FreeRTOS, Timers & Watchdog Challenge

The Challenge

The aim of this challenge is to create a system of PWN signals, which can be controlled by either a Manual Control Panel or the Serial Bus.

Requirements

- Make use of an MCP (Manual Control Panel)
    - LEDs & Buttons
- Make use of Serial Communication
    - UART
- PWM Input & Output
- At least one timer
- Responsive

Chosen Hardware

Hardware	Description
STM32F404XE	Microcontroller
2x Buttons	Manual Control Panel
2x LEDs	Manual Control Panel
1x Servo	PWM Input

Design

Timers

I have made a general purpose timer class, which includes all the types of timers needed within the course, this includes:

- Time Base
- PWM Input
- PWM Output
- Counting Pulse

This class targets 3 General Purpose timers, 2/3/4, as these are the only timers which have 4 channels, which is required for the PWM Input/Output. For this challenge, I'll be using PWM Input and PWM Output.

Servo PWM

The control signal requires a 5V PWM signal, Additionally, it expects the duty cycle to around 1280 microseconds, or 1.280ms. This means for the purposes of PWMOutput, we need to configure it as such:

```
  const int PWM_PSC = 72;
  const int PWM_ARR = 200;
  BasicTimerPackage basicTimerPackageOutput = {PWM_PSC, PWM_ARR,
TimerBit::TIMER2};
  PWMOutputPackage pwmOutputPackage = {basicTimerPackageOutput, 1,
CC_ChannelType::CC_CHANNELTYPE_PWMOutput, OCM_Type::OCM_TYPE_PWM1, 1280};
```

For the Feedback signal, a 3.3v signal at 910Hz is expected. in my experience during the Proof of Concept stage, I can use the same PSC and ARR.

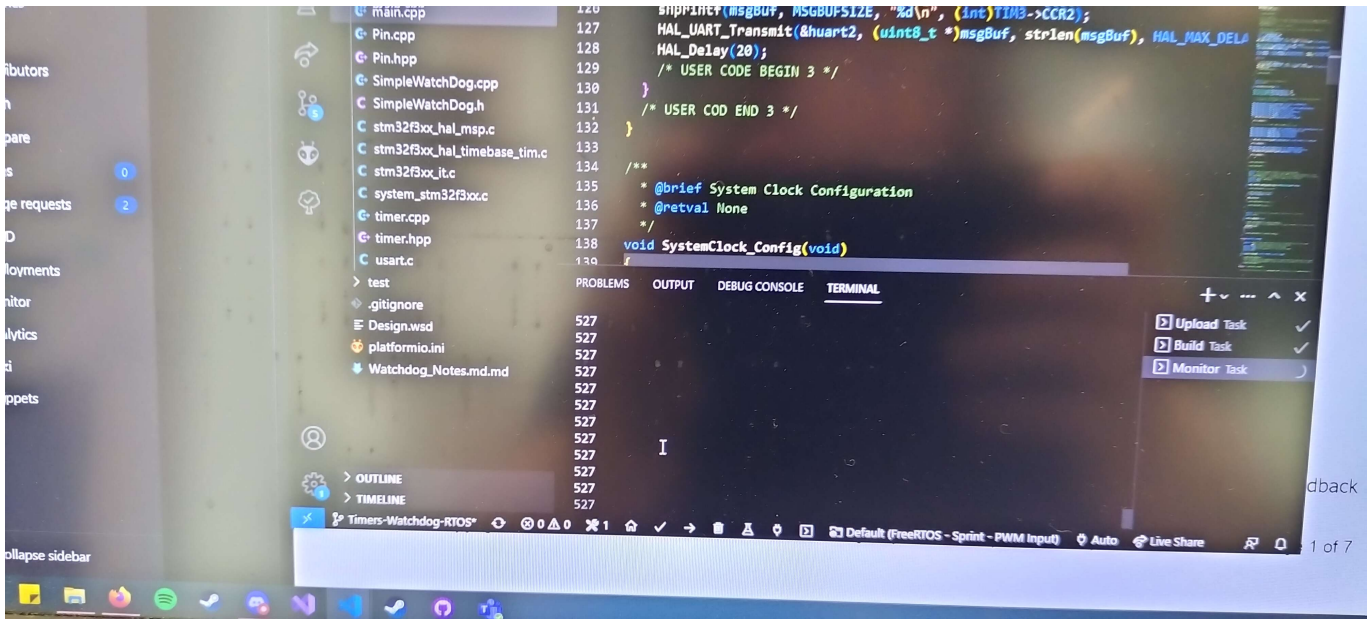
Alternate Functions

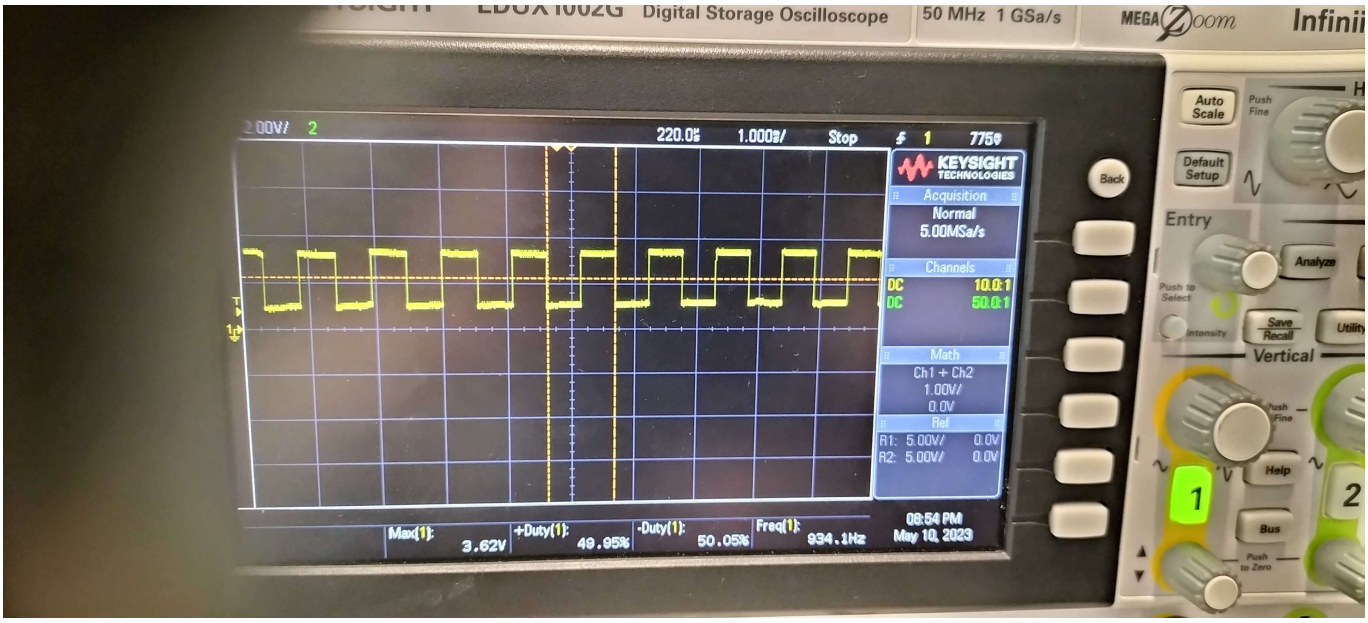
For the PWM input, I'll be using the Servo as an input device. I'll be taking the data from the servo, and relaying it to the user via UART.

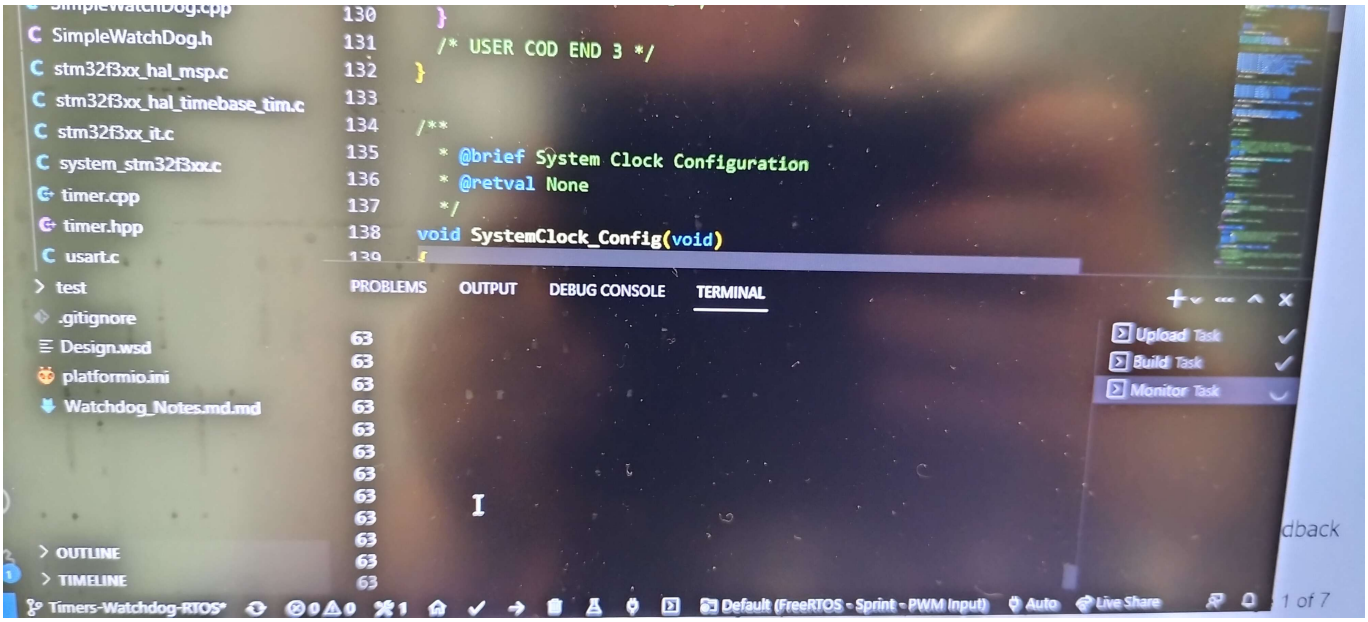
```
  // PIN B4 (D5) -> TIM3 INPUT
  GPIOB->MODER = (GPIOB->MODER & ~GPIO_MODER_MODER4) | (0b10 <<
GPIO_MODER_MODER4_Pos);
  GPIOB->AFR[0] = (GPIOB->AFR[0] & ~GPIO_AFRL_AFRL4) | (0B0010 <<
GPIO_AFRL_AFRL4_Pos);
```

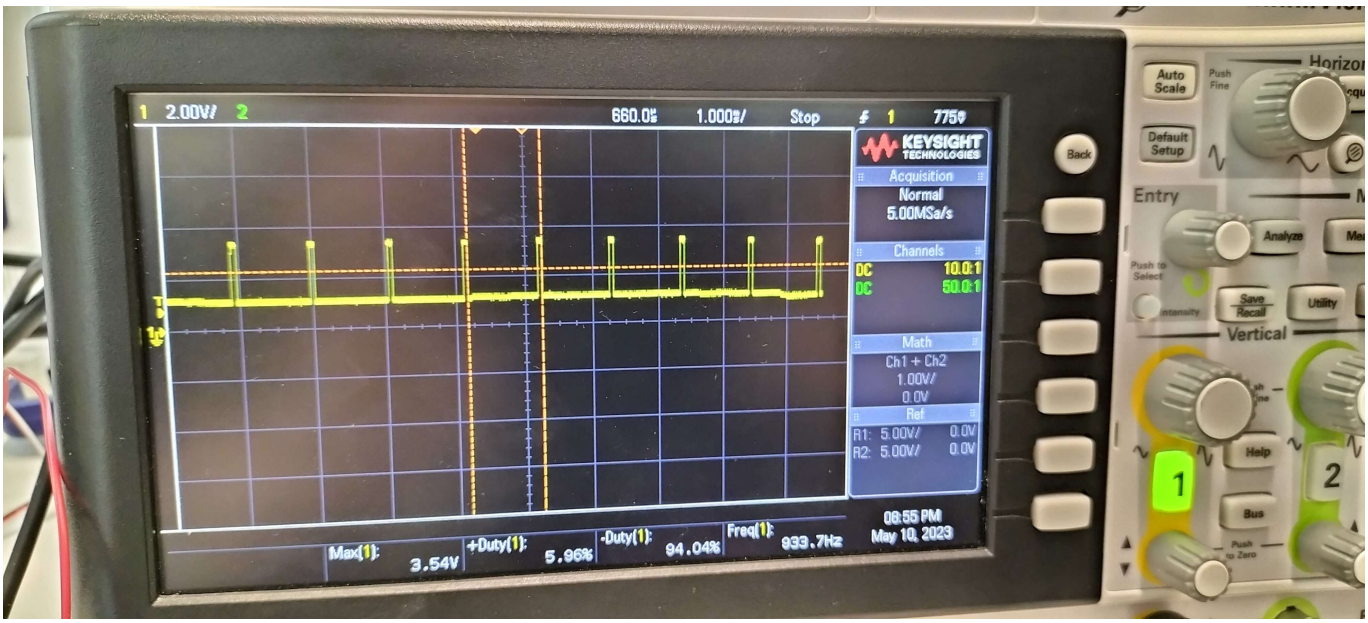
This sets D5 (Which is a PWM pin), as an actual PWM input pin. We can then read the value via the `CCR2` register of the timer. I'll be doing the same for D6, which is also a PWM pin.

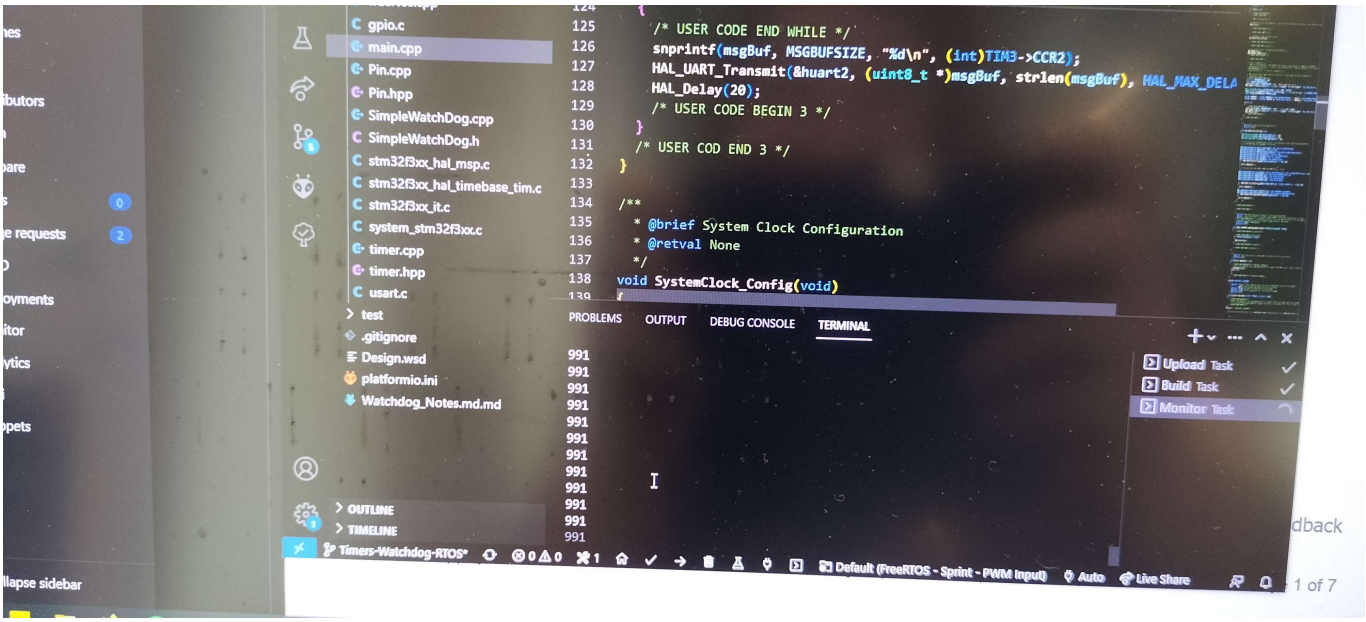
Measuring PWM Input & Output

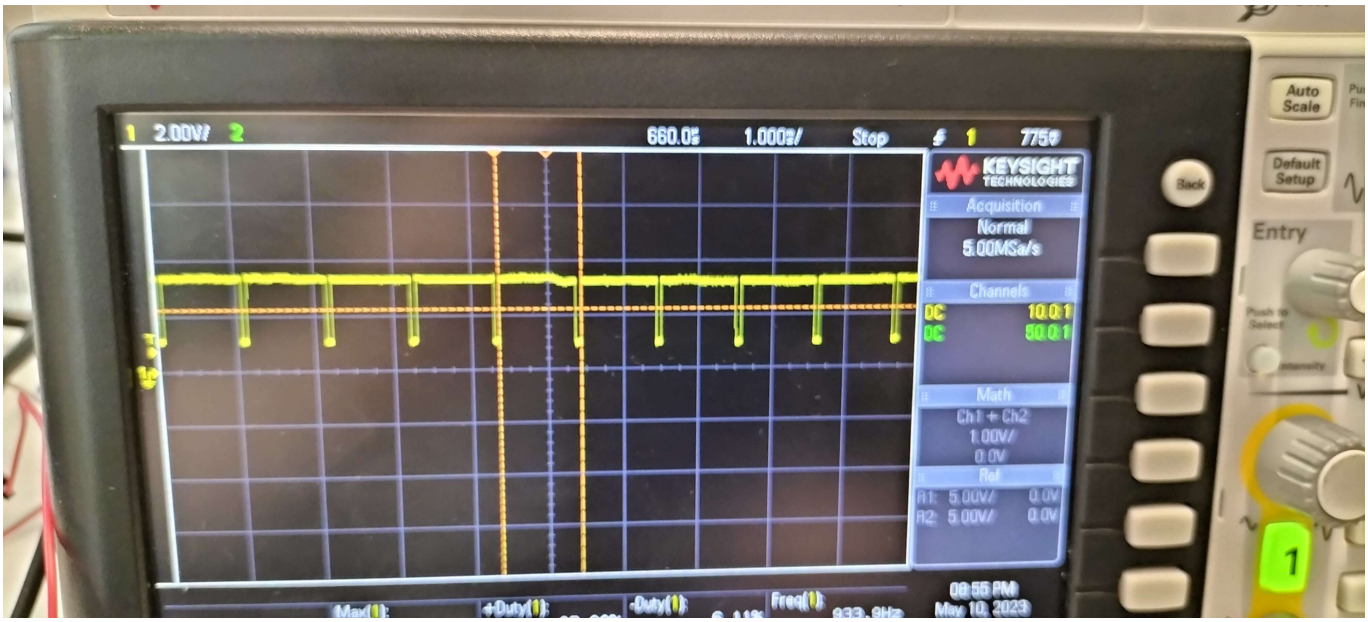



Digital Storage Oscilloscope
50 MHz 1 GSa/s
KEYSIGHT TECHNOLOGIES
Acquisition
Normal
5.00MSa/s
Channels
Math
Ch1 + Ch2
1.00V/
0.0V
Ref
Max(1): 3.62V
+Duty(1): 49.95%
-Duty(1): 50.05%
Freq(1): 934.1Hz
08:54 PM
May 10, 2023


SimpleWatchDog.h
stm32f3xx_hal_msp.c
stm32f3xx_hal_timebase_tim.c
stm32f3xx_it.c
system_stm32f3xx.c
timer.cpp
timer.hpp
usart.c
test
.gitignore
Design.wsd
platformio.ini
Watchdog_Notes.md.md
/* USER COD END 3 */
/**
* @brief System Clock Configuration
* @retval None
*/
void SystemClock_Config(void)
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
Upload Task
Build Task
Monitor Task
OUTLINE
TIMELINE
Timers-Watchdog-RTOS*
Default (FreeRTOS - Sprint - PWM Input)
Auto
Live Share
1 of 7


2.00V/
660.0ns
1.000s/
Stop
KEYSIGHT TECHNOLOGIES
Acquisition
Normal
5.00MSa/s
Channels
Math
Ch1 + Ch2
1.00V/
0.0V
Ref
Max(1): 3.54V
+Duty(1): 5.96%
-Duty(1): 94.04%
Freq(1): 939.7Hz
08:55 PM
May 10, 2023





Differentiation

One of the main problems with using the Servo as an input device, alongside the fact that it is not intended to be used as such, is the fact that the servo can turn bi-directionally. As Rene pointed out, this could prove to be a problem in the future, as it would be hard to detect which direction the servo is turing in. However, in testing, I have found that the servo cannot turn fast enough to lose track of itself. This means that the servo can be used as a PWM input device, without the need for a differentiation circuit.

This is primarily due to the significant amount of force required to turn the servo, which limits the rotational velocity. Even when forcing the fastest rotational speed via PWM input, the servo itself cannot turn fast enough to lose track of itself.

Watchdog

I have made a simple watchdog class, which is used to reset the microcontroller if the system is unresponsive. This is done by using the `IWDG` peripheral, which is a simple watchdog timer, which can be used to reset the microcontroller if the system is unresponsive.

The watchdog is reset every time the system is responsive, this is done by using the `Feed` function, which the system needs to trigger to indiciate that it is still responsive. If this is not activated in time, the system performs a hard reset.

FreeRTOS

Making use of FreeRTOS is very easy. For my current plan, I have decided to use the following tasks:

Task	Description
`SerialTask`	Handles serial input
`ManualTask`	Handles all manual control panel input

SerialTask

The `SerialTask` is responsible for handling serial input. This includes:

- Receiving commands

This can be achieved using the `scanf` function, as supplied by the `stdio.h` library. The scanf functions allows me to enforce a certain format for the commands, which makes it easier to parse the commands.

The main reason for not including serial output in the task as well, is due to the blocking nature of scanf. This means that the task will be blocked until a command is received, which means that the task will not be able to handle any other tasks, until a command is received. This in turn, would mean that there is no feedback to the user, which would make the system seem unresponsive.

This however, should not impact the other parts of the system, as the serial input will not be responisble for calling the watchdog reset function.

Side Note

After some additional consideration, I decided to use the `HAL_UART_Recieve` function for the purposes of reading the serial input. This is a fair bit cleaner, as well as adding less bloat due to not including the large `stdio.h` library.

Implementation

Results