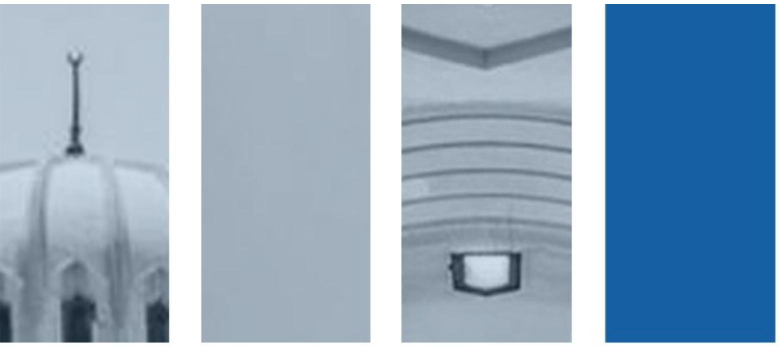


RESEARCH & PROJECT SUBMISSIONS

Program: Computer and Systems

Course Code: CSE312

Course Name: Microprocessor Based Systems

Examination Committee:

Prof. Dr. Ashraf M. M. Elfarghly Salem

Ain Shams University
Faculty of Engineering
Spring Semester – 2020

Student Personal Information

Student Name: أحمد عبد الحكيم عبد الله محمد
Student Code: 1600122
Class/Year: Third Year

Plagiarism Statement

I certify that this assignment / report is my own work, based on my personal study and/or research and that I have acknowledged all material and sources used in its preparation, whether they are books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this assignment / report has not been previously been submitted for assessment for another course. I certify that I have not copied in part or whole or otherwise plagiarized the work of other students and / or persons.

Signature/Student Name: احمد عبد الحكيم عبد الله محمد **Date:** 5/6/2020

Submission Contents

01: **Bill of Materials and Design**
02: **Detailed Circuit Diagram**
03: **C Code**
04: **Real Time Operating Systems Comparison**

01

First Topic

Bill of Materials and Design

Bill of Materials:

1- Tiva C Series TM4C123G LaunchPad Evaluation Board.

2- 4x4 Keypad module.

3- Solenoid lock (Dowonsol 3v – 12v DC 80mA-350mA Micro Solenoid).

4- 330 Ohm resistance.

5- Power source.

Design:

The Idea of the project is pretty simple and straightforward, although the implementation isn't the best. The design separates the setup mode from the other modes, where the receptionist enters the rooms numbers through UART connection with receptionist PC, incrementing the noOfRooms variable with every new number, and they are stored in roomsNo array, and when he or she enters the same number twice; that means that the setup mode has ended. The setup variable is set to one, and the roomNumber variable is set to the entered room number (the variable first insures that the room number won't be asked twice at the first time). Then through UART connection with receptionist PC, the status of the room is given and stored in status array according to the given room number, where simple if-else statements differentiate between different statuses. If the status is one, the room is occupied, and a password must be entered through UART connection with receptionist PC, the four-digit password is taken digit by digit and stored in the savedpassarr two-dimensional array according to the room number given. Then the guest should enter the password through the keypad, the password is taken digit by digit and stored in the in array, then validatepassword function is called, which validates the password, by comparing character by character. If the password is correct, the door opens (solenoid unlocked), else it stay closed. When the receptionist enters another room number, and its status, if the status is zero the room is free, and the door is closed, else if the status is two the room is in room cleaning mode, and the door is open.

02

Second Topic

Circuit Diagrams

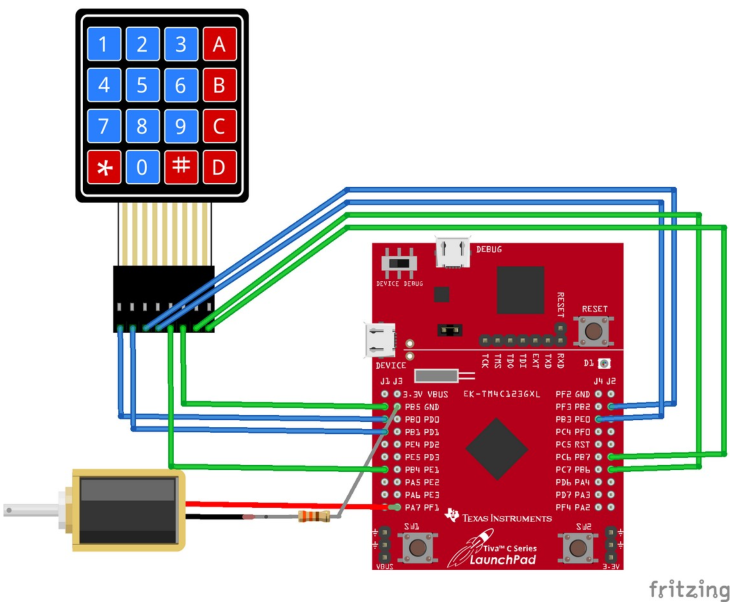


Fig (1.1): Circuit configuration using Solenoid.

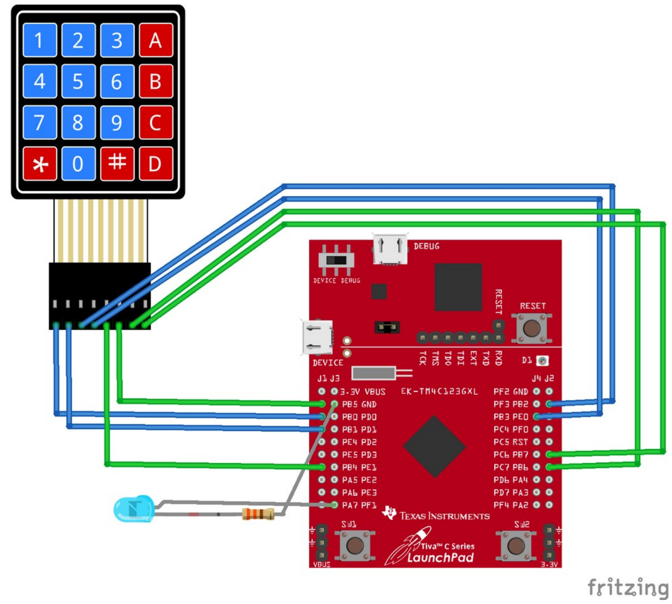


Fig (1.2): Circuit configuration using LED

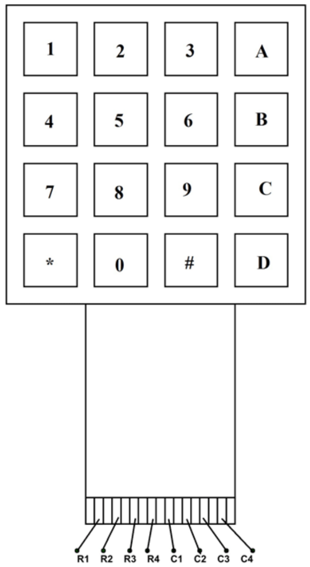


Fig (2.1): 4x4 Keypad module pinout.

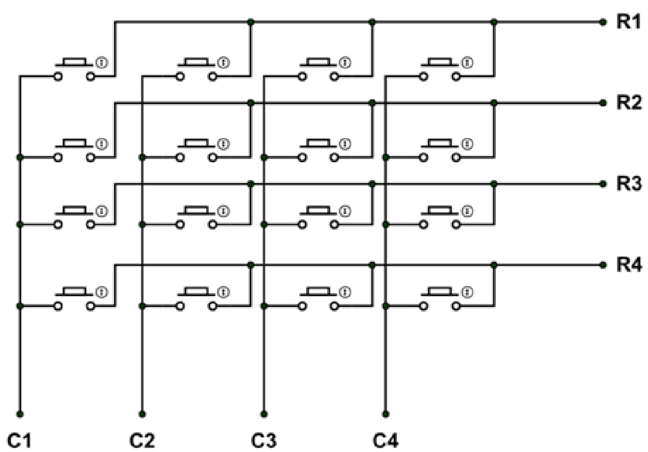


Fig (2.2): 4x4 Keypad internal structure.

03
Third Topic

C Code

```
#include "KeyPad.h"
#include "uart.h"

int8_t validatepassword(uint8_t *input , uint8_t *rightpass);
void initportf()
{
   SYSCTL_RCGC2_R |= 0x20;
   GPIO_PORTF_LOCK_R = 0x4C4F434B;
   GPIO_PORTF_CR_R = 0XFF;
   GPIO_PORTF_AFSEL_R = 0;
   GPIO_PORTF_PCTL_R = 0;
   GPIO_PORTF_AMSEL_R = 0;
   GPIO_PORTF_DIR_R |= 0X02;
   GPIO_PORTF_DIR_R &= ~0X01;
   GPIO_PORTF_DEN_R |= 0X03;
   GPIO_PORTF_PUR_R |= 0X01;
   GPIO_PORTF_DATA_R &= ~0x02; // Solenoid Locked
}
int main(void)
{
   uint8_t noOfRooms = 0;
   uint8_t key;
   uint8_t counter = 0;
   uint8_t status[10];
   uint8_t roomsNo[10];
   uint8_t roomNo;
   uint8_t found = 0;
   uint8_t in[4];
   uint8_t savedpassarr[10][4];
   uint8_t setup = 0;
   uint8_t first = 0;
   uint8_t i = 0;
   uint8_t j = 0;
   uint8_t k = 0;
   volatile uint8_t check;
   volatile uint8_t roomNumber;
   initportf();
   UART_init();
   KeyPad_init();
   for(;;)
   {
      if(setup == 0)
```

```
{
  while(1)
  {
    roomNo = UART0_read();
    for(k = 0; k <= noOfRooms && noOfRooms != 0; k++)
    {
      if(roomsNo[k] == roomNo)       // Setup finished, same room number
      {
        roomNumber = roomNo;
        found = 1;
      }
    }
    if(found == 0)
    {
      roomsNo[noOfRooms] = roomNo;
      noOfRooms++;
    }
    else{break;}
  }
  setup = 1;
}
if(setup == 1)
{
  if(first == 1)
  {
    roomNumber = UART0_read();
  }
  first = 1;
  status[roomNumber] = UART0_read();

  if(status[roomNumber] == 1)
  {
    for(j = 0; j < 4; j++)
    {
      savedpassarr[roomNumber][j] = UART0_read();
    }
    if(counter < 4)
    {
      key = KeyPad_getPressedKey();
      if(key == '#' || key == '*'){continue;}
      in[i] = key;
      i++;
      counter++;
    }
    if(counter >= 4)
    {
      key = KeyPad_getPressedKey();
      if(key == '#')
      {
        check = validatepassword(in,savedpassarr[roomNumber]);
        if(check == 1)                      // Correct password
        {
```

```
                    GPIO_PORTF_DATA_R |= 0x02; //Solenoid open
                  }
                  i = 0;
                  counter = 0;
                }}}
          else if(status[roomNumber] == 2)
          {
            GPIO_PORTF_DATA_R |= 0x02;
          }
          else if(status[roomNumber] == 0)
          {
            GPIO_PORTF_DATA_R &= ~0x02;
          }}}}
int8_t validatepassword(uint8_t *input , uint8_t *rightpass)
{
  uint8_t z;
  for(z = 0; z < 4; z++)
  {
    if (input[z] != rightpass[z])
    return 0;
  }
  return 1;
}

#ifndef __KEYPAD__H__
#define __KEYPAD__H__
#include "tm4c123gh6pm.h"
#include "stdint.h"
uint8_t KeyPad_getPressedKey(void);
void KeyPad_init(void);
#endif

#include "KeyPad.h"
uint8_t KeyPad_4x3_adjustKeyNumber(uint8_t button_number)
{
    switch(button_number)
    {
        case 10: return '*'; // ASCII Code of *
        case 11: return 0;
        case 12: return '#'; // ASCII Code of #
        default: return button_number;
    }
}
uint8_t KeyPad_getPressedKey(void)
{
    uint8_t col,row;
    while(1)
    {
        for(col = 0; col < 3; col++) // loop for columns
        {
      GPIO_PORTB_DATA_R |= 0xF0;        // first make all cols (PB7-PB4) as output high
```

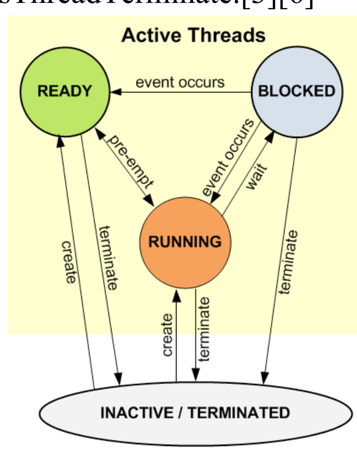
```
        GPIO_PORTB_DATA_R &= (~(0x10<<col)); // second make one of the columns output low at each
iteration in order PB4 to PB7
            for(row = 0; row < 4; row++) // loop for rows
            {
                if(!(GPIO_PORTB_DATA_R & (0x01<<row))) // if the switch is press in this row
                {
                    return KeyPad_4x3_adjustKeyNumber((row*3) + col + 1);
                }}}}}
void KeyPad_init(void)
{
  volatile unsigned long delay;
  SYSCTL_RCGC2_R |= 0x00000002;     // 1) activate clock for Port B
  delay = SYSCTL_RCGC2_R;           // allow time for clock to start
  GPIO_PORTB_AMSEL_R &= 0x00;       // 3) disable analog on PB
  GPIO_PORTB_PCTL_R &= 0x00000000;  // 4) PCTL GPIO on PB7-0
  GPIO_PORTB_DIR_R |= 0xF0 ;        // make cols as output (PB7-PB4)
  GPIO_PORTB_DIR_R &= 0xF0 ;        // make rows as input (PB3-PB0)
  GPIO_PORTB_AFSEL_R &= 0x00;       // 6) disable alt funct on PB7-0
  GPIO_PORTB_PUR_R |= 0x0F ;        // enable pull-up for (PB3-PB0)
  GPIO_PORTB_DEN_R = 0xFF;          //7) enable digital I/O on PB7-0
}

#ifndef UART_H_
#define UART_H_
#include "tm4c123gh6pm.h"
#include "stdint.h"
void UART_init(void);
uint8_t UART0_read (void);
#endif

#include "uart.h"
void UART_init(void)
{
  SYSCTL_RCGCUART_R |= 0x0001;
  SYSCTL_RCGCGPIO_R |= 0x0001;
  UART0_CTL_R &= ~ 0x0001;
  UART0_IBRD_R = 104; //9600
  UART0_FBRD_R = 11;
  UART0_LCRH_R = 0x0070;
  UART0_CTL_R = 0x0301;
  UART0_CC_R = 0x05;  //16MHz
  GPIO_PORTA_AFSEL_R |= 0x03;
  GPIO_PORTA_PCTL_R = (GPIO_PORTA_PCTL_R & 0xFFFFFF00) + 0x00010001;
  GPIO_PORTA_DEN_R |= 0x03;
  GPIO_PORTA_AMSEL_R &= ~0x03;
}
uint8_t UART0_read (void)
{
  while((UART0_FR_R & 0x0010) != 0) {}
  return (uint8_t)(UART0_DR_R & 0xFF);
}
```

04

Fourth Topic

RTOS Comparison

	FreeRTOS	Keil RTX	TI-RTOS

Multithreading and Scheduling	The scheduling mechanism is to make sure that the highest priority task (each task must have a priority) that can execute is the task given to the processor. This sometimes requires dividing processing time reasonably between tasks that have equal priorities if they are prepared to run simultaneously, where the kernel makes a round robin pattern, where each thread gets a full tick before switching to the next.
	The FreeRTOS creates the idle task, which will execute only when there are no other tasks executing, and it can always execute.[1]
	Thread Switching code:
	The switching from a lower priority task A, to higher priority task B. Task B has previously been suspended so its context has already been stored on the task B stack.
	`//ISR for the RTOS tick`
	`void SIG_OUTPUT_COMPARE1A( void ){`
	`vPortYieldFromTick();`
	`asm volatile ( "reti" );}`
	
	`void vPortYieldFromTick( void ){`
	`portSAVE_CONTEXT();`
	`vTaskIncrementTick();`
	`vTaskSwitchContext();`
	`portRESTORE_CONTEXT();`
	`asm volatile ( "ret" );}`
		Keil RTX has flexible scheduling like round-robin, pre-emptive, and collaborative.
		A thread is made by the function osThreadNew, and relying upon the thread priority, it is either placed in the ready or running state.
		The active thread with the most priority turns into the running thread as long as it doesn't hold up to any event.
		The running thread moves into the blocked state when it is belated, waiting for an event or suspended.
		Active threads can be ended whenever through the function osThreadTerminate.[3][6]
		
		
		Fig. (3): State transitions.
			Tasks are the same as threads that conceptually execute functions simultaneously. In the running state there is always only one task, regardless of whether it is just the idle task. Priorities are given to Tasks, and more than one task can have similar priority. Tasks are prepared to execute by most noteworthy to least priority; tasks of a priorities are planned for according to arrival times. The running task is seized and rescheduled to execute if a task of higher priority is in the ready state. When a task is set, it has its own runtime stack for saving local variables and nesting of function calls. All tasks executing inside a program share a joint group of global variables. The stack is the context of the task. The task creation is made in the main() function, before the kernel's scheduler is set by BIOS_start().[7][8]

Tick	The kernel measures time using a **tick** count variable. The tick is just a timer that has a period set to match the desired tick rate. By default, this is set to 1ms which provides a good balance between task speed and overhead of task switching. For ARM core processors, there is a special timer designed specifically for providing an RTOS its tick. The SysTick Timer provides its own clock configuration, counter value, and interrupt flag. This allows you to set up the SysTick to provide FreeRTOS' tick and use all the other timers for your program's needs. A timer interrupt (the RTOS **tick interrupt**) increments the tick count with strict temporal accuracy – allowing the real time kernel to measure time to a resolution of the chosen timer interrupt frequency. Each time the tick count is incremented, the real time kernel must check if it is now time to unblock or wake a task. It is possible that a task woken or unblocked during the tick ISR will have a priority higher than that of the interrupted task. If this is the case the tick ISR switches context to the newly woken/unblocked task – effectively interrupting one task but returning to another. Higher priority tasks are immediately switched to if they become unblocked due to a semaphore or queue releasing them and the next task takes over if any task goes into the blocked state (waiting for a semaphore or queue).[2]	The **OS Tick API** is an interface to a system timer that generates the Kernel Ticks. All Cortex-M processors provide a unified System Tick Timer that is typically used to generate the RTOS Kernel Tick. The CMSIS-RTOS RTX functions provide delays in milliseconds that are derived from the RTX Timer Tick. #define OS_TICK Specifies the RTX Timer Tick interval in microseconds (us). This value is used to calculate timeout values.When the SysTick core timer is enabled the value is also used to configure the SysTick timer. It is recommended to configure the RTX Timer tick to 1000 us which results in a timeout granularity of 1 millisecond. int32_t OS_Tick_Setup(uint32_t freq, IRQHandler_t handler); OS_Tick_Setup sets the tick timer to generate periodic kernel ticks. The timer should be configured to generate periodic interrupts at frequency specified by *freq*. The parameter *handler* defines the interrupt handler function that is called. The timer should only be initialized and configured but must not be started to create interrupts.[6]	The ti.sysbios.knl.Clock module is accountable for the periodic tick that the kernel utilizes to hold time track. All SYS/BIOS APIs that require a timeout parameter explicate the timeout in terms of clock ticks. The clock module utilizes the ti.sysbios.hal.Timer module to make a timer to create the tick, which is fundamentally a periodic call to Clock_tick() function. The clock module supplies APIs to start, stop and set the tick, which allow you to make frequency modifications at runtime. These three APIs aren't reentrant and gates need to be utilized to guard them. Clock_tickStart() function restarts the timer utilized to make the clock tick by calling Timer_start(). Clock_tickStop() function stops the timer by calling Timer_stop(). Clock_tickReconfig() Calls Timer_setPeriodMicroseconds() internally to reconfigure the timer, and it malfunctions if the timer can't give Clock.tickPeriod at the actual CPU frequency. Clock_getTicks() gets the number of clock ticks that have passed since startup. The value restored wraps back to zero after it comes to the highest value.[8]

Task Management	**Create**	BaseType_t xTaskCreate(TaskFunction_t pvTaskCode, const char * const pcName, configSTACK_DEPTH_TYPE usStackDepth, void *pvParameters, UBaseType_t uxPriority, TaskHandle_t *pxCreatedTask); Forms a new task, and appends it to the queue of tasks that are ready to run. *PvTaskCode:* Pointer to the task entree function. *PcName:* A name for the task. *usStackDepth:* The number of words to assign for the task's stack. *PvParameters:* The task's parameter. *UxPriority:* The task's priority. If the task was made correctly, then pdPASS is returned. Otherwise errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY is returned.[1]	OS_TID os_tsk_create (void (**task*) (void),U8 *priority*); Forms the task defined by the *task* function pointer argument and then appends the task to the ready queue. It assigns a task identifier value (TID) to the new task. The priority argument defines the priority for the task. If the new priority has a higher priority than the as of now executing task, at that point a task switch happens quickly to execute the new task. Returns the task identifier value (TID) of the new task. If the function malfunctions, it returns zero.[6]	Task_Handle Task_create(Task_FuncPtr fxn, const Task_Params *params, Error_Block *eb); Forms a new task object. The *fxn* parameter utilizes the FuncPtr type to give a pointer to the function the Task object must run. params: per-instance configuration parameters, or NULL to select default values. eb: active error-handling block, or NULL to select default policy. If it successes, it will return the handle of the new task object. If it fails, it will return NULL unless it aborts.[9]

	Suspend/ Delete	void vTaskSuspend(TaskHandle_t xTaskToSuspend);
		Suspends any task.
		XtaskToSuspend: Handle to the task being suspended. Passing a NULL handle will suspend the calling task.
		void vTaskResume(TaskHandle_t xTaskToResume);
		Resumes a suspended task.
		XtaskToResume: Handle to the task being readied.[1]
			OS_RESULT os_tsk_delete (OS_TID *task_id*);
			
			Terminate a task that has finished all its work or that is not required anymore.
			Returns OS_R_OK if the task was successfully terminated and deleted, else it returns OS_R_NOK.[6]
				void Task_sleep(Uint nticks);
				Delay execution of the current task.
				Changes the running task to blocked state and delays its execution for nticks increments. After the interval of time has passed, the task returns to the ready state and is tabled for execution.
				nticks: Number of ticks.[9]

	Set Priority	void vTaskPrioritySet(TaskHandle_t xTask, UBaseType_t uxNewPriority); Set the execution priority of a task. *Xtask:* Handle to the task for which the priority is being set. Passing a NULL handle results in the priority of the calling task being set. *uxNewPriority:* Task's new priority.[1]	OS_RESULT os_tsk_prio (OS_TID *task_id*, U8 *new_prio*); Modifies the priority of the task specified by the *task_id*. Returns OS_R_OK, if the priority of the task is changed, and OS_R_NOK, if the task with *task_id* doesn't exist or hasn't been started.[6]	Uint Task_setPri(Task_Handle handle, Int newpri); Sets the priority of a task. *handle:* handle of a previously-created Task instance object. *newpri:* Task's new priority. Returns the old priority of the task.[9]

Hardware Abstraction Layer		There are multiple hardware abstraction layer (HAL) drivers that are provided by third parties for the FreeRTOS. One of which is FreeRTOS Hardware Abstraction Layer (http://freertoshal.github.io/), which has simple architecture at user, and driver level, with quick response of drivers.
		Its main idea is a hal struct which is the main interface for drivers, where each driver shall implement the interface #include <hal.h>.
		All driver interfaces will have an init function and a deinit function.
		`int32_t hal_init(void * data);`
		`int32_t hal_deinit(void * data);`
		The init function initializes the hal driver. The deinit function disable the driver instance.
		Each driver can make the instances threadsave, with the function lock and unlock the driver can lock and unlock the instances.
		`#define HAL_LOCK(data, waittime, errcode);`
		data: A driver Struct like hal.
		waittime: Maximum waiting time.
		Errcode: Error code that is returned automatically.
		All pointers to all instants are stored in a separated section sorted by driver type (like uart, pwm, and timer).
		For driver without a specified type a generic array is created named hal use HAL_ADD() Macro to create a new entry.
		`#define HAL_ADD(ns, p)`
		HAL_ADDDEV(**hal**, ns, p)
		Add some devices without global namespace. All driver instants are static allocated.[5]
			ARM's Cortex Microcontroller Software Interface Standard (CMSIS) is a generic Hardware Abstraction Layer (HAL) for the Cortex-M processor arrangement. Keil RTX5 is an open-source, deterministic continuous working framework executing the CMSIS-RTOS v2 API.
			
			CMSIS-RTOS gives a generic API to programming parts that require RTOS functionality and gives great features to the clients and the software field, where it gives fundamental properties that are required in numerous applications and innovations.
			The standard generic list of capabilities of the CMSIS-RTOS API encourages sharing of programming parts and decreases learning attempts. Middleware parts that utilize the CMSIS-RTOS API don't depend on the type of RTOS used.
			The CMSIS-RTOS v2 API provides these features:
			
			Dynamic object creation no longer requires static memory, static memory buffers are now optional.
			Provisions for message passing in multi-core systems and C++ run-time environments.
			C interface which is binary compatible across application binary interface (ABI) compatible compilers.[3]
				SYS/BIOS gives setup and management of interrupts, cache, and timers, which are modules that straightforwardly program parts of a device's hardware and are assembled in the Hardware Abstraction Layer (HAL) bundle.
				
				The HAL APIs fall into two classifications:
				
				1-Generic APIs that are accessible over all targets and devices.
				
				2-Target/device-specific APIs that are accessible just for a particular gadget or ISA family.
				
				The standard APIs are intended to cover the vast majority utilization cases. Software engineers who are worried about guaranteeing simple portability between various TI devices are best served by utilizing the standard APIs however much as could reasonably be expected.
				
				In situations where the standard APIs can't empower utilization of a device-specific hardware attribute that is invaluable to the software application, you may decide to utilize the target/device-specific APIs, which give full hardware privilege.[8]

APIs and Modules	**Event Groups/Flags**	**1- Event Groups/Flags API Function (xEventGroupCreate):** EventGroupHandle_t xEventGroupCreate(void); Forms a new event group, and returns a handle by which the newly created event group can be referenced. If there was scanty heap untaken to create the event group then NULL is returned.[4]	**1- Event Flags CMSIS RTOS API Function (osEventFlagsNew):** osEventFlagsId_t osEventFlagsNew(const osEventFlagsAttr_t *attr); Forms a new event flags object that to send events across threads. *attr*: Identifies additional event flags attributes. Regular attributes if set to *NULL*. Returns event flags object identifier for reference or NULL in fault situation.[6]	**1- Event Kernel Module Function (Event_create):** Event_Handle Event_create(const Event_Params *params, Error_Block *eb); Forms a new Event object. params: per-instance configuration parameters, or NULL to select default values. eb: active error-handling block, or NULL to select default policy.[9]
	Message buffer	**2- Message buffer API function (xMessageBufferCreate):** MessageBufferHandle_t xMessageBufferCreate(size_t xBufferSizeBytes); Sets up a new message buffer using dynamically allocated memory. *XbufferSizeBytes:* The total number of bytes the message buffer will be able to hold at any one time. A non-NULL value being returned indicates that the message buffer has been created successfully.[4]	**2- Message Queue CMSIS RTOS API Function (osMessageQueueNew):** osMessageQueueId_t osMessageQueueNew(uint32_t msg_count, uint32_t msg_size, const osMessageQueueAttr_t * attr); Forms and initializes a message queue object. msg_count: Maximum number of messages in queue. msg_size: Maximum message size in bytes. Returns message queue ID or NULL in fault situation.[6]	**2- Mailbox Kernel Module Function (Mailbox_create):** Mailbox_Handle Mailbox_create(SizeT msgSize, Uint numMsgs, const Mailbox_Params *params, Error_Block *eb); Forms a mailbox object that is set to hold numMsgs messages of size msgSize. Mailbox messages are kept in a queue that needs a header in front of each message. msgSize: Message size. numMsgs: Mailbox length. [9]
	Kernel Control	**3- Kernel Control API Function (vTaskSuspendAll):** void vTaskSuspendAll(void); Suspends the scheduler. Suspending the scheduler forbids a context switch from happening but leaves interrupts allowed. If an interrupt requests a context switch while the scheduler is suspended, then the request is held	**3- Kernel Control CMSIS RTOS API Function (osKernelSuspend):** uint32_t osKernelSuspend(void); The return value can be used to find the amount of ticks until the next tick-based kernel event will occur, for example a delayed thread becomes ready	**3- Task Kernel Module Function (Task_disable):** Uint Task_disable(); Disables all other Tasks from running until restore is called, where together they control task scheduling, in which they allow you to

		pending and is performed only when the scheduler is resumed. Calls to xTaskResumeAll() shifts the scheduler out of the Suspended state.[4]	again. It is advisable to set up the low power timer to make a wake-up interrupt according to this return value.[6]	ensure that statements that must be performed together during critical processing are not preempted by other Tasks.[9]
	Timer	**4- Timer API Function (xTimerCreate):** TimerHandle_t xTimerCreate (const char * const pcTimerName, const TickType_t xTimerPeriod, const UBaseType_t uxAutoReload, void * const pvTimerID, TimerCallbackFunction_t pxCallbackFunction); Forms a new software timer instance and returns a handle by which the timer can be referenced. *PcTimerName:* A name for the timer. x*timerPeriod:* The period (in ticks). u*xAutoReload:* If uxAutoReload is set to pdTRUE, then the timer will expire repeatedly with a frequency set by the xTimerPeriod parameter, and if set to pdFALSE, then it is a one-shot timer. *pvTimerID:* An identifier that is assigned to the timer being created. *pxCallbackFunction:* The function to call when the timer expires.[4]	**4- Timer CMSIS RTOS API Function (osTimerNew):** osTimerId_t osTimerNew(osTimerFunc_t func,osTimerType_t type, void * argument, const osTimerAttr_t * attr) Forms a one-shot or periodic timer and associates it with a callback function with *argument*. The timer is in stopped state until it is started with osTimerStart. *func*: Function pointer to callback function. *type*: osTimerOnce for one-shot or osTimerPeriodic for periodic behavior. *argument*: Argument to the timer callback function. Returns timer ID or NULL in fault situation.[6]	**4- Timer HAL Module Function (Timer_create):** Timer_Handle Timer_create(Int id, Timer_FuncPtr tickFxn, const Timer_Params *params, Error_Block *eb); Sets up a timer (that is, to mark a timer for use) and configure it to call a tickFxn when the timer expires. Use this module only if you don't require any custom configuration of the timer peripheral. Create malfunctions if timer peripheral is inaccessible. To request any available timer use "ANY" as the id. *id*: Timer id range from zero to a platform particular value. *tickFxn*: Function that runs upon timer expiry.[9]
	Semaphore	**5- Semaphores Module Function (xSemaphoreCreateBinary):** SemaphoreHandle_t xSemaphoreCreateBinary(void); Creates a binary semaphore, and returns a semaphore handle to be referenced with. The semaphore is created in the empty state, which means that the semaphore must first be set using the xSemaphoreGive() API function before it can later be obtained using the xSemaphoreTake() function.[4]	**5- Semaphores CMSIS RTOS API Function (osSemaphoreNew):** osSemaphoreId_t osSemaphoreNew(uint32_t max_count, uint32_t initial_count, const osSemaphoreAttr_t * attr); Creates a semaphore object, and returns the pointer to the semaphore object identifier or *NULL* in fault situation. *max_count*: Maximum number of available tokens. *initial_count:* Initial number of addressable tokens.[6]	**5- Semaphore Kernel Module Function (Semaphore_create):** Semaphore_Handle Semaphore_create(Int count, const Semaphore_Params *params, Error_Block *eb); Forms a new Semaphore object which is set to count. *count*: Initial semaphore count. Returns a semaphore handle to be referenced with.[9]

<table>
<tr>
<td>RTOS Configuration</td>
<td>
Customizattion is done using a configuration file called FreeRTOSConfig.h, which joins the RTOS kernel to the application being built, where every application must have this header file in its pre-processor include path.
Constants that start with “config” specifies attributes, and features of the kernel, while those start with “INCLUDE_” are used to include or exclude API functions.
Configuration File Example:
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H
// Include header files
#define configCPU_CLOCK_HZ 60000000
The frequency in Hz at which the internal clock that drives the peripheral used to make the tick interrupt will be executing – this is reguraly the same clock that drives the internal CPU clock. This value is needed in order to correctly set timer peripherals.
#define configTICK_RATE_HZ 250
The frequency of the tick interrupt. The tick interrupt is used to calculate time. Hence a higher tick frequency means time can be calculated to a higher resolution. Nevertheless, a high tick frequency also means that the RTOS kernel will use more CPU time so will be less effective.
More than one task can have the same priority. The RTOS scheduler will divide the processor time between tasks of the same priority by switching
</td>
<td>
The file "RTX_Config.h" characterizes the configuration parameters of CMSIS-RTOS RTX and must be part of every project that is utilizing the CMSIS-RTOS RTX kernel. The document "RTX_Config.c" contains default implementations of the functions osRtxIdleThread and osRtxErrorNotify. Both functions can be overwritten with a customized behavior by redefining them as feature of the user code.
RTX_Config.h System Configuration:
#define OS_DYNAMIC_MEM_SIZE
Defines the combined global dynamic memory size for the Global Memory Pool.
#define OS_TICK_FREQ
Defines base time unit for delays and timeouts in Hz.
#define OS_ROBIN_ENABLE
Enables Round-Robin context switching.
#define OS_ROBIN_TIMEOUT
Specifies the time slice for threads.
#define OS_ISR_FIFO_QUEUE
RTOS Functions called from ISR save requests in this buffer.
#define OS_OBJ_MEM_USAGE
Enables object memory usage counters to measure the
</td>
<td>
The *.cfg configuration files characterize the configuration parameters of SYS/BIOS applications.
By modifying these files in the project configuration of SYS/BIOS applications can be achieved.
These files are written in a scripting language like JavaScript. While you can edit this file with a text editor, CCS provides a graphical configuration editor called XGCONF.
XGCONF is helpful; because it gives you a simple method to see the accessible alternatives and your current configuration. Since modules and instances are excited behind-the-scenes when the configuration is processed, XGCONF is a useful tool for viewing the effects of these internal actions and for discovering conflicts.
In the cfg Script tab of a configuration, you can choose to Revert File to get the last saved configuration file you can also save the current configuration.
The next line in the *.cfg file for a TI-RTOS application makes all TI-RTOS drivers to be available to the application build.
var driversConfig = xdc.useModule('ti.drivers.Co
</td>
</tr>
</table>

	between the tasks with each tick. The context switch overhead must be taken into consideration. #define configMAX_TASK_NAME_LEN 16 The maximum length of the illustrative name given to a task when the task is made. #define configMINIMAL_STACK_SIZE 128 The size of the stack that idle task uses. Generally this should not be decreased from the value set in the file supplied with the demo application for the port in use. The stack size is specified in words. If each item placed on the stack is 32-bits, then a stack size of 200 means 800 bytes (each 32-bit stack item consuming 4 bytes). //Software timer related definitions. #define configUSE_TIMERS 1 Set to one to include software timer functionality, or zero to exclude software timer functionality.[4]	maximum memory pool requirements singly for each RTOS object type. **Thread Configuration:** #define OS_THREAD_NUM Defines maximum number of user threads that can be active simultaneously. Employs to user threads with system given memory for control blocks. #define OS_STACK_SIZE Defines stack size for threads with zero stack size specified. #define OS_STACK_CHECK Enable stack overrun checks at thread switch. **Timer Configuration:** #define OS_TIMER_NUM Defines maximum number of objects that can be active simultaneously. Employs to objects with system given memory for control blocks.[6]	nfig'); This doesn't mean that all the drivers will be compiled into the application. To minimize the memory footprint of the application, only driver library code called by the application will be included in the compiled and linked executable. By default, the application is configured to use non-instrumented libraries, which doesn't process 'Log' events and Asserts. You can choose the instrumented libraries by using XGCONF or by adding this line to your application's *.cfg file: driversConfig.libType = driversConfig.LibType_Instrumented;[8]

References:

[1] https://www.freertos.org/features.html 7 June 2020.
[2] http://www.learnitmakeit.com/freertos-tick/ 7 June 2020.
[3] Jonathan W. Valvano, Real-Time Operating Systems for Arm Cortex-M microcontrollers, V3, Fourth Edition, January 2017.
[4] The FreeRTOS Reference Manual.
[5] http://freertoshal.github.io/doxygen/group__HAL.html 7 June 2020.
[6] https://www.keil.com/pack/doc/CMSIS/RTOS2/html/index.html 7 June 2020.
[7] http://software-dl.ti.com/lprf/simplelink_cc26x2_sdk-1.60/docs/thread/html/tirtos/rtos-overview.html#tasks 7 June 2020.
[8] TI-RTOS Kernel (SYS/BIOS) User's Guide.
[9] http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/tirtos/2_14_04_31/exports/tirtos_full_2_14_04_31/products/bios_6_42_03_35/docs/cdoc/ti/sysbios/knl/package.html 7 June 2020.