Alpha Design

Final Report

3-in-1 Game with Remote

Group 7
Project Members

Michael Shoniker
shoniker@ualberta.ca

Timothy Bauman
tbauman@ualberta.ca

Table of Contents

Abstract

The goal of this project is to implement a 3-in-1 game system using the Altera DE2 board. The board will be programmed to display menus and games to an external LCD and receive user input from a remote control. The LCD will have a power circuit to power the back light capabilities as well as adjust screen contrast. Specifically is will be a 4x20 character LCD that will be interfaced to the DE2 over GPIO connections. To receive user input via a remote control, the on-board Agilent HSDL-3201 low power infrared transceiver will be used. The infrared (IR) signals will then be passed to FPGA instantiated components for decoding. Decoded IR signals will then be used by the NIOSII soft processor to interpret user commands. Users will be able to select from three games in a main menu by cycling through them using corresponding “next” or “previous” button pushes. The three games that were implemented are Blackjack, Skill Tester, and Repeat After Me. After each play though of a game the user will be prompted to continue or return to the main menu. The user will also be able to return to the main menu at any type by simply pressing the “exit” button.

Functional Requirements

The device must be able to receive and properly decode IR signals. This was accomplished with the combination of the on-board IrDA receiver and an FPGA instantiated decoder component. The component was creating using VHDL and converts a serial NEC IR standard protocol signal [1] to a 32-bit parallel output command. The NIOSII system then checks the received 32-bit command for validity using a hard coded mapping of all possible button pushes. If a command is not valid the system will reject it and wait for valid input. In this regard our system operates almost flawlessly. The only known issue is that, as the remote moves out of range, our decoder can become unsynchronized from the incoming signal. This is caused by not receiving an entire NEC signal and puts the decoder in an erroneous state. This is a rare occurrence, but has been addressed. A decoder reset has been implemented with a switch on the board, SW(0). A simple toggle of the switch resets the decoder and resynchronizes it. This has no effect to the rest of the NIOSII system and the system retains the last valid command.

The device must be able to display to an external LCD that has a back light. Originally the need of a back light was conceived for use in low light environments and to add a bit of flare to the device. The LCD chosen was a versatile 4x20 Seiko L2034 display that allowed us to display more than double the number of characters than the on-board 2x16 LCD. The GPIO connection was able to meet voltage and current requirements for the normal operation of the LCD, but not the back light operation. This was due to high power consumption of the yellow back light LED. Instead a DC power supply was used to meet the 480mA and 5V requirements of the back light. Power requirements for the LCD can be viewed below in Fig. 1 and Fig. 2.

Ta = 25°C

Item	Symbol	Specifications	Unit
LED forward current consumption*	I_F	480	mA
LED reverse voltage	V_R	8	V
LED allowable dissipation	P_D	2.0	W

Fig. 1 LED Requirements [2]

V_{DD} = 5 V ± 5% V_{SS} = 0 V Ta = −20°C to +70°C

Item		Symbol	Conditions	Min.	Typ.	Max.	Unit
Power supply voltage		V_{DD}		4.75	5.00	5.25	V
		$V_{DD}-V_{LC}$		4.0	–	11.0	V
* Input voltage	High	V_{IH1}		2.2	–	V_{DD}	V
	Low	V_{IL1}		0	–	0.6	V
** Output voltage	High	V_{OH1}	$-I_{OH}$ = 0.205 mA	2.4	–	–	V
	Low	V_{OL1}	I_{OL} = 1.2 mA	–	–	0.4	V
Current consumption		I_{DD}	Ta = 25°C, V_{DD} = 5 V, V_{LC} = 0.2 V	–	2.9	4.0	mA
		I_{LC}		–	1.2	2.0	mA
Clock oscillation frequency		f_{osc}	Resistance oscillation	140	220	300	kHz

Fig. 2 LCD Electrical Characteristics [2]

Interfacing the NIOSII system with the LCD is done using a simple parallel input/output (PIO) component in the Quartus SOPC tool set. This component directly maps to 11 GPIO pins, to which the LCD is connected. By having this SOPC component it allows our NIOSII system software to directly interact with the hardware. We used CMPE401 lab supplied LCD driver code from Nancy Minderman as basis for our system's operation. The code was modified to work in a µC/OS-II environment and the C++ files were converted into C to be successfully integrated into our project.

A major issue found while interfacing this LCD component was the timing requirements. At first, we were able to successfully power the LCD and clear the screen, but it seemed like all other commands sent were not producing desired responses. We consulted the data sheet and lab instructors for clues as to what was causing this issue. After much thought, and trial and error, we were convinced it was a signal timing issue. With some experimentation we were able to meet the timing requirements a described in Fig. 3.

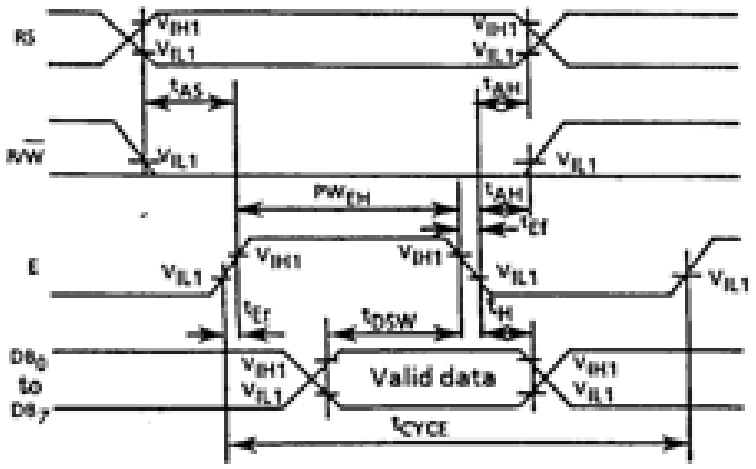


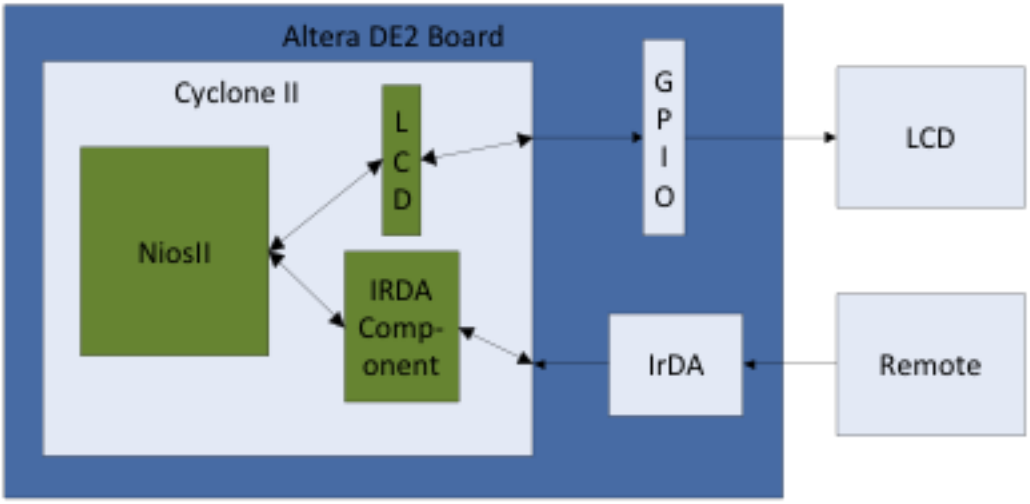
Fig. 3 LCD Write Operation Timing Requirements [2]

Our system meets all functional requirements for the LCD and has no known issues.

Requirements for the game aspect of our design include having a main menu where games could be selected, multiple games available and that each game can be exited and returned to the main menu. The software components we have created adhere to these requirements. The menu allows the user to cycle through all the games, and returning to the menu at any time has been achieved. At this point all bugs found during testing have been dealt with and there are no known issues with the game operation. However, while users can cycle through and enter each game, only one game is functional at this time.

Design and Description of Operation

To accomplish the production of a 3-in-1 game system this project was comprised of an LCD screen, remote control, and the Altera DE2 board. The DE2 board was the glue between all the pieces and the core of our device. User commands are given via the remote control and are executed on the DE2 board by the CycloneII FPGA. We chose to use the Altera supplied NIOSII soft processor as the heart of our embedded system on the FPGA. The CycloneII will communicate with the on-board IrDA to receive all incoming IR signals. Custom created IR components are implemented on the CycloneII to decode IR signals and feed them to the NIOSII processor.



System Overview

The functional operation of this project is to display a welcome screen on the LCD upon power up and then display a main menu. The main menu screen allows users to cycle through the three playable games from corresponding “next” and “previous” button pushes on the remote. To select a game the user must press “OK” and the game will begin. Each game is different, but all have similar functionality. They all require user input from the remote and all respond to the “EXIT” command. The “EXIT” button push will return the user to the main menu at any point while within a game. At the end of each play through the user will be prompted to play again by pushing “OK” or to return to the main menu by pushing any button that is not “OK”.

IrDA Module:

Our IrDA soft core design is based off of vhdl from Xilinx Corp. [3] The soft core IrDA module receives a 38 KHz signal and samples it at a 1780 Hz frequency. The signal processing is done in two steps. The first module smooths the signal out buffering the 38 KHz IrDA pulses into its high and low signals. The second module samples the buffered signal, at 1780 Hz, 128 times and outputs a 32-bit vector to the NIOSII processor.

The IrDA module does not send IrDA signals.

Operation:

The IrDA buffer runs off of a 16x clock, so for our purpose we feed it a 608 KHz clock signal. The IrDA decoders serial output goes low as each IrDA pulse is received. Upon receiving a falling edge from the IrDA pin the decoder starts a count to 16. If a new falling edge is not received within that time the serial output of the decoder goes high. The serial output is then inverted and passed into the serial to parallel portion of the module.

On the Cyclone II FPGA our IrDA SP converter operates off of a 28.5 KHz clock. Like the IrDA serial decoder, the serial to parallel converter operates on a 16x clock. The NEC signal comes in at a frequency of 888 Hz and we sample the signal at 1780 Hz to obtain 128 bits. The sampling of the serial NEC signal is done in two 64 bit intervals. These 128 bits are parsed in the top level.

The top level of the IrDA module runs off of the same 50 Hz clock that the NIOSII processor runs off of. The IrDA top module is responsible for scaling down the clock signals for the two stage IrDA signal processing. The top level of the IrDA also

generates an interrupt every time 64 bits are processed and are ready to be read.

To interface the IrDA module to the nios II processor we designed some top level vhdl. This top level vhdl receives two 64 bit vectors from the IrDA module and parses out 32 data bits to be sent to the nios II processor. These 32 bits are chosen to provide a one to one mapping of all the buttons on the remote into the nios world. This top level code writes 0x00000000 to the IrDA's register between successful IrDA receives so we are able to poll the register from software. The IrDA's interrupt is also fed into the nios II processor but due to technical difficulties we do not use it.

Hardware Requirements

1. Altera DE2 board - 8 MB SDRAM
- 512 KB SRAM
- 4 MB Flash
- IrDA Transceiver
- 50 MHz clock

2. LCD with back light
- Black on green 5 V
- 4x20 Character LCD
- Due to current/power requirements for the back light, the LCD's LED circuit was powered by a DC power supply
- The rest of the LCD was able to be powered directly from the DE2 GPIO pins

3. Philips 4-in-1 Remote Control
- Carrier frequency range: 24 - 55 kHz - Operating distance: 33 ft (10 m)
- Transmission angle: 45 degree
- Transmitting LEDs: 1
- Universal IR code database
- Our system focuses on the NEC standard which is specific to many tv remotes, so our system was configured to operate on when the remote is in "tv device mode"

Parts List

Part	Estimated Price ($)
LCD screen	Lab supplied stock (~15)
Remote control	15
Total	~30

LCD
Seiko L2034B1J000 4x 20 Character LCD
http://ca.mouser.com/ProductDetail/Seiko-Instruments/L2034B1J000/?qs=7GPkEwWWW0en4BEFfeNIzw%3D%3D

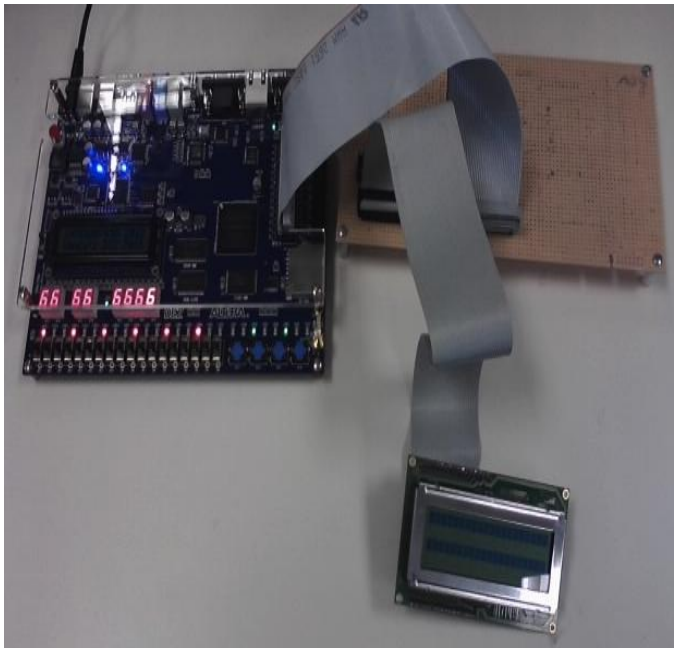
Remote Control
Philips 4-in-1 Remote Control (SRP4004)
http://www.futureshop.ca/en-CA/product/philips-philips-4-in-1-remote-control-srp4004-srp4004-27/10178381.aspx?path=e348c57c40a38786417de3d84d72f941en02

Device IO

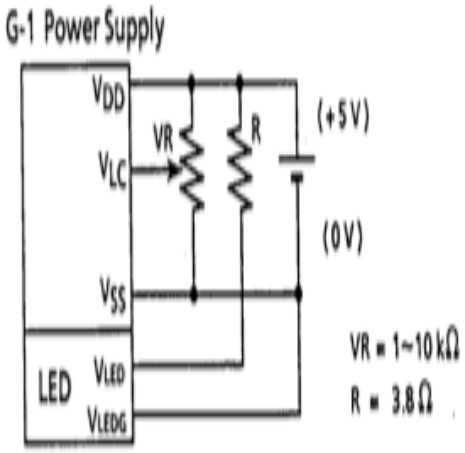
This project has two main components to its I/O. They are the LCD and the IrDA.

LCD

The Seiko L2034B1J000 4x 20 Character LCD will be used for displaying messages to the user. It will be connected to the DE2 board via GPIO pins. The 5V power supply provides sufficient current draw at 2.2 mA for basic screen operation, but a secondary power supply is required to provide 480 mA current consumption for the backlight LED. A secondary power supply will be built on perfboard using a 9V 500 mA power supply and a voltage regulator.



LCD Connected to DE2

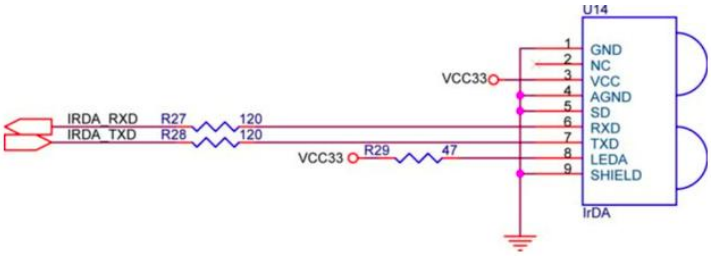


LCD Power Suppy Circuit Diagram

IrDA

User commands will be in the form of button presses on the Philips SRP4004 remote control. This remote has 1 transmitting LED capable of operating at a distance of up to 10 meters and has a signal carrier frequency range of 24 -55 kHz. The DE2 board is able to receive this signal by the on-board Agilent HSDL-3201 low power infrared transceiver.

Philips Remote



IRDA Schematic

Software Design

We implement the functionality of our 3 in 1 game box using the RTOS uC OS-II. uC provides a real time multitasking environment and is easy to use making it an ideal choice for our design. We uses uC's semaphores, queues and mutexes extensively to synchronize the operation of our system. We use one semaphore, three mutexes, and two queues. Overall our system makes use of seven priorities: two, four and five are mutexes. Priorities one and three are used by the remote driver and six and seven are used by the game tasks.

Remote Driver:

The remote driver consists of a main task a callback task and a mutex which is cooperatively shared amongst tasks. The remotes highest priority task polls the IrDA's 32 bit data register and checks to see if the value has changed and checks to see if the client task is waiting for a button push. The mutex is set at priority 2. Every time a task desires to read a button press it pend's to this mutex and in doing so acquires the second highest priority in the system. The second main task of the remote driver at priority 3 handles special signals from the remote and calls callback functions provided by the remotes client task.

Design Choices:

We chose to set the priority of the remote higher than any task so that we would have assurance that any button push will not be ignored. This allows for callback commands to be sent to the client when the client is not pending on a button push. The callback task is set at one priority lower than the remote mutex so that the client's callback functions will be able to pend on the remote without deadlock occurring. We set the task priority of the callback task to one lower than the remote mutex to ensure that it will be the next task to acquire the remote mutex.

Data Flow:

The remote and client tasks synchronize with each other via two queues and one semaphore. One queue forwards button signals to the client task and the other queue forwards signals to the callback task. The semaphore is posted to by the task requesting the next button signal.

To receive a signal from the remote a task must first take control of the remotes mutex and acquire its priority. This ensures that the task will execute in a timely fashion and promptly release the remote. Then the task post's to the remotes semaphore to request a button and waits for a signal to arrive on the button queue. The remotes main task monitors the semaphore and posts the latest remote signal to the button queue when on is requested.

There are two designated buttons which will initiate a callback routine: the "exit" button and the "back" button. The call back task pends to the callback queue. In the meantime the main remote task monitors the IrDA register for these special signals and posts them to the callback queue when they occur. Upon receiving a button signal from the main task the callback task pends on the remote mutex waiting for the latest task to give up the remote. The callback task then seizes the remote mutex and ensures that the callback functions are executed.

Games:

There is one main game task which hosts all other tasks. To ensure only one game task runs at a time we have each game task pend on a mutex before they begin. We have three games each implemented as their own task which will be created and deleted at the user's request.

Game Host:

The game host task displays the welcome screen and allows the user to sift through the games. At the user's command the host task creates the specified game task, posts to the game mutex to allow the new game to run and then pends back to the game mutex to wait for the game to finish.

The game host is responsible for initializing the remote driver and the LCD. The initialization function takes two function pointers to the callback tasks to be called when the “exit” and “back” buttons are pushed. We pass the same callback function to the remote for both “exit” and “pause” button presses. The callback function does two things: it signals the remote task to only send “EXIT” signals which are handled by the game task, and it sets a game status bit which can be checked by the current game task. The game task can rely on both of these mechanisms to decide when it should end.

Game Tasks:

Each game is implemented as a task which is created by the game host. Each game task is responsible for deleting itself and releasing any resources they have acquired. Each game task must be aware of when it must delete itself. There are two mechanisms provided by the game host which it can use for this purpose. The game task can rely on the remote to send a consist stream of “EXIT” signals, or the task can call a function supplied by the host to check if it must delete itself. The task then exits its main loop, releases the game mutex and deletes itself.

LCD:

The external LCD is wired up to the NIOSII via the GPIO pins. The NIOSII processor then drives these lines to write to the LCD.

LCD Driver:

The LCD driver we use is a modified version of our lab instructors code used with her permission. [4] The LCD is governed by a mutex which is pended to every time a task writes to the LCD ensuring only one task is controlling the LCD at a time.

Test Plan

Software:

We tested the software as we wrote it. The remote was verified by a test function which displayed all the received button pushes to stdout. We found every button was mapped correctly except for the “enter” and “blue” buttons. These two buttons are decoded to be the same. See design and functionality and future Work for more about this bug.

The LCD code was developed from Nancy Minderman’s code and was tested as we modified it. It works as expected.

The game tasks were tested with input from the remote.

Hardware:

The IrDA component was tested and verified if modelsim and the LCD was tested with a modified version of Nancy Minderman’s code.

Integrated Circuit Design

Our integrated circuit was comprised of all of our FPGA instantiated components excluding the NIOSII system components. These were:

- niosII_microc_lab1.vhd (this is a modified top level component with no reference to the NIOSII system)
- clock_scalar.vhd
- irda_xlinx.vhd
- irda_com.vhd
- jk_ff.vhd
- sirendec.vhd

To perform analysis on our design we used Synopsys, that was licensed for the ETLC 5013 AICT lab.

This is our .tcl script used to compile, synthesize, place, route and verify our circuit:

```
analyze -format vhdl ./niosII_microc_lab1.vhd
analyze -format vhdl ./clock_scaler.vhd
analyze -format vhdl ./irda_xlinx.vhd
analyze -format vhdl ./irda_com.vhd
analyze -format vhdl ./jk_ff.vhd
analyze -format vhdl ./sirendec.vhd
```

```
elaborate niosII_microc_lab1
check_design -multiple_designs
uniquify
create_clock "CLOCK_50" -period 20
current_design niosII_microc_lab1
compile -map_effort high -boundary_optimization
report_area
report_power
report_timing -path  full -delay max -max_paths 1 -nworst 1
```

A full copy of the report produced by this script has been included in the IC Design section of the Appendix. Although all files were successfully compiled and run, the area, power and timing reports indicate that the system has not been correctly synthesized. The area report shows 0 area required and similarly the power report shows 0W consumption. Please note that the Appendix section also includes a schematic view of the irda component.

We could not determine our error so instead preformed the analysis again, but only on our clock scaler component. A schematic view and corresponding reports have been included in the appendix section.

```
Area report generated:
Number of ports:              35
Number of nets:              343
Number of cells:             148
Number of references:         12
Combinational area:         0.000000
Noncombinational area:      0.000000
Net Interconnect area:      undefined  (No wire load specified)
Total cell area:           0.000000
Total area:                undefined

Power report generated:
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1nW
  Cell Internal Power  = 891.9057 uW   (85%)
  Net Switching Power  = 161.3475 uW   (15%)
                         ---------
Total Dynamic Power    =   1.0533 mW  (100%)
Cell Leakage Power     =  98.6818 nW

Timing report generated:
```

data required time	19.84
data arrival time	-3.11
slack (MET)	16.73

Since our design was created to specifically operate using the 50 MHz clock from the Altera board, optimizing for a faster frequency of clock would require the entire component to be modified. With the FPGA implementation we were unable to determine power consumption so we could not compare. The FPGA also required 333 logic elements in order to instantiate our components. We are unable to directly compare size of implementation but reasonably believe that the IC circuit will require less area.

References Used

[1] http://wiki.altium.com/display/ADOH/NEC+Infrared+Transmission+Protocol
NEC Infrared Transmission Protocol

[2] Seiko Instruments Character LCD Modules Datasheet

[3] Xilinx IrDA
Irda_com.vhd is a modified version of a supplied code from Xilinx.com.
Sirendec.vhd, jk_ff.vhd, and pkg_util.vhd are unmodified codes from Xilinx.com.

[4] LCD Code
seiko_lcd.c , seiko_lcd.h, seiko_lcd_private.h are adapted versions of CMPE401 lab supplied code from Nancy Minderman.

Appendix
Hardware Documentation

Appendix A: Quick Start Manual

This manual assumes you have the Altera development environment with Quartus II anbd the NIOS II IDE and assumes that the NIOS II IDE is setup with uC OS-II. All our files are available in the appendices below. A brief list of all the files required is listed here:

VHDL files:

- clock_scaler.vhd
- jk_ff.vhd
- cyclone.vhd
- irda_xlinx.vhd
- pkg_util.vhd
- sirendec.vhd
- irda_com.vhd
- niosII_microc_lab1.vhd
- sram_glue.vhd

C files:

- alt_lcd.c
- game2.c
- game3.h
- hello_ucosii.c
- remote.c
- test_game.c
- alt_lcd.h
- game2.h
- game_menu.c
- my_error.h
- remote.h
- test_game.h
- game3.c

game_menu.h
tasks.h

Software Requirements:

All of our code was created and compiled use Altera's Quartus II and NIOSII IDE. The version we used is 10.1

Hardware Requirments:

We used the Altera DE2 educational development board with the Cyclone II EP2C35F672C6 on board.

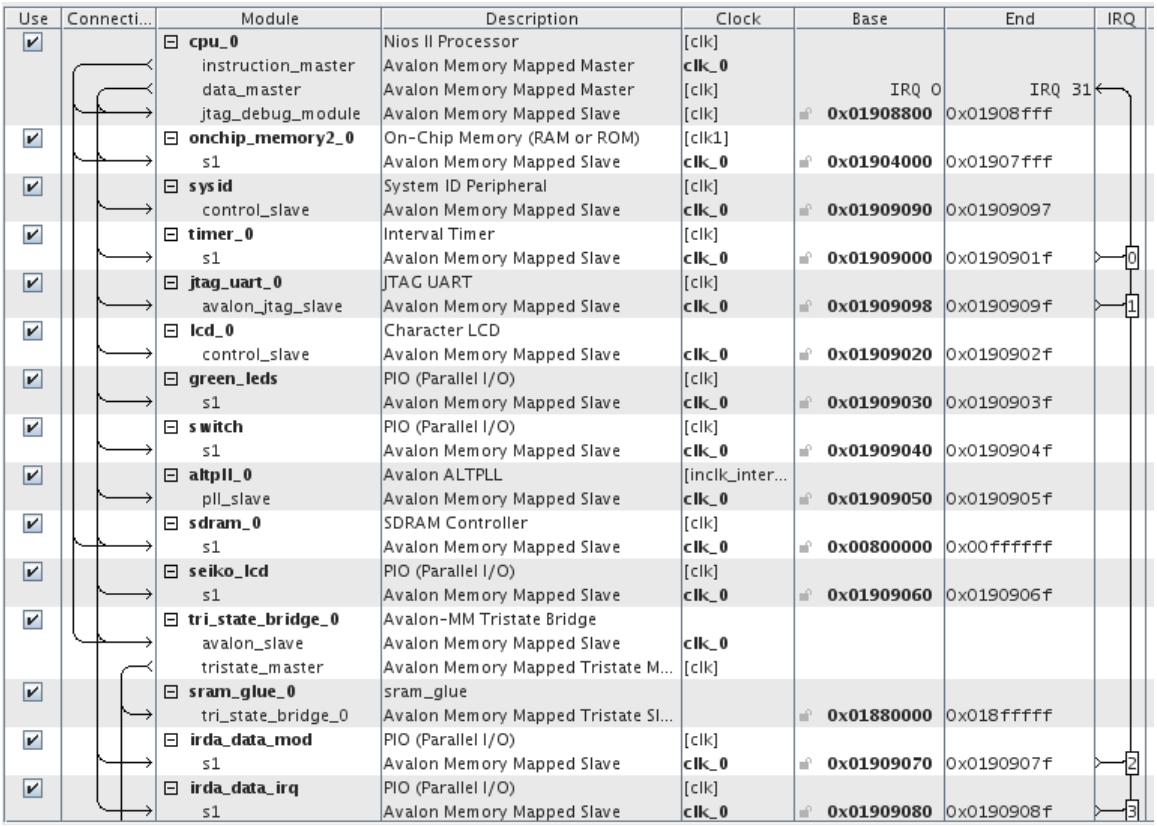
The LCD requires a 5 V supply to power the backlight. The backlight is optional and the LCD runs fine on the GPIO power pins on the DE2 board.

Environment Setup:

Step a new quartus project and make sure you select the correct device. (Cyclone II EP2C35F672C6)

Programming the Cyclone II FPGA:

First under the "tasks" tab to the right of the Quartus II environment click the SOPC Builder tab. Build the system as seen below. You may exclude timer, lcd_0, the leds, the pll, and the sdram if you wish but you will have to change the top level file later to make the project compile. Add the Avalon tristate bridge. Click add new component and add the sram_glue.vhd. Click finish and connect the sram component to the trisate bridge as shown. Create two PIO pins call irda_data_mod and irda_data_irq and set them as input. Click System-> auto assign base addresses and irqs. Save the system and click generate.



Use	Connecti...	Module	Description	Clock	Base	End	IRQ
☑		**cpu_0**	Nios II Processor	[clk]			
		instruction_master	Avalon Memory Mapped Master	**clk_0**			
		data_master	Avalon Memory Mapped Master	[clk]		IRQ 0	IRQ 31
		jtag_debug_module	Avalon Memory Mapped Slave	[clk]	**0x01908800**	0x01908fff	
☑		**onchip_memory2_0**	On-Chip Memory (RAM or ROM)	[clk1]			
		s1	Avalon Memory Mapped Slave	**clk_0**	**0x01904000**	0x01907fff	
☑		**sysid**	System ID Peripheral	[clk]			
		control_slave	Avalon Memory Mapped Slave	**clk_0**	**0x01909090**	0x01909097	
☑		**timer_0**	Interval Timer	[clk]			
		s1	Avalon Memory Mapped Slave	**clk_0**	**0x01909000**	0x0190901f	0
☑		**jtag_uart_0**	JTAG UART	[clk]			
		avalon_jtag_slave	Avalon Memory Mapped Slave	**clk_0**	**0x01909098**	0x0190909f	1
☑		**lcd_0**	Character LCD				
		control_slave	Avalon Memory Mapped Slave	**clk_0**	**0x01909020**	0x0190902f	
☑		**green_leds**	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	**clk_0**	**0x01909030**	0x0190903f	
☑		**switch**	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	**clk_0**	**0x01909040**	0x0190904f	
☑		**altpll_0**	Avalon ALTPLL	[inclk_inter...			
		pll_slave	Avalon Memory Mapped Slave	**clk_0**	**0x01909050**	0x0190905f	
☑		**sdram_0**	SDRAM Controller	[clk]			
		s1	Avalon Memory Mapped Slave	**clk_0**	**0x00800000**	0x00ffffff	
☑		**seiko_lcd**	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	**clk_0**	**0x01909060**	0x0190906f	
☑		**tri_state_bridge_0**	Avalon-MM Tristate Bridge				
		avalon_slave	Avalon Memory Mapped Slave	**clk_0**			
		tristate_master	Avalon Memory Mapped Tristate M...	[clk]			
☑		**sram_glue_0**	sram_glue				
		tri_state_bridge_0	Avalon Memory Mapped Tristate Sl...		**0x01880000**	0x018fffff	
☑		**irda_data_mod**	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	**clk_0**	**0x01909070**	0x0190907f	2
☑		**irda_data_irq**	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	**clk_0**	**0x01909080**	0x0190908f	3

Once the system is generated return to the Quartus II environment. Under the "project navigator" tab right click on files add/remove files and click add all. Under the task tab double click compile design. Once the system is compiled click open the device programmer. Make sure you have the usb jtag port configured properly. Add the .sof file and click start.

Building the Project:

Open the NIOS II IDE and create a new workspace. Create a new project and chose a uC OS-II template. Delete the hello microc file and add all the c files mentioned above. Right click the corresponding syslib file and click properties. Set all sections to sram_glue_0.

Right click the mircoc template file and select "build". Once it has been built select Runs as -> niosII application. The project should run.

Appendix B: Future Work

Our design is far from optimal. There are several modifications that could be made to our project to improve it.

Also, we originally intended to design and implement a WiFi Radio which would stream MP3's from a host over the internet.

IrDA Module:

Our IrDA module is far from perfect. The serial to parallel (SP) converter samples the NEC signal at 1780 Hz which is twice that of the bit frequency of the NEC standard. Given that a NEC standard signal is 68 ms long this means that we the SP converted samples a total of 128 bits from the IrDA serial signal. The IrDA module also reads the start pulse of the NEC signal as if it were data.

Also in the top level that routes the IrDA signal to the NIOSII processor does not supply a one to one mapping of all the buttons on the remote. In particular, the "Enter" and "Blue" buttons on the remote are registered as the same signal. This is a consequence of the data sample that we chose to forward to the NIOSII processor.

Future modifications to the IrDA module could appropriately handle the 9 ms start bit in each NEC signal and could sample the serial signal at the NEC's bit rate of 888 bits per second as opposed to 1780 bits per second. It is also recommended that the data vector forwarded to the NIOSII be changed to ensure a one to one mapping of all the signals sent from the Phillips remote.

Software:

There are known deadlock issues in the software due to mishandled resources. Also the Mutex for the LCD is not necessary and can be replaced with a semaphore.

Deadlock tends to occur when the callback functions of the remotes client try to write to the LCD. This because the client's callback functions are call from a task running at priority two. Future changes can fix this issue by implementing proper interprocess control routines.

MP3 Streaming:

As originally planned we intended to implement a internet radio. This would involve connecting to the internet, decoding MP3's and streaming audio out through the onboard Audio Codec. [X]
To connect to the internet we planned on using the RN-174 module. [X] This device

creates and handle the TCP/IP protocol and feeds data out through a RS232 interface. This device is functional however due to time constraints we never had time to implement an interface.

Decoding the MP3's was to be done with an IP core we obtained from OpenCores. [X] This core was originally intended for SOC design and needed to be integrated into the NIOSII sopc builder.

Appendix C: Source Code Section

Remote Driver:

Remote.h

```
#ifndef REMOTE_H_
#define REMOTE_H_

#include "includes.h"


#define WAIT_FOREVER    0

// signals used to control button flow
enum {
   CNTRL_FEED_BUTTONS  =   0x0,
   CNTRL_FEED_ZEROS    =   0x1,
   CNTRL_FEED_EXITS    =   0x2
};


enum {
      BTN_INPUT    = 0x77777555,
      BTN_POWER    = 0x77775d55,
      BTN_MENU     = 0x7575d577,
      BTN_INFO     = 0x75555777,
      BTN_EXIT     = 0xd75755d7,
      BTN_UP       = 0x777775dd,
      BTN_DOWN     = 0x77775775,
      BTN_LEFT     = 0x5dd75775,
      BTN_RIGHT    = 0x7775d775,
      BTN_OK       = 0x55d75777,
      BTN_RED      = 0xd75ddd5d,
      BTN_GREEN    = 0x5d75575d,
      BTN_YELLOW   = 0xdddd75d5,
      BTN_BLUE     = 0xd5575777,
```

```
    BTN_VOL_UP   = 0x77755557,
    BTN_VOL_DWN  = 0x775d555d,
    BTN_CH_UP    = 0x75dd755d,
    BTN_CH_DWN   = 0x75dddd57,
    BTN_MUTE     = 0x77555577,
    BTN_BACK     = 0x75dd5575,
    BTN_THMB_DWN = 0xd775d575,
    BTN_FFW      = 0x775ddd75,
    BTN_RRW      = 0x7775d5d5,
    BTN_PLAY     = 0xddd55d57,
    BTN_STOP     = 0x777575d5,
    BTN_REC      = 0xdd775757,
    BTN_PAUSE    = 0xdd75d757,
    BTN_1        = 0x7775dd55,
    BTN_2        = 0xdddd7555,
    BTN_3        = 0xdddd5d55,
    BTN_4        = 0xddd77755,
    BTN_5        = 0x775dd557,
    BTN_6        = 0x775d7557,
    BTN_7        = 0x77577557,
    BTN_8        = 0xddd5d557,
    BTN_9        = 0xddd57557,
    BTN_0        = 0x75ddd55d,
    BTN_ENTER    = 0xd5575777,
    BTN_MINUS    = 0x5ddd55d5,
    BTN_ERROR    = 0x80000000
};


int init_remote( int (*)(int(*)()), int (*)(int(*)()) );
void remote_test();
int get_button_push();

void set_remote_flags( int );
void remote_test_lcd();

#endif /*REMOTE_H_*/
```

Remote.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <system.h>
#include "remote.h"
#include "includes.h"
#include "tasks.h"
#include "SeikoLCD.h"
```

```
#include "my_error.h"
#include "seiko_lcd.h"

#define OS_SEM_ACCPEPT_EN              1
#define REMOTE_BUTTON_Q_SIZE           2
#define REMOTE_CONTROL_Q_SIZE          2
#define REMOTE_BUTTON_REQUEST_SEM_INIT  0

/* this is the priviledged get button press that is used exclusively by
 *  remote_main_task
 */


int abp(int);
int gbp(); // private function

struct _remote {
    int defined;    // remote only functional if this is "true"
    void * remote_button_q[REMOTE_BUTTON_Q_SIZE];
    void * remote_control_q[REMOTE_CONTROL_Q_SIZE];
    OS_EVENT * button_queue;    // Q for buttons
    OS_EVENT * button_request_sem;
    OS_EVENT * button_prompt_mutex; // only hanlde one button prompt at a time
    OS_EVENT * control_queue;    // Q for special buttons
    //special button routines
    int (*exit)(int (*)() );  // exit and pause receive a pointer to gbp() so they can
    int (*pause)(int (*)() );   // read button presses accordingly
    int flags;
} remote;

/*  for use by client
 *  sets the desired flags
 */
void set_remote_flags( int flag ){
    remote.flags = flag;
}

/*  remote initialization function
 *
 *  called by client
 *  client must provide exit and pause functions
 *  if exit and pause functions are NULL then remote main task will
 *  pass zeros until flags is set otherwise
 */
int  init_remote( int (*exit)(int(*)() ), int (*pause)(int(*)() ) ){


    INT8U muterr = OS_NO_ERR;
```

```
    if ( !remote.defined ){
        remote.button_request_sem = OSSemCreate(
REMOTE_BUTTON_REQUEST_SEM_INIT);
        if ( muterr != OS_NO_ERR ){
            ERROR(Error creating remote semaphore);
            return muterr;
        }
        remote.button_queue = OSQCreate(remote.remote_button_q,
REMOTE_BUTTON_Q_SIZE);
         if ( !remote.button_queue ){
            ERROR(Error creating remote button queue\n);
            return -1;
        }
        remote.control_queue = OSQCreate(remote.remote_control_q,
REMOTE_BUTTON_Q_SIZE);
        if ( !remote.control_queue ){
            ERROR(Error creating remote control queue\n);
            return -1;
        }
        remote.button_prompt_mutex = OSMutexCreate( BUTTON_PROMPT_MUTEX,
&muterr );
        if ( muterr != OS_NO_ERR ){
            ERROR(Error creating remote button prompt mutex);
            return muterr;
        }

        remote.exit = exit;
        remote.pause = pause;
        remote.defined = 1;
    }
    return OS_NO_ERR;


}


/* REMOTE_MAIN_TASK
 *  this task feeds button values into the function get_button_press
 *  which is called by remote clients
 *
 *  init remote must be called by the client program
 *
 *  if EXIT is pushed then the exit routine will be called
 *
 *  if PAUSE is pused then the pause function will be called
 *
 */
```

```
void remote_main_task( void * pdata ){

   note(started);

   unsigned int           *irda_data, irdata, irdata_mask ;
   irda_data       = (int*)IRDA_DATA_MOD_BASE;
   INT8U semnum    =       OS_NO_ERR;
   INT8U qerr      =       OS_NO_ERR;

   irdata = *irda_data;

   while (1) {

      if ( remote.defined ) {
         //note(Remote defined);

         OSTimeDlyHMSM( 0, 0, 0, 15 ); // wait 108 milliseconds,  or in interrupt
routine

         while ( irdata != *irda_data ){       // post to semaphore

            irdata_mask = irdata = (*irda_data);
            if ( irdata != 0 ) {

               // if EXIt or PAUSE pushed
               switch (irdata) {
                  // insert special buttons here
                  case BTN_EXIT   :
                  case BTN_BACK   :
                     // signal remote_control task
                     note(Posting to control_queue);
                     qerr = OSQPost( remote.control_queue, (void*) irdata );
                     if ( qerr != OS_NO_ERR ){
                        ERROR(Error posting to remote control queue);
                     }
                     OSTimeDlyHMSM( 0, 0, 1, 0 );
                     //irdata_mask = 0; // do not pass button press to client
                     break ;
                  default :   while(0);
               }

               // remote control flags
               switch (remote.flags){
                  case CNTRL_FEED_ZEROS   : irdata_mask = 0; break;
                  case CNTRL_FEED_EXITS   : irdata_mask = BTN_EXIT; break;
                  case CNTRL_FEED_BUTTONS : break;
                  default :   while (0);
               }
```

```
                note(Pending to button_request_sem);
                // checking to see if button is requested
                semnum = OSSemAccept( remote.button_request_sem );
                if ( semnum == 1 ) {  // if button requested then..
                   note(Posting to button_queue);
                   qerr = OSQPost( remote.button_queue, (void*)irdata_mask ); // send
next button press
                   if ( qerr != OS_NO_ERR ){
                      ERROR(Error posting to remote button queue);
                      return;
                   }
                }
                else if ( semnum != 0) { // other case is no resources
                   ERROR(Error posting to remote button semaphore);  // if not time out
then..
                   return;
                }
                OSTimeDlyHMSM( 0, 0, 0, 200 );
             }
          }
       } // if remote note defined pend for a bit
       else OSTimeDlyHMSM(0,0,1,0); // allow other tasks to start
    }
}

/*  this tasks deals with special buttons presses ( BTN_EXIT, BTN_PAUSE, etc. )
 *
 *  control of to remote is seized and not given up until the special button
 *  cases are resolved.
 *
 *  predefined special button routines are called from the remote struct
 */
void remote_control( void * pdata ){


      note(started);

      INT8U qerr = OS_NO_ERR;
      INT8U muterr = OS_NO_ERR;
      int button = 0;

      while (1) {
       if ( remote.defined ) {
          note(Pending to control_queue);

          button = (int) OSQPend(remote.control_queue, WAIT_FOREVER, &qerr);  //
wait for special button press
          if ( qerr != OS_NO_ERR ){
```

```
            ERROR(Error pending to remote control queue);

        }
            note(Pending to button_prompt_mutex);
            //taking control of remote
            OSMutexPend( remote.button_prompt_mutex, WAIT_FOREVER, &muterr
);
            if ( muterr != OS_NO_ERR ){
               ERROR(Error pending to remote button prompt mutex);
            }

            // calling special button routines
            // passing priviledge get button press function to routines
            switch ( button ) {
               // parse special button press
               case BTN_EXIT   :
                        if ( remote.exit != NULL ){
                           note(calling remote exit button routine);
                           (remote.exit)( gbp );
                        } else {
                           note(remote has NULL exit button routine);
                           set_remote_flags( CNTRL_FEED_ZEROS );
                        } break;
               case BTN_BACK   :
                        if ( remote.pause != NULL ){
                           note(calling remote pause button routine);
                           (remote.exit)( gbp );
                        } else {
                           note(remote has NULL pause button routine);
                           set_remote_flags( CNTRL_FEED_ZEROS );
                        } break;
               default :   while(0);
            }
            note(Posting to button_prompt_mutex);
            //releasing remote
            muterr = OSMutexPost( remote.button_prompt_mutex );
            if ( muterr != OS_NO_ERR ){
               ERROR(Error pending to remote button prompt mutex);

            }
      } else OSTimeDlyHMSM(0,0,1,0); // if remote not defined pend for a bit
     }
     note(returned);
}



/*  public get_button_push function
```

```
 *
 *  used by client
 */
int get_button_push(){
    int loop = 0;
    int button = 0;
    INT8U muterr = OS_NO_ERR;

    if (remote.defined){

        note(Pending to button_prompt_mutex);
        // taking control of remote
        while ( !loop ){
            OSTimeDly(15);
            loop = OSMutexAccept( remote.button_prompt_mutex, &muterr );
            if ( muterr != OS_NO_ERR ){
               ERROR(Error pending to remote button prompt mutex);
               return -1;
            }
        }
//            OSMutexPend( remote.button_prompt_mutex, WAIT_FOREVER, &muterr
);
//            if ( muterr != OS_NO_ERR ){
//               ERROR(Error pending to remote button prompt mutex);
//               return -1;
//            }

            button = gbp();

            note(Posting to button_prompt_mutex);
            // releasing remote
            muterr = OSMutexPost( remote.button_prompt_mutex );
            if ( muterr != OS_NO_ERR ){
               ERROR(Error posting to remote button prompt mutex);
               printf("muterr: %x\n", muterr);
               PRINT_MUTEX_ERR_CODES;
               return -1;
            }

        return button;
    }
    return 0;
}

/*  priviledged get_button_push function does not pend to remote mutex
 *
 *  should not be used by client
 *
```

```
 *  BE CAREFUL USING THIS!!!!
 */
int gbp(){
    int button = '0';
    INT8U muterr = OS_NO_ERR, qerr = OS_NO_ERR;

    if ( remote.defined ) {
        note(Posting to button_request_sem);
         // prompting for button
        muterr = OSSemPost( remote.button_request_sem );
        if ( muterr != OS_NO_ERR ){
          ERROR(Error posting to remote semaphore);
          return -1;
        }
        note(Pending to button_queue);
        // listening for button
        button = (int) OSQPend( remote.button_queue, WAIT_FOREVER, &qerr );
        if ( qerr != OS_NO_ERR ){
          ERROR(Error pending to remote queue);
          return -1;
        }
        return button;
    }
    return 0;
}

/*  remote test function to verify remote functionality
 *
 *  writes to stdout
 */
void remote_test(){
    int button = 0;
    while(1){
        OSTimeDlyHMSM( 0, 0, 0, 200 ); //don't get values too fast
        button = get_button_push();
        switch (button) {
            case BTN_INPUT    : printf("INPUT");    break;
            case BTN_POWER    : printf("POWER");    break;
            case BTN_MENU     : printf("MENU");     break;
            case BTN_INFO     : printf("INFO");     break;
            case BTN_EXIT     : printf("EXIT");     break;
            case BTN_UP       : printf("UP");       break;
            case BTN_DOWN     : printf("DOWN");     break;
            case BTN_LEFT     : printf("LEFT");     break;
            case BTN_RIGHT    : printf("RIGHT");    break;
            case BTN_OK       : printf("OK");       break;
            case BTN_RED      : printf("RED");      break;
            case BTN_GREEN    : printf("GREEN");    break;
```

```
            case BTN_YELLOW   : printf("YELLOW");   break;
            case BTN_BLUE     : printf("BLUE");     break;
            case BTN_VOL_UP   : printf("VOL-UP");   break;
            case BTN_VOL_DWN  : printf("VOL-DOWN"); break;
            case BTN_CH_UP    : printf("CHNL-UP");  break;
            case BTN_CH_DWN   : printf("CHNL-DWN"); break;
            case BTN_MUTE     : printf("MUTE");     break;
            case BTN_BACK     : printf("BACK");     break;
            case BTN_THMB_DWN : printf("THMB-DWN"); break;
            case BTN_FFW      : printf("FFW");      break;
            case BTN_RRW      : printf("RRW");      break;
            case BTN_PLAY     : printf("PLAY");     break;
            case BTN_STOP     : printf("STOP");     break;
            case BTN_REC      : printf("REC");      break;
            case BTN_PAUSE    : printf("PAUSE");    break;
            case BTN_1        : printf("1");        break;
            case BTN_2        : printf("2");        break;
            case BTN_3        : printf("3");        break;
            case BTN_4        : printf("4");        break;
            case BTN_5        : printf("5");        break;
            case BTN_6        : printf("6");        break;
            case BTN_7        : printf("7");        break;
            case BTN_8        : printf("8");        break;
            case BTN_9        : printf("9");        break;
            case BTN_0        : printf("0");        break;
            case BTN_MINUS    : printf("MINUS");    break;
            case 0 : default  : printf("Error");
        }
    }
}

void remote_test_lcd(){
    int button = 0;
    int i = 0;
    while (1){

        button = get_button_push();
        // BTN_BACK will signal that the user wishes to return to the main menu
//        if  ( ( button == BTN_EXIT ) || ( button == BTN_BACK ) ) return 0;
//        else
         switch (button) {
            case BTN_INPUT    : print_lcd("Welcome to game_funcbutton: INPUT and
more and more and more and more and more blaaaaabaa");    break;
            case BTN_POWER    : print_lcd("Welcome to game_funcbutton: POWER");
break;
            case BTN_MENU     : print_lcd("Welcome to game_func\nMENU");     break;
            case BTN_INFO     : print_lcd("Welcome to game_func\nINFO");     break;
            case BTN_EXIT     : print_lcd("Welcome to game_func\nEXIT");     break;
```

```
        case BTN_UP       : print_lcd("Welcome to game_func\nUP");       break;
        case BTN_DOWN     : print_lcd("%20s%-20s%20s%-
20s","DOWN","Brown","Cow","I like scotch");     break;
        case BTN_LEFT     : print_lcd("Welcome to game_func\nLEFT");     break;
        case BTN_RIGHT    : print_lcd("Welcome to game_func\nRIGHT");    break;
        case BTN_OK       : print_lcd("Welcome to game_func\nOK");       break;
        case BTN_RED      : print_lcd("Welcome to game_func\nRED");      break;
        case BTN_GREEN    : print_lcd("Welcome to game_func\nGREEN");
break;
        case BTN_YELLOW   : print_lcd("Welcome to game_func\nYELLOW");
break;
        case BTN_BLUE     : print_lcd("Welcome to game_func\nBLUE");     break;
        case BTN_VOL_UP   : print_lcd("Welcome to game_func\nVOL-UP");
break;
        case BTN_VOL_DWN  : print_lcd("Hello\n\nWelcome to game_func\nVOL-
DOWN\n"); break;
        case BTN_CH_UP    : print_lcd("Welcome to game_func\nCHNL-UP");
break;
        case BTN_CH_DWN   : print_lcd("Welcome to game_func\nCHNL-DWN");
break;
        case BTN_MUTE     : print_lcd("Hi\nMy\nname\nis");     break;
        case BTN_BACK     : print_lcd("Welcome to game_func\nBACK");     break;
        case BTN_THMB_DWN : print_lcd("Welcome to game_func\nTHMB-DWN");
break;
        case BTN_FFW      :  {
                        for ( i = 0; i < 80; i ++ ) {
                           ShiftDisplayRight(LCD_UPPER_SCR);
OSTimeDlyHMSM(0,0,0,200);
                        }       break;}
        case BTN_RRW      : {
                        for ( i = 0; i < 80; i ++ ) {
                           ShiftDisplayLeft(LCD_UPPER_SCR);
OSTimeDlyHMSM(0,0,0,200);
                        }       break;}
        case BTN_PLAY     : print_lcd("Welcome to game_func\nPLAY");     break;
        case BTN_STOP     : print_lcd("Welcome to game_func\nSTOP");     break;
        case BTN_REC      : print_lcd("Welcome to game_func\nREC");      break;
        case BTN_PAUSE    : print_lcd("Welcome to game_func\nPAUSE");    break;
        case BTN_1        : print_lcd("Welcome to game_func\n1");        break;
        case BTN_2        : print_lcd("Welcome to game_func\n2");        break;
        case BTN_3        : print_lcd("Welcome to game_func\n3");        break;
        case BTN_4        : print_lcd("Welcome to game_func\n4");        break;
        case BTN_5        : print_lcd("Welcome to game_func\n5");        break;
        case BTN_6        : print_lcd("Welcome to game_func\n6");        break;
        case BTN_7        : print_lcd("Welcome to game_func\n7");        break;
        case BTN_8        : print_lcd("Welcome to game_func\n8");        break;
        case BTN_9        : print_lcd("Welcome to game_func\n9");        break;
        case BTN_0        : print_lcd("Welcome to game_func\n0");        break;
```

```
            case BTN_MINUS    : print_lcd("Welcome to game_func\nMINUS");    break;
            case 0 : default  : print_lcd("Welcome to game_func\nError");
        }

    }
}
```

Game Host:

Game_menu.c:

```
#include <stdlib.h>
#include "game_menu.h"
#include "remote.h"
//#include "alt_lcd.h"
//#include "seiko_lcd.h"
#include "my_error.h"
#include "tasks.h"
#include "test_game.h"
#include "game2.h"
#include "game3.h"
#include "SeikoLCD.h"

#define print_lcd(a,b,c,d) print_lcd("%-20s%-20s%-20s%-20s",a,c,b,d );

int play_current_game();



struct _GAME {
    // all game tasks pend to this mutex
    OS_EVENT * game_mutex;  // ensures only one game task can run at a time
    //  game menu data ...
    game_type games[MAX_GAMES];
    int current_game;
    int enable_callback;
    int end_game;
} GAME;


int game_exit( int (*)() );

void rotate_games_left();
void rotate_games_right();

void start_menu( void * pdata ){
```

```
    note(started);
    // initialization

    note(Initializing remote);
    init_remote( game_exit, game_exit );

    note(Initializing games);
    init_games();

    note(Initializing print_lcd);
    init_print_lcd();

    int button = 0;
    int loop = 1;

    INT8U err = OS_NO_ERR;
    while (1) {
        // of game tasks only this task can run
        note(pending to game mutex);
        OSMutexPend( GAME.game_mutex, WAIT_FOREVER, &err );
        if ( err != OS_NO_ERR ) {
            _num(err);
            ERROR(could not pend to game mutex);
        }
        // prepare everything for the next game
        GAME.end_game = GAME_CONTINUE;
        set_remote_flags( CNTRL_FEED_BUTTONS );

        note(Acquired game mutex);
        //set_remote_flags( CNTRL_FEED_BUTTONS );
        /*  ensures we get the buttons
         *  this is incase we were just exiting from a game
         *  get_button_push should only return BTN_EXIT when exiting a game
         */
         //print_lcd("%-20s%-20s%-20s%-20s","Welcome","Press `OK'to", "",
"continue." );
        print_lcd("Welcome", "", "Press `OK'to",  "continue." );
        note(Printed to the LCD);

        while ( loop ) switch ( button = get_button_push() ) {
            case BTN_OK : loop = 0; break;
            default : while(0) ;
        } loop = 1; // for next loop



        while (loop) {  // get game selection from user
```

```
        print_lcd( "Pick a Program:", "", GAME.games[GAME.current_game].name,
"");
        switch (button = get_button_push() ){
           case BTN_LEFT   :   rotate_games_left();    break;
           case BTN_RIGHT  :   rotate_games_right();   break;
           case BTN_OK     :   loop = 0; break;
           default         :   while(0);
        }
      } loop = 1;

      // play current game
      if( play_current_game() != 0 )
         ERROR(could not play game);

      note(Posting to game_mutex);
      // give up control
      err = OSMutexPost( GAME.game_mutex );
      if ( err != OS_NO_ERR ) {
         //if( err !=
         ERROR(could not post to game mutex);
         printf("muterr: %x\n", err);
         PRINT_MUTEX_ERR_CODES;
      }
      OSTimeDlyHMSM( 0, 0, 1, 0 ); // give game a chance to sieze control



   }
}

/* games[ A B C D E F ]
 *            i
 *      ====>i ======
 */
void rotate_games_left(){
   GAME.current_game =  ( GAME.current_game + MAX_GAMES - 1 ) %
MAX_GAMES;
}

/* games[ A B C D E F ]
 *            i
 *            =>i
 */
void rotate_games_right(){
   GAME.current_game =  ( GAME.current_game + 1 ) % MAX_GAMES;
}
/* PLAY_CURRENT_GAME
 *  called when a game is selected to play
```

```
 *  enables games exit and pause functions
 *
 */
int play_current_game(){


    GAME.enable_callback = 1;
    GAME.games[GAME.current_game].create_game();

    return 0;
}
/*  called by start_menu task
 *
 *
 *
 *
 */
int init_games() {
    INT8U err = OS_NO_ERR;

    // initializing game menu struct
    GAME.game_mutex = OSMutexCreate( GAME_MUTEX_PRIO, &err );
    if ( err != OS_NO_ERR ) {
        ERROR(Could not create game mutex);
    }
    // setting current game
    GAME.current_game = 0;
    GAME.enable_callback = 0;
    GAME.end_game = GAME_CONTINUE;

    init_test_game(&GAME.games[0]); // important
    init_game2( &GAME.games[1] );
    init_game3( &GAME.games[2] );

    return 0;
}


/*  these functions are called when buttons exit or pause are pushed
 *
 */
int game_exit(int (*get_button_push)() ){
    note(called);
    int cmd_exit = 0;

    if ( GAME.enable_callback ){

        set_remote_flags(CNTRL_FEED_EXITS);
```

```
        GAME.end_game = GAME_EXIT;
        GAME.enable_callback = 0;
        note(Returned);
        return cmd_exit;
    }
    return 0;
}


// this is called something in OOP ...
INT8U post_to_game_mutex(){
    note(Called);
    return OSMutexPost( GAME.game_mutex );
}

INT8U pend_to_game_mutex(){
    note(Called);
    INT8U err = OS_NO_ERR;
    OSMutexPend( GAME.game_mutex, WAIT_FOREVER, &err );
    return err;
}

/*  lets client no if it is supposed to end
 *
 */
int end_game(){
    if (GAME.end_game == GAME_EXIT)
        return 1;
    return 0;
}
```

Game_menu.h

```
#ifndef GAME_MENU_H_
#define GAME_MENU_H_

#define STR_LEN     64
#define MAX_GAMES   3

#include "includes.h"
#include "tasks.h"

enum {
```

```
    GAME_EXIT       = 0xFFFFFFFF,
    GAME_CONTINUE   = 0xEEEEEEEE,
    GAME_ERROR      = 0xF8F8F8F8
};

typedef struct {
    int game_code;
    char name[STR_LEN]; // name of game
    void (*create_game)();
} game_type;

int init_games();
INT8U post_to_game_mutex();
INT8U pend_to_game_mutex();

int end_game();

#endif /*GAME_MENU_H_*/
```

Test_game.c

```
#include "includes.h"
#include "SeikoLCD.h"
#include "my_error.h"
#include "remote.h"
#include "game_menu.h"
#include "tasks.h"
#include "seiko_lcd.h"
#include <stdlib.h>
#include <sys/alt_alarm.h>
#include <string.h>

#define true 1
#define false 0

#define print_lcd(a,b,c,d) print_lcd("%-20s%-20s%-20s%-20s",a,c,b,d )
#define print_lcd_top(a)   { Home(LCD_UPPER_SCR);  PrintString(
LCD_UPPER_SCR, a ); }
#define print_lcd_middle(a) { MoveCursor(LCD_UPPER_SCR, 27);  PrintString(
LCD_UPPER_SCR, a ); }
#define clear_lcd()        Clear(LCD_UPPER_SCR)

#define REP_PAUSE_DISPLAY   1       //REP_PAUSE_DISPLAY
#define REACTION_TIME       3000    //milliseconds
#define DISPLAY_TIME        600     //milliseconds
#define MAX_LEN_SEQ         100     //max sequence to repeat

enum {
```

```
//      REP_INPUT    ,
//      REP_POWER    ,
//      REP_MENU     ,
//      REP_INFO     ,
//      REP_UP       ,
//      REP_DOWN     ,
//      REP_LEFT     ,
//      REP_RIGHT    ,
//      REP_OK       ,
//      REP_RED      ,
//      REP_GREEN    ,
//      REP_YELLOW   ,
//      REP_BLUE     ,
//      REP_VOL_UP   ,
//      REP_VOL_DWN  ,
//      REP_CH_UP    ,
//      REP_CH_DWN   ,
//      REP_MUTE     ,
//      REP_BACK     ,
//      REP_THMB_DWN ,
//      REP_FFW      ,
//      REP_RRW      ,
//      REP_PLAY     ,
//      REP_STOP     ,
//      REP_REC      ,
//      REP_PAUSE    ,
        REP_1        ,
        REP_2        ,
        REP_3        ,
        REP_4        ,
        REP_5        ,
        REP_6        ,
        REP_7        ,
        REP_8        ,
        REP_9        ,
        REP_0        ,
//      REP_ENTER    ,
//      REP_MINUS    ,
//      REP_ERROR    ,
        REP_MAX_BUTTON,
        REP_ERROR
};

void create_test_game();
void test_game_task( void * );

OS_STK      test_game_stk[TASK_STACKSIZE];
```

```
void init_test_game( game_type * game_struct ){
   game_struct->game_code = GAME_CONTINUE;    strncpy( game_struct->name,
"Repeat After Me", STR_LEN );
   game_struct->create_game = create_test_game;
}

void create_test_game(){
   OSTaskCreateExt(test_game_task,
               NULL,
               (void *)&test_game_stk[TASK_STACKSIZE-1],
               GAME_PRIO,
               GAME_PRIO,
               test_game_stk,
               TASK_STACKSIZE,
               NULL,
               0);
}

void test_game_task( void * pdata ){
   note(Started);

   INT8U err = pend_to_game_mutex();
   if ( err != OS_NO_ERR ){
      ERROR(Could not pend to game mutex);
   }
   note(Acquired game mutex);
   long int seed = alt_nticks();
   srand48(seed);

   INT32U time1 = 0, time2 = 0;
   int game_over = false;
   int button = 0;
   int seq_len = 0;  // the length of sequence the user has to repeat
   int sequence[MAX_LEN_SEQ] = {0};
   int i = 0;
   int loop = 1;

   // generate random sequence
   for ( i = 0; i < MAX_LEN_SEQ; i++ ){
      sequence[i] = ( lrand48() % REP_MAX_BUTTON );
   }


   char * welcome = "Welcome to:";
   char * name = "Repeat After Me" ;

   print_lcd( welcome, name, "", "" );
```

```
    get_button_push();

    print_lcd(name, "", "Begin?", "" );

    while ( !end_game() && loop ){
        switch ( get_button_push() ){
            case BTN_OK : loop = 0; break;
            default   : while (0);
        }
    } loop = 1;



    while (!end_game()) {
        if (game_over) break;
        while (!end_game()){
            if (game_over) break;
            seq_len ++;
            print_lcd("Starting sequence","","","");
            note(Waiting);
            if ( OSTimeDlyHMSM( 0, 0, 1, 0 ) != OS_NO_ERR)
                ERROR(Could not wait);


            for ( i = 0; i < seq_len ; i++ ){
                if (end_game()) break;
                clear_lcd();
                switch ( sequence[i] )  {
//                  case REP_INPUT    : print_lcd_middle("INPUT");    OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
//                  case REP_POWER    : print_lcd_middle("POWER");
OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME );  break;
//                  case REP_MENU     : print_lcd_middle("MENU");
OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME );  break;
//                  case REP_INFO     : print_lcd_middle("INFO");     OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
//                  case REP_UP       : print_lcd_middle("UP");       OSTimeDlyHMSM( 0,
0, 0, DISPLAY_TIME );  break;
//                  case REP_DOWN     : print_lcd_middle("DOWN");
OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME );  break;
//                  case REP_LEFT     : print_lcd_middle("LEFT");     OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
//                  case REP_RIGHT    : print_lcd_middle("RIGHT");    OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
//                  case REP_OK       : print_lcd_middle("OK");       OSTimeDlyHMSM( 0,
0, 0, DISPLAY_TIME );  break;
//                  case REP_RED      : print_lcd_middle("RED");      OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
```

```
//                case REP_GREEN    : print_lcd_middle("GREEN");
OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME );  break;
//                case REP_YELLOW   : print_lcd_middle("YELLOW");
OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME );  break;
//                case REP_BLUE     : print_lcd_middle("BLUE");     OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
//                case REP_VOL_UP   : print_lcd_middle("VOL-UP");
OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME );  break;
//                case REP_VOL_DWN  : print_lcd_middle("VOL-DOWN");
OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME );  break;
//                case REP_CH_UP    : print_lcd_middle("CHNL-UP");
OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME );  break;
//                case REP_CH_DWN   : print_lcd_middle("CHNL-DWN");
OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME );  break;
//                case REP_MUTE     : print_lcd_middle("MUTE");     OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
//                case REP_BACK     : print_lcd_middle("BACK");     OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
//                case REP_THMB_DWN : print_lcd_middle("THMB-DWN");
OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME );  break;
//                case REP_FFW      : print_lcd_middle("FFW");      OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
//                case REP_RRW      : print_lcd_middle("RRW");      OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
//                case REP_PLAY     : print_lcd_middle("PLAY");     OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
//                case REP_STOP     : print_lcd_middle("STOP");     OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
//                case REP_REC      : print_lcd_middle("REC");      OSTimeDlyHMSM(
0, 0, 0, DISPLAY_TIME );  break;
//                case REP_PAUSE    : print_lcd_middle("PAUSE");
OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME );  break;
                case REP_1        : print_lcd_middle("1");        OSTimeDlyHMSM( 0, 0, 0,
DISPLAY_TIME );  break;
                case REP_2        : print_lcd_middle("2");        OSTimeDlyHMSM( 0, 0, 0,
DISPLAY_TIME );  break;
                case REP_3        : print_lcd_middle("3");        OSTimeDlyHMSM( 0, 0, 0,
DISPLAY_TIME );  break;
                case REP_4        : print_lcd_middle("4");        OSTimeDlyHMSM( 0, 0, 0,
DISPLAY_TIME );  break;
                case REP_5        : print_lcd_middle("5");        OSTimeDlyHMSM( 0, 0, 0,
DISPLAY_TIME );  break;
                case REP_6        : print_lcd_middle("6");        OSTimeDlyHMSM( 0, 0, 0,
DISPLAY_TIME );  break;
                case REP_7        : print_lcd_middle("7");        OSTimeDlyHMSM( 0, 0, 0,
DISPLAY_TIME );  break;
                case REP_8        : print_lcd_middle("8");        OSTimeDlyHMSM( 0, 0, 0,
DISPLAY_TIME );  break;
```

```
                case REP_9       : print_lcd_middle("9");       OSTimeDlyHMSM( 0, 0, 0,
DISPLAY_TIME );  break;
                case REP_0       : print_lcd_middle("0");       OSTimeDlyHMSM( 0, 0, 0,
DISPLAY_TIME );  break;
//                 case REP_MINUS    : print_lcd_middle("MINUS");
OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME );  break;
                default  : print_lcd( name, "ERROR:", "invalid sequence", "" );
              }
          }  // end display sequence

          print_lcd("","   YOUR TURN! ","","");
          note(Waiting);
          if ( OSTimeDlyHMSM( 0, 0, 1, 0 ) != OS_NO_ERR)
             ERROR(Could not wait);


          print_lcd("","     GO! ","","");
          note(Waiting);
          if ( OSTimeDlyHMSM( 0, 0, 0, DISPLAY_TIME ) != OS_NO_ERR)
             ERROR(Could not wait);

          clear_lcd();

          for ( i = 0; i < seq_len ; i++ ){

             int tracker = REP_ERROR;
             if (end_game()) break;
             if (game_over) break;

             time1 = OSTimeGet();
             switch (get_button_push())  {
                case BTN_INPUT    : clear_lcd();    print_lcd_middle("INPUT");
break;
                case BTN_POWER    : clear_lcd();    print_lcd_middle("POWER");
break;
                case BTN_MENU     : clear_lcd();    print_lcd_middle("MENU");
break;
                case BTN_INFO     : clear_lcd();    print_lcd_middle("INFO");       break;
                case BTN_UP       : clear_lcd();    print_lcd_middle("UP");         break;
                case BTN_DOWN     : clear_lcd();    print_lcd_middle("DOWN");
break;
                case BTN_LEFT     : clear_lcd();    print_lcd_middle("LEFT");       break;
                case BTN_RIGHT    : clear_lcd();    print_lcd_middle("RIGHT");
break;
                case BTN_OK       : clear_lcd();    print_lcd_middle("OK");         break;
                case BTN_RED      : clear_lcd();    print_lcd_middle("RED");        break;
                case BTN_GREEN    : clear_lcd();    print_lcd_middle("GREEN");
break;
```

```
            case BTN_YELLOW   : clear_lcd();    print_lcd_middle("YELLOW");
break;
            case BTN_BLUE     : clear_lcd();    print_lcd_middle("BLUE");       break;
            case BTN_VOL_UP   : clear_lcd();    print_lcd_middle("VOL-UP");
break;
            case BTN_VOL_DWN  : clear_lcd();    print_lcd_middle("VOL-DOWN");
break;
            case BTN_CH_UP    : clear_lcd();    print_lcd_middle("CHNL-UP");
break;
            case BTN_CH_DWN   : clear_lcd();    print_lcd_middle("CHNL-DWN");
break;
            case BTN_MUTE     : clear_lcd();    print_lcd_middle("MUTE");
break;
            case BTN_BACK     : clear_lcd();    print_lcd_middle("BACK");
break;
            case BTN_THMB_DWN : clear_lcd();    print_lcd_middle("THMB-
DWN");   break;
            case BTN_FFW      : clear_lcd();    print_lcd_middle("FFW");        break;
            case BTN_RRW      : clear_lcd();    print_lcd_middle("RRW");        break;
            case BTN_PLAY     : clear_lcd();    print_lcd_middle("PLAY");       break;
            case BTN_STOP     : clear_lcd();    print_lcd_middle("STOP");
break;
            case BTN_REC      : clear_lcd();    print_lcd_middle("REC");        break;
            case BTN_PAUSE    : clear_lcd();    print_lcd_middle("PAUSE");
break;
            case BTN_1        : clear_lcd();    print_lcd_middle("1");
                        tracker = REP_1;                        break;
            case BTN_2        : clear_lcd();    print_lcd_middle("2");
                        tracker = REP_2;                        break;
            case BTN_3        : clear_lcd();    print_lcd_middle("3");
                        tracker = REP_3;                        break;
            case BTN_4        : clear_lcd();    print_lcd_middle("4");
                        tracker = REP_4;                        break;
            case BTN_5        : clear_lcd();    print_lcd_middle("5");
                        tracker = REP_5;                        break;
            case BTN_6        : clear_lcd();    print_lcd_middle("6");
                        tracker = REP_6;                        break;
            case BTN_7        : clear_lcd();    print_lcd_middle("7");
                        tracker = REP_7;                        break;
            case BTN_8        : clear_lcd();    print_lcd_middle("8");
                        tracker = REP_8;                        break;
            case BTN_9        : clear_lcd();    print_lcd_middle("9");
                        tracker = REP_9;                        break;
            case BTN_0        : clear_lcd();    print_lcd_middle("0");
                        tracker = REP_0;                        break;
            case BTN_MINUS    : clear_lcd();    print_lcd_middle("MINUS");
break;
            default  : clear_lcd();    print_lcd( name, "ERROR:", "invalid sequence",
```

```
"" );
            }
            time2 = OSTimeGet();
            if ( sequence[i] != tracker )
               game_over = true;
            if ( ( time2 - time1 ) > ( alt_ticks_per_second()*REACTION_TIME)/1000 )
               game_over = true;
         } // end your turn for

         OSTimeDlyHMSM( 0, 0, REP_PAUSE_DISPLAY, 0);
         // is took longer  1 second ?
         if ( game_over ){
            print_lcd( name, "GAME OVER", "", "Play Again?" );

            switch (get_button_push()){
               case BTN_OK :   game_over = false;
                               //generate new sequence
                               seq_len = 0;
                               for ( i = 0; i < MAX_LEN_SEQ; i++ ){
                                  sequence[i] = ( lrand48() % REP_MAX_BUTTON );
                               } break;
               default     :   while(0);
            }

         } else {
            print_lcd( "", "Great!", "", "" );
         }
      } // end game started


   }
   // if we ever get here
   set_remote_flags(CNTRL_FEED_BUTTONS);
   note(Posting to game_mutex);
   err = post_to_game_mutex();
   if ( err != OS_NO_ERR ){ERROR(Could not post to game mutex);}
   OSTaskDel(OS_PRIO_SELF);
   note(Ended);
   OSTimeDlyHMSM( 0, 0, 5, 0 );  // wait until deleted
}
```

Test_game.h

```
#ifndef TEST_GAME_H_
#define TEST_GAME_H_

void test_game_task( void * pdata );
```

```
void init_test_game( game_type * );


#endif /*TEST_GAME_H_*/
```

Game2.c

```
#include "game_menu.h"
#include "my_error.h"
#include "remote.h"
#include "SeikoLCD.h"
#include <string.h>

void create_game2();
void skill_tester_task( void * );

OS_STK      skill_tester_stk[TASK_STACKSIZE];

void init_game2( game_type * game_struct ){
   strncpy( game_struct->name, "Skill Tester", STR_LEN );
   game_struct->create_game = create_game2;
}

void create_game2(){
        OSTaskCreateExt(skill_tester_task,
                NULL,
                (void *)&skill_tester_stk[TASK_STACKSIZE-1],
                GAME_PRIO,
                GAME_PRIO,
                skill_tester_stk,
                TASK_STACKSIZE,
                NULL,
                0);
}


void skill_tester_task(void *pdata){
   INT8U err = pend_to_game_mutex();
   if ( err != OS_NO_ERR ){
      ERROR(Could not pend to game mutex);
   }
   note(Acquired game mutex);
   char * welcome = "Welcome to the \nSkill Tester";

   print_lcd( welcome );

   while (1) {
```

```
        int button = get_button_push();

        switch (button) {
           case  BTN_EXIT  :   set_remote_flags(CNTRL_FEED_BUTTONS);
                               note(Posting to game_mutex);
                               err = post_to_game_mutex();
                               if ( err != OS_NO_ERR ){ERROR(Could not post to game
mutex);}
                               OSTaskDel(OS_PRIO_SELF);
                               note(Ended);
                               OSTimeDlyHMSM( 0, 0, 5, 0 ); break;
           default   :  while (0);
        }



    }
}
```

Game2.h

```
#ifndef GAME2_H_
#define GAME2_H_

void init_game2( game_type * );

#endif /*GAME2_H_*/
```

Game3.c

```
#include "game_menu.h"
#include "my_error.h"
#include "remote.h"
#include "SeikoLCD.h"
#include <string.h>

#define CARD_LEN    9

void create_game3();
void blackjack_task( void * );



OS_STK      blackjack_stk[TASK_STACKSIZE];

void init_game3( game_type * game_struct ){
```

```
    strncpy( game_struct->name, "BlackJack", STR_LEN );
    game_struct->create_game = create_game3;
}

void create_game3(){
            OSTaskCreateExt(blackjack_task,
                NULL,
                (void *)&blackjack_stk[TASK_STACKSIZE-1],
                GAME_PRIO,
                GAME_PRIO,
                blackjack_stk,
                TASK_STACKSIZE,
                NULL,
                0);
}

//struct {
//    char * playerCards[CARD_LEN];
//     hitOrStay = "";
//        static int total, count, dealerTotal;
//        static Random cardRandomizer = new Random();
//} BlackJack;

void blackjack_task( void * pdata ){
    INT8U err = pend_to_game_mutex();
    if ( err != OS_NO_ERR ){
        ERROR(Could not pend to game mutex);
    }
    note(Acquired game mutex);
    char * welcome = "Welcome to the \nBlackJack";

    print_lcd( welcome );

    while (1) {

        int button = get_button_push();

        switch (button) {
            case  BTN_EXIT  :   set_remote_flags(CNTRL_FEED_BUTTONS);
                                note(Posting to game_mutex);
                                err = post_to_game_mutex();
                                if ( err != OS_NO_ERR ){ERROR(Could not post to game
mutex);}
                                OSTaskDel(OS_PRIO_SELF);
                                note(Ended);
                                OSTimeDlyHMSM( 0, 0, 5, 0 ); break;
          default   :  while (0);
        }
```

```
    }

}
```

Game3.h

```
#ifndef GAME3_H_
#define GAME3_H_


void init_game3( game_type * );


#endif /*GAME3_H_*/
```

LCD:

SeikoLCD.c

```
#include "includes.h"
#include "SeikoLCD.h"
#include "seiko_lcd.h"
#include "my_error.h"
#include "stdarg.h"
#include "tasks.h"
#include <stdio.h>
#include <string.h>


static struct lcd_struct {
    OS_EVENT * mutex;
    char line[LCD_SIZE + 1];  // current line in lcd
//    char secondline[LINE_MAX];
} LCD;
```

```
int init_print_lcd(){
    INT8U err = OS_NO_ERR;
    LCD.mutex = OSMutexCreate( LCD_PRIO, &err );
    if ( err != OS_NO_ERR ) {
        ERROR(Could not create LCD mutex);
        return -1;
    }

    Init(LCD_UPPER_SCR);

    return 0;
}
//int delete_lcd(){
//
//
//}
int print_lcd( char * string, ... ){

    int num = 0;
    int loop = 0;


//    char line3[LCD_COLS+1];
//    char line4[LCD_COLS+1];

    INT8U err = OS_NO_ERR;
    va_list arguments;


    va_start ( arguments, string );
    while ( !loop ){
        OSTimeDly(15);
        loop = OSMutexAccept(LCD.mutex, &err);
        if ( err != OS_NO_ERR ) {
            ERROR(Could not pend to LCD mutex);
            return -1;
        }
    }

    note(Acquired LCD mutex);

        char line[LCD_SIZE + 1];
        num = vsnprintf( line, LCD_SIZE + 1, string, arguments ); // so thats how...
        //vprintf(string,arguments);
        strncpy( LCD.line, line, LCD_SIZE + 1 );
        Clear(LCD_UPPER_SCR);
```

```
        //MoveCursor(LCD_UPPER_SCR,40);


        //printf("%s\n",line);
        PrintString( LCD_UPPER_SCR, line );
        //printf("%s\n",line);

    err = OSMutexPost(LCD.mutex);
    if ( err != OS_NO_ERR ) {
        ERROR(Could not post to LCD mutex);
        return -1;
    }
    return num;

}
void read_lcd( char * return_string ){
    strncpy( return_string, LCD.line, LCD_SIZE + 1 );

}
// for more control
INT8U post_to_lcd_mutex(){
    return OSMutexPost( LCD.mutex );

}
INT8U pend_to_lcd_mutex(){
    INT8U err = OS_NO_ERR;
    int loop = 0;
    while ( !loop ){
        OSTimeDly(1);
        loop = OSMutexAccept(LCD.mutex, &err);
        if ( err != OS_NO_ERR ) {
            ERROR(Could not pend to LCD mutex);
            return -1;
        }
    }
    return err;

}
int plcd( char * string, ... ){
    int num = 0;
    va_list arguments;
    va_start ( arguments, string );
    char line[LCD_SIZE + 1];
    num = vsnprintf( line, LCD_SIZE + 1, string, arguments ); // so thats how...
    strncpy( LCD.line, line, LCD_SIZE + 1 );
    Clear(LCD_UPPER_SCR);

    PrintString(LCD_UPPER_SCR, line);
```

```
    return num;

}  // bypasses lcd mutex
```

SeikoLCD.h

```
#ifndef SEIKOLCD_H_
#define SEIKOLCD_H_

#define LCD_SIZE    80

int init_print_lcd();
//int delete_lcd();
int print_lcd( char *, ... );
void read_lcd( char * );
// for more control
INT8U post_to_lcd_mutex();
INT8U pend_to_lcd_mutex();
int plcd( char*, ... );  // bypasses lcd mutex



#endif /*SEIKOLCD_H_*/
```

Seiko_lcd.c

```
/* This code was made for use on a NetBurner MOD5234
 * Justin Smalley, 2008
 *
 * Nancy Minderman
 * nem@ece.ualberta.ca
 * Modified for the CMPE401 Fall2009 semester
 * This file now lives in user space to ensure student access.
 *
 * Modified for the CMPE401 Fall2010 semester
 * Added screen granularity so we can control both screens independently
 *
 *
 * Modified for the CMPE401 Fall2011 semester
 * Cleaned up some methods and removed some dead code
 *
 * */

#define false 0
#define true 1

//#include "predef.h"
```

```
#include <stdio.h>
#include <ctype.h>
//#include <startnet.h>
//#include <http.h>
//#include <ucosmcfc.h>
#include <system.h>
//#include <iosys.h>
//#include <autoupdate.h>
//#include <serial.h>
//#include <dhcpclient.h>
//#include <taskmon.h>
//#include <NetworkDebug.h>
//#include <pins.h>
//#include "pinmap.h"
#include "seiko_lcd.h"
#include "seiko_lcd_private.h"


// added
#include "includes.h"
#include <stdlib.h>
#include <sys/alt_alarm.h>
//#include "system.h"

#define SEIKO_LCD_DATA_BITS     0x07F8


/* Name:  send_cmd
 * Description: Sends a command to the LCD.
 *
 * A proper command is specified by a constant that starts with
 * CMD_* and is ored with a valid option selection for that command.
 * see lcd.h for valid options
 * Inputs: unsigned char screen unsigned char cmd
 * Outputs: none
 */
void send_cmd(unsigned char screen, unsigned char cmd)
{

   set_data_lines(cmd);
   set_rs_line(RS_CMD);

   //debug outputs. These can be turned on and off in lcd.h
   LCD_DEBUG(iprintf("(%3u) <%u%u%u%u %u%u%u%u>\n", cmd, (cmd >> 7) &
0x01,(cmd >> 6) & 0x01,(cmd >> 5) & 0x01,(cmd >> 4) & 0x01,(cmd >> 3) &
0x01,(cmd >> 2) & 0x01,(cmd >> 1) & 0x01,cmd & 0x01));

   // echo outputs to serial console. These can be turned on and off
```

```
  // in lcd.h
  LCD_ECHO(iprintf("LCD_ECHO command: %X\n", cmd));
  OSTimeDly(1);

  if (screen == LCD_UPPER_SCR){
     enable_upper();
  } else if (screen == LCD_LOWER_SCR) {
     enable_lower();
  } else if ((screen == LCD_BOTH_SCR) || (screen == LCD_SINGLE_SCR)) {
     enable_upper();
     enable_lower();
  } else {
     LCD_DEBUG(iprintf("send_cmd wrong screen type\n"));
  }
}

/* Name: send_data
 * Description: Sends a single character to the LCD.
 * Given that the LCD is divided into 2 separate screens
 * it must be sent to the correct one based on the current position.
 * Inputs: unsigned char data must be a printable ASCII character.
 * Outputs: none
 */
void send_data(unsigned char  screen, unsigned char data)
{
   set_data_lines(data);
   set_rs_line(RS_DATA);
   LCD_ECHO(iprintf("LCD_ECHO data: %c\n", data));
   OSTimeDly(1);
   // upper screen data
   //if(position < (LCD_COLS * 2))
   if (screen == LCD_UPPER_SCR)
   {
      enable_upper();
   }else if (screen == LCD_LOWER_SCR) {
      enable_lower();
   } else if (screen == LCD_BOTH_SCR){
      enable_upper();
      enable_lower();
   } else if (screen == LCD_SINGLE_SCR) {
      if (Lcd.position_single < LCD_COLS * 2) {
         enable_upper();
      } else if (Lcd.position_single < LCD_COLS * 4) {
         enable_lower();
      }
   } else {
      LCD_DEBUG(iprintf("send_data invalid screen"));
   }
```

```
}

/* Name: set_data_lines
 * Description: Bit bang to set the GPIO lines corresponding to the
 * data lines.
 * Inputs: unsigned char (8 bits) of data to send.
 * Outputs: none
 */
void set_data_lines(unsigned char data)
{
    //set data lines only
    // 0x7f8 = 0000011111111000
    int  * base;
    base = (int*) SEIKO_LCD_BASE;
    unsigned int Data;
    Data = data;
    *base &= 0x7; // set data bits to 0 and leave contrl bits alone
    *base |= ((Data << 3 ) & SEIKO_LCD_DATA_BITS); // DATA_BITS

     /* LCD_D0 = data & LSBIT;
      LCD_D1 = (data >> 1) & LSBIT;
      LCD_D2 = (data >> 2) & LSBIT;
      LCD_D3 = (data >> 3) & LSBIT;
      LCD_D4 = (data >> 4) & LSBIT;
      LCD_D5 = (data >> 5) & LSBIT;
      LCD_D6 = (data >> 6) & LSBIT;
      LCD_D7 = (data >> 7) & LSBIT;*/
}

/* Name: enable_upper
 * Description: Generates toggle of enable line to write data or
 * commands to the upper two lines of the LCD.
 * Inputs: none
 * Outputs: none
 */
void enable_upper(void)
{
       // simple toggle from low to high to low
       int  * base;
       base = (int*) SEIKO_LCD_BASE;
       *base |= 0x4;
       *base &= 0x7fb;
}

/* Name:enable_lower
 * Description: Generates toggle of enable line to write data or
 * commands to the lower two lines of the LCD.
 * Inputs:
```

```
 * Outputs:
 */
void enable_lower(void)
{
        // simple toggle from low to high to low
//      LCD_E2 = 1;
//      LCD_E2 = 0;
        int  * base;
        base = (int*) SEIKO_LCD_BASE;
        *base |= 0x4;
        *base &= 0x7fb;
}

/* Name: print_char
 * Description: Prints a single character to the LCD to
 * the current cursor position. It wraps as position hits the
 * end of each of the four lines (or rows).
 * Inputs: char c must be a printable ASCII character, a carriage return
 * or a newline character. Anything else will look like garbage.
 * Outputs: none
 */
void print_char(unsigned char screen, char c)
{
    switch (screen) {
        case LCD_UPPER_SCR :
            if (c =='\n') {
                if (Lcd.position_upper < LCD_COLS){
                    move(screen, LCD_COLS);
                } if (Lcd.position_upper < LCD_COLS*2){
                    move(screen, LCD_COLS*2);
                } if (Lcd.position_upper < LCD_COLS*3){
                    move(screen, LCD_COLS*3);
                } else if (Lcd.position_upper < (LCD_COLS*4)) {
                    move(screen, LCD_ORIGIN);
                }
            } else if (c == '\r') {
                if (Lcd.position_upper < LCD_COLS) {
                    move(screen, LCD_ORIGIN);
                } else if ( Lcd.position_upper < (LCD_COLS*2) ) {
                    move(screen, LCD_COLS);
                }
            } else {
                send_data(screen, c);
                Lcd.position_upper++;
                if (Lcd.position_upper > (LCD_COLS*4-1) ) {
                    move(screen, LCD_ORIGIN);
                }
            }
```

```
            break;
        case LCD_LOWER_SCR :
            if (c =='\n') {
                if (Lcd.position_lower < LCD_COLS){
                    move(screen, LCD_COLS);
                } else if (Lcd.position_lower < (LCD_COLS*2)) {
                    move(screen, LCD_ORIGIN);
                }
            } else if (c == '\r') {
                if (Lcd.position_lower < LCD_COLS) {
                    move(screen, LCD_ORIGIN);
                } else if ( Lcd.position_lower < (LCD_COLS*2) ) {
                    move(screen, LCD_COLS);
                }
            } else {
                send_data(screen, c);
                Lcd.position_lower++;
                if (Lcd.position_lower > (LCD_COLS*2-1) ) {
                    move(screen, LCD_ORIGIN);
                }
            }
            break;
        case LCD_BOTH_SCR :
            break;
        case LCD_SINGLE_SCR :
            break;
        default:
            break;
    }

}

/* Name:set_rs_line
 * Description:Sets the RS line to be either hi (1) or low (0).
 * Inputs: unsigned char rs must be either RS_DATA or RS_CMD.
 * Unpredictable results will occur if you send it something else.
 * Outputs: none
 */
void set_rs_line(unsigned char rs)
{
    int  * base;
    unsigned int RS;
    base = (int *) SEIKO_LCD_BASE;
    RS = rs;
    *base ^= ((*base & 0x001)^ RS);

    //LCD_RS = rs;
}
```

```
/* Name: move
 * Description: Moves the current cursor position but does not make
 * any change to data currently being displayed.
 * Inputs: unsigned char col, row. For predictable results col should
 * be between 0 and 39 and row should be between 0 and 3
 * Outputs: none
 */

void move(unsigned char screen, unsigned char position)
{
    if( position > (LCD_COLS * 4)) {
        LCD_DEBUG(iprintf("move bad position"));
        return;
    }
    if (screen == LCD_UPPER_SCR){
       Lcd.position_upper = position;
        send_cmd(LCD_UPPER_SCR, (CMD_DGRAM_ADR + position));
    }
//    else if (screen == LCD_LOWER_SCR){
//        Lcd.position_lower = position;
//        send_cmd(LCD_LOWER_SCR, (CMD_DGRAM_ADR + position));
//    } else if (screen == LCD_BOTH_SCR){
//       Lcd.position_upper = position;
//        Lcd.position_lower = position;
//        send_cmd(LCD_BOTH_SCR, (CMD_DGRAM_ADR + position));
//    } else if (screen == LCD_SINGLE_SCR){
//        Lcd.position_single = position;
//        if (position < (LCD_COLS*2) ) {
//            send_cmd(LCD_UPPER_SCR, (CMD_DGRAM_ADR + position));
//        } else if (position < (LCD_COLS * 4) ) {
//            send_cmd(LCD_LOWER_SCR, (CMD_DGRAM_ADR + (position -
(LCD_COLS*2))));
//        }
//    }
    else {
        LCD_DEBUG(iprintf"move invalid screen\n");
    }
}
/* Name: constructor
 * Description: Constructor for Lcd class. Sets ivar position to known
 * good initial value
 * Inputs: none
 * Outputs: none
 */
/*Lcd()
{
    Lcd.position_single = 0;
```

```
   Lcd.position_upper = 0;
    Lcd.position_lower = 0;
    Lcd.sem_inited = false;

}*/

/* Name: Init
 * Description: Initialises the hardware and the internal semaphore
 * to a known good state. The semaphores guarantee atomic instructions.
 * Inputs: none
 * Outputs: none
 */
void Init(unsigned char screen)
{
     INT8U err = OS_NO_ERR;

    //set hardware lines to gpio, and put into initial state
//  LCD_RS.function(PIN_GPIO); LCD_RS = 0;
//  LCD_E1.function(PIN_GPIO); LCD_E1 = 0;
//  LCD_E2.function(PIN_GPIO); LCD_E2 = 0;
//  LCD_D0.function(PIN_GPIO); LCD_D0 = 0;
//  LCD_D1.function(PIN_GPIO); LCD_D1 = 0;
//  LCD_D2.function(PIN_GPIO); LCD_D2 = 0;
//  LCD_D3.function(PIN_GPIO); LCD_D3 = 0;
//  LCD_D4.function(PIN_GPIO); LCD_D4 = 0;
//  LCD_D5.function(PIN_GPIO); LCD_D5 = 0;
//  LCD_D6.function(PIN_GPIO); LCD_D6 = 0;
//  LCD_D7.function(PIN_GPIO); LCD_D7 = 0;

    int  * base;
    base = (int *) SEIKO_LCD_BASE;
    unsigned int set;
    set = 0x000;
    *base &= set; // set all bits to 0



    //Initialise Semaphore here and not in constructor to avoid
    // and endless trap loop
    if (Lcd.sem_inited == false) {
        Lcd.sem = OSSemCreate(1);
        if(Lcd.sem == NULL){
            LCD_DEBUG(iprintf("Error In SemInit\n"));
        } else {
            Lcd.sem_inited = true;
        }
    }
```

```
   if (Lcd.sem_inited == true){
      OSSemPend( Lcd.sem, WAIT_FOREVER, &err);
      //err = OSSemPend(&sem, WAIT_FOREVER);
      if (err == OS_NO_ERR)
      {
         // exercise 2: decode and replace the send_cmd lines
         //            turn the cursor on top screen
         //            and the blink feature on on the bottom screen

         switch (screen)
         {
            case LCD_UPPER_SCR :
               LCD_DEBUG(printf( "Entering LCD initialization sequence\n");)
               send_cmd(LCD_UPPER_SCR, CMD_FUNCTION | FUNCTION_8BIT |
FUNCTION_1LINE | FUNCTION_5x8 );
               OSTimeDlyHMSM( 0, 0, 0, 5 );
               send_cmd(LCD_UPPER_SCR, CMD_FUNCTION | FUNCTION_8BIT |
FUNCTION_1LINE | FUNCTION_5x8 );
               OSTimeDlyHMSM( 0, 0, 0, 1 );
               send_cmd(LCD_UPPER_SCR, CMD_FUNCTION | FUNCTION_8BIT |
FUNCTION_1LINE | FUNCTION_5x8 );

               send_cmd(LCD_UPPER_SCR, CMD_FUNCTION | FUNCTION_8BIT |
FUNCTION_2LINE | FUNCTION_5x8 );
               send_cmd(LCD_UPPER_SCR, CMD_DISPLAY | DISPLAY_OFF |
DISPLAY_NOCURSOR | DISPLAY_NOBLINK );
               send_cmd(LCD_UPPER_SCR, CMD_CLEAR);
               OSTimeDlyHMSM(0, 0, 0, 2);
               send_cmd(LCD_UPPER_SCR, CMD_ENTRY_MODE |
ENTRY_CURSOR_INC | ENTRY_NOSHIFT );

               send_cmd(LCD_UPPER_SCR, CMD_DISPLAY | DISPLAY_ON |
DISPLAY_NOCURSOR | DISPLAY_NOBLINK );
               send_cmd(LCD_UPPER_SCR, CMD_SHIFT | SHIFT_CURSOR |
SHIFT_RIGHT );
               send_cmd(LCD_UPPER_SCR, CMD_HOME );
               OSTimeDlyHMSM(0, 0, 0, 2);
               break;
//             case LCD_LOWER_SCR :
//               send_cmd(LCD_LOWER_SCR, 56);
//               send_cmd(LCD_LOWER_SCR, 1);
//               send_cmd(LCD_LOWER_SCR, 12);
//               send_cmd(LCD_LOWER_SCR, 6);
//               send_cmd(LCD_LOWER_SCR, 20);
//               send_cmd(LCD_LOWER_SCR, 2);
//               break;
//          case LCD_BOTH_SCR :
//               send_cmd(LCD_BOTH_SCR, 56);
```

```
//                  send_cmd(LCD_BOTH_SCR, 1);
//                  send_cmd(LCD_BOTH_SCR, 12);
//                  send_cmd(LCD_BOTH_SCR, 6);
//                  send_cmd(LCD_BOTH_SCR, 20);
//                  send_cmd(LCD_BOTH_SCR, 2);
//                  break;
//           case LCD_SINGLE_SCR :
//                  send_cmd(LCD_BOTH_SCR, 56);
//                  send_cmd(LCD_BOTH_SCR, 1);
//                  send_cmd(LCD_UPPER_SCR, 12);
//                  send_cmd(LCD_LOWER_SCR, 9);
//                  send_cmd(LCD_BOTH_SCR, 6);
//                  send_cmd(LCD_BOTH_SCR, 20);
//                  send_cmd(LCD_BOTH_SCR, 2);
//                  break;
              default:
                 LCD_DEBUG(iprintf("Init incorrect screen\n"));
          }
          OSSemPost(Lcd.sem);
       } else {
          LCD_DEBUG(iprintf("Sem pend error\n"));
       }
    }
}


/* Name:Clear
 * Description: Clears both screens and resets the hardware and position
 * variable to 0 (upper left position) on both screens.
 * The semaphores guarantee atomic instructions.
 * Inputs: none
 * Outputs: none
 */
void Clear(unsigned char screen)
{
     INT8U err = OS_NO_ERR;

    //err = OSSemPend(&sem,WAIT_FOREVER);
    OSSemPend( Lcd.sem, WAIT_FOREVER, &err);
    if (err == OS_NO_ERR)
    {
       // clear only the upper screen
       if( screen == LCD_UPPER_SCR) {
          Lcd.position_upper = 0;
           send_cmd(LCD_UPPER_SCR, CMD_CLEAR);
       // clear only the lower screen
       }
//        else if (screen == LCD_LOWER_SCR) {
```

```
//            Lcd.position_lower = 0;
//            send_cmd(LCD_LOWER_SCR, CMD_CLEAR);
//        // in this case clearing mirrored screens or the two screens as a
//        // single screen is the same
//        } else if ( (screen == LCD_SINGLE_SCR) || (screen == LCD_BOTH_SCR) ) {
//           Lcd.position_upper = 0;
//            Lcd.position_lower = 0;
//            Lcd.position_single = 0;
//            send_cmd(LCD_BOTH_SCR, CMD_CLEAR);
//        }
        OSTimeDly(alt_ticks_per_second()/5);
        OSSemPost(Lcd.sem);
    } else {
        LCD_DEBUG(iprintf("Clear sem error\n"));
    }
    OSTimeDlyHMSM(0, 0, 0, 2);
}

/* Name: Home
 * Description: Resets the current cursor position to 0 (upper left)
 * on both screens. The semaphores guarantee atomic instructions.
 * Inputs: none
 * Outputs: none
 */
void Home(unsigned char screen)
{
     INT8U err = OS_NO_ERR;
    OSSemPend( Lcd.sem, WAIT_FOREVER, &err);
//    err = OSSemPend(&sem,WAIT_FOREVER);
    if (err == OS_NO_ERR )
    {
        send_cmd(screen,CMD_HOME);
        if (screen == LCD_UPPER_SCR){
           Lcd.position_upper = 0;
        }
//        else if (screen == LCD_LOWER_SCR) {
//            Lcd.position_lower = 0;
//        } else if (screen == LCD_BOTH_SCR) {
//           Lcd.position_upper=0;
//            Lcd.position_lower=0;
//        } else if (screen == LCD_SINGLE_SCR) {
//            Lcd.position_single=0;
//        } else {
//            LCD_DEBUG(iprintf("Home unknown screen");)
//        }
        //OSTimeDly(TICKS_PER_SECOND/10);
        OSTimeDly(alt_ticks_per_second()/10);
        OSSemPost(Lcd.sem);
```

```
    } else {
        LCD_DEBUG(iprintf("Home sem error\n"));
    }
    OSTimeDlyHMSM(0, 0, 0, 2);
}

/* Name: MoveCursor
 * Description: Moves the current position of the cursor to the supplied
 * screen and position. The semaphores guarantee atomic instructions.
 * Inputs: unsigned char screen and position. screen should be
 * one of LCD_UPPER_SCR, LCD_LOWER_SCR, LCD_BOTH_SCR,
LCD_SINGLE_SCR
 * position should be between 0-79 for all screens except for LCD_SINGLE_SCR.
 * If screen is LCD_SINGLE_SCR position can be between 0-159
 * Outputs: none
 */
void MoveCursor(unsigned char screen, unsigned char position)
{

     INT8U err = OS_NO_ERR;

    //err = OSSemPend(&sem,WAIT_FOREVER);
    OSSemPend( Lcd.sem, WAIT_FOREVER, &err);

    if( err == OS_NO_ERR)
    {
        move(screen,position);
        OSSemPost(Lcd.sem);
    } else {
        LCD_DEBUG(iprintf("MoveCursor sem error\n"));
    }
}

/* Name: PrintChar
 * Description: Prints a single character to the LCD at the current
 * cursor position. The semaphores guarantee atomic instructions.
 * Inputs: char c must be a printable ascii character, a carriage
 * return or a newline character
 * Outputs: none
 */
void PrintChar(unsigned char screen, char c)
{
     INT8U err = OS_NO_ERR;

    OSSemPend( Lcd.sem, WAIT_FOREVER, &err);
    //err = OSSemPend(&sem,WAIT_FOREVER);
    if(err == OS_NO_ERR)
    {
```

```
        print_char(screen, c);
        OSSemPost(Lcd.sem);
    } else {
        LCD_DEBUG(iprintf("Print sem error\n"));
    }
}

/* Name: PrintString
 * Description: Prints a c-style string to the LCD at the current cursor
 * position. The semaphores guarantee atomic instructions.
 * Inputs: const char * points to the first letter of the string.
 * Unpredictable results will occur if the string is not
 * null-terminated.
 * Unpredictable results will occur if a NULL pointer is passed in.
 * Outputs: none
 */
void PrintString(unsigned char screen, const char * str)
{
     INT8U err = OS_NO_ERR;
    int i = 0;

    //err = OSSemPend(&sem,WAIT_FOREVER);
    OSSemPend( Lcd.sem, WAIT_FOREVER, &err);
    if(err == OS_NO_ERR)
    {
        while (str[i] != '\0') {
            print_char(screen, str[i]);
            i++;
        }
        OSSemPost(Lcd.sem);
    } else {
        LCD_DEBUG(iprintf("Print sem error\n"));
    }
}

/* Name: ShiftDisplayRight
 * Description:
 * Inputs:
 * Outputs:
 */
void ShiftDisplayRight(unsigned char screen)
{
     INT8U err = OS_NO_ERR;
     OSSemPend( Lcd.sem, WAIT_FOREVER, &err);
     if(err == OS_NO_ERR){
        send_cmd(screen, CMD_SHIFT | SHIFT_SCREEN | SHIFT_RIGHT ); // not
used
        OSSemPost(Lcd.sem);
```

```
    } else {
        LCD_DEBUG(iprintf("Print sem error\n"));
    }


}
void ShiftDisplayLeft(unsigned char screen)
{
     INT8U err = OS_NO_ERR;
     OSSemPend( Lcd.sem, WAIT_FOREVER, &err);
     if(err == OS_NO_ERR){
        send_cmd(screen, CMD_SHIFT | SHIFT_SCREEN | SHIFT_LEFT ); // not used
        OSSemPost(Lcd.sem);
    } else {
        LCD_DEBUG(iprintf("Print sem error\n"));
    }


}
```

Seiko_lcd.h

```
/* This code was modified and tailored for use on a NetBurner MOD5234
 * for University of Alberta CMPE 401 lab sections
 * Justin Smalley, 2008 */


/* Nancy Minderman
 * Fall2009 CMPE401 Modifications.
 * Removed unused routines and simplified class
 */
/*****************************************************************************/
/* GUIDE TO USING LCD CLASS

    1) Connect the LCD hardware as indicated in the lab1 schematic.
    2) pinmap.h contains the PinIO class mappings for the LCD. They
       correspond directly to the schematic version. Don't mess with
       the mappings unless you change the hardware to match.
    3) Construct class at compile time in the heap.
       Lcd myLcd();
    4) Call Initialize function: i.e.
       myLcd.Init();

    Useful LCD Functionalities
    a) wraps around at the end of line
    b) '\n' (ASCII 10) will move to the next line
```

```
   Semaphores:
   The LCD uses a semaphore to ensure that LCD commands are atomic.
   All public functions manage the semaphore so the user
   does not have to worry about contention for the LCD for each
   character and command write. Groups of characters and commands need
   to be dealt with by the application. This issue is the subject of
   lab1.

*****************************************************************************/



#ifndef _SEIKO_LCD_H_
#define _SEIKO_LCD_H_


//#define LCD_COLS        40
#define LCD_COLS        20
#define LCD_ORIGIN      0

#define LCD_UPPER_SCR   0
#define LCD_LOWER_SCR   1
#define LCD_BOTH_SCR    2
#define LCD_SINGLE_SCR  3
#define LCD_UNSET_SCR   4


#define LSBIT           0x01

/******** Display Debug Information **********/
// #define LCD_DEBUG(x) (x)
#define LCD_DEBUG(x)

/**** Echo Characters sent to LCD over serial *****/
// #define LCD_ECHO(x) (x)
#define LCD_ECHO(x)

/* List of available commands */
#define CMD_CLEAR       0x01
#define CMD_HOME        0x02
#define CMD_ENTRY_MODE  0x04
#define CMD_DISPLAY     0x08
#define CMD_SHIFT       0x10
#define CMD_FUNCTION    0x20
#define CMD_CGRAM_ADR   0x40
#define CMD_DGRAM_ADR   0x80
```

```
// Function constants to initialise LCD hardware
#define FUNCTION_8BIT   0x10 // enable 8 pin mode
#define FUNCTION_4BIT   0x00 // enable 4 pin mode (not supported)
#define FUNCTION_2LINE  0x08 // LCD has two lines, line two starts at addr 0x40
#define FUNCTION_1LINE  0x00 // LCD has one continuous line
#define FUNCTION_5x10   0x04 // use 5x10 custom characters
#define FUNCTION_5x8    0x00 // use 5x8 custom characters

/* CMD_DISPLAY constants,  can be OR'd together to select one feature
 * from each pair
 */
#define DISPLAY_ON          0x04    // display on
#define DISPLAY_OFF         0x00    // display off
#define DISPLAY_CURSOR      0x02    // cursor on
#define DISPLAY_NOCURSOR    0x00    // cursor off
#define DISPLAY_BLINK       0x01    // cursor blink on
#define DISPLAY_NOBLINK     0x00    // cursor blink off

/* CMD_ENTRY_MODE constants, can be OR'd together to select one feature
 * from each pair
 */
#define ENTRY_CURSOR_INC    0x02 // increment cursor position
#define ENTRY_CURSOR_DEC    0x00 // decrement cursor position
#define ENTRY_SHIFT         0x01 // shift entire display
#define ENTRY_NOSHIFT       0x00 // don't shift display

/* CMD_SHIFT constants, can be OR'd together to select one feature
 * from each pair
 */
#define SHIFT_SCREEN    0x08 // shift display
#define SHIFT_CURSOR    0x00 // shift cursor
#define SHIFT_RIGHT     0x04 // to the right
#define SHIFT_LEFT      0x00 // to the left

#define RS_DATA         1   // to send data set rs line high
#define RS_CMD          0   // to send a command set rs line low

#define LCD_ERROR       0xff // To signal general LCD errors

#ifndef WAIT_FOREVER
#define WAIT_FOREVER    0
#endif
```

```
//********************** PUBLIC *********************************************



    /* Initialize LCD hardware*/
    void Init(unsigned char screen);

    /* Clear the display of the specified screen. */
    void Clear(unsigned char screen);

    /* Move cursor to home position (top/left) on specified screen. */
    void Home(unsigned char screen);

    /* Moves the cursor to a position on the specified screen */
    void MoveCursor(unsigned char screen, unsigned char position);

    /* Print a single ASCII character to the LCD screen at the current cursor position.
*/
    void PrintChar(unsigned char screen, char c);

    /*Print a c-style string to the LCD screen at the current cursor position */
    void PrintString(unsigned char screen, const char * str);

    /* Shift the screen to the right by one position */
    void ShiftDisplayRight(unsigned char screen);

    void ShiftDisplayLeft(unsigned char screen);


#endif
```

Seiko_lcd_private.h

```
#ifndef SEIKO_LCD_PRIVATE_H_
#define SEIKO_LCD_PRIVATE_H_

#include "includes.h"

struct _LCD {
    unsigned char  position_single;     // current position of cursor
                                // value should be from 0 - 159
    unsigned char  position_upper; // current value of cursor on upper screen
                                // value should be from 0-79
```

```
    unsigned char  position_lower;  // current value of cursor on lower screen
                                    // value should be from 0-79

    OS_EVENT *          sem;            //internal semaphore to guarantee atomicity of
each LCD instruction

    unsigned char   sem_inited;     // Calling Init method more than once should not
reinitialize the semaphore
} Lcd;

    /* pulses the enable pin E1 to write to the upper LCD */
    void enable_upper(void);

    /* pulses the enable pin E2 to write to the lower LCD */
    void enable_lower(void);

    /* print a character to the LCD at the current position
     * of the cursor on the specified screen */
    void print_char(unsigned char screen, char data);


    /* Move the cursor to a screen and position.
     * screen can be upper, lower, or both simultaneously
     * LCD_UPPER_SCR, LCD_LOWER_SCR, LCD_BOTH_SCR
     * position can be from 0-79
     * LCD_SINGLE_SCR
     * position can be from 0-159
     */
    void move(unsigned char screen, unsigned char position);

    /* Sets the RS line for sending either data or commands
     * to the LCD, two possible values RS_DATA or RS_CMD
     */
    void set_rs_line(unsigned char  rs);

    /* Sends a command to the LCD over data lines*/
    void send_cmd(unsigned char screen, unsigned char cmd);

    /* Sends data to the LCD to display.
     * This data must be printable ASCII characters
     * for the data to show up
     */
    void send_data(unsigned char screen, unsigned char data);

    /* Sets the data lines D0 to D7 */
    void set_data_lines(unsigned char data);

#endif /*SIEKO_LCD_PRIVATE_H_*/
```

uC OS-II:

tasks.h

```
#ifndef TASKS_H_
#define TASKS_H_
#include "includes.h"
/* Definition of Task Stacks */
#define     TASK_STACKSIZE       2048
OS_STK      remote_main_task_stk[TASK_STACKSIZE];
OS_STK      start_menu_stk[TASK_STACKSIZE];
OS_STK      test_stk[TASK_STACKSIZE];
OS_STK      remote_control_stk[TASK_STACKSIZE];
//OS_STK      Game_stk[TASK_STACKSIZE];
//OS_STK    start_menu_stk[TASK_STACKSIZE];
//OS_STK    start_menu_stk[TASK_STACKSIZE];

#define REMOTE_MAIN_TASK_PRIORITY   1  // remote defined functionality
#define BUTTON_PROMPT_MUTEX         2  // get_button_push will have a higher
priority than the game
#define REMOTE_CONTROL_PRIO         3
#define LCD_PRIO                    4
#define GAME_MUTEX_PRIO             5  // mutex for game
#define START_MENU_PRIORITY         6  //
#define GAME_PRIO                   7
#define TEST_PRIO                   9

void remote_control( void * pdata );
void remote_main_task( void * pdata );
void start_menu( void * pdata );
void test_task( void * pdata );

#ifndef WAIT_FOREVER
#define WAIT_FOREVER 0
#endif

#endif /*TASKS_H_*/
```

hello_microc.c

```
#include <stdio.h>
#include "includes.h"
```

```
#include "tasks.h"
#include "SeikoLCD.h"
#include "remote.h"

/* Definition of Task Stacks */
#define   TASK_STACKSIZE       2048
OS_STK    task1_stk[TASK_STACKSIZE];
OS_STK    task2_stk[TASK_STACKSIZE];

/* Definition of Task Priorities */

int test_exit( int (*gbp)() ){
   return(BTN_EXIT);
}

int test_pause( int (*gbp)() ){
   return(BTN_PAUSE);
}


/* Prints "Hello World" and sleeps for three seconds */
void test_task(void* pdata)
{
  init_print_lcd();
  init_remote(test_exit,test_pause);
  remote_test_lcd();
}

/* The main function creates two task and starts multi-tasking */
int main(void)
{
//
//  OSTaskCreateExt(test_task,
//                  NULL,
//                  (void *)&test_stk[TASK_STACKSIZE-1],
//                  TEST_PRIO,
//                  TEST_PRIO,
//                  test_stk,
//                  TASK_STACKSIZE,
//                  NULL,
//                  0);
              OSTaskCreateExt(remote_main_task,
                NULL,
                (void *)&remote_main_task_stk[TASK_STACKSIZE-1],
                REMOTE_MAIN_TASK_PRIORITY,
                REMOTE_MAIN_TASK_PRIORITY,
                remote_main_task_stk,
                TASK_STACKSIZE,
```

```
                NULL,
                0);

        OSTaskCreateExt(start_menu,
                NULL,
                (void *)&start_menu_stk[TASK_STACKSIZE-1],
                START_MENU_PRIORITY,
                START_MENU_PRIORITY,
                start_menu_stk,
                TASK_STACKSIZE,
                NULL,
                0);
     OSTaskCreateExt(remote_control,
                NULL,
                (void *)&remote_control_stk[TASK_STACKSIZE-1],
                REMOTE_CONTROL_PRIO,
                REMOTE_CONTROL_PRIO,
                remote_control_stk,
                TASK_STACKSIZE,
                NULL,
                0);

  OSStart();
  return 0;
}
```