Implementation of a Real Time Operating System on Altera’s Cyclone IV FPGA

Sagar Kakade, Hamid Shahnasser
School or Engineering
San Francisco State University
ksagard@yahoo.com

Abstract: An operating system is a piece of software written for managing hardware resources and applications that run on top of the operating system. A real time operating system RTOS also does the same task, but it is specifically designed to serve applications that need stimulus in real time and provide the desired response in real time and with high reliability. Real time applications guarantee response within strict time constraints especially in milliseconds or sometimes even in microseconds. This response is important in measurement, control and automation systems. Porting a real time operating system to a microprocessor is a critical step in the field of embedded system engineering. To study this crucial step, this paper outlines the Board Support Package port of uC/OS-II to a custom RISC processor built using FPGA. The aim of this paper is to study the steps in designing a raw processor from logic elements, developing a BSP for it, porting a real time operating system on it and finally demonstrating task context switches, interrupts and inter-task communication using mailboxes on Altera’s DE2-115 development kit.

Keywords: Embedded Systems; Altera DE2-115; Nios II; uC/OS-II; RTOS

I. INTRODUCTION

The number of tasks that a modern day embedded system schedules and executes is increasing rapidly. This has in turn emphasized the importance of response time of the system. Operating systems traditionally tend to improve the user responsiveness of the system with every new version; which may or may not improve the response factor of the system. The design and operation of operating systems that provide time critical response and throughput is different from that of normal user oriented operating systems like Windows or the Mac. Such operating systems have been designed with complex scheduling algorithms to minimize the response time of processor to a task. These operating systems are called Real Time Operating Systems for Embedded Systems. An RTOS processes applications in parallel using threads of execution. This is done by allocating fixed time slots to processes and implementing them in parallel. This method is called Multitasking. In a Multitasking environment, processes and applications are executed in parallel using methods like multithreading and synchronization. Embedded systems are constrained in memory space and resources usage. To accommodate the OS in such constrained memory, it becomes imperative for its footprint to be minimal. Hence to order to keep the memory footprint small, the RTOS contains only few task management routines, inter-task communication mechanisms, memory management and scheduler codes. Although multi-core architectures are being introduced in embedded systems like smart phones, many
other embedded systems cannot incur the overhead of having another core for a processor on the chip. All these constraints greatly influence the design of a RTOS. An example of such a RTOS is UC/OS-II developed by Jean J Labrosse and currently maintained by Micrium Incorporated[1]. The basics of porting this RTOS on to a custom developed processor from logical elements are outlined in this paper.

II. IDEA OF THE PROJECT

The main aim of this project is to port an operating system on a microprocessor. For this purpose, a custom Nios II architecture along with a real time operating system was chosen. The steps for porting a real time operating system and general operating system like uCLinux are almost similar. The ways of compiling the source of the operating system vary according to the type of operating system. The Nios II architecture supports real time operating system like TinyOS, FreeRTOS and uC/OS-II. The Nios II architecture is to be built on a Field Programmable Gate Array (FPGA). A FPGA is an integrated circuit which contains logical blocks and a hierarchy of reconfigurable interconnects that allow a programmer to reconfigure the FPGA if the application at hand needs any interconnectivity post production (The FPGA Place and Route Challenge). Beside these reconfigurable interconnects, a FPGA contains memory elements like Random Access Memory (RAM), Read Only Memory (ROM), and Flash memory. These can be used as on chip memory or off chip memory for the microprocessor to load/store from it. The Nios II microprocessor is a 32 bit general purpose RISC processor. It is property of Altera Corporation. It is a type of soft processor which means it can be used to configure the instruction set, peripherals, memory management unit and memory protection unit.

uC/OS-II is a real time operating system written by Jean J Labrosse and maintained by Micrium Corporation[1]. It is mostly written in C programming language. A port of uC/OS-II was available and distributed through Altera Corporation. But this port was specific to the development board. Software distributed by Altera Corporation is to be used while programming with a port of uC/OS-II for the Nios II architecture. To make this independent, a Board Support Package to port this operating system was proposed to be developed so that the basics of developing a BSP, compiling an operating system and porting it could be learnt in detail.

Once the operating system port is developed, an application to test the peripherals using Hardware Abstraction procedure calls super-imposed by uC/OS-II task management calls will be written. This application will allocate memory for the tasks, allocate a priority number to the task and schedule it according to its priority number. A task in uC/OS-II is an infinite loop. This loop will contain application code to test all the peripherals. One task is responsible for the functioning of one peripheral. Four such tasks will demonstrate the working of four different peripherals. An interrupt subroutine will be written for a hardware interrupt that will display a message on the LCD conveying the fact that an interrupt has actually occurred.
III. BACKGROUND KNOWLEDGE

To get a better understanding of this paper, this topic explains some of the technical terms used to implement the real time operating system on the development kit. Firstly, a detailed explanation of how the operating system sits on top of the hardware is given. This includes concepts like the functioning of the device drivers, middleware and that of the application software itself. After this, the development kit, the real time operating system used and the types of porting methods available are discussed. An overview of the steps to complete the project is discussed in the next few paragraphs.

A. Layered Architecture of an Operating System

Figure 1 shows the layers of software and hardware that build up a system with a functional application running on it. The hardware layer consists of the actual physical hardware that makes a system. It includes the FPGA, the peripherals and the interfaces connected to the FPGA in reference to this paper. The device drivers run on top of the hardware. It enables the above layers to interact actively with the hardware. It is part of the software that runs the hardware. With reference to this project, device drivers are pieces of codes written to make the FPGA recognize the peripherals connected to it and make them functional. They are termed as Hardware Abstraction Layer (HAL) in Altera’s code library. Above the device drivers, resides the complete operating system and its kernel. The operating system is responsible for task management, hardware functions like memory management and input output. It acts as an intermediary between the hardware and the application programs that run on it. uC/OS-II is the real time operating system that runs on the FPGA for this project. In its most general sense, middleware is computer software that provides services to software applications beyond those available from the operating system. Middleware can be described as “software glue” (Defining Technology Inc). Middleware is used for services that provide a layer of abstraction that hides details about the hardware from the application running from it. The application software that is at the top of the hierarchy is the user application written for a piece of hardware which runs the operating system. This is mainly the user space where the user has freedom to write a code that interacts with the operating system and in turn with the hardware. The code written for creating the tasks and implementing their functionality is the user space application software.

Fig.1. Layered Architecture of a system running an OS

B. UC/OS II - Real Time Operating System

UC/OS-II (Micro Controller Operating System Version 2) is a simple priority based pre-emptive real time embedded operating system used typically in applications where
requests by the interfaces and peripherals are made in real time[1]. This operating system is mostly written in C programming language. A key consideration here is not the throughput but the guarantee of performance based on the time of request and time of service. This core operating system has a footprint of about 20 Kbytes. This real time operating system can manage task creation, building task stack and check the task stack, task deletion, static changes to the task priority, suspension and resumption of tasks and extract information from the task control block. This kernel can handle up to 64 tasks out of which 56 tasks are user defined tasks. Each task has a priority assigned to it. Inter task communication is handled by message mailbox, semaphores and message queues. General purpose operating systems are designed to maintain user responsiveness for all the multiple program instances running. On the other hand, real-time operating systems are designed to run critical applications reliably and with precise timing barring the user response.

C. Architecture of uC/OS-II

Figure 2 takes deeper look at the architecture described above. This layered architecture is specific to the system that runs uC/OS-II. The CPU is the Nios II processor that is built out of the Cyclone IV FPGA from Altera. This is a 32 bit RISC processor and has a timer configured according to the requirement of uC/OS-II. The timer is required for the accurate functioning of the real time operating system because it is responsible for interrupt generation and task service time determination. The operating system takes care of task and interrupts management. Processor specific code is the actual port of the uC/OS-II operating system. This consists of the header, assembly and C source code of the operating system which is dependent on the type of processor and architecture. Hence this is called processor specific code. The application specific code for uC/OS-II is used to configure the operating system for the application that the user is willing to run on the hardware. This includes the file OS_CFG.H and INCLUDES.H. The processor independent code is the actual C source code for real time operating system kernel and its functions like task management, inter-task communication and time management. It consists of files like OS_CORE.C, uCOS_II.C, OS_TASK.C, OS_TIME.C, OS_SEM.C, OS_MEM.C and many more (Porting uC/OS-II). This code is the actual system call that the application software layer makes to the operating system. This may be termed as the kernel space which is invoked by the application software through a system call. This layer of abstraction is important to hide the kernel from the user level applications.

---

Fig.2. Layered Architecture of a system running uC/OS-II
D. The Development Kit – Altera DE2-115 with Cyclone IV Field Programmable Gate Array

Altera’s DE2-115 FPGA development kit is equipped with Cyclone IV family of FPGA as the nucleus of the board. This FPGA supports the Nios II architecture of microprocessor family. The Nios II processor is built using hardware design language like VHDL or Verilog. The Nios II processor is a 32-bit RISC processor which can support speeds up to 200 MHz and is compatible with operating systems like UC/OS-II and uCLinux. The processor can drive various GPIO present on the board, Ethernet, USB ports and a variety of I/O terminals which makes it suitable for a wide range of applications from Signal Processing to System Control. Besides, Nios II architecture is a soft-core architecture that allows the user to build a custom Nios II processor designed for a specific application.

The development kit has on-chip and off-chip memories that make the process of porting an operating system easier. One can select SRAM or SDRAM for the kernel of the desired operating system to be loaded. A Flash memory is also provided and it the recommended memory for the operating system kernel. Given these features of the development kit, it is ideal for the aforementioned project at hand.

E. Porting UC/OS-II On FPGA

Porting is the process of adapting software so that an executable program can be created for a computing environment that is different from the one for which it was originally designed for (Porting, 2012). Porting involves making the hardware platform, its architecture and the code running on it recognizable to each other. Thus porting is essential for the software of the operating system to understand the hardware it is running on. Porting an operating system involves two types of ports[2].

1. Architecture port
2. Board port which is sub divided into Basic and Board Support Package.

Architecture port

The main task of the Operating system is to support multi-tasking. Architecture porting is basically only about providing context switches and exception handling which are the basic tasks of the operating system.

Context switches

A context switch, also known as process or task switch is the switching of the central processing unit from one process or thread to another. A process or a task is an executing instance of a program in the running application code. Context switch is the responsibility of the operating system kernel. When a multitasking kernel requires pre-empting a running task to deploy a waiting task, it needs to save the current task’s context which includes the CPU registers, task state and task control block in the current task’s context storage area. This storage area is specific to every task and is called its stack. Once this is done, it restores the new task’s context from its stack and then resumes execution of the new task’s code. This process is called a context switch or a task switch. Context switching adds overhead to the application. The more registers a CPU has, the higher the overhead because of the time the kernel takes to
save the content of the registers. The time required to perform a context switch is determined by how many registers have to be saved and restored by the CPU.

**Exception Handling**

In operating systems, the process of changing the normal program flow by responding to the occurrence of exceptions - anomalous or exceptional situations requiring special processing during computation, is called exception handling (Exception Handling, 2012). Exception handling in operating systems is provided by specialized programming language constructs or software that directs the program to a handler. Normally, an exception is handled by saving the current state of execution in a predefined place and switching the execution to a specific subroutine known as an exception handler. If exceptions are continual, the handler may later resume the execution at the original location using the saved information.

**Basic Board Port**

This type of port includes developing a code to map the peripherals from the board to the memory addresses in the operating system’s processor dependent code. Thus, the Basic Board Porting involves the following[2]:

1. Check proper board operation
2. Provide service from at least basic peripherals – Memory, ICU, PLL, Timer, Input / Output

**Board Support Package**

A board support package or a BSP is an implementation specific support code for a given microprocessor board that conforms to a given operating system. It is generally built with a boot loader that contains the minimal device support to load the operating system and device drivers for all the devices on the board. A Minimum BSP developed for the hardware is similar to basic board porting for the same hardware.

The BSP developed for this project takes care of the following functions[2]:

1. Initializing the processor
2. Initializing the memory
3. Initializing the bus and the interrupt controller
4. Initializing the clock

uC/OS-II porting on any type of microprocessor requires the following conditions fulfilled or has the following pre-requisites.

1. The proposed microprocessor should support interrupts
2. The proposed microprocessor should generate interrupts at regular rates
3. Interrupts can be enabled or disabled by the microprocessor
4. The microprocessor can support hardware stack
5. The microprocessor can load or store stack pointer and CPU registers

**IV. Implementing UC/OS-II ON FPGA**

The implementation of this project can be divided into two parts, the hardware development and the software development. The
design, development and testing of a processor based system using Verilog hardware description language can be termed as the Hardware aspect of the project. The application code for the processor and its interfaces, a Board Support Package for the operating system, kernel development and port, and uC/OS-II API comprise of the Software development aspect of the project. The hardware aspect of design is explained in the next few paragraphs.

A. Hardware Implementation

For the operating system to be functional, it needs to have a base system on which it will be ported. For the scope of this project, the base system is built using Altera’s Quartus II IDE. The development kit used for this project is Altera’s DE2-115 with Cyclone IV family of FPGA. Using SOPC builder utility, the system components are connected to each other in a desired manner and their instance is created. The programming language used to link all the interfaces, memory subsystem and the processor is Verilog. The system runs at 200 MHz and is connected to a video buffer pipeline and a memory subsystem consisting of a SDRAM and Flash.

Hardware Architecture of System

Figure 3 describes the hardware architecture of the system built using the Nios II processor. The Nios II processor is built from the logical elements or blocks present in the FPGA. Almost all the components listed in Figure 3 are part of the Cyclone IV FPGA present on the development kit. The Flash memory and the SDRAM are off chip memory connected to the FPGA using a tri state bridge and wires respectively. The SDRAM, FLASH and ONCHIP memory constitute the memory subsystem of the proposed system. The specification of this subsystem is given in Table 1. The Phased Locked Loop (PLL) is used to operate components at a frequency which is different than the global frequency used for the system. In our case, the PLL is used to switch phase of the frequency to communicate with the SDRAM and VGA controller. The exact values of the frequencies used are given in Table 1 below. The peripherals connected to the Nios II processor are shown below. A series of 8 seven segment displays, one 16x2 LCD, eight red LEDs and sixteen green LEDs have been assigned different tasks. The functioning of each of these peripherals is described in the task behavior. The JTAG UART is used to download the text section of the code on to the processor. If this peripheral is not used, it is not possible for the user to download his application on to the processor.

Fig.3. Hardware Architecture of the proposed system
The detailed specification for the system developed in Quartus is given in the table below.

<table>
<thead>
<tr>
<th>Development Kit</th>
<th>Altera DE2-115 Cyclone IV FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Family</td>
<td>32-bit RISC Nios II F processor</td>
</tr>
<tr>
<td>Operating Frequency</td>
<td>200 MHz</td>
</tr>
<tr>
<td>Memory Subsystem</td>
<td>128 MB SDRAM, 8M Flash</td>
</tr>
<tr>
<td>Video Pipeline</td>
<td>SDRAM used as buffer</td>
</tr>
<tr>
<td>Video Operating Frequency</td>
<td>50 MHz</td>
</tr>
<tr>
<td>SDRAM Operating Frequency</td>
<td>50 MHz (-60 degree phase shift)</td>
</tr>
<tr>
<td>GPIO Interfaces</td>
<td>LED, LCD, Seven Segment, Push buttons and Switches</td>
</tr>
</tbody>
</table>

Table.1. System specifications

This covers the hardware design aspect of this project. From this step, the project requires creation of a Board Support Package, development a port of RTOS kernel and finally the development of an application level firmware for complete functioning.

B. Software Implementation

Software development process for this project takes place in two steps. Firstly, a board support package for uC/OS-II is to be developed in order to port the operating system on the development kit. After this is done, an application for creating and running uC/OS-II needs to be written in C programming language. The process of creating a board support package is explained in the section below.

Board Support Package

The source for Micrium’s uC/OS-II is acquired from the Downloads section of Micrium homepage. This version if free for academic purposes and is not supposed to be distributed for any commercial purposes. This source code is independent of the processor and is native to the application written and hence is called the processor independent or application specific code. This source consists of the following files (Micrium uC/OS-II Source Code)

<table>
<thead>
<tr>
<th>Configuration Files of uC/OS-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. os_cfg_r.h</td>
</tr>
<tr>
<td>2. os_core.c</td>
</tr>
<tr>
<td>3. os_dbg_r.c</td>
</tr>
<tr>
<td>4. os_flag.c</td>
</tr>
<tr>
<td>5. os_mbox.c</td>
</tr>
<tr>
<td>6. os_mem.c</td>
</tr>
<tr>
<td>7. os_mutex.c</td>
</tr>
<tr>
<td>8. os_q.c</td>
</tr>
<tr>
<td>9. os_sem.c</td>
</tr>
<tr>
<td>10. os_task.c</td>
</tr>
<tr>
<td>11. os_time.c</td>
</tr>
<tr>
<td>12. os_tmr.c</td>
</tr>
<tr>
<td>13. ucos_ii.c</td>
</tr>
<tr>
<td>14. ucos_ii.h</td>
</tr>
</tbody>
</table>

System initialization is done by alt_sys_init.c which is used in the Makefile for the target in the Nios II application. This file should initialize the timer required for uC/OS-II application to function. system.h file is the header file which contains the mappings to the peripherals that are used in the Nios II application. These mappings can be manually done or can be done with the help of Nios II Software Build Tools. The processor dependent files are os_cpu.h, os_cpu.c.c and os_cpu_a.asm. These files contain the mapping of the processor to that of
the operating system. Task management functions like creation and deletion of task from memory are listed here. Also, the tick rate of the RTOS is decided in this file. A function OSTimeTick() is called every tick and it updates the uC/OS-II timers by signaling the timer task. OS_CPU_SysTickInit() is used to initialize the system tick which is the uC/OS-II time tick.

Porting the RTOS µC/OS-II requires the creation of file os_cpu_a.s which is the ASM file for the processor. It contains the definition of the following five functions. These functions are needed because you normally cannot save/restore registers from C functions. These functions are:

1. OS_CPU_SR_Save()
2. OS_CPU_SR_Restore()
3. OSStartHighRdy()
4. OSCtxSw()
5. OSGlobalSw()

The Board Support Package file settings.bsp is created by the Nios II Software Build Tools. It is a XML file that contains the map of the hardware peripheral and its definition in the system.h file. This is important because this file basically acts as an intermediary between the application code and the target platform on which the operating system is ported.

The relationship between the modules of the board support package, the actual application and the uC/OS-II port code is given in the figure below.

---

**Application Development**

Four tasks have been created and have been assigned different functionalities. The context switches between the tasks have been demonstrated by the use of printf statements which display the name and number of task on the Nios II console. A push button has been assigned an interrupt number and when pressed, it triggers an interrupt which lights up a LED and displays ‘Interrupt’ on the LCD. Inter-task communication has been demonstrated with the use of semaphores and mailboxes. A task waiting for a mailbox from another task suspends its operation till it gets a mailbox signal from the posting task.
To generate the task, following stack size and specifications have been used.

1. Task Priority
2. Task Stack Size - 2048 bytes. This is basically an array of type OS_STK with the size of 2048 elements.
3. Number of tasks. This is specifically important because the total size of all tasks should not exceed the stack storage space on the processors on chip memory.

Every task in an uC/OS-II application is an infinite loop. The behavior of the task is written inside an infinite loop. Task 1 has been created to switch the RED LED’s ON and OFF after a certain time interval. Task 2 displays ‘uC OS ||’ on the seven segment display. Task 3 prints ‘uC/OS – II OS’ on the 16x2 LCD display. Task 4 has been assigned to activate the GREEN LED’s available on the development kit. The aim of this project is to demonstrate the task switching process carried out by the real time operating system. A delay of 3 seconds has been given in each task to make the process visible to the naked eye.

One button out of the four buttons present on the development kit generates an interrupt when pressed. This in turn suspends the operation of the running task and services the interrupt. Generally, sleeping in the interrupt context is not recommended but in order to make this ISR visible, a small delay has been given. This process is time critical and of utmost importance in the functioning of uC/OS-II because the time to service an interrupt should be minimal in case of a real time operating system which is demonstrated.

V. Observations

The aforementioned system was successfully compiled, verified and tested for functionality. To test the system, application level code was written in C. This code writes user defined values into internal registers of the peripherals and activates the desired sequence. Altera’s Nios II IDE was used to write code for the above generated system file. The observations after compiling the system in Quartus are as given below. This is a summary report of the compiled application.

<table>
<thead>
<tr>
<th>Total logic elements</th>
<th>5,329 / 114,480 (5 %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total combinational functions</td>
<td>4,492 / 114,480 (4 %)</td>
</tr>
<tr>
<td>Dedicated logic registers</td>
<td>3,187 / 114,480 (3 %)</td>
</tr>
<tr>
<td>Total registers</td>
<td>3354</td>
</tr>
<tr>
<td>Total pins</td>
<td>293 / 529 (55 %)</td>
</tr>
<tr>
<td>Total memory bits</td>
<td>1,772,224 / 3,981,312 (45 %)</td>
</tr>
<tr>
<td>Total PLLs</td>
<td>1 / 4 (25 %)</td>
</tr>
</tbody>
</table>

Table 2. Compilation Report

After the above system was built, task context switches were observed. Justifying the operation of the RTOS, it was observed that the RTOS takes over scheduling of tasks instead of the users. The tasks were run according to their priority numbers. These tasks even though were defined separately, would run according to one task at a time according to their priority. In case of an interrupt, these tasks would suspend their operation and the Interrupt Subroutine would take over. Following is the sequence of operations that was observed in the first run of the application program that was written.

**Task 1.** Red LEDs would go on and at the end of the task, they would go off. The task executes instantaneously and hence some delay was
added to make the task sleep and thus make the next higher priority task ready to run. This demonstrated the process of context switching.

**Task 2.** uC/OS-II was displayed on the Seven Segment Display one at a time. Sleep operation was called in the middle of this task to make the next higher priority task to run. It was observed that if Task 1 was ready to run having completed its sleep time before the completion of Task 2, it would pre-empt the running task and continue its execution. If the timer was not complete, Task 3 would change its state from ready to run to running.

**Task 3.** uC/OS-II was displayed on the LCD after initializing it and similar observations like that of Task 2 were made when Task 3 was run. If any of the higher priority tasks changed their state from blocked to ready to run, they would be scheduled. If not, after Task 3 was suspended, either because of completion or because of change of state, Task 4 would run.

**Task 4.** Green LEDs went on once this task began executing. This being the lowest priority task, became running only when all other tasks were blocked by the sleep() call or because of pre-emption.

Inter-task communication takes place with the help of semaphores and mailboxes. Semaphore is used to send an integer value from one Task 2 to a Task 1. If the value of the integer remains unchanged, then ‘Semaphore Received’ message is printed on the LCD screen and the console. This was verified with the help of a global integer and successfully demonstrated. Mailboxes work in a similar way but instead of integers, they communicate using pointers. This was verified by using an integer pointer and a similar message ‘Mailbox received’ was displayed on the LCD on successful inter-task communication.

**VI. CONCLUSION AND FUTURE WORK**

The aim of this paper is to study the steps in designing a raw processor from logic elements, developing a BSP for it, porting a real time operating system on it and finally demonstrating task context switches, interrupts and inter-task communication using semaphores and mailboxes on Altera’s DE2-115 development kit. The hardware development stage was completed successfully by implementing a Nios II processor and the system mentioned above. The Board Support Package was developed for the system which was built in the hardware development stage and uC/OS-II was successfully ported on to the microprocessor. An application to demonstrate the use of RTOS was written and demonstrated which explained the concepts of context switching, interrupt handlers and inter-task communication using semaphores and mailboxes. This custom system can be developed to run any single large scale application depending on the peripherals used because of the reconfigurable architecture of FPGA.

**VII. References**