Automatic Vacuum Chamber Controller Demo

Joshua Karch
December 3,2021

Introduction

For a long time, I wanted to develop a project employing use of pressure sensors, motors, valves, human interface devices, and displays, controlled by a microcontroller. At the same time I wanted to tinker with the concept of using an RTOS, such as FreeRTOS and an industry standard microcontroller, such as the STM32 platform to accomplish this. I came up with a concept to make a vacuum chamber for use in home projects. Vacuum chambers can be used for a variety of interesting experiments, like changing boiling points or causing rapid temperature changes. Simply hooking a container to a vacuum chamber can be dangerous as the container may implode, so having the ability to control vacuum to a certain level was an interesting concept I wanted to tinker around with. I started with some parts off Adafruit and SparkFun, along with an STM32F4 Discovery board.

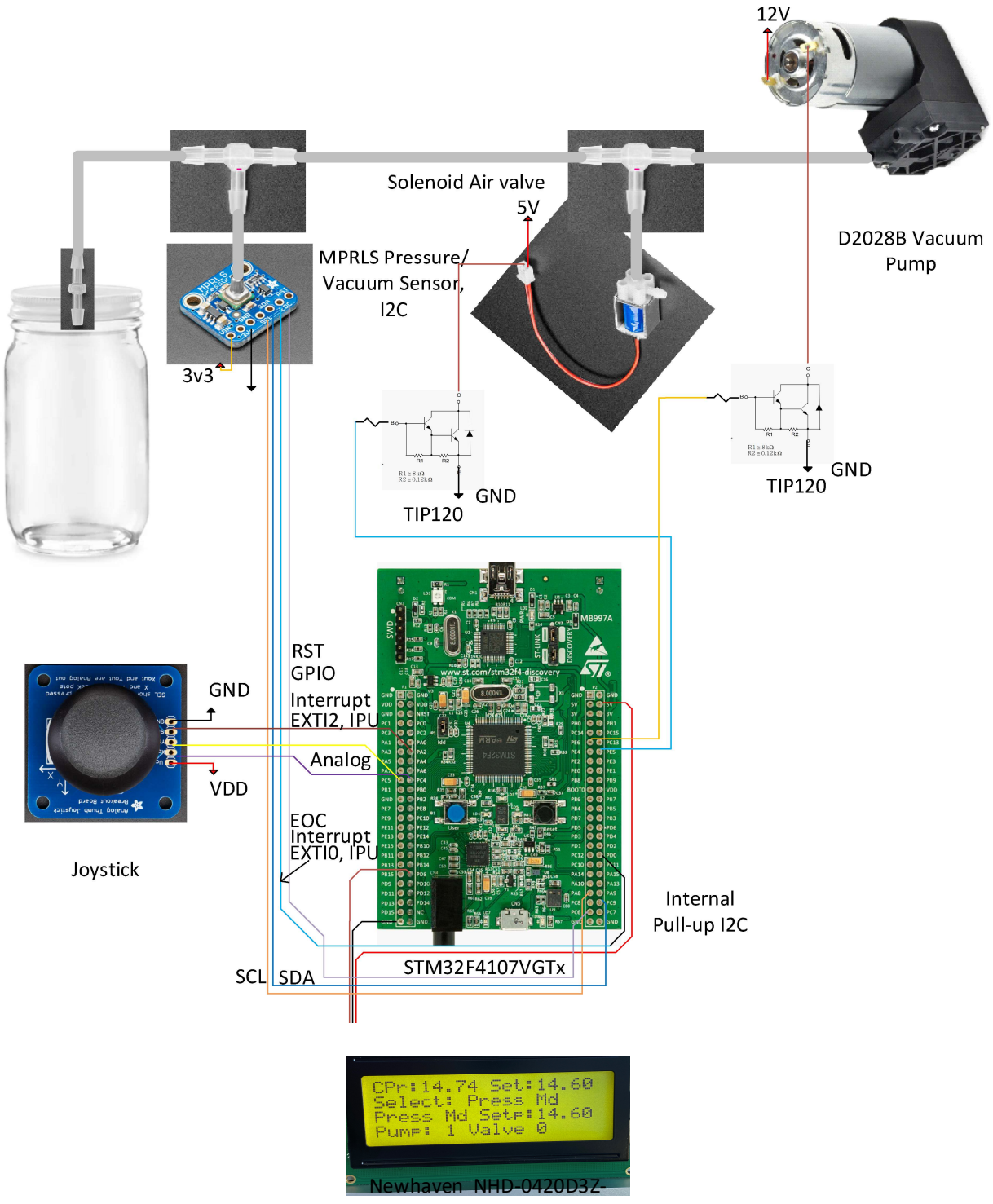


Figure 1: Pictographical Schematic of the Vacuum Chamber Controller

Hardware Architecture

The hardware setup consists of an STM32F4 Discovery board, a 4x20 line LCD display, a joystick with a pushbutton, a vacuum pump, Darlington NPN transistor drivers, a solenoid valve, and a Honeywell MPRLS pressure sensor that can sense both positive and negative pressure from 0-25 PSI, where 14.7PSI is standard atmosphere at standard temperature and pressure. The general behavior of this system is not unlike an A/C and Heating system where instead of temperature, vacuum level is controlled within a band. If too much vacuum is drawn, a relief valve will release until the desired level is reached, whereas the vacuum pump will pull a hysteresis value more than the setpoint to lower the duty cycle.

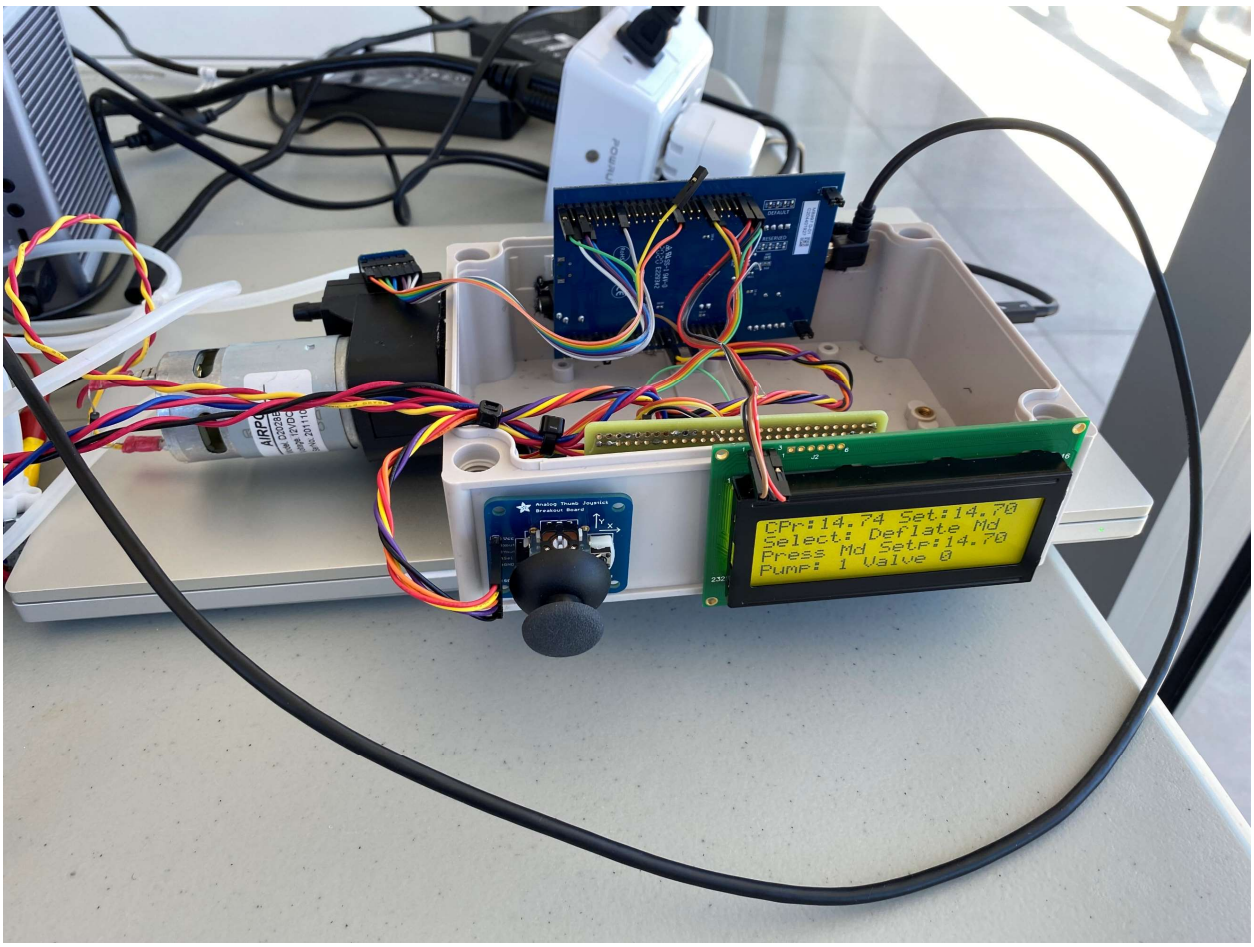


Figure 2: Vacuum controller setup. Vacuum device under test not shown

Software Architecture

I decided to use Microsoft Visual Studio Community Edition with VisualGDB extensions. This replaces the STM32 Eclipse based environment and provides a common platform for development on a wide variety of embedded platforms from STM32 to ESP32, Arduino platforms, and embedded Linux Platforms. It's a one-stop shop that turns VS into an embedded development environment, and uses OpenOCD for debugging. It can even perform remote GDB debugging on embedded Linux hardware and deploy embedded Linux applications to the Raspberry Pi. In order to speed up the hardware development process, I decided to use ST Microelectronics' Hardware Abstraction Layer libraries and STM32CubeMX for GPIO, timer, RTOS, Queues, and Peripheral configuration. This tool makes it easy to set the individual control bits on registers, enable interrupts, pull-up or pull-down resistors, slew rates, serial data interface rates, and even create the templates for FreeRTOS threads and Queues using the ARM CMSIS V2 API(https://www.keil.com/pack/doc/cmsis/RTOS2/html/index.html). The way this system works is that graphical configuration leads to configuration of the main source code and peripheral libraries, only enabling perippherals that have been selected for operation. Then, the code blocks contain guarded snippets where a user can enter custom code. If custom code is placed outside these snippets it will be erased on subsequent configuration sessions with STM32CubeMX. See Using FreeRTOS with STM32 Devices and VisualGDB – VisualGDB Tutorials (https://visualgdb.com/tutorials/arm/stm32/freertos/) for more information on setting the system up with FreeRTOS, STM32CubeMX, VisualGDB, and Visual Studio Community edition.

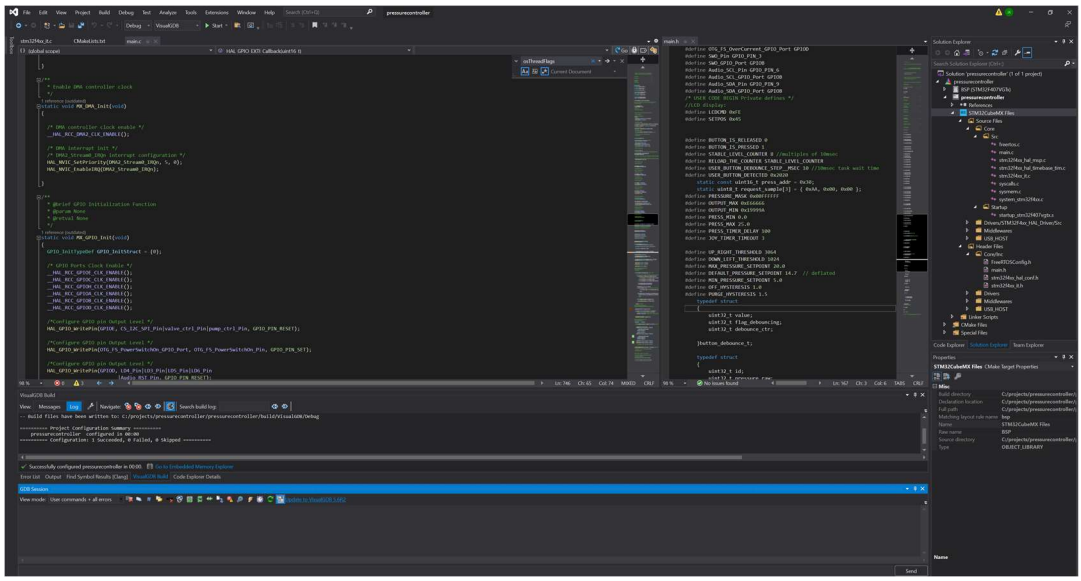


Figure 3: VisualStudio Code Community Edition with VisualGDB Extensions

RTOS Code Organization

The firmware is developed using FreeRTOS with CMSISv2 extensions. Hardware interactions are interrupt-driven, with care taken to minimize intensive code in ISRs. There is no polling in this application.

For the I2C bus, transmit and receive interrupts are enabled, whereas for the UART, transmit interrupts are enabled. An internal timer, Timer2 triggers ADC readings which write to a circular buffer on the DMA. Only when ADC acquisition is complete on all channels is an interrupt event triggered.

This is a description of the program flow, Figures 4 and 5 below should help visually explain the operation. In some ways, this program flow is a bit overkill, but is something I would use in more complex RTOS-based embedded designs.

The primary driving source of this control system is the vacuum/pressure sensing loop. Control System activities run specifically based on this driving source. A single RTOS timer running at 10Hz triggers a request over I2C to read from the MPRLS sensor. A single interrupt-driven write to address 0x18 (0x30) triggers the internal A/D to start sampling on the MPRLS component. When complete, an EOC signal drives high. That signal triggers an interrupt on EXTI0, which causes a call for I2C read, which is also interrupt driven and reads out four bytes, the MSByte being a status byte, and bytes 2-0 are 24 bit representations of pressures from 0-25PSI.This data is unpacked into a single uint32_t and passed over a sensor queue used for all data acquisition with an ID representing a pressure read, followed by a uint32 of data. This queue send unlocks the control thread which blocks for data reception. This queue is shared with an analog to digital measurement thread and a button debounce thread. Conversion of the uint32 data to floating point pressures is done in the control task as it is not appropriate to do a the end of an ISR.

The ADC data acquisition chain reads from two analog channels, 14 and 15 representing joystick axes Y and X. These values drive the controls for the user interface. The ADC acquisition is triggered by a timer at 25Hz and when both channels are read the data is passed on via DMA to a memory location. Then an interrupt is triggered, and that data is copied by the MCU into the same sensing queue with an ID of “Joystick Data”. There is some inefficiency here no doubt, and it may be possible to point directly to the data structure memory locations pointing to the queue, so that the queue can send off the data without extra copying. This message gets sent to the control thread, which determines if the joystick is setting pressure setpoints up or down or setting control modes, left and right.

A third and final data source is the button debounce task. When a button gets pressed on the joystick, an interrupt on EXTI0 is triggered. When this happens, if the thread for button debounce is not running, a thread Flag signal is sent once to start the debounce process. This debounce Flag starts the debounce event, which runs a number of cycles

and determines if the GPIO has settled “pressed” sufficiently before sending over the data acquisition queue an empty message that a button was pressed.

The three sources of data, vacuum data, joystick data, and button data all trigger the control thread which blocks until one of these events occur. The control of pumps is driven solely on vacuum pressure measurement, while the joysticks are used by the same thread to control user interface selections, such as mode or pressure settings. It’s possible to split this into two threads, but I decided to keep them together due to the slow dynamics of the system. A button press queue simply commits changes to modes and pressure settings that were selected with the joystick, while the main control portion uses the current mode and setpoints to perform simple bang-bang control of pressure with hysteresis. The vacuum pump will run until the setpoint minus a hysteresis value is triggered, then the pump will turn off. As a safety feature however, if another threshold value that is lower in pressure is measured, the pressure relief valve will open until the setpoint is reached. This system is still single string, and so ideally it’s a good idea to have a watchdog timer watching the application, but this functionality has not yet been implemented. All of the data for provisional setpoints and selected setpoints is sent over a second queue to an LCD print over UART task, which sends the information to the 4 line by 20 character LCD display. That device has requirements for latency in sending data, so the print routines contain thread sleep events which enable the RTOS to perform other duties. As a result, the LCD task is of the lowest priority. However, since this thread uses the fairly heavy sprintf functions, the stack size had to be increased considerably.

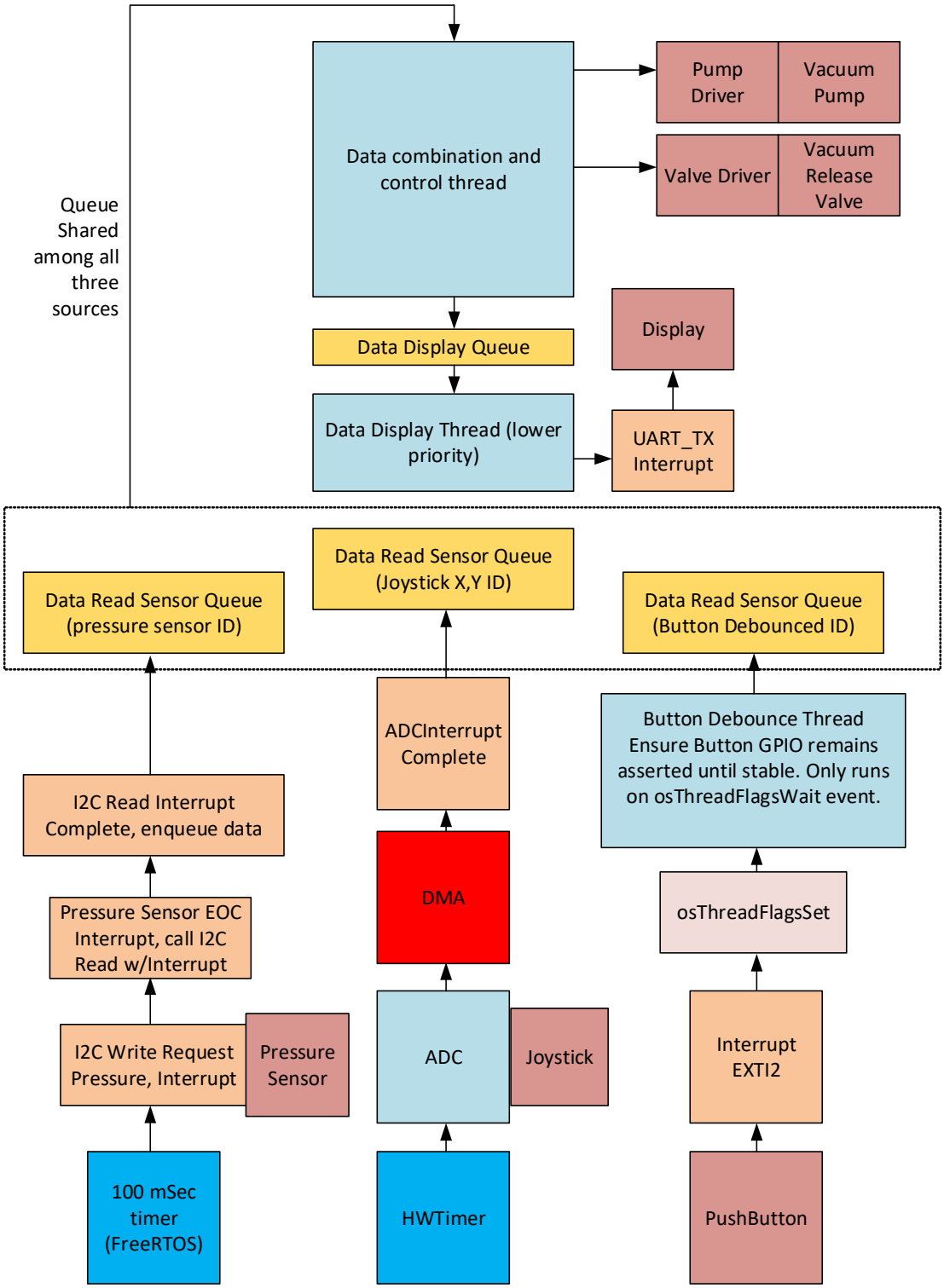


Figure 4: FreeRTOS Configuration

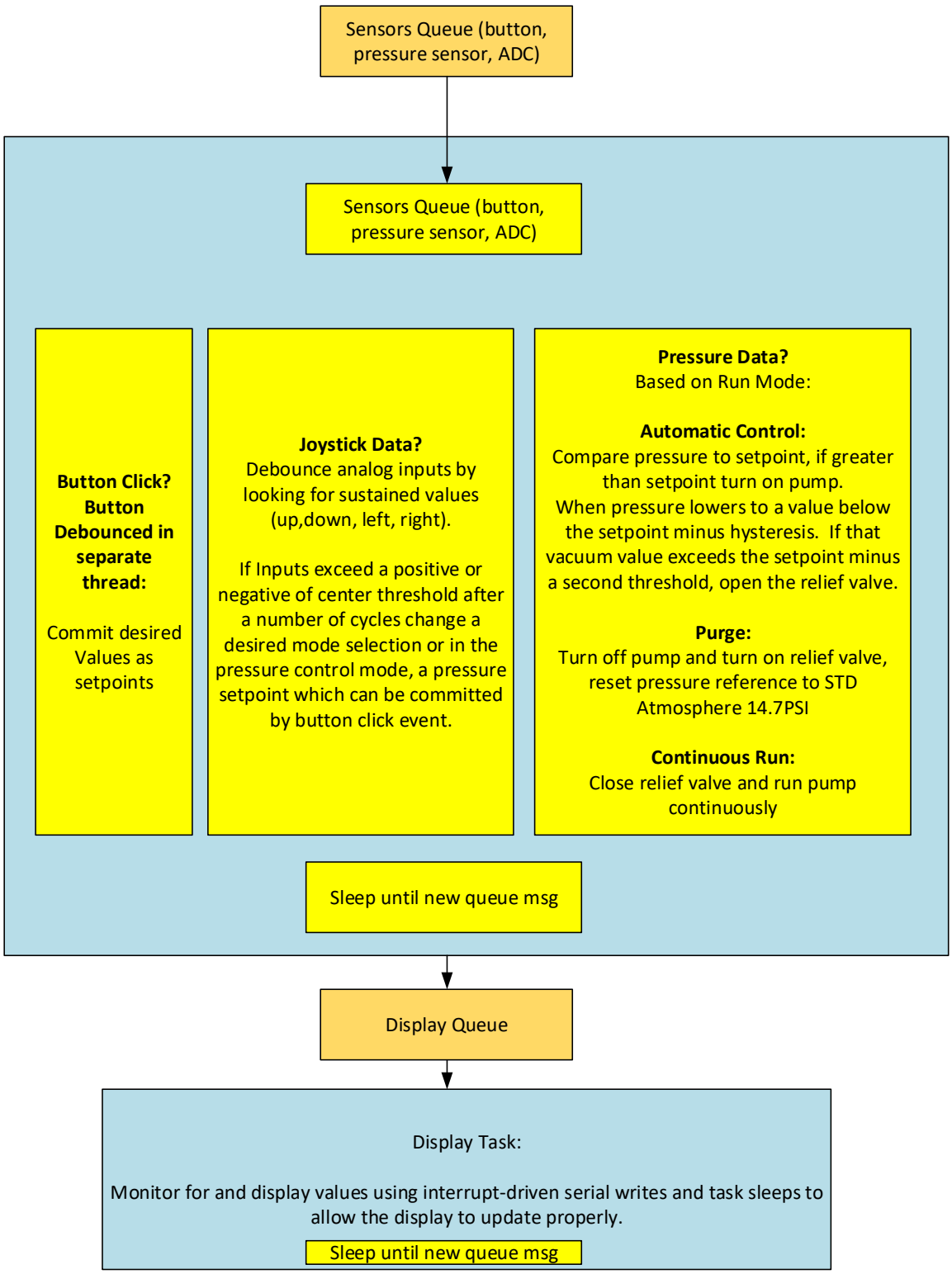


Figure 5: Sensor Data Processing Thread

Display Modes of Operation: Purge, Control, and Continuous

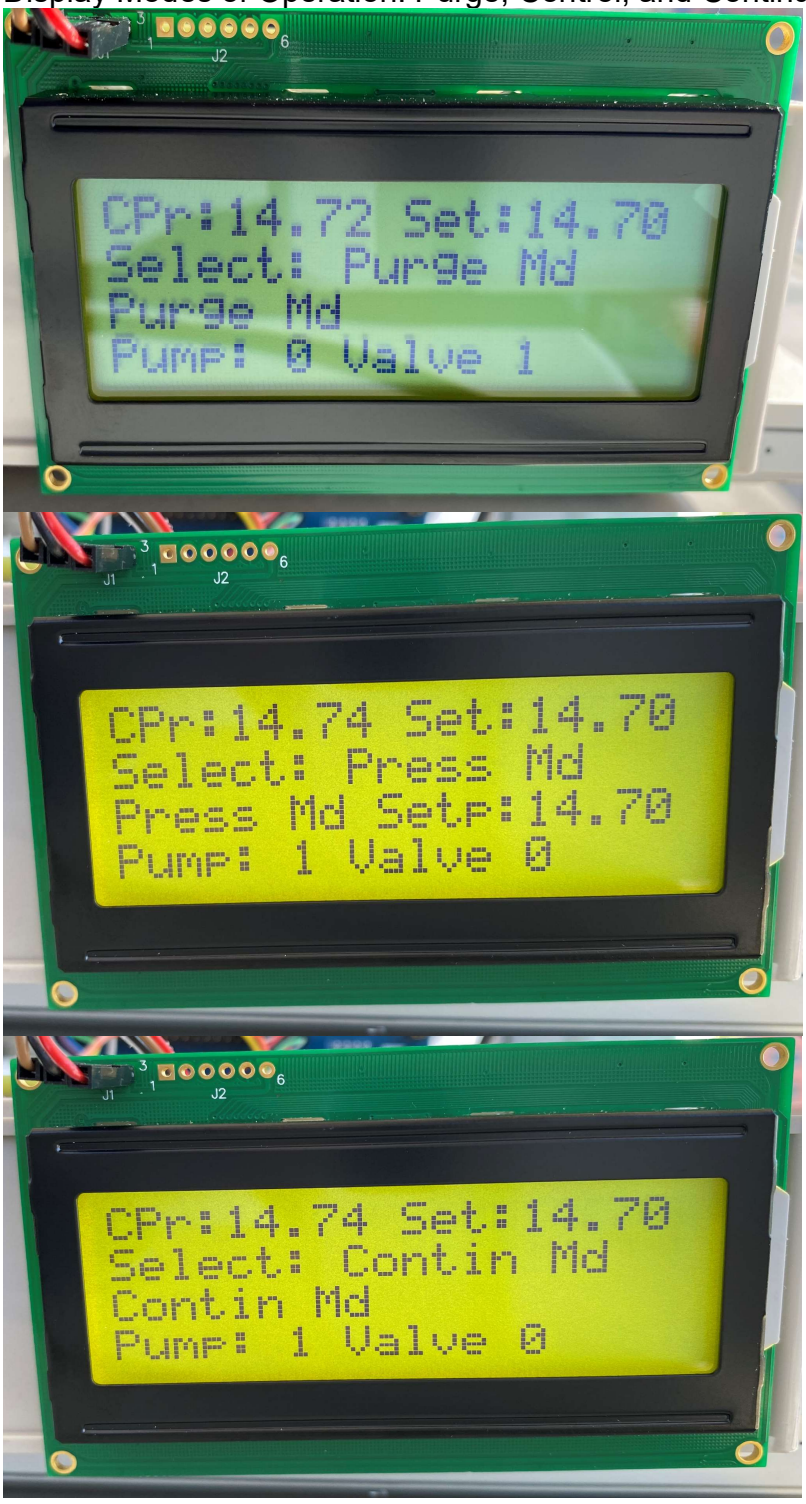


Figure 6: Modes of operation.

STM32CubeMX Configurations

The sections below show screenshots of critical configurations needed to make this system work.

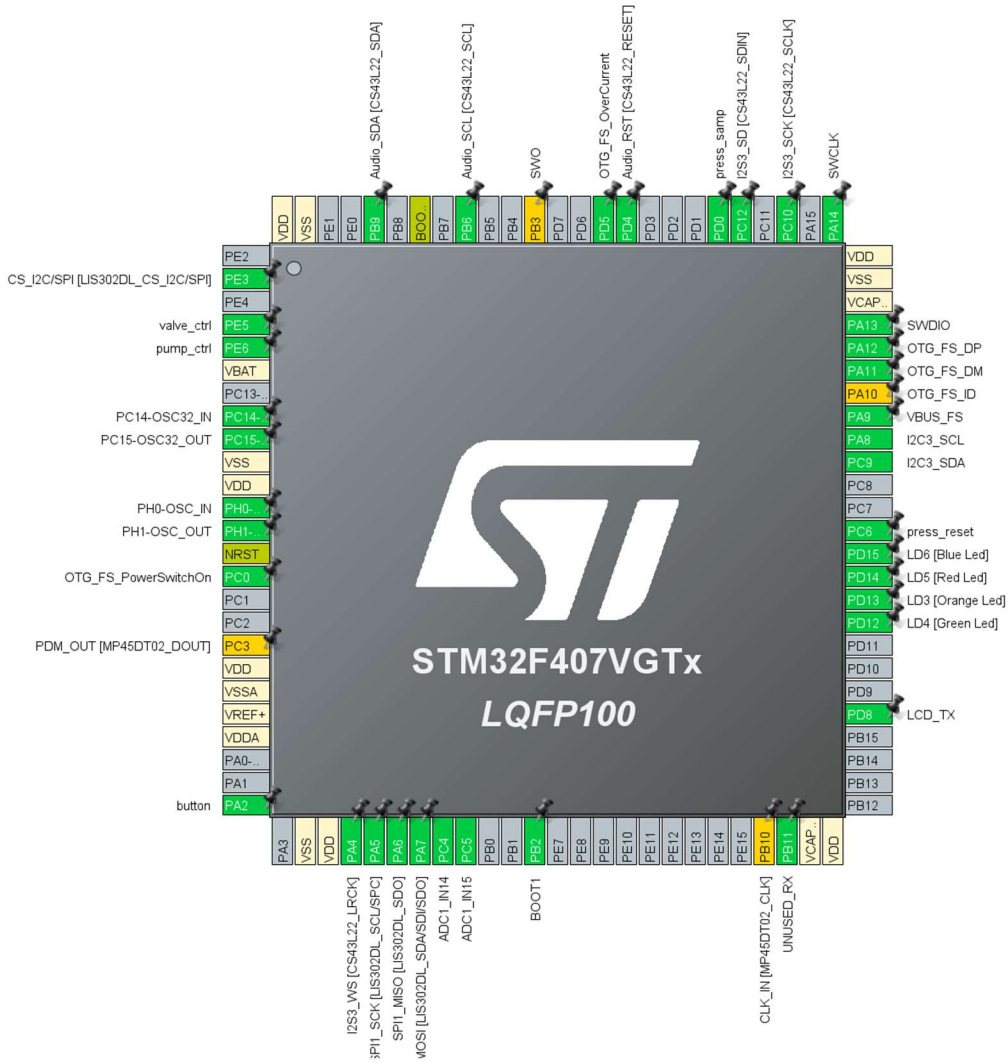
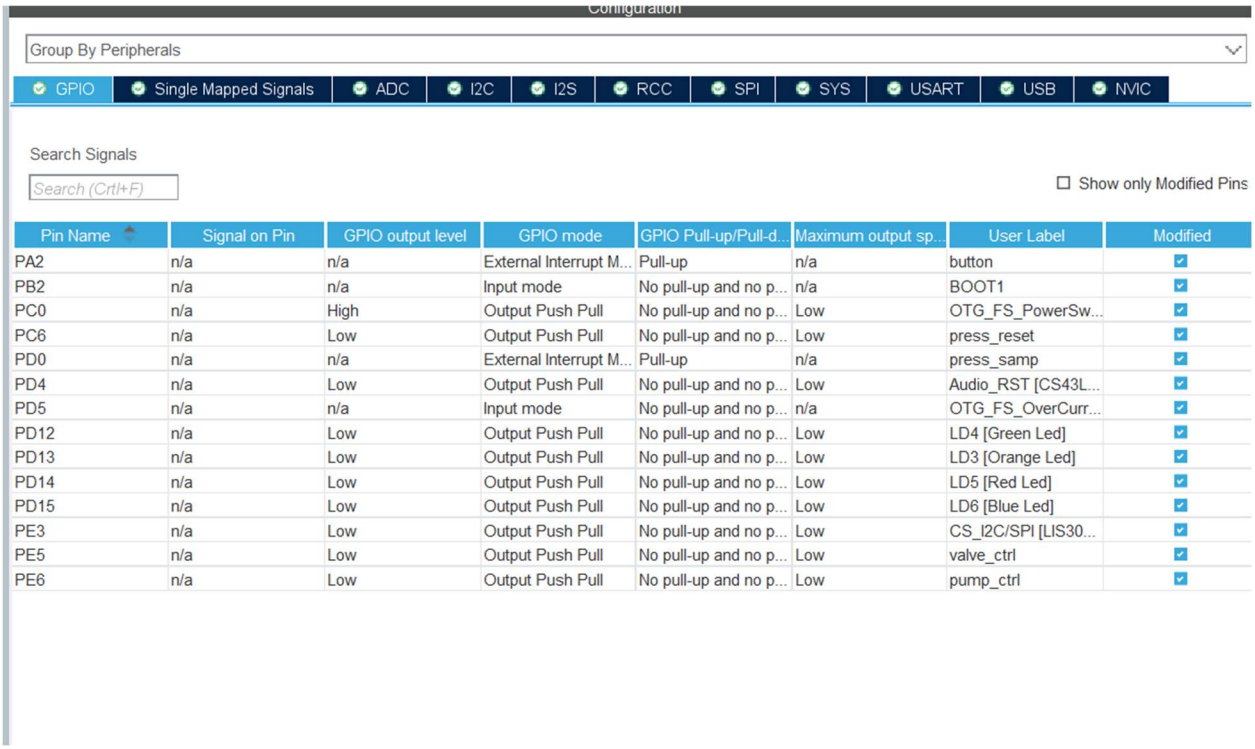


Figure 7: STM32CubeMX Hardware Configuration, derived from the base STM32F4Discovery board setup, coupled with customizations for GPIO.



Pin Name	Signal on Pin	GPIO output level	GPIO mode	GPIO Pull-up/Pull-d...	Maximum output sp...	User Label	Modified
PA2	n/a	n/a	External Interrupt M...	Pull-up	n/a	button	☑
PB2	n/a	n/a	Input mode	No pull-up and no p...	n/a	BOOT1	☑
PC0	n/a	High	Output Push Pull	No pull-up and no p...	Low	OTG_FS_PowerSw...	☑
PC6	n/a	Low	Output Push Pull	No pull-up and no p...	Low	press_reset	☑
PD0	n/a	n/a	External Interrupt M...	Pull-up	n/a	press_samp	☑
PD4	n/a	Low	Output Push Pull	No pull-up and no p...	Low	Audio_RST [CS43L...	☑
PD5	n/a	n/a	Input mode	No pull-up and no p...	n/a	OTG_FS_OverCurr...	☑
PD12	n/a	Low	Output Push Pull	No pull-up and no p...	Low	LD4 [Green Led]	☑
PD13	n/a	Low	Output Push Pull	No pull-up and no p...	Low	LD3 [Orange Led]	☑
PD14	n/a	Low	Output Push Pull	No pull-up and no p...	Low	LD5 [Red Led]	☑
PD15	n/a	Low	Output Push Pull	No pull-up and no p...	Low	LD6 [Blue Led]	☑
PE3	n/a	Low	Output Push Pull	No pull-up and no p...	Low	CS_I2C/SPI [LIS30...	☑
PE5	n/a	Low	Output Push Pull	No pull-up and no p...	Low	valve_ctrl	☑
PE6	n/a	Low	Output Push Pull	No pull-up and no p...	Low	pump_ctrl	☑

Figure 8: GPIO Port Configurations

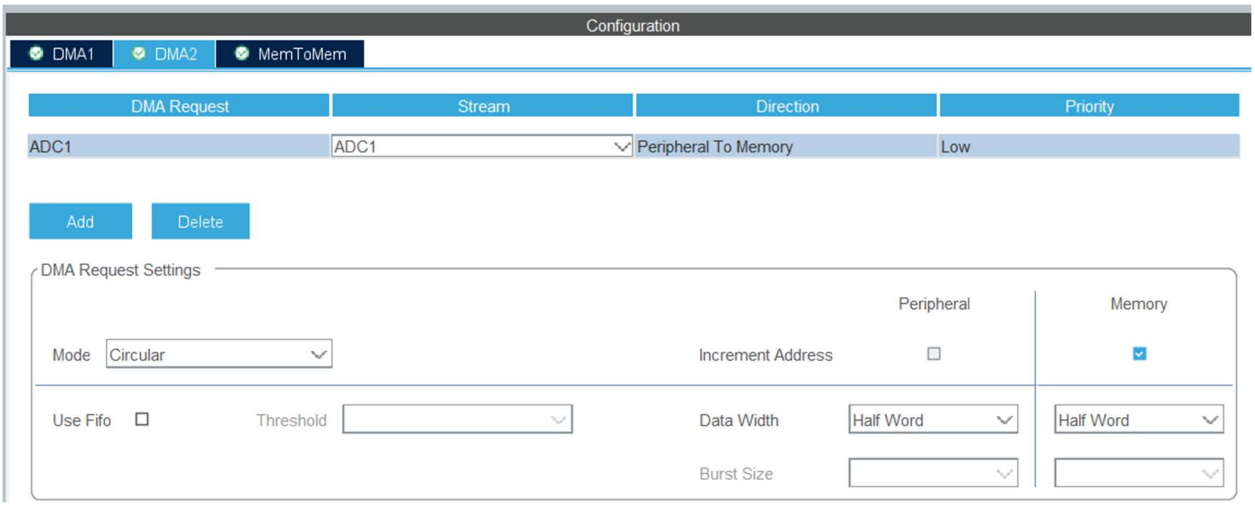


Figure 9:ADC DMA Settings

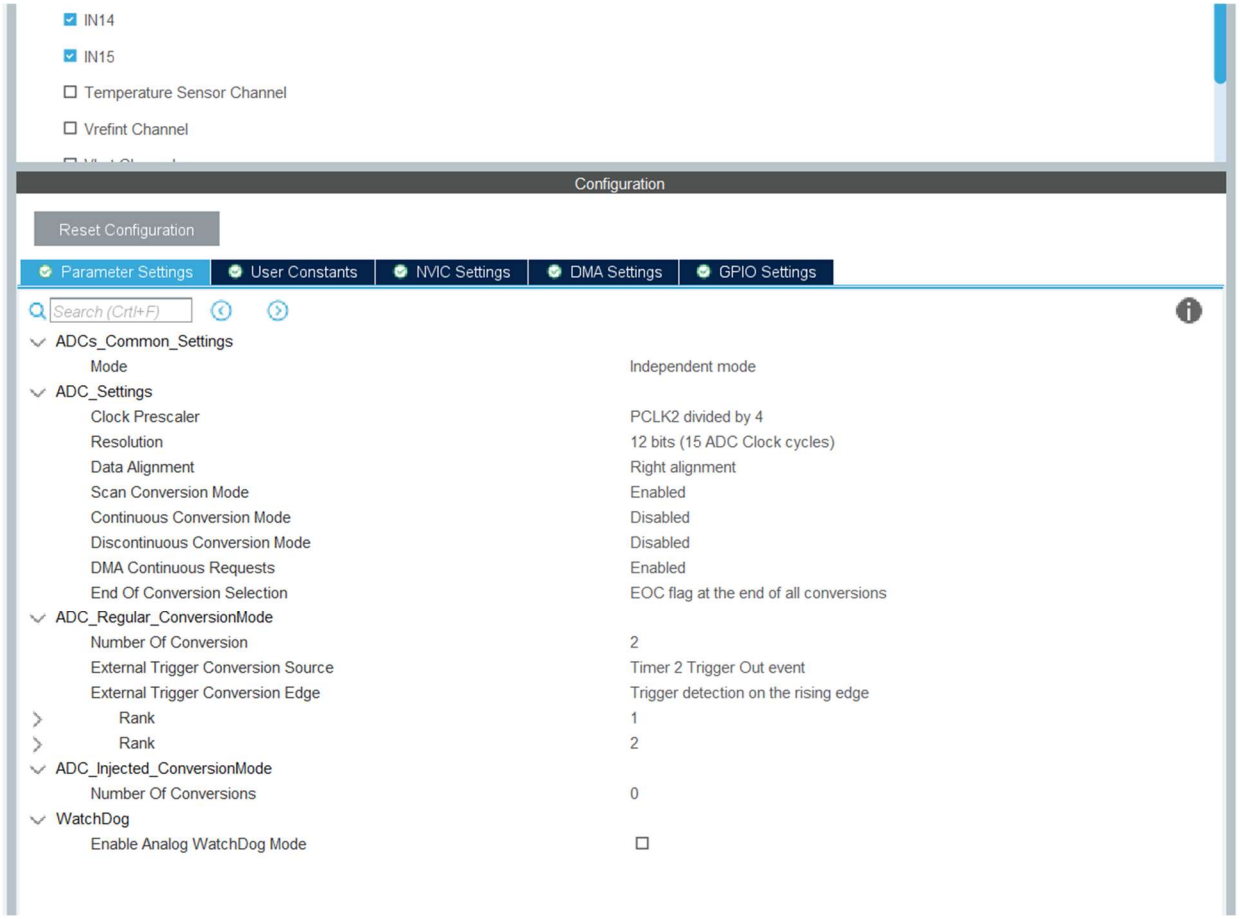


Figure 10: ADC settings, note the ADC read is triggered by timer and writes to DMA.

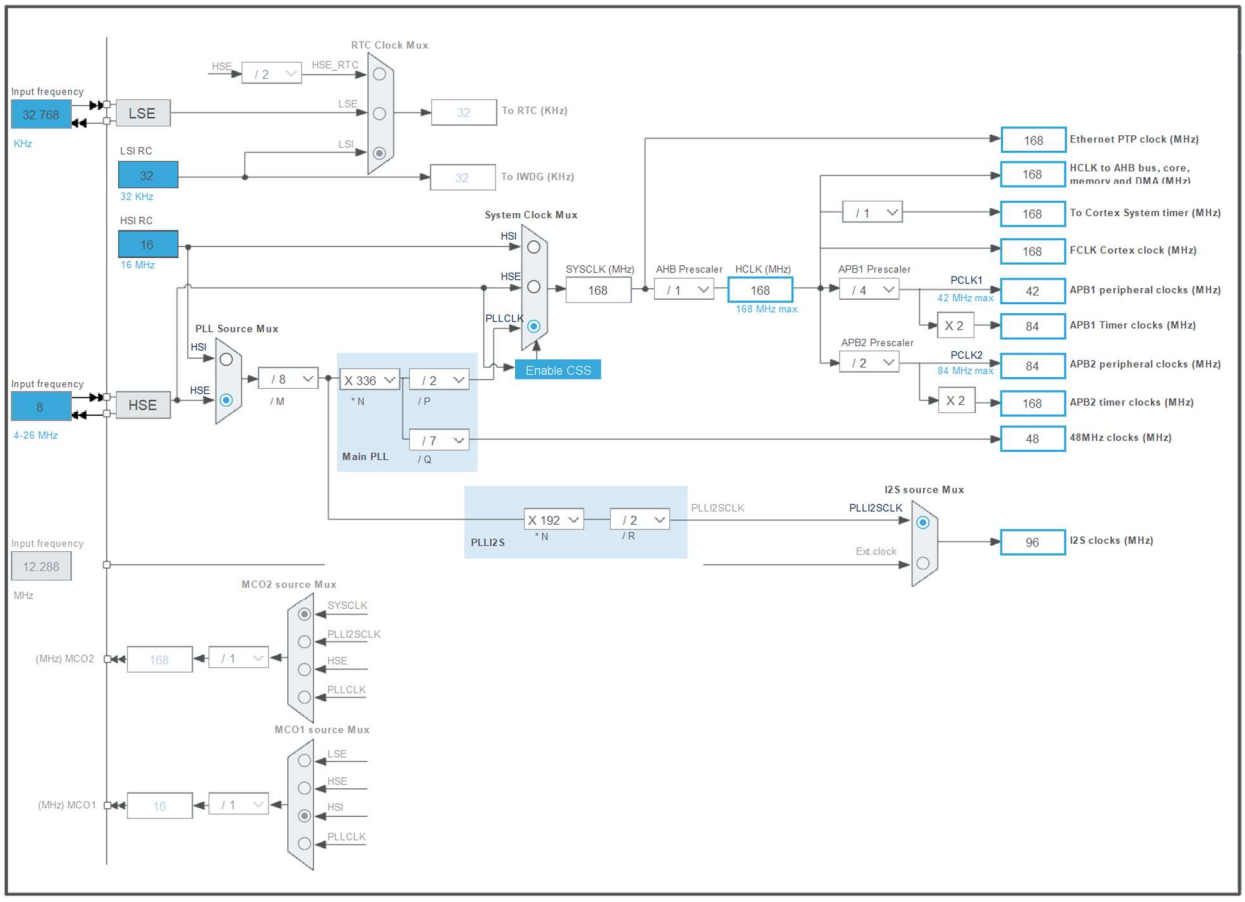


Figure 11: System Clock Settings

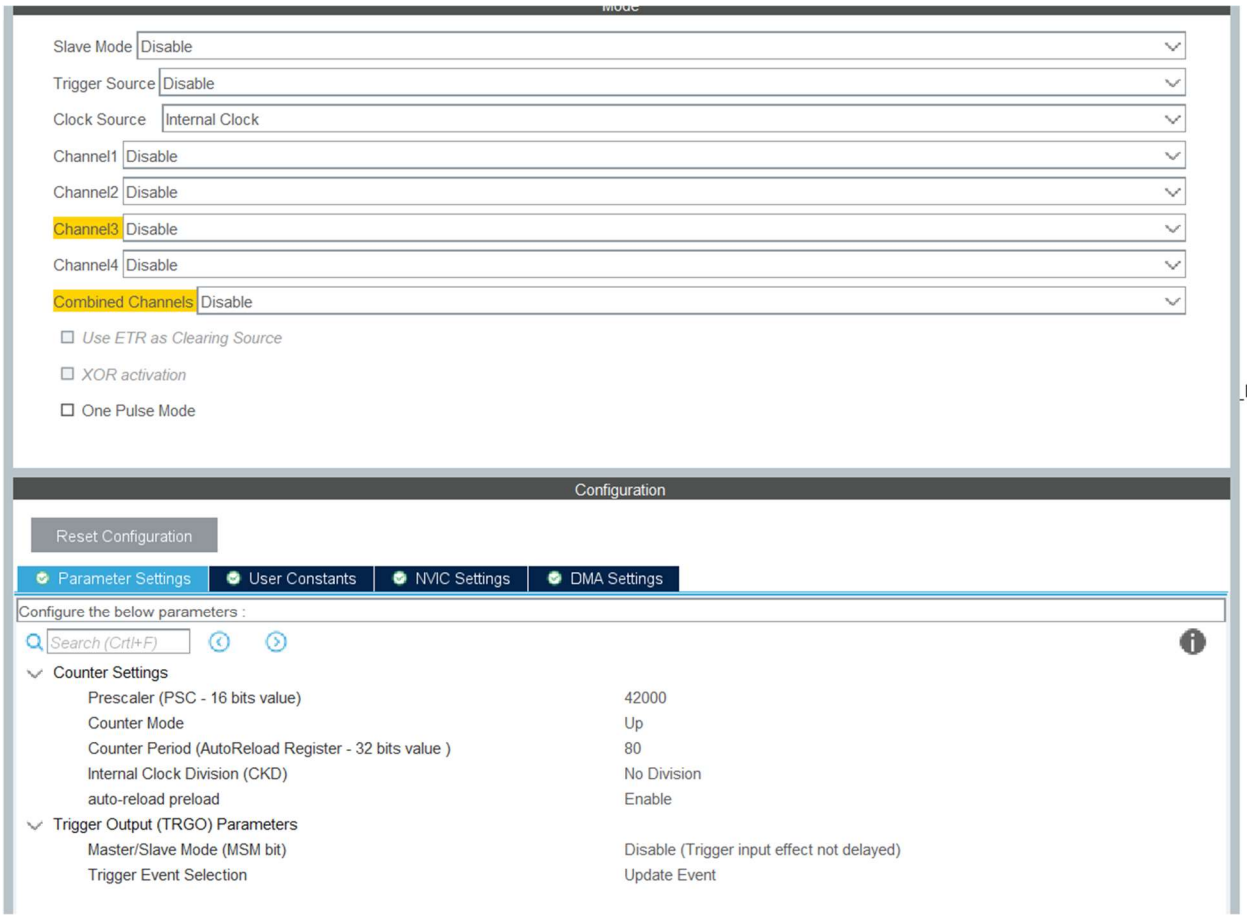


Figure 12: ADC timer settings (off APB2 Peripheral clock at 84 MHz)
The rate of data triggering is 82,000,000/(42000*80) = 25 Hz.

FreeRTOS Configuration:

In order to set up FreeRTOS to work with the STM32 the following configurations are recommended per STM32CubeMX.

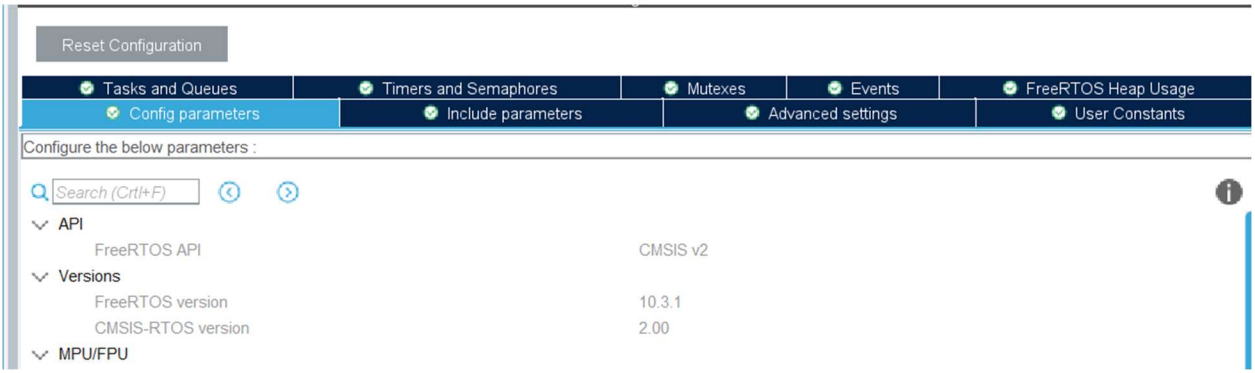


Figure 13: FreeRTOS Settings: CMSIS v2. It is also possible to code in the native FreeRTOS API.

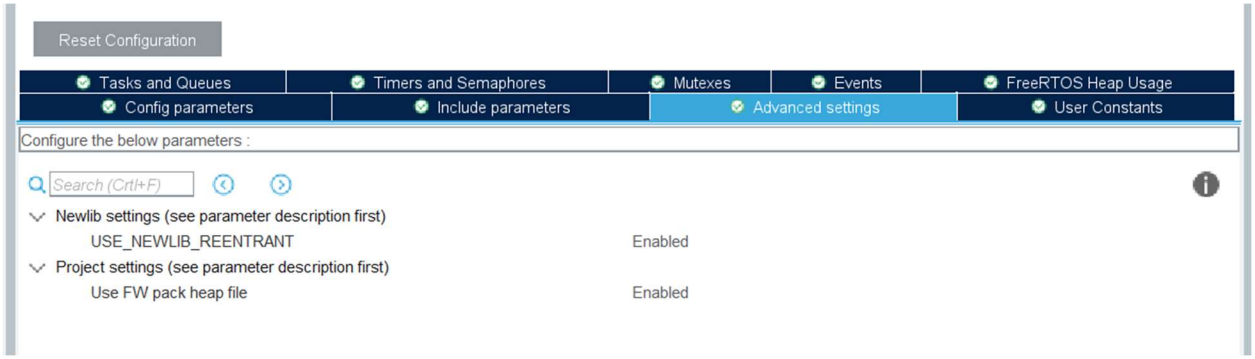


Figure 14: It is critical to set USE_NEWLIB_REENTRANT

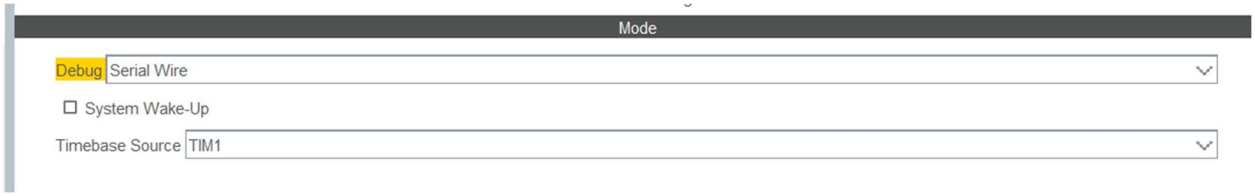


Figure 15: Use Timebase Source TIM1 in SYS.

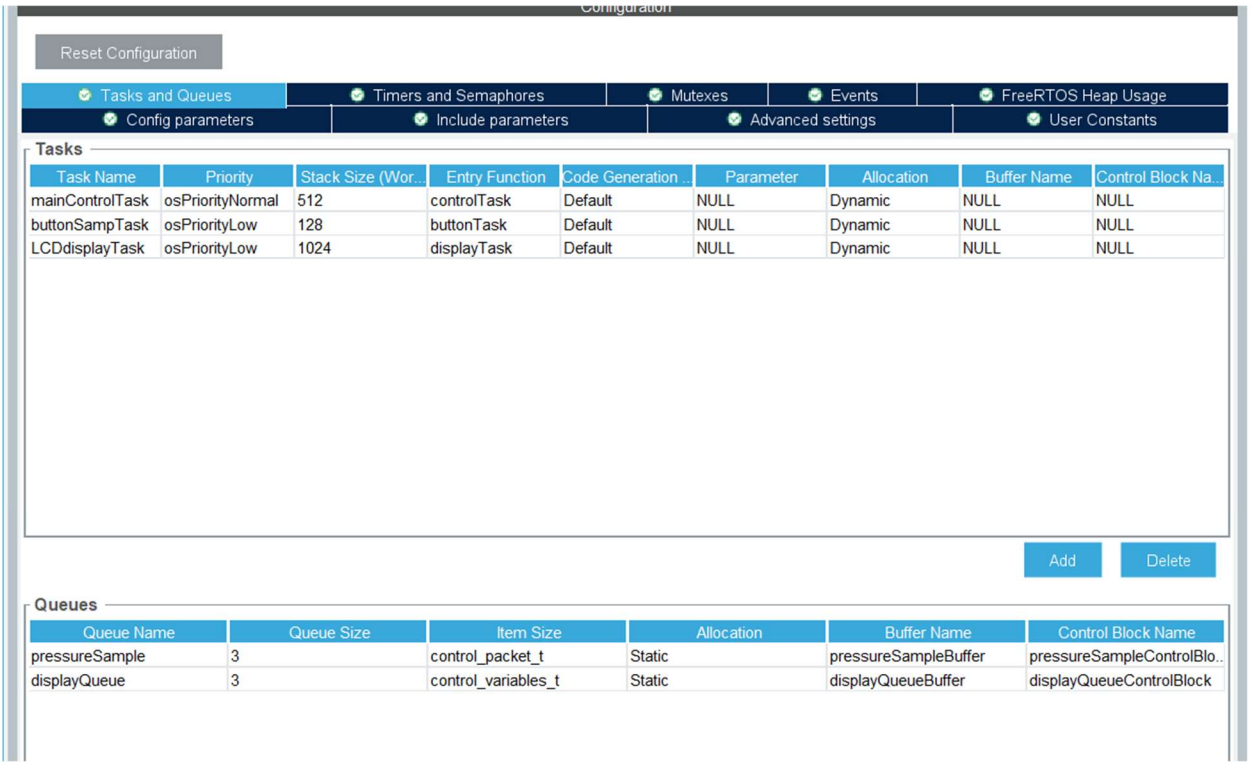


Figure 16: Set up tasks here. You can also manually create those tasks. Regarding Queues, Stm32CubeMX does not properly make queue buffers correctly, as it likes to use the uint8_t data type. I had to change the automatically declared buffers from uint8_t pressureSampleBuffer[3*sizeof control_packet_t] to control_packet_t pressureSampleBuffer[3];

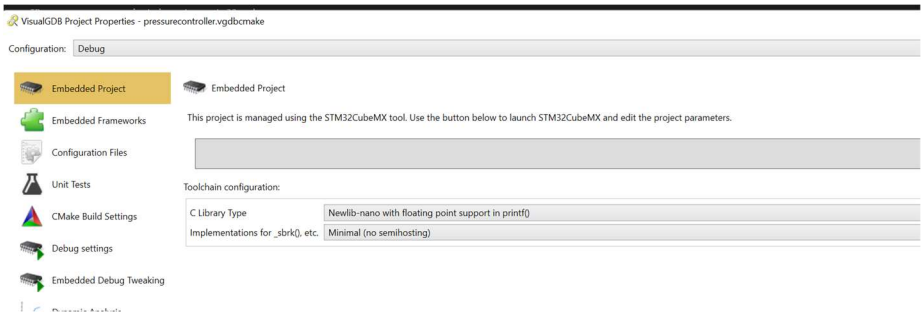


Figure 17: Be sure to set Newlib-nano in project properties to enable printf/sprintf support.

Final Notes

I originally tried to develop this application in a similar setup without FreeRTOS using a single main loop but with interrupt driven activities, and used timers to run debounce routines. This ended up working pretty well for this application, but anything much more complex than this type of activity makes sense using an RTOS. Queues and flags are helpful to block tasks from running unless needed.

Item	Link
Vacuum Sensor	https://learn.adafruit.com/adafruit-mprls-ported-pressure-sensor-breakout
Air Pump	https://www.sparkfun.com/products/10398
Air Valve	6V Air Valve with 2-pin JST PH Connector [FA0520E] : ID 4663 : $2.95 : Adafruit Industries, Unique & fun DIY electronics and kits
STM32	STM32F4DISCOVERY STMicroelectronics | Development Boards, Kits, Programmers | DigiKey
LCD	NHD-0420D3Z-FL-GBW-V3 Newhaven Display Intl | Optoelectronics | DigiKey
Darlington NPN	TIP120 Power Darlington Transistors - 3 pack : ID 976 : $2.50 : Adafruit Industries, Unique & fun DIY electronics and kits
Joystick	Analog 2-axis Thumb Joystick with Select Button + Breakout Board : ID 512 : $5.95 : Adafruit Industries, Unique & fun DIY electronics and kits

Figure 18: Parts List