특집기사

# FreeRTOS 실시간 커널과 Core-A 프로세서(4)

## FreeRTOS 응용 프로그램

Core-A프로세서를 사용한 플랫폼에FreeRTOS 실시간 커널을 이식하고 Core-A 개발 보드에서 실행해 본다. FreeRTOS에 대해 살펴보았고, Core-A 기반 플랫폼에 이식하는데 필요한 주요 코드와 응용 프로그램을 개발 할 때 사용하는 API에 대해 살펴 보았다.

마지막으로 FreeRTOS가 이식된 Core-A 개발 보드에서 몇 가지 멀티 태스킹 예제를 실행시켜 본다.



## FreeRTOS 활용 Core-A 응용 프로그램

FreeRTOS를 이식한 Core-A 프로세서 기반 플랫폼에 비교적 간단하지만 가장 핵심적인 예제 프로그램 몇 가지를 실행하여 본다. 첫 번째 프로그램은 여러 태스크를 생성하고, 각 태스크가 일정 시간 간격으로 실행되는 것을 확인해 본다. 이 예제를 통해 실시간 특성을 어떻게 지키는지 등을 확인해 볼 수 있다. 두 번째 프로그램은 여러 태스크 사이에 정보를 메시지 큐를 통해 전달하는 것을 확인해 본다. 예제를 통해 여러 태스크로 나누어 일할 때 처리할 데이터를 어떻게 받고, 처리한 결과 데이터를 어떻게 전달하는 것인지 확인해 본다. 세 번째 프로그램은 뮤텍스를 이용하여 배타적으로 실행하는 경우를 살펴본다.

### ■ Core-A platform

Core-A에 관한 상세한 내용은 참고자료 [1]를 참조하고, Core-A 프로세서에서 실행될 프로그램을 컴파일하기 위한 크로스 컴파일 환경은 참고자료 [2]와 [3]를 참조한다.

[그림 1]에 FreeRTOS를 실행하는 환경을 개략적으로 보였다. Core-A 개발 보드[4]에 장착된 FPGA에 하드웨어를 iCON-USB를 통해 구성한다. FreeRTOS응용 프로그램을 컴파일하고, 그 결과를 iCON-USB와 BFM(Bus Functional Model)을 통해 메모리에 다운로드 한 후, Core-A가 해당 프로그램을 실행하도록 한다. Core-A가 프로그램을 실행하는 과정에서 출력하는 결과는 직렬통신 포트를 통해 문자 출력 모니터에 출력된다. [그림 1]에서 설명한 하드웨어와 이후 설명한 여러 예제를 컴파일하고 실행하는 방법의 상세한 것은 참고 자료 [7]을 참조한다.

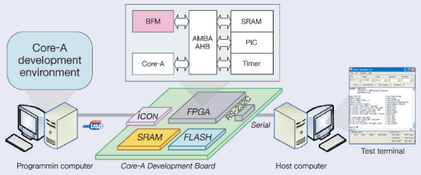


그림 1. FreeRTOS를 실행하는 환경

### ■ 멀티 태스킹 예제

이 예제 프로그램은 정해진 문자열을 출력하는 태스크를 세 개 생성한다. 각 태스크에는 0, 1, 2로 우선순위를 지정하고, 우선순위가 0인 태스크는 3초, 1인 태스크는 2초, 3인 태스크는 1초씩 기다린 후, 문자열 출력을 반복한다[1].

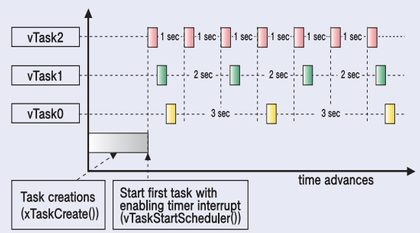


그림 2. 멀티 태스킹의 실행 일례

[그림 2]에서와 같이 'vTaskStartScheduler()'를 호출하여 멀티 태스킹을 시작하면, 'vTask2'는 매 1초마다 실행되고, 'vTask1'은 매 2초마다 실행되며, 'vTask0'는 매 3초마다 실행된다. 두 개 이상의 태스크가 동시에 실행되는 시점에는 우선순위가 높은 태스크가 먼저 실행되므로, 다음 순서로 실행된다.

```
(vTask2, vTask1, vTask0) – (vTask2) – (vTask2, vTask1) – (vTask2, vTask0)
– (vTask2, vTask1, vTask0) — … …
```

다음은 이 예제의 메인 루틴이며, 'prvSetupHadware()'를 호출하여 하드웨어를 초기화하고, 'xTaskCreate()' 루틴을 통해 태스크(vTask())를 세 개 생성한 후, 'vTaskStartScheduler()'를 호출하여 멀티 태스킹을 시작한다. 태스크를 생성할 때 각 태스크에 서로 다른 우선순위를 지정하고, 딜레이 시간을 파라메터로 전달한다.

```
#include "includes.h"
static void prvSetupHardware(void);
void vTask (void *pvParameters);
int main(void) {
    portBASE_TYPE delay[3] = {3000, 2000, 1000}; // delay in msec
    prvSetupHardware();
    uart_put_string("FreeRTOS on Core-A\r\n");
    uart_put_string("Dynalith Systems (www.Dynalith.com)\r\n");
    xTaskCreate(vTask, (signed char*)"vTask0", 100,
                (void*)&delay[0], 0 , (xTaskHandle*)NULL);
    xTaskCreate(vTask, (signed char*)"vTask1", 100,
                (void*)&delay[1], 1 , (xTaskHandle*)NULL);
    xTaskCreate(vTask, (signed char*)"vTask2", 100,
                (void*)&delay[2], 2 , (xTaskHandle*)NULL);
    vTaskStartScheduler();
    return 0;
}
```

다음은 태스크 코드인데, 자신의 우선순위를 'uxTaskPriorityGet()'으로 확인하고, 파라메터로 전달받은 딜레이를 확인한다. 이후 무한 루프를 수행하는데, 내부에서 정해진 문자열을 출력하고, 우선 순위를 출력하고, 현재의 틱 카운터 값을 출력한 후, 'vTaskDelay()'를 호출하여 일정 시간 기다린다.

```
void vTask(void *pvParameters) {
    unsigned portBASE_TYPE priority = uxTaskPriorityGet(NULL);
    portTickType delay = *(portTickType*)pvParameters;
    while (1) {
        portENTER_CRITICAL();
        uart_put_string("vTask"); uart_put_int(priority); uart_put_char(' ');
        uart_put_int(xTaskGetTickCount()); uart_put_string("\r\n");
        portEXIT_CRITICAL();
        vTaskDelay(delay/portTICK_RATE_MS); // wait ms
    }
}
```

위 코드에서 사용한 'uart_put_strin()'과 'uart_put_int()' 그리고 'uart_put_char()'는 UART를 통해 문자열과 수 그리고 문자를 출력하는 루틴이다. 위 태스크 코드에서 'portENTER_CRITICAL()'과 'portEXIT_CRITICAL()'을 통해 하드웨어를 참조하는 코드를 배타적으로 실행할 수 있도록 하고, 딜레이를 'portTICK_RATE_MS'로 나누어 절대 시간을 계산하여 사용하였다.

다음은 앞에서 설명한 멀티 태스크 예제의 실행 결과인데, 틱 카운터가 0일 때 모든 태스크가 실행되었음을 확인할 수 있고, 이때 우선순위에 따라 실행된 것을 알 수 있다. 이후, 매 1초(즉, 1msec 타이머 이터럽트에서 1,000개 티 카운터)마다 'vTask2'가 실행되고 있고, 매 2초마다 'vTask1'이 실행되고 있고, 매 3초마다 'vTask0'이 실행된다.

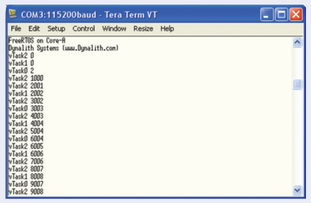


그림 3. FreeRTOS 멀티 태스크 예제 실행 결과

이 예제에서는 'xTaskDelay()'를 사용했는데, 'vTask2'가 '0', '1000', '2001', '3002', '4003' 등과 같이 정확하게 1000 틱 타이머 간격을 유지하지 못한다. 따라서 태스크가 정확한 시간 간격으로 실행되기 위해서는 'xTaskDelayUntil()'을 이용하는 것이 필요하다.

## ■ 큐를 활용한 태스크 사이 통신 예제

[그림 4]에 FreeRTOS의 큐(queue)를 이용한 태스크 사이의 통신을 이용한 예제의 동작을 개념적으로 표현하였다. 'vTaskSnd()' 태스크는 큐에 빈 방이 생기면 새로운 메시지를 만들어 채운다. 'vTaskRcv0()/vTaskRcv1()/vTaskRcv2()'는 큐에 메시지가 도착하면 읽고, 메시지에 전달된 내용을 출력한 후, 그 내용에 따라 일정 시간 기다린다. 여러 태스크가 큐에서 메시지를 동시에 읽는 경우를 방지하는 기능이 'xQueueReceive()'에 구현되어 있다.

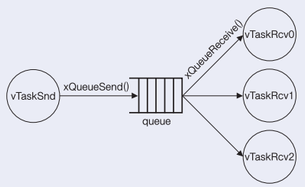


그림 4. 메시지큐를 이용한 태스크간 통신

다음 코드에서 보인 바와 같이, 'main()'에서 'prvSetupHardware()'를 호출하여 하드웨어를 초기화 하고, 이후 'xQueueCretate()'를 호출하여 사용할 큐를 준비하고 이때 만들어진 큐 핸들러는 이후 큐를 구별할 때 사용한다. 'vTaskCrate()'로 태스크 들을 생성하는데, 'TASK_CREATE()' 마크로를 이용하여 좀 더 용이하게 태스크를 생성했다. 이후 'vTaskStartScheduler()'를 호출하여 멀티 태스킹을 시작한다.

```
#include "includes.h"
xQueueHandle xQh;
static void prvSetupHardware(void);
void vTaskSnd (void *pvParameters);
void vTaskRcv (void *pvParameters);
#define TASK_CREATE(func,name,stack,priority)\
      if (xTaskCreate( (func), (signed char*)(name)\
                     , stack, (void*)(name), priority\
                     , (xTaskHandle*)NULL)!=pdPASS) {\
      uart_put_string(name); uart_put_string(" not created\r\n");\
      return 1;\
}
int main(void) {
  prvSetupHardware();

  uart_put_string("FreeRTOS on Core-A\r\n");
  uart_put_string("Dynalith Systems (www.Dynalith.com)\r\n");

  xQh = xQueueCreate(10, sizeof(long));

  TASK_CREATE(vTaskSnd,"vTaskSnd",100,1)
  TASK_CREATE(vTaskRcv,"vTaskRcv0",100,1)
  TASK_CREATE(vTaskRcv,"vTaskRcv1",100,1)
  TASK_CREATE(vTaskRcv,"vTaskRcv2",100,1)

  vTaskStartScheduler();
  return 0;
}
```

다음 코드는 큐에 메시지를 채우는 태스크인데, 난수 생성 함수를 통해 수를 만든 후 큐를 통해 다른 태스크로 전달한다. 메시지에 전달되는 수는 해당 메시지를 받은 태스크가 몇 초를 기다릴 것인지를 정한다.

```
void vTaskSnd(void *pvParameters) {
  long  msg;
  char *name = (char *)pvParameters;
  portENTER_CRITICAL();
  uart_put_string(name); uart_put_string(" started\r\n");
  portEXIT_CRITICAL();
  while (1) {
    msg = (rand())%5; msg = (msg==0) ? 1 : msg;
    xQueueSend(xQh, (void*)&msg, portMAX_DELAY);
  }
}
```

다음 코드는 큐에서 메시지를 읽어서 그 내용에 따라 문자열을 출력하고 일정 시간 기다리는 태스크이다.

```
void vTaskRcv(void *pvParameters)
{
  long msg;
  char *name = (char *)pvParameters;
  portENTER_CRITICAL();
  uart_put_string(name); uart_put_string(" started\r\n");
  portEXIT_CRITICAL();
  while (1) {
    xQueueReceive(xQh, (void*)&msg, portMAX_DELAY
      portENTER_CRITICAL();
      uart_put_string(name); uart_put_string(" got ");
      uart_put_int(msg); uart_put_string("\r\n");
      portEXIT_CRITICAL();
    vTaskDelay((portTickType)(msg*1000/portTICK_RATE_MS));
  }
}
```

[그림 5]에 이제까지 설명한 응용 프로그램이 실행된 결과를 보였다.

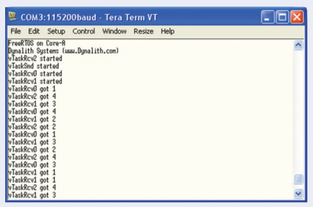


그림 5. FreeRTOS 큐를 이용한 태스크간 통신 결과

## ■ 뮤텍스를 이용한 예제

여러 태스크가 실행되는 환경에서 공유 자원을 참조하거나 이와 관련된 코드를 실행할 때 특정 코드를 방해 없이 수행할 필요가 생긴다.

앞 예제들에서 문자열을 출력할 때 'portENTER_CRITICAL()'와 'portEXIT_CRITICAL()'을 사용했다. 이 예제에서는 FreeRTOS에서 제공하는 뮤텍스를 이용한다.

다음 여러 태스크 사이의 동기화를 위해 사용할 뮤텍스를 생성하고, 비교적 긴 문자열을 출력하는 세 개 태스크를 생성하는 코드이다. 각 태스크에는 우선순위가 다르게 지정되어 선점형 스케줄링에서 우선순위가 높은 태스크가 낮은 우선순위의 태스크가 실행 중이라도 프로세서를 선점할 수 있도록 한다.

```
#include "includes.h"
static void prvPrintTask( void *pvParameters );
int main( void ) {
    prvSetupHardware();

    uart_put_string("FreeRTOS on Core-A\r\n");
    uart_put_string("Dynalith Systems (www.Dynalith.com)\r\n");

    xMutex = xSemaphoreCreateMutex();

    xTaskCreate( prvPrintTask, (signed char*)"Print1", 1000
                , "Task 1 ****************************************\r\n"
                , 1, NULL );
    xTaskCreate( prvPrintTask, (signed char*)"Print2", 1000
                , "Task 2 ----------------------------------------\r\n"
                , 2, NULL );
    xTaskCreate( prvPrintTask, (signed char*)"Print3", 1000
                , "Task 3 ||||||||||||||||||||||||||||||||||||||||\r\n"
                , 3, NULL );
    vTaskStartScheduler();
    return 0;
}
```

다음은 파라메터로 전달받은 비교적 긴 문자열을 출력하는 태스크인데, 문자열을 완전하게 출력할 때까지 해당 코드 부분을 배타적으로 실행할 수 있도록 한다.

```
static void prvPrintTask( void *pvParameters ) {
    char *pcStringToPrint;
    pcStringToPrint = ( char * ) pvParameters;
    for( ;; ) {
        xSemaphoreTake( xMutex, portMAX_DELAY );
        uart_put_string((char*)pcStringToPrint);
        xSemaphoreGive( xMutex );
        vTaskDelay( ( rand() & 0xFF ) );
    }
}
```

위 코드에서 'xSemaphoreTake()'에서부터 'xSemaphoreGive()' 사이의 코드는 다른 태스크의 방해 없이 실행할 수 있도록 보장한다. 만약 이들 두 루틴이 없다면, 우선순위가 높은 태스크가 중간에 문맥 교환을 통해 끼어들 수가 있다.
[그림 6]은 뮤텍스 예제를 실행한 결과인데, 왼쪽 것은 뮤텍스를 사용한 경우이고, 오른쪽은 뮤텍스를 사용하지 않은 경우이다. 그림에서 알 수 있듯이 뮤텍스를 사용하지 않은 경우, 우선순위가 높은 Task2가 우선 순위가 낮은 Task1이 실행 중인 중간에 끼어들어서 출력 결과가 예상과 다르게 나옴을 알 수 있다.

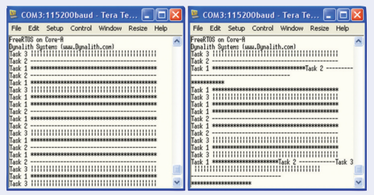


그림 6. 뮤텍스 예제의 결과 (왼쪽이 뮤텍스를 사용한 경우)

**▶ 우선 순위 반전과 승계**

뮤텍스를 사용할 경우, 선점형 스케줄링과정에 순수한 우선순위에 따라 태스크의 실행 순서가 결정되지 않을 수 있다. [그림 7]에 우선순위가 반전되는 경우를 보였다.

태스크 'vTaskHP'는 가장 우선순위가 높고, 태스크 'vTaskLP'는 가장 우선순위가 낮다. 그 중간에 태스크 'vTaskMP'가 있다. 'vTaskHP'와 'vTaskLP'는 상호 배타적으로 실행해야 할 코드가 있어 뮤텍스를 사용하고, 'vTaskMP'는 해당 뮤텍스를 사용하지 않는다.

(1)vTaskLP가 우선순위가 낮음에도 시간상으로 먼저 실행되어 vTaskHP 보다 먼저 뮤텍스를 차지하고 배타적 코드를 수행하고 있다. (2)vTaskHP가 실행하게 되어 우선순위에 따라 vTaskLP를 선점하지만 뮤텍스를 얻지 못해 곧 문맥 교환이 되어 뮤텍스가 가용할 때까지 기다린다.

(3)vTaskLP가 뮤텍스를 내 놓기 전에 이 보다 우선순위가 높은 vTaskMP가 vTaskLP를 선점한다. 여기서 vTaksMP는 뮤텍스와 무관하므로 자신의 일을 마칠 때까지 실행된다.
(4)vTaksMP가 문맥 교환되어 vTaskLP가 실행된다. 이때까지 가장 높은 우선순위의 vTaskHP는 뮤텍스를 기다리고 있다.
(5)vTaskLP가 뮤텍스를 내 놓으면 곧바로 vTaskHP가 재개된다. 이 과정에서 vTaksMP는 자신보다 우선순위가 높은 vTaskHP보다 먼저 실행되며, 이것을 우선순위 반전(priority inversion)이라 한다.

여기서는 세 개 태스크만을 고려했지만 vTaskHP보다 우선순위가 낮고 vTaskLP보다 우선순위가 높은 태스크가 여럿 존재하면 우선순위 반전 구간이 오래 지속할 수 있다. 이는 시간 제약을 반드시 지켜야 할 실시간 응용에 심각한 문제가 된다.

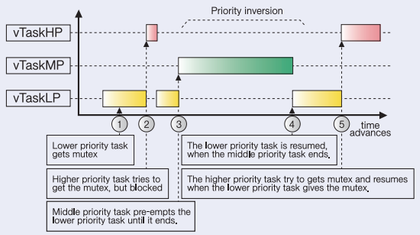


그림 7. 우선 순위 반전의 일례

우선순위 반전 문제를 해결하는 방법으로 FreeRTOS는 우선순위 승계(priority inheritance) 기법을 사용한다. [그림 7]에서 (2)번 과정에서 vTaskLP가 vTaskHP의 우선순위를 승계하여 뮤텍스를 되돌려 줄 때까지 vTaskHP보다 낮은 우선순위의 태스크가 중간에 실행될 수 없도록 한다.

## 종합

시스템집적 반도체와 내장형 시스템에 사용할 목적으로 개발된 32-비트 RISC 프로세서인 Core-A를 실시간 응용에 적용하기 위해, 실시간 특성이 있고 멀티 태스킹을 지원하는 소프트웨어 실행 환경이 필요하다. 여기서는 FreeRTOS 실시간 커널을 Core-A에 이식하는 과정과 Core-A 개발 보드에서 실행하는 것을 다루었다.

FreeRTOS는 우선순위 기반 스케줄러 기법을 사용하며, 같은 우선순위에 대해서는 순환 순서(라운드 로빈, round-robin)를 사용한다. 스케줄러는 선점형(preemptive)과 비선점형 중 하나를 선택할 수 있다. 선점형인 경우, 높은 우선 순위의 태스크가 언제든 프로세서를 선점한다. 태스크(task)와 더불어 코루틴(co-routine)도 지원한다. 태스크 간 통신과 동기를 위해 큐(queue), 세마포(semaphore), 뮤텍스(mutex) 등을 제공한다.

특히 큐는 태스크 간뿐 아니고 인터럽트 서비스 루틴과 태스크 간의 정보 전달에도 사용할 수 있다. 비교적 단순한 동적 메모리 할당 기법을 제공한다. 태스크를 관리하기 위해 TCB(Task Control Block) 자료 구조를 사용하고, 필요에 따라 기능 대부분과 그 기능에 관련된 코드들을 선택적으로 사용할 수 있어서 코드를 크기 면에서 최적화하는데 적합하다.

FreeRTOS는 원천 코드가 공개되어 있고, 사용료가 없으며, 상용으로도 활용할 수 있고 프로세서 의존적인 부분이 매우 제한적이라 새 프로세서에 이식하는데 큰 어려움이 없다. Core-A는 간단한 협약을 통해 원천 코드를 받을 수 있고, 사용료도 없으며, 하드웨어와 소프트웨어 개발 환경이 준비되어 있어서 새로운 시스템을 구성하기 용이하다. 이들 둘을 조합하여 실시간 응용에 적합한 시스템을 구현하는데, 본 고가 다룬 내용이 도움되었기를 바라며, 지면이 제한된 관계로 상세하게 다루지 못한 부분이 있지만, 독자들이 큰 어려움 없이 이해할 수 있을 것이라 기대한다.

[1] RMS(Rate Monotonic Scheduling, 비율 단조 스케줄링): 자주 처리해야 될 태스크에 높은 우선 순위를 지정하는 방식

## 저작권과 라이센싱

- Core-A는 무상으로 사용할 수 있지만, Core-A 개발자와 협약을 맺어야 한다. (www.Core-A.com)
- FreeRTOS는 공개코드 소프트웨어이며 무상으로 상용 제품에도 사용할 수 있다. (www.FreeRTOS.org)
- 본고에서 다루는 내용과 코드 중, FreeRTOS를 Core-A에 이식하기 위해 추가된 FreeRTOS와 분리된 코드는 교육용인 경우에 한하여 자유롭게 사용할 수 있지만, 정부과제나 기업과제 그리고 제품에 사용할 경우 개발자와 협의하여야 한다. (www.Dynalith.com, corea@Dynalith.com)

## Reference


[1] 박인철, Core-A Computer Architecture and Design, 홍릉과학출판사, 2009.10.
[2] 기안도, Core-A 프로세서를 활용한 플랫폼 설계, 홍릉과학출판사, 2010.3.
[3] 이종열, Core-A 프로그래머 가이드, 홍릉과학출판사, 2009.11.
[4] Core-A 개발 보드 사용자 매뉴얼, 다이나릿시스템, 2009.
[5] 기안도, Core-A 프로세서와 실시간 커널 MicroC/OS-II, IDEC Newsletter 155호, 156호, 157호, 2010.5~7.
[6] Richard M. Stallman et. al., Using the GNU Compiler Collection, GNU Press.
[7] Ando Ki, FreeRTOS Port for Core-A Platform, Application Note, DS-AN-2010-10-001, Dynalith Systems.



(주)다이나릿시스템
기안도 박사
연구분야 : 시스템집적반도체 설계와 검증
E-mail : adki@dynalith.com
http://dynalith.com