Real-Time Embedded Systems

CMPE.663.01

Project II(b)

QNX PWM Control of Servo Motors

Christoffer Rosén & Lennard Streat

cbr4830@rit.edu, lgs8331@rit.edu

Submitted 11/18/2012

Class Section 1

Instructor: *Dr. Nathan Ransom*

Table of Contents

Areas of Focus

List of the project assignments for each team member

Christoffer Rosén

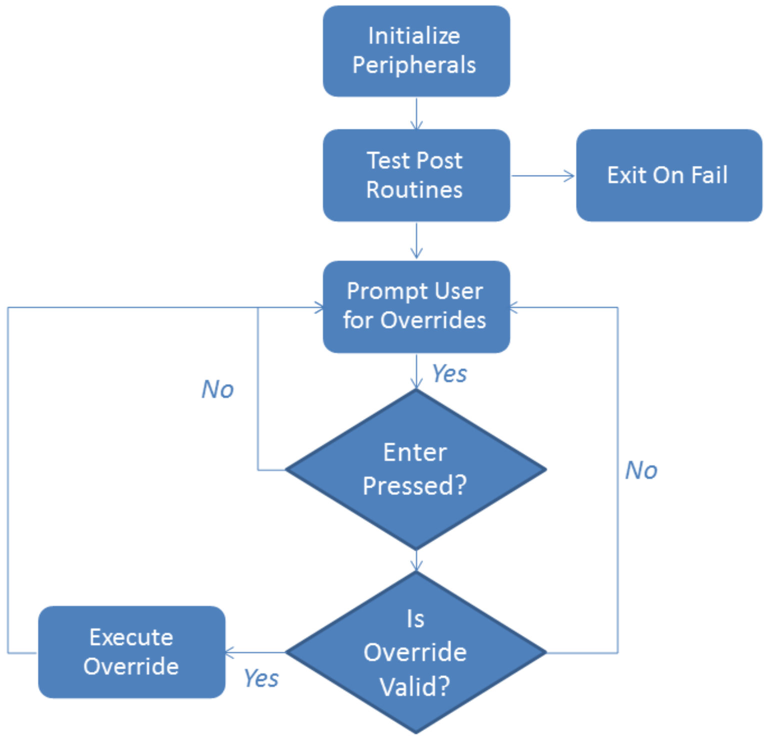
1. ***Hardware Configuration—****wired the board to specifications.*
2. ***Lead Architect-*** *Setup the structure of the program and how different portions (i.e. input and motors) interacted with each other.*
3. ***Lead Debugger—****debugged code and solved many of the issues.*
4. ***Lead Subversion Manager****—created the base directory for the project via github.*

Lennard Streat

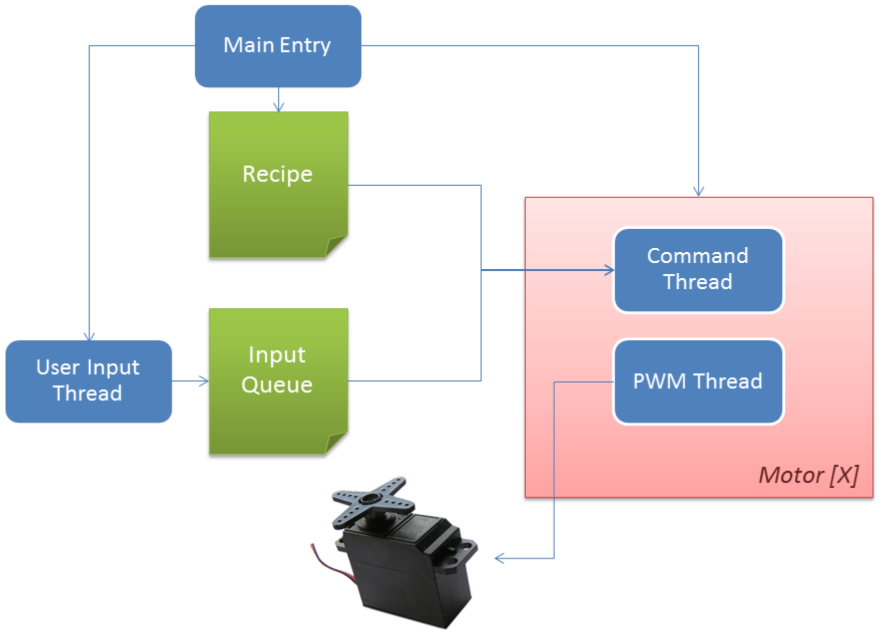
1. ***Code Review****—reviewed code to ensure that it met the project requirements. Furthermore, provided the overview and layout for the overall program. Translated requirements into a code flow.*
2. ***Tester—****Wrote test cases and extra opcode.*
3. ***Diagram Designer****—created diagrams for the project report.*
4. ***Quality Assurance—****debugged code and solved many of the issues related to hardware configuration.*

Analysis & Design

Include an outline of the approach taken; block diagrams of the hardware and software, and supplementary diagrams to illustrate the required behavior.



Figure—1a. *High-level software design flow (main execution loop).*



Figure—1b. *High level overview of the thread structure.*

1. Requirements Analysis:
    a. *The requirements for this investigation were determined and then incorporated into the design.*
2. Project Mock-up:
    a. *A rough layout of the anticipated design was created on paper—this design was later modified.*
3. Initialization of Peripherals:
    a. *Determined the appropriate initialization parameters for the timer to execute properly.*
4. Created Program
    a. *The program was coded and designed in several steps—first the proper usage of the timer, then the proper usage of the servos. Lastly, the delay system and instruction decoder were created.*
5. Debug & Verification:
    a. *After the initial phase of creating the program, extensive testing procedures were implemented to ensure that the system functioned properly. Most "tests" were coded into constant arrays.*
6. Project Reflection
    a. *Wrote a write-up containing all of the aforementioned process and steps taken to complete the project.*

Test Plan

Include a description of your POST. Include a list of the tests and expected results.

Description of POST Method

Two power-on self-test routines were utilized for this investigation. The first was the timer post routine, and the second was the Pulse Width Modulator (PWM) post routine. The latter ensured that the PWM was properly functional in two ways: through inspection and by verifying that the internal counter for the PWM was active. For the former case, the servos were turned to its two extremes. This verified that the servo was properly functional, and that the delay capabilities were properly implemented.

Since the inspective case could not be verified in code, the output for the PWM post routine programmatically depended solely on the state of the internal counter. The inspective case was solely for the benefit of the developers.

Expected Results

1. Boot up the program—default states. This should result in the servos executing the POST routine (the servo POST routine will be visually noticeable on the servos). Furthermore, the servos should be in the paused state after the execution of the POST, and the command line should be actively waiting for the user to enter overrides.
2. A one second delay should be executed and timed against an external timer to verify that the timer delay has been properly calibrated. This was made visible by having one move command in the front and behind the delay command, with an argument of 10. For instance, **MOV+0, DELAY+10, MOV+5**; the aforementioned combination should make estimating the period of the timer possible.
3. An execution of the default recipe (provided in the project requirements). This should result in the proper visually verifiable output on the servos of the code sequence. This also demonstrates the operation of a normal/valid recipe, as per the requirements.

4. Execute a program with a **Nested Loop Error** (NLE) situation (two nested loops) followed by a valid command. This should result in the NLE status bit being set for the servo—thus being displayed on the output LEDs. The continue command should not function.
5. Execute a program with an invalid input for the MOV command (e.g. MOV+6) and a valid command following the invalid command. This should result in the **Recipe Command Error** (RCE) status bit being set for the servo—thus being displayed on the output LEDs. The continue command should not function.
6. Proper reset after an error. This situation should result in the LED statuses being cleared—thus all LEDs should be turned off.
7. Typing the pause command on the command line. The recipe will enter the pause state if it was executing. Otherwise, the servo should retain its current state.
8. Typing the continue command on the command line. If the recipe was paused, this should continue the execution of the recipe from the point in the recipe that it last left off at. Otherwise, the servo should not respond. Also, the LED status must respond accordingly—all LEDs turned off.
9. Typing the left/right command on the command line. If the servo is paused, this should result in the servo turning to the respective adjacent position. Otherwise, the servo will not respond.
10. Attempt to move the servo "***out of bounds***" with override statements. This command should not result in the servo attempting to move—the servo should remain in its position.
11. Typing the restart command. This command will respond by clearing all the status bits (displayed on the LEDs) and jumping to the beginning of the execution block for the code and executing the recipe.
12. **Executing the added opcode**. This opcode will result in the mirroring of the servo—for instance, if the servo was located at position 0, the servo will move to 0. If it was at 1 it will move to 4; from 2, to 3, and vice versa.
13. Executing the no-op command. This command should result in nothing happening

within the code—no values should change, including the states for the servos—displayed on the LEDs.

14. Testing the PWM period. This was done to optimize the operation of the servos—the PWM was set to be within the ideal range for the servos. The servos were designed to operate within the kHz range.
15. Execute a recipe that moves between all positions from position 0 to position 1. This should result in a ticking-clock motion.
16. Execute a recipe with a **RECIPE_END** command followed by another command—this demonstrates that the recipe continue command works properly. The command following the recipe end should not be executed.
17. Use overrides to execute different combinations of commands on the paused servos. This results in the servos moving independently of one another and demonstrates that the functionality of the servos is not coupled.
18. Use different recipes for each of the servos. This should demonstrate that the functionality of the servos is not coupled.
19. Note: *Servos should be started in the 0 position.*

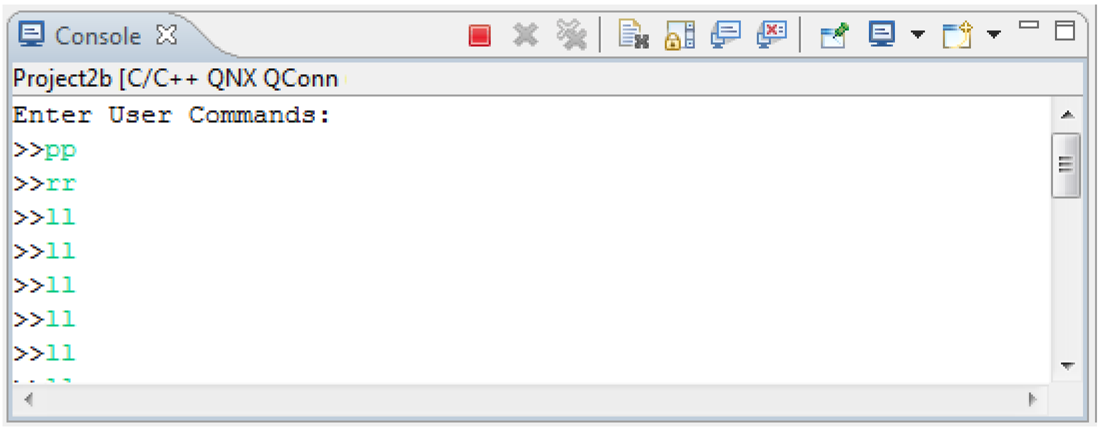
Lessons Learned

Include a brief summary of what was learned in this project and a brief description of any difficulties encountered.

Several lessons concerning multi-threading were learned. Particularly, the division of concerns amongst threads, modularity, and timer event signaling. A timer was utilized that would signal updates to the servo motor on predefined intervals, to simulate the functionality of a PWM module. Initially, we tried to get this to work through only using nanospin. We would nanospin for the required uptime, then nanospin for the required downtime (period - uptime) in a loop. However, these caused considerable jitter. We learned that the timer made it considerably better, which is likely because it is much more accurate in always sending a pulse every 20ms when the servos is expecting a signal than through the use of nanonspins. This investigation also provided an opportunity to gain a more thorough understanding of timing systems for QNX. Issues with bit-banging were reinforced.

Proof of Operation

Include appropriate output or screen shots of the user interface with representative results.



```
Enter User Commands:
>>pp
>>rr
>>ll
>>ll
>>ll
>>ll
>>ll
```

Figure—1: *A simulation of a proper execution of the program.*

Source Code

This is a listing of source code and test code.

See attached file

1. **Project2b.cpp—***The main entry point for the program; creates the pointers to the memory mapped io for the GPIO port and initializes the motors and input listener.*
2. **Inputs.cpp—***This is the input listener. It creates a thread that listens to user input and places them into the proper motor input queue.*
3. **Inputs.h—***The header for the Inputs.cpp file.*
4. **Motor.cpp—***The motor class represents the servos motor. It contains two threads: 1) One is responsible for executing recipes and user input commands. It appropriately controls how long the uptime is required for the next time signal is sent. 2) The second thread generates the signal, or PWM, to the motor. It sends the pulse width uptime required to move the servos motor to required position every 20ms which is manipulated by thread 1. This is done by setting 1 on the digital IO board for the required uptime to set position of the motor and then setting it back to zero* ***every*** *20 milliseconds through the use of a timer.*
5. **Motor.h—***The header for the Motor.cpp file.*