University of Cincinnati

Date: 3/26/2013

I, Eric B Swegert, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:
RTOS Tutorials for a Heterogeneous Class of Senior and Beginning Graduate Students

Student’s name: Eric B Swegert

This work and its defense approved by:

Committee chair: Carla Purdy, Ph.D.
Committee member: Yiming Hu, Ph.D.
Committee member: George Purdy, Ph.D.
RTOS Tutorials for a Heterogeneous Class of Senior and Beginning Graduate Students

A Thesis submitted to the
Graduate School
Of
The University of Cincinnati
in partial fulfillment of the
requirements for the degree of

Master of Science

in the
Department of Electrical and Computer Engineering
Of the
College of Engineering
May 2013

by
Eric Swegert
Bachelor’s of Science in Computer Engineering, Miami University 2010

Committee Chair: Dr. Carla Purdy
ABSTRACT

Students with an interest in embedded systems come from a variety of backgrounds: EE, COMPE, CS, Telecommunications, etc. The students may be on-campus students or distance learning students. In particular, they have different levels of expertise in hardware and software and they will work or do research in a variety of settings, with many different system environments. For an embedded systems course to serve all the aforementioned groups, the course material has to follow an approach from hardware to software illustrating general concepts as well as practical experience. The course should be self-contained and flexible enough to support distance learning courses as well. However, there do not exist teaching materials at the appropriate levels and sufficiently detailed to properly train beginning students in the many areas they need to master. This thesis attempts to remedy that lack in the area of real time operating systems (RTOS).

The proposed RTOS tutorials on an embedded, FPGA-based development platform will start with specifying the necessary hardware system then progress to the RTOS application. The necessary operating system (OS) and real time operating system (RTOS) concepts will be introduced to facilitate successful RTOS application development. The RTOS tutorials will be built upon the Altera Nios II soft-core processor and the Micrium MicroC/OS II RTOS utilizing RTOS concepts that can be applied to different RTOS’s and embedded platforms.
# TABLE OF CONTENTS

1. **INTRODUCTION** ........................................................................................................1
   1.1. Motivation ........................................................................................................2
   1.2. Thesis Goals ......................................................................................................2
   1.3. Outline .............................................................................................................2

2. **BACKGROUND** .....................................................................................................4
   2.1. Operating Systems ............................................................................................4
      2.1.1. Process ......................................................................................................6
      2.1.2. Process Scheduling & Multitasking ..............................................................7
      2.1.3. Synchronization ........................................................................................7
      2.1.4. Deadlock ....................................................................................................9
      2.1.5. Interrupts ...................................................................................................9
   2.2. Real-Time Operating Systems ........................................................................10
      2.2.1. Hard vs. Soft Real-Time Systems .................................................................10
      2.2.2. RTOS Implementation ...............................................................................11
   2.3. Micrium MicroC/OS-II ...................................................................................12
      2.3.1. Task .........................................................................................................12
      2.3.2. Preemptive, Priority-Based Scheduler .......................................................13
      2.3.3. Semaphores ............................................................................................13
   2.4. Altera Nios II Processor .................................................................................14
   2.5. Tool Chain and Design Flow ..........................................................................15
   2.6. Altera DE1 Development Board ....................................................................18
3. TUTORIALS........................................................................................................20
   3.1. Quartus II/Qsys Tutorial..............................................................................20
       3.1.1. Quartus II..........................................................................................20
       3.1.2. Qsys..................................................................................................22
   3.2. Nios II SBT Tutorial..................................................................................23

4. LABS..................................................................................................................24
   4.1. Producer-Consumer...................................................................................24
       4.1.1. Goal..................................................................................................24
       4.1.2. Prerequisites......................................................................................25
       4.1.3. To Do..............................................................................................25
       4.1.4. Conclusion.........................................................................................27
   4.2. Readers-Writers.........................................................................................27
       4.2.1. Goal..................................................................................................27
       4.2.2. Prerequisites......................................................................................28
       4.2.3. To Do..............................................................................................29
       4.2.4. Conclusion.........................................................................................31
   4.3. Dining Philosophers...................................................................................31
       4.3.1. Goal..................................................................................................32
       4.3.2. Prerequisites......................................................................................32
       4.3.3. To Do..............................................................................................33
       4.3.4. Conclusion.........................................................................................34
4.4. Automated External Defibrillator (AED) Controller……………………………………35
  4.4.1. Goal…………………………………………………………………………………35
  4.4.2. Prerequisites……………………………………………………………………...35
  4.4.3. To Do…………………………………………………………………………….36
  4.4.4. Conclusion……………………………………………………………………….37
4.5. Sleeping Barber…………………………………………………………………………37
  4.5.1. Goal………………………………………………………………………………37
  4.5.2. Prerequisites……………………………………………………………………...38
  4.5.3. To Do…………………………………………………………………………….38
  4.5.4. Conclusion……………………………………………………………………….39
5. RESULTS…………………………………………………………………………………...40
  5.1. Lab Setup……………………………………………………………………………..40
    5.1.1. Development Environment……………………………………………………40
    5.1.2. Development Board…………………………………………………………...41
    5.1.3. Working Lab Implementation……………………………………………….41
  5.2. Quartus II/Qsys Tutorial………………………………………………………………...41
    5.2.1. Qsys……………………………………………………………………………41
    5.2.2. Quartus II………………………………………………………………………...41
  5.3. Nios II SBT Tutorial…………………………………………………………………..42
  5.4. Labs and OS Concepts…………………………………………………………………..42
6. CONCLUSIONS AND FUTURE WORK ................................................................. 43

6.1. Conclusions .................................................................................................. 43

6.2. Future Work .................................................................................................. 44

REFERENCES ........................................................................................................ 45

APPENDIX A: Quartus II/Qsys Tutorial .............................................................. 47

APPENDIX B: Nios II SBT Tutorial ..................................................................... 66

APPENDIX C: MicroC OS-II Labs Source Code .................................................. 69

| C.1: Producer-Consumer Source Code ............................................................ 69 |
| C.1.1: producer_consumer.c ........................................................................ 69 |
| C.1.2: producer_consumer.h ....................................................................... 77 |

| C.2: Readers-Writers Source Code ................................................................. 79 |
| C.2.1: reader_writer.c ................................................................................ 79 |
| C.2.2: reader_writer.h .............................................................................. 82 |

| C.3: Sleeping Barber Source Code ................................................................. 83 |
| C.3.1: sleeping_barber.c ............................................................................ 83 |
| C.3.2: sleeping_barber.h ........................................................................... 87 |
LIST OF FIGURES

Figure 2.1: Abstract computer system components [1] ..............................................................5
Figure 2.2: Process state diagram [1] ........................................................................................6
Figure 2.3: Example of a Nios II Processor System [3] ...............................................................14
Figure 2.4: HAL-based system layers [8] ..................................................................................15
Figure 2.5: The design flow .........................................................................................................17
Figure 2.6: DE1 development board [29] ..................................................................................19
Figure 4.1: Producer-consumer lab ..........................................................................................25
Figure 4.2: Readers-writers lab ..................................................................................................29
Figure 4.3.1: Classic dining philosophers [1] ..........................................................................31
Figure 4.3.2: Cardinal dining philosophers ...............................................................................33
Figure 4.4: AED controller lab ......................................................................................................36
Figure 4.5: Sleeping barber lab ..................................................................................................38
Figure A.1: New project wizard [page 1 of 5] .........................................................................47
Figure A.2: New project wizard [page 3 of 5] .........................................................................48
Figure A.3: Qsys project settings ...............................................................................................49
Figure A.4: SDRAM controller parameter settings .....................................................................50
Figure A.5: ALTPLL parameter settings [page 1 of 7] ..............................................................51
Figure A.6: ALTPLL parameter settings [page 2 of 7] ..............................................................52
Figure A.7: ALTPLL output clocks [page 4 of 7] ......................................................................53
Figure A.8: ALTPLL output clocks [page 5 of 7] ......................................................................54
Figure A.9: ALTPLL output clocks [page 6 of 7] ......................................................................55
Figure A.10: Clock bridge parameters ......................................................................................56
Figure A.11: PIO settings (HEX0) ............................................................................................57
Figure A.12: PIO settings (green_LEDs) ...................................................................................58
Figure A.13: Nios II processor settings.................................................................61
Figure A.14: de1_tutorial.v...............................................................................63
Figure A.15: toggle.v.........................................................................................64
INTRODUCTION

1.1 Motivation

Real-Time Operating System (RTOS) application development for an embedded systems platform requires knowledge of both hardware and software concepts. Modern RTOS’s abstract the hardware, allowing developers from various backgrounds to concentrate in their area of specialization [11]. This poses a barrier to an approach aimed at traversing the embedded computer system hierarchy. Students entering a class covering such a topic will bring with them varying aspects of the prerequisite knowledge and therefore each topic should be given equal coverage and lab material should be provided to give each student practical experience.

Altera provides a tool chain of easy to use tools that allows for a smooth design flow from hardware specification to RTOS application download [7]. The tutorials provided by Altera provide an adequate understanding of how to use the tools but fall short in the area of hardware specification. Unless a beginning student is following along with the same Altera development board the tutorial was written for, it is unlikely a valid Nios II system will result. And rightly so, a Nios II system is a highly configurable product that is meant to be custom fitted to the application.

We propose a compendium of self-contained tutorials and subsequent labs that will allow the beginning student to successfully deploy an RTOS application on an embedded computing platform. The tutorials will enable all the students to specify and generate a hardware system on which RTOS applications can be executed. The labs will allow the students to utilize the hardware system created from the tutorials to develop and deploy RTOS applications. The
RTOS applications created from the labs will demonstrate fundamental RTOS and OS concepts while giving the students practical experience.

These labs can be used as either group or individual projects. For a group, the typical make-up should be 2-4 people with a mix of COMPE and EE students and a mix of senior and graduate students. The members of which should be chosen, if possible, to span the necessary skill set e.g., programming, HDL, OS, etc. For the individual, the labs are designed to cover the breadth of knowledge required and are intended to be developed into a web-based lab for individual distance learning in the future.

1.2. Thesis Goals

This thesis sets out to provide a collection of tutorials and labs targeted at a heterogeneous class of senior and beginning graduate students to allow them to develop Micrium MicroC/OS-II [5] RTOS applications on the Nios II processor [3].

1.3. Outline

The thesis is organized as follows:

Chapter 2 provides an overview of the necessary OS concepts required to introduce RTOS’s. It then covers the specialized OS concepts and design constraints that comprise an RTOS. The Micrium MicroC/OS II RTOS implementation of an RTOS is then covered, as this will be the RTOS used in the tutorials and labs. The embedded platform, Altera’s DE1 development board, is then introduced followed by a brief synopsis of Altera’s Nios II processor. Chapter 2 is brought to a close with an overview of the design flow and tool chain used in the tutorials and labs.
Chapter 3 presents the tutorials, Quartus II/Qsys and Nios II SBT (Software Build Tools). The Qsys part of the Quartus II/Qsys tutorial allows the students to specify and generate a Nios II system that will be used in the labs to run RTOS applications. The Quartus II part of the Quartus II/Qsys tutorial allows the students to instantiate the top level of the Nios II system as an HDL (Verilog or VHDL) module, compile that Nios II system, and then program the DE1 development board with the system netlist using Quartus II Programmer. The Nios II SBT tutorial allows the students to compile and download a MicroC/OS-II RTOS application and BSP to the DE1 development board.

Chapter 4 presents the labs which build upon the Nios II SBT knowledge and hardware system obtained from the tutorials. The labs include: producer-consumer, readers-writers, dining philosophers, AED controller, and sleeping barber.

Chapter 5 presents the results of the application of the tutorials and labs in a heterogeneous class of senior and beginning graduate students.

Chapter 6 presents conclusions and proposed future work.
2. BACKGROUND

In this chapter we briefly cover the basic operating system and real-time operating system concepts necessary to develop and reason about the MicroC OS-II programs in the tutorials and labs. The MicroC OS-II implementation specifics are also covered and correlated with the operating system and real-time operating system sections. The Nios II processor, Altera tool chain and design flow, and Altera DE1 development board are also briefly discussed.

Two key terms to understand in this section are microcontroller and embedded system. A microcontroller is a lightweight computer system, on a single chip, containing a processor, memory, and I/O peripherals [27]. An embedded system is a computer system with a specific purpose and it is often embedded within a larger system to provide an application specific solution [26]. A microcontroller is often part of an embedded system and the two terms are usually used interchangeably.

2.1. Operating Systems

An operating system abstracts the underlying hardware of a computer system, creating a safe environment for programs to execute in. The computer system is comprised of the hardware, the operating system, the application programs, and the users [1]. A graphical representation of this can be seen in Figure 2.1 [1]. The operating system enables the separation of these components by enforcing read/write permissions, controlling I/O, controlling processing resources, and acting as the application’s liaison to the hardware [2]. This separation allows the “untrusted” application programs to execute without direct (and possibly damaging) access to the computer system’s hardware, and allows for sharing of resources.
The operating system can be defined as one program, which is always running, conventionally named the kernel [1]. The remaining programs that accompany the kernel (while remaining separate) can be classified as application programs and system programs. System programs are ancillary programs to the kernel; they provide services to the application programs and users. The kernel consists of the core operating system functions that execute with elevated privileges with regards to the application and system programs. The kernel provides services to the application programs to allow for safe and efficient use of the computer system. These services are accessed through the use of system calls and defined by the operating system’s application programming interface. The abstraction that the application programming interface provides allows applications to be developed independent of the hardware.

Figure 2.1: Abstract computer system components [1]
2.1.1. Process

A fundamental service that an operating system provides is the time-sharing of a computer system. A time-shared system can load multiple programs into memory and execute them concurrently, thus facilitating tighter control and compartmentalization; the concept of a process allows this to come to fruition [1]. A process is a specific, executing instance of an application or system program which—generally speaking—includes the process’s registers, call stack, and memory space. Any given program may have multiple processes executing its code; the context of each process is what makes them unique. Every process also has associated with it a state of execution: New, the process has just been spawned; Running, the process’s associated program is executing; Waiting, the process is blocking for I/O or an interrupt; Ready, the process is ready to be executed; and Terminated, the process is done executing [1]. A visualization of these states can be seen in Figure 2.2. When a process is swapped out from the running state and enters either the ready or waiting states, its context and state information are stored in an operating system data structure known as the process control block (PCB) [1]. A process control block can also be called a task control block, as will become evident later in this chapter when Micrium MicroC/OS-II RTOS is introduced.

Figure 2.2: Process state diagram [1]
2.1.2. Process Scheduling & Multitasking

A basic service that the operating system provides is multitasking. Multitasking allows seemingly concurrent process execution (at least on a single core embedded system) and it accomplishes this through process switching. This task is handled by the operating system’s process scheduler [2]. The process scheduler decides which process will be allowed to execute at certain junctures in system execution where the operating system has been given control: usually after an interrupt or I/O request. How the process scheduler makes its decisions is based upon the specific scheduling algorithm it implements. The scheduling algorithm allows the characteristics of an operating system’s process management to be fine-tuned to specific applications. A complete chronicle of all available scheduling algorithms is outside the scope of this operating system introduction; however, the specific algorithm implemented by Micrium MicroC/OS-II RTOS will be introduced later in this chapter.

2.1.3. Synchronization

Synchronization mechanisms provide the means by which the shared resources of a computer system can be safely utilized by multiple executing processes. In this section the discussion of synchronization will remain general—abstracting the different types of synchronization mechanisms with the simple lock. A detailed discussion of the specific synchronization mechanisms necessary to complete the MicroC OS-II labs will be covered in section 2.3.3. The lock—as we are abstracting it at this point—provides mutual exclusion. Mutual exclusion ensures that only one process can access a shared resource at a time and is necessary to protect a process’s critical section in order ensure data consistency. The critical section is a segment of code in which the process is accessing shared data structures [1]. The critical section requires
mutually exclusive access which constrains the access of shared data to one process. Before entering its critical section, a process must acquire the lock corresponding to the shared data structure that it wishes to read from or write to. Once the process is finished modifying the resource, it releases the lock. Without locks to synchronize access to shared resources, the data would become corrupted and process execution would become unpredictable.
2.1.4. Deadlock

Deadlock is a situation where two or more processes, sharing a resource, each preclude the other(s) from accessing said shared resource, thus resulting in system lockup [2]. More specifically, deadlock occurs when two or more processes share a resource protected by a synchronization mechanism, such as a lock, that ensures each process has mutually exclusive access to the shared resource; and the process holding the lock enters a state where it will never release the lock. Another deadlock related issue is starvation, which causes a process to wait indefinitely while other processes are allowed to execute [1]. Processes must be written in such a way as to prevent deadlock, which leads to system inoperability. Deadlock can embrace a system if the following conditions occur simultaneously: mutual exclusion, resource holding, no preemption, and circular waiting [1]. Mutual exclusion is introduced in section 2.1.3 above. Resource holding occurs when a process is holding one resource but requires another resource held by another process; this problem will be illustrated in the dining philosophers lab. No preemption, pertains to resources held by certain processes, it means that resources can only be voluntarily released by a process [1]. Circular waiting is a concept, related to resource holding, wherein there is group of processes each requiring a resource, one of the others holds. All four of the above conditions must exist for deadlock to occur [1].

2.1.5. Interrupts

An interrupt is a signal to the operating system that service is required or that a context switch is necessary. Interrupts allow a computer system to interact with the outside world without the wasteful use of polling loops, which check for a specific condition, looping until it occurs. At the basic level, interrupts can be classified into hardware and software interrupts. Hardware
interrupts indicate to the operating system that some asynchronous event has occurred such as device I/O completion [1]. Software interrupts indicate to the operating systems that a synchronous event has occurred such as a process invoking a system call [1]. In either case, the operating system must save the process state and service the interrupt.

2.2. Real-Time Operating Systems

Real-time operating systems control real-time systems. The term “real-time” has entered the vernacular and is understood by most people to mean “at once” or “instantaneously” [10]. However—in a computing context—a real-time system is a computer system that demands “correct” results delivered by a specified deadline [1]. Real-time systems encompass a wide range of applications but most of them are embedded systems [4]. In summary, a real-time operating system is a specialized subclass of operating system with a different set of constraints that allow it to successfully control and react to a real-time system.

2.2.1. Hard vs. Soft Real-Time Systems

Real-time systems can be classified into hard and soft real-time systems. A soft real-time system is degraded but not destroyed by missing a deadline; a hard real-time system on the other hand may be destroyed by missing a deadline [10]. A hard real-time system must deterministically meet every deadline. An example of a soft real-time system is a video game where the game play lags with missed deadlines but continues to function; an example of a hard real-time system is a weapons system where missing a deadline could mean life or death for a fighter pilot [10]. A majority of real-time systems have both soft and hard requirements [4].
2.2.3. RTOS Implementation

On a basic level, a real-time operating system will have the following attributes: preemptive, priority-based scheduling; preemptive kernel; and minimized latency [1]. A preemptive, priority-based scheduling system is a system in which a higher priority process preempts (interrupts) a lower priority, executing process [10]. Priorities are assigned to processes based upon their priority level relative to other processes; the actual priority enumeration scheme is arbitrarily chosen to either correlate higher priorities to higher values or higher priorities to lower values. A preemptive kernel augments a preemptive, priority-based scheduling system by allowing the highest priority process to always run even if the kernel is processing a system call for a lower priority process [1]. This can be done by inserting preemption points within long-duration system calls to allow the scheduler to assess possible scheduling changes or by using synchronization mechanisms to protect kernel data structures [1]. The preemptive kernel, priority-based scheduling system described above is poised to respond to any software or hardware event in real-time, and the metric that describes the amount of time that elapses between when an event occurs and when it is serviced is called event latency [1]. It is highly desirable for any RTOS implementation to minimize this metric by minimizing both interrupt latency and dispatch latency. Interrupt latency refers to the time period from interrupt arrival to when the system begins to service the interrupt and dispatch latency refers to the time period required for the scheduler to stop one process and schedule another [1]. An RTOS with these characteristics is well equipped to control almost any real-time system.
2.3. Micrium MicroC/OS-II

“Micrium’s MicroC OS/II is a completely portable, ROMable, scalable, preemptive, real-time, multitasking kernel that is easy to learn and use” [4]. The MicroC OS/II kernel has been ported to Altera’s Nios II processor architecture and the full ANSI C source code is available with the Nios II EDS (Embedded Design Suite) as an evaluation edition free for academic use [5]. Students can study the source code implementation of MicroC OS/II allowing them a better understanding of how to use the RTOS services in their applications. Design examples are also included, allowing students to reference working MicroC OS/II applications using them as starting points to develop their own applications [7]. MicroC OS/II coupled with Altera’s Nios II EDS is a powerful teaching tool for RTOS on an embedded system platform.

2.3.1. Task

The MicroC OS/II RTOS does not have the concept of a process as described in section 2.1.1. MicroC OS/II has tasks, which are defined in [4] as: “a simple program that thinks it has the CPU all to itself”. A task has a unique priority, its own context registers, and its own stack [4]. A task will normally have a cyclic structure running in an infinite loop which can be in the following states: dormant, ready, running, waiting, or ISR (Interrupt Service Routine) [4]. A dormant task is in memory but has not been made ready for execution; a ready task is ready for execution but has a priority less than the currently executing task; a running task is executing; a waiting task is waiting for a specific event to occur; an ISR task has been interrupted and the operating system is servicing that interrupt [4].
2.3.2. Preemptive, Priority-Based Scheduler

The MicroC OS-II task scheduler is a preemptive priority-based scheduler. This means that the highest priority ready task will always be allowed to execute [4]. When a higher priority task becomes ready to execute, a task switch occurs; the scheduler preempts the execution of the current task, performing a task switch [4]. A task switch pushes the current task’s CPU registers (context) onto the current task’s stack, restores the preemitting task’s CPU registers, and then gives the preemitting task control of the CPU [4]. Starvation of lower priority tasks will occur unless the highest priority task makes a system call to give up execution time. As a general rule, within the task loop, each task should do work, make system calls, etc., then give up the CPU with the appropriate time delay system call. In general, a preemptive, priority-base scheduler provides the granularity of control and short response time that a real-time system demands.

2.3.3. Semaphores

MicroC OS-II provides a number of synchronization services, including: semaphores, event flags, message mailboxes, and message queues [4]. However, for the context of this paper—and the tutorials and labs introduced here—the semaphore is the only synchronization mechanism that will be needed. The concept of a semaphore is similar to the lock abstraction used in section 2.1.3. MicroC OS-II provides semaphores that can be configured as binary semaphores (providing mutual exclusion) or counting semaphores (controlling access to a limited shared resource) [4].
2.4. Altera Nios II Processor

The Nios II processor system is equivalent to a microcontroller (embedded system) that includes a processor, peripherals, and memory on a single chip [3]. Contrary to a typical microcontroller, the Nios II processor is a configurable soft IP core which can be modified for a specific application [3]. A basic Nios II system consists of the Nios II processor, on-chip peripherals, on-chip memory, and interfaces to off-chip memory, all implemented on a single Altera FPGA [3]. An example of such a Nios II system can be seen in Figure 2.3.

Figure 2.3: Example of a Nios II Processor System [3]
Similar to most microcontroller families, all Nios II systems use a consistent instruction set and programming model [3]. The Altera provided Hardware Abstraction Layer (HAL) is part of the Nios II programming model. The HAL provides the following services: a C standard library, device drivers, interrupt handling, alarm facilities, system initialization, and device initialization [8]. The standard interface to HAL services allows application programs to be written independent of the underlying hardware specific drivers. A visualization of the HAL can be seen in Figure 2.4.

![Figure 2.4: HAL-based system layers [8]](image)

### 2.5. Tool Chain and Design Flow

For the purposes of this paper, a tool chain is a set of tools used to build a project, and a design flow is the flow of design files and artifacts between each tool in the chain. That said, these elements are crucial to the successful implementation of an embedded systems RTOS lab. Altera’s Embedded Design Suite (EDS) supplies the necessary tool chain for a smooth design flow from Nios II system specification to a deployed MicroC OS-II application [7]. To deploy a
MicroC OS-II application on an embedded development board, the student must follow a bottom up approach starting with Altera’s Qsys system integration tool where the hardware system is assembled from Altera’s IP modules and Avalon Interconnect to create a Nios II system [6]. Qsys then passes off the parameterized IP (Intellectual Property) modules and HDL (hardware description language) Nios II instantiations to Quartus II. Quartus II compiles the HDL, yielding a SRAM Object File (SOF) which is used by the Quartus II programmer to configure the FPGA on the development board [6]. The tool chain terminates with the Nios II SBT for Eclipse, which uses information about the chip generated by Qsys (.sopcinfo file) to configure, build, and download a MicroC OS/II application and Board Support Package (BSP) [13]. The design flow is illustrated in Figure 2.5 below. The design flow and tool chain described above can be daunting for the beginning student, but with the proper documentation the student can complete a manageable process that delivers successful RTOS application development and deployment.
Figure 2.5: The design flow
2.6. Altera DE1 Development Board

The hardware platform for an RTOS embedded systems course needs to be compact, durable, and affordable. All of these qualities can be found in Altera’s DE1 development board. With a 6” x 6” footprint, standoff mounted rubber feet, standoff mounted plexiglas cover, and an academic reduced price of $125 (April 2013), the DE1 development board encompasses all of these qualities [12]. The DE1 development board is also conveniently powered by USB and contains a built-in USB-Blaster which allows it to be deployed in a desktop environment with ease, requiring only a USB cable connection for programming and power. Aptly equipped with Altera’s Cyclone II 2C20 Field Programmable Gate Array (FPGA), the DE1 development board is compatible with Altera’s Nios II soft processor [12]. The DE1 development board is capable of persistent configuration with Altera’s EPCS4 4-Mbit serial configuration device with which stores SRAM configuration data in non-volatile flash memory for configuration on power up [12]. With all of the above stated capabilities, the DE1 development board can support an RTOS embedded systems course. The DE1 development board—with important features labeled—can be seen in Figure 2.6 below.
During the development board selection process, multiple Altera and Xilinx boards were examined. At the time, the DE1 development board had the best availability, pricing, and features to suit our needs. Altera’s Nios II port of the MicroC OS-II RTOS was a deciding factor as well. Other affordable Altera boards, compatible with the tutorials and labs herein, include the DE0 ($79) and the DE0-Nano ($59). Affordable Xilinx boards were also available but Xilinx did not offer anything comparable to Altera’s EDS especially when [4] is considered as a complementary text. It is important to note that the OS concepts and principles introduced in this paper carry over to Xilinx boards and can be utilized after following Xilinx tutorials for board setup. Whether Altera or Xilinx, manufacturers can only list specifications and it is
difficult to tell if a design will fit on a device, therefore it is recommended that the design be tested on the board before any sizable orders are placed. This is especially prudent if the board has been selected with minimum specifications. However, it is always good practice to allow for expansion in development boards so future projects are not left constrained to fit minimal hardware.

3. TUTORIALS

In this chapter we cover two tutorials, Quartus II/Qsys and Nios II SBT, aimed at taking the student from Nios II system specification and generation to Nios II application download. The Quartus II/Qsys tutorial is a specific adaptation of Altera documents: [14], [15], and [16] tailored to Altera’s DE1 development board. The Nios II SBT (Software Build Tools) tutorial, based on [13], will get the students through the final step of downloading their Nios II BSP (Board Support Package) and application onto the DE1 development board.

3.1. Quartus II/Qsys Tutorial

The Quartus II/Qsys tutorial is the first step to a functional Nios II system. The tutorial can be found in Appendix A. This section will leave the step-by-step specifics to the appendix and cover general principles on Quartus II and Qsys that will make the tutorial easier to follow.

3.1.1. Quartus II

The Altera Quartus II design software is a multiplatform design environment that can be adapted to specific needs, for use during all phases of FPGA design [18]. For the purposes of this paper, we will be using the Quartus II IDE to create a project, specify FPGA pin assignments (specific to DE1 development board), and HDL design entry and compilation. The Quartus II compiler
targets the specific FPGA (associated with the Quartus II project) and generates a .sof (SRAM Object File). The Quartus II programmer downloads the .sof onto the FPGA, leaving it configured and running.

The Quartus II pin assignments are crucial to a functioning design. A pin assignment associates an internal I/O port with an external pin on the FPGA package. This reconfigurable assignment is made possible by the programmable FPGA interconnect fabric, the details of which are beyond the scope of this paper and can be explored in [25]. The pin assignments can be found in the Quartus II Pin Planner, which allows for viewing and editing of pin assignments. Since we are using a development board with a static layout, the pin assignments can be imported into Quartus II from the Altera provided .csv file. Once imported, each pin has an associated name which must be matched by any HDL port connected to it.

The Quartus II compilation flow generates a .sof, which is used to configure the SRAM based FPGA. The Quartus II programmer downloads the .sof onto the FPGA, which configures it for operation. The Quartus II programmer has many settings and capabilities outside the scope of this paper; the default settings and basic program feature will suffice for our purposes. One caveat to using the Quartus II programmer is ensuring that the USB-Blaster is selected properly and that the USB-Blaster drivers are installed correctly. For most Windows installations, the drivers are bundled with the Quartus II installation files and installed by Windows plug and play manager when the USB-Blaster is plugged in. As for selecting the USB-Blaster, following the tutorial in Appendix A will ensure it is selected properly.
3.1.2. Qsys

Qsys is a system generation tool which captures system-level hardware designs at a high level, automating the task of defining and integrating custom HDL components: IP Cores, verification IP, and other custom design modules [6]. To stitch together the Nios II system, Qsys automatically generates Qsys interconnect logic [6]. The Qsys interconnect is a high-bandwidth framework for connecting components that implement standard Avalon interfaces [19].

The Qsys interconnect connects the following interface types: memory-mapped, streaming, interrupts, clocks, resets, and conduits [6]. Qsys interconnect interfaces may be exported in order to interface with signals external to the Nios II system. The Nios II system that is created using the Quartus II/Qsys tutorial in Appendix A will make use of all Qsys interconnect types except the streaming interface. The memory-mapped interface implements a partial-crossbar interconnect structure which provides a duplex communication link between master and slaves [6]. Transactions between masters and slaves are encapsulated and transmitted in packets [19]. Masters send command packets to slaves over a command network and slaves send response packets over a response network [19]. The conduit interface is for interfaces that do not fall under any of the other Qsys interconnect interface types; it can be used to group any arbitrary collection of signals [19]. Application specific conduit interfaces, exported to the top level, usually represent I/O signals in a component’s HDL logic that are not part of any other Qsys interconnect type connected to an external device [6].
3.2. Nios II SBT Tutorial

The Nios II SBT contains many templates and example projects included as part of the Altera EDS (Embedded Design Suite). The Nios II SBT tutorial found in Appendix B utilizes the Nios II Application and BSP from Template to quickly bring-up a Nios II application (MicroC OS-II) on the DE1 development board.

The MicroC OS-II kernel runs on top of the HAL (Hardware Abstraction Layer) -based BSP (Board Support Package) for the Nios II system [5]. The HAL is a runtime environment that provides a device driver interface for applications to connect to the underlying Nios II system [8]. The HAL API is integrated with the ANSI C library, providing applications access to standard C library functions [8]. The Nios II SBT utilizes the .sopcinfo artifact generated by Qsys to generate a custom HAL based BSP. Modifications to the Nios II hardware configuration are reflected in the HAL device driver configuration when the BSP is re-generated in Nios II SBT [8]. It is important to note that if any changes are made to the hardware configuration (reflected in the .sopcinfo) or the BSP settings, the BSP will need to be re-generated to reflect those changes. The HAL device driver abstraction delivers a clear distinction between application and driver, allowing for reusable application code that is robust to changes in hardware [8].
4. LABS

The MicroC OS-II labs in this chapter can be found in Appendix C. The labs are intended to illustrate classic OS concepts in a MicroC OS-II RTOS environment. Each section is structured as follows: The Prerequisite section will outline what OS concepts the lab will be leveraging. The Goal section will outline what is to be accomplished in the lab. The To Do section will describe the code provided (if any) as well as what functionality will need to be implemented. Finally, the Conclusion will wrap up the lab. This section includes questions to expand upon.

4.1. Producer-Consumer

The classic producer-consumer problem is an example of a multi-task synchronization problem in which two tasks, a producers and a consumer, share a common buffer data structure [20]. In the case of this lab, we are using a ring buffer as the shared buffer. The producer fills (produces) a slot in the buffer while the consumer takes away (consumes) a slot in the buffer.

4.1.1. Goal

This lab serves as an introduction to using the MicroC OS-II. All the functionality is implemented and is meant to serve as a coding example for subsequent labs. First and foremost, this lab outlines the MicroC OS-II startup and initialization process. This lab also provides an example of the correct usage of MicroC OS-II semaphores and illustrates PIO (Programmed I/O) and BAR (Base Address Register) manipulation by displaying the state of the ring buffer on the DE1 LEDs and seven segment displays.
4.1.2. Prerequisites

The code provided for this lab can be found in Appendix C.1 and includes the following files:

producer_consumer.c and producer_consumer.h.

This lab requires the completion of both the Quartus II/Qsys and Nios II SBT tutorials.

This lab requires an understanding of the concept of binary semaphores and mutual exclusion. The shared ring buffer can only be modified by one task (producer or consumer) at a time and this mutual exclusion is provided by the MicroC OS-II semaphores. PIO and BAR manipulation is required to illustrate the state of the ring buffer on the DE1 board. Access to the MicroC OS-II API, which can be found in [4], will also assist any student working on this lab.

4.1.3. To Do

An illustration of this lab can be seen in Figure 4.1.1 below.

![Figure 4.1: Producer-consumer lab](image-url)
In order for the student to properly understand and reason about any MicroC OS-II application, the API, found in [4], should be reviewed for the following OS functions: OSSemCreate(), OSTaskCreate(), OSStart(), OSSemPend(), OSSemPost(), and OSTimeDlyHMSM().

It is important to notice that the initialization of the OS data structures is taken care of in the main—before MicroC OS-II is started. A binary semaphore is initialized with a call to the MicroC OS-II function OSSemCreate(). Passing a 1 here initializes the semaphore as a binary semaphore which provides mutual exclusion. At least one task must be created before the OS is started. However, other tasks are capable of creating new tasks once the OS is started [4]. In this lab both the producer and consumer task are created in the main—before OSStart()—by calling OSTaskCreate(). At the end of main the OS can be started with a call to OSStart(); this starts task switching and gives the OS control.

Next, study the usage of semaphores in producer_consumer.c. A call to OSSemPend() attempts to acquire the semaphore and a call to OSSemPost() releases the semaphore. Any task wishing to modify a shared data structure must first acquire the associated semaphore, modify the data, and then release the semaphore.

The structure of a MicroC OS-II task should also be studied. From looking at the source code it should become evident that each task can be simplified to a basic structure. The main body of every task (other than tasks that run once and return) lies within an infinite loop. The cyclic structure, within the body of the loop, is as follows: the task performs computations and then makes a system call to OSTimeDlyHMSM() to give up the CPU. It is important to remember that the MicroC OS-II preemptive scheduler always runs the highest priority, ready task and if that task never gives up the CPU, lower priority tasks would be starved of execution time [4].
4.1.4. Conclusion

This producer-consumer lab is an important introductory lab where the student should have gained the skills to complete the subsequent labs in this series. It provides a working MicroC OS-II application that implements the producer-consumer classic OS problem in order to illustrate to the student the proper usage of MicroC OS-II tasks and semaphores as well as the proper of use of PIO and BARs in order to interface with the DE1’s LEDs and seven segment displays.

4.2. Readers-Writers

The classic readers-writers problem is an example of a common problem in concurrency, where many tasks must access a shared data structure concurrently; some tasks are performing reads and others are performing writes [21]. It logically follows that concurrent reads are allowable if and only if there are no active writes and a write is allowable if and only if there are no active reads. In the readers-writers lab there are two tasks, to start with, in the provided code: the reader and writer. The writer writes the pangram: “A quick brown fox jumps over the lazy dog” to an array of strings. The writer will overwrite the book array in a ring buffer like fashion. The reader reads the contents of the array and prints it to the Nios II console. Access to the shared array is synchronized with a semaphore.

4.2.1. Goal

The readers-writers lab reinforces the MicroC OS-II startup and initialization process covered in the producer-consumer lab as well as illustrating another application of semaphores in a shared data situation. The students will leverage knowledge gained from the producer-consumers lab by
extending the functionality of the given code to allow for multiple readers as well as conveying the application’s state to the Nios II console.

4.2.2. Prerequisites

The code provided for this lab can be found in Appendix C.2 and includes the following files: reader_writer.c and reader_writer.h.

This lab requires the completion and understanding of the producer-consumer lab and all of its prerequisites.

The lab also requires an understanding of reentrancy. A reentrant function can be called by multiple tasks without the threat of data corruption [4]. Reentrant functions are constrained to using local variables, allocated on the stack, and semaphore protected global variables. The use of reentrant functions allows for cleaner code because the application is not filled up with many copies of functions that do essentially the same task. Debugging is a much simpler task because modifications automatically propagate to all tasks and copy-paste bugs are all but completely eliminated. On the other hand, reentrant functions can introduce hard to debug behavior when logical bugs in a seemingly reentrant function (making it non-reentrant) create unintended side effects. The use of reentrant functions requires a different mindset than traditional sequential programming. While the function remains locally sequential, the student must keep in mind that there are other tasks running an instance of the same code and the only thing differentiating them is a unique task identifier passed to the task when it was created.
4.2.3. To Do

An illustration of this lab can be seen in Figure 4.2 below.

This lab tasks the students with extending the functionality of the existing code to allow three readers and one writer as well as to display the state of execution on the Nios II console. The readers should be created in a loop using successive calls to OSTaskCreate() passing the reentrant reader function, a unique task identifier, a pointer to the top of the task’s stack, and a unique task priority. Each task has its own stack data structure and global scope is shared among tasks. Since the readers will essentially be copies of one another the task identifier is the only way they tell themselves apart.
Multiple binary semaphores will be needed to protect housekeeping data structures. These semaphores include a read lock and a write lock. Readers acquire the read lock to update the read count. After updating the read count, each reader checks to see if they are the only reader. If they are the first reader they need to acquire the write lock (which locks out the writer) else they can proceed with the read because another reader has already locked out the writer. When a reader is done, it decrements and checks the read count, releasing the writer lock if the read count is zero.

Displaying the execution state on the Nios II console will be no simple task since there are going to be three reader tasks and one writer task trying to print to the console. Using bare printf() statements to convey the state of execution will muddy up the Nios II console. There are many ways to accomplish this task so the student should take into account the following notes and implement a suitable solution. It is recommended that statements be printed in places where they won't end up garbling the console. For example, the writer should always execute mutually exclusive of the readers so any print statements in the writer will print fine. Place print statements at synchronization check points such as acquiring and releasing the writer semaphore. Embed reader task ids into the print statements so the reader task can be identified. Taken in sum, these print statements are meant to give the student feedback as to what is going on in their application so the student can correct any bugs or unintentional side effects that may crop up.

It is important that the student experiment with inserting delays—using OSTimeDlyHMSM()—into the code. The higher priority readers should delay longer allowing other readers a time slice and the writer will need to delay as well.
4.2.4. Conclusion

After completing this lab the student should be proficient in the use of semaphores and understand where data synchronization with semaphores is necessary.

This lab can be modified to add multiple writers. In this scenario it may be easier for the student to treat the execution state as a log and have one task print it to the console at certain intervals.

4.3. Dining Philosophers Lab

The classic dining philosophers problem is used to propose a solution to a concurrent design in which resources must be shared among competing tasks [22]. A total of N silent philosophers sit around a table with rice in front of them and chopsticks between them [22]. N is commonly chosen to be five as can be seen in Figure 4.1 below. Each philosopher can be in one of two states: thinking or eating. In order to eat, a philosopher needs to pick up the chopsticks at his or her left and right. This poses an inherent threat of deadlock because it is likely that a philosopher will pick up one chopstick and find that the other chopstick has been taken.

![Figure 4.3.1: Classic dining philosophers](1)

31
4.3.1. Goal

The dining philosophers lab builds upon the semaphore, mutual exclusion, and reentrancy concepts introduced in the producer-consumer and readers-writers labs. The classic dining philosophers problem is modified for simple graphical representation and logical reasoning. This lab challenges the student to implement a solution that will prevent deadlock.

4.3.2. Prerequisites

There will be no code provided in the appendix for this lab.

This lab requires the completion and understanding of the producer-consumer lab and all of its prerequisites as well as the readers-writers lab and all of its prerequisites.

The lab also requires an understanding of deadlock. If a philosopher picks up a utensil only to find that the other utensil required to eat is taken (assuming that the philosopher does not release the utensil he or she already has) deadlock will ensue and all progress will halt. The four conditions for deadlock are met by the following: Mutual exclusion, the shared utensils must be used mutually exclusive of adjacent philosophers. Device holding; each philosopher must hold two utensils (devices) any of which can be held by adjacent philosophers. No preemption; once a philosopher holds a utensil (device) it can only cooperatively release it. Circular waiting; since each philosopher shares a utensil with adjacent philosophers, it is easy to see how a circular waiting condition can occur; for example, when every philosopher picks up a right utensil.
4.3.3. To Do

There are four dining philosophers that sit at a table with a compass oriented at its center. \( N \) is chosen to be four to simplify the problem illustration (Figure 4.2) and to make it easier for the student to model and reason about the problem; as long as adjacent philosophers share a common resource, \( N \) is unimportant. A philosopher sits at each of the cardinal directions on the compass—namely North, South, East, and West; this is illustrated in Figure 4.2 below. The philosophers exist in two states, either thinking or eating—if it helps to visualize a plate of food on the compass, make it so. If a philosopher wishes to stop thinking and eat, he or she must pick up the adjacent eating utensils (left and right) and enter the eating state. A shared utensil sits at each of the intercardinal directions—namely Northwest, Southwest, Northeast, and Southeast. The utensils are shared by neighboring philosophers.

![Figure 4.3.2: Cardinal dining philosophers](image)
In this scenario each philosopher is a task and the shared utensils are semaphores. Each philosopher is created in the thinking state and attempts to enter the eating state. The code for each philosopher should be generic and reentrant so that it may be reused in a similar fashion to the multiple readers in the readers-writers lab.

An array of structs should be created that each philosopher task can index into with its unique task identifier. Each struct will store the left and right utensils (semaphores) for each philosopher. Each philosopher’s task should index into the array of structs to find which semaphores it needs to eat. The philosopher tasks should acquire the semaphores in a way which will prevent deadlock, perhaps by utilizing the timeout feature of MicroC OS-II semaphores. Once a philosopher acquires the utensils (semaphores) to eat, it should print to the Nios II console: "<Direction> philosopher is eating". The console should be protected by a semaphore to ensure that it does not get muddied up. Each philosopher should eat for five seconds and then release its resources (left and right semaphores). When finished eating, the philosopher task should enter the thinking state for a fixed length of time, which may need to be empirically determined if some of the lower priority philosopher tasks are getting starved.

4.3.4. Conclusion

After completing this lab the student should be well versed in the philosophy of deadlock prevention and shared resource contention. Each philosopher task should have reused the same generic reentrant function facilitating cleaner code and easier debugging.
4.4. Automated External Defibrillator (AED) Controller

“An automated external defibrillator (AED) is a portable electronic device that automatically diagnoses the potentially life threatening cardiac arrhythmias of ventricular fibrillation and ventricular tachycardia in a patient, and is able to treat them through defibrillation, the application of electrical therapy which stops the arrhythmia, allowing the heart to reestablish an effective rhythm” [24]. For the purposes of this lab, the AED controller is considered a hard real-time system because the patient’s life (hopefully returning from interrupt) depends on its timely operation.

4.4.1. Goal

Model a basic AED controller. The operation of the AED will be controlled by a switch on the DE1 development board. Since we know the RC time constant of the AED capacitor bank, we can theoretically calculate how long it will take to charge the bank to a given threshold voltage suitable for defibrillation of the patient. For the purposes of this lab, the time required to charge the capacitor bank has been arbitrarily chosen to be 1 second. After the switch has been flipped and the time delay finished, defibrillation can be triggered. This is an over simplification of an AED controller and it serves purely pedagogical purposes.

4.4.2. Prerequisites

There will be no code provided in the appendix for this lab.

This lab requires the completion and understanding of the producer-consumer lab and all of its prerequisites as well as the readers-writers lab and all of its prerequisites.
4.4.3. To Do

An illustration of this lab can be seen in Figure 4.4 below.

This lab requires two tasks: a polling task and a pending task. The pending task engages the defibrillator when signaled by the polling task via a semaphore.

The pending task begins its task loop with a call to OSSemPend(), waiting for the signal OSSemPost() from the polling task to begin defibrillation. Once the semaphore is acquired, the pending task prints to the Nios II console indicating that defibrillation has successfully occurred.

Since user defined interrupts are beyond the scope of this paper (see Future Work section) this lab will simulate an interrupt by using a high frequency, high priority polling task. When the polling task detects that the switch has been flipped, it prints to the Nios II console and delays for one second, using OSTimeDlyHMSM(), then signals the pending task with a call to OSSemPost().
The Nios II system used for the previous labs will need to be modified to add a switch or push button as a PIO input device.

4.4.4. Conclusion

This lab provides the student with a practical application of real-time timing constraints and pseudo-interrupt servicing. The semaphore is serviced as a type of inter-process communication (see Future Work section) that allows the polling task to signal to the (waiting) pending task that it should engage the defibrillator after a fixed capacitor charging period.

4.5. Sleeping Barber

The sleeping barber problem is a classic synchronization problem between multiple tasks attempting to utilize a shared resource [23]. The premise of the problem is that there is one barber who must service multiple customers from a fixed size waiting room. When the barber finishes cutting a customer’s hair he checks the waiting room. If there is a customer the customer takes the empty barber’s chair and receives a haircut. If the waiting room is empty the barber goes back to his chair and sleeps in it. When a customer arrives he or she checks to see what the barber is doing. If the barber is asleep, the customer wakes up the barber, sits in the empty barber’s chair and receives a haircut. If the barber is busy, the customer checks the waiting room for an available seat leaving to return later if the waiting room is full.

4.5.1. Goal

Implement a MicroC OS-II application with eight tasks (customers) sharing a single barber with a single barber chair (semaphore) and a waiting room displaying the state of the barber shop on the DE1 development board.
4.5.2. Prerequisites

The code provided for this lab can be found in Appendix C.3 and includes the following files: sleeping_barber.c and sleeping_barber.h.

This lab requires the completion and understanding of the producer-consumer lab and all of its prerequisites as well as the readers-writers lab and all of its prerequisites.

4.5.3. To Do

An illustration of this lab can be seen in Figure 4.5 below.

![Diagram](image)

**Figure 4.5: Sleeping barber lab**

The implementation provided here creates eight processes representing the customers. Each customer has a three letter name that can be displayed on the seven segment displays: HEX3, HEX2, & HEX1. HEX0 displays a ten second timer that counts down ten seconds for each haircut. The barber function is protected by the semaphore, barber_sem. The waiting room is implemented as a counting semaphore, waiting_room_sem, initialized to three thus allowing 3 customers in the waiting room.
The generic customer task is provided with an empty task loop. Implement the task loop per the sleeping barber problem described above. If a task cannot enter the waiting room it should delay using OSTimeDlyHMSM().

It should be noted that (due to the deterministic nature of MicroC OS-II) once a working implementation has been achieved the pattern of customers receiving haircuts will continue in the same order. This is a side effect of using an RTOS; therefore, attempting to achieve randomness in the order of customers is unnecessary.

4.5.4. Conclusion

This lab provides the student with a real-time visualization of the application’s execution state while illustrating the deterministic execution of an RTOS. The side effects of semaphores, priority, and timing are conveyed through the use of on-board displays.
5. RESULTS

The tutorials and labs (not including the AED controller lab) were deployed in an embedded system laboratory environment where they were put to the test by groups of students. The actual lab will be described as well suggestions for improvement. The lab groups were formed at the beginning of the semester and the students remained together throughout the completion of the course. The lab took place in a computer lab, which is the ideal lab setup.

5.1. Lab Setup

There are many considerations when setting up a lab for senior and beginning graduate students. How the lab is setup has an impact on the efficient use of lab time and the efficacy of the labs. Lab problems can easily eat up a majority of the allocated lab time, so proper planning and foresight is crucial. One caveat to the ideal computer lab setup is allowing the students to use their own laptops. If this is allowed, expect to spend a considerable amount of time assisting students setup their environments. It is suggested that the computer lab be provided with all of the necessary software and drivers installed and that any student wishing to use his or her laptop be instructed to set it up before coming to lab. This way, any questions can be addressed outside of class and valuable lab time is not wasted.

5.1.1. Development Environment

The development environment OS for the tutorials in this paper was Windows XP. Windows was chosen because Altera has limited support for Linux, and setting up the environment on an unsupported Linux distribution was time consuming and complicated. Since each OS contributes its own unique set of pitfalls to the tutorial, it is best to pick one and stick with it. Regardless of these considerations, a majority of time in the first few lab sessions was spent helping students get the tools working on Windows 7, Windows 8, and Ubuntu Linux.
5.1.2. Development Board

Each group of 3-4 students was required to purchase and share a DE1 development board. It should not need stating that the development board needs be taken to lab. Poor group communication and preparation often left groups without a development board to complete the lab.

5.1.3. Working Lab Implementation

It is beneficial to have a working copy of the assigned lab on display during the lab session. The running development board can be on display and the Nios II console displayed on the classroom projector.

5.2. Quartus II/Qsys Tutorial

The Quartus II/Qsys tutorial is by far the most complicated of the tutorials. The steps have to be followed exactly and—most of the time—in order.

5.2.1. Qsys

The Qsys Nios II system specification and generation is by far the most time consuming and tedious part of the Quartus II/Qsys tutorial. Even if the system generates without errors it is possible to end up with a system that is inoperable, requiring the student to either thoroughly inspect the Nios II system within Qsys or start the tutorial over. Many times, incorrect connections in the Qsys interconnect are the culprit.

5.2.2. Quartus II

At lot of lab time was spent helping students fix syntax related compilation errors in the Verilog HDL. Ensuring design files were properly added to Quartus II and the top-level entity was properly set was also an issue. Students also had trouble with FPGA configuration, mostly
remembering (at least for our setup) that FPGAs lose configuration when turned off and need to be reprogrammed with the Quartus programmer.

5.3. Nios II SBT Tutorial

The Nios II SBT tutorial is by far the shortest and easiest of the tutorials. The only problem with this tutorial is that each Nios II system generated by Qsys is given an identifier which is stored in the system id hardware module. When Nios II SBT goes to download a Nios II application, it checks this identifier against the Nios II system the BSP was generated for and if there is a mismatch the download fails. This mechanism is meant to ensure that the hardware and drivers stay in sync. This can become a tedious process when the student is simply trying to get the Nios II system working and is not making any architectural changes. The Nios II system generated in Qsys does not have a system id module and to get around the system identifier mismatch error the Nios II SBT run configurations have to be modified to ignore mismatched system identifiers.

5.4. Labs and OS Concepts

Students did not dissect the introductory producer-consumer lab as intended and were not effective in MicroC OS-II application development until the basic concepts were properly introduced. A lot of students had trouble with the concept of mutual exclusion and the critical section when dealing with shared data. Students either failed to protect shared data access altogether or they would protect only writes. Once these issues were taken care of, the students were able to successfully complete the rest of the labs.
6. CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

We briefly discussed the necessary OS and RTOS concepts necessary to implement and reason about MicroC OS-II applications in the labs. The tool chain and design flow were described so that the student could better understand and follow the tutorials.

The Altera tools were discussed and described in detail, providing the student with an idea of their purpose and how they fit into the design flow. The relationships between the tools were emphasized so that the student could follow the design flow as outlined in the tutorials.

Labs were provided to challenge the student to develop MicroC OS-II applications leveraging the OS and RTOS concepts discussed in the background section. The introductory labs included source code intended to guide the student through the MicroC OS-II application development process. Later labs only offered specifications and ideas, leaving all the MicroC OS-II application development to the student.

Therefore, we can conclude that for a student to successfully develop a MicroC OS-II application on an embedded systems platform the student needs to understand the entire process from Nios II system specification and generation to application download.
6.2. Future Work

This work presents a self-contained set of tutorials and labs aimed at taking the student from the hardware system specification to MicroC OS-II application download. The hardware system does not however include any user defined interrupt devices. A future research goal would be to develop a tutorial that would take the student through the process of MicroC OS-II ISR development and registering interrupts with Altera’s HAL.

Future updates to this document will be available at http://www.ece.uc.edu/~cpurdy/.
REFERENCES

APPENDIX A

Quartus II/Qsys Tutorial
Eric Swegert
March 2013

1. Start Quartus II.

2. Create a new project. From the Quartus II toolbar select File > New > New Quartus II Project; click Next to skip the Introduction page. In the Directory, Name, Top-Level Entity (1 of 5) page create a working directory (without spaces) then append \de1_tutorial to the end (this creates a project folder). Enter de1_tutorial as the project name, which is also the name of the top level entity. Confirm that the page is the same as in Figure A.1; click Next and Yes when asked to create the project directory. Click Next to skip the Add Files (2 of 5) page. In the Family & Device Settings (3 of 5) page under Target device select the option Specific device selected in ‘Available devices’ list then choose EP2C20F484C7 as the target device. Confirm that the page is the same as in Figure A.2; click Finish to create the project.

Figure A.1: New project wizard [page 1 of 5]
From the Quartus II toolbar, select Tools > Qsys. Then, from the Qsys toolbar, select File > Save As..., then enter nios_system; this will be the name of the system generated by Qsys.

4. Within Qsys, confirm that the Project Settings tab is the same as in Figure A.3.
Note: Steps 5-26 use the Qsys Components Library to build nios_system from individual modules.

5. Within Qsys, where it says Components Library, select Embedded Processors > Nios II Processor; click Add.

6. Within the Nios II MegaCore page, select the Nios II/e core; click Finish.

7. Within Qsys, where it says Components Library, select Memories and Memory Controllers > On-Chip > On-Chip Memory (RAM or ROM); click Add.

8. Within the On-Chip Memory MegaCore page; click Finish.

9. Within Qsys, where it says Components Library, select Interface Protocols > Serial > JTAG UART; click Add.

10. Within the JTAG UART MegaCore page; click Finish.

11. Within Qsys, where it says Components Library, select Peripherals > Microcontroller Peripherals > Interval Timer; click Add.

12. Within the Interval Timer MegaCore page; click Finish.
13. Within Qsys, where it says Components Library, select Memories and Memory Controllers > External Memory Interfaces > SDRAM Interfaces > SDRAM Controller; click Add.

14. Within the SDRAM Controller MegaCore Parameter Settings page, change the page to look like Figure A.4; click Finish.

Figure A.4: SDRAM controller parameter settings
Note: The system will use the DE1’s 50 MHz clock source (external clock source), however it will need an ALTPLL to create a -3ns shifted clock signal for the SDRAM controller/SDRAM chip (accounts for clock skew on the DE1 board) and a 25 MHz clock source for the toggling red LEDs.

15. Within Qsys, where it says Components Library, select PLL > Avalon ALTPLL; click Add.

16. Within the MegaWizard Plug-In Manager change the Parameter Settings > General/Modes to the settings in Figure A.5; click Next.

Figure A.5: ALTPLL parameter settings [page 1 of 7]
17. Within the MegaWizard Plug-In Manager change the Parameter Settings > Inputs/Lock tab to the settings in Figure A.6; click Next.

Figure A.6: ALTPLL parameter settings [page 2 of 7]
18. Within the MegaWizard Plug-In Manager leave the Parameter Settings > Clock switchover tab unchanged; click Next.

19. Within the MegaWizard Plug-In Manager change the Output Clocks > clk c0 tab to the settings in Figure A.7; click Next.

Figure A.7: ALTPLL output clocks [page 4 of 7]
20. Within the MegaWizard Plug-In Manager change the Output Clocks > clk c1 tab to the settings in Figure A.8; click Next.

Figure A.8: ALTPLL output clocks [page 5 of 7]
Note: Clock source c2 will be exported to the toggle.v module which implements the flashing LEDs on the DE1 board.

21. Within the MegaWizard Plug-In Manager change the Output Clocks > clk c2 tab to the settings in Figure A.9; click Next.

Figure A.9: ALTPLL output clocks [page 6 of 7]
22. Within the MegaWizard Plug-In Manager leave the EDA tab unchanged; click Finish to add the PLL to the system.

Note: In a Qsys system a clock source can be exported or connected internally, but not both. This necessitates the use of a clock bridge. The clock bridge allows clock source `c1` to be exported to the SDRAM chip as well as internally connected to the SDRAM controller. The Qsys interconnect will be added in steps 28-30.

23. Within Qsys, where it says Components Library, select Bridges > Clock Bridge; click Add.

24. Within the MegaCore Clock Bridge page, change the settings to look like Figure A.10; click Finish.

Figure A.10: Clock bridge parameters
25. Within Qsys, where it says Components Library, select Peripherals > Microcontroller Peripherals > PIO (Parallel I/O); click Add.

26. Within the MegaCore PIO (Parallel I/O) page, change the settings to look like Figure A.11; click Finish.

Figure A.11: PIO settings (HEX0)

27. Within Qsys, select the newly added component and, from the toolbar, select Edit > Rename: HEX0; press enter.

28. Repeat steps 26-27 three more times, changing the names to: HEX1, HEX2, and HEX3.

25. Within Qsys, where it says Components Library, select Peripherals > Microcontroller Peripherals > PIO (Parallel I/O); click Add.
26. Within the MegaCore PIO (Parallel I/O) page, change the settings to look like Figure A.12; click Finish.

Figure A.12: PIO settings (green_LEDs)

27. Within Qsys, select the newly added component and, from the toolbar, select Edit > Rename: green_LEDs; press enter.
Note: Exporting signals allows them to be used outside the Nios II system in other hardware modules implemented on the FPGA.

28. Export the necessary signals. To export a module’s signal, find its row and, in the Export column, click where it says Click to export. Once the signal name is typed, press enter to add the export; pressing esc will cancel the export. The following exports will need to be created:

- green_LEDs > external_connection > (Click and press enter)
- HEX0 > external_connection > (Click and press enter)
- HEX1 > external_connection > (Click and press enter)
- HEX2 > external_connection > (Click and press enter)
- HEX3 > external_connection > (Click and press enter)
- sram_0 > wire > (Click and press enter)
- altpll_0 > inclk_interface > (Click and press enter)
- altpll_0 > c2 > (Click and press enter)
- clock_bridge_0 > out_clk_1 > (Click and press enter)

29. Create clock connections using the ALTPLL and the Clock Bridge. First, delete the auto generated clock source clk_0 by selecting it and, from the Qsys toolbar, selecting Edit > Remove. Make the following clock connections by clicking on the corresponding empty circles within the Connections column (dark circles indicate connections):

- altpll_0.c0 > nios2_qsys.clk
- altpll_0.c0 > onchip_memory2_0.clk1
- altpll_0.c0 > jtag_uart_0.clk
- altpll_0.c0 > timer_0.clk
- altpll_0.c0 > HEX0.clk
- altpll_0.c0 > HEX1.clk
- altpll_0.c0 > HEX2.clk
- altpll_0.c0 > HEX3.clk
- altpll_0.c0 > green_LEDs.clk
- altpll_0.c1 > clock_bridge_0.in_clk
- clock_bridge_0.out_clk > sram_0.clk
30. Create master/slave connections using the Nios II Processor. Make the following
master/slave connections by clicking on the corresponding empty circles within the
Connections column (dark circles indicate connections):

- `nios2_qsys_0.data_master > onchip_memory2_0.s1`
- `nios2_qsys_0.instruction_master > onchip_memory2_0.s1`
- `nios2_qsys_0.data_master > jtag_uart_0.avalon_jtag_slave`
- `nios2_qsys_0.data_master > timer_0 > s1`
- `nios2_qsys_0.data_master > sdram_0.s1`
- `nios2_qsys_0.data_master > sdram_0.s1`
- `nios2_qsys_0.data_master > altpll_0.pll_slave`
- `nios2_qsys_0.data_master > HEX0.s1`
- `nios2_qsys_0.data_master > HEX1.s1`
- `nios2_qsys_0.data_master > HEX2.s1`
- `nios2_qsys_0.data_master > HEX3.s1`
- `nios2_qsys_0.data_master > green_LEDs.s1`
31. Assign reset and exception vectors. Within Qsys, select the Nios II Processor and, from the toolbar, select Edit > Edit.. Within the MegaCore page, select sdram_0.s1 from the Reset vector memory drop-down menu as well as the Exception Vector drop-down menu. The page should look like Figure A.13; click Finish.

Figure A.13: Nios II processor settings

32. Assign Base Addresses. From the Qsys toolbar, select System > Assign Base Addresses.

33. Create Global Reset Network. From the Qsys toolbar, select System > Create Global Reset Network.

34. Assign interrupt numbers and connect the IRQs for the JTAG UART and the Interval Timer. Within Qsys, find the jtag_uart_0 row and connect its IRQ in the IRQ column, then select the IRQ number: 16. Within Qsys, find the timer_0 row and connect its IRQ in the IRQ column, then select the IRQ number: 0.

35. Within Qsys, in the Generation tab, click Generate.

36. Within Qsys, in the Address Map tab, copy the base addresses of HEX0, HEX1, HEX2, HEX3, and green_LEDs for later use in application software.

37. Within Qsys, in the HDL Example tab, copy the instantiation of nios_system (Verilog) for later use in the Quartus II top-level entity.
38. Within Quartus II, select File > New > Verilog HDL File.

39. Within Quartus II, select File > Save As… del_tutorial.v.

40. Within Quartus II, paste the code from Figure A.14 into del_tutorial.v.

41. Within Quartus II, select File > New > Verilog HDL File.

42. Within Quartus II, select File > Save As… toggle.v.

43. Within Quartus II, paste the code from Figure A.15 into toggle.v.

44. Within Quartus II, select Assignments > Import Assignments… DE1_pin_assignments.csv.

45. Within Quartus II, select Processing > Start Compilation.

46. Within Quartus II, select Tools > Programmer.

47. Within Programmer, click Hardware Setup.

48. Within Hardware Setup under the Hardware Settings tab, select USB-Blaster from the drop-down menu; click Close.

49. Within Programmer, leave the Mode settings as default; click Start.

Note: At this point the .sof has been downloaded to the FPGA, leaving it configured and running. If the FPGA loses power at any point, it will need to be reconfigured. This is accomplished by repeating steps 46-49.

50. Exit Quartus II.
module de1_tutorial(CLOCK_50, LEDG, LEDR, DRAM_CLK, DRAM_CKE, DRAM_ADDR, DRAM_BA_1, DRAM_BA_0, DRAM_CS_N, DRAM_CAS_N, DRAM_RAS_N, DRAM_WE_N, DRAM_DQ, DRAM_UDQM, DRAM_LDQM, HEX3, HEX2, HEX1, HEX0);

input CLOCK_50;
output [7:0] LEDG;
output [9:5] LEDR;
output [6:0] HEX3;
output [6:0] HEX2;
output [6:0] HEX1;
output [6:0] HEX0;
output [11:0] DRAM_ADDR;
output DRAM_BA_1, DRAM_BA_0, DRAM_CS_N, DRAM_RAS_N, DRAM_CLK;
output DRAM_CKE, DRAM_CS_N, DRAM_WE_N, DRAM_UDQM, DRAM_LDQM;
inout [15:0] DRAM_DQ;

wire toggle_clock;

nios_system u0 (  
    .pio_0_external_connection_export (LEDG),  
    .hex3_external_connection_export (HEX3),  
    .hex2_external_connection_export (HEX2),  
    .hex1_external_connection_export (HEX1),  
    .hex0_external_connection_export (HEX0),  
    .sdram_0_wire_addr (DRAM_ADDR),  
    .sdram_0_wire_ba ({DRAM_BA_1, DRAM_BA_0}),  
    .sdram_0_wire_cas_n(DRAM_CAS_N),  
    .sdram_0_wire_cke (DRAM_CKE),  
    .sdram_0_wire_cs_n(DRAM_CS_N),  
    .sdram_0_wire_dq (DRAM_DQ),  
    .sdram_0_wire_dqm({DRAM_UDQM, DRAM_LDQM}),  
    .sdram_0_wire_ras_n(DRAM_RAS_N),  
    .sdram_0_wire_we_n(DRAM_WE_N),  
    .altpll_0_inclk_interface_clk (CLOCK_50),  
    .altpll_0_c2_clk(toggle_clock),  
    .clock_bridge_0_out_clk_1_clk (DRAM_CLK));

toggle t(toggle_clock, LEDR);

dendmodule
module toggle(clock, toggle_output);

input clock;
output[9:5] toggle_output;

reg[30:0] toggling;

always @(posedge clock)
  begin
    toggling<=toggling + 1;
  end

assign toggle_output[9] = toggling[21];
assign toggle_output[5] = toggling[21];

assign toggle_output[8] = toggling[26];
assign toggle_output[6] = toggling[26];

assign toggle_output[7] = toggling[27];
endmodule
References: Quartus II/Qsys Tutorial

APPENDIX B

Nios II SBT Tutorial
Eric Swegert
March 2013

1. Start Nios II SBT for Eclipse (Windows 7 users: Right-click > Run as administrator).

2. Select File > New > Nios II Application and BSP from Template.

3. In the new project wizard, where it says SOPC Information File name, browse to the sopcinfo file created in the Quartus II/Qsys tutorial.

4. In the new project wizard, where it says Project Name type de1_uosii_tutorial.

5. In the new project wizard, select Hello MicroC/OS-II as the template then click Finish.

6. In the project explorer, expand de1_uosii_tutorial and delete hello_uosii.c.

7. Drag and drop the provided source files (if any) into de1_uosii_tutorial; select to copy the files.

8. If the project uses PIO, ensure that the PIO base addresses are the same as the addresses copied from Qsys.

9. Right-click de1_uosii_tutorial_bsp, select Nios II > BSP Editor; the Nios II BSP Editor opens.

10. In the BSP Editor Main tab, locate the expandable tree of settings, expand Advanced, and then expand and select ucosii. You can see that the BSP settings for MicroC/OS-II are highly configurable. The settings determine which MicroC/OS-II options are included in the binary image. The Nios II SBT also saves the setting values to the system.h file; to see descriptions of the settings, expand and select each setting.

11. Expand the Common settings, expand and select hal, and ensure that stderr, stdin, and stdout are all set to jtag_uart_0.

12. If you make any changes, save the BSP and click Generate to recreate the BSP files; click Exit to close the BSP Editor.

13. Select Project > Build Project.

14. Right click de1_uosii_tutorial and select Run As > Run Configurations.
15. In the Run Configurations window, under the Target Connection tab, check the Ignore mismatched system ID and the Ignore mismatched system timestamp checkboxes; click Apply and Close.

16. Right click de1_uosii_tutorial and select Run As > Nios II Hardware.
References: Nios II SBT Tutorial


[7] Altera Embedded Software Development,


APPENDIX C: MicroC OS-II Labs Source Code

C.1. Producer-Consumer Lab Source Code

C.1.1. producer_consumer.c

```c
#include <stdio.h>
#include <unistd.h>
#include "includes.h" // includes header full of necessary includes
#include "alt_uosii_simple_error_check.h"
#include "producer_consumer.h"

// ring buffer data structure
struct ringBuffer{
    INT8U readMe; // index of oldest element to be read
    INT8U bufSize; // fixed size of ring buffer
    INT8U numElements; // number of elements currently in ring buffer
    INT16U ring[BUFFER_SIZE]; // ring buffer
}ringBuffer;

// task stack definitions
OS_STK    consumer_stk[TASK_STACKSIZE];
OS_STK    producer_stk[TASK_STACKSIZE];

// ring buffer semaphore definition
OS_EVENT *buffer_sem;

// light up green LEDs to reflect state of buffer
void lightLEDs(int rw, INT8U stateOfLEDs)
{
    switch(stateOfLEDs){
    case 0:
        if(rw == WRITE){
            *green_LEDs = *green_LEDs | BIT0;
        }else{
            *green_LEDs = *green_LEDs & BIT0NOT;
        }
        return;
    case 1:
        if(rw == WRITE){
            *green_LEDs = *green_LEDs | BIT1;
        }else{
            *green_LEDs = *green_LEDs & BIT1NOT;
        }
        return;
    case 2:
        if(rw == WRITE){
            *green_LEDs = *green_LEDs | BIT2;
        }else{
```
*green_LEDs = *green_LEDs & BIT2NOT;
)
return;
case 3:
    if(rw == WRITE){
        *green_LEDs = *green_LEDs | BIT3;
    }else{
        *green_LEDs = *green_LEDs & BIT3NOT;
    }
return;
case 4:
    if(rw == WRITE){
        *green_LEDs = *green_LEDs | BIT4;
    }else{
        *green_LEDs = *green_LEDs & BIT4NOT;
    }
return;
case 5:
    if(rw == WRITE){
        *green_LEDs = *green_LEDs | BIT5;
    }else{
        *green_LEDs = *green_LEDs & BIT5NOT;
    }
return;
case 6:
    if(rw == WRITE){
        *green_LEDs = *green_LEDs | BIT6;
    }else{
        *green_LEDs = *green_LEDs & BIT6NOT;
    }
return;
case 7:
    if(rw == WRITE){
        *green_LEDs = *green_LEDs | BIT7;
    }else{
        *green_LEDs = *green_LEDs & BIT7NOT;
    }
return;
}

// BCD arithmetic to increment runningTotal
INT16U incrementBCD(INT16U runningTotal)
{
    if(runningTotal == 0x0255){
        printf("runningTotal reset\n");
        return 0x0001;
    }

    INT16U ONE = 0x0001;
    INT16U carry = 0x0000;

INT16U retVal = 0x0000;
INT16U upperNibble = (runningTotal & 0x0f00) >> 8;
INT16U middleNibble = (runningTotal & 0x00f0) >> 4;
INT16U lowerNibble = runningTotal & 0x000f;

lowerNibble = lowerNibble + ONE;
if(lowerNibble > 0x0009){
    carry = (lowerNibble + 0x0006);//convert carry to BCD
    lowerNibble = carry & 0x000f;//mask off carry, leaving
shift right 4
    carry = (carry & 0x000f0) >> 4;//mask off lowerNibble and
    middleNibble = (middleNibble + carry);
    if(middleNibble > 0x0009){
        carry = (middleNibble + 0x0006);//convert carry to
        BCD
        middleNibble = carry & 0x000f;//mask off carry, leaving
middleNibble
        carry = (carry & 0x000f0) >> 4;//mask off middleNibble
and shift right 4
        upperNibble = (upperNibble + carry);
        if(upperNibble > 0x0009){
            carry = (upperNibble + 0x0006);//convert carry
to BCD
            upperNibble = carry & 0x000f;
            carry = (carry & 0x000f0) >> 4;
            lowerNibble = lowerNibble + carry;
            //reconstruct packed BCD
            retVal = retVal | (upperNibble << 8);
            retVal = retVal | (middleNibble << 4);
            retVal = retVal | lowerNibble;
            return retVal;
        }
    }
}
else{
    //reconstruct packed BCD
    retVal = retVal | (upperNibble << 8);
    retVal = retVal | (middleNibble << 4);
    retVal = retVal | lowerNibble;
    return retVal;
}
}
}
return retVal;
}

//returns seven segment encoding of encode_me
INT8U sevenSegEncoder(int encode_me)
{
    switch(encode_me){
    case 0:
        return SevenSeg0;
        break;
    case 1:
        return SevenSeg1;
        break;
    case 2:
        return SevenSeg2;
        break;
    case 3:
        return SevenSeg3;
        break;
    case 4:
        return SevenSeg4;
        break;
    case 5:
        return SevenSeg5;
        break;
    case 6:
        return SevenSeg6;
        break;
    case 7:
        return SevenSeg7;
        break;
    case 8:
        return SevenSeg8;
        break;
    case 9:
        return SevenSeg9;
        break;
    }
    return SevenSeg0;
}

//prints info to NIOS II Console,
//and writes displayNumBCD to the seven segment displays
void displayDigits(INT8U RW, INT16U displayNumBCD){
    INT8U i = 0;
    //mask and shift to extract digits
    INT16U upperNibble = (displayNumBCD & 0x0f00) >> 8;
    INT16U middleNibble = (displayNumBCD & 0x00f0) >> 4;
    INT16U lowerNibble = displayNumBCD & 0x000f;
}
if(RW == READ){
  *hex3 = SevenSegDASH;
  printf("Consuming: ");
} else{
  *hex3 = SevenSegOFF;
  printf("Producing: ");
}

printf("%d", upperNibble);
printf("%d", middleNibble);
printf("%d\n", lowerNibble);

for(i = 0; i < ringBuffer.bufSize; i++)
  printf(" %d ", ringBuffer.ring[i]);

printf(" numElements = %d\n", ringBuffer.numElements);
printf("******************************************************\n");

*hex2 = sevenSegEncoder(upperNibble);
*hex1 = sevenSegEncoder(middleNibble);
*hex0 = sevenSegEncoder(lowerNibble);

//consumer task consumes numbers from the ring buffer
void consumer(void* pdata){
  //error check argument
  INT8U return_code;

  //task loop
  while(1){
    OSSemPend(buffer_sem, SEMAPHORE_TIMEOUT, &return_code);//acquire semaphore (system call)
    alt_uosii_check_return_code(return_code);//check for errors during system call
    if(ringBuffer.numElements <= 3){//buffer is empty release share_resource_sem
      OSSemPost(buffer_sem);//release semaphore (system call)
      alt_uosii_check_return_code(return_code);//check for errors during system call
      OSTimeDlyHMSM(0, 0, 0, 100);//delay to allow producer to catch up (system call)
    } else{//read from buffer
      INT16U temp = ringBuffer.ring[ringBuffer.readMe];//read next number from ring[]
      ringBuffer.ring[ringBuffer.readMe] = 0;//Set that member to 0
      ringBuffer.numElements = ringBuffer.numElements - 1;//update numElements
      displayDigits(READ, temp);//display read number on
    }
  }
}

//producer task produces numbers in the ring buffer
void producer(void* pdata){
  //error check argument
  INT8U return_code;

  //task loop
  while(1){
    OSSemPost(buffer_sem, SEMAPHORE_TIMEOUT, &return_code);//release semaphore (system call)
    alt_uosii_check_return_code(return_code);//check for errors during system call
    if(ringBuffer.numElements < ringBuffer.bufSize){//buffer is not full, consume from ring
      INT16U temp = ringBuffer.ring[ringBuffer.readMe];//read next number from ring[]
      ringBuffer.ring[ringBuffer.readMe] = 0;//Set that member to 0
      ringBuffer.numElements = ringBuffer.numElements + 1;//update numElements
      displayDigits(READ, temp);//display read number on
    }
  }
}
seven seg displays
    lightLEDs(READ, ringBuffer.readMe);//update LEDs to
    reflect ring[] population

    OSTimeDlyHMSM(0, 0, 1, 0);//delay half a second to
    allow viewing of seven seg displays and LED array (system call)

    ringBuffer.readMe = (ringBuffer.readMe + 1) %
    ringBuffer.bufSize;//increment readMe to next location

    OSSemPost(buffer_sem);//release semaphore (system
call)
}
}
}

//producer fills the ring buffer with incrementing numbers
void producer(void* pdata)
{
    //error check argument
    INT8U return_code = OS_NO_ERR;

    //producer's running total encoding packed BCD (easier to
display)
    INT16U runningTotalBCD = 0x0001;

    //task loop
    while(1){
        OSSemPend(buffer_sem, SEMAPHORE_TIMEOUT,
        &return_code);//acquire semaphore (system call)
        alt_uosii_check_return_code(return_code);//check for
        errors during system call

        if(ringBuffer.numElements == ringBuffer.bufSize){//buffer
        is full, release share_resource_sem
            OSSemPost(buffer_sem);//release semaphore (system
        call)
            alt_uosii_check_return_code(return_code);//check for
        errors during system call
            OSTimeDlyHMSM(0, 0, 0, 100);//delay to allow producer
        to catch up (system call)
        }else{//proceed with write operation, protected by
        shared_resource_sem
            INT8U writeHere = (ringBuffer.readMe +
            ringBuffer.numElements) % ringBuffer.bufSize;
            ringBuffer.ring[writeHere] = runningTotalBCD;//Write
            runningTotal to buffer

            if(ringBuffer.numElements ==
            ringBuffer.bufSize){//producer is overwriting elements, readMe is
            incremented accordingly, numElements unchanged
                ringBuffer.readMe = (ringBuffer.readMe + 1) %
ringBuffer.bufSize;
    } else {// no elements overwritten, increment
        numElements accordingly
        ringBuffer.numElements = ringBuffer.numElements + 1;
    }

displayDigits(WRITE, runningTotalBCD);// display read
number on seven seg displays
lightLEDs(WRITE, writeHere);// update LEDs to reflect
ring[] population

OSTimeDlyHMSM(0, 0, 1, 0);// delay half a second to
allow viewing of seven seg displays and LED array (system call)

    runningTotalBCD =
incrementBCD(runningTotalBCD); // increment runningTotal

    OSSemPost(buffer_sem);// release semaphore (system
call)

// main initializes OS data structures,
// creates producer and consumer,
// and starts the OS
int main (int argc, char* argv[], char* envp[])
{
    // initialize buffer_sem as binary semaphore
    buffer_sem = OSSemCreate(1);

    // initialize ring buffer meta data
    ringBuffer.readMe = 0;
    ringBuffer.bufSize = BUFFER_SIZE;
    ringBuffer.numElements = 0;

    // error check argument
    INT8U return_code = OS_NO_ERR;

    // create producer task
    return_code = OSTaskCreate(producer,
                              NULL, // task pointer
                              // task argument pointer (not used)
                              (void*)&producer_stk[TASK_STACKSIZE - 1], // top of
                              stack pointer (stack grows down)
                              PRODUCER_PRIORITY);

    // error check
    alt_uCOSii_check_return_code(return_code);

    // create consumer task
return_code = OSTaskCreate(consumer, NULL,
    //task argument pointer (not used)
    (void*)&consumer_stk[TASK_STACKSIZE - 1],//top of
    stack pointer (stack grows down)
    CONSUMER_PRIORITY);
    //task priority (must be unique)
    //error check
    alt_uicosii_check_return_code(return_code);

    //Start the OS task switching
    OSStart();

    return 0;
}
C.1.2.producer_consumer.h

//ensure these PIO BARS the correct
//values taken from Qsys
//(Programmed I/O Base Address Registers)
#define green_LEDs (INT8U *) 0x01003030
#define hex3 (INT8U *) 0x01003040
#define hex2 (INT8U *) 0x01003050
#define hex1 (INT8U *) 0x01003060
#define hex0 (INT8U *) 0x01003070

//task priorities, the producer one step higher (lower # = higher
//priority)
#define CONSUMER_PRIORITY 10
#define PRODUCER_PRIORITY 9
#define TASK_STACKSIZE 2048
#define BUFFER_SIZE 8
#define SEMAPHORE_TIMEOUT 0 //The task will block indefinitely for
//this semaphore
#define WRITE 0
#define READ 1

//bit patterns lighting green LEDs
//LEDs active high
//used by lightLEDs()
#define BIT0 0x01
#define BIT0NOT 0xFE
#define BIT1 0x02
#define BIT1NOT 0xFD
#define BIT2 0x04
#define BIT2NOT 0xFB
#define BIT3 0x08
#define BIT3NOT 0xF7
#define BIT4 0x10
#define BIT4NOT 0xEF
#define BIT5 0x20
#define BIT5NOT 0xDF
#define BIT6 0x40
#define BIT6NOT 0xBF
#define BIT7 0x80
#define BIT7NOT 0x7F

//seven segment display encodings
//seven segment display active low
//used by sevenSegEncoder()
#define SevenSegDASH 0x3f;
#define SevenSegOFF 0x7f;
#define SevenSeg0 0x40
#define SevenSeg1 0x79
#define SevenSeg2 0x24
#define SevenSeg3 0x30
#define SevenSeg4 0x19
#define SevenSeg5 0x12
#define SevenSeg6 0x02
#define SevenSeg7 0x58
#define SevenSeg8 0x00
#define SevenSeg9 0x18

//end producer_consumer.h
C.2: Readers-Writers Source Code

C.2.1. reader_writer.c

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include "includes.h"
#include "alt_ucosii_simple_error_check.h"
#include "reader_writer.h"

/* Definition of shared_buf_sem Semaphore */
OS_EVENT *shared_buf_sem;

/* Definition of Task Stacks */
OS_STK reader_stk[TASK_STACKSIZE];
OS_STK writer_stk[TASK_STACKSIZE];

void elipsis(){
    INT8U i;
    for(i=0; i<3; i++){
        printf(".");
        OSTimeDlyHMSM(0,0,1,0);
    }
}

void reader(void *pdata){
    INT8U return_code;
    while(1)
    {
        OSSemPend(shared_buf_sem, 0, &return_code);
        if(book_mark == 0){
            OSSemPost(shared_buf_sem);
            OSTimeDlyHMSM(0,0,2,0);
        }else{
            INT8U index = 0;
            printf("Reader is reading");
            elipsis();
            while(index < book_mark){
                printf("%s ", book[index][0]);
                index += 1;
            }
            printf("\n");
        }
        OSSemPost(shared_buf_sem);
    }
}
```
void writer(void *pdata){
    INT8U return_code;

    while(1){
        OSSemPend(shared_buf_sem, 0, &return_code);
        if(book_mark == WORDS_IN_BOOK -1){
            book_mark = 0;
        }
        printf("Writer is writing");
        elipsis();
        book[book_mark][0] = pangram[book_mark][0];
        printf("%s \n", book[book_mark][0]);
        book_mark += 1;
        OSSemPost(shared_buf_sem);
        OSTimeDlyHMSM(0,0,1,0);
    }
}

void reader_writer_init()
{
    INT8U return_code = OS_NO_ERR;
    book_mark = 0;

    //initialize pangram
    pangram[0][0] = "A\0";
    pangram[1][0] = "quick\0";
    pangram[2][0] = "brown\0";
    pangram[3][0] = "fox\0";
    pangram[4][0] = "jumps\0";
    pangram[5][0] = "over\0";
    pangram[6][0] = "the\0";
    pangram[7][0] = "lazy\0";
    pangram[8][0] = "dog\0";

    //create writer
    return_code = OSTaskCreate(writer, NULL,
                                (void*)&writer_stk[TASK_STACKSIZE-1], WRITER_PRIO);
    alt_ucosii_check_return_code(return_code);

    //create reader
    return_code = OSTaskCreate(reader, NULL,
(void*)&reader_stk[TASK_STACKSIZE-1], READER_Prio);
    alt_uCosII_check_return_code(return_code);
}

int main (int argc, char* argv[], char* envp[])
{
    shared_buf_sem = OSSemCreate(1);//binary semaphore
    reader_writer_init();
    OSStart();
    return 0;
}

//end reader_writer.c
C.2.2. reader_writer.h

#define MAX_WORD_SIZE 6
#define WORDS_IN_BOOK 9

#define TASK_STACKSIZE 2048

#define WRITER_PRIO 16
#define READER_PRIO 15

char* pangram[WORDS_IN_BOOK][MAX_WORD_SIZE];
char* book[WORDS_IN_BOOK][MAX_WORD_SIZE];
INT8U book_mark;

//end reader_writer.h
C.3: Sleeping Barber Source Code

C.3.1. sleeping_barber.c

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include "includes.h"
#include "alt_ucesii_simple_error_check.h"
#include "sleeping_barber.h"

static char* cust_tbl[NUM_CUSTOMERS][NM_LEN];
static INT8U* cust_args[NUM_CUSTOMERS];

/* Definition of Semaphore */
OS_EVENT *barber_sem;
OS_EVENT *waiting_room_sem;

/* Definition of Task Stacks */
OS_STK cust_stks[NUM_CUSTOMERS][TASK_STACKSIZE];

INT8U sevenSegEncoder(int seven_seg_encoding){

    switch(seven_seg_encoding){
    case 0:
        return SevenSeg0;
        break;
    case 1:
        return SevenSeg1;
        break;
    case 2:
        return SevenSeg2;
        break;
    case 3:
        return SevenSeg3;
        break;
    case 4:
        return SevenSeg4;
        break;
    case 5:
        return SevenSeg5;
        break;
    case 6:
        return SevenSeg6;
        break;
    case 7:
        return SevenSeg7;
        break;
    case 8:
        return SevenSeg8;
        break;
    }
}
case 9:
    return SevenSeg9;
    break;
}
return SevenSeg0;

void printCustNameSevenSegs(int customerID){
    switch(customerID){
    case 0:
        *hex3 = SevenSegR;
        *hex2 = SevenSegO;
        *hex1 = SevenSegN;
        break;
    case 1:
        *hex3 = SevenSegB;
        *hex2 = SevenSegO;
        *hex1 = SevenSegB;
        break;
    case 2:
        *hex3 = SevenSegJ;
        *hex2 = SevenSegO;
        *hex1 = SevenSegN;
        break;
    case 3:
        *hex3 = SevenSegD;
        *hex2 = SevenSegO;
        *hex1 = SevenSegN;
        break;
    case 4:
        *hex3 = SevenSegB;
        *hex2 = SevenSegE;
        *hex1 = SevenSegN;
        break;
    case 5:
        *hex3 = SevenSegH;
        *hex2 = SevenSegA;
        *hex1 = SevenSegN;
        break;
    case 6:
        *hex3 = SevenSegJ;
        *hex2 = SevenSegA;
        *hex1 = SevenSegN;
        break;
    case 7:
        *hex3 = SevenSegD;
        *hex2 = SevenSegA;
        *hex1 = SevenSegN;
        break;
    }
}
void barber(INT8U cust_id){
  printf("*********************************************************
  ********
  
  Barber: Hello %s, get on up in the chair son\n", 
cust_tbl[cust_id][0]);
  printf("Barber: This will only take 10 seconds\n");

  int i = 9;
  for(i; i >= 0; i--){
    *hex0 = sevenSegEncoder(i);
    OSTimeDlyHMSM(0,0,1,0);
    printf(".");
  }
  printf(" Done\n");
}

void enterServiceInterval(INT8U service_interval_sec){
  OSTimeDlyHMSM(0,0,service_interval_sec,0);
}

void customer(void *pdata){
  INT8U return_code;

  INT8U *cust_id = (INT8U*)pdata;

  INT8U customer_prio = OSTCBCur->OSTCBPrio; //Obtains customer
  priority from Task Control Block

  INT8U service_interval = 45 - customer_prio;

  while(true){
    //Implement solution here
  }
}

void open_shop()
{
  INT8U return_code = OS_NO_ERR;
  INT8U CUST_PRI0 = BASE_CUST_PRI0;
  INT8U i;

  //initialize customer table
cust_tbl[RON][0] = "Ron\0";
cust_tbl[BOB][0] = "Bob\0";
cust_tbl[JON][0] = "Jon\0";
cust_tbl[DON][0] = "Don\0";
cust_tbl[BEN][0] = "Ben\0";
cust_tbl[HAN][0] = "Han\0";
cust_tbl[JAN][0] = "Jan\0";
cust_tbl[DAN][0] = "Dan\0";
//initialize customer args
//Purely educational since integers (C being pass by value)
//can be passed directly not worrying about corrupting
//stack memory.
cust_args[RON] = RON;
cust_args[BOB] = BOB;
cust_args[JON] = JON;
cust_args[DON] = DON;
cust_args[BEN] = BEN;
cust_args[HAN] = HAN;
cust_args[JAN] = JAN;
cust_args[DAN] = DAN;

printf("***********************************\n");

//create customers
for(i=0; i<NUM_CUSTOMERS; i++)
{
    return_code = OSTaskCreate(customer, (void*)&cust_args[i],
    (void*)&cust_stks[i][TASK_STACKSIZE-1], CUST_PRIO++);
    alt_ucosii_check_return_code(return_code);
    printf("Hello %s, you are customer number %d\n",
cust_tbl[i][0], i);
    printf("Priority: %d\n", CUST_PRIO);
    printf("***********************************\n");
    printCustNameSevenSegs(i);
}

int main (int argc, char* argv[], char* envp[])
{
    barber_sem = OSSemCreate(1);//Initialize barber_sem as a binary
    semaphore
    waiting_room_sem = OSSemCreate(3);//Initialize waiting_room_sem
    to allow 3 customers
    open_shop();

    OSStart();

    return 0;
}

//end sleeping_barber.c
C.3.2. sleeping_barber.h

//Make sure base addresses are correct
#define green_LEDs (INT8U *) 0x01003030
#define hex3 (char *) 0x01003040
#define hex2 (char *) 0x01003050
#define hex1 (char *) 0x01003060
#define hex0 (char *) 0x01003070
#define true 1
#define false 0
#define TASK_STACKSIZE 2048
#define NUMustomers 8
#define BASE_CUST_PRIO 7
#define NM_LEN 4
#define SevenSeg0 0x40
#define SevenSeg1 0x79
#define SevenSeg2 0x24
#define SevenSeg3 0x30
#define SevenSeg4 0x19
#define SevenSeg5 0x12
#define SevenSeg6 0x02
#define SevenSeg7 0x58
#define SevenSeg8 0x00
#define SevenSeg9 0x18
#define SevenSegA 0x08
#define SevenSegB 0x03
#define SevenSegD 0x21
#define SevenSegE 0x06
#define SevenSegH 0x09
#define SevenSegJ 0x61
#define SevenSegN 0x2B
#define SevenSegO 0x23
#define SevenSegR 0x4C
#define SevenSegDASH 0x3f
#define SevenSegOFF 0x7f
#define RON 0
#define BOB 1
#define JON 2
#define DON 3
#define BEN 4
#define HAN 5
#define JAN 6
#define DAN 7

//end sleeping_barber.h