Real-Time and Embedded Systems

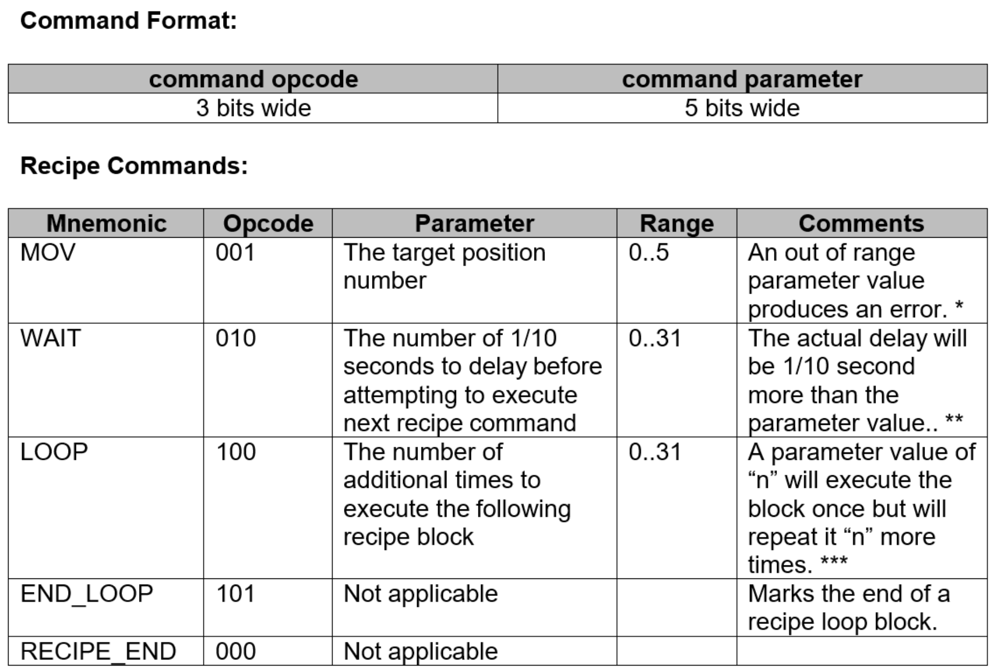
Project 02: Servo Control

Author: Liam Sullivan
Email: lds7381@rit.edu
Submission Date: 10/03/2022

Analysis and Design
This laboratory required two servo motors connected to the STM32 microcontroller which would be controlled to move to 6 separate degrees. Recipes containing bit strings that represented instructions giving an op code and a parameter were used to control these motor's positions. Each motor was given their own recipe to follow and the two servo motor's movements were independent of each other. While a servo was performing a recipe it also needed to be able to be paused. During this pause state the servo motor also needed to be able to take instructions outside of the recipe such as move right or left one position. The recipe could also then be continued any time and restart where the recipe had left off. To enable control of the servo to work with recipes like this many different peripherals on the STM32 microcontroller were used.

One of the most important peripherals used on the microcontroller was UART. UART enabled the microcontroller to take user's commands as inputs to tell the servo what to do as well as prompt the user. Two letters corresponding to commands are pressed and then the program will check for the enter key press to execute the commands.

Another peripheral used on the board was the general purpose timers. This included TIM2, TIM3, and TIM4. TIM2 and TIM3 were used to generate and output a Pulse Width Modulation (PWM) singal to the servo. These timers were given a prescaler of 79 to make the clock run at 1MHz or count every one microsecond. To create the PWM output from the timers, the clock's period had to be set, this was done by setting the ARR register to 19999 or a 20 millisecond period. Next, the CCR1 register was used to change the duty cycle of the PWM signal. This duty cycle was from 0.4 ms to 2.0ms. This range gave the servo the full range of movement 0-160 degrees. From this range 6 distinct positions equidistant from each other were chosen to be the six positions. The respective duty cycles were then given to the CCR1 register to change the position of the servo. The final timer that was used was TIM4. TIM4 was used to be the base timer for getting a new command in the servos recipe. All of the opcodes and how the parameters are used can be seen in Figure 1.



Command Format:

command opcode	command parameter
3 bits wide	5 bits wide

Recipe Commands:

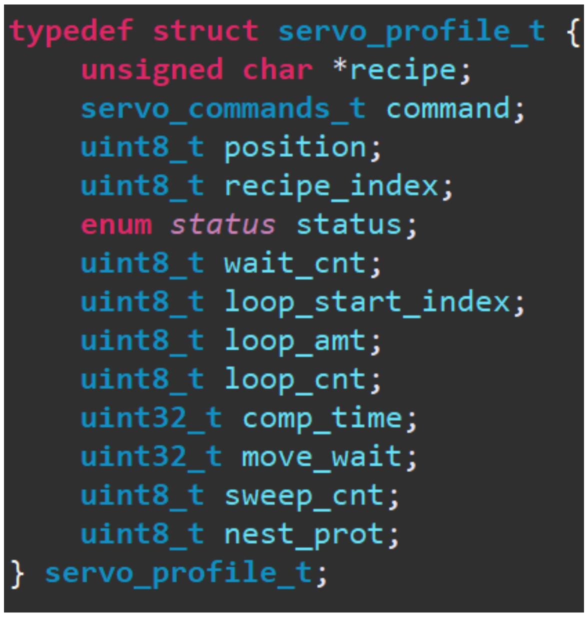
Mnemonic	Opcode	Parameter	Range	Comments
MOV	001	The target position number	0..5	An out of range parameter value produces an error. *
WAIT	010	The number of 1/10 seconds to delay before attempting to execute next recipe command	0..31	The actual delay will be 1/10 second more than the parameter value.. **
LOOP	100	The number of additional times to execute the following recipe block	0..31	A parameter value of “n” will execute the block once but will repeat it “n” more times. ***
END_LOOP	101	Not applicable		Marks the end of a recipe loop block.
RECIPE_END	000	Not applicable		

Figure 1: Command Opcodes and Parameter Usage

In addition to these commands, another command, SWEEP, was added. This opcode was 011 and took in a parameter 0-31. The command would “sweep” from position 0 to 5 and back to 0 the amount of times given by the parameter. The prescaler to get these commands for this timer was set to 7999 making the count for the servo 10KHz or once every 100 microseconds. This allowed for the program to compare the difference in time to 1000, which would be 100ms, and get the next instruction in the recipe.

The final peripherals used were the LEDs on the RTOS shield. These LEDs were used to indicate the recipes status of the first servo being controlled. This was done with the D3 and D4 LEDs on the shield. For the case of the recipe running the only light on was D3. If the recipe was paused then none of the lights were on. If there was a command error from the opcode then only the D4 LED would be on. Finally, if there was a nested loop error then both of the LEDs would come on. This allowed for the user to see the real-time status of the first servo when a recipe was running on it.

With all of these peripherals set up this way, the program could be written. The biggest challenge in writing this program was that everything had to be non-blocking as well as independent of one another. The first step of this to make the program non-block was to have the while loop in the main of the program be as small as possible so that each function could be run as fast as possible to check to see if they needed to be performed. This main would only call three methods, *get_servo_process(), update_servos(),* and *update_leds().* All of the methods used a created servo profile struct that contained all of the servos information. This struct can be seen below in Figure 2..



```c
typedef struct servo_profile_t {
    unsigned char *recipe;
    servo_commands_t command;
    uint8_t position;
    uint8_t recipe_index;
    enum status status;
    uint8_t wait_cnt;
    uint8_t loop_start_index;
    uint8_t loop_amt;
    uint8_t loop_cnt;
    uint32_t comp_time;
    uint32_t move_wait;
    uint8_t sweep_cnt;
    uint8_t nest_prot;
} servo_profile_t;
```

Figure 2: Servo Profile Struct

As can be seen this struct contains all of the profile's information including its recipe, command, position and status. As well as some of its variables used to ensure that the execution is independent of the other servo (such as the loop variables and the time comparator). A setup function to initialize each of these structs to have the correct starting values was used as well prior to the main loop. Each function used its obtained information or the pointer of this struct to set these variables.

The first method *get_servo_process()* used the UART peripheral to check to see if a user had entered any commands to the servo and would set the commands to the corresponding servo profile.

The second method *update_servos* would check the command of the servo with a switch statement. The servo would then perform one of the following commands: begin, pause, continue, left, right, and no-op. The begin command would start the execution of the recipe and everytime the *update_servo()* method is being called, if the command remains, it checks TIM4 to see if the time after the last execution is greater than 100ms and if it is, executes the next command. Another wait was also added to this time comparison, this was the move wait. This wait was the time it took to move the servo from one position to another. This was calculated by finding the absolute value of the difference in the positions and multiplying it by 75 ms, or the found time it takes to go from one position to another. The execution of this recipe can also be paused and continued in this function when given the commands. A right and left one position was also added as commands and could be used any time the servo was not actively executing the recipe (paused). A no-op instruction was also allowed to make it so that only one servo could be controlled at a time as well. This function also was used to set the status of the servo which would be used by the next function, *update_leds()*.

update_leds() Was a function that used the current status of the first servo motor and would change the LEDs D3 and D4 like described above to match the current status of the servo.

With all of these functions and peripherals coded and functional the overall program was a non-blocking program that could take inputs in to control the servo at any moment. Designing the overall program this way was imperative to having the complete functionality required by this exercise.

Test Plan

The first part of testing this program to work was finding the correct duty cycles for the 6 separate positions on the servo. As described in the Analysis and Design section the duty cycle of the 20ms PWM signal was 0.4 ms to 2ms, this corresponds to a CCR1 register value of 400-2000. These values were then split into 6 equidistant duty cycles and tested. The resulting servo positions were good except for the 1st position which was too far to the right and the gearbox of the servo would strain. The minimum value that was found that would not cause this straining was 470 or .47ms and the position 0 was set to this and the new position values calculated.

The second test that had to be performed was on the servo's movement time. When all of the functions were first being created the movement time was not included and if the servo was to move to position 0 to 5 then back to 0 it would just stay at position 0. So different timing values were tested to be added to wait until the servo gets to the position prior to moving again. The value found that best fit this was 75ms per position being moved. This allowed the servo plenty of time to get to whatever position it needed to get to before executing its next command.

Finally once all of the peripherals and servos were set up the demo code recipe was tested. This recipe was the perfect test for testing since it used all of the opcodes that are available while testing the parameter functionality of these opcodes. For example, the servo on start attempts to go position 0 to 5 then back to 0. This is a test for the movement time for the servo to get to the next position. There was also a LOOP command which had a parameter of 0, testing the functionality of the loop parameter and making sure that it doesn't get performed twice. This demo recipe also demonstrates the WAIT command making sure that the 3 WAIT commands with a max parameter waits for approximately 9.3 seconds. A custom SWEEP opcode was also added into the end to test the custom opcode functionality. This test recipe goes over many of the different commands that could cause issues in a recipe. A full successful execution of all of these commands in this demo recipe would show that any correct recipe given to the servo will be able to be run.

Another test to make sure that the servos would run independently of each other was the use of the right and left commands. Giving one command to only one of the servos and only that servo moving shows that the servos can function independently of one another regardless of the recipe or status.

Project Results

The resulting program with all of the design decisions above and using what was found during testing created a robust and independent control for two separate servos. Each one of these servos were given a recipe to perform. The first servo executed the demo recipe and the second executed another custom recipe that just performed a bunch of MOVE commands, some of which contained in a loop. When performing the recipe on both of the servo many aspects were tested to make sure that the program was executing successfully.

One of these aspects was independent servo control. This was seen to be correct by performing a beginning on both recipes, then followed by a pause on servo two. This kept servo one following its recipe. Then a move right position was given to servo two while servo one was still running. This servo moved its position one to the right. All of these commands on the servo whilst it is in the middle of the recipes proved that the servos had no issues in performing independently.

Another aspect of the program was the correct use of the recipe's bit strings that were the commands in the recipe. The demo recipe used all of these premade opcodes and parameters described in Figure 1. When this demo recipe was executed the servo had no issues. The bit string commands were able to be decoded properly and control the servo in the way that was desired. This meant that any correct recipe given to the servo would be able to run without issue. However, if there was an error in the recipe the program would also recognize this. The two errors that the program looked for in the recipe were unknown opcode and nested loop errors. Both of these errors can occur if the recipe is not set up properly, and the execution of the recipe would stop and update the status of the servo.

The final aspect was the LED lights as an indicator to display the current status of the first servo. To ensure that the LEDs were working correctly two recipes were made. One that contained a nested loop error and one with an opcode error. These both successfully updated the LEDs to show that an error had occurred as well as having the run and pause statuses being displayed.

Lessons Learned

One lesson that was learned during this exercise was about dealing with the duty cycle percentage being sent to the servo motor. The duty cycle range of the PWM period is 20ms, and the duty cycle for the servo was expected to be between 0.4ms and 2ms. However, when going about making these PWM duty cycles and testing them, it was discovered that the 0.4ms duty cycle tried to go past its lower limit and would cause strain on the gearbox in the servo. It was found that the minimum duty cycle to be sent to the servo without this occurring was 0.47ms and this was used throughout the program. From this it was learned that even though a specification may say it can handle that duty cycle, it is best to test and ensure that the servo will not break with the 0.4ms and find the minimum that that specific servo can run at.

Non-blocking code was an essential part of this program, it allowed for the servos to be controlled independently without having to wait on any part of the program. Without this non-blocking code, the servo would not be able to pause whenever wanted or separately control the servos. The coding structure to make everything non-blocking seems to be really important in embedded systems so that no inputs are missed. Forcing myself to code this in a non-blocking way led to a different way than I am used to programming but I very much enjoyed programming the servo this way and see the clear advantages that it provides.

Again, in this lab timers were quite important and this lab further shows how versatile timers on the STM32 do many functions. Without these versatile timers these projects would not be possible. Getting more familiar with how these timers work is great and will lead to future success in future exercises.