Xilinx Zynq FPGA, TI DSP, MCU 기반의 회로 설계 및 임베디드 전문가 과정

강사 - Innova Lee(이상훈)

gcccompil3r@gmail.com

학생 - TaeYoung Eun(은태영)

zero_bird@naver.com

1.7 소프트웨어 타이머 API

1.7.1 타이머 생성 및 제거

1.7.1.1 xTimerCreate : 타이머 동적 생성

```
#include “FreeRTOS.h”
#include “timers.h”

TimerHandle_t xTimerCreate ( const char * pcTimerName, const TickType_t xTimerPeriod, const UBaseType_t uxAutoReload,
                             void * const pvTimerID, TimerCallbackFunction_t pxCallbackFunction );
```

새로운 소프트웨어 타이머를 만들고 생성된 소프트웨어 타이머를 참조할 수 있는 핸들을 반환한다.

각 소프트웨어 타이머에는 타이머의 상태를 유지하는데 사용되는 소량의 RAM 이 필요하다. 소프트웨어 타이머가 xTimerCreate()을 사용하여 생성되면 이 RAM 은 FreeRTOS 힙 메모리에서 자동으로 할당된다. xTimerCreateStatic()을 사용하여 소프트웨어 타이머를 만들면 추가 매개 변수가 필요하지만, 컴파일 타임에 RAM 을 정적으로 할당할 수 있도록 응용프로그램 작성자가 RAM 을 제공한다.

타이머를 생성해도 타이머는 실행되지 않는다. xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod(), xTimerChangePeriodFromISR() API 함수를 사용하여 타이머 실행을 할 수 있다.

xTimerCreate()를 사용하려면 FreeRTOSConfig.h 의 configUSE_TIMERS 및 configSUPPORT_DYNAMIC_ALLOCATION 을 1 로 설정해야 한다. configSUPPORT_DYNAMIC_ALLOCATION 은 정의되지 않아도 1 로 기본 설정되어 있다.

1.7.1.1.1 매개 변수

pcTimerName : 순전히 디버깅을 돕기 위해 타이머에 할당되는 일반 텍스트 이름이다.

xTimerPeriod : 타이머의 기간이다. 타이머 기간은 tick 으로 지정된다. pdMS_TO_TICKS()매크로는 밀리 초 단위의 시간을 tick 단위의 시간으로 변환하는데 사용할 수 있다. 예를 들어 타이머가 100 tick 후에 만료되어야만 할 경우 xNewPeriod 를 직접 100 으로 설정할 수 있다. 또는, 타이머가 500ms 후에 만료해야 하면, configTICK_RATE_HZ 가 1000 이하일 경우 xNewPeriod 를 pdMS_TO_TICKS(500)으로 설정할 수 있다.

uxAutoReload : Autoreload 타이머를 생성하려면 pdTRUE 로 설정한다. One-Shot 타이머를 작성하려면 pdFALSE 로 설정한다. Autoreload 타이머는 xTimerPeriod 매개 변수에 의해 설정된 주기로 반복적으로 만료된다. One-Shot 타이머는 한번만 만료된다. 만료된 후 One-Shot 타이머는 수동으로 다시 시작할수 있다.

pvTimerID : 만들려는 타이머에 할당된 식별자다. 식별자는 나중에 vTimerSetTimerID()API 함수를 사용하여 업데이트 할 수 있다. 동일한 콜백 함수가 여러 타이머에 할당된 경우, 콜백 함수 내에서 타이머 식별자를 검사하여 실제로 만료된 타이머를 확인할 수 있다. 또한, 타이머 식별자는 타이머의 콜백 함수 호출 사이에 값을 저장하는데 사용할 수 있다.

pxCallbackFunction : 타이머가 만료될 때 호출할 함수다. 콜백 함수는 TimerCallbackFunction_t 유형의 프로토 타입을 갖는다. 필요한 프로토 타입은 다음과 같다.

```
void vCallbackFunctionExample (TimerHandle_t xTimer);
```

1.7.1.1.2 반환 값

NULL : 타이머 데이터 구조를 할당하는데 사용되는 FreeRTOS 힙 메모리가 충분하지 않아 소프트웨어 타이머를 만들수 없다.

다른 값 : 소프트웨어 타이머가 성공적으로 생성되었으며, 반환된 값은 생성된 소프트웨어 타이머를 참조할 수 있는 핸들이다.

1.7.1.1.3 기타

```
/* 여러 타이머 인스턴스가 사용할 콜백 함수를 정의한다. 콜백 함수는 관련 타이머가 만료된 횟수를 계산하고 아무것도 하지 않으며
타이머가 10 회 만료되면 타이머를 중지한다. 카운트는 타이머의 ID 로 저장한다. */
void vTimerCallback( TimerHandle_t xTimer ) {
    const uint32_t ulMaxExpiryCountBeforeStopping = 10;
    uint32_t ulCount;

    // 이 타이머가 만료된 횟수가 타이머의  ID 로 저장된다. 카운트를 얻어 온다.
    ulCount = ( uint32_t ) pvTimerGetTimerID( xTimer );

    // 카운트를 증가시키고 타이머가 만료됐는지 확인하기 위해 ulMaxExpiryCountBeforeStopping 을 테스트한다.
    ulCount++;

    // 타이머가 10 회 만료되면 실행을 중지한다.
    if( ulCount >= xMaxExpiryCountBeforeStopping ) {
      /* 타이머 콜백 함수에서 타이머 API 함수를 호출하는 경우, 차단 시간을 사용하면 안된다.
      사용할 경우 교착 상태가 발생할 수 있다. */
      xTimerStop( pxTimer, 0 );
    } else {
      // 타이머의 ID 필드에 증가된 카운트를 다시 저장하여, 다음에 이 소프트웨어 타이머가 만료될 때 다시 읽도록 한다.
      vTimerSetTimerID( xTimer, ( void * ) ulCount );
    }
}
```

```
#define NUM_TIMERS 5

// 생성된 타이머에 대한 핸들을 저장하기 위한 배열이다.
TimerHandle_t xTimers[ NUM_TIMERS ];

void main( void ) {
    long x;

    /* 타이머를 만들고 시작한다. RTOS 스케줄러가 시작되기 전에 타이머를 시작할 경우,
    타이머는 RTOS 스케줄러가 시작되는 즉시 실행된다. */
    for( x = 0; x < NUM_TIMERS; x++ ) {
      /* tick 타이머 기간은 0 보다 커야한다, 타이머가 만료되면 자동으로 다시 로드된다, ID 는 타이머가 만료된 횟수를 저장하는데
      사용되며 0 으로 초기화된다, 각 타이머는 만료되면 같은 콜백을 호출한다. */
      xTimers[ x ] = xTimerCreate( "Timer", ( 100 * x ) + 100, pdTRUE, ( void * ) 0, vTimerCallback );
      if( xTimers[ x ] == NULL ) {
        // 타이머 생성에 실패하였다.
      } else {
        // 타이머를 시작한다. 차단 시간은 지정되지 않았고, RTOS 스케줄러가 아직 시작되지 않았기 때문에 차단 시간이 무시된다.
        if( xTimerStart( xTimers[ x ], 0 ) != pdPASS ) {
          // 타이머를 활성화 상태로 설정할 수 없다.
        }
      }
    }
    // 여기서 작업을 만든다.
    // RTOS 스케줄러를 시작하면 타이머가 이미 활성화 상태로 설정되어 있으므로, 타이머가 작동한다.
    vTaskStartScheduler();

    // 여기에 도달하지 않는다.
    for( ;; );
}
```

1.7.1.2 xTimerCreateStatic : 타이머 정적 생성

```
#include “FreeRTOS.h”
#include “timers.h”

TimerHandle_t xTimerCreateStatic (const char * pcTimerName, const TickType_t xTimerPeriod,
                                  const UBaseType_t uxAutoReload, void * const pvTimerID,
                                  TimerCallbackFunction_t pxCallbackFunction, StaticTimer_t * pxTimerBuffer );
```

새로운 소프트웨어 타이머를 만들고 생성된 소프트웨어 타이머를 참조할 수 있는 핸들을 반환한다.

각 소프트웨어 타이머에는 타이머의 상태를 유지하는데 사용되는 소량의 RAM 이 필요하다. 소프트웨어 타이머가 xTimerCreate()을 사용하여 생성되면 이 RAM 은 FreeRTOS 힙 메모리에서 자동으로 할당된다. xTimerCreateStatic()을 사용하여 소프트웨어 타이머를 만들면 추가 매개 변수가 필요하지만, 컴파일 타임에 RAM 을 정적으로 할당할 수 있도록 응용프로그램 작성자가 RAM 을 제공한다.

타이머를 생성해도 타이머는 실행되지 않는다. xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod(), xTimerChangePeriodFromISR() API 함수를 사용하여 타이머 실행을 할 수 있다.

xTimerCreateStatic()를 사용하려면 FreeRTOSConfig.h 의 configUSE_TIMERS 및 configSUPPORT_STATIC_ALLOCATION 을 1 로 설정해야 한다.

1.7.1.2.1 매개 변수

pcTimerName : 순전히 디버깅을 돕기 위해 타이머에 할당되는 일반 텍스트 이름이다.

xTimerPeriod : 타이머의 기간이다. 타이머 기간은 tick 으로 지정된다. pdMS_TO_TICKS()매크로는 밀리 초 단위의 시간을 tick 단위의 시간으로 변환하는데 사용할 수 있다. 예를 들어 타이머가 100 tick 후에 만료되어야만 할 경우 xNewPeriod 를 직접 100 으로 설정할 수 있다. 또는, 타이머가 500ms 후에 만료해야 하면, configTICK_RATE_HZ 가 1000 이하일 경우 xNewPeriod 를 pdMS_TO_TICKS(500)으로 설정할 수 있다.

uxAutoReload : Autoreload 타이머를 생성하려면 pdTRUE 로 설정한다. One-Shot 타이머를 작성하려면 pdFALSE 로 설정한다. Autoreload 타이머는 xTimerPeriod 매개 변수에 의해 설정된 주기로 반복적으로 만료된다. One-Shot 타이머는 한번만 만료된다. 만료된 후 One-Shot 타이머는 수동으로 다시 시작할수 있다.

pvTimerID : 만들려는 타이머에 할당된 식별자다. 식별자는 나중에 vTimerSetTimerID()API 함수를 사용하여 업데이트 할 수 있다. 동일한 콜백 함수가 여러 타이머에 할당된 경우, 콜백 함수 내에서 타이머 식별자를 검사하여 실제로 만료된 타이머를 확인할 수 있다. 또한, 타이머 식별자는 타이머의 콜백 함수 호출 사이에 값을 저장하는데 사용할 수 있다.

pxCallbackFunction : 타이머가 만료될 때 호출할 함수다. 콜백 함수는 TimerCallbackFunction_t 유형의 프로토 타입을 갖는다. 필요한 프로토 타입은 다음과 같다.

```
void vCallbackFunctionExample (TimerHandle_t xTimer);
```

pxTimerBuffer : 타이머의 상태를 유지하는데 사용되는 StaticTimer_t 유형의 변수를 가리킨다.

1.7.1.2.2 반환 값

NULL : pxTimerBuffer 가 NULL 이라 소프트웨어 타이머를 만들수 없다.

다른 값 : 소프트웨어 타이머가 성공적으로 생성되었으며, 반환된 값은 생성된 소프트웨어 타이머를 참조할 수 있는 핸들이다.

1.7.1.2.3 기타

```
/* 여러 타이머 인스턴스가 사용할 콜백 함수를 정의한다. 콜백 함수는 관련 타이머가 만료된 횟수를 계산하고 아무것도 하지 않으며,
타이머가 10 회 만료되면 타이머를 중지한다. 카운트는 타이머의 ID 로 저장한다. */
```

```
void vTimerCallback( TimerHandle_t xTimer ) {
    const uint32_t ulMaxExpiryCountBeforeStopping = 10;
    uint32_t ulCount;

    // 이 타이머의 만료된 횟수가 타이머 ID 로 저장된다. 카운트를 얻는다.
    ulCount = ( uint32_t ) pvTimerGetTimerID( xTimer );

    // 카운트를 증가시키고 타이머가 만료되었는지 확인하기 위해 ulMaxExpiryCountBeforeStopping 을 테스트한다.
    ulCount++;

    // 타이머가 10 회 만료되면 실행을 중지한다.
    if( ulCount >= xMaxExpiryCountBeforeStopping ) {
      // 타이머 콜백 함수에서 타이머 API 함수를 호출하는 경우, 차단 시간을 사용하면 안된다.
      // 사용할 경우 교착 상태가 발생할 수 있다.
      xTimerStop( pxTimer, 0 );
    } else {
      // 타이머의 ID 필드에 증가된 카운트를 다시 저장하여 다음에 이 소프트웨어 타이머가 만료될때 읽도록 한다.
      vTimerSetTimerID( xTimer, ( void * ) ulCount );
    }
}
```

```
#define NUM_TIMERS 5

// 생성된 타이머에 대한 핸들을 저장하기 위한 배열이다.
TimerHandle_t xTimers[ NUM_TIMERS ];

void main( void ) {
    long x;

    /* 타이머를 만들고 시작한다. RTOS 스케줄러가 시작되기 전에 타이머를 시작할 경우,
    타이머는 RTOS 스케줄러가 시작되는 즉시 실행된다. */
    for( x = 0; x < NUM_TIMERS; x++ ) {
      /* tick 타이머 기간은 0 보다 커야한다, 타이머가 만료되면 자동으로 다시 로드된다,
      ID 는 타이머가 만료된 횟수를 저장하는데 사용되며 0 으로 초기화된다, 각 타이머는 만료되면 같은 콜백을 호출한다,
      생성되는 타이머와 관련된 데이터를 보유할 StaticTimer_t 변수의 주소를 전달한다. */
      xTimers[ x ] = xTimerCreateStatic( "Timer", ( 100 * x ) + 100, pdTRUE, ( void * ) 0,
                                         vTimerCallback, &( xTimerBuffers[ x ] ); );
      if( xTimers[ x ] == NULL ) {
        // 타이머 생성에 실패하였다.
      } else {
        // 타이머를 시작한다. 차단 시간은 지정되지 않았고, RTOS 스케줄러가 아직 시작되지 않았기 때문에 차단 시간이 무시된다.
        if( xTimerStart( xTimers[ x ], 0 ) != pdPASS ) {
          // 타이머를 활성화 상태로 설정할 수 없다.
        }
      }
    }
    // 여기서 작업을 만든다.
    // RTOS 스케줄러를 시작하면 타이머가 이미 활성화 상태로 설정되어 있으므로, 타이머가 작동한다.
    vTaskStartScheduler();

    // 여기에 도달하지 않는다.
    for( ;; );
}
```

1.7.1.3 xTimerDelete : 타이머 삭제

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerDelete ( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

타이머를 삭제한다. 타이머는 먼저 xTimerCreate()API 함수를 사용하여 만들어져야 한다.

xTimerDelete()를 사용하려면 FreeRTOSConfig.h 의 configUSE_TIMERS 를 1 로 설정해야 한다.

1.7.1.3.1 매개 변수

xTimer : 삭제하고자 하는 타이머의 핸들이다.

xTicksToWait : 타이머 기능은 핵심 FreeRTOS 코드에 의해 제공되는게 아니라 타이머 서비스 task(데몬)에 의해 제공된다. FreeRTOS 타이머 API 는 타이머 명령 대기열이라고 하는 대기열의 타이머 서비스 task 에 명령을 보낸다. xTicksToWait 는 대기열이 이미 꽉 찼으면 타이머 명령 대기열에서 공간을 사용할 수 있을때까지 task 가 차단 상태로 대기하는데, 그 최대 시간을 지정한다.

차단 시간은 tick 으로 지정되며 tick 시간은 tick 주파수에 따라 달라진다. pdMS_TO_TICKS()매크로는 밀리 초 단위로 지정된 시간을 tick 으로 변환하는데 사용할 수 있다.

xTicksToWait 을 portMAX_DELAY 로 설정하면, FreeRTOSConfig.h 의 INCLUDE_vTaskSuspend 가 1 일때, task 가 시간초과 없이 무기한 대기하게 된다. xTicksToWait 는 xTimerDelete()가 스케줄러 시작 전에 호출되면 무시한다.

1.7.1.3.2 반환 값

pdPASS : 삭제 명령이 타이머 명령 대기열에 성공적으로 전송되었다. 차단 시간이 지정된 경우(xTicksToWait 이 0 이 아닐 때), 함수를 반환하기 전에 호출 task 가 차단 상태로 설정되어 타이머 명령 대기열에서 공간을 사용할 수 있을때까지 대기할 수 있으며, 차단 시간이 만료되기 전에 대기열에 성공적으로 기록했을 때 반환된다. 명령이 실제로 처리될 때는 시스템의 다른 task 와 관련된 타이머 서비스 task 의 우선 순위에 따라 달라진다.

타이머 서비스 task 의 우선순위는 configTIMER_TASK_PRIORITY 구성 상수에 의해 설정된다.

pdFAIL : 대기열이 이미 가득차 있기 때문에, 삭제 명령이 타이머 명령 대기열로 보내지지 않았다. 차단 시간을 지정했을 경우(xTicksToWait 이 0 이 아닐 때), 타이머 서비스 task 가 대기열에 공간을 만들때까지 대기하지만, 지정된 차단 시간이 만료될 경우 반환한다.

1.7.1.3.3 기타

xTimerChangePeriod()API 함수의 예제를 참조한다.

1.7.2 타이머 기본 제어

1.7.2.1 xTimerStart : 타이머 시작

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerStart ( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

실행 상태의 타이머를 시작한다.

xTimerStartFromISR()은 인터럽트 서비스 루틴에서 호출할 수 있는 동일한 기능이다.

타이머가 아직 실행되고 있지 않으면 타이머는 xTimerStart()가 호출되었을 때를 기준으로 만료 시간을 계산한다. 타이머가 이미 실행중이면 xTimerStart()는 xTimerReset()와 동일한 기능을 한다. 타이머가 정지, 삭제 또는 재설정되지 않으면 xTimerReset()이 호출된 후 타이머와 연관된 콜백 함수가 ‘n’tick 으로 호출된다. 여기서 ‘n’은 타이머의 정의된 기간이다.

xTimerStart()을 사용하려면 FreeRTOSConfig.h 의 configResult_TIMERS 을 1 로 설정한다.

1.7.2.1.1 매개 변수

xTimer : 재설정, 시작 또는 재시작할 타이머의 핸들이다.

xTicksToWait : 타이머 기능은 FreeRTOS 코드에 의해 제공되는 것이 아니라, 타이머 서비스(데몬)task 에 의해 제공된다. FreeRTOS 타이머 API 는 타이머 명령 대기열에서 타이머 서비스 task 에 명령을 보낸다. xTicksToWait 는 대기열이 이미 꽉 찼으면 타이머 명령 대기열에서 공간을 사용할 수 있을때까지 대기할 task 가, 차단 상태로 유지되는 최대 시간이다. 차단 시간은 tick 으로 지정되며 tick 의 절대시간은 tick 주파수에 따라 달라진다.

pdMS_TO_TICKS()매크로는 밀리 초 단위로 지정된 시간을 tick 으로 변환하는데 사용할 수 있다.

FreeRTOSConfig.h 의 INCLUDE_vTaskSuspend 가 1 로 설정된 경우 xTicksToWait 을 portMAX_DELAY 할 경우 task 가 시간초과 없이 무기한 대기한다. xTicksToWait 는 xTimerStart()가 스케줄러에서 시작되기 전에 호출되면 무시한다.

1.7.2.1.2 반환 값

pdPASS : 시작 명령이 타이머 명령 대기열로 성공적으로 전송되었다. 차단 시간이 지정된 경우(xTicksToWait 가 0 이 아닐 때), 함수를 반환하기 전에 호출 task 가 차단 상태로 설정되어 타이머 명령 대기열에서 공간을 사용할 수 있을때까지 기다리며, 차단 시간이 만료되기 전에 데이터가 성공적으로 기록되었을때 반환한다.

명령이 처리되는 타이머의 만료 시간은 xTimerStart()가 실제로 호출되었을 때의 시간이지만, 시스템의 다른 task 의 우선순위에 따라 상대적인 타이머 서비스 task 의 처리 시간이 달라진다. 타이머 서비스 task 의 우선 순위는 configTIMER_TASK_PRIORITY 구성 상수에 의해 설정된다.

pdFAIL : 대기열이 이미 가득차서 시작 명령이 타이머 명령 대기열로 보내지지 않았다. 차단 시간이 지정된 경우(xTicksToWait 가 0 이 아닐 때), 타이머 서비스 task 가 대기열에 공간을 만들때까지 대기하지만, 지정된 차단 시간이 만료될때까지 처리가 되지 않을 경우 반환한다.

1.7.2.1.3 기타

xTimerCreate()API 함수에 제공된 예제를 참조한다.

1.7.2.2 xTimerStartFromISR : ISR 에서 타이머 시작

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerStartFromISR ( TimerHandle_t xTimer, BaseType_t * pxHigherPriorityTaskWoken );
```

인터럽트 서비스 루틴에서 호출할수 있는 xTimerStart()버전이다.

xTimerStartFromISR()을 사용하려면 FreeRTOSConfig.h 의 configUSE_TIMERS 를 1 로 설정해야 한다.

1.7.2.2.1 매개 변수

xTimer : 시작, 재설정, 재시작 중인 타이머의 핸들이다.

pxHigherPriorityTaskWoken : xTimerStartFromISR()은 타이머 명령 대기열에 명령을 기록한다. 타이머 명령 대기열에 기록하면 타이머 서비스 task 가 차단 상태에서 해제되고, 타이머 서비스 task 의 우선 순위가 현재 실행중인 task(인터럽트 된 task)보다 크거나 같으면 xTimerStartFromISR()함수 내부적으로 *pxHigherPriorityTaskWoken 를 pdTRUE 로 변경한다. xTimerStartFromISR()에서 이 값을 pdTRUE 로 설정하면 인터럽트가 종료되기 전에 컨텍스트 스위치를 수행해야 한다.

1.7.2.2.2 반환 값

pdPASS : 시작 명령이 타이머 명령 대기열로 성종적으로 전송되었다. 명령이 처리되는 타이머의 만료 시간은 xTimerStartFromISR()이 실제로 호출되었을 때의 시간이지만, 시스템의 다른 task 의 우선순위에 따라 상대적인 타이머 서비스 task 의 처리 시간이 달라진다. 타이머 서비스 task 의 우선 순위는 configTIMER_TASK_PRIORITY 구성 상수에 의해 설정된다.

pdFAIL : 대기열이 이미 가득차서 시작 명령이 타이머 명령 대기열로 보내지지 않는다.

1.7.2.2.3 기타

```
/* 이 예제는 xBacklightTimer 가 이미 생성되었다고 가정한다. 키를 누르면 LCD 백라이트가 켜진다. 키를 누르지 않고 5 초가 지나면
One-Shot 타이머로 LCD 백라이트가 꺼진다. xTimerReset()함수에 대해 제공된 예제와 달리 키 누르기 이벤트 핸들러는 인터럽트
서비스 루틴이다. */

// One-Shot 타이머에 할당된 콜백 함수다. 이 경우 매개 변수가 사용되지 않는다.
void vBacklightTimerCallback( TimerHandle_t pxTimer ) {
    // 타이머가 만료되었으므로, 키를 누른 후 5 초가 지나야 한다. LCD 백라이트를 끈다.
    vSetBacklightState( BACKLIGHT_OFF );
}

// 키 입력 인터럽트 서비스 루틴이다.
void vKeyPressEventInterruptHandler( void ) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* LCD 백라이트가 켜져있는지 확인 후, 5 초동안 키가 작동하지 않으면 백라이트를 끄는 타이머가 다시 시작한다.
    이것은 인터럽트 서비스 루틴이므로 “FromISR”로 끝나는 FreeRTOS API 만 호출할 수 있다. */
    vSetBacklightState( BACKLIGHT_ON );

    /* xTimerStartFromISR() 또는 xTimerResetFromISR()둘 다 타이머가 만료 시간을 다시 계산하도록 하기 때문에 여기에서
    호출할 수 있다. xHigherPriorityTaskWoken 은 선언되었을 때 pdFALSE 로 초기화 한다. */
    if( xTimerStartFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS ) {
      // 시작 명령이 성공적으로 실행되지 않았다. 여기서 조치를 취한다.
    }

    // 여기서 나머지 키 처리를 수행한다.
```

```
    /* xHigherPriorityTaskWoken 이 pdTRUE 와 같으면 컨텍스트 전환이 수행되어야합니다.
    ISR 내부에서 컨텍스트 스위치를 수행하는 데 필요한 구문은 포트마다 다르기 때문에, 사용중인 포트에 대한 데모를
    확인하여 필요한 실제 구문을 찾아서 사용한다. * /
    if( xHigherPriorityTaskWoken != pdFALSE ) {
      // 여기서 인터럽트 안전 출력 함수를 호출한다. (실제 함수는 사용중인 FreeRTOS 포트에 따라 다르다.)
    }
}
```

1.7.2.3 xTimerStop : 타이머 정지

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerStop ( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

타이머 실행을 중지한다.

xTimerStopFromISR()은 인터럽트 서비스 루틴에서 호출할 수 있는 동일한 기능이다.

xTimerStop()을 사용하려면 configRootConfig.h 의 configUSE_TIMERS 를 1 로 설정해야 한다.

1.7.2.3.1 매개 변수

xTimer : 정지할 타이머의 핸들이다.

xTicksToWait : 타이머 기능은 FreeRTOS 코드에 의해 제공되는 것이 아니라, 타이머 서비스(데몬)task 에 의해 제공된다. FreeRTOS 타이머 API 는 타이머 명령 대기열에서 타이머 서비스 task 에 명령을 보낸다. xTicksToWait 는 대기열이 이미 꽉 찼으면 타이머 명령 대기열에서 공간을 사용할 수 있을때까지 대기할 task 가, 차단 상태로 유지되는 최대 시간이다. 차단 시간은 tick 으로 지정되며 tick 의 절대시간은 tick 주파수에 따라 달라진다.

pdMS_TO_TICKS()매크로는 밀리 초 단위로 지정된 시간을 tick 으로 변환하는데 사용할 수 있다.

FreeRTOSConfig.h 의 INCLUDE_vTaskSuspend 가 1 로 설정된 경우 xTicksToWait 을 portMAX_DELAY 할 경우 task 가 시간초과 없이 무기한 대기한다. xTicksToWait 는 xTimerStop()이 스케줄러에서 시작되기 전에 호출되면 무시한다.

1.7.2.3.2 반환 값

pdPASS : 중지 명령이 타이머 명령 대기열로 성공적으로 전송되었다. 차단 시간이 지정된 경우(xTicksToWait 가 0 이 아닐 때), 함수를 반환하기 전에 호출 task 가 차단 상태로 설정되어 타이머 명령 대기열에서 공간을 사용할 수 있을때까지 기다리며, 차단 시간이 만료되기 전에 데이터가 성공적으로 기록되었을때 반환한다.

명령이 실제로 처리될 때 시스템의 다른 task 와 타이머 서비스 task 의 우선 순위에 따라 달라진다. 타이머 서비스 task 의 우선 순위는 configTIMER_TASK_PRIORITY 구성 상수에 의해 설정된다.

pdFAIL : 대기열이 이미 가득차서 중지 명령이 타이머 명령 대기열로 보내지지 않았다. 차단 시간이 지정된 경우(xTicksToWait 가 0 이 아닐 때), 타이머 서비스 task 가 대기열에 공간을 만들때까지 대기하지만, 지정된 차단 시간이 만료될때까지 처리가 되지 않을 경우 반환한다.

1.7.2.3.3 기타

xTimerCreate() API 함수에 제공된 예제를 참조한다.

1.7.2.4 xTimerStopISR : ISR 에서 타이머 정지

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerStopFromISR ( TimerHandle_t xTimer, BaseType_t * pxHigherPriorityTaskWoken );
```

인터럽트 서비스 루틴에서 호출할 수 있는 xTimerStop()버전이다.

xTimerStopFromISR()을 사용하려면 FreeRTOSConfig.h 의 configUSE_TIMERS 를 1 로 설정해야 한다.

1.7.2.4.1 매개 변수

xTimer : 정지할 타이머의 핸들이다.

pxHigherPriority : xTimerStopFromISR()은 타이머 명령 대기열에 명령을 기록한다. 타이머 명령 대기열에 기록하면 타이머 서비스 task 가 차단 상태에서 해제되고, 타이머 서비스 task 의 우선 순위가 현재 실행중인 task(인터럽트 된 task)보다 크거나 같으면 xTimerStopFromISR()함수 내부적으로 *pxHigherPriorityTaskWoken 를 pdTRUE 로 변경한다. xTimerStopFromISR()에서 이 값을 pdTRUE 로 설정하면 인터럽트가 종료되기 전에 컨텍스트 스위치를 수행해야 한다.

1.7.2.4.2 반환 값

pdPASS : 중지 명령이 타이머 명령 대기열로 성종적으로 전송되었다.

명령이 실제로 처리될 때 시스템의 다른 task 와 타이머 서비스 task 의 우선 순위에 따라 달라진다. 타이머 서비스 task 의 우선 순위는 configTIMER_TASK_PRIORITY 구성 상수에 의해 설정된다.

pdFAIL : 대기열이 이미 가득차서 중지 명령이 타이머 명령 대기열로 보내지지 않았다.

1.7.2.4.3 기타

```
// 이 예제는 xTimer 가 이미 생성되어 시작되었다고 가정한다. 인터럽트가 발생하면 타이머를 정지한다.

// 타이머를 멈추게하는 인터럽트 서비스 루틴이다.
void vAnExampleInterruptServiceRoutine( void ) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // 인터럽트가 발생했다. 타이머를 중지한다. xHigherPriorityTaskWoken 은 pdFALSE 로 설정되어 있다.
    // 인터럽트 서비스 루틴이므로 “FromISR”로 끝나는 FreeRTOS API 함수만 사용할 수 있다.
    if( xTimerStopFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS ) {
      // 중지 명령이 성공적으로 실행되지 않았다. 여기에 적절한 조치를 취한다.
    }

    /* xHigherPriorityTaskWoken 이 pdTRUE 와 같으면 컨텍스트 전환이 수행되어야합니다.
    ISR 내부에서 컨텍스트 스위치를 수행하는 데 필요한 구문은 포트마다 다르기 때문에, 사용중인 포트에 대한 데모를
    확인하여 필요한 실제 구문을 찾아서 사용한다. * /
    if( xHigherPriorityTaskWoken != pdFALSE ) {
      // 여기서 인터럽트 안전 출력 함수를 호출한다. (실제 함수는 사용중인 FreeRTOS 포트에 따라 다르다.)
    }
}
```

1.7.2.5 xTimerReset : 타이머 리셋

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerReset ( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

타이머를 다시 시작한다. xTimerResetFromISR()은 인터럽트 서비스 루틴에서 호출할 수 있는 동일한 기능이다.

타이머가 이미 실행중이면 타이머는 xTimerReset()가 호출되었을 때 상대적인 만료 시간을 다시 계산한다. 타이머가 실행되고 있지 않으면 타이머는 xTimerReset()이 호출되었을 때를 기준으로 만료 시간을 계산하고, 타이머가 실행된다. 이 경우 xTimerReset()은 xTimerStart()와 기능적으로 동일하다. 타이머를 재설정하면 타이머가 실행된다. 그동안 타이머가 정지, 삭제, 재설정되지 않으면 xTimerReset()이 호출된 후 타이머와 연관된 콜백 함수가 ‘n’tick 으로 호출된다. 여기서 ‘n’tick 은 타이머의 정의된 기간이다. 스케줄러가 시작되기 전에 xTimerReset()가 호출되면 스케줄러가 시작될때까지 타이머가 실행되지 않고 타이머의 만료 시간은 스케줄러가 시작된 시간에 비례한다.

xTimerReset()을 사용하려면 FreeRTOSConfig.h 의 configUSE_TIMERS 을 1 로 설정해야 한다.

1.7.2.5.1 매개 변수

xTimer : 재설정, 시작 또는 재시작할 타이머의 핸들이다.

xTicksToWait : 타이머 기능은 핵심 FreeRTOS 코드에 의해 제공되는게 아니라 타이머 서비스 task(데몬)에 의해 제공된다. FreeRTOS 타이머 API 는 타이머 명령 대기열이라고 하는 대기열의 타이머 서비스 task 에 명령을 보낸다. xTicksToWait 는 대기열이 이미 꽉 찼으면 타이머 명령 대기열에서 공간을 사용할 수 있을때까지 task 가 차단 상태로 대기하는데, 그 최대 시간을 지정한다.

차단 시간은 tick 으로 지정되며 tick 시간은 tick 주파수에 따라 달라진다. pdMS_TO_TICKS()매크로는 밀리 초 단위로 지정된 시간을 tick 으로 변환하는데 사용할 수 있다.

xTicksToWait 을 portMAX_DELAY 로 설정하면, FreeRTOSConfig.h 의 INCLUDE_vTaskSuspend 가 1 일때, task 가 시간초과 없이 무기한 대기하게 된다. xTicksToWait 는 xTimerDelete()가 스케줄러 시작 전에 호출되면 무시한다.

1.7.2.5.2 반환 값

pdPASS : 재설정 명령이 타이머 명령 대기열에 성공적으로 전송되었다. 차단 시간이 지정된 경우(xTicksToWait 이 0 이 아닐 때), 함수를 반환하기 전에 호출 task 가 차단 상태로 설정되어 타이머 명령 대기열에서 공간을 사용할 수 있을때까지 대기할 수 있으며, 차단 시간이 만료되기 전에 대기열에 성공적으로 기록했을 때 반환된다. 명령이 실제로 처리될 때는 시스템의 다른 task 와 관련된 타이머 서비스 task 의 우선 순위에 따라 달라진다.

타이머 서비스 task 의 우선순위는 configTIMER_TASK_PRIORITY 구성 상수에 의해 설정된다.

pdFAIL : 대기열이 이미 가득차 있기 때문에, 삭제 명령이 타이머 명령 대기열로 보내지지 않았다. 차단 시간을 지정했을 경우(xTicksToWait 이 0 이 아닐 때), 타이머 서비스 task 가 대기열에 공간을 만들때까지 대기하지만, 지정된 차단 시간이 만료될 경우 반환한다.

1.7.2.5.3 기타

```
// 이 예에서 키를 누르면 LCD 백라이트가 켜진다. 키를 누르지 않고 5 초가 지나면 One-Shot 타이머로 LCD 백라이트가 꺼진다.
TimerHandle_t xBacklightTimer = NULL;

// One-Shot 타이머에 할당된 콜백 함수다. 이 경우 매개 변수가 사용되지 않았다.
void vBacklightTimerCallback( TimerHandle_t pxTimer ) {
    // 타이머가 만료되었으므로 키를 누른 후 5 초가 지났다. LCD 백라이트를 끈다.
    vSetBacklightState( BACKLIGHT_OFF );
}
```

```
// 키 누름을 관리하는 이벤트 핸들러다.
void vKeyPressEventHandler( char cKey ) {
    /* LCD 백라이트가 켜져있는지 확인하고, 5 초동안 키가 작동하지 않으면 백라이트를 끈다.
    즉시 전송할 수 없는 경우 reset 명령이 성공적으로 전송되도록 10 tick 씩 대기한다. */
    vSetBacklightState( BACKLIGHT_ON );

    if( xTimerReset( xBacklightTimer, 10 ) != pdPASS ) {
      // reset 명령이 성공적으로 실행되지 않았다. 여기서 적절한 조치를 취한다.
    }

    // 여기서 나머지 키 처리를 수행한다.
}

void main( void ) {
    // 5 초 이내에 아무 키도 누르지 않으면 백라이트를 끈다.
    // 타이머 기간(tick 단위), One-shot 타이머, 콜백에 사용되지 않은 ID, LCD 백라이트 종료 콜백 함수
    xBacklightTimer = xTimerCreate( "BcklghtTmr", pdMS_TO_TICKS( 5000 ), pdFALSE, 0, vBacklightTimerCallback );

    if( xBacklightTimer == NULL ) {
      // 타이머가 생성되지 않았다.
    } else {
      // 타이머를 시작한다. 차단 시간이 지정되지 않았고 스케줄러가 아직 시작되지 않았으므로 무시한다.
      if( xTimerStart( xBacklightTimer, 0 ) != pdPASS ) {
        // 타이머를 활성화 상태로 설정할 수 없다.
      }
    }
    // 여기서 task 를 만든다.
    // xTimerStart 가 이미 호출되었기 때문에 스케줄러를 시작하면 타이머가 시작된다.
    xTaskStartScheduler();
}
```

1.7.2.6 xTimerResetFromISR : ISR 에서 타이머 리셋

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerResetFromISR ( TimerHandle_t xTimer, BaseType_t * pxHigherPriorityTaskWoken );
```

인터럽트 서비스 루틴에서 호출할 수 있는 xTimerReset()의 버전이다.

xTimerResetFromISR()을 사용하려면 FreeRTOSConfig.h 의 configUSE_TIMERS 를 1 로 설정해야 한다.

1.7.2.6.1 매개 변수

xTimer : 재설정, 시작 또는 재시작할 타이머의 핸들이다.

pxHigherPriorityTaskWoken : xTimerResetFromISR()을 호출하면 대기열에서, RTOS 타이머 데몬 task 로 보내는 메시지다. FreeRTOSConfig.h 의 configTIMER_TASK_PRIORITY 값으로 설정된 데몬 task 의 우선 순위가 현재 실행중인 task(인터럽트 된 task)의 우선 순위보다 높으면 xTimerResetFromISR()에서 *pxHigherPriorityTaskWoken 가 pdTRUE 로 설정된다. 인터럽트가 종료되기 전에 컨텍스트 스위치가 요청되야 하는 것을 나타낸다.

1.7.2.6.2 반환 값

pdPASS : 재설정 명령이 타이머 명령 대기열에 성공적으로 전송되었다. 명령이 실제로 처리될 때 타이머의 만료 시간은 xTimerResetFromISR()이 실제로 호출되는 시간과 관련이 있지만, 시스템의 다른 task 의 우선 순위에 따라 달라진다. 타이머 서비스 task 의 우선 순위는 configTIMER_TASK_PRIORITY 구성 상수에 의해 설정된다.

pdFAIL : 대기열이 이미 가득차 있기 때문에, 삭제 명령이 타이머 명령 대기열로 보내지지 않았다.

1.7.2.6.3 기타

```
/* 이 예제는 xBacklightTimer 가 이미 생성되어 있다고 가정한다. 키를 누르면 LCD 백라이트가 켜진다. 키를 누르지 않고 5 초가
지나면 One-shot 타이머로 LCD 백라이트가 꺼진다. xTimerReset()함수에 대해 제공된 예제와 달리 키누르기 이벤트 핸들러는 인터럽트
서비스 루틴이다. */
void vBacklightTimerCallback( TimerHandle_t pxTimer ) {
    // 키를 누른 후 5 초가 지나 타이머가 만료되었으므로, LCD 백라이트를 끈다.
    vSetBacklightState( BACKLIGHT_OFF );
}

// 키 누름 인터럽트 서비스 루틴이다.
void vKeyPressEventInterruptHandler( void ) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* LCD 백라이트가 켜져 있는지 확인한 후 5 초동안 키가 작동하지 않으면 백라이트를 끈다. 이것은 인터럽트 서비스 루틴이므로
    “FromISR”로 끝나는 FreeRTOS 함수만 호출할 수 있다. */
    vSetBacklightState( BACKLIGHT_ON );

    /* xTimerStartFromISR () 또는 xTimerResetFromISR () 둘 다 타이머가 만료 시간을 다시 계산하도록하기 때문에 여기에서
    호출할수 있다. xHigherPriorityTaskWoken 은 선언되었을때 pdFALSE 로 초기화되있다 */
    if( xTimerResetFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS ) {
      // reset 명령이 성공적으로 실행되지 않았다. 여기서 적절한 조치를 취한다.
    }

    // 여기서 나머지 키 처리를 수행한다.
    /* xHigherPriorityTaskWoken 가 pdTRUE 와 같으면 컨텍스트 스위치를 수행해야 한다. ISR 내부에서 컨텍스트 스위치를 수행하는
    구문은 포트마다 다르며 사용중인 포트에 대한 데모를 확인하여 필요한 실제 구문을 찾아야 한다. */
    if( xHigherPriorityTaskWoken != pdFALSE ) {
      // 여기서 인터럽트 안전 출력 함수를 호출한다. (실제 함수는 사용중인 FreeRTOS 포트에 따라 다르다.)
    }
}
```

1.7.2.7 xTimerChangePeriod : 타이머 기간 변경

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerChangePeriod ( TimerHandle_t xTimer, TickType_t xNewPeriod, TickType_t xTicksToWait );
```

타이머 기간을 변경한다. 인터럽트 서비스 루틴(ISR)에서 사용하고자 한다면 xTimerChangePeriodFormISR()을 사용한다.

xTimerChangePeriod()을 사용하여 이미 실행중인 타이머시 기간을 변경하면, 타이머는 새로운 기간 값을 사용하여 만료 기간을 다시 계산한다. 다시 계산된 만료 기간은 xTimerChangePeriod()가 호출된 때를 기준으로 하여, 타이머가 원래 시작되었을 때와 관련이 없다. xTimerChangePeriod()가 실행되고 있지 않은 타이머의 기간을 변경하는 경우, 타이머는 새 기간 값을 사용하여 만료 기간을 계산하고 타이머가 실행한다.

xTimerChangePeriod()을 사용하려면 configRes0_Config.h 의 configUSE_TIMERS 를 1 로 설정한다.

1.7.2.7.1 매개 변수

xTimer : 새로운 기간을 할당할 타이머 핸들이다.

xNewPeriod : xTimer 매개 변수에서 참조하는 타이머의 새 기간이다. 타이머 기간은 tick 으로 지정된다. pdMS_TO_TICKS()매크로는 밀리 초 단위의 시간을 tick 시간으로 변환하는데 사용할 수 있다. 예를들어 타이머가 100tick 후에 만료되어야 한다면 xNewPeriod 를 직접 100 으로 설정할 수 있다. 타이머가 500ms 후에 만료되어야 하면 configTICK_RATE_HZ 가 1000 이하인 경우 xNewPeriod 를 pdMS_TO_TICKS (500)로 설정할수 있다.

xTicksToWait : 타이머 기능은 핵심 FreeRTOS 코드에 의해 제공되는 것이 아니라 타이머 서비스(데몬)task 에 의해 제공된다. FreeRTOS 타이머 API 는 타이머 명령 대기열의 타이머 서비스 task 에서 명령을 보낸다. xTicksToWait 는 대기열이 꽉 찼으면 타이머 명령 대기열에서 공간을 사용할수 있을때까지 대기할 task 가 차단 상태로 유지되어야 하는데 최대 시간이다. 차단 시간은 tick 으로 지정되므로 tick 시간의 절대 시간은 tick 주파수에 따라 달라진다. xNewPeriod 매개 변수와 마찬가지로, pdMS_TO_TICKS()매크로는 밀리 초 단위로 지정된 시간을 tick 으로 변환하는데 사용할 수 있다.

FreeRTOSConfig.h 에서 INCLUDE_vTaskSuspend 가 1 일 때, xTicksToWait 을 portMAX_DELAY 로 설정하면 task 가 시간 초과 없이 무기한 대기하게 된다.

xTimerChangePeriod()는 스케줄러가 시작되기 전에 호출되면 xTicksToWait 가 무시된다.

1.7.2.7.2 반환 값

pdPASS : 기간 변경 명령이 타이머 명령 대기열에 성공적으로 전송되었다. 차단 시간이 지정된 경우(xTicksToWait 이 0 이 아닐 경우), 함수를 반환하기 전 호출 task 가 차단 상태로 설정되어 타이머 명령 대기열에서 공간을 사용할 수 있을 때까지 호출 task 가 차단 상태로 전환될 수 있다. 명령이 실제로 처리될 때, 타이머의 만료 기간은 xTimerChangePeriod()가 실제 호출될 때의 상태임에도 불구하고, 시스템의 다른 task 와 관련하여 타이머 서비스 task 의 우선 순위에 따라 달라진다.

타이머 서비스 task 의 우선 순위는 configTIMER_TASK_PRIORITY 구성 상수에 의해 설정된다.

pdFAIL : 대기열이 이미 가득차서 기간 변경 명령이 타이머 대기열로 전송되지 않았다. 차단 시간을 지정하면(xTicksToWait 이 0 이 아닐 경우), 타이머 서비스 task 가 대기열에 공간을 만들 때까지 대기하지만, 지정된 차단 시간이 만료할 경우 반환한다.

1.7.2.7.3 기타

```
/* 이 함수는 xTimer 가 이미 생성되어 있다고 가정한다. xTimer 에 의해 참조된 타이머가 호출될 때 이미 활성화되어 있으면 타이머를
삭제한다. xTimer 가 참조하는 타이머가 호출될 때 활성화되지 않으면 타이머의 주기를 500ms 로 설정하고 타이머가 시작된다. */

void vAFunction( TimerHandle_t xTimer ) {
    if( xTimerIsTimerActive( xTimer ) != pdFALSE ) {
      // xTimer 가 이미 활성화 되어 있다. 삭제한다.
      xTimerDelete( xTimer );
    } else {
      /* xTimer 가 활성화 되어 있지 않다. 주기를 500ms 로 변경한다. 또한, 타이머가 시작된다.
      기간 변경 명령 즉시, 타이머 명령 대기열로 보낼수 없으면 최대 100tick 동안 대기한다. */
      if( xTimerChangePeriod( xTimer, pdMS_TO_TICKS( 500 ), 100 ) == pdPASS ) {
        // 명령이 성공적으로 전송되었다.
      } else {
        // 100 tick 을 기다린 후에도 명령을 보낼수 없다. 여기서 적절한 조치를 취한다.
      }
    }
}
```

1.7.2.8 xTimerChangePeriodFromISR : ISR 에서 타이머 기간 변경

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerChangePeriodFromISR ( TimerHandle_t xTimer, TickType_t xNewPeriod,
                                       BaseType_t * pxHigherPriorityTaskWoken );
```

인터럽트 서비스 루틴(ISR)에서 호출할 수 있는 xTimerChangePeriod()의 버전이다.

xTimerChangePeriodFromISR()을 사용하려면 configRootConfig.h 의 configUSE_TIMERS 를 1 로 설정해야 한다.

1.7.2.8.1 매개 변수

xTimer : 새로운 기간을 할당할 타이머다.

xNewPeriod : xTimer 매개 변수에서 참조하는 타이머의 새 기간이다. 타이머 기간은 tick 으로 지정된다. pdMS_TO_TICKS()매크로는 밀리초 단위의 시간을 tick 시간으로 변환하는데 사용할 수 있다. 예를들어 타이머가 100tick 후에 만료되어야 한다면 xNewPeriod 를 직접 100 으로 설정할 수 있다. 타이머가 500ms 후에 만료되어야 하면 configTICK_RATE_HZ 가 1000 이하인 경우 xNewPeriod 를 pdMS_TO_TICKS (500)로 설정할수 있다.

pxHigherPriorityTaskWoken : xTimerChangePeriodFromISR()은 타이머 명령 대기열에 명령을 기록한다. 타이머 명령 대기열에 기록하면 타이머 서비스 task 의 우선순위가 현재 실행중인 task(인터럽트 된 task)보다 크거나 같으면 xTimerChangePeriodFromISR() 함수 내부적으로 *pxHigherPriorityTaskWoken 가 pdTRUE 가 된다. xTimerChangePeriodFromISR()이 값을 pdTRUE 로 설정되면 인터럽트가 종료되기 전에 컨텍스트 스위치를 수행해야한다.

1.7.2.8.2 반환 값

pdPASS : 기간 변경 명령이 타이머 명령 대기열로 전송되었다. 명령이 실제로 처리될 때 타이머의 만료 기간은 xTimerChangePeriodFromISR()이 실제 호출될 때와 관련이 있지만, 시스템의 다른 task 에 상대적인 타이머 서비스 task 우선 순위에 따라 달라진다. 타이머 서비스 task 의 우선순위는 configTIMER_TASK_PRIORITY 구성 상수에 의해 설정된다.

pdFAIL : 대기열이 이미 가득차서 기간 변경 명령이 타이머 명령 대기열로 전송되지 않는다.

1.7.2.8.3 기타

```
/* 이 예제는 xTimer 가 이미 생성되어 시작되고 있다 가정한다. 인터럽트가 발생하면 xTimer 주기를 500ms 로 변경한다.
xTimer 의 주기를 변경하는 인터럽트 서비스 루틴이다. */
void vAnExampleInterruptServiceRoutine( void ) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* 인터럽트가 발생했다. xTimer 의 주기를 500ms 로 변경한다. xHigherPriorityTaskWoken 은 pdFALSE 로 설정되어 있다.
    이것은 인터럽트 서비스 루틴이므로 “FromISR”로 끝나는 FreeRTOS API 함수만 사용할 수 있다. */
    if( xTimerChangePeriodFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS ) {
      // 타이머 기간을 변경하는 명령이 성공적으로 실행되지 않았다. 여기서 적절한 조치를 취한다.
    }
    /* xHigherPriorityTaskWoken 가 pdTRUE 와 같으면 컨텍스트 스위치가 수행되야 한다. ISR 내부에서 컨텍스트 스위치를
    수행하는데 필요한 구문은 포트마다 다르며, 사용중인 포트에 대한 뎀모를 검색하여 필요한 실제 구문을 찾아야 한다. */
    if( xHigherPriorityTaskWoken != pdFALSE ) {
      // 여기서 인터럽트 안정성 출력 함수를 호출한다. (실제 함수는 사용중인 FreeRTOS 포트에 따라 다르다.)
    }
}
```

1.7.3 타이머 출력 및 고급 기능

1.7.3.1 xTimerIsTimerActive : 타이머 실행 여부 확인

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerIsTimerActive ( TimerHandle_t xTimer );
```

타이머가 실행중인지 확인한다. 다음과 같은 경우 타이머가 실행되지 않는다.

1. 타이머가 생성되었지만 시작되지 않았다.

2. 타이머는 만료된 이후 다시 시작되지 않은 One-Shot 타이머다.

xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod(), xTimerChangePeriodFromISR() API 함수를 사용하여 타이머를 실행할 수 있다.

xTimerIsTimerActive()를 사용하려면 FreeRTOSConfig.h 의 configUSE_TIMERS 를 1 로 설정해야 한다.

1.7.3.1.1 매개 변수

xTimer : 확인하고자 하는 타이머의 핸들이다.

1.7.3.1.2 반환 값

pdFALSE : 타이머가 실행되고 있지 않는다.

다른 값 : 타이머가 실행중이다.

1.7.3.1.3 기타

```
// 이 함수는 xTimer 가 이미 생성되어 있다고 가정한다.
void vAFunction( TimerHandle_t xTimer ) {
    // 다음 줄은 “ if( xTimerIsTimerActive( xTimer ) ) “ 로 대체할수 있다.
    if( xTimerIsTimerActive( xTimer ) != pdFALSE ) {
      // xTimer 가 활성화 되어 있다.
    } else {
      // xTimer 가 활성화 되어 있지 않는다. 다른것을 사용한다.
    }
}
```

1.7.3.2 xTimerGetPeriod : 타이머 기간 출력

```
#include “FreeRTOS.h”
#include “timers.h”

TickType_t xTimerGetPeriod ( TimerHandle_t xTimer );
```

소프트웨어 타이머의 기간을 반환한다. 기간은 RTOS tick 단위로 지정된다. 소프트웨어 타이머의 기간은 처음 타이머를 생성하는데 사용된 xTimerCreate()호출의 xTimerPeriod 매개 변수에 의해 지정된다. 나중에 xTimerChangePeriod() 및 xTimerChangePeriodFromISR()API 함수를 사용하여 변경할 수 있다.

1.7.3.2.1 매개 변수

xTimer : 확인하고자 하는 타이머의 핸들이다.

1.7.3.2.2 반환 값

tick 단위로 지정된 타이머의 기간이다.

1.7.3.2.3 기타

```
// 소프트웨어 타이머에 할당된 콜백 함수다.
static void prvTimerCallback( TimerHandle_t xTimer ) {
    TickType_t xTimerPeriod;

    // 만료된 타이머의 기간을 확인한다.
    xTimerPeriod = xTimerGetPeriod( xTimer );
}
```

1.7.3.3 xTimerGetExpiryTime : 타이머 만료 시간 출력

```
#include “FreeRTOS.h”
#include “timers.h”

TickType_t xTimerGetExpiryTime ( TimerHandle_t xTimer );
```

소프트웨어 타이머의 콜백 기능이 실행되는 시간인, 소프트웨어 타이머가 만료되는 시간을 반환한다.

xTimerGetExpiryTime()에 의해 반환된 값이 현재 tick 수보다 작으면 tick 수가 오버플로우 되고 다시 0 으로 wrapped 될 때까지 타이머가 만료되지 않는다. 오버 플로우는 RTOS 구현 자체에서 처리되기 때문에 타이머의 콜백 함수는 tick 수가 오버플로우 되기 전후에 상관없이 정확한 시간에 실행된다.

xTimerGetExpiryTime()을 사용하려면 FreeRTOSConfig.h 의 configRetry_TIMERS 를 1 로 설정해야한다.

1.7.3.3.1 매개 변수

xTimer : 확인하고자 하는 타이머의 핸들이다.

1.7.3.3.2 반환 값

xTimer 가 참조하는 타이머가 활성화되어 있다면, 타이머의 콜백 함수가 실행될 시간이 반환된다. 시간은 RTOS tick 으로 지정된다.

xTimer 가 참조하는 타이머가 활성화되어 있지 않으면, 반환값은 정의되지 않는다. xTimerIsTimerActive()API 함수로 타이머가 활성화 상태인지 확인할 수 있다.

1.7.3.3.3 기타

```
static void vAFunction( TimerHandle_t xTimer ) {
    TickType_t xRemainingTime;

    /* xTimer 가 참조하는 타이머가 만료되기 전의 시간을 계산하고 콜백 함수를 실행한다. TickType_t 는 부호없는 유형이므로
    tick 수가 오버플로우 될때까지 타이머가 만료되지 않더라도 감산 결과가 올바른 응답을 한다. */
    xRemainingTime = xTimerGetExpiryTime( xTimer ) - xTaskGetTickCount();
}
```

1.7.3.4 pcTimerGetName : 타이머 텍스트 이름 출력

```
#include “FreeRTOS.h”
#include “timers.h”

const char * pcTimerGetName ( TimerHandle_t xTimer );
```

타이머 작성시, 인간이 읽을수 있는 텍스트 명을 반환한다. 자세한 내용은 xTimerCreate()를 참조한다.

pcTimerGetName()를 사용하려면 FreeRTOSConfig.h 의 configUSE_TIMERS 를 1 로 설정해야 한다.

1.7.3.4.1 매개 변수

xTimer : 확인하고자 하는 타이머다.

1.7.3.4.2 반환 값

타이머의 이름은 표준 NULL 종료 C 문자열이다. 반환된 값은 대상 타이머의 이름을 가리키는 포인터다.

1.7.3.5 vTimerSetTimerID : 타이머 식별자 설정

```
#include “FreeRTOS.h”
#include “timers.h”

void vTimerSetTimerID ( TimerHandle_t xTimer, void * pvNewID );
```

식별자(ID)는 타이머가 생성될 때 타이머에 할당되며, vTimerSetTimerID()API 함수를 사용하여 언제든지 변경할 수 있다.

동일한 콜백 함수가 여러 타이머에 할당된 경우 콜백 함수내에서 타이머 식별자를 검사하여 실제로 만료된 타이머를 확인할 수 있다.
타이머 식별자는 타이머의 콜백 함수 호출 시, 타이머에 데이터를 저장하는데 사용할수도 있다.

xTimerSetTimerID()를 사용하려면 FreeRTOSConfig.h 의 configUSE_TIMERS 를 1 로 설정해야 한다.

1.7.3.5.1 매개 변수

xTimer : 새로운 식별자로 업데이트 할 타이머의 핸들이다.

pvNewID : 타이머의 식별자가 설정될 값이다.

1.7.3.5.2 기타

```
// 타이머에 할당된 콜백 함수다.
void TimerCallbackFunction( TimerHandle_t pxExpiredTimer ) {
    uint32_t ulCallCount;

    // 콜백 함수는 이 타이머가 만료되어 실행된 횟수를 ID 에 저장한다. 카운트를 가져와 증가시키고, 타이머 ID 에 다시 저장한다.
    ulCallCount = ( uint32_t ) pvTimerGetTimerID( pxExpiredTimer );
    ulCallCount++;

    vTimerSetTimerID( pxExpiredTimer, ( void * ) ulCallCount );
}
```

1.7.3.6 pvTimerGetTimerID : 타이머 식별자 출력

```
#include “FreeRTOS.h”
#include “timers.h”

void * pvTimerGetTimerID ( TimerHandle_t xTimer );
```

타이머에 할당된 식별자(ID)를 반환한다.

식별자는 타이머가 생성될 때 타이머에 할당되며 vTimerSetTimerID()API 함수를 사용하여 업데이트 할 수 있다. 자세한 내용은 xTimerCreate()함수를 참조한다.

동일한 콜백 함수가 여러 타이머에 할당된 경우 콜백 함수 내에서 타이머 식별자를 검사하여 실제로 만료된 타이머를 확인할 수 있다. 이는 xTimerCreate()API 함수에 제공된 예제 코드에 설명되어 있다. 또한 타이머 식별자는 타이머의 콜백 함수 호출 사이에 값을 저장하는데 사용할 수 있다.

pvTimerGetTimerID()를 사용하려면 FreeRTOSConfig.h 의 configuse_TIMERS 가 1 로 설정되어 있어야 한다.

1.7.3.6.1 매개 변수

xTimer : 확인하고자 하는 타이머의 핸들이다.

1.7.3.6.2 반환 값

확인하고자 하는 타이머에 할당된 식별자다.

1.7.3.6.3 기타

```
// 타이머에 할당된 콜백 함수다.
void TimerCallbackFunction( TimerHandle_t pxExpiredTimer ) {
    uint32_t ulCallCount;

    // 이 타이머가 만료되어 콜백 함수가 실행된 카운트는 타이머의 ID 에 저장된다. 카운트를 가져와 증가시킨 후 다시 저장한다.
    ulCallCount = ( uint32_t ) pvTimerGetTimerID( pxExpiredTimer );

    ulCallCount++;
    vTimerSetTimerID( pxExpiredTimer, ( void * ) ulCallCount );
}
```

1.7.3.7 xTimerGetTimerDaemonTaskHandle : 타이머 데몬 task 핸들 출력

```
#include “FreeRTOS.h”
#include “timers.h”

TaskHandle_t xTimerGetTimerDaemonTaskHandle ( void );
```

소프트웨어 타이머 데몬 task(또는 서비스)와 관련된 task 핸들을 반환한다.

FreeRTOSConfig.h 의 configUSE_TIMERS 이 1 로 설정되면 스케줄러가 시작될 때 타이머 데몬 task 가 자동적으로 작성된다. 모든 FreeRTOS 소프트웨어 타이머 콜백 함수는 타이머 데몬 task 의 컨텍스트에서 실행된다.

xTimerGetTimerDaemonTaskHandle()을 사용하려면, configRootConfig.h 의 configUSE_TIMERS 을 1 로 설정해야 한다.

1.7.3.7.1 반환 값

타이머 데몬 task 의 핸들이다. FreeRTOS 소프트웨어 타이머 콜백 기능은 소프트웨어 데몬 task 의 컨텍스트에서 실행된다.

1.7.3.8 xTimerPendFunctionCall : RTOS 데몬 task 에서 함수 실행

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerPendFunctionCall ( PendedFunction_t xFunctionToPend, void * pvParameter1,
                                    uint32_t ulParameter2, TickType_t xTicksToWait );
```

함수의 실행을 RTOS 데몬 task(타이머 서비스 task. timers.c 에 구현되고 ‘Timer’접두어가 붙어있다.)로 지연시킨다.

이 함수는 인터럽트 서비스 루틴에서 호출하면 안된다. 인터럽트 서비스 루틴에서 호출할 수 있는 버전은 xTimerPendFunctionCallFromISR()를 참조한다.

RTOS 데몬 task 로 연기하는 함수는 아래와 같은 프로토 타입을 가져야 한다.

```
void vPendableFunction (void * pvParameter1, uint32_t ulParameter2);
```

pvParameter1 및 ulParameter2 매개 변수는 응용 프로그램 코드에서 사용하도록 제공된다.

xTimerPendFunctionCall()을 사용하려면 FreeRTOSConfig.h 의 INCLUDE_xTimerPendFunctionCall() 및 configUSE_TIMERS 을 1 로 설정한다.

1.7.3.8.1 매개 변수

xFunctionToPend : 타이머 서비스 / 데몬 task 에서 실행할 함수이다. 이 함수는 PendedFunction_t 프로토 타입이어야 한다.

pvParameter1 : 함수의 첫번째 매개 변수로 콜백 함수에 전달할 값이다. 매개 변수는 void * 유형이므로 모든 유형을 전달하는데 사용할 수 있다.

ulParameter2 : 함수의 두번째 매개 변수로 콜백 함수에 전달할 값이다.

xTicksToWait : xTimerPendFunctionCall()을 호출하면 대기열에서 타이머 데몬 task(타이머 서비스 task)로 메시지를 전송한다. xTicksToWait 은 대기열이 가득차면 대기열에서 공간을 사용할 수 있을때까지 task 가 차단 상태로 대기하는 시간을 지정한다.

1.7.3.8.2 반환 값

pdPASS : 메시지가 RTOS 데몬 task 에 성공적으로 전송되었다.

다른 값 : 메시지 대기열이 이미 가득차서 메시지가 RTOS 데몬 task 로 전송되지 않았다. 대기열의 길이는 FreeRTOSConfig.h 의 configTIMER_QUEUE_LENGTH 값으로 설정된다.

1.7.3.9 xTimerPendFunctionCallFromISR : ISR 에서 RTOS 데몬 task 에서 함수 실행

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerPendFunctionCallFromISR ( PendedFunction_t xFunctionToPend, void * pvParameter1,
                                           uint32_t ulParameter2, BaseType_t * pxHigherPriorityTaskWoken );
```

응용 프로그램을 인터럽트 서비스 루틴에서 RTOS 데몬 task 로 함수의 실행을 지연시키는데 사용된다. (타이머 서비스 task 라고도 하며, timers.c 에 구현되며, ‘Timer’접두사가 붙는다.) 이상적으로 인터럽트 서비스 루틴(ISR)은 가능한 짧게 유지되지만, 때로는 ISR 이 많은 처리를 해야 하거나 결정적이지 않은 처리를 수행해야 하는 경우가 있다. 이러한 경우 xTimerPendFunctionCallFromISR()을 사용하여 RTOS 데몬 task 를 이용해 함수 처리를 지연시킬 수 있다. 인터럽트 이후에 보류된 기능을 실행하는 task 로 직접 리턴할 수 있게하는 메커니즘이 제공된다. 이것은 마치 콜백이 인터럽트 자체에서 실행된 것처럼 콜백 함수가 인터럽트와 함께 시간 내에 연속적으로 실행되도록 허용한다. RTOS 데몬 task 로 연기할 수 있는 함수는 아래 프로토 타입을 가져야 한다.

```
void vPendableFunction (void * pvParameter1, uint32_t ulParameter2);
```

pvParameter1 및 ulParameter2 매개 변수는 응용 프로그램 코드에서 사용하도록 제공한다.

xTimerPendFunctionCallFromISR()를 사용하려면 FreeRTOSConfig.h 의 INCLUDE_xTimerPendFunctionCall() 및 configUSE_TIMERS 모두 1 로 설정해야 한다.

1.7.3.9.1 매개 변수

xFunctionToPend : 타이머 서비스 / 데몬 task 에서 실행할 함수다. 함수는 PendedFunction_t 프로토 타입이어야 한다.

pvParameter1 : 콜백 함수의 첫번째 매개 변수로 전달되는 값이다. 매개 변수에는 void *유형이 있으므로, 모든 유형을 전달하는데 사용할 수 있다. 예를들어 정수 타입은 void *로 캐스트 될 수도 있고, 구조체를 가리키는데 사용될 수 있다.

ulParameter2 : 함수의 두번째 매개 변수로 콜백 함수에 전달되는 값이다.

pxHigherPriorityTaskWoken : xTimerPendFunctionCallFromISR()을 호출하면 대기열에서, RTOS 타이머 데몬 task 로 보내는 메시지다. FreeRTOSConfig.h 의 configTIMER_TASK_PRIORITY 값으로 설정된 데몬 task 의 우선 순위가 현재 실행중인 task(인터럽트 된 task)의 우선 순위보다 높으면 xTimerPendFunctionCallFromISR()에서 *pxHigherPriorityTaskWoken 가 pdTRUE 로 설정된다. 인터럽트가 종료되기 전에 컨텍스트 스위치가 요청되야 하는 것을 나타낸다. 이러한 이유로 * pxHigherPriorityTaskWoken 은 pdFALSE 로 초기화되야 한다.

1.7.3.9.2 반환 값

pdPASS : 메시지가 RTOS 데몬 task 에 성공적으로 전송되었다.

다른 값 : 메시지 대기열이 이미 가득차서 메시지가 RTOS 데몬 task 로 전송되지 않았다. 대기열의 길이는 FreeRTOSConfig.h 의 configTIMER_QUEUE_LENGTH 값으로 설정된다.

1.7.3.9.3 기타

```
// 데몬 task 의 컨텍스트에서 실행할 콜백 함수다. 콜백 함수는 모두 이 동일한 프로토 타입을 사용해야 한다.
void vProcessInterface( void *pvParameter1, uint32_t ulParameter2 ) {
    BaseType_t xInterfaceToService;

    // 서비스가 필요한 인터페이스는 두번째 매개 변수로 전달된다. 첫번째 매개 변수는 이 경우 사용하지 않는다.
    xInterfaceToService = ( BaseType_t ) ulParameter2;

    // 여기서 처리를 수행한다.
}
```

```
// 여러 인터페이스에서 데이터 패킷을 받는 ISR 이다.
void vAnISR( void ) {
    BaseType_t xInterfaceToService, xHigherPriorityTaskWoken;

    // 처리할 인터페이스를 결정하기 위해 하드웨어를 확인한다.
    xInterfaceToService = prvCheckInterfaces();

    /* 실제 처리는 task 로 연기된다. vProcessInterface()콜백 함수가 요청되면 처리가 필요한 인터페이스의 번호를 전달한다.
    서비스 할 인터페이스는 두번째 매개 변수로 전달한다. 첫번째 매개 변수는 이 경우 사용하지 않는다. */
    xHigherPriorityTaskWoken = pdFALSE;
    xTimerPendFunctionCallFromISR( vProcessInterface, NULL, ( uint32_t ) xInterfaceToService, &xHigherPriorityTaskWoken );

    /* xHigherPriorityTaskWoken 이 pdTRUE 로 설정되면 컨텍스트 스위치가 요청되야 한다. 사용된 매크로는 특정 포트 용도이며,
    portYIELD_FROM_ISR() 또는 END_SWITCHING_ISR()가 있다. 사용중인 포트의 설명서 페이지를 참조한다. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```