Swift Act Internship Project

Electric Water Heater

Submitted by: Mostafa Amgad Ahmed

Faculty: Faculty of Engineering

University: Ain Shams University

Major: Computer

Academic Year: Senior

Table of Contents

1. Project Description:

A Layered architecture design that consists of:

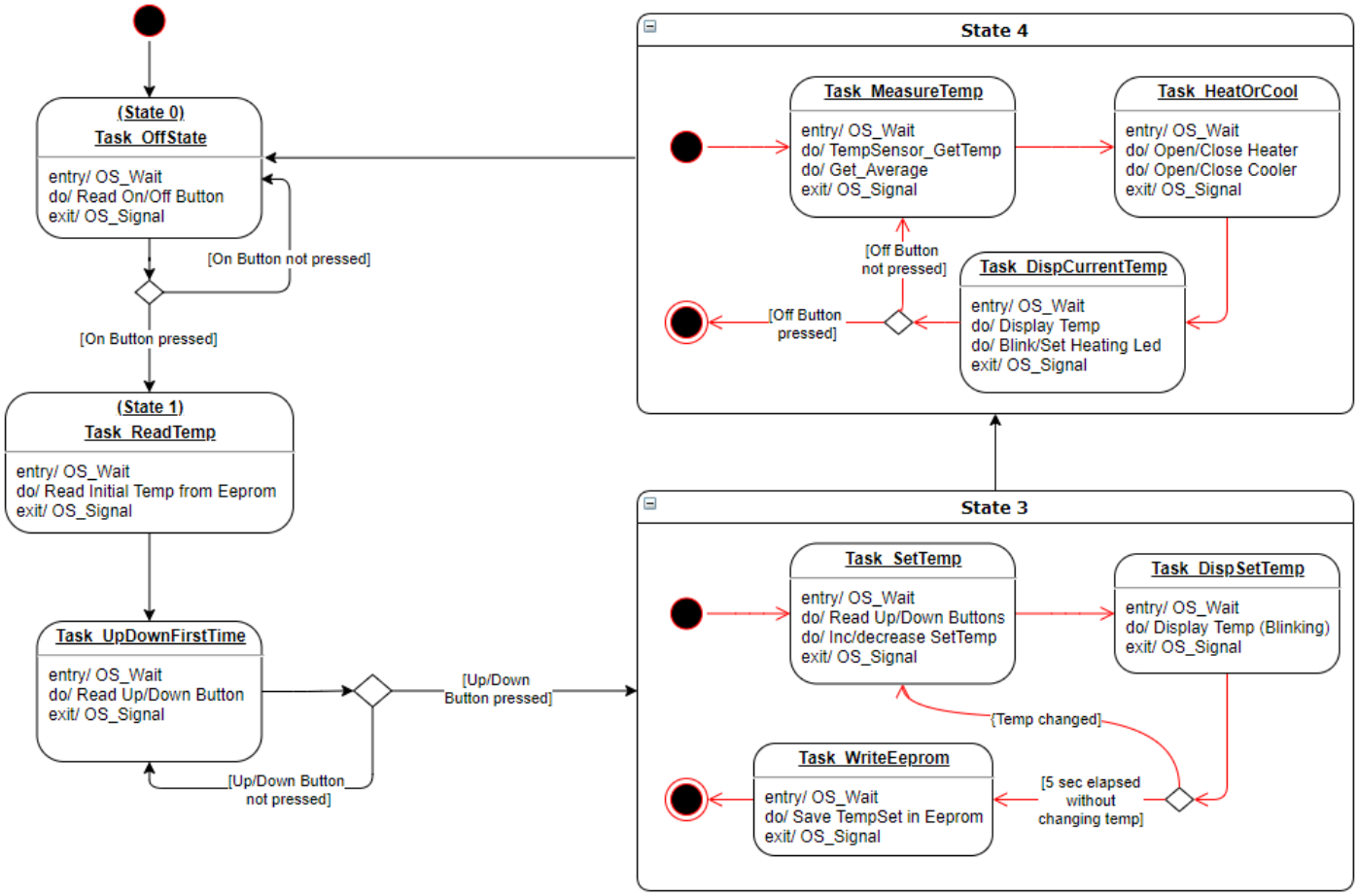
- Application Layer: a Finite State Machine design where each state contains one or more tasks.
- Operating System: I included two designs:
  - A configurable Linked-List based OS using a preemptive Round-Robin scheduler. Tasks have two layers of protection: semaphores for each global variable (wait & signal fns) and critical sections. Preemption is accomplished using context-switching macros implemented in assembly instructions, and the context of the task is saved in a stack provided for each task.
  - A counter-driven OS using a tick counter to calculate the time elapsed and decide which task to run based on a timeline specified by the user.
- All MCAL modules are implemented following the AUTOSAR Specifications with pre-compile and post-build configurations. The user specifies the desired configurations in a structure which is sent as a parameter in the Init function. The implemented modules are: Port, Dio, Adc, Pwm, I2c, Gpt, and Eeprom(HAL). I took it upon myself, as a challenge, to support as many features required by AUTOSAR as possible using the PIC16F; any features not supported by hardware are realized in software.
- Some HAL modules such as: TempSensor, TempSystem (Heater & Cooler) and Display (7 segments).
- A Default Error Tracing (Det) module that permits the user to discover any development errors (ex: using a module's api without initialization), also following AUTOSAR Specs. The Det module reports the error id, the api id and the module id where the error occurred. Then, it sends it through the serial terminal using the Uart module. And finally, it forces the microcontroller to enter Sleep mode.

2. Project Specification:

2.1. Application Layer:

2.1.1. Finite State Machine:

The figure down-below shows the FSM implementation of the App layer. Flags are used as guard conditions along with the scheduler to switch form one state to another.



2.1.2. States Specification:

State 0:

- *Task_OffState():* waits for the On button to be pressed to request the OS to change the current state.

State 1:

- *Task_ReadTemp()*: it's only a transitional state where it reads the saved value in a specified address in Eeprom. If the data in the Eeprom is invalid (hasn't been saved yet), it loads 60. Then proceeds to the next state directly.

State 2:

- *Task_UpDownFirstTime()*: waits for the Up/Down button to be pressed one time to enter the Setting Temperature State.

State 3:

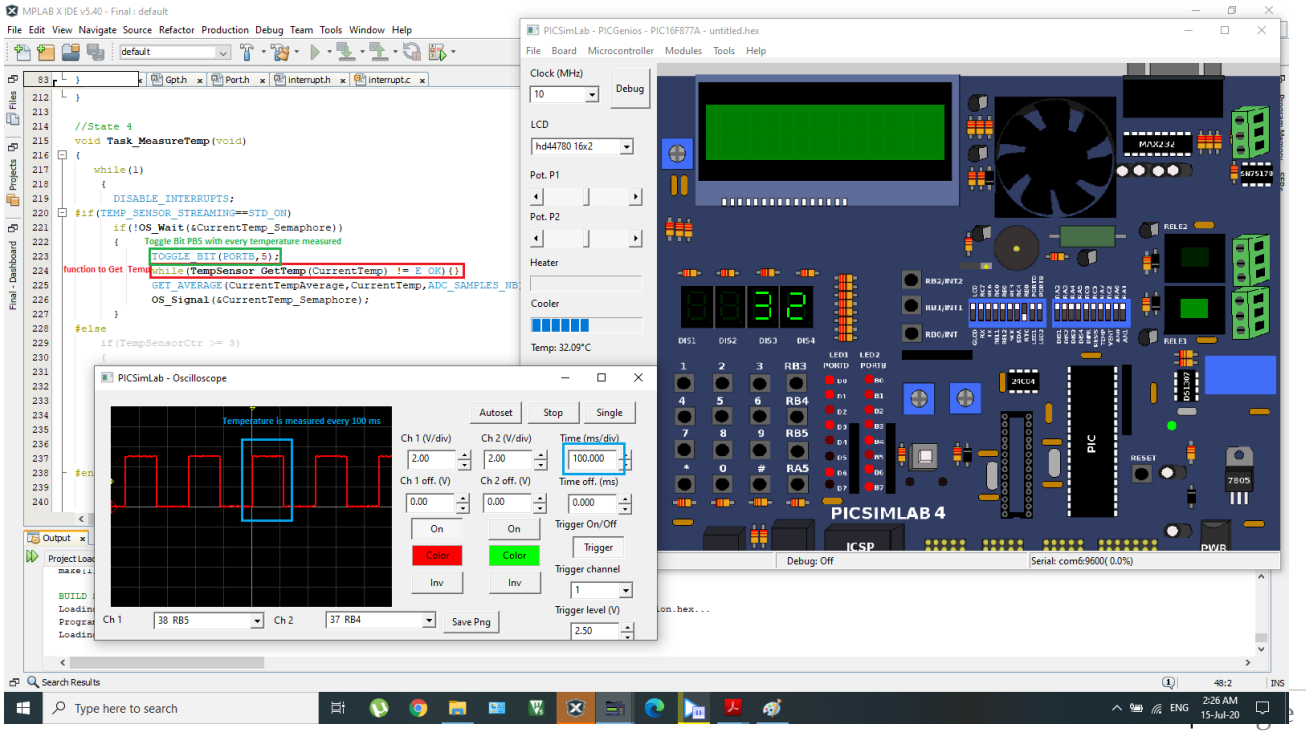
- *Task_SetTemp()*: user changes the SetTemp using Up/Down buttons.
- *Task_DispSetTemp()*: display SetTemp on the 7 segments (Blinking). (The application is stuck at State 3 executing these 2 tasks one after the other while a counter increments every tick to count the time elapsed since the last button is pressed, when this counter reaches a value equivalent to 5 seconds passed, we proceed to the 3rd task)
- *Task_WriteEeprom()*: a transitional state where we save the SetTemp value in the Eeprom to be loaded later, then proceeds the final State.

State 4:

- *Task_MeasureTemp()*: our ADC is configured in streaming mode; we call the function GetTemp, then the last set of samples are loaded in the array CurrentTemp. We use it to calculate CurrentTempAverage. The Timer is configured to call the TickHandler every 100ms so that the OS schedules this task to run every 100ms for the TempSensor to take readings correctly (as shown in the figure down-below).
- *Task_HeatOrCool():* it decides based on CurrentTempAverage whether to open the heater(dio) or the cooler(pwm).
- *Task_DispCurrentTemp():* display the CurrentTempAverage on 7 segments, blink the heating led when the heater is on and set it when the cooler is on.

(The temp continues oscillating until the On/Off button is pressed and the application is stopped)

***Note:** at any moment during runtime, when the On/Off button is pressed the system stops until the On/Off button is pressed again so that the system can restart.



2.1.3. Some important notes:

- On/Off button is mapped to RB1.
- For the 1st OS design: Up/Down are mapped to RB2/RB0 respectively, for the 2nd they are mapped to RB3/RB4.
- It's very important to know that I used a PRO compiler (XC8) to reach optimization level –s, without it the code won't compile, because after optimization the code uses 97% of data memory and 91% of program memory. That's because the project's configurations is based on struct with multiple members.
- The modules are arranged in folders corresponding to their layer.
- I added two extra files: a file.c for the post-build configurations that contains the ConfigStructs, a file.h for the pre-compile configurations which contains some preprocessor directives to turn On/Off some apis or specifies the maximum numbers of channels for each module. These files are usually generated using a GUI.
- I chose to include a PWM module in my design to be used with the temperature system (cooler & heater). I use it to control the speed of the increase/decrease of the temperature and because turning On/Off the heater doesn't happen abruptly in reality as in the simulation; that's going to cause the temperature over the specified limit.
- For the code to fit the memory, some features are turned off using #define EXTRA_FEATURES, but they work fine.

2.2. Operating System:

2.2.1. First Design:

❖ OS Entities:

- **Tcb_Array[TASKS_NB]:**
  The main entity in the OS. An array of TCBs where every task has a corresponding TCB (structure).
  Members of TCB struct:
  - Task_Id: Id of task
  - State : Which State includes this task
  - StateChangeTrigger: trigger that is set to change state
  - void (*Task) (void); pointer to task function
  - struct TCB *Next_Task; linked-list pointer, points to the next task
  - int8_t *Blocked_Sem; pointer to blocked resource
  - uint8_t stack[5]; a stack for every TCB used in context-switching

- **OS_ConfigType:**
  I designed the OS to be configurable for every user.
  Members of OS_Config struct:
  - NbOfStates: the user specifies the number of states for his FSM.
  - NbOfTasks_Array: the user specifies the number of tasks included in each state.
  - TaskTable[TASKS_NB]: the user defines an array of pointer to functions that points to all tasks, then passes it as a parameter.

- **CurrentThread**: the thread(tcb) currently running on the OS.

- **Semaphores:** for each global variable shared by more than one task like (SetTemp & CurrentTemp), I assigned a semaphore to manage concurrent access using OS_Wait() and OS_Signal() functions.

❖ APIs:

- **Context-Switching Macros:**
  Context-Switching in PIC16F is extremely tricky for a lot of reasons:
  - PIC16F supports only one level of interrupts, which means that the ISR context cannot be interrupted. This causes problems for the scheduler as it is called within the ISR context of timer and switches to another task, so now this task cannot be interrupted. **Solution:** in every task, while using the critical sections to protect our re-entrant functions from being interrupted, we also enable GIE bit at the end of the task, which allows the task to be interrupted once it finishes.
  - Some Shadow Registers (CPU core registers used for the context-saving): WREG - STATUS - FSR, are not accessible in C-code. **Solution:** Inline assembly instructions are used to save these registers along with PCL - PCLATH in the task's stack.
  - PIC16F uses a HW stack that is not accessible at all. **Solution:** every task (tcb) has its own stack where these registers are saved. Save/Restore Context macros are used in the beginning and the end of the critical section of the task. I used Macros and not functions so as not to change the context by calling a function.
  - Assigning a stack for each task uses a LOT of memory and the code already after optimization used most of the memory. **Solution:** context-switching feature is turned off by #define, but it works perfectly fine and can be used if used independently.

- **OS_Init():**
  Assign every task to its corresponding TCB and add the user's states configurations.

- **OS_Start()/OS_Restart():**
  Launches/re-launches the first task after On/Off button is pressed.

- **OS_Schedule():**
  - This Api is called at the TickHandler (timer's callbackFn).
  - It switches between tasks within the same State in a circular and round-robin way.
  - Whenever we want to change the state, OS_ChangeState api is called and it set the trigger in the TCB, then the scheduler does the rest.
  - If thread is blocked, the scheduler will skip to the next thread.
  - If all threads are blocked, will then send the processor to sleep;
  - Interrupt will be the only event that will wake the processor to sleep, and resume the OS_Scheduler.

- **OS_Wait()/ OS_Signal:**
  These apis are used to manage concurrent access of shared resources through semaphores. Whenever a task wants to use a shared resource, it calls OS_Wait. If the resource is available, this api decrements its semaphore to prevent other tasks from using it; and if not, the task is blocked from this resource. After finishing using the resource, OS_Signal is called to increment the semaphore.

2.2.2. Second Design:

- This OS is counter-driven; which means that the main entity in this OS is a tick counter to calculate the time elapsed.
- The Scheduler assigns the current thread according to a certain timeline provided by the user.
- The user defines an array containing the start/end times of each task (when the task should start/end).

APIs:

- Os2_Init(*uint16_t); defines the timeline for task execution.
- Os2_Start(); executes the Init_Task and starts the scheduler.
- Os2_AddThreads(); takes as a parameter an array of pointers to function, each one pointing to a task.
- Os2_Scheduler();

2.3. MCAL Modules:

As I mentioned before, I tried to support as many features as I can in every module, in order to implement fully functional and reusable modules. Supported features appears as enums and typedefs used in defining members of the configuration struct. This is the struct passed as a parameter in the Init function of the module. Also, AUTOSAR requires that some APIs can be turned-off using preprocessor directives

2.3.1.Port Module:

Enums & typedefs used	6/6
APIs implemented	5/5
Det errors handled	5/6

Configuration Structure members:

1) Port: the configured port
2) Pin: the configured pin
3) PinDirection: direction of the configured pin (input/output)
4) PinMode: direction of the configured pin (dio, uart, …)
5) PinInitLevel: initial output value of the configured pin
6) PullConfig: activating internal pull-up resistors
7) PinDirChangeable: indicates if the pin direction can be changed in runtime.
8) PinModeChangeable: indicates if the pin mode can be changed in runtime.
9) PortStatus: used for error checking and handling

APIs:

1) Port_Init(Port_ConfigType* ConfigPtr); Initialize a specific pin
2) Port_InitPort(Port_ConfigType* ConfigPtr); Initialize the whole port
3) Port_SetPinDirection; changes the direction of a specific pin (if possible)
4) Port_SetPinMode();changes the mode of a specific pin(if possible)
5) Port_RefreshPortDirection(); restores the original direction of the port

Note: to reduce data memory consumption, and because this modules's struct is used by every other module, I turned off the unused members using preprocessor directives , but the code is usable.

2.3.2.Dio Module:

Enums & typedefs used	5/5
APIs implemented	5/8
Det errors handled	1/5

Configuration Structure members:

1) Port: the configured port
2) Channel: the configured channel
3) Array of configured channels (static variable)

APIs:

1) Dio_WriteChannel(channel, level);
2) Dio_ReadChannel(channel);
3) Dio_ToggleChannel(channel);
4) Dio_WritePort (port, value);
5) Dio_ReadPort (port);

Note: In this module and in all the others, I didn't support any features concerning the version info, the power state as it's not supported at all in PIC16F.

2.3.3.Interrupts Module (Not-AUTOSAR):

Any module that wants to enable interrupts calls the API Interrupt_Enable and passes as parameter the module itself and a pointer to a callback function to be called when the interrupt occurs. This modules handles the interrupts assignment and contains the ISR function.

2.3.4.Adc Module(the best implemented module):

Enums & typedefs used	16/24
APIs implemented	10/12
Det errors handled	6/8

Configuration Structure members:

1) NbChannels; number of adc channels
2) ArrayOfAdcChannels; array of adc channels used
3) GroupAccessMode; single or streaming
4) NbSamples; number of samples taken for a single channel (streaming)
5) GroupConvMode; oneShot or continuous
6) ResultAlignment; align the conversion result to the left or the right
7) Prescale; the prescaler used in the conversion
8) CallbackFn; a function to be called after finishing the conversion
9) Status; used for error checking and handling
10) Group Id;
11) StreamBufferMode; linear or circular buffer (for streaming)

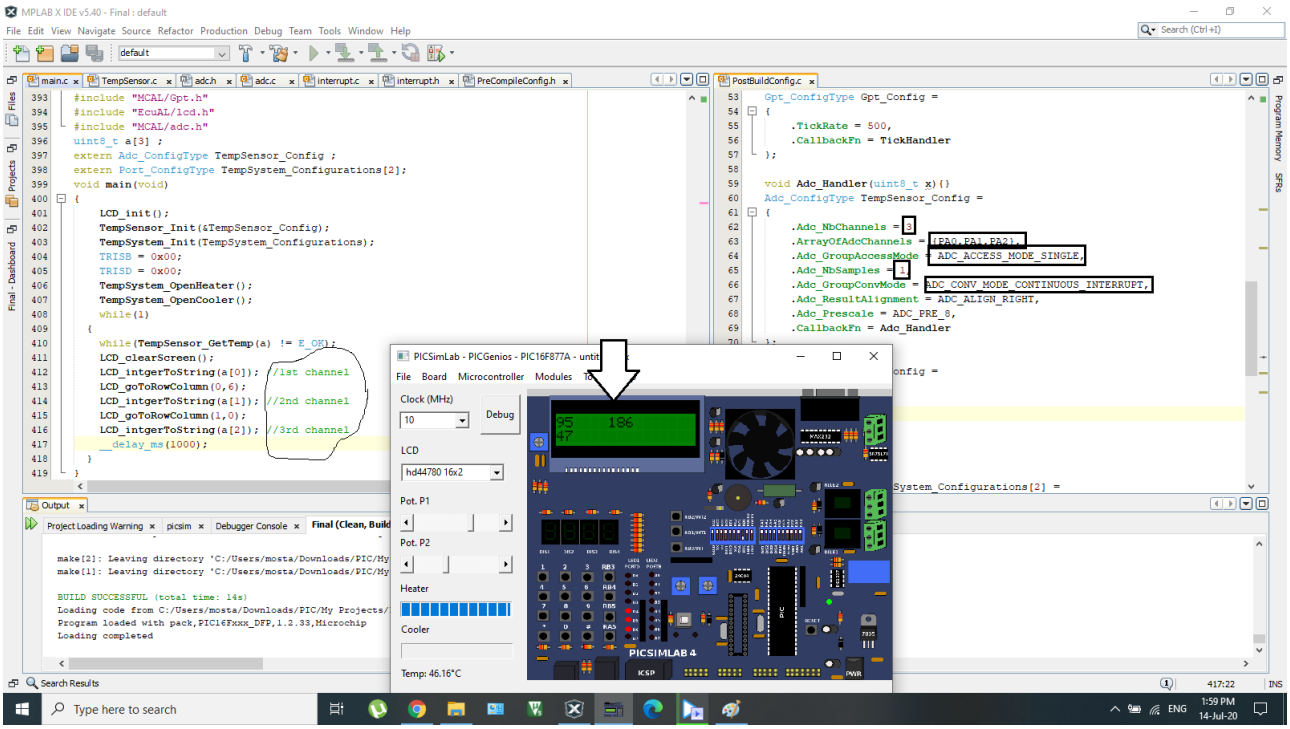
APIs:

1) Adc_Init(Adc_ConfigType*);
2) Adc_ReadGroup (GroupId, *DataBufferPtr);
3) Adc_StartGroupConversion (groupId);
4) Adc_StopGroupConversion (groupId);
5) Adc_DeInit()
6) Adc_GetStatus();
7) Std_ReturnType Adc_SetupResultBuffer(Group, * DataBufferPtr);
8) Adc_EnableGroupNotification (groupId); enable interrupts for adc
9) Adc_DisableGroupNotification(groupId);

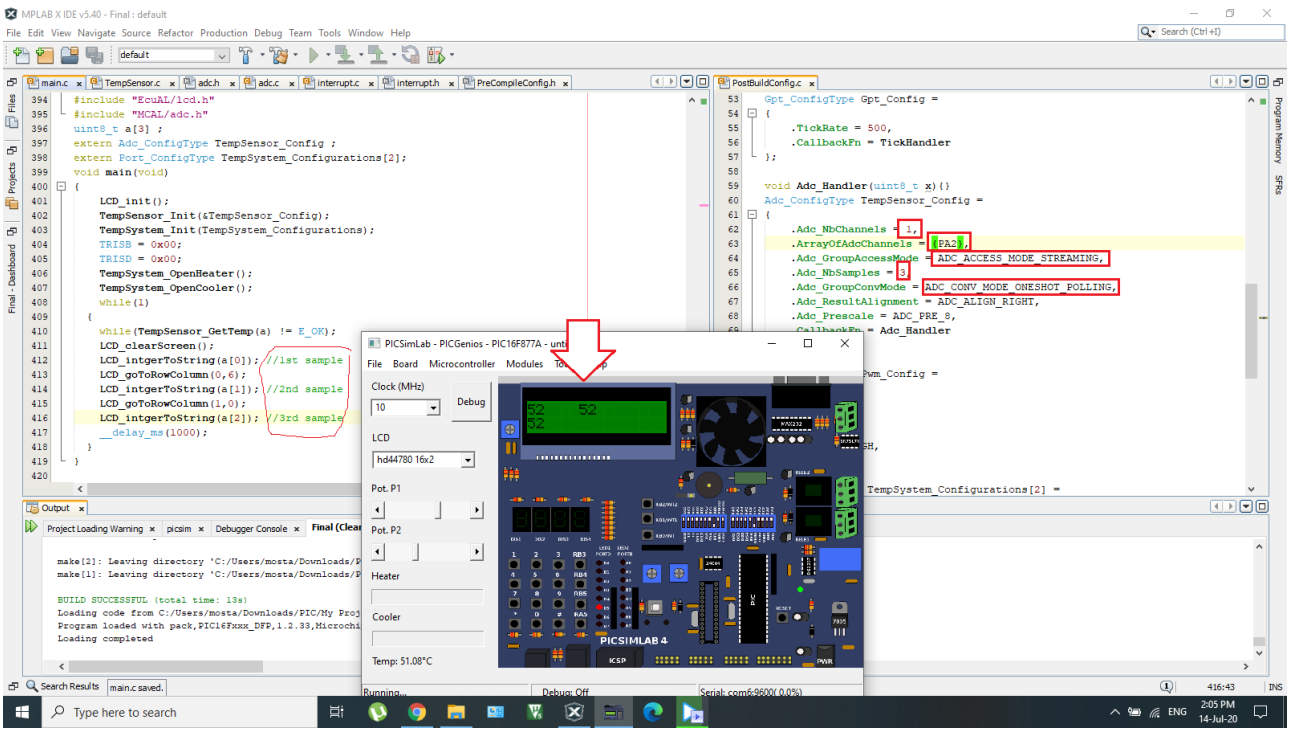
Note: This is the best implemented module because (continuous, streaming) modes are not supported by PIC16 HW, so I supported them using SW (interrupt and polling). That's why the adc is the largest module in the project. The streaming mode is used by the Temperature sensor to get the 10 samples which are used to decide of cooling/heating.

These are some test benches for the Adc Module:

- ❖ Continuous conversion of a group of channels



- ❖ Streaming conversion (taking multiple samples) of a single channel



2.3.5.Gpt Module:

Enums & typedefs used	8/15
APIs implemented	3/14
Det errors handled	7/10

Configuration Structure members:

1) Module; timer0, timer1, or timer2
2) Channel;
3) TickRate; rate at which the counter increments
4) CallbackFn; function to be called at every overflow (TickHandler for OS)
5) Mode; timer, counter or pwm
6) ClkSource; internal or external
7) Status; used for error checking and handling

APIs:

1) Gpt_Init(Gpt_ConfigType*);
2) Gpt_DeInit(Module);
3) Gpt_GetStatus(Gpt_ModuleType Module);

Note:

- The module chooses the prescaler suitable for the input TickRate by itself.
- Timer1 is used by the OS, Timer2 is used by the PWM module.
- The Init function initializes the 2 timers each one with the configurations suitable for the module that uses it.

2.3.6.Pwm Module:

Enums & typedefs used	5/9
APIs implemented	3/9
Det errors handled	7/10

Configuration Structure members:

1) Channel;
2) Period;
3) DutyCycle;
4) Polarity; the entered duty cycle is for the HIGH or LOW state
5) Status; used for error checking and handling

APIs:

1) Pwm_Init(Pwm_ConfigType*);
2) Pwm_DeInit(Module);
3) Pwm_SetDutyCycle (Channel, DutyCycle);
4) Pwm_SetPeriodAndDuty Channel, DutyCycle ,Period);
5) Pwm_SetOutputToIdle (Channel);
6) Pwm_GetStatus();

Note:

- Only the cooler uses the PWM module because the simulation pins of the heater aren't connected to any PWM pins. This can be solved using spare parts in picsimlab but the temperature won't be embedded with the project.

2.3.7.I2c Module:

Enums & typedefs used	8/15
APIs implemented	3/14
Det errors handled	7/10

Configuration Structure members:

1) Module;
2) Channel;
3) Baudrate;
4) Mode; Master/Slave & 7bit/10bit
5) DataWidth: 8bits (fixed in pic16)
6) BufferSource; internal or external
7) TransferStart; LSB (fixed in pic16)
8) DefaultTransmitValue; zero (fixed in pic16)
9) Status;

APIs:

1) I2c_Init(*Config_Ptr);
2) I2c_WriteIB (Channel, *DataBufferPtr);
3) I2c_ReadIB (Channel, *DataBufferPtr);
4) I2c_WriteSlaveAddress(data);
5) I2c_DeInit();
6) I2c_GetStatus();

Note:

- The I2C module isn't exactly supported by AUTOSAR, but the SPI is. The two communication protocols are very similar so I based the I2C module implementation on the AUTOSAR Specs of SPI.

2.3.8.Uart Module (AUTOSAR-like):

Configuration Structure members:

1) Baudrate;
2) ComMode; Master transmitter ot Slave reciever
3) OpMode ; Synchronous or Asynchronous
4) SysMode ; polling or interrupt
5) NinthBitEn; ninth bit communication enable
6) AddrDetEn; address detection property enable
7) TransferStart; LSB or MSB
8) Status;

APIs:

1) Uart_Init(Uart_ConfigType*);
2) Uart_WriteByte(uint8_t);
3) Uart_WriteString(char*);
4) Uart_WriteInt(uint8_t);
5) void Uart_DeInit();
6) Uart_GetStatus();

Note:

- This module is used by the Det module.

2.4. HAL Modules:

2.4.1.Eeprom Module:

Enums & typedefs used	3/3
APIs implemented	6/10
Det errors handled	6/10

Configuration Structure members:

1) SlaveAddress;
2) Size; size of the used eeprom
3) Status;

APIs:

1) Eep_Init(*ConfigPtr);
2) Eep_Write(EepromAddress, *DataBufferPtr, Length);
3) Eep_Read (EepromAddress, *DataBufferPtr, Length);
4) Eep_DeInit();
5) Eep_Erase (EepromAddress, Length);
6) Eep_Compare (EepromAddress, * DataBufferPtr, Length);
7) Eep_GetStatus ();

Note:

- This is the only HAL module following the AUTOSAR Specs.

2.4.2.Display Module:

Configuration Structure members:

1) DisplayPort; which port to put the displayed data
2) EnablePort; which port to be used to enable the 7 segment display
3) EnablePinsArray; enable pins
4) NbDigits; number of digits used
5) BlinkingPeriod;

APIs:

1) Disp_Init(Disp_ConfigType*);
2) Disp_DisplayNb(var, Mode);
3) Disp_Close();

2.4.3.TempSensor Module:

Configuration Structure members:

The Pin used by the temperature sensor

APIs:

1) TempSensor_Init(Adc_ConfigType*);
2) TempSensor_GetTemp(* DataBufferPtr);

2.4.4.TempSystem Module:

Configuration Structure members:

The Pins used by the heater/cooler

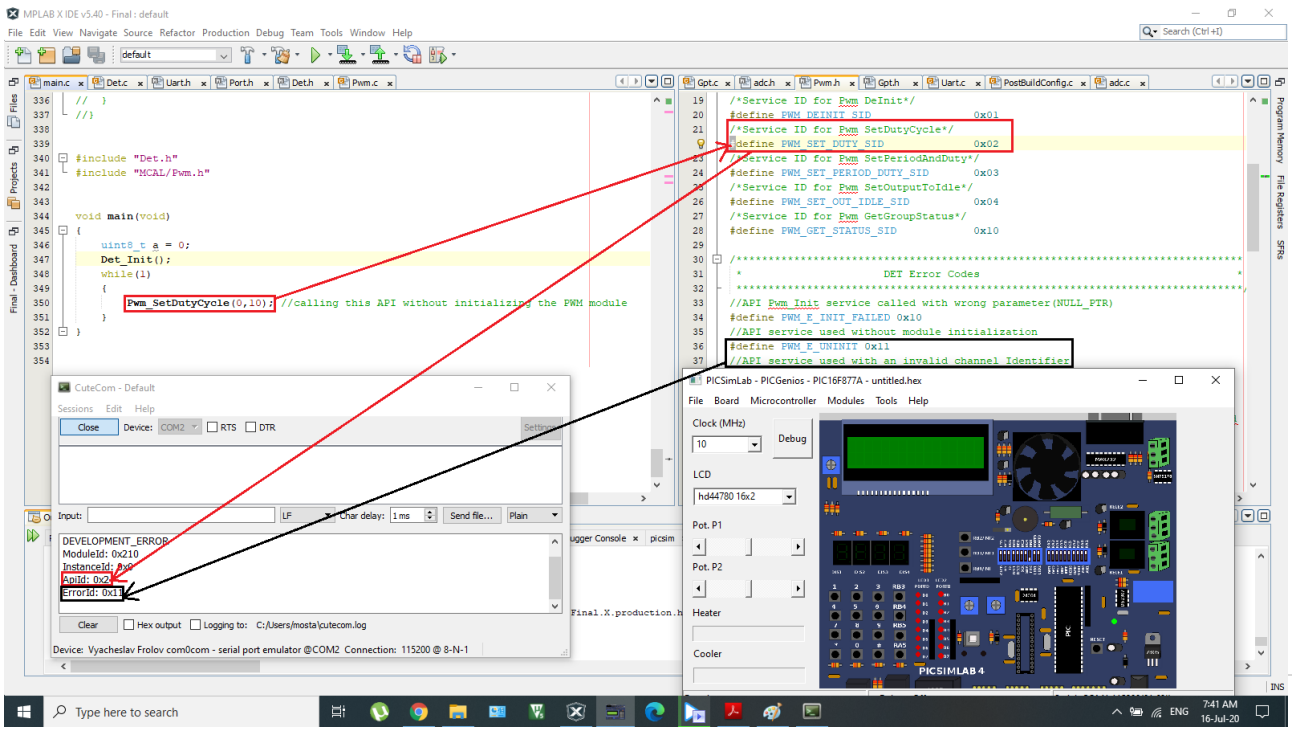
APIs:

1) TempSystem_Init(Port_ConfigType*);
2) TempSystem_OpenHeater/ TempSystem_CloseHeater
3) TempSystem_SetCoolerPower / TempSystem_CloseCooler

Note: Only the cooler uses the PWM module because the simulation pins of the heater aren't connected to any PWM pins. This can be solved using spare parts in picsimlab but the temperature won't be embedded with the project.

2.5. DET Module:

- This module is used to handle development errors in every module.
- It can be turn On/Off in the pre-compile configurations.
- It uses the Uart to display the error before going to sleep mode.
- Every error has a code that can be found in the module Specs.
- The most famous errors:
  - <module>_E_UNINIT
    API service called without module initialization
  - <module>_E_ALREADY_INITIALIZED
    Init api has been called while module is already initialized
  - <module>_E_PARAM_POINTER
    API parameter checking: invalid pointer (NULL_PTR)
  - <module>_E_PARAM_CHANNEL
    API parameter checking: invalid channel requested
  - <module>_E_PARAM_GROUP
    API parameter checking: invalid group ID requested
  - <module>_E_PARAM_VALUE
    API parameter checking: invalid value requested
  - <module>_E_PARAM_MODE
    API parameter checking: invalid mode requested
- API: Det_ReportError (ModuleId, InstanceId, ApiId, ErrorId)



3. Deliverables:

- Hex file ready to be loaded to PicSimlab.
- "Final.X": Complete version of the code using OS1.
- "Final with OS2 .X": 2nd version of the code using OS2 (4 states only, it's used to prove that OS2 is working correctly).
- "Final without error habdling.X": 3rd version of the code without the error handling because the complete version uses 98% of RAM and 93% of ROM which on some laptops, it causes problems in PicSimLab.
- A link to a video showing that the 3 codes work perfectly: https://www.youtube.com/watch?v=TCeqcKRjvbw&feature=youtu.be
  OR **https://youtu.be/TCeqcKRjvbw**
- My CV.