Quinn Leydon

Dominic Merchelski

**Problem Statement**

**Overview**

The goal of this lab was to control the position of two SG90 Micro Servos using a 100ms loop on an MSP32 Nucleo. The loop must complete a cycle every 100ms regardless of the commands within the loop. The system has two objectives, move to position and wait. Two LEDs display the state of the system. The commands to wait or move can be chosen through a prewritten list of 8-bit op codes or through a command from UART input.

**Recipe**

The recipe is a series of one-byte (8bit) commands. The chart below depicts the recipe command format and recipe commands.

|  |  |
| --- | --- |
| **command opcode** | **command parameter** |
| 3 bits wide | 5 bits wide |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |  |  |

The D1 and D2 LEDs as described below indicate status.

## Recipe Running (only D1 on)

## Recipe Paused (all off)

## Recipe Command Error (D2 on)

## Recipe Nested Loop Error (D1 and D2 both on)

**User Commands:**

User commands are entered via system console keyboard and transferred to the board via UART. Each command is 2 characters followed by a carriage return. The two letters are commands for motor 1 and motor two, respectively. If “X” or “x” is entered, the entire override command is canceled. The command details are shown below.

User Command Details:

|  |  |  |
| --- | --- | --- |
| **Command** | **Mnemonic** | **Comments** |
| Pause Recipe execution | P or p | Not operative after recipe end or error |
| Continue Recipe execution | C or c | Not operative after recipe end or error |
| Move 1 position to the right if possible | R or r | Not operative if recipe isn’t paused or at extreme right position |
| Move 1 position to the left if possible | L or l | Not operative if recipe isn’t paused or at extreme left position |
| No-op no new override entered for selected servo | N or n |  |
| Begin or Restart the recipe | B or b | Starts the recipe’s execution immediately |

**Solution:**

**Timming:**

The servos require 200ms PWM loops. To control two motors two outputs are necessary. The main loop is 100ms. Our solution was to use two timers. All internal clocks were set to 16MHz. Timer 2 was used for the main loop. This had a prescaler of 15999+1 and a period of 100 counts, resulting in a 100ms period. After the 100ms, the ARR2 interrupt is triggered. This is handled by the HAL\_TIM\_PeriodElapsedCallback, where a timing flag is set to 1, allowing the while loop to reset. Timer 3 has channels 1 and 2 set to PWM Generation to control motors 1 and two. The prescaler is set to 159 + 1 and the period is 2000, resulting in a 200ms period. The PWM is controlled by CCR1 and CCR2 for each motor.

**Servos:**

According to the datasheet “Position "0" (1.5 ms pulse) is middle, "90" (~2ms pulse) is all the way to the right, "-90" (~1 ms pulse) is all the way to the left.” “Operating speed: 0.1 s/60 degree” To achieve 6 positions with a maximum range of ~160 degrees, the difference in pulse time between two positions is 0.18ms, or about 32.4 degrees per shift. The datasheet reports “ Operating speed: 0.1 s/60 degree”, which is about 54ms per shift. the count set in the ccrx register is 110+18\*difference.

**while Loop**

A white screen with black text

Description automatically generated

**process\_timestep:**This is processed every 100ms and accounts for what happened in the previous 100ms. The arguments are a pointer to the struct containing servo information, as shown in Figure 1 and a pointer to the recipe for that servo.

A screenshot of a computer program

Description automatically generated

Figure 1: SERVO Struct.

If the servo state is state\_moving, the count is decremented. Some movements of the servo take longer than 100ms. This counter is used to delay the program while the servo is moving. If the servo state is moving and the count is zero then the servo\_state is set to state\_at position, and the recipe index is incremented and the recipe is processed.

If servo state is state\_at\_position, then the recipe is processed. This could have resulted from a pause and continue from the user.

If servo state is state\_waiting and the status is not paused decrement the wait count. If count now equals zero, then set state to state\_at\_position, increase index, and process next recipe.

If servo state is state\_recipe\_ended, then the status is set to pause, and the index is reset.

After the state is delt with, the LEDS are set using process\_status\_LED, so they follow the guidelines.

**process\_recipe**

This function first validates that the servo status is running. If it is not running, then the recipe is not processed. The 3 most significant bits are read as op codes and launch the respective function. For example if op code is 001, the function is start\_move(SERVO, recipe[SERVO->INDEX] & 0b00011111). For the graduate opcodes, one will print the index of the servo, and one will print the uart command options.

**start\_move:**

If the commanded position is not a valid entry, the servo status is set to status\_command\_error. If valid, the goal position is set to the commanded position. The difference is found between the goal position and the current position, and the count is set according to the time necessary to move. Finally the servo state is set to state\_moving, and the timer3 ccrx is set to the correct position.

**start\_loop:**

The start loop function checks if the servo is currently in a loop. If it is, there is a nested loop error. If not, it records that it is in a loop and the index of the start of the loop. It sets a counter for the amount it should count and increments the index.

**end\_loop:**

If the count does not equal zero, decrement count and move servo Index to one after the loop starts. If the count is zero, the index is incremented, and the loop flag is set to 0.

**read\_user\_command**

When the user types in the keyboard an interrupt is triggered. This reads the character from the UART buffer and stores it in the RX\_command buffer. If the character is an X or x, the buffer is cleared, and a newline is sent. If it is a valid character, the character is repeated to the user.

The read\_user\_command function reads the Rx\_command buffer. If the command is not complete, it does nothing. If it is invalid, it reports to the user that the buffer is invalid, and that the user should try again. If the commands are valid, then command one is sent to process\_user\_command along with the pointer to the struct containing servo1 information. This is repeated for Servo 2. Process\_user\_command validates then performs the command. For example, if ‘P’ was entered, the function checks that the servo is not in an error state, and the recipe has not ended. After the command has been validated, the servo status is set to paused and the status LED is set to paused.

**Lessons Learned:**

1. PID control using CCRX register.
2. How to control servo position and measure time to move to each position.
3. How to use a real time system where a loop is a specific amount of time.

**Potential Improvement:**

There is redundancy in our code. We keep track of positions which are not used. We update the LEDs multiple times per timestep. There are unnecessary variables within the servo struct.