FPGA Design for Monitoring CANbus Traffic in a Prosthetic Limb Sensor Network

by

A. Bochem, J. Deschenes, J. Williams,
Y. Losier and K. B. Kent

TR 10-204, June 15, 2010

Faculty of Computer Science
University of New Brunswick
Fredericton, NB, E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
http://www.cs.unb.ca

Design Document

Executive Summary

The prosthesis industry is moving towards an open standards bus, with this comes the need to monitor the information being transmitted. This monitor is necessary to help develop new prosthetics as well as to assess rehabilitation effectiveness. Current monitors are unacceptable as they are unable to capture information at 1 Mbps, which is required. Altera's FPGA platform allows for rapid development, fast problem targeting and most important, the ability to operate at the speed required to process the data. The design of the system will be flexible enough to meet future needs and follow current standards, simplifying the work required for end users to utilize the system. The system in its current iteration is well on its way to achieving the goals that must be met to call the project a success.

Design Purpose

In the prosthetics field, various research institutions and commercial vendors are currently developing prosthetic limb components with a newly proposed open bus communication standard. The goal is to simplify the interconnection of these components within a prosthetic limb system and to allow the interchangeability of devices from different manufacturers. This initiative is still in development and will undoubtedly face some obstacles during its implementation, as there are currently no devices available to reliably monitor the activity occurring on both the sensor and actuator buses.

The design of a FPGA-based prosthetic limb data monitor will allow embedded system engineers to monitor the communication activity occurring in the system thereby providing an effective developmental tool to not only help develop new prosthetic limb components but also advance the open bus standard initiative. Furthermore, the monitor's data logging capabilities will allow the prosthetic fitting rehabilitation team to analyze the amputee's daily use of the system in order to assess its rehabilitation effectiveness. The evaluation of the data monitor's capabilities will be performed by the team in conjunction with UNB researchers who are leading members of the Standardized Communication Interface for Prosthetics forum.

Application Scope

The application consists of a data monitor attached to a prosthesis computer processing unit. This monitor will receive all data and messages from multiple 1-Mbps Controller Area Network buses. Based on input from the users and the current operating mode a number of filters will be applied and the appropriate messages logged.

The logging will consist of writing the messages to a cache and then when appropriate compress the information and send it to (or through) one of the following devices: SD

flash memory, a serial connection, a Bluetooth connection or the ANT ultra low power wireless connection.

Due to the open standards nature of this project, great care must be taken to design and implement modules and cores that are flexible, interchangeable and will allow vendors the freedom to do what is necessary within the framework of the data monitor. Other constraints include a low power design, real-time error handling and robustness.

Targeted Users

The targeted users for this application are prosthesis creators and members of the prosthesis research and design community. The prosthesis community has shown great interest in the open standard and the data logger is a vital component. This data logger will provide the community with the ability to implement and test their prosthetics in compliance with the new standard. The future of this data logger also includes collecting motor and servo statistics for vendors, allowing them to monitor the running conditions of the prosthesis.

Altera FPGA

Altera's FPGA technology and Quartus development environment will allow rapid prototyping through the use of available cores and custom modules. Specifically the DE2 board allows quick validation of numerous communication and storage technologies. Altera also provides a long term cost effective solution as prototyping can quickly become an end-market product.

Function Description

The open bus standard-based data monitor system for Prosthetic Limbs captures and collects the serial information from two separate Controller Area Network (CAN) buses: the sensor and control buses. With the collected information it then would transform the data into the correct CAN message format. A timestamp would be added and the messages would be passed through a user controlled filter to dictate which messages should be logged. After the filter the two buses' messages are merged and sorted according to their timestamp. Once sorted, the information is then sent to an output device for processing. Currently the output devices consist of an RS-232 to Computer monitor and an SPI SD Flash card interface. This project will allow the end-user the ability to understand how the sensor data is being used to create control messages. This gives them a method for troubleshooting prosthetic systems, an ability they currently do not have.

The implementation consists of various "working" cores, cores that can be individually tested each containing one piece of the functionality which is required for the overall system. These cores interface with one another through a standard FIFO, alleviating timing issues. A breakdown of each core is as follows:

1. "CAN-Core" x 2 (one for each CAN-bus "sensor" and "control")

Functionality: The module sets up the internal CANBUS Module so it is compatible with the end-users CANBUS system. The module receives information from the CANBUS via a 2-wire serial interface, properly checks all required aspects of a CAN message (as per CANBUS specification). After parsing the message the module would then add the timestamp to each message and finally pass the message to the FIFO.

2. “Filter” x 2
Functionality: Allows the end-user to specify filters based on node-id or message priority. This filters the appropriate messages.

3. “Merge-Sorter”
Functionality: Sorts the various messages based on timestamp, merges the two buses messages into one bus.

4. “Output”
Functionality: This core contains the functionality required to get the information out to the user is some manner. Currently an RS-232 Serial connection and a small program allow the user to see real-time CAN-bus data. An SPI SD flash card core allows information to be stored on a flash card, giving the end user the ability to capture real field data.

5. FIFO Core:
This core serves as a communication interface between the system’s modules and can hold up to eight 128-bit words. In the current system design, a CAN message plus an internal timestamp is 128 bit long.

Performance Parameters

The overall performance parameters are dictated by the speed at which the CAN-bus operates. Each CAN-bus operates in high speed at a maximum speed of 1Mbps. Therefore our system must be able to handle a theoretical maximum throughput of 2Mbps. Further performance parameters include an error rate of zero over an operating time of twenty-four hours.

The test being performed involves a CAN-bus with two nodes which are sending messages to each other. Using a transceiver the messages are digitized and sent to the Altera DE2 Development Board by the GPIO pins. Once the system is fully stable, we will bring the second CAN-bus online continuing the test until both CAN-buses are operating at maximum speed.

As of this submission, the following results have been achieved:

1 Mbps throughput of the entire system has been achieved, with one CAN-bus consisting of two node devices sending information as fast as possible. A serial writing bottleneck is currently holding back the full potential of the system, potential solutions to

this problem involves increasing the clock rate, and sending the value over the GPIO pins.

The error rate is not yet at zero, however excluding the first few hundred milliseconds the error rate is very close to zero. Initialization techniques are being devised and it is suspected that this method will reduce the error rate to its target.

The system itself can run without critical errors indefinitely. This means the throughput of the system is even and no bottlenecks are causing the system to stop functioning.

Throughput and error rates have been steadily improving. Ideas for increasing the output speed through compression show great potential in overcoming their inherent slowness.

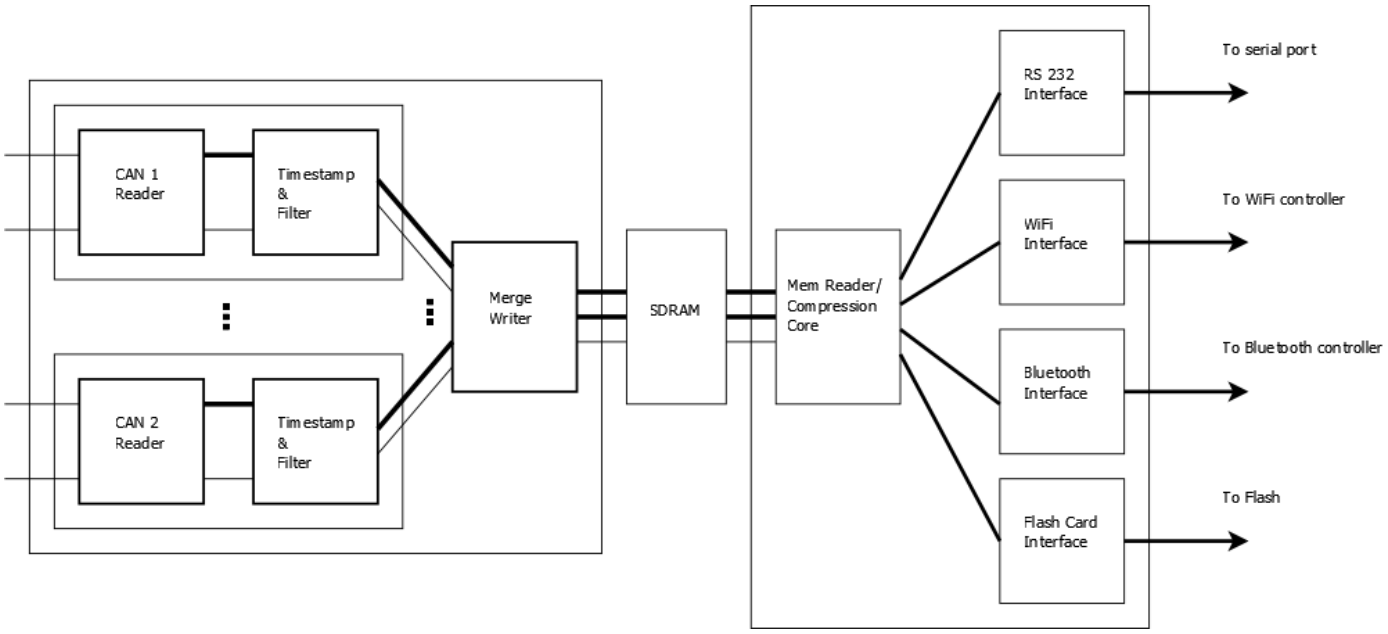
The Altera Quartus II Development Environment has allowed us to rapidly prototype solutions to the various problems that have come up. The system has given us the ability to utilize open source products and cores to greatly reduce the time to implementation. Furthermore through the various properties of the target board, we were able to add functionality required in the prosthesis monitoring system.

For verification and validation of our design we used Altera's Model Sim. Model Sim is a simulation tool that allowed us to observe the systems' behavior during runtime for finding and eliminating errors in the design. This greatly reduced frustration and again allowed us to work through issues faster.

Design Architecture

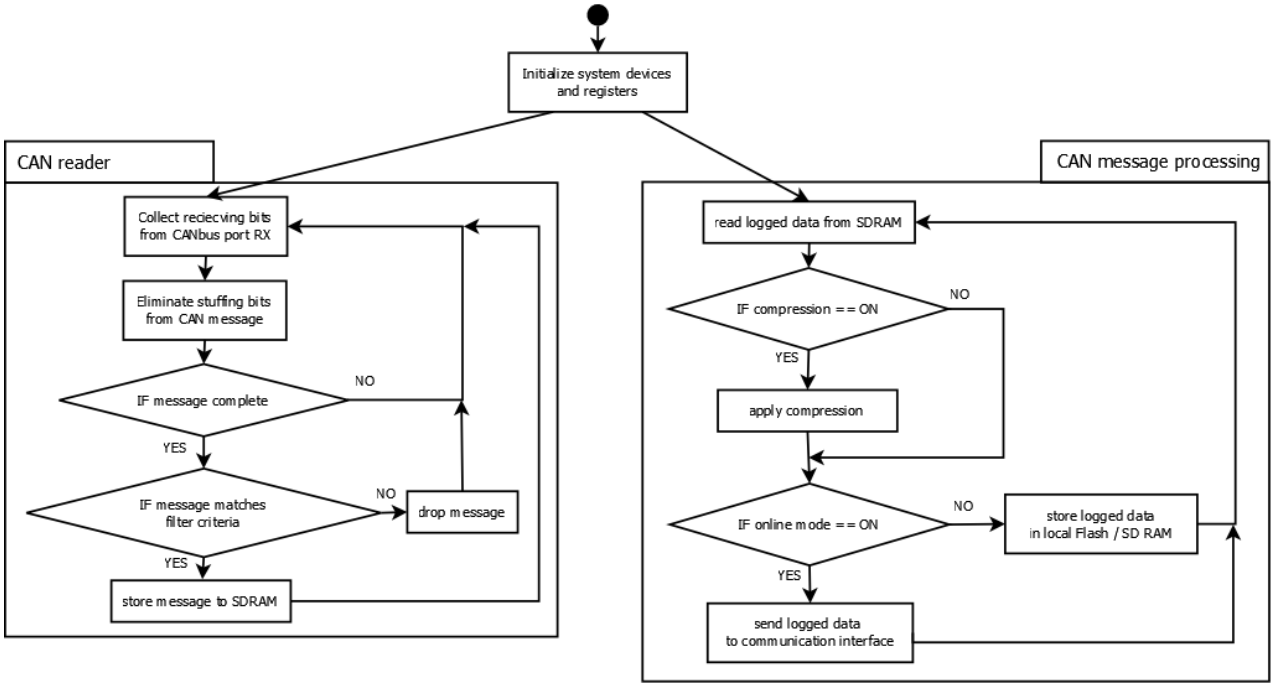
Design block diagram

This diagram shows the modules required to implement the system. The CAN-bus



reader receives information from the external bus, it is then filtered and merged (according to the timestamp value) out to the output core. From there it could be transformed by the compression core and transmitted through or to the appropriate medium (for storage or analysis).

Software flow diagram



This diagram shows the internal processing of the system at an abstract level. It also shows that the core functions can be separated into two modules and run in parallel. This is an important feature for the proposed system as it allows for the scalability necessary to connect multiple CAN-buses and have the busses, and therefore the overall monitor to run more efficiently.

Design Methodology

Our design methodology consisted of three governing principles. The first principle was a bottom up approach with a unified module design goal. This was facilitated through a weekly accountability meeting and brief bi-weekly code review meetings. The accountability meetings consisted of actors from the end user group and embedded system mentors. During this meeting the project goals were continually tweaked and our assumptions were tested. This promoted a dynamic and adaptive development environment. Furthermore, each module was designed to be self-testable. This allowed verification of each module to occur, reducing total system errors.

The second principal was universal code ownership. Each of the three members were responsible for understanding the entire codebase. This was done in conjunction with pair programming to reduce the complexity of each person's task. This methodology allowed for all members to form different solutions to the most complex problems. Then through discussion and implementation the best solution was selected and modified to fit the groups problem solving style. Through this process each section of code had to be justified and any superfluous code was removed. This contributed to a more unified codebase, with each member responsible for the entire system and problems that occurred were solved quickly and without ownership disputes.

Finally our last principle has been that of "Continuous process" which has pushed the team to continually improve, integrate and push out many small releases. Continual integration has helped us keep to our deadlines. Each week we set out to integrate new features and components into our design. Through this process we were able to continually re-evaluate what we had time for, what was feasible and what was necessary for the project. With close integration occurring every week, it allowed a much smoother move from simulation to implementation, saving us from hunting for system wide bugs. Continual improvements helped us to maintain high coding standards. The codebase was continually cleaned up by simplifying existing architectures and structures. This was done in an effort to decrease the amount of work required to bring our code to the level required by an open standard. Finally, by continually pushing out small releases we were able to bring functional components to our weekly meetings, promoting instant feedback on design decisions. With each release we were able to realign our goals and make strides towards achieving a singular vision of the final product.

Design Features

The project's target is the implementation of a CAN-bus monitoring system. It should allow the connection of multiple CAN-buses and save received messages for further evaluation. Decisions about which messages are used for further processing will be made based upon configurable filters. These filters can identify messages based up on priority or node ID, which are part of the CAN message protocol.

For the purpose of connecting multiple buses, the central unit of the design is a data monitor that processes the filtered CAN messages and stores them in FIFO's on the DE2 board. From these FIFO's the CAN messages are stored into a larger external memory device or directly transmitted to a connected Host-PC. The system can operate in one of two modes. These two modes are specified as offline and online-mode. Online-mode is preferable for engineers and vendors during the development and testing phase. This allows maximum performance tests on the CAN-bus communication. The offline-mode should allow long term observation and error evaluation before and after introduction to the market.

If time permits, a compression module will be designed. This will allow extended long term error logging. Another optional feature will be the application of a wireless communication interface unit. This would allow the transmission of logged data to an external receiver without the use of a wired connection to a Host PC.

Conclusion

A great deal was learned about the Altera FPGA platform specifically, and low level embedded system development in general. The most important lesson learned was understanding the differences between, and problems that can occur when transitioning from simulation to the actual implementation. We spent a number of hours trying to solve what we thought were random bugs only to realize we needed to set an internal register or put a pin value high. Reading the Altera documentation beforehand would have saved us time and allowed us to handle the implementation in a more efficient manner.

Another tip would be to understand all the tools Altera has to make your life easier. During this project we learned Model Sim, which saved us a great deal of time in the simulation testing environment. When we implemented the system, most of the problems we had solved in Model Sim were absent, however additional problems needed to be solved. We were informed by a colleague that Altera indeed had a platform to assist us, but at this point it was too late.

Some tips, or guidelines that would have allowed for a much more successful project would be to read up on all the tools, learn them through online tutorials and then incorporate that understanding into project development. From the beginning design an embedded system which utilizes the various LED's, switches and displays to test and verify (in a reconfigurable manner) a number of system properties. The reconfigurability offered in FPGAs can allow for a dynamic and robust testing tool. Plan ahead and understand where in your design problems may occur and attempt to incorporate both simulated and practical testing into your system design.

FPGA Data Monitor for Prosthetic Limb System Documentation

Module: can_controller

Justin Deschenes, Jeremy Williams, Alexander Bochem

Faculty of Computer Science
University of New Brunswick
Fredericton, Canada

Revision Table

Changes	**Revision**	**Date**
Module Description, Input, Output, Parameters, Registers, Internal Description	0.1	3/22/2010
revision changes	0.11	6/9/2010

Module Description

The *can_controller* module instantiates the subjacent modules *can_io_adapter* and *can_top*. The *can_top* module itself is responsible for receiving CAN-messages and provides them for further processing in the receive buffer. The *can_io_adapter* serves as configuration interface between the wishbone interface of the *can_top* module and the *can_controller* module.
Bus timing, acceptance mask/code and interrupt configuration are performed by parameter settings in the *can_controller* module.

Input

iClock:	Clock signal. Drives the *can_controller* module.
iCANcoreClk:	Clock signal. Drives the *can_top* module. Timing parameters of the module are computed based on this clock.
iWishboneClk,	Clock signal. Drives the Wishbone-Interface of the *can_top* module and the interface module *can_io_adapter*.
iReset	Reset signal. Reinitializes the *can_controller* and all underlying modules.
iCANbus_Rx	Receive pin from CANbus. Connected to the JP2 pins on the DE2 board.
iWriteFifoFull	Control signal from FIFO *CANmsgFIFO*. Received CAN messages are stored in the FIFO for further processing if FIFO is not full.

Output

oCANbus_Tx	CANbus transmit pin from *can_top* module. (not used)
oDataToFIFO	128 bit data bus to *CANmsgFIFO.*
oWriteRequest	Control signal to send a write request to the CANmsgFIFO.
oCountSteps	6 bit debug bus to control internal proceedings in the module (not required)
oReadSteps	6 bit debug bus to control internal proceedings in the module (not required)
oIObusy	Debug signal to control internal proceedings in the module (not required)

Parameters

CAN_TIMING0_BRP	Baud rate prescaler (BRP) Prescale formula: (2 * iCANcoreClk * (BRP+1))
CAN_TIMING0_SJW	SJW = (value+1)
CAN_TIMING1_TSEG1	TSEG1 segment (value+1)
CAN_TIMING1_TSEG2	TSEG2 segment (value+1)
CAN_TIMING1_SAM	Triple sampling (Enable=1 / Disable=0)
EXT_CLK_OFF	External clock, (Enable=1 / Disable=0)
CAN_ACCEPTANCE_CODE	Message acceptance code
CAN_ACCEPTANCE_MASK	Message acceptance mask

IRQ_ENABLE_OIE	Enable Overrun Interrupt, (Enable=1 / Disable=0)
IRQ_ENABLE_EIE	Enable Error Interrupt, (Enable=1 / Disable=0)
IRQ_ENABLE_RIE	Enable Receive Interrupt, (Enable=1 / Disable=0)
IRQ_ENABLE_TIE	Disable Transmit Interrupt, (Enable=1 / Disable=0)

Registers

reg [5:0] controller_state	Control value to control main state machine of *can_controller* module.
reg [5:0] init_state	Control value for initialization phase state machine. Placed inside the INIT state of main state machine.
reg [5:0] init_step	Control value for init_state, selecting registers for perform configuration steps.
reg [5:0] read_rr_state	Control value for state machine inside READ_RECEIVE_BUFFER, to collect CAN-message from receive buffer of *can_top* module.
reg [7:0] Register_address	Used to address internal registers in the can_top module. Interface to can_io_adapter
reg [7:0] writeRegister_data	Used to pass register values to *can_top* module

Wires

CAN_IO_BUSY	Control signal, set to 1 while *can_io_adapter* is processing a read or write request from *can_controller*.
IRQflag_CANcore	Control signal, set to 1 if interrupt occurred in the *can_top* module.
[7:0] readRegister_data	Used to pass register values from *can_top* module

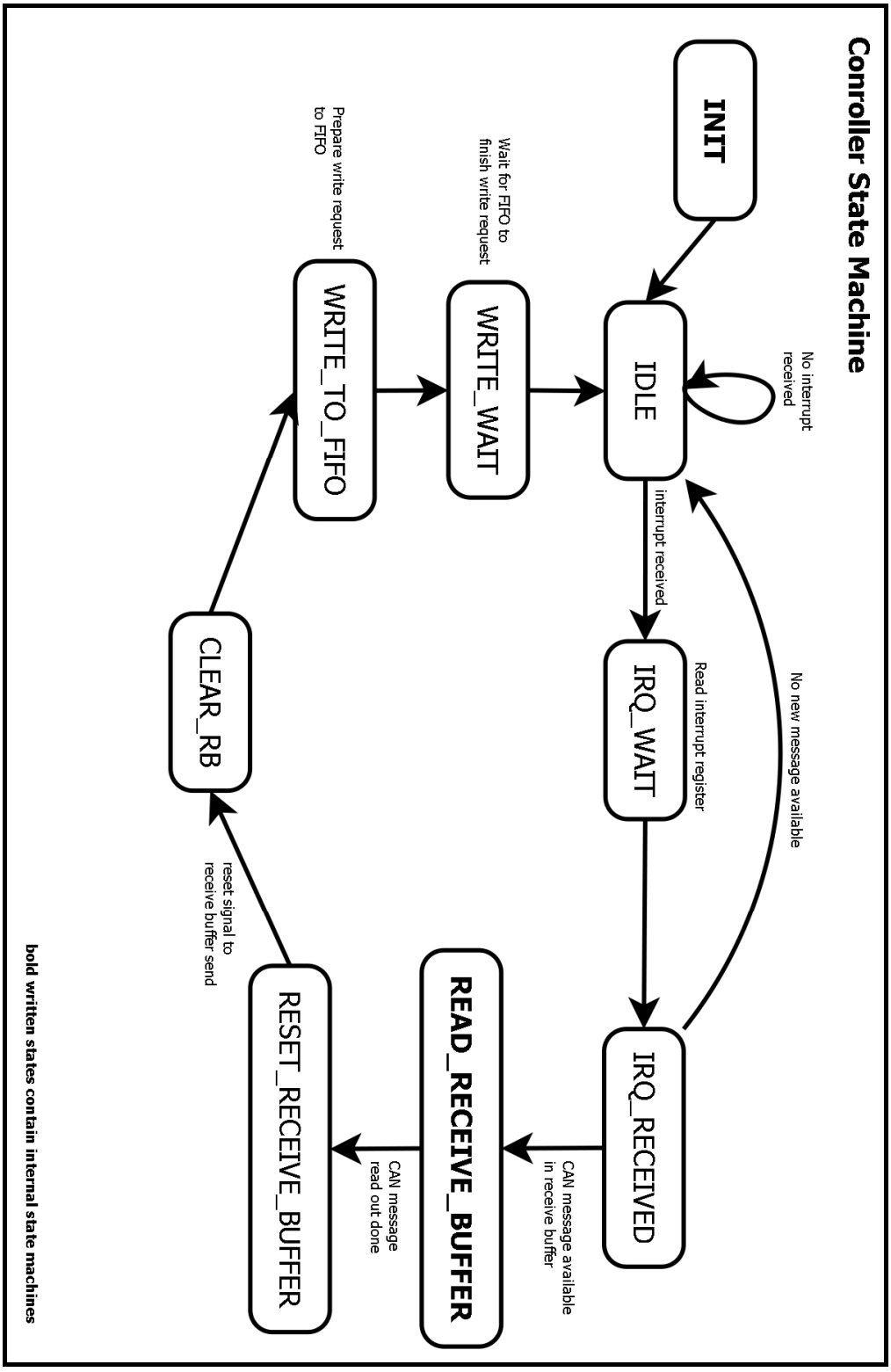
Internal Description

The processing flow of the module is split into several state machines which form a hierarchical tree structure.

The *can_controller* module configures the *can_top* module registers on startup. Then it waits for an interrupt signal from the *can_top* module and checks if a CAN message has been received. (IDLE state) If it has, the receive buffer is read out (READ_RECEIVE_BUFFER), cleared (RESET_RECEIVE_BUFFER) and the can_controller goes back to IDLE state.

The *can_top* module only allows the reading of one Byte per read request. For reading out a complete CAN message, ten reading requests need to be processed. When reading a message from *can_top* module, a timestamp of 48bit is attached and the data is written to an external FIFO module.

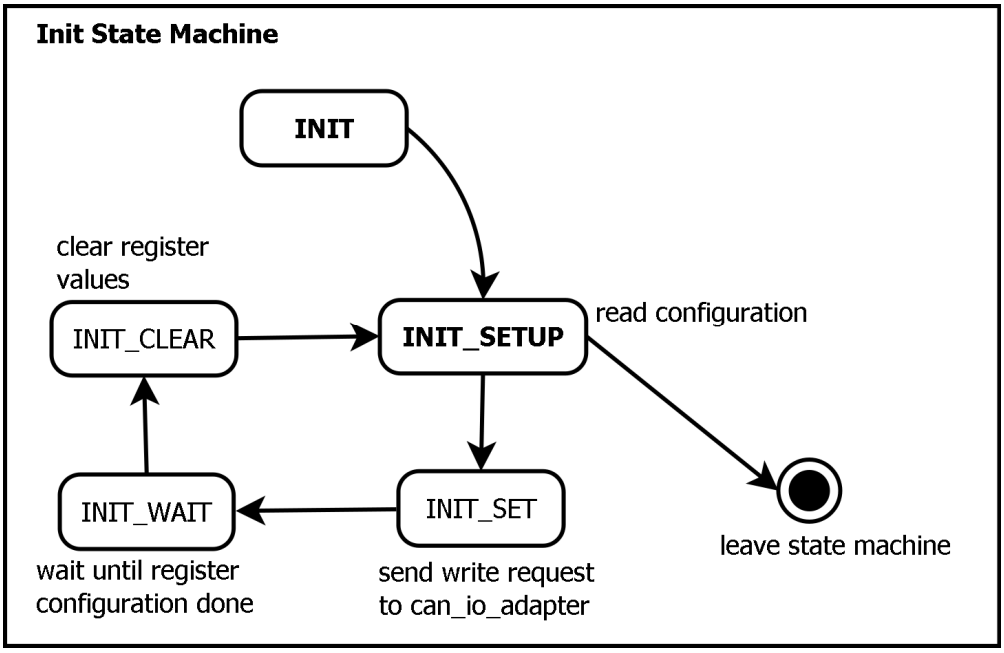
Controller State Machine



The controller state machine is responsible for the main processing of the *can_controller* module. After startup it activates the **INIT** state (Init State Machine) which

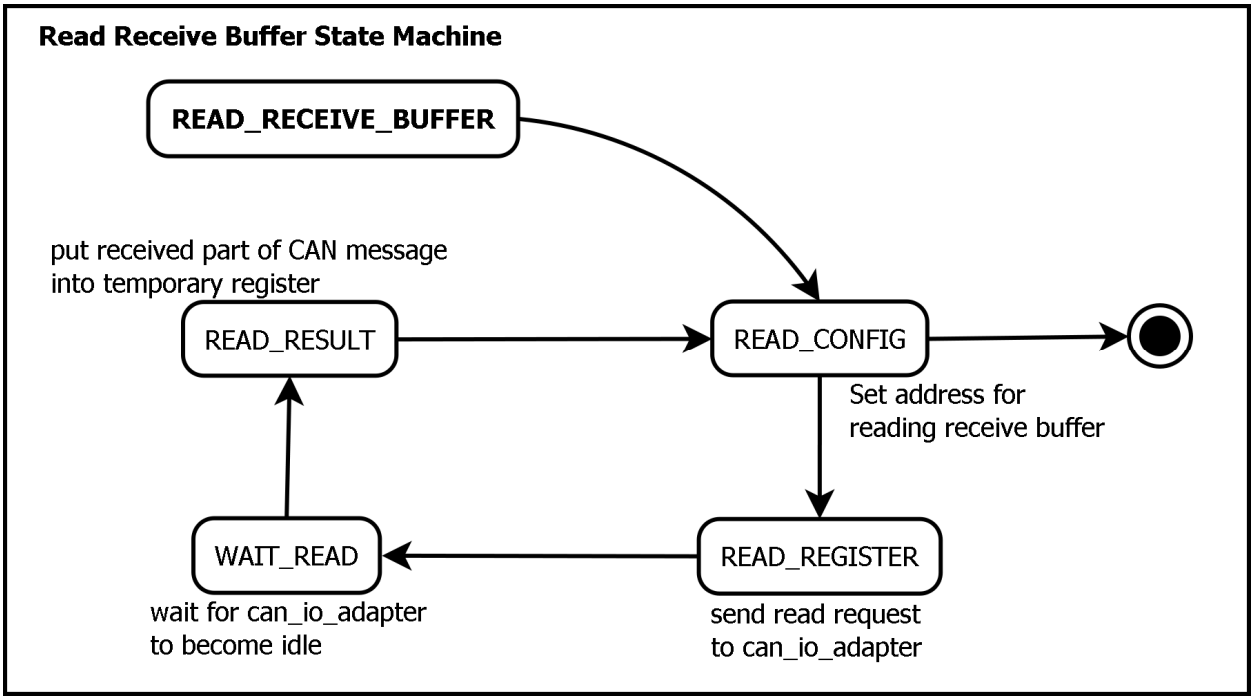
configures internal registers in the *can_top* module. The state READ_RECEIVE_BUFFER performs a readout of the receive buffer in the *can_top* module which is described in the Read Receive Buffer State Machine diagram.

Init State Machine



The **INIT** state performs the configuration of internal registers in the *can_top* module. The initialization runs until all defined values in state **INIT_SETUP** have been set. Afterwards control is passed back to the controller state machine.

Read Receive Buffer State Machine



The READ_RECEIVE_BUFFER state performs a read out of the receive buffer in the *can_top* module. The *can_top* module only allows reading out one Byte per read request. For reading a complete CAN-message up to ten read requests need to be processed. Afterwards the control is handed back to the controller state machine.

Module Usage

Dependencies

- The *can_controller* module instantiates the *can_io_adapter* module.
- The module expects a FIFO connected to the correspondent I/O-ports (iWriteFifoFull, oDataToFIFO, oWriteRequest).
- The FIFO has to run with the same clock speed as iClock.

Instantiation example

```
//Receive CAN messages and write them into FIFO
can_controller CANcontroller(
            iClock(PinName for ModuleClock),
            .iCANcoreClk(PinName for CANcore Clock),
            .iWishboneClk(PinName for Wishbone Clock),
            .iReset(PinName for Reset Signal),
            .iCANbus_Rx(PinName for .CANbus Receive),
            .iWriteFifoFull(PinName for .FIFO is full signal),
            .oCANbus_Tx(PinName for .CANbus Transmit),
            .oDataToFIFO(PinName for data bus to FIFO),
            .oWriteRequest(PinName for write request to FIFO));
```

The ports for oCountSteps, oReadSteps and oIObusy are not required.

Test case description

None

Additional Information

Problems section

After troubleshooting a CAN message dropping issue, it is likely that this module or the *can_io_adapter* is not fast enough to handle a full 1 MBps throughput (although the specification says it is). If the issues lie with this module, architectural optimizations must be made.

References

None

FPGA Data Monitor for Prosthetic Limb System Documentation

Module: can_io_adapter

Justin Deschenes, Jeremy Williams, Alexander Bochem

Faculty of Computer Science
University of New Brunswick
Fredericton, Canada

Revision Table

Changes	**Revision**	**Date**
Module Description, Internal Description, Module Usage	0.1	3/22/2010
revision changes	0.11	6/9/2010

Module Description

The module can_io_adapter serves as a communication interface to perform register read and write request to the can_top module. The interface behavior is based on the Wishbone Bus standard.

Input

iCANcoreClk	system clock signal
iWishboneClk	system clock for Wishbone interface
iReset	system reset signal
iReadRequest	read request from can_controller to can_top registers
iWriteRequest	write request from can_controller to can_top registers
iCANbus_Receive	Receive pin from CANbus, forwarded to can_top module
[7:0] iAddr	address bus for read/write request to can_top module
[7:0] iDataWrite	data bus for write request to can_top module

Output

[7:0] oDataRead	received value from can_top register after READ request
oBusy	HIGH=io_adapter busy, LOW=io_adapter ready for request
oCANbus_Transmit	Transmit pin from CANbus
oIRQ	IRQ flag from can_top module, used in can_controller

Registers

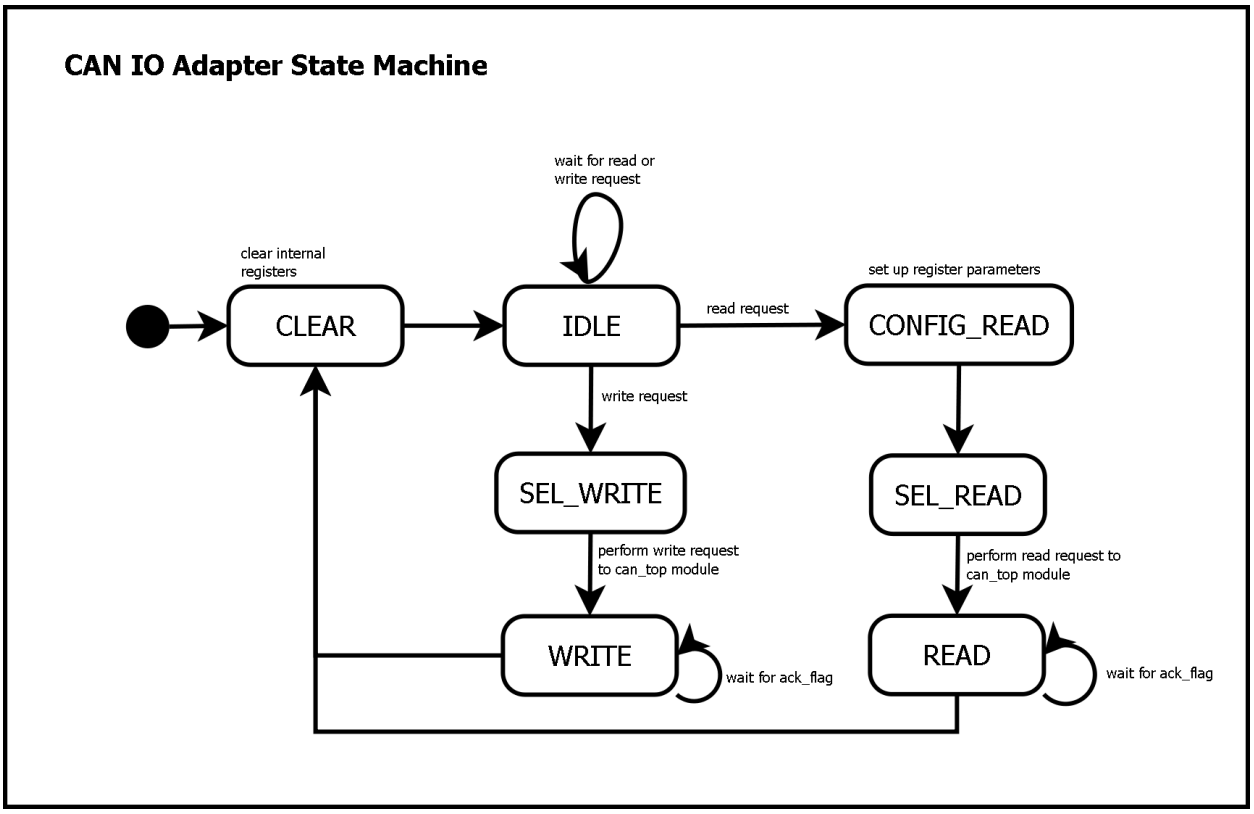
reg [5:0] state	control value for state machine in can_io_adapter
reg wb_free	internal status flag, HIGH= while can_io_adapter is busy
reg wb_we_i	input signal to can_top, write register enable
reg [7:0] wb_adr_i	input address bus to can_top
reg [7:0] wb_dat_i	input data bus to can_top, write register value
reg wb_cyc_i;	input can_top, "control signal"
reg wb_stb_i	input can_top, "control signal"

Wires

wb_clk_i	clock for wishbone interface in can_top module
wb_rst_i	reset signal for wishbone interface in can_top module
[7:0] wb_dat_o	output data bus from can_top, results from read register
wb_ack_o	output control signal HIGH=can_top finished read- or write-request LOW = can_top module busy processing request

Internal Description

The can_io_adapter handels read and write requests from the can_controller module and handles interface requirements for the wishbone bus interface on the can_top module.



Module Usage

Dependencies

The can_io_adapter module instantiates the can_top module.

The can_top module requires the additional modules can_bsp, can_ibo, can_acf, can_crc, can_fifo, can_ibo, can_btl, can_registers and can_register.

Instantiation example

```
//IO adapter handles Write and Read requests for registers in CAN core
can_io_adapter CAN_IO(
       .iAddr(Register_address for can_top module),
       .iCANcoreClk(CANcore Clock),
       .iWishboneClk(Wishbone Clock),
       .iDataWrite(data bus for write request),
       .iReadRequest(Read request flag to can_io_adapter),
       .iReset(Reset signal),
       .iCANbus_Receive(CANbus receive pin),
       .iWriteRequest(Write request flag to can_io_adapter),
       .oBusy(status signal from can_io_adapter busy),
       .oDataRead(data bus for result from read request),
       .oCANbus_Transmit(CANbus transmission pin),
       .oIRQ(IRQ pin from can_top module));
```

Test case description

None

Additional Information

Problems section

When troubleshooting a CAN message dropping issue, it is likely that the module or the *can_controller* is not fast enough to handle a full 1 MBps throughput (although the specification says it is). If the problem lies in this module, more reading must be done to determine if it is the result of misconfiguration or a deeper design issue.

References

- Datasheet for SJA1000, Stand-alone CAN controller (SJA1000.pdf)
- Application Note for SJA 1000, Stand-alone CAN controller (AN97076.pdf)

FPGA Data Monitor for Prosthetic Limb System Documentation

Module: CANfilter

Justin Deschenes, Jeremy Williams, Alexander Bochem

Faculty of Computer Science
University of New Brunswick
Fredericton, Canada

Revision Table

Changes	**Revision**	**Date**
Module Description, Internal Description, Module Usage	0.1	3/23/2010
revision changes	0.11	6/9/2010

Module Description

The module CANfilter reads CAN messages with a timestamp from the CANmsgFIFO. The message part is compared with a set of filter parameters which specify node IDs or message priorities.
If a message fits the parameters it is dropped from further processing. Otherwise the CANfilter module passes the CAN message, including timestamp to another FIFO (FilterMergeSortFIFO).
Configuration of parameters in the current design requires the modification and recompilation of the system's design.

Input

iCLK	System clock, same used for read-clock of CANmsgFIFO
iReset	System reset signal
iReadFifoEmpty,	Signal for empty CANmsgFIFO
[127:0] iData	128 bit data bus read from CANmsgFIFO
iWriteFifoFull	Signal from FilterMergeSortFIFO, if FIFO is full

Output

oReadRequest	signal read request to CANmsgFIFO module
oWriteRequest	signal write request to FilterMergeSortFIFO module
[127:0] oData	128bit data bus write to FilterMergeSortFIFO module

Parameters

//FILTER PARAMETER	
HIGH_PRIORITY_MSG = 2'b0	filter high priority messages
NORMAL_PRIORITY_MSG = 2'b0	filter normal priority messages
LOW_PRIORITY_MSG = 2'b10	filter low priority messages
NODE_ID_1 = 8'd8	filter node with particular ID 8
NODE_ID_2 = 8'h0A;	filter node with particular ID 10

Registers

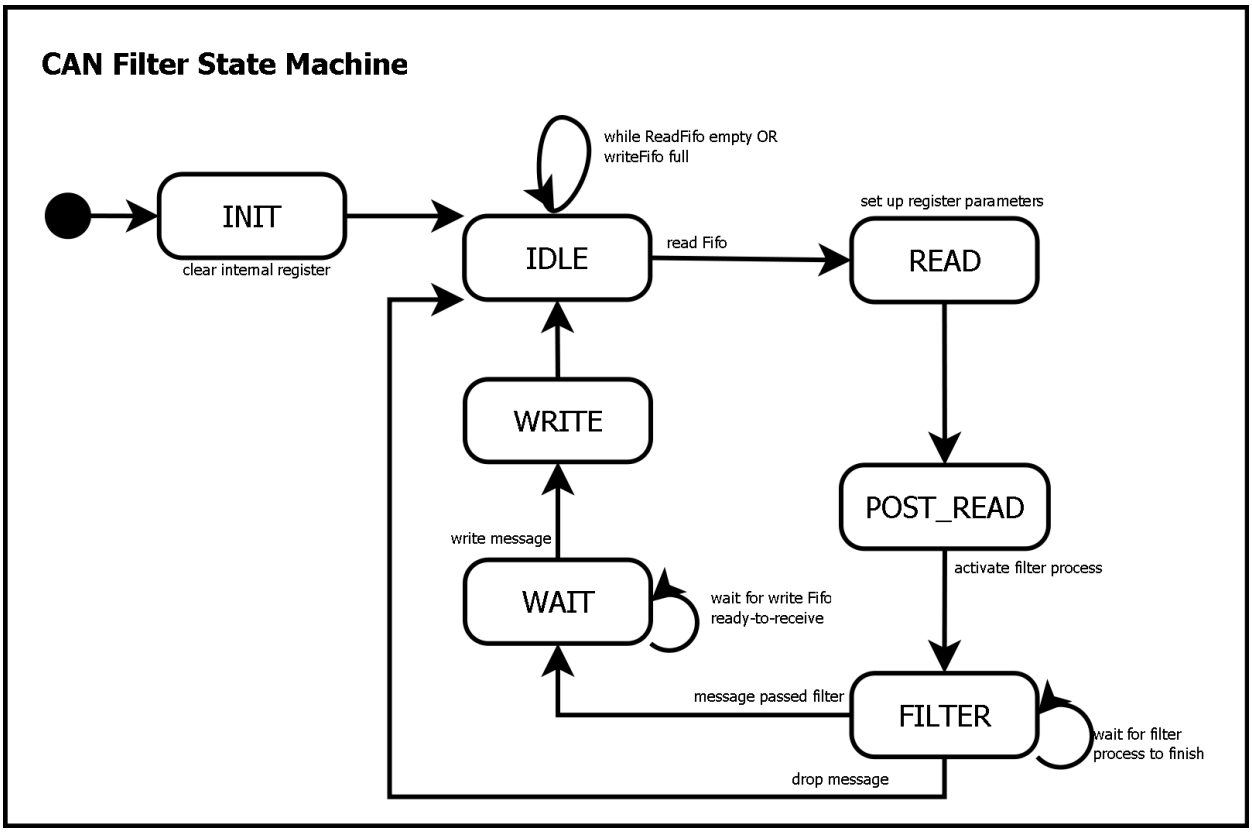
None of importance

Wires

None

Internal Description

The processing flow of the CANfilter module is controlled by an internal state machine.



Module Usage

Dependencies

The module has no internal instantiations.
It uses a FIFO to read input data and another FIFO to provide output data. The FIFOs need a data bus width of 128bit.

Instantiation example

```
//Read data from FIFO IOtoFilterFIFO and drop messages which fit filter criteria
CANfilter    CANmsgFilter(
.iCLK(CLOCK_50),                        //system clock
.iReset(iRESET),                        //system reset signal
.iReadFifoEmpty(FifoEmpty_Filter),      //1 bit signal from IOtoFilterFIFO, if is
                                        //empty
iData(CAN_Message_to_Filter),           //128bit CAN message, read from
                                        //IOtoFilterFIFO
.oReadRequest(ReadRequest_Filter),      //1 bit signal, read request to
                                        //IOtoFilterFIFO
.oData(CAN_Message_from_Filter),        //128bit block read from //FilterMergeSortFIFO
.iWriteFifoFull(FifoFull_Filter),       //signal from FilterMergeSortFIFO,If full
.oWriteRequest(writeRequest_Filter));   //write request signal to Filter
```

Test case description

none

Additional Information

Problems section
none

References
none

FPGA Data Monitor for Prosthetic Limb System Documentation

Module: CANmerge

Justin Deschenes, Jeremy Williams, Alexander Bochem

Faculty of Computer Science
University of New Brunswick
Fredericton, Canada

Revision Table

Changes	**Revision**	**Date**
Initial document created, Module description, Internal Description created	0.1	3/25/2010
revision changes	0.11	6/9/2010

Module Description

The CANmerge module reads from two FIFOs, each of which is responsible for a different bus, Signal or Control, and sorts input based on its 48 bit timestamp. The output is then passed into a FIFO which is then passed on to an output module.

Input

clk_i	System clock, same speed as the FIFO's
rst_i	System reset signal
[127:0] data1_i	128 bit data bus connected to the 1st CANbus
[127:0] data2_i	128 bit data bus connected to the 2nd CANbus
empty1_i	Signal to determine if the first data bus is empty
empty2_i	Signal to determine if the second data bus is empty
wrfull	Signal from the output FIFO, if FIFO is full.

Output

req1_o	Signal requesting a read operation from first FIFO
req2_o	Signal requesting a read operation from second FIFO
[127:0] data_o	128bit data bus writing to output FIFO
en_o	
wrReq	Signal a write request operation to output FIFO

Parameters

None

Registers

None

Wires

None

Internal Description

This module determines which of the two input FIFOs' message is next to be added to the output according to timestamp. To do this it holds a buffer of 2 messages per FIFO and determines which of the buffered messages must be put to the output FIFO.

Module Usage

Dependencies

This module has no internal instantiations.

It uses two FIFOs to read inputs, and another FIFO to provide output data. The FIFOs need a data bus width of 128bit.

Instantiation example

```
can_merge MergeSorter(
      .clk_i(CLOCK_50),                      In: System Clock
      .rst_i(iRESET),                        In: Global Reset
      .data1_i(CAN_Message_to_MergeSort1),   In: 128bit output of C1 after Filter
      .data2_i(CAN_Message_to_MergeSort2),   In: 128bit output of C2 after Filter
      .empty1_i(FifoEmpty_MergeSort1),       in: output of C1 FIFO empty sig
      .empty2_i(FifoEmpty_MergeSort2),       in: output of C2 FIFO empty sig
      .wrfull(retFifoFull));                 in: signal from output filter, if full
      .req1_o(ReadRequest_MergeSort1),       out: to C1 FIFO
      .req2_o(ReadRequest_MergeSort2),       out: to C2 FIFO
      .data_o(oCANmerged),                   out: 128bit msg output of module
      .wrReq(writeRequest_Merge),            out: write request to output FIFO
```

Test case description

none

Additional Information

Problems section

Currently only tested using one input functionality, the other input was only sending a high signal. The logic guiding the single input is duplicated for dual input so no big issues are predicted when using two inputs.

References

None

FPGA Data Monitor for Prosthetic Limb System Documentation

Module: serial_out

Justin Deschenes, Jeremy Williams, Alexander Bochem

Faculty of Computer Science
University of New Brunswick
Fredericton, Canada

Revision Table

Changes	**Revision**	**Date**
Module description, Internal description	0.1	3/25/2010
revision changes	0.11	6/9/2010

Module Description

This module converts CANbus messages received over the FIFO into proper serialized 8bit messages which can be sent over a serial connection and read by a terminal program.

Input

clk_i	Serial clock, runs at the necessary speed to support a specific BAUD rate
rst_i	System reset signal
[127:0] data_i	128 bit data bus read from input FIFO (Merged FIFO)
empty_i	Signal for empty input FIFO

Output

req_o	Signal outputs whether it is requesting to send
txd_o	Signal outputs the serialized transmission data
[1:0] oState	2 bit signal used to setup next state in module

Parameters

Various state machine parameters.

Registers

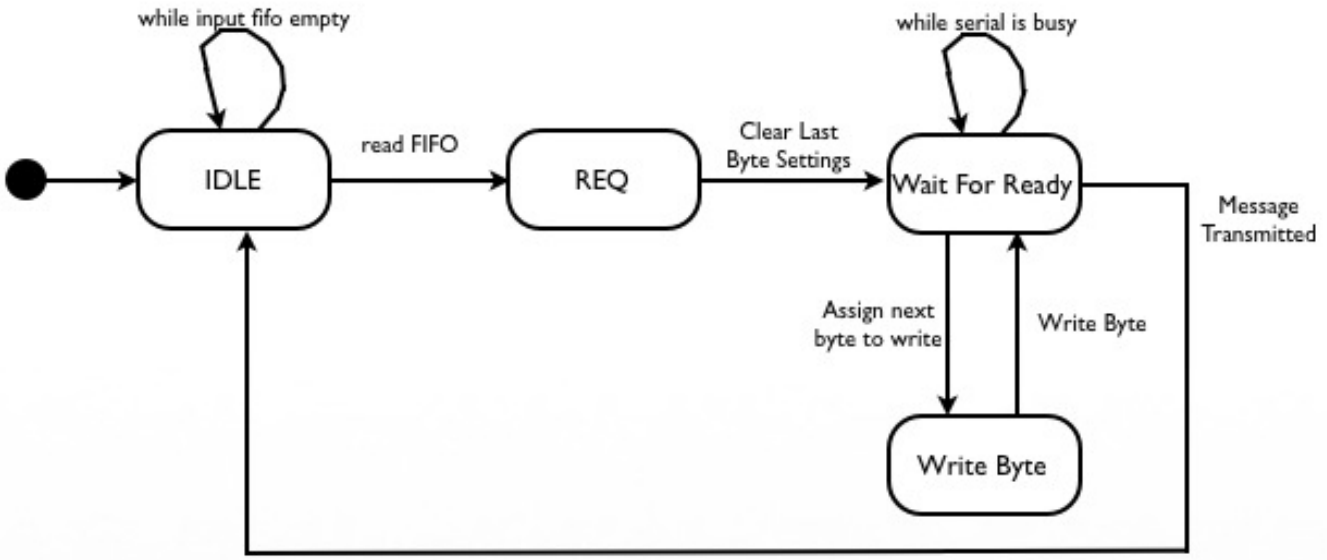
[7:0] serial_bus	8bit register value passed into the serial UART core internal, used as data bus

Wires

serial_wr	Passed into the serial UART core, Connected to Write input, used to determine if a byte is ready to be written
eot	Passed into serial UART core, determines when transmission ends
serial_ready	Passed into serial UART core, determines when serial is ready

Internal Description

The processing flow of the Serial Out module is controlled by an internal state machine.



Module Usage

Dependencies

The SerialOut module instantiates the Serial module. The module expects a FIFO connected to the corresponding input ports (data_i and empty_i). It also expects a RS-232 compliant hardware device on the output side, connecting to the corresponding output ports (req_o, txd_o).

Instantiation example

```
serial_out serial(
    .clk_i(PinName for SerialClock),
    .rst_i(PinName for Global Reset),
    .data_i(CAN_Message_to_Serial),
    .empty_i(PinName for FIFO is empty signal),
    .req_o(PinName for REQ bit (Connected to RS-232 comp. dev.),
    .txd_o(PinName for TX bit (Connected to RS-232 comp. dev.),
    .oState(not currently implemented ));
```

Test case description

None

Additional Information

Problems section

Currently this module is working with the RS-232 chip on the Altera FPGA board, however this chip only allows for a max BPS of 119200. A faster Baud rate might be required when the full system (Signal and Control Canbuses) is being tested.

References

None

FPGA Data Monitor for Prosthetic Limb System Documentation

Module: spi_controller <ALPHA>

Justin Deschenes, Jeremy Williams, Alexander Bochem

Faculty of Computer Science
University of New Brunswick
Fredericton, Canada

Revision Table

Changes	**Revision**	**Date**
Module Description, Input, Output, Parameters, Registers, Internal Description	0.1	3/25/2010
revision changes	0.11	6/0/2010

Module Description

The *spi_controller* module instatiates the *spi_io_adapter.* The module handles writing of CAN messages into a SD flash card attached to the Serial Peripheral Interface. This module is in early alpha stages.

Input

iClock	System Clock. Drives the FIFO Read logic.
iSPIClock	SPI Clock. Drives the SPI R/W logic.
[127:0]iData	128bit CAN message data received from input FIFO.
iReset	Reset signal. Reinitializes internal registers.
empty_i	Received from FIFO. Is high if no data is in FIFO.

Output

oReadRequest	output communicates ReadRequest state with io adapter module

Other outputs are currently undefined, however the documentation discusses how they should be included. Currently in the process of setting up outputs.

Parameters

The Following parameters match the registers and values required to interact with the SPI device. See the PDF's in the reference section for more information.

```
//////// SPI register addresses  ////////
TRANS_TYPE_REG = 5'h2;
TRANS_CTRL_REG = 5'h3;
TRANS_STS_REG = 5'h4;
TRANS_ERROR_REG = 5'h5;
TX_FIFO_DATA_REG = 5'h20;
SD_ADDR_7_0_REG = 5'h7;
SD_ADDR_15_8_REG = 5'h8;
SD_ADDR_23_16_REG = 5'h9;
SD_ADDR_31_24_REG = 5'ha;
//////// SPI Values ////////
SPI_INIT_SD = 2'd1;
SPI_TRANS_START = 1'd1;
TRANS_BUSY = 1'd1;
INIT_NO_ERROR = 2'd0;
SPI_RW_READ_SD_BLOCK = 2'd2;
WRITE_NO_ERROR = 2'd0;
```

Registers

As the module is still in alpha, the register assignments are in flux.

Wires

spiBusy	Connected to *spi_io_adapter.* Determines if the SPI device is busy.
[7:0]rdRegData	8bit wire. Data returned from *spi_io_adapter* (SPI register value).

Internal Description

More testing is required to determine if the current state machine is working properly.

Module Usage

Dependencies

The *spi_controller* module instantiates the *spi_io_adapter* module.
The module expects a FIFO to be connected to the corresponding I/O ports (iData, empty_i). Various input clock rates are required to control interaction between the FIFO and the serial peripheral interface and more testing is required to determine exact rates.

Instantiation example

Not yet implemented

Test case description

None

Additional Information

Problems section

The spi_controller module has not yet been tested, thus correctness and accuracy cannot be verified.

References

spiMaster_FSM.pdf, spiMaster_specification.pdf

FPGA Data Monitor for Prosthetic Limb System Documentation

Module: spi_io_adapter<ALPHA>

Justin Deschenes, Jeremy Williams, Alexander Bochem

Faculty of Computer Science
University of New Brunswick
Fredericton, Canada

Revision Table

Changes	**Revision**	**Date**
Module Description, Input, Output, Parameters, Registers, Internal Description	0.1	3/25/2010
revision changes	0.11	6/9/2010

Module Description

The *spi_io_adapter* module instantiates the spiMaster core. The *spi_io_adapter* module is responsible for reading, writing and working out the timing issues between the *spi_controller* and the *spiMaster.* The *spi_controller* utilizes the wishbone interface of the *spiMaster* allowing simplified communication and timing.

Input

iClk	System clock signal
iSPIClk	System clock for SPI interface (wishbone?)
iReadRequest	Read request from spi_controller to spiMaster registers
iWriteRequest	Write request from spi_controller to spiMaster registers
[7:0]iAddr	Address bus for read/write request to spiMaster module
[7:0]iDataWrite	Data bus for write request to spiMaster module
iReset	System reset signal

Output

[7:0]oDataRead	Received value from spiMaster register after READ req.
oBusy	Status of whether spi_io_adapter is busy(high) or ready(low)

additional outputs are expected to be added soon. These outputs would correspond with the physical parameters of the SPI device.

Parameters

Only state machine parameters exist in module.

Registers

wb_free	Internal status flag, HIGH= while can_io_adapter is busy
wb_we_i	Input signal to spiMaster, write register enable
wb_cyc_i	Input spiMaster, "control signal"
wb_stb_i	Input spiMaster, "control signal"
[7:0]wb_adr_i	Input address bus to input address bus to spiMaster
[7:0]wb_dat_i	Input data bus to can_top, write register value

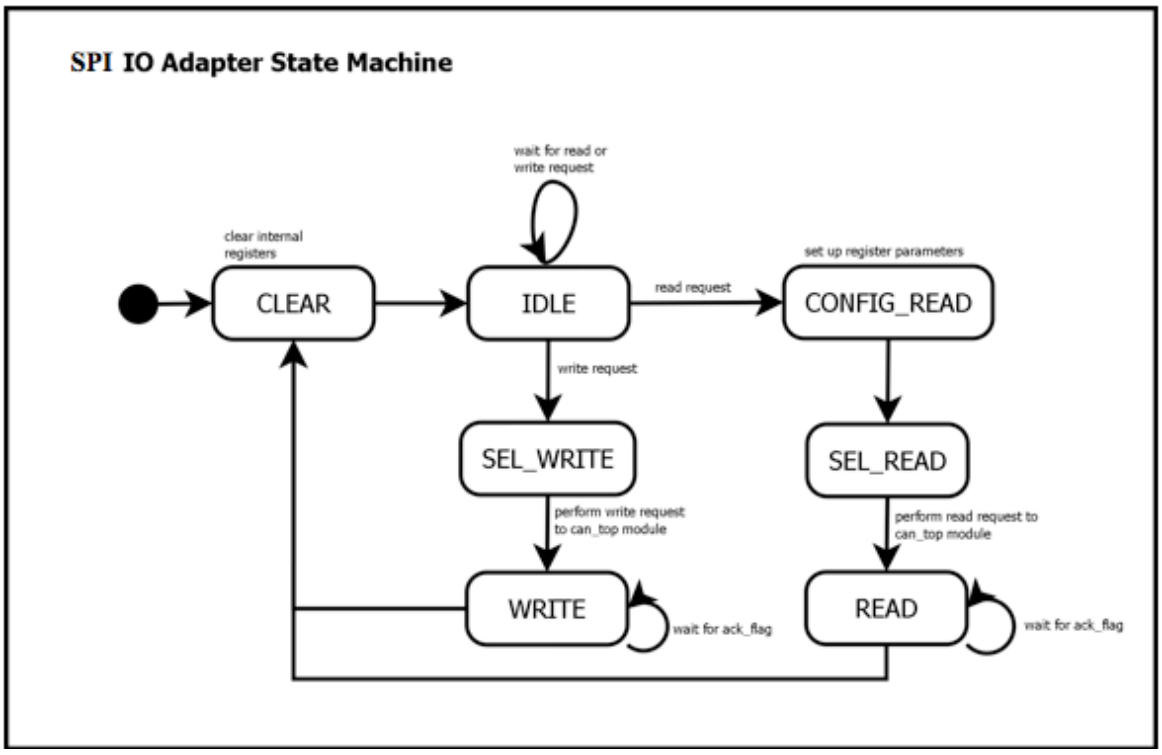
Wires

wb_ack_o;	Determines accessibility of spiMaster module, high = rdy, low= busy
[7:0]wb_dat_o;	Output data bus from spiMaster, results from read register
wb_clk_i;	Clock for wishbone interface in spiMaster mod., currently == SPIclk
wb_rst_i;	Reset signal for wishbone interface in spiMaster module

Internal Description

The *spi_io_adapter* handles read and write requests from the *spi_controller* module and handles interface requirements for the wishbone bus interface on the *spiMaster* module.

The following diagram defines the functional operation of the *spi_io_adapter*.



Module Usage

Dependencies

The *spi_io_adapter* module instatiates the spiMaster module. The module expects the spi_controller module to connect to all the I/O ports.

Instantiation example

```
spi_io_adapter spi
(
       .iClk(pin name for system clock),
       .iSPIClk(pin name for SPI clock),
       .iReadRequest(wire for read request from controller),
       .iWriteRequest(wire for write request from controller),
       .iAddr(8 bit wire for passing address info to device (from
       controller)),
```

```
        .iDataWrite(8 bit wire for data to be written to device (from
        controller)),
        .iReset(pin name for global reset),
        .oDataRead(8 bit wire for passing information to
        controller),
        .oBusy(wire connected to controller passing busy status)
);
```

Test case description

None

Additional Information

Problems section

The spi_controller is in alpha, so issues could arise specifically with interfacing. This module uses the wishbone interface and because of this, this module should be in decent shape.

References

spiMaster_FSM.pdf, spiMaster_specification.pdf, Wishbone specification document.

Future Developments

Compression

It would be beneficial to have compression so that the bandwidth of the system could be used more effectively. The compression core would either need to be developed in house which could be time consuming or purchased from a third party which could be costly. Either way the compression would need to be fast enough so that it does not become a bottleneck for the system. Ideally the core would take in a stream of data and output a stream of compressed data. This could plug directly into the system we have built. If that is not feasible a system could potentially be devised where data gets buffered and then compressed after the buffer gets filled but this method would require some reorganization of the system.

Wireless

It would be useful to have wireless functionality so that the prosthesis does not need to be tethered to a computer to retrieve the logged data. It would straight forward to include this functionality. The biggest hurdle to overcome is to understand how to communicate with the wireless controller on the hardware level. Most of the theory for interfacing with the controller will probably be similar to that of the flash memory. But the flash memory uses the Wishbone interface and the wireless chip uses a different interface.

CAN controller configurability

At the moment all the values for the CAN controller are hard coded and thus can not be changed after the design has been synthesized. This could be improved in several ways ranging from full configurability at runtime to a bit of code reorganization so that the values can easily be changed for re-synthesis. Full configurability at run time would be a lot of work and is probably overkill for this project and what needed. A compromise could be the ability to configure a small number of values at runtime. The most time effective solution would be to reorganize the code so that the hard coded values could easily be changed and then the design could be re-synthesized with the new values.

Filtering configurability

The filtering is currently fairly rudimentary and inflexible. It would be advantageous to have more flexibility with how to filter messages. Reconfiguring the filters, and turn some filters on and some off would be helpful. This becomes even more crucial as the design moves from lab testing to real world testing where re-synthesizing the design becomes less feasible. How this would actually be implemented would still needs to be thought through. The system could potentially be a memory mapped device and masks could be stored in registers which could be written to by a micro controller. A simpler solution would be to have some sort of persistent memory, a couple kilobytes of flash memory to which masks could be written, to be used during the filtering of the messages.