Using EDK to Run Xilkernel on a MicroBlaze Processor

Example Design

UG758 October 19, 2011

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/02/2003	1.0	EDK 6.1i release
02/15/2005	2.0	EDK 7.1i release
10/24/2005	3.0	EDK 8.1i release
06/23/2006	3.1	EDK 8.2i release
01/08/2007	4.0	EDK 9.1i release
09/02/2007	4.1	EDK 9.2i release
09/19/2008	5.0	EDK 10.1 release
09/16/2009	6.0	EDK 11.1 release
08/11/2010	7.0	Revised to fit new template. EDK 12.2 release.
10/19/2011	13.3	EDK 13.3 release. Version changed to match release number. Design updated for AXI. Board platform changed to ML605. IP source versions updated. Document content structure redesigned for clarity.

Xilkernel on a MicroBlaze Processor Example Design in EDK

Xilkernel on a MicroBlaze Processor Example Design in EDK

This document details an embedded system example design for the ML605 Evaluation Platform board that illustrates the features of Xilkernel executing on the MicroBlaze™ soft processor, using the Xilinx® Embedded Development Kit (EDK). The hardware consists of an ARM AMBA infrastructure connecting the CPU to numerous peripherals using Advanced eXtensible Interface (AXI) and Local Memory Bus (LMB) to build a complete system. This document describes the contents of the reference system and provides information about how the system is organized and implemented.

The design illustrates the usage of each API of Xilkernel. Xilkernel is a small light-weight easy to use kernel, providing features like scheduling, threads, IPC and synchronization with a POSIX subset interface. The hardware design used to illustrate the kernel consists of a MicroBlaze processor connected to two AXI timers, a UART Lite, an AXI interrupt controller, external DDR3 SDRAM controller, and the MicroBlaze Debug Module (MDM) for debugging the processor. The software application consists of Xilkernel and application threads executing on top of the kernel, each illustrating various concepts of the kernel.

Xilkernel and each of the APIs are described in great detail in the chapter on "Xilkernel" in the *OS and Libraries Document Collection*, available on the EDK documentation web page: http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/oslib_rm.pdf.

Run Demos on the Board

Hardware Requirements

- ML605 Evaluation Platform
- Two USB cables (provided in the ML605 package): one each for JTAG and UART

Software Requirements

- Embedded Development Kit (EDK) 13.3
- ISE® 13.3 or later

Implement the Hardware Design

1. Click the **Generate Netlist** button. This generates the synthesis, HDL, and implementation directories.
2. Click the **Generate Bitstream** button. This runs translate, MAP, PAR, and Bitgen. A file called `system.bit` is generated in the implementation directory.

Refer to the EDK documentation for further details.

Connect the Board

1. Connect two USB cables to the board: one for the JTAG USB, and the other for UART USB.
2. Select **Device Manager > Ports (COM & LPT)** to verify the UART COM port number.

   For more information about installing UART USB drivers, refer to *Getting Started with the Xilinx Virtex®-6 FPGA ML605 Evaluation Kit* (UG533).

   Note: The JTAG chain for the ML605 Demonstration Board has been defined by the `download.cmd` file included in the `/etc` directory of the project.

Download the Bitstream

You can download the bitstream to the board using various tools, such as iMPACT, XPS, and SDK. The following steps explain how to download the bitstream in SDK. This is the main development and debug environment in this manual.

1. In XPS, select **Project > Export Hardware Design to SDK**.
2. Make sure the **Include bitstream and BMM file** check box is selected.
3. Click **Export and Launch SDK**.
4. Select the SDK Workspace (located in `<PROJECT>/SDK/workspace`).
5. In SDK, select **Xilinx Tools > Program FPGA**.
6. Make sure that the software configuration is bootloop.

Run the Xilkernel Application

The following steps explain how to run the program in SDK.

1. In the SDK Project Explorer, right-click `xilkernel_demo` and select **Clean Project**.

   Because the Build Automatically option is turned on by default, the application builds automatically after a clean process runs.
2. Select **Run > Run Configurations**.
3. Click the **STDIO Connection** tab and specify the following settings:
   - Enable **STDIO to Console**.
   - Select the corresponding port viewed from Device Manager. Recall that you determined the port in "Connect the Board," page 6.
   - Set the BAUD Rate to **9600**.

4. Select **Run > Run As > Launch on Hardware**.
5. You should see the kernel start up and print messages in the SDK Console, as displayed below. The shell starts to run, clears the screen, and presents a prompt for you to type your commands.

```
XMK: Starting kernel.
XMK: Initializing Hardware.
XMK: System initialization.
XMK: Process scheduling starts.
Idle Task
Idle Task
Idle Task SHELL: Starting clock...
CLOCK: Successfully registered a handler for extra timer interrupts.
CLOCK: Configuring extra timer to generate one interrupt per second..
CLOCK: Enabling the interval timer interrupt...
```

For more information about the demo application, refer to "Demo Application Description," page 14.

Hardware Design

The design is described entirely within EDK to simplify the design flow. It is created using Base System Builder with the cores being configured as mentioned in the memory map.

The design contains the following peripherals in the EDK installation directory.

Hardware Design Files

A listing of the relevant files in the design is shown in Table 1.

Table 1: **EDK Hardware Design Files Description**

Directory	File	Description
<project>	system.xmp	XPS project file
	system.mhs	Microprocessor Hardware Specification file
	system.mss	Microprocessor Software Specification file
<project>/data	system.ucf	Timing constraints and pin locations for ML605 demonstration board
<project>/etc	bitgen.opt	Bitgen command line options
	download.cmd	IMPACT command file for downloading download.bit
	fast_runtime.opt	XFLOW option file for Translate, MAP, and PAR

IP Source Version

The reference design uses the latest IP cores from EDK 13.3. A list of used IP cores and their versions is shown in Table 2.

Table 2: **IP Source Version**

Core Name	Version
microblaze	8.20a
axi_interconnect	1.04a
mdm	2.00b
lmb_v10	2.00b
lmb_bram_if_cntlr	3.00b
axi_uartlite	1.02a
proc_sys_reset	3.00.a
axi_timer	1.03a
axi_v6_ddrx	1.04.a
clock_generator	4.03.a
axi_intc	1.01a
bram_block	1.00a

Memory Map

The memory for this design consists of 8 KB of block RAM connected via the LMB bus to both the data and instruction side of the processor and 64 MB of external DDR3 SRAM with 8 KB ICACHE and 8 KB DCACHE.

Table 3: **Memory Map**

Bus	IP	Base Address	High Address	Size
LMB Bus	microblaze_0_i_bram_ctrl	0x00000000	0x00001FFF	8K
	microblaze_0_d_bram_ctrl	0x00000000	0x00001FFF	8K
AXI Interconnect	RS232_Uart_1	0x40600000	0x4060FFFF	64K
	microblaze_0_intc	0x41200000	0x4120FFFF	64K
	axi_timer_0	0x41C00000	0x41C0FFFF	64K
	axi_timer_1	0x41C200000	0x41C2FFFF	64K
	debug_module	0x74800000	0x7480FFFF	64K
	DDR3_SDRAM	0xBC000000	0XBFFFFFFF	64M

Clocking

The oscillator on the board generates a 200 MHz clock that is connected to the Clock Generator core in EDK. The Clock Generator core uses an MMCM to output the required 100 MHz, 200 MHz, or 400 MHz clock.

Table 4: **Clocking Explanations**

Input	Output	Usage
200 MHz	100 MHz	MicroBlaze, AXI Interconnect, LMB bus, and peripherals
	200 MHz	DDR Controller
	400 MHz	DDR Controller

Software Design

Software Demo Application and Associated Files

The Xilkernel demonstration is organized as one software application project. You can browse through the sources in the `SDK/workspace/xilkernel_demo` directory as you work through this demo. It consists of the following application threads:

Table 5: **Application Threads and Associated Files**

Thread	Description	Associated File
shell	Main controlling thread. Presents a shell with a few simple commands and from which you can launch the other demo threads.	shell.c
prodcon	Producer consumer example thread(s) using message queues.	prodcon.c
llist	Linked list demo using the buffer memory allocation interfaces.	llist.c
sem	Semaphore example showing multiple competing threads using semaphores to co-ordinate.	sem.c
TicTacToe	Simple tic-tac-toe game, which illustrates how to dynamically assign stack memory to a thread when creating it.	tictac.c
TimerTest	Simple time management demo.	timertest.c
prio	Thread illustrating dynamically changing priorities and priority queues in the kernel structures.	prio.c
mutex	Mutex demo, illustrating pthread mutex locks	mutexdemo.c
clock	Simple thread, using the second timer device and handling interrupts from it, to keep track of wall-clock time. This illustrates user-level interrupt handling.	clock.c
push	Simple interrupt system that detects interrupt on the external port when a button is push on the board.	pushbutton.c
standby	Simple illustration of how priority affects execution of threads, by showing that all of the application threads can be put into standby.	standby.c

Software Platform Specification

The example system has a pre-configured software platform with chosen values for each parameter. These Software Platform Specification values are stored in the `<PROJECT>/SDK/Workspace/xilkernel_bsp_0/system.mss` file. Review the following MSS snippet from the MicroBlaze processor example, and examine some parameters and their definitions. The Xilkernel MSS specification is enclosed within an OS block. The MSS example shown below snippet is generated for this software platform.

Click a parameter to view its description.

```
BEGIN OS
 PARAMETER OS_NAME = xilkernel
 PARAMETER OS_VER = 5.01.a
 PARAMETER PROC_INSTANCE = microblaze_0
# Specification of the intc device
 PARAMETER sysintc_spec = axi_intc_0
# Specifying the standard IN / OUT device
 PARAMETER STDIN = RS232_Uart_1
 PARAMETER STDOUT = RS232_Uart_1
# Enhanced features
 PARAMETER enhanced_features = true
 PARAMETER config_kill = true
# Enable diagnostic/debug messages
 PARAMETER config_debug_support = true
 PARAMETER verbose = true
# MSGQ's require config_bufmalloc to be true.
 PARAMETER config_bufmalloc = true
 PARAMETER mem_table = ((4,30),(8,20))
# Semaphore specification
 PARAMETER config_sema = true
 PARAMETER max_sem_waitq = 10
 PARAMETER max_sem = 4
# MSGQ specification
 PARAMETER config_msgq = true
 PARAMETER msgq_capacity = 10
 PARAMETER num_msgqs = 1
# Configure time related features
 PARAMETER config_time = true
 PARAMETER max_tmrs = 10
# Scheduling type
 PARAMETER config_sched = true
 PARAMETER max_readyq = 10
 PARAMETER n_prio = 6
 PARAMETER sched_type = SCHED_PRIO
# Configure pthreads
 PARAMETER config_pthread_support = true
 PARAMETER static_pthread_table = ((shell_main,1))
 PARAMETER config_pthread_mutex = true
 PARAMETER max_pthreads = 10
# MicroBlaze system timer device specification
 PARAMETER systmr_spec = true
 PARAMETER systmr_interval = 100
 PARAMETER systmr_freq = 100000000
 PARAMETER systmr_dev = axi_timer_1
END
```

The Xilkernel MSS specification is enclosed within an OS block. The parameters in bold are the required parameters that you must provide values for. The following table describes each parameter.

Table 6: Software Platform Parameters

Parameter	Description
OS_NAME	These parameters combine to tell Libgen what the OS is.
OS_VER	
STDIN	These parameters provide input and output for the example applications. In this example, you must tell Xilkernel the instance names of the peripherals that are to be used for standard inputs and outputs. In this case, we use the PLB UART Lite peripheral in the system, named RS232_Uart_1, as the input-output device.
STDOUT	
proc_instance	This parameter ties the OS to a particular processor in the hardware system. In this case, it is tied to the processor whose instance name is microblaze_0.
config_debug_support	This parameter controls various aspects of debugging the kernel.
verbose	This sub-parameter of config_debug_support, if set to true, will enable verbose messages from the kernel, such as on error conditions. We set this to true in our example.
systmr_dev	Xilkernel requires a timer to tick the kernel. This parameter works with the systmr_freq and systmr_interval parameters.
systmr_freq	This parameter specifies the frequency at which the timer is clocked, measured in Hertz. In this case, the timer is clocked at 100 MHz.
systmr_interval	This parameter allows you to control the time interval at which the kernel ticks are to arrive. This is automatically determined if a fit_timer is used, since the interval cannot be programmed. Otherwise, a value in milliseconds is provided. In this case, we provide a large granularity tick of 100 ms. This ensures that the output from the application threads are not interleaved and appear mangled on the screen. This setting directly controls the CPU budget of each thread.
sysintc_spec	The example hardware system has multiple interrupting devices, so an interrupt controller is tied to the external interrupt pin of the processor. In this case, the sysintc_spec parameter is set to the instance name of the corresponding peripheral.
config_sched	You configure the scheduling scheme of the kernel with the config_sched parameter. In this example, the scheduling type is set as SCHED_PRIO with six priority levels, and the length of each ready queue is 10.
sched_type	
N_prio	
max_readyq	

Table 6: Software Platform Parameters ***(Cont'd)***

Parameter	Description
config_pthread_support	This parameter specifies whether to support threads. Since this example works with threads, this parameter is set to true.
max_pthreads	This parameter controls the maximum number of pthreads that can run on Xilkernel at any instant in time.
config_sema	This parameter determines whether to include semaphore support in the kernel. In this example, config_sema parameter is enabled.
max_sem	This parameter specifies the maximum number of semaphores required at run time.
max_sem_waitq	This parameter specifies the length of each semaphore's wait queue.
config_msgq	This parameter enables the message queue category. The message queue module depends on both the semaphore and buffer memory allocation modules. Each message queue uses two semaphores internally and uses block memory allocation to allocate memory for messages in the queue. This dependency is enforced both in the GUI and in the library generation process with Libgen. Libgen Design Rule Checkers (DRCs) will catch any errors due to missing dependencies.
num_msgqs	This parameter specifies the number of message queues. This example requires a single message queue for the producer consumer demo thread.
msgq_capacity	This parameter specifies the maximum number of messages that the queue accommodates. In this example, the message queue can contain a maximum of 10 messages.
config_bufmal loc	This parameter defines whether or not to use buffer memory allocation. In this example, buffer memory allocation is needed by both the message queue module and the linked list demo thread, so the parameter is set to true.
config_pthread_mutex	This parameter defines whether to enable mutex lock support. This example has a mutex lock demo thread, so it sets this parameter to true.
config_time	This parameter defines whether to enable software timer support. This example requires software timer support, so it sets this parameter to true.
max_tmrs	This parameter specifies the maximum number of timers. This example uses a maximum of 10 timers for the kernel to support.

Table 6: Software Platform Parameters ***(Cont'd)***

Parameter	Description
`mem_table`	This parameter statically specifies a list of block sizes that the kernel will support when it starts. It consists of a tuple (m,n) where m is the block size, and n is the number of such blocks to be allocated. We have two entries in this array - (4,30) and (8,20). This means that the kernel must support up to 30 requests for memory of size 4 bytes and 20 requests for memory of size 8 bytes.
`enhanced_features` `config_kill`	The enhanced_features parameter defines whether to use enhanced features. One such feature is the ability to kill a process, `config_kill`, as defined in this example. In order to use enhanced features, the enhanced_features parameter must be defined as true, and each feature must also be separately included and defined as true.
`static_pthread_table`	This parameter specifies a list of threads to create at kernel startup. It is made up of an array of tuples (`start_func`, `prio`) which specifies, for each thread, the starting point of execution (`start_func`) and the priority at which it starts (`prio`). For this example, the `static_pthread_table` parameter is specified as (shell_main, 1). The `shell_main()` function is the start of the shell, and the priority is 1.

Parameters whose default values have not been changed do not appear in the MSS file. All of the parameters in the platform specification are translated into configuration directives and definitions in header files. Specifically, for Xilkernel, the files `os_config.h` and `config_init.h` are generated header files that contain C-language equivalents of the specifications in the MSS file. For the system timer device and system interrupt controller device specifications, the header files contain definitions of base addresses of these devices, which are in turn used by the Xilkernel code. These header files are generated under the main processor include directory. In this example, the include directory is *<SDK Workspace>*`/xilkernel_bsp0/microblaze_0/include`.

The information regarding these parameters is available in the Xilkernel section of the *OS and Libraries Reference Guide* (UG643), available at http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/oslib_rm.pdf

Demo Application Description

Shell

The messages display the output of the first two programs from the execution of Xilkernel. The lines of output that begin with "`XMK:` " are debug messages from the kernel. The kernel begins by initializing the hardware. This step includes initializing the interrupt controller and the timer device so that the kernel is interrupted at the configured time interval. Then Xilkernel performs a system initialization, which includes initializing data structures inside the kernel, initializing kernel flags, and creating the statically specified processes and threads. The threads and processes are created in the same order that they were specified in the MSS file. There is also an idle task created by the kernel.

The shell starts, creating a clock thread that illustrates user-level interrupt handling in the following manner:

- Initializes the extra timer peripheral to generate interrupts every one second.
- Registers a handler for this interrupt with Xilkernel.
- Initializes a semaphore with the value `0`.
- Enables the interrupt.
- Performs a `sem_wait()` operation on the semaphore, causing the clock thread to be blocked.

Upon each interrupt, the `extra_timer_int_handler()` handler resets the timer counter device and invokes a semaphore system call to post to the semaphore. This causes the clock thread to be unblocked once every second. Whenever the clock thread is unblocked, it increments its concept of time, and returns to a wait operation on the semaphore, blocking again. This simple thread illustrates how you can register handlers for other interrupts in the system and perform communication between the interrupt handler and your application threads, such as a semaphore post operation as in this example. Your user-level interrupt handlers cannot invoke blocking system calls.

Using the Application Threads

There are three active threads: the idle task, the shell, and the clock. The idle task never runs while there are other higher priority threads in the system, so you do not see it run. The shell runs, providing a prompt on the user computer and responding to user commands. The shell can be used to launch other example threads. The clock thread executes every one second, when the extra timer provides an interrupt.

Consider the POSIX threads API before examining the different application threads. The pthread API implemented in Xilkernel is very close to the POSIX standards; therefore, many applications in the demo can be directly compiled using the compiler for any POSIX operating system and execute without any changes.

One of the first commands used by an application creating threads is:

```
retval = pthread_attr_init(&attr);
```

This system call initializes an attributes structure `pthread_attr_t` that can be used to configure the creation parameters for a new thread. It includes fields to specify the scheduling priority, the detach state of the created thread, and the stack that is to be used. The `pthread_attr_init` system call inserts default attributes in the attribute structure specified. The default attributes are that the scheduling priority is the lowest priority of the system, with no custom stack space and with the threads created in the default detach state, PTHREAD_CREATE_DETACHED. This detach state specifies that the thread's

resources will be automatically reclaimed and completely flushed from the system upon the thread's termination. Specifying the detach state as PTHREAD_CREATE_JOINABLE causes the thread's storage to be reclaimed only when another thread joins, as described in "Making a Thread Joinable." Use `pthread_attr_init()` only if you want to modify some of these attributes later. With other system calls, you can change just the other creation attributes, and untouched parameters will have default values. If only default attributes are required, then do not change attributes at all; instead, use a NULL pointer in the `pthread_create()` system call.

The `pthread_create()` system call dynamically creates a new thread:

```
retval = pthread_create(&tid1,&attr,thread_func,&arg1);
```

At the end of the create system call, a new thread is created, starting from the function specified in the call. This thread starts in the ready state, waiting to be scheduled. The system call returns the ID of the newly created thread in the location of the first parameter. The thread identifier is of type `pthread_t`. This identifier must be used in identifying the target of many thread operations. The created thread executes based on the scheduling policy. For example, if the thread was created with a higher priority, then it executes immediately, whereas if it was created with a lower priority, it executes when the scheduling allows it to. A thread exits by invoking the following code:

```
pthread_exit(&ret);
```

Alternatively, a thread that does not use the `pthread_exit()` call in its body invokes it implicitly. Therefore, use `pthread_exit()` only if you want to return a value to the joining thread.

Making a Thread Joinable

If a thread is configured to be joinable, the call to `pthread_exit()` suspends the calling thread without reclaiming resources and context switches to the next schedulable process or thread. The exit routine takes a pointer argument that points to a return value data structure. This pointer argument can be reclaimed by any thread that joins with this thread. Joins are performed with the `pthread_join()` system call, as shown here:

```
print ("-- shell going into wait-mode to join with launched
program.\r\n");
ret = pthread_join (tid, NULL);
```

This code snippet is run by the shell whenever it launches the application threads. It performs a join, which causes the shell to suspend while waiting for the thread. The thread's identifier is the value contained in tid to terminate. When the target thread terminates, the shell is unblocked and it goes about reclaiming the resources of the target thread. The second parameter to `pthread_join()`, if provided, is used for reclaiming the return value of the terminated thread.

1. Type **`help`** in the shell to see possible commands:

```
shell>help

List of commands
run <program_num>: Run a program. For e.g. "run 0" loads the first
program.
time ?HHMM?      : Set/Display the current time.
standby          : Suspend all tasks for 10 seconds. Idle task executes.
clear            : Clear the screen
list             : List the programs loaded for this example system
help             : This help screen
exit             : Exit this shell
```

2. Set the current time using the time command:

```
shell>time 0637
shell>time
Time is: 06:37:01.
shell>time 0954
shell>time
Time is: 09:54:06
shell>
```

Note: The clear command clears a screenful of the hyperterminal window.

3. Use the `shell>standby` command to have the shell create the standby thread at a higher priority than itself:

```
shell>standby
Idle Task
Idle Task
Idle Task
shell>
```

The standby command illustrates strict priority scheduling. The shell is the highest priority process when it is waiting for input. The shell also always performs a `pthread_join` with the thread that it launches to wait for it to complete. Our standby thread, however, performs a sleep(1000) call, causing it to suspend for 10 seconds. Therefore, all threads are suspended and only the idle task executes.

4. Type **`list`** in the shell to view a list of programs that the shell can load. When the standby thread terminates, then the shell gets control again and it returns the prompt.

```
shell>list
List of programs loaded in this example system
0: MEMORY : Linked list example using buffer memory allocation
1: SEM    : Semaphores example
2: PRODCON: Producer consumer example using message queues
3: TIMER  : Timer example, illustrating software timers
4: TICTAC : TicTacToe thread with dynamically assigned large stack
5: MUTEX  : Mutex lock demo
6: PRIO   : Priority queue demo
7: PUSH   : Push Button Interrupt demo
shell>
```

5. Type **`run 0`** in the hyperterminal session to run the memory allocation example. The shell creates a new thread using the pthread_create system call with default attributes. The output from the execution of llist.elf is shown below:

```
shell>run 0
-- shell going into wait-mode to join with launched program.

LLIST: Sorted Linked List Implementation.
LLIST: Demonstrates memory allocation interfaces.
LLIST: Creating block memory pool....
LLIST: Adding to list 10 statically defined elements....
( 0 1 2 3 4 5 6 7 8 9 )
LLIST: Deleting the list elements.. 0,5,9
LLIST: The list right now is,
( 1 2 3 4 6 7 8 )
LLIST: Adding to list 1535, 661, 2862 and 8.
LLIST: The list right now is,
( 1 2 3 4 6 7 8 8 661 1535 2862 )
LLIST: Deleting block memory pool...
LLIST: Done. Good Bye..
shell>
```

6. This is a linked list program which uses the dynamic buffer memory allocation interface of Xilkernel. It starts off by creating a buffer memory pool of 20 buffers, each of size 8 bytes with the `bufcreate()` system call.

```
i = bufcreate (&mbuft, membuf, 20, 8);
If the call returns successfully, it returns the identifier of the
created memory buffer in mbuft. Every time the application thread needs
to add an element to the linked list, it invokes bufmalloc().
temp = (list_t *)bufmalloc (MEMBUF_ANY, sizeof(list_t));
```

The first parameter should be the identifier of the memory buffer that was created; however, programs can obtain a buffer from any memory buffer available. This program illustrates such a usage of the `bufmalloc()` interface. When you invoke `bufmalloc()` with MEMBUF_ANY as the memory buffer identifier, it requests that the kernel finds the requested sized block from any memory buffer available. The linked list thread, t, deletes an element and releases storage for that element with the `buffree()` method. The application starts by adding some elements, then removing some, then again adding some and terminating.

Semaphore Example

Among the system calls used in this example thread, of interest are the POSIX compliant semaphore calls, as described below:

- Creating a semaphore and getting a handle to it:

  ```
  sem_init (&protect, 1, 1)
  ```

  The `sem_init()` system call creates a new semaphore inside the kernel and returns the identifier of the created semaphore in the location passed as the first parameter. This identifier is of type `sem_t`. The second argument is ignored. The third argument provides the initial value of the semaphore.

- Performing a wait operation on the semaphore:

  ```
  sem_wait (&rzvous_1)
  ```

  The wait operation blocks execution of the calling process until the semaphore is successfully acquired. The semaphore's value indicates the current state of the semaphore. If the semaphore value is greater than zero, then the process decrements this value and successfully acquires the semaphore. If it is less than or equal to zero, then the process blocks.

- Performing a post operation on the semaphore:

  ```
  sem_post (&rzvous_1)
  ```

  The post operation increments the value of the referenced semaphore. If the semaphore value indicates that there are processes waiting to acquire the semaphore, then it unblocks exactly one waiting process from the waiting queue. This queue is a priority queue when scheduling is priority driven.

- Destroying the semaphore:

  ```
  sem_destroy (&protect)
  ```

  This call de-allocates the semaphore resources and removes it from the system. This call fails if there are processes blocked on the semaphore.

The semaphore main thread initializes a total of three semaphores. It uses two of these as flags to converge with two dynamically created threads. That is, it creates these semaphores with an initial value of 0. The created threads, as one of their first steps, perform `sem_wait()` on these converged semaphores. The main thread performs all the thread creation and initialization operations, and flags the threads off by running a post on

both threads. This ensures that both threads start their critical sections as closely as possible.

The threads then contend for the console to perform some message output operations. To prevent the interleaving of the output on the console, they do this inside a critical section. The protect semaphore is used to ensure mutual exclusion while executing in the critical section. The protect semaphore has an initial value of 1. The threads use `sem_wait()` to acquire the semaphore and then use `sem_post()` to release it when they are completed with the critical section.

The two threads contend a couple of times and then terminate. The main thread, which was waiting to join with these two threads, now reclaims their resources, destroys the semaphores, and terminates. Type **`run 1`** in the hyperterminal shell prompt to run this example:

```
shell>run 1
-- shell going into wait-mode to join with launched program.
SEM: Starting...
SEM: Spawning 1...
SEM: Returned TID: 00000003
SEM: Spawning 2..
SEM: Returned TID: 00000004
SEM: Rendezvousing with 1.
SEM: Thread 1: Doing sem_wait.
SEM: Thread 1: 00000000
SEM: Thread 1: 00000001
SEM: Thread 1: 00000002
SEM: Thread 1: 00000003
SEM: Thread 1: 00000004
SEM: Thread 1: 00000005
SEM: ThreadSEM: Rendezvousing with 2.
 1: 00000006
SEM: Thread 1: 00000007
SEM: Thread 1: 00000008
SEM: Thread 1: 00000009
SEM: Thread 1: Doing sem_post.
SEM: Thread 1: Doing sem_wait.
SEM: Thread 1SEM: Thread 2: Doing sem_wait.
: 00000000
SEM: Thread 1: 00000001
SEM: Thread 1: 00000002
SEM: Thread 1: 00000003
SEM: Thread 1: 00000004
SEM: Thread 1: 00000005
SEM: Thread 1: 00000006
SEM: Thread 1: 00000007
SEM: Thread 1: 00000008
SEM: Thread 1: 00000009
SEM: Thread 1: Doing sem_post.
SEM: Thread 2: 00000000
SEM: Thread 2: 00000001
SEM: Thread 2: 00000002
SEM: Thread 2: 00000003
SEM: Thread 2: 00000004
SEM: Thread 2: 00000005
SEM: Thread 2: 00000006
SEM: Thread 2: 00000007
SEM: Thread 2: 00000008
SEM: Thread 2: 00000009
```

```
SEM: Thread 2: Doing sem_post.
SEM: Thread 2: Doing sem_wait.
SEM: Thread 2: 00000000
SEM: Thread 2: 00000001
SEM: ThreadSEM: Successfully joined with thread 1. Return value of
terminated thread: 00000064
SEM: Thread 2: 00000002
SEM: Thread 2: 00000003
SEM: Thread 2: 00000004
SEM: Thread 2: 00000005
SEM: Thread 2: 00000006
SEM: Thread 2: 00000007
SEM: Thread 2: 00000008
SEM: Thread 2: 00000009
SEM: Thread 2: Doing sem_post.
SEM: Successfully joined with thread 2. Return value of terminated
thread: 00000
0c8
SEM: Releasing misc resources..
SEM: Good bye !
shell>
```

Next, execute the producer consumer example. This application solves the producer consumer problem using message queues. An application that uses message queues must include the `sys/ipc.h` and `sys/msg.h` header files to make available standard declarations. Consider the POSIX compliant message queue API that is used in this example:

- Creating a message queue and getting a handle to it:

  ```
  msgid = msgget (key, IPC_CREAT);
  ```

  The `msgget()` system call creates a new message queue inside the kernel and returns an identifier to it. When obtaining a message queue, a unique key is used to identify the message queue. Thus two threads, by agreeing upon a command key, can operate on the same message queue and co-ordinate.

- Performing a blocking message send:

  ```
  msgsnd (msgid, &msg_p, 4, 0)
  ```

  The message send operation blocks execution of the calling process until the message in the buffer msg_p is successfully stored in the message queue. The size of the message to be sent is passed in as an argument. You can make the message send operation non-blocking by using IPC_NOWAIT in the flags.

- Performing a blocking message receive:

  ```
  msgrcv (msgid, &msg_c, 4, 0,0)
  ```

  The receive operation blocks the calling thread until a message is placed on the message queue. The target buffer to store the message and the size of the target buffer are also passed in as parameters. You can make the receive non-blocking by using the IPC_NOWAIT flag. When this call returns successfully, a message is stored in the requested buffer.

- Retrieving the message queue statistics:

  ```
  msgctl(msgid, IPC_STAT, &stats)
  ```

  Statistics about the message queue can be retrieved using the `msgctl()` API. The statistics are placed in the stats structure, which is of type msgid_ds. The statistics that can be retrieved are the number of messages in the queue, the maximum size of the message queue, the identifier of the last process that performed a send operation on the queue, and the identifier of the last process that performed a receive on the queue.

- Removing the message queue:

  ```
  msgctl(msgid, IPC_RMID, NULL)
  ```

  The message queue is removed, again, with the `msgctl()` API. The operation requested should be IPC_RMID. This forcefully clears the message queue and flushes out messages that are in the queue and processes that are waiting on the message queue, either on a send or a receive.

The message queue application begins by creating two threads: a producer thread and a consumer thread. The producer thread continues producing all of the english alphabets from `a` to `t` while the consumer consumes the same. One way to synchronize both the producer and the consumer is to use message queues to store whatever the producer produces, and have the consumer consume from the message queue, which acts as the synchronizing agent in this case. Both the producer and the consumer block sends and receives. Therefore, the producer blocks when all the production buffers (message queue) are full and is unblocked whenever the consumer consumes a message. Similarly, the consumer blocks on empty buffers and gets unblocked whenever the producer produces an item. The main thread also performs some additional operations, such as requesting the statistics of the message queue, verifying that it cannot acquire an existing message queue in exclusive mode, removing the message queue from the system, and ensuring that processes that are blocked on the queue are flushed out of the queue.

Type **`run 2`** in the hyperterminal shell prompt. The following code displays the output from this example:

```
shell>run 2
-- shell going into wait-mode to join with launched program.
PRODCON: Starting..
PRODCON: Spawning Producer..
PRODCON: Producer -- Start !
PRODCON: Producer -- a
PRODCON: Producer -- b
PRODCON: Producer -- c
PRODCON: Producer -- d
PRODCON: Producer -- e
PRODCON: Producer -- f
PRODCON: Producer -
PRODCON: Returned TID: 00000003
PRODCON: Spawning consumer...
- g
PRODCON: Producer -- h
PRODCON: Producer -- i
PRODCON: Producer -- j
PRODCON: Consumer -- Start !

PRODCON: Returned TID: 00000004
PRODCON: Waiting for these guys to finish.
PRODCON: Producer -- k
PRODCON: Consumer -- a
PRODCON: Producer -- l
PRODCON: Consumer -- b
```

```
PRODCON: Producer -- m
PRODCON: Consumer -- c
PRODCON: Producer -- n
PRODCON: Consumer -- d
PRODCON: Producer -- o
PRODCON: Consumer -- e
PRODCON: Producer -- p
PRODCON: Consumer -- f
PRODCON: Producer -- q
PRODCON: Consumer -- g
PRODCON: Producer -- r
PRODCON: Consumer -- h
PRODCON: Producer -- s
PRODCON: Consumer -- i
PRODCON: Producer -- t
PRODCON: Producer done !
PRODCON: Consumer -- j
PRODCON: Consumer -- k
PRODCON: Consumer -- l
PRODCON: Consumer -- m
PRODCON: Consumer -- n
PRODCON: Consumer -- o
PRODCON: Consumer -- p
PRODCON: Consumer -- q
PPRODCON: Successfully joined with producer. Return value of terminated
thread:
00000000
RODCON: Consumer -- r
PRODCON: Consumer -- s
PRODCON: Consumer -- t
PRODCON: Consumer -- Done. ERRORS (1 indicates error in corresponding
message):
00000000
PRODCON: Consumer -- Signalling main.
PRODCON: Starting other tests..
PRODCON: Trying to create a message queue with the same key.
PRODCON: EXCL mode...
PRODCON: SuccePRODCON: Consumer -- Doing other tests...Blocking on
message queue
ssfully failed :). Errno: 00000011
PRODCON: Retrieving msgid for already created msgQ.
PRODCON: Retrieving statistics from message queue.
PRODCON: MsgQ stats:
        msg_qnum    : 00000000
        msg_qbytes  : 00000000
        msg_lspid   : 00000004
        msg_lrpid   : 00000005
End Stats
PRODCON: Attempting to destroy message Q while a process is occupying
it.
PRODCON: Consumer -- Great! Got Kicked out of msgrcv appropriately.
PRODCON: Consumer -- Terminating.
PRODCON: Successfully removed message queue.
PRODCON: Successfully joined with consumer. Return value of terminated
thread: 0
0000000
PRODCON: Releasing misc resources..
PRODCON: Done !
shell>
```

You can run this example an unlimited number of times. Since the semaphore and producer consumer examples are based on the POSIX API, they should be completely portable onto a POSIX OS without any change.

Timer Example

The following are some of the basic interfaces used in this example:

- Getting number of clock ticks elapsed since kernel start:

  ```
  ticks = xget_clock_ticks ();
  ```

  This routine returns the number of times the kernel received a timer interrupt, such as a kernel tick, since the kernel was started. This is a useful measure as a kind of timestamp, or in other time-related calculations.

- Suspending the current task for a certain number of milliseconds:

  ```
  sleep(1000);
  ```

  This routine causes the kernel to suspend the invoking thread for the specified number of milliseconds. The thread regains control after the time elapses.

The timer test thread begins by creating multiple threads, each of which reports the current timestamp. Each thread then attempts to sleep for a different time amount. This can be seen by the idle task executing in between the suspension of the threads. The time actually slept by each thread can be compared against wall clock time and verified.

Type **`run 3`** in the shell prompt. The following code displays output from this demo thread:

```
shell>run 3
-- shell going into wait-mode to join with launched program.
TIMER_TEST: Starting...
TIMER_TEST: Creating 3 threads...
Thread 0 starting..
Thread 0 clock ticks currently: 00021e6a. Sleeping for 6 seconds
Thread 1 starting..
Thread 1 clock ticks currently: 00021e6a. Sleeping for 1 second
Thread 2 starting..
Thread 2 clock ticks currently: 00021e6a. Sleeping for 2 seconds
TIMER_TEST: Clock ticks before: 138858.
Thread 1 done.
TIMER_TEST: Clock ticks after: 138868.
TIMER_TEST: Clock ticks before: 138868.
TIMER_TEST: Creating 1 threads...
Thread 1 starting..
Thread 1 clock ticks currently: 00021e74. Sleeping for 1 second
Idle Task
Thread 2 done.
Thread 1 done.
Idle Task
Idle Task
Thread 0 done.
TIMER_TEST: Clock ticks after: 138918.
TIMER_TEST: End demo...
shell>
```

Tic-Tac-Toe Game

This application thread does not illustrate any kernel interfaces; however, it is slightly unique and brings to light a common run-time requirement, the run-time stack of a thread. Since this thread has nine levels of recursion in its MIN-MAX algorithm, it makes sense to give a bigger stack to it. The static pthread stack size specification, however, applies globally to all threads. You control the stack size selectively for a few special threads by using the shell when creating the tictac thread. This example is displayed in the following code snippet:

```
static char tictac_stack[TICTAC_STACK_SIZE]
    __attribute__ ((aligned(4)));
.
.
.
pthread_attr_init (&attr);
if (opt == 4) {
  /* Special attention to tictac thread */
  pthread_attr_setstack (&attr, tictac_stack, TICTAC_STACK_SIZE);
}
ret = pthread_create (&tid, &attr, (void*)proginfo[opt].start_addr,
NULL);
```

Use the `pthread_attr_setstack()` interface to modify the default thread creation stack attributes. Do this by assigning a memory buffer, tictac_stack in this example, to be used as a stack and by also telling the implementation the size of the stack. Therefore, you can selectively choose to assign a different stack space to threads, preferring over the default fixed stack size allocated by the kernel.

Type **`run 4`** in the shell prompt. Here is a part of the output from the tictac thread:

```
shell>run 4
-- shell going into wait-mode to join with launched program.
TICTAC: Game Starting
TICTAC: Current board -->
   |   |
------------
   |   |
------------
   |   |
TICTAC: Make a move (1-9):
TICTAC: Current board -->
 A |   |
------------
   |   |
------------
   |   |
TICTAC: I am thinking..
TICTAC: Current board -->
 A |   |
------------
   | X |
------------
   |   |
TICTAC: Make a move (1-9):
```

Mutex Demo Application

The following are some of the basic mutex operations that are performed by this application.

- Initializing mutex lock:

  ```
  pthread_mutex_init (&mutex, NULL)
  ```

  The `pthread_mutex_init()` system call creates a new mutex lock within the kernel and returns the identifier of the mutex in the location passed as the first parameter. The type of this mutex identifier is `pthread_mutex_t`. The initialization call requires a second parameter, which gives a pointer to a mutex initialization attributes structure. Because only the basic mutex types are supported, this parameter is unused and NULL or an attribute initialized with `pthread_mutexattr_init()` should be passed in.

  There is an alternative way to initialize the mutex lock statically, by assigning the value PTHREAD_MUTEX_INITIALIZER to the `pthread_mutex_t` structure. This allows the kernel to initialize the mutex lock, in a lazy fashion, whenever it is operated on for the first time.

- Performing a lock operation on the mutex:

  ```
  pthread_mutex_lock (&mutex)
  ```

  This call locks the mutex for the calling process or thread and returns. If the mutex is already locked, then the calling process or thread blocks until it is unblocked by some mutex unlock operation.

- Performing a mutex unlock operation:

  ```
  pthread_mutex_unlock (&mutex)
  ```

  This call unlocks the mutex, which must be currently locked by the calling process or thread, and returns. If there are processes blocked on the mutex, this call unlocks exactly one of them. If scheduling is priority-driven, then it unlocks the highest-priority process in the wait queue. If scheduling is round- robin, then it unlocks the first process in the wait queue.

- Destroying the mutex lock:

  ```
  pthread_mutex_destroy (&mutex)
  ```

  This call destroys the mutex lock. No consideration is given for blocked processes and mutex lock and unlock state.

  The mutex demo application creates some configured number of threads (3 in this case). Each thread contends for a critical section in which it increments a global variable. The main thread looks at this global variable to reach a particular value and then proceeds to join with the threads. The threads use the lock and unlock primitives to access the critical section. The threads also delay inside the critical section to demonstrate contention by other threads. There is also a "bad thread" that tries to do an illegal operation and therefore runs into an error. There are also interfaces for testing the recursive type pthread mutex locks.

Type **run5** to start the mutex demo application. The following code sample displays the output for this example:

```
shell>run 5
-- shell going into wait-mode to join with launched program.
MUTEXDEMO: Starting..
MUTEXDEMO: Launching 3 contending threads..
MUTEXDEMO: Thread(0) starting...
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication frMUTEXDEMO: Thread(1)
starting...
MUTEXDEMO: Thread(1) waiting for indication from main.
MUTEXDEMO: Thread(1) waiting for indication from main.
MUTEXDEMO: Thread(1) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting fm main.
MUTEXDEMO: Thread(1) waiting for indication from main.
MUTEXDEMO: Thread(1) waiting for indication from main.
MUTEXDEMO: Thread(1) waiting for indication from main.
MUTEXDEMO: ThreaMUTEXDEMO: Thread(2) starting...
MUTEXDEMO: Thread(2) waiting for indication from main.
MUTEXDEMO: Thread(2) waiting for indication from main.
MUTEXDEMO: Thread(2) waiting for indication fro
MUTEXDEMO: Providing indication to waiting threads to start
contending...
MUTEXDEMO: Waiting for threads to get past critical section...
or indication from main.
MUTEXDEMO: Thread(0) contending...
MUTEXDEMO: Thread(0) in critical section..Will spend some time here.
d(1) waiting for indication from main.
MUTEXDEMO: Thread(1) contending...
m main.
MUTEXDEMO: Thread(2) contending...
MUTEXDEMO: Waiting for threads to get past critical section...
MUTEXDEMO: Thread(1) in critical section..Will spend some time here.
MUTEXDEMO: Thread(0) mutex done...
MUTEXDEMO: Waiting for threads to get past critical section...
MUTEXDEMO: Thread(2) in critical section..Will spend some time here.
MUTEXDEMO: Thread(1) mutex done...
MUTEXDEMO: Thread(2) mutex done...
MUTEXDEMO: BAD_THREAD: Starting..
MUTEXDEMO: I am going to try to unlock a mutex I don't own and force an
error.
MUTEXDEMO: Good! I got the right error!
MUTEXDEMO: BAD_THREAD Done.
MUTEXDEMO: Destroying mutex locks...
MUTEXDEMO: Done. Good Bye !
shell>
```

You can run this demo an unlimited number of times.

Next, run the last application thread. This thread illustrates priority scheduling and dynamic priority. It also illustrates how priority is ingrained even in the wait queues of primitives like semaphores, mutex locks, and message queues.

Create several threads with different priority ordering. All the threads block on a single semaphore which is initialized to 0. The main thread unblocks one single thread using sem_post. Subsequently, threads must be unblocked in priority order, instead of the original blocking order. This is confirmed by the outputs issued by the threads. One single LOW priority thread also tests the `sem_timedwait()` API by repeatedly attempting to acquire the semaphore with a time out specified. This thread is the last thread that acquires the semaphore, but should have timed out several times in between. The main thread joins with the remaining threads and terminates this portion of the test.

In the second stage of the test, the main thread creates another thread at highest priority. It then tests this thread by changing its priority at regular intervals and sleeping in the intervals. This is confirmed by the puppet thread not producing any output in the intervals that the main thread sleeps. The main thread then kills this puppet thread using the kill system call.

The following are the relevant interfaces illustrated with this example:

- Specifying thread priority during creation:

  ```
  spar.sched_priority = prio[i];
  pthread_attr_setschedparam(&attr,&spar);
  ```

  This snippet illustrates how the priority of a thread is controlled while creating a thread. By setting the priority attribute of the thread creation attributes and passing the same attributes structure to the `pthread_create()` call, the priority of a thread is controlled.

- Dynamically changing the priority of a thread:

  ```
  retval = pthread_create(&lowpriotid, &attr, low_prio_thread_func,
  NULL);
  spar.sched_priority = (NUM_PUPPET_THREADS - 1) - i;
  if ((retval = pthread_setschedparam (puppet_tid[i], 0, &spar)) != 0)
  ```

  The `pthread_setschedparam()` call changes the priority of a thread. This snippet shows how the priority of the puppet threads are flipped at runtime by the main thread.

- Advanced features: killing a thread:

  ```
  if (kill (main_thread_pid) != 0) {
  ```

  This snippet shows how the low priority thread kills the main thread. Notice that the identifier passed to the kill interface is a different identifier, not of type pthread_t. This is because POSIX threads does not define a `kill()` interface. This is a custom interface exported by Xilkernel. Therefore, use the underlying process context identifier to kill the thread. The process context identifier is retrieved by using the following call:

  ```
  main_thread_pid = get_currentPID ();
  ```

Type **`run 6`** in the shell prompt to run this application thread. The following code snippet displays the output for this example:

```
shell>run 6
-- shell going into wait-mode to join with launched program.
PRIOTEST: Starting...
PRIOTEST: Spawning: 0.
Thread(0): Starting...
PRIOTEST: Returned TID: 3.
PRIOTEST: Spawning: 1.
Thread(1): Starting...
PRIOTEST: Returned TID: 4.
Thread(LOWPRIO): Starting. I should be the lowest priority of them
all...
PRIOTEST: Returned TID: 5.
PRIOTEST: Yawn..sleeping for a while (400 ms)
Thread(LOWPRIO): TIMEDOUT while trying to acquire sem.
Thread(LOWPRIO): TIMEDOUT while trying to acquire sem.
Thread(LOWPRIO): TIMEDOUT while trying to acquire sem.
PRIOTEST: Time to wake up the sleeping threads...
Thread(1): Acquired sem. Doing some processing...
Thread(1): Done...
Thread(LOWPRIO): TIMEDOUT while trying to acquire sem.
Thread(0): Acquired sem. Doing some processing...
Thread(0): Done...
PRIOTEST: Joining with threads...
PRIOTEST: Allowing the LOWPRIO thread to finish...
PRIOTEST: Joining with LOWPRIO thread...
Thread(LOWPRIO): TIMEDOUT while trying to acquire sem.
Thread(LOWPRIO): Acquired sem. Doing some processing...
Thread(LOWPRIO): Done...
PRIOTEST: Dynamic priority test phase starting !
PRIOTEST: Initializing barrier semaphore to value 0...
PRIOTEST: Creating puppet threads...
PUPPET(0): Starting...
PUPPET(0): Blocking on semaphore...
PUPPET(1): Starting...
PUPPET(1): Blocking on semaphore...
PRIOTEST: Now I am flipping the priorities of all the blocked puppet
threads.
PRIOTEST: Now I am posting to the semaphores and releasing all the
puppets.
PUPPET(1): Got semaphore...
PUPPET(1): Releasing semaphore...
PUPPET(1): DONE...
PUPPET(0): Got semaphore...
PUPPET(0): Releasing semaphore...
PUPPET(0): GRR Taking revenge for being demoted in priority..
PUPPET(0): Killing main thread before I die...
shell>PUPPET(0): SUCCESS.
PUPPET(0): DONE...

shell>
```

Push Button Application

Xilkernel takes care of all primary interrupt handling requirements from the user application. Xilkernel can handle multiple interrupts when connected through an interrupt controller. An instance of interrupt controller should be provided in Xilkernel setup in software platform settings. When interrupt occurs in Xilkernel system, an interrupt handling routine within the kernel is called first. This handler then calls the interrupt handler routine of the interrupt controller. From this point, the handler for the interrupt controller invokes the user specified interrupt handlers for the various interrupting peripherals.

There are five user level interrupt handling APIs in Xilkernel:

```
unsigned int register_int_handler(int_id_t id, void *handler)(void*),
void *callback)
void unregister_int_handler(int_id_t id)
void enable_interrupt(int_id_t id)
void disable_interrupt(int_id_t id)
void acknowledge_interrupt(int_id_t id)
```

The `register_int_handler( )` function registers the specified user level interrupt handler as the handler for a specified interrupt. Internally this function calls `XIntc_Connect (&sys_intc, intr_id, (XInterruptHandler)handler, callback)`, which registers interrupt handler in the interrupt vector table of the Interrupt controller for the particular source of the interrupt defined by `intr_id`.

When the kernel detects an interrupt, the interrupt handler in the kernel calls the interrupt controller handler, which calls the handler registered by `XIntc_connect( )`.

`Unregister_int_handler( )` calls `XIntc_Disconnect (&sys_intc, intr_id)`,

`enable_interrupt( )` calls `XIntc_Enable (&sys_intc, intr_id)`,

`disable_interrupt( )` calls `XIntc_Disable (&sys_intc, intr_id)`, and

`acknowledge_interrupt( )` calls `XIntc_Acknowledge (&sys_intc, intr_id)`.

With interrupt in Xilkernel, it is not necessary to call in `acknowledge_interrupt( );` it is automatically handled by the interrupt controller handler. However, you can change the interrupt controller interrupt handling routine by using this API.

Interrupt Handling Mechanism in Xilkernel

This section describes the list of steps required to handle interrupts from a peripheral.

Following are the steps that were mentioned for handling interrupt in the standalone system:

1. Define Interrupt handler for Push Button
2. Initialize exception handling for MicroBlaze processors
3. Register external interrupt handler
4. Register Push Button interrupt handler
5. Start the interrupt controller
6. Enable Push Button interrupts in the interrupt controller
7. Enable MicroBlaze processor non-critical interrupts

The steps in handling interrupt in Xilkernel is the same, but some of the steps are taken care of by the kernel. Also, the kernel provides an API to register a peripheral handler. Following are the general steps for handling an interrupt in Xilkernel system:

1. Define the peripheral thread.
2. In the peripheral thread, register the peripheral interrupt handler using the register_int_handler API available in Xilkernel.
3. In the peripheral thread, enable peripheral interrupt and enable interrupt for that particular peripheral in the interrupt controller.
4. In `shell_main( )`, start the peripheral thread.

The following steps are listed in the standalone system is handled by the kernel itself:

1. Initialization of exception handling for MicroBlaze processors
2. Registering external interrupt handler
3. Starting interrupt controller
4. Enabling MicroBlaze processor non-critical interrupts

Type **`run 7`** in the shell prompt to run this application thread. The following code snippet displays the output for this example:

```
-- shell going into wait-mode to join with launched program.
PUSH BUTTON: push_button.c
PUSH BUTTON: ************************************************
PUSH BUTTON: --                                            --
PUSH BUTTON: --                 Press Button               --
PUSH BUTTON: --                                            --
PUSH BUTTON: ************************************************
PUSH BUTTON: Press Button ten times to exit...
```

After pressing the middle Push Button 10 times, the application will return to the shell main:

```
PUSH BUTTON: Push button interrupt[0]
PUSH BUTTON: Push button interrupt[1]
PUSH BUTTON: Push button interrupt[2]
PUSH BUTTON: Push button interrupt[3]
PUSH BUTTON: Push button interrupt[4]
PUSH BUTTON: Push button interrupt[5]
PUSH BUTTON: Push button interrupt[6]
PUSH BUTTON: Push button interrupt[7]
PUSH BUTTON: Push button interrupt[8]
PUSH BUTTON: Push button interrupt[9]
PUSH BUTTON: Exiting push_main...
shell>
```

When you are done with the examples, type **`exit`** in the shell prompt. This ends all the application threads and leaves only the idle task to continue.