



솔루션 공과 전자학원
SOLUTION ENGINEERING
ELECTRONIC ACADEMY

실시간 운영체제

(FreeRTOS : ARM Cortex-M)

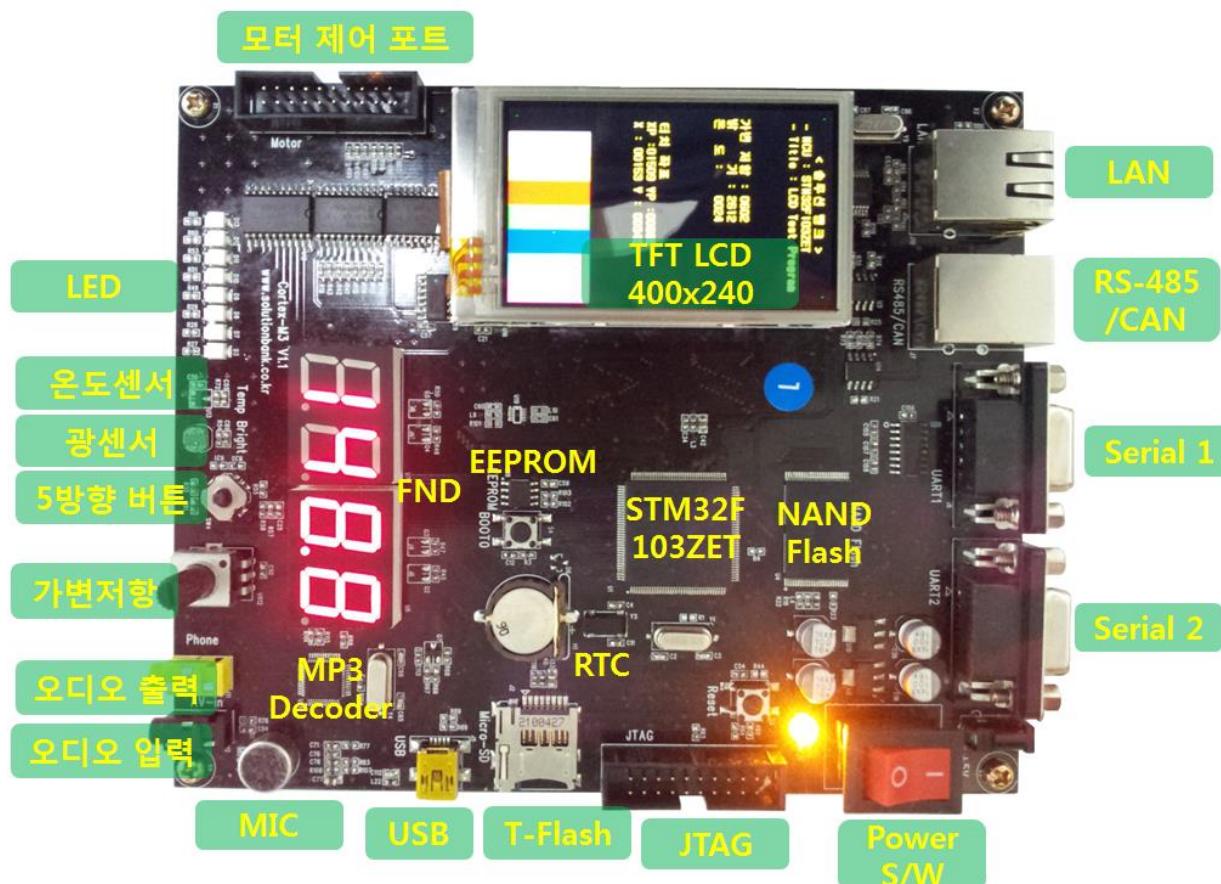


목 차

1.	Cortex 개발 환경	2
1.1	타겟 보드	2
1.2	라이팅 장비	2
1.3	컴파일러	3
2.	Cortex-M3	9
2.1	기본 구조	9
2.2	예외 모델	10
2.3	Core peripherals	15
3.	프로젝트 구성	32
3.1	프로젝트 생성	32
3.2	CMSIS(Cortex Microcontroller Software Interface Standard)	38
3.3	라이브러리 설치	40
3.4	FreeRTOS 설치	44
4.	FreeRTOS 시작하기	49
4.1	실시간 시스템	49
4.2	RTOS(Real Time Operating System) 란?	50
4.3	멀티태스킹(Multitasking)	50
4.4	비선점형/선점형 커널	52
4.5	문맥 전환(Context Switching)	54
4.6	FreeRTOS 개요	55
4.7	FreeRTOS 시작시키기	59
5.	FreeRTOS	61
5.1	태스크 관리(Task Management)	61
5.2	큐 관리(Queue Management)	82
5.3	인터럽트 관리(Interrupt Management)	98
5.4	소프트웨어 타이머	105
5.5	자원 관리	108
5.6	참조	116
6.	드라이버 작성	118
6.1	개요	118
6.2	전송 모드(Transfer Mode)	119
6.3	Poll 모드	119
6.4	Interrupt driven zero copy 모드	120
6.5	Interrupt driven circular buffer 모드	120
6.6	Interrupt driven character queue 모드	120

1. Cortex 개발 환경

1.1 타겟 보드



1.2 라이팅 장비

- J-Link (<http://www.segger.com/index.html>)



1.3 컴파일러

1.3.1 IAR workbench



홈페이지 : www.iar.com

1.3.1.1 컴파일러 다운로드

IAR Systems 에서는 다음과 같은 사이트에서 ARM 컴파일러를 구할 수 있다.

여기서 Time-limited license 는 30일 동안 사용해 볼 수 있는 버전이고 Size-limited license는 컴파일의 크기가 32k로 제한된 버전이다.

The evaluation license for IAR Systems software is free of charge. The only requirement is that you need to register with us. The evaluation license is intended for prospective customers to test and evaluate IAR Systems software.

IAR Embedded Workbench

30-day time-limited evaluation license:
Fully functional for 30 days after installation with the following limitations: no MISRA C support, source code for runtime libraries is not included. The 30-day time-limited evaluation must not be used for product development or any other kind of commercial use.

Kickstart, size-limited evaluation license:
Code size limited license without any time limitation but, no MISRA C support, no power debug functionality, source code for runtime libraries is not included.

Processor or core	Time-limited license	Size-limited license
ARM	v6.50	v6.50 (32K)

1.3.1.2 주요 특징

주요 구성 요소:

- 프로젝트 관리 및 Editor 를 포함한 통합 개발 환경
- ARM 을 위한 가장 최적화된 C 와 C++ compiler
- 자동 MISRA C rules (MISRA C:2004) 확인
- ARM EABI 와 CMSIS 호환성
- 광범위한 HW system 지원
- I-jet 과 JTAGjet hardware debug probe 지원
- Source code 에 대한 가시적인 소비 전력 측정 및 Power debugging 기능 제공
- Source code 를 포함한 실시간 libraries
- Relocating ARM assembler
- Linker 및 librarian tools
- ARM simulator 를 통한 C-SPY® debugger, JTAG 지원 및 RTOS-aware debugging 지원

- IAR Systems 와 RTOS 사들과의 제휴를 통한 RTOS 플러그인 지원
- 다양한 하드웨어 보드를 위한 3100 개 이상의 예제 프로젝트 제공
- PDF 형 사용자 설명서 및 사용 가이드 제공
- Online 기술 지원

상세 Chip 지원 사항:

- 3,100 개 이상의 예제 프로젝트를 제공하고 있으며, Actel, Analog Devices, Aiji Systems, ARM, Atmel, Cirrus Logic, EnergyMicro, Freescale, Fujitsu, Holtek, Keil, LogicPD, Micronas, Nohau, Nuvoton, NXP, OKI, Olimex, ON Semiconductor, Pasat, Phytec, Samsung, ST, Texas Instruments 및 Toshiba chip들을 지원
- ARM 과 Thumb mode 향 4 Gbyte application 지원
- ARM 혹은 Thumb mode로 Compile 가능
- VFP(Vector Floating Point coprocessor) code 생성
- Intrinsic NEON 지원

하드웨어 디버깅 지원:

Probe (JTAG/SWD)	Note
I-jet	ARM7/ARM9/ARM11 및 Cortex-M/R/A core 지원
IAR J-Trace	모든 ARM7, ARM9, Cortex-M3/M4 cores에 대한 ETM 지원, IAR J-Link와 같은 JTAG/SWD 지원, USB를 통한 통신
IAR J-Link	모든 ARM7/ARM9/ARM11 및 Cortex-M0/M1/M3/M4/R4(F)/A5 core 지원
IAR J-Link Ultra	Cortex core 향 Power debugging 지원
JTAGjet	Debugger에의 Trace module 기능 추가 지원
RDI	Debug probe에의 RDI (Remote Debug Interface) 기능 추가 지원
GDB server	
Jeeni EPI	
Stellaris FTDI	LMI FTDI driver
Stellaris ICDI	
Macraigor OCDemon	mpDemon, usbDemon, usb2Demon, usb2Sprite
P&E Micro JTAG probes	Multilink, Cyclone, OSJTAG
STMicroelectronics ST-LINK V2	STM32 기기 지원
STMicroelectronics ST-LINK	STM32 기기 지원
SAM-ICE	Atmel AT91SAM 기기 지원
J-Link Lite LPC Edition	ARM 기반 NXP 기기 지원

mIDASLink	Analog Devices 기기 지원
DIGI JTAG Link	DIGI 기기 지원
XDS100	TI 기기 지원
ROM-monitor	Note
IAR ROM-monitor	Analog Devices, NXP, 및 OKI 보드 등에서 사용
Angel ROM-monitor	Atmel 및 Cirrus Logic 보드 등에서 사용

RTOS 지원: real time operating system

Operating system	Built-in plugin	Vendor plugin
AVIX	Yes	
CMX-RTX	Yes	
CMX-Tiny+	Yes	
e-Force µC3/Compact	Yes	
eSysTech X Realtime kernel	Yes	
Express Logic ThreadX	Yes	
FreeRTOS, OpenRTOS, SafeRTOS	Yes	
Freescale MQX	Yes	
Micrium µC/OS-II	Yes	
Micro Digital SMX RTOS	Yes	
NORTi MiSPO	Yes	
OSEK (ORTI)	Yes	
Quadros RTXC	Yes	
Segger embOS	Yes	
Unicor Fusion	Yes	

각각의 RTOS 는 특정 breakpoint 를 포함하는 작업이나 thread list 창을 포함한 Plugin들은 C-SPY안의 각각 새로운 windows에 설치됩니다. RTOS의 내부 Data structure와 같은 timer, queue, semaphore, resource 및 mailbox등은 inspector windows를 통하여 볼 수 있습니다.

Supported devices:

IAR Embedded Workbench 는 ARM7, ARM9, ARM9E, ARM10E, ARM11, SecurCore, Cortex M0, M0+, M1, M3, M4(F), R4(F), R5, R7, A5, A7, A8, A9, A15, 및 광범위한 XScale 기기들을 지원합니다.

1.3.2 Keil

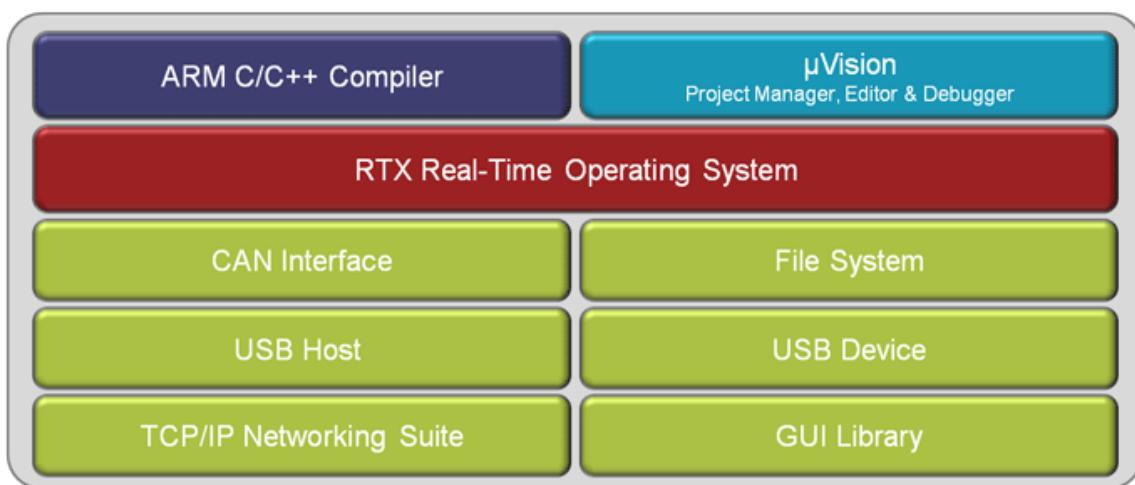


1.3.2.1 컴파일러 다운로드

아래의 주소에서 평가판을 다운로드 받을 수 있다.

<http://www.keil.com/arm/mdk.asp>

MDK-ARM Microcontroller Development Kit



[Request a Quote](#)

[Download](#)

1.3.2.2 주요 특징

- Cortex-M, Cortex-R4, ARM7, ARM9 기반 디바이스 완벽 지원
- 산업표준 ARM C/C++ Compiler 내장
- μVision IDE4: 통합개발환경, 디버거, 시뮬레이터
- Real-Time Operating System RTX Kernel 제공 – **소스코드 포함**
- 멀티 프로토콜 및 다양한 어플리케이션을 제공하는 TCP Networking Suite 제공
- 표준 드라이버단 USB Device / Host 스택 제공
- ULINKpro 를 이용한 Cortex-M3 인스트럭션 분석 – on-the-fly 분석
- 프로그램 실행에 대한 완벽한 Code Coverage 정보
- 프로그램 최적화를 위한 실행 프로파일러 및 퍼포먼스 어널라이저
- Device Database 를 통한 각 디바이스별 스타트업 코드 제공
- 다양한 어플리케이션 예제
- CMSIS(Cortex Microcontroller Software Interface Standard) 적용

1.3.3 GCC

- Tool chain

Target 시스템의 Software 개발을 진행하기 위해 필요한 host system의 cross compile (교차 컴파일) 환경을 우리는 툴체인이라고 부른다.

여기서 Cross라는 말이 의미하는 것은 우리가 개발 환경으로 사용하는 PC는 CPU가 어떤 것이든 Target에서 사용하는 CPU와는 다른 것이 될 것이다. PC에서 개발을 수행하고, 그 PC에서 컴파일 해서 바이너리를 생성한다. 그러나 이 바이너리 파일은 PC에서 동작하지 않고 Target 보드에 있는 CPU에서 동작하는 바이너리 파일이다. 이와 같이 개발하는 곳의 환경과, 실제 개발된 코드가 동작될 환경이 다를 경우 이러한 것을 Cross라고 부르고 있다.

툴체인은 컴파일러만을 의미하지는 않는다. source code를 compile하고 build하여 바이너리 실행 파일을 생성하는데 필요한 각종 Utility 및 Library의 모음이라고 생각하면 된다. 기본적으로는 Assembler, Linker, C compiler, C library 등으로 구성되어 있습니다.

ARM Cortex 개발에서도 GCC로 개발하기 위한 Tool chain이 많이 있지만 대표적인 것을 두 가지만을 설명한다.

1.3.3.1 Sourcery CodeBench



1.3.3.1.1 다운로드

아래 사이트에서 다운로드 할 수 있다.

<http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/>

Sourcery CodeBench Lite 2013.11-24

Status: Release

This is a fully-validated release.

This release was made on 4 December 2013.

Software

Download	MD5 Checksum
Recommended Packages	
IA32 GNU/Linux Installer	7267c1490a84e72478fa49c15b052690
IA32 Windows Installer	be97d65bd77cd44eed75f2894acae9
Advanced Packages	
IA32 GNU/Linux TAR	8ba2bddb0a28f5606db3b6f1b9ffe6af
IA32 Windows TAR	6b3ff249e3feed53d20784824c7f5758
Source TAR	3d89a0fb72e1e3ccb3fed500dbeec8f

What's in this release?

The datasheet provides information about key components of Sourcery CodeBench Lite 2013.11-24.

Most users prefer the easy-to-install recommended packages. Expert users may prefer the advanced packages.

You may use the md5sum utility to verify that your download has completed correctly.

1.3.3.2 GNU Tools for ARM Embedded Processors



1.3.3.2.1 다운로드

아래 사이트에서 다운로드 할 수 있다.

<https://launchpad.net/gcc-arm-embedded>

Downloads

Latest version is 4.8-2014-q1-update

release.txt



gcc-arm-non...4-win32.exe



gcc-arm-non...4-win32.zip



gcc-arm-non...nux.tar.bz2



gcc-arm-non...mac.tar.bz2



gcc-arm-non...src.tar.bz2



How-to-buil...olchain.pdf



readme.txt



license.txt



released on 2014-03-28

2. Cortex-M3

2.1 기본 구조

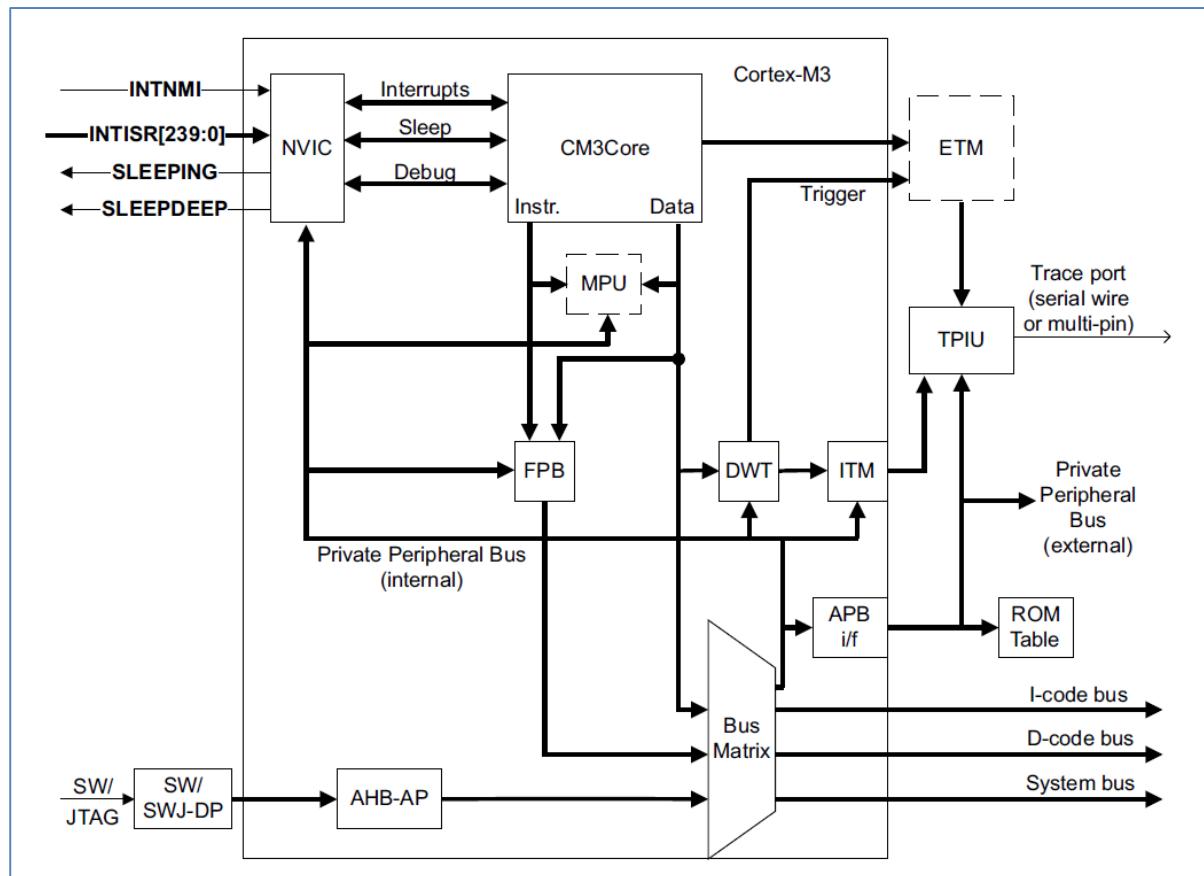
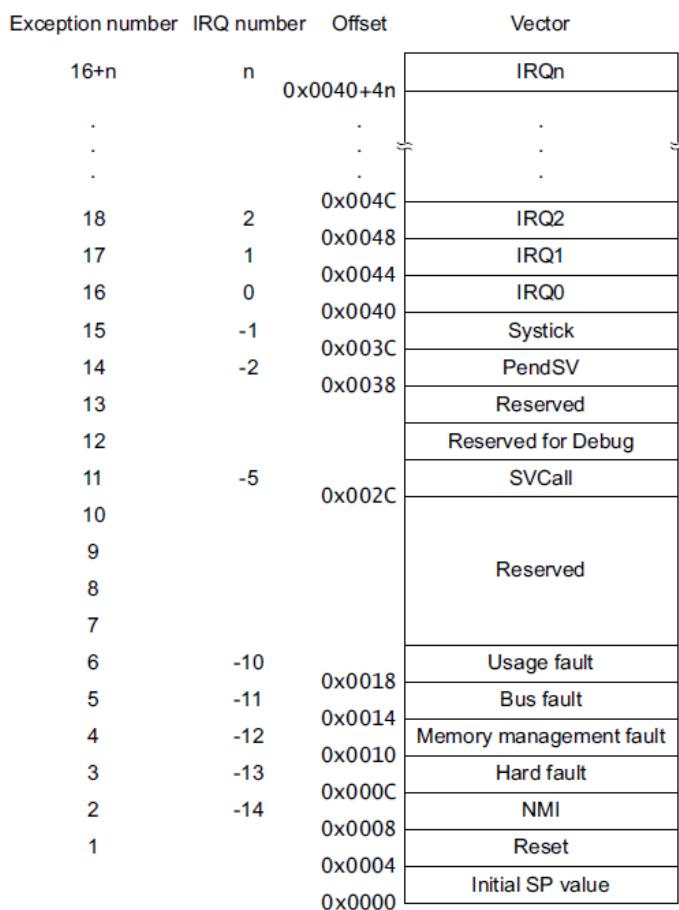


그림 1 Cortex 블록도

2.2 예외 모델

Exception Number	Exception Type	IRQ Number	Priority	설명
0				리셋시 vector table의 첫 entry
1	Reset	-	-3	Power up과 Warm reset시
2	Non-maskable Interrupt(NMI)	-14	-2	Pre-empted 되지 않는다.
3	Hard Fault	-13	-1	모든 종류의 Fault
4	Memory Management	-12	Configurable	MPU(Memory Protection Unit) mismatch
5	Bus Fault	-11	Configurable	Pre-fetch, memory fault
6	Usage Fault	-10	Configurable	Undefined instruction
7 ~ 10	-	-		Reserved
11	SVCALL	-5	Configurable	System service 호출
12	Debug Monitor	-		Not halting Debug monitor
13		-		Reserved
14	PendSV	-2	Configurable	Pendable request
15	SysTick	-1	Configurable	System tick timer
16~255	External Interrupt	0 ~	Configurable	Core 밖 외부 인터럽트(0 ~ 239)



2.2.1 예외 우선순위와 처리

Cortex-M3는 내부에 NVIC(Nested Vectored Interrupt Controller) 가 있어서 각종 인터럽트를 효율적으로 제어할 수 있다. 모든 Exception의 우선순위를 결정하고 처리한다. 우선순위는 인터럽트가 동시에 여러 개가 발생할 경우 발생한 인터럽트를 중요도가 높은 작업부터 우선적으로 처리하게 된다.

Nested 라는 말에서도 인지할 수 있듯이 하나의 인터럽트가 처리되고 있는 중에 이보다 더 높은 우선 순위를 갖는 인터럽트가 발생하였을 경우 처리되고 있던 인터럽트는 처리를 잠시 멈추고 새로 발생한 높은 우선순위의 인터럽트가 처리될 수 있다. 이러한 것을 인터럽트가 nested 되었다고 말하는 것이고, 이것이 가능하도록 설계되어 있다.

Interrupt 번호가 낮을 수록 높은 Priority를 가진다. Reset은 가장 높은 Priority를 가진다 우선순위 -3, -2, -1을 가지는 Reset, Non-maskable Interrupt (NMI), Hard Fault는 우선 순위를 변경할 수 없다. 고정된 우선 순위를 가지고 있고, 가장 높은 우선순위를 가진다.

어떤 경우에는 하나의 우선 순위를 공유할 수도 있다. 이를 위해 Preempting Priority와 Subpriority를 두어서 그룹으로 만들어서 관리하는 방법을 제시하고 있다.

2.2.2 인터럽트 비교

ARM7, ARM9에서는 실제적으로 두 개의 인터럽트만을 지원한다. 실제로 집에서 수많은 인터럽트가 존재하는데 단 2개 만을 지원한다는 것이 잘 와 닳지 않을 수 있다. 하지만 이것은 ARM core에 대한 설명이다. ARM core를 기반으로 CPU칩을 제조하였고, 그 집의 내부에는 interrupt controller가 들어 있어서 수많은 인터럽트들을 처리하게 된다. 하지만 ARM core에게 전달되는 인터럽트는 단 2가지이다 IRQ, FIQ가 그것이다. 물리적으로는 UART, Keypad, USB 등등 많은 peripheral들이 interrupt로 동작하고 있지만 실제 ARM core에는 IRQ 나 FIQ로 전달되었던 것이다. 내부에 있는 interrupt controller가 외부에 많은 peripheral들의 interrupt를 IRQ나 FIQ로 매핑하는 것이다.

nested와 관련해서 같은 IRQ가 발생하게 되면 이것은 구분이 되지 않기 때문에 ARM core는 하나의 IRQ가 모두 처리될 때까지 다른 IRQ를 받아들이지 않게 된다. 물론 기존 ARM7, ARM9에서도 IRQ가 수행되는 동안에 FIQ가 발생되면 이것은 중첩이 되고 IRQ의 상태가 저장되고 FIQ를 수행하게 된다.

이것은 서로 다른 수행 모드에 대한 처리와도 같다. Cortex-M3에서는 모든 인터럽트가 중첩이 가능하다. Cortex-M3는 코어 입장에서 255개의 인터럽트를 받아들일 수 있다. 이중 15개는 고정된 코어 내부 인터럽트이고, 240개는 외부에서 코어로 입력되는 인터럽트이다. ARM7, ARM9에서 단 2 개밖에 없던 것에서 서로 다른 우선순위를 가지고 nested 처리까지 해주는 매우 강력하게 변한 모습을 가지는 것이다.

240개의 외부인터럽트에 대한 구현은 제조사에서 결정하게 된다. STM에서도 240개가 모두 구성되어 있지는 않다. 이 또한 STM에서 제공되는 칩의 종류에 따라서도 모두 다르다. 인터럽트 발생시 기존에 수행하던 상태를 Stack에 저장하고, 인터럽트 수행이 종료되면 Stack으로부터 복구되어야 한다. 기존의 ARM7, ARM9에서는 소프트웨어 방식으로 이 저장과 복구를 처리하였고 이 부분이 프로그램 상에서 반드시 존재했어야 한다. 하지만 Cortex-M3에서는 하드웨어적으로 자동으로 처리가 된다.

2.2.3 Interrupt priority grouping

인터럽트를 사용하는 시스템에서 우선순위의 제어의 필요성이 증가함에 따라서 NVIC에서는 우선 순위를 그룹화 시키는 기능을 제공한다. 이 것은 Interrupt priority register entry 를 2부분으로 나눈다.

- 상위 부분은 group priority 를 정의한다.
- 하위 부분은 그룹안에서의 subpriority를 정의한다.

Group priority 는 인터럽트가 걸렸을 경우에 선점이 되는냐 여부를 결정한다. 만약 인터럽트 핸들러가 실행되고 있는 도중에 같은 인터럽트 그룹에 속하는 인터럽트가 발생하면 실행되고 있는 인터럽트 핸들러는 선점되지 않는다.

그리고 같은 그룹에 속해 있는 인터럽트가 동시 다발적으로 발생이 되어 팬딩이 되어있는 상태라고 하면 subpriority에 따라 순서적으로 실행이 된다.

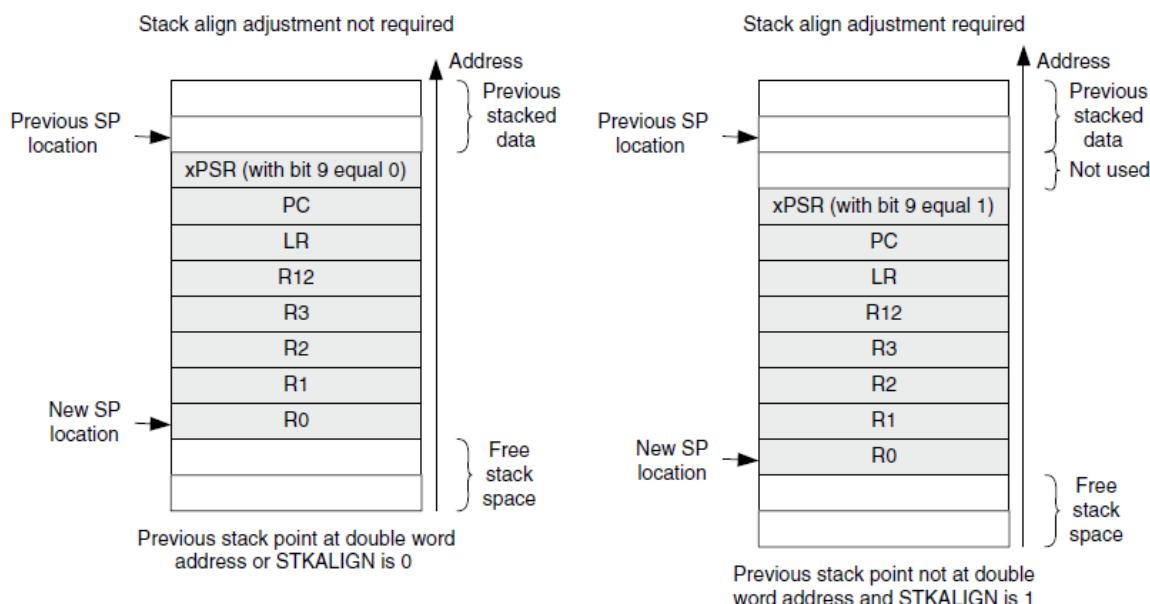
2.2.4 인터럽트 / 예외 처리 순서

예외가 발생하면 다음과 같은 일이 발생하게 된다.

- **Stacking** : 스택에 8개의 레지스터 값을 Push 한다.
- **벡터 패치(Vector Fetch)** : 벡터 테이블에서 예외 핸들러의 시작 주소를 가지고 온다.
- SP(Stack Pointer), LR(Link Register), PC(Program Counter)를 갱신한다.

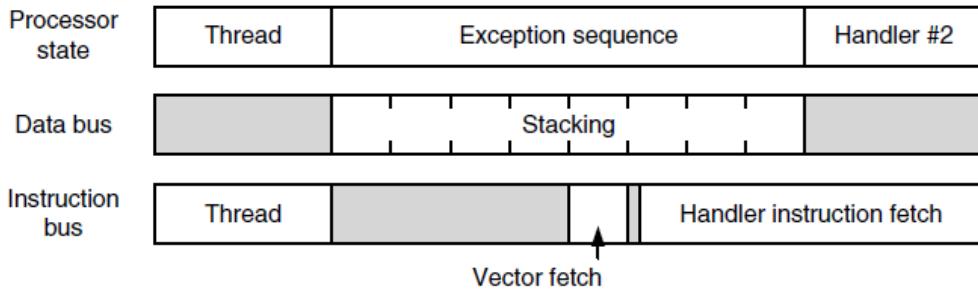
2.2.4.1 Stacking

예외가 발생하게 되면 R0~R3, R12, LR, PC, Program Status(PSR)이 스택에 Push 된다. 코드가 PSP(Process Stack Pointer)를 사용하고 있었으면 Process stack을 사용하고, MSP(Main Stack Pointer)를 사용하고 있었으면 Main stack을 사용하게 된다.



2.2.4.2 Vector Fetch

벡터 테이블에서 예외 핸들러의 시작 주소를 가지고 있는 예외 벡터를 Fetch한다. Stacking 과 Vector Fetch는 다른 버스에서 수행되므로 동시에 이루어 진다.



2.2.4.3 레지스터 갱신(Register Update)

Stacking과 Vector fetch가 끝나면 예외 벡터가 실행을 하게 된다. 예외 핸들러의 진입시에 레지스터가 갱신된다.

- SP : SP는 Stacking 동안 새로운 주소의 값으로 갱신된다. 인터럽트 서비스 루틴이 실행되는 동안이라면 MSP가 사용된다.
- PSR : IPSR이 새로운 예외 번호로 갱신된다.
- PC : Vector fetch에 따라 Vector handler로 갱신이 되고, 새로운 명령어 패치를 시작한다.
- LR : EXEC_RETURN 이라고 불리는 특수한 값으로 갱신된다.

2.2.5 예외 처리 종료(Exception Exit)

예외 핸들러의 끝에서 일반 실행 루틴을 다시 시작하기 위해서 시스템 상태를 복구시켜야 한다. 몇몇 마이컴에서는 Interrupt Return하기 위해 특별한 명령어를 사용한다 (8051에서는 RETI). Cortex-M3에서는 C함수처럼 동작시키기 위해 일반 명령어가 사용된다.

이 Interrupt Return를 동작시키기 위해서 3가지 방법이 사용된다. 이 방법들 모두는 핸들러 진입시에 LR레지스터에 저장되는 EXEC_RETURN 값을 사용한다.

리턴 명령어	설명
BX reg	EXEC_RETURN 값이 여전히 LR에 저장되어 있다면 BX LR 명령을 사용하여 Interrupt Return를 실행할 수 있다. BX LR ; Return from function call
POP {PC}, or POP {..., PC}	대부분 예외 핸들러 진입시에 스택에 LR의 값을 저장한다. PC에 EXEC_RETURN을 넣기 위해 POP를 사용하게 되면 Interrupt return 를 실행한다. POP <reglist>, PC ; Pop and return
Load (LDR) or Load multiple (LDM)	Destination register로 PC를 사용한 LDR/LDM 명령어를 사용하여 Interrupt return를 실행할 수 있다.

2.3 Core peripherals

2.3.1 Nested Vectored Interrupt Controller(NVIC)

2.3.1.1 NVIC 개요

- 240개 까지의 범위의 인터럽트를 관리한다.
- 각각의 인터럽트는 0 ~ 255레벨의 우선순위를 가질 수 있다. (레벨 값이 작을수록 높은 우선순위를 가진다. 0이 가장 높은 우선순위를 가지게 된다.)
- 동적으로 인터럽트 우선순위 변경 가능.
- 우선순위의 그룹화(group priority and subpriority field)

Address	Name	설명
0xE000E100 ~ 0xE000E11C	NVIC_ISER0 ~ NVIC_ISER7	<i>Interrupt Set-enable Registers</i>
0xE000E180 ~ 0xE000E19C	NVIC_ICER0 ~ NVIC_ICER7	<i>Interrupt Clear-enable Registers</i>
0xE000E200 ~ 0xE000E21C	NVIC_ISPR0 ~ NVIC_ISPR7	<i>Interrupt Set-pending Registers</i>
0xE000E280 ~ 0xE000E29C	NVIC_ICPR0 ~ NVIC_ICPR7	<i>Interrupt Clear-pending Registers</i>
0xE000E300 ~ 0xE000E31C	NVIC_IABR0 ~ NVIC_IABR7	<i>Interrupt Active Bit Registers</i>
0xE000E400 ~ 0xE000E4EF	NVIC_IPR0 ~ NVIC_IPR59	<i>Interrupt Priority Registers</i>
0xE000EF00	STIR	<i>Software Trigger Interrupt Register</i>

2.3.1.2 Interrupt Set-enable Registers (NVIC_ISER)

인터럽트를 활성화 시킨다.

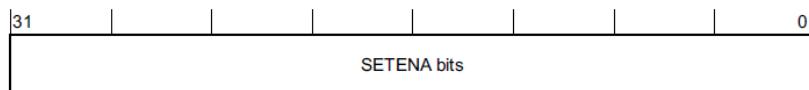


Table 4-4 ISER bit assignments

Bits	Name	Function
[31:0]	SETENA	Interrupt set-enable bits. Write: 0 = no effect 1 = enable interrupt. Read: 0 = interrupt disabled 1 = interrupt enabled.

2.3.1.3 Interrupt Clear-enable Registers (NVIC_ICER)

인터럽트를 비활성화 시킨다.

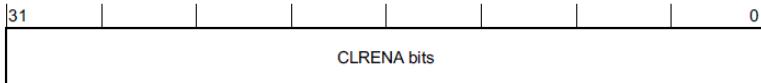


Table 4-5 ICER bit assignments

Bits	Name	Function
[31:0]	CLRENA	Interrupt clear-enable bits. Write: 0 = no effect 1 = disable interrupt. Read: 0 = interrupt disabled 1 = interrupt enabled.

2.3.1.4 Interrupt Set-pending Registers (NVIC_ISPR)

인터럽트 Pending 상태를 강제한다.

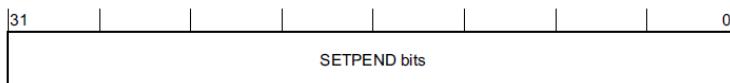


Table 4-6 ISPR bit assignments

Bits	Name	Function
[31:0]	SETPEND	Interrupt set-pending bits. Write: 0 = no effect 1 = changes interrupt state to pending. Read: 0 = interrupt is not pending 1 = interrupt is pending.

2.3.1.5 Interrupt Clear-pending Registers (NVIC_ICPR)

인터럽트 Pending 상태를 제거한다.

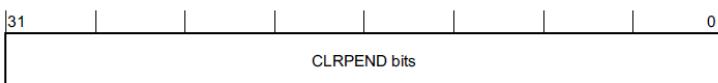


Table 4-7 ICPR bit assignments

Bits	Name	Function
[31:0]	CLRPEND	Interrupt clear-pending bits. Write: 0 = no effect 1 = removes pending state an interrupt. Read: 0 = interrupt is not pending 1 = interrupt is pending.

2.3.1.6 Interrupt Active Bit Registers (NVIC_IABR)

인터럽트가 걸렸는지 표시한다.

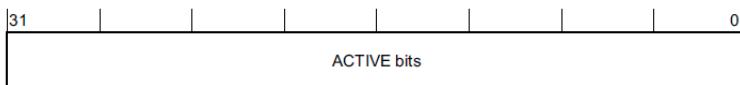
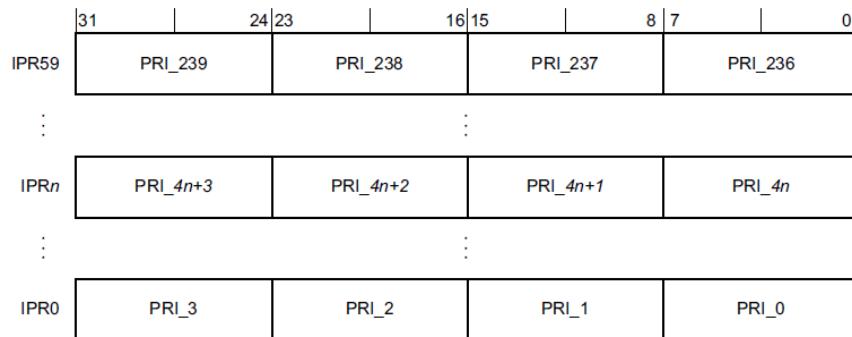


Table 4-8 IABR bit assignments

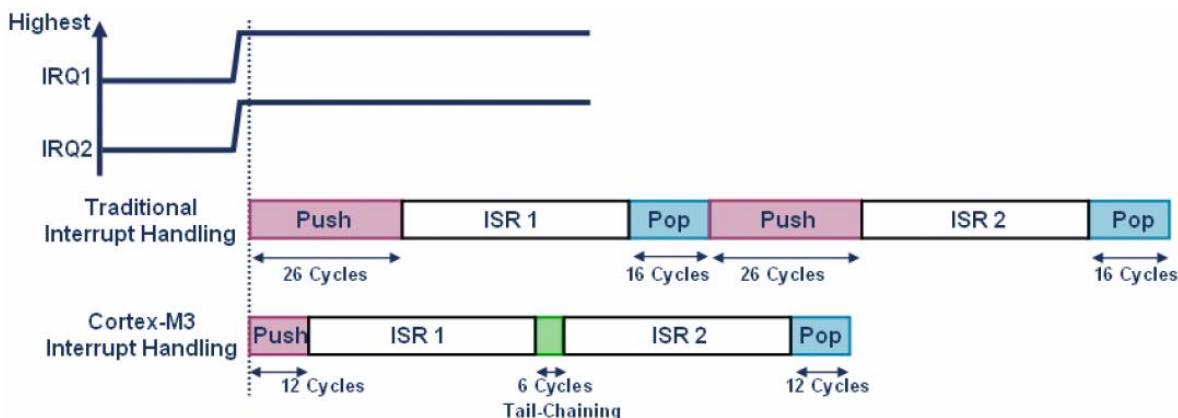
Bits	Name	Function
[31:0]	ACTIVE	Interrupt active flags: 0 = interrupt not active 1 = interrupt active.

2.3.1.7 Interrupt Priority Registers (NVIC_IPR)

인터럽트 우선 순위를 설정한다.



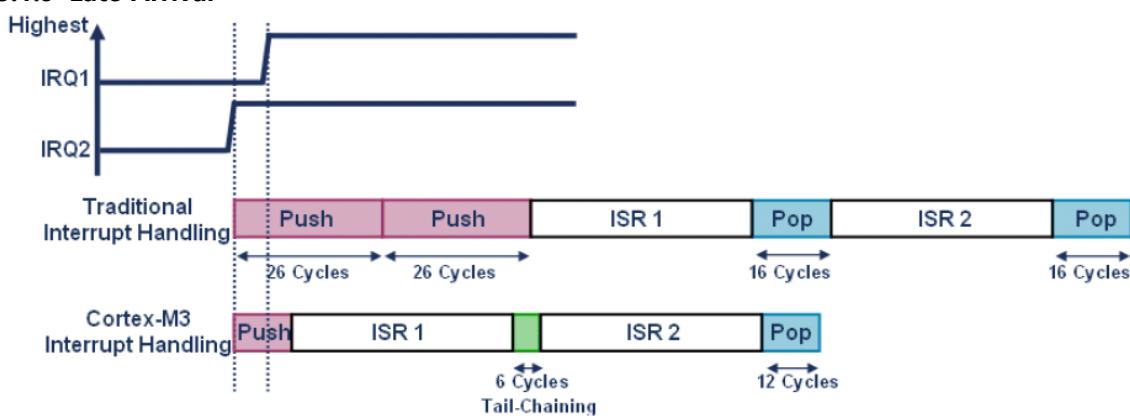
2.3.1.8 Tail Chaining



동시에 여러 개의 인터럽트가 쌓여 있는 경우에 처리하는 방식이다. back-to-back interrupts라고 부를 수 있다. 전통적인 방식에서는 하나의 인터럽트를 처리하면서 Stack push pop 이 2회 발생하게 되지만 tail-chaining 방법을 이용해서 이 횟수를 한 번으로 줄이는 것이다. 사실 연속된 인터럽트의 경우에 있어서 Stack에 저장하는 데이터는 인터럽트가 발생하기 이전의 정상적인 수행 상태가 될 것이다. 인터럽트의 처리를 위해서 Stack 이 저장하고 복구하는 작업을 반복해서 수행해야 할 이유가 없는 것이다.

그림은 동시에 도달한 IRQ1과 IRQ2에 대해서 처리를 하는 것을 시뮬레이션 하고 있다. 전통적인 방식에서 두 인터럽트의 처리 부분에 대한 실행 시간을 제외하고 오버헤드로 붙는 클럭 수가 총 84 cycle이 된다. 하지만 tail-chaining을 사용하는 Cortex-M3에서는 30 cycle로 무려 54 cycle을 절약하게 되는 것이다. 이 tail-chaining 기술은 NVIC hardware 내에 구현되어 있어서 보다 빠르고 쉽게 사용할 수 있도록 해주고 있는 것이다. 100 MHz 이하의 성능을 가지는 micro-processor 시장에서 이 부분은 획기적인 성능 향상을 가져온다.

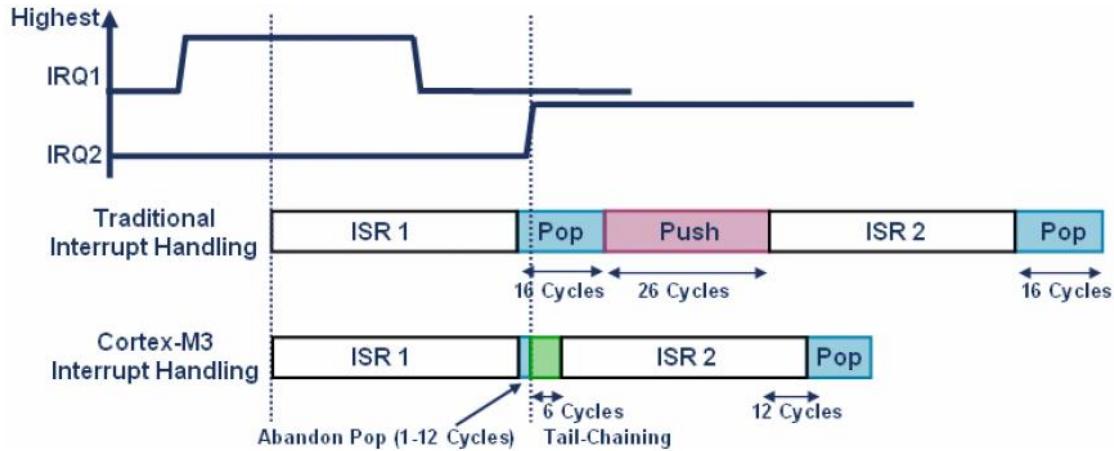
2.3.1.9 Late Arrival



위의 그림의 경우는 보다 높은 우선순위를 가지는 인터럽트가 더 늦게 도착한 경우이다. 보다 낮은 우선순위를 가지는 IRQ2가 먼저 발생을 했고 이를 위해서 현재의 프로세서 상태를 저장하기 위한 Push 작업을 수행하고 있는 동안 (즉, 아직까지 IRQ2의 ISR2가 실행되지 않은 상태인 것이 중요하다) 이때 보다 높은 우선순위를 가진 IRQ1이 발생하게 되면, 이전에 수행하던 Push 작업은 그대로 수행을 하면서 Push 작업이 끝난 이 후에 먼저 발생한 ISR2를 수행하는

것이 아니고 ISR1을 먼저 수행하고, 이 이후의 동작은 위에 설명한 tail-chaining과 같이 ISR2에 대한 수행을 이어서 하게 된다.

2.3.1.10 Pop pre-emption



ISR 동작을 종료한 이후 이전 상태를 복구하기 위한 Pop을 수행하고 있는 시점에 다른 IRQ가 발생하는 경우에 만약 Pop을 종료하지 않은 상황이라면 이전의 Pop이 되고 있던 상황 자체를 무시하고 Stack pointer도 원래의 자리로 옮기고 Pop이 수행되던 것을 무시하고, 새로 들어온 interrupt를 처리하게 된다 이 경우 자연 시간 부분에서 Pop을 수행하던 cycle이 얼마나 진행되고 있었는가에 따라서 추가되는 cycle의 수는 달라질 수 있다.

2.3.2 System Control Block (SCB)

System Control Block (SCB)은 시스템 실행 정보와 시스템 제어를 제공한다 여기에는 system exceptions에 대한 설정, 제어 및 많은 reporting을 가지고 있다. Cortex-M3 SCB registers의 CMSIS mapping 이다.

software 효율을 향상시키기 위해서 CMSIS는 SCB register 표현을 간략화 시켰다. CMSIS에서 byte array SHP[0]부터 SHP[12]는 레지스터 SHPR1-SHPR3에 해당한다.

Table 47. SCB register map and reset values

Offset	Register	31	30	29	28	Implementer	27	26	Variant	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	SCB_CPUID Reset Value	0	1	0	0	PENDSVSET	0	0	PENDSVCLR	0	0	0	1	0	0	0	1	1	1	1	1	0	0	0	1	1	0	0	0	1	0	0	0
0x04	SCB_ICSR Reset Value	NMIPENDSET 0	Reser ved	0	0	PENDSVSET 0	0	0	PENDSVCLR 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0x08	SCB_VTOR Reset Value	Reser ved	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0x0C	SCB_AIRCR Reset Value	VECTKEY[15:0]												ENDIANESS 0	Reserved	PRIGROUP[2:0] 0 0 0	Reserved	SYSRESETREQ 0 0 0 0 0 0 0 0 0 0 0 0												VECTACTIVE 0	VECTRESET 0		
0x10	SCB_SCR Reset Value	Reserved																															
0x14	SCB_CCR Reset Value	Reserved																		STKALIGN 1 0	BFHFIGN 0	Reserved	DIV_0_TRP 0 0	UNALIGN_TRP 0 0	SLEEPDEEP 0 0	SEVONPEND 0	Reserved	SYSRESETREQ 0 0 0 0 0 0 0 0 0 0 0 0	VECTACTIVE 0 0 0 0 0 0 0 0 0 0 0 0	VECTRESET 0 0 0 0 0 0 0 0 0 0 0 0			
0x18	SCB_SHPR1 Reset Value	Reserved				PRI6 0 0 0 0 0 0	0	0	PRI5 0 0 0 0 0 0	0	0	0	0	0	0	0	PRI4 0 0 0 0 0 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 47. SCB register map and reset values (continued)

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1C	SCB_SHPR2																																
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x20	SCB_SHPR3																																
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x24	SCB_SHCRS																																
	Reset Value																																
0x28	SCB_CFSR																																
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x2C	SCB_HFSR																																
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x34	SCB_MMAR																																
	Reset Value	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
0x38	SCB_BFAR																																
	Reset Value	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		

2.3.2.1 CPUID base register(SCB_CPUID)

CPUID 레지스터는 프로세서에 대한 정보를 가지고 있다.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementer								Variant							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PartNo								Revision							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

비트	레지스터 이름	내 용
31:24	Implementer	시행사 코드 0x41: ARM
23:20	Variant	제품 개정 번호(rnnpn)에서 r값을 표시
19:16	Constant	0x0F 으로 고정됨
15:4	PartNo	부품 번호(Part Number) 0xC23 = Cortex-M3
3:0	Revision	제품 개정 번호(rnnpn)에서 p값을 표시

2.3.2.2 Interrupt control and state register (SCB_ICSR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
NMIE NDSET	Reserved		PEND SVSET	PEND SVCLR	PEND STSET	PENDS TCLR	Reserved			ISRPEN DING	VECTPENDING[9:4]					
			rw	w	rw	w				r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
VECTPENDING[3:0]				RETOB ASE	Reserved			VECTACTIVE[8:0]								
								rw	rw	rw	rw	rw	rw	rw	rw	

비트	레지스터 이름	내 용
31	NMIPENDSET	<p>NMI 팬딩 비트를 Set</p> <p>쓰기 :</p> <p>0 : 영향 없음</p> <p>1 : NMI 예외 상태를 팬딩으로 변경.</p> <p>읽기 :</p> <p>0 : NMI 팬딩 없음.</p> <p>1 : NMI 팬딩 있음.</p>
30:29	Reserved	<p>PendSV 팬팅 비트를 Set</p> <p>쓰기 :</p> <p>0 : 영향 없음</p> <p>1 : PendSV 예외 상태를 팬딩으로 변경.</p> <p>읽기 :</p> <p>0 : PendSV 팬딩 없음.</p> <p>1 : PendSV 팬딩 있음.</p>
27	PENDSVCLR	<p>PendSV 팬팅 비트를 Clear</p> <p>0 : 영향 없음</p> <p>1 : PendSV 예외 상태를 Clear한다.</p>
26	PENDSTSET:	<p>SysTick 팬팅 비트를 Set</p> <p>쓰기 :</p> <p>0 : 영향 없음</p> <p>1 : SysTick 예외 상태를 팬딩으로 변경.</p> <p>읽기 :</p> <p>0 : SysTick 팬팅 없음.</p> <p>1 : SysTick 팬팅 있음.</p>
25	PENDSTCLR:	<p>SysTick 팬팅 비트를 Clear</p> <p>0 : 영향 없음</p> <p>1 : SysTick 예외 상태를 Clear한다.</p>
24	Reserved	
23	Reserved	
22	ISRPENDING	<p>인터럽트 팬팅 플래그 (NMI와 Faults는 제외)</p> <p>0 : 팬딩 되지 않음.</p> <p>1 : 팬딩 됨</p>
21:12	VECTPENDING[9:0]	<p>활성화된 예외 중에서 가장 우선순위가 높은 팬딩된 예외의 번호</p> <p>0 : 팬딩된 예외가 없음.</p> <p>그 외 : 활성화된 예외 중에서 가장 우선순위가 높은 팬딩된 예외의 번호</p>

11	RETOBASE:	0 : 선점된 활성 예외가 있음. 1: 활성화된 예외가 없음.
10:9	Reserved	활성화된 예외 번호
8:0	VECTACTIVE[8:0]	0 : 스레드 모드 그 외 : 현재 활성 예외의 예외 번호

2.3.2.3 Vector table offset register (SCB_VTOR)

벡터 테이블의 오프셋을 결정한다.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		TBLOFF[29:16]													
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TBLOFF[15:9]								Reserved							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

비트	레지스터 이름	내 용
31:30	Reserved	
29:11	TBLOFF[29:9]	<p>이 필드는 메모리 0x00000000에서부터 벡터 테이블(Vector Table)의 오프셋을 나타낸다. 최소 정렬(Alignment)은 128 word가 되어야 한다.</p> <p>비트29는 벡터 테이블의 위치가 Code 또는 SRAM 영역인지를 결정한다.</p> <p>0 : Code 1 : SRAM</p>
10:0	Reserved	

2.3.2.4 Application interrupt and reset control register (SCB_AIRCR)

예외 모델에서의 우선순위 그룹을 제어한다.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
VECTKEYSTAT[15:0](read)/ VECTKEY[15:0](write)																
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ENDIANESS	Reserved				PRIGROUP			Reserved						SYS RESET REQ	VECT CLR ACTIVE	VECT RESET
r					rw	rw	rw							w	w	w

비트	레지스터 이름	내 용
31:16	VECTKEYSTAT[15:0]/ VECTKEY[15:0]	<p>읽기 : 0xFA05로 항상 읽어진다.</p> <p>쓰기 : 이 레지스터에 쓰기 위해서는 이 필드에 VECTKEY인 0x5FA를 넣어 주어야 한다. 그렇지 않으면 쓰기 동작은 무시된다.</p>

15	ENDIANESS	데이터의 Endian 설정 0 : Little-endian
14:11	Reserved	
10:8	PRIGROUP[2:0]	인터럽트 그룹 설정
7:3	Reserved	
2	SYSRESETREQ	System reset 요청
1	VECTCLRACTIVE	디버그 용도로 예약되어 있음.
0	VECTRESET	디버그 용도로 예약되어 있음

■ Binary point

Table 45. Priority grouping

PRIGROUP [2:0]	Interrupt priority level value, PRI_N[7:4]			Number of	
	Binary point ⁽¹⁾	Group priority bits	Subpriority bits	Group priorities	Sub priorities
0b011	0bxxxx	[7:4]	None	16	None
0b100	0bxxx.y	[7:5]	[4]	8	2
0b101	0bxx.yy	[7:6]	[5:4]	4	4
0b110	0bx.yyy	[7]	[6:4]	2	8
0b111	0b.yyyy	None	[7:4]	None	16

1. PRI_n[7:4] field showing the binary point. x denotes a group priority field bit, and y denotes a subpriority field bit.

2.3.2.5 System control register (SCB_SCR)

저전력상태(Low power state)로 들어가고 나가는 방법을 설정한다.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															
										SEVONPEND	Res.	SLEEPDEEP	SLEEPONEXIT	Res.	
										rw		rw	rw		

비트	레지스터 이름	내 용
31:5	Reserved	저전력상태(Low power state)로 들어가고 나가는 방법을 설정한다.
4	SEVEONPEND	0 : 활성화된 인터럽트만 프로세서를 깨울 수 있다. 1 : 모든 이벤트/인터럽트가 프로세서를 깨울 수 있다.(비활성화된 인터럽트 포함)
3	Reserved	Sleep 모드를 설정한다.
2	SLEEPDEEP	0 : Sleep 1 : Deep sleep
1	SLEEPONEXIT	핸들러 모드에서 스레드 모드로 돌아갈 때 Sleep 모드로 갈

것인지 설정한다.

0 : Sleep 모드로 돌아가지 않는다.

1 : Sleep 모드로 돌아간다.

0	Reserved
----------	-----------------

2.3.2.6 Configuration and control register (SCB_CCR)

예외 모델에서의 우선순위 그룹을 제어한다.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved					STK ALIGN	BFHF NMIGN	Reserved			DIV_0 TRP	UN ALIGN TRP	Res.		USER SET MPEND	NON BASE THRD ENA
					rw	rw				rw	rw			rw	rw

비트	레지스터 이름	내 용
31:10	Reserved	저전력상태(Low power state)로 들어가고 나가는 방법을 설정한다.
9	STKALIGN	0 : 활성화된 인터럽트만 프로세서를 깨울 수 있다. 1 : 모든 이벤트/인터럽트가 프로세서를 깨울 수 있다.(비활성화된 인터럽트 포함)
8	BFHFNMIGN	0 : Load/Store 명령에 의해 Data bus fault가 생기는 것을 허용. 1 : 우선순위가 -1/-2인 핸들러가 실행 중일 때 Load/Store 명령어에 의해 Data bus fault가 발생하면 무시한다.
7:5	Reserved	
4	DIV_0_TRP	0 : 0으로 나눌 때 trap 하지 않는다. 1 : 0을 나눌 때 trap 한다.
3	UNALIGN_TRP	0 : 정렬(unalign)이 안된 halfword/word 접근을 trap하지 않는다. 1 : 정렬(unalign)이 안된 halfword/word 접근을 trap한다.
2	Reserved	
1	USERSETMPEND	0 : STIR(Software trigger interrupt register)에 대한 권한이 없는 소프트웨어 접근 금지. 1 : STIR(Software trigger interrupt register)에 대한 권한이 없는 소프트웨어 접근 허용.
0	NONBASETHRDENA	0 : 프로세서는 예외가 활성화되지 않았을 때만 스레드 모드로 들어갈 수 있다. 1 : EXC_RETURN 값에 따라 어떤 레벨에서도 스레드 모드로 들어갈 수 있다.

2.3.2.7 System handler priority registers (SHPRx)

SHPR1 ~ SHPR3 레지스터는 시스템 핸들러에 대한 우선순위(0 ~ 15)를 설정한다.

■ System handler priority register 1 (SCB_SHPR1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								PRI_6[7:4]				PRI_6[3:0]			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PRI_5[7:4]				PRI_5[3:0]				PRI_4[7:4]				PRI_4[3:0]			
rw	rw	rw	rw	r	r	r	r	rw	rw	rw	rw	r	r	r	r

비트	레지스터 이름	내 용
31:24	Reserved	
23:16	PRI_6[7:0]	시스템 핸들러 6(Usage fault)에 대한 우선 순위
15:8	PRI_5[7:0]	시스템 핸들러 5(Bus fault)에 대한 우선 순위
7:0	PRI_4[7:0]	시스템 핸들러 4(Memory management fault)에 대한 우선 순위

■ System handler priority register 2 (SCB_SHPR2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PRI_11[7:4]				PRI_11[3:0]				Reserved							
rw	rw	rw	rw	r	r	r	r	15	14	13	12	11	10	9	8
Reserved								7	6	5	4	3	2	1	0

비트	레지스터 이름	내 용
31:24	PRI_11[7:0]	시스템 핸들러 11(SVCall)에 대한 우선 순위
23:0	Reserved	

■ System handler priority register 3 (SCB_SHPR3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PRI_15[7:4]				PRI_15[3:0]				PRI_14[7:4]				PRI_14[3:0]			
rw	rw	rw	rw	r	r	r	r	rw	rw	rw	rw	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

비트	레지스터 이름	내 용
31:24	Reserved	
23:16	PRI_15[7:0]	시스템 핸들러 15(SysTick exception)에 대한 우선 순위
15:0	PRI_14[7:0]	시스템 핸들러 14(PendSV)에 대한 우선 순위

2.3.2.8 System handler control and state register (SCB_SHCSR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Reserved														USG FAULT ENA	BUS FAULT ENA	MEM FAULT ENA
														rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
SV CALL PEND ED	BUS FAULT PEND ED	MEM FAULT PEND ED	USG FAULT PEND ED	SYS TICK ACT	PEND SV ACT	Res.	MONIT OR ACT	SV CALL ACT	Reserved		USG FAULT ACT	Res.	BUS FAULT ACT	MEM FAULT ACT		
rw	rw	rw	rw	rw	rw		rw	rw			rw		rw		rw	

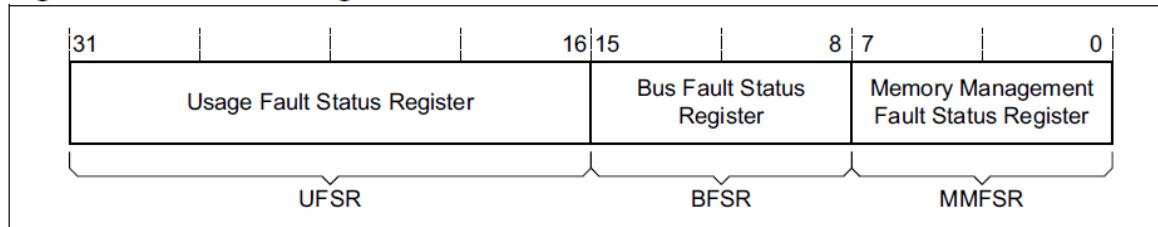
비트	레지스터 이름	내 용
31:19	Reserved	
18	USGFAULTENA	Usage fault enable 0 : Disable 1 : Enable
17	BUSFAULTENA	Bus fault enable 0 : Disable 1 : Enable
16	MEMFAULTENA	Memory management fault enable 0 : Disable 1 : Enable
15	SVCALLPENDED	SVC call pending bit
14	BUSFAULTPENDED	Bus fault exception pending bit
13	MEMFAULTPENDED	Memory management fault exception pending bit
12	USGFAULTPENDED	Usage fault exception pending bit
11	SYSTICKACT	SysTick exception active bit
10	PENDSVACT	PendSV exception active bit
9	Reserved	
8	MONITORACT	Debug monitor active bit
7	SVCALLACT	SVC call active bit
6:4	Reserved	
3	USGFAULTACT	Usage fault exception active bit
2	Reserved	
1	BUSFAULTACT	Bus fault exception active bit
0	MEMFAULTACT	Memory management fault exception active bit

2.3.2.9 Configurable fault status register (SCB_CFSR)

CFSR 은 바이트 접근이 가능한다.

- 0xE000ED28에 word 접근하면 전체 CFSR에 접근하고,
- 0xE000ED28에 halfword 접근하면 MMFSR과 BFSR에 접근하고,
- 0xE000ED28에 byte 접근하면 전체 MMFSR에 접근하게 된다.

Figure 20. CFSR subregisters



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				DIVBYZERO	UNALIGNED	Reserved				NOCP	INVPC	INVSTATE	UNDEFINSTR		
		rc_w1	rc_w1							rc_w1	rc_w1	rc_w1	rc_w1		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BFARVALID ALID	Reserved	STKERR	UNSTKERR	IMPRECISERR	PRECISERR	IBUSERR	MMARVALID	Reserved	MSTKERR	MUNSTKERR	Res.	DACCVIOL	IACCVIOL		
		rw	rw	rw	rw	rw	rw		rw	rw		rw	rw		

비트	레지스터 이름	내 용
31:26	Reserved	
25	DIVBYZERO	Divide by zero usage fault 0 : 안 발생 1 : 발생
24	UNALIGNED	Unaligned access usage fault 0 : 안 발생 1 : 발생
23:20	Reserved	
19	NOCP	No coprocessor usage fault (지원하지 않는 coprocessor 명령어 실행시) 0 : 안 발생 1 : 발생
18	INVPC	Invalid PC load usage fault (EXC_RETURN 에 의해 로드된 PC 값이 유효하지 않은 경우) 0 : 안 발생 1 : 발생
17	INVSTATE	Invalid state usage fault: 0 : 안 발생 1 : 발생
16	UNDEFINSTR	정의되지 않은 명령어 사용시 0 : 안 발생 1 : 발생
15	BFARVALID	유효한 주소에서 Bus Fault가 발생함. 0 : 안 발생 1 : 발생
14:13	Reserved	PendSV exception active bit
12	STKERR	Stacking 시에 Bus fault 발생 0 : 안 발생 1 : 발생
11	UNSTKERR	Unstacking 시에 Bus fault 발생 0 : 안 발생 1 : 발생
10	IMPRECISERR	알 수 없는 Bus fault 에러 발생 0 : 안 발생 1 : 발생
9	PRECISERR	Precise data bus error

		0 : 안 발생 1 : 발생
8	IBUSERR	Instruction bus error 0 : 안 발생 1 : 발생
7	MMARVALID	Memory Management Fault Address Register (MMAR)가 유효한지 표시 0 : 유효하지 않음 1 : 유효
6:5	Reserved	
4	MSTKERR	Stacking 중에 Memory manager fault 발생 0 : 안 발생 1 : 발생
3	MUNSTKERR	Unstacking 중에 Memory manager fault 발생 0 : 안 발생 1 : 발생
2	Reserved	
1	DACCVIOL	데이터 접근 위반 플래그 0 : 없음 1 : 데이터 접근 위반
0	IACCVIOL	명령어 접근 위반 플래그 0 : 없음 1 : 명령어 접근 위반

2.3.2.10 Hard fault status register (SCB_HFSR)

hard fault를 발생시킨 이벤트에 대한 정보를 준다.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DEBU G_VT	FORC ED														
rc_w1	rc_w1														
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
														VECT TBL	Res.
														rc_w1	

비트	레지스터 이름	내 용
31	DEBUG_VT	디버그용으로 예약됨
30	FORCED	강제된 Hard fault 0 : 없음 1 : 강제된 Hard fault
29:2	Reserved	Unaligned access usage fault 0 : 안 발생 1 : 발생
1	VECTTBL	Vector table hard fault 0 : 없음

1 : Vector table 읽는 과정에 fault 발생															
0	Reserved														

2.3.2.11 Memory management fault address register (SCB_MMFAR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MMFAR[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MMFAR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

비트	레지스터 이름	내 용
31:0	MMFAR[31:0]	Memory management fault 가 발생한 주소

2.3.2.12 Bus fault address register (SCB_BFAR)

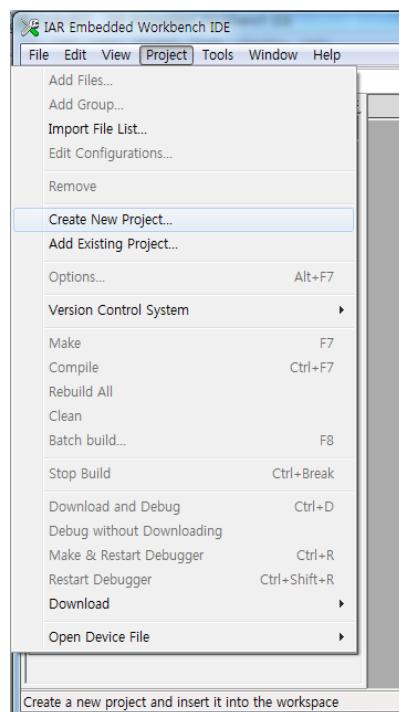
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BFAR[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BFAR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

비트	레지스터 이름	내 용
31:0	BFAR[31:0]	Bus fault address가 발생한 주소

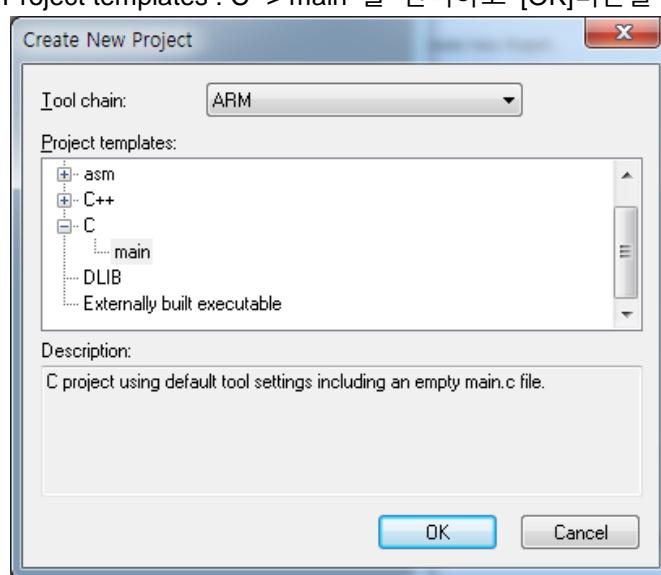
3. 프로젝트 구성

3.1 프로젝트 생성

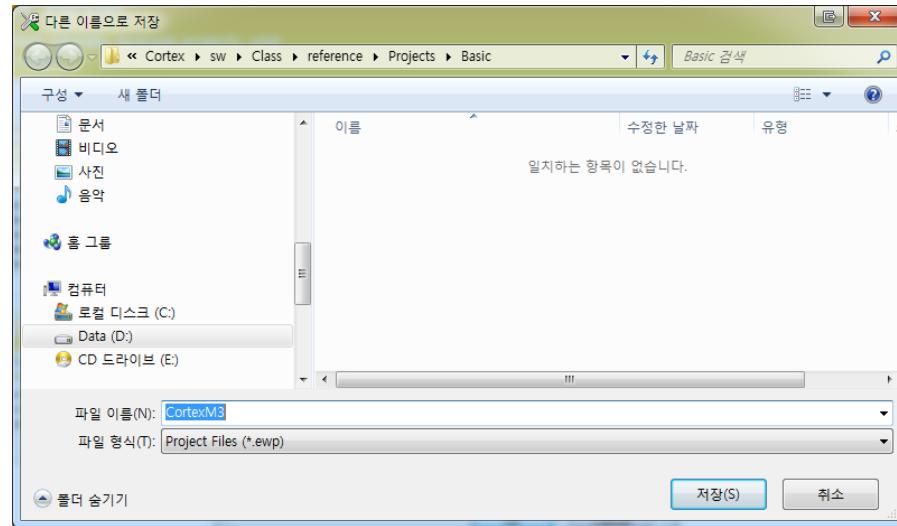
- 1) 새로운 프로젝트를 생성하기 위해 상단의 메뉴에서 [Project] -> [Create New Project] 를 선택한다.



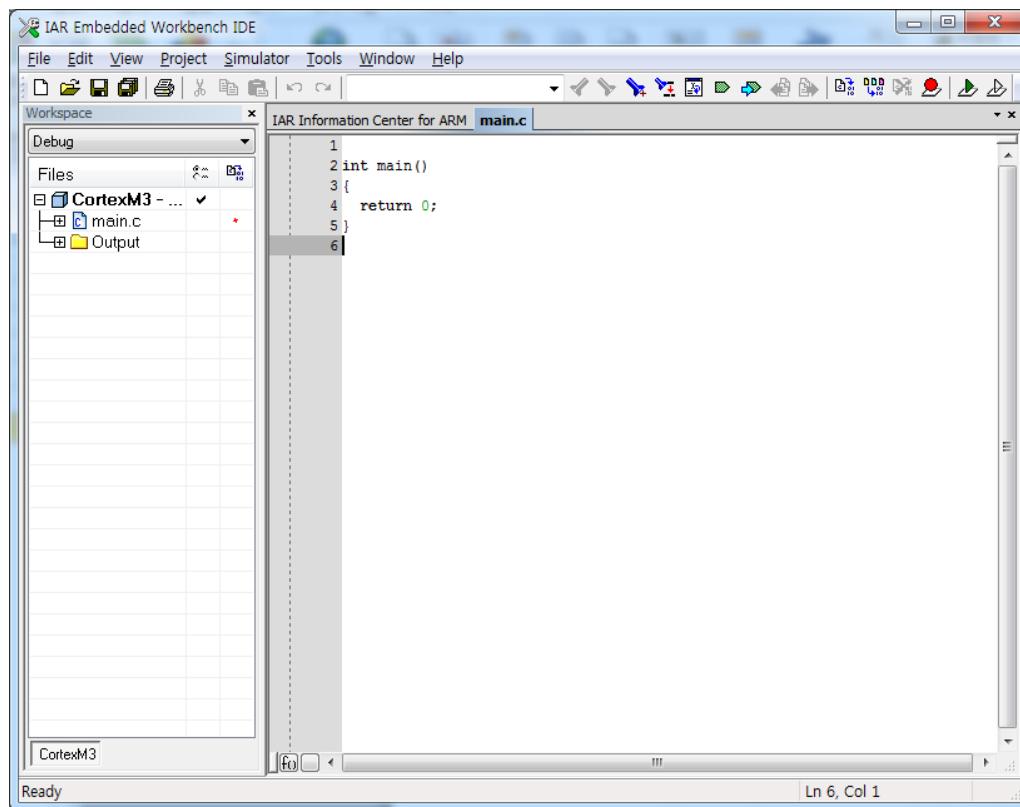
- 2) Tool chain : ARM, Project templates : C -> main 를 선택하고 [OK]버튼을 누른다.



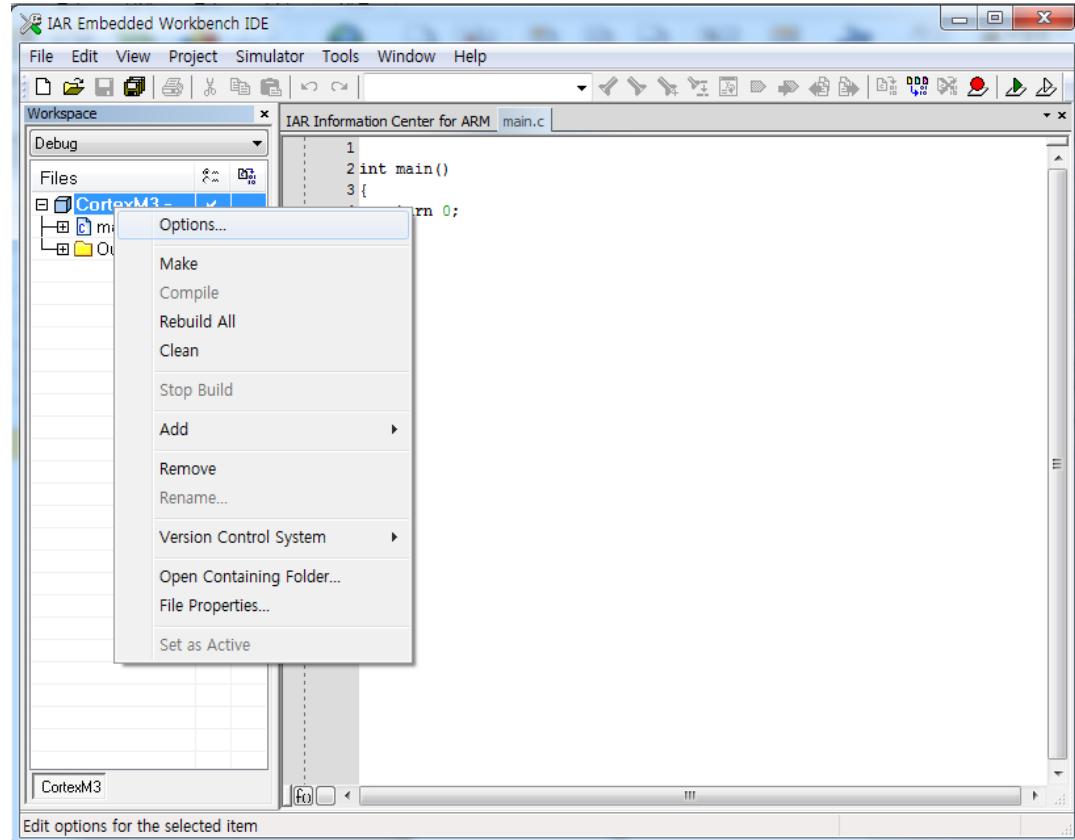
3) 프로젝트가 생성될 폴더 경로로 이동한 다음 프로젝트 파일명을 적어 준 다음 [저장]을 누른다.



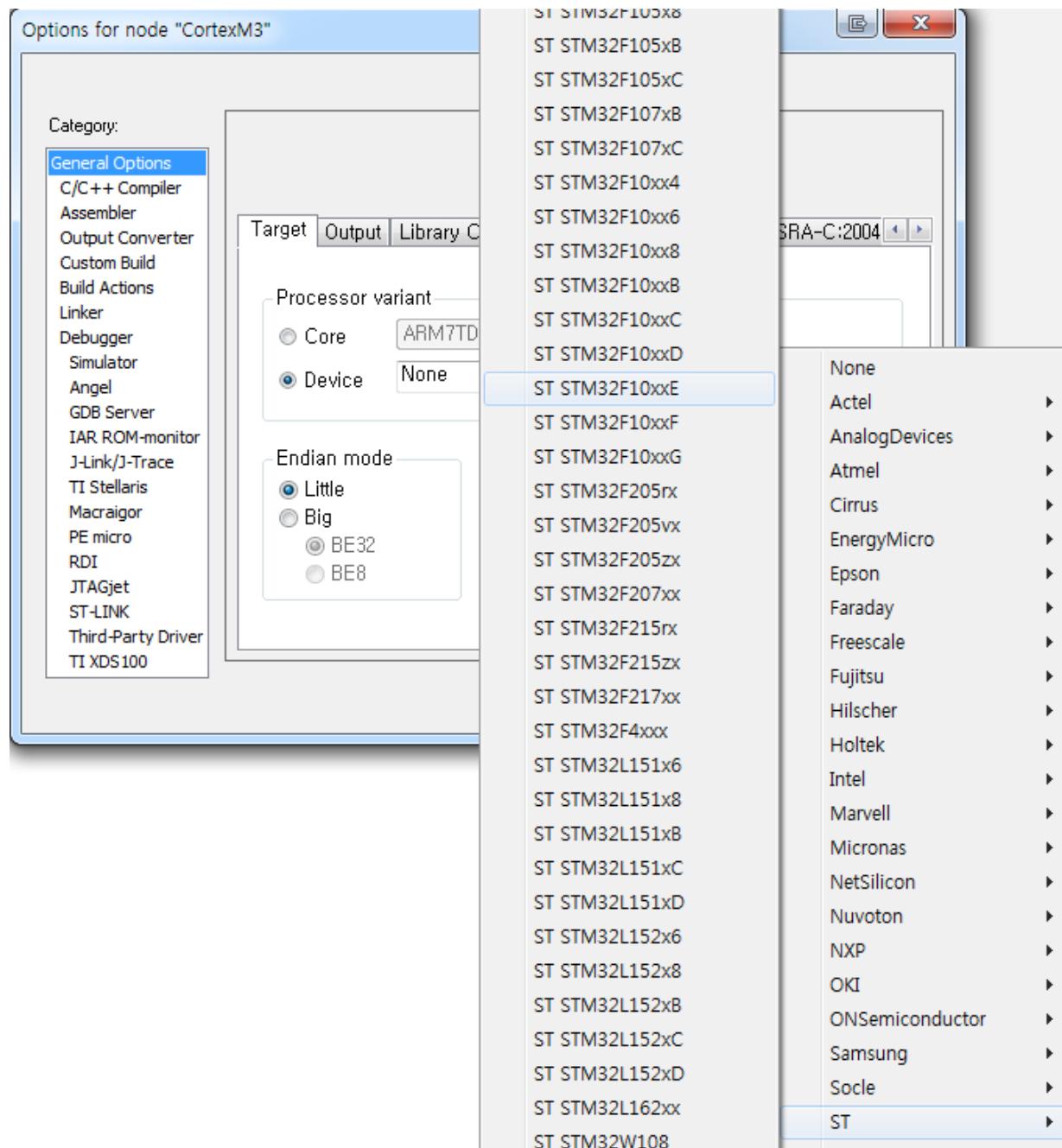
4) 프로젝트가 생성이 되고 오른쪽 화면에 main.c 파일의 내용이 보이게 된다.



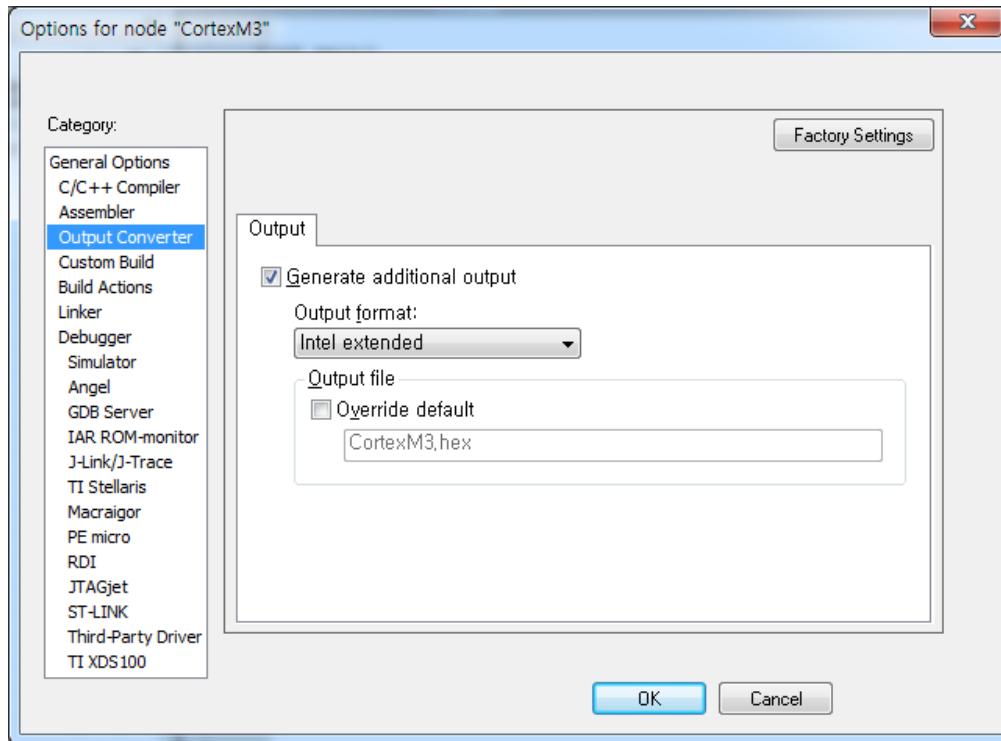
- 5) 프로젝트가 생성이 되었으면 컴파일을 어떻게 할 것인지에 대한 설정이 필요하다.
왼쪽의 Workspace에서 아래와 같이 팝업 메뉴를 실행시켜서 [Options]를 실행시킨다.



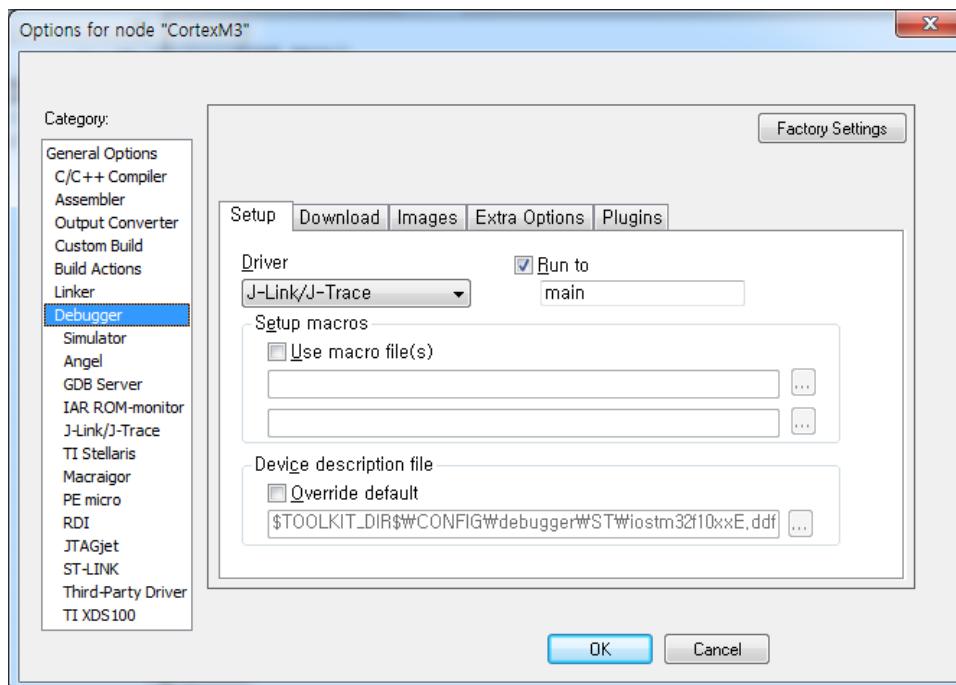
- 6) [General Options] -> [Target] -> [Device]에서 사용하고자 하는 디바이스를 선택한다.
 우리가 사용하는 칩은 ST사의 STM32F103ZET 이므로 [ST STM32F10xxE]를 선택한다.



7) Hexa 파일을 생성하기 위해서 [Output Converter]에서 아래 그림과 같이 설정한다.



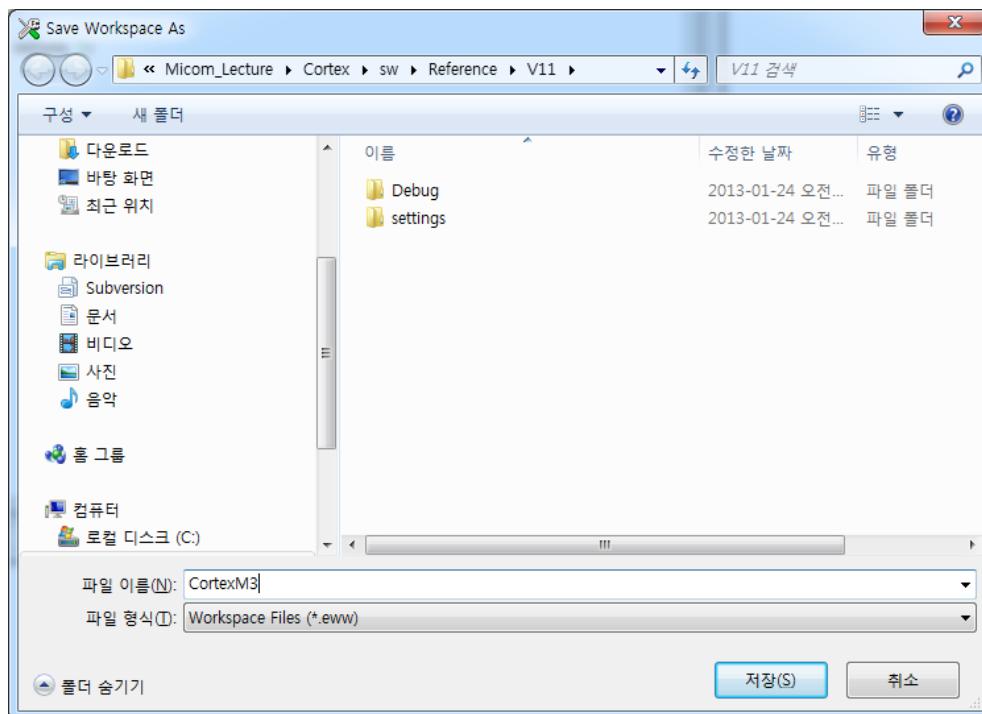
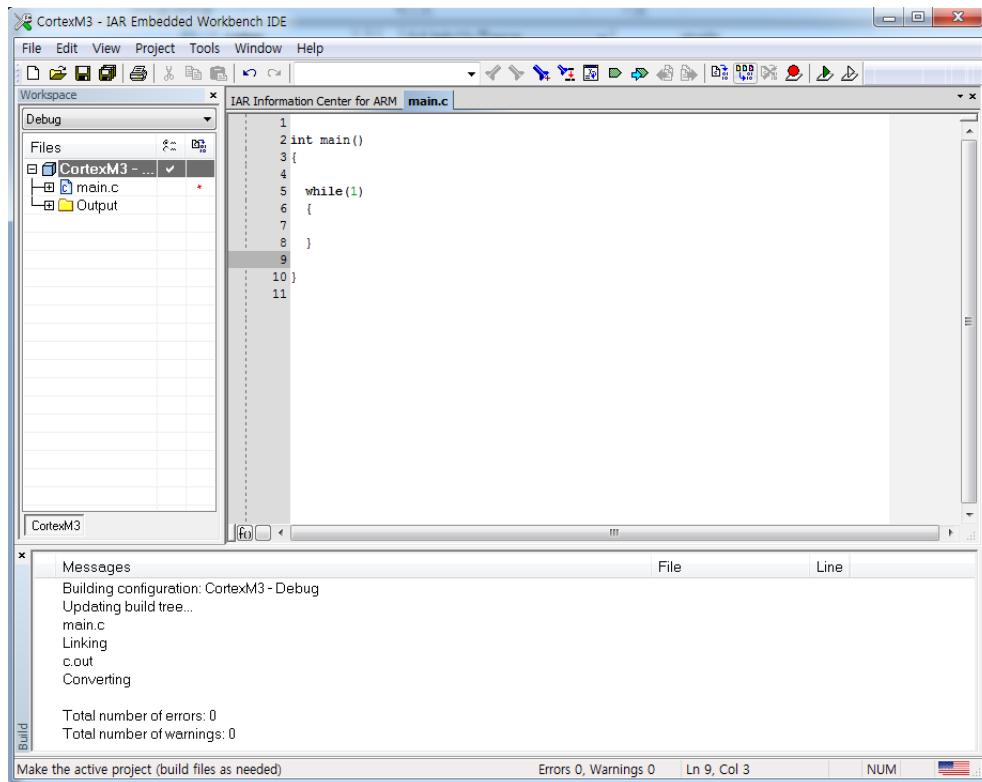
8) Debugger 장비로는 J-Link를 사용하고 있으므로 [Debugger] -> [J-Link/J-Trace] 선택한다.



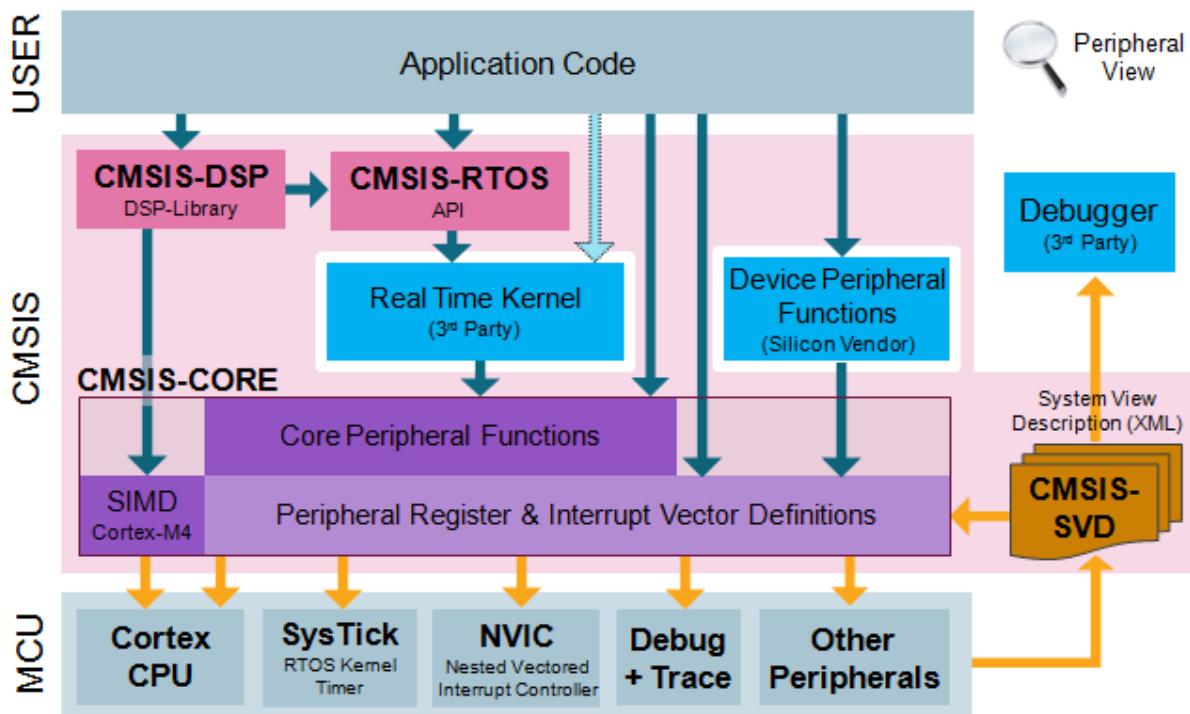
9) 기본 설정이 끝났으므로 하단의 [OK] 버튼을 누르고 설정화면을 빠져 나온다.

10) 컴파일이 정상적으로 되는지 확인하기 위해서 아래와 같이 코딩을 하고 화면 상단의 [Make] 아이콘을 눌러 Make를 실행시킨다. 그러면 workspace 파일을 저장하라는 대화 상자가 나오고 여기에 적당한 이름의 파일명을 적어 준 다음 [저장] 버튼을 눌러 주면 컴파일이

정상적으로 되었다는 메시지가 화면 하단에 표시된다.



3.2 CMSIS(Cortex Microcontroller Software Interface Standard)



CMSIS는 ARM 사에서 제시하고 있는 표준입니다 물론 권고 사항이고 이를 반드시 따라야 하는 규정은 아닙니다. 하지만 특별한 이유가 있지 않은 한 칩제조사의 입장에서 이를 따르지 않을 이유는 사실 별로 없습니다. 자사의 집을 많이 판매하는 것을 목적으로 하고 있는 것이고 이를 위해서 그다지 어려운 것도 아니고 다만 소프트웨어의 수정만 조금 하면 되는 것을 회사들이 마다할 이유는 없는 것입니다

ARM사는 자사의 M3 Core를 사용하는 서로 다른 반도체 회사들 간의 Firmware의 호환성을 높이기 위해서 CMSIS (Cortex Microcontroller Software Interface Standard)를 발표하게 됩니다. 현재 모든 칩 제조사들은 모든 예제 코드들을 CMSIS 에 맞춰서 제공하고 있습니다. 결국 사용자의 입장에서 서로 다른 회사의 집을 사용하더라도 내가 만든 소스 코드를 보다 많이 재활용할 수 있는 길이 열리는 것이고 무척이나 고무적인 일이라고 할 수 있습니다.

Cortex-M3의 Core 설계에도 이러한 개념은 많이 포함이 되어 있습니다. Memory Map에 있어서도 하드웨어적으로 시켜 놓아서 서로 다른 칩 제조사간의 호환성을 높이려는 시도를 하드웨어적으로도 추진하고 있는 것입니다.

3.2.1 동기

CMSIS는 다음과 같은 것을 목표로 만들어 졌습니다.

- **일관된 소프트웨어 인터페이스로 인해 소프트웨어의 이식성 및 재사용이 향상시킨다.**
일반적인 소프트웨어 라이브러리는 다양한 반도체 회사에서 제공하는 디바이스 라이브러리와 인터페이스 할 수 있다.
- **개발 기간을 단축시킨다.**
표준화된 소프트웨어를 사용을 통해 쉽고 빠르게 소프트웨어를 생성한다.

- 컴파일러 독립적인 층을 제공한다.
CMSIS는 주로 사용되는 컴파일러(ARMCC, IAR, GNU...)에 의해 지원이 된다.
- 주변 장치 정보를 통해 향상된 프로그램 디버깅을 제공한다.

3.2.2 코딩 규칙

CMSIS는 다음과 같은 필수 코딩 규칙을 사용한다.

- ANSI C/C++ 를 따른다.
- <stdint.h>에 정의에 ANSI C standard data type를 사용한다.
- 변수와 매개 변수는 완전한 데이터 형을 가진다.
- #define 상수 표현은 괄호 안에 넣어서 사용한다.
- MISRA 2004 따른다.

다음은 권고 코딩 규칙이다.

- Core Registers, Peripheral Registers, CPU Instructions 는 대문자를 사용한다.
- 함수 이름과 인터럽트 함수 이름은 CamelCase 규칙을 따른다.

CamelCase xxxxYyyy

PascalCase XxxxYyyy

- Namespace_prefix를 사용하여 식별자 간의 충돌을 피하고 함수 그룹화를 제공한다.

3.3 라이브러리 설치

ST사에서는 Cortex 마이컴 개발을 위해서 수 많은 소프트웨어 코드를 제공하고 있다. 그 중에서 가장 기본이 되는 라이브러리를 받아서 사용하도록 한다.

3.3.1 라이브러리 다운로드

1) 라이브러리 다운로드

<http://www.st.com/stonline/stappl/productcatalog/app?page=partNumberSearchPage&levelid=SS1576&parentid=1743&resourcetype=SW>

에 들어가면

STM32F10x, STM32L1xx and STM32F3xx USB full speed device library (UM0424)

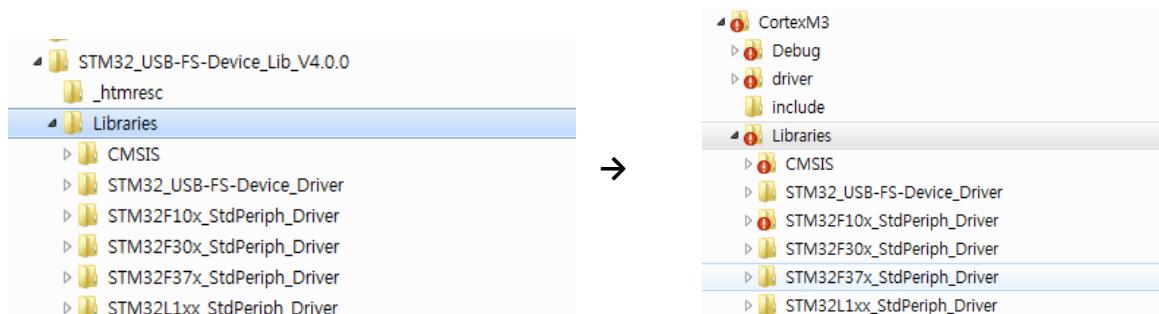
이라고 되어 있는 부분이 있다.

Part Number Search			
STM32 Firmware : 39			
Part Number	Orderable Part Number(s)	Marketing Status	Description
HCC-FFS	HCC-FFS	Active	All Flash File Systems with fail-safe mechanisms for STM32, from HCC Embedded
HCC-MISRA-TCP/IP	HCC-MISRA-TCP/IP	Active	TCP/IP v4 or v6 stack for STM32, from HCC Embedded
HCC-USB	HCC-USB	Active	USB Host & Device stacks for STM32, from HCC Embedded
Men-Nucleus-SF	Men-Nucleus-SF	Active	Nucleus SmartFit for STM32
STSW-MCU005	STSW-MCU005	Active	STM32 and STM8 Flash loader demonstrator (UM0462)
STSW-STM32013	STSW-STM32013	Active	STM32F10xxx LCD glass driver firmware (AN2656)
STSW-STM32017	STSW-STM32017	Active	STM32F10xxx Speex library firmware STM32_StdPeriph Lib, speex, audio (AN2812)
STSW-STM32025	STSW-STM32025	Active	Managing the Driver Enable signal for RS-485 and IO-Link communications with the STM32 USART
STSW-STM32028	STSW-STM32028	Active	STM32's ADC modes and their applications (AN3116)
STSW-STM32047	STSW-STM32047	Active	Implementing receivers for infrared remote control protocols using STM32F1 microcontrollers (AN3174)
STSW-STM32094	STSW-STM32094	Active	STM32 in-application programming over the I2C bus (AN3078)
STSW-STM32098	STSW-STM32098	Active	STM32 embedded GUI library (AN3128)
STSW-STM32100	STSW-STM32100	Active	STM32 PMSM FOC SDK motor control firmware library (UM1052)
STSW-STM32108	STSW-STM32108	Active	STM32F30x/31x DSP and standard peripherals library, including 81 examples for 25 different peripherals and template project for 5 different IDEs (UM1581)
STSW-STM32111	STSW-STM32111	Active	STM32F3xx in-application programming (IAP) using the USART (AN4045)
STSW-STM32112	STSW-STM32112	Active	EEPROM emulation in STM32F3xx microcontrollers (AN4046)
STSW-STM32115	STSW-STM32115	Active	STM32F37x/38x DSP and standard peripherals library, including 73 examples for 26 different peripherals and template project for 5 different IDEs (UM1585)
STSW-STM32121	STSW-STM32121	Active	STM32F10x, STM32L1xx and STM32F3xx USB full speed device library (UM0424)
STSW-STM32126	STSW-STM32126	Active	I2C timing configuration tool for STM32F3xx and STM32F0xx microcontrollers (AN4235)
STSW-STM32AN4187	STSW-STM32AN4187	Active	Using CRC peripheral in STM32 family (AN4187)
TAP-KNX-KAISStack	TAP-KNX-KAISStack	Active	KNX home automation for STM8 and STM32, from Tapko Technologies GMBH

위의 설명한 부분을 선택하면 아래 그림과 같은 창이 뜬다. 여기서 우측 하단의 [Download] 버튼을 눌러 라이브러리 파일을 받는다.

Part Number	Version	Marketing Status	Order From ST
STSW-STM32121	4.0.0	Active	Download

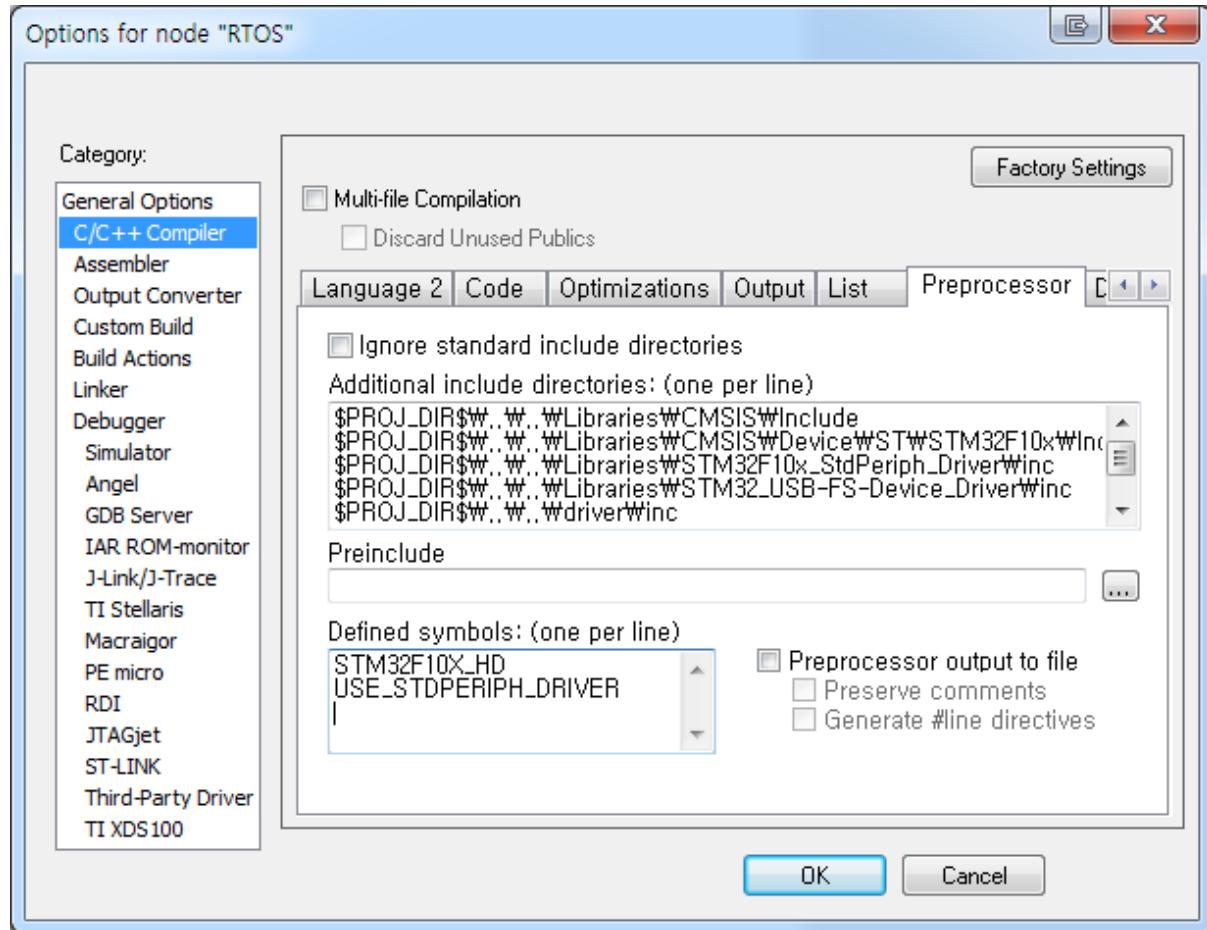
2) 다운로드한 파일을 압축해제하고 필요한 라이브러리가 있는 Libraries 폴더만 복사해서 프로젝트 파일에 복사한다.



3) stm32f10x_conf.h 파일을 /Projects/RTOS/inc 밑에 복사한다.

3.3.2 라이브러리 설정

- 프로젝트 [Option]으로 들어가서 다음과 같이 설정한다.



1) Additional include directories 에 header 파일이 들어있는 include 디렉토리를 추가해 준다.

```
$PROJ_DIR$\..\..\Libraries\CMSIS\Include
$PROJ_DIR$\..\..\Libraries\CMSIS\Device\ST\STM32F10x\Include
$PROJ_DIR$\..\..\Libraries\STM32F10x_StdPeriph_Driver\inc
$PROJ_DIR$\..\..\Libraries\STM32_USB-FS-Device_Driver\inc
$PROJ_DIR$\..\..\driver\inc
$PROJ_DIR$\inc
```

2) Defined symbols 에 STM32F10X_HD와 USE_STDPERIPH_DRIVER를 정의한다.

```
STM32F10X_HD
USE_STDPERIPH_DRIVER
```

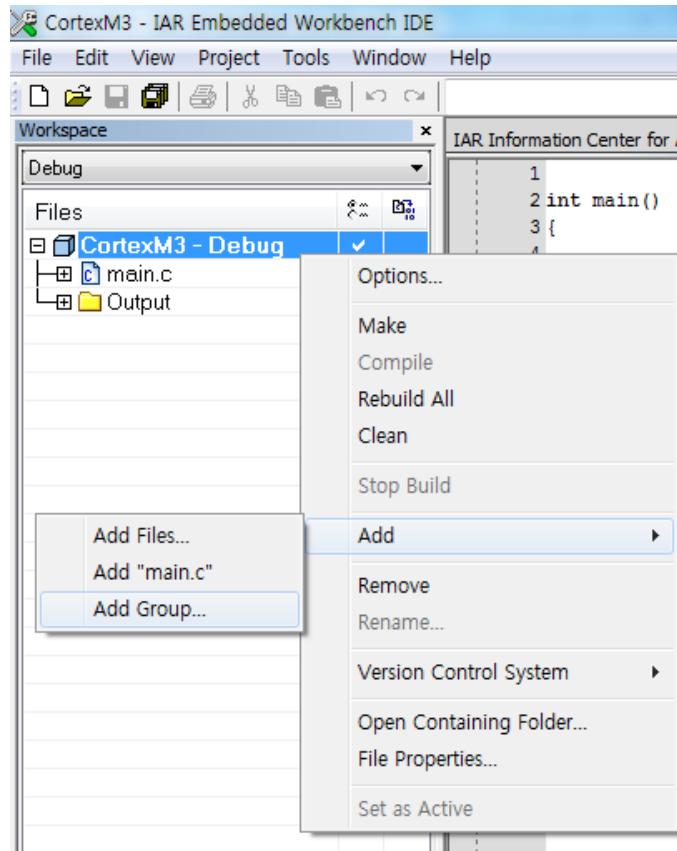
우리가 사용하고 있는 라이브러리는 STM32F10X 시리즈의 위한 것인데 이 중에서 STM32 High density devices 용으로 컴파일 하기위해 정의를 추가한다.

3.3.3 프로젝트 설정

Workspace에 설치한 라이브러리를 추가한다.

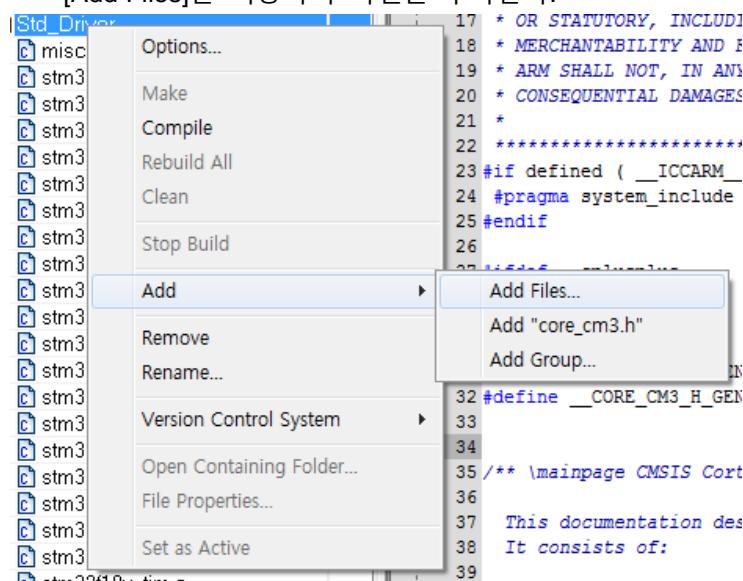
■ Group 추가

[Add Group]를 선택하여 Group를 추가한다.

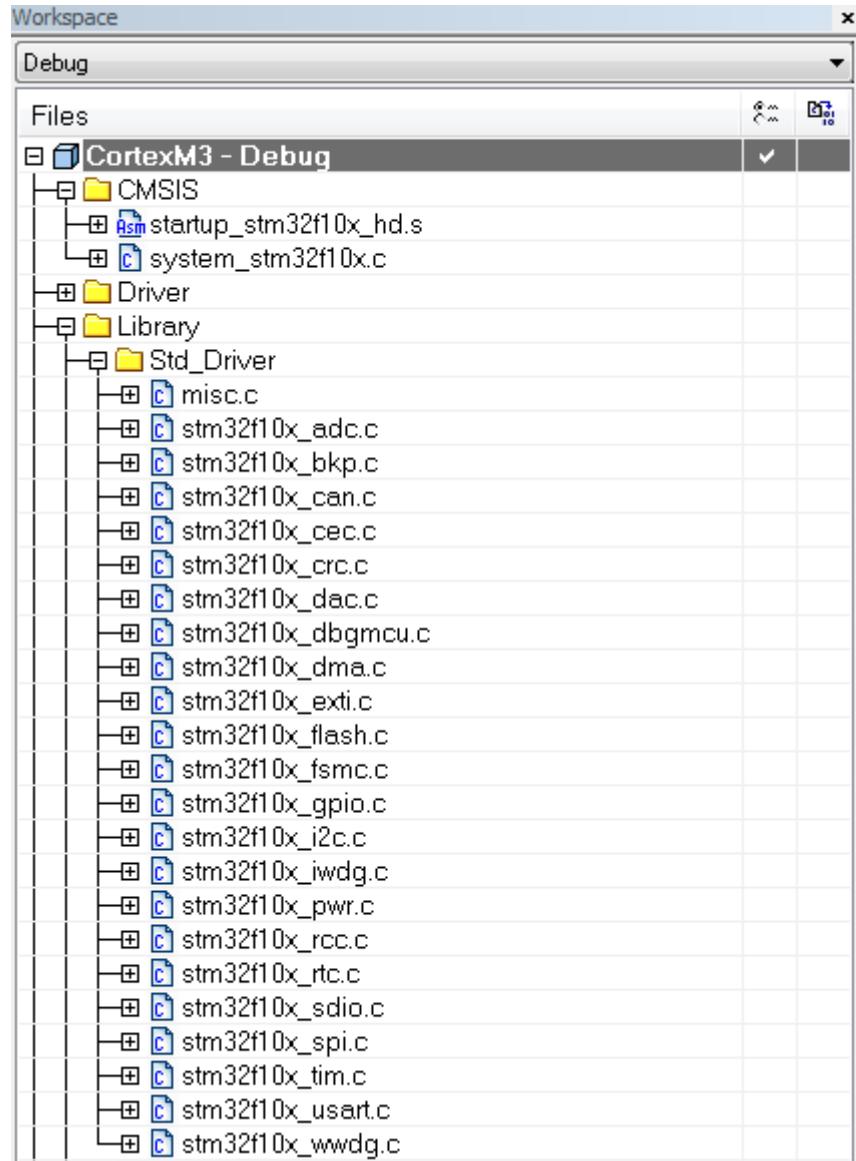


■ 파일 추가

[Add Files]를 이용하여 파일을 추가한다.



- ST에서 받은 라이브러리 파일을 다음과 같이 추가시킨다.



각각의 파일 위치는 다음과 같다.

파일명	위치
startup_stm32f10x_hd.s	\Libraries\CMSIS\Device\ST\STM32F10x\Source\Templates\iar
system_stm32f10x.c	\Libraries\CMSIS\Device\ST\STM32F10x\Source\Templates
기타 드라이버 파일	\Libraries\STM32F10x_StdPeriph_Driver\src

3.4 FreeRTOS 설치

3.4.1 소스코드 다운로드

FreeRTOS 홈페이지의 다음 페이지에서 최신 버전의 코드를 다운로드 받는다.

<http://sourceforge.net/projects/freertos/files/>

FreeRTOS Real Time Kernel (RTOS)

Market leading real time kernel for 32 microcontroller architectures

Brought to you by: rtel

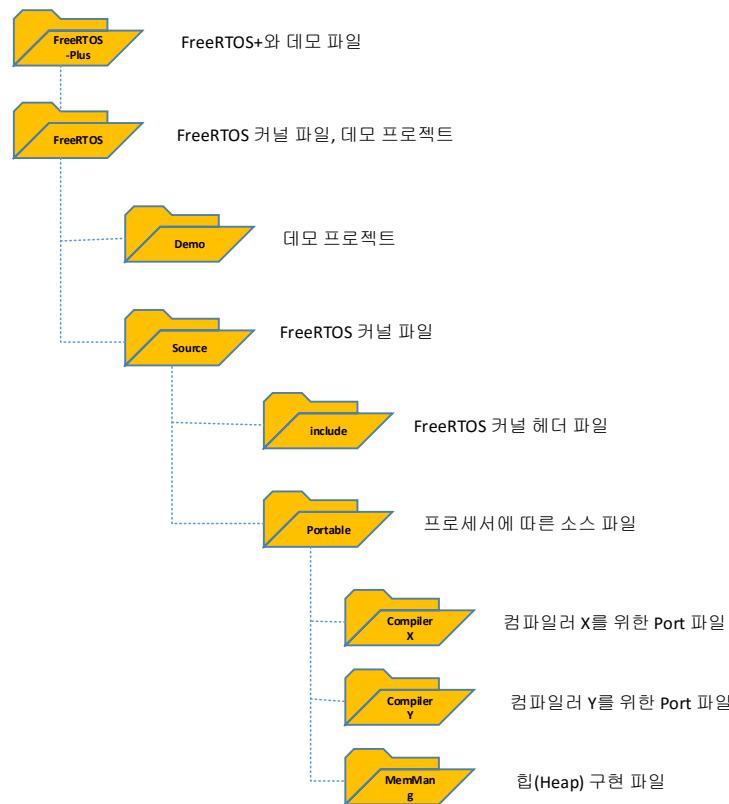
Summary	Files	Reviews	Support	Wiki	Tickets ▾	News	Discussion	Donate	Code
---------	-------	---------	---------	------	-----------	------	------------	--------	------

Looking for the latest version? [Download FreeRTOSV8.0.0.exe \(9.6 MB\)](#)

Home			
Name	Modified	Size	Downloads / Week
■ FreeRTOS	2014-02-19		
readme.txt	2013-07-23	1.5 kB	8
README	2013-07-19	91.7 kB	4
Totals: 3 Items		93.2 kB	12

3.4.2 FreeRTOS 파일

■ FreeRTOS 폴더 구조



FreeRTOS 소스코드를 홈페이지에서 다운받아서 압축을 해제하면 위의 왼쪽 그림과 같은 폴더 구조를 가지고 있다. \FreeRTOS 와 \FreeRTOS-Plus 폴더를 \에 복사를 한다.

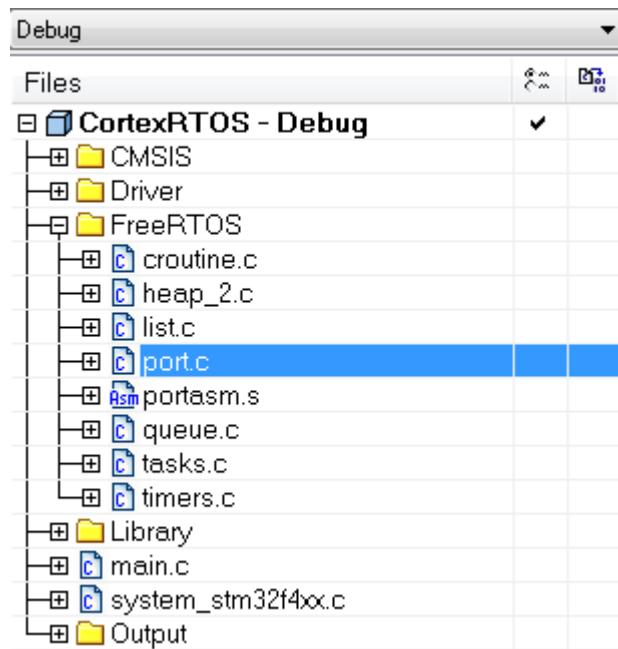
■ FreeRTOSIOConfig.h 설치

이 파일은 FreeRTOS의 설정을 사용자의 환경에 맞게 설정하는 파일으로 사용자가 직접 작성해야 되는 파일이다. FreeRTOS에서 예제로 주는 FreeRTOSIOConfig.h 를 사용하기로 한다. 아래의 폴더에 있는 이 파일을 \Projects\RTOS\inc\에 복사한다.

\FreeRTOS\FreeRTOS\Demo\CORTEX_STM32F103_IAR\FreeRTOSIOConfig.h

3.4.3 IAR 프로젝트 구성

3.4.3.1 FreeRTOS 파일 구성

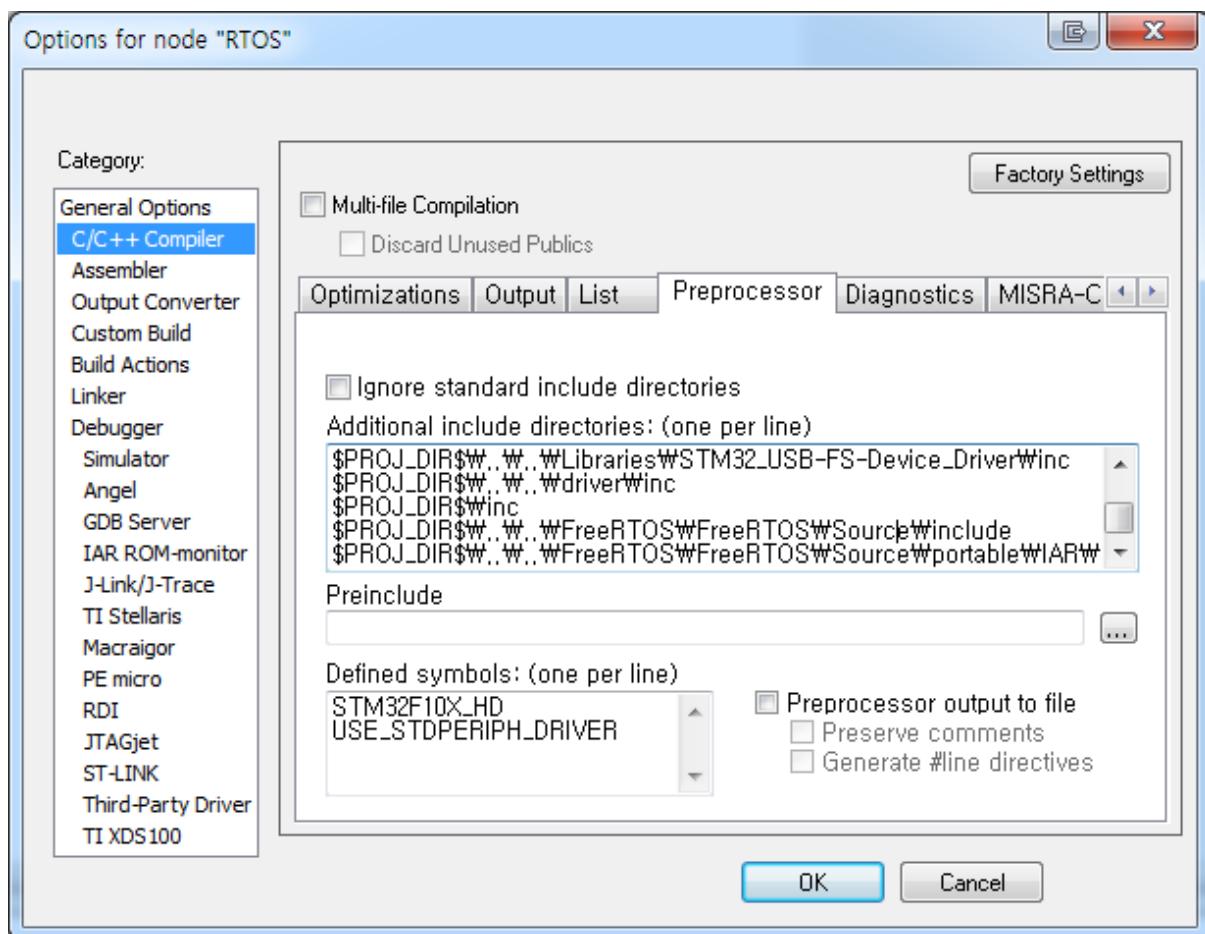


위의 그림과 같이 FreeRTOS 파일을 [Workspace]에 추가한다. 파일의 위치는 아래와 같다.

파일명	파일 위치
list.c	\FreeRTOS\
queue.c	\FreeRTOS\
tasks.c	\FreeRTOS\
timers.c	\FreeRTOS\
croutine.c	\FreeRTOS\
port.c	\FreeRTOS\portable\IAR\ARM_CM3
portasm.s	\FreeRTOS\portable\IAR\ARM_CM3
heap_2.c	\FreeRTOS\portable\memmang

3.4.3.2 프로젝트 옵션 구성

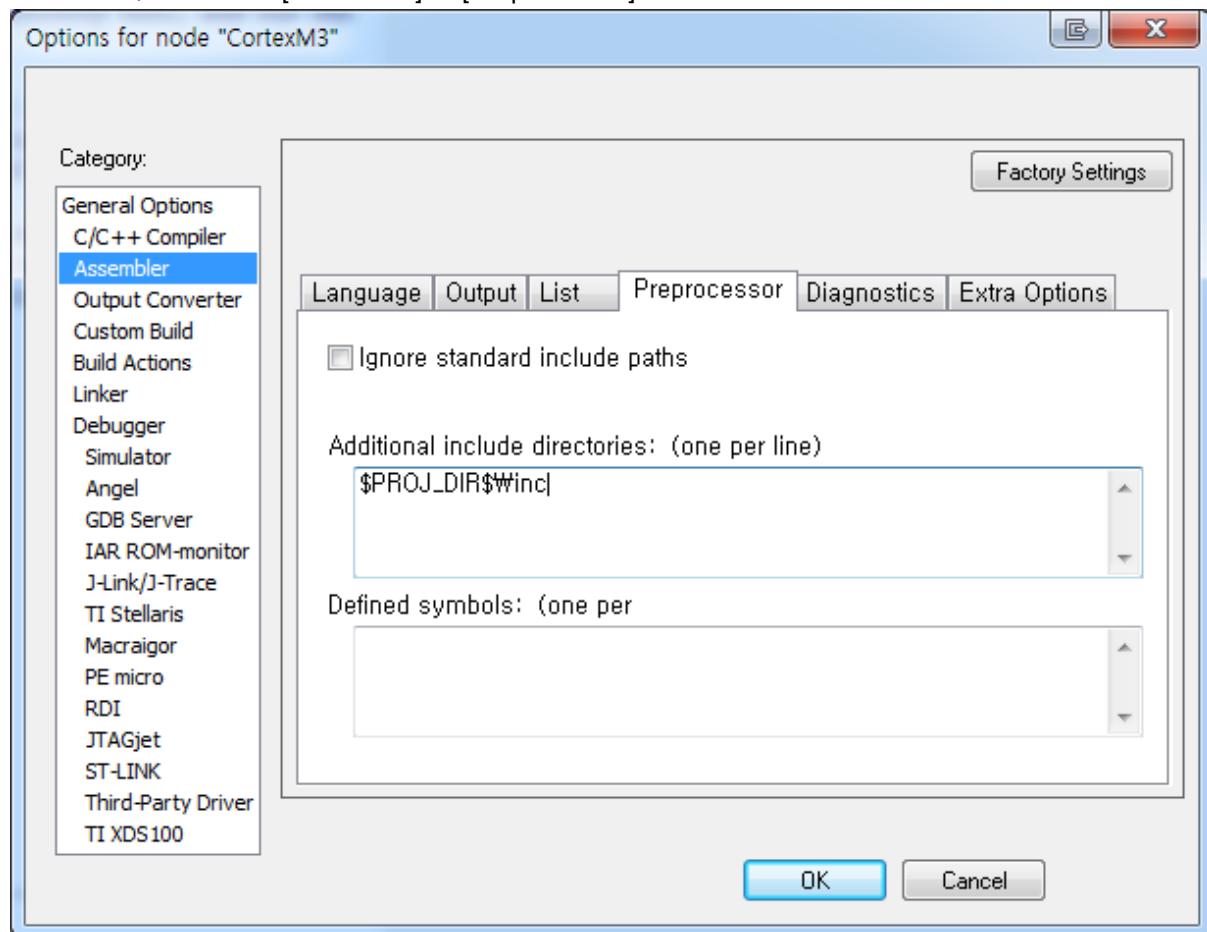
■ 프로젝트 옵션 -> [C/C++ Compiler] -> [Preprocessor]



```
$PROJ_DIR$\\..\\..\\Libraries\\CMSIS\\Include
$PROJ_DIR$\\..\\..\\Libraries\\CMSIS\\Device\\ST\\STM32F10x\\Include
$PROJ_DIR$\\..\\..\\Libraries\\STM32F10x_StdPeriph_Driver\\inc
$PROJ_DIR$\\..\\..\\Libraries\\STM32_USB-FS-Device_Driver\\inc
$PROJ_DIR$\\..\\..\\driver\\inc
$PROJ_DIR$\\inc
$PROJ_DIR$\\..\\..\\FreeRTOS\\FreeRTOS\\Source\\include
$PROJ_DIR$\\..\\..\\FreeRTOS\\FreeRTOS\\Source\\portable\\IAR\\ARM_CM3
```

FreeRTOS 관련 헤더파일이 있는 폴더를 추가한다.

■ 프로젝트 옵션 -> [Assembler] -> [Preprocessor]



어셈블러에서 필요한 헤더파일 폴더를 추가한다.

\$PROJ_DIR\$\inc

3.4.4 FreeRTOS에 필요한 ISR(Interrupt Service Routine) 연결

FreeRTOS에서는 다음과 같은 3가지 ISR를 연결을 시켜 주어야 한다.

ISR	FreeRTOS
SVC_Handler	vPortSVCHandler
PendSV_Handler	xPortPendSVHandler
SysTick_Handler	xPortSysTickHandler

위와 같이 하기 위해서 FreeRTOSConfig.h 에 다음과 같은 코드를 삽입한다.

```
#define configLIBRARY_KERNEL_INTERRUPT_PRIORITY      15

#define vPortSVCHandler SVC_Handler
#define xPortPendSVHandler PendSV_Handler
#define xPortSysTickHandler SysTick_Handler

#endif /* FREERTOS_CONFIG_H */
```

4. FreeRTOS 시작하기

4.1 실시간 시스템

실시간 시스템이란 임의의 **정보가 시스템에 입력되었을 때 주어진 시간 내에 작업이 완료되어 결과가 주어지는 것을** 의미한다.

실시간 시스템은 Hard Real Time과 Soft Real Time으로 나뉜다.

Hard Real Time의 특징은 어떤 task를 일정 시간 내에 반드시 처리해야 하며 그 시간이 지난 후의 결과 값은 정확해도 의미가 없는 경우로 군사장비, 비행기 내부 제어장비, 산업용 제어 시스템 등에 사용된다.



Soft Real Time은 어떤 시간 내에 처리하면 좋지만 그렇지 못한 경우 그 시간에서 약간 경과한 후의 처리 결과값도 인정하는 경우이다. Soft Real Time은 주로 휴대용 단말기, 라우터 등 일반적인 가전제품에 많이 사용된다.



4.2 RTOS(Real Time Operating System) 란?

RTOS란 주어진 작업을 정해진 시간 안에 수행 할 수 있도록(실시간) 도와주는 환경을 제공하는 OS이다. RTOS를 사용한다고 해서 실시간성이 보장되는 것은 아니다.

RTOS에서는 실시간성을 보장하기 위해서 예측가능하고 일정한 응답 시간을 요구하는 응용 프로그램의 자원을 지원한다. 이러한 특성 때문에 RTOS는 하드웨어 자원을 좀 낭비하더라도 작업의 시간제한을 맞추는데 주력하며, 우선순위를 기반으로 하는 Task 스케줄링을 많이 사용한다.

4.3 멀티태스킹(Multitasking)

단일 태스크에서 동시에 두 가지 이상의 일을 해야 될 경우에 일 처리는 동시에 두 가지 일을 할 수 없고 하나의 일이 끝나야지만 다른 일을 할 수가 있다. 이렇게 되면 실시간성을 유지하기가 어렵게 된다. 그래서 대부분의 OS는 동시에 여러 개의 프로그램을 실행 시키는 것을 가능하게

한다. 이것을 멀티 태스킹이라고 한다. 그러나 실제의 프로세서는 동시에 여러 개의 프로그램(쓰레드)를 실행할 수 없고 한번에 하나의 프로그램만을 실행시킬 수 있다. OS의 한 부분을 차지하고 있는 스케줄러(scheduler)는 언제 프로그램을 실행시킬 것인가를 결정하고 또한 각각의 프로그램을 빠르게 전환(Switching) 시킴으로써 동시에 동작하는 것처럼 보이게 한다.

OS의 타입에 따라 스케줄러가 언제 프로그램을 실행시킬 것인가가 결정된다. 유닉스 같은 멀티유저OS는 각각의 유저에게 합당한 처리 시간을 보장하면서 여러 프로그램을 실행시킨다. 또한 윈도우즈 같은 데스크탑용 OS의 스케줄러는 유저가 응답 가능한 상태에 있는지 확인하면서 실행을 한다.

RTOS에서의 스케줄러는 예상 가능한(predictable, deterministic) 실행 패턴을 제공하도록 설계되었다. 이것은 실시간성을 가진 임베디드 시스템으로서의 특별한 관심사항이다. 이 실시간성 요구에 대한 보장은 OS의 스케줄러의 동작이 예상 가능할 경우에만 가능하다.

그래서 일반적인 RTOS의 스케줄러에서는 사용자가 프로그램의 우선순위를 정하는 것을 허용함으로서 예상 가능한 시간안에 동작이 가능하도록 한다. 스케줄러는 이 우선순위에 의해 다음에 실행할 프로그램을 결정한다.



	
예상 가능한 상태	

4.4 비선점형/선점형 커널

멀티태스킹은 한 번에 두가지 일을 CPU가 할 수 없기 때문에 시분할 방식을 통하여 여러 태스크가 동시에 이루어지 지는 것처럼 보이게 한다. 이 때 스케줄러가 언제 어떤 태스크가 CPU를 점유할지를 결정한다.

스케줄러가 태스크를 결정하는 방식에 따라 비선점형과 선점형 커널로 나눌수가 있다.

- 비선점형 커널(Non-preemptive Kernel)

어떤 프로세스가 CPU를 할당 받으면 그 프로세스가 종료되거나 입출력 요구가 발생하여 자발적으로 중지될 때까지 계속 실행되도록 보장한다. 순서대로 처리되는 공정성이 있고 다음에 처리해야 할 프로세스와 관계없이 응답 시간을 예상할 수 있으며 선점 방식보다 스케줄러 호출 빈도 낮고 문맥 교환에 의한 오버헤드가 적다. 일괄 처리 시스템에 적합하며, CPU 사용 시간이 긴 하나의 프로세스가 CPU 사용 시간이 짧은 여러 프로세스를 오랫동안 대기시킬 수 있으므로, 처리율이 떨어질 수 있다는 단점이 있다

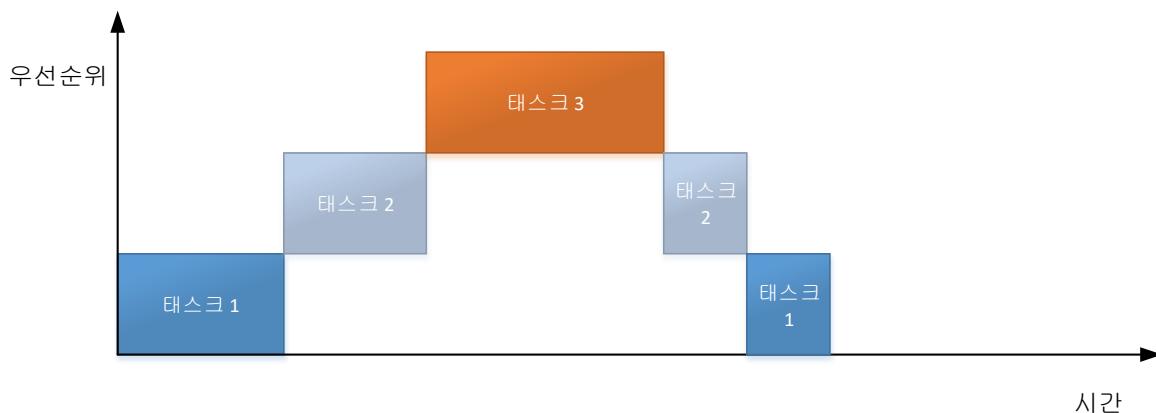
- FCFS 스케줄링(First Come First Served Scheduling)
 - : 대기 큐에 먼저 들어온 태스크순으로 CPU를 할당
- SJF 스케줄링(Short Job First Scheduling)
 - : 소요시간이 짧은 태스크순으로 할당
- HRRN 스케줄링(Highest Response Ratio Next Scheduling)
 - : CPU 처리 기간이 길고 해당 프로세스의 대기 시간이 긴 태스크를 먼저 할당.



● 선점형 커널(Preemptive Kernel)

어떤 프로세스가 CPU를 할당받아 실행 중에 있어도 다른 프로세스가 실행 중인 프로세스를 중지하고 CPU를 강제로 점유할 수 있다. 모든 프로세스에게 동일한 우선순위를 주고 라운드로빈 방식으로 CPU 사용 시간을 동일하게 부여할 수 있다. 빠른 응답시간을 요하는 대화식 시분할 시스템에 적합하며 긴급한 프로세서를 제어할 수 있다. '운영 체제가 프로세서 자원을 선점'하고 있다가 각 프로세스의 요청이 있을 때 특정 요건들을 기준으로 자원을 배분하는 방식이다

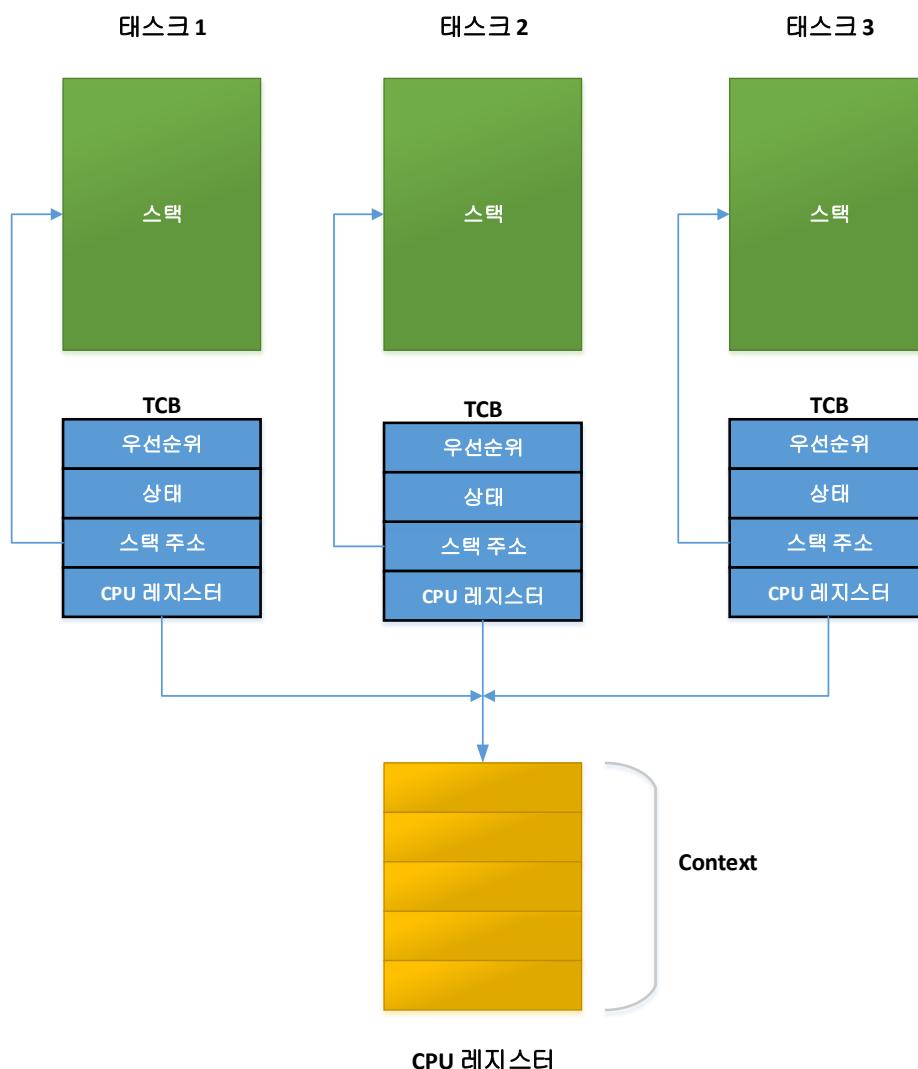
- RR 스케줄링(Round Robin Scheduling)
 - : 프로세스들 사이에 우선순위를 두지 않고, 순서대로 시간단위(Time Quantum)로 CPU를 할당
- 비율단조 스케줄링(Rate Monotonic Scheduling)
 - : 수행 주기가 가장 짧은 프로세스에 가장 높은 우선순위를 부여하는 방식



4.5 문맥 전환(Context Switching)

Task는 메모리에 독립된 Stack 영역 가지게 되고, 다른 Task들은 이 영역을 액세스 할 수 없다. 함수 안에 선언한 지역변수, 함수의 argument, 함수 수행 후 돌아갈 return 어드레스 등이 Stack에 저장된다. 이것을 Task Control Block (TCB)라 한다. 어떤 프로그램이 수행될 때 함수를 호출할 것이며 그 과정에서 Stack 이 쓰이게 된다.

Scheduler에 의해서 새로이 RUNNING 상태가 될 Task가 결정되면 현재 Running 상태의 Task가 사용하던 문맥을 메모리 특정 영역에 저장한다. 그리고 새로이 수행될 Task의 문맥을 TCB에서 CPU의 레지스터 영역으로 복사하여 새로운 Task가 수행되도록 하는 작업을 문맥교환 (Context Switch, C/S) 라 한다. Task가 사용하는 CPU 레지스터들의 값을 문맥이라고 한다. 문맥은 Task가 유지해야 하는 정보들의 모임으로서 위에서 설명한 것 보다 좀 더 포괄적인 의미를 가지고 있다.



4.6 FreeRTOS 개요

4.6.1 FreeRTOS 의 특징

Pre-emptive scheduling option	Easy to use message passing
Co-operative scheduling option	Round robin with time slicing
ROMable	Mutexes with priority inheritance
6K to 10K ROM footprint	Recursive mutexes
Configurable / scalable	Binary and counting semaphores
Compiler agnostic	Very efficient software timers
Some ports never completely disable interrupts	Easy to use API

4.6.2 FreeRTOS가 이식된 마이컴

- Actel (now Microsemi)
 - Supported processor families: SmartFusion, SmartFusion2 - see Microsemi listing
 - Supported tools: IAR, Keil, SoftConsole (GCC with Eclipse)
- Altera
 - Supported processor families: Nios II
 - Supported tools: Nios II IDE with GCC
- Atmel
 - Supported processor families: SAM3 (Cortex-M3), SAM4 (Cortex-M4), SAM7 (ARM7), SAM9 (ARM9), AT91, AVR and AVR32 UC3
 - Supported tools: IAR, GCC, Keil, Rowley CrossWorks
- Cortus
 - Supported processor families: APS3
 - Supported tools: Cortus IDE with GCC
- Cypress
 - Supported processor families: PSoC 5 ARM Cortex-M3
 - Supported tools: GCC, ARM Keil and RVDS - all in the PSoC Creator IDE
- Energy Micro
 - Supported processor families: EFM32 (Cortex-M3, ARM Cortex-M4)
 - Supported tools: A demo project is provided for IAR, although this uses the standard ARM Cortex-M3 port so projects for GCC and Keil could also be created.
- Freescale
 - Supported processor families: Kinetis ARM Cortex-M4, Coldfire V2, Coldfire V1, other Coldfire families, HCS12, PPC405 & PPC440 (Xilinx implementations) (small and banked memory models), plus contributed ports

- Supported tools: Codewarrior, GCC, Eclipse, IAR
- Infineon
 - Supported processor families: TriCore, XMC4000 (Cortex-M4F)
 - Supported tools: GCC, Keil, Tasking, IAR
- Fujitsu
 - Supported processor families: FM3 ARM Cortex-M3, 32bit (for example MB91460) and 16bit (for example MB96340 16FX)
 - Supported tools: Softune, IAR, Keil
- Luminary Micro / Texas Instruments
 - Supported processor families: All Luminary Micro ARM Cortex-M3 and ARM Cortex-M4 based Stellaris microcontrollers
 - Supported tools: Keil, IAR, Code Red, CodeSourcery GCC, Rowley CrossWorks
- Microchip
 - Supported processor families: PIC32, PIC24, dsPIC, (PIC18)
 - Supported tools: MPLAB C32, MPLAB C30, (MPLAB C18 and wizC)
- Microsemi
 - Supported processor families: SmartFusion, SmartFusion2
 - Supported tools: IAR, Keil, SoftConsole (GCC with Eclipse)
- NEC (now Renesas)
 - Supported processor families: V850 (32bit), 78K0R (16bit)
 - Supported tools: IAR
- NXP
 - Supported processor families: LPC1700 (Cortex-M3), LPC1800 (Cortex-M3), LPC1100 (Cortex-M0), LPC4300 (Cortex-M4F) LPC2000 (ARM7), LPC4300 (Cortex-M4F/Cortex-M0)
 - Supported tools: GCC, Rowley CrossWorks, IAR, Keil, Red Suite, Eclipse
- Renesas
 - Supported processor families: RZ (ARM Cortex-A9), RX600 / RX62N / RX63N, RX200, RX100, SuperH, RL78, H8/S plus contributed ports
 - Supported tools: GCC, HEW (High Performance Embedded Workbench), IAR Embedded Workbench
- Silicon Labs [ex Cygnal]
 - Supported processor families: Super fast 8051 compatible microcontrollers.
 - Supported tools: SDCC
- ST
 - Supported processor families: STM32 (Cortex-M0, ARM Cortex-M3 and ARM Cortex-M4F), STR7 (ARM7), STR9 (ARM9)
 - Supported tools: IAR, Atollic TrueStudio, GCC, Keil, Rowley CrossWorks
- TI
 - Supported processor families: RM48, TMS570, MSP430, MSP430X, Stellaris (Cortex-M3, ARM Cortex-M4F)
 - Supported tools: Rowley CrossWorks, IAR, GCC, Code Composer Studio
- Xilinx

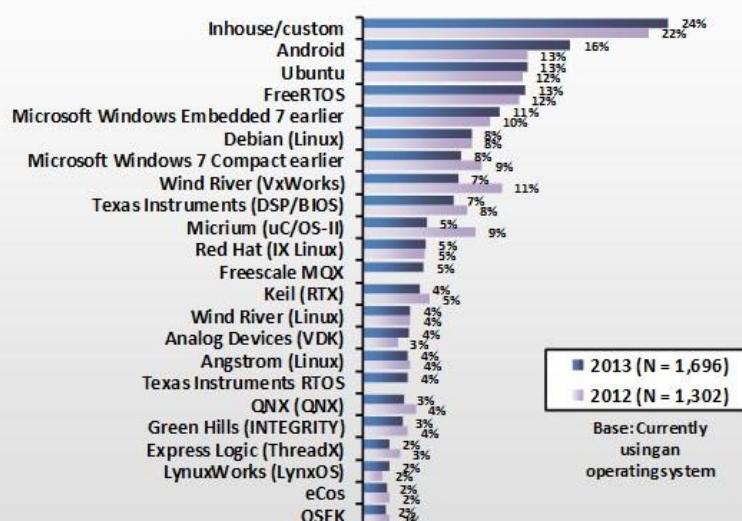
- Supported processor families: PPC405 running on a Virtex4 FPGA, PPC440 running on a Virtex5 FPGA, Microblaze (a Zynq port is available in the FreeRTOS Interactive site)
- Supported tools: GCC
- x86
 - Supported processor families: Any x86 compatible running in Real mode only, plus a Win32 simulator
 - Supported tools: Visual Studio 2010 Express, MingW, Open Watcom, Borland, Paradigm

4.6.3 FreeRTOS의 시장 점유율

2013 Embedded Market Study

23

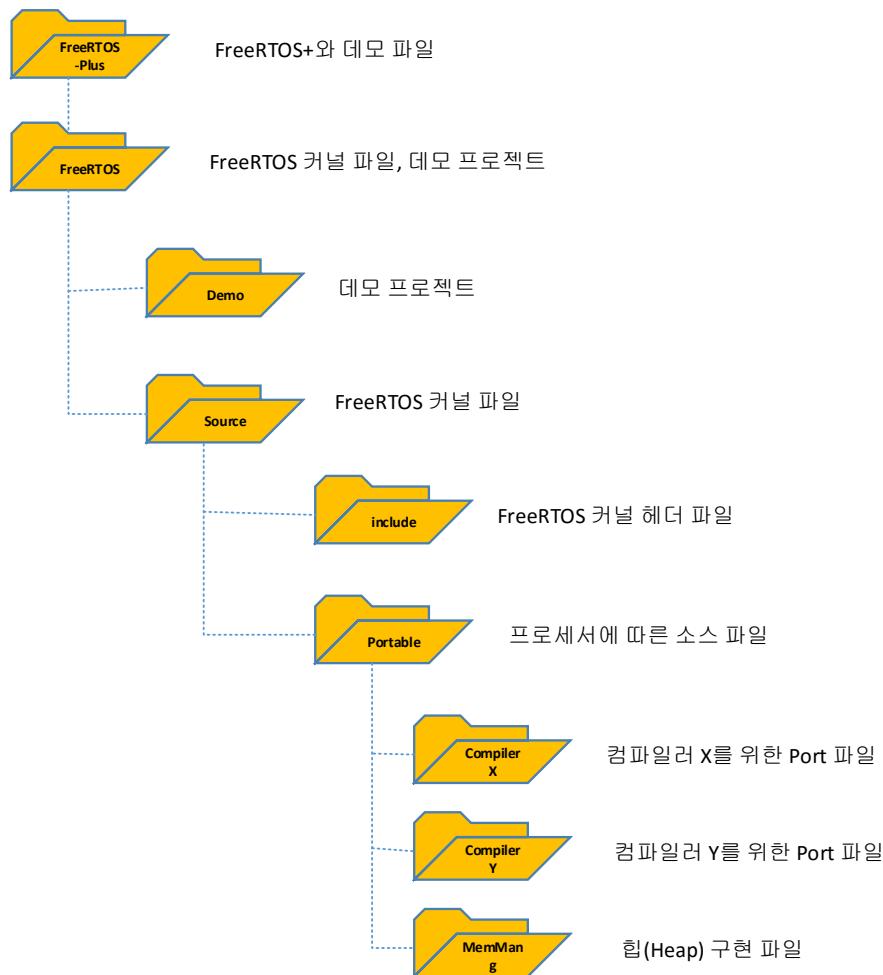
Please select ALL of the operating systems you are currently using.



Only Operating Systems that had 2% or more are shown.

Copyright © 2013 by UBM / EE Times Group. All rights reserved.

4.6.4 FreeRTOS 디렉토리 구조



4.7 FreeRTOS 시작시키기

4.7.1 태스크의 생성

멀티 태스크를 하려면 당연히 태스크를 먼저 생성해야 된다. FreeRTOS에서의 태스크 생성 방법은 다음과 같이 하면 된다.

```
xTaskCreate(Task1,           /* 태스크를 실행할 함수 포인터. */
            "Task 1", /* 태스크 이름 */
            240,      /* 스택의 크기 */
            NULL,     /* 태스크 매개변수 */
            2,        /* 태스크의 우선순위 */
            NULL );   /* 태스크 핸들 */
```

`xTaskCreate()` 라는 함수가 태스크를 생성해 주는 함수이다. 첫번째 매개변수는 태스크를 실행한 함수 포인터이다. 그냥 함수 하나 만들어 주고 이름만 여기다 붙여 주면 된다. 이 함수에 대한 세부적인 것은 밑에 설명한다.

세번째 매개변수에 태스크에서 사용할 스택의 크기를 적어 준다.

다섯번째 매개변수에는 태스크 우선순위를 적는다. FreeRTOS에서는 우선순위 숫자의 크기가 크면 우선순위가 높다. 1 보다 2가 우선 순위가 높다.

그럼 태스크를 실행할 Task1라는 함수를 만들어 보자

```
static void vTask1(void *pvParam)
{
    for(;;)
    {
        vLED_On(0xFF);
        vDelay(1000000);
        vLED_On(0x00);
        vDelay(1000000);
    }
}
```

태스크는 매개 변수가 `void * pvParam`이고 리턴값이 없는 함수여야 한다.

여기서는 단순히 `for`문은 사용해서 무한히 LED를 일정한 간격으로 점멸하는 프로그램을 작성하였다.

4.7.2 태스크의 실행

`xTaskCreate()`를 사용해서 태스크를 생성했다고 해서 태스크가 바로 돌아가는 것은 아니다. 태스크를 실행시키기 위해서는 `vTaskStartScheduler()` 라는 함수를 실행하여야 FreeRTOS에서 태스크를 스케줄링하면서 태스크를 실행시킨다.

```
int main()
{
    Init();           // 시스템 초기화

    // 태스크 생성
    xTaskCreate(vTask1,    // 테스크를 실행할 함수 포인터.
                "Task1",    // 테스크 이름
                240,       // 스택의 크기
                NULL,      // 테스크 매개변수
                1,         // 테스크의 우선순위
```

```
        NULL          // 테스크 핸들
    );

/* 태스크 실행 */
vTaskStartScheduler();

return 0;
}

static void vTask1(void *pvParam)
{
    for(;;)
    {
        vLED_On(0xFF);
        vDelay(1000000);
        vLED_On(0x00);
        vDelay(1000000);
    }
}

static void vDelay(unsigned portLONG ulDelay)
{
    volatile unsigned portLONG i;

    for(i = 0; i < ulDelay;i++);
}
```

main() 함수에서 Task1 태스크를 생성하고 vTaskStartScheduler()라는 함수를 실행시켜 태스크를 실행시킨다. 이 함수는 무한히 태스크를 관리하면서 태스크를 실행시키므로 그 밑으로는 빠져 나오지 않는다.

이 프로그램을 실행하게 되면 Task1 실행되면서 일정한 간격으로 LED를 점멸하게 된다

5. FreeRTOS

5.1 태스크 관리(Task Management)

5.1.1 태스크(Task)의 구현

C언어에서 어떤 일을 표현하는 방법은 함수를 하나 만드는 것이다. 태스크도 하나의 일을 처리하는 것이므로 태스크의 구현은 함수를 하나 만드는 것이 된다. 단순히 함수를 만드는 것이라면 일반 함수와 구별이 가질 않을 것이다. 그래서 태스크는 특정한 함수의 형태를 가지는데 FreeRTOS에서의 원형은 다음과 같이 표시한다.

```
Void Task1(void *pvParameters);
```

■ 태스크(Task)의 정의

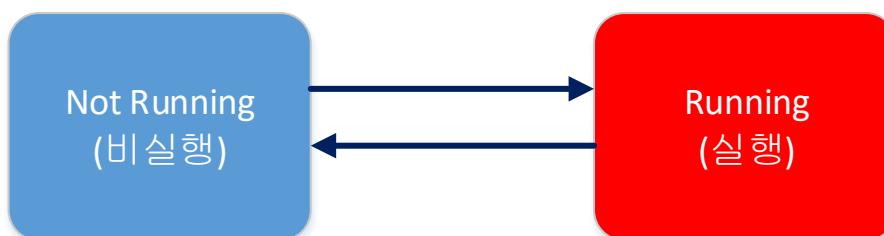
자신만의 일정한 프로그램 영역(코드메모리, 스택 메모리..)을 가지고 돌아가는 작은 프로그램으로, 무한히 돌아가는 루프에서 자기의 일을 실행한다.

```
Void Task1(void *pvParameters)
{
    for(;;)
    {
        // 태스크에서 실행할 코드를 넣는다.
    }
}
```

5.1.2 태스크(Task)의 기본 상태

프로그램은 여러 개의 태스크로 이루어 질 수 있다. 그러나 프로세서는 한 번에 하나의 태스크만을 실행시킬 수 있다. 그러므로 각각의 태스크는 다음의 그림과 같이 두 가지의 상태(Running/Not Running)를 가지게 된다. 뒤에 설명하겠지만 Not Running 상태는 보다 많은 하위 상태가 존재한다. 지금은 간단히 두 가지 상태만을 정의한다.

태스크가 Running 상태에 있을 때는 프로세서가 해당 코드를 실행하고 있는 상태를 말한다. Not Running 상태에 있을 때는 태스크가 휴면(dormant) 상태에 있는 것을 말한다. 스케줄러가 다음에 Running상태로 만들 때까지 준비하고 기다리는 상태이다. 만약 태스크가 다시 시작하게 되면 최근에 멈춘 상태로부터 다시 시작하게 된다.



태스크가 Not Running 상태에서 Running 상태로의 전환을 태스크 스위치(Task switch)라고 한다. 이런 태스크 스위치는 RTOS의 스케줄러에서 이루어지게 된다.

5.1.3 태스크의 생성(Create Tasks)

태스크의 생성은 다음과 같은 FreeRTOS xTaskCreate() API를 사용한다

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed char * const pcName,
                           unsigned short usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           TaskHandle_t *pxCreatedTask
);
```

pvTaskCode	태스크로 사용 할 함수의 함수명을 넣는다.
pcName	태스크의 문자열 이름, 실질적으로 FreeRTOS에서 사용하지는 않지만 디버깅용도로 사용함.
usStackDepth	태스크에서 사용할 스택의 크기, 단위는 워드(Word)이다.
pvParameters	태스크로 전달하고 싶은 값을 이 파라미터로 전달한다. 없으면 NULL로 설정.
uxPriority	태스크의 우선 순위, 0은 가장 낮은 우선순위를 가지며, 숫자가 커지면 우선 순위가 높아진다. 가장 높은 우선순위는 configMAX_PRIORITIES – 1 이 된다.
pxCreatedTask	태스크의 핸들을 받는다. 태스크의 우선순위를 변경하거나 지울 때 사용된다. 사용하지 않으면 NULL로 설정한다.

예제 1) 우선 순위가 같은 두 개의 태스크 동작

두 개의 태스크의 우선순위를 모두 1로 해서 생성하고 스케줄러를 동작시켜 멀티 테스킹이 어떻게 동작하는지 알아본다.

Task1은 주기적으로 LED를 0xFF와 0x00를 점멸한다.

```
static void vTask1(void *pvParam)
{
    for(;;)
    {
        vLED_On(0xFF);
        vDelay(1000000);
        vLED_On(0x00);
        vDelay(1000000);
    }
}
```

Task2은 Count 변수를 주기적으로 하나씩 증가해서 FND로 출력해서 보여준다.

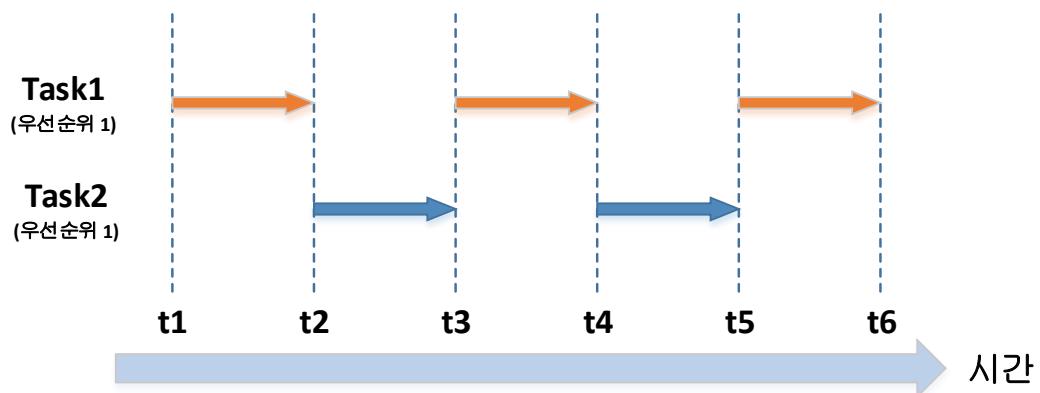
```
static void vTask2(void *pvParam)
{
    unsigned portLONG Count = 0;

    for(;;)
    {
        vFND_Write(Count % 10, 0, 0);
        vFND_Write(Count / 10 % 10, 1, 1);
        vFND_Write(Count / 100 % 10, 2, 0);
        vFND_Write(Count / 1000 % 10, 3, 0);
        vDelay(1000000);
        //vTaskDelay(100/portTICK_PERIOD_MS);
        Count++;
    }
}
```

```
void vTask_SimpleTask(void)
{
    // 태스크 생성
    xTaskCreate(vTask1,                                // 태스크를 실행할 함수 포인터.
                "Task1",                                // 태스크 이름
                240,                                     // 스택의 크기
                NULL,                                    // 태스크 매개변수
                1,                                       // 태스크의 우선순위
                NULL);                                  // 태스크 핸들
    );

    // 태스크 생성
    xTaskCreate(vTask2,                                // 태스크를 실행할 함수 포인터.
                "Task2",                                // 태스크 이름
                240,                                     // 스택의 크기
                NULL,                                    // 태스크 매개변수
                1,                                       // 태스크의 우선순위
                NULL);                                  // 태스크 핸들
    );
}
```

FreeRTOS는 선점형 커널을 가지고 있기 때문에 항상 우선순위가 높은 태스크가 프로세스를 점유하게 되어있다. 그런데 FreeRTOS에서는 하나 이상의 태스크가 같은 우선 순위를 가질 수 있도록 설계되어 있다. 같은 우선 순위 태스크끼리의 프로세스 할당은 라운드 로빈 스케줄링(Round Robin Scheduling) 방식을 취한다. 이 방법은 태스크를 순서대로 시간 단위(Time Quantum/Time Slice)로 프로세스를 할당하는 방식을 취한다. 지금 Task1 과 Task2의 우선순위가 1로 같기 때문에 아래와 형태로 각각의 프로세스가 일정한 시간동안 Running/Non Running 상태를 번갈아 가면서 동작하게 된다.



5.1.4 태스크의 우선순위 (Task Priorities)

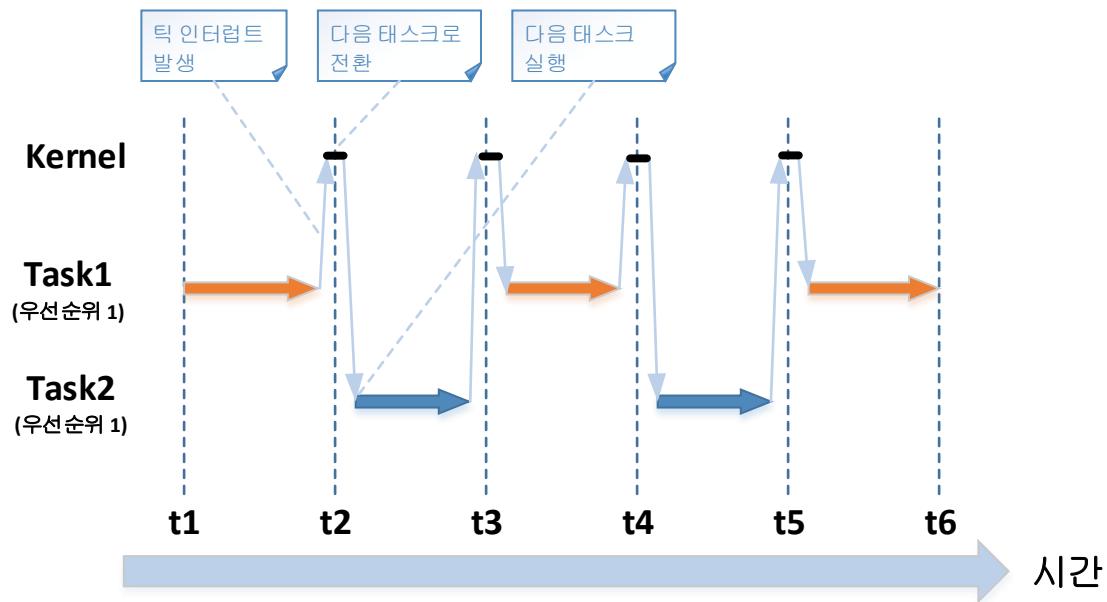
태스크는 xTaskCreate() 함수를 사용해서 생성하였다. 여기서 uxPriority라는 매개변수를 통해서 생성하는 태스크의 우선순위를 설정하게 된다. 태스크의 우선 순위는 항상 생성할 때의 값으로 고정되는 것이 아니라 실행 도중이라도 vTaskPrioritySet() API를 통해서 언제든지 변경할 수 있다.

우선 순위는 0이 가장 낮은 우선순위를 나타내고 값이 증가하면 우선순위도 또한 높아지게 된다. 다시 말해 값이 크면 우선순위도 높다는 것이다. 우선 순위의 최대값은 <FreeRTOSConfig.h> 안의 **configMAX_PRIORITY**의 값으로 설정할 수가 있다.

태스크에 우선순위를 정하는데 있어서 FreeRTOS는 특별한 제약은 없다. 최대 우선순위 값보다 작은 값 안에서 어떠한 우선순위를 가져도 된다. 다른 태스크끼리 같은 우선순위를 가져도 상관이 없다.

스케줄러는 항상 **실행할 수 있는(블록되지 않은) 가장 우선순위가 높은 태스크**가 실행이 되도록 한다. 실행할 준비가 된 태스크가 하나 이상이고 같은 우선 순위를 가졌으면 스케줄러는 태스크를 순서대로 일정한 시간(time slice) 동안 실행 상태(Running)로 전환시킨다.

다음 태스크를 실행시키기 위해서는 Time slice의 끝에 잠시 동안 스케줄러가 실행이 되고, 태스크를 전환(switch)할 다음 태스크를 실행시켜야만 한다. 틱(Tick) 인터럽트라고 불리는 주기적인 인터럽트가 이런 목적을 위해 사용된다. 이 틱의 주기는 <FreeRTOSConfig.h>의 **configTICK_RATE_HZ** 값을 설정하면 된다.



예제 1) 우선 순위가 다른 두 개의 태스크 동작

두 개의 태스크의 우선순위는 Task1은 1, Task2는 2해서 생성하고 스케줄러를 동작시켜 멀티 테스킹이 어떻게 동작하는지 알아본다.

Task1은 주기적으로 LED를 0xFF와 0x00를 점멸한다.

```
static void vTask1(void *pvParam)
{
    for(;;)
    {
        vLED_On(0xFF);
        vDelay(1000000);
        vLED_On(0x00);
        vDelay(1000000);
    }
}
```

Task2은 Count 변수를 주기적으로 하나씩 증가해서 FND로 출력해서 보여준다.

```
static void vTask2(void *pvParam)
{
    unsigned portLONG Count = 0;

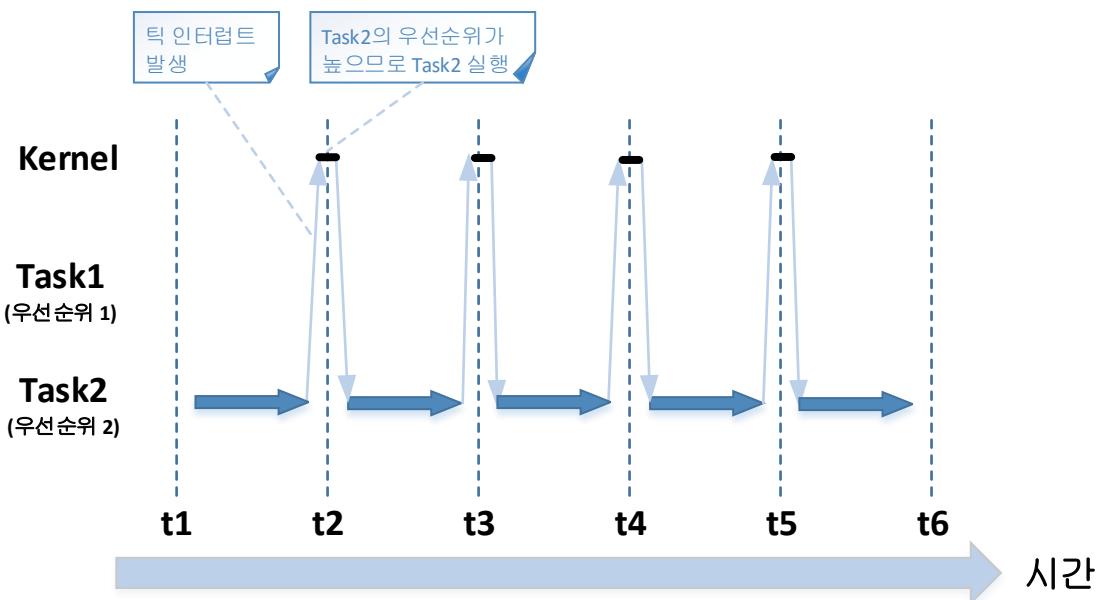
    for(;;)
    {
        vFND_Write(Count % 10, 0, 0);
        vFND_Write(Count / 10 % 10, 1, 1);
        vFND_Write(Count / 100 % 10, 2, 0);
        vFND_Write(Count / 1000 % 10, 3, 0);
        vDelay(1000000);
        Count++;
    }
}
```

Task1, Task2 를 생성해서 실행시킨다.

```
void vTask_SimpleTask(void)
{
    // 태스크 생성
    xTaskCreate(vTask1,                                // 태스크를 실행할 함수 포인터.
                "Task1",                               // 태스크 이름
                240,                                    // 스택의 크기
                NULL,                                   // 태스크 매개변수
                1,                                     // 태스크의 우선순위
                NULL);                                // 태스크 핸들
    );

    // 태스크 생성
    xTaskCreate(vTask2,                                // 태스크를 실행할 함수 포인터.
                "Task2",                               // 태스크 이름
                240,                                    // 스택의 크기
                NULL,                                   // 태스크 매개변수
                2,                                     // 태스크의 우선순위
                NULL);                                // 태스크 핸들
    );
}
```

Task2의 우선순위(2)가 Task1의 우선순위(1) 보다 높기 때문에 Task2가 항상 동작해서 FND의
에 표시되는 값은 증가하고 있지만 Task1의 LED 점멸은 전혀 일어나지 않는다.

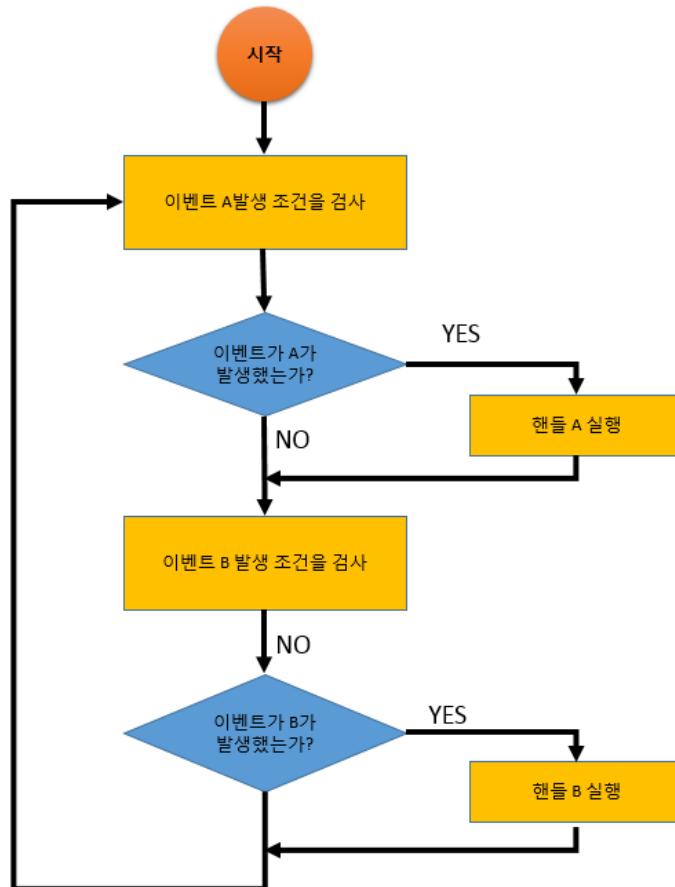


5.1.5 Not Running 상태의 확장

지금까지의 태스크는 항상 기다림 또는 기다림 없이 지속적으로 코드를 실행했었다. 이런 태스크를 연속 처리 태스크(Continuous processing task)라고 한다. 이 태스크보다 우선 순위가 낮은 태스크는 실행할 기회를 가지기 어렵게 되고, 다른 우선 순위의 태스크를 실행 시키기 위해서는 이 태스크의 우선순위가 매우 낮아야 한다.

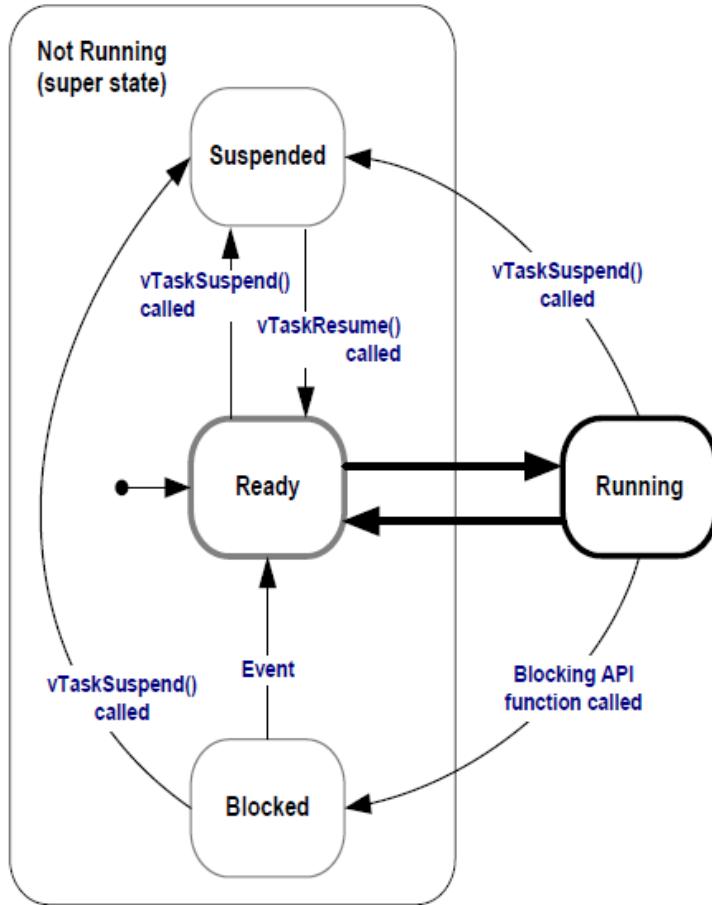
이런 문제점을 해결하고, **태스크 실행의 효율성을 높이기 위해서는 이벤트 기반의 코딩이 필요하다.** 이벤트 기반으로 코딩한 태스크는 이벤트가 발생했을 경우에만 일을 실행(Running 상태)하고 이벤트가 발생하기 전에는 Blocked 상태로 있다. 스케줄러는 항상 실행할 준비가 되어 있고, 우선순위가 가장 높은 태스크를 실행시킨다. 그러므로 우선 순위가 높은 태스크가 있더라도 실행할 준비가 되어있지 않다면 태스크(Blocked 상태의 태스크)는 실행시키지 않고 보다 낮은 우선 순위를 가진 태스크를 실행시키게 된다. 이렇게 함으로써 낮은 우선 순위를 가진 태스크가 높은 우선순위를 가진 태스크에 의해 항상 선점되지 않고 동시에 같이 실행 할 수 있게 된다.

RTOS를 사용하지 않는 경우에는, 이벤트를 기다리는 동안에는 루프에 의해 이벤트의 발생을 감시하는데 루프를 이용하여 감시하는 것은(폴링에 의한 방식) 폴링 속도에 따라 응답 속도가 일정하지 않고, 또한 감시를 위해 CPU를 동작시키기 때문에 CPU에 부하가 많이 걸리게 된다.



OS 가 없을 경우의 이벤트 처리

지금까지, 태스크의 상태를 두 가지(Running / Not Running)로만 구분을 했지만, Not Running 상태를 3가지 상태(Blocked/Suspended/Ready)로 확장하면 전체 상태는 다음과 같다



■ Blocked 상태

태스크가 이벤트를 기다리고 있는 상태이다.

이 상태는 두가지의 다른 종류의 이벤트를 기다린다.

- 1) 시간 관련(Temporal(Time-related))이벤트: 딜레이 시간 만료, 또는 특정 시간 도달등에 대한 이벤트
- 2) 동기(Synchronization) 이벤트: 다른 태스크나 인터럽트에 의한 이벤트로 FreeRTOS에서는 큐, 세마포어, 뮤텍스를 동기 이벤트가 발생하도록 사용할 수 있다.

■ Suspended 상태

Suspended 상태에서는 스케줄링이 되지 않는다. 이 상태는 오직 vTaskSuspend() API를 통해서 가능하며, vTaskResume()/vTaskResumeFromISR() API를 통해서 빠져 나올 수 있다.

■ Ready 상태

이 상태는 Blocked/Suspended 상태가 아닌 Running 상태가 되기 위한 준비상태이다. 그러므로 현재 실행하고 있지는 않지만 실행 할 수 있는 준비가 된 상태이다.

예제 1) Blocked 상태를 사용한 딜레이 생성

이전의 예제에서 사용한 딜레이는 Count 값을 증가 시키면서 Count값이 0이 되었는지 폴링(Polling)하는 것이다. 그래서 딜레이를 하는 동안 실질적으로 태스크는 대기 상태가 아닌 Count를 폴링하는 실행 상태로 동작하게 된다. 그래서 우선순위가 낮은 태스크는 딜레이 하는 동안에도 실행이 될 수가 없다. 예제2 에서와 같이 Task1은 전혀 실행이 되지 못하는 상황이 된다. 폴링 형태의 프로그램은 매우 비효율적이며 프로세스 사이클을 낭비하게 된다.

여기서 폴링 형태의 Delay함수를 FreeRTOS에서 제공하는 vTaskDelay() API로 교체한다. 이 함수는 특정 시간(틱)동안 태스크를 Blocked 상태로 만든다. 그러므로 실제로 필요한 시간만 실행하고 나머지 시간은 Blocked 상태로 있게 된다.

void vTaskDelay(portTickType xTicksToDelay);	
xTicksToDelay	태스크가 Ready 상태로 돌아오기 전 Blocked 상태로 있을 시간(틱)의 개수
	틱의 주기는 portTICK_RATE_MS에 의존한다.

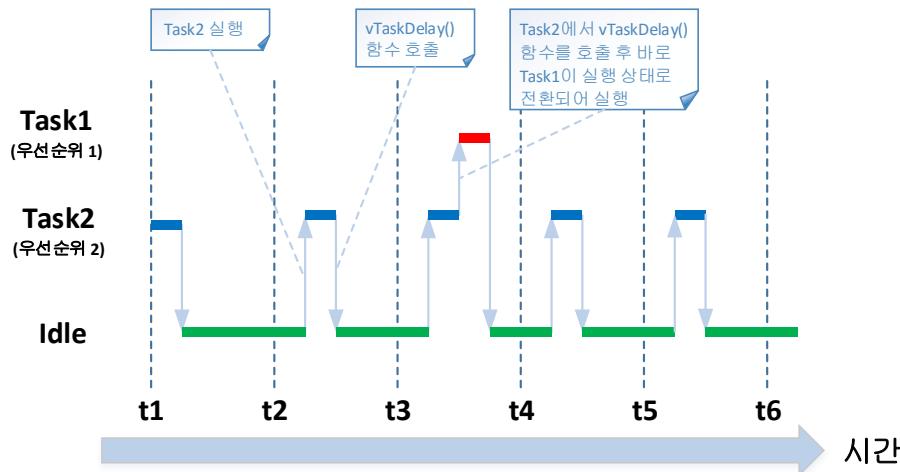
Task1 : vTaskDelay() 함수를 사용하여 LED를 0xFF와 0x00를 점멸한다.

```
static void vTask1(void *pvParam)
{
    for(;;)
    {
        vLED_On(0xFF);
        // vDelay(1000000);
        vTaskDelay(500/portTICK_PERIOD_MS);
        vLED_On(0x00);
        // vDelay(1000000);
        vTaskDelay(500/portTICK_PERIOD_MS);
    }
}
```

Task2은 vTaskDelay() 동안 딜레이하고 Count 하나씩 증가해서 FND로 출력해서 보여준다.

```
static void vTask2(void *pvParam)
{
    unsigned portLONG Count = 0;

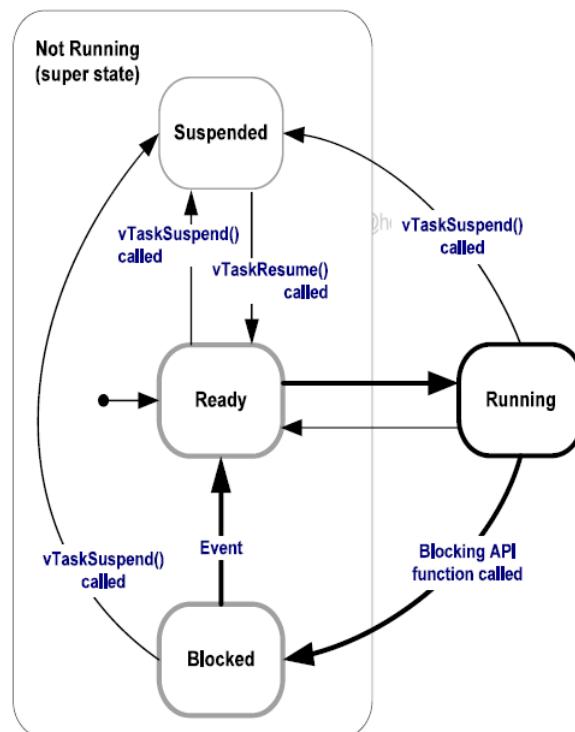
    for(;;)
    {
        vFND_Write(Count % 10, 0, 0);
        vFND_Write(Count / 10 % 10, 1, 1);
        vFND_Write(Count / 100 % 10, 2, 0);
        vFND_Write(Count / 1000 % 10, 3, 0);
        //vDelay(1000000);
        vTaskDelay(100/portTICK_PERIOD_MS);
        Count++;
    }
}
```



위의 그림에서 Idle 태스크는 스케줄러가 시작 될 때 생성되며 어떠한 태스크도 Ready 상태에 없을 때 Idle 태스크가 실행된다.

Task2는 일을 처리하고 vTaskDelay()를 호출하여 Blocked 상태로 들어가면 Idle 태스크가 실행이 된다. 만약에 Task1이 Ready 상태에 있다면 Task2가 vTaskDelay()를 호출할 때 Task1이 실행되게 된다. 다시 Task1이 vTaskDelay()를 호출하면 Task1은 Blocked 상태로 바뀌고 Idle 태스크가 동작하게 된다.

폴링을 사용하는 Delay() 대신에 vTaskDelay()를 사용함으로서 우선 순위가 높은 태스크만 동작을 했다가 우선 순위에 상관없이 두 개의 태스크가 동작하는 것을 볼 수가 있다.



위의 그림은 상태의 변화를 그린 것이다. 굵은 화살표가 이 예제에서의 태스크의 상태가 변화는 방향을 표현하고 있다. 태스크는 Running -> Blocked -> Ready -> Running상태를 반복하게 된다.

● vTaskDelayUntil() API

vTaskDelayUntil() 은 vTaskDelay()와 유사하지만 vTaskDelay()는 이 함수가 호출되는 때를 기준으로 매개변수로 전달되는 상대적인 시간(틱)만큼 Blocked 상태로 들어가지만 vTaskDelayUntil()은 절대적인 시간까지 태스크를 Blocked 상태로 만들고 다시 빠져나온다.

<code>void vTaskDelayUntil(portTickType * pxPreviousWakeTime, portTickType xTimeIncrement);</code>	
<code>pxPreviousWakeTime</code>	가장 최근에 태스크가 Blocked 상태에서 깨어났을 때의 시간(틱)값. 이 값은 포인터로 포인터가 가르키는 곳의 값은 vTaskDelayUntil() 함수 내부에서 자동으로 Blocked에서 빠져 나올 때의 시간이 갱신된다.
<code>xTimeIncrement</code>	가장 최근에 깨어난 시간(pxPreviousWakeTime) + xTimeIncrement 을 더한 시간후에 Blocked 상태에서 깨어나게 한다.

Task2에서 vTaskDelayUntil() 사용하여 분/초를 나타내는 시계를 구현해 보도록 한다.

예제 2) 정확한 1초 딜레이를 갖는 타이머 구현

매번 깨어날 때의 시간 xLastWakeTime에서 1초(1000ms) 후에 깨어나도록 해서 타이머를 구현한다.

```
static void Task2(void * pvParam)
{
    unsigned portLONG Count = 0;
    portTickType xLastWakeTime;

    /* 초기 시작시간을 위해 xLastWakeTime에 현재의 시간을 알아낸다. */
    xLastWakeTime = xTaskGetTickCount();

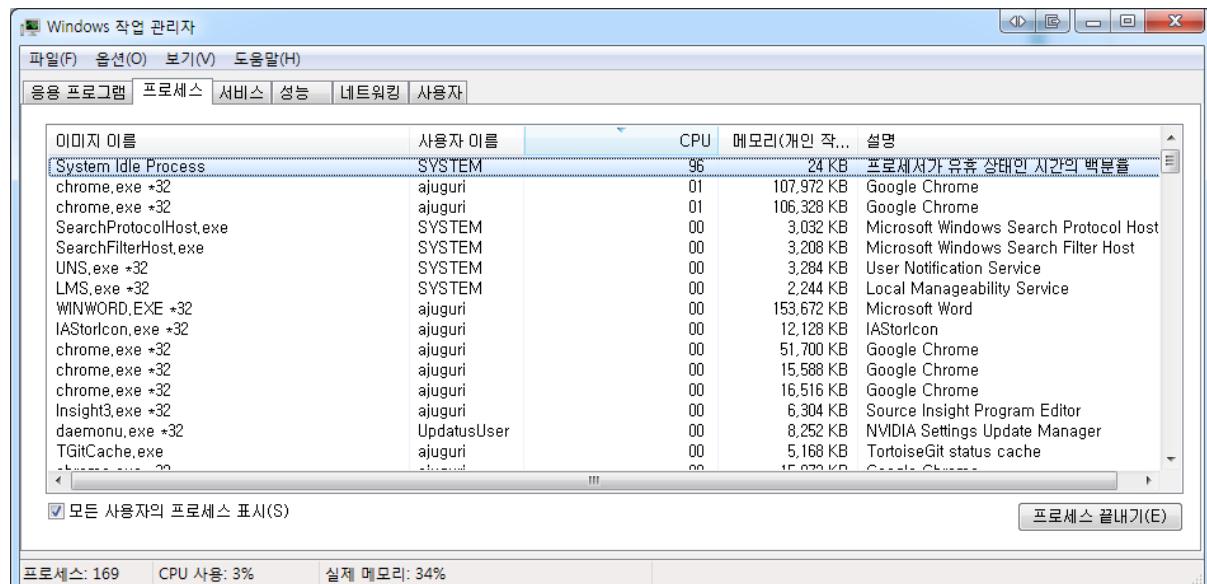
    for(;;)
    {
        vFND_Write( Count % 10, 0, 0);                                // 초의 1의 자리
        vFND_Write( Count / 10 % 6, 1, 0);                            // 초의 10의 자리
        vFND_Write( Count / 60 % 10, 2, 1);                          // 분의 1의 자리
        vFND_Write( Count / 600 % 6, 3, 0);                           // 분의 10의 자리
        vTaskDelayUntil(&xLastWakeTime, 1000/portTICK_PERIOD_MS);
        Count++;
    }
}
```

5.1.6 Idle 태스크

대부분의 태스크들은 Blocked 상태에서 시간을 보낸다(이벤트 기반의 프로그래밍일 경우). 이 상태에서는 실행하지도 되지도 못하고 스케줄러의 대상이 되지도 못한다.

프로세서는 항상 실행 할 무엇인가가 있어야 한다. 다시 말해 적어도 하나의 태스크가 Running 상태로 있어야 한다는 것이다. 그래서 vTaskStartScheduler() 를 호출할 때 만들어지는 태스크가 있는데 이것이 Idle 태스크이다.

Idle 태스크는 가장 낮은 우선순위를 가지고 있다. 그래서 다른 어떠한 태스크가 Running 상태로 전환하는데 전혀 영향을 미치지 않는다. 단지 실행할 수 있는 어떠한 태스크도 존재하지 않는 경우에만 Running 상태로 전환한다.



이미지 이름	사용자 이름	CPU	메모리(개인 작...)	설명
System Idle Process	SYSTEM	96	24 KB	프로세서가 유동 상태인 시간의 백그라운드
chrome.exe +32	ajuguri	01	107,972 KB	Google Chrome
chrome.exe +32	ajuguri	01	106,328 KB	Google Chrome
SearchProtocolHost.exe	SYSTEM	00	3,092 KB	Microsoft Windows Search Protocol Host
SearchFilterHost.exe	SYSTEM	00	3,208 KB	Microsoft Windows Search Filter Host
UNS.exe +32	SYSTEM	00	3,284 KB	User Notification Service
LMS.exe +32	SYSTEM	00	2,244 KB	Local Manageability Service
WINWORD.EXE +32	ajuguri	00	153,672 KB	Microsoft Word
IAStorIcon.exe +32	ajuguri	00	12,128 KB	IAStorIcon
chrome.exe +32	ajuguri	00	51,700 KB	Google Chrome
chrome.exe +32	ajuguri	00	15,588 KB	Google Chrome
chrome.exe +32	ajuguri	00	16,516 KB	Google Chrome
Insight3.exe +32	ajuguri	00	6,304 KB	Source Insight Program Editor
daemonu.exe +32	UpdatusUser	00	8,252 KB	NVIDIA Settings Update Manager
TGitCache.exe	ajuguri	00	5,168 KB	TortoiseGit status cache

원도우 7에서의 Idle 태스크

■ Idle Task Hook Functions

Idle 태스크에서 사용자가 원하는 기능을 특정 함수를 통해서 시킬 수 있다. 이 특정 함수가 Idle Hook Function 이다. 이 함수는 Idle 태스크에서 Idle task loop 반복당 한번 호출이 된다.

함수의 원형은 반드시 다음과 같아야 한다.

```
void vApplicationIdleHook(void);
```

이 함수를 사용하기 위해서는 <FreeRTOSConfig.h>의 configUSE_IDLE_HOOK 를 반드시 1로 만들어야 한다.

이 함수는 다음과 같은 용도로 사용할 수 있다.

- 낮은 우선순위로 백그라운드에서 연속적으로 프로세스를 실행.
- 남은 프로세스 여유를 측정.
- 프로세스를 저전력모드로 사용.

예제 1) Idle Task Hook 사용예

Idle Task Hook 함수에서 ullidleTickCount 값을 증가시키고 Task1에서 ullidleTickCount 을 출력하는 예제이다.

```

void vTask_IdleHook(void)
{
    // 태스크 생성
    xTaskCreate(vIdleHook_Task,
                "IdleHook_Task",
                240,
                NULL,
                1,
                NULL
            );
}

```

```

static void vIdleHook_Task(void *pvParam)
{
    for(;;)
    {
        // Idle Hook 테스트용
        xSerial_PutString("\r\n");
        xSerial_PutChar((ulIdleCycleCount / 100000000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 10000000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 1000000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 100000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 10000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 1000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 100 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 10 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount % 10 + '0', 0);

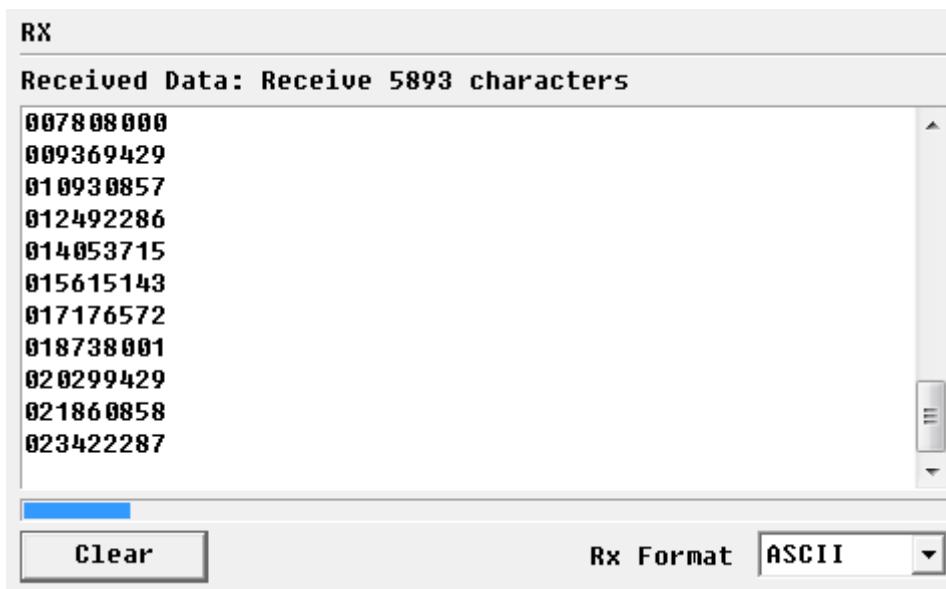
        vLED_On(0xAA);
        vTaskDelay(500/portTICK_PERIOD_MS);
        vLED_On(0x55);
        vTaskDelay(500/portTICK_PERIOD_MS);
    }
}

```

```

void vApplicationIdleHook(void)
{
    ulIdleCycleCount++;
}

```



ulldleCycleCount의 값이 약 1,560,000씩 증가하는 것을 알 수 있다. 그러므로 1초당 vApplicationIdleHook() 함수가 1,560,000 호출된 것이다.

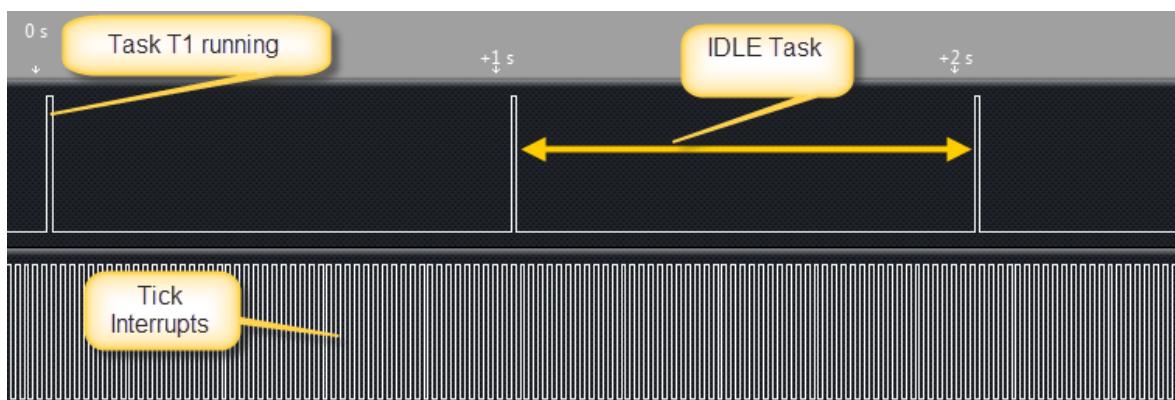
예제 2) Idle Task Hook를 이용한 단순 저전력(Low power)모드 사용

STM32F103ZET에서 제공하는 저전력 모드는 다음과 같이 3가지 모드(Sleep, Stop, Standby)가 있다. 여기서 실행할 태스크가 없는 동안 Idle Task가 동작하므로 이 기간동안에 Sleep 모드로 들어가서 CPU에서 사용하는 전력소비를 줄여 보도록 한다.

Sleep 모드에서는 어떠한 인터럽트가 발생해도 Sleep 모드에서 빠져 나오게 된다. 그러면 FreeRTOS에서는 Time tick이 주기적으로(1ms으로 설정됨) 발생하므로 이 때마다 CPU가 깨어나게 된다.

Table 11. Low-power mode summary

Mode name	Entry	wakeup	Effect on 1.8V domain clocks	Effect on V _{DD} domain clocks	Voltage regulator
Sleep (Sleep now or Sleep-on - exit)	WFI	Any interrupt	CPU clock OFF no effect on other clocks or analog clock sources	None	ON
	WFE	Wakeup event			
Stop	PDDS and LPDS bits + SLEEPDEEP bit + WFI or WFE	Any EXTI line (configured in the EXTI registers)	All 1.8V domain clocks OFF	HSI and HSE oscillators OFF	ON or in low-power mode (depends on <i>Power control register (PWR_CR)</i>)
Standby	PDDS bit + SLEEPDEEP bit + WFI or WFE	WKUP pin rising edge, RTC alarm, external reset in NRST pin, IWDG reset			OFF



```

void vTask_IdleHook(void)
{
    // 태스크 생성
    xTaskCreate(vIdleHook_Task,
                "IdleHook_Task",
                240,
                NULL,
                1,
                NULL
            );
}

```

```

static void vIdleHook_Task(void *pvParam)
{
    for(;;)
    {
        // Idle Hook 테스트용
        xSerial_PutString("\r\n");
        xSerial_PutChar((ulIdleCycleCount / 100000000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 10000000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 1000000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 100000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 10000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 1000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 100 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 10 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount % 10 + '0', 0);

        vLED_On(0xAA);
        vTaskDelay(500/portTICK_PERIOD_MS);
        vLED_On(0x55);
        vTaskDelay(500/portTICK_PERIOD_MS);
    }
}

```

```

void vApplicationIdleHook(void)
{
    ulIdleCycleCount++;
    __WFI();           // Sleep 모드로 진입
}

```

RX**Received Data: Receive 176726 characters**

```

000032670
000033880
000035090
000036300
000037510
000038720
000039930
000041140
000042350
000043560
000044770

```

ulIdleCycleCount 의 값이 약 1200씩 증가하는 것을 알 수 있다 그러므로 1초당 vApplicationIdleHook() 함수가 1200 호출된 것이다.

이전 예제에서는 약 1,560,000 번 호출되었었는데 확실히 줄은 것을 확인 할 있다. CPU는 Time Tick 동안에 Sleep 모드로 들어가서 동작을 하지 않은 것이다.

예제 3) FreeRTOS의 Tickless Idle Mode를 이용한 저전력(Low power)모드 사용

Idle 태스크가 실행 될 때 호출되는 vApplicationIdleHook() 함수에서 Sleep 모드로 들어갈 경우에 매번 Time Tick 이 발생할 때마다 Sleep 모드에서 깨어나게 된다. 가장 효율적인 방법은 Sleep 모드에 있는 동안 CPU 가 깨어나지 않는 것이다. 그러나 실직적으로 FreeRTOS 가 Idle 태스크가 돌아가고 있는 동안에도 커널 내부에서 여전히 해야 될 일이 있다.

FreeRTOS 에서는 Tickless Idle Mode 라는 것을 제공한다. 이것은 Idle 태스크가 실행이 되면 vPortSuppressTicksAndSleep() 함수를 호출한다. 이 함수는 Time Tick 를 적게 발생이 되도록 만들고 CPU 를 Sleep 모드로 만들어 버린다. 이렇게 해서 되도록이면 CPU 를 깨어나지 않도록 하고 있다.

이 기능은 <FreeRTOSConfig.h>의 #define configUSE_TICKLESS_IDLE 1 로 만들어 주면 된다.

FreeRTOSConfig.h

```
#define configUSE_TICKLESS_IDLE 1
```

```
void vTask_IdleHook(void)
{
    // 태스크 생성
    xTaskCreate(vIdleHook_Task,           // 테스크를 실행할 함수 포인터.
                "IdleHook_Task",        // 테스크 이름
                240,                   // 스택의 크기
                NULL,                 // 테스크 매개변수
                1,                     // 테스크의 우선순위
                NULL);                // 테스크 핸들
}
```

```
static void vIdleHook_Task(void *pvParam)
{
    for(;;)
    {
        // Idle Hook 테스트용
        xSerial_PutString("\r\n");
        xSerial_PutChar((ulIdleCycleCount / 1000000000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 10000000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 1000000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 100000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 10000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 1000 % 10 + '0', 0);
        xSerial_PutChar((ulIdleCycleCount / 100 % 10 + '0', 0);
```

```

        xSerial_PutChar(ulIdleCycleCount /100 % 10 + '0', 0);
        xSerial_PutChar(ulIdleCycleCount /10 % 10 + '0', 0);
        xSerial_PutChar(ulIdleCycleCount % 10 + '0', 0);

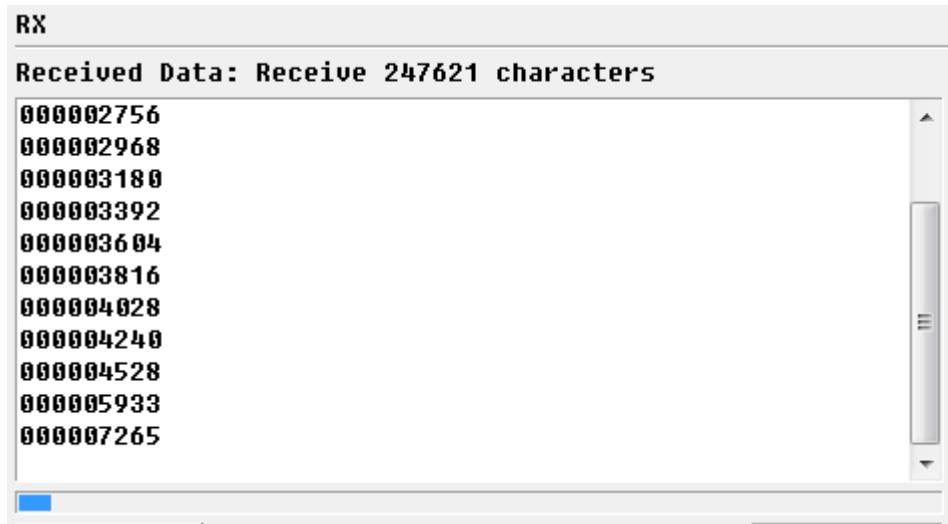
        vLED_On(0xAA);
        vTaskDelay(500/portTICK_PERIOD_MS);
        vLED_On(0x55);
        vTaskDelay(500/portTICK_PERIOD_MS);
    }
}

```

```

void vApplicationIdleHook(void)
{
    ulIdleCycleCount++;
}

```



ulIdleCycleCount 의 값이 약 200씩 증가하는 것을 알 수 있다 그러므로 1초당

vApplicationIdleHook() 함수가 200 호출된 것이다.

이전 예제에서 vApplicationIdleHook () 호출시에 Sleep 모드로 들어갈 경우 에서는 약 1200 번 호출되었던 것에 비해서도 더 많이 줄은 것을 확인 할 있다. CPU는 더 많은 시간 동안 Sleep 모드로 들어간 것을 알 수가 있다.

5.1.7 우선순위 변경

- **vTaskPrioritySet()**: 태스크 우선순위 변경

이 API는 태스크의 우선순위를 변경한다. 함수의 원형은 다음과 같다.

```
void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority );
```

pxTask	우선 순위를 바꾸고자 하는 태스크의 핸들.
--------	-------------------------

	태스크 자신의 우선순위를 바꾸고자 하면 여기에 NULL값을 넣는다
--	---

uxNewPriority	우선 순위 값
---------------	---------

- **uxTaskPriorityGet()**: 태스크 우선순위 획득.

이 API는 태스크의 우선순위 값을 얻어온다. 함수의 원형은 다음과 같다.

```
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

pxTask	우선 순위값을 알고자 하는 태스크의 핸들
--------	------------------------

반환값	우선 순위 값
-----	---------

5.1.8 태스크 삭제

- **vTaskDelete()**: 태스크 삭제

이 API는 태스크를 삭제한다.

```
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

pxTaskToDelete	삭제하고자 하는 태스크의 핸들
----------------	------------------

5.1.9 태스크 중지

- **vTaskSuspend ()**: 태스크 중지

이 API는 태스크를 태스크를 중지한다.

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend );
```

xTaskToSuspend	중지하고자 하는 태스크의 핸들, 만약 NULL 값을 사용하면 현재 자신의 태스크가 중지된다.
----------------	---

5.1.10 태스크 재시작

- **vTaskResume ()**: 태스크 재시작

이 API는 중지된 태스크를 다시 재시작 시킨다.

```
void vTaskResume( TaskHandle_t xTaskToResume );
```

xTaskToResume	재시작시킬 태스크의 핸들
---------------	---------------

실습 1 태스크 Suspend/Resume/Change Priority

시리얼로 명령을 받아서 Task1(태스크번호 1), Task2(태스크 번호 2)를 조정한다.

- 시리얼로 데이터를 받을 수 있는 테스크를 생성을 한다.
- 시리얼로 받은 명령을 처리하기 위한 태스크에서 Task1 과 Task2를 생성한다. 이 두 태스크의 생성 초기에는 Suspend 모드로 있어야 한다.
- 아래와 같은 데이터를 받으면 그 명령대로 Task1/2를 조정한다.

'\$'	'R'	'T'	'T'	'S'	'K'	'S' / 'R'/P'	' '	' '	'**'
시작 문자	송신자 식별자	명령어				'S' : Suspend	태스크 번호	사용안함	종료문자
						'R' : Resume	태스크 번호	사용안함	
						'P' : Change Priority	태스크 번호	우선순위	

실습 2 LED 디밍 제어 태스크

LED 8개에 대해서 각각의 PWM를 출력하여 LED의 밝기를 제어하고, 표시 형태는 전격 Z작전의 키드의 LED와 비슷하게 표시되도록 한다.

- LED 8개 각각에 대해서 On/Off를 제어하는 태스크 8개를 생성한다.
- 각각의 태스크에서 On/Off 시간을 조절하여 PWM 파형을 만들어 낸다.

5.2 큐 관리(Queue Management)

FreeRTOS에서 사용하는 어플리케이션들은 각각의 독립적인 태스크로 구조화된다. 이런 독립적인 태스크들은 서로 정보를 주고 받아야 하고 그렇게 함으로써 전체적으로 원활한 시스템 기능을 제공할 수 있다.

큐는 이 FreeRTOS에 있어서 통신과 동기화의 기본 요소로서 사용되고 있다

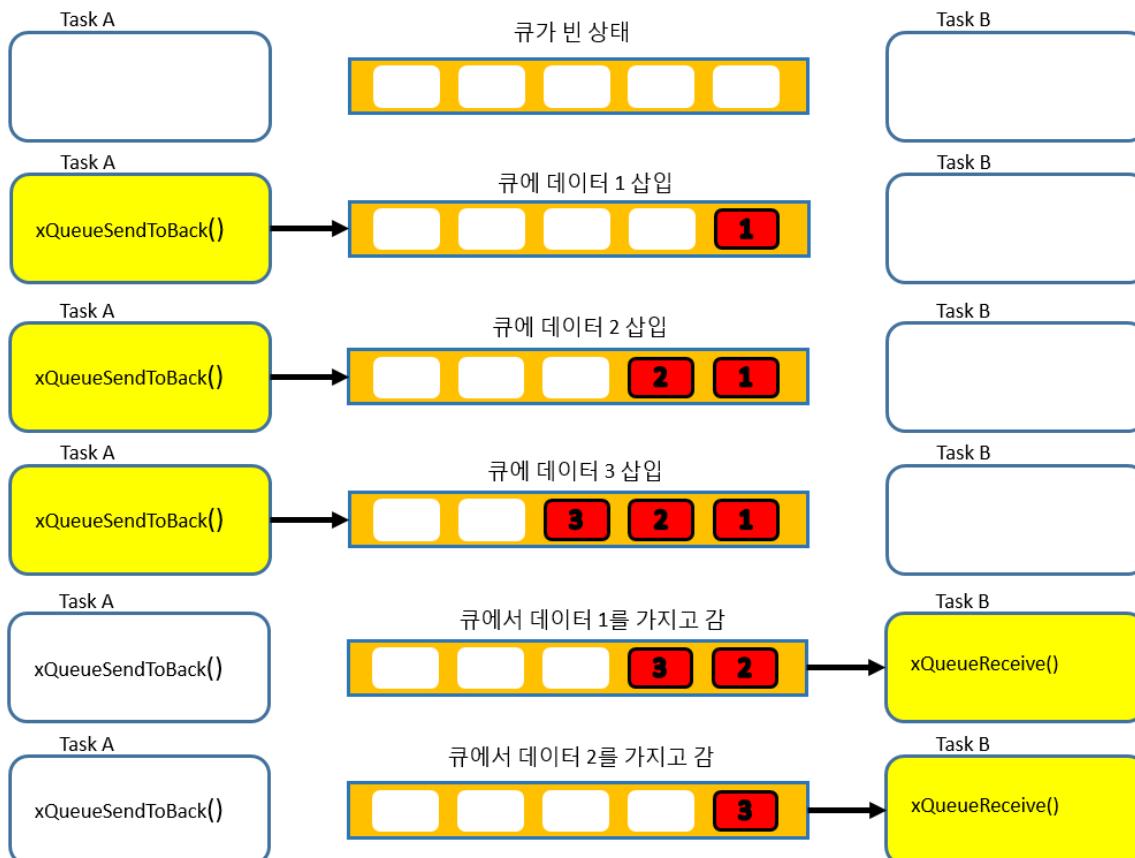
5.2.1 큐의 특징

1) 데이터의 저장

큐는 고정 크기의 데이터를 가질 수가 있다. 길이(Length)와 데이터의 최대 크기는 큐를 생성할 때 설정한다.

일반적으로 큐는 FIFO(First In First Out) 버퍼로서 사용되는데 꼬리(Tail)에 데이터를 쓰고, 머리(Head)에서 데이터를 가지고 온다. 이 반대로 가능한다.

큐에 데이터를 쓰게 되면 그 데이터가 큐에 저장이 되고, 큐에서 읽게 되면 그 값은 큐에서 제거가 된다.



2) 큐 접근

큐는 특정한 태스크에 의해 할당되거나 소유되는 것이 아니라 자신만의 영역을 가진 객체이다. 그러므로 어떤 태스크도 큐에 읽고 쓸 수가 있다.

3) 큐 읽기와 대기

큐에서 데이터를 읽을 경우, 데이터가 있으면 바로 데이터를 가지고 오면 된다. 그러나 데이터가 없을 경우에는 데이터가 들어올 때까지 Blocked 상태에서 기다리거나 일정 시간 동안 Blocked 상태에서 기다리다가 데이터가 들어오지 않으면 빠져 나와야 한다. 이 때 기다리는 일정 시간은 큐를 읽을 때 명시를 하면 된다.

Blocked 상태에서 큐로 데이터가 들어오면 Blocked 상태에서 대기 중이던 태스크는 Ready 상태로 자동 전환이 된다. 마찬가지로 일정 시간동안 Blocked 상태로 있는 태스크도 시간이 만료가 돼도 데이터가 들어오지 않으면 Ready 상태로 자동 전환된다.

하나의 큐에 여러 태스크가 데이터 읽기를 요청하면서 Blocked 상태로 있을 수가 있다. 이 때 큐로 데이터가 들어온다면 데이터를 기다리고 있는 태스크 중 어느 한 태스크는 깨어나서 데이터를 받아가야 한다. **여러 대기 태스크 중에서 Blocked 상태에서 깨어나는 규칙은 우선순위가 가장 높은 태스크가 먼저 깨어나게(Unblocked) 된다. 만약 우선순위가 같을 경우에는 가장 오랫동안 기다리고 있던 태스크가 깨어나게 된다.**

4) 큐 쓰기와 대기

읽기와 같이 큐 쓰기도 큐에 쓸 여유 공간이 있으면 바로 쓰지만 여유 공간이 없을 경우에는 여유 공간이 빌 때까지 Blocked 상태로 기다리거나, 일정 시간 동안 Blocked 상태에서 기다리다가 공간이 비지 않으면 빠져 나온다.

만약에 여러 태스크가 큐에 쓰기 위해서 Blocked 상태로 있을 때, 큐의 여유 공간이 생겨서 대기 중이던 태스크 중 하나가 깨어나야 한다면, **가장 우선순위가 높은 태스크가 먼저 깨어나서 Ready상태가 된다. 대기 중이던 태스크의 우선순위가 같다면 가장 오래 기다린 태스크가 깨어나게 된다**

5.2.2 큐 관련 API

- 큐의 생성 (`xQueueCreate()`)

```
QueueHandle_t xQueueCreate( unsigned portBASE_TYPE uxQueueLength, unsigned
portBASE_TYPE uxItemSize);
```

`uxQueueLength` 큐의 길이(큐에 들어가는 데이터 항목의 개수)

`uxItemSize` 데이터 항목의 바이트 단위 크기

- 큐에 데이터 삽입 (`xQueueSendToBack()` / `xQueueSendToFront()`)

큐에 데이터를 삽입은 하는데 `xQueueSendToBack()`의 큐의 꼬리에 데이터를 삽입하고 `xQueueSendToFront()`는 큐의 머리(Head)에 데이터를 삽입한다.

```
portBASE_TYPE xQueueSendToFront(QueueHandle_t xQueue, const void *
pvItemToQueue, portTickType xTicksToWait);
```

```
portBASE_TYPE xQueueSendToBack(QueueHandle_t xQueue,
const void * pvItemToQueue, portTickType xTicksToWait);
```

`xQueue` 큐 핸들

`pvItemToQueue` 큐에 넣은 항목의 포인터

`xTicksToWait` 큐가 꽉 찾을 경우 태스크가 Blocked 상태로 기다리는 시간(틱).

- 값이 0 이면 큐가 꽉 차있더라도 기다리지 않고 바로 빠져 나온다.

- INCLUDE_vTaskSuspend = 1 이고, 이 값이 portMAX_DELAY 이면 큐가 빌 때까지 무한히 기다린다.

- 큐에서 데이터 인출 (`xQueueReceive()` / `xQueuePeek()`)

큐에서 데이터를 가지고 오는데 머리(Head)에서 데이터를 가지고 온다. `xQueueReceive()`은 데이터를 가지고 온 후 데이터가 큐에서 삭제가 되지만 `xQueuePeek()`는 큐에서 삭제가 되지 않고 그래서 보존이 된다.

```
portBASE_TYPE xQueueReceive(QueueHandle_t xQueue, const void * pvBuffer,
portTickType xTicksToWait);
```

```
portBASE_TYPE xQueuePeek(QueueHandle_t xQueue, const void * pvBuffer,
portTickType xTicksToWait);
```

`xQueue` 큐 핸들

`pvBuffer` 큐에서 받아온 데이터를 넣을 메모리의 포인터

`xTicksToWait` 큐가 비었을 경우 태스크가 Blocked 상태로 기다리는 시간(틱).

- 큐에 들어있는 데이터 항목의 개수 (`uxQueueMessagesWaiting()`)

큐 안에 들어있는 데이터 항목의 개수를 알아낸다.

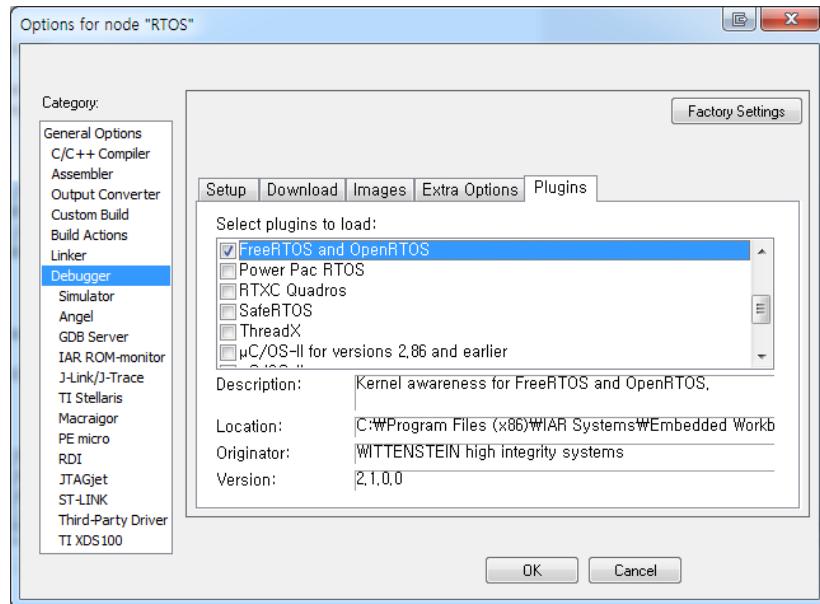
```
unsigned portBASE_TYPE uxQueueMessagesWaiting( QueueHandle_t xQueue );
```

`xQueue` 큐 핸들

`반환값` 큐 안에 들어있는 데이터 항목의 개수

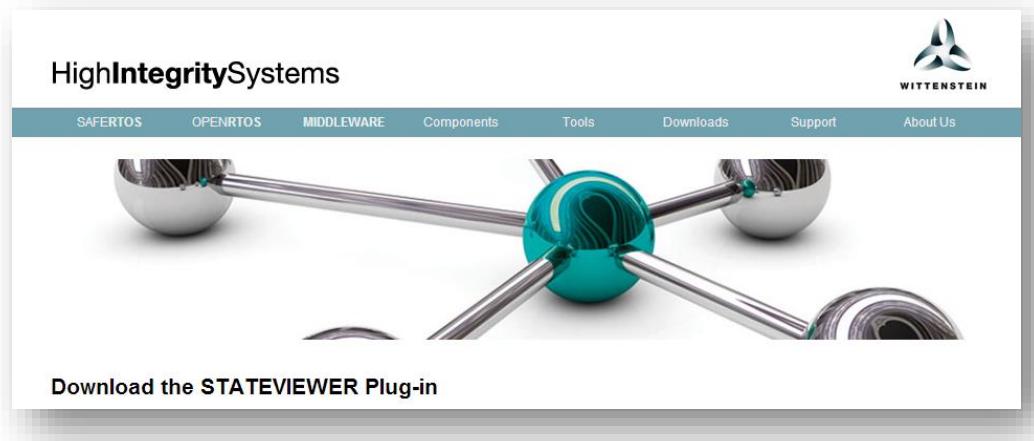
5.2.3 StateView 설치

IAR의 옵션 중에서 [Debug] -> [FreeRTOS and OpenRTOS]를 선택하면 FreeRTOS의 동작 상황을 쉽게 볼수 있는 플러그인을 동작 시킬 수 있다.



그러나 IAR 설치시 사용되는 버전이 옛날것이여서 현재 버전의 FreeRTOS를 사용시 Queue가 보이지 않는 현상이 발생한다. 이 플러그인의 최신 버전을 다음 사이트에서 다운 받아서 설치한다.

<http://www.highintegritysystems.com/down-loads/stateviewer-plug-in/>



- 설치 순서

- 1) IAR Embedded Workbench 동작을 중지시킨다.
- 2) [IAR 설치 폴더]\arm\plugins\rtos\OpenRTOS 이동한다.
- 3) 다운로드한 파일중에 OpenRTOSPlugin.dll, OpenRTOSPlugin.ewplug로 기존의 파일을 덮쳐쓴다.
- 4) 프로그램을 실행시킨다.

예제 1) 1초 간격으로 큐에 데이터 주고 받기.

main에서 큐(xQueue)를 생성하고 vSenderTask에서 1초 간격으로 데이터를 큐에 삽입한다. 그러면 vReceiverTask에서는 큐에 데이터가 들어오기를 기다렸다가 데이터가 들어오면 메시지를 표시한다.

큐와 vSenderTask, vReceiverTask를 생성한다.

```
QueueHandle_t xQueue;

void vQueue_SimpleQueue(void)
{
    // 큐의 생성
    xQueue = xQueueCreate(5,sizeof(portCHAR));
    vQueueAddToRegistry(xQueue, "xQueue");

    xTaskCreate(vSenderTask,           // 태스크를 실행할 함수 포인터.
                "Sender Task",      // 태스크 이름
                240,                 // 스택의 크기
                NULL,                // 태스크 매개변수
                1,                   // 태스크의 우선순위
                NULL                 // 태스크 핸들
            );

    // 수신 태스크 생성
    xTaskCreate(vReceiverTask,         // 태스크를 실행할 함수 포인터.
                "Receiver Task",    // 태스크 이름
                240,                 // 스택의 크기
                NULL,                // 태스크 매개변수
                1,                   // 태스크의 우선순위
                NULL                 // 태스크 핸들
            );
}
```

1초 간격으로 큐에 데이터를 삽입한다.

```
static void vSenderTask(void * pvParam)
{
    portCHAR cTemp = 'a';
    portBASE_TYPE xStatus;

    for(;;)
    {
        xStatus = xQueueSendToBack(xQueue , &cTemp, 0);

        if(xStatus == pdPASS)
        {
            xSerial_PutString("\r\nSended ");
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

큐에 데이터가 들어오기를 기다렸다가 데이터가 들어오면 시리얼로 메시지를 표시한다.

```
static void vReceiverTask(void * pvParam)
{
    portCHAR cReceivedData;
```

```

for(;;)
{
    // 큐에 데이터가 들어올 때까지 기다린다.
    xQueueReceive(xQueue, &cReceivedData, portMAX_DELAY);

    xSerial_PutString("\r\nReceived");

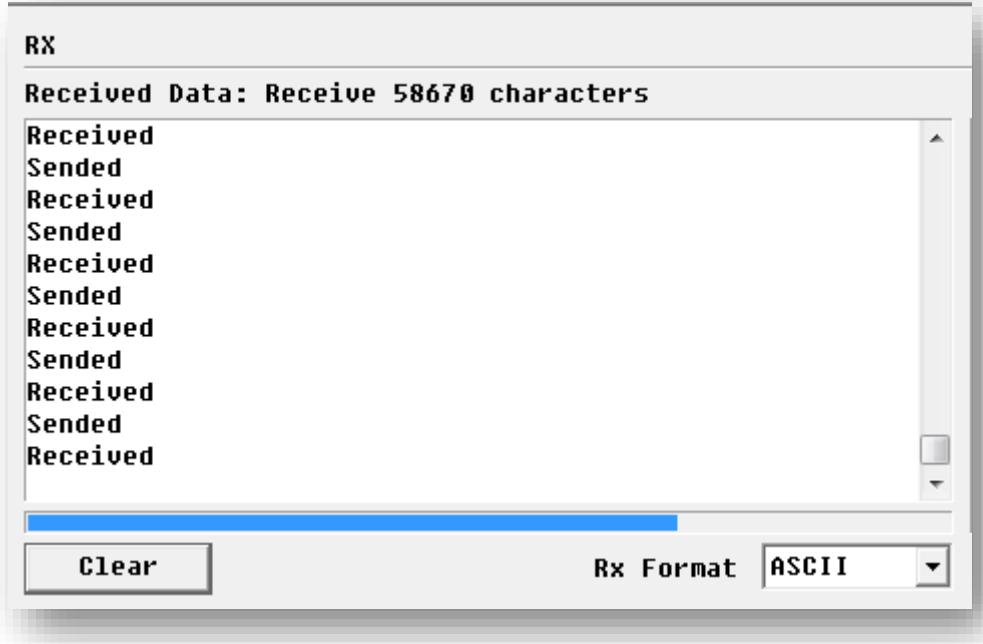
}

```

위의 프로그램의 결과는 아래 그림과 같다. 지금 같은 경우에는 vSenderTask와 vReceiverTask의 우선 순위가 같기 때문에 vSenderTask에서 데이터를 보내고 “Sended”를 표시한 다음 Block 상태로 되고 이때 Block 상태에 있던 vReceiverTask가 큐를 기다리다가 깨어나고 큐로부터 데이터를 받고 “Received”를 표시하게 된다.

만약에 vSenderTask가 vReceiverTask 보다 우선 순위가 높은 경우에는 먼저 “Received”가 표시되고 그 다음에 “Sended”가 표시된다. 이 경우에는 vSenderTask가 큐에 데이터를 보내면 데이터를 받은 큐는 태스크 전환을 시도한다. 만약에 자신 보다 높은 우선순위를 가진 태스크가 큐의 데이터를 기다리고 있으면 높은 우선순위를 가진 태스크로의 전환이 이루어진다. 그러므로 우선 순위가 높은 vReceiverTask가 큐에 데이터가 들어오자 마자 실행상태가 되고 “Received”라고 표시하고 다시 Block상태로 돌아가면 다시 vSenderTask가 다시 실행상태로 되고 “Sended”를 표시하게 된다.

참고로 현재의 태스크가 우선순위가 같은 Ready 상태에 있는 다른 태스크를 실행시키고자 한다면 taskYIELD()를 호출하면 된다.



예제 2) 2개의 태스크가 큐에 송신하고 1개의 태스크 수신하기 (우선순위)

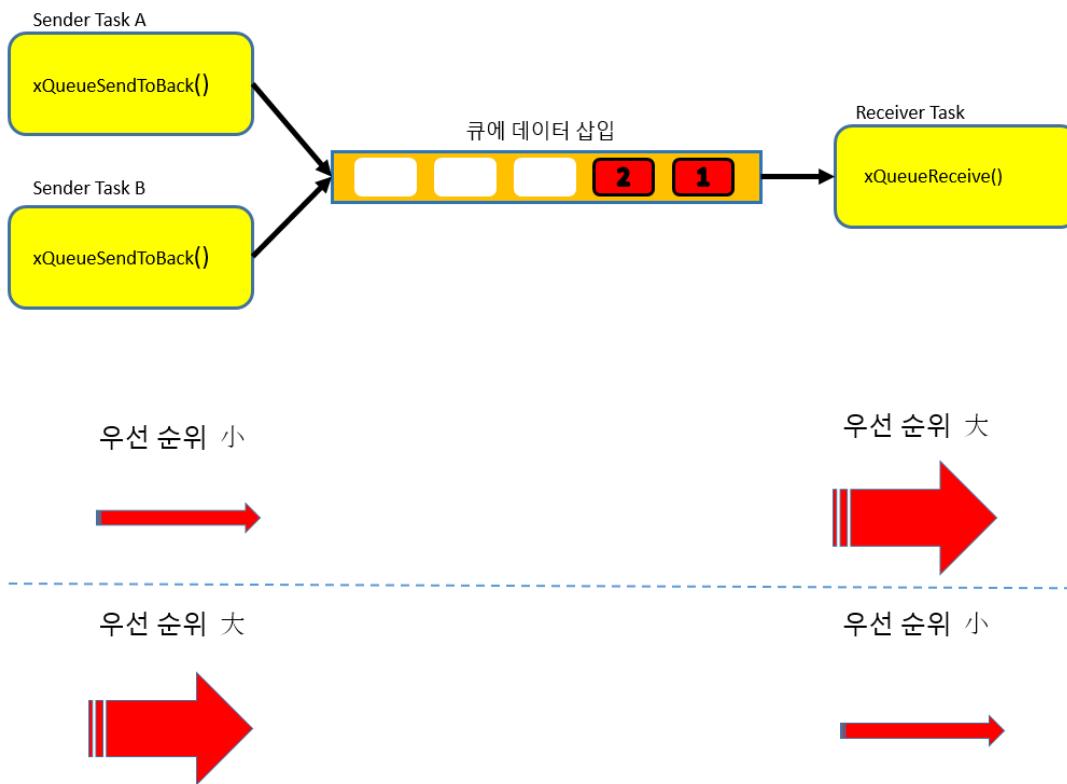
큐에 데이터를 송신하는 태스크를 2개 만들고 큐에서 데이터를 받는 태스크를 1개 만들어서 송/수신 데이터를 시리얼로 출력하는 프로그램을 한다.

1) 우선순위: 송신 데스크(Sender Task A/B) < 수신 태스크(Receiver Task)

- 수신 태스크의 우선 순위가 송신 태스크 보다 높기 때문에 큐에서 데이터를 가져가는 속도가 넣는 속도보다 높아서 큐가 넘치지 않아서 정상동작 하게 된다.

2) 우선순위: 송신 데스크(Sender Task A/B) > 수신 태스크(Receiver Task)

- 수신 태스크의 우선 순위가 송신 태스크 보다 낮기 때문에 큐에서 데이터를 가져가는 속도가 넣는 속도보다 낮아서 큐가 넘치는 현상이 발생한다. 두 개의 송신 태스크의 우선 순위가 같으면 라운드 로빙 방식으로 큐에 데이터를 넣으려 할 것이다.



```

QueueHandle_t xQueue;

void vQueue_2Sender1Receiver(void)
{
    xQueue = xQueueCreate(5,sizeof(portCHAR));
    vQueueAddToRegistry(xQueue, "xQueue");

    // 송신 태스크 A/B생성
    xTaskCreate(vSenderTask,
                "Sender Task A",
                240,
                (void *)'A',
                1,
                NULL
            );
    xTaskCreate(vSenderTask,
                "Sender Task B",
                // 태스크를 실행할 함수 포인터.
                // 태스크 이름
                // 스택의 크기
                // 태스크 매개변수
                // 태스크의 우선순위
                // 태스크 핸들
            );
}

```

```

        240,                      // 스택의 크기
        (void *)'B',              // 태스크 매개변수
        1,                        // 태스크의 우선순위
        NULL,                     // 태스크 핸들
    );

// 수신 태스크 생성
xTaskCreate(vReceiverTask,
            "Receiver Task",
            240,
            NULL,
            1,
            NULL
);
}

```

```

static void vSenderTask(void *pvParam)
{
    portCHAR cSendData = (portBASE_TYPE)pvParam;

    for(;;)
    {
        if(xQueueSendToBack(xQueue,&cSendData, 0) != pdPASS)
        {
            xSerial_PutString("Could not send to the queue.\r\n");
        }
        // 같은 우선 순위의 테스트가 Ready 상태에 있다면 실행을 그 테스크로 넘긴다.
        taskYIELD();
    }
}

```

```

static void vReceiverTask(void *pvParam)
{
    portCHAR cReceiveData;

    for(;;)
    {
        if(xQueueReceive(xQueue,&cReceiveData, 100) == pdPASS)
        {
            xSerial_PutString("\r\nReceived : ");
            xSerial_PutChar(cReceiveData, 10);
        }
        else
        {
            xSerial_PutString("Could not receive from the queue.\r\n");
        }
    }
}

```

RX

Received Data: Receive 393846 characters

```
eue.  
CoCould not suld not sendend to the to the queuqueue.  
e..■Could not se■nd to the Could not sequeue.  
nd Could not seto the queund to the que.  
eue.  
CCould not seould not send to the qund to the que.  
ueue..■Could not s■end to the Could not squeue.  
end Could not sto the queueend to the q.  
ueue.  
CCould not seould not send to the qund to the que.  
ueue..■Could not s■end to the Could not sq
```

Rx Format

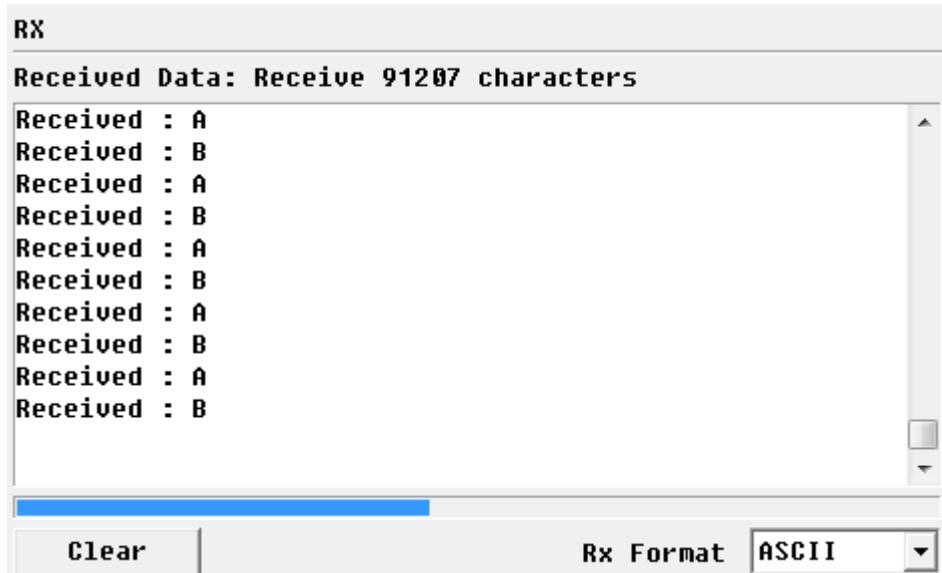
Sender Task A = Sender Task B = Receiver Task

RX

Received Data: Receive 3972055 characters

```
Could not send to the queue.  
Could not send to the queue.
```

Sender Task A > Sender Task B = Receiver Task



Sender Task A = Sender Task B < Receiver Task

예제 3) 1개의 테스크가 큐에 송신하고 2개의 태스크가 수신하기

큐에 데이터를 송신하는 태스크를 1개 만들고 2개의 태스크가 큐에서 데이터 받아 가도록 한다. 이 때 받는 태스크의 우선 순위를 변경하면 어떻게 되는지 확인한다.



```

QueueHandle_t xQueue;
void vQueue_1Sender2Receiver(void)
{
    xQueue = xQueueCreate(5,sizeof(portCHAR));
    vQueueAddToRegistry(xQueue, "xQueue");

    // 송신 태스크 생성
    xTaskCreate(vSenderTask,
                "Sender Task",
                240,
                (void *)'S',
                1,
                NULL
    );
}

// 수신 태스크 A/B 생성
xTaskCreate(vReceiverTask,
            "Receiver Task A",
            240,
            (void *)'A',
            2,
            NULL
);

xTaskCreate(vReceiverTask,
            "Receiver Task B",
            240,
            (void *)'B',
            2,
            NULL
);
}

```

```

static void vSenderTask(void *pvParam)
{
    portCHAR cSendData = (portBASE_TYPE)pvParam;

    for(;;)
    {
        if(xQueueSendToBack(xQueue,&cSendData, 0) != pdPASS)
        {
            xSerial_PutString("Could not send to the queue.\r\n");
        }
        vTaskDelay(100 / portTICK_PERIOD_MS);
    }
}

```

```

static void vReceiverTask(void *pvParam)
{
    portCHAR cReceiveData;
    portCHAR cReceiver = (portLONG)pvParam;

    for(;;)
    {
        if(xQueueReceive(xQueue,&cReceiveData, 1000) == pdPASS)
        {
            xSerial_PutString("\r\n");
            xSerial_PutChar(cReceiver, 10);
            xSerial_PutString("-Received : ");
            xSerial_PutChar(cReceiveData, 10);
        }
        else
        {
            xSerial_PutString("\r\n");
            xSerial_PutChar(cReceiver, 10);
            xSerial_PutString("-Could not receive from the queue.");
        }
    }
}

```

RX

Received Data: Receive 4176 characters

```

B-Received : S
A-Received : S

```

Sender Task A < Receiver Task A = Receiver Task B

```

RX
Received Data: Receive 1086 characters
A-Received : S
A-Received : S
A-Received : S
A-Received : S
B-Could not receive from the queue.
A-Received : S

```

Sender Task A < (Receiver Task A > Receiver Task B)

```

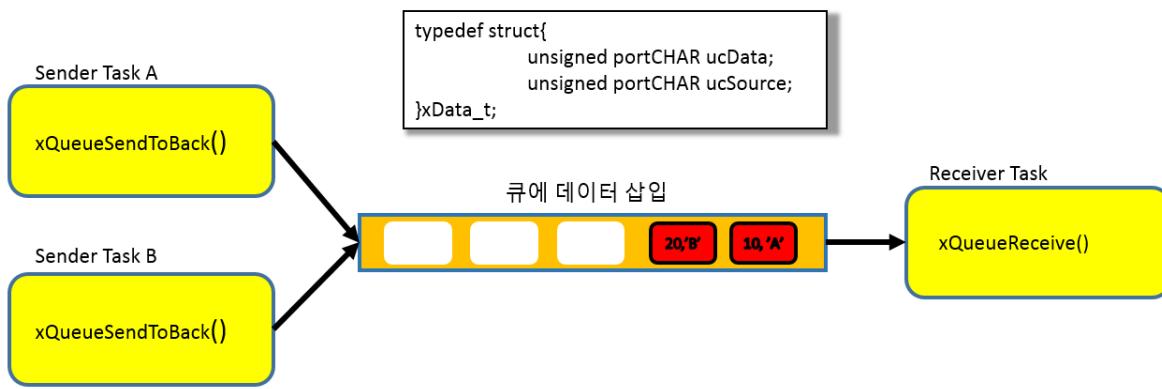
RX
Received Data: Receive 378 characters
B-Received : S
B-Received : S
B-Received : S
A-Could not receive from the queue.
B-Received : S

```

Sender Task A < (Receiver Task A < Receiver Task B)

예제 4) 복수 테스크가 큐에 송신하고 수신하기 (꼬리표)

여러 개의 소스(Source)에서 하나의 큐로 데이터를 보내는 경우가 흔히 발생한다. 이럴 경우에는 데이터가 어디에서 왔는지를 구별해서 데이터를 처리해야 한다. 이를 구현하기 위해 단순히 데이터만을 큐에 넣는 것이 아니라 소스와 데이터를 같이 묶어서 하나의 구조체로 만들어서 보냄으로서 데이터를 받는쪽에서 소스와 데이터를 알 수 있게 처리한다.



```

QueueHandle_t xQueue;

typedef struct{
    unsigned portCHAR ucData;
    unsigned portCHAR ucSource;
}xData_t;

xData_t xData_A = {'1', 'A'};
xData_t xData_B = {'2', 'B'};

void vQueue_CompoundType(void)
{
    xQueue = xQueueCreate(5,sizeof(xData_t));
    vQueueAddToRegistry(xQueue, "xQueue");

    // 송신 태스크 A/B생성
    xTaskCreate(vSenderTask,
                "Sender Task A",
                240,
                &xData_A,
                1,
                NULL
                );
    xTaskCreate(vSenderTask,
                "Sender Task B",
                240,
                &xData_B,
                1,
                NULL
                );

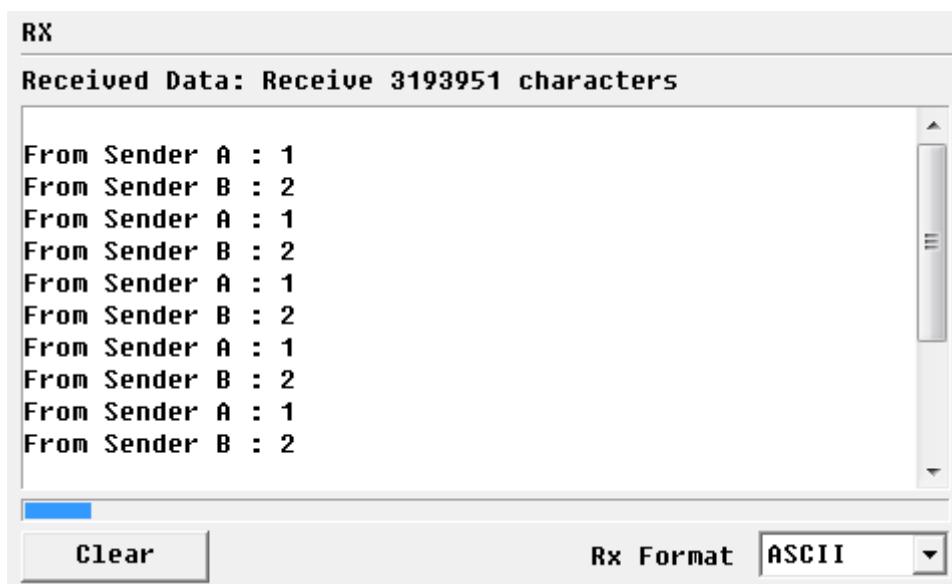
    // 수신 태스크 생성
    xTaskCreate(vReceiverTask,
                "Receiver Task",
                240,
                NULL,
                2,
                NULL
                );
}

```

```
static void vSenderTask(void *pvParam)
{
    for(;;)
    {
        if(xQueueSendToBack(xQueue, pvParam, 100) != pdPASS)
        {
            xSerial_PutString("Could not send to the queue.\r\n");
        }
        vTaskDelay(500 / portTICK_PERIOD_MS);
    }
}
```

```
static void vReceiverTask(void *pvParam)
{
    xData_t xReceiveData;

    for(;;)
    {
        if(xQueueReceive(xQueue, &xReceiveData, 10000) == pdPASS)
        {
            if(xReceiveData.ucSource == 'A')
            {
                xSerial_PutString("\r\nFrom Sender A : ");
                xSerial_PutChar(xReceiveData.ucData, portMAX_DELAY);
            }
            else if(xReceiveData.ucSource == 'B')
            {
                xSerial_PutString("\r\nFrom Sender B : ");
                xSerial_PutChar(xReceiveData.ucData, portMAX_DELAY);
            }
        }
        else
        {
            xSerial_PutString("Could not receive from the queue.\r\n");
        }
    }
}
```



복수 테스트가 큐에 송신하고 수신하기 (꼬리표)

5.2.4 많은 양의 데이터 다루기

큐에 넣어야 할 데이터의 양이 많을 경우에는 이 데이터 전부를 큐에 넣기 보다는 데이터의 주소를 넣는다. 이렇게 함으로써 큐에서 새로운 데이터를 위해 만들어야 하는 RAM의 크기도 줄일 수도 있고, 데이터를 처리하기 위한 시간도 줄일 수 있게 된다. 그러나 다음과 같은 점을 반드시 주의해야 된다.

1) 데이터에 대한 접근 보장

여러 태스크가 공유 메모리를 포인터를 통해서 사용하고 있다면 메모리의 내용에 대한 수정이 여러 태스크에 의해 동시에 일어나지 않으며 메모리의 내용이 유효하고 일관성이 있음을 보장해야 한다.

큐에 데이터를 송신하는 태스크가 있다면 이 태스크가 데이터의 주소값을 큐에 넣을 때까지 메모리의 접근이 허용되어야 하며, 또한 큐에서 데이터를 수신하는 태스크가 있다면, 이 태스크가 데이터의 주소값을 받아서 데이터에 접근할 수 있어야 한다.

2) 데이터의 유효성

만약에 동적 메모리에 있는 데이터에 대한 주소값을 큐에 넣는다고 하면, 큐에서 데이터를 수신하는 태스크가 메모리에 접근할 시기에 동적메모리가 사라지고 없으면 커다란 문제가 발생하게 된다. 동적 메모리를 사용할 경우에는 이 메모리를 해제(Freeing)할 경우에는 다른 태스크에서 이 메모리에 대한 접근이 없을 경우에만 해제해야만 한다.

그리고 태스크 스택에 할당된 메모리의 접근은 되도록이면 하지 않아야 한다.

5.3 인터럽트 관리(Interrupt Management)

5.3.1 개요

임베디드 시스템은 주변 환경에서 발생하는 이벤트에 대한 응답을 취해야 한다. 예를 들면 키가 눌리면 이에 대한 응답이 있어야 하고 시리얼로 데이터가 들어오면 이 데이터를 받아서 처리해야 한다. 여러 가지 소스에서 발생하는 이벤트들은 각각의 다른 처리 방법과 시간이 필요하게 된다. 이 같은 경우에 어떻게 각기 다른 이벤트에 가장 효과적으로 처리할 것인가에 대한 판단이 필요하다.

1) 어떻게 이벤트를 검출할 것인가?

- 인터럽트 또는 폴링(Polling)

2) 인터럽트 방식을 쓸 경우, ISR(Interrupt Service Routine)에서 어떤 일을 얼마큼 하고, 메인 루틴(non-ISR)에서 어떤 일 얼마큼 할 것인가?

- 가능하면 ISR에서 짧은 시간동안 일을 하도록 한다.
- 지연 인터럽트 처리(Deferred Interrupt Processing)

3) 이벤트가 발생했을 경우에 메인 루틴(non-ISR)과 어떻게 소통할 것인가? 또 어떻게 비동기적으로(이벤트가 발생하면 바로 전달) 이벤트를 처리할 것인가?

FreeRTOS에서는 직접적으로 위에 대한 방법을 제시하지는 않지만 이 것을 해결하기 위한 쉽고 간편한 도구들을 제시하고 있다.

5.3.2 상호 배제(Mutual Exclusion)

태스크가 가장 쉽게 서로 정보를 교환할 수 있는 방법은 공유 데이터를 통해서이다. 특히 모든 태스크가 단일 주소영역에 존재하고 전역 변수, 포인터, 버퍼, 링크드 리스트, 링버퍼등을 참조할 수 있을 때 더욱 용이하다. 데이터를 공유하는 것이 정보 교환을 쉽게 해주기는 하지만, 각 태스크의 경쟁과 공유 데이터의 내용이 손상되는 것을 막으려면 **데이터를 독점해서 사용**할 수 있게 해줘야 한다.

공유 자원의 독점적인 엑세스를 얻는 가장 일반적인 방법은

- 인터럽트 비활성화
- 전역 플래그 사용
- 스케줄링 비활성화
- 세마포어

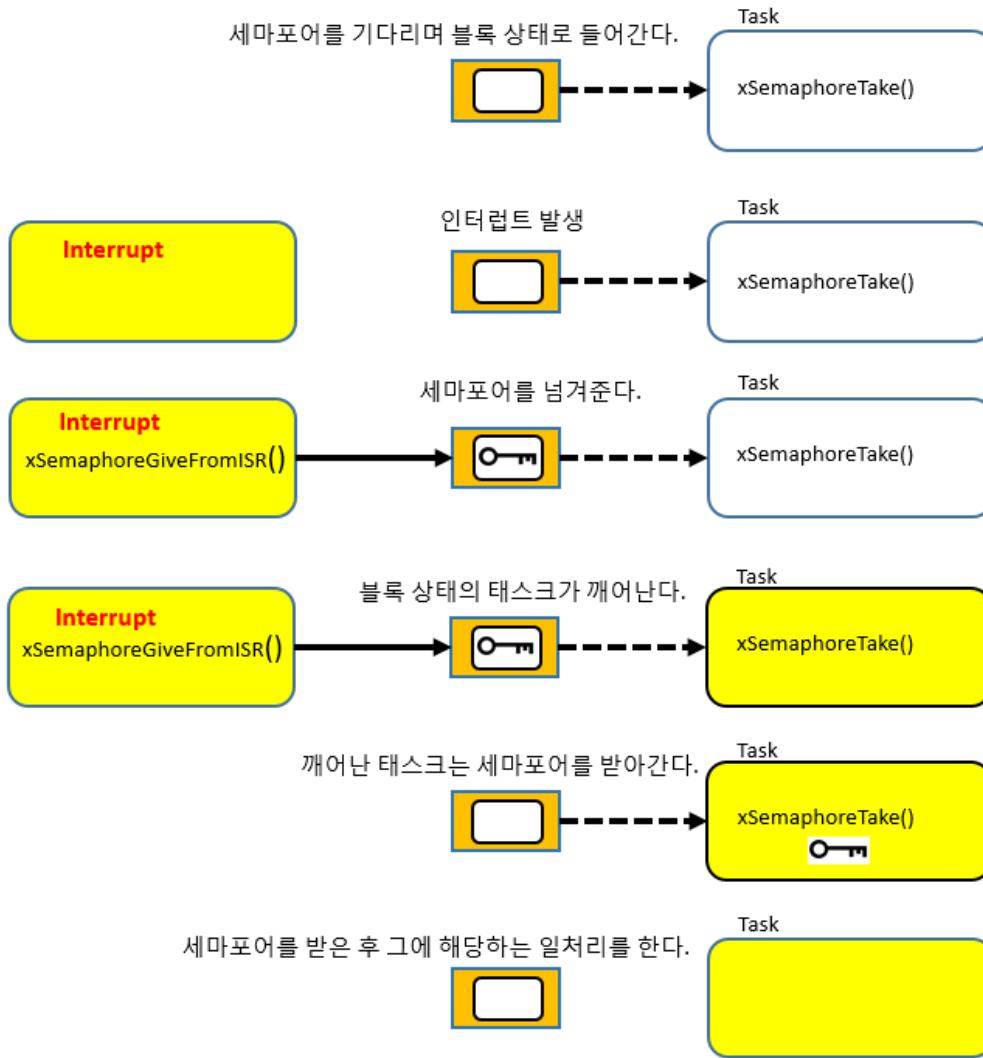
5.3.3 세마포어(Semaphore)

세마포어는 대부분의 멀티태스킹 커널이 제공하는 프로토콜 메커니즘이다. 세마포어는 다음과 같은 용도로 많이 사용되고 있다.

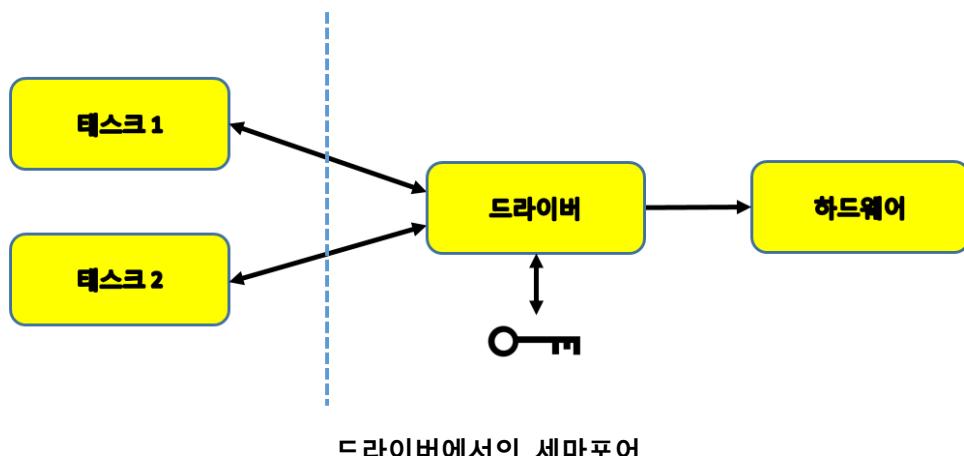
- 공유 자원간의 액세스를 제어(Mutual exclusion)
- 이벤트의 발생을 알려줌(Signaling)
- 두 태스크 간의 동작을 동기화(synchronization)

세마포어는 코드를 계속 실행하기 위해서 획득해야 하는 열쇠와 비슷하다. 세마포어가 이미 사용되고 있다면 세마포어를 요구한 태스크는 현재 세마포어 소유자가 세마포어를 양도할 때까지

대기 상태가 된다. 세마포어는 바이너리(Binary) 세마포어와 카운팅(Counting) 세마포어 두 가지 종류가 있다. 바이너리 세마포어는 단지 0과 1 두 가지의 값만 가질 수 있다. 카운팅 세마포어는 특정한 값을 가질 수 있고 세마포어를 생성할 때 값이 정해진다.



세마포어 처리 과정



5.3.4 세마포어 관련 API

- 세마포어 생성

- Binary 세마포어

```
void vSemaphoreCreateBinary(SemaphoreHandle_t xSemaphore );
```

xSemaphore 세마포어 핸들 (함수를 호출한후 xSemaphore의 값이 NULL이면 세마포어 생성에 실패)

- Counting 세마포어

```
xSemaphoreHandle xSemaphoreCreateCounting( unsigned portBASE_TYPE uxMaxCount,
                                         unsigned portBASE_TYPE uxInitialCount );
```

uxMaxCount 세마포어 최대 Count 개수

uxInitialCount 초기의 세마포어 Count 개수

- 세마포어 주기(xSemaphoreGive() / xSemaphoreGiveFromISR())

```
portBASE_TYPE xSemaphoreGive(SemaphoreHandle_t xSemaphore );
```

```
portBASE_TYPE xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore,
                                     portBASE_TYPE *pxHigherPriorityTaskWoken);
```

xSemaphore 세마포어 핸들

pxHigherPriorityTaskWoken 현재 실행할 태스크보다 우선순위가 높거나 같은 태스크가 Ready 상태에 있을 경우에 pdTRUE, 아니면 pdFAIL를 돌려준다.

이 값이 pdTRUE일 경우에는 ISR를 빠져나가기 전에 문맥전환이 이루어 져야한다.

- 세마포어 받기 (xSemaphoreTake())

```
portBASE_TYPE xSemaphoreTake(SemaphoreHandle_t xSemaphore, portTickType
                           xBlockTime);
```

xSemaphore 세마포어 핸들

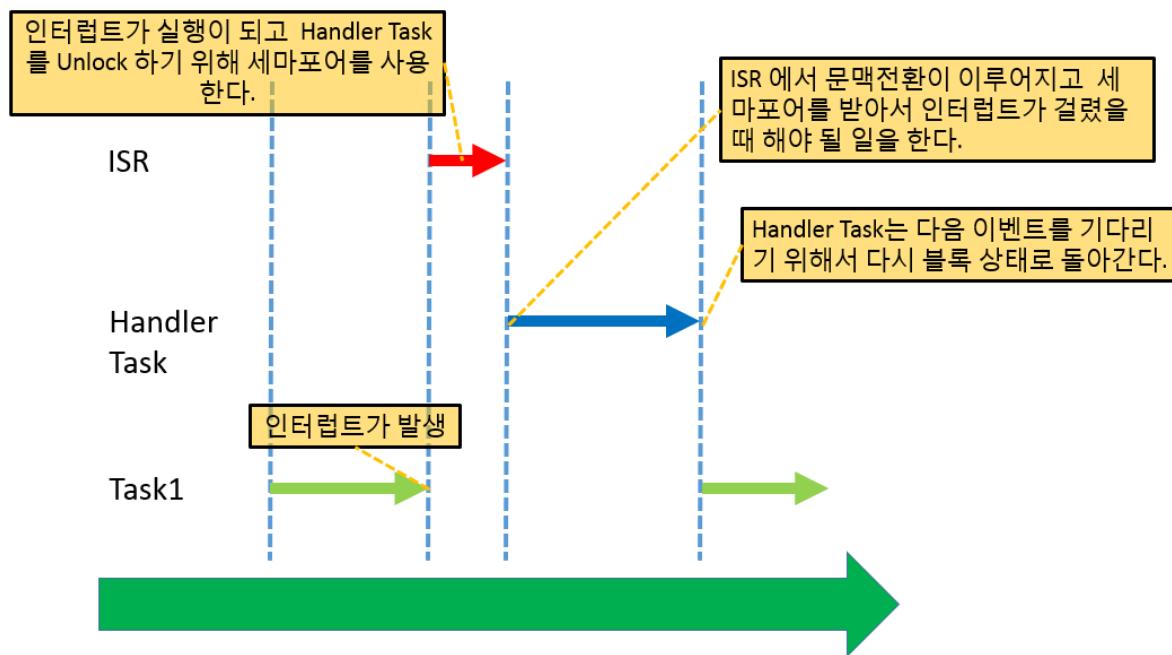
xBlockTime 세마포어를 받을 때까지 태스크가 Blocked 상태로 기다리는 시간(틱).

반환값 pdPASS : 성공
pdFAIL : 실패

5.3.5 Deferred Interrupt Processing(지연 인터럽트 처리)

5.3.5.1 바이너리 세마포어(Binary Semaphore)

바이너리 세마포어는 불특정한 시간에 발생하는 인터럽트가 발생했을 경우에 태스크를 대기상태(Blocked)에서 깨어나게 해서 인터럽트와 태스크를 동기화시키는 목적으로 사용될 수 있다. 인터럽트 이벤트 처리에 있어서 당장 빨리 처리해야 하는 일을 ISR에서 처리하도록 하고 나머지 일들은 동기화된 태스크에서 실행되도록 만드는 것이다. 이러한 인터럽트 처리를 지연 인터럽트 처리(Deferred Interrupt Processing)이라고 한다.



5.3.5.2 Counting Semaphore

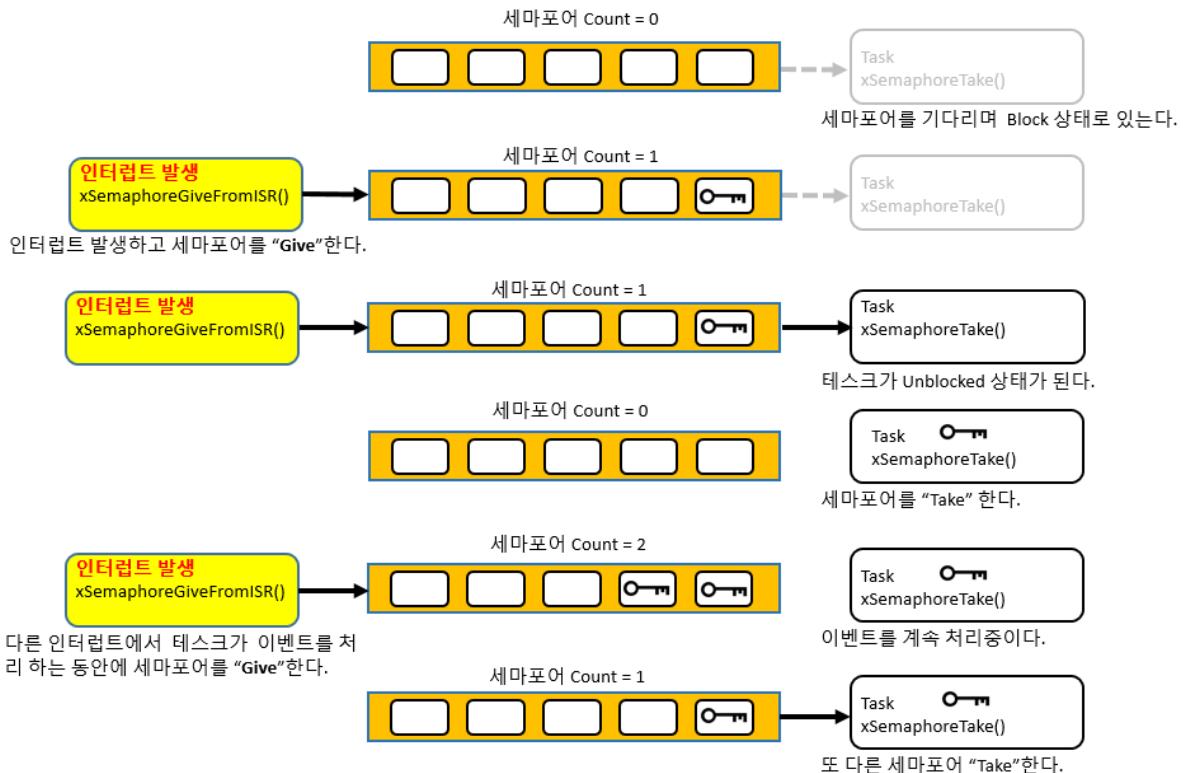


Figure Counting Semaphore를 이용 한 Events Count의 처리 과정

Counting Semaphore 는 Binary Semaphore와는 달리 세마포어의 개수가 1개가 아닌 여러 개가 존재한다. 그 용도는 다음 2가지로 요약할 수 있다.

1) 이벤트의 개수를 센다.

이벤트(인터럽트) 핸들러는 이벤트가 발생할 때마다 세마포어를 “Give”한다. 이 때마다 Counting Semaphore 의 Count 개수는 증가한다. 핸들러 테스크가 “Take”할 때마다 Count의 개수는 감소하게 된다. Count의 개수는 이벤트가 발생한 개수와 이 이벤트를 처리하 개수의 차를 표시하게 된다.

2) 자원 관리

세마포어의 Count는 사용 가능한 자원의 개수를 나타내는 용도로 사용될 수 있다. 자원을 제어하기 위한 권한을 획득하기 위해서 테스크는 세마포어를 먼저 얻어야(“Take”) 한다. 이 때 Count 값은 하나 감소하게 된다. 만약 Count 값이 0이 된다면 사용 가능한 자원이 없게 된다. 자원을 다 사용한 테스크가 다시 세마포어를 돌려 주면(“Give”) Count값이 증가하고 자원을 다시 테스크가 사용할 수 있는 권한을 획득할 수 있게 된다.

5.3.6 인터럽트 서비스 루틴(ISR) 안에서의 큐사용

xQueueSendToFrontFromISR(), xQueueSendToBackFromISR() 그리고 xQueueReceiveFromISR()은 기존의 큐 함수를 인터럽트 서비스 루틴에서 안정적으로 사용할 수 있도록 만든 함수이다.

```
portBASE_TYPE xQueueSendToFrontFromISR( xQueueHandle xQueue,
                                         void *pvItemToQueue,
                                         portBASE_TYPE *pxHigherPriorityTaskWoken );
```

xQueue	큐 핸들
pvItemToQueue	큐에 넣은 항목의 포인터
pxHigherPriorityTaskWoken	현재 실행할 태스크보다 우선순위가 높거나 같은 태스크가 Ready 상태에 있을 경우에 pdTRUE, 아니면 pdFAIL를 돌려준다. 이 값이 pdTRUE일 경우에는 ISR를 빠져나가기 전에 문맥전환이 이루어 져야한다.

```
portBASE_TYPE xQueueSendToBackFromISR( xQueueHandle xQueue,
                                         void *pvItemToQueue,
                                         portBASE_TYPE
```

xQueue	큐 핸들
pvItemToQueue	초기의 세마포어 Count 개수
pxHigherPriorityTaskWoken	현재 실행할 태스크보다 우선순위가 높거나 같은 태스크가 Ready 상태에 있을 경우에 pdTRUE, 아니면 pdFAIL를 돌려준다. 이 값이 pdTRUE일 경우에는 ISR를 빠져나가기 전에 문맥전환이 이루어 져야한다.

- **효과적인 큐 사용법**

인터럽트 핸들러가 데이터를 큐를 통해 전달하고 받는 방법은 상당히 비효율적인 방법이다. 특히 데이터가 수시로 오고 가야하는 통신에서는 속도가 올라감에 따라 두드러지게 나타나게 된다. 그래서 다음과 같은 두가지 방법을 추천하다.

- 1) 간단한 RAM 버퍼 사용

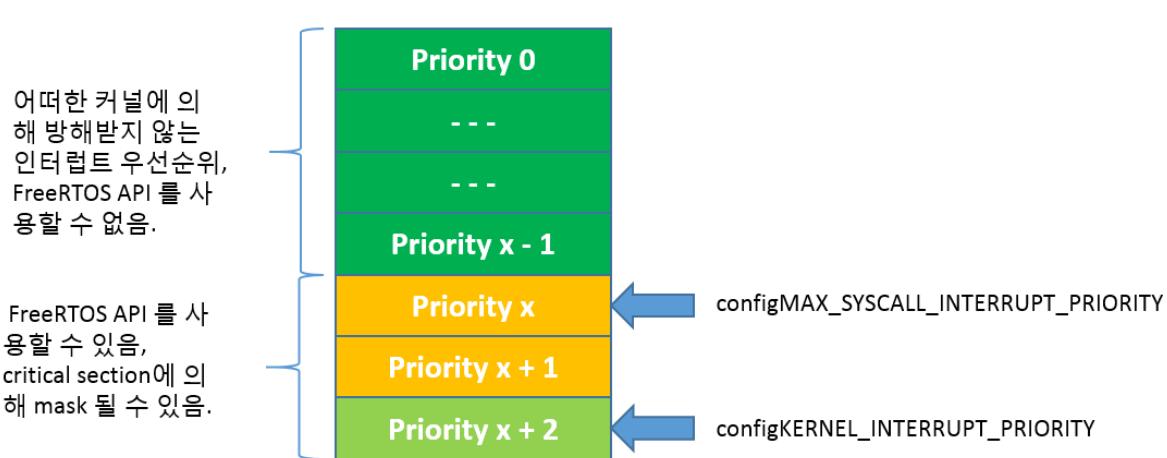
간단한 RAM를 사용하여 데이터가 모두 들어올 때까지 RAM에 데이터를 저장하고 그 동안 세마포어를 이용하여 데이터를 처리하는 프로세스를 블록시킨다. 데이터가 다 들어오면 그 때 데이터 처리 함수를 언블록시켜 데이터를 처리한다.

- 2) 인터럽트 서비스 루틴에서 데이터 직접 처리

데이터가 들어오면 인터럽트 서비스 루틴에서 데이터를 처리하고 여기서 나온 결과를 큐를 통해서 데이터 처리 프로세스에게 넘겨 처리하도록 한다.

5.3.7 인터럽트 증첩

상수	설명
configKERNEL_INTERRUPT_PRIORITY	커널 자체내에서 사용하는 인터럽트, 우선순위로, Time tick 으로 사용하는 타이머의 인터럽트와 API안에서 사용되는 PendSV (Pend Service Call)이 이에 해당한다. configKERNEL_INTERRUPT_PRIORITY은 가능한 가장 낮은 우선순위를 갖도록 설정된다.
configMAX_SYSCALL_INTERRUPT_PRIORITY	FreeRTOS에서 호출 되어질 수 있는 가장 높은 인터럽트 우선순위, 'FromISR' 로 끝나는 API 함수만이 인터럽트 서비스 루틴안에서 호출할 수 있다. 이 상수는 반드시 0이 아니어야 한다. 그렇게 되면 이를 mask할 수 없게 되기 때문이다.



- 인터럽트 우선순위 ($x \sim x + 2$)
이 범위의 우선순위는 critical section에 의해 실행이 중지될 수 있다.
'FromISR'로 끝나는 API함수를 사용할 수 있다.
- 인터럽트 우선순위 ($0 \sim x - 1$)
critical section에 의해 실행이 영향받지 않는다. 따라서 커널에 의해 실행이 영향받지 않는다.
타이밍에 민감한 기능(예를 들어, 모터제어) 구현에 사용한다.

5.4 소프트웨어 타이머

5.4.1 개요

Software Timer는 설정된 시간에 Timer의 Callback Function을 실행하도록 한다. Timer가 Start 되고 설정된 시간 후(Period)에 Timer Callback Function이 실행 된다.

Timer Callback Function은 Timer Service Task로 실행 되기 때문에 Blocking 상태가 될 수 없다. 그렇기 때문에 Timer Callback Function에서는 vTaskDelay(), vTaskDelayUntil() API를 Call 할 수 없고, Queue나 Semaphore를 Accessing 할 때 Block Time을 zero로 하여야 한다.

5.4.2 Timer Command Queue

FreeRTOS는 여러 종류의 Timer 관련 API 함수를 갖고 있다. 이들 함수는 Timer Command Queue를 사용하여 Timer Service Task에 명령을 전달 한다.

Timer Command Queue는 Timer가 구현될 때 생성되고, Private 속성을 갖기 때문에 사용자가 직접 Access 할 수 없다.

5.4.3 소프트웨어 타이머를 사용하기 위한 설정

FreeRTOSConfig.h Header File에 다음 상수를 Define 한다.

Constant	Description
configUSE_TIMERS	1로 Set하면 RTOS Scheduler가 Start할 때 Timer Service Task를 자동으로 Create 한다.
configTIMER_TASK_PRIORITY	Timer Service Task의 Priority를 설정 한다. 다른 Task와 마찬 가지로 0 – (configMAX_PRIORITIES-1) 사이의 값으로 설정 할 수 있다.
configTIMER_QUEUE_LENGTH	Timer Command Queue에 미처 처리되지 못한 Command를 Hold 할 수 있는 수
configTIMER_TASK_STACK_DEPTH	Timer Service Task에서 사용 할 수 있는 Stack Size(Word)

5.4.4 One-shot Timer 와 Auto-reload Timer

One-shot Timer	Auto-reload Timer
<p>One-shot Timer는 한번 Start되면 Callback function을 한번만 실행 한다.</p> <p>One-shot Timer를 다시 Start시키려면 xTimerStart() 등의 API 명령을 사용하여야 한다.</p>	<p>Auto-reload Timer는 Callback function이 실행된 다음에 자동으로 Restart되어 일정한 주기로 반복하여 실행된다.</p>

5.4.5 소프트웨어 타이머 관련 API

● 타이머 생성

- 타이머를 생성한다.

<code>TimerHandle_t xTimerCreate</code>	
	(const char * const pcTimerName, const TickType_t xTimerPeriod, const unsigned BaseType_t uxAutoReload, void * const pvTimerID, TimerCallbackFunction_t pxCallbackFunction);
<code>pcTimerName</code>	타이머에 대한 문자열 이름. 디버깅 용도로만 사용됨.
<code>xTimerPeriod</code>	타이머 주기. 단위는 Time tick이된다. 예를 들어 100ms 후에 콜백함수를 호출하려면 100/portTICK_PERIOD_MS 라고 설정하면 된다.
<code>uxAutoReload</code>	- Auto Reload : pdTRUE - One-Shot : pdFALSE
<code>pvTimerID</code>	타이머 식별자. 하나의 콜백함수에 여러 개의 타이머가 할당되었을 경우에 타이머를 식별하기 위해 사용될 수 있다.
<code>pxCallbackFunction</code>	타이머의 시간이 되었을 경우에 백을 받을 콜백함수. 콜백함수의 프로토 타입(Prototype)은 <code>void vCallbackFunction(TimerHandle_t xTimer);</code> 이다.
리턴값	타이머 핸들

● 타이머 시작

- 타이머를 시작시킨다.

<code>BaseType_t xTimerStart(TimerHandle_t xTimer,</code>	
	<code>TickType_t xBlockTime);</code>
<code>xTimer</code>	타이머 핸들.
<code>xBlockTime</code>	명령이 타이머 명령 큐에 전달 될 때까지 블록된(Blocked) 상태로 있을 틱 단위의 시간.

리턴값

- pdFAIL : 명령이 타이머 명령 큐에 전달되지 않음.
- pdPASS : 명령이 타이머 명령 큐에 전달됨.

● 타이머 정지

- 타이머를 정지시킨다.

<code>BaseType_t xTimerStop(TimerHandle_t xTimer,</code>	
	<code>TickType_t xBlockTime);</code>
<code>xTimer</code>	타이머 핸들.
<code>xBlockTime</code>	명령이 타이머 명령 큐에 전달 될 때까지 블록된(Blocked) 상태로 있을 틱 단위의 시간.

리턴값

- pdFAIL : 명령이 타이머 명령 큐에 전달되지 않음.
- pdPASS : 명령이 타이머 명령 큐에 전달됨.

● 타이머 초기화

- 타이머를 초기화시킨다.

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

xTimer	타이머 핸들.
xBlockTime	명령이 타이머 명령큐에 전달 될 때까지 블럭된(Blocked) 상태로 있을 틱 단위의 시간.
리턴값	<ul style="list-style-type: none"> - pdFAIL : 명령이 타이머 명령큐에 전달되지 않음. - pdPASS : 명령이 타이머 명령큐에 전달됨.

● 타이머 주기 설정

- 타이머의 주기를 설정한다.

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

xTimer	타이머 핸들.
xNewPeriod	새로운 주기값
xBlockTime	명령이 타이머 명령큐에 전달 될 때까지 블럭된(Blocked) 상태로 있을 틱 단위의 시간.
리턴값	<ul style="list-style-type: none"> - pdFAIL : 명령이 타이머 명령큐에 전달되지 않음. - pdPASS : 명령이 타이머 명령큐에 전달됨.

● 타이머 제거

- 타이머를 제거한다.

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

xTimer	타이머 핸들.
xBlockTime	명령이 타이머 명령큐에 전달 될 때까지 블럭된(Blocked) 상태로 있을 틱 단위의 시간.
리턴값	<ul style="list-style-type: none"> - pdFAIL : 명령이 타이머 명령큐에 전달되지 않음. - pdPASS : 명령이 타이머 명령큐에 전달됨.

5.5 자원 관리

5.5.1 멀티 테스킹에서의 자원관리의 고려사항

- 동시에 여러 테스크가 한 장치에 접근
시리얼 포트로 두 개의 테스크가 접근하는 경우
 - 1) 테스크A가 시리얼 포트에 접근하여 “Hello world”를 보내기 시작한다.
 - 2) 테스크A가 “Hello w”를 보내다가 테스크B에 의해 선점이 된다.
 - 3) 테스크B는 “Abort, Retry, Fail?”를 보내고, 블록 상태로 들어간다.
 - 4) 테스크A가 다시 실행되고 “orld”를 시리얼로 보낸다.

결국, 시리얼 포트는 “Hello wAbort, Retry, Fail?orld”를 보내게 된다.

- RMW(Read, Modify, Write)

C 코드	어셈블리어
<pre>GlobalVar = 0x01;</pre>	<pre>LDR r4, [pc,#284] LDR r0, [r4,#0x08] /* Load the value of GlobalVar into r0. */ ORR r0, r0,#0x01 /* Set bit 0 of r0. */ STR r0, [r4,#0x08] /* Write the new r0 value back to GlobalVar. */</pre>

C언어에서 한 줄의 코드로 변수에 대한 처리를 하지만, 어셈블리어로는 데이터를 읽고(Read), 이것을 수정하고(Modify), 쓰기(Write)를하게 된다. 이 경우에

- 1) 테스크A가 GlobalVar의 값을 레지스터로 값을 읽어 들인다.
- 2) 테스크A가 테스크B에 의해 선점이 된다.
- 3) 테스크B에서 GlobalVar를 수정하고 블록(Block)상태로 들어간다.
- 4) 테스크A가 다시 실행되고 1)에서 읽어온 값이 있는 레지스터를 수정한 다음 GlobalVar에 값을 쓰게 된다.

이 경우에 테스크의 A/B에 의해 GlobalVar의 일관성이 깨지게 된다.

- 비원자적 변수에 대한 접근(Non-atomic Access to Variables)

구조체나 워드(word)보다 큰 변수에 대한 변수에 대한 값을 수정할 경우에 한 번에 데이터를 가지고 올 수 없고 여러 번에 걸쳐 데이터를 가지고 와야 하기 때문에 그 중간에 다른 테스크에 의해 선점되거나 인터럽트에 의해 중지 될 경우에는 데이터를 잃거나 깨지게 된다.

- 함수의 재진입성

하나 이상의 테스크 또는 인터럽트에 의해 함수가 호출될 때 함수의 동작에 있어서 문제가 없다면(안전하다면) 이 함수를 재진입 가능하다고 한다.

재진입 함수(Reentrant function)

```
long Increase(long Data)
{
    longRetVal;
    RetVal = Data + 1;
```

```

        return RetVal;
    }
}

```

비재진입 함수(Reentrant function)

```

long Temp;
void swap(long *x, long *y)
{
    Temp = *x;
    *x = *y;
    *y = *Temp;
}

```

5.5.2 상호 배제 (Mutual Exclusion)

여러 테스크와 인터럽트의 사이에서 공통으로 사용하는 자원에 대한 접근은 상호배제라는 방법을 사용해야만 한다(일종의 공유 자원을 독점하는 방식). 공유 자원에 대해 여러 테스크가 동시에 접근해서 사용하게 대한 자원에 대한 일관성을 유지할 수가 없게 된다. 그래서 RTOS에서는 공유 자원에 대해 한번에 하나의 테스크나 인터럽트가 접근할 수 있도록 아래와 같은 여러 방법을 제공하고 있다.

- 크리티컬 섹션
- 스케줄링의 정지
- 세마포어

5.5.3 크리티컬 섹션(Critical Section)

크리티컬 섹션은 공유 자원을 독점하여 사용하기 위해 인터럽트를 중지시키는 방법으로 공유 자원을 사용할 때 인터럽트를 정지 시켰다가 공유 자원을 다 사용하면 인터럽트를 다시 동작시키는 것입니다.

크리티컬 섹션 (Reentrant function)

```

/* 크리티컬 섹션으로 진입 */
taskENTER_CRITICAL();
/* taskENTER_CRITICAL()이 호출 된 이후에는 문맥 전환이 일어나지 않는다. 그러나
인터럽트는 configMAX_SYSCALL_INTERRUPT_PRIORITY 보다 높은 인터럽트만 가능하다.*/
GlobalVar |= 0x01;
/* 크리티컬 섹션으로 빠져나감 */
taskEXIT_CRITICAL();

```

크리티컬 섹션은 완벽한 상호배제 방법은 아니다. 이 방법은 configMAX_SYSCALL_INTERRUPT_PRIORITY 값까지의 인터럽트만을 비활성화시킨다. taskENTER_CRITICAL()를 호출한 테스크는 크리티컬 섹션을 빠져 나올 때까지 Running 상태를 보장 받고 taskEXIT_CRITICAL()를 호출하면 이 섹션을 빠져 나온다. 크리티컬 섹션은 가능한 짧은 시간동안 사용해야만 한다. 그렇지 않으면 인터럽트 응답 시간에 영향을 미치게 된다.

FreeRTOS에서의 크리티컬 섹션은 nesting의 개수를 세고 있기 때문에 크리티컬 섹션이

nesting 이 된다고 해도 안전하다.

5.5.4 스케줄링의 정지

크리티컬 섹션은 또한 스케줄러를 정지시켜도 가능하다. 기본적으로 크리티컬 섹션은 다른 테스크나 인터럽트로부터 코드를 보호하는 것이다. 이 방법은 크리티컬 섹션안에서의 코드는 단지 다른 테스크의 접근만을 막을 수 있고 인터럽트는 막을 수가 없다.

FreeRTOS에서는 다음과 같은 두 가지 함수를 제공한다.

스케줄링 정지

```
void vTaskSuspendAll( void );
```

스케줄링 다시 활성화

```
portBASE_TYPE xTaskResumeAll( void );
```

5.5 뮤텍스 (Mutexes : Binary Semaphores)

뮤텍스는 여러 테스크 사이의 공유 자원 접근 제어에 사용되는 바이너리 세마포어의 특별한 형태이다. MUTEX 는 “MUTual EXclusion”의 약어이다.

뮤텍스를 사용하게 되면 테스크가 공유 자원을 사용하려고 하면 먼저 토큰(Token)을 먼저 획득(take)해야 한다. 획득하고 나면 자원을 사용하고 끝나면 반드시 토큰을 다시 반납(give)해야 한다.

- 세마포어와 뮤텍스의 차이점
 - 1) 뮤텍스로 사용되는 세마포어는 항상 토큰이 리턴되어야 한다.
 - 2) 동기의 목적으로 사용되는 세마포어는 획득한 세마포어를 버리고 리턴하지 않는다.

뮤텍스를 위한 함수는 다음과 같다.

뮤텍스의 생성

```
xSemaphoreHandle xSemaphoreCreateMutex( void );
```

뮤텍스의 획득

```
portBASE_TYPE xSemaphoreTake( xSemaphoreHandle xSemaphore, portTickType xBlockTime );
```

뮤텍스의 반납

```
portBASE_TYPE xSemaphoreGive( xSemaphoreHandle xSemaphore );
```

자원을 보호하기 위한 뮤텍스



Task A

Task B

자원이 뮤텍스에 의해 보호됨

보호된
자원

두 개의 테스크가 자원에 접근하기를 원하지만 뮤텍스를 획득하지 못했기 때문에 접근할 수가 없다.

자원을 보호하기 위한 뮤텍스



Task A

`xSemaphoreTake()`

Task B

보호된
자원

Task A가 뮤텍스를 획득을 시도해서 뮤텍스를 획득한 후 자원에 접근할 수 있게 된다.

자원을 보호하기 위한 뮤텍스



Task A



Task B

`xSemaphoreTake()`

보호된
자원

Task B가 실행이 되고 같은 뮤텍스 획득을 시도하지마 Task A가 여전히 뮤텍스를 가지고 있기 때문에 Task B는 자원에 접근할 수가 없다.

자원을 보호하기 위한 뮤텍스



Task A

`xSemaphoreGive()`

Task B

보호된
자원

Task B는 뮤텍스를 얻기 위해 블록 상태로 들어가고 Task A는 다시 실행 상태가 된다.
Task A는 자원의 사용을 마치고 뮤텍스를 반환한다.

자원을 보호하기 위한 뮤텍스



Task A

Task B

`xSemaphoreTake()`

보호된
자원

Task A가 뮤텍스 반환을 하게 되면 Task B가 블록 상태에서 깨어나게 되고 뮤텍스를 성공적으로 획득한다. Task B는 이제 자원에 접근을 할 수 있게 된다.

자원을 보호하기 위한 뮤텍스



Task A

Task B

`xSemaphoreGive()`

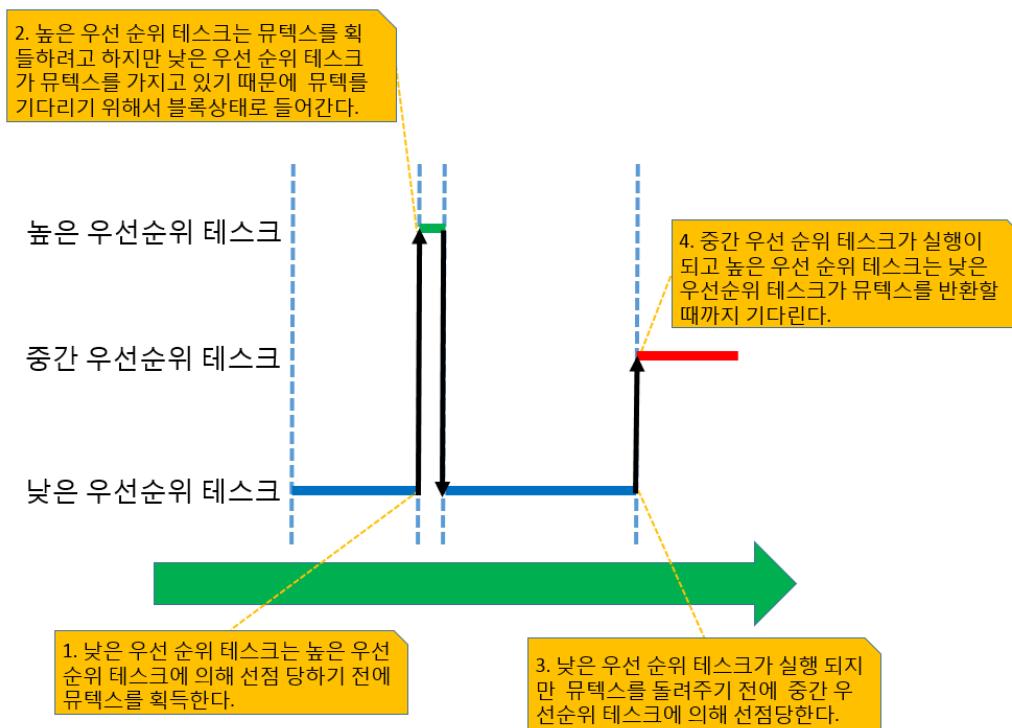
보호된
자원

Task B는 자원의 사용을 마치고 다시 뮤텍스를 반납한다. 이제 다른 테스크에서 뮤텍스를 획득할 수 있다.

5.5.5.1 우선순위 역전(Priority Inversion)

상호 배제를 위해 뮤텍스를 사용하다보면 잠재적인 위험에 빠질 경우가 있다.

아래 그림과 같이 낮은 우선순위 테스크가 높은 우선순위 테스크에 의해 선점 당하기 전에 뮤텍스를 획득하고, 이 후에 높은 우선순위 테스크에 의해 선점을 당한다. 높은 우선순위 테스크는 자원을 사용하기 위해 뮤텍스를 획득하려고 하면 낮은 우선순위 테스크가 이미 뮤텍스를 획득했기 때문에 높은 우선순위 테스크는 뮤텍스를 기다리기 위해 블록상태로 들어가게 된다. 그러면 다시 낮은 우선순위가 실행되게 된다. 이렇게 되면 높은 우선순위 테스크의 우선순위가 낮은 우선순위 테스크보다 높지만 실행되지 못하고 낮은 우선순위 테스크가 실행이 되게 된다. 이것을 우선순위 역전(Priority Inversion)이라고 한다.

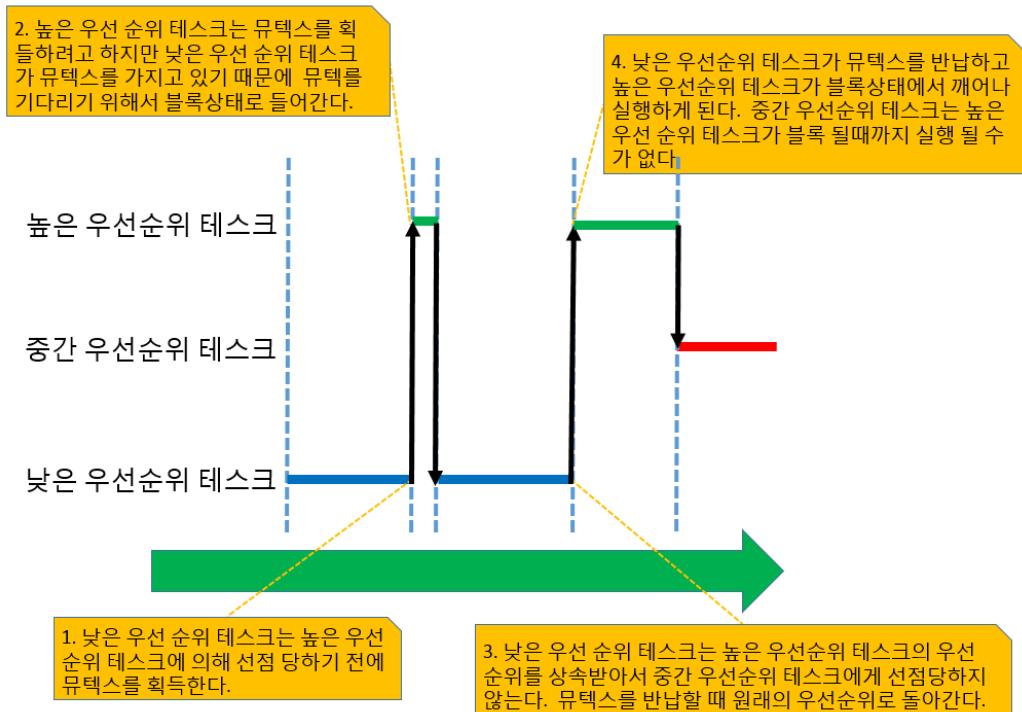


5.5.5.2 우선순위 상속(Priority Inheritance)

FreeRTOS에서 뮤텍스와 바이너리 세마포어는 서로 비슷하지만 다른점은 뮤텍스에서는 우선순위 상속이 이루어지지만, 바이너리 세마포어에서는 그렇지 않다는 점이다.

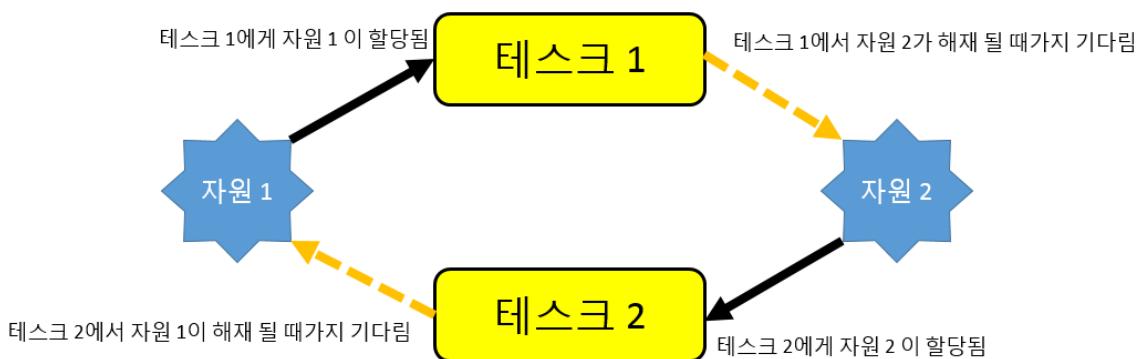
우선순위 상속은 우선순위 역전의 역효과를 최소화한다. 그러나 완전하게 해결책은 되는 것은 아니며, 또한 시스템 타이밍 예측을 어렵게 만든다.

우선순위 상속은 뮤텍스를 획득한 테스크가 같은 뮤텍스를 얻으려고 하는 가장 높은 테스크의 우선순위를 상속받아서 테스크의 우선순위를 임시적으로 높인다. 그렇게 함으로써 이 보다 낮은 테스크에게 선점당하지 않는다. 이렇게 함으로써 우선순위가 역전되는 시간을 줄일 수 있다. 그리고 공유자원을 다 사용하고 나면 다시 원래의 우선순위로 돌아간다.



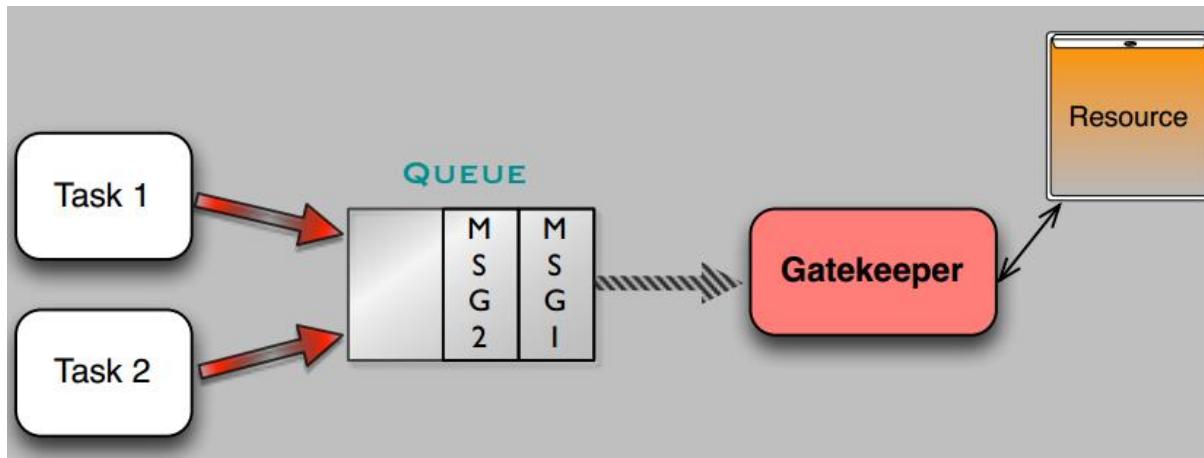
5.5.3 교착 상태(Deadlock, Deadly Embrace)

교착상태는 두 테스크가 각자 다른 테스크가 사용하고 있는 자원을 무한정 기다리는 상태를 말한다. 테스크1이 자원1을 사용하고 있고, 테스크2는 자원2를 사용하고 있을 때, 테스크1이 자원2가 필요하고, 테스크2는 자원1이 필요하게 되면 어느 테스크도 계속 진행 할 수 없다. 즉 교착 상태에 빠지는 것이다. 대부분의 커널은 세마포어를 획득할 때 타임아웃을 설정할 수 있다. 이 것을 이용하여 어떤 형태의 에러코드를 테스크에 전달할 수 있다.



5.5.4 문지기 테스크(Gatekeeper Tasks)

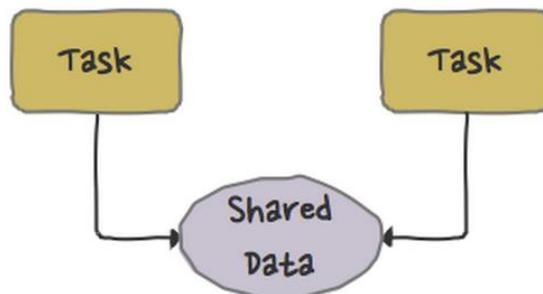
문지기 테스크는 우선순위 역전 또는 교착상태없이 상호배제를 할 수 있는 확실한 방법이다. 문지기 테스크는 자신만의 공유자원을 가지고 있는 테스크이다. 오직 문지기 테스크만이 직접적으로 자원에 접근 가능하다. 자원에 접근하고자 하는 다른 테스크들은 간접적으로 문지기 테스크의 서비스를 사용해야만 한다.



5.5.5.5 뮤텍스와 세마포어의 차이점

- 공유 데이터를 어떻게 직렬로 공유할 것인가? (뮤텍스의 관점)

여러 개의 태스크가 실행되고 있는 상황에서 공유된 데이터는 동시에 여러 태스크가 접근해서 데이터를 사용하게 되면 데이터의 일관성을 유지할 수 없다. 항상 한번에 하나의 태스크가 접근해서 데이터를 사용하도록 해야만 데이터의 일관성을 유지할 수 있다. 이것을 데이터 접근의 직렬화(Serialized)라고 한다.



- 데이터가 사용할 수 있다고 어떻게 알려 줄 것인가? (세마포어의 관점)

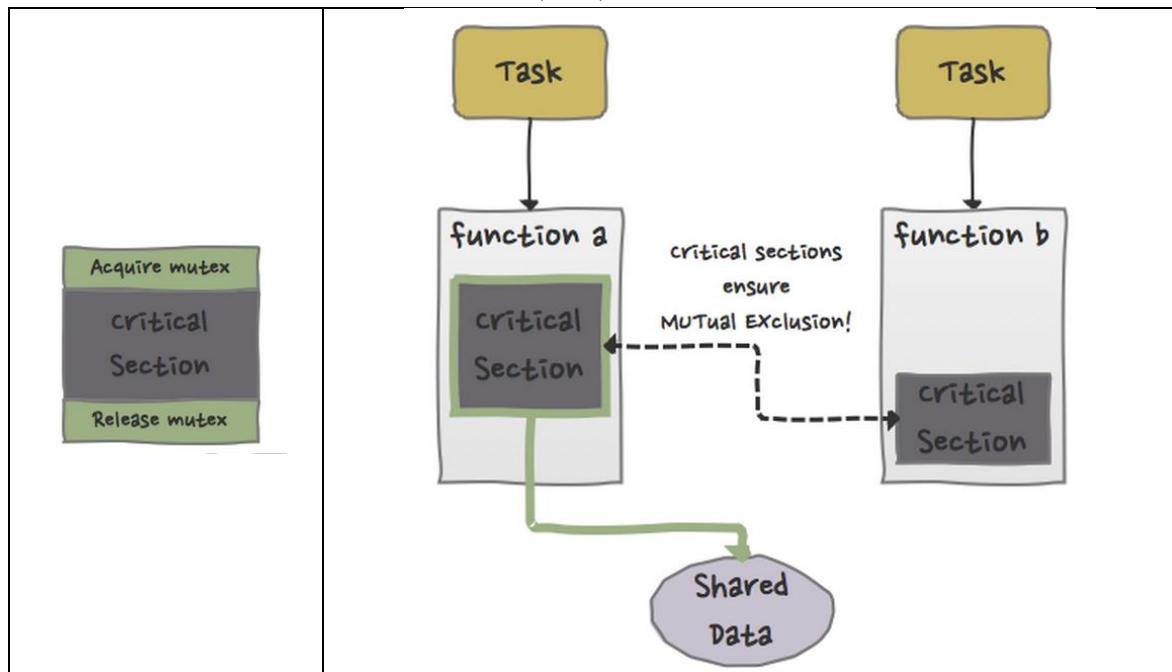
데이터를 생산하는 쪽(생산자 : Producer)과 데이터를 처리하는 쪽(소비자 : Consumer)이 있다고 하면 생산자는 새로운 데이터가 생산이 되면 이 데이터를 바로 처리할 수 있도록 소비자에게 바로 알려 주어야 한다.



- 뮤텍스

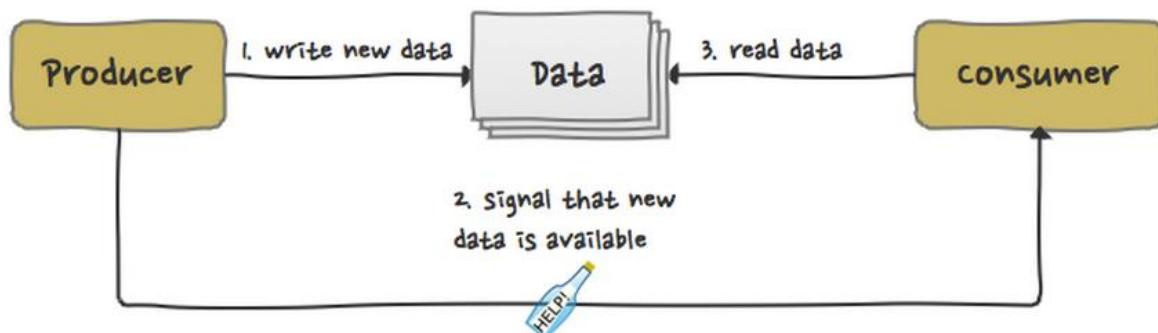
뮤텍스는 크리티컬 섹션(Critical Section)을 이용해서 데이터를 접근하도록 만든 장치이다. 함수가 크리티컬 섹션에서 동작하게 되면 다른 태스크에서 접근할 수 없기 때문에 자연스럽게 상호배제(Mutual Exclusion)이 실행되게 된다.

뮤텍스에서 크리티컬 섹션 들어가기 위해 토큰(Token)을 얻어야 하고(Take)를 해야 하고 크리티컬 섹션을 나올 때 반드시 토큰을 반납(Give) 해야만 한다.



- 세마포어

세마포어의 경우는 공유 데이터에 대해서 새로운 데이터가 생성이 되면(생산자에서) 데이터를 처리해야 하는 쪽에게(소비자에게) 데이터가 있다고 신호를 보내는 방식이다.



5.6 참조

5.6.1 코딩 스타일

- 변수

변수형	접두어(Prefix)
char	c
short	s
long	l
Enum	e
기타 다른 형(struct...)	x
Pointer	p
unsigned	u

예제)

```
unsigned portCHAR ucData;
unsigned portSHORT usData;
unsigned portLONG ulData;
QueueHandle_t xQueue
```

- 함수

함수	접두어(Prefix)
File private 함수	prv
API 함수	함수리턴 데이터 형
함수 이름	함수가 정의된 파일 이름

예제)

```
void prvExtIF_GPIOConfig(void);
void vSenderTask(void *pvParam);
void vReceiverTask(void *pvParam);
portBASE_TYPE xSerial_PutString(const portCHAR* pcString);
```

5.6.2 매크로

매크로는 기본적으로 대문자를 사용하고 접두어는 소문자를 사용한다.

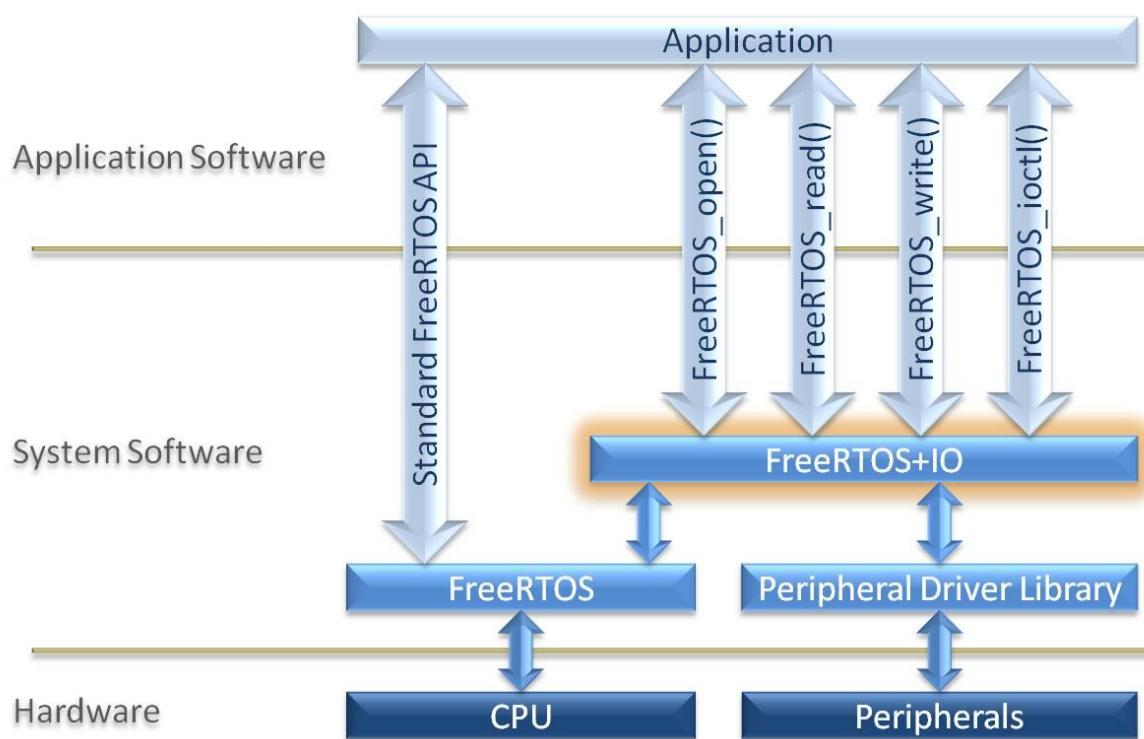
Prefix	Location of macro definition
port (for example, portMAX_DELAY)	portable.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

- 자주 사용되는 매크로 정의

매크로	값
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

6. 드라이버 작성

6.1 개요



6.2 전송 모드(Transfer Mode)

드라이버에서 데이터를 주고 받는 방법에는 다음과 같이 나누어 볼 수 있다.

전송 모드	데이터 방향	설명
Polled	읽기 / 쓰기	인터럽트를 사용하지 않고 폴링(Polling) 방식으로 데이터가 처리될 때까지 지속적으로 기다리는(busy wait)를 사용한다.
Interrupt driven circular buffer	읽기	ISR(Interrupt Service Routine)에서 받은 데이터를 버퍼에 저장했다가 읽으면 버퍼에서 제거한다.
Interrupt driven zero copy	쓰기	ISR은 내부에 전송할 데이터의 버퍼를 따로 만들지 않고 직접 데이터를 전송한다.
Interrupt driven character queue	읽기 / 쓰기	ISR에서 데이터의 읽기/쓰기 동작을 하기 위해서 데이터 버퍼로 큐(Queue)를 사용한다.

6.3 Poll 모드

이 모드는 데이터가 읽고/쓰기 동작이 완료가 될 때까지 완료 상태비트를 지속적으로 폴링해서 완료가 되면 리턴하는 방식이다.

장점	단점
<ul style="list-style-type: none"> 1. 구현이 간단하다. 2. 모든 데이터가 읽거나 쓰여진 다음에 리턴한다. 3. 드라이버는 전송할 데이터 버퍼가 필요 없다. 	<ul style="list-style-type: none"> 1. 데이터를 전송하거나 받는 동안 이 것이 완료가 되었는지 확인하기 위해 지속적으로 폴링을 하기 때문에 태스크가 Ready 또는 Running 상태로 남게 되어서 CPU 동작 시간을 Polling으로 낭비하게 된다. 2. 여러 태스크가 같은 주변장치를 접근할 경우에 상호배제를 위해 mutex 등을 사용해야 한다.

6.3.1 사용예제

시리얼로 데이터를 1바이트 전송하는 함수이다. 데이터를 전송하기 위해 먼저 데이터 레지스터(DR) 가 빌 때까지 폴링하고 있다가 비면 데이터를 전송하다.

```
portBASE_TYPE xSerial_PutChar(portCHAR cChar, portTickType xBlockTime)
{
    (void) xBlockTime; // Warning 방지용

    /* 데이터 레지스터(DR)이 빌 때까지 기다렸다가 비면 데이터를 DR에 써 넣어서 전송한다. */
    while(USART_GetFlagStatus(USART1, USART_SR_TXE) == RESET);
    USART_SendData( USART1, cChar );

    return pdTRUE;
}
```

6.4 Interrupt driven zero copy 모드

이 모드는 쓰기 동작에 한정된다.

태스크는 드라이버에게 데이터를 쓰기 함수를 호출할 때 쓸 데이터의 개수와 전송할 데이터의 첫번째 주소만 넘기고 호출을 끝낸다. 그러면 드라이버는 바로 직접적으로 쓰지 않고, 인터럽트를 통해서 데이터 쓰기(전송)을 한다.

장점	단점
<ul style="list-style-type: none"> 1. 인터럽트로 동작하기 때문에 직접적으로 쓰기 동작만 할 때만 CPU시간을 소비한다. 2. 쓰기 함수를 콜하자마자 즉시 리턴하기 때문에 데이터 쓰기 동작하고 있는 동안에도 태스크는 다른 일을 처리할 수 있다. 3. 데이터를 위한 버퍼가 필요하다. 	<ul style="list-style-type: none"> 1. 사용하기가 복잡하다.

6.5 Interrupt driven circular buffer 모드

이 모드는 읽기 동작에 한정된다.

태스크는 드라이버에게 데이터를 읽기 함수를 호출하면 드라이버는 직접 주변장치에서 데이터를 읽어 오지 않고 인터럽트 서비스 루틴에서 데이터를 받아서 저장해 놓은 버퍼에서 데이터를 받아오는 방식이다.

장점	단점
<ul style="list-style-type: none"> 1. 구현이 간단하다. 2. 타이아웃을 구현할 수 있다. 3. 데이터를 받는 동작이 인터럽트에서 자동으로 이루어지므로 읽기 함수를 호출하지 않아도 데이터를 놓치지 않는다. 	<ul style="list-style-type: none"> 1. 버퍼를 위한 여분의 메모리 공간이 필요하다

6.6 Interrupt driven character queue 모드

이 모드는 데이터가 읽고/쓰기에 모두 사용할 수 있다.

드라이버는 쓰기를 할 때 바로 주변장치에 접근해서 쓰지 않고 데이터를 큐에 넣는다. 그리고 인터럽트 서비스 루틴에서는 큐에서 데이터를 가지고 와서 주변장치에 데이터를 쓴다.

읽기를 할 때는 인터럽트에서 데이터를 받아서 큐에 넣어 놓으면 이 것을 받아서 리턴하는 방식이다.

장점	단점
----	----

- | | |
|--|---|
| <ol style="list-style-type: none"> 1. 구현이 간단하다. 2. 타임아웃을 구현할 수 있다. 3. 주변장치에서 받은 데이터를 자동으로 큐에 저장이 되기 때문에 데이터를 잃어버리지 않는다. | <ol style="list-style-type: none"> 1. 큐를 위한 메모리 공간이 필요하다. 2. 많은 양의 데이터를 주고 받을 경우에는 큐를 사용하는 것은 불리하다. |
|--|---|

6.6.1 사용예제 1

시리얼로 데이터를 1바이트 전송하는 함수이다. 데이터를 전송하기 위해 큐에 데이터를 넣고 나서 인터럽트를 활성화 시킨다.

```
portBASE_TYPE xSerial_PutChar(portCHAR cChar, portTickType xBlockTime)
{
    /* Queue 에 한 문자를 넣는다. 만약 Queue가 꽉 찾을 경우는 errQUEUE_FULL 를 리턴한다 */
    if(xQueueSendToBack(xSerialTxQueue, &cChar, xBlockTime) != pdPASS)
    {
        return pdFAIL;
    }

    /* 인터럽트를 활성화시킨다 */
    USART_ITConfig( USART1, USART_IT_TXE, ENABLE );
    return pdTRUE;
}
```

6.6.2 사용예제 2

시리얼로 데이터를 받는 함수이다. 데이터를 큐에서 데이터를 받아온다.

```
portBASE_TYPE xSerial_GetChar(portCHAR *pcChar, portTickType xBlockTime)
{
    if(xQueueReceive(xSerialRxQueue, pcChar, xBlockTime) == pdFAIL)
    {
        return pdFAIL;
    }
    else
    {
        return pdTRUE;
    }
}
```

인터럽트 서비스 루틴에서는 전송할 데이터를 하나씩 시리얼로 보내고, 받은 데이터는 큐에 저장한다.

```
void USART1_IRQHandler(void)
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    char cChar;

    /** ===== 송신 인터럽트 처리 ===== */
    /**
     *      Tx 버퍼(DR 레지스터)가 비어서 인터럽트가 발생하면
     *      Queue에서 데이터를 하나씩 가지고 와서 UART로 송신한다
     */
    if( USART_GetITStatus( USART1, USART_IT_TXE ) == SET )
    {
```

```
/* Queue 핸들이 있는지 확인한다. */
if(xSerialTxQueue == NULL)
{
    return;
}

/* Queue 에서 데이터를 가지고 와서 UART로 송신한다. */
if( xQueueReceiveFromISR( xSerialTxQueue, &cChar, &xHigherPriorityTaskWoken ) ==
pdTRUE )
{
    USART_SendData( USART1, cChar );
}
else
{
    USART_ITConfig( USART1, USART_IT_TXE, DISABLE );
}

/** ===== 수신 인터럽트 처리 ===== */
/*
*      Rx 버퍼(DR 레지스터)가 비지 않음 인터럽트가 발생하면
*      Queue에 데이터를 하나씩 집어 넣는다.
*/
if( USART_GetITStatus( USART1, USART_IT_RXNE ) == SET )
{
    cChar = (portCHAR)USART_ReceiveData(USART1);

    /* Queue 핸들이 있는지 확인한다. */
    if(xSerialRxQueue == NULL)
    {
        return;
    }

    /* Queue 에 UART로 들어온 데이터를 넣는다. */
    xQueueSendToBackFromISR(xSerialRxQueue, &cChar, &xHigherPriorityTaskWoken );
}

/* 높은 우선 순위가 있으면 테스크 스위칭을 한다. */
portEND_SWITCHING_ISR(xHigherPriorityTaskWoken);
}
```