1 Introduction

This reference design demonstrates the ease-of-use and functionality of the Xilinx real time operating system, Xilkernel. A digital alarm clock design was chosen to exhibit some Xilkernel features due to its simplicity and familiarity to users.

Please refer to the accompanying *Digital Alarm Clock Users Manual* to quickly configure the board and test alarm clock features. *The Digital Alarm Clock Users Manual* includes a complete description of how to use the alarm clock.

1.1 Design Features

- External and Internal Interrupt Handling
- Static POSIX threads
- Message Queues for task communication
- Priority-based scheduling
- Round Robin scheduling
- Custom 7-segment time multiplexed display
- Custom Color Video Controller
- Bootloader—using XAPP482s Platform Flash Boot Loader

1.2 Hardware and Software Requirements

Hardware:
Spartan™-3 Starter Kit Board, Rev E
Optional RS232 serial cable for UART output
Optional VGA Monitor with VGA cable
Parallel 4 Download Cable or JTAG3 Low Cost Download Cable

Software:
Xilinx Platform Studio 7.1.02i (Build EDK_H.12.5.1+0)
ISE 7.1.03i (Version H.41)

2 Design Contents

2.1 Hardware

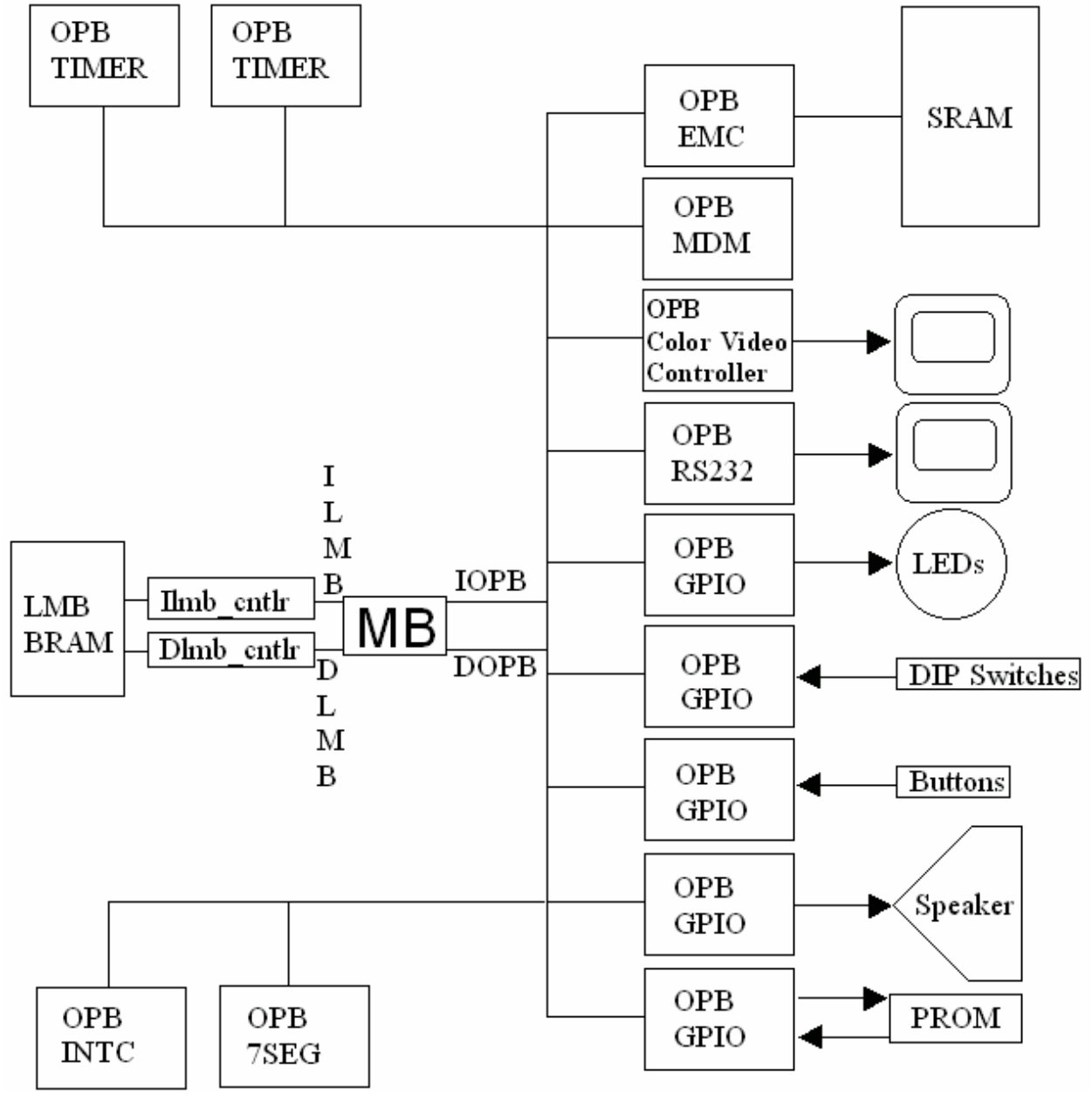

Figure 1. Block Diagram.

OPB Master

- MicroBlaze

OPB Slaves

- Opb Timer 0: System timer used for Xilkernel tick.

- Opb Timer 1: Timer used to generate an interrupt every 60 seconds.
- Opb EMC: External Memory Controller for on-board SRAM.
- Opb MDM: Microprocessor Debug Module used for software debug/download.
- Opb Uartlite: Used for SW debug and optional HyperTerminal output.
- Opb GPIO: Used for output to LEDs, input from DIP switches, input from push buttons, output to optional speaker. GPIO is also used to communicate with the Platform flash during the bootload routine immediately following system power up.
- Opb Intc: Interrupt controller is used to capture kernel tick interrupt, 60-second interrupt, and user push button interrupts.
- Opb 7seg: Custom peripheral. Controller for 7 segment time multiplexed display.
- Opb Color Video Controller: Connects to VGA cable of a color monitor.

Table 1. Memory Map.

Peripheral	Instance	Prefix	Lock	Base Address	High Address	Min Size	Size (KB)
opb_intc	system_intc		☐	0x0c000600	0x0c0006ff	0x20	
opb_timer	opb_timer_2		☐	0x90100b00	0x90100bff	0x100	
opb_timer	opb_timer_1		☐	0x80100b00	0x80100bff	0x100	
opb_color_video_...	opb_color_video_ctrl_0		☐	0xb2000000	0xb200FFFF		
opb_mdm	debug_module		☐	0x80100000	0x801000ff	0x100	
opb_gpio	buzzer		☐	0xFFF00000	0xFFF001ff	0x1FF	
opb_emc	SRAM_256Kx32		☐	0x80200200	0x802003ff	0x200	
opb_emc	SRAM_256Kx32	MEM0	☐	0x80300000	0x803fffff	0	1024
opb_uartlite	RS232		☐	0x80100100	0x801001ff	0x100	
opb_gpio	Push_Buttons_3Bit		☐	0x80100600	0x801007ff	0x1FF	
opb_gpio	PROMREAD		☐	0x90101200	0x901013ff	0x1FF	
opb_gpio	LEDs_8Bit		☐	0x80100200	0x801003ff	0x1FF	
opb_7segled	LED_7SEGMENT		☐	0xFFFF0000	0xFFFF00FF	0x100	
opb_gpio	DIP_Switches_8Bit		☐	0x80100800	0x801009ff	0x1FF	
lmb_bram_if_cntlr	ilmb_cntlr		☐	0x00000000	0x00001fff	0x800	8
lmb_bram_if_cntlr	dlmb_cntlr		☐	0x00000000	0x00001fff	0x800	8

2.2 Software

Upon powering up the system, a bootloader is configured into the FPGA. The bootloader simply copies the final executable image stored in the platform flash prom to external SRAM. Once the program is loaded into SRAM, we enter the main( ) routine where the operating system—Xilkernel—is called.

Once Xilkernel is called and initialized, it calls its scheduler to schedule our tasks[*]. We have configured Xilkernel to have priority-based scheduling as opposed to round robin scheduling.

[*] NOTE: The terms tasks and threads are used interchangeably in this discussion.

Therefore the highest priority task that is not blocked will run. Tasks at the same priority will run round robin in a time-sliced basis. The first task to run is the *Hello Main Task*; It has the highest priority and therefore always runs before any other tasks. The purpose of the *Hello Main Task* is to register the interrupt handlers, enable interrupts, and initialize a timer to interrupt in 60 seconds. The thread then exits and is never scheduled to run again.

The next tasks that will run are the *Display Task*, the *Calculation Task*, and the *Check Alarm Task*. Please refer to *Figure 2* below for the RTOS flow diagram. These three tasks have the same priority level and therefore will run continuously in a round robin time-sliced basis unless the higher priority task—*Sound Alarm Task*—becomes un-blocked.

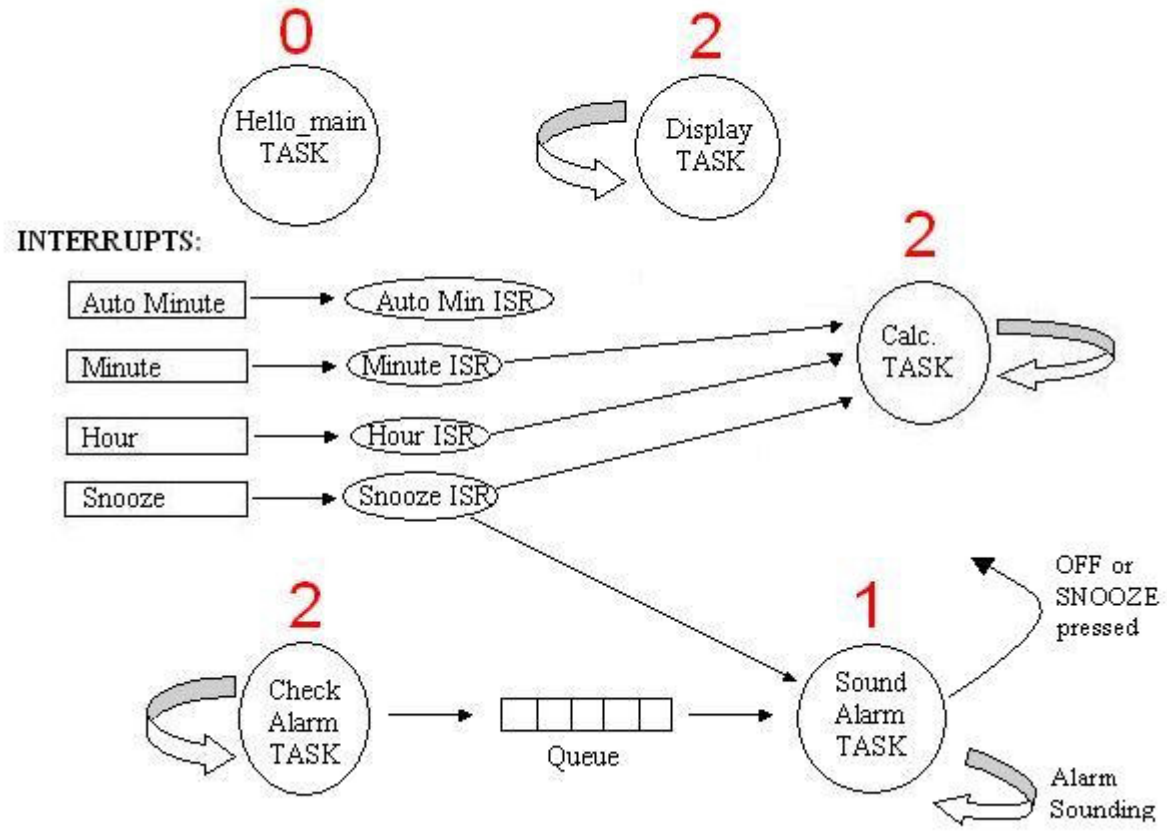

Figure 2. ***Flow diagram.*** *Depicts the software system interrupts, ISRs, queues, and threads. Red numbers represent the priorities of each thread.*

The *Display Task* continually outputs to the 7-segment display, the VGA display, and to the on board LEDs. The 7-segment display will display the actual time or alarm clock time depending on what the user chooses to view. The on board LEDs will display the AM/PM of both the real time and alarm time. A LED also represents whether or not the Alarm is ON or OFF. One can distinguish the real time from the alarm time because the real time has a blinking decimal point.

The VGA will always display both the real and alarm times. The *Display Task* also keeps track of the snooze state and prints a "Snoozing ZZZ" message to the VGA accordingly.

The *Calculation Task* continually calculates what time it is based on user inputs of HOUR and MIN buttons. It keeps track of both the real time and the alarm time. It also keeps track of whether it is AM or PM for both the real and alarm time.

The *Check Alarm Task* continually checks to see whether the real time is equal to the alarm time. If so, and the alarm is ON, the *Check Alarm Task* sends a message to the *Sound Alarm Task* to wake it. The *Sound Alarm Task* is now awake and is unblocked.

The *Sound Alarm Task* will continue to sound the alarm until either the snooze button is pressed or until the user turns OFF the alarm clock. Because the *Sound Alarm Task* has the highest priority in the system (after hello_main exits), it will continue to run by itself until snooze is hit or alarm is turned OFF. As the alarm sounds, *this task* sends a pulse train to a FPGA output pin that can be connected to a speaker to provide audible sound. It also forces the alarm clock to count up to 9999 on the 7-segment display to provide visual stimulation. Optionally, the user can see the alarm sounding in the hyperterminal if he has a RS-232 cable connected. Furthermore, *this task* forces a flashing message "wake up buddy!!" to the VGA screen. When the snooze button is hit or the alarm is turned OFF, the *Sound Alarm Task* exits a loop and then becomes blocked, going back to sleep. As the *Sound Alarm Task* is now blocked, priority-two tasks will now continue to run again in a round robin time-sliced fashion.

Once snooze is hit, the *Calculation Task* adds 10 minutes to the alarm time to give the user a 10-minute snooze. Hitting the snooze button does nothing if the alarm is not sounding. The alarm time will continue to be incremented by 10 minutes each time snooze is hit. The alarm time will not be set back to the original alarm time until the user turns OFF the alarm. In this way, the user is never blinded from the alarm time—opposed to most conventional alarm clocks—and can also wake up at the same time every day.

Every 60 seconds, a timer interrupts to automatically increment the real time minute. The one-minute auto update is put into the *auto minute ISR* (as opposed to the *Calculation Task*) to ensure that an accurate time is maintained regardless of what else may be happening in the system. User push button MINUTE will manually increment the minute of either the real time or the alarm time depending on what the user wants to set. When the user manually increments the real time minute, the auto 60 second timer is reset. User push button HOUR similarly will increment the hour of either the real time or the alarm time.

3 Directory structure

Table 2. Relevant Directory Structure & File Description.

Directory	*File*	*Description*
boot_loop	bootloop.c	Boot loop application used with XMD downloads.
data	system.ucf	User constraints file.
Pcores	opb_7segled_v1_00_a	Custom IP 7-seg controller.
	opb_color_video_ctrl_v1_00_a	Custom VGA Video Controller.
	bus_pad	Custom IP bus pad.
	dabounce	Custom debounce circuitry.
SRAM_boot	promread.c	Bootloader source.
	TestApp.c	Initializes BRAM, calls promread and jumps to final image in SRAM.
	TestAppLinkScr	Custom Linker Script for TestApp.c
	promread.h	Bootloader header file.
RUN_FROM_SRAM	cmd.bat	Batch file to create new MCS file with ELF file appended to original MCS file, Invokes iMPACT.
	pc_ublaze.pl	Pearl script used in cmd.bat
	simple.c	Main application code.
	LED_disp.c	7-seg display driver.
	vga.c , vga.h, def.h	VGA driver, header files.
	new_prom_load.mcs	Pre-made prom file for quick board test.

4 Building Design

There are three software projects in this design: boot_loop, SRAM_boot, and RUN_FROM_SRAM. Project boot_loop is only used as a boot loop application to initialize FPGA BRAM and start the processor until the main application— RUN_FROM_SRAM .elf file—can be downloaded to SRAM with XMD.

SRAM_boot is the platform flash bootloader as described in XAPP482. There are two possible flows included with this design:

1. Configure the FPGA with the boot_loop application, then use XMD to download the main application to SRAM. This flow is useful for debugging since one can quickly re-download new .elf files via XMD. However, once power is cycled on the board, the .elf file will need to be re-downloaded to SRAM. As such, there is a dependency on XMD and a host machine. Consequently, this flow is not a complete stand-alone embedded system.

2. Use the platform flash bootload scheme described in XAPP482. In this flow, we target SRAM_boot to create a download.bit file. Then we create a MCS prom file from download.bit . Then we use the scripts in XAPP482 to create a new MCS file that concatenates the RUN_FROM_SRAM .elf file to the original MCS file. You now have a MCS file that will initialize the FPGA with the bootloader. The bootloader, now running from the FPGA, goes on to fetch the RUN_FROM_SRAM .elf file—located in the platform flash—and copies it into SRAM. This flow serves as a complete stand-alone embedded system. The alarm clock does not have any dependency on a host machine and can therefore operate independently of any other computer system.

The two flows described above are broken into steps in the following two sections.

4.1 Flow 1: Downloading with XMD

1. Press the Generate Netlist icon
   - Generates the synthesis, hdl, and implementation directories.
2. Press the Generate Bitstream icon
   - Runs translate, map, par, and bitgen
3. Make sure **boot_loop** is "marked to initialize BRAMs". You can right-click the boot_loop project in the software applications tab to verify that it is checked.
4. Press the Update Bitstream icon
   - Runs data2mem to update the system.bit with the executable.elf file. A new bit file containing the BRAM initialization strings is created called download.bit
5. Right-click the RUN_FROM_SRAM project and select "Build Project". This will create the main application ELF file—our.elf.
6. Connect Parallel-IV cable to the Spartan™-3 Starter Kit Board (Rev E).
7. Optionally, connect a serial cable to the UART on the Spartan™-3 Starter Kit Board and to any COM port on your computer.
   Start hyper-terminal with the following settings:
   - Baud Rate – 115200
   - Data – 8 bits
   - Parity – none
   - Stop – 1 bit
   - Flow control – none
8. In XPS, press the Download icon within XPS, this will download the boot loop application code which is located in BRAM.
9. Press the XMD icon to start XMD. XMD should automatically identify the JTAG chain and connect to the MDM.
10. Type **source dow-1.cmd** which downloads and runs the application from SRAM.

4.2 Flow 2: Using the Platform Flash Bootloader

1. Press the Generate Netlist icon
   - Generates the synthesis, hdl, and implementation directories.
2. Press the Generate Bitstream icon

- Runs translate, map, par, and bitgen

3. Make sure **SRAM_boot** is "marked to initialize BRAMs". You can right-click the SRAM_boot project in the software applications tab to verify that it is checked.
4. Be sure to associate the linker script **TestAppLinkScr** with SRAM_boot project by double-clicking the SRAM_boot project → directories tab → browse to TestAppLinkScr in the SRAM_boot directory.
5. Press the Update Bitstream icon
   - Runs data2mem to update the system.bit with the executable.elf file. A new bit file containing the BRAM initialization strings is created called *download.bit*
6. Create a MCS file named *prom_load.mcs* from the *download.bit* file created in step 5. It is important to name the MCS file *prom_load.mcs* as step 7 is dependent on this name. The MCS file can be created using iMPACT. Please **save** *prom_load.mcs* into the RUN_FROM_SRAM project subdirectory.
7. Right-click the RUN_FROM_SRAM project and select "Build Project". This will create the main application ELF file—our.elf.
8. Navigate to the RUN_FROM_SRAM project subdirectory. Double-click cmd.bat to create the new prom file. A new prom file called *new_prom_load.mcs* will be created and placed into this directory. iMPACT will also be invoked.
9. Connect Parallel-IV cable to the Spartan™-3 Starter Kit Board (Rev E).
10. Optionally, connect a serial cable to the UART on the Spartan™-3 Starter Kit Board and to any COM port on your computer.
    Start hyper-terminal with the following settings:
    - Baud Rate – 115200
    - Data – 8 bits
    - Parity – none
    - Stop – 1 bit
    - Flow control – none
11. In iMPACT, assign *new_prom_load.mcs* to the platform flash device (You may bypass the FPGA). Erase, Program, and Verify the prom. Press the PROG button to start the alarm clock.

5 Conclusion

This reference design serves as a building block for more complicated Xilkernel operating system designs. Please refer to the EDK OS and Libraries Reference Guide for a thorough description of Xilkernel API. Also see the Platform Studio Users Guide for a complete description on how to use the kernel.

6 References/Credit

1. XAPP482 by Shalin Sheth : http://direct.xilinx.com/bvdocs/appnotes/xapp482.pdf
2. Thanks to Shalin Sheth for 7-seg display and VGA cores
3. XAPP694 by Stephan Neuhold : http://direct.xilinx.com/bvdocs/appnotes/xapp694.pdf
4. EDK OS and Libraries Reference Guide: http://www.support.xilinx.com/ise/embedded/edk_docs.htm
5. Platform Studio Users Guide : http://www.support.xilinx.com/ise/embedded/edk_docs.htm