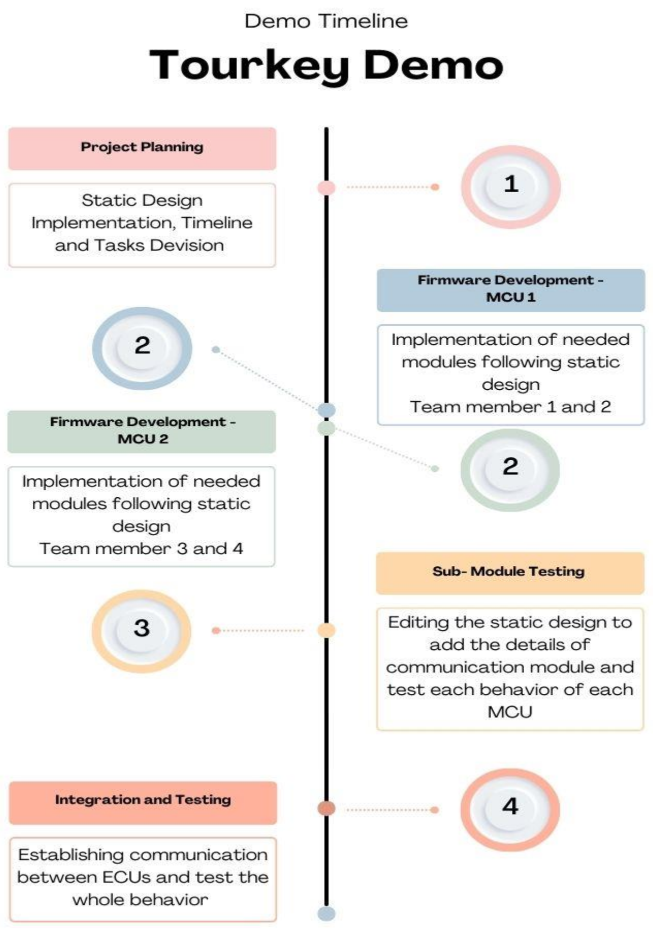
Static Design Analysis

Description:

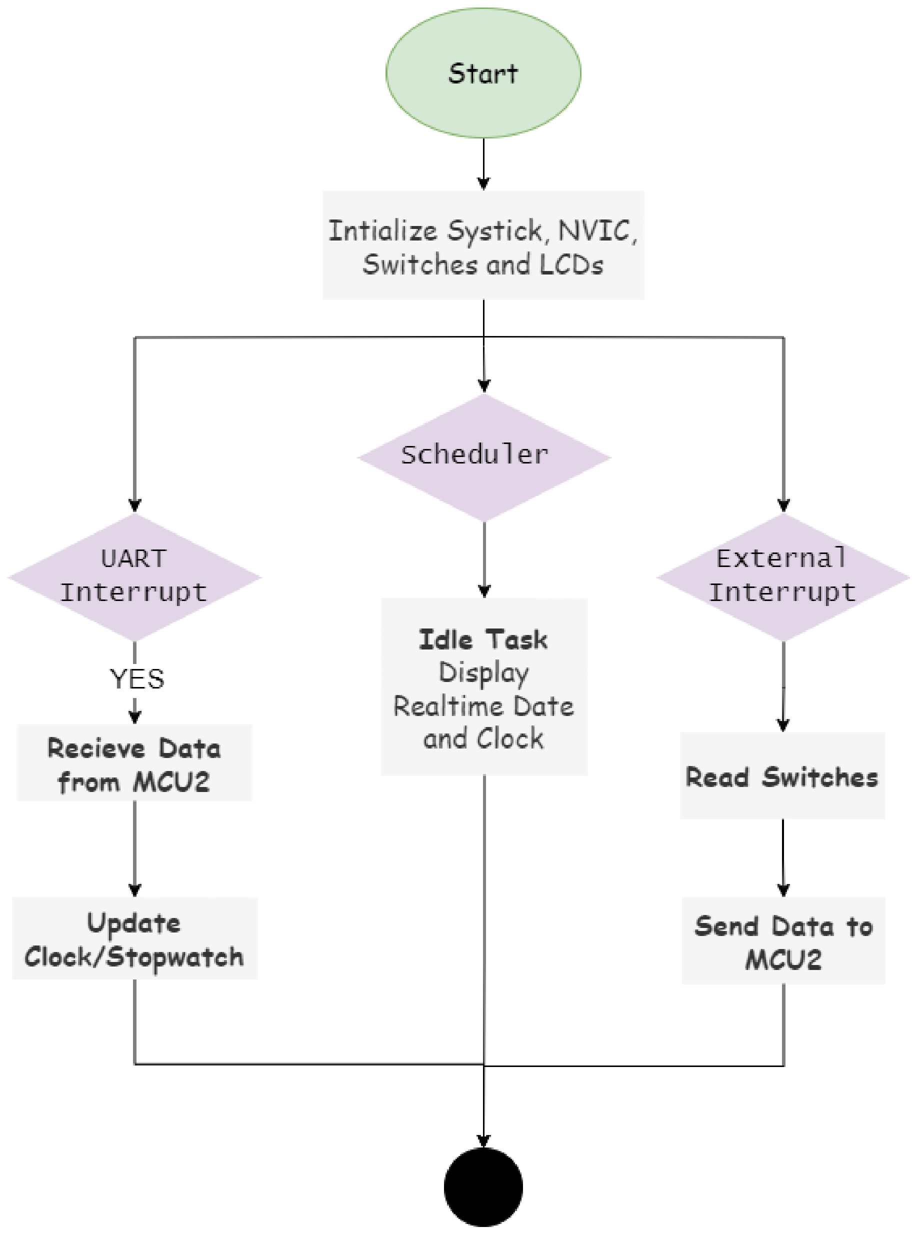
- Each MCU has an LCD which displays:
  - Date and Time (with seconds)
  - Stop watch
- Each MCU has switches to control the LCD of the other MCU
- The communication between MCUs could be UART or any other protocol
- When switching from Stop watch mode to Date & Time mode then going back to Stop watch. The Stop watch should be resuming its previous work

Timeline

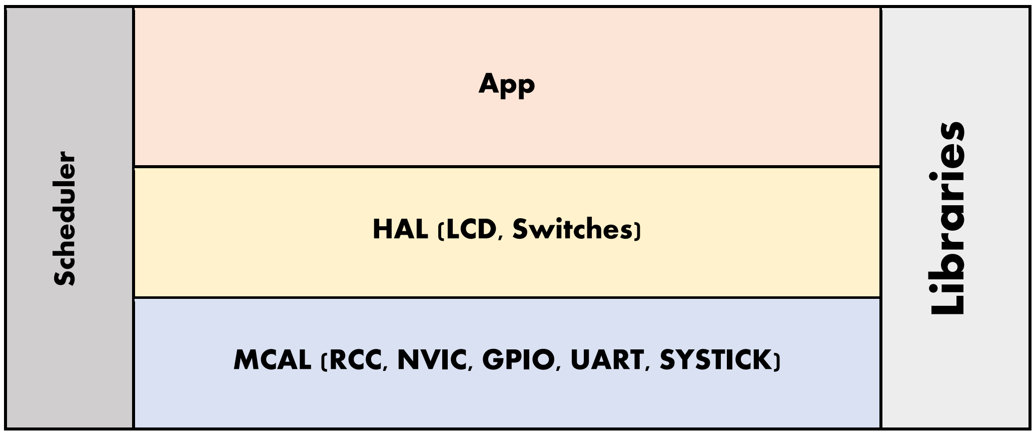

Demo Timeline
Tourkey Demo
Project Planning
Static Design Implementation, Timeline and Tasks Devision
1
Firmware Development - MCU 1
Implementation of needed modules following static design
Team member 1 and 2
2
Firmware Development - MCU 2
Implementation of needed modules following static design
Team member 3 and 4
2
Sub- Module Testing
Editing the static design to add the details of communication module and test each behavior of each MCU
3
Integration and Testing
Establishing communication between ECUs and test the whole behavior
4

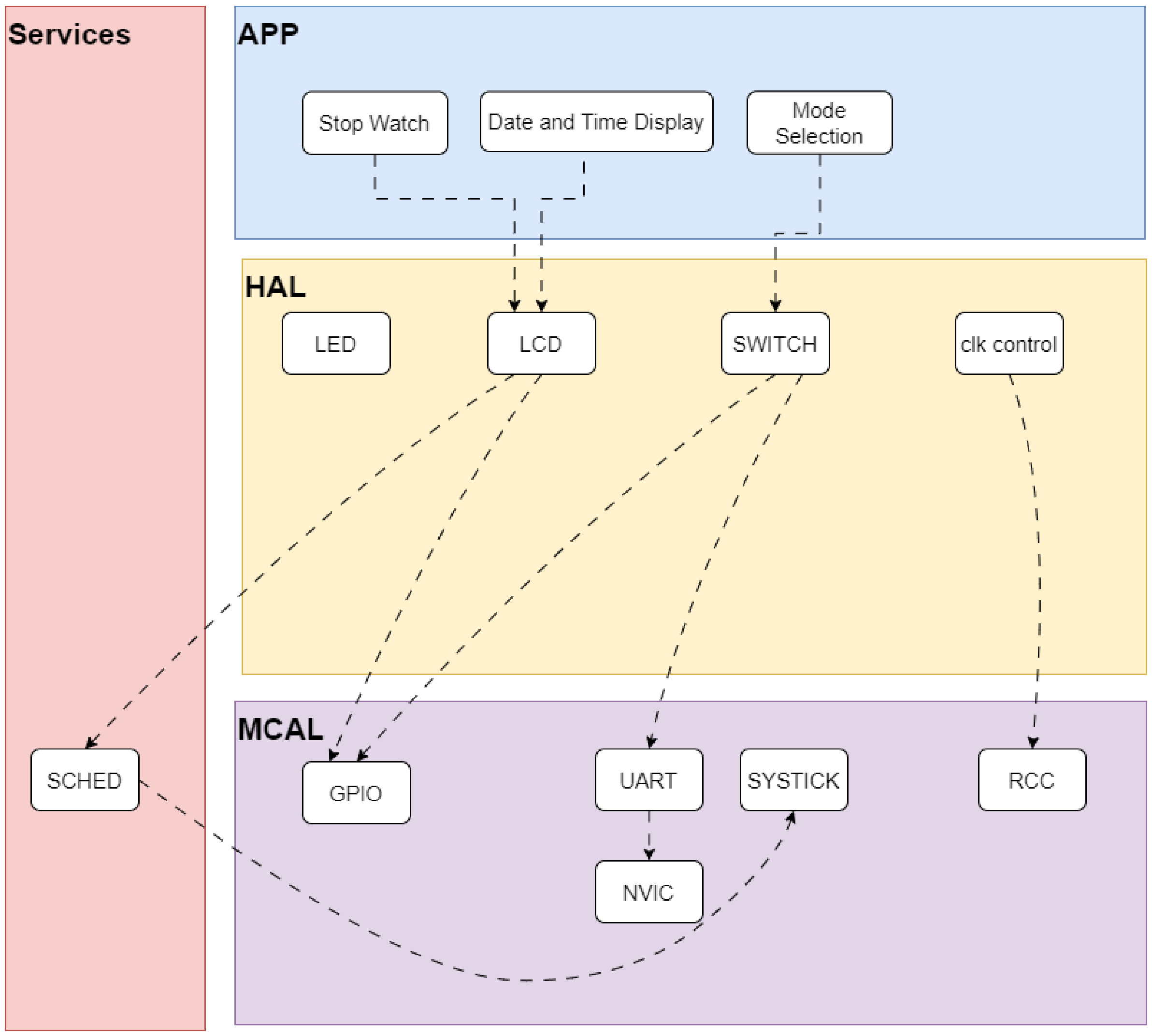
For ECU 1

Flow Chart



The layered architecture





ECU components

1) Switches
   - Increment Clock Switch
   - Decrement Clock Switch
   - Mode Selection Switch
   - UP and Down switches for mode selection
2) LCD

Full detailed APIs for each module as well as a detailed description for the used typedefs

1. NVIC Module

1.1 void NVIC_DecodePriority (uint32 Priority, uint32 PriorityGroup, uint32* pPreemptPriority, uint32* pSubPriority);

Function	void NVIC_DecodePriority (uint32 Priority, uint32 PriorityGroup, uint32* pPreemptPriority, uint32* pSubPriority);
Input Arguments	**Priority:** Priority value, which can be retrieved with the function \ref NVIC_GetPriority(). **Type:** Unsigned Integer **Range:** from zero to 128 **PriorityGroup:** Used priority group. **Type:** Unsigned Integer **Range:** from 1 to pPreemptPriority: Preemptive priority value **Type:** Pointer to unsigned Integer **Range:** from 0 to pSubPriority: Subpriority value **Type:** Pointer to unsigned Integer **Range:** from 0 to
Output Arguments	None
Description	- The function decodes an interrupt priority value with a given priority group to preemptive priority value and subpriority value.

	- In case of a conflict between priority grouping and available priority bits (__NVIC_PRIO_BITS) the smallest possible priority group is set.
Return	None

1.2 `void NVIC_SoftwareInterrupt(IRQn_t IRQn);`

1.3 `uint32 NVIC_EncodePriority (uint32 PriorityGroup, uint32 PreemptPriority, uint32 SubPriority);`

1.4 `uint32 NVIC_GetPriority(IRQn_t IRQn);`

1.5 `void NVIC_SetPriority(IRQn_t IRQn, uint32 priority);`

1.6 `void NVIC_SetPriorityGrouping(uint32 PriorityGroup);`

1.7 `uint32 NVIC_GetPriorityGrouping(void);`

1.8 `void NVIC_EnableIRQ(IRQn_t IRQn);`

2. GPIO Module

2.1 `GPIO_ERROR_t GPIO_init(GPIO_ConfigPin_t *Config_ptr);`

2.2 `GPIO_ERROR_t GPIO_setPinValue(void* port,uint32 pin, uint32 value);`

2.3 `uint32 GPIO_getPinValue(void* port,uint32 pin);`

3. RCC Module

3.1 `RCC_ERROR_STATUS RCC_enableClk(CLK_SRC clk);`

3.2 `RCC_ERROR_STATUS RCC_disableClk(CLK_SRC clk);`

3.3 `RCC_ERROR_STATUS RCC_setSysClk(CLK_SRC clk);`

3.4 `RCC_ERROR_STATUS RCC_pllConfig(uint8 PLLM,uint8 PLLN,uint8 PLLP,uint8 PLLQ,PLL_SRC PLLclk);`

3.5 `RCC_ERROR_STATUS RCC_enablePeri(RCC_BUS_ID_t busID, uint32 P_id);`

3.6 RCC_ERROR_STATUS RCC_disablePeri(RCC_BUS_ID_t busID, uint32 P_id);

3.7 RCC_ERROR_STATUS RCC_prescalerConfig(RCC_BUS_ID_t busID,uint32 prescalar);

4. UART Module

Function	***void* UART_init(*void*)**
Input Arguments	None
Output Arguments	None
Description	This function initializes the UART module based on the configuration settings. It iterates over the UART modules that are used (as indicated by _UART_USED_NUM). For each UART module, it retrieves the base address and applies the configuration settings such as stop bits, oversampling, data size, and parity. It also enables the UART module. The configuration settings are retrieved from the usart_config array, which is presumably defined elsewhere in your code. The function also starts to calculate the Baud rate based on the oversampling setting, but the provided code snippet does not show the complete calculation.
Return	None

Function	***void* UART_sendByte(UART_UserReq_t **Ptr_UserReq*)**
Input Arguments	**UART_UserReq_t *Ptr_UserReq**: This argument is a pointer to a UART_UserReq_t structure. This structure likely contains information about a user request to send a single byte of data over UART. The exact contents of this structure would depend on its definition, which is not provided in the code snippet.
Output Arguments	None
Description	This function sends a single byte of data over UART. The data to be sent is likely contained in the UART_UserReq_t structure pointed to by Ptr_UserReq. The function likely waits for the UART transmitter to be ready, then writes the byte to the UART data register to start the transmission. The exact behavior of

	the function would depend on its implementation, which is not provided in the code snippet.
Return	None

Function	***void* USART_TxBufferAsyncZeroCopy(<u>UART UserReq t</u> *Ptr_UserReq*)**
Input Arguments	**UART_UserReq_t *Ptr_UserReq**: This argument is a pointer to a UART_UserReq_t structure. This structure likely contains information about a user request to transmit data over UART. The exact contents of this structure would depend on its definition, which is not provided in the code snippet.
Output Arguments	None
Description	This function performs an asynchronous, zero-copy transmission of data over UART. The "zero-copy" part means that the function does not create a copy of the data to be transmitted; instead, it transmits the data directly from the buffer provided by the user. The function likely uses DMA or interrupts to perform the transmission, allowing it to return immediately while the transmission continues in the background. The exact behavior of the function would depend on its implementation, which is not provided in the code snippet.
Return	None

Function	***void* USART_RxBufferAsyncZeroCopy(<u>UART UserReq t</u> *Ptr_UserReq*)**
Input Arguments	**UART_UserReq_t *Ptr_UserReq**: This argument is a pointer to a UART_UserReq_t structure. This structure likely contains information about a user request to receive data over UART. The exact contents of this structure would depend on its definition, which is not provided in the code snippet. Output Arguments:
Output Arguments	None

Description	This function performs an asynchronous, zero-copy reception of data over UART. The "zero-copy" part means that the function does not create a copy of the received data; instead, it stores the data directly into a buffer provided by the user. The function likely uses DMA or interrupts to perform the reception, allowing it to return immediately while the reception continues in the background. The exact behavior of the function would depend on its implementation, which is not provided in the code snippet.
Return	None

5. SYSTICK Module

Function	***void*** **SYSTICK_Init(<u>uint8</u>** ***mode*****)**
Input Arguments	uint8 mode: This argument represents the mode in which the SysTick timer should operate. The specific modes available would be defined elsewhere in your code, likely as #define statements or an enum.
Output Arguments	None
Description	This function initializes the SysTick timer. It first sets the systick_mode global variable to the provided mode. Then it resets the SysTick current value register (Systick->VAL) to 0. The function appears to be incomplete as the comments suggest that it should also select the system clock, enable the SysTick interrupt, and enable the SysTick timer, but the code to do these operations is not present.
Return	None

Function	***void*** **SYSTICK_set_ms(<u>uint32</u>** ***time)*****)**
Input Arguments	**uint32 time**: This argument represents the time in milliseconds for which the SysTick timer should be set.

Output Arguments	None
Description	This function sets the SysTick timer to generate an interrupt every time milliseconds. It first checks the CLKSOURCE bit in the SysTick->CTRL register to determine the clock source of the SysTick timer. If the CLKSOURCE bit is set, the SysTick timer is using the processor clock; otherwise, it's using the processor clock divided by 8. The function then calculates the reload value based on the clock speed and the desired time, and sets the LOAD and VAL registers of the SysTick timer accordingly.
Return	None

Function	***void* SYSTICK_stop ()**
Input Arguments	None
Output Arguments	None
Description	This function stops the SysTick timer. It does this by clearing the ENABLE bit in the SysTick->CTRL register. This bit controls whether the SysTick timer is running or stopped. When the ENABLE bit is cleared (0), the SysTick timer is stopped.
Return	None

Function	***void* SYSTICK_setCallBack(systick_cbf_t *cbf*)**
Input Arguments	**systick_cbf_t cbf**: This argument is a callback function that will be called when the SysTick interrupt occurs. The systick_cbf_t type is likely a function pointer type, defined something like this: typedef void (*systick_cbf_t)(void);.
Output Arguments	None

Description	This function sets a callback function for the SysTick interrupt. The provided callback function cbf will be stored in the global variable APP_cbf and will be called when the SysTick interrupt occurs. If cbf is NULL_PTR, the function does nothing.
Return	None

Function	***void* SysTick_Handler(*void*)**
Input Arguments	None
Output Arguments	None
Description	This function is the interrupt service routine for the SysTick interrupt. When the SysTick timer expires, this function will be called. If the APP_cbf callback function is set (not NULL), the function will first check the systick_mode global variable. If systick_mode is ONESHOT, the function will stop the SysTick timer by clearing the ENABLE bit in the SysTick->CTRL register. Then, the function will call the APP_cbf callback function.
Return	None

6. Scheduler Module

6.1 void sched_init(void);

6.2 void sched_start(void);

7. Switches Module

7.1 void SWITCH_init(void);

7.2 SWITCH_State_t SWITCH_Getstatus(uint32 switch_num);

7.3 void SWITCH_Runnable(void)

8. LCD Module

8.1 void LCD_Init(void);

8.2 `void LCD_writeStringAsync(uint8* string,uint8 length);`

8.3 `void LCD_setCursorPosAsync(uint8 posX, uint8 posY);`

8.4 `void LCD_clearScreenAsynch(void);`

8.5 `uint8 LCD_getStatus(void);`

8.6 `static void LCD_initSM();`

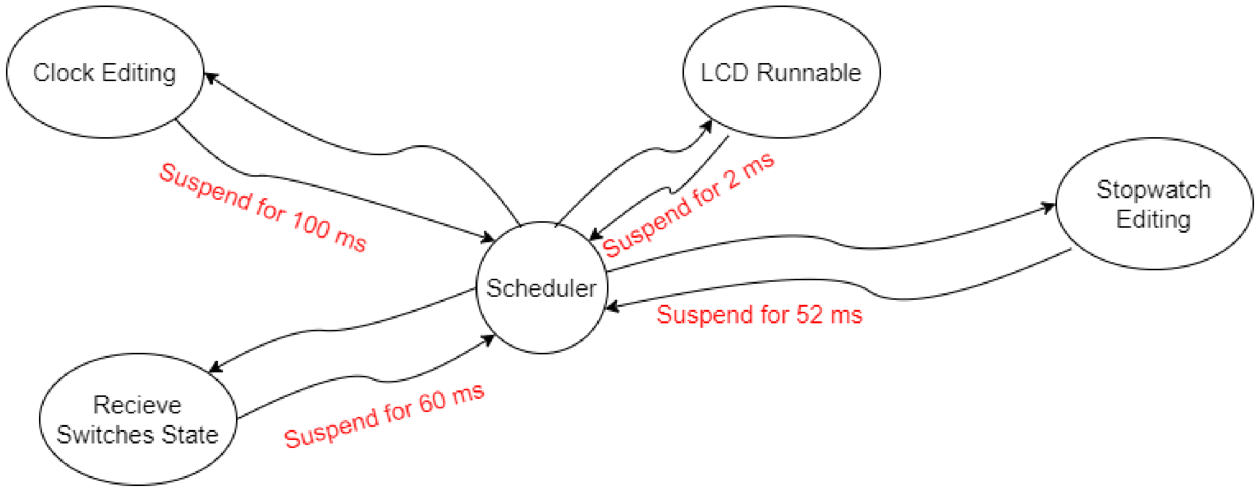
8.7 `static void LCD_writeProcess();`

8.8 `static void LCD_clearProcess();`

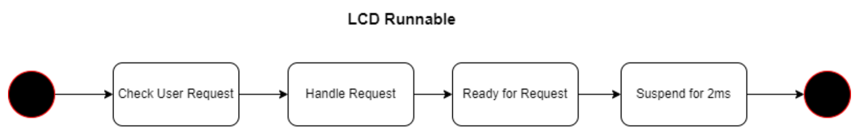
8.9 `static void LCD_setPose_Process();`

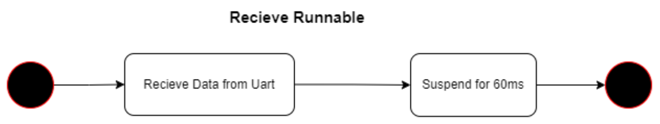
Dynamic Design Analysis

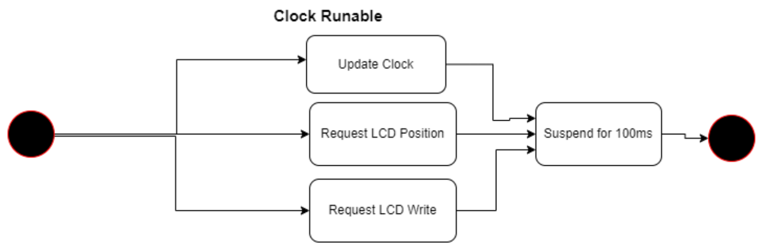
2.1 Draw a state machine diagram for the ECU operation

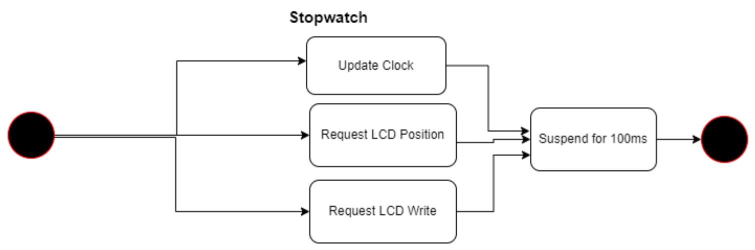


2.2 Draw a state machine diagram for each ECU component

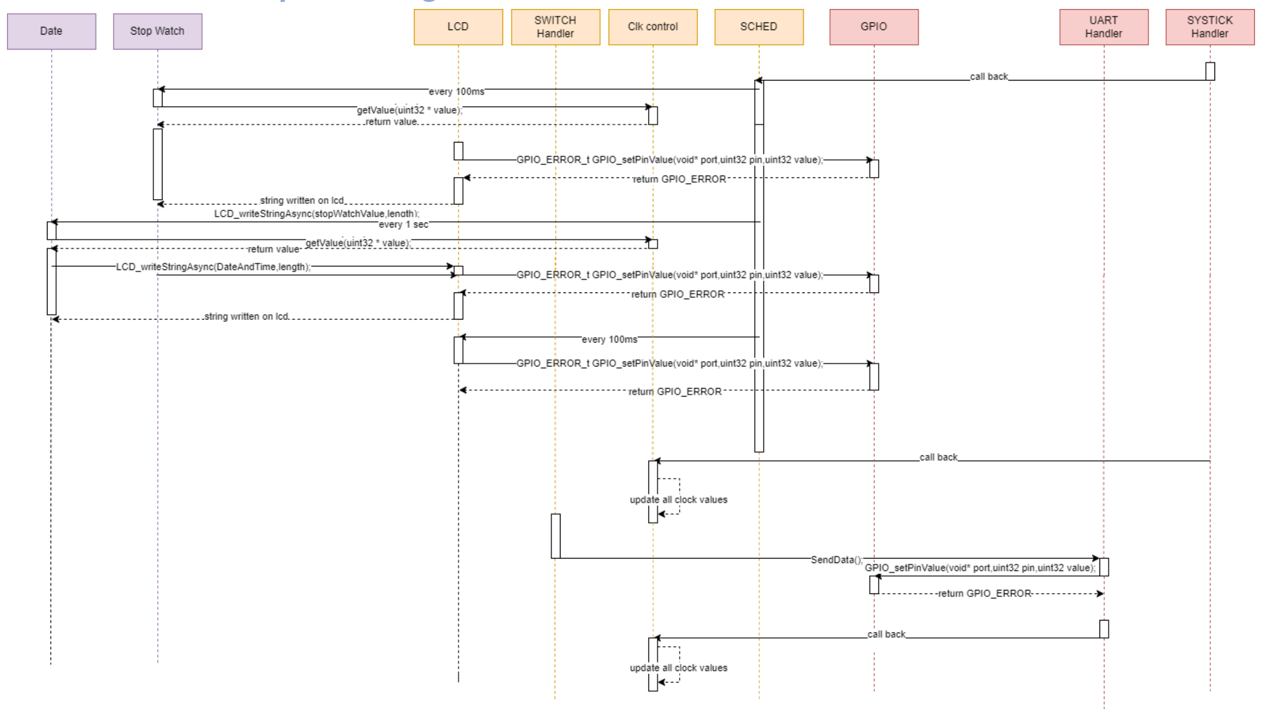








2.3 Draw sequence diagram for each ECU


Date
Stop Watch
LCD
SWITCH Handler
Clk control
SCHED
GPIO
UART Handler
SYSTICK Handler
call back
every 100ms
getValue(uint32 * value);
return value
GPIO_ERROR_t GPIO_setPinValue(void* port,uint32 pin,uint32 value);
return GPIO_ERROR
string written on lcd
LCD_writeStringAsync(stopWatchValue,length);
every 1 sec
getValue(uint32 * value);
return value
LCD_writeStringAsync(DateAndTime,length);
GPIO_ERROR_t GPIO_setPinValue(void* port,uint32 pin,uint32 value);
return GPIO_ERROR
string written on lcd
every 100ms
GPIO_ERROR_t GPIO_setPinValue(void* port,uint32 pin,uint32 value);
return GPIO_ERROR
call back
update all clock values
SendData();
GPIO_setPinValue(void* port,uint32 pin,uint32 value);
return GPIO_ERROR
call back
update all clock values