# Introduction

This design is ½ of a demonstration for Electronica 2018.

This document describes the main controller. The remote-control design is described in a separate document.

The main controller consists of 2 water tanks that are controlled and measured by a PSoC 6 WiFi/Bluetooth kit. It also has LEDs, a speaker, and a UART interface. There will two teams of three people each (chosen randomly from the audience). The goal is for each team to try to fill their water tank first. To do this, they use remote controls that we will provide containing another PSoC 6 WiFi/Bluetooth module, CapSense, OLED Display, and LEDs. The contestants will use a CapSense slider on their remote to swipe left to try to fill the left tank or swipe right to try to fill the right tank. Messages will be sent to the main controller via either WiFi (using AWS) or BLE.

# Hardware

### List

1. CY8CKIT-062 PSoC 6 Pioneer WiFi Kit
2. Custom Shield with:
   1. Dual Motor Driver (TB6612FNG)
   2. Dual LED Control for WS2812 RGB LEDs
   3. Amplifier and Speaker Output
   4. Dual connectors for Liquid Level Sensing
   5. Start Button
3. CY8CKIT-022 (2) to do liquid level sensing on two bottles.
   1. Note: The sensors and bottles from this kit are used but not the shield.
4. Submersible Pumps (2)
   1. <https://www.amazon.com/Mavel-Star-Submersible-Fountain-Upgraded/dp/B0713T9PRP/ref=pd_lpo_vtph_200_lp_t_2?_encoding=UTF8&refRID=CAMHGDA9695DQ80AY3AR&dpID=41Da%252BTqs1VL&preST=_SY300_QL70_&dpSrc=detail&th=1>
   2. These need to have their connectors cut off and wires stripped and tinned so that they can be connected to the shield.
5. Plastic Tubing
   1. <https://www.amazon.com/dp/B000E62TCC/ref=twister_B07GZVYNXF?_encoding=UTF8&th=1>
6. Bucket (4L or larger)
7. Duct Tape (or is it Duck Tape?)
8. (Optional) Base to connect various parts of the setup.
9. (Optional) Caps for the bottles (2) with holes for the tubing drilled in them.

### Assembly

To assemble the demonstration:

1. Attach 2 liquid level sensors to 2 bottles from the CY8CKIT-022.
2. Connect the custom shield to the CY8CKIT-062.
3. Connect the 2 liquid level sensors/bottles to the custom shield.
4. Place the entire assembly on the base and attach if necessary/desired.
5. Place the bucket near the bottles (on the floor under the table is probably OK) and fill with at least 3L of water.
6. Connect plastic tubing to each of the 2 pumps. The tubing goes on the outlet port which is on the side (opposite the power cord).
7. Connect the pump power leads to the custom shield.
8. Place the pumps inside the bucket.
9. Connect the tubes from the pumps to the bottles – attaching them right at the top of the bottles is best – if they are inside the bottles you can get siphoning back into the bucket.
   1. We might want to use caps on the bottles with a hole drilled in them to just fit the tubing.

# Firmware

## Functional Description

### Startup

Upon powerup or reset, the firmware will be in a state that is ready to start the game. It will:

1. Connect to WiFi – SSID and Password will be hard coded into the FW.
2. Connect to AWS and Subscribe to MQTT game messages.
   1. The broker name is: "amk6m51qrxr2u.iot.us-east-1.amazonaws.com"
   2. Messages will be received from the MQTT Topic "PumpAWS".
   3. We will use the AWS account wiciedwifi101.
   4. GJL will supply certificates.
3. Act as a BLE Central and scan for BLE devices with specific custom manufacturer data (vendor ID and product ID).
4. Allow up to 6 BLE devices to connect as Peripherals.
5. Act as a GATT Server. The server will have:
   1. One Custom Service Containing the following Characteristics:
      1. WaterLevelLeftBLE
      2. WaterLevelRightBLE
      3. PumpLeftBLE
      4. PumpRightBLE
   2. The first 2 Characteristics will be Readable by the Clients and the last two will be Writable.
6. Display pertinent information on a UART terminal. For example, messages regarding WiFi connection, MQTT connection, BLE, etc. to make sure things are running properly.
7. After initialization, measure the water level in each tank (presumably will be 0) and:
   1. Publish the values to the MQTT broker. See the Game Operation details below for the shadow topic to use.
   2. Update the values in the WaterLevelLeftBLE and WaterLevelRightBLE Characteristics.
8. Wait for user input to start the game (using a mechanical button or UART command).

### Game Operation

A mechanical button or UART command will be used to start game operation. Once the game starts the firmware will:

1. Play a sound for a short time (fight bell).
2. Monitor water level in both tanks. LEDs will be lit to represent the amount of water in each tank.
3. Look at MQTT messages received from the PumpAWS Topic and increment the appropriate counter for each message it receives. The counter will increment by the amount of the value passed by the message. Messages will be JSON and will indicate which counter to increment and by how much.
   * 1. Example message for the left pump counter is: {"Left" : 2}
     2. Example message for the right pump counter is: {"Right" : 3}
4. Increment the appropriate counters when the PumpLeftBLE and PumpRightBLE Characteristics are written. The counter will increment by the amount of the value written to the Characteristic.
   1. Note that the value doesn't need to be stored anywhere – the GATT Write callback just needs to increment the appropriate counter.
5. Activate the appropriate pump when that pump's counter is greater than 1.
   1. The firmware will use a varying PWM duty cycle in which it pumps faster for larger values in the counter. This will be determined through experimentation for good game play.
   2. The firmware will decrement the counter on a periodic basis until it reaches 0 at which point the pump will be stopped. The decrementing rate will also be determined through experimentation for good game play.
6. Publish MQTT messages to the "Electonica2018" Thing Shadow with water levels. The topic name will be *$aws/things/Electronica2018/shadow/update*. This can be done periodically (e.g. every 250ms) or just when the water level crosses a threshold (every 5%). An example message is:
   1. {"state" : {"reported" : {" WaterLevelLeftAWS" : 20.0, " WaterLevelRightAWS " : 25.0}}}
   2. GJL will create the *Thing* named Electronica2018 and will provide the necessary certificates.
7. Update the water levels in the WaterLevelLeftBLE and WaterLevelRightBLE Characteristics in the GATT database. This should be done on the same frequency as the MQTT publish messages (i.e. either on a periodic time bases or when water level crosses a threshold).
8. Display water level messages on the UART – use the same frequency as MQTT/BLE.
9. When one tank is full (>95%), end the game by doing the following:
   1. Shut off both pumps.
   2. Play an "end of game sound".
      1. Could make the game end sound different depending on which team wins.
   3. Flash the LEDs rapidly on/off for the side that won the game.
   4. Display an appropriate message on the UART.
10. If possible, a game "pause" feature would be useful. That is, the game master will be able to pause the game in progress and then restart after adjusting teams, etc.
11. An abort feature should be implemented. Possible ideas: kit reset, another button, wall power.

To restart the game, the kit will be reset. If feasible, the firmware will support restarting the game without reset.

## Firmware Implementation

The firmware for the main controller is RTOS thread based, with supporting state machines for some functions. Firmware implementation will be highly modular. All modules and threads will be written to be as independent as possible. There will be four global variables used for inter-thread communication: two for the liquid level readings and two pump speed increase requests.

### Game State Machine

The game state machine is handled by a thread. The game states are power-on, idle, start, running, finish, and abort. Only this thread reads the liquid level, and controls sounds, LED status, and motors. The most recent water level readings will be globals so they are available to other threads.

If feasible the *power-on* state should be implemented so that re-entering *power-on* after a game finish or *abort* resets all applicable connections and data so the game can be restarted without resetting the system.

1. Power-on:

This state performs the start-up operations as defined in the firmware operational description. On completion of these operations the state machine transitions to *idle*.

1. Idle:

During *idle* the game state machine is waiting for a start command, either pushbutton switch, or via UART, which causes a transition to the *start* state. During idle no sound is played, and all LEDs are off.

1. Start:

The *start* state triggers the start.wav sound, optionally flashes the strip LEDs to indicate game start, and drives the output for the illuminated start/stop switch active. The global motor request variables are cleared. After completion of the start sound and/or optional LED sequence, the game state machine transitions to the *running* state. During the *start* state actuating the abort button will cause a transition to the *abort* state.

1. Running:

In this state the game state machine accepts inputs from BT, AWS, and UART (debug/development use) for motor control and sets the motor driver PWMs accordingly. Bottle liquid levels are measured via capsense, and the strip LEDs updated to reflect the current liquid level. Water levels expressed as percentage are saved to global uint8 variables.

A press of start button transitions to the *pause* state.

A press of the abort button will stop the pump motors and transition to the *abort* state.

If either liquid level reaches the “win” level, the pump motors are stopped and the state machine transitions to the finish state.

1. Finish:

The finish state flashes the LED strip on the winning side in some pattern/colors TBD, and triggers the finish sound. After completion of the LED sequence and/or finish sound, the illuminated start switch is turned off, and the state machine transitions back to either *wait* or *power-on*.

1. Abort:

The abort state insures that the pump motors are stopped/disabled, and optionally gives a visual indication on the strip LEDs. After completion of any indication sequence, the illuminated start switch is turned off, any currently playing sound aborted, and the state machine transitions to either *power-on* or *wait* depending on the development path implemented.

1. Pause:

In the *pause* state, the motors are halted, and the illuminated start switch is in a blinking state. The liquid levels are read and LED strip status maintained. Incoming pump commands are ignored. Pressing the start button again transitions to the running state. An abort switch detection causes a transition to the abort state. The firmware shall include sufficient hysteresis to prevent unintended transitions between the running and pause states.

1. Wait

If it proves problematic to cleanly re-run the *power-on* state after a finish or abort, the firmware will include a wait state in which all outputs are off. This state will not transition, the user must reset the system to start a new game.

### AWS

The AWS interface is an independent thread. This thread has access to the global pump request and liquid level variables. The AWS module shall provide initialization, and if feasible, reset, functions so the game state machine can initialize the AWS module during the power-on state. The AWS module shall track its state so the thread will perform accordingly to the current state (i.e. uninitialized, running, etc).

### BT

The Bluetooth interface is an independent thread. This thread has access to the global pump request and liquid level variables. The BT module shall provide initialization, and if feasible, reset, functions so the game state machine can initialize the BT module during the power-on state. The BT module shall track its state so the thread will perform accordingly to the current state (i.e. uninitialized, running, etc).

### Motor drive

At the lowest level, two PWMs are provided to drive the pump motors through the dual h-bridge. There is also GPIO interface that controls motor direction (not used) and braking. When stopped or disabled, the GPIO is set to brake; when running it is set for the correct pump motor rotation. Support functions are provided for braking and setting the PWM duty cycle.

During game play, the game state machine thread increases the pump duty cycle a fixed amount up to 100% per count in the pump request global variable. This allows any running thread to request pump speed increases, including by scaled amounts as is desired for the remote capsense slider input. A requesting thread merely adds the scaled amount to the pump request global. When in the running *state*, the game state machine decrements the global request and increases the motor PWM duty cycle. A timer decays the duty cycle by a fixed amount until it reaches zero.

### WS2812b LED driver

The WS2812b individually addressable LED RGB strips are driven by an SPI peripheral. The data transfer is handled by DMA. The WS2812b supports 24-bit RGB data. Each bit of color information is translated to three bits of SPI data to generate the proper timing required by the LEDs. This array of nine bytes per LED is continuously transferred to the LEDs by DMA.

Support functions allow the caller to control individual LEDs by 24-bit RGB values, as well as set an entire string to a color, including off.

### Sound player

The sound player is an interrupt driven state machine. Sounds are generated by a PWM passed through a low pass filter. If in the “play” state, a timer interrupt running at 11.025 kHz either places the next sound value into the PWM duty cycle, or upon reaching the end of the sound sample, the interrupt sets the state machine status to “idle”. The state machine is non-retriggerable; sounds can only be started if the state machine is in the “idle” state. A currently playing sound can be aborted, which stops playback and sets the state to “idle”.

A support function is supplied allowing the caller to trigger a sound, abort a currently playing sound, and to read the current sound state machine status.

Sounds are included in the firmware build by placing a .wav file of the appropriate format (11.025 kHz sample rate, resolution TBD) in the resources folder of the mainapp Modus project. The .wav file is translated to an array in a .c file by the process resources script. The sound could be quickly changed during a demonstration by simply changing the .wav file in the project.

### UART

The firmware supports an asynchronous serial channel for control and debug. The various states provide debug and operational status over UART. The UART shall be configured for 115200 baud, N81.

# Questions:

1. Do we want to use Micrium to show the two tank levels on the screen? This is done in the water level sensing kit example project for a single bottle. Can we leverage that?
2. Which of the 2 liquid level sensors do we want to use? There is one with 2 sensors (backgammon style) and one with 12 sensors. Presumably the 12 sensor one is more accurate, so we should probably use that one.