Program 1.

```
/* File Name: freertos.c
 File Description: This is the main file that contains the primary function
calls.
 Author Name: Nitik Satish Gupta and Rakesh Kumar */
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "semphr.h"
#include "fsl_dac.h"
#include "fsl_device_registers.h"
#include "fsl_debug_console.h"
#include "fsl_port.h"
#include "fsl_gpio.h"
#include "board.h"
#include "fsl_port.h"
#include "pin_mux.h"
#include "fsl_common.h"
#include "clock_config.h"
#include "math.h"
/***************************************************************************
*****
 * Definitions

****************************************************************************
***/

#define DEMO_DAC_BASEADDR DAC0

#define BUFF_LENGTH 4
#define DMA_CHANNEL 0
#define DMA_SOURCE 63
/* clang-format on */

/***************************************************************************
*****
 * Variables

****************************************************************************
***/


static void SwTimerCallback(TimerHandle_t xTimer);
#define SINUS_LENGTH 51
#define SW_TIMER_PERIOD_MS (100 / portTICK_PERIOD_MS)
dac_config_t dacConfigStruct;

static int dacValue[SINUS_LENGTH] ;
/*  Name: DACInit()
    Description: This is the function that does the initialization for the
DAC.
    Inputs: void
    Returns: void */

void DACInit()
{
         DAC_GetDefaultConfig(&dacConfigStruct);
               DAC_Init(DEMO_DAC_BASEADDR, &dacConfigStruct);
               DAC_Enable(DEMO_DAC_BASEADDR, true);             /* Enable
output. */
               DAC_SetBufferReadPointer(DEMO_DAC_BASEADDR, 0U);
}


/*****************************************************************************
*****
 * Code

*****************************************************************************
***/
/*!
 * @brief Main function
 */
//BaseType_t DAC_task;
//TaskHandle_t* DAC_task;
int main(void)
{
/* Define the init structure for the input switch pin */
#ifdef BOARD_SW_NAME
    gpio_pin_config_t sw_config = {
        kGPIO_DigitalInput, 0,
    };
#endif
    uint8_t index=0;
    TimerHandle_t SwTimerHandle = NULL;
    //Initializing the Queue
    LED_GREEN_INIT(1);
    for(index = 0;index<51;index++)
       {
          dacValue[index] = ( ( ( sin(index * (6.28/50)))+2)*4096/3.3 );
          PRINTF("SINE = %d\n\r",dacValue[index]);
       }
    DACInit();
   //  lptmrInit();
#if configUSE_TICKLESS_IDLE

#endif
    BOARD_InitPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();

    /*Create tickless task*/
  /* Make sure the read pointer to the start. */
                                                         /*
                                                         * The buffer is not
enabled, so the read pointer can not move automatically. However, the
buffer's read pointer
                                                         * and itemss can be
written manually by user.
                                                         */
       SwTimerHandle = xTimerCreate("SwTimer1",           /* Text name. */
                                    SW_TIMER_PERIOD_MS, /* Timer period. */
                                    pdTRUE,             /* Enable auto
reload. */
                                    0,                  /* ID is not used.
*/
                                    SwTimerCallback);   /* The callback
function. */
    /*Task Scheduler*/
```

```
       xTimerStart(SwTimerHandle, 0);
    vTaskStartScheduler();
    for (;;)
        ;
}

/* Tickless Task */
uint8_t i=0;
/*  Name: SwTimerCallback
    Description: This is the function that does the calls the Timer
Function.
    Inputs: xTimer
    Returns: void */
static void SwTimerCallback(TimerHandle_t xTimer)
{

    LED_GREEN_TOGGLE();
    DAC_SetBufferValue(DEMO_DAC_BASEADDR, 0U, dacValue[i]);
    PRINTF("\n\rDAC Value:%d", dacValue[i]);
    i++;
    if(i==SINUS_LENGTH)
    {
        i=0;
    }
    //vTaskResume(Hello_task);
}
```

Program 2.

```
/* File Name: freertos.c
File Description: This is the main file that contains the primary function
calls.
Author Name: Nitik Satish Gupta and Rakesh Kumar */
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "semphr.h"
#include "fsl_dac.h"
#include "fsl_device_registers.h"
#include "fsl_debug_console.h"
#include "fsl_port.h"
#include "fsl_gpio.h"
#include "board.h"
#include "fsl_port.h"
#include "pin_mux.h"
#include "fsl_common.h"
#include "clock_config.h"
#include "fsl_adc16.h"
#include "fsl_dma.h"
#include "fsl_dmamux.h"
#if configUSE_TICKLESS_IDLE
#include "fsl_lptmr.h"

#endif
#include "math.h"
#include "circular_buffer.h"
/***************************************************************************
*****
 * Definitions

****************************************************************************
***/

#define BOARD_SW_GPIO BOARD_SW2_GPIO
#define BOARD_SW_PORT BOARD_SW2_PORT
#define BOARD_SW_GPIO_PIN BOARD_SW2_GPIO_PIN
#define BOARD_SW_IRQ BOARD_SW2_IRQ
#define BOARD_SW_IRQ_HANDLER BOARD_SW2_IRQ_HANDLER
#define BOARD_SW_NAME BOARD_SW2_NAME
#define MAX_LOG_LENGTH 20
/* @brief FreeRTOS tickless timer configuration. */
#define BOARD_LPTMR_IRQ_HANDLER LPTMR0_IRQHandler /*!< Timer IRQ handler.
*/
#define TICKLESS_LPTMR_BASE_PTR LPTMR0            /*!< Tickless timer base
address. */
#define TICKLESS_LPTMR_IRQn LPTMR0_IRQn           /*!< Tickless timer IRQ
number. */

/* Task priorities. */
/* clang-format off */
#define tickless_task_PRIORITY   ( configMAX_PRIORITIES  )
#define SW_task_PRIORITY   ( configMAX_PRIORITIES - 1 )
#define TIME_DELAY_SLEEP      100
#define DEMO_DAC_BASEADDR DAC0
#define DEMO_ADC16_BASE ADC0
#define DEMO_ADC16_CHANNEL_GROUP 0U
```

```
#define DEMO_ADC16_USER_CHANNEL 0U /*PTE20, ADC0_SE0 */
/* Interrupt priorities. */
#define SW_NVIC_PRIO 2
#define SIZE 64
#define BUFF_LENGTH 4
#define DMA_CHANNEL 0
#define DMA_SOURCE 63
/* clang-format on */
/**************************************************************************
*****
 * Prototypes

***************************************************************************
***/
extern void vPortLptmrIsr(void);
LPTMR_Type *vPortGetLptrmBase(void);
IRQn_Type vPortGetLptmrIrqn(void);

/**************************************************************************
*****
 * Variables

***************************************************************************
***/
static void DAC_TRANSFER_task(void *pvParameters);
static void ADC_TRANSFER_task(void *pvParameters);
static void DSP_Task(void *pvParameters);
static void SW_task(void *pvParameters);

#define SINUS_LENGTH 51
volatile uint32_t i=0;
SemaphoreHandle_t xSWSemaphore = NULL;
adc16_config_t adc16ConfigStruct;
adc16_channel_config_t adc16ChannelConfigStruct;
dac_config_t dacConfigStruct;
circularbuff_handle_t cbuf_handle;
uint16_t * buffer;
uint8_t count=0;
dma_handle_t g_DMA_Handle;
dma_transfer_config_t transferConfig;
uint32_t counter = 0;
/* Data log queue handle */
static QueueHandle_t log_queue = NULL;

//Semaphore for the LED resource
SemaphoreHandle_t xSemaphore_LED = NULL;
const TickType_t xDelay1ms = pdMS_TO_TICKS( 1 );
volatile bool g_Transfer_Done = false;
volatile uint8_t flag=0;
uint16_t ADC_Buffer[51];
uint16_t DSP_Buffer[51],Max,Min;
char logg[MAX_LOG_LENGTH + 1];
double mean,StdDev;
static const int dacValue[SINUS_LENGTH];

/*  Name: log_task()
    Description: This is the function to display the logging information
stored in the queue.
    Inputs: void
    Returns: void */
static void log_task()
```

```
{

    char log_temp[MAX_LOG_LENGTH + 1];
    while (counter--)
    {
        //Get the logging informatin stored in the queue
        xQueueReceive(log_queue, log_temp, portMAX_DELAY);
        PRINTF("Log %d: %s\r\n", counter, log_temp);
    }
}

/*  Name: ConvertTime()
    Description: This is the function to process time in the required
format.
    Inputs: void
    Returns: void */
void ConvertTime(double ticks)
{       double mil;
        uint8_t S=0,M=0;
        mil=ticks;
        while(mil>1000)
        {
                mil=mil-1000;
                S++;
        }
        while(S>60)
        {
                S=S-60;
                M++;
        }
                PRINTF("\n\r%d:%d.%d",M,S,mil);
                PRINTF("\n\r%d",ticks);
}

/*  Name: ADCInit()
    Description: This is the function that does the initialization for the
ADC.
    Inputs: void
    Returns: void */
void ADCInit()
{

         ADC16_GetDefaultConfig(&adc16ConfigStruct);
        #ifdef BOARD_ADC_USE_ALT_VREF
            adc16ConfigStruct.referenceVoltageSource =
kADC16_ReferenceVoltageSourceValt;
        #endif
            //Initializing the ADC
            ADC16_Init(DEMO_ADC16_BASE, &adc16ConfigStruct);
            /* Make sure the software trigger is used. */
            ADC16_EnableHardwareTrigger(DEMO_ADC16_BASE, false);
        #if defined(FSL_FEATURE_ADC16_HAS_CALIBRATION) &&
FSL_FEATURE_ADC16_HAS_CALIBRATION
            if (kStatus_Success ==
ADC16_DoAutoCalibration(DEMO_ADC16_BASE))
            {
                PRINTF("ADC16_DoAutoCalibration() Done.\r\n");
            }
            else
            {
                PRINTF("ADC16_DoAutoCalibration() Failed.\r\n");
            }
        #endif /* FSL_FEATURE_ADC16_HAS_CALIBRATION */
            PRINTF("Press any key to get user channel's ADC value
...\r\n");

            adc16ChannelConfigStruct.channelNumber =
DEMO_ADC16_USER_CHANNEL;
            adc16ChannelConfigStruct.enableInterruptOnConversionCompleted =
false;
        #if defined(FSL_FEATURE_ADC16_HAS_DIFF_MODE) &&
FSL_FEATURE_ADC16_HAS_DIFF_MODE
            adc16ChannelConfigStruct.enableDifferentialConversion = false;
        #endif /* FSL_FEATURE_ADC16_HAS_DIFF_MODE */

}

/*  Name: DACInit()
    Description: This is the function that does the initialization for the
DAC.
    Inputs: void
    Returns: void */
void DACInit()
{
        //Getting the default configuration for the DAC
         DAC_GetDefaultConfig(&dacConfigStruct);
         //Initializing the DAC
         DAC_Init(DEMO_DAC_BASEADDR, &dacConfigStruct);
         //Enabling the DAC
         DAC_Enable(DEMO_DAC_BASEADDR, true);             /* Enable output.
*/
         DAC_SetBufferReadPointer(DEMO_DAC_BASEADDR, 0U);
}

/*  Name: CircularBuffInit()
    Description: This is the function that does the initialization for the
circular buffer.
    Inputs: void
    Returns: void */
void CircularBuffInit()
{
        //Memory allocation for the buffer to be provided to the circular
buffer handle
        buffer = (uint16_t *)malloc(SIZE*sizeof(uint16_t));
        assert(buffer);
        //Memory allocation for the circular buffer structure
        cbuf_handle = (circular_buffer_t
*)malloc(SIZE*sizeof(circular_buffer_t));
        bool res = circular_buffer_valid(cbuf_handle);
        if(res)
        {
                 PRINTF("\n\r The buffer pointer is valid");
        }
        else
        {
                 PRINTF("\n\r Invalid Buffer pointer");
        }
        //Initializing the circular buffer
        circular_buffer_init(buffer, SIZE, cbuf_handle);

        //Checking for the proper initialization of the buffer
        res = circular_buffer_init_check(cbuf_handle);
        if(res)
        {
                PRINTF("\n\r The buffer pointer is initialized");
        }
        else
        {
                PRINTF("\n\r Couldn't initialize buffer");
        }
        cbuf_handle->tail = 0;
}

/*  Name: DMA_Callback()
    Description: This is the callback function for the DMA.
    Inputs: void
    Returns: void */
void DMA_Callback(dma_handle_t *handle, void *param)
{
        //Indicating the transfer done status
    g_Transfer_Done = true;
}

/*  Name: queue_init()
    Description: Queue init for the logging functionality.
    Inputs: void
    Returns: void */
void queue_init(uint32_t queue_length, uint32_t max_log_length)
{
        //Creating the queue
    log_queue = xQueueCreate(queue_length, max_log_length);
}

/*  Name: log_add()
    Description: This is the function that adds the logginf information
into the queue.
    Inputs: void
    Returns: void */
void log_add(char *log)
{
        //Adding information inside the queue
    xQueueSend(log_queue, logg, 0);
}

//void DMA_func()
//{
//      DMA_PrepareTransfer(&transferConfig, srcAddr, sizeof(srcAddr[0]),
destAddr, sizeof(destAddr[0]), sizeof(srcAddr),
//                              kDMA_MemoryToMemory);
//          DMA_SubmitTransfer(&g_DMA_Handle, &transferConfig,
kDMA_EnableInterrupt);
//          DMA_StartTransfer(&g_DMA_Handle);
//          /* Wait for DMA transfer finish */
//          while (g_Transfer_Done != true)
//          {
//          }
//}

/*
 * @brief Main function
 */
int main(void)
{
```

```
/* Define the init structure for the input switch pin */
#ifdef BOARD_SW_NAME
    gpio_pin_config_t sw_config = {
        kGPIO_DigitalInput, 0,
    };
#endif

    //Initializing the Queue
    queue_init(10, MAX_LOG_LENGTH);
    vSemaphoreCreateBinary(xSemaphore_LED);
    if (xSemaphore_LED == NULL)
    {
        PRINTF("xSemaphore_producer creation failed.\r\n");
        vTaskSuspend(NULL);
    }
    for(uint8_t index = 0;index<50;index++)
    {
        dacValue[index] = ( ( ( sin(index * (6.28/50)))+2)*4096/3.3 );
        PRINTF("SINE = %d\n\r",dacValue[index]);
    }
    LED_GREEN_INIT(1);
    LED_BLUE_INIT(1);
    xSemaphoreGive(xSemaphore_LED);
    uint8_t status = uxSemaphoreGetCount(xSemaphore_LED);
    if(status)
    {
       LED_GREEN_ON();
       //xSemaphoreGive(xSemaphore_LED);
       xSemaphoreTake(xSemaphore_LED, 0);
       sprintf(logg, "LED status: %d", 1);
       log_add(logg);
       counter++;
    }
    else
    {
       LED_GREEN_OFF();
       xSemaphoreGive(xSemaphore_LED);
       sprintf(logg, "LED status: %d", 0);
       log_add(logg);
       counter++;
    }
    status = uxSemaphoreGetCount(xSemaphore_LED);
    if(!status)
    {
        LED_GREEN_OFF();
        xSemaphoreGive(xSemaphore_LED);
        sprintf(logg, "LED status: %d", 0);
        log_add(logg);
        counter++;
    }
    else
    {
        LED_GREEN_ON();
        sprintf(logg, "LED status: %d", 1);
        log_add(logg);
        counter++;
    }
    log_task();
    ADCInit();
    DACInit();
    CircularBuffInit();
```

```
#if configUSE_TICKLESS_IDLE

#endif
    BOARD_InitPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();

    /* Print a note to terminal. */
    PRINTF("Tickless Demo example\r\n");
#ifdef BOARD_SW_NAME
    PRINTF("Press %s to wake up the CPU\r\n", BOARD_SW_NAME);
    /* Init input switch GPIO. */
//    PORT_SetPinInterruptConfig(BOARD_SW_PORT, BOARD_SW_GPIO_PIN,
kPORT_InterruptFallingEdge);
//    NVIC_SetPriority(BOARD_SW_IRQ, SW_NVIC_PRIO);
//    EnableIRQ(BOARD_SW_IRQ);
//    GPIO_PinInit(BOARD_SW_GPIO, BOARD_SW_GPIO_PIN, &sw_config);
#endif
    /*Create tickless task*/
  /* Make sure the read pointer to the start. */
     /*
     * The buffer is not enabled, so the read pointer can not move
automatically. However, the buffer's read pointer
     * and itemss can be written manually by user.
     */
    xTaskCreate(DAC_TRANSFER_task, "DAC_TRANSFER_task",
configMINIMAL_STACK_SIZE + 100, NULL, tickless_task_PRIORITY-2, NULL);
    xTaskCreate(ADC_TRANSFER_task, "ADC_TRANSFER_task",
configMINIMAL_STACK_SIZE + 1000, NULL, tickless_task_PRIORITY-3, NULL);
    xTaskCreate(SW_task, "Tickless_task", configMINIMAL_STACK_SIZE + 38,
NULL, tickless_task_PRIORITY, NULL);

    /*Task Scheduler*/
    vTaskStartScheduler();
    for (;;)
        ;
}

/*  Name: DMA0_IRQHandler()
    Description: This is the IRQ handler for the DMA-0.
    Inputs: void
    Returns: void */
void DMA0_IRQHandler(void)
{       PRINTF("\n\r DMA Done");
        ConvertTime(xTaskGetTickCount());
        //Task creation for the DSP
        xTaskCreate(DSP_Task, "DSP_Task", configMINIMAL_STACK_SIZE-20 ,
NULL, tickless_task_PRIORITY-1, NULL);
        //Disabling the DMA interrupt
        DisableIRQ(DMA0_IRQn);
}
/* Tickless Task */
/*  Name: DAC_TRANSFER_task()
    Description: This is the function that executes the DMA transfer
operation.
    Inputs: pvParameters
    Returns: void */
static void DAC_TRANSFER_task(void *pvParameters)
{
    for (;;)
    {
        i++;
        LED_GREEN_TOGGLE();

        DAC_SetBufferValue(DEMO_DAC_BASEADDR, 0U, dacValue[i]);
        PRINTF("\n\rDAC Value:%d", dacValue[i]);
        if(i==SINUS_LENGTH)
        {
               i=0;
        }
        vTaskDelay(TIME_DELAY_SLEEP);
    }
}

/*  Name: ADC_TRANSFER_task()
    Description: This is the function that executes the ADC transfer
operation.
    Inputs: pvParameters
    Returns: void */
static void ADC_TRANSFER_task(void *pvParameters)
{       TickType_t DMA_Start,DMA_Stop;                      //Adding the
data to the circular buffer
        for (;;)
    {
        ADC16_SetChannelConfig(DEMO_ADC16_BASE, DEMO_ADC16_CHANNEL_GROUP,
&adc16ChannelConfigStruct);
        while (0U == (kADC16_ChannelConversionDoneFlag &

        //Getting the status flags of the ADC
        ADC16_GetChannelStatusFlags(DEMO_ADC16_BASE,
DEMO_ADC16_CHANNEL_GROUP)))
        {
        }
        //Getting the channel conversion value
        ADC_Buffer[i]=ADC16_GetChannelConversionValue(DEMO_ADC16_BASE,
DEMO_ADC16_CHANNEL_GROUP);
        PRINTF("ADC Value: %d\r\n",ADC_Buffer[i] );
        PRINTF("\n\r%d",i );
        //Adding values into the circular buffer
        circular_buffer_add(cbuf_handle, ADC_Buffer[i]);
        uint16_t* Add=cbuf_handle->buffer;
        if(i==0)
        {
               count++;
               PRINTF("\n\r INSIDE DMA");
               //Getting the current tick count
               DMA_Start = xTaskGetTickCount();
               ConvertTime(DMA_Start);
               //Adjusting for the required delay
               DMA_Stop = DMA_Start + (500*xDelay1ms);
            while(DMA_Start != DMA_Stop)
               {
                       DMA_Start = xTaskGetTickCount();
                       GPIOB->PDDR&=0x00;
                       LED_BLUE_ON();
               }
               LED_BLUE_OFF();
               GPIOB->PDDR|=0xFF;
               LED_GREEN_INIT(1);
               LED_BLUE_INIT(1);
               //xTaskCreate(LED_Task,"Led_Task",configMINIMAL_STACK_SIZE
+ 500, NULL, tickless_task_PRIORITY-1, NULL);
```

```
                DMAMUX_Init(DMAMUX0);
            DMAMUX_SetSource(DMAMUX0, DMA_CHANNEL, DMA_SOURCE);
            DMAMUX_EnableChannel(DMAMUX0, DMA_CHANNEL);
            /* Configure DMA one shot transfer */
            DMA_Init(DMA0);
            DMA_CreateHandle(&g_DMA_Handle, DMA0, DMA_CHANNEL);
                DMA_SetCallback(&g_DMA_Handle, DMA_Callback, NULL);
                DMA_PrepareTransfer(&transferConfig, Add, sizeof(uint16_t),
DSP_Buffer, sizeof(uint16_t), SIZE*sizeof(uint16_t),kDMA_MemoryToMemory);
            DMA_SubmitTransfer(&g_DMA_Handle, &transferConfig,
kDMA_EnableInterrupt);
            DMA_StartTransfer(&g_DMA_Handle);
                            /* Wait for DMA transfer finish */
//                          while (g_Transfer_Done != true)
//                          {
//                          }
        }
        vTaskDelay(TIME_DELAY_SLEEP);
    }
}

/*  Name: SW_task()
    Description: This is the function that is responsible for the CPU
wakeup on the external interrupt.
    Inputs: pvParameters
    Returns: void */
static void SW_task(void *pvParameters)
{
        //Creating a binary semaphore
    xSWSemaphore = xSemaphoreCreateBinary();
    for (;;)
    {
        if (xSemaphoreTake(xSWSemaphore, portMAX_DELAY) == pdTRUE)
        {
            PRINTF("CPU waked up by EXT interrupt\r\n");
        }
    }
}

volatile uint16_t cnt=0;
/*  Name: DSP_Task()
    Description: This is the function that carries out the DSP related
operation.
    Inputs: pvParameters
    Returns: void */
static void DSP_Task(void *pvParameters)
{       for(;;)
        {       cnt++;
                PRINTF("\n\r DSP Task");
                uint16_t j;
                double sum=0,sumVar=0,Var;
//              for(j=0;j<51;j++)
//              {
//                      PRINTF("\n\r%d",DSP_Buffer[j]);
//              }
                for(j=0;j<51;j++)
                {       if(j==0)
                        {
                                Max=DSP_Buffer[j];
                                Min=DSP_Buffer[j];
                        }
                        if(DSP_Buffer[j]>Max)
                        {
                                Max=DSP_Buffer[j];
                        }
                        if(DSP_Buffer[j]<Min)
                        {
                                Min=DSP_Buffer[j];
                        }
                        sum+=DSP_Buffer[j];
                }
                mean=sum/50;
                for(j=0;j<51;j++)
                {
                        sumVar+=pow((DSP_Buffer[j]-mean),2);
                }
                Var=sumVar/50;
                StdDev=sqrt(Var);
                PRINTF("\n\rMAX=%d",Max);
                PRINTF("\n\rMIN=%d",Min);
                PRINTF("\n\rMEAN=%f",mean);
                PRINTF("\n\rSTANDARD DEVIATION=%f",StdDev);
                if(cnt==5)
                {
                        vTaskSuspendAll();
                }
                vTaskSuspend(NULL);
        }
}
```