Super IO

6.111 : Introductory Digital Systems Laboratory

Ross Glashan (rng@mit.edu)
James Wnorowski (jamwno@mit.edu)

TA: Chris Forker

May 12, 2005

Abstract

This report details the implementation of the Super IO controller chip. The Super IO is a FPGA-based Input/Output controller chip designed for use on small autonomous robots. A host processor communicates with the Super IO over a serial bus, giving the Super IO control commands. The Super IO drives a variety of modules including motors, servos, shaft encoders, push buttons, analog inputs, and an LCD. These modules make the Super IO incredibly scalable, allowing it to control hundreds of IO devices.

1 Introduction

The 6.270 Lego Robot Competition[1] is a competition run every IAP where students build and program Lego robots to compete in a special game. The "brain" of 6.270 robots is a board known as the Handyboard. The Handyboard was designed in 1995, specifically for 6.270, and performs admirably at the task of running lego robots. As 6.270 grows however, demands placed on the Handyboard have begun to exceed its capabilities. The board is beginning to show its age, and so this project details the core of what we hope will become the future 6.270 board.

One of the major problems the Handyboard suffers from, is a relatively small processor (68HC11). This processor performs all the task on the board - running user code, driving motors, servos, LCD, etc. Thus, when the processor is under heavy load, servos begin to jitter, motors start to jerk and sensor input becomes unreliable. To solve this problem, we designed the Super IO, a dedicated high speed Input/Output controller chip. The new Handyboard will contain a processor (to run user code), and a Super IO (to drive the motors, servos, sensors ...). This architecture is shown in Fig. 1. This ensures that no matter what happens to the host processor, the robot's sensors and actuators will work perfectly.

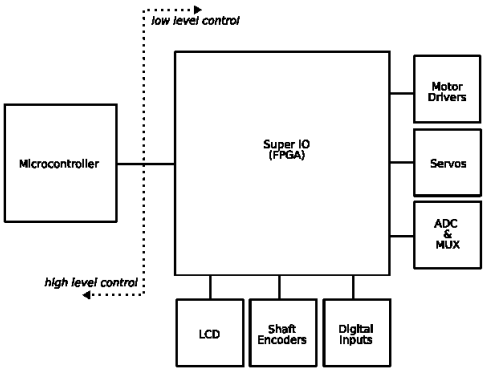

Figure 1: Super IO Usage Scenario

The Super IO off-loads as much processing from the host processor as is practical. For this implementation, all the waveforms required to drive motors, servos and the LCD are generated in the FPGA. Sensors also automatically polled. The Super IO even includes built in velocity feedback for the motors, a feature that normally required a sizeable portion of dedicated code on the Handyboard.

Our hope is that with the Super IO, contestants can completely ignore the complexities of low level IO control, and instead focus on the writing code to make their robots win the competition.

[1] http://web.mit.edu/6.270

2 Overview

Our design communicates with the user-programmable microprocessor by way of a serial interface. An address byte and a data byte are sent through the serial interface to the serial bus controller. The serial bus controller will relay this information over the shared bus. Each modules register controller will check the address byte to see if it is the module that is being accessed. Our modules include a motor controller, a servo controller, a shaft-encoder, digital inputs and outputs, an analog input, and a Character LCD.

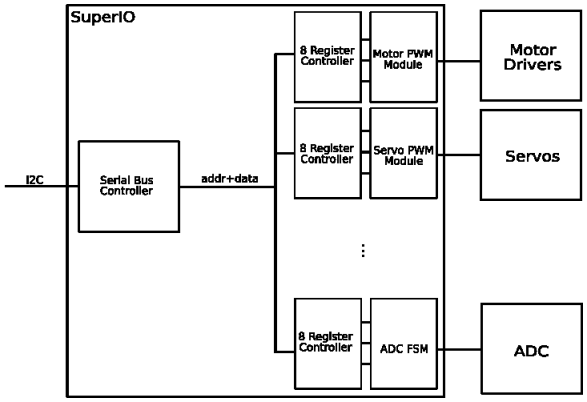

Figure 2: Super IO Block Diagram

3 Module Description

3.1 Serial Bus Controller

The Serial Bus Controller forms a bridge between the host processor serial bus and the internal parallel bus.

Initially we planned to use I^2C as the bus connecting the Super IO to the host processor. An I^2C slave module was implemented and simulated correctly. Sadly, the module did not perform as expected in hardware, likely due to resistor pull-up issues on the bus lines. In order to save time, we replaced the I^2C bus with a simple RS232 module which emulates the functionality of the I^2C module. We plan to debug and implement the I^2C module at a later date.

The Bus Controller is remarkably simple. It is essentially a wrapper around a pair of RS232 UART[2] modules. The controller emulates the functionality of I^2C very closely. The communication protocol is shown in Fig. 3. Each message is 3 bytes long.

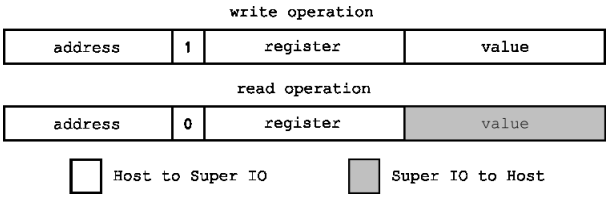

Figure 3: Super IO - Host Communication

The first byte contains the address of the Super IO and a read/write bit. The following byte contains the address of the register to read from or write to. The final byte contains the register data. The state machine receives the first 2 bytes, then either reads or writes a byte over the internal bus.

3.2 Register Controller

The register controller is the module that goes between the serial bus controller and the other modules. There is a separate instantiation of the register controller for each module. Each register controller has eight 8-bit registers, although most of the modules do not utilize all of them.

The serial bus controller sends eight bits of address and eight bits of data down the bus to all of the register controllers. The five high order bits address a specific register controller, while the three low order bits address individual registers in the register controller.

[2]RS232 UART from http://www.fpga4fun.com/SerialInterface.html

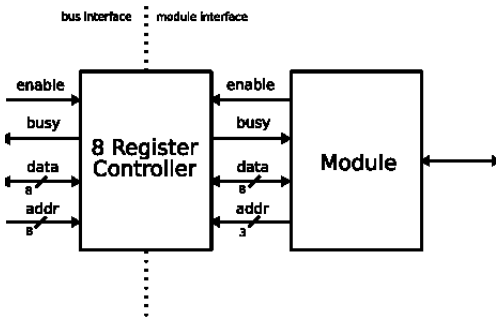

Figure 4: Register Controller Operation

When the serial controller writes to a register, it sends an enable signal along with the address and data. Upon receiving the enable, the register controller will then send a busy signal to the module. This busy signal serves two purposes. First, it prevents the module from reading from or writing to the register controller while it is interacting with the bus. Second, by detecting the falling edge of this busy signal, the module knows that the register controller has been updated, and can then request the new data.

The register controller can also receive data from the modules. In this case, the module will send an enable and then write to the register controller. However, the register controller cannot write to the bus, so it must wait for the serial controller to poll it.

3.3 Generic Module

All of the modules used in our system were built around a single skeleton module. Since all the modules have to either read or write from the register controller our skeleton module incorporated a both read and write functionality. The simple FSM for this module is shown in Fig. 5. When we needed to implement a new module, the skeleton code was copied, the register names were changed, and the external actuator/sensor code was added. This allowed us to add new modules quickly, and with relative easy. In fact, we plan on releasing the skeleton code to new contestants, so that they may augment the functionality of the Super IO with special actuator/sensor support.

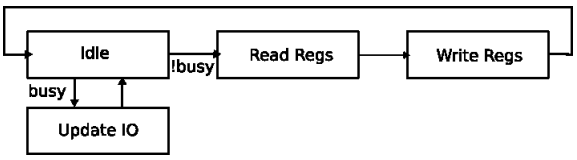

Figure 5: Generic Module FSM

The generic Read code revolves around the busy signal coming from the register controller. This signal is high whenever the register controller is being accessed from the bus. Thus as soon as it goes low, we assume a value in the register controller has changed and we perform a read, updating our local copies of the registers.

For the generic write module we simply wait until we have a new value to send to the register controller, check for the busy to be low, and then write the data.

3.4 Motor Module

The motor driver module has 3 registers, 2 for controlling the velocity and direction of the motor, and 1 to control feedback. The registers are shown in Table 1.

Addr	Name	R/W
0	Vel	W
1	Dir	W
3	Feedback	W

Table 1: Motor Module Registers

The motor driver module contains two main components - a PWM driver and a feedback controller. The PWM driver outputs a 100Khz square wave with a variable duty cycle, controlled by the *vel* register. This signal is passed to the enable signal of a L293D H-bridge connected to the Super IO. The *dir* register is passed directly to the direction signal of the L293D. These two signals allow the Super IO to driver the motor in both directions and at variable speed.

The feedback module provides closed loop control of the motors, using a shaft encoder module as an input device. When a motor module is instantiated in the Super IO, it is always linked to a corresponding shaft encoder module by a special 8 bit bus. This bus provides the motor module with the current velocity of its corresponding shaft encoder. When feedback is enabled, the motor module compares the control velocity (the *vel* register) to the actual velocity (from the shaft encoder) and then adjusts the PWM velocity accordingly. This ensures that the motor velocity is exactly the value of the *vel* register, making for more accurate robot control.

3.5 Servo Module

The servo driver has only 1 register, a position. This value corresponds to the angle of the servo (between 0 and 180 degrees). The register map is shown in table 2. Since each servo only uses 1 register, 8 servo drivers could be placed in a single module.

Addr	Name	R/W
0	Servo 0 Pos	W
...	...	...
7	Servo 7 Pos	W

Table 2: Servo Module Registers

A servo is controlled by a PWM signal similar to the motors, except that the duty cycle now relates to position, not velocity. Servos also place more stringent requirements on the PWM waveform. Firstly the the PWM frequency needs to be approximately 50Hz. More importantly, the portion of the square wave must be between 0.5ms and 2.5ms - corresponding to 0 and 180 degrees rotation. The servo module is simply the motor module code, copied and modified to take these specifications into account.

3.6 Analog Input Module

The analog input module uses an ADC to take an analog signal and convert it into an 8-bit value. We decided to use an analog multiplexer at the inputs, so that we could take up to sixteen analog inputs with only one ADC.

The analog input process starts with a request from the serial bus controller. It will send a data byte corresponding to the multiplexer input value to be selected to the register controller connected to the analog input module. The analog input module will then use that value as the selector for the multiplexer, and send the multiplexer output through the ADC. Once the ADC has finished its conversion, it will write the value to the register controller. At this point, the analog input is identical to the digital input, and the serial bus controller will poll the analog input controller for its value.

3.7 Digital Input Module

The digital input module uses 1 register per 8 inputs, thus a single 8 Register Controller could sample up to 64 inputs.The register map is shown in table 3.

Addr	Name	R/W
0	Digital inputs 0-7	R
...	...	...
7	Digital inputs 56-63	R

Table 3: Digital Input Module Registers

The input module continuously polls the inputs and when a change occurs (and the busy signal is low), the new input value is written to the register

controller. For the push-button input module, all of the inputs are fed through switch debouncers[3] before being passed to the input module.

3.8 Digital Output Module

The digital output module is very similar to the digital output module - the code is very similar, the only change being read operations have changed to writes.

3.9 Shaft Encoder Module

The shaft encoder module has 3 registers, 2 for shaft position, and 1 for velocity. Since shaft position changes very rapidly (up to 100 counts per second), 2 registers are used to hold the current position. The register map is shown in table 4.

Addr	Name	R/W
0	Pos Hi	R
1	Pos Lo	R
2	Vel	R

Table 4: Shaft Encoder Module Registers

The shaft encoder counter is based on a single 16 bit register which is clocked off the encoder input signal, incrementing its value at each encoder tick. The register controller is updated on a 1KHz clock. On each of these ticks the position value is written to the register controller. The velocity is calculated using a position delta and then is written to the register controller as well. The velocity value is also passed out to the related motor module.

3.10 LCD Module

The character LCD module interfaces with a 1602A Character Liquid Crystal Display. This is a 16 character by 2 line LCD that has been used with the Handyboard in the past. This LCD follows the standard character LCD protocol. There are 256 addresses for characters, and a subset of those addresses actually contains valid characters. Some of the characters are actually user-programmable.

Our LCD responds to the HD44780 instruction set. This protocol uses ten inputs: a read/write, a register select, and the eight data bits. Since we never read information from the Character LCD module, we can safely leave the read/write

[3]Debouncer from http://www.opencores.org

pin grounded. The register select bit, however, needs to be low when issuing a command byte and high when writing a character to the display. This is somewhat problematic, as the bus is only eight bits wide. To accommodate this, we used one of the addresses that did not contain a valid character to send a clear screen command. When we received a new data byte, we first checked to see if it was the reserved command byte or just a character to be displayed, than acted accordingly. This method can be extended to more command characters, so that users can change the cursor style or other options.

Another issue we ran into with the display was the fact that the HD44780 instruction set assumed that a two line display was 40 characters to a line. The display would not automatically wrap lines, and would instead write to memory locations that did not display to the screen. To correct this, the Character LCD module keeps count of the number of characters written to the display, and automatically sets the address to the next line once sixteen characters have been written.

4 Testing and Debugging

The first two elements that we designed were the serial bus controller and the register controllers. We tested these modules by looking at the waveforms that were generated in the simulator and comparing them to our specifications. We then tested them together, checking to see if the serial bus controller could read and write from a register controller using the shared data bus. Once this was successfully debugged and tested, we knew that the core of our design worked.

Since each component module was isolated from the data bus by a register controller, we just needed to insure that the individual modules interacted correctly with a register controller. Thus, we were able to set up a testbench that included the module to be tested, a register controller, and a simple serial bus controller that would write test data to the bus at a specified time. This made testing and debugging each component module relatively simple.

During the implementation of our project, we ran into a timing issue. While writing the code for the Analog Input, we reused the parts of the FSM controlling the ADC chip from Lab Three. When we tested this functionality in the new labkit, however, the ADC did not give us valid data. We soon realized that the timing constrains in Lab 3 were based on a 10 MHz clock, while the new labkit ran off of a 27 MHz clock. When we adjusted the setup time in the FSM to account for this, our ADC gave us valid values.

Finally, when working on a project in a team, it is important to make sure that people share a set of conventions, and stick to them. Much debugging time would have been saved if we had agreed on an order of variable declarations inside our component modules and adhered to it.

5 Conclusion

We feel the initial prototype of the Super IO was a remarkable success. Our implementation allowed us to reliably control over 10 IO devices, with minimal resource usage on the host processor. All this done with only 3% usage of the FPGA. The highly modular design allows for the Super IO to be scaled up or down - from a tiny single motor and shaft encoder controller chip, up to a 6M Gate Super IO controlling hundreds of IO Devices. The main issue we would run up against with a scaled up version of the Super IO, is that the internal bus would begin to saturated with an exceedingly large number of modules. I don't think this would be a problem until we reach around the 100 Module range.

6 Acknowledgments

We would like to thank the following people for their help and support during the development of this project:

- *Professor Chandrakasan* - for great lectures and help in lab.
- *Nathan Ickes* - for an incredible new lab kit.
- *The 6.111 TAs and LAs* - for spending endless hours in lab, and solving our problems.
- *Keith Kowal* - for being generally awesome
- *Our fellow 6.111 students* - for not getting annoyed by the incessant whine of Lego motors
- *6.270* - for the inspiration