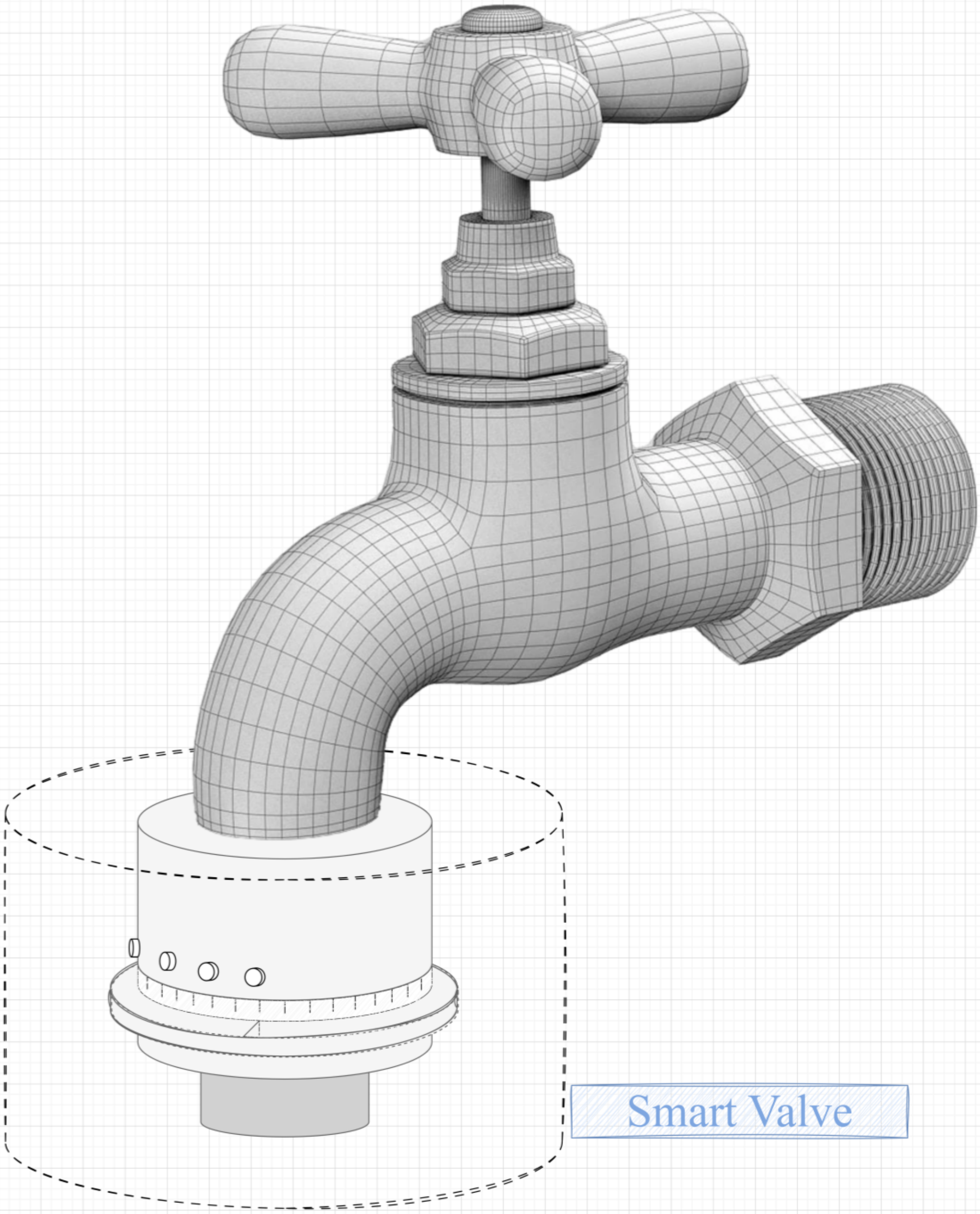
SMART VALVE



A project by

Smile Guleria

Electronics & Communication Engineering Department
National Institute of Technology Hamirpur
Hamirpur, Himachal Pradesh (India)

Ph: +91 82199-15039
e-mail: smileguleriazplus@gmail.com

TABLE OF CONTENTS

1 Problem statement

Statement: To prevent water wastage from hostel taps that are left open by negligent students during periods of regular and irregular water supply through the installation of additional accessories onto conventional tap fixtures.

As per the survey conducted, water wastage in the hostels are primarily observed in following three cases in different water supply scenarios:

- Regular water supply cases:
  a) Water bucket is kept below the tap and tap is turned on to fill the bucket with water, but most of the time tap is turned off after substantial amount of water being overflown out of bucket due to student's negligent behavior.
  b) Many times, students may want tap, to be kept turned on for some amount of time, but usually due to busy schedule or negligence, it is kept on for a longer period. It finally results in students pouring out and flushing the extra water being filled during extra time.
- Irregular water supply cases: Tap is opened by students to hand-wash or fill the bucket but is kept open in negligence when water supply is not there, but when water supply comes up then substantial amount of water is wasted until someone explicitly comes to close the tap by hearing or watching running water.

One existing solution to above stated problem is to replace all the conventional tap fixtures with sensor-based touchless tap fixtures, but it comes with it's own following set of disadvantages:

- Cost: sensor-based touchless tap fixtures are expensive compared to conventional tap fixtures. On top of fixture cost, additional installation cost as well as requirement to place to replace all the conventional tap fixtures with sensor-based touchless tap fixtures counterparts in public places like student hostels, makes sensor-based touchless tap fixtures to be an expensive and impractical solution.
- Operational limitations: Apart from handwash requirements, other daily requirements including bucket filling or installation of pipe on the top of nozzle to channelize the water flow to other locations, are not practically possible with sensor-based touchless tap fixtures.

Thus, an additional easily installable, cost-effective electronic accessory is required that can be fitted on the top of existing conventional tap fixtures nozzle to regulate the water supply for water wastage avoidance in above listed scenarios.

2 Requirement Analysis

2.1 Problem space modeling

Desired electronic accessory can be modeled as an embedded control system. It should include an electronically controlled valve on the top of tap nozzle, along with sensor to sense the hand or bucket filling. Based on defined water wastage cases in previous section, overall tap operation should be modeled as four distinct modes of operations such that there should be external provision to manually switch among these modes of operation.

S.No.	Mode	Description
1.	Handwash mode	• Use case: In this operational mode, students are expected to turn on the tap to wash their hands. • Operation: If hand is under a pre-decided proximity distance, then valve fitted at the top of nozzle end, should open, else should remain closed.
2.	Bucket mode	• Use case: In this operational mode, students are expected to keep the bucket under tap nozzle, turn on the tap and switch to bucket mode to fill the bucket. • Operation: Valve should open and bucket should be continuously sensed for it's filling and once bucket is filled, tap should automatically toggle to handwash mode followed by closing of valve. Students can also manually switch to handwash mode during filling of bucket.
3.	Timer mode	• Use case: In this operational mode, students are expected to rotate the knob of regulator to desired timing value, turn on the tap and switch to timer mode to start the timer. • Operation: Valve should open and remain open until desired timing is reached. Once desired time is reached, tap should automatically toggle to handwash mode followed by closing of the valve. Students can also manually switch to handwash mode before reaching of desired timing.
4.	Static mode	• Use case: In this operational mode, students are expected to fit the pipe on the top of nozzle end, and switch to static mode to channelize the water flow to other locations. • Operation: Valve should open and remain open until student doesn't explicitly switch to handwash mode or other operation mode.

2.2 Input/Output requirements

Handwash mode is most frequently used mode of operation, so handwash mode should be kept default mode of operation, whereas for other modes, manual switching is expected from student's end.

To manually switch to bucket, timer and static mode, three distinct push buttons shall be required. Students can switch to these modes and move back to default handwash mode by pressing on same push button.

Apart from push buttons, a potentiometer will be required as a regulator to tune to desired timing value. A proximity distance measuring sensor will be required to implement hand sensing and bucket filling logic. In addition to it, a servo motor will also be required to have controlled action of valve.

Other explicit requirements include LED indicators to represent the entry and exit into a distinct operational mode. Based on above stated requirements, following inputs and output are listed:

- Inputs:
  a) A push button to switch to bucket mode.
  b) A push button to switch to timer mode.
  c) A push button to switch to static mode.
  d) A potentiometer to set the desired timing value.
- Outputs:
  a) Opening or closing of valve
  b) Turning on/off mode's LEDs.

2.2 Assumptions and constraints

Following assumptions and constraints are considered while doing the requirement analysis:

a) After system initialization, controller of embedded control system should go to default handwash operation mode.
b) In handwash mode, valve remains close by default.
c) Position of bucket remains fixed during filling of the bucket.
d) Time duration between stoppage and re-start of water supply is considerably long, so controller goes to default handwash mode if such stoppage of water supply is encountered in the middle of bucket filling in bucket mode.

2.3 Behavioral representation

It is evident from previous sections that required embedded control system is reactive and sequential in nature. Thus, its behavioral modeling is done through finite state machine as follows:

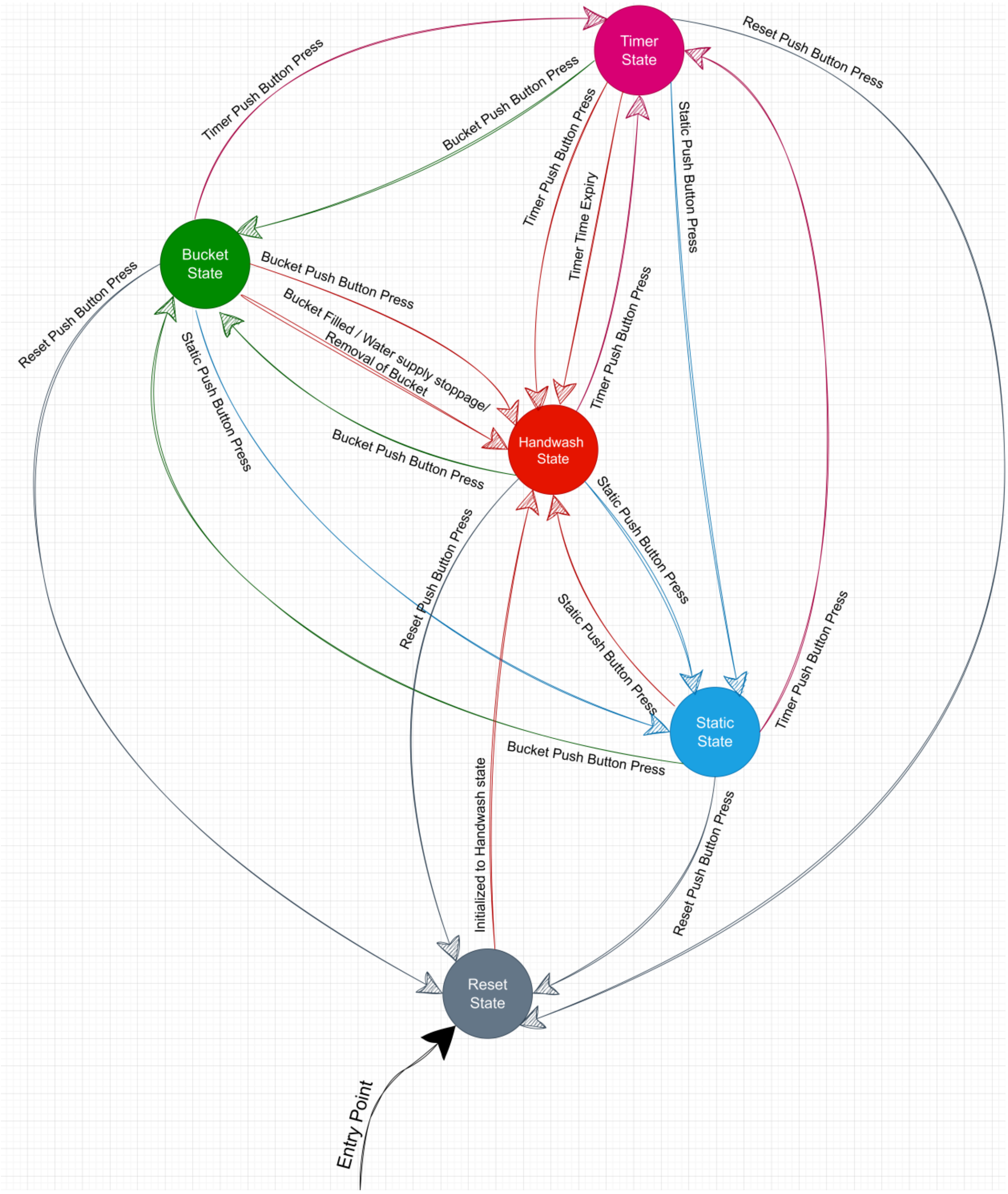


Fig. 1 Finite state machine representation for behavioral modeling of required embedded control system.

3 Design and Implementation

3.1 Prototyping

Platform based design is used for rapid prototyping of the required embedded control system. Arduino Uno R3 is used for rapid prototyping with Atmega380p micro-controller as controller. Other components are listed as follows:

a) SG90 Servo motor is used for controlled valve action.
b) HC-SR04 ultrasonic sensor is used as proximity distance measuring sensor.
c) RGB LEDs as LED indicators.
d) Potentiometer as timer's timing regulator.

3.2 Firmware architecture

Following firmware architecture is designed to run on controller to implement finite state machine behavioral representation specified in section 2.3, so that all the requirements listed in section 2.1 – 2.4 are catered.

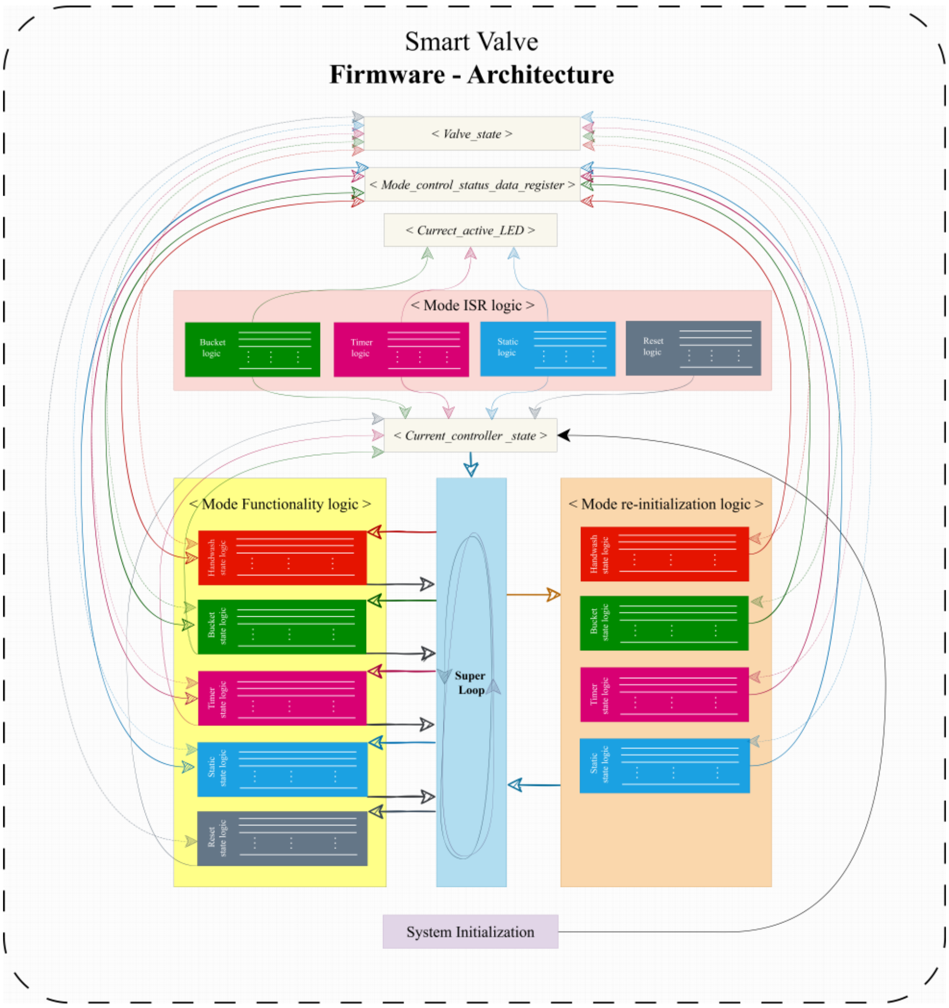


Fig. 2 Firmware architecture designed to implement the finite state machine behavioral representation of desired embedded control system.

Firmware architecture specified in Fig. 1 is described further in the terms of data-structures used as well as independent logics meant for specific mode's operation:

- Data-structures: Four primary data-structures are defined as follows:
  a) <*Current_controller_state*> : It defines currently active state from finite state machine described in section 2.3. As per this data-structure, <Super-Loop logic> decides which mode functionality's logic is required to be executed as well as requirement to invoke <mode re-initialization logic>. Sources and corresponding scenario's in which *Current_controller_state* can be modified in run-time, are listed as follows :

     1. During system initialization: Once system initialization is done and mode is required to be reverted to default handwash mode then *Currrent_controller_state* will be modified to handwash state
     2. From <mode functionality logic>: When a certain mode's functionality is completed and mode is required to be reverted to default handwash mode then *Current_controller_state* will be modified to handwash state.
     3. From <mode ISR logic>: When a push button is pressed by students to either switch to a distinct mode or revert back to handwash mode then an interrupt will be generated and corresponding interrupt's ISR will modify *Current_controller_state* as per the currently ongoing execution.

  b) < *Currect_active_LED* > : There are total 3 LEDs corresponding to Bucket, Timer and Static state. Controller can only be present in one of the state at a time as defined in finite state machine of section 2.3. There-fore only one LED can be active at a time. This data structure defines currently active LED. It can be modified by <mode ISR logic> as per push button being pressed by the student.

  c) <*Mode_control_status_data_register*> : *Mode_control_status_data_register* data-structure defines the status and control bits as well as control data, used by <mode_functionality_logic> during mode's functionality. As only controller can only be in one state as per finite state machine defined in section 2.3, so there can only be one active mode and correspondingly only currently executing mode's control, status and data can be operational. Once state transition has occurred, previously running mode's control, status and data can be simply over-ridded. Thus a single *Mode_control_status_data_register* is defined for all the modes. There are two sources and corresponding scenarios in which, it can be modified in run time:
     1. From <mode functionality logic>: During mode's functional execution, it can be modified by corresponding mode's <mode functionality logic>
     2. From <mode re-initialization logic>: During mode's re-initialization, it can be modified as per mode's initialization state by corresponding mode's <mode re-initialization logic>

  d) < *Valve_state* > : *Valve_state* data-structure defines the current valve state. It can take two values, one depicts valve to be open where as depicts it to be closed. As soon as this data structure is modified, servo motor actuation is performed for actual valve actuation to either open or closed as per modified value. It can be modified by <mode re-initialization logic> or <mode_functionality_logic>.

- Algorithm : The finite state machine described in Section 2.3 is implemented through four independent logic components: state functionality logic, external event logic, state transition logic, and a central coordinator logic. Each state possesses its own state functionality logic, which is executed when the controller operates in the corresponding state. External event logic is activated upon the reporting of an external event. This event logic determines the subsequent state to which the controller should transition, based on the reported event. Once the subsequent state to be executed has been determined by either the external event logic or the state functionality logic, the continuously looping control flow is transferred to the state transition logic by the central coordinator logic (if required), to facilitate the transition to the initial conditions required for the designated subsequent state's functionality logic.

  In the implemented firmware architecture, all state functionality logics, external event logics , state transition logic and central coordinator logic are segregated as follows:

  a) *<Mode Functionality logic>*: For segregating all the state functionality logic.
  b) *<Mode re-initialization logic>*: For segregating all the state transition logic.
  c) *<Mode ISR logic>*: For segregating all the external event logic.
  d) *<Super Loop>*: For central coordinator logic.

3.2 Firmware implementation

Firmware implementation is described for data structure and associated algorithm (described in section 3.2.) as follows:

- Data-Structures:

  <Current_controller_state> : C_state.present
  <Current_mode>: modSel.active
  <Capured_mode>: modSel.capture
  < Currect_active_LED >: LED.present
  <Mode_control_status_data_register> : cap
  <Valve_state> : valve_state

  a) Controller state definition:

```
//-------------Controlstate_reg definiton--------------
//definition
typedef enum {
    handwash_state,
    bucket_state,
    timer_state,
    static_state,
    reset_state,
    null_state,
}state;

typedef struct {
    state present;
    state past;
}state_reg;

//declaration
state_reg C_state;// C_state refers to controller state here
```

Controller can be in all these possible states.

- *Present defines the currently active controller state.*
- *Past defines just previously active controller state.*

b) Mode selection definition:

```
typedef struct{
    state active;
    state capture;
}modeselection;
//declaration
modeselection modeSel;
```

- *Active defines the currently running mode of operation, it is same as that of present controller state.*
- *Capture defines last the mode of operation decided by external event logic. It is same as that of active, until an external event logic or static functionality logic changes it to reflect the change in state.*
- *Followed by change in state, active as well as controller's present and past state will also be modified.*

c) LED indicators definition:

```
typedef struct{
    const int Bucket = 9;
    const int Timer = 8;
    const int Static = 7;
    int present = 0;//to check the LED which is currently turned on
}indicators;
indicators LED;
```

- *It includes GPIO pin number to which corresponding LED is connected.*
- *Present is assigned the pin number of currently active LED*
- *Current LEDs turn on/off action is made through the pin number stored in present.*

d) Control, status bits and control data definition:

```
//------------Capability_reg definiton-----------------
//definition
typedef struct {
    bool handwash_status_bit;
    bool proximity_bit;
    int max_hand_proximity;
}handwash_cap;

typedef struct{
    int decremental_rate;
    int past_value;
}filling_info;

typedef struct {
    bool bucketfull_bit;
    bool buckfilling_bit;
    filling_info filling_stat;
}bucket_cap;

typedef struct {
    bool timer_status_bit;
    bool timer_start_bit;
    int timer_input;
}timer_cap;

typedef struct {
    bool static_status_bit;
    bool static_running_bit;
    bool static_input;
}static_cap;

typedef struct {
    bool status_bit;
    bool control_bit;
    long data;
}reg;

typedef union {
    reg cap_reg;
    handwash_cap handwash;
    bucket_cap bucket;
    timer_cap timer;
    static_cap static_mode;
}capability_reg;
//declaration
capability_reg cap;
```

- *handwash_status_bit : status bit of handwash mode capability register.*
- *Proximity_bit : control bit of handwash mode capability register.*
- *max_hand_proximity: control data for handwash mode functionality.*

- *bucketfull_bit : status bit of bucket mode capability register.*
- *bucketfilling_bit : control bit of bucket mode capability register.*
- *filling_stat: control data for bucket mode functionality.*

- *timer_status_bit : status bit of timer mode capability register.*
- *timer_bit : control bit of timer mode capability register.*
- *filling_stat: control data for timer mode functionality.*

- *static_status_bit : status bit of static mode capability register.*
- *static_running_bit : control bit of static mode capability register.*
- *static_input: placeholder dummy data for future scope.*

- *With union only, one capability register active at a time.*
- *A generic cap_reg defined to refer to any capability register's status, control bits and control data.*

e) Valve state definition:

```
bool valve_state;
```

- *False value indicates, valve to be closed, whereas true value indicated valve to be open*

e) Servo motor declaration for valve:

```
Servo valve;
```

- Algorithms:

  Independent logics defined in firmware architecture in section 3.2, uses few generic functions defined as follows:

  a) Valve-control:

```
void valve_action(bool valve_state){
    int open_angle=90; //measured in degree
    int closing_angle=0; //measured in degree
    if(valve_state){
        //opening of valve
        valve.write(open_angle);
        //waiting for valve to get there
        delay(15);
    }
    else{
        //closing of valve
        valve.write(closing_angle);
        //waiting for valve to get there
        delay(15);
    }
};
```

  b) Proximity distance measurement:

```
float d_measure(){
    digitalWrite(proximity.trigPin, LOW);
    delayMicroseconds(2);
    digitalWrite(proximity.trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(proximity.trigPin, LOW);

    float duration = pulseIn(proximity.echoPin, HIGH);
    float distance = (duration*.0343)/2;
    return distance;
};
```

  c) Control, valve state and mode toggling:

```
void toggleModeSelectionInput(){
    //toggling back to handwash_state
    modeSel.active=handwash_state;
    digitalWrite(LED.present,LOW);
    LED.present=0;
};


//toggle control bit of capability register
void toggleControlBit(){
    cap.cap_reg.control_bit=!cap.cap_reg.control_bit;
};
//toggle valve state
void toggleValveState(){
    valve_state!=valve_state;
};
```

  Independent logics defined in firmware architecture are defined as follows:

  a) *<Mode Functionality logic>* :

  1. Handwash state logic:

     In handwash mode, an ultrasonic sensor continuously sense the proximity distance, and as soon as sensed distance becomes less than a threshold value, valve state is toggled, and corresponding valve action is made. As soon as sensed distance becomes greater than threshold, valve state is again toggled followed by corresponding valve action. This process should go on continuously until a manual switch is not made to another mode of operation it:

     Implementation:

     Mode specific data structure used:

```
typedef struct{
    int max_hand_proximity_macro = 20.00;
    int max_proximity = 300.00;
    int trigPin = 5;
    int echoPin = 6;
}HCSR04;
volatile HCSR04 proximity;
```

Code implementation:

```
bool isInProximity(){
    return (d_measure()<=cap.handwash.max_hand_proximity);
};

void handwash_handler(){
    if(isInProximity()!= cap.handwash.proximity_bit){
        toggleControlBit();
        toggleValveState();
        valve_action(valve_state);
    }
};
```

2. <u>Bucket mode logic:</u>

Bucket to be kept below the tap followed by pressing of bucket mode push button, valve should be opened and subsequently ultrasonic sensor should start sensing the distance from water interface from nozzle of the tap.

- At first measurement sample, two distances should be measured after a certain distance measuring delay.
- Difference of two measured distances should be calculated and stored as a decremental rate.
- Each subsequent measurement sample should be taken after same measuring delay.
- If measured distance comes out to be greater than the expected distance for a consecutive pre-defined number of samples then three cases are indicated:

(Note: expected distance should be calculated from previously measured distance, decremental rate as well as certain threshold compensation for minimum water flow to account for decrease in water flow)

> Bucket is filled.
> Water supply is gone or less the the minimum allowable water flow.
> Bucket is removed in the middle of water filling by the student.

In all these cases, tap is expected to automatically switch to handwash mode operation followed by handwash mode re-initialization which causes closing of valve.

<u>Implementation:</u>

```
bool isBucketFull(){
    if(cap.bucket.buckfilling_bit==0)
        return 0;
    if(cap.bucket.filling_stat.decremental_rate==0){
        int d1=d_measure();
        delay(decremental_cal_delay);
        int d2=d_measure();
        cap.bucket.filling_stat.decremental_rate=d1-d2;
        if(cap.bucket.filling_stat.decremental_rate==0){
            cap.bucket.bucketfull_bit=1;
            return 1;
        }// to include the case of no water supply initially
        cap.bucket.filling_stat.past_value=d2;
    }
    delay(decremental_cal_delay);
    if(d_measure()>cap.bucket.filling_stat.past_value-(max_water_flow_comp*cap.bucket.filling_stat.decremental_rate)){
        cap.bucket.bucketfull_bit=1;
        return 1;
    }
    else
        return 0;
};
void bucket(){

    if(isBucketFull()==cap.bucket.buckfilling_bit)
    {
        toggleControlBit();
        toggleValveState();
        valve_action(valve_state);
    }
};
void bucket_handler(){
    if(cap.bucket.bucketfull_bit==0)
        bucket();
    else
        toggleModeSelectionInput();
};

```

3. Timer mode logic:

Potentiometer representing the regulator to tune timing requirements, should be rotated to an appropriate position, followed by pressing of timer push button. Value should be sampled from potentiometer followed by scaling it to appropriate time value. Further, Timer interrupt should be enabled, and timer should be initialized to zero. Total number of timer overflow interrupts should be monitored and the time it reaches the required number of calculated overflows as per desired time value, timer interrupt should be disabled, and tap is expected to automatically switch to handwash mode.

Implementation

Mode specific data-structure used:

```
typedef struct{
  int tickCount = 0; //Holding total number of timer interrupt overflows required to have the delay as per scaled potentiometer value
  int tickStep = 0; // Holding current no of timer interrupt overflows.
  bool set = false; // a boolean value to indicate if timer is initialized and corresponding tickCounts have been calculated.
  bool met = false; // a boolean value to indicate if intentional amount of delay has been occured or not.
}timer_control_data;
timer_control_data tConData;// a structure to hold the timer control data

```

Code implementation:

```
//----------------------------------------------------------timer capability------------------
bool isTimesUp(){
    if(cap.timer.timer_start_bit==0)
        return 0;
    if(tConData.met){
        cap.timer.timer_status_bit=0;
        return 1;
    }
};
void timer(){
    if(isTimesUp()==cap.timer.timer_start_bit){
        toggleControlBit();
        toggleValveState();
        valve_action(valve_state);
    }
};

void timer_handler(){
    if(!tConData.set){
       tConData.tickCount=0.41946*cap.timer.timer_input;
       TCNT1 = 0; //initializing timer1 counting from 0
       TIMSK1 |= B00000001; //enabling Timer Overflow Interrupt
       cap.timer.timer_status_bit==1; // enabling the timer_status
      tConData.set=!tConData.set; //toggling set boolean variable to disable
                                  //the re-enabling of timer over flow interrupt
                                  //as well as other re-assignments in the continuous loop
    }
    if(cap.timer.timer_status_bit==1){
        timer();
    }
    else
        tConData.set=!tConData.set;
        tConData.met=!tConData.met;
        tConData.tickCount=0;
        tConData.tickStep=0;
        toggleModeSelectionInput();
};
```

4. Static mode logic:

Static push button should be pressed to enter to static mode. Once entered, valve will open and it will remain open until same push button is not explicitely pressed again.

Implementation

```
//-----------------------------------static_capability----------

void static_handler(){
    if(!cap.static_mode.static_running_bit){
        toggleControlBit();
        toggleValveState();
        valve_action(valve_state);
    }
};

```

5. <u>Reset mode logic:</u>

Reset push button should be pressed to enter to reset mode. This mode causes, all LEDs re-initialization, valve reinitialization, controller state reinitialization as well as mode re-initialization to handwash mode.

<u>Implementation</u>

```
//----------------------Reset Capablity-----------------------------------------------
void reset_handler(){

    // LEDs re-initialization
    digitalWrite(LED.Bucket,LOW);
    digitalWrite(LED.Timer,LOW);
    digitalWrite(LED.Static,LOW);
    LED.present=0;//No LED is selected initially

    // Valve re-initilization
    valve_state=false;// False valve_state mean valve is open
                      // and true valve_state mean valve is closed.
    valve_action(valve_state);//initializing valve state to be closed

    // Controller state re-initialization
    C_state.present=null_state;
    C_state.past=null_state;

    // Mode re-initialization
    modeSel.capture=handwash_state;
    modeSel.active=handwash_state;//initialized to handwash state ->
   //control,status and data fields are not required as they will be done by reinit()
    Serial.println("Reinitialization done > Cuurent Operational state: handwash_state");

};

```

b) *<Mode re-initialization logic>* :

```
//function to reinitialize Capability Register when a state change occurs
void reinit(int present){
    switch(present){
        case handwash_state:
            cap.handwash.handwash_status_bit=1;
            cap.handwash.proximity_bit=0;
            cap.handwash.max_hand_proximity=proximity.max_hand_proximity_macro;
            if(valve_state){
                valve_state=false;//initializing valve state to be closed
                valve_action(valve_state);
            }
            Serial.println("Entering Handwash mode...");
            break;

        case bucket_state:
            cap.bucket.bucketfull_bit=0;
            cap.bucket.buckfilling_bit=0;
            cap.bucket.filling_stat.decremental_rate=0;
            cap.bucket.filling_stat.past_value=0;
            if(valve_state){
                valve_state=false;//initializing valve state to be closed
                valve_action(valve_state);
            }
            Serial.println("Entering Bucket mode...");
            break;

        case timer_state:
            cap.timer.timer_status_bit=1;
            cap.timer.timer_start_bit=0;
            cap.timer.timer_input=analogRead(timer_input_pin);//initializing to the value of potentiometer
            if(valve_state){
                valve_state=false;//initializing valve state to be closed
                valve_action(valve_state);
            }
            Serial.println("Entering Timer mode...");
            break;

        case static_state:
            cap.static_mode.static_status_bit=1;
            cap.static_mode.static_running_bit=0;
            cap.static_mode.static_input=0;
            if(valve_state){
                valve_state=false;//initializing valve state to be closed
                valve_action(valve_state);
            }
            Serial.println("Entering Static mode...");
    }
};
```

c) *<Mode ISR logic>* :

1. Bucket state logic:

This interrupt service routine is invoked by halting the current ongoing execution when bucket mode's push button is pressed by the students. First it updates the captured mode selection to bucket_state followed by comparing it with active mode selection. If captured and active mode selection are same, then it implies

>The case of already running bucket mode, such that bucket is not filled upto it's brim but student has decided to stop the bucket filling.

>It should result in corresponding bucket mode's LED's turning off and subsequent automatic transition of operational mode to handwash mode.

However, if captured mode selection is different mode is different from active mode selection then it implies

>The case of manual switching from another already running mode (Timer or Static or Handwash mode) to bucket mode.

>It should result in corresponding running mode's LED's turning off and subsequent automatic transition of operation mode to bucket mode.

Implementation

```
ISR(PCINT1_vect){
    bucket_isr();
};
void bucket_isr(){
        modeSel.capture=bucket_state;
        if(modeSel.active==modeSel.capture){
            modeSel.active=handwash_state;
            digitalWrite(LED.present,LOW);
            LED.present=0;
            Serial.println("Leaving Bucket mode...");
        }
        else{
            modeSel.active=modeSel.capture;
            if(LED.present!=0){
                digitalWrite(LED.present,LOW);
            }
            LED.present=LED.Bucket;
            digitalWrite(LED.present,HIGH);
        }
};
```

2. Timer state logic:

This interrupt service routine is invoked by halting the current ongoing execution when timer mode's push button is pressed by the students. First it updates the captured mode selection to timer_state followed by comparing it with active mode selection. If captured and active mode selection are same, then it implies

>The case of already running timer mode, such that timing has not reached upto desired timing value but student has decided to stop the timer.

>It should result in disabling of timer overflow interrupt followed by corresponding bucket mode's LED's turning off and subsequent automatic transition of operational mode to handwash mode.

However, if captured mode selection is different mode is different from active mode selection then it implies

>The case of manual switching from another already running mode (Bucket or Static or Handwash mode) to timer mode.

>It should result in capturing of current timer value from potentiometer and storing it in timer mode capability register's control data. Currently running mode's LED should be turned off and subsequently operational mode should be transitioned to timer mode.

Implementation

```
ISR(PCINT0_vect){
    timer_isr();
};

void timer_isr(){
        modeSel.capture=timer_state;
        if(modeSel.active==modeSel.capture){
            TIMSK1 = B00000000; // disable timer overflow interrupt
            modeSel.active=handwash_state;
            digitalWrite(LED.present,LOW);
            LED.present=0;
            Serial.println("Leaving Timer mode...");
        }
        else{
            modeSel.active=modeSel.capture;
            //capture the current timer value from potentiometer and store it in cap.timer.timer_input
            cap.timer.timer_input=analogRead(timer_input_pin);
            if(LED.present!=0){
                digitalWrite(LED.present,LOW);
            }
            LED.present=LED.Timer;
            digitalWrite(LED.present,HIGH);
        }
};
```

3. Static state logic:

This interrupt service routine is invoked by halting the current ongoing execution when static mode's push button is pressed by the students. First it updates the captured mode selection to static_state followed by comparing it with active mode selection. If captured and active mode selection are same, then it implies

>The case of already running static mode, and student has decided to stop it.

>It should result in corresponding static mode's LED's turning off and subsequent automatic transition of operational mode to handwash mode.

However, if captured mode selection is different mode is different from active mode selection then it implies

>The case of manual switching from another already running mode (bucket or timer or handwash mode) to timer mode.

>It should result in currently running mode's LED turning off and subsequent operational mode transition to static mode.

Implementation

```
ISR(PCINT2_vect){
    static_isr();
};
void static_isr(){
        modeSel.capture=static_state;
        if(modeSel.active==modeSel.capture){
            modeSel.active=handwash_state;
            digitalWrite(LED.present,LOW);
            LED.present=0;
            Serial.println("Leaving Static mode...");
        }
        else{
            modeSel.active=modeSel.capture;
            if(LED.present!=0){
                digitalWrite(LED.present,LOW);
            }
            LED.present=LED.Static;
            digitalWrite(LED.present,HIGH);
        }
};
```

4. Reset state logic:

This interrupt service routine is invoked by halting the current ongoing execution when static mode’s push button is pressed by the students. It simply modifies active mode selection to reset_state.

Implementation

```
ISR(INT0_vect){
    modeSel.active=reset_state;
};
```

d) <*Super-Loop*> :

```
//------------------------Array of function pointers to invoke mode functionality logic handlers------------------------------
void (*state_handler[])() ={&handwash_handler,&bucket_handler,&timer_handler,&static_handler,&reset_handler};

// function to run Statemachine
void runStateMachine(){
    if(C_state.past!=C_state.present && C_state.present!=reset_state){
        reinit(C_state.present);
        C_state.past=C_state.present;
    }
    //invoke the appropriate handler
    state_handler[C_state.present]();
};

void loop(){
    switch(modeSel.active){
        case handwash_state:
            C_state.present=handwash_state;
            break;
        case bucket_state:
            C_state.present=bucket_state;
            break;
        case timer_state:
            C_state.present=timer_state;
            break;
        case static_state:
            C_state.present=static_state;
            break;
        case reset_state:
            C_state.present=reset_state;
            break;
    }
    runStateMachine();
};
```

e) <*System initialization* > :

System specific data-structures definition:

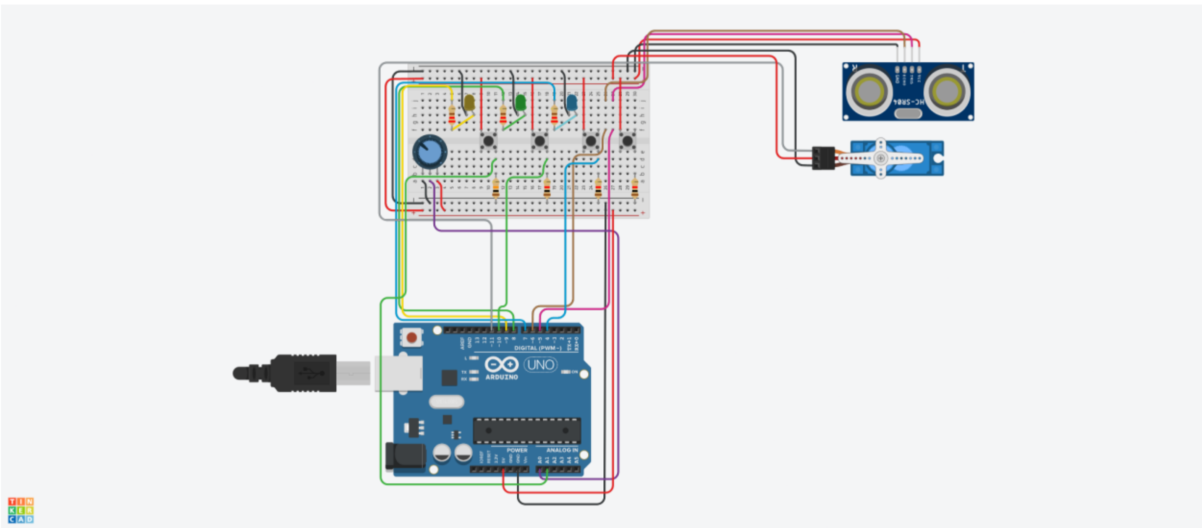
```
//-----------------------------------------------------------------------------------------
//arduino micro specific definitions

#define decremental_cal_delay 10000 // 10s is taken as arbitirary value
#define max_water_flow_comp 0.8 // 20% variability in flow rate

//pin numbers
typedef struct{
    const int Bucket = 15;//PCINT13 -> PCIE1[14:8] -> PCMSK1
    const int Timer = 10;//PCINT2 -> PCIE0[7:0] -> PCMSK0
    const int Static = 4;//PCINT20 -> PCIE2[23:16] ->PCMSK2
    const int Reset = 2;//External Interrupt -> INT0
}ir;
const ir interrupt;

const int valve_pin = 11;
const int timer_input_pin = A0;

typedef struct{
    int max_hand_proximity_macro = 20.00;
    int max_proximity = 300.00;
    int trigPin = 5;
    int echoPin = 6;
}HCSR04;
const HCSR04 proximity;
```

Code Implementation for system initialization:

```
void setup(){
    //Serial print for debugging
    Serial.begin(9600);
    Serial.println("Initialization started...");

    //pinmodes configuration
    pinMode(LED.Bucket, OUTPUT);
    pinMode(LED.Timer, OUTPUT);
    pinMode(LED.Static, OUTPUT);//for LEDs

    pinMode(proximity.trigPin, OUTPUT);
    pinMode(proximity.echoPin, INPUT);//for HCSR04 ultrasonic sensor

    pinMode(valve_pin,OUTPUT);//for servo motor to control the valve

    pinMode(timer_input_pin,INPUT);//for potentiometer-> timer input

    pinMode(interrupt.Bucket, INPUT_PULLUP);
    pinMode(interrupt.Timer, INPUT_PULLUP);
    pinMode(interrupt.Static, INPUT_PULLUP);
    pinMode(interrupt.Reset, INPUT_PULLUP);//for push-button interrupts

    //defining and attaching servo motor to valve_pin:
    valve.attach(valve_pin);

    //attaching interrupts----->

    //Pin change interrupts :
    PCICR=0b00000111;//enable PCIE0,PCIE1,PCIE2 for bucket, timer,static
    PCMSK0=0b00000100;//PCINT2
    PCMSK1=0b0000010;//PCINT9
    PCMSK2=0b00010000;//PCINT20
    //External Interrupts"
    EICRA=0b00000011;
    EIMSK=0b00000001;//enabling int0

    //Initializing timer interrupts's configuration:
    TCCR1A=0; //Init Timer1
    TCCR1B=0; //Init Timer1
    TCCR1B |= B00000101; //Prescalar = 1024

    sei();//enabling interrupts gloabally

    reset_handler();

};
```

4 Logic Verification

TINKER CAD circuit simulation is used for logic verification with following circuit schematic:



Code Reference

Please refer to source code directory for reference.