석 사 학 위 논 문

상용게임기를 이용한 임베디드 소프트웨어 교육 환경 구성 연구

2013년

한성대학교 대학원

컴 퓨 터 공 학 과

컴퓨터공학전공

장 영 준

석 사 학 위 논 문
지도교수 이민석

상용게임기를 이용한 임베디드 소프트웨어 교육 환경 구성 연구

Implementation of Embedded Software Educational Environment Using Portable Game Console

2012년 12월 일

한성대학교 대학원
컴 퓨 터 공 학 과
컴퓨터공학전공
장 영 준

석 사 학 위 논 문
지도교수 이민석

상용게임기를 이용한 임베디드 소프트웨어 교육 환경 구성 연구

Implementation of Embedded Software Educational Environment Using Portable Game Console

위 논문을 공학 석사학위 논문으로 제출함

2012년 12월 일

한성대학교 대학원
컴 퓨 터 공 학 과
컴 퓨 터 공 학 전 공
장 영 준

장영준의 공학 석사학위논문을 인준함

2012년 12월 일

심사위원장 장 재 영 인

심 사 위 원 이 민 석 인

심 사 위 원 허 준 영 인

국 문 초 록

상용 게임기를 이용한 임베디드 소프트웨어 교육 환경 구성 연구

한성대학교 대학원
컴퓨터공학과
컴퓨터공학전공
장 영 준

본 논문은 기존의 임베디드 시스템 교육 환경이 가지고 있는 여러 가지 문제점에 대한 분석을 통해 이를 해결하고자 닌텐도 DS와 같은 상용 게임기를 이용한 임베디드 소프트웨어 교육 환경 구성에 대해 소개한다. 연구에서는 임베디드 소프트웨어 교육을 위한 실습 환경이 갖추어야 할 여러 가지 요소들을 상용 게임기의 게임팩과 같은 형태로 개발, 설치한 뒤 게임기로의 프로그램 다운로드 및 실행, 원격 디버깅이 가능하게 하는 게임기용 소프트웨어와 개발 호스트에서 실행되는 프로젝트 관리, 소스 프로그램 편집, 컴파일, 다운로드 및 원격 디버깅을 통합 처리하는 통합 개발 환경을 개발함으로서 기존의 임베디드 소프트웨어 교육 환경보다 좀 더 낮은 비용으로 현실감 있고, 품질이 높은 임베디드 소프트웨어 교육 환경을 제공한다.

【주요어】 임베디드 소프트웨어, 상용 게임기, 통합 개발 환경, 공개 소스, 소프트웨어 교육, 실시간 운영체제, 원격 디버깅

목 차

【 표 목 차 】

【 그 림 목 차 】

제 1 장 서 론

제 1 절 연구 동기

임베디드 소프트웨어는 IT 뿐만 아니라 전 융합 산업에 걸쳐 활용되는 기술 분야로서 전문 인력에 대한 수요가 지속적으로 증가하고 있다. 임베디드 시스템 또는 임베디드 소프트웨어 교육은 일반적으로 임베디드 하드웨어 보드와 그와 연관된 소프트웨어 개발 도구를 이용하여 이루어진다.

기존의 임베디드 시스템 교육 환경은 다음과 같은 단점을 가지고 있다. 첫째, 임베디드 시스템 개발 장비는 세부 사양에 따라 적어도 백만 원에서 수백만 원에 이르기 때문에 교육 기관에 상당한 비용 부담을 주며, 개인 개발자가 구입하기에는 가격이 매우 비싸다. 둘째, 기존 장비는 스마트폰, 게임기 등 실생활에서 익숙해져 있는 임베디드 시스템과는 전혀 다른 모양으로 임베디드 시스템을 처음 배우는 학생들에게 친근하지 않아 현실감이 부족하여 동기 유발이 어렵다. 셋째, 기존의 임베디드 시스템 개발 장비의 대부분은 고가의 고급 디버깅 장비를 필요로 하는 등 원격 디버깅 기능이 부족하여 품질 높은 임베디드 소프트웨어 개발 방법을 교육하는데 부족함이 많다.

본 논문에서는 위와 같은 문제를 해결하기 위하여 비교적 가격이 저렴한 상용 게임기 하드웨어, 공개 소스 개발 도구, 공개 소스 실시간 운영체제 등을 활용하여 적은 비용으로 높은 교육 효과를 얻을 수 있는 임베디드 시스템 또는 임베디드 소프트웨어 교육 실습을 수행하기 위한 개발 환경을 구성하였다.

제 2 절 연구 목적 및 필요성

본 논문에서는 닌텐도 DS-lite와 같은 상용 게임기 하드웨어를 이용하여 임베디드 시스템 또는 임베디드 소프트웨어 교육 실습을 수행하기 위한 개발 환경을 구성하여, 그 결과를 대학에서의 임베디드 소프트웨어 연구와 교육, 전문 교육기관, 고등학교에서의 임베디드 소프트웨어 교육, 초등학교의 로봇 프로그래밍 도구 등으로 활용할 수 있도록 하는데 목적이 있다.

본 연구는 기존 임베디드 시스템 개발 환경이 가진 여러 가지 문제점을 극복하고자 하는 동기에서 시작되었으며, 연구의 필요성은 다음과 같이 정리할 수 있다.

1. 기존의 임베디드 시스템 개발 장비는 가격이 매우 비싸다.

임베디드 시스템 개발 장비는 세부 사양에 따라 적어도 백만 원에서 수백만 원에 이르기 때문에 교육 기관에 상당한 비용 부담을 주며, 개인 개발자가 구입하기에는 가격이 매우 비싸다. 반면에 닌텐도 DS-lite와 같은 휴대용 게임기의 경우 현재도 시중에서 10만원 내외의 가격으로 구입이 가능하다.

2. 학생들에게 친근한 환경이 필요하다.

기존의 임베디드 시스템 개발 장비들은 임베디드 시스템을 처음 배우는 학생들에게 친근하지 않은 형상이다. 실제 많은 학생들은 임베디드 시스템 관련 업종으로 취업을 하는 경우, 휴대폰, 스마트폰, 게임기와 같은 휴대용 단말의 하드웨어, 소프트웨어를 개발하게 되며, 이미 그런 임베디드 시스템에 익숙해져 있는 반면에 기존 임베디드 시스템 개발 장비는 전혀 다른 모양으로 임베디드 시스템을 처음 배우는 학생들에게 친근하지 않아 현실감이 부족하여 교육 실습을 진행하는데 동기 유발이 어렵다. 반면에 게임

기에 자신의 응용 프로그램을 만들어 실행하는 경우, 흥미와 성취감을 불러 일으켜서 교육의 성과를 높일 수 있는 기대효과를 가지고 있다.

3. 기존 임베디드 시스템 개발 환경에서는 완벽한 개발 환경의 제공이 미흡하다.

기존의 임베디드 시스템 개발 장비의 대부분은 원격 디버깅 기능이 부족하여 품질 높은 임베디드 소프트웨어 개발 방법을 교육하는데 부족함이 많았다. 물론, 이러한 점도 하드웨어 디버깅 장비를 이용하면 해결할 수 있는 부분이지만, 하드웨어 디버깅 장비 역시 매우 고가의 장비이기 때문에 교육 기관 또는 개인 개발자가 구입하기에는 상당한 부담을 준다. 반면에 본 논문에서는 이클립스 기반의 통합 개발 환경 하에서 원격 디버깅을 가능하게 함으로써 품질이 높은 소프트웨어를 개발하기 위한 충분한 환경을 제공한다.

4. 게임기는 다양한 임베디드 장치들을 내장하고 있다.

게임기들은 종류에 따라 차이는 있지만, 고성능, 저전력 CPU, 그래픽 가속 장치, 버튼, 터치스크린, 진동 모터, 각종 센서, 사운드 등 임베디드 시스템 교육 장비들이 제공하고 있는 거의 모든 입출력 장치를 모두 가지고 있기 때문에, 그런 장치들을 개념적으로 연계하는 고급 임베디드 프로그램이 매우 용이하다. 하지만 게임기의 특성과는 다른 입출력 장치, 외부 장치와의 연결 관점에서는 상용 임베디드 시스템 교육 장비에 비하여 부족한 점이 있기 때문에 교육에 필요한 경우 이 부분을 가상 디바이스, 외부와 원활한 인터페이스를 하기 위한 간단한 장치의 제공을 통해서 보완할 필요가 있다.

제 3 절 연구 범위

본 논문에서는 임베디드 소프트웨어 교육을 위한 실습 환경이 갖추어야 할 여러 가지 요소들을 상용 게임기의 게임팩과 같은 형태로 개발 설치한 뒤, 게임기로의 프로그램 다운로드 및 실행, 원격 디버깅이 가능하게 하는 게임기용 소프트웨어와 개발 호스트에서 실행되는 프로젝트 관리, 소스 프로그램 편집, 컴파일, 다운로드 및 원격 디버깅을 통합 처리하는 일련의 개발 도구 및 환경을 구현하였다. 성공적인 임베디드 소프트웨어 교육을 위해 고려가 필요한 소프트웨어 요소들은 다음과 같다.

1. 타겟 시스템(상용 게임기) 측 개발 환경

타겟 시스템은 개발 과정에서 호스트 컴퓨터에서 실행되는 통합 개발 환경의 동작에 대응하는 모듈들과 응용 프로그램의 실행의 기반이 되는 운영체제 및 라이브러리를 포함하여야 한다. 여기에는 실행 바이너리를 수신하고 수신된 바이너리들을 관리하고 실행하기 위한 쉘, IDE(통합 개발 환경)의 디버깅 기능에 대응하여 타겟 시스템에 다운로드 된 소프트웨어를 제어하기 위한 디버그 스텁 모듈, 실시간 운영체제, 이 운영체제 위에서 여러 편의 기능을 제공하기 위한 통신, 파일 시스템 등의 라이브러리 등이 포함된다.

2. 호스트 시스템 측 개발 환경

개발 호스트와 타겟 시스템이 같지 않은 환경에서, 호스트 시스템에서 모든 개발 과정을 마치고, 타겟 시스템에 개발된 소프트웨어(바이너리)를 다운로드하고 원격으로 디버깅을 할 수 있는 환경, 교차 개발 환경에는 호스트 시스템 측에서 실행되는 IDE(통합 개발 환경 - 프로젝트 관리 도구, 컴파일러, 어셈블러, 링커, 디버거), 다운로더, 타겟 시뮬레이터 등이 포함된다.

제 4 절 논문 구성

본 논문의 구성은 다음과 같다.

- 2장 임베디드 소프트웨어 교육 환경 구성

  상용 게임기를 이용하여 임베디드 소프트웨어 교육 환경을 구성하기 위한 개발 도구의 활용 및 구현에 대하여 기술한다.

- 3장 교육 실습 개발 환경

  본 논문에서 구성한 교육 환경을 이용하여 교육 실습을 하기 위해 개발 환경을 구축하기 위한 과정에 대하여 기술한다.

- 4장 활용 방안

  본 논문에서 기술한 상용 게임기를 이용한 임베디드 소프트웨어 교육 환경에 대하여 실제 활용할 수 있는 방안에 대하여 기술한다.

- 5장 결론 및 향후 연구

  본 논문에서 구성한 교육 환경에 대한 결론을 내리며 향후 연구 사항들에 대하여 기술한다.

제 2 장 임베디드 소프트웨어 교육 환경

제 1 절 전체 교육 환경 구성

본 논문에서는 닌텐도 DS-lite와 같은 상용 게임기 하드웨어를 이용하여 임베디드 시스템 또는 임베디드 소프트웨어 교육 실습을 수행하기 위한 개발 환경을 구성하여, 그 결과를 대학에서의 임베디드 소프트웨어 연구와 교육, 전문 교육기관, 고등학교에서의 임베디드 소프트웨어 교육, 초등학교의 로봇 프로그래밍 도구 등으로 활용하기 위한 것으로 기술적으로는 임베디드 소프트웨어 교육을 위한 실습 환경이 갖추어야 할 여러 가지 요소들을 상용 게임기의 게임팩과 같은 형태로 개발 설치한 뒤, 게임기로의 프로그램 다운로드 및 실행, 원격 디버깅이 가능하게 하는 게임기용 소프트웨어와 개발 호스트에서 실행되는 프로젝트 관리, 소스 프로그램 편집, 컴파일, 다운로드 및 원격 디버깅을 통합 처리하는 일련의 개발 도구 및 환경을 구현하였다. 논문에서 구현한 전체 임베디드 소프트웨어 교육 환경은 그림 2.1과 같다.

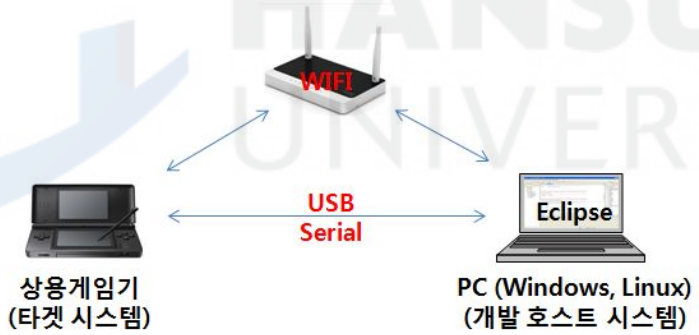


그림 2.1 임베디드 소프트웨어 교육 환경

임베디드 소프트웨어 개발 환경의 소프트웨어 구성은 그림 2.2와 같다. 그림에서 볼 수 있는 바와 같이 구성에 포함된 소프트웨어 가운데 가능한 모든 부분에 기존 공개 소스 소프트웨어를 활용하였다.

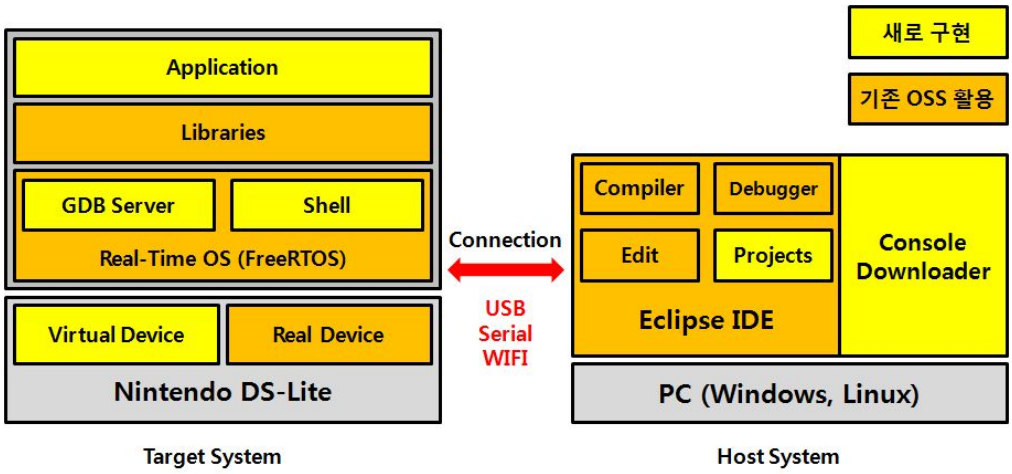


그림 2.2 임베디드 소프트웨어 교육 환경 세부 구성

타겟 시스템인 닌텐도 DS-lite에는 실시간 운영체제로 공개 소스인 FreeRTOS를 이식하여 사용하였으며, 원격 디버깅을 위하여 GDB 서버(스텁 모듈)을 구현하였다. 그리고 호스트 시스템에서 개발된 프로그램을 다운로드 받고, 관리하기 위한 쉘 프로그램을 새로 구현하였다. 또 임베디드 소프트웨어 교육에 활용하기 위하여 실제 닌텐도 DS-lite가 가지고 있지 않은 장치들을 가상적으로 시뮬레이트 하는 가상 디바이스를 추가하였다. 타겟 시스템에서 실행되는 프로그램과 링크되는 라이브러리는 devkitPro, Palib를 수정하여 사용하였다.

호스트 시스템은 리눅스 운영체제나 윈도우즈 운영체제를 모두 활용할 수 있도록 하였다. 호스트의 통합 개발 환경으로는 이클립스 CDT를 이용하며, 컴파일러는 devkitPro를 사용한다. 이클립스 CDT는 devkitPro가 제공하는 GNU C 컴파일러, 어셈블러, 링커, 디버거를 사용한다. 이클립스는 Java 개발에 주로 사용되는 IDE로서 많은 학생들이 이미 익숙해져 있기 때문에, 별도로 사용 교육이 필요 없는 도구이다. 버전 관리도구로는 SVN, Git을 적용하였다. 또한, 개발된 실행 파일을 타겟 시스템에 USB또는 무선 랜을 통하여 다운로드하기 위한 콘솔 다운로더를 구현하였다. 모든 개발은 이클립스 IDE에서 이루어지며, 빌드 된 바이너리 이미지는 다운로더를 통해 타겟 시스템인 닌텐도 DS-lite로 전송되어 실행된다. 또한,

호스트 시스템에서 닌텐도 DS 시뮬레이터를 이용하여 실행하는 것도 가능하며, USB 연결을 통해 원격 디버깅 모드를 이용할 수도 있다.

본 논문에서는 호스트 시스템과의 연결 방법이 무선 랜 밖에 없는 닌텐도 DS-lite에 USB 또는 시리얼 포트 연결성을 부여하고, 가속 센서와 같은 추가적인 장치의 제공, 외부 장치 제어가 가능하도록 그림 2.3의 어댑터 하드웨어를 개발하였다. 무선 랜은 개인 개발자들에게는 매우 유용하나, 보안이 적용된 환경에서는 여러 제약이 있기 때문에 사용이 편리한 USB 연결을 지원하였다. 어댑터 하드웨어는 독자적인 32비트 CPU를 가지고 있으며 기본적으로는 호스트 시스템과 타겟 시스템인 닌텐도 DS-lite 사이의 통신을 중계한다. 호스트 시스템과 어댑터 하드웨어는 USB 또는 시리얼 인터페이스로 연결되며, 어댑터 하드웨어와 닌텐도 DS-lite 사이는 닌텐도의 게임팩 슬롯이 제공하는 SPI 인터페이스를 이용한 통신이 이루어진다. 어댑터 하드웨어에도 본 논문에서 제공하는 실험 환경과 같은 실시간 운영체제인 FreeRTOS를 사용한다.

그림 2.3 확장 어댑터 하드웨어

제 2 절 상용 게임기 하드웨어 (Nintendo DS-lite)

본 논문에서 타겟 시스템으로 사용한 상용 게임기로는 국내에서 가장 높은 시장 점유율을 가진 휴대용 게임기인 닌텐도사의 닌텐도 DS-lite이다. 닌텐도 DS-lite는 국내뿐만 아니라 세계에서 가장 많이 팔린 휴대용 게임기 가운데 하나로, 지금도 시중에서 10만원 내외의 가격으로 구입이 가능하다. 이 게임기에는 ARM9, ARM7등 두 개의 32비트 CPU, 두 개의 LCD 화면, 그래픽 가속 장치, 버튼, 터치스크린, 사운드 입출력, 무선 랜 등 기존 임베디드 시스템 교육 장비가 필요로 하는 대부분의 장치들을 가지고 있다. 표 2.1, 그림 2.4는 각각 닌텐도 DS-lite의 하드웨어 사양 및 닌텐도 DS-lite를 나타낸다.

분 류	사 양
Processor	ARM946E-S 32bit RISC CPU, 66Mhz
	ARM7TDMI 32bit RISC CPU, 33Mhz
LCD	상단: 18bit, 256x192 pixel, LCD only
	하단: 18bit, 256x192 pixel, Touch Screen
RAM	4M Byte SRAM
ROM	256K Byte Flash Memory
통신	무선랜 : 802.11b (WEP)
그래픽	2D, 3D 가속 엔진
기타	키패드, 사운드(마이크, 스피커), 확장 메모리(Dual Slot), 100mAh 배터리

표 2.1 닌텐도 DS-lite 주요 하드웨어 사양

그림 2.4 닌텐도 DS-lite

제 3 절 실시간 운영체제 (FreeRTOS)

실시간 운영체제(RTOS, Real-Time Operating System)는 실시간 응용 프로그램을 위해 개발된 운영체제이다. 본 논문에서는 Richard Berry에 의해 설계된 작고 사용하기 쉽도록 디자인되어 있는 공개 소스 실시간 운영체제인 FreeRTOS를 타겟 시스템인 닌텐도 DS-lite에 이식하고 그 위에서 실시간 운영체제를 이용한 실습 교육 환경을 제공한다. 이 절에서는 본 논문에서 사용하는 실시간 운영체제인 FreeRTOS에 대해 설명한다.

1. FreeRTOS 소프트웨어 구조

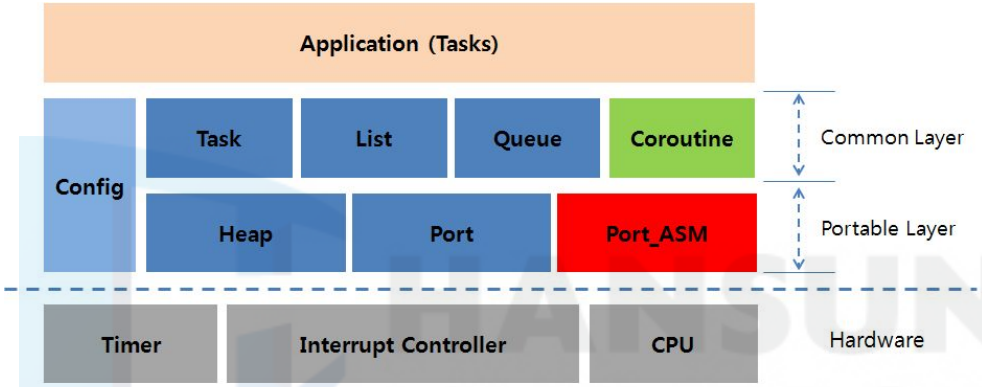


그림 2.5 FreeRTOS 소프트웨어 구조

FreeRTOS의 소프트웨어 구조는 그림 2.5와 같다. 그림에서 볼 수 있는 바와 같이 FreeRTOS는 크게 Common Layer와 Portable Layer로 구분할 수 있다. Common Layer는 시스템에 의존적이지 않아 플랫폼에 상관없이 공통적으로 사용할 수 있는 소스 코드를 포함하는 계층을 말하며, Portable Layer는 FreeRTOS를 이용하려는 사용자의 목적에 맞도록 운영체제에 대한 기능을 설정하거나 시스템에 의존적인 기능을 직접 구현 또는 적절하게 수정하여 사용해야 하는 소스 코드를 포함하는 계층을 말한다.

FreeRTOS의 대부분의 기능은 Task에 밀집되어 있으며 내부적으로 데

이터를 관리하기 위해 List와 Queue가 존재한다. 그 중에서 Queue는 프로세스 간 통신, 동기화 문제를 해결하기 위해 사용된다.

FreeRTOS는 메모리 관리, 인터럽트 서비스 루틴, 타이머, 디바이스 관리 등에 관한 부분을 Portable Layer에 위치해놓고 있다. 따라서 이 부분은 FreeRTOS를 이용하는 사용자가 이용하려는 환경에 맞게 적절하게 구현하여 사용하여야 한다.

2. FreeRTOS 소스 구성

FreeRTOS는 기본적으로 그림2.6과 같은 디렉토리 구조로 되어있다.

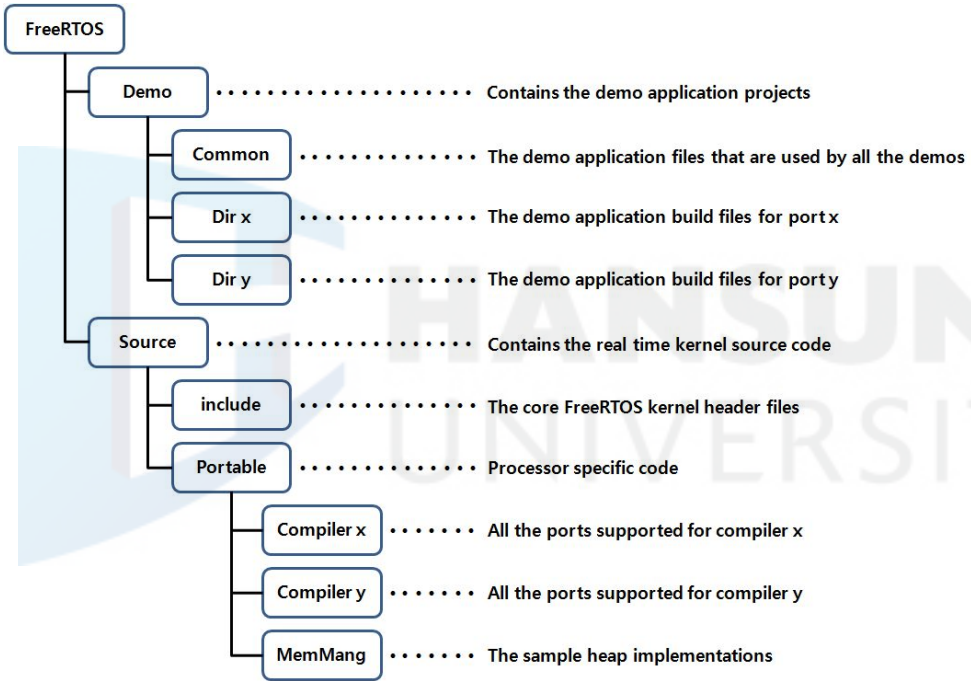


그림 2.6 FreeRTOS 디렉토리 구조

Demo 디렉토리는 각 포트(프로세서, 컴파일러)에 맞게 구현된 데모 어플리케이션 소스 코드와 프로젝트들로 구성되어 있으며, Source 디렉토리는 FreeRTOS 커널 소스 코드를 포함하고 있다.

FreeRTOS/Source 디렉토리에는 모든 프로세서 아키텍처에서 공통적으

로 사용할 수 있는 tasks.c, queue.c, list.c, coroutine.c 4개의 파일이 위치하고 있다. 이 중 tasks.c queue.c, list.c 3개의 파일이 FreeRTOS의 핵심 커널 소스코드이며, 코루틴의 기능은 운영체제의 환경 설정을 통해 사용 여부를 결정할 수 있다. (최신 버전의 FreeRTOS에는 소프트웨어 타이머 기능을 구현한 timers.c가 추가되어 있다.)

FreeRTOS의 Portable Layer에 해당하는 프로세서 아키텍처에 의존적인 코드들은 FreeRTOS/Source/Portable/[Compiler]/[Architectures] 디렉토리에 위치하고 있으며, 메모리 관리에 대한 기능 또한 Portable Layer에 포함하고 있다. 이러한 디렉토리 구조는 사용자가 원하는 방식으로 바꿀 수 있으며, 디렉토리 구조를 변경할 경우 이에 맞도록 Makefile을 수정해야 한다.

본 논문에서는 FreeRTOS를 닌텐도 DS-lite의 CPU인 ARM 프로세서에 맞게 이식하고 FreeRTOS를 이용한 임베디드 소프트웨어 실습 교육을 진행할 수 있도록 그림 2.7과 같이 프로젝트의 디렉토리 구조를 구성하였다.

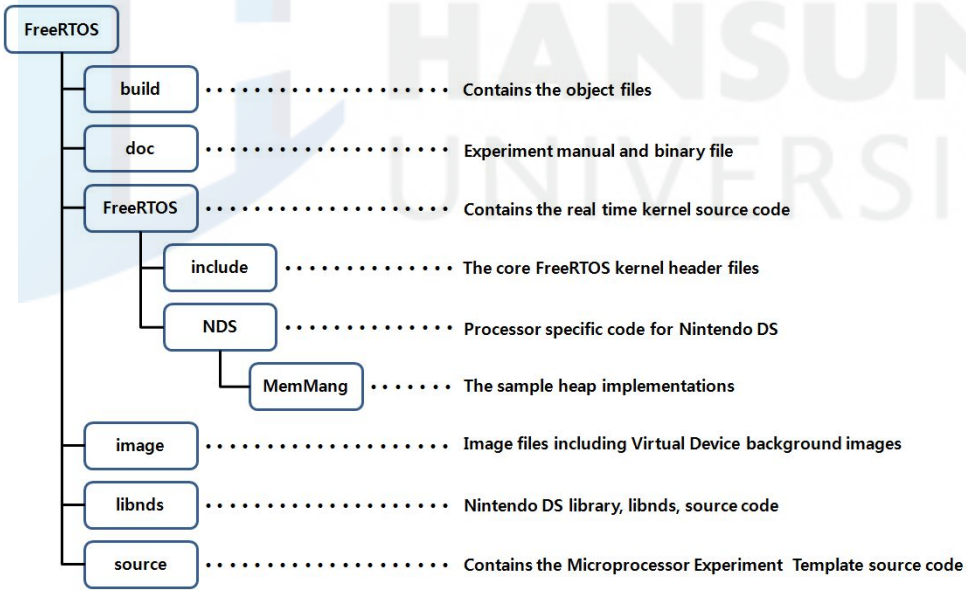


그림 2.7 임베디드 소프트웨어 교육 환경 프로젝트 디렉토리 구조

제 4 절 실시간 운영체제 이식

FreeRTOS의 이식은 닌텐도 DS-lite에서 사용하고 있는 ARM9 프로세서의 레지스터와 중첩 인터럽트 모델(Nested Interrupt Model)을 적용하기 위한 FreeRTOS의 변수 등 문맥(Context)을 구성하고, 문맥 전환(Context Switching) 기능을 하기 위한 함수의 구현, 닌텐도 DS-lite의 타이머 장치를 이용하여 타이머 인터럽트 등록을 통해 이루어진다. 이 과정은 닌텐도 DS-lite와 ARM9 프로세서에 맞도록 플랫폼 종속적인 데이터 구조의 수정을 포함한다. 따라서 이 절에서는 실시간 운영체제를 이식하기 위해 필요한 ARM 프로세서의 동작모드, 레지스터 구성 등에 대해 설명하고 실시간 운영체제를 이식하는 과정에 대해 설명한다.

1. ARM 프로세서 동작모드

동작(Operation)모드는 프로세서가 어떤 권한을 가지고 어떤 종류의 작업을 처리하고 있는지를 나타내는 것으로 ARM 프로세서는 7가지의 동작모드를 지원한다. 각 동작모드에 대한 설명은 다음과 같다.

1) User 모드

ARM 프로세서가 사용자 태스크나 어플리케이션을 수행할 때의 동작모드이다. User 모드는 ARM 프로세서의 7가지 동작모드 중 유일하게 비특권(Un-privileged)모드이다. User 모드는 메모리, I/O 장치와 같은 시스템 자원을 사용하는데 제한을 두어 사용자의 실수를 미연에 방지할 수 있도록 관리할 수 있다.

2) FIQ 모드

ARM 프로세서는 크게 2개의 인터럽트 소스를 가지고 있는데, 하나는

일반적인 인터럽트 처리 용도로 사용하고, 다른 하나는 아주 빠르게 처리할 수 있도록 구성하여 사용하고 있다. FIQ(Fast Interrupt Request)모드는 여기서 빠른(Fast) 인터럽트를 처리하는 모드이다. 외부 장치에서 요청되는 하드웨어적인 FIQ의 발생에 의해 ARM 프로세서는 FIQ 모드로 전환하고 인터럽트를 처리한다.

3) IRQ 모드

ARM 프로세서에서 일반적으로 사용되는 인터럽트인 IRQ(Interrupt Request)를 처리하는 모드이다. 외부 장치에서 요청되는 하드웨어적인 IRQ의 발생에 의해 ARM 프로세서는 IRQ 모드로 전환하고 인터럽트를 처리한다.

4) SVC(Supervisor) 모드

SVC(Supervisor) 모드는 대부분의 시스템 자원을 자유롭게 관리할 수 있는 동작모드이다. 따라서 주로 운영체제의 커널이나 디바이스 드라이버를 처리할 때 사용되는 동작모드이다. ARM 프로세서는 리셋 신호가 입력되거나 소프트웨어 인터럽트(SWI)가 사용되면 SVC 모드로 전환된다.

5) Abort 모드

메모리에서 명령을 읽거나 데이터를 읽거나 쓸 때 오류가 발생하면 ARM 프로세서는 Abort 모드로 전환하여 메모리 오류를 처리한다. Abort 오류는 MMU, MPU 또는 외부의 메모리 제어기에서 구동되는 Abort 신호에 의해서 발생된다.

6) Undefined 모드

ARM 프로세서가 명령어를 읽어 실행하고자 하나 읽어온 명령이 디코더에 정의되어 있지 않은 명령인 경우 발생되는 오류를 처리하기 위한 동작모드이다.

7) System 모드

User 모드와 동일한 용도로 사용된다. 하지만 특권(Privilege) 모드라는 점이 다르다. User 모드와 System 모드를 제외한 다른 동작모드는 모두 외부의 요청이나 오류에 의해서 모드가 전환된다. 이와 같은 외부의 오류나 요청을 예외처리라 한다. User 모드와 System 모드는 외부의 요청이나 오류가 아닌 프로그래머의 필요에 의해 소프트웨어적으로만 변경이 가능한 동작모드이다.

본 논문에서 사용한 실시간 운영체제인 FreeRTOS는 System 모드에서 사용자 태스크가 실행되며, 멀티태스킹이 이루어지는 과정에서는 System 모드만이 아닌 다른 여러 동작모드로 전환되며 실행된다. ARM 프로세서는 각 동작 모드마다 별도의 레지스터를 갖도록 구성되어 있으므로 문맥 전환이 이루어지는 과정에서는 레지스터의 값을 동작 모드에 맞게 정확하게 저장, 복원되도록 하는 것이 필요하다.

2. ARM 프로세서 레지스터

ARM 프로세서는 총 37개의 32비트 길이의 레지스터를 가지고 있다. 37개의 레지스터는 데이터 연산 등에 사용되는 범용 레지스터 30개와 프로그램 제어 목적으로 사용되는 1개의 PC(프로그램 카운터), 프로세서의 동작 상태를 나타내는 상태 레지스터 6개로 구성된다. ARM 명령에서는 1개의 PC를 포함하여 모두 16개의 레지스터를 이용하여 각종 데이터 처리 및 연산을 하기 위한 프로그램을 작성한다. 프로그램에서는 레지스터를 쉽게 사용할 수 있도록 R0에서 R15까지의 키워드를 부여하여 사용하며,

R15는 PC가 사용된다. 일반적으로 PC는 명령어를 읽어올 위치 정보를 가지고 있다. ARM 프로세서 레지스터에 대한 자세한 내용은 다음과 같다.

1) 범용 레지스터

범용 레지스터(R0 - R15)는 데이터 처리나 데이터 전송 등 다양한 목적으로 사용된다. 범용 레지스터는 다음과 같이 3개의 그룹으로 분류할 수 있다.

- The unbanked registers, R0-R7
- The banked registers, R8-R14
- Register 15, the PC, is described in The Program counters, R15

Banked 레지스터는 ARM 프로세서의 모든 동작모드에서 공유하여 사용할 수 있는 레지스터(R0 - R7)를 의미한다. Unbanked 레지스터는 각 동작모드별로 할당되어 있는 레지스터를 의미하며, 각 동작모드별로 할당되어 있는 레지스터는 서로 다른 동작모드에서는 사용할 수 없다. 이렇게 일부 레지스터를 별도로 할당하여 사용하는 것은 ARM 프로세서가 동작모드를 전환하여 특정 작업을 처리하도록 되어있는 특징과 밀접한 관계가 있다. ARM 프로세서의 동작모드의 전환은 예외처리에 의해서 발생한다. 예외처리가 발생하면 ARM 프로세서는 기존에 처리하던 사용자 프로그램을 멈추고 미리 정해진 위치에 있는 새로운 프로그램을 처리하도록 되어 있다. 이때 예외처리 후 되돌아가기 위해서는 기존에 사용자 프로그램에서 사용되던 레지스터나 상태 값을 그대로 복원해야 하고, 이 복원할 값을 백업하기 위해서 이와 같은 별도의 레지스터를 가지게 된다. SPSR에는 CPSR을 백업하고, R14(LR) 레지스터에는 기존 프로그램 위치, 죽 PC 값을 백업하여 보관한다. 그리고 R13(SP) 레지스터에는 각 동작모드의 프로그램에서 사용되는 스택 메모리의 위치 정보를 가지고 있도록 한다. 그림 2.8은 ARM 프로세서의 전체 레지스터 구성을 나타낸다.

Modes						
	Privileged modes					
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

그림 2.8 ARM 프로세서 전체 레지스터 구성

2) 상태 레지스터

ARM 프로세서는 두 종류의 상태 레지스터(Program Status Register)를 가진다. 상태 레지스터는 1개의 CPSR와 5개의 SPSR로 구성되어 있다. CPSR(Current Program Status Register)은 현재 동작중인 프로세서의 상태를 나타내고 있는 레지스터이며, SPSR(Saved Program Status Register)은 이전 동작모드의 CPSR의 복사본으로 표현되는 정보는 CPSR과 동일하다. ARM 프로세서는 User 모드, System 모드를 제외한 나머지 5개의 동작모드는 예외처리에 의해서 모드가 전환된다. SPSR은 예외처리에 의하여 동작 모드가 전환될 때 이 전 동작모드의 CPSR을 복사해 보관하는 레지스터이다. SPSR은 User 모드, System 모드를 제외한 SVC(Supervisor), Abort, Undefined, IRQ. FIQ의 5개 동작모드에 각각 하나씩 할당되어 사용된다.

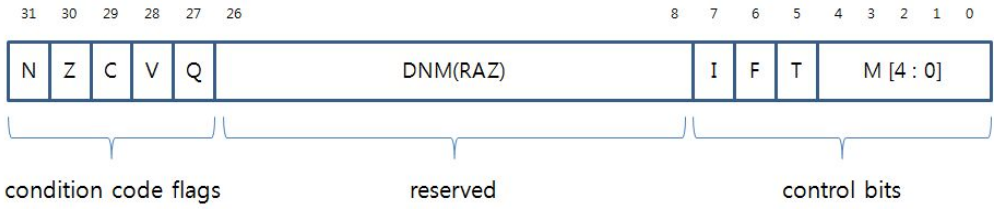


그림 2.9 ARM 프로세서 상태 레지스터 구성

그림 2.9는 상태 레지스터의 비트 구성을 나타낸다. 상태 레지스터는 ALU 연산 결과에 대한 정보를 나타내는 플래그 비트와 동작모드, IRQ/FIQ 인터럽트의 제어(enable / disable), ARM/Thumb 상태 정보를 나타내는 컨트롤 비트로 구성되어 있다. 각 비트에 대한 설명은 다음과 같다.

(1) 플래그 비트

플래그(Flag) 비트는 ALU의 연산 결과에 대한 N, Z, C, V, Q 상태 정보를 나타낸다. 표 2.2는 각 비트에 대한 설명을 나타낸다.

비 트	설 명
Negative 플래그(N)	연산의 결과가 마이너스인 경우에 세트된다.
Zero 플래그(Z)	연산의 결과가 0인 경우에 세트된다.
Carry 플래그(C)	자리 올림이나 내림이 발생한 경우, 그리고 쉬프트 연산 등에서 캐리가 발생되는 경우에 세트된다.
Overflow 플래그(V)	연산의 결과가 오버플로우가 되었을 때 세트된다.
Q 플래그(Q)	포화(saturation)연산 명령의 수행 결과 포화가 발생하면 세트된다.

표 2.2 ARM 프로세서 상태 레지스터 플래그 비트

(2) 컨트롤 비트

컨트롤(Control) 비트는 프로세서의 7가지 동작모드와 IRQ/FIQ 인터럽트를 제어(enable / disable)하고, ARM/Thumb 상태 정보를 가지고 있다. 표 2.3은 각 비트에 대한 설명을 나타내며, 표 2,4는 모드 비트에 대한 동작 모드를 나타낸다.

비 트	설 명
I / F 비트	'I'와 'F'비트를 세트(1)하면 CPU는 외부로부터 입력되는 IRQ와 FIQ를 처리하지 않는다.
T 비트	'T'비트가 세트(1)되어 있으면 Thumb 상태, 클리어(0)되어 있으면 ARM 상태를 나타낸다. 'T'비트는 읽기 전용이며, 사용자가 강제로 이 비트를 제어할 수 없다.
Mode 비트	Mode 비트는 ARM 프로세서의 7가지 동작 모드를 나타낸다.

표 2.3 ARM 프로세서 상태 레지스터 컨트롤 비트

M [4 : 0]	동작 모드
10000	User 모드
10001	FIQ 모드
10010	IRQ 모드
10011	SVC 모드
10111	Abort 모드
11011	Undefined 모드
11111	System 모드

표 2.4 ARM 프로세서 상태 레지스터 동작 모드 비트

3. FreeRTOS 자료형 및 기능 설정

FreeRTOS를 닌텐도 DS-lite에 이식하는 과정은 닌텐도 DS-lite와 ARM9 프로세서에 맞도록 플랫폼 종속적인 데이터 구조를 수정하고, 코루

틴(Coroutine), 소프트웨어 타이머 기능의 사용, 선점(Preemptive)/비선점(Nonpreemptive) 스케줄링 정책 등 FreeRTOS에서 제공하는 여러 기능들의 사용 여부를 설정하는 과정을 포함한다.

표2.5는 본 논문에서 구성한 portmacro.h 헤더 파일을 나타낸다. portmacro.h 헤더 파일에는 닌텐도 DS-lite의 ARM9 프로세서에 맞게 FreeRTOS 내부에서 사용하는 기본 자료형 및 스택이 자라는 방향 등을 설정하고, 인터럽트, 임계구역, 함수 및 함수의 프로토타입 등과 관련된 처리를 하기 위한 매크로 함수 설정에 대한 내용을 포함하고 있다.

```
#ifndef PORTMACRO_H
#define PORTMACRO_H

#ifdef __cplusplus
extern "C" {
#endif

/* Type definitions. */
#define portCHAR          char
#define portFLOAT         float
#define portDOUBLE        double
#define portLONG          long
#define portSHORT         short
#define portSTACK_TYPE    unsigned portLONG
#define portBASE_TYPE     portLONG

#if( configUSE_16_BIT_TICKS == 1 )
  typedef unsigned portSHORT portTickType;
  #define portMAX_DELAY ( portTickType ) 0xffff
#else
  typedef unsigned portLONG portTickType;
  #define portMAX_DELAY ( portTickType ) 0xffffffff
#endif
```

```
/*-----------------------------------------------------------*/
/* Architecture specifics. */
#define portSTACK_GROWTH        ( -1 )
#define portTICK_RATE_MS  ((portTickType) 1000 / configTICK_RATE_HZ)
#define portBYTE_ALIGNMENT      8
#define portNOP()               __asm volatile ( "NOP" );

extern void portRESTORE_CONTEXT_FIRST(void) __attribute__ ((naked));

/*-----------------------------------------------------------*/
#define portYIELD_FROM_ISR()    vTaskSwitchContext()
extern void vPortYIELD(void)    __attribute__ ((naked));
#define portYIELD()             vPortYIELD()
/*-----------------------------------------------------------*/

/* Critical section management. */
#ifdef THUMB_INTERWORK

  extern void vPortDisableInterruptsFromThumb(void) __attribute__ ((naked));
  extern void vPortEnableInterruptsFromThumb(void) __attribute__ ((naked));

  #define portDISABLE_INTERRUPTS()      PortDisableInterruptsFromThumb()
  #define portENABLE_INTERRUPTS()       PortEnableInterruptsFromThumb()

#else
  #define portDISABLE_INTERRUPTS()                                        \
    __asm volatile (                                                      \
      "STMDB SP!, {R0}   \n\t"      /* Push R0. */                        \
      "MRS R0, CPSR      \n\t"      /* Get CPSR. */                       \
      "ORR R0, R0, #0xC0 \n\t"      /* Disable IRQ, FIQ.*/                \
      "MSR CPSR, R0      \n\t"      /* Write back modified value.*/       \
      "LDMIA SP!, {R0}   " )        /* Pop R0.*/
  #define portENABLE_INTERRUPTS()                                         \
    __asm volatile (                                                      \
```

```
        "STMDB SP!, {R0}     \n\t"     /* Push R0. */                    \
        "MRS R0, CPSR        \n\t"     /* Get CPSR. */                   \
        "BIC R0, R0, #0xC0   \n\t"     /* Enable IRQ, FIQ. */            \
        "MSR CPSR, R0        \n\t"     /* Write back modified value.*/   \
        "LDMIA SP!, {R0}     " )       /* Pop R0. */

#endif /* THUMB_INTERWORK */

extern void vPortEnterCritical( void );
extern void vPortExitCritical( void );

#define portENTER_CRITICAL()        vPortEnterCritical();
#define portEXIT_CRITICAL()         vPortExitCritical();
/*-----------------------------------------------------------*/

/* Task function macros as described on the FreeRTOS.org WEB site. */
#define portTASK_FUNCTION_PROTO( vFunction, pvParameters ) void vFunction(
void *pvParameters )
#define portTASK_FUNCTION( vFunction, pvParameters ) void vFunction( void
*pvParameters )

#ifdef __cplusplus
}
#endif

#endif /* PORTMACRO_H */
```

표 2.5 portmacro.h

FreeRTOS는 사용자가 사용하려는 시스템의 특성에 맞게 매개 변수를 통하여 FreeRTOS의 커널을 설정할 수 있는 환경을 제공하고 있다. 환경 설정을 위한 매개 변수는 FreeRTOSConfig.h에서 확인할 수 있으며, 본 논문에서는 표 2.6과 같이 구성하였다.

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

#define configUSE_WATCHDOG_TICK         1
#define configUSE_PREEMPTION            1
#define configUSE_IDLE_HOOK             0
#define configUSE_TICK_HOOK             0
#define configCPU_CLOCK_HZ              ((unsigned long) 66000000)
#define configCPU_PERIPH_HZ             ((unsigned long) 48000000)
#define configTICK_RATE_HZ              ((portTickType) 1000)
#define configMAX_PRIORITIES            ((unsigned portBASE_TYPE) 255)
#define configMINIMAL_STACK_SIZE        ((unsigned short) 1024)
#define configTOTAL_HEAP_SIZE           ((size_t) 131072)
#define configMAX_TASK_NAME_LEN         32
#define configUSE_TRACE_FACILITY        1
#define configUSE_16_BIT_TICKS          0
#define configIDLE_SHOULD_YIELD         1
#define configUSE_MUTEXES               1

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES           0
#define configMAX_CO_ROUTINE_PRIORITIES 2
#define THUMB_INTERWORK                 1

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */
#define INCLUDE_vTaskPrioritySet            1
#define INCLUDE_uxTaskPriorityGet           1
#define INCLUDE_vTaskDelete                 1
#define INCLUDE_vTaskCleanUpResources       0
#define INCLUDE_vTaskSuspend                1
#define INCLUDE_vTaskDelayUntil             1
#define INCLUDE_vTaskDelay                  1
#define INCLUDE_xTaskGetCurrentTaskHandle   1
```

```
#endif /* FREERTOS_CONFIG_H */
```

표 2.6 FreeRTOSConfig.h

4. 문맥 교환(Context Switching)

문맥 교환(Context Switching)이란 하나의 프로세스가 CPU를 사용 중인 상태에서 다른 프로세스가 CPU를 사용하도록 하기 위해, 이전의 프로세스의 상태(문맥)를 보관하고 새로운 프로세스의 상태를 적재하는 작업을 말하며, 문맥(Context)이란 프로세스가 중지되었다가 다시 실행될 때 복원되어야 하는 프로세스의 실행 정보를 의미한다. 한 프로세스의 문맥은 그 프로세스의 프로세스 컨트롤 블록(PCB, Process Control Block)에 기록되어 있으며, 본 논문에서 사용하고 있는 실시간 운영체제에서는 태스크 컨트롤 블록(TCB, Task Control Block)에 기록되어 있다. 커널 수준에서의 프로세스 문맥은 범용 레지스터, 프로그램 카운터(PC), 스택 포인터(SP), 프로세스의 상태 레지스터 등의 내용을 포함한다.

닌텐도 DS-lite에서 FreeRTOS를 사용하기 위해서는 닌텐도 DS-lite의 ARM 프로세서에 맞게 문맥을 구성하여 태스크의 전환이 이루어지도록 해야 한다. 본 논문에서는 그림 2.10과 같이 문맥을 구성하였다. 문맥은 ARM 프로세서의 레지스터(R0 – R15, CPSR)와 FreeRTOS에서 중첩 인터럽트 모델을 지원하기 위한 변수(ulCriticalNesting)로 이루어져 있다.

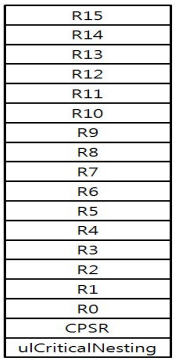


그림 2.10 FreeRTOS 문맥 구성

본 논문에서 사용하는 실시간 운영체제인 FreeRTOS는 새로운 태스크를 생성할 때마다 태스크 컨트롤 블록(TCB)이라는 자료구조를 생성해서 태스크를 관리한다. 각 태스크는 태스크별로 고유의 스택을 가지고 있으며, 태스크 컨트롤 블록을 통해 참조되어질 수 있다. 태스크가 생성 될 때에는 생성된 태스크가 처음으로 태스크 함수로 진입하여 실행될 수 있도록 하는 문맥 정보를 스택에 저장해 주어야 한다. FreeRTOS에서 태스크의 생성은 xTaskCreate() 함수의 호출을 통해 이루어지며, xTaskCreate() 함수 내부에서 태스크 스택 초기화 함수인 pxPortInitialiseStack() 함수의 호출을 통해 태스크의 스택을 초기화 하도록 되어 있다. 표 2.7은 닌텐도 DS-lite에서 FreeRTOS를 사용하기 위해 구현한 태스크 스택 초기화 함수의 코드를 나타낸다.

```
portSTACK_TYPE *pxPortInitialiseStack( portSTACK_TYPE *pxTopOfStack,
pdTASK_CODE pxCode, void *pvParameters )
{
  portSTACK_TYPE *pxOriginalTOS;
  pxOriginalTOS = pxTopOfStack;

  *pxTopOfStack = ( portSTACK_TYPE ) pxCode + portINSTRUCTION_SIZE;
  pxTopOfStack--;
  /* R14 */
  *pxTopOfStack = ( portSTACK_TYPE ) pxCode + portINSTRUCTION_SIZE;
  pxTopOfStack--;
  /* Stack used when task starts goes in R13. */
  *pxTopOfStack = ( portSTACK_TYPE ) pxOriginalTOS;
  pxTopOfStack--;
  *pxTopOfStack = ( portSTACK_TYPE ) 0x12121212;    /* R12 */
  pxTopOfStack--;
```

```
    *pxTopOfStack = ( portSTACK_TYPE ) 0x11111111;    /* R11 */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) 0x10101010;    /* R10 */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) 0x09090909;    /* R9 */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) 0x08080808;    /* R8 */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) 0x07070707;    /* R7 */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) 0x06060606;    /* R6 */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) 0x05050505;    /* R5 */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) 0x04040404;    /* R4 */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) 0x03030303;    /* R3 */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) 0x02020202;    /* R2 */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) 0x01010101;    /* R1 */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) pvParameters; /* R0 */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) portINITIAL_SPSR;
    pxTopOfStack--;
    *pxTopOfStack = portNO_CRITICAL_SECTION_NESTING;
    return pxTopOfStack;
}
```

표 2.7 태스크 스택 초기화 함수

표 2.8은 본 논문에서 구성한 port_asm.s 소스 파일을 나타낸다. port_asm.s 소스 파일은 문맥 교환을 위해 필요한 함수들을 구현한 코드로서, 인터럽트의 활성화/비활성화, 틱 인터럽트 또는 함수의 호출을 통해 문맥 교환이 이루어지기 위해 문맥을 저장, 복원하는 내용을 포함하고 있다.

```
    .section ".text"
    .align 4
    .arm

    .global vPortDisableInterruptsFromThumb
    .global vPortEnableInterruptsFromThumb
    .global portRESTORE_CONTEXT_FIRST
    .global portRESTORE_CONTEXT
    .global vPreemptiveTick
    .global vPortYIELD

vPortDisableInterruptsFromThumb:
    STMDB   SP!, {R0}
    MRS     R0, CPSR
    ORR     R0, R0, #0xC0
    MSR     CPSR, R0
    LDMIA   SP!, {R0}
    BX      R14

vPortEnableInterruptsFromThumb:
    STMDB   SP!, {R0}
    MRS     R0, CPSR
    BIC     R0, R0, #0xC0
    MSR     CPSR, R0
    LDMIA   SP!, {R0}
    BX      R14
```

```
    .extern  pxCurrentTCB
    .extern  ulCriticalNesting
portRESTORE_CONTEXT_FIRST:
    MOV      R3, #0x4000000
    STR      R3, [R3, #0x208]
    MRS      R3, CPSR
    BIC      R3, R3, #0xDF
    ORR      R3, R3, #0x92
    MSR      CPSR, R3

portRESTORE_CONTEXT:
    LDR      R0, =pxCurrentTCB
    LDR      R0, [R0]
    LDR      R0, [R0]

    LDR      R1, =ulCriticalNesting
    LDMFD    R0!, {R2}
    STR      R2, [R1]

    LDMFD    R0!, {R2}
    MSR      SPSR, R2

    LDR      R1, [R0, #60]
    SUB      LR, R1, #4

    MOV      R3, #0x4000001
    STR      R3, [R3, #0x207]

    LDMFD    R0, {R0-R14}^

    MOVS     PC, LR

    .extern  vTaskIncrementTick
    .extern  vTaskSwitchContext
```

```
vPreemptiveTick:
    MOV     R3, #0x4000000
    STR     R3, [R3, #0x208]

    POP     {R2, LR}

    SUB     R0, SP, #72
    ADD     R3, R0, #24
    STMIA   R3, {R4-R14}^

    MSR     CPSR, R2
    LDMFD   SP!, {R5-R6, R7, R8}
    STR     R5, [R0, #4]

    LDMFD   SP!, {R5-R8, R12, LR}

    ADD     R3, R0, #8
    STMIA   R3, { R5-R8 }
    STR     R12, [R0, #56]
    STR     LR, [R0, #68]

    LDR     R5, =ulCriticalNesting
    LDR     R5, [R5]
    STR     R5, [R0]

    LDR     R5, =pxCurrentTCB
    LDR     R5, [R5]
    STR     R0, [R5]

    BL      vTaskIncrementTick
    BL      vTaskSwitchContext
    B       portRESTORE_CONTEXT

vPortYIELD:
```

```
    MOV     R0, #0x4000000
    STR     R0, [R0, #0x208]

    SUB     R0, SP, #60
    STMIA   R0, { R1-R14 }

    ADR     R1, exit + 4
    STR     R1, [SP, #-4]

    MRS     R0, CPSR
    STR     R0, [SP, #-68]

    SUB     R0, SP, #72
    LDR     R1, =ulCriticalNesting
    LDR     R1, [R1]
    STR     R1, [R0]

    LDR     R1, =pxCurrentTCB
    LDR     R1, [R1]
    STR     R0, [R1]

    MRS     R0, CPSR
    BIC     R0, R0, #0xDF
    ORR     R0, R0, #0x92
    MSR     CPSR, R0

    BL      vTaskSwitchContext
    B       portRESTORE_CONTEXT
exit:
    BX      LR
```

표 2.8 port_asm.s

5. 타이머

실시간 운영체제에서는 내부적으로 시간 관리를 하기 위하여 일정한 주기의 타이머를 사용한다. FreeRTOS에서는 하드웨어의 타이머 장치를 이용하여 타이머 인터럽트(틱, Tick)를 발생시키고, 타이머 인터럽트가 발생할 때마다 시간을 나타내는 변수인 Tick Count의 값을 증가시키도록 구현되어 있다. 스케줄러는 이 Tick Count 값을 이용하여 각 태스크가 중지된 시점, 실행되어야 하는 시점을 파악할 수 있다. 이를 통해 시간 지연, 소프트웨어 타이머의 기능을 제공함으로써 태스크의 주기성을 보장해 줄 수 있다. 또한, 현재 실행중인 태스크보다 더 높은 우선순위를 가진 태스크가 있는 경우 타이머 인터럽트가 발생할 때마다 문맥 전환이 이루어지도록 함으로써 우선순위 기반의 선점형 스케줄링이 가능하도록 한다.

본 논문에서는 닌텐도 DS-lite의 하드웨어 장치를 제어하기 위하여 devkitPro의 libnds 라이브러리를 사용한다. libnds 라이브러리는 표 2.9와 같은 타이머 정보를 포함하고 있다.

```
#define TIMER_CR(n)          (*(vu16*)(0x04000102+((n)<<2)))
#define TIMER0_CR            (*(vu16*)0x04000102)
#define TIMER1_CR            (*(vu16*)0x04000106)
#define TIMER2_CR            (*(vu16*)0x0400010A)
#define TIMER3_CR            (*(vu16*)0x0400010E)

#define TIMER_DATA(n)        (*(vu16*)(0x04000100+((n)<<2)))
#define TIMER0_DATA          (*(vu16*)0x04000100)
#define TIMER1_DATA          (*(vu16*)0x04000104)
#define TIMER2_DATA          (*(vu16*)0x04000108)
#define TIMER3_DATA          (*(vu16*)0x0400010C)

#define BUS_CLOCK            (33513982)
#define TIMER_ENABLE         (1<<7)
#define TIMER_IRQ_REQ        (1<<6)
```

```
#define TIMER_CASCADE           (1<<2)
typedef enum {
    ClockDivider_1 = 0,     // divides the timer clock by 1 (~33513.982 kHz)
    ClockDivider_64 = 1,    // divides the timer clock by 64 (~523.657 kHz)
    ClockDivider_256 = 2,   // divides the timer clock by 256 (~130.914 kHz)
    ClockDivider_1024 = 3   // divides the timer clock by 1024 (~32.7284 kHz)
}ClockDivider;

#define TIMER_DIV_1     (0)
#define TIMER_DIV_64    (1)
#define TIMER_DIV_256   (2)
#define TIMER_DIV_1024  (3)

#define TIMER_FREQ(n)           (-BUS_CLOCK/(n))
#define TIMER_FREQ_64(n)        (-(BUS_CLOCK>>6)/(n))
#define TIMER_FREQ_256(n)       (-(BUS_CLOCK>>8)/(n))
#define TIMER_FREQ_1024(n)      (-(BUS_CLOCK>>10)/(n))
```

표 2.9 닌텐도 DS-lite 타이머 장치 정보

표 2.10은 FreeRTOS에 닌텐도 DS-lite의 타이머를 이용하여 타이머 인터럽트를 등록하는 코드를 나타낸다. 본 논문에서는 표 2.8의 정보를 바탕으로 1 밀리 초(MS)의 틱 발생 주기를 갖도록 타이머를 설정하여 타이머 인터럽트를 호출하도록 구현하였다.

```
static void prvSetupTimerInterrupt( void )
{
    vPortDisableInterruptsFromThumb();

    TIMER0_DATA = TIMER_FREQ_256(1000);
    TIMER0_CR = TIMER_ENABLE | TIMER_IRQ_REQ | TIMER_DIV_256;
    irqSet(IRQ_TIMER0, vPreemptiveTick);
    irqEnable(IRQ_TIMER0);
}
```

표 2.10 타이머 장치 초기화 함수

6. 메모리 관리

일반적으로 동적으로 메모리를 할당하고 해제하기 위해서는 표준 C 라이브러리인 malloc() 함수와 free() 함수를 사용한다. 하지만, 임베디드 / 실시간 시스템에서는 메모리 할당과 해제에 따른 메모리 단편화 현상 때문에 연속적인 단일 메모리 공간을 확보하지 못할 수도 있다는 점과 malloc(), free() 함수가 연속된 메모리를 할당하기 위해 사용하는 복잡한 알고리즘 때문에 수행 시간이 예측 불가능하다는 점, 멀티태스킹 환경에서 thread-safe 하지 않다는 점 등에서 이 함수들을 사용하는 것이 바람직하지 않을 수도 있다.

본 논문에서 사용한 실시간 운영체제인 FreeRTOS에서는 기본적으로 4가지 방식의 동적 메모리 할당 구현 샘플을 제공하고 있으며, 위에서 언급한 문제들 때문에 메모리를 관리하는 API를 Portable 계층에 위치시켜 FreeRTOS를 사용하는 사용자가 개발하려는 시스템에 적절한 방식의 메모리 관리 모델을 적용하도록 하고 있다. FreeRTOS에서 제공하는 메모리 관리 모델에 대한 설명은 표 2.11과 같으며, 본 논문에서는 heap_3.c 소스 파일로 제공하는 동적 메모리 할당 구현을 사용하여 메모리 관리를 하고 있다.

구 현	설 명
heap_1.c	가장 간단한 메모리 관리 모델 구현 샘플
heap_2.c	best fit 알고리즘을 이용한 메모리 관리 모델 구현 샘플
heap_3.c	표준 C 라이브러리인 malloc(), free() 함수를 이용한 메모리 관리 모델 구현 샘플
heap_4.c	first fit 알고리즘을 이용한 메모리 관리 모델 구현 샘플

표 2.11 FreeRTOS 메모리 관리 모델 구현 샘플

제 5 절 FreeRTOS 성능 측정

1. 성능 측정 방법

본 논문에서는 실시간 운영체제인 FreeRTOS를 닌텐도 DS-lite에 이식하여 실시간 운영체제 환경을 제공하고 있다. FreeRTOS를 이용한 실시간 운영체제 환경에서 작성한 응용 프로그램이 교육 실습을 진행하기위해 무리가 없는지 검증하기 위하여 FreeRTOS의 멀티태스킹 환경에서 성능에 영향을 미치는 요인, 우선순위 기반의 선점형 스케줄링이 제대로 이루어지는지 확인하였다.

성능을 측정하기 위한 방법으로는 닌텐도 DS-lite의 타이머를 이용하여 시간을 측정하기 위한 기능을 표 2.12와 같이 구현하고, 성능에 영향을 미치는 요인을 파악할 수 있는 시나리오를 작성하여 문맥 교환이 이루어지는 시점에 각 태스크를 식별할 수 있는 태스크의 번호, 태스크 전환이 이루어지는 상황, 타이머 레지스터의 값을 기록한 뒤, 기록된 데이터를 분석하여 응용 프로그램을 설계 하였을 때 예상할 수 있는 결과와 비교하였다.

```
#define TIMER0_DATA     (*(vuint16*)0x04000100)
#define TIMER1_DATA     (*(vuint16*)0x04000104)
#define TIMER2_DATA     (*(vuint16*)0x04000108)
#define TIMER3_DATA     (*(vuint16*)0x0400010C)

#define TIMER0_CR       (*(vuint16*)0x04000102)
#define TIMER1_CR       (*(vuint16*)0x04000106)
#define TIMER2_CR       (*(vuint16*)0x0400010A)
#define TIMER3_CR       (*(vuint16*)0x0400010E)

#define TIMER_DIV_1     (0)
#define TIMER_DIV_64    (1)
#define TIMER_DIV_256   (2)
```

```
#define TIMER_DIV_1024    (3)

#define TIMER_ENABLE    (1<<7)
#define TIMER_IRQ_REQ    (1<<6)
#define TIMER_CASCADE  (1<<2)

#define SWITCHED_IN       0
#define SWITCHED_OUT    1

#define MAX_LOGGING_NUM          1024

struct TASK_INFO
{
    int taskID;
    int taskState;
    int timerValue;
};

struct TASK_INFO taskInfo[MAX_LOGGING_NUM];
int taskInfoIndex = 0;

int elapsedTimeIndex = 0;
int elapsedTime[MAX_LOGGING_NUM] = {0, };

#define initTimer()                                                    \
{                                                                      \
    TIMER1_CR = 0;                                                     \
    TIMER2_CR = 0;                                                     \
                                                                       \
    TIMER1_DATA = 0;                                                   \
    TIMER2_DATA = 0;                                                   \
                                                                       \
    TIMER1_CR = TIMER_ENABLE;                                          \
    TIMER2_CR = TIMER_ENABLE | TIMER_CASCADE;                          \
```

```
}

#define updateTimer()                                                    \
{                                                                        \
    TIMER1_CR = 0;                                                       \
    TIMER2_CR = 0;                                                       \
}

#define getElapsedTime() ( TIMER1_DATA | (TIMER2_DATA<<16) )

void logSwitchedIn()
{
    if(taskInfoIndex < MAX_LOGGING_NUM)
    {
        taskInfo[taskInfoIndex].taskID = pxCurrentTCB->uxTCBNumber;
        taskInfo[taskInfoIndex].taskState = SWITCHED_IN;
        taskInfo[taskInfoIndex].timerValue = getElapsedTime();
        taskInfoIndex++;
    }
}

void logSwitchedOut()
{
    if(taskInfoIndex < MAX_LOGGING_NUM)
    {
        taskInfo[taskInfoIndex].taskID = pxCurrentTCB->uxTCBNumber;
        taskInfo[taskInfoIndex].taskState = SWITCHED_OUT;
        taskInfo[taskInfoIndex].timerValue = getElapsedTime();
        taskInfoIndex++;
    }
}
```

표 2.12 시간 측정을 위한 함수

문맥 교환이 이루어지는 시점에 성능 측정을 위한 데이터를 기록하기 위한 방법으로는 표 2.12에서 구현한 logSwitchedIn(), logSwitchedOut()

함수를 표 2.13과 같이 FreeRTOS에서 성능 측정 또는 디버깅을 위한 용도로 제공하는 매크로 함수를 이용하여 호출하도록 하였다.

```
#ifndef traceTASK_SWITCHED_IN
    /* Called after a task has been selected to run.  pxCurrentTCB holds a pointer
       to the task control block of the selected task. */
    #define traceTASK_SWITCHED_IN()        logSwitchedIn();
#endif

#ifndef traceTASK_SWITCHED_OUT
    /* Called before a task has been selected to run.  pxCurrentTCB holds a
       pointer to the task control block of the task being switched out. */
    #define traceTASK_SWITCHED_OUT()    logSwitchedOut();
#endif
```

표 2.13 성능 측정 데이터를 기록하기 위한 매크로 함수

2. 성능 측정 실험 및 결과

FreeRTOS 성능 측정 실험은 위에서 언급한 바와 같이, 성능을 측정하기 위한 방법 및 함수를 구현하고 ARM 프로세서를 사용하는 닌텐도 DS와 TI사의 DSP 프로세서인 TMS320F28335를 사용하는 임베디드 보드에서 실험을 진행하였다. 분석에 사용된 데이터는 TMS320F28335에서 측정된 결과를 사용하였다.

1) FreeRTOS 성능에 영향을 미치는 요인

논문에서 사용된 실시간 운영체제인 FreeRTOS의 성능에 영향을 미치는 요인을 파악하기 위한 실험은 표 2.14의 시나리오와 같이 응용 프로그램을 구현한 뒤, 각 시나리오에서 얻어진 데이터를 비교 분석하여 FreeRTOS의 성능에 영향을 미치는 요인을 파악하였다. 문맥 교환 시간은 표 2.12에서 구현한 logSwitchedIn(), logSwitchedOut() 함수를 문맥 교환

을 수행하는 함수 내부에서 문맥 저장, 문맥 복원을 수행하기 직전에 호출하도록 함으로써 틱 인터럽트를 통해 문맥 교환이 이루어지는데 소요되는 사이클 수를 측정하였다.

시나리오1			시나리오2			시나리오3		
우선순위	태스크	주기	우선순위	태스크	주기	우선순위	태스크	주기
5	Task1	1000us	5	Task1	1000us	5	Task1	300us
5	Task2	1000us	4	Task2	1000us	4	Task2	1000us
5	Task3	1000us	3	Task3	1000us	3	Task3	1000us
5	Task4	1000us	2	Task4	1000us	2	Task4	2000us
5	Task5	1000us	2	Task5	1000us	2	Task5	2000us
5	Task6	1000us	2	Task6	1000us	2	Task6	2000us

표 2.14 FreeRTOS 성능 측정 시나리오

표 2.15는 FreeRTOS의 성능에 영향을 미치는 요인을 파악하기 위한 실험의 시나리오에 대한 설명을 나타내며, 실험을 통해 얻어진 데이터는 표 2.16에서 표 2.23까지의 내용과 같다. 각 표에서 Task 항목의 번호는 표 2.14의 각 시나리오 태스크 번호와 같으며, 0으로 표시된 태스크는 IDLE 태스크를 나타낸다. State 항목은 태스크가 전환되는 상황에 따른 분류를 하기 위해 표시한 값이며, Time, Elapsed Time 항목은 타이머 레지스터의 값, 스케줄러가 시작된 이후부터 누적된 타이머 레지스터의 값을 나타낸다. C/S 항목은 문맥 교환을 위해 소요되는 시간(타이머의 값)을 나타낸다.

시나리오	설 명
시나리오 1	같은 우선순위, 같은 주기를 갖는 멀티태스킹 환경 실험
시나리오 2	다른 우선순위, 같은 주기를 갖는 멀티태스킹 환경 실험
시나리오 3	다른 우선순위, 다른 주기를 갖는 멀티태스킹 환경 실험

표 2.15 성능 측정 시나리오 설명

Task	State	Time	InterruptCount	ElapsedTime	C/S	Reason
0	0	14734	11	165266	223	T
0	1	14958	12	180042		
0	0	14735	12	180265	223	T
0	1	14956	13	195044		
0	0	14733	13	195267	223	T
0	1	14957	14	210043		
0	0	14734	14	210266	223	T
0	1	14957	15	225043		
0	0	14734	15	225266	223	T
0	1	14958	16	240042		
0	0	14735	16	240265	223	T
0	1	14953	17	255047		
0	0	14730	17	255270	223	T
0	1	14958	18	270042		
0	0	14735	18	270265	223	T
0	1	14958	19	285042		
0	0	14735	19	285265	223	T

표 2.16 시나리오1 실험 결과(실행될 수 있는 태스크가 없는 경우)

Task	State	Time	InterruptCount	ElapsedTime	C/S	Reason
0	0	14734	19	285266	223	T
0	1	14958	20	300042		
4	0	13505	20	301495	1453	T
4	3	13226	20	301774		
6	2	13091	20	301909	135	P
6	3	12626	20	302374		
1	2	12491	20	302509	135	P
1	3	12001	20	302999		
2	2	11866	20	303134	135	P
2	3	11351	20	303649		

3	2	11216	20	303784	135	P
3	3	10676	20	304324		
5	2	10541	20	304459	135	P
5	3	9976	20	305024		
4	2	9827	20	305173	149	P
4	3	9272	20	305728		
0	2	9053	20	305947	219	P

표 2.17 시나리오1 실험 결과(실행될 수 있는 태스크가 6개 있는 경우)

Task	State	Time	InterruptCount	ElapsedTime	C/S	Reason
0	0	14734	11	165266	223	T
0	1	14958	12	180042		
0	0	14735	12	180265	223	T
0	1	14956	13	195044		
0	0	14733	13	195267	223	T
0	1	14957	14	210043		
0	0	14734	14	210266	223	T
0	1	14957	15	225043		
0	0	14734	15	225266	223	T
0	1	14958	16	240042		
0	0	14735	16	240265	223	T
0	1	14953	17	255047		
0	0	14730	17	255270	223	T
0	1	14958	18	270042		
0	0	14735	18	270265	223	T
0	1	14958	19	285042		
0	0	14735	19	285265	223	T

표 2.18 시나리오2 실험 결과(실행될 수 있는 태스크가 없는 경우)

Task	State	Time	InterruptCount	ElapsedTime	C/S	Reason
0	0	14735	19	285265	223	T

0	1	14958	20	300042		
1	0	13502	20	301498	1456	T
1	3	13037	20	301963		
2	2	12874	20	302126	163	P
2	3	12384	20	302616		
3	2	12221	20	302779	163	P
3	3	11706	20	303294		
4	2	11543	20	303457	163	P
4	3	11264	20	303736		
4	2	11115	20	303885	149	P
4	3	10610	20	304390		
5	2	10447	20	304553	163	P
5	3	9907	20	305093		
6	2	9772	20	305228	135	P
6	3	9207	20	305793		
0	2	9044	20	305956	163	P

표 2.19 시나리오2 실험 결과(실행될 수 있는 태스크가 6개 있는 경우)

Task	State	Time	InterruptCount	ElapsedTime	C/S	Reason
0	2	13839	6	91161	219	P
0	1	14952	7	105048		
0	0	14729	7	105271	223	T
0	1	14957	8	120043		
0	0	14734	8	120266	223	T
0	1	14958	9	135042		

표 2.20 시나리오3 실험 결과(실행될 수 있는 태스크가 없는 경우)

Task	State	Time	InterruptCount	ElapsedTime	C/S	Reason
0	0	14735	19	285265	223	T
0	1	14952	20	300048		
2	0	13683	20	301317	1269	T

2	3	13193	20	301807		
3	2	13030	20	301970	163	P
3	3	12515	20	302485		
4	2	12352	20	302648	163	P
4	3	12073	20	302927		
5	2	11938	20	303062	135	P
5	3	11398	20	303602		
6	2	11263	20	303737	135	P
6	3	10698	20	304302		

표 2.21 시나리오3 실험 결과(실행될 수 있는 태스크가 5개 있는 경우)

Task	State	Time	InterruptCount	ElapsedTime	C/S	Reason
0	0	14735	38	570265	223	T
0	1	14958	39	585042		
1	0	14524	39	585476	434	T
1	3	13934	39	586066		
0	2	13715	39	586285	219	P
0	1	14953	40	600047		

표 2.22 시나리오3 실험 결과(실행될 수 있는 태스크가 1개 있는 경우)

Task	State	Time	InterruptCount	ElapsedTime	C/S	Reason
0	0	14735	49	735265	223	T
0	1	14954	50	750046		
2	0	14312	50	750688	642	T
2	3	13747	50	751253		
3	2	13584	50	751416	163	P
3	3	12994	50	752006		
0	2	12803	50	752197	191	P
0	1	14958	51	765042		

표 2.23 시나리오3 실험 결과(실행될 수 있는 태스크가 2개 있는 경우)

실험을 통해 얻어진 데이터를 바탕으로 분석해 본 결과, 생성되는 태스

크의 총 개수와 우선순위는 문맥 교환에 따른 오버헤드에 영향을 미치지 않고, 스케줄링 될 때 실행할 수 있는 태스크의 개수가 많을수록 문맥 교환에 따른 오버헤드가 증가하는 것을 파악할 수 있다. 이는, FreeRTOS가 태스크를 관리하기 위해 리스트 자료구조를 사용하고 있기 때문에 생기는 현상이며, 태스크의 반복 주기가 짧을수록 문맥 교환에 따른 오버헤드가 증가하는 것을 확인할 수 있다.

2) 우선순위 기반의 선점형 스케줄링

논문에서 사용된 실시간 운영체제인 FreeRTOS는 태스크 컨트롤을 위해 vTaskDelay(), vTaskDelayUntil() 함수를 제공하고 있다. 우선순위 기반의 선점형 스케줄링 측정 실험은 교육 실습을 위한 수준에서 응용 프로그램을 작성하였을 때 우선순위 기반의 선점형 스케줄링이 제대로 이루어지는지 확인하기 위하여 표 2.24의 시나리오와 같이 응용 프로그램을 구현한 뒤, 각 시나리오에서 얻어진 데이터를 응용 프로그램을 설계하였을 때 예상할 수 있는 결과와 비교하였다.

		틱 인터럽트 주기	100 us	
태스크	우선순위	주기	실행시간	비율
Task1	5	1000 us	200 us	20
Task2	4	1500 us	300 us	20
Task3	3	3000 us	600 us	20
IDLE	0			40

표 2.24 우선순위 기반의 선점형 스케줄링 확인을 위한 시나리오

그림 2.11, 그림 2.13은 표 2.24의 내용을 바탕으로 vTaskDelay(), vTaskDelayUntil() 함수를 이용하여 예상할 수 있는 스케줄링 결과를 나타내며, 그림 2.12, 그림 2.14는 실험을 통해 얻어진 데이터를 이용하여 그래프 화 한 그림을 나타낸다.

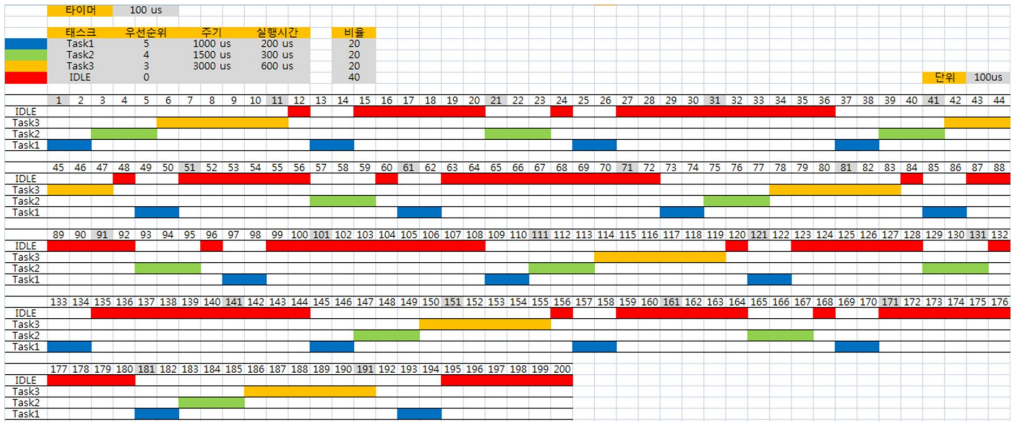


그림 2.11 vTaskDelay() 함수를 이용한 실험 예상 결과

Task1	0
Task2	1
Task3	2
IDLE	3

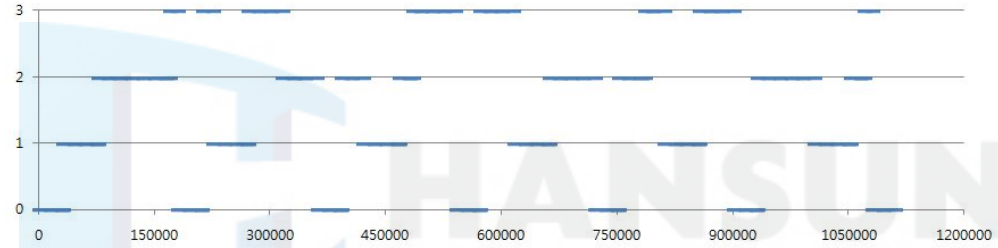


그림 2.12 vTaskDelay() 함수를 이용한 실험 결과

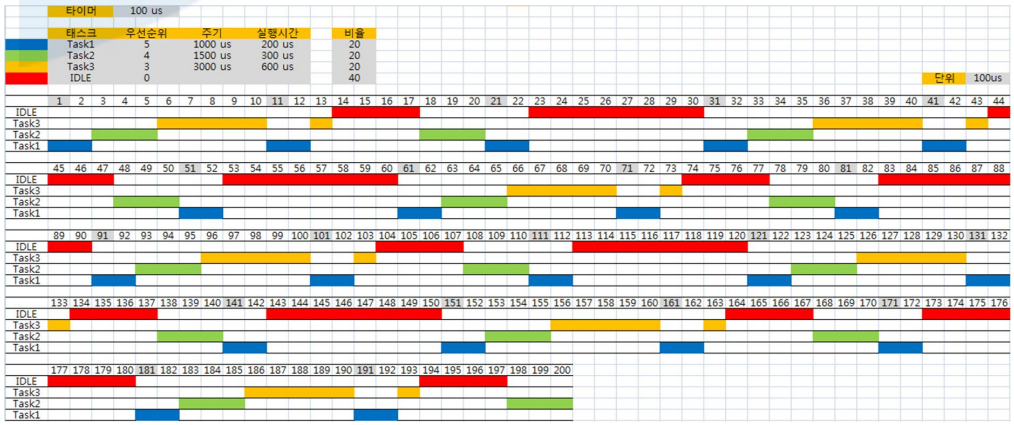


그림 2.13 vTaskDelayUntil() 함수를 이용한 실험 예상 결과

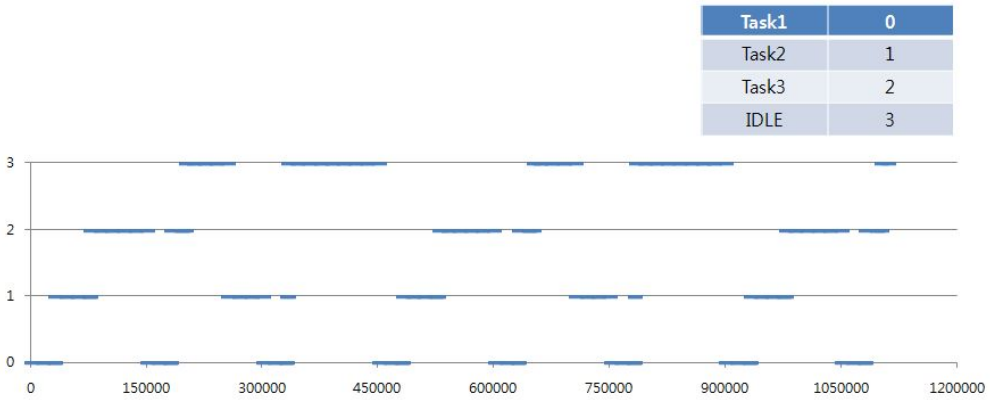


그림 2.14 vTaskDelayUntil() 함수를 이용한 실험 결과

실험을 통해 얻어진 데이터를 바탕으로 비교, 분석해 본 결과, 교육 실습을 위한 수준의 응용 프로그램을 작성하였을 때 사용자가 설계 시 예측할 수 있는 수준의 스케줄링이 이루어질 수 있다는 점을 확인할 수 있다.

제 6 절 원격 디버깅

1. 원격 디버깅 개요

원격 디버깅이란 그림 2.15와 같이 호스트 머신에서 동작하는 디버거와 타겟 머신에서 동작하는 디버깅 대상이 되는 소프트웨어가 이더넷(Ethernet) 또는 시리얼(Serial) 등을 통해 원격으로 이루어지는 디버깅 방법을 말한다.

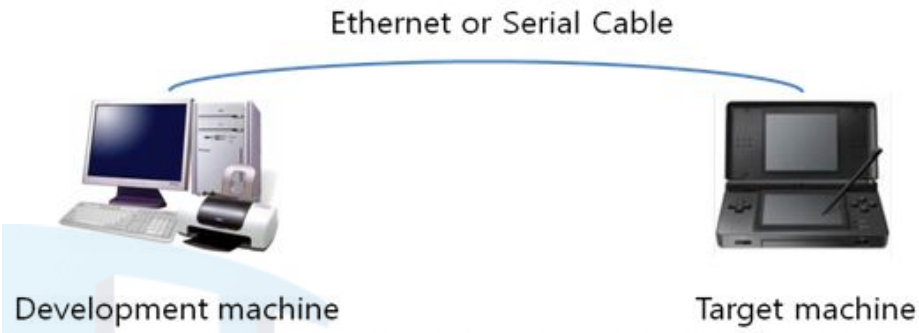


그림 2.15 원격 디버깅

원격 디버깅이 이루어지기 위해 호스트 머신의 디버거는 타겟 머신에서 동작하는 소프트웨어와 통신하기 위한 방법을 알아야 하고, 타겟 머신에서 동작하는 소프트웨어는 호스트 머신에서 동작하는 디버거의 요청 또는 명령에 대한 처리를 해주어야 할 필요가 있다.

본 논문에서는 디버거로 공개 소스 소프트웨어인 GDB를 사용하였다. GDB는 원격 디버깅을 위한 방법으로 리모트 시리얼 프로토콜(Remote Serial Protocol)이 정의되어 있고, 원격 디버깅을 하기 위한 방법을 모두 알고 있으므로 약간의 사용법만 익힌다면 일반적인 컴퓨터 시스템에서 디버깅을 하는 것과 같이 원격 디버깅을 할 수 있다. 반면, 타겟 머신의 소프트웨어에서는 GDB의 요청에 대한 처리를 해주기 위한 모듈이 필요한데 이렇게 GDB의 요청에 대한 처리를 해주기 위해 타겟 시스템에서 돌아가

는 코드를 스텁(Stub)이라 한다. 그림 2.16은 원격 디버깅 모듈의 소프트웨어 구조로 타겟, 호스트 머신 간 원격 디버깅이 이루어지는 모습을 나타낸다.

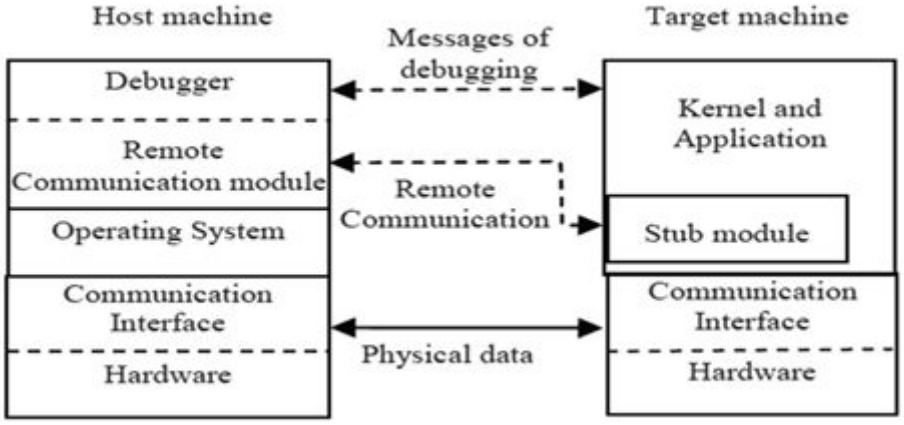


그림 2.16 원격 디버깅 모듈 소프트웨어 구조

본 논문에서는 GDB의 요청을 처리하는 디버거 스텁 모듈을 작성하여 실시간 운영체제(FreeRTOS)에 포함시킴으로써 FreeRTOS상에서 원격 디버깅이 가능하도록 구현하였다.

2. ARM 프로세서 예외처리 메커니즘

원격 디버깅을 처리하기 위해서는 타겟 머신에서 실행되는 소프트웨어에서 원하는 시점에 GDB 스텁 모듈로 진입할 수 있는 방법이 필요하다. 또한 GDB 스텁 모듈로 진입했던 시점으로 복귀하여 프로그램을 재개 할 수 있는 방법이 필요하다. 다음은 ARM 프로세서의 예외처리 메커니즘을 이용하여 GDB 스텁 모듈로 진입하고 복귀하는 방법에 대해 설명한다.

예외처리(Exception)란 외부의 요청이나 오류에 의해 정상적으로 진행되는 프로그램의 동작을 멈추고 프로세서의 동작모드를 변환하여 미리 정해진 프로그램으로 외부의 요청이나 오류를 처리하는 것을 의미한다.

ARM 프로세서의 예외처리 종류에는 Reset, Undefined Instruction,

Software Interrupt, Prefetch Abort, Data Abort, IRQ, FIQ 총 7개가 있고, 각 예외처리가 발생하면 프로세서는 해당하는 동작모드로 전환된다. 표 2.25는 ARM 프로세서의 예외처리들과 각각의 발생 시기를 나타낸다.

예외처리	발생 시기
Reset	시스템에 전원 인가 시 또는 리셋 신호 입력되면 발생
FIQ (Fast Interrupt Request)	외부 장치에서 발생된 인터럽트 신호가 ARM에 입력되면 발생
IRQ (Interrupt Request)	외부 장치에서 발생된 인터럽트 신호가 ARM에 입력되면 발생
SWI (Software Interrupt)	(사용자의 요청에 의한) SWI 명령어 실행 시 발생
Prefetch Abort	잘못된 주소 공간에서 명령을 읽으려고 할 때 MMU나 메모리 제어기에서 발생한 Abort 신호의 입력에 의해 발생
Data Abort	잘못된 주소 공간에서 데이터를 읽거나 쓰려고 할 때 MMU나 메모리 제어기에서 발생한 Abort 신호의 입력에 의해서 발생
Undefined Instruction	ARM에 정의되지 않은 명령을 수행하려고 하거나, 코프로세서에서 응답이 없으면 발생

표 2.25 ARM 프로세서 예외처리 및 발생 시기

ARM 프로세서에서는 리셋과 소프트웨어 인터럽트 예외처리가 발생하면 SVC 모드, Undefined Instruction 예외처리가 발생하면 Undefined 모드, Prefetch Abort와 Data Abort 예외처리가 발생하면 Abort 모드, IRQ가 발생하면 IRQ모드, FIQ가 발생하면 FIQ 모드로 전환된다. 예외처리는 동시에 발생 할 수도 있으므로 우선순위를 미리 정해놓고 있으며, 예외처리가 발생하면 미리 정해진 주소에 있는 프로그램을 수행하도록 되어 있다. 이 위치를 예외처리 벡터(Vector)라 부르고, 각 예외처리에 대하여 벡터 주소를 정의해 놓은 표를 벡터 테이블이라고 한다. 기본적으로 ARM 프로세서는 0x00000000 번지에 벡터 테이블을 둔다.

예외처리	벡터 주소	동작모드	우선순위
Reset	0x0000 0000	SVC (Supervisor)	1(High)
Undefined Instruction	0x0000 0004	Undefined	6(Low)
SWI (Software Interrupt)	0x0000 0008	SVC (Supervisor)	6
Prefetch Abort	0x0000 000C	Abort	5
Data Abort	0x0000 0010	Abort	2
Reserved	0x0000 0014		
IRQ	0x0000 0018	IRQ	4
FIQ	0x0000 001C	FIQ	3

표 2.26 ARM 프로세서 예외처리

표 2.26은 ARM 프로세서의 예외처리와 벡터 주소, 전환되는 동작모드 및 우선순위를 나타낸다. ARM 프로세서에서는 벡터 테이블에 분기(Branch) 명령 또는 이와 유사한 명령을 사용하여 실제 예외처리를 처리하기 위한 핸들러로 분기하도록 되어 있다.

ARM 프로세서는 예외처리가 발생하면 하드웨어적으로 다음 순서에 따른 동작을 수행한다.

1) CPSR을 SPSR_<mode>에 복사한다.
2) CPSR의 비트를 수정한다.
   - ARM 상태로 바꾼다.
   - 프로세서의 모드 비트를 수정한다.
   - 예외처리에 따라 필요하면 인터럽트(I, F 비트)를 Disable 한다.
3) 복귀할 주소를 LR_<mode>에 저장한다.
4) PC 값을 벡터 주소로 바꾼다.

예외처리 벡터에는 예외처리 핸들러로 분기하는 명령이 있고, 예외처리 핸들러에서는 사용할 범용 레지스터를 스택에 저장하고 요청된 예외처리

를 한다. 예외처리가 완료되면 스택에 저장했던 범용 레지스터 값을 다시 복원하고, PC 값과 CPSR 값을 복원하여 예외처리가 발생하기 이전 상태로 되돌아간다.

본 논문에서는 ARM 프로세서의 예외처리 메커니즘을 이용하여 인위적으로 예외처리를 발생시키고 예외처리 핸들러에서 GDB 스텁 모듈로 분기하도록 함으로써 원하는 시점에 GDB 스텁 모듈로 진입하도록 구현하였다.

```
BreakPoint:
    BKPT
    BX LR
```

표 2.27 예외처리 발생 명령

표 2.27은 GDB 스텁 모듈로 진입하기 위해 본 논문에서 사용하는 예외처리 발생 명령을 나타낸다. ARM 프로세서의 명령어인 BKPT(Breakpoint) 명령은 Prefetch Abort 예외처리를 발생시킨다. 예외처리가 발생하면 다음 표 2.28과 같은 예외처리 핸들러 함수로 분기하게 된다. 예외처리 핸들러 함수에서는 복귀해야 할 주소를 계산하고, 원격 시리얼 프로토콜 파싱 함수를 호출한다. 원격 시리얼 프로토콜 파싱 함수에서는 호스트 머신의 디버거인 GDB의 디버깅 요청 명령을 받아 명령에 해당하는 기능을 처리하고 요청 명령에 대한 응답을 하도록 구현되어 있다. 원격 시리얼 프로토콜 파싱 함수가 종료되면 예외처리가 발생하였을 때 계산한 복귀 주소를 이용하여 예외처리가 발생하기 전의 상태로 복귀한다.

```
static void exceptionHandler()
{
        exceptionRegisters[15] = *(uint32_t *)0x027FFD98;

        retAddr = computeReturnAddress();
```

```
        build_t_packet();

        putPacket(remcomOutBuffer);

        debugStub();

        jumpBack(retAddr);
}
```

표 2.28 예외처리 핸들러 함수

ARM 프로세서가 외부에서 발생되는 예외처리를 인식하고 처리하는 시점은 명령어 파이프라인과 밀접한 관계가 있다. 이는 프로그램에서 예외처리를 한 후에 되돌아가는 위치 정보를 아는 데 매우 중요하다. 각 예외처리 별로 발생되는 명령어 파이프라인과 되돌아갈 때 PC에 저장되는 위치 정보는 표 2.29와 같다. 본 논문에서는 Prefetch Abort 예외처리를 발생시켜 GDB 스텁 모듈로 진입하도록 되어있으므로, 표 2.30과 같은 방법으로 예외처리에서 복귀하도록 하도록 구현하였다. 해당 함수에서는 CPSR의 모드 비트를 검사하여 현재 동작모드에 따른 복귀 주소를 계산하도록 되어있다.

예외처리	명령 파이프라인	PC
Undefinded Instruction	Decode 단계	PC = LR
SWI (Software Interrupt)	Decode 단계	PC = LR
Prefetch Abort	Execute 단계	PC = LR - 4
Data Abort	Memory 단계	PC = LR - 8
IRQ	Execute 단계	PC = LR - 4
FIQ	Execute 단계	PC = LR - 4

표 2.29 ARM 프로세서 예외처리 및 명령 파이프라인

```
static uint32_t computeReturnAddress()
{
	uint32_t returnAddress;
	uint32_t currentMode = getCPSR() & 0x1f;
	uint32_t thumbState = getSPSR() & 0x20;
	// Prefetch Abort
	if(currentMode == 0x17)
	{
		returnAddress = exceptionRegisters[PC] - 4;

		if((uint32_t)BreakPoint == returnAddress)
		{
			exceptionRegisters[PC] += 4;
			returnAddress += 4;
		}
	}
	else
	{
		returnAddress = exceptionRegisters[PC] - 4;

		if(thumbState)
		{
			returnAddress += 2;
		}
	}
	return returnAddress;
}
```

표 2.30 복귀 주소 계산 함수

3. 원격 시리얼 프로토콜 구현

본 논문에서 사용한 GDB는 원격 디버깅을 위한 방법으로 리모트 시리얼 프로토콜(Remote Serial Protocol)이 정의되어 있다. 타겟, 호스트 시스템 간 원격 디버깅을 위한 GDB의 모든 명령 및 응답은 패킷으로 전송된

다. 패킷은 표 2.31과 같이 패킷의 시작을 알리는 문자인 '$', 패킷의 실제 데이터, 패킷의 종료를 알리는 문자인 '#', 두 자리의 체크섬 순으로 구성되어 있다.

$packet-data#checksum

표 2.31 GDB 원격 시리얼 프로토콜 패킷 형식

타겟, 호스트 시스템 간 원격 디버깅이 이루어지기 위해서는 GDB 스텁 모듈에서 'g', 'G', 'm', 'M', 'c', 's' 명령에 대해 필수적으로 처리해 주어야 하며, 멀티 쓰레드를 지원하기 위해서는 'vCont' 명령에 대한 처리를 해야 한다. 그 외의 나머지 명령은 모두 선택적이다. 표 2.32는 본 논문에서 구현한 GDB 원격 디버깅 명령 및 명령의 의미를 나타낸다.

명 령	의 미
+	응답 (패킷이 정상적으로 수신)
-	응답 (패킷을 재전송해 줄 것을 요청)
?	타겟이 정지된 이유
g	레지스터 읽기
G	레지스터 쓰기
m	메모리 읽기
M	메모리 쓰기
c	컨티뉴 요청
s	스텝 요청
vCont	특정 쓰레드에 컨티뉴, 스텝, 정지 요청

표 2.32 GDB 원격 디버깅 명령

제 7 절 가상 디바이스

본 논문에서 구성한 가상 디바이스는 닌텐도 DS-lite에 있는 LCD 화면과 터치패드 상에서 동작하는 하드웨어 시뮬레이터이다. 본 논문에서 제공하는 임베디드 소프트웨어 교육을 위한 환경에서는 자원의 추가에 다른 비용 문제로 하드웨어를 추가할 수 없지만 교육에 필요한 하드웨어들을 가상 디바이스 형태로 제공하고, 그 장치에 대한 제어를 실제 하드웨어와 같은 방법으로 제어할 수 있도록 함으로써 실제 하드웨어를 사용한 것과 같은 교육 효과를 얻을 수 있도록 하는 환경을 제공한다. 본 논문에서는 많은 임베디드 시스템에 존재하고 교육 대상이 되는 7 Segment LED, Bar LED, Switch, Key-Matrix를 구현하여 실시간 운영체제의 태스크 형태로 제공하고, 가상 디바이스들에 대한 데이터시트를 실제 하드웨어와 같은 형태로 제공한다. 그림 2.17은 실시간 운영체제 환경에서 실행되는 가상디바이스의 모습을 나타낸다.

그림 2.17 가상 디바이스

제 8 절 통합 개발 환경

본 논문에서 구성한 통합 개발 환경은 현재 임베디드 시스템 개발에 가장 많이 사용하고 있는 이클립스 CDT를 사용하여 개발 호스트 상에서 이루어지는 모든 개발 관련 작업(프로젝트 관리, 프로그램 편집, 컴파일, 다운로드 및 원격 디버깅)을 논문에서 구성한 환경 내에서 처리할 수 있도록 하는 기능을 제공한다. 콘솔 다운로더, 에뮬레이터 실행 등 개발 호스트 쪽에 추가되는 모든 소프트웨어는 이클립스 플러그인 또는 외부 프로그램 실행을 통한 형식으로 개발하여 연동하고 있으며, GDB를 사용하여 이클립스 IDE가 제공하는 디버깅 UI를 통해 원격 디버깅을 실행할 수 있도록 하는 기능을 제공한다. 그림 2.18은 통합 개발 환경 내에서 원격 디버깅을 진행하는 모습을 나타낸다.

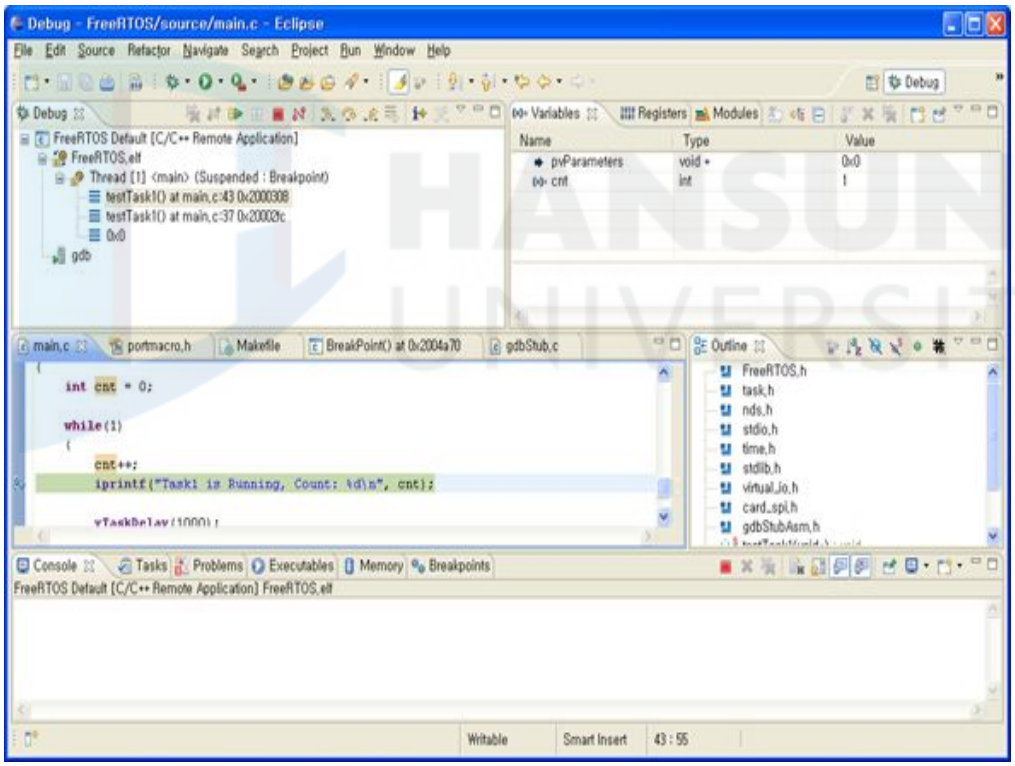


그림 2.18 이클립스 IDE를 통한 원격 디버깅

제 7 절 콘솔 다운로더

본 논문에서 구현한 콘솔 다운로더는 호스트 시스템에서 개발 한 결과물을 타겟 시스템인 닌텐도 DS로 다운로드 할 수 있게 하는 기능을 제공하는 프로그램으로, 무선 랜, 시리얼, USB를 통한 다운로드가 가능하다. 다운로드는 다운로드의 시작을 나타내는 문자열, 다운로드 할 데이터의 정보(파일 이름, 크기 등)를 나타내는 헤더, 실제 다운로드 할 파일, 파일의 무결성을 위한 체크섬, 다운로드의 끝을 나타내는 문자열 데이터를 타겟, 호스트 간 송수신함으로써 진행된다. 표 2.33은 다운로드 프로그램에서 사용되는 프로토콜 정보를 나타낸다.

```
// Download Port Number for WIFI connection
#define DOWNLOAD_PORT    3389     // Just it, No special meaning.

// File Transfer protocol
#ifndef MAX_FILE_NAME_LEN
#define MAX_FILE_NAME_LEN       32
#endif

#define MAX_FILE_SIZE    (2 * 1024 * 1024)
#define MAGIC_LEN        16
#define MAGIC_DOWNLOAD_STRING          "(&^@%=DALL=}\~+)"
#define MAGIC_CLEANUP_STRING           ")+~\}=LLAD=%@^&("

#define MAX_FILE_LENGTH_LEN    16
#define RESV_LEN               8
struct header {
    char filename[MAX_FILE_NAME_LEN + 4];  // MAX 32 byte ASCIZ file name
    char flash;     // 'F' if write after download, otherwise 'X'
    char run;       // 'R' if run after download, otherwise 'X'
    char debug;     // 'D' if run with debug, otherwise 'X'
```

```
    char zero;      // set 0
    char filelength[MAX_FILE_LENGTH_LEN];     // ASCIZ file length
    char reserved[RESV_LEN];        // Reserved
};
#define HEADER_SIZE sizeof(struct header)

#define RESP_CODE_LEN             4
#define RESP_STRING_LEN           28
struct response {
    char code[RESP_CODE_LEN];               // 3 Byte Reponse Code and Blank
    char string[RESP_STRING_LEN];           // Response String (ASCIZ)
};
#define RESPONSE_SIZE sizeof(struct response)

// Response Code definition
// "000" : No Error
// "001" : No Error, with Some Comments as follows
// "100" ~ "899" : Error and Cause
// "999" : Internal Error

// Protocol Description
// 1. HOST -> NDS  : MAGIC DOWNLOAD STRING
// 2. HOST -> NDS  : FILE HEADER
// 3. NDS  -> HOST : OK
// 4. HOST -> NDS  : File Data (in Bytes)
// 5. HOST -> NDS  : CHECKSUM
// 6. NDS  -> HOST : OK
// 7. HOST -> NDS  : MAGIC CLEANUP STRING
```

표 2.33 다운로드 프로그램 프로토콜

시리얼, USB 연결을 통한 프로그램 다운로드 기능은 논문에서 구현한 어댑터 하드웨어를 이용하여 사용할 수 있으며, 어댑터 하드웨어가 없는 경우에도 무선 랜 연결을 통해 다운로드 기능을 사용할 수 있다.

무선 랜 연결을 통한 프로그램 다운로드는 개발 호스트 시스템과 타겟

시스템이 같은 AP의 무선 랜 상에서 서버, 클라이언트로 동작하여 파일을 전송할 수 있도록 하는 모듈과 개발 호스트 시스템이 무선 랜을 지원하지 않는 경우를 고려하여 파일 중계 서버를 사용하여 파일을 전송할 수 있도록 하는 모듈을 포함하고 있다. 그림 2.19는 다운로드 프로그램을 개발 호스트 시스템과 타겟 시스템에서 실행한 모습을 나타낸다.

그림 2.19 콘솔 다운로더를 이용한 프로그램 다운로드

제 8 절 어댑터 하드웨어

본 논문에서는 호스트 시스템과의 연결 방법이 무선 랜 밖에 없는 닌텐도 DS-lite에 USB 또는 시리얼 포트 연결성을 부여하고, 가속 센서와 같은 추가적인 장치의 제공, 외부 장치 제어가 가능하도록 그림 2.20의 어댑터 하드웨어를 개발하였다. 어댑터 하드웨어는 Atmel 32 bit CPU, USB, GPIO, LED, 시리얼, 지자기 센서, 자이로 센서, 3D-가속센서 등으로 구성되어 있으며, 닌텐도 DS에서 사용되는 정품 게임팩을 장착하는 것과 같은 방법으로 게임기와 연결할 수 있다. 또한, 어댑터 하드웨어에 있는 USB, 시리얼, GPIO를 통해 외부의 장치, 회로와 연결 가능하다.

본 논문에서는 어댑터 하드웨어를 기본적으로는 호스트 시스템과 타겟 시스템인 닌텐도 DS-lite 사이의 통신을 중계하는 역할을 하기 위한 장치로 사용하고 있다. 호스트 시스템과 어댑터 하드웨어는 USB 또는 시리얼 인터페이스로 연결되며, 어댑터 하드웨어와 닌텐도 DS-lite 사이는 닌텐도의 게임팩 슬롯이 제공하는 SPI 인터페이스를 이용한 통신이 이루어진다. 어댑터 하드웨어에도 본 논문에서 제공하는 실험 환경과 같은 실시간 운영체제인 FreeRTOS를 사용한다.

그림 2.20 확장 어댑터 하드웨어

제 3 장 교육 실습 개발 환경

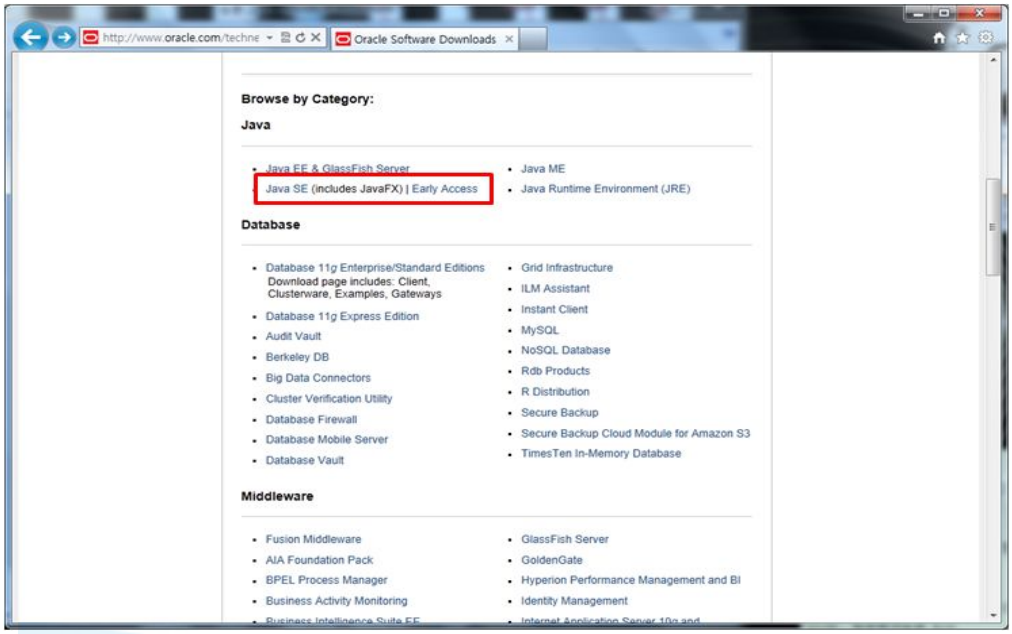
제 1 절 개발 환경 구성

1. JDK

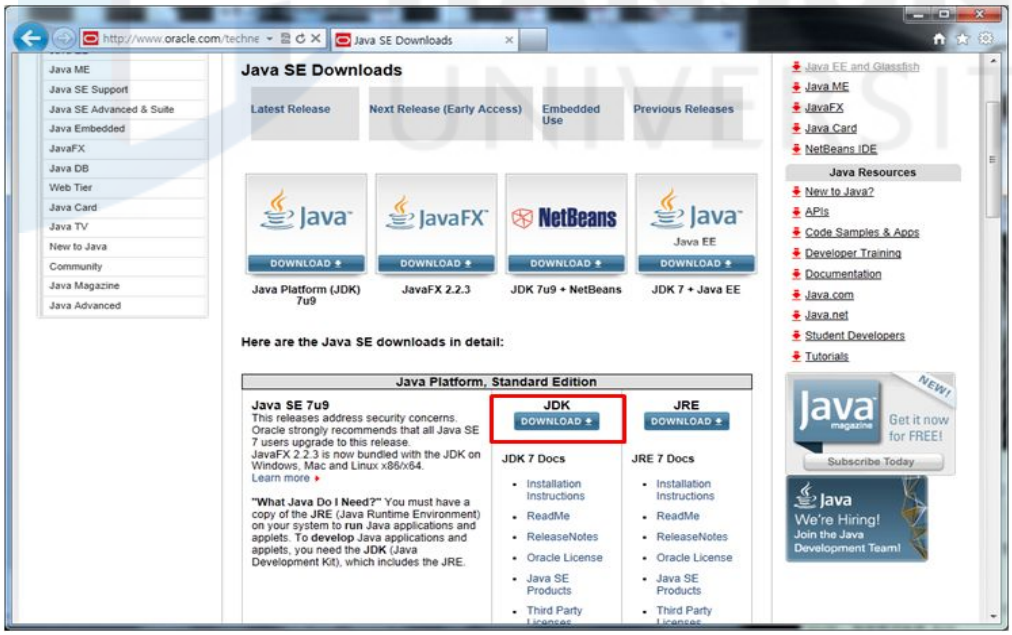
JDK(Java Development Kit)는 본 논문에서 사용하는 통합 개발 환경인 Eclipse 개발 도구가 Java로 만들어져 있으므로 Eclipse를 실행하기 위해 필요한 도구로 Eclipse를 설치하기 전에 먼저 설치한다.

1) http://www.oracle.com에서 Downloads 메뉴를 선택한다.

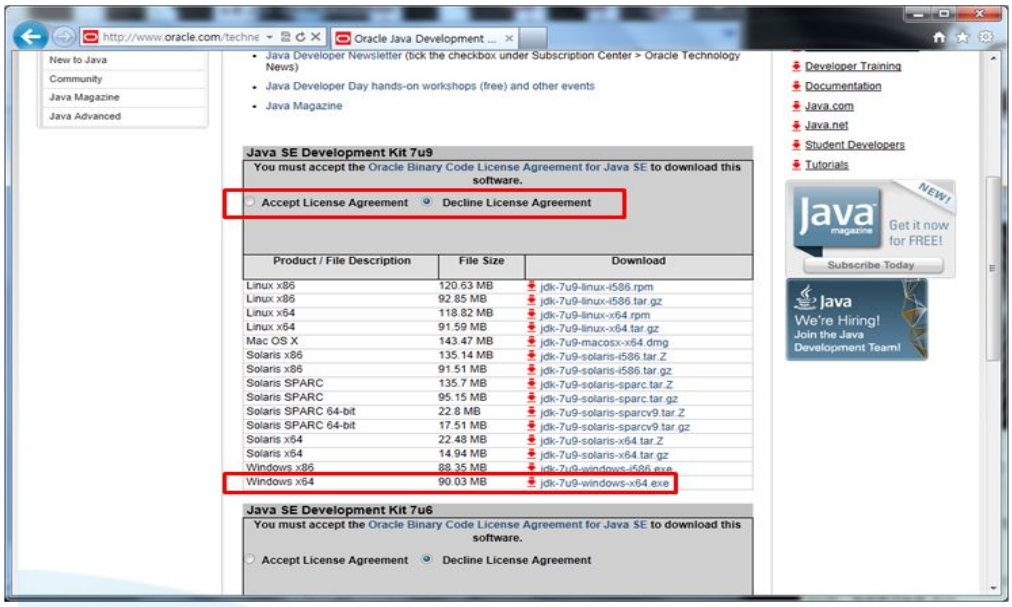
2) Java 항목의 Java SE 메뉴를 선택한다.



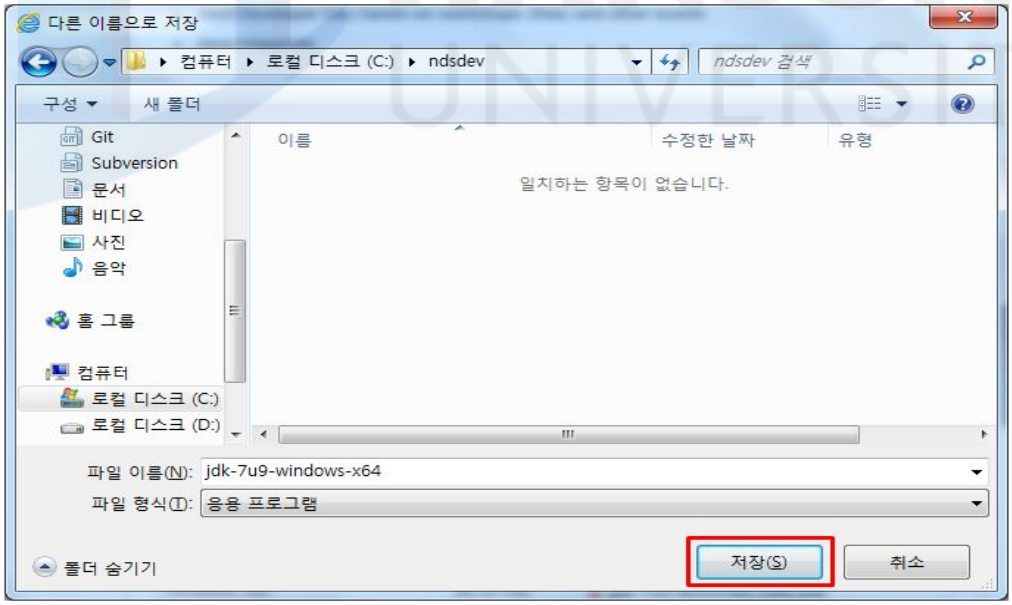
3) JDK Download 버튼을 선택한다.



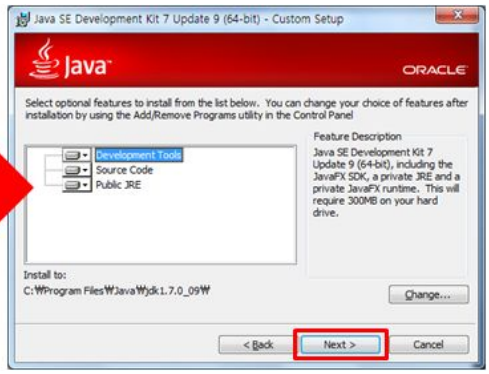
4) Accept License Agreement 항목을 체크한 뒤, 운영체제에 맞는 버전을 선택한다.

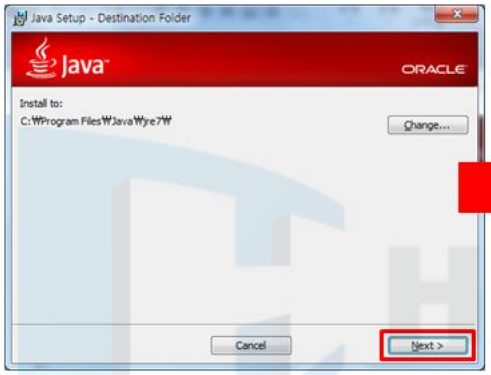


5) 다운로드 파일 저장을 위한 창이 나오면 적당한 곳에 저장한다.



6) 다운로드 받은 파일을 실행하여 JDK를 설치한다. (Next, Accept 등을 누르면 별 문제없이 설치된다.)

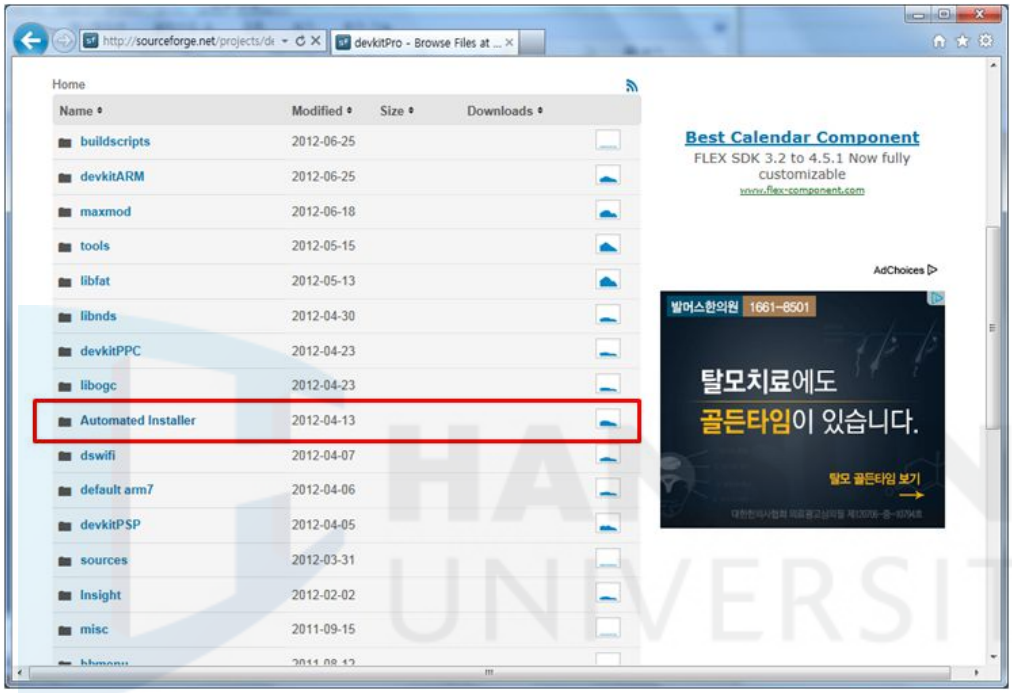




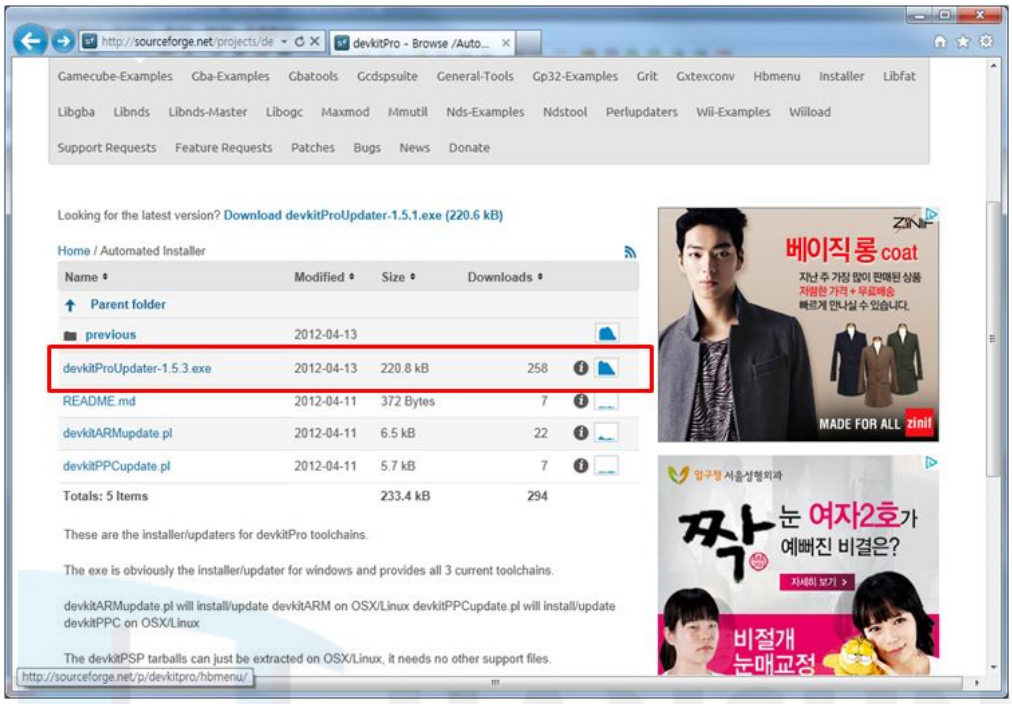
2. devkitPro

devkitPro는 닌텐도 DS를 이용한 소프트웨어 개발에 필요한 툴 체인(컴파일러, 링커 등) 및 라이브러리를 제공하는 도구이다.

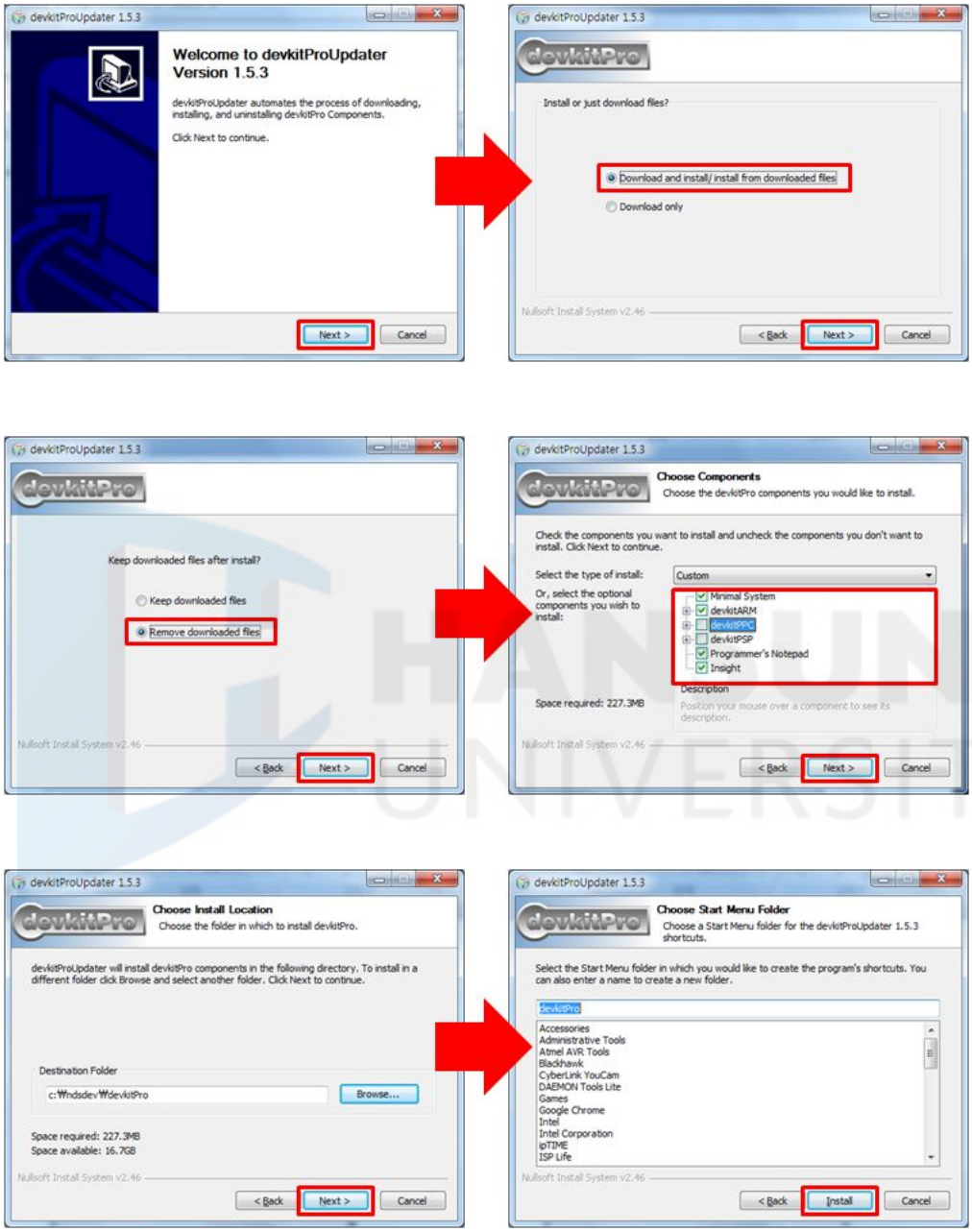
1) http://sourceforge.net/projects/devkitpro/files/에서 Automated Installer 메뉴를 선택한다.

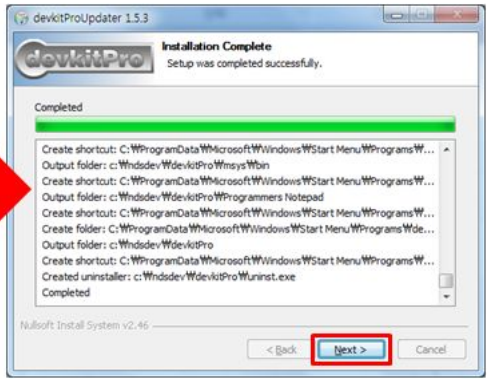


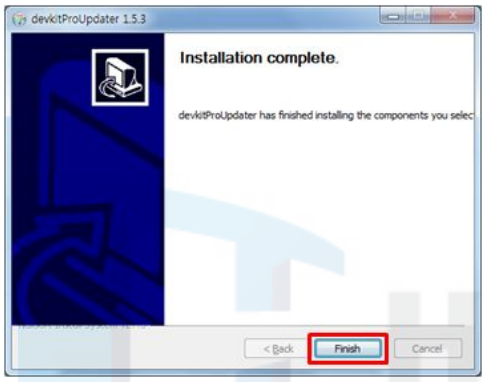
2) devkitProUpdater-1.5.3.exe 파일을 선택한 뒤, 다운로드 파일 저장을 위한 창이 나오면 적당한 곳에 저장한다.



3) 다운로드 받은 파일을 실행하여 devkitPro를 설치한다. (Next, Accept 등을 누르면 별 문제없이 설치된다.)



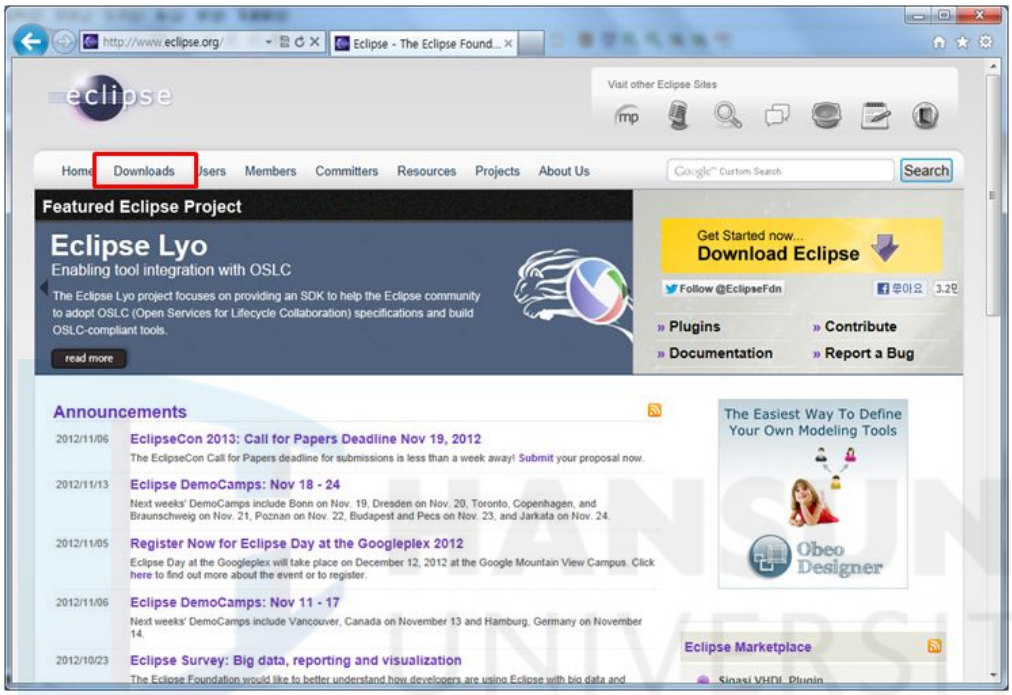




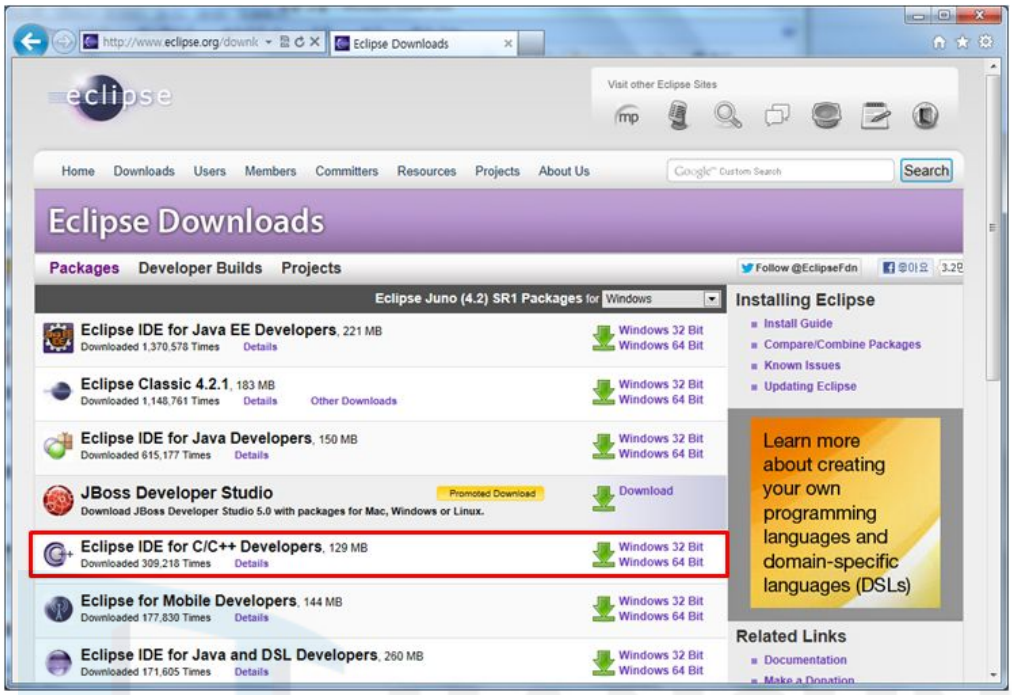
3. Eclipse

Eclipse는 프로젝트 수준에서 닌텐도 DS를 이용한 소프트웨어 개발을 하기 위한 IDE(Integrated Development Environment) 도구이다.

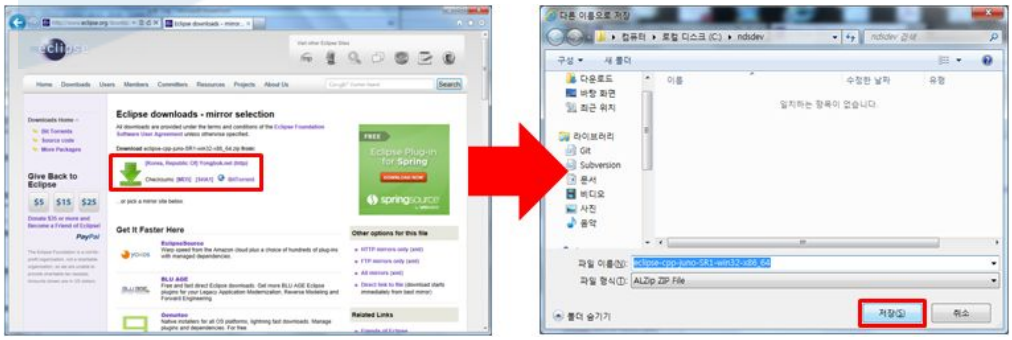
1) http://www.eclipse.org에서 Downloads 메뉴를 선택한다.



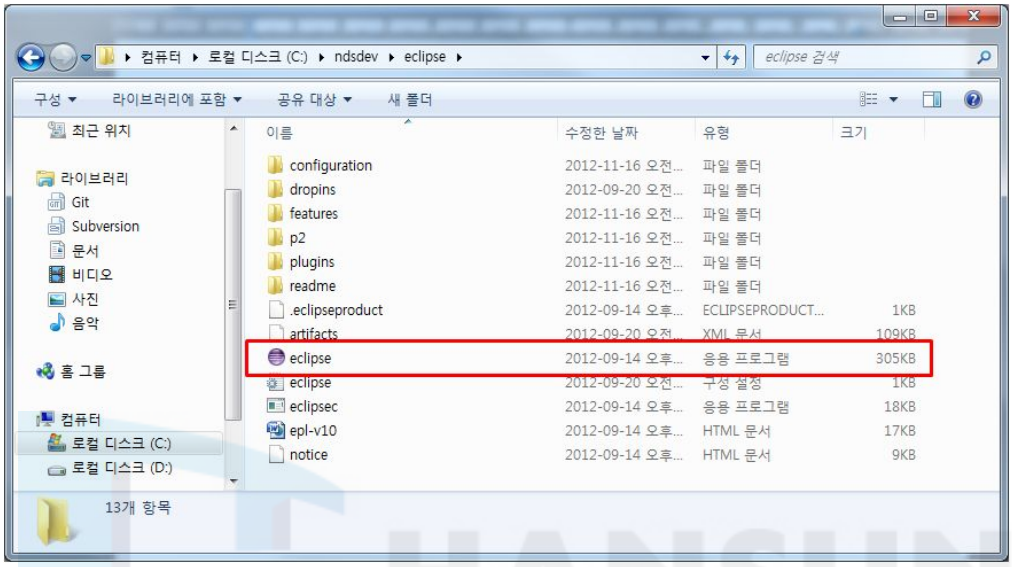
2) Eclipse IDE for C/C++ Developers 항목에서 운영체제에 맞는 버전을 선택한다.

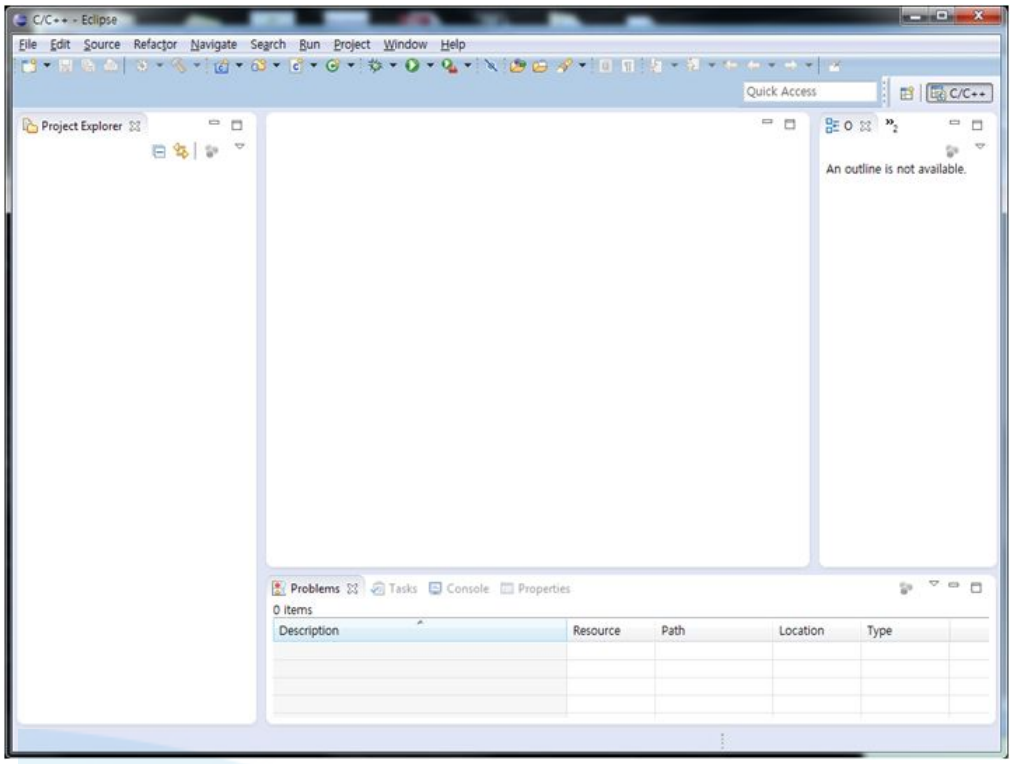


3) 녹색화살표를 누른 뒤, 다운로드 파일 저장을 위한 창이 나오면 적당한 곳에 저장한다.



4) 다운로드 받은 파일을 압축 해제한 뒤, Eclipse 실행파일을 실행하고 워크스페이스를 지정하기 위한 창이 나오면 C:\ndsdev\workspace로 프로젝트를 저장할 기본 폴더를 지정한 뒤, OK 버튼을 눌러 Eclipse를 실행한다. (Eclipse CDT는 별도의 설치가 필요 없이 앞서 설치했던 JDK만 있으면 실행 가능하다.)

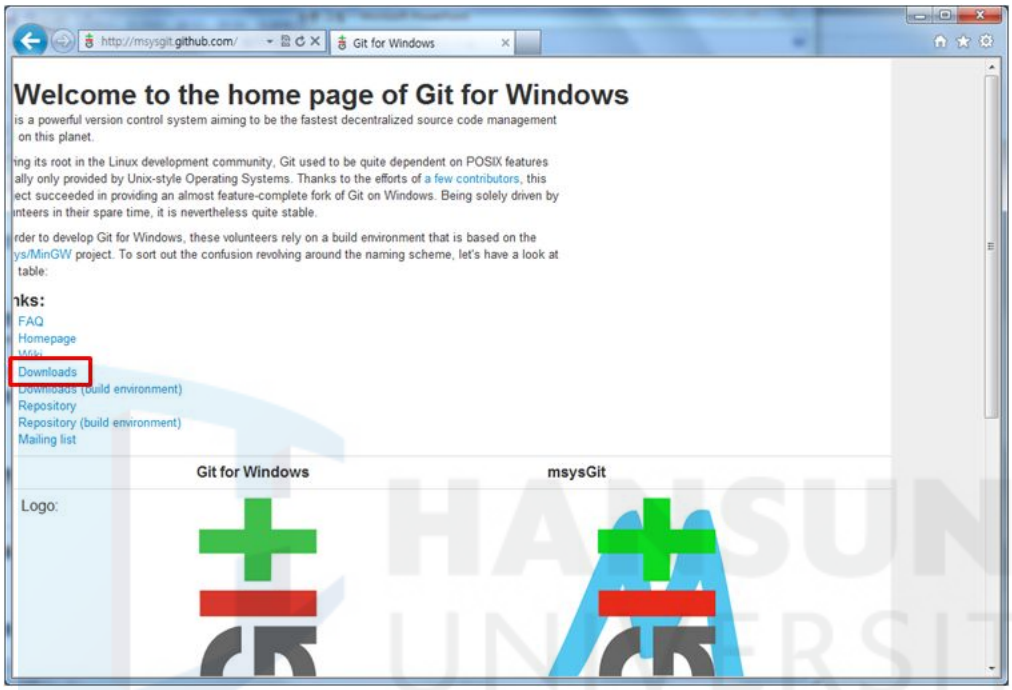




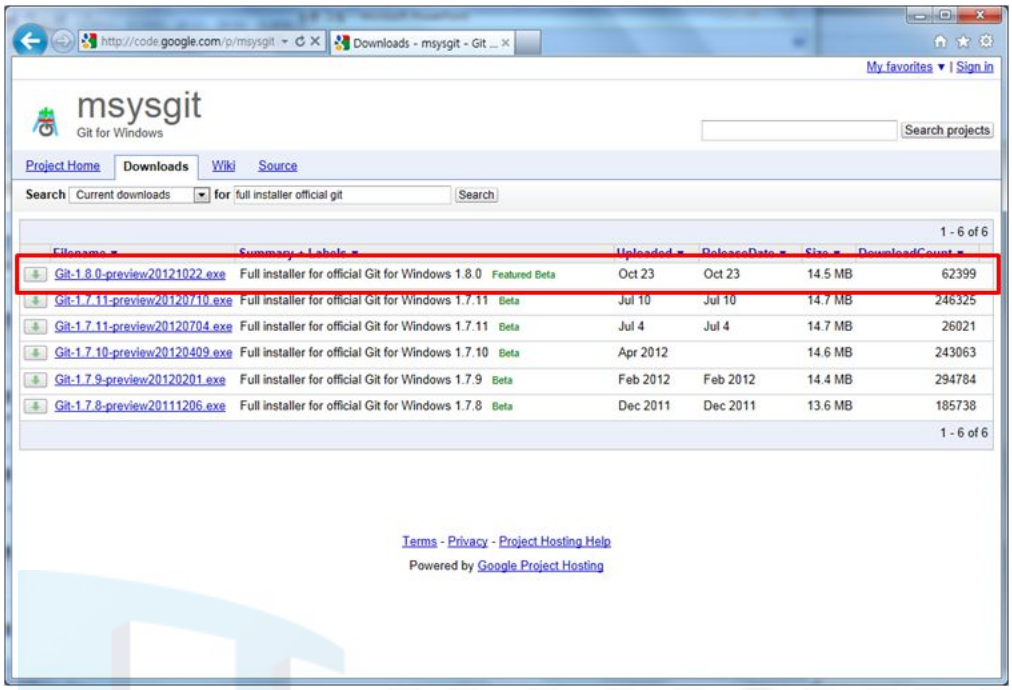
4. Git

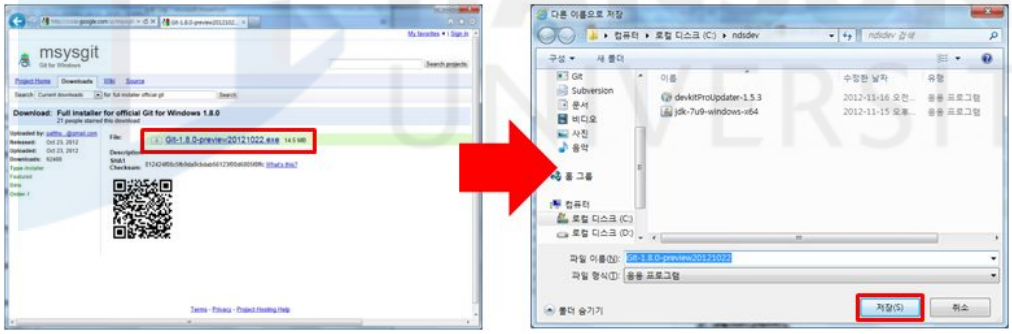
Git은 프로그램 등의 소스 코드 관리를 위한 분산형 버전 관리 시스템(Distributed Version Control System)이다.

1) http://msysgit.github.com에서 Downloads 메뉴를 선택한다.

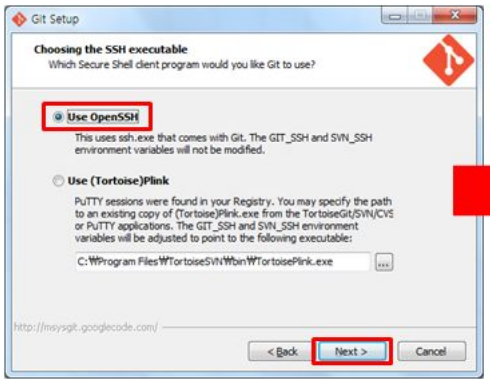


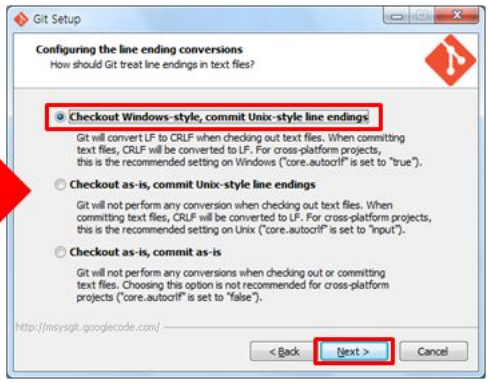
2) Git-1.8.0-preview20121022.exe 항목을 선택한 뒤, 다운로드 파일 저장을 위한 창이 나오면 적당한 곳에 저장한다.

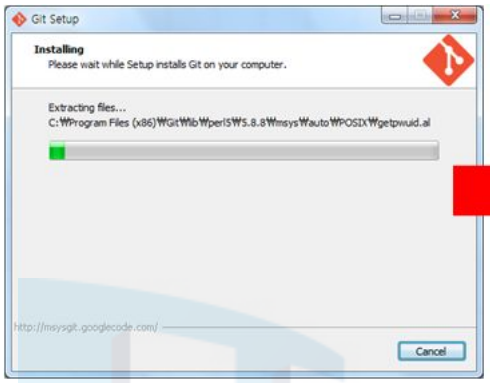




3) 다운로드 받은 파일을 실행하여 Git를 설치한다. (Next, Accept 등을 누르면 별 문제없이 설치된다.)





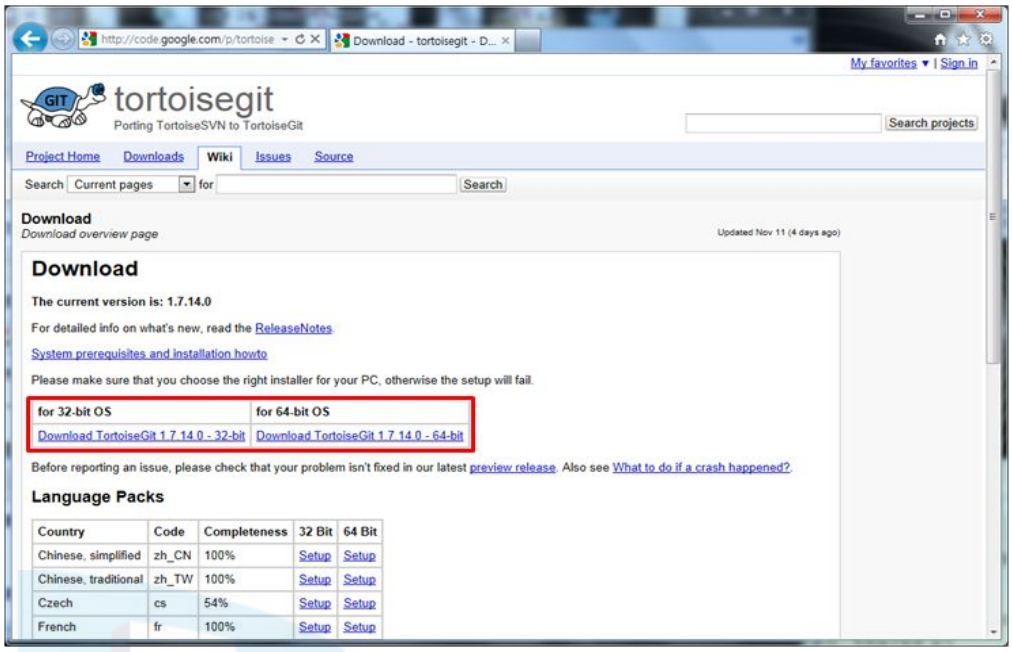


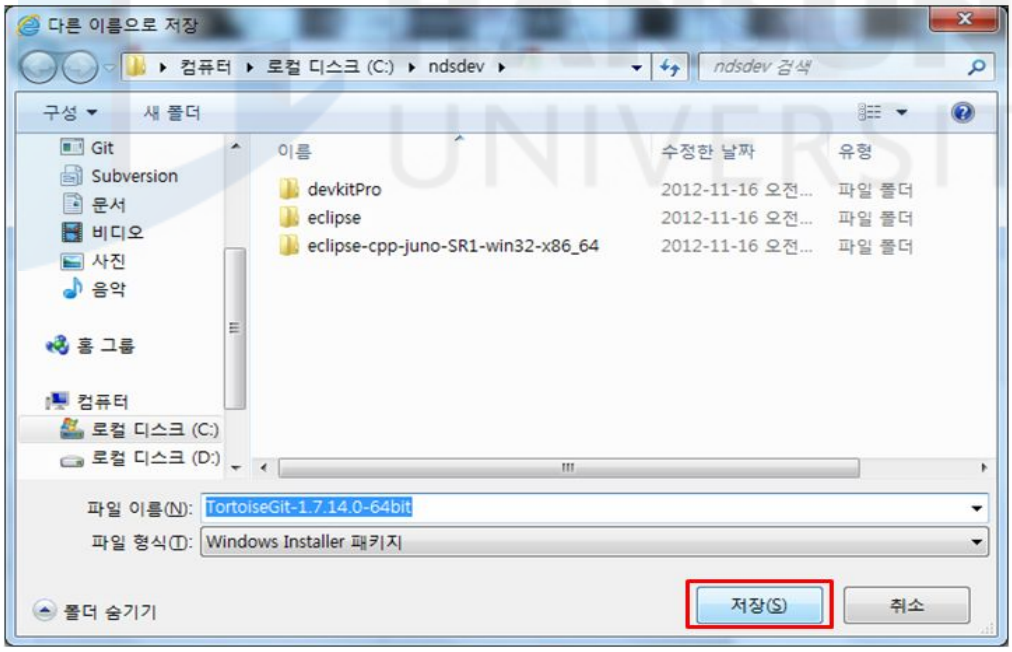
5. Tortoise Git

Tortoise Git은 Windows 운영체제에서 GUI를 통해 Git을 사용할 수 있도록 하는 도구이다.

1) http://code.google.com/p/tortoisegit/에서 Download TortoiseGit 메뉴를 선택한다.

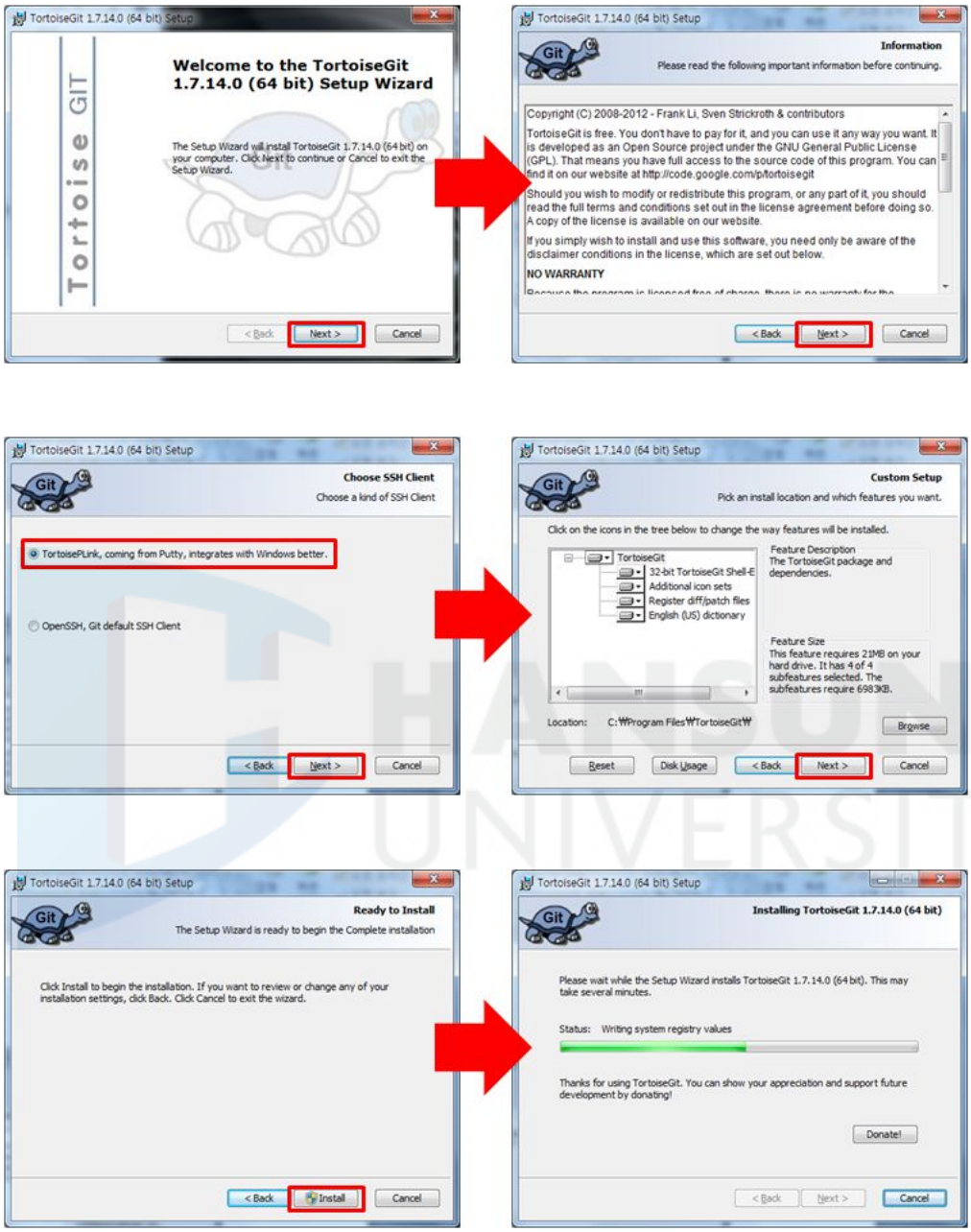
2) 운영체제에 맞는 버전을 선택한 뒤, 다운로드 파일 저장을 위한 창이 나오면 적당한 곳에 저장한다.

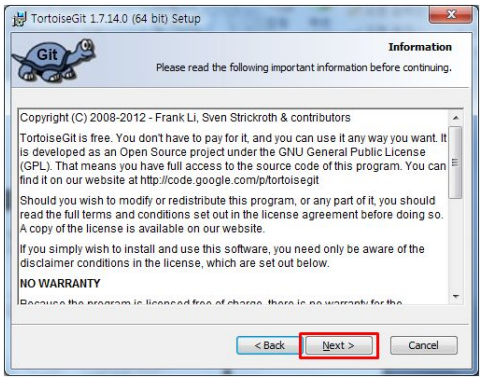




3) 다운로드 받은 파일을 실행하여 Tortoise Git를 설치한다.

(Next, Accept 등을 누르면 별 문제없이 설치된다.)



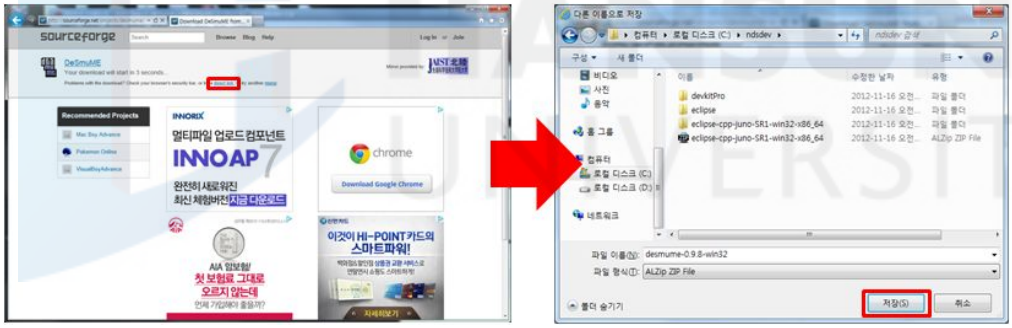


6. Emulator (DeSmuME)

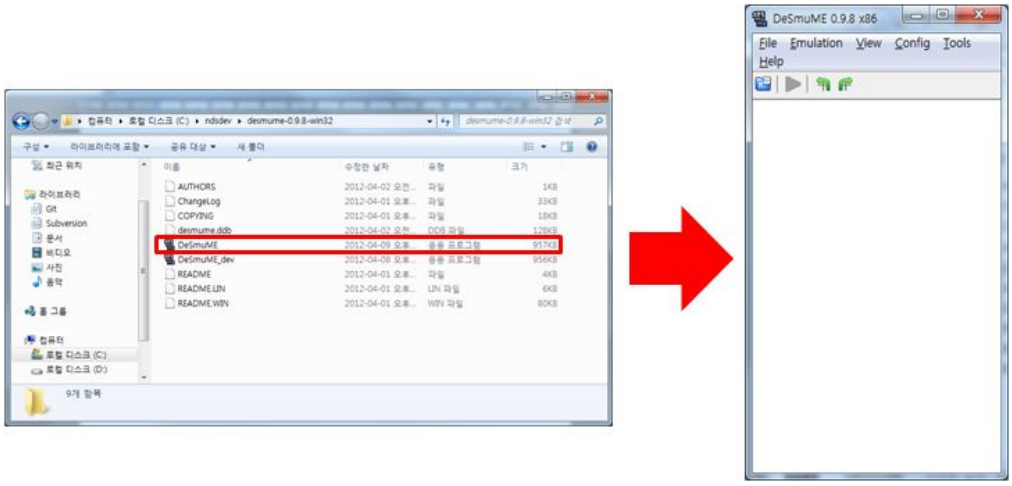
DeSmuME는 닌텐도 DS 응용 프로그램을 실행할 수 있는 에뮬레이터이다.

1) http://desmume.org/에서 Download 메뉴를 선택한다.

2) 운영체제에 맞는 버전을 선택한 뒤, 다운로드 파일 저장을 위한 창이 나오면 적당한 곳에 저장한다.



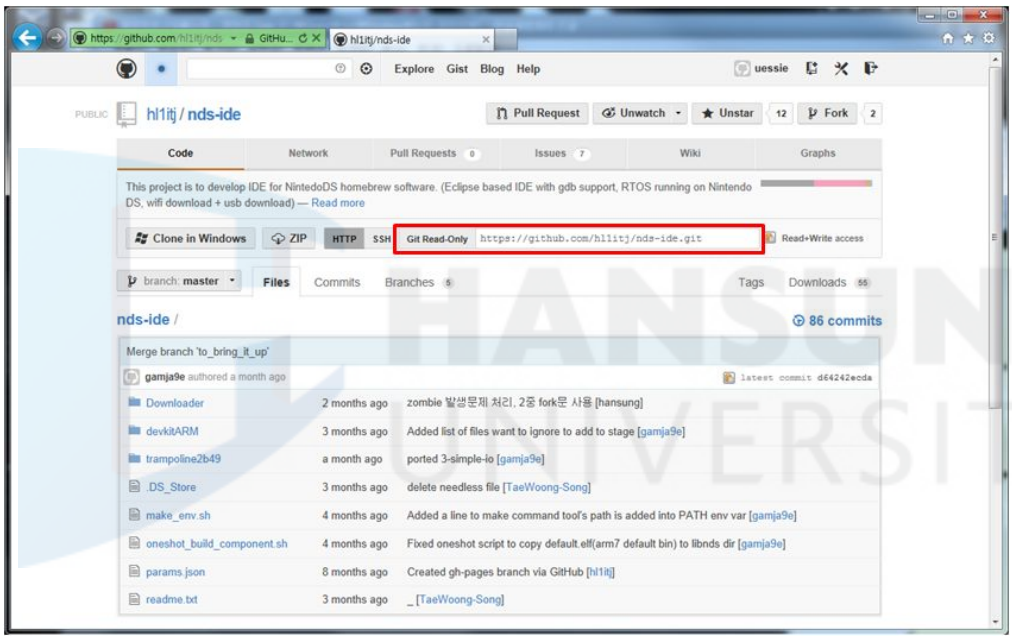
3) 다운로드 받은 파일을 압축 해제한 뒤, DeSmuME 실행파일을 실행한다.



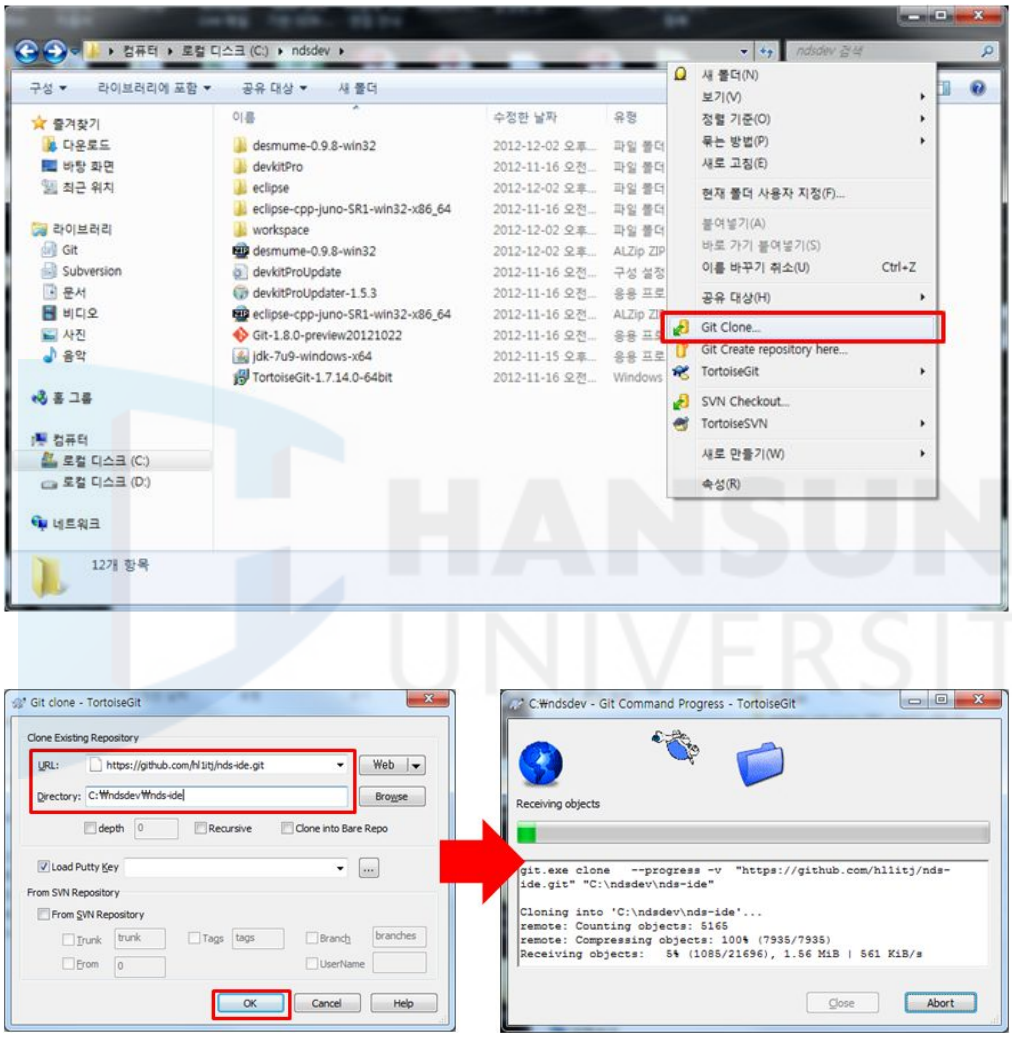
7. NDS-IDE Repository

본 논문에서 제공하는 모든 결과물(개발 환경, 실험 교육에 사용된 문서, 소스 등)은 Github, Sourceforge를 통해 공개하고 있으며, 현재는 모든 작업을 Github을 이용하여 진행하고 있다. Git를 이용하여 실습을 위해 필요한 다운로드 프로그램, 실습교재 및 템플릿 소스, 샘플 프로그램 등을 Github 홈페이지의 Repository를 Clone하여 다운로드 받을 수 있다.

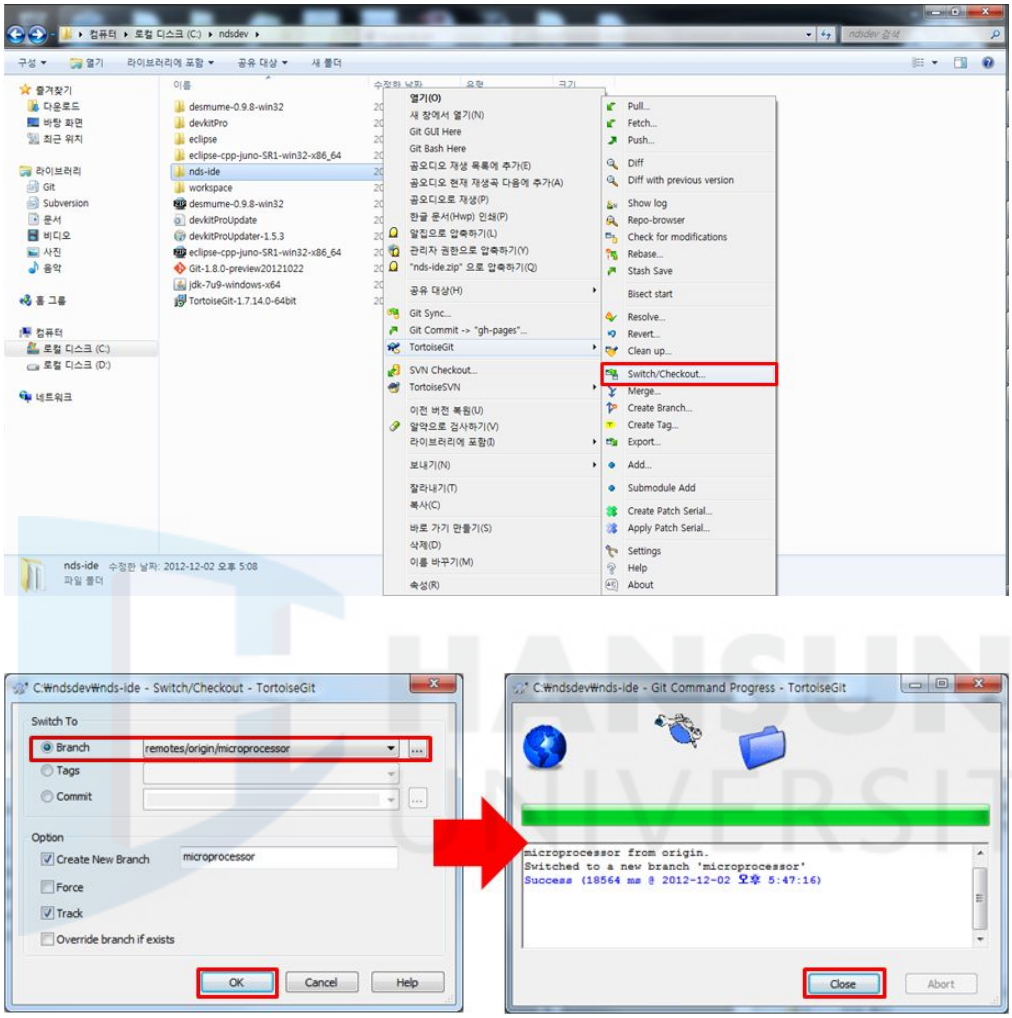
Sourceforge 홈페이지 : http://nintendo.sourceforge.net
Github 홈페이지 : http://github.com/hl1itj/nds-ide



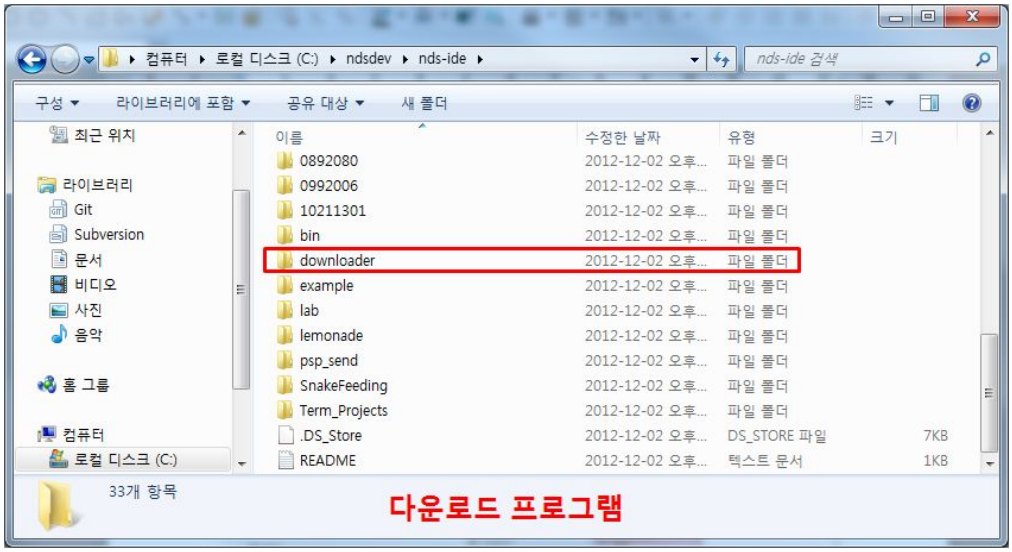
1) Tortoise Git를 이용하여 실습을 위해 필요한 다운로드 프로그램, 실습 교재 및 템플릿 소스, 샘플 프로그램 등을 Github 홈페이지의 Repository를 Clone하여 다운로드 받는다. (마우스 오른쪽 버튼을 클릭하여 Git Clone... 메뉴를 선택한 뒤, Github 홈페이지의 Repository 주소와 다운로드 받을 디렉토리 위치를 입력하고 OK 버튼을 눌러 Clone 한다.)

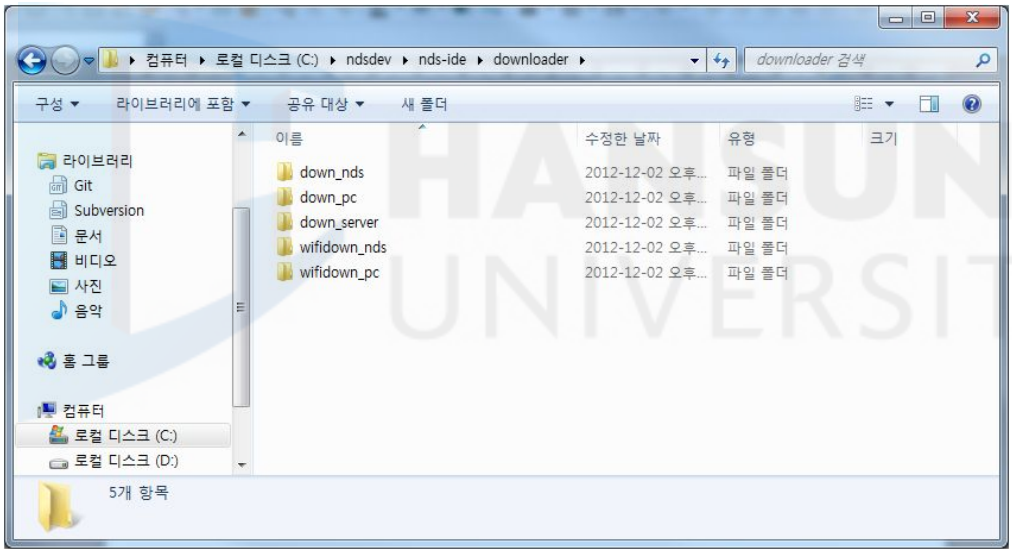


2) 다운로드 받은 디렉토리를 마우스 오른쪽 버튼을 클릭하여 TortoiseGit → Switch/Checkout... 메뉴를 선택한 뒤, microprocessor 브랜치를 선택하고 OK 버튼을 눌러 microprocessor 브랜치로 변경한다.

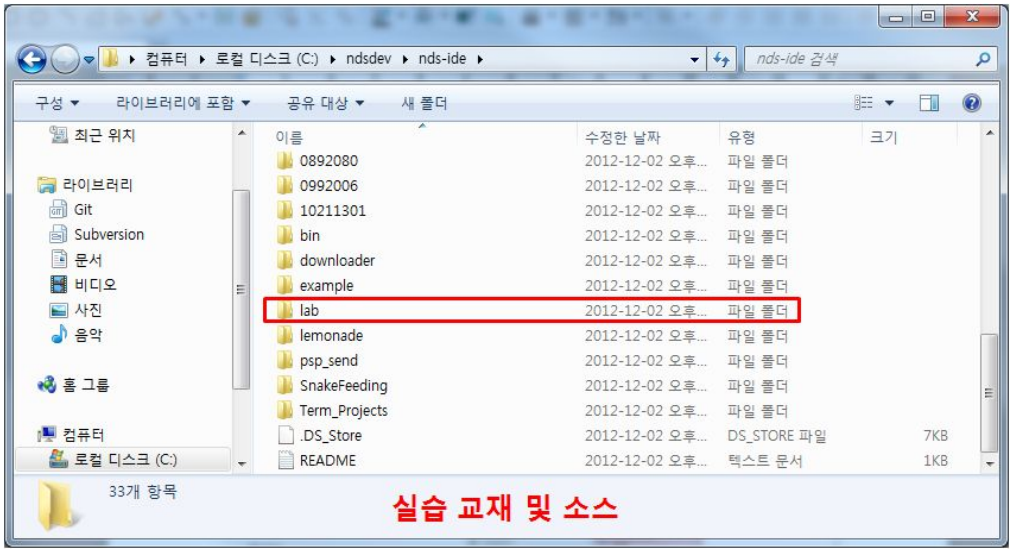


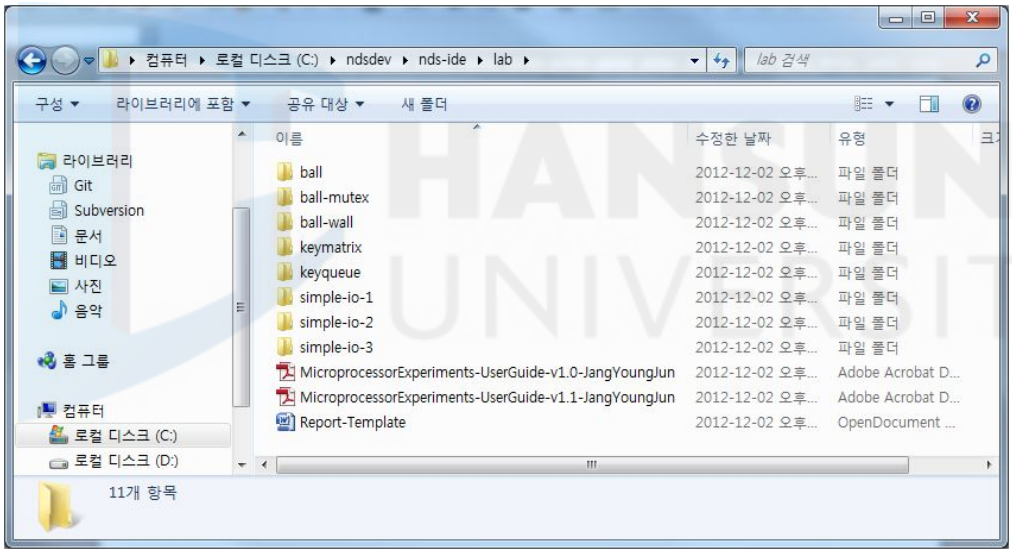
※ 다운로드 프로그램


컴퓨터 ▸ 로컬 디스크 (C:) ▸ ndsdev ▸ nds-ide ▸
nds-ide 검색
구성
라이브러리에 포함
공유 대상
새 폴더
최근 위치
라이브러리
Git
Subversion
문서
비디오
사진
음악
홈 그룹
컴퓨터
로컬 디스크 (C:)
이름
수정한 날짜
유형
크기
0892080
0992006
10211301
bin
downloader
example
lab
lemonade
psp_send
SnakeFeeding
Term_Projects
.DS_Store
README
2012-12-02 오후...
파일 폴더
DS_STORE 파일
7KB
텍스트 문서
1KB
33개 항목
다운로드 프로그램


컴퓨터 ▸ 로컬 디스크 (C:) ▸ ndsdev ▸ nds-ide ▸ downloader ▸
downloader 검색
구성
라이브러리에 포함
공유 대상
새 폴더
라이브러리
Git
Subversion
문서
비디오
사진
음악
홈 그룹
컴퓨터
로컬 디스크 (C:)
로컬 디스크 (D:)
이름
수정한 날짜
유형
크기
down_nds
down_pc
down_server
wifidown_nds
wifidown_pc
2012-12-02 오후...
파일 폴더
5개 항목

※ 실습 교재 및 소스


컴퓨터 ▸ 로컬 디스크 (C:) ▸ ndsdev ▸ nds-ide ▸
nds-ide 검색
구성
라이브러리에 포함
공유 대상
새 폴더
최근 위치
라이브러리
Git
Subversion
문서
비디오
사진
음악
홈 그룹
컴퓨터
로컬 디스크 (C:)
이름
수정한 날짜
유형
크기
0892080
0992006
10211301
bin
downloader
example
lab
lemonade
psp_send
SnakeFeeding
Term_Projects
.DS_Store
README
2012-12-02 오후...
파일 폴더
DS_STORE 파일
7KB
텍스트 문서
1KB
33개 항목
실습 교재 및 소스

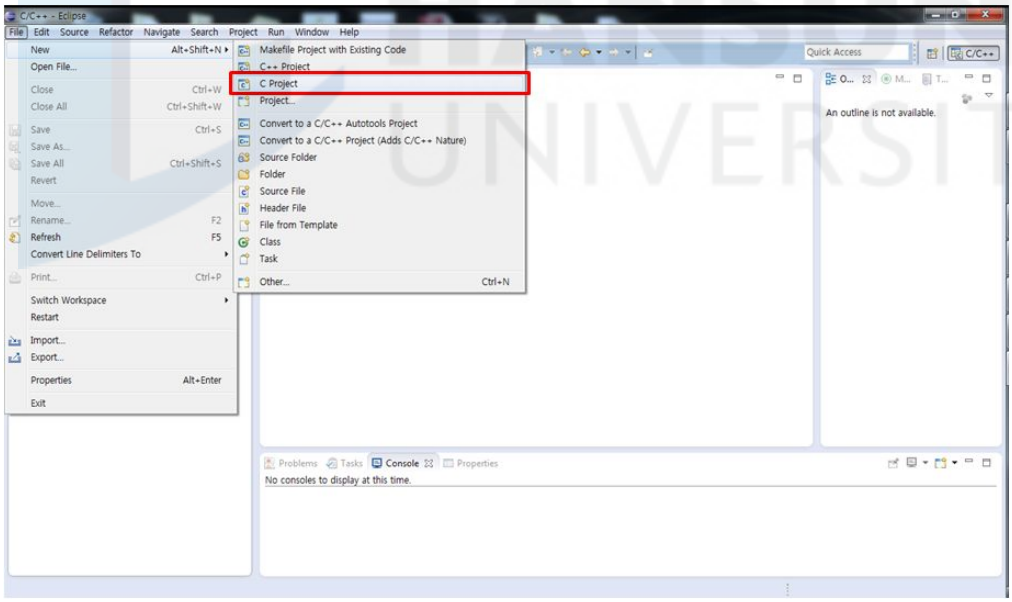

컴퓨터 ▸ 로컬 디스크 (C:) ▸ ndsdev ▸ nds-ide ▸ lab ▸
lab 검색
구성
라이브러리에 포함
공유 대상
새 폴더
라이브러리
Git
Subversion
문서
비디오
사진
음악
홈 그룹
컴퓨터
로컬 디스크 (C:)
로컬 디스크 (D:)
이름
수정한 날짜
유형
ball
ball-mutex
ball-wall
keymatrix
keyqueue
simple-io-1
simple-io-2
simple-io-3
MicroprocessorExperiments-UserGuide-v1.0-JangYoungJun
MicroprocessorExperiments-UserGuide-v1.1-JangYoungJun
Report-Template
2012-12-02 오후...
파일 폴더
Adobe Acrobat D...
OpenDocument ...
11개 항목

제 2 절 교육 환경 사용 (실험 항목: simple-io-1)

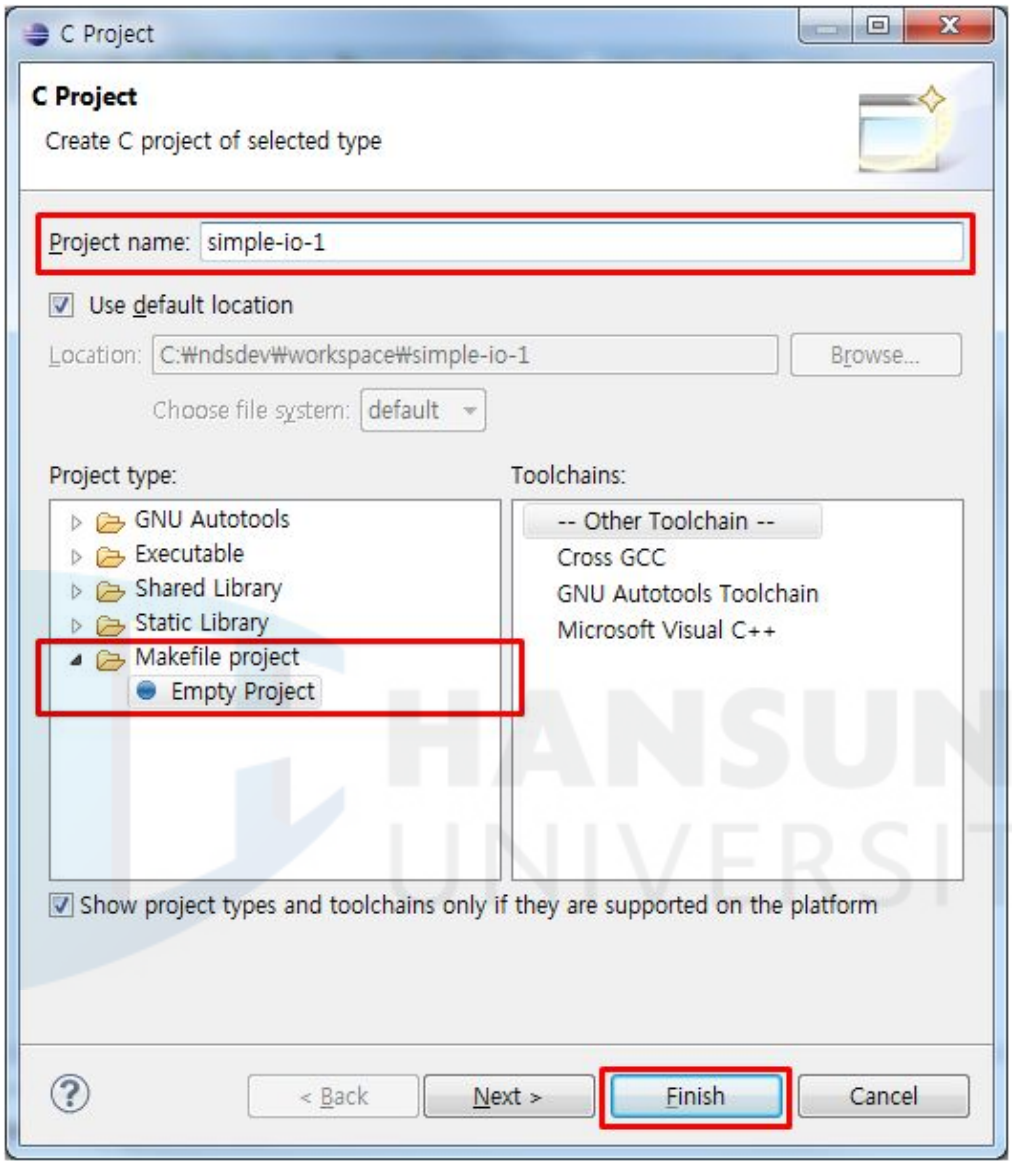
1. 프로젝트 생성

1) Eclipse 실행파일을 실행하고 워크스페이스를 지정하기 위한 창이 나오면 C:\ndsdev\workspace로 프로젝트를 저장할 기본 폴더를 지정한 뒤, OK 버튼을 눌러 Eclipse를 실행한다.

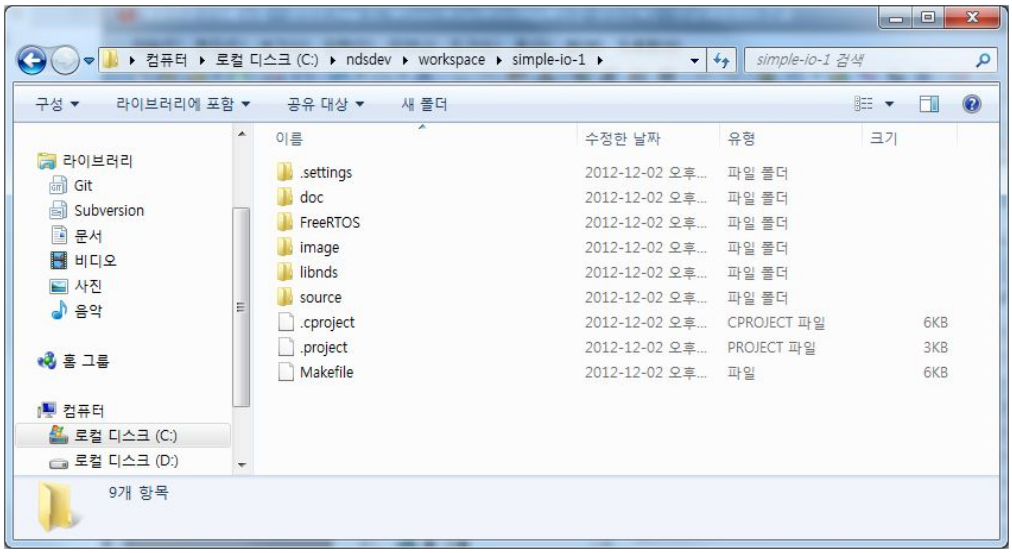
2) File → New → C Project 메뉴를 선택한다.



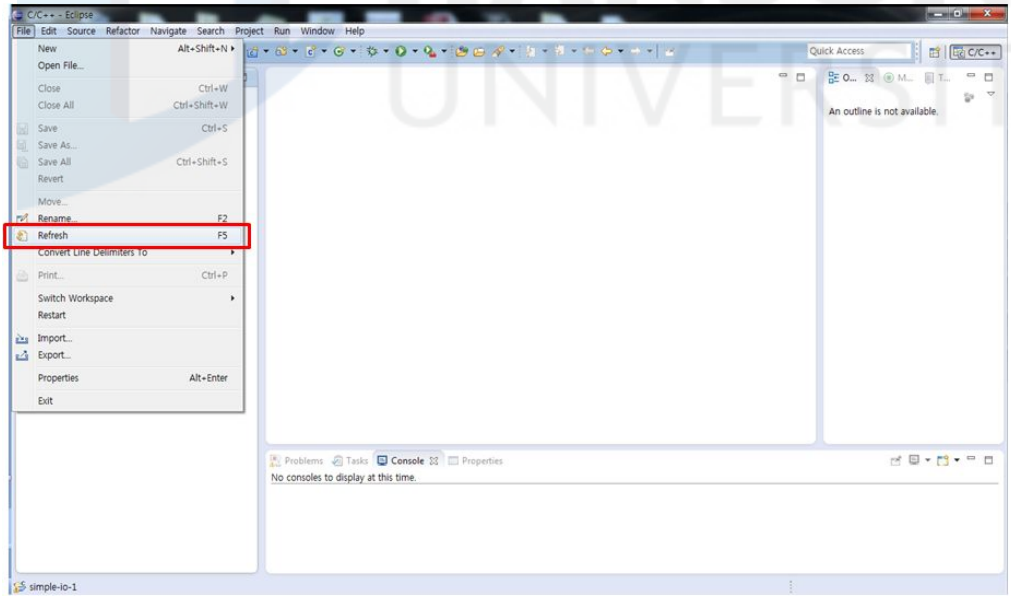
3) 프로젝트 이름, 프로젝트 타입을 다음과 같이 설정한 뒤, Finish 버튼을 눌러 프로젝트를 생성한다.

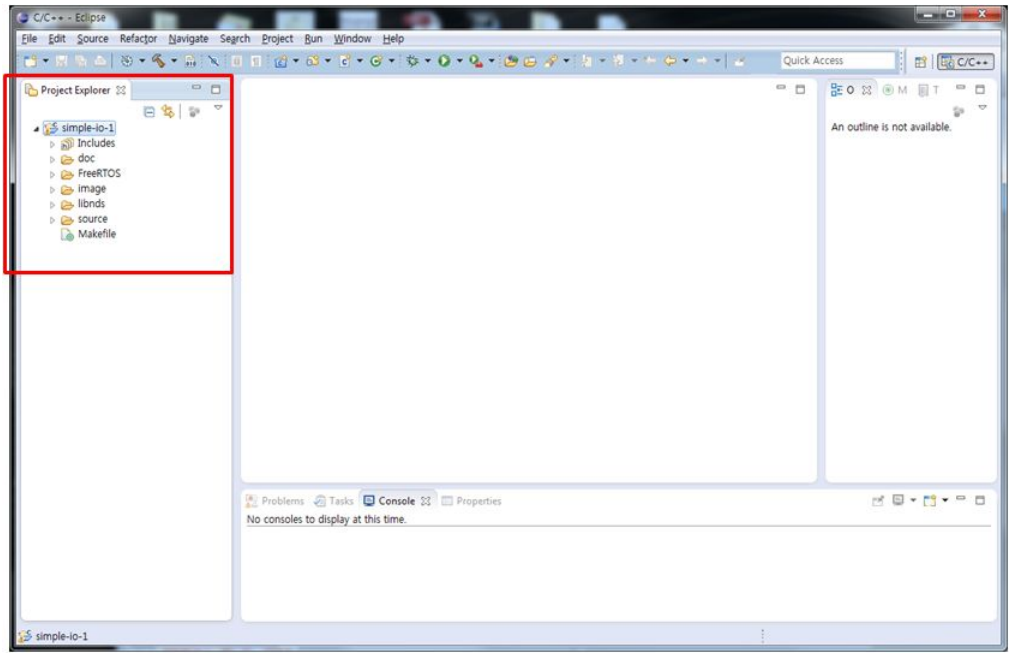


4) Github 홈페이지의 Repository에서 다운로드 받은 실습 소스를 복사한다.



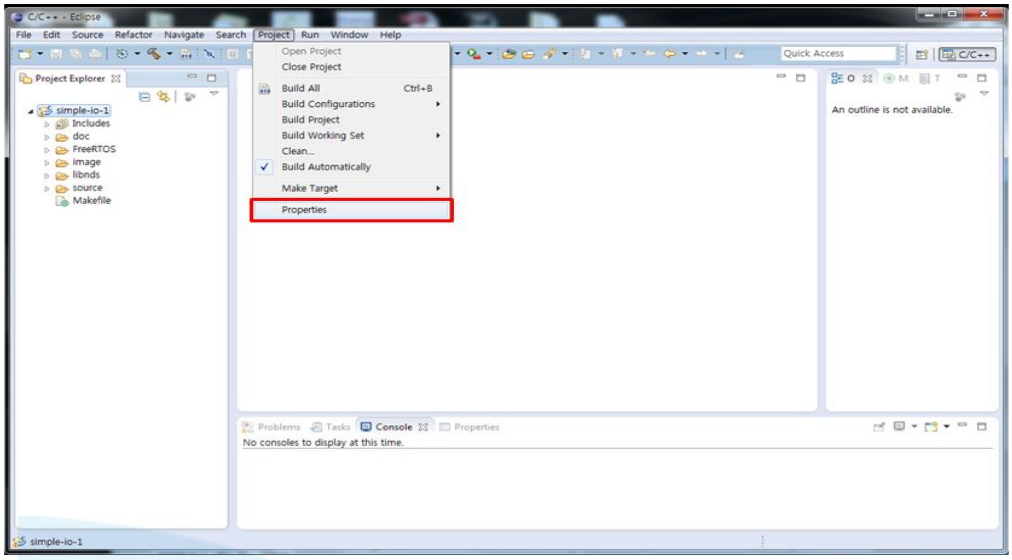
5) File → Refresh 메뉴를 선택하면 Project Explorer에서 프로젝트를 구성하는 파일을 확인할 수 있다.



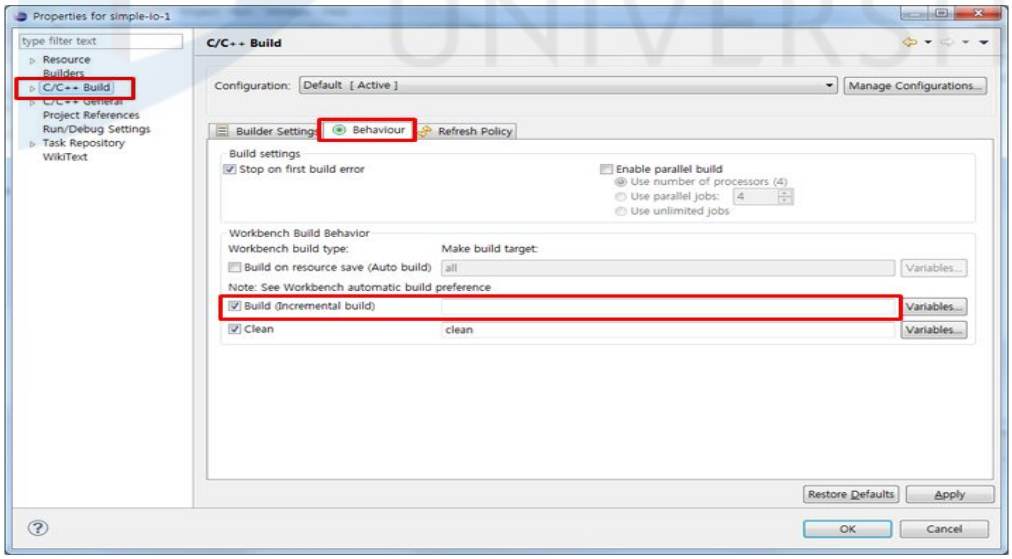


2. 프로젝트 설정

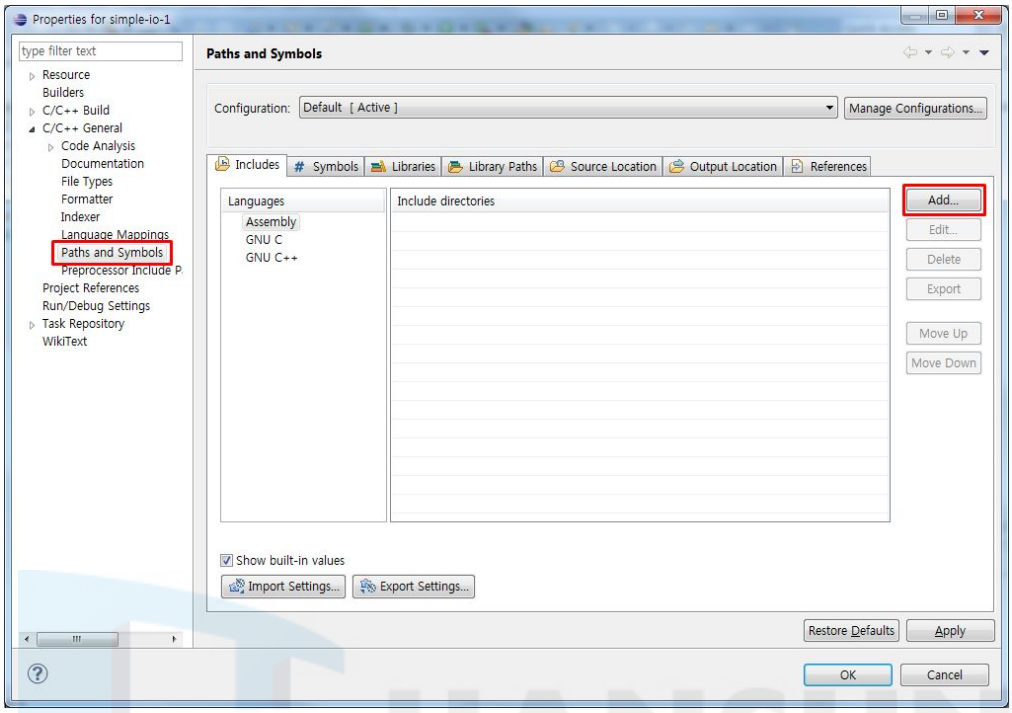
1) Project → Properties 메뉴를 선택한다.



2) C/C++ Build 메뉴의 Behaviour 탭에서 Build(Incremental build) 항목을 다음과 같이 설정한다.



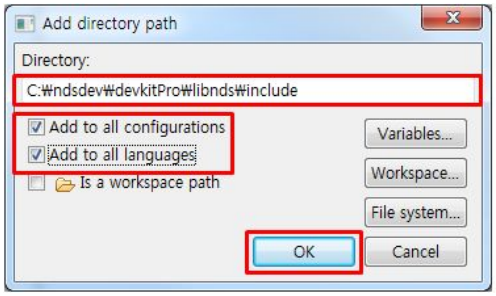
3) C/C++ General → Paths and Symbols 메뉴를 선택한 뒤, Add 버튼을 누른다.

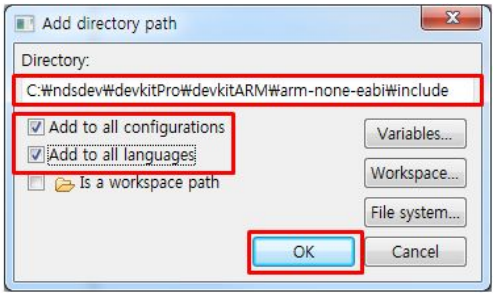


4) 디렉토리 경로를 다음과 같이 추가한다.

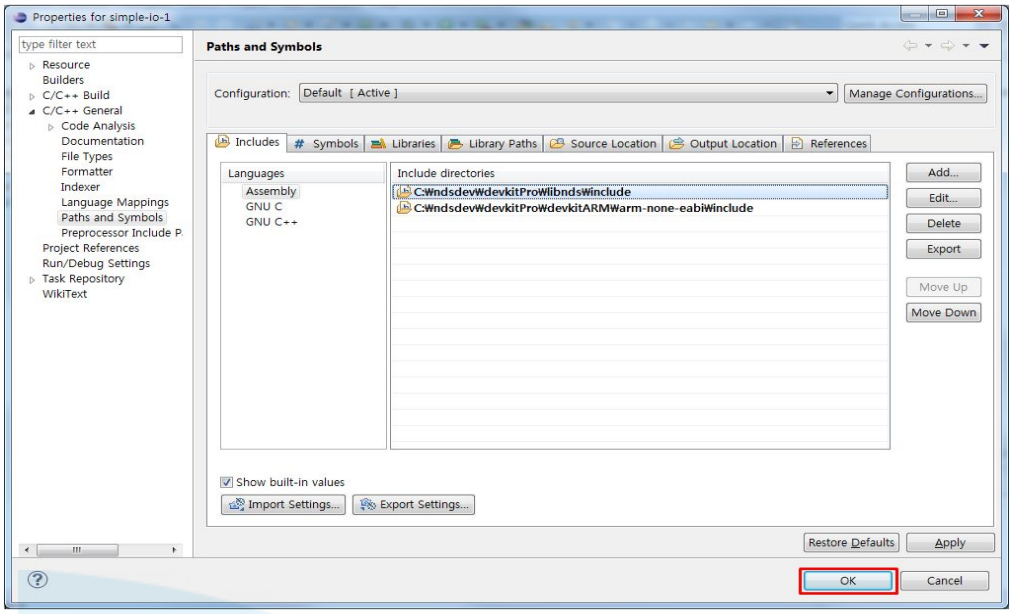
(c:\ndsdev\devkitPro\libnds\include)

(c:\ndsdev\devkitPro\devkitARM\arm-none-eabi\include)

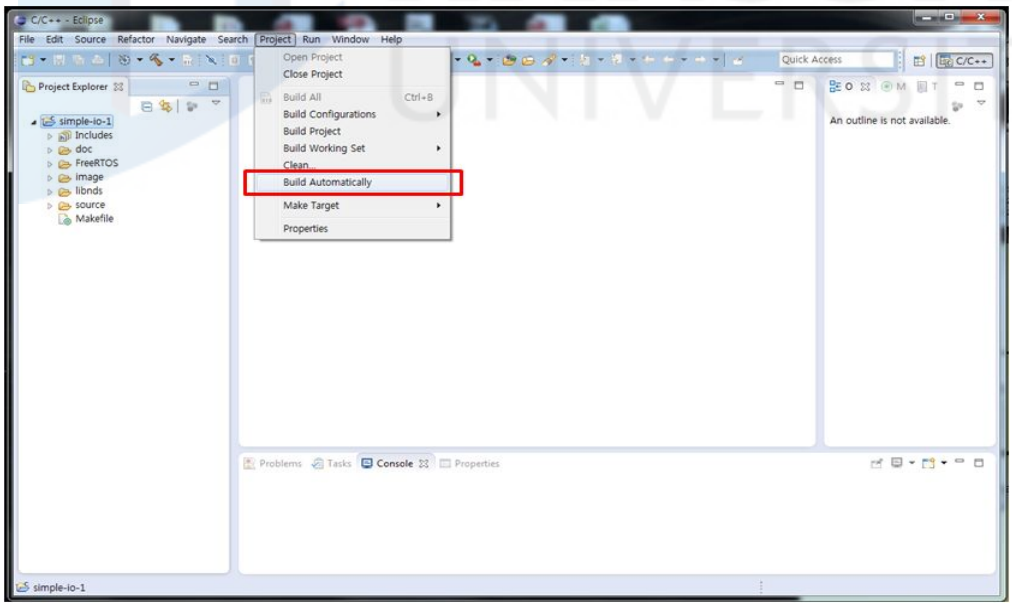




5) 프로젝트 설정이 제대로 되었는지 확인한 뒤, OK 버튼을 눌러 설정을 완료한다.

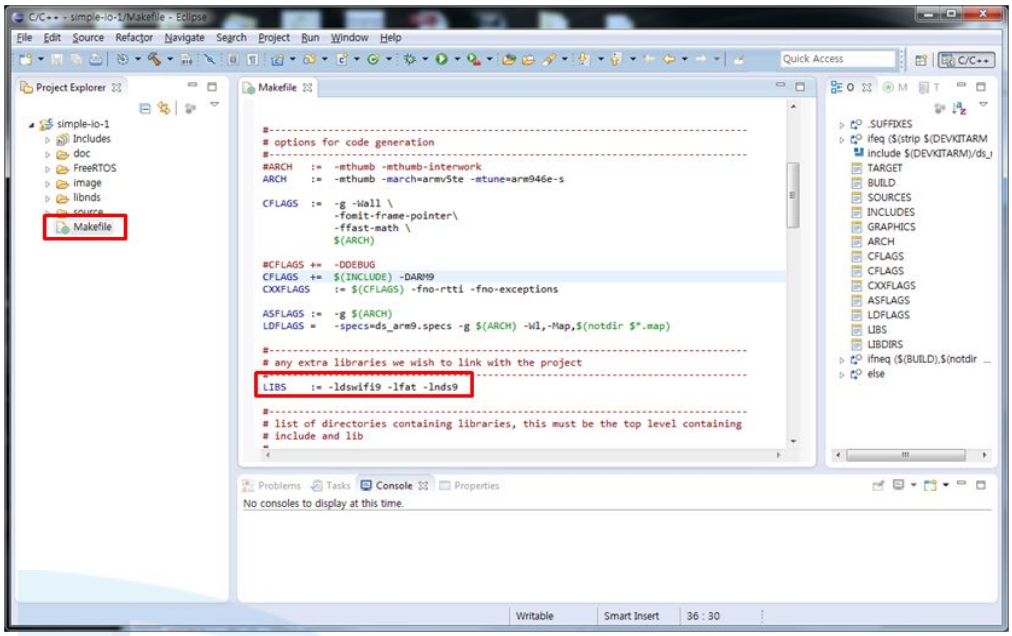


6) Project → Build Automatically 메뉴를 선택하여 해제한다.



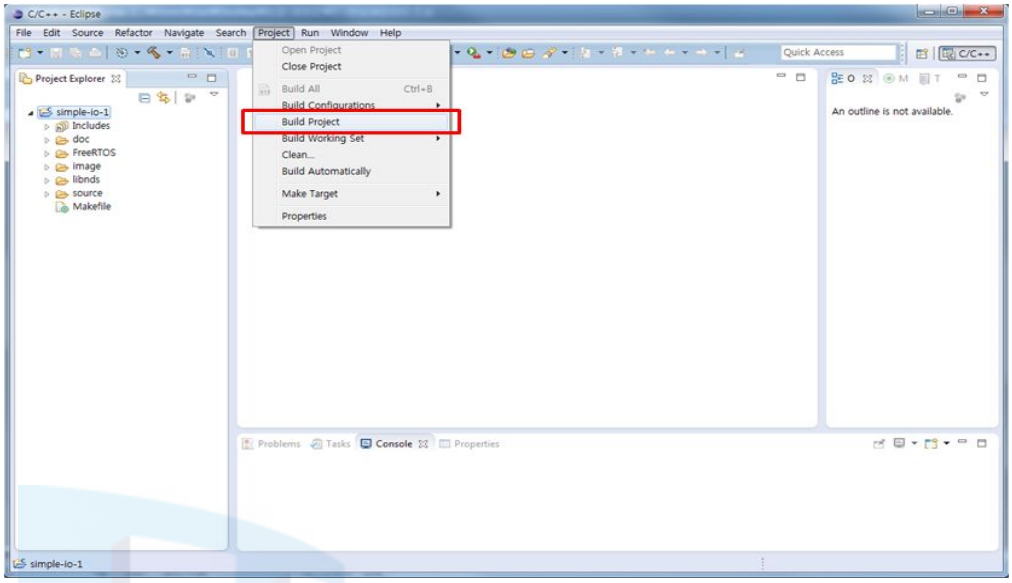
7) Makefile을 열어 LIBS 항목을 다음과 같이 설정한다.

(LIBS := -ldswifi9 -lfat -lnds9)

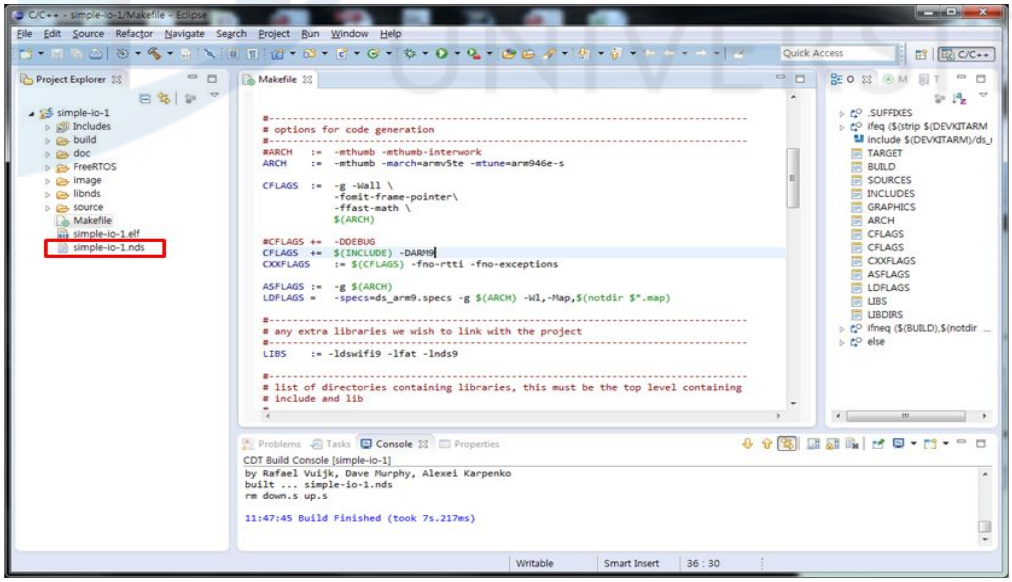


3. 프로젝트 빌드

1) Project → Build Project 메뉴를 선택하거나,  툴바를 이용하여 프로젝트를 빌드 한다.



2) 성공적으로 빌드 되면, simple-io-1.nds 파일이 생성된다.

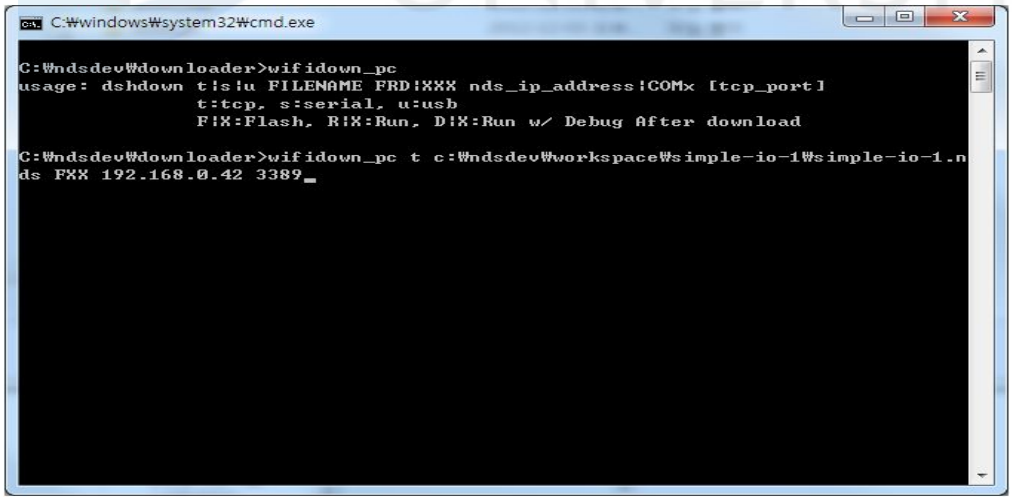


4. 프로그램 다운로드 및 실행

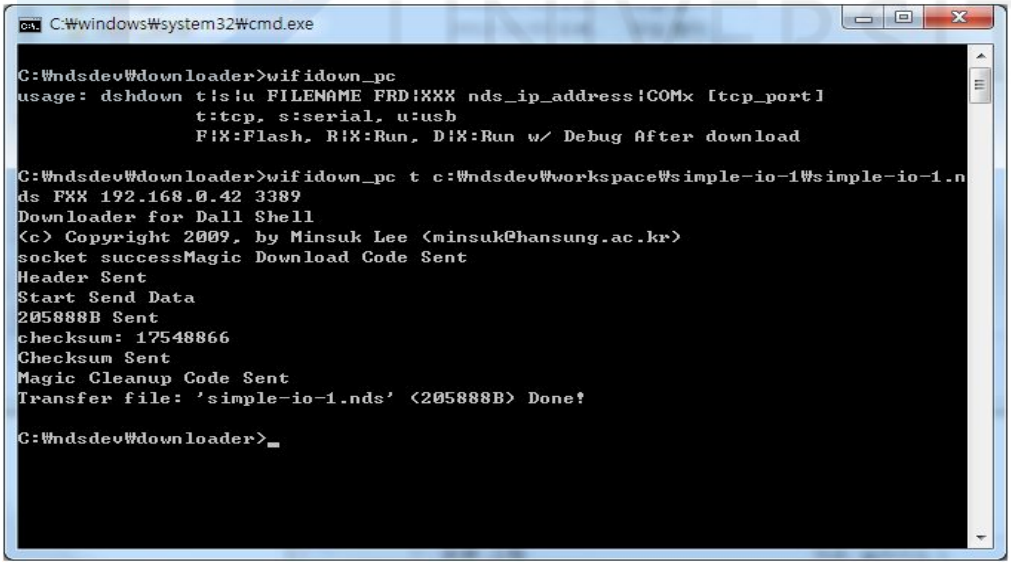
※ 닌텐도 DS, PC 간 무선 랜 연결을 통한 다운로드
(PC에 무선 랜이 설치되어 있는 경우 - 닌텐도 DS와 PC가 같은 AP를 사용하여야 다운로드가 이루어지며, Github 홈페이지의 Repository에서 다운로드 받은 wifidown_pc, wifidown_nds 프로그램을 이용한다.)

1) 닌텐도 DS의 다운로드 프로그램을 실행시킨 뒤, 무선 랜 연결이 올바르게 이루어지고, 다운로드가 가능한 상태인지 확인한다.

2) PC의 다운로드 프로그램의 사용법에 맞게 IP주소, 포트, 파일 경로 등을 올바르게 입력하고 실행시킨다.



3) 다운로드가 완료되면 닌텐도 DS에서는 “Download Successfully Done” 이라는 메시지가 나오고, PC에서는 ”Transfer file - Done!”이라는 메시지가 나온다. (닌텐도 DS는 다운로드가 완료되면 다시 다운로드 대기상태로 돌아간다.)

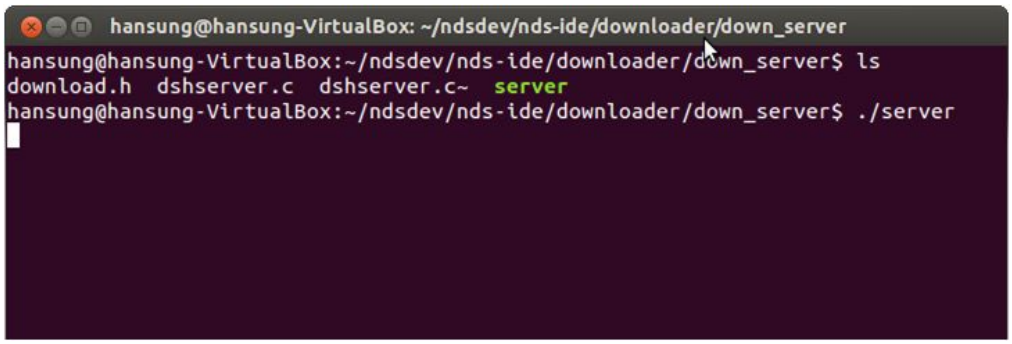


4) 다운로드가 완료되면 닌텐도 DS의 전원을 다시 킨 뒤, 다운로드 파일이 있는지 확인하고 다운로드 된 파일을 실행시킨다.

※ 파일 중계 서버를 통한 다운로드

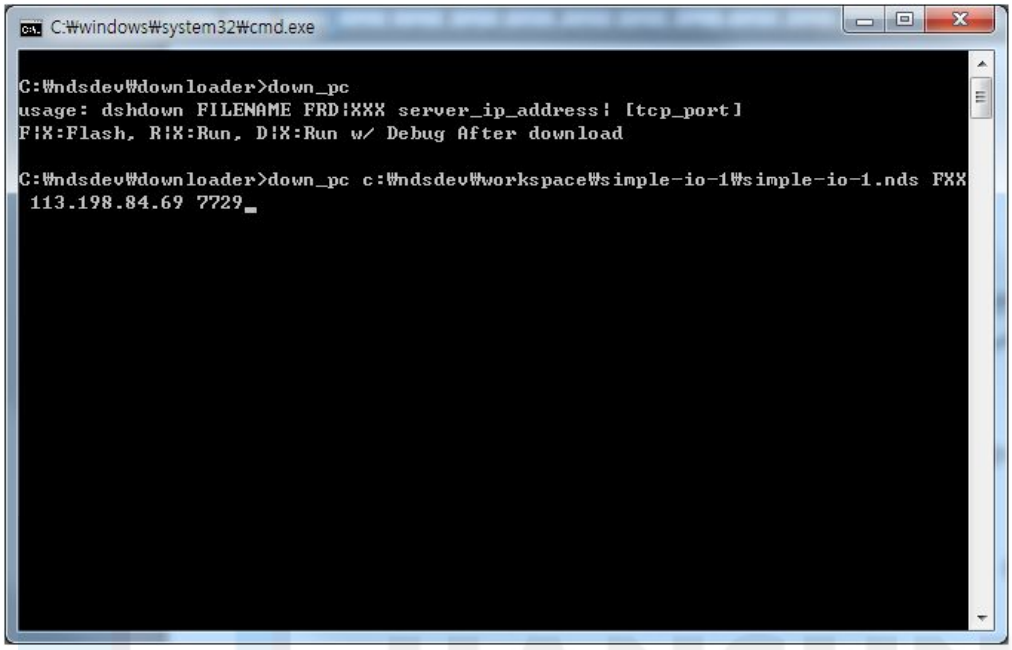
(PC에 무선 랜이 설치되어 있지 않은 경우 - 파일 중계 서버가 실행중인 상태에서 다운로드가 이루어지며, Github 홈페이지의 Repository에서 다운로드 받은 down_pc, down_nds, down_server 프로그램을 이용한다.)

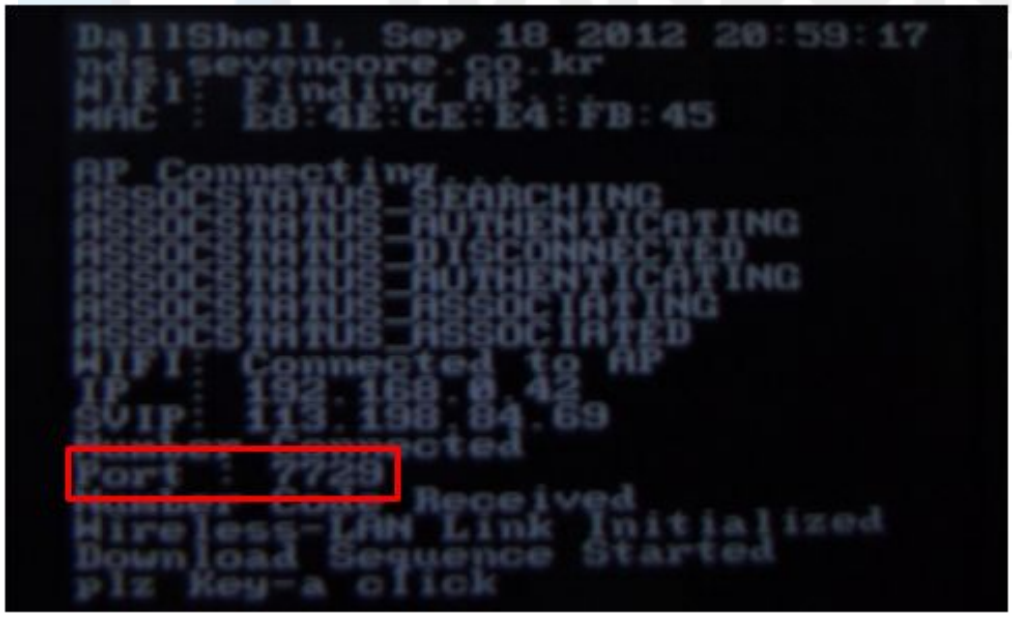
1) 파일 중계 서버 프로그램을 실행시킨다.



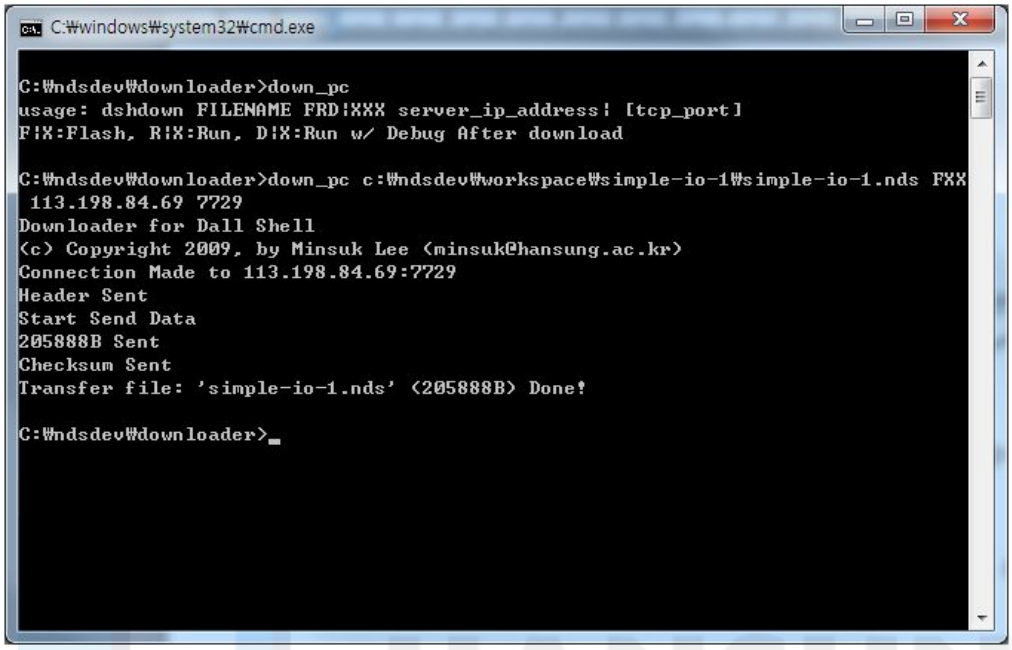
2) 닌텐도 DS의 다운로드 프로그램을 실행시킨 뒤, 무선 랜 연결이 올바르게 이루어지고, 다운로드가 가능한 상태인지 확인한다.

3) PC의 다운로드 프로그램의 사용법에 맞게 IP주소, 포트, 파일 경로 등을 올바르게 입력하고 실행시킨다. (포트는 닌텐도 DS에 표시되는 번호를 사용하여야 한다.)





4) PC에서 파일 중계 서버로의 다운로드가 완료되면 닌텐도 DS의 “A” 키 입력을 통해 PC에서 전송한 파일을 파일 중계 서버로부터 다운로드 한다.



5) 다운로드가 완료되면 닌텐도 DS의 전원을 다시 킨 뒤, 다운로드 파일이 있는지 확인하고 다운로드 된 파일을 실행시킨다.

제 4 장 활용 방안

본 논문에서는 가격이 비교적 저렴한 상용 게임기 하드웨어, 공개 소스 개발도구, 공개 소스 실시간 운영체제 등을 활용하여 마이크로프로세서 응용 또는 임베디드 소프트웨어에 대한 개발, 교육 실습을 수행하기 위한 개발 환경을 구성함으로써, 기존의 임베디드 소프트웨어 교육 장비 또는 환경이 가지고 있던 여러 가지 단점들을 해소하고, 원격 디버깅 등의 기능 부족으로 개발 생산성이 낮았던 기존 상용 게임기 활용 개발 방법을 획기적으로 개선하는 시발점이 되었다.

본 논문의 결과물은 대학, 고등학교, 전문 교육 기관에서 임베디드 시스템, 소프트웨어 관련 교육 실습에 활용될 수 있으며, 개인 임베디드 소프트웨어 개발자들의 개발 도구로 활용될 수 있다. 또한, 각종 임베디드 소프트웨어 관련 연구에 활용될 수 있으며, 실시간 운영체제, 다중 프로세서용 실시간 운영체제 연구 등에 활용될 수 있다.

본 논문의 결과물을 활용하여 얻을 수 있는 구체적인 기대효과는 다음과 같다.

1. 교육에의 활용

- 대학, 전문 대학에서 임베디드 시스템, 소프트웨어 관련 교육 실습에의 활용
- OSEK 표준을 준수하는 실시간 운영체제 교육
- 주로 대학 4학년 대상의 종합 설계 과목 (캡스톤 디자인)에서 도구로 이용
- 개인 임베디드 소프트웨어 개발자들의 개발 도구로 활용
- 타 분야 소프트웨어 개발자들이 임베디드 시스템으로 이직을 할 때, 훈련 도구로 활용
- 각종 임베디드 소프트웨어, 시스템 경진대회에서의 활용

2. 게임 개발에의 활용

- 제공된 개발 도구, SDK를 이용한 게임 개발
- 어댑터 하드웨어를 이용하여 통신, 여러 센서를 활용한 새로운 개념의 게임 개발
- 새로운 입출력 장치를 어댑터 보드에 연결하여 그를 이용하는 게임 개발

3. 상업적 활용

- 자동차 산업을 포함한 임베디드 산업에서의 활용
- 특히 상업적 활용에 있어서는 커뮤니티 개발이 활성화 되어 있는 미국, 유럽, 일본의 경우, 어댑터 보드를 온라인/오프라인으로 수출 판매하는 방식이 상당히 유망할 것으로 기대되며, 외국의 경우 이미 이런 시장이 상당히 활성화 되어 있다. 또, 과제에서 개발되는 어댑터보드의 경우, 교육에 필요한 최소한의 저렴한 하드웨어를 추구하겠지만, 좀 더 많은 장치를 포함하여, 일부 산업용으로도 게임기를 충분히 활용할 수 있는 여지가 있을 것으로 사료된다.

4. 인력 양성에의 기여

- 현재 임베디드 소프트웨어 산업은 산업의 IT 융합화에 따라 많은 개발 인력의 유입을 필요로 하고 있다. 이 과제의 결과물은 기존 개발자를 임베디드 소프트웨어 개발자로 유인하는 훌륭한 도구가 될 수 있으며, 게임기를 가진 많은 학생들에게 임베디드 소프트웨어 또는 적어도 소프트웨어 개발에 대한 중요한 동기를 부여하여 소프트웨어 산업이 인력 양성에 일조할 것으로 기대된다.
- 게임기는 완벽한 제품으로 다른 상용 임베디드 시스템보다는 성능이 전반적으로 우수하며, 다양한 입출력 장치를 가지고 있어, 임베디드 시스템, 임베디드 소프트웨어를 연구하는 그룹이 항상 겪고 있는 프로토타입 구현을 매

우 적은 비용으로 수행할 수 있어서, 연구의 활성화와 인력 양성에 기여할 수 있다.

제 5 장 결 론

본 논문에서는 임베디드 소프트웨어 교육을 위한 실습 환경이 갖추어야 할 여러 가지 요소, 즉 프로그램 다운로드 및 실행, 원격 디버깅이 가능하게 하는 게임기용 소프트웨어와 개발 호스트에서 실행되는 프로젝트 관리, 소스 프로그램 편집, 컴파일, 다운로드 및 원격 디버깅을 통합 처리하는 통합 개발 환경을 구현하였다. 논문에서는 다양한 공개 소스 소프트웨어를 적극적으로 활용하였으며, 부족한 요소들은 역시 공개 소스 소프트웨어 형태로 개발하였다. 또, 교육에 필요한 추가적인 기능을 제공하기 위하여 가상 디바이스, 하드웨어 어댑터도 구현하였다.

논문에서 개발된 임베디드 소프트웨어 교육 환경은 대학이나 학원 등의 교육 기관이나 동호회, 개인들이 현실감 높고 품질이 높은 임베디드 소프트웨어를 대발할 수 있는 환경을 매우 저렴하게 제공하며, 실제 임베디드 시스템 실험 수업에 활용되어 그 효용성이 검증되었다.

앞으로 닌텐도 DS-lite에 이식된 실시간 운영체제를 자동차 산업의 표준인 OSEK/VDX 호환 운영체제로 교체하여, 학생들이 산업체에서 더 많이 사용하는 환경을 경험할 수 있도록 할 예정이며, 더욱 체계적인 임베디드 소프트웨어 교육을 위한 환경을 위해서 추가적인 교육 실습 콘텐츠의 개발 및 본 논문에서 제공하는 환경을 통한 교육 효과에 대한 비교 연구가 필요하다. 또한, 이 환경을 임베디드 운영체제 연구, 로봇 제어와 같은 타 분야에의 응용에 적용하여 저렴한 상용 게임기가 임베디드 소프트웨어 연구, IT 융합 산업에서의 실용적 이용, 임베디드 시스템 교육이라는 다양한 목적으로 보일 수 있는 추가적인 연구를 할 예정이다.

본 논문에서 구현한 임베디드 소프트웨어 교육 환경의 모든 결과물은 Github, Sourceforge 사이트를 통해 공개하여, 누구든 자유롭게 사용, 개발에 참여할 수 있는 기회를 제공하고 있다.

【참고문헌】

1. 국내문헌

Andrew N Sloss et al., 씨랩시스 역, 『ARM System Developer's Guide』, 사이텍미디어, 2005.

박창빈, 김기창, 「ARM 코어 기반 임베디드 시스템을 위한 GDB-stub의 구현」, 한국정보처리학회 추계학술발표대회, 2006

안효복, 『ARM으로 배우는 임베디드 시스템』, 한빛미디어, 2008.

이민석, 「휴대용 게임기를 이용한 공개 SW 기반 임베디드 소프트웨어 교육 IDE 환경 구성」, 대한임베디드공학회논문지, 2012년 4월

장영준 외 4인, 「닌텐도 DS를 이용한 공개 SW 기반 임베디드 소프트웨어 교육 환경 구성」, 대한임베디드공학회 추계학술대회, 2012

_____ 외 5인, 「상용 게임기를 이용한 임베디드 소프트웨어 교육 환경 구성」, 대한임베디드공학회 추계학술대회, 2011

_____, 이민석, 「닌텐도 DS를 이용한 임베디드 소프트웨어 교육」, 정보과학회 한국컴퓨터종합학술대회 논문집, 2012

2. 국외문헌

Hongwei, Li et al., 2009, Research of "Stub" Remote Debugging Technique. Proceedings of 2009 4th International Conference on Computer Science & Education.

Minheng, Tan, 2002, A minimal GDB stub for embedded remote debugging. Columbia University.

Richard, M. Stallman et al., 2002, Debugging with GDB:The GNU Source-Level Debugger. Free Software Foundation.

Stan, Shebs, 1999, An Open Source Debugger for Embedded Development. In Embedded Systems Conference.

【부 록】

FreeRTOS API

FreeRTOS API Index

Task Creation

Task Control

Task Utilities

RTOS Kernel Control

Queues

Semaphore / Mutexes

Software Timers

[NAME]

xTaskHandle

[SYNOPSIS]

#include "task.h"

[DESCRIPTION]

태스크를 참조할 수 있는 자료 형 (태스크는 xTaskHandle을 통해 참조되어진다.)
예를 들어, xTaskCreate() 함수의 호출은 태스크를 삭제하기 위한 vTaskDelete() 함수의 매개변수로 사용되는 xTaskHandle 변수를 반환한다.

[PARAMETERS]

None

[RETURN VALUES]

None

[NAME]

xTaskCreate

[SYNOPSIS]

```
#include "task.h"

portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const portCHAR * const pcName,
                           unsigned portSHORT usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pvCreatedTask );
```

[DESCRIPTION]

새로운 태스크를 생성하고, 실행 준비 된 리스트(Ready List)에 해당 태스크를 추가한다.

FreeRTOS-MPU를 사용중이면 xTaskCreate() 함수보다 xTaskCreateResricted() 함수를 사용하는 것이 바람직하다. FreeRTOS-MPU를 사용하는 환경에서 xTaskCreate() 함수를 사용하는 것은 권한(Privileged) 모드 또는 사용자(User) 모드에서 태스크를 실행할 수 있도록 만들어준다. 권한(Privileged) 모드에서는 태스크가 전체 메모리 영역을 접근할 수 있는 권한을 가지고, 사용자(User) 모드에서는 오직 태스크의 스택에만 접근할 수 있는 권한을 가진다. 이 두 경우 모두 MPU는 표준 FreeRTOS 스택 오버플로우 감지 메커니즘이 사용되고 있을지라도, 스택 오버플로우 문제를 자동적으로 해결해주지는 못한다. xTaskCreateRestricted() 함수는 좀 더 유연성을 제공한다.

[PARAMETERS]

pvTaskCode : 생성할 태스크 함수의 포인터. 태스크 함수는 절대 리턴 되지 않도록 구현되어야 한다. (무한루프)

pcName : 태스크를 설명하는 이름
디버깅 기능을 위해서만 사용되며, 태스크 이름의 최대 길이는 configMAX_TASK_NAME_LEN를 통해 정의된다.

usStackDepth : 태스크의 스택 크기로 바이트 단위가 아닌 스택을 정의하는 자료형 크기의 개수로써 구체화된다. 예를 들어, 스택이 16비트 크기(Width)이고, usStackDepth가 100으로 정의되어 있다면, 스택을 위한 공간으로 200바이트가 할당된다. 스택의 크기(Width)와 깊이(Depth)를 곱한 값(스택이 가지는 총 크기)은 size_t 자료형 변수가 포함할 수 있는 최대값을 초과할 수 없다. (예. int stack[100]의 경우 스택의 크기는 스택의 자료형(int)

의 크기를 의미하고, 스택의 깊이는 100을 의미한다.)

pvParameters : 태스크가 생성될 때 전달할 수 있는 매개변수의 포인터

uxPriority : 태스크의 우선순위
MPU가 제공되는 시스템에서는 권한(Privileged) 모드에서 우선순위 매개변수로 portPRIVILEGE_BIT를 설정함으로써 태스크를 생성할 수 있다. 예를 들어, 우선순위 2인 권한을 갖는 태스크를 생성하기 위해서는 uxPriority 매개 변수를 (2 |portPRIVILEGE_BIT)로 설정해야 한다.

pvCreatedTask : 생성된 태스크를 참조하기 위해 사용되어지는 태스크 핸들

[RETURN VALUES]

태스크가 성공적으로 생성되었을 경우 pdPASS, 태스크를 생성할 충분한 메모리 공간이 없을 경우 errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY를 반환한다.

[NAME]

vTaskDelete

[SYNOPSIS]

#include "task.h"

void vTaskDelete(xTaskHandle pxTask);

[DESCRIPTION]

vTaskDelete() 함수를 사용하기 위해서는 INCLUDE_vTaskDelete를 반드시 1로 설정해야 한다.

실시간 운영체제 커널의 관리 대상에서 제외된다. 해당 태스크는 준비(Ready), 대기(Blocked), 중지(Suspended), 이벤트(Event) 리스트에서 삭제된다.

NOTE: IDLE 태스크는 삭제된 태스크에 할당된 메모리 해제를 담당한다. 그러므로 응용 프로그램 내에서 vTaskDelete() 함수를 호출한다면 마이크로 컨트롤러 수행 시간에 IDLE 태스크가 기아 현상을 겪지 않도록 하는 것이 중요하다. 태스크 코드 내에서 할당되었던 메모리는 자동적으로 해제되지 않으므로 태스크가 삭제되기 전에 해제해야 한다.

[PARAMETERS]

pxTask : 삭제할 태스크의 핸들
NULL로 사용 될 경우 vTaskDelete() 함수를 호출한 태스크가 삭제된다.

[RETURN VALUES]

None

[NAME]

vTaskDelay

[SYNOPSIS]

#include "task.h"

void vTaskDelay(portTickType xTicksToDelay);

[DESCRIPTION]

vTaskDelay() 함수를 사용하기 위해서는 INCLUDE_vTaskDelay를 반드시 1로 설정해야 한다.

주어진 시간의 틱만큼 태스크를 지연시킨다. 태스크가 대기 상태로 지연되는 실제 시간은 틱 발생빈도(Tick Rate)에 의존적이다. portTICK_RATE_MS는 틱 발생 빈도로부터 실제 시간을 계산하는데 사용된다. (한 틱 주기를 알기 위해서)

vTaskDelay() 함수를 통해 태스크가 대기 상태로 지연되는 시간은 vTaskDelay()함수를 호출한 시점과 밀접한 관계를 갖고 있다. 예를 들어, 지연시간으로 100틱을 지정하는 것은 태스크가 vTaskDelay() 함수가 호출된 시점으로부터 100틱 이후에 재개하도록 한다. vTaskDelay() 함수는 코드, 다른 태스크, 인터럽트는 vTaskDelay() 함수 호출을 통한 주기에 영향을 미치고, 이로 인해 다음 번 실행 시간에도 영향을 끼치기 때문에 반복적인 태스크의 주기를 결정하는 좋은 방법을 제공하지 못한다. 고정된 실행 주기를 활용하기 위해서는 vTaskDelayUntil() 함수를 사용하도록 한다. vTaskDelayUntil() 함수는 상대 시간이 아닌 절대 시간을 사용함으로써 태스크의 주기를 보장한다.

[PARAMETERS]

xTicksToDelay : 생성할 태스크 함수의 포인터

[RETURN VALUES]

None

[NAME]

vTaskDelayUntil

[SYNOPSIS]

```
#include "task.h"

void vTaskDelayUntil( portTickType *pxPreviousWakeTime,
                      portTickType xTimeIncrement );
```

[DESCRIPTION]

vTaskDelay() 함수를 사용하기 위해서는 INCLUDE_vTaskDelay를 반드시 1로 설정해야 한다.

주어진 시간의 틱 만큼 태스크를 지연시킨다. vTaskDelayUntil() 함수는 반복적으로 실행되는 태스크의 실행 주기를 보장하기 위해 사용된다.

vTaskDelayUntil() 함수는 vTaskDelay() 함수와 한 가지 중요한 차이점이 있다. vTaskDelay() 함수는 태스크를 지연시키기 위해 vTaskDelay() 함수가 호출된 시점부터 상대적인 시간을 지정하도록 하는 반면에, vTaskDelayUntil() 함수는 태스크를 지연시키기 위해 절대적인 시간을 지정한다.

vTaskDelay() 함수는 vTaskDelay() 함수가 호출된 시점으로부터 지정된 틱 수만큼 태스크를 지연시킨다. 그러므로 vTaskDelay() 함수를 통해 시간적으로 고정된 실행 주기를 생성하는 것은 어렵다.

vTaskDelay() 함수는 태스크가 재개될 시간을 함수가 호출된 시점에 상대적인 시간으로 지정하는 반면, vTaskDelayUntil() 함수는 재개시키고자 하는 시간을 절대적인 시간으로 지정한다.

vTaskDelayUntil() 함수는 재개시키려는 시간이 이미 지나간 시간이라면 즉시 리턴 된다. 그러므로 주기적인 실행이 특정 이유(예를 들어, 태스크가 일시적으로 중지된 상태로 존재)로 인해 중단되었다면, 태스크가 주기적인 실행을 하지 못하게 될 수도 있기 때문에 vTaskDelayUntil() 함수를 사용하여 태스크가 주기적으로 실행되도록 하기 위해서는 재개시키려는 시간을 다시 계산해야 한다. 이러한 문제는 현재의 틱 카운트 값이 아닌 pxPreviousWakeTime 매개변수로 전달된 변수의 값을 확인함으로써 감지될 수 있다.

portTICK_RATE_MS는 틱 발생 빈도로부터 실제 시간을 계산하는데 사용된다. (한 틱 주기를 알기 위해서)

vTaskDelayUntil() 함수는 vTaskSuspendAll() 함수를 통해 스케줄러가 중지되어 있는 동안 호출되어서는 안 된다.

[PARAMETERS]

xTicksToDelay : 생성할 태스크 함수의 포인터

[RETURN VALUES]

None

[NAME]

uxTaskPriorityGet

[SYNOPSIS]

#include "task.h"

unsigned portBASE_TYPE uxTaskPriorityGet(xTaskHandle pxTask);

[DESCRIPTION]

uxTaskPriorityGet() 함수를 사용하기 위해서는 INCLUDE_vTaskPriorityGet을 반드시 1로 설정해야 한다.

특정 태스크의 우선순위를 반환한다.

[PARAMETERS]

pxTask	:	우선순위를 확인할 태스크의 핸들. pxTask를 NULL로 사용하는 것은 uxTaskPriorityGet() 함수를 호출한 태스크의 우선순위를 반환하는 결과를 얻는다.

[RETURN VALUES]

pxTask의 우선순위

[NAME]

vTaskPrioritySet

[SYNOPSIS]

#include "task.h"

void vTaskPrioritySet(xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority);

[DESCRIPTION]

vTaskPrioritySet() 함수를 사용하기 위해서는 INCLUDE_vTaskPrioritySet을 반드시 1로 설정해야 한다.

특정 태스크의 우선순위를 설정한다.

만약 우선순위가 현재 실행중인 태스크의 우선순위보다 높은 우선순위로 설정된다면, vTaskPrioritySet() 함수가 리턴되기 전에 문맥 교환이 실행되게 된다.

[PARAMETERS]

pxTask	:	우선순위를 설정할 태스크의 핸들. pxTask를 NULL로 사용하는 것은 vTaskPrioritySet() 함수를 호출한 태스크의 우선순위를 설정하는 결과를 얻는다.
uxNewPriority	:	태스크에 설정할 우선순위

[RETURN VALUES]

None

[NAME]

vTaskSuspend

[SYNOPSIS]

#include "task.h"

void vTaskSuspend(xTaskHandle pxTaskToSuspend);

[DESCRIPTION]

vTaskSuspend() 함수를 사용하기 위해서는 INCLUDE_vTaskSuspend을 반드시 1로 설정해야 한다.

특정 태스크를 중지시킨다. 태스크가 중지되었을 때에는 우선순위에 상관없이 마이크로 컨트롤러 처리(수행) 시간을 얻지 못한다. (즉, 실행되지 못한다.)

vTaskSuspend() 함수의 호출은 누적되지 않는다. 즉, 같은 태스크 내에서 vTaskSuspend() 함수를 두 번 호출할지라도, 중지된 태스크를 준비(Ready) 상태로 만들기 위해서는 vTaskResume() 함수를 한 번만 호출하면 된다.

[PARAMETERS]

pxTaskToSuspend : 중지하게 될 태스크의 핸들. pxTaskToSuspend를 NULL로 사용하는 것은 vTaskSuspend() 함수를 호출한 태스크를 중지시키는 결과를 얻는다.

[RETURN VALUES]

None

[NAME]

vTaskResume

[SYNOPSIS]

#include "task.h"

void vTaskResume(xTaskHandle pxTaskToResume);

[DESCRIPTION]

vTaskResume() 함수를 사용하기 위해서는 INCLUDE_vTaskSuspend를 반드시 1로 설정해야 한다.

중지된 태스크를 재개시킨다.

한 번 이상의 vTaskSuspend() 함수의 호출로 중지된 태스크는 단 한번의 vTaskResume() 함수 호출을 통해 실행 가능하도록 만들 수 있다.

[PARAMETERS]

pxTaskToResume : 준비시키게 될(준비 상태로 바꿀) 태스크의 핸들

[RETURN VALUES]

None

[NAME]

xTaskResumeFromISR

[SYNOPSIS]

#include "task.h"

portBASE_TYPE xTaskResumeFromISR(xTaskHandle pxTaskToResume);

[DESCRIPTION]

xTaskResumeFromISR() 함수를 사용하기 위해서는 INCLUDE_vTaskSuspend, INCLUDE_xTaskResumeFromISR을 반드시 1로 설정해야 한다.

중지된 태스크를 재개시키기 위한 함수로 ISR(인터럽트 서비스 루틴)에서 호출될 수 있다.

한 번 이상의 vTaskSuspend() 함수의 호출로 중지된 태스크는 단 한번의 vTaskResumeFromISR() 함수 호출을 통해 실행 가능하도록 만들 수 있다.

xTaskResumeFromISR() 함수는 태스크가 중지되기 전에 인터럽트가 발생한다면 인터럽트를 놓치게 되기 때문에 태스크를 인터럽트와 동기화하는데 사용되지 않는다.

[PARAMETERS]

pxTaskToResume : 준비시키게 될(준비 상태로 바꿀) 태스크의 핸들

[RETURN VALUES]

만약 태스크를 재개시키는 것이 문맥 교환을 수행해야 하는 결과를 발생시킨다면 pdTRUE, 그렇지 않다면 pdFALSE를 반환한다. 반환 값은 ISR(인터럽트 서비스 루틴) 내에서 문맥 교환이 수행되어야 하는지에 대한 사항을 결정하기 위해 사용된다.

[NAME]

xTaskGetApplicationTaskTag

[SYNOPSIS]

#include "task.h"

pdTASK_HOOK_CODE xTaskGetApplicationTaskTag(xTaskHandle xTask);

[DESCRIPTION]

xTaskGetApplicationTaskTag() 함수를 사용하기 위해서는 configUSE_APPLICATION_TASK_TAG를 반드시 1로 설정해야 한다.

태스크와 관련된 태그 값을 반환한다. 태그 값의 의미와 사용은 응용프로그램을 작성하는 사용자에 의해서 정의되어진다. 실시간 운영체제 커널은 스스로 태그 값에 접근하지 않는다.

[PARAMETERS]

xTask	:	태그 값을 전달받고자 하는 태스크의 핸들. xTask를 NULL로 사용하는 것은 xTaskGetApplicationTaskTag() 함수를 호출한 태스크의 태그 값을 반환하는 결과를 얻는다.

[RETURN VALUES]

태스크의 태그

[NAME]

xTaskGetCurrentTaskHandle

[SYNOPSIS]

#include "task.h"

xTaskHandle xTaskGetCurrentTaskHandle(void);

[DESCRIPTION]

xTaskGetCurrentTaskHandle() 함수를 사용하기 위해서는 INCLUDE_xTaskGetCurrentTaskHandle를 반드시 1로 설정해야 한다.

[PARAMETERS]

None

[RETURN VALUES]

현재 실행중인(함수를 호출한) 태스크의 핸들

[NAME]

xTaskGetIdleTaskHandle

[SYNOPSIS]

#include “task.h”

xTaskHandle xTaskGetIdleTaskHandle(void);

[DESCRIPTION]

xTaskGetCurrentTaskHandle() 함수를 사용하기 위해서는 INCLUDE_xTaskGetIdleTaskHandle를 반드시 1로 설정해야 한다.

[PARAMETERS]

None

[RETURN VALUES]

Idle 태스크의 핸들.
Idle 태스크는 실시간 운영체제의 스케줄러가 시작될 때 자동적으로 생성된다.

[NAME]

uxTaskGetStackHighWaterMark

[SYNOPSIS]

#include "task.h"

unsigned portBASE_TYPE uxTaskGetStackHighWaterMark(xTaskHandle xTask);

[DESCRIPTION]

uxTaskGetStackHighWaterMark() 함수를 사용하기 위해서는
INCLUDE_uxTaskGetStackHighWaterMark를 반드시 1로 설정해야 한다.

스택은 태스크의 실행과 인터럽트 처리에 따라 성장, 축소되면서 태스크에 의해 사용된다. uxTaskGetStackHighWaterMark() 함수는 태스크의 실행이 시작된 이후 사용가능한 스택의 남아있는 공간의 양을 반환한다. 스택의 남아있는 양은 태스크의 스택이 최고(가장 깊은) 값일 때 사용하지 않은 스택의 남아있는 양이며, 스택의 'high water mark'로 참조된다.

[PARAMETERS]

xTask	:	스택의 남아있는 공간을 확인할 태스크의 핸들. xTask를 NULL로 사용하는 것은 uxTaskGetStackHighWaterMark() 함수를 호출한 태스크의 스택의 남아있는 공간을 확인하는 결과를 얻는다.

[RETURN VALUES]

반환하는 값은 워드 크기의 high water mark이다. (예를 들어, 32비트 머신에서 1을 반환하는 것은 스택에 4바이트의 사용하지 않은 공간이 있다는 것을 의미한다.) 만약, 반환 값이 0이라면 스택이 오버플로우 되었을 가능성이 있다는 것을 의미한다. 반환 값이 0에 가깝다면 스택 오버플로우가 발생할 상황이 가까워진다는 것을 의미한다.

[NAME]

pcTaskGetTaskName

[SYNOPSIS]

#include “task.h”

signed char * pcTaskGetTaskName(xTaskHandle xTaskToQuery);

[DESCRIPTION]

pcTaskGetTaskName() 함수를 사용하기 위해서는 INCLUDE_pcTaskGetTaskName를 반드시 1로 설정해야 한다.

[PARAMETERS]

xTaskQuery	:	태스크 이름을 확인할 태스크의 핸들. xTaskQuery를 NULL로 사용하는 것은 pcTaskGetTaskName() 함수를 호출한 태스크의 이름을 확인하는 결과를 얻는다.

[RETURN VALUES]

태스크의 이름을 가리키는 포인터

[NAME]

xTaskGetTickCount

[SYNOPSIS]

#include "task.h"

volatile portTickType xTaskGetTickCount(void);

[DESCRIPTION]

None

[PARAMETERS]

None

[RETURN VALUES]

vTaskStartScheduler() 함수가 호출된 이후의 틱 카운트

[NAME]

xTaskGetTickCountFromISR

[SYNOPSIS]

#include “task.h”

volatile portTickType xTaskGetTickCountFromISR(void);

[DESCRIPTION]

ISR(인터럽트 서비스 루틴)에서 호출될 수 있는 xTaskGetTickCount() 함수

[PARAMETERS]

None

[RETURN VALUES]

vTaskStartScheduler() 함수가 호출된 이후의 틱 카운트

[NAME]

xTaskGetSchedulerState

[SYNOPSIS]

#include "task.h"

unsigned portBASE_TYPE uxTaskGetNumberOfTasks(void);

[DESCRIPTION]

None

[PARAMETERS]

None

[RETURN VALUES]

다음의 상수들 (task.h에 정의되어 있는) 중 하나 : taskSCHEDULER_NOT_STARTED, taskSCHEDULER_RUNNING, taskSCHEDULER_SUSPENDED

[NAME]

uxTaskGetNumberOfTasks

[SYNOPSIS]

#include "task.h"

unsigned portBASE_TYPE uxTaskGetNumberOfTasks(void);

[DESCRIPTION]

None

[PARAMETERS]

None

[RETURN VALUES]

실시간 커널이 현재 관리하고 있는 태스크의 수. 모든 레디, 대기, 그리고 중지된 상태의 태스크를 포함 한다. 태스크가 이미 삭제되었지만 아직 IDLE 태스크에 의해 메모리가 해제되지 않은 태스크 또한 카운트에 포함된다.

[NAME]

vTaskList

[SYNOPSIS]

#include “task.h”

void vTaskList(portCHAR *pcWriteBuffer);

[DESCRIPTION]

vTaskList() 함수를 사용하기 위해서는 configUSE_TRACE_FACILITY, INCLUDE_vTaskDelete, INCLUDE_vTaskSuspend를 모두 반드시 1로 설정해야 한다.

NOTE : vTaskList() 함수는 이 함수가 동작하는 동안 인터럽트를 비활성화 시킨다. 일반적인 응용 프로그램 내에서가 아닌 디버깅을 위한 용도로 사용된다.

모든 현 태스크들의 현재 상태와 스택 사용량을 리스트 화 한다.

태스크는 대기상태('B'), 준비상태('R'), 삭제된 상태('D') 또는 중지 상태('S')로 기록된다.

[PARAMETERS]

pcWriteBuffer	:	위에서 언급한 자세한 내용들은 버퍼에 ASCII 형식으로 쓰여 진다. 버퍼는 생성된 보고서(Report)를 포함하는데 충분히 크다고 가정한다. 대략 한 태스크 당 40 바이트면 충분하다.

[RETURN VALUES]

None

[NAME]

vTaskStartTrace

[SYNOPSIS]

#include "task.h"

void vTaskStartTrace(portCHAR * pcBuffer, unsigned portLONG ulBufferSize);

[DESCRIPTION]

[vTaskStartTrace() 함수는 본래 trace 유틸리티와 연관이 있다. 하지만 사용자들은 사용하기에 더 쉽고 강력한 새로운 Trace Hook Macros를 찾을 수 있을 것이다.]

실시간 커널 활동 trace를 시작한다. Trace는 어떤 태스크가 언제 사용 중인지 기록한다.

Trace 파일은 바이너리 형식으로 저장된다. convtrce.exe를 호출한 독립된 DOS 유틸리티는 trace 파일을 탭 간격을 가진 텍스트 파일로 변환하여 보여 지고 스프레드시트에 기록된다.

[PARAMETERS]

pcBuffer	:	Trace가 기록될 버퍼
ulBufferSize	:	바이트 크기로 pcBuffer의 크기. Trace는 버퍼가 꽉 찬 상태이거나 ulTaskEndTrace() 함수가 호출될 때까지 계속된다.

[RETURN VALUES]

None

[NAME]

ulTaskEndTrace

[SYNOPSIS]

#include “task.h”

unsigned portLONG ulTaskEndTrace(void);

[DESCRIPTION]

[vTaskStartTrace함수는 본래 trace 유틸리티와 연관이 있다. 하지만 사용자들은 사용하기에 더 쉽고 강력한 새로운 Trace Hook Macros를 찾을 수 있을 것이다.]

커널 활동 trace를 중단한다.

[PARAMETERS]

None

[RETURN VALUES]

Trace 버퍼에 기록된 바이트의 수

[NAME]

vTaskGetRunTimeStats

[SYNOPSIS]

```
#include "task.h"

void vTaskGetRunTimeStats( portCHAR *pcWriteBuffer );
```

[DESCRIPTION]

vTaskGetRunTimeStats() 함수를 사용하기 위해서는 configGENERATE_RUN_TIME_STATS를 반드시 1로 정의해야 한다.
응용프로그램에서는 반드시 주변 타이머/카운터를 설정하기 위한 portCONFIGURE_TIMER_FOR_RUN_TIME_STATS(), 타이머의 현재 카운트 값을 반환하는 portGET_RUN_TIME_COUNTER_VALUE를 정의해 주어야한다.

NOTE : vTaskGetRunTimeStats() 함수는 이 함수가 동작하는 동안 인터럽트를 비활성화 시킨다. vTaskGetRunTimeStats() 함수는 일반적인 응용프로그램 내에서가 아닌 디버깅을 위한 용도로 사용 된다.

configGENERATE_RUN_TIME_STATS값을 1로 설정하는 것은 각 태스크에 축적된 총 실행시간이다. 축적된 시간 값의 해결책은 portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() macro에 의해 설정된 타이머의 주기에 의존하는 것이다. vTaskGetRunTimeStats() 함수를 호출하는 것은 각 태스크의 총 실행 시간을 절대 카운트 값과 시스템 총 실행 시간의 퍼센트 비율로써 버퍼에 쓰는 것이다.

[PARAMETERS]

pcWriteBuffer	:	위에 버퍼에 대해 자세히 설명한 것은 아스키 형식으로 쓰여 있다. 버퍼는 생성된 보고서(Report)를 포함하는데 충분히 크다고 가정한다. 대략한 태스크 당 40 bytes면 충분하다.

[RETURN VALUES]

태스크가 성공적으로 생성되었을 경우 pdPASS, 태스크를 생성할 충분한 메모리 공간이 없을 경우 errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY를 반환한다.

[NAME]

vTaskSetApplicationTaskTag

[SYNOPSIS]

```
#include "task.h"

void vTaskSetApplicationTaskTag( xTaskHandle xTask,
                                 pdTASK_HOOK_CODE pxTagValue );
```

[DESCRIPTION]

configUSE_APPLICATION_TASK_TAG는 이 함수를 사용하기 위하여 반드시 1로 정의되어야 한다.

태그 값은 각 태스크에 할당된다. 태그 값은 오직 응용프로그램의 사용을 목적으로 한다. 실시간 운영체제 커널 자체는 어떠한 방법으로도 사용될 수 없다. RTOS trace macros 문서페이지에서는 응용 프로그램이 어떻게 이러한 특징을 사용하는지에 대한 좋은 예를 소개한다.

[PARAMETERS]

xTask	:	태스크에 대한 태그 값의 핸들이 할당된다. xTask가 NULL값을 전달하는 것은 태그 값이 이 함수를 호출한 태스크에 할당된다는 것을 의미한다.
pxTagValue	:	태스크 태그에 할당된 값. 값이 실제로 할당되었음에도 태그로써 할당되도록 함수포인터를 가능하게 하는 pdTASK_HOOK_CODE의 형태이다.

[RETURN VALUES]

None

[NAME]

xTaskCallApplicationTaskHook

[SYNOPSIS]

```
#include "task.h"

portBASE_TYPE xTaskCallApplicationTaskHook( xTaskHandle xTask,
                                            void *pvParameter );
```

[DESCRIPTION]

configUSE_APPLICATION_TASK_TAG는 이 함수를 사용하기 위하여 반드시 1로 정의되어야 한다.

태그 값이 각 태스크에 할당될 수 있다. 일반적으로 태그 값은 오직 응용프로그램에서 사용하며 실시간 운영체제 커널은 태그 값에 접근하지 않는다. 하지만, 태스크에 hook (또는 콜백) 함수 (xTaskCallApplicationTaskHook() 함수를 호출함으로써 실행되는 콜백 함수)를 할당하기 위해 태그를 사용하는 것은 가능하다. 각 태스크는 각 태스크는 콜백 함수를 정의 할 수도, 정의하지 않을 수도 있다.

함수의 첫 번째 매개변수를 사용하여 특정 태스크의 hook 함수를 호출하는 것이 가능 하지만, 보통 태스크의 hook 함수는 trace hook macros를 통해 사용된다.

태스크 hook 함수는 pdTASK_HOOK_CODE형(즉, void * 매개변수)을 가져야 하며, portBASE_TYPE 형 값을 반환한다. void * 매개변수는 hook 함수에 특정 정보를 전달하기 위해 사용된다.

[PARAMETERS]

xTask	:	태스크의 hook 함수가 호출된 핸들. xTask가 NULL값을 전달하는 것은 현재 실행중인 태스크와 연관된 hook 함수를 호출하는 것이다.
pvParameter	:	hook 함수로 전달할 매개변수. 구조체의 포인터 또는 단순 상수 값으로 사용된다.

[RETURN VALUES]

None

[NAME]

taskYIELD

[SYNOPSIS]

#include "task.h "

taskYIELD();

[DESCRIPTION]

문맥 전환을 강제로 발생시키는 매크로 함수

[PARAMETERS]

None

[RETURN VALUES]

None

[NAME]

taskENTER_CRITICAL

[SYNOPSIS]

#include “task.h ”

taskENTER_CRITICAL();

[DESCRIPTION]

임계 구역의 시작 위치에 사용하는 매크로 함수

NOTE: 스택이 변경될 수 있으므로 신중하게 사용해야 한다.

[PARAMETERS]

None

[RETURN VALUES]

None

[NAME]

taskEXIT_CRITICAL

[SYNOPSIS]

#include “task.h ”

taskEXIT_CRITICAL();

[DESCRIPTION]

임계 구역의 끝 위치에 사용하는 매크로 함수.
선점형 문맥 전환은 임계 구역 내에서 발생시킬 수 없다.

NOTE: 스택이 변경될 수 있으므로 신중하게 사용해야 한다.

[PARAMETERS]

None

[RETURN VALUES]

None

[NAME]

taskDISABLE_INTERRUPTS

[SYNOPSIS]

#include "task.h "

taskDISABLE_INTERRUPTS();

[DESCRIPTION]

마스크 가능한 모든 인터럽트를 비활성화(disable) 시키는 매크로 함수

[PARAMETERS]

None

[RETURN VALUES]

None

[NAME]

taskENABLE_INTERRUPTS

[SYNOPSIS]

#include "task.h "

taskENABLE_INTERRUPTS();

[DESCRIPTION]

마이크로 컨트롤러 인터럽트를 활성화(enable) 시키는 매크로 함수

[PARAMETERS]

None

[RETURN VALUES]

None

[NAME]

vTaskStartScheduler

[SYNOPSIS]

#include “task.h ”

void vTaskStartScheduler(void);

[DESCRIPTION]

실시간 커널 틱 발생을 시작시킨다. 커널은 vTaskStartScheduler() 함수가 호출된 이후에 태스크가 실행되도록 제어한다.

Idle 태스크는 vTaskStartScheduler() 함수가 호출될 때 자동적으로 생성된다.

vTaskStartScheduler() 함수가 성공적으로 동작하면 vTaskEndScheduler() 함수가 호출되기 전 까지는 절대 리턴 되지 않는다. Idle 태스크를 생성할 충분한 메모리 공간이 없다면 vTaskStartScheduler() 함수는 (실패하고) 바로 리턴 된다.

demo application 파일의 main.c에서 태스크를 생성하고 커널을 시작시키는 예제를 확인할 수 있다.

[PARAMETERS]

None

[RETURN VALUES]

None

[NAME]

vTaskEndScheduler

[SYNOPSIS]

#include "task.h "

void vTaskEndScheduler(void);

[DESCRIPTION]

실시간 커널 틱 발생을 정지시킨다. 생성된 모든 태스크는 자동적으로 삭제되며 멀티태스킹은 정지된다.

vTaskEndScheduler() 함수를 호출하면 마치 vTaskStartScheduler() 함수가 바로 리턴 한 것처럼 vTaskStartScheduler() 함수가 호출된 곳부터 실행은 재개된다.

vTaskEndScheduler() 함수를 사용하는 예제는 demo/PC 디렉터리에 있는 demo application의 main.c에서 확인할 수 있다.

vTaskEndScheduler() 함수는 종료 함수를 필요로 하기 때문에 이식 계층(portable layer)에서 정의되어야 한다. (PC port의 port.c에 있는 vPortEndScheduler() 함수를 통해 확인할 수 있다.)
커널 틱 발생을 정지시키는 것과 같은 하드웨어의 특정 동작을 수행한다.

vTaskEndScheduler() 함수는 커널에 의해 할당되어진 모든 자원을 할당 해제 시킨다. 하지만, 해제된 자원들은 어플리케이션 태스크에 의해 할당되어지지 않는다.

[PARAMETERS]

None

[RETURN VALUES]

None

[NAME]

vTaskSuspendAll

[SYNOPSIS]

#include “task.h ”

void vTaskSuspendAll(void);

[DESCRIPTION]

커널 틱을 포함하여 인터럽트를 enable 상태로 유지하면서 모든 실시간 커널 활동을 중지시킨다.

vTaskSuspendAll() 함수를 호출한 후에는 xTaskResumeAll() 함수가 호출되기 전까지 태스크가 swapped out될 위험 없이 실행을 유지시킬 수 있다.

vTaskDelayUntil(), xQueueSend() 와 같은 API들은 문맥 전환을 발생시킬 수 있으므로 스케줄러가 정지된 동안에는 호출하면 안 된다.

[PARAMETERS]

None

[RETURN VALUES]

None

[NAME]

xTaskResumeAll

[SYNOPSIS]

#include “task.h ”

void xTaskResumeAll(void);

[DESCRIPTION]

xTaskSuspendAll() 함수가 호출 된 이후의 실시간 커널 활동을 재개시킨다.
vTaskSuspendAll() 함수가 호출된 후에 커널은 실행 중이었던 태스크의 제어권을 가져올 수 있다.

[PARAMETERS]

None

[RETURN VALUES]

None

[NAME]

uxQueueMessagesWaiting

[SYNOPSIS]

```
#include "queue.h"

unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

[DESCRIPTION]

큐에 저장되어 있는 메시지의 수를 반환한다.

[PARAMETERS]

xQueue : 메시지의 수를 확인할 큐의 핸들

[RETURN VALUES]

큐에 저장되어 있는 메시지의 수

[NAME]

uxQueueMessagesWaitingFromISR

[SYNOPSIS]

#include "queue.h"

unsigned portBASE_TYPE uxQueueMessagesWaitingFromISR(xQueueHandle xQueue);

[DESCRIPTION]

ISR(인터럽트 서비스 루틴)에서 호출될 수 있는 uxQueueMessageWaiting() 함수.
큐에 저장되어 있는 메시지의 수를 반환한다.

[PARAMETERS]

xQueue : 메시지의 수를 확인할 큐의 핸들

[RETURN VALUES]

큐에 저장되어 있는 메시지의 수

[NAME]

xQueueCreate

[SYNOPSIS]

```
#include "queue. h "

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

[DESCRIPTION]

새로운 큐 인스턴스를 생성한다. 생성될 큐를 위해 요구되는 저장 공간이 할당되고 큐를 위한 핸들이 리턴 된다.

[PARAMETERS]

uxQueueLength	:	큐가 포함할 수 있는 아이템의 최대 수
uxItemSize :		큐에 저장할 아이템의 크기. 아이템은 큐에 참조되는 것이 아니라 복사되기 때문에 uxItemSize는 큐에 전달되는 아이템을 복사하기 위한 크기를 의미한다.

[RETURN VALUES]

큐가 성공적으로 생성되면 새로 생성된 큐의 핸들이 리턴 되며, 큐를 만들 수 없을 때 0이 반환된다.

[NAME]

vQueueDelete

[SYNOPSIS]

```
#include "queue.h"

void vQueueDelete( xQueueHandle xQueue );
```

[DESCRIPTION]

큐를 삭제한다. 큐에 아이템을 저장하기 위해 할당한 메모리를 해제한다.

[PARAMETERS]

xQueue : 삭제할 큐의 핸들

[RETURN VALUES]

None

[NAME]

xQueueReset

[SYNOPSIS]

```
#include "queue.h"

portBASE_TYPE xQueueReset( xQueueHandle xQueue );
```

[DESCRIPTION]

큐를 초기 상태로 초기화한다.

[PARAMETERS]

xQueue : 초기화할 큐의 핸들

[RETURN VALUES]

큐가 성공적으로 초기화되었을 경우 pdPASS, 큐에 아이템을 추가하거나 큐로부터 아이템을 전달 받기 위해 대기중인 태스크가 있어 초기화할 수 없는 경우 pdFALSE를 반환한다.

[NAME]

xQueueSend

[SYNOPSIS]

```
#include "queue. h "

portBASE_TYPE xQueueSend( xQueueHandle xQueue,
                          const void * pvItemToQueue,
                          portTickType xTicksToWait );
```

[DESCRIPTION]

xQueueGenericSend() 함수를 호출하는 매크로 함수이다.
xQueueGenericSend() 매크로는 xQueueSendToFront(), xQueueSendToBack() 매크로를 포함하지 않는 버전의 FreeRTOS와의 호환성을 위해 포함되어 있다.

큐에 아이템을 추가한다. 아이템은 큐에 참조되는 것이 아니라 복사된다. 이 함수는 인터럽트 서비스 루틴에서 호출될 수 없다. ISR(인터럽트 서비스 루틴)에서 큐에 아이템을 추가하기 위해서는 xQueueSendFromISR() 함수를 사용해야 한다.

xQueueSend() 함수는 태스크 간 통신 API중 fully featured API에 속하며, xQueueAltSend() 함수는 동일한 기능을 하는alternative API이다. 두 버전 모두 동일한 매개 변수와 동일한 반환 값을 가진다.

[PARAMETERS]

xQueue	:	아이템을 추가할 큐의 핸들
prvItemToQueue	:	큐에 추가하려고 하는 아이템의 포인터. 아이템의 크기는 큐가 생성될 때 정의된 크기이며, prvItemToQueue가 가리키는 영역에서부터 이 길이만큼의 바이트 영역이 큐의 저장 영역으로 복사된다.
xTicksToWait	:	이미 꽉 찬 큐에 아이템을 추가할 수 있게 되기까지 대기하는 최대 시간. 만약 큐가 꽉 차 있는 상태에서 xTicksToWait가 0으로 설정되어 있다면, 즉시 리턴 된다. 틱 주기로 정의된 시간은 portTICK_RATE_MS를 통해 실제 시간으로 바뀌게 된다. 만약 INCLUDE_vTaskSuspend가 1로 설정되어 있다면, portMAX_DELAY를 대기 시간으로 사용하는 것은 태스크가 무기한으로 대기하게 만들 수 있다.

[RETURN VALUES]

성공적으로 아이템을 추가했다면 pdTRUE, 그렇지 않다면 errQUEUE_FULL이 반환된다.

[NAME]

xQueueSendToBack

[SYNOPSIS]

```
#include "queue. h "

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

[DESCRIPTION]

xQueueGenericSend() 함수를 호출하는 매크로 함수이다. xQueueSend() 함수와 동일한 기능을 한다.

큐에(큐의 뒤부터) 아이템을 추가한다. 아이템은 큐에 참조되는 것이 아니라 복사된다. 이 함수는 인터럽트 서비스 루틴에서 호출될 수 없다. ISR(인터럽트 서비스 루틴)에서 큐에 아이템을 추가하기 위해서는 xQueueSendToBackFromISR() 함수를 사용해야 한다.

xQueueSendToBack() 함수는 태스크 간 통신 API중 fully featured API에 속하며, xQueueAltSendToBack() 함수는 동일한 기능을 하는 alternative API이다. 두 버전 모두 동일한 매개 변수와 동일한 반환 값을 가진다.

[PARAMETERS]

xQueue	:	아이템을 추가할 큐의 핸들
prvItemToQueue	:	큐에 추가하려고 하는 아이템의 포인터. 아이템의 크기는 큐가 생성될 때 정의된 크기이며, prvItemToQueue가 가리키는 영역에서부터 이 길이만큼의 바이트 영역이 큐의 저장 영역으로 복사된다.
xTicksToWait	:	이미 꽉 찬 큐에 아이템을 추가할 수 있게 되기까지 대기하는 최대 시간. 만약 큐가 꽉 차 있는 상태에서 xTicksToWait가 0으로 설정되어 있다면, 즉시 리턴 된다. 틱 주기로 정의된 시간은 portTICK_RATE_MS를 통해 실제 시간으로 바뀌게 된다. 만약 INCLUDE_vTaskSuspend가 1로 설정되어 있다면, portMAX_DELAY를 대기 시간으로 사용하는 것은 태스크가 무기한으로 대기하게 만들 수 있다.

[RETURN VALUES]

성공적으로 아이템을 추가했다면 pdTRUE, 그렇지 않다면 errQUEUE_FULL이 반환된다.

[NAME]

xQueueSendToFront

[SYNOPSIS]

```
#include "queue. h "

portBASE_TYPE xQueueSendToToFront( xQueueHandle xQueue,
                                   const void * pvItemToQueue,
                                   portTickType xTicksToWait );
```

[DESCRIPTION]

xQueueSendToFront() 함수는 xQueueGenericSend() 함수를 호출하는 매크로 함수이다.

큐에(큐의 처음부터) 아이템을 추가한다. 아이템은 큐에 참조되는 것이 아니라 복사된다. 이 함수는 인터럽트 서비스 루틴에서 호출될 수 없다. ISR(인터럽트 서비스 루틴)에서 큐에 아이템을 추가하기 위해서는 xQueueSendToBackFromISR() 함수를 사용해야 한다.

xQueueSendToFront() 함수는 태스크 간 통신 API중 fully featured API에 속하며, xQueueAltSendToFront() 함수는 동일한 기능을 하는 alternative API이다. 두 버전 모두 동일한 매개 변수와 동일한 반환 값을 가진다.

[PARAMETERS]

xQueue	:	아이템을 추가할 큐의 핸들
prvItemToQueue	:	큐에 추가하려고 하는 아이템의 포인터. 아이템의 크기는 큐가 생성될 때 정의된 크기이며, prvItemToQueue가 가리키는 영역에서부터 이 길이만큼의 바이트 영역이 큐의 저장 영역으로 복사된다.
xTicksToWait	:	이미 꽉 찬 큐에 아이템을 추가할 수 있게 되기까지 대기하는 최대 시간. 만약 큐가 꽉 차 있는 상태에서 xTicksToWait가 0으로 설정되어 있다면, 즉시 리턴 된다. 틱 주기로 정의된 시간은 portTICK_RATE_MS를 통해 실제 시간으로 바뀌게 된다. 만약 INCLUDE_vTaskSuspend가 1로 설정되어 있다면, portMAX_DELAY를 대기 시간으로 사용하는 것은 태스크가 무기한으로 대기하게 만들 수 있다.

[RETURN VALUES]

성공적으로 아이템을 추가했다면 pdTRUE, 그렇지 않다면 errQUEUE_FULL이 반환된다.

[NAME]

xQueueReceive

[SYNOPSIS]

```
#include "queue. h "

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

[DESCRIPTION]

xQueueReceive() 함수는 xQueueGenericReceive() 함수를 호출하는 매크로 함수이다.

큐로부터 아이템을 전달받는다. 아이템은 복사되어 전달받기 때문에 적절한 버퍼 크기가 제공되어야 한다. 큐가 생성될 때 정의한 길이만큼의 영역이 버퍼로 복사된다.

xQueueReceive() 함수는 인터럽트 서비스 루틴에서 사용될 수 없다. ISR(인터럽트 서비스 루틴)에서 아이템을 전달받기 위해서는 xQueueReceiveFromISR() 함수를 사용해야 한다.

xQueueReceive() 함수는 태스크 간 통신 API중 fully featured API에 속하며, xQueueAltReceive() 함수는 동일한 기능을 하는 alternative API이다. 두 버전 모두 동일한 매개 변수와 동일한 반환 값을 가진다.

[PARAMETERS]

xQueue	:	아이템을 전달받으려고 하는 큐의 핸들
pvBuffer	:	전달받을 아이템이 복사될 버퍼를 가리키는 포인터
xTicksToWait	:	함수를 호출한 시점에 큐가 비어 있어 아이템을 전달 받을 수 있을 때까지 대기하는 최대 시간. 만약 큐가 비어있는 상태에서 xTicksToWait가 0으로 설정되어 있다면, 즉시 리턴 된다. 틱 주기로 정의된 시간은 portTICK_RATE_MS를 통해 실제 시간으로 바뀌게 된다. 만약 INCLUDE_vTaskSuspend가 1로 설정되어 있다면, portMAX_DELAY를 대기 시간으로 사용하는 것은 태스크가 무기한으로 대기하게 만들 수 있다.

[RETURN VALUES]

성공적으로 아이템을 전달받았다면 pdTRUE, 그렇지 않다면 pdFALSE가 반환된다.

[NAME]

xQueuePeek

[SYNOPSIS]

```
#include "queue. h "

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

[DESCRIPTION]

xQueuePeek() 함수는 xQueueGenericReceive() 함수를 호출하는 매크로 함수이다.

큐에 있는 아이템을 삭제하지 않고 아이템을 전달받는다. 아이템은 복사되어 전달받기 때문에 적절한 버퍼 크기가 제공되어야 한다. 큐가 생성될 때 정의한 길이만큼의 영역이 버퍼로 복사된다.

큐에 남아있는 아이템은 xQueuePeek() 함수를 다시 호출하거나, xQueueReceive() 함수를 호출하여 다시 전달받을 수 있다.

xQueuePeek() 함수는 인터럽트 서비스 루틴에서 사용될 수 없다.

xQueuePeek() 함수는 태스크 간 통신 API중 fully featured API에 속하며, xQueueAltPeek() 함수 는 동일한 기능을 하는 alternative API이다. 두 버전 모두 동일한 매개 변수와 동일한 반환 값을 가진다.

[PARAMETERS]

xQueue	:	아이템을 전달받으려고 하는 큐의 핸들
pvBuffer	:	전달받을 아이템이 복사될 버퍼를 가리키는 포인터 이 값은 큐가 생성될 때 정의한 큐 아이템의 크기를 복사할 수 있을 만큼 커야한다.
·xTicksToWait	:	함수를 호출한 시점에 큐가 비어 있어 아이템을 전달 받을 수 있을 때까지 대기하는 최대 시간. 만약 큐가 비어있는 상태에서 xTicksToWait가 0으로 설정되어 있다면, 즉시 리턴 된다. 틱 주기로 정의된 시간은 portTICK_RATE_MS를 통해 실제 시간으로 바뀌게 된다. 만약 INCLUDE_vTaskSuspend가 1로 설정되어 있다면, portMAX_DELAY를 대기 시간으로 사용하는 것은 태스크가 무기한으로 대기하게 만들 수 있다.

[RETURN VALUES]

성공적으로 아이템을 전달받았다면 pdTRUE, 그렇지 않다면 pdFALSE가 반환된다.

[NAME]

xQueueSendFromISR

[SYNOPSIS]

```
#include "queue. h "

portBASE_TYPE xQueueSendFromISR( xQueueHandle pxQueue,
                                 const void *pvItemToQueue,
                                 portBASE_TYPE *pxHigherPriorityTaskWoken );
```

[DESCRIPTION]

xQueueGenericSendFromISR() 함수를 호출하는 매크로 함수이다.
xQueueGenericSend() 매크로는 xQueueSendToFrontFromISR(), xQueueSendToBackFromISR() 매크로를 포함하지 않는 버전의 FreeRTOS와의 호환성을 위해 포함되어 있다.

큐에(큐의 뒤부터) 아이템을 추가한다. 이 함수는 인터럽트 서비스 루틴에서 사용될 수 있다.

아이템은 참조되는 것이 아니라 복사되어 큐에 추가되기 때문에 특히 ISR(인터럽트 서비스 루틴)에서 호출될 때는 작은 크기의 아이템을 추가하는 것이 적절하다. 대부분의 경우 아이템의 포인터를 큐 에 추가하는 것이 바람직하다.

[PARAMETERS]

xQueue	:	아이템을 추가할 큐의 핸들
prvItemToQueue	:	큐에 추가하려고 하는 아이템의 포인터. 아이템의 크기는 큐가 생성 될 때 정의된 크기이며, prvItemToQueue가 가리키는 영역에서부터가 길이만큼의 바이트 영역이 큐의 저장 영역으로 복사된다.
pxHigherPriorityTaskWoken	:	큐에 아이템을 추가하는 것이 태스크를 Unblock 상태로 만들고, Unblock된 태스크의 우선순위가 현재 실행중인 태스크의 우선순위 보다 높은 경우 *pxHigherPriorityTaskWoken은 pdTRUE로 설정된다. pxHigherPriorityTaskWoken값이 pdTRUE로 설정된다면, 인터럽트가 끝나기 전에 문맥 교환이 요청된다.

[RETURN VALUES]

성공적으로 아이템을 추가했다면 pdTRUE, 그렇지 않다면 errQUEUE_FULL이 반환된다.

[NAME]

xQueueSendToBackFromISR

[SYNOPSIS]

```
#include "queue. h "

portBASE_TYPE
xQueueSendToBackFromISR( xQueueHandle pxQueue,
                         const void *pvItemToQueue,
                         portBASE_TYPE *pxHigherPriorityTaskWoken );
```

[DESCRIPTION]

xQueueGenericSendFromISR() 함수를 호출하는 매크로 함수이다.

큐에(큐의 뒤부터) 아이템을 추가한다. 이 함수는 인터럽트 서비스 루틴에서 사용될 수 있다.

아이템은 참조되는 것이 아니라 복사되어 큐에 추가되기 때문에 특히 ISR(인터럽트 서비스 루틴)에서 호출될 때는 작은 크기의 아이템을 추가하는 것이 적절하다.

[PARAMETERS]

xQueue	:	아이템을 추가할 큐의 핸들
prvItemToQueue	:	큐에 추가하려고 하는 아이템의 포인터. 아이템의 크기는 큐가 생성 될 때 정의된 크기이며, prvItemToQueue가 가리키는 영역에서부터가 길이만큼의 바이트 영역이 큐의 저장 영역으로 복사된다.
xTaskPreviouslyWoken	:	큐에 아이템을 추가하는 것이 태스크를 Unblock 상태로 만들고, Unblock된 태스크의 우선순위가 현재 실행중인 태스크의 우선순위보다 높은 경우 *xTaskPreviouslyWoken은 pdTRUE로 설정된다. xTaskPreviouslyWoken값이 pdTRUE로 설정된다면, 인터럽트가 끝나기 전에 문맥 교환이 요청된다.

[RETURN VALUES]

성공적으로 아이템을 추가했다면 pdTRUE, 그렇지 않다면 errQUEUE_FULL이 반환된다.

[NAME]

xQueueSendToFrontFromISR

[SYNOPSIS]

```
#include "queue. h "

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

[DESCRIPTION]

xQueueGenericSendFromISR() 함수를 호출하는 매크로 함수이다.

큐에(큐의 처음부터) 아이템을 추가한다. 이 함수는 인터럽트 서비스 루틴에서 사용될 수 있다.

아이템은 참조되는 것이 아니라 복사되어 큐에 추가되기 때문에 특히 ISR(인터럽트 서비스 루틴)에서 호출될 때는 작은 크기의 아이템을 추가하거나 아이템의 포인터를 큐에 추가하는 것이 바람직하다.

[PARAMETERS]

xQueue	:	아이템을 추가할 큐의 핸들
prvItemToQueue	:	큐에 추가하려고 하는 아이템의 포인터. 아이템의 크기는 큐가 생성 될 때 정의된 크기이며, prvItemToQueue가 가리키는 영역에서부터가 길이만큼의 바이트 영역이 큐의 저장 영역으로 복사된다.
xTaskPreviouslyWoken	:	큐에 아이템을 추가하는 것이 태스크를 Unblock 상태로 만들고, Unblock된 태스크의 우선순위가 현재 실행중인 태스크의 우선순위 보다 높은 경우 *xTaskPreviouslyWoken은 pdTRUE로 설정된다. xTaskPreviouslyWoken값이 pdTRUE로 설정된다면, 인터럽트가 끝나기 전에 문맥 교환이 요청된다.

[RETURN VALUES]

성공적으로 아이템을 추가했다면 pdTRUE, 그렇지 않다면 errQUEUE_FULL이 반환된다.

[NAME]

xQueueReceiveFromISR

[SYNOPSIS]

```
#include "queue. h "

portBASE_TYPE xQueueReceiveFromISR( xQueueHandle pxQueue,
                                    void *pvBuffer,
                                    portBASE_TYPE *pxTaskWoken );
```

[DESCRIPTION]

큐로부터 아이템을 전달받는다. 이 함수는 인터럽트 서비스 루틴에서 사용될 수 있다.

[PARAMETERS]

pxQueue	:	아이템을 전달받을 큐의 핸들
pvBuffer	:	전달받을 아이템을 복사하게 될 버퍼를 가리키는 포인터
pxTaskWoken	:	태스크는 큐를 사용할 수 있게 될 때까지 대기상태로 기다리게 된다. xQueueReceiveFromISR() 함수가 태스크를 Unblock 상태로 만들면 *pxTaskWoken은 pdTRUE로 설정되고, 그렇지 않다면 값은 변하지 않는다.

[RETURN VALUES]

성공적으로 아이템을 전달받았다면 pdTRUE, 그렇지 않다면 pdFALSE가 반환된다.

[NAME]

vQueueAddToRegistry

[SYNOPSIS]

```
#include "queue.h"

void vQueueAddToRegistry( xQueueHandle xQueue,
                          signed portCHAR *pcQueueName );
```

[DESCRIPTION]

큐에 이름을 할당하고 레지스트리에 큐를 추가한다.

[PARAMETERS]

xQueue	:	레지스트리에 추가하게 될 큐의 핸들
pcQueueName	:	큐에 할당되는 이름. 디버깅을 용이하게 만들어주는 텍스트로 된 문자열이다.

큐 레지스트리는 두 가지 목적을 가지며, 두 가지 목적 모두 커널을 디버깅 하기 위한 용도이다.

1. 큐에 할당된 텍스트 이름을 통해 디버깅 GUI에서 쉽게 큐를 인식할 수 있게 한다.
2. 등록된 큐와 세마포어를 찾기 위해 디버거가 요구하는 정보를 가지고 있다.

큐 레지스트리는 커널 디버깅을 위한 목적으로만 사용된다.

configQUEUE_REGISTRY_SIZE는 등록될 수 있는 큐와 세마포어의 최대 개수를 정의한다.
커널 디버거를 사용하여 큐와 세마포어를 보고자 할 때만 등록될 필요가 있다.

[RETURN VALUES]

None

[NAME]

vQueueUnregisterQueue

[SYNOPSIS]

```
#include "queue.h"

void vQueueUnregisterQueue( xQueueHandle xQueue );
```

[DESCRIPTION]

큐 레지스트리에서 큐를 삭제한다.

[PARAMETERS]

xQueue : 레지스트리에서 삭제하게 될 큐의 핸들

큐 레지스트리는 두 가지 목적을 가지며, 두 가지 목적 모두 커널을 디버깅 하기 위한 용도이다.

1. 큐에 할당된 텍스트 이름을 통해 디버깅 GUI에서 쉽게 큐를 인식할 수 있게 한다.
2. 등록된 큐와 세마포어를 찾기 위해 디버거가 요구하는 정보를 가지고 있다.

큐 레지스트리는 커널 디버깅을 위한 목적으로만 사용된다.

configQUEUE_REGISTRY_SIZE는 등록될 수 있는 큐와 세마포어의 최대 개수를 정의한다.
커널 디버거를 사용하여 큐와 세마포어를 보고자 할 때만 등록될 필요가 있다.

[RETURN VALUES]

None

[NAME]

xQueueIsQueueFullFromISR

[SYNOPSIS]

#include "queue.h"

portBASE_TYPE xQueueIsQueueFullFromISR(xQueueHandle pxQueue);

[DESCRIPTION]

큐가 가득 찬(full) 상태인지 확인한다.

[PARAMETERS]

xQueue : 상태를 확인할 큐의 핸들

[RETURN VALUES]

큐가 가득 찬 상태가 아니라면 pdFALSE, 큐가 가득 찬 상태라면 pdFALSE가 아닌 다른 값이 반환된다.

[NAME]

xQueueIsQueueEmptyFromISR

[SYNOPSIS]

#include "queue.h"

portBASE_TYPE xQueueIsQueueEmptyFromISR(xQueueHandle pxQueue);

[DESCRIPTION]

큐가 비어 있는(empty) 상태인지 확인한다.

[PARAMETERS]

xQueue : 상태를 확인할 큐의 핸들

[RETURN VALUES]

큐가 비어 있는 상태가 아니라면 pdFALSE, 큐가 비어 있는 상태라면 pdFALSE가 아닌 다른 값이 반환된다.

[NAME]

vSemaphoreCreateBinary

[SYNOPSIS]

#include "semphr.h"

vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore);

[DESCRIPTION]

vSemaphoreCreateBinary() 함수는 큐 메커니즘을 통해 세마포어를 생성하는 매크로 함수이다.
바이너리 세마포어기 때문에 큐의 길이는 1로 만들어지며, 큐에는 특정 데이터를 저장하는 용도로 사용하는 것이 아니라 큐가 비었는지 차있는지만 알면 되기 때문에 큐의 데이터 크기 (큐에 저장할 데이터의 크기)는 0으로 만들어진다.

바이너리 세마포어와 뮤텍스는 매우 유사하지만 미묘한 차이점을 가지고 있다.
뮤텐스는 우선순위 상속 메커니즘을 포함하고 있지만, 바이너리 세마포어는 이러한 기능을 지원하지 않는다. 따라서 바이너리 세마포어는 태스크 간 또는 태스크와 인터럽트 간 동기화를 위해, 뮤텍스는 간단한 상호배제를 위해 사용하는데 적절하다.

바이너리 세마포어는 태스크 동기화를 하나의 태스크 또는 인터럽트에서 세마포어를 지속적으로 반환하도록 하고 다른 태스크 또는 인터럽트에서는 세마포어를 지속적으로 얻도록 함으로써 구현될 수 있기 때문에 세마포어를 얻은 뒤 다시 반환할 필요가 없다.
이러한 내용은 xSemaphoreGiveFromISR() 페이지에 있는 샘플 코드에서 보여주고 있다.

뮤텍스를 소유한 태스크의 우선순위는 더 높은 우선순위를 갖는 다른 태스크가 동일한 뮤텍스를 획득하려고 할 때 일시적으로 상승하게 되는 것이 가능하다. 뮤텍스를 소유한 태스크는 동일한 뮤텍스를 획득하려고 하는 더 높은 우선순위 태스크의 우선순위를 상속받을 수 있다.

이것은 뮤텍스를 반드시 반환해야 한다는 것을 의미한다. 뮤텍스를 반환하지 않는다면, 높은 우선순위의 태스크는 뮤텍스를 절대 획득할 수 없게 되고, 낮은 우선순위의 태스크는 상속받은 우선순위를 놓지 못하게 된다.

뮤텍스와 바이너리 세마포어는 xSemaphoreHandle 타입의 변수에 할당되며, xSamaphoreHandle 타입의 변수를 통해 다른 API 함수들에서 사용될 수 있다.

[PARAMETERS]

xSemaphore : xSemaphoreHandle 타입으로, 생성될 세마포어에 대한 핸들

[RETURN VALUES]

None

[NAME]

xSemaphoreCreateCounting

[SYNOPSIS]

```
#include "semphr.h"

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

[DESCRIPTION]

xSemaphoreCreateCounting() 함수는 큐 메커니즘을 통해 카운팅 세마포어를 생성하는 매크로 함수이다.

카운팅 세마포어는 일반적으로 다음과 같은 상황에서 사용된다.

1. 카운팅 이벤트
이벤트 핸들러에서는 이벤트가 발생될 때 마다 세마포어를 반환하고(세마포어 카운트 값을 증가시키고), 핸들러 태스크에서는 세마포어를 얻어(세마포어 카운트 값을 감소시키고) 이벤트를 처리한다. 그러므로 카운트 값은 발생되는 이벤트의 횟수와 이벤트를 처리하는 횟수에 따라 차이를 보인다. 위와 같은 상황에서는 초기 카운트 값으로 0을 설정하는 것이 적절하다.

2. 자원 관리
세마포어의 카운트 값은 이용 가능한 자원의 수를 나타낸다. 태스크가 자원의 제어권을 획득하기 위해서는 먼저 세마포어를 획득해야 한다.(세마포어의 카운트 값이 감소된다.) 세마포어의 카운트 값이 0이라는 것은 이용 가능한 자원이 없다는 것을 의미한다. 태스크는 자원에 대한 사용이 끝났을 때 세마포어를 반환한다. (세마포어의 카운트 값이 증가된다.) 위와 같은 상황에서는 모든 자원이 이용 가능하다는 것을 나타낼 수 있는 최대 값을 초기 카운트 값으로 설정하는 것이 적절하다.

[PARAMETERS]

uxMaxCount	:	접근할 수 있는 세마포어의 최대 카운트 값 세마포어의 카운트 값이 uxMaxCount 일 때, 더 이상 반환될 수 없다는 것을 의미한다.
uxInitialCount	:	세마포어가 생성될 때 할당되는 세마포어의 카운트 값

[RETURN VALUES]

세마포어가 성공적으로 생성된다면 xSemaphoreHandle 타입의 핸들이 반환되며, 세마포어가 생성될 수 없다면 NULL이 반환된다.

[NAME]

xSemaphoreCreateMutex

[SYNOPSIS]

```
#include "semphr.h"

xSemaphoreHandle xSemaphoreCreateMutex( void );
```

[DESCRIPTION]

xSemaphoreCreateMutex() 함수는 큐 메커니즘을 통해 뮤텍스를 생성하는 매크로 함수이다.

생성된 뮤텍스는 xSemaphoreTake(), xSemaphoreGive() 매크로 함수를 통해 사용할 수 있다. xSemaphoreTakeRecursive(), xSemaphoreGiveRecursive() 매크로 함수는 사용할 수 없다.

바이너리 세마포어와 뮤텍스는 매우 유사하지만 미묘한 차이점을 가지고 있다. 뮤텐스는 우선순위 상속 메커니즘을 포함하고 있지만, 바이너리 세마포어는 이러한 기능을 지원하지 않는다. 따라서 바이너리 세마포어는 태스크 간 또는 태스크와 인터럽트 간 동기화를 위해, 뮤텍스는 간단한 상호배제를 위해 사용하는데 적절하다.

뮤텍스를 소유한 태스크의 우선순위는 더 높은 우선순위를 갖는 다른 태스크가 동일한 뮤텍스를 획득하려고 할때 일시적으로 상승하게 되는 것이 가능하다. 뮤텍스를 소유한 태스크는 동일한 뮤텍스를 획득하려고 하는 더 높은 우선순위 태스크의 우선순위를 상속받을 수 있다.

이것은 뮤텍스를 반드시 반환해야 한다는 것을 의미한다. 뮤텍스를 반환하지 않는다면, 높은 우선순위의 태스크는 뮤텍스를 절대 획득할 수 없게 되고, 낮은 우선순위의 태스크는 상속받은 우선순위를 놓지 못하게 된다.

바이너리 세마포어는 태스크 동기화를 하나의 태스크 또는 인터럽트에서 세마포어를 지속적으로 반환하도록 하고 다른 태스크 또는 인터럽트에서는 세마포어를 지속적으로 얻도록 함으로써 구현될 수 있기 때문에 세마포어를 얻은 뒤 다시 반환할 필요가 없다.
이러한 내용은 xSemaphoreGiveFromISR() 페이지에 있는 샘플 코드에서 보여주고 있다.

뮤텍스와 바이너리 세마포어는 xSemaphoreHandle 타입의 변수에 할당되며, xSamaphoreHandle 타입의 변수를 통해 다른 API 함수들에서 사용될 수 있다.

[PARAMETERS]

xSemaphore : xSemaphoreHandle 타입으로, 생성될 세마포어에 대한 핸들

[RETURN VALUES]

None

[NAME]

xSemaphoreCreateRecursiveMutex

[SYNOPSIS]

#include "semphr.h"

xSemaphoreHandle xSemaphoreCreateRecursiveMutex(void);

[DESCRIPTION]

xSemaphoreCreateRecursiveMutex() 함수는 큐 메커니즘을 통해 재귀 뮤텍스를 생성하는 매크로 함수이다.

생성된 뮤텍스는 xSemaphoreTakeRecursive(), xSemaphoreGiveRecursive() 매크로 함수를 통해 사용할 수 있다. xSemaphoreTake(), xSemaphoreGive() 매크로 함수는 사용할 수 없다.

xSemaphoreCreateRecursiveMutex() 함수를 이용하여 생성된 뮤텍스는 뮤텍스의 소유자에 의해 반복적으로 획득될 수 있다. 뮤텍스는 성공적인 획득 요청을 위해 소유자에 의해 xSemaphoreGiveRecursive() 함수를 호출하기 전까지 사용할 수 없게 된다. 만약 특정 태스크가 동 일한 뮤텍스를 5번 획득하였다면, 정확히 5번 반환하기 전 까지는 어떤 태스크도 해당 뮤텍스를 사용할 수 없게 된다.

이런 유형의 세마포어는 우선순위 상속 메커니즘을 사용하기 때문에 세마포어를 획득한 태스크는 세마포어가 더 이상 필요가 없다면 반드시 세마포어를 반환해야 한다.

뮤텍스 타입의 세마포어는 인터럽트 서비스 루틴에서 사용될 수 없다.

FreeRTOS는 순수 동기화 (하나의 태스크 또는 인터럽트에서는 세마포어를 항상 반환하고 다른 태스크 또는 인터럽트에서는 세마포어를 항상 얻는)기능을 위해, 그리고 인터럽트 서비스 루틴에서 세마포어를 사용할 수 있는 방법으로 vSemaphoreCreateBinary() 함수를 제공한다.

[PARAMETERS]

xSemaphore : xSemaphoreHandle 타입으로, 생성될 세마포어에 대한 핸들

[RETURN VALUES]

None

[NAME]

vSemaphoreDelete

[SYNOPSIS]

#include "semphr.h"

void vSemaphoreDelete(xSemaphoreHandle xSemaphore);

[DESCRIPTION]

세마포어를 삭제한다.

세마포어는 삭제하려고 하는 세마포어를 획득하기 위해 대기중인 태스크가 있는 경우 삭제할 수 없다.

[PARAMETERS]

xSemaphore : 삭제할 세마포어의 핸들

[RETURN VALUES]

None

[NAME]

xSemaphoreGetMutexHolder

[SYNOPSIS]

#include "semphr.h"

xSemaphoreGetMutexHolder(xSemaphoreHandle xMutex);

[DESCRIPTION]

매개변수에 의해 지정된 뮤텍스를 소유하고 있는 태스크의 핸들을 반환한다.

xSemaphoreGetMutexHolder() 함수는 함수를 호출한 태스크가 뮤텍스를 소유하고 있는지 확실하게 알아보는데 사용될 수 있다. 하지만, 뮤텍스가 함수를 호출한 태스크가 아닌 다른 태스크에 의해 소유되어져 있다면 어느 태스크가 뮤텍스를 소유하고 있는지는 확인할 수 없다. 태스크가 함수를 호출한 시점과 함수의 반환 값을 확인하는 시점 사이 뮤텍스의 소유자가 바뀔 수도 있기 때문이다.

xSemaphoreGetMutexHolder() 함수를 사용하기 위해서는 configUSE_MUTEXES를 반드시 1로 설정해야 한다.

[PARAMETERS]

xSemaphore : 확인할 뮤텍스의 핸들

[RETURN VALUES]

xMutex 매개변수에 지정된 뮤텍스를 소유하고 있는 태스크의 핸들. xMutex 매개변수로 전달된 세마포어가 뮤텍스 타입의 세마포어가 아니거나, 뮤텍스가 어느 태스크에도 소유되지 않아 사용할 수 있는 경우 NULL을 반환한다.

[NAME]

xSemaphoreTake

[SYNOPSIS]

#include "semphr.h"

xSemaphoreTake(xSemaphoreHandle xSemaphore, portTickType xBlockTime);

[DESCRIPTION]

xSemaphoreTake() 함수는 세마포어를 획득하는 매크로 함수이다.
xSemaphoreTake() 함수를 사용하기 위해서는 xSemaphoreTake() 함수를 호출하기 전에 vSemaphoreCreateBinary(), xSemaphoreCreateMutex(),
xSemaphoreCreateCounting() 함수를 호출하여 세마포어를 생성해 놓아야 한다.

xSemaphoreTake() 매크로 함수는 ISR(인터럽트 서비스 루틴)에서 호출될 수 없다.
ISR에서 xSemaphoreTake() 함수를 호출하는 것은 정상적인 방법이 아니지만, 인터럽트 내에서 필요하다면 세마포어를 얻기 위하여 xQueueReceivedFromISR() 함수를 사용할 수 있다.
세마포어는 이러한 메커니즘으로써 큐를 사용하기 때문에 함수들은 상호 정보 교환이 가능할 정도의 규모를 가진다.

xSemaphoreTake() 함수는 태스크 간 통신 API중 fully featured API에 속하며, xSemaphoreAltTake() 함수는 동일한 기능을 하는 alternative API 이다. 두 버전 모두 동일한 매개변수와 동일한 반환 값을 가진다.

[PARAMETERS]

xSemaphore	:	이미 생성된 세마포어를 얻기 위한 세마포어의 핸들
xBlockTime	:	세마포어를 사용할 수 있을 때까지 기다리는 시간 (틱) portTICK_RATE_MS 매크로는 xBlockTime을 실제 시간으로 바꾸는데 사용된다. xBlockTime을 0으로 사용함으로써 세마포어를 폴링하는데 사용될 수 있다.

[RETURN VALUES]

세마포어를 획득하면 pdTRUE, xBlockTime 만큼의 시간(틱)이 지난 후에도 세마포어를 획득할 수 없으면 pdFALSE가 반환된다.

[NAME]

xSemaphoreTakeRecursive

[SYNOPSIS]

#include "semphr.h"

xSemaphoreTakeRecursive(xSemaphoreHandle xMutex, portTickType xBlockTime);

[DESCRIPTION]

뮤텍스 타입의 세마포어를 반복적으로 획득하기 위한 매크로 함수이다.
xSemaphoreTakeRecursive() 함수를 사용하기 위해서는 xSemaphoreTakeRecursive() 함수를 호출하기 전에 xSemaphoreCreateRecursiveMutex() 함수를 통해 뮤텍스를 생성해야 한다.

xSemaphoreTakeRecursive() 함수를 사용하기 위해서는 FreeRTOSConfig.h에 있는 configUSE_RECURSIVE_MUTEXS를 1로 설정해야 한다.

xSemaphoreTakeRecursive() 함수는 xSemaphoreCreateMutex()를 이용하여 생성된 뮤텍스를 사용할 수 없다.

xSemaphoreCreateRecursiveMutex() 함수를 이용하여 생성된 뮤텍스는 뮤텍스의 소유자에 의해 반복적으로 획득될 수 있다. 뮤텍스는 성공적인 획득 요청을 위해 소유자에 의해 xSemaphoreGiveRecursive() 함수를 호출하기 전까지 사용할 수 없게 된다. 만약 특정 태스크가 동일한 뮤텍스를 5번 획득하였다면, 정확히 5번 반환하기 전 까지는 어떤 태스크도 해당 뮤텍스를 사용할 수 없게 된다.

[PARAMETERS]

xSemaphore	:	뮤텍스를 획득하기 위한 핸들로 xSemaphoreCreateRecursiveMutex()에 의해 반환된 핸들이다.
xBlockTime	:	세마포어를 사용할 수 있을 때까지 기다리는 시간 (틱) portTICK_RATE_MS 매크로는 xBlockTime을 실제 시간으로 바꾸는데 사용된다. xBlockTime을 0으로 사용함으로써 세마포어를 폴링하는데 사용될 수 있다. 태스크가 이미 세마포어를 획득한 상태라면 xSemaphoreTakeRecursive() 함수는 xBlockTime의 값과는 상관없이 즉시 반환된다.

[RETURN VALUES]

세마포어를 획득하면 pdTRUE, xBlockTime 만큼의 시간(틱)이 지난 후에도 세마포어를 획득할 수 없으면 pdFALSE가 반환된다.

[NAME]

xSemaphoreGive

[SYNOPSIS]

```
#include "semphr.h"

xSemaphoreGive( xSemaphoreHandle xSemaphore );
```

[DESCRIPTION]

xSemaphoreGive() 함수는 세마포어를 반환하는 매크로 함수이다.
xSemaphoreTake() 함수를 사용하기 위해서는 xSemaphoreTake() 함수를 호출하기 전에 vSemaphoreCreateBinary(), xSemaphoreCreateMutex(), xSemaphoreCreateCounting() 함수를 호출하여 세마포어를 생성해 놓아야 한다.

xSemaphoreGive() 매크로 함수는 ISR(인터럽트 서비스 루틴)에서 호출될 수 없다.
FreeRTOS는 ISR(인터럽트 서비스 루틴)에서 호출될 수 있는 xSemaphoreGiveFromISR() 함수를 제공한다.

xSemaphoreGive() 함수는 xSemaphoreCreateRecursiveMutex() 함수를 이용하여 생성된 뮤텍스를 사용할 수 없다.

xSemaphoreGive() 함수는 태스크 간 통신 API중 fully featured API에 속하며, xSemaphoreAltGive() 함수는 동일한 기능을 하는 alternative API이다. 두 버전 모두 동일한 매개변수와 동일한 반환 값을 가진다.

[PARAMETERS]

xSemaphore : 반환할 세마포어의 핸들로 세마포어가 생성될 때 반환된 핸들이다.

[RETURN VALUES]

세마포어를 반환하면 pdTRUE, 에러가 발생하면 pdFALSE를 반환한다.
세마포어는 큐를 사용하여 구현되며, 에러는 큐에 메시지를 전달할 공간이 없을 때 발생할 수 있다.

[NAME]

xSemaphoreGiveRecursive

[SYNOPSIS]

```
#include "semphr.h"

xSemaphoreGiveRecursive( xSemaphoreHandle xMutex );
```

[DESCRIPTION]

뮤텍스 타입의 세마포어를 반복적으로 반환하기 위한 매크로 함수이다.
xSemaphoreGiveRecursive() 함수를 사용하기 위해서는 xSemaphoreGiveRecursive() 함수를 호출하기 전에 xSemaphoreCreateRecursiveMutex() 함수를 통해 뮤텍스가 생성되어 있어야 한다.

xSemaphoreGiveRecursive() 함수를 사용하기 위해서는 FreeRTOSConfig.h에 있는 configUSE_RECURSIVE_MUTEXS를 1로 설정해야 한다.

xSemaphoreGiveRecursive() 함수는 xSemaphoreCreateMutex()를 이용하여 생성된 뮤텍스를 사용할 수 없다.

xSemaphoreCreateRecursiveMutex() 함수를 이용하여 생성된 뮤텍스는 뮤텍스의 소유자에 의해 반복적으로 획득될 수 있다. 뮤텍스는 성공적인 획득 요청을 위해 소유자에 의해 xSemaphoreGiveRecursive() 함수를 호출하기 전까지 사용할 수 없게 된다. 만약 특정 태스크가 동일한 뮤텍스를 5번 획득하였다면, 정확히 5번 반환하기 전 까지는 어떤 태스크도 해당 뮤텍스를 사용할 수 없게 된다.

[PARAMETERS]

xMutex	:	뮤텍스를 반환하기 위한 핸들로 xSemaphoreCreateRecursiveMutex()에 의해 반환 된 핸들이다.

[RETURN VALUES]

세마포어를 성공적으로 반환하면 pdTRUE를 반환한다.

[NAME]

xSemaphoreGiveFromISR

[SYNOPSIS]

```
#include "semphr.h"

xSemaphoreGiveFromISR( xSemaphoreHandle xSemaphore,
                       signed portBASE_TYPE *pxHigherPriorityTaskWoken );
```

[DESCRIPTION]

xSemaphoreGiveFromISR() 함수는 세마포어를 반환하는 매크로 함수이다.

xSemaphoreGiveFromISR() 함수를 사용하기 위해서는 xSemaphoreGiveFromISR() 함수를 호출하기 전에 vSemaphoreCreateBinary(), xSemaphoreCreateCounting() 함수를 호출하여 세마포어를 생성해 놓아야 한다.

뮤텍스 타입의 세마포어(xSemaphoreCreateMutex() 함수를 이용하여 생성된 세마포어)는 xSemaphoreGiveFromISR() 함수를 사용할 수 없다.

xSemaphoreGiveFromISR() 함수는 ISR(인터럽트 서비스 루틴)에서 사용할 수 있다.

[PARAMETERS]

xSemaphore	:	세마포어를 반환하기 위한 핸들로 세마포어가 생성될 때 반환된 핸들이다.
pxHigherPriorityTaskWoken	:	세마포어를 반환하는 것이 태스크를 Unblock 상태로 만들고, Unblock된 태스크의 우선순위가 현재 실행중인 태스크의 우선순위보다 높은 경우 *pxHigherPriorityTaskWoken은 pdTRUE로 설정된다. pxHigherPriorityTaskWoken 값이 pdTRUE로 설정된다면, 인터럽트가 끝나기 전에 문맥 교환이 요청된다.

[RETURN VALUES]

세마포어를 성공적으로 반환하면 pdTRUE, 그렇지 않다면 errQUEUE_FULL이 반환된다.

[NAME]

xTimerCreate

[SYNOPSIS]

```
#include "timers.h"

xTimerHandle xTimerCreate( const signed char *pcTimerName,
                           portTickType xTimerPeriod,
                           unsigned portBASE_TYPE uxAutoReload,
                           void * pvTimerID,
                           tmrTIMER_CALLBACK pxCallbackFunction );
```

[DESCRIPTION]

소프트웨어 타이머 객체를 생성한다. xTimerCreate() 함수는 새로 생성되는 타이머의 내부 상태를 초기화하기 위한 저장 공간을 할당하고 타이머를 참조할 수 있는 핸들을 리턴 한다.
타이머는 휴면(동작하고 있지 않은) 상태로 생성되며, xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod(), xTimerChangePeriodFromISR() 함수를 사용하여 타이머의 상태를 바꿀 수 있다.

[PARAMETERS]

pcTimerName : 타이머에 할당되는 이름
pcTimerName은 오직 디버깅 용도로만 사용된다. 커널은 타이머를 참조하기 위해 pcTimerName이 아닌 타이머의 핸들을 사용한다.

xTimerPeriod : 타이머 주기. 시간은 틱 주기로 정의되며 portTICK_RATE_MS는 명시 된 시간을 밀리 초로 변경하는데 사용된다. 예를 들어, 타이머가 100틱 이후에 만료되어야 한다면, xTimerPeriod는 100으로 설정되어야 한다. 또 다른 방법으로 타이머가 500틱 이후에 만료되어야 한다면, xPeriod는 1000 이하의 configTICK_RATE_HZ에 의해 제공된 500 / portTICK_RATE_MS로 설정 될 수 있다.

uxAutoReload : uxAutoReload를 pdTRUE로 설정하는 경우, xTimerPeriod 매개 변수로 설정한 주기마다 반복해서 타이머가 만료된다. 만약 uxAutoReload를 pdFALSE로 설정하는 경우, one-shot 타이머로 사용되며 타이머가 만료된 후 휴면 상태가 된다.

pvTimerID : 타이머가 생성될 때 부여되는 식별자. pvTimerID는 동일한 콜백 함수가 하나 이상의 타이머에 등록되어 있을 때 어떤 타이머에 의해 호출되는

것인지 식별하기 위해 타이머 콜백 함수에서 사용한다.

pxCallbackFunction : 타이머가 만료되었을 때 호출되는 함수
콜백 함수는 반드시 tmrTIMER_CALLBACK 매크로
(즉, void vCallbackFunction(xTimerHandle xTimer);)를 통해 정의된 프로토타입을 가져야 한다.

[RETURN VALUES]

타이머가 성공적으로 생성되었을 경우 새로 생성된 타이머의 핸들이 리턴 된다.
타이머가 생성될 수 없다면(타이머 구조체를 위한 충분한 힙 메모리 영역이 부족하거나 타이머 주기가 0으로 설정되어 있기 때문에) 0을 리턴 한다.

[NAME]

xTimerIsTimerActive

[SYNOPSIS]

#include "timers.h"

portBASE_TYPE xTimerIsTimerActive(xTimerHandle xTimer);

[DESCRIPTION]

타이머가 실행중인지 휴면 상태인지 확인한다.
타이머가 휴면 상태를 가지는 경우는 다음과 같다.

1. 타이머가 생성되었지만 실행되지 않음
2. One-shot 타이머로 생성되어 타이머가 만료되고 재 시작되지 않음

타이머는 휴면 상태로 생성된다. xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod(), xTimerChangePeriodFromISR() 함수들은 타이머의 상태를 변화 시키는데 사용할 수 있다.

[PARAMETERS]

xTimer : 상태를 확인할 타이머의 핸들

[RETURN VALUES]

타이머가 휴면 상태일 경우 pdFALSE가 리턴 된다.
타이머가 실행 중일 경우 pdFALSE가 아닌 다른 값이 리턴 된다.

[NAME]

xTimerStart

[SYNOPSIS]

```
#include "timers.h"

portBASE_TYPE xTimerStart( xTimerHandle xTimer, portTickType xBlockTime );
```

[DESCRIPTION]

FreeRTOS의 타이머 기능은 타이머 서비스/데몬 태스크에 의해 제공된다.
FreeRTOS에서 제공하는 많은 수의 타이머 API 함수는 타이머 명령 큐라고 명명된 큐를 통해 타이머 서비스 태스크에게 명령을 전송한다. 타이머 명령 큐는 커널만 접근하도록 하기 위해 Private 속성을 가지며, 어플리케이션 코드에서의 직접적인 접근이 불가능하다. 타이머 명령 큐의 길이는 configTIMER_QUEUE_LENGTH를 설정함으로써 변경할 수 있다.

xTimerStart() 함수는 xTimerCreate() 함수를 통해 생성된 타이머를 구동시킨다.
만약, 타이머가 이미 실행중이라면 xTimerReset() 함수와 상응하는 동작을 한다.

타이머의 구동은 타이머가 실행(Active) 상태가 되도록 한다. 타이머가 구동되는 동안 멈추거나 삭제되거나 초기화되지 않는다면, 타이머에 등록된 콜백 함수는 xTimerStart() 함수가 호출된 시점으로부터 'n'틱(타이머의 주기) 후에 호출된다.

스케줄러가 구동되기 전에 xTimerStart() 함수를 호출하는 것은 가능하지만, 실제로는 스케줄러가 구동되기 전까지 타이머는 구동되지 않으며, 타이머의 만료 시간은 xTimerStart() 함수가 호출된 시점이 아니라 스케줄러가 구동된 시점과 연관이 있다.

xTimerStart() 함수를 사용하기 위해서는 configUSE_TIMERS를 반드시 1로 설정해야 한다.

[PARAMETERS]

xTimer	:	시작/재시작 될 타이머의 핸들
xBlockTime	:	xTimerStart() 함수가 호출될 때 이미 큐가 꽉 차 있다면 타이머 명령 큐에 타이머 시작 명령을 성공적으로 보낼 수 있을 때 까지 대기 상태에서 태스크가 기다리게 되는 시간(틱). 스케줄러가 구동되기 전에 xTimerStart() 함수가 호출된다면 xBlockTime은 무시된다.

[RETURN VALUES]

xBlockTime 만큼의 틱이 경과한 후에도 타이머 명령 큐에 타이머 시작 명령이 전송될 수 없다면 pdFAIL이 리턴 된다. 타이머 명령 큐에 타이머 시작 명령이 성공적으로 전송된다면 pdPASS가 리턴 된다. 타이머 만료 시간은 xTimerStart() 함수가 호출된 시점과 관련이 있지만, 타이머 시작 명령이 실질적으로 처리되는 시점은 시스템의 다른 태스크들과 관계를 갖는 타이머 서비스/데몬 태스크의 우선순위에 의해 영향을 받는다. 타이머 서비스/데몬 태스크의 우선순위는 configTIMER_TASK_PRIORITY를 설정함으로써 변경할 수 있다.

[NAME]

xTimerStop

[SYNOPSIS]

```
#include "timers.h"

portBASE_TYPE xTimerStop( xTimerHandle xTimer, portTickType xBlockTime );
```

[DESCRIPTION]

FreeRTOS의 타이머 기능은 타이머 서비스/데몬 태스크에 의해 제공된다.
FreeRTOS에서 제공하는 많은 수의 타이머 API 함수는 타이머 명령 큐라고 명명된 큐를 통해 타이머 서비스 태스크에게 명령을 전송한다. 타이머 명령 큐는 커널만 접근하도록 하기 위해 Private 속성을 가지며, 어플리케이션 코드에서의 직접적인 접근이 불가능하다. 타이머 명령 큐의 길이는 configTIMER_QUEUE_LENGTH를 설정함으로써 변경할 수 있다.

xTimerStop() 함수는 xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod(), xTimerChangePeriodFromISR() 함수를 사용해 이미 실행 상태에 있는 타이머를 정지 시킨다.

타이머의 정지는 타이머가 실행(Active) 상태가 아니라는 것을 의미한다.

[PARAMETERS]

xTimer	:	정지하게 될 타이머의 핸들
xBlockTime	:	xTimerStop() 함수가 호출될 때 이미 큐가 꽉 차 있다면 타이머 명령 큐에 타이머 정지 명령을 성공적으로 보낼 수 있을 때 까지 대기 상태에서 태스크가 기다리게 되는 시간(틱)

[RETURN VALUES]

xBlockTime 만큼의 틱이 경과한 후에도 타이머 명령 큐에 타이머 정지 명령이 전송될 수 없다면 pdFAIL이 리턴 된다. 타이머 명령 큐에 타이머 정지 명령이 성공적으로 전송된다면 pdPASS가 리턴 된다. 타이머 정지 명령이 처리되는 시점은 시스템의 다른 태스크들과 관계를 갖는 타이머 서비스/데몬 태스크의 우선순위에 의해 영향을 받는다. 타이머 서비스/데몬 태스크의 우선순위는 configTIMER_TASK_PRIORITY를 설정함으로써 변경할 수 있다.

[NAME]

xTimerChangePeriod

[SYNOPSIS]

```
#include "timers.h"

portBASE_TYPE xTimerChangePeriod( xTimerHandle xTimer,
                                  portTickType xNewPeriod,
                                  portTickType xBlockTime );
```

[DESCRIPTION]

FreeRTOS의 타이머 기능은 타이머 서비스/데몬 태스크에 의해 제공된다.
FreeRTOS에서 제공하는 많은 수의 타이머 API 함수는 타이머 명령 큐라고 명명된 큐를 통해 타이머 서비스 태스크에게 명령을 전송한다. 타이머 명령 큐는 커널만 접근하도록 하기 위해 Private 속성을 가지며, 어플리케이션 코드에서의 직접적인 접근이 불가능하다. 타이머 명령 큐의 길이는 configTIMER_QUEUE_LENGTH를 설정함으로써 변경할 수 있다.

xTimerChangePeriod() 함수는 xTimerCreate() 함수를 통해 이미 생성된 타이머의 주기를 변경한다. xTimerChangePeriod() 함수는 타이머의 상태(실행 상태, 휴면 상태)에 상관없이 타이머의 주기를 변경하기 위해 호출될 수 있다.

xTimerChangePeriod() 함수를 사용하기 위해서는 configUSE_TIMERS를 반드시 1로 설정해야 한다.

[PARAMETERS]

xTimer	:	주기를 변경할 타이머의 핸들
xNewPeriod	:	타이머의 새로운 주기 시간은 틱 주기로 정의되며 portTICK_RATE_MS는 명시된 시간을 밀리 초로 변경하는데 사용된다. 예를 들어, 타이머가 100틱 이후에 만료되어야 한다면, xTimerPeriod는 100으로 설정되어야 한다. 또 다른 방법으로 타이머가 500틱 이후에 만료되어야 한다면, xPeriod는 1000이하의 configTICK_RATE_HZ에 의해 제공된 500 / portTICK_RATE_MS로 설정 될 수 있다.
xBlockTime	:	xTimerChangePeriod() 함수가 호출될 때 이미 큐가 꽉 차 있다면 타이머 명령 큐에 타이머 주기 변경 명령을 성공적으로 보낼 수 있을 때 까지 대기 상태에서 태스크가 기다리게 되는 시간(틱)

[RETURN VALUES]

xBlockTime 만큼의 틱이 경과한 후에도 타이머 명령 큐에 타이머 주기 변경 명령이 전송될 수 없다면 pdFAIL이 리턴 된다. 타이머 명령 큐에 타이머 주기 변경 명령이 성공적으로 전송된다면 pdPASS가 리턴 된다. 타이머 주기 변경 명령이 처리되는 시점은 시스템의 다른 태스크들과 관계를 갖는 타이머 서비스/데몬 태스크의 우선순위에 의해 영향을 받는다. 타이머 서비스/데몬 태스크의 우선순위는 configTIMER_TASK_PRIORITY를 설정함으로써 변경할 수 있다.

[NAME]

xTimerDelete

[SYNOPSIS]

#include "timers.h"

portBASE_TYPE xTimerDelete(xTimerHandle xTimer, portTickType xBlockTime);

[DESCRIPTION]

FreeRTOS의 타이머 기능은 타이머 서비스/데몬 태스크에 의해 제공된다.
FreeRTOS에서 제공하는 많은 수의 타이머 API 함수는 타이머 명령 큐라고 명명된 큐를 통해 타이머 서비스 태스크에게 명령을 전송한다. 타이머 명령 큐는 커널만 접근하도록 하기 위해 Private 속성을 가지며, 어플리케이션 코드에서의 직접적인 접근이 불가능하다. 타이머 명령 큐의 길이는 configTIMER_QUEUE_LENGTH를 설정함으로써 변경할 수 있다.

xTimerDelete() 함수는 xTimerCreate() 함수를 통해 이미 생성된 타이머를 삭제한다.

xTimerDelete() 함수를 사용하기 위해서는 configUSE_TIMERS를 반드시 1로 설정해야 한다.

[PARAMETERS]

xTimer	:	삭제할 타이머의 핸들
xBlockTime	:	xTimerDelete() 함수가 호출될 때 이미 큐가 꽉 차 있다면 타이머 명령 큐에 타이머 삭제 명령을 성공적으로 보낼 수 있을 때 까지 대기 상태에서 태스크가 기다리게 되는 시간(틱)

[RETURN VALUES]

xBlockTime 만큼의 틱이 경과한 후에도 타이머 명령 큐에 타이머 삭제 명령이 전송될 수 없다면 pdFAIL이 리턴 된다. 타이머 명령 큐에 타이머 삭제 명령이 성공적으로 전송된다면 pdPASS가 리턴 된다. 타이머 삭제 명령이 처리되는 시점은 시스템의 다른 태스크들과 관계를 갖는 타이머 서비스/데몬 태스크의 우선순위에 의해 영향을 받는다. 타이머 서비스/데몬 태스크의 우선순위는 configTIMER_TASK_PRIORITY를 설정함으로써 변경할 수 있다.

[NAME]

xTimerReset

[SYNOPSIS]

#include "timers.h"

portBASE_TYPE xTimerReset(xTimerHandle xTimer, portTickType xBlockTime);

[DESCRIPTION]

FreeRTOS의 타이머 기능은 타이머 서비스/데몬 태스크에 의해 제공된다.
FreeRTOS에서 제공하는 많은 수의 타이머 API 함수는 타이머 명령 큐라고 명명된 큐를 통해 타이머 서비스 태스크에게 명령을 전송한다. 타이머 명령 큐는 커널만 접근하도록 하기 위해 Private 속성을 가지며, 어플리케이션 코드에서의 직접적인 접근이 불가능하다. 타이머 명령 큐의 길이는 configTIMER_QUEUE_LENGTH를 설정함으로써 변경할 수 있다.

xTimerReset() 함수는 xTimerCreate() 함수를 통해 이미 생성된 타이머를 재시작 시킨다. xTimerReset() 함수가 호출되었을 때 타이머가 이미 시작되어 실행 상태에 있다면 xTimerReset() 함수가 호출된 시점을 기준으로 타이머 만료시간이 재 측정되고, 타이머가 휴면 상태에 있다면 xTimerStart() 함수와 동일한 기능을 한다.

타이머의 재시작은 타이머가 실행(Active) 상태가 되도록 한다. 타이머가 구동되는 동안 멈추거나 삭제되거나 초기화되지 않는다면, 타이머에 등록된 콜백 함수는 xTimerReset() 함수가 호출된 시점으로부터 'n'틱(타이머의 주기) 후에 호출된다.

스케줄러가 구동되기 전에 xTimerReset() 함수를 호출하는 것은 가능하지만, 실제로는 스케줄러가 구동되기 전까지 타이머는 구동되지 않으며, 타이머의 만료 시간은 xTimerReset() 함수가 호출된 시점이 아니라 스케줄러가 구동된 시점과 연관이 있다.

xTimerReset() 함수를 사용하기 위해서는 configUSE_TIMERS를 반드시 1로 설정해야 한다.

[PARAMETERS]

xTimer	:	재시작 될 타이머의 핸들
xBlockTime	:	xTimerReset() 함수가 호출될 때 이미 큐가 꽉 차 있다면 타이머 명령 큐에 타이머 재시작 명령을 성공적으로 보낼 수 있을 때 까지 대기 상태에서 태스크가 기다리게 되는 시간(틱)

[RETURN VALUES]

xBlockTime 만큼의 틱이 경과한 후에도 타이머 명령 큐에 타이머 재시작 명령이 전송될 수 없다면 pdFAIL이 리턴 된다. 타이머 명령 큐에 타이머 재시작 명령이 성공적으로 전송된다면 pdPASS가 리턴 된다. 타이머 재시작 명령이 처리되는 시점은 시스템의 다른 태스크들과 관계를 갖는 타이머 서비스/데몬 태스크의 우선순위에 의해 영향을 받는다. 타이머 서비스/데몬 태스크의 우선순위는 configTIMER_TASK_PRIORITY를 설정함으로써 변경할 수 있다.

[NAME]

xTimerStartFromISR

[SYNOPSIS]

```
#include "timers.h"

portBASE_TYPE xTimerStartFromISR( xTimerHandle xTimer,
                                  portBASE_TYPE *pxHigherPriorityTaskWoken );
```

[DESCRIPTION]

xTimerStartFromISR() 함수는 인터럽트 서비스 루틴에서 호출할 수 있는 xTimerStart() 함수의 한 버전이다.

[PARAMETERS]

xTimer	:	시작/재시작 될 타이머의 핸들
pxHigherPriorityTaskWoken	:	타이머 서비스/데몬 태스크는 타이머 명령 큐에 전달되는 메시지를 기다리고 있기 때문에 대부분 대기 상태에 있다. xTimerStartFromISR() 함수를 호출하는 것은 타이머 명령 큐에 메시지를 전송하여 타이머 서비스/데몬 태스크가 대기 상태에서 벗어날 수 있도록 한다. 타이머 서비스/데몬 태스크가 대기 상태에서 벗어날 수 있도록 하는 xTimerStartFromISR() 함수를 호출하고 타이머 서비스/데몬 태스크가 현재 실행중인 태스크(인터럽트 서비스 루틴) 보다 크거나 같은 우선순위를 가지고 있다면 *pxHigherPriorityTaskWoken이 xTimerStartFromISR() 함수 내에서 pdTRUE로 설정된다. pxHigherPriorityTaskWoken 값이 pdTRUE로 설정되면, 인터럽트가 끝나기 전에 문맥 교환이 수행된다.

[RETURN VALUES]

xBlockTime 만큼의 틱이 경과한 후에도 타이머 명령 큐에 타이머 시작 명령이 전송될 수 없다면 pdFAIL이 리턴 된다. 타이머 명령 큐에 타이머 시작 명령이 성공적으로 전송된다면 pdPASS가 리턴 된다. 타이머 시작 명령이 처리되는 시점은 시스템의 다른 태스크들과 관계를 갖는 타이머 서비스/데몬 태스크의 우선순위에 의해 영향을 받는다. 타이머 서비스/데몬 태스크의 우선순위는 configTIMER_TASK_PRIORITY를 설정함으로써 변경할 수 있다.

[NAME]

xTimerStopFromISR

[SYNOPSIS]

```
#include "timers.h"

portBASE_TYPE xTimerStopFromISR( xTimerHandle xTimer,
                                 portBASE_TYPE *pxHigherPriorityTaskWoken );
```

[DESCRIPTION]

xTimerStopFromISR() 함수는 인터럽트 서비스 루틴에서 호출할 수 있는 xTimerStop() 함수의 한 버전이다.

[PARAMETERS]

xTimer	:	생성할 태스크 함수의 포인터
pxHigherPriorityTaskWoken	:	타이머 서비스/데몬 태스크는 타이머 명령 큐에 전달되는 메시지를 기다리고 있기 때문에 대부분 대기 상태에 있다. xTimerStopFromISR() 함수를 호출하는 것은 타이머 명령 큐에 메시지를 전송하여 타이머 서비스/데몬 태스크가 대기 상태에서 벗어 날 수 있도록 한다. 타이머 서비스/데몬 태스크가 대기 상태에서 벗어날 수 있도록 하는 xTimerStopFromISR() 함수를 호출하고 타이머 서비스/데몬 태스크가 현재 실행중인 태스크(인터럽트 서비스 루틴)보다 크거나 같은 우선순위를 가지고 있다면 *pxHigherPriorityTaskWoken이 xTimerStartFromISR() 함수 내에서 pdTRUE로 설정된다. pxHigherPriorityTaskWoken 값이 pdTRUE로 설정되면, 인터럽트가 끝나기 전에 문맥 교환이 수행된다.

[RETURN VALUES]

xBlockTime 만큼의 틱이 경과한 후에도 타이머 명령 큐에 타이머 정지 명령이 전송될 수 없다면 pdFAIL이 리턴 된다. 타이머 명령 큐에 타이머 정지 명령이 성공적으로 전송된다면 pdPASS가 리턴 된다. 타이머 정지 명령이 처리되는 시점은 시스템의 다른 태스크들과 관계를 갖는 타이머 서비스/데몬 태스크의 우선순위에 의해 영향을 받는다. 타이머 서비스/데몬 태스크의 우선순위는 configTIMER_TASK_PRIORITY를 설정함으로써 변경할 수 있다.

[NAME]

xTimerChangePeriodFromISR

[SYNOPSIS]

```
#include "timers.h"

portBASE_TYPE
xTimerChangePeriodFromISR( xTimerHandle xTimer,
                           portTickType xNewPeriod,
                           portBASE_TYPE *pxHigherPriorityTaskWoken );
```

[DESCRIPTION]

xTimerChangePeriodFromISR() 함수는 인터럽트 서비스 루틴에서 호출할 수 있는 xTimerChangePeriod() 함수의 한 버전이다.

[PARAMETERS]

xTimer	:	주기를 변경할 타이머의 핸들
xNewPeriod	:	타이머의 새로운 주기 시간은 틱 주기로 정의되며 portTICK_RATE_MS는 명시된 시간을 밀리 초로 변경하는데 사용된다. 예를 들어, 타이머가 100틱 이후에 만료되어야 한다면, xTimerPeriod는 100으로 설정되어야 한다. 또 다른 방법으로 타이머가 500틱 이후에 만료되어야 한다면, xPeriod는 1000이하의 configTICK_RATE_HZ에 의해 제공된 500 / portTICK_RATE_MS로 설정 될 수 있다.
pxHigherPriorityTaskWoken	:	타이머 서비스/데몬 태스크는 타이머 명령 큐에 전달되는 메시지를 기다리고 있기 때문에 대부분 대기 상태에 있다. xTimerStartFromISR() 함수를 호출하는 것은 타이머 명령 큐에 메시지를 전송하여 타이머 서비스/데몬 태스크가 대기 상태에서 벗어 날 수 있도록 한다. 타이머 서비스/데몬 태스크가 대기 상태에서 벗어날 수 있도록 하는 xTimerStartFromISR() 함수를 호출하고 타이머 서비스/데몬 태스크가 현재 실행중인 태스크(인터럽트 서비 스 루틴)보다 크거나 같은 우선순위를 가지고 있다면 *pxHigherPriorityTaskWoken이 xTimerStartFromISR() 함수 내에서 pdTRUE로 설정된다. pxHigherPriorityTaskWoken 값이 pdTRUE로 설정되면, 인터럽트가 끝나기 전에 문맥 교환이 수행된다.

[RETURN VALUES]

xBlockTime 만큼의 틱이 경과한 후에도 타이머 명령 큐에 타이머 주기 변경 명령이 전송될 수 없다면 pdFAIL이 리턴 된다. 타이머 명령 큐에 타이머 주기 변경 명령이 성공적으로 전송된다면 pdPASS가 리턴 된다. 타이머 주기 변경 명령이 처리되는 시점은 시스템의 다른 태스크들과 관계를 갖는 타이머 서비스/데몬 태스크의 우선순위에 의해 영향을 받는다. 타이머 서비스/데몬 태스크의 우선순위는 configTIMER_TASK_PRIORITY 를 설정함으로써 변경할 수 있다.

[NAME]

xTimerResetFromISR

[SYNOPSIS]

```
#include "timers.h"

portBASE_TYPE xTimerResetFromISR( xTimerHandle xTimer,
                                  portBASE_TYPE *pxHigherPriorityTaskWoken );
```

[DESCRIPTION]

xTimerResetFromISR() 함수는 인터럽트 서비스 루틴에서 호출할 수 있는 xTimerReset() 함수의 한 버전이다.

[PARAMETERS]

xTimer	:	재시작 될 타이머의 핸들
pxHigherPriorityTaskWoken	:	타이머 서비스/데몬 태스크는 타이머 명령 큐에 전달되는 메시지를 기다리고 있기 때문에 대부분 대기 상태에 있다. xTimerStartFromISR() 함수를 호출하는 것은 타이머 명령 큐에 메시지를 전송하여 타이머 서비스/데몬 태스크가 대기 상태에서 벗어 날 수 있도록 한다. 타이머 서비스/데몬 태스크가 대기 상태에서 벗어날 수 있도록 하는 xTimerStartFromISR() 함수를 호출하고 타이머 서비스/데몬 태스크가 현재 실행중인 태스크(인터럽트 서비스 루틴)보다 크거나 같은 우선순위를 가지고 있다면 *pxHigherPriorityTaskWoken이 xTimerStartFromISR() 함수 내에서 pdTRUE로 설정된다. pxHigherPriorityTaskWoken 값이 pdTRUE로 설정되면, 인터럽트가 끝나기 전에 문맥 교환이 수행된다.

[RETURN VALUES]

xBlockTime 만큼의 틱이 경과한 후에도 타이머 명령 큐에 타이머 재시작 명령이 전송될 수 없다면 pdFAIL이 리턴 된다. 타이머 명령 큐에 타이머 재시작 명령이 성공적으로 전송된다면 pdPASS가 리턴 된다. 타이머 재시작 명령이 처리되는 시점은 시스템의 다른 태스크들과 관계를 갖는 타이머 서비스/데몬 태스크의 우선순위에 의해 영향을 받는다. 타이머 서비스/데몬 태스크의 우선순위는 configTIMER_TASK_PRIORITY를 설정함으로써 변경할 수 있다.

[NAME]

pvTimerGetTimerID

[SYNOPSIS]

#include “timers.h”

void *pvTimerGetTimerID(xTimerHandle xTimer);

[DESCRIPTION]

타이머의 ID를 리턴한다.

타이머의 ID는 xTimerCreate() 함수를 사용하여 타이머를 생성할 때 매개변수로 전달 한 pvTimerID가 할당된다.

ID는 동일한 콜백 함수가 다수의 타이머에 등록되어 있을 때 어떤 타이머가 만료되어 호출되는 것인 지 식별하기 위해 타이머 콜백 함수 내부에서 사용한다.

[PARAMETERS]

xTimer : ID를 확인 할 타이머의 핸들

[RETURN VALUES]

매개변수로 전달 받은 타이머의 ID

[NAME]

xTimerGetTimerDaemonTaskHandle

[SYNOPSIS]

#include “timers.h”

xTaskHandle xTimerGetTimerDaemonTaskHandle(void);

[DESCRIPTION]

xTimerGetTimerDaemonTaskHandle() 함수를 사용하기 위해서는
INCLUDE_xTaskGetTimerDaemon, configUSE_TIMERS를 반드시 1로 설정해야 한다.

[PARAMETERS]

None

[RETURN VALUES]

소프트웨어 타이머 서비스/데몬 태스크와 관련된 태스크의 핸들을 반환한다. configUSE_TIMERS가 1로 설정되어 있다면, 타이머 데몬 태스크는 스케줄러가 시작될 때 자동으로 생성된다.

ABSTRACT

Implementation of Embedded Software Educational Environment Using Portable Game Console

Jang, Young Jun
Major in Computer Engineering
Dept. of Computer Engineering
Graduate School, Hansung University

In this paper, we proposed an embedded software education environment with game console such like Nintendo DS through an analysis of several problems which existing education system has. We developed a console game pack with several compositions which practical training environment requires. Game software of the pack is available for downloading, running and remote-debugging to console machine. And we also developed an integrated development tool for project managing, source editing, compiling, downloading and remote-debugging. With both the game software and development tool, more real and better quantitative education environment can be provided with lower cost than before.

Keywords : Embedded Software, Game Console, IDE, Open Source, Software Education, Real-Time Operating System, Remote Debugging