SWEN 563/CMPE 663/EEEE 663
REAL-TIME & EMBEDDED SYSTEMS
ROCHESTER INSTITUTE OF TECHNOLOGY

PROJECT-2b: CONTROL OF TWO SERVO MOTORS BY INDEPENDENT RECIPES ON QNX

Submitted to: Larry Kiser(llkiee@rit.edu)

Submitted by:

Nicolas Delanou (nxd7208@rit.edu)

D.O.S: 12/05/2016

Areas of Focus

Writing of the code:

- Nicolas Delanou: 100%

This is my personal report. We agreed with my teammate, Mokshan Shettigar, that we will both submit independent reports and code for this project as he never showed up in class for this project.

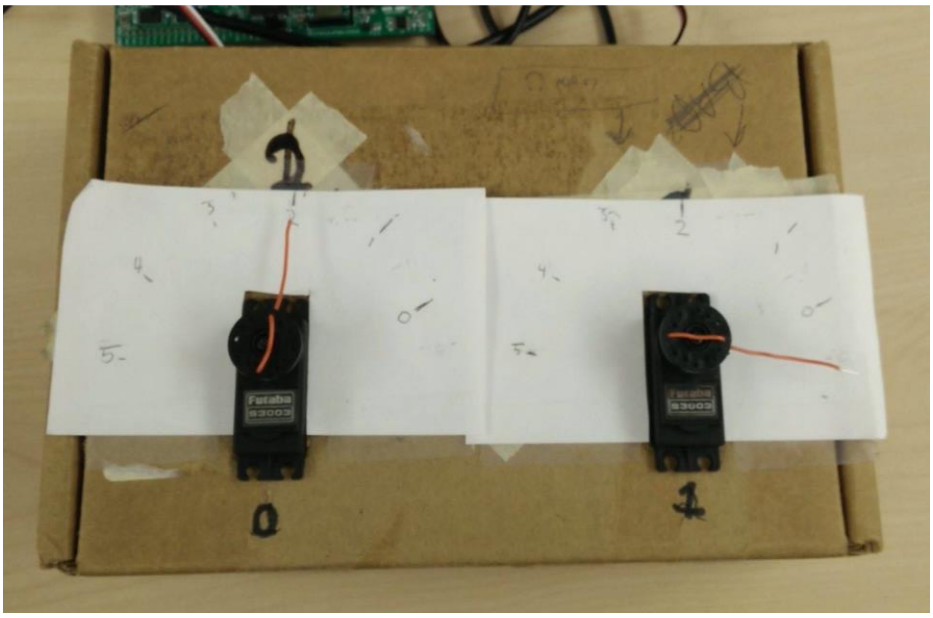
Analysis / Design

In this project, we are asked to adapt the Project 2a, developed for STM32 boards, for QNX systems.

For that, we will adapt the code (C language) in order to run on the QNX “purple box” from the lab. We will use the QNX Momentics IDE.

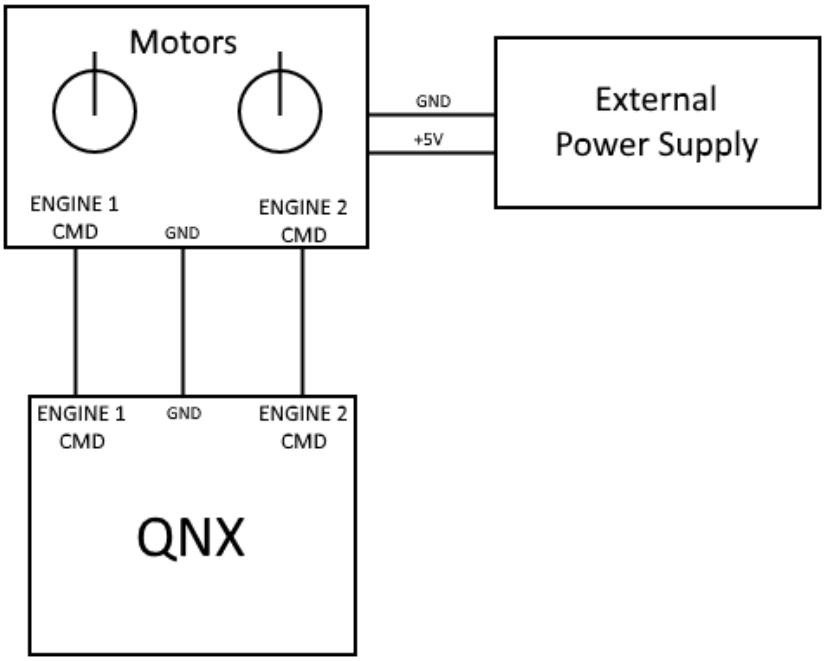
The goal of this project is to control independently two servo motors S3003 with PWM. The PWM are independently generated by the QNX board on PORT A and PORT B. Both motors have to follow different recipes and can be commanded by a user interface on the QNX system using command line.

Here is a picture of the servo motors:



Test Plan

Hardware diagram



PWM generation

The first part of the project has been the generation of a correct PWM signal. For that, we created a thread function, named "*Generate_PMW*", taking as parameter an engine structure.

The function launched by the thread creates a timer that will be called every 20ms, in order to have a correct period for the signal. The code called every 20ms is the following:

```
pthread_mutex_lock(&mutex_writingPWM);
out8(data_handler, 0xFF);
nanospin(upTime);
out8(data_handler, 0);
pthread_mutex_unlock(&mutex_writingPWM);
```

In this code, we are using a mutex to protect the signal writing. We decided to use this trick in order to have a more stable signal and this was very effective. After the mutex has been lock, we set the signal to the high level on the right port. The port used is defined by the "*data_handler*", which is an attribute of the engine passed as parameter. Then, the program waits for "*upTime*", which is a "*timespec*" structure and also a parameter of the engine. This value determines the time spent on high level by the signal, and therefore the position of the motor. After setting the signal back to low level, we release the mutex.

Position	Angle (°)	Up Time (ns)	PWM high level (%)
Position 0	0	600000	3
Position 1	36	900000	4.5
Position 2	72	1200000	6
Position 3	108	1500000	7.5
Position 4	144	1800000	9
Position 5	180	2400000	12

In order to generate two PWMs, we create two threads as shown in the code below:

```
int ret = 0;
ret += pthread_create( &thread_engine_1, NULL, Generate_PWM, &eng1);
ret += pthread_create( &thread_engine_2, NULL, Generate_PWM, &eng2);
```

We see that the last parameter passed to this function is the address of the engine structure.

Command management

Servos are supposed to follow recipes given by the user. Those recipes are implemented as an array of consecutive instructions. Every instruction is a byte (8 bits). We are using the variable type “uint8_t” to store the value of an instruction.

The 3 most significant bits of an instruction represent the opcode of the instruction. The 5 other bits are used to as parameters. Here is a list of all the instruction implemented in this project:

Instruction name	Opcode	Purpose	Range of the argument
MOV	001	Move the servo to position X	0..5
WAIT	010	Wait for X*100ms	0..31
LOOP_START	100	Start of the loop. X repetition	0..31
LOOP_END	101	End of the loop	
RECIPE_END	000	End of the recipe	
Additional instructions (graduated student)			
JUMP	110	Jump to the instruction number X	0..31
SYNC	101	Wait for both recipes to be in Synching state to start both recipe again.	

We then create a new function that will be called in a thread called “Update_Recipe”. The main purpose of this thread is to follow the recipes independently for each motor. This thread creates a timer that will call the correct functions every 100ms in order to update the state of the recipe.

We change the structure from the Project 2a in order to store more information like the *data_handler* or the *upTime*.

```
typedef struct Engine Engine;
struct Engine
{
    unsigned char recipe[100];
    int wait_cpt;
    int index_recipe;
    int index_loop;
    int start_loop_index;
    int current_postion;
    int recipe_ended;
    enum Channel chan;
    int in_Pause;
    int in_Sync;
    struct timespec upTime;
    uintptr_t data_handler;
};
```

User Terminal

We use the QNX Momentic IDE Console in order to communicate with the user. The user can enter a two-character command that allow him to control the engines independently. This feature is handled in the main function of the program.

Here is all the available command:

Character	Command
P or p	Pause the running recipe
C or c	Continue the recipe
R or r	If recipe stopped, move the engine one position to the left
L or l	If recipe stopped, move the engine one position to the left
N or n	Nothing changes for this engine
B or b	Restart the recipe

Here are some command examples:

- *pc* : pause engine’s 1 recipe and continue engine’s 2 recipe
- *Rb* : Move engine 1 one position to the right if his recipe is stopped and restart engine 2 recipe
- *nB* : Nothing happened for engine 1 and restart engine’s 2 recipe

Project Results

After verifying with the professor that the PWMs generated were correct, we plugged the PORT A and PORT B to the servos.

At first, motor was following the recipe but the PWM generated was not really stable. We then decided to had a mutex to protect the PWM writing and this solved our problem. The writing of the two PWMs was shifted for the second one, but we kept the 20ms period on both signals so this was not a problem.

All the features from project 2a are working correctly, even the two additional opcodes.

The servo motors can be commanded independently.

Comparing and contrasting

The main difference between the two project is the fact that this one is using a real-time operating system. This can be a good point because it is easier to give a specific schedule or a specific priority to a task. However, it is less predictable because of the operating system running in background that can preempt the code execution at some points.

The way PWM are generated on bare-metal programming, with timers, is a big plus because you don't use CPU for the signal generation once it's launched. You only consume CPU when you want to change the signal period.

In QNX, we had to create two different threads for that. This is not a problem because the QNX board we are using is running at 80Mhz, but it will have been even more effective if we were using a multi-core system.

Lessons learned

During this project, we learned to adapt code from a system to another. This is not an easy thing, especially when you convert a bare-metal project into a program running on a real-time operating system.

In my opinion, the hardest thing on real-time system is to generate a stable output because of the operating system running in background.

This is the first time I have to adapt a project like that and I learned to be extremely methodic during this process in order to don’t forget anything.