연재3회 Embedded Systems에의 RTOS포팅

Intel 계열로의 포팅 실제 (x86 Core)

삼성 소프트웨어 멤버쉽 **Digital System Design Lab.**
김종수(promise@secsm.org)

이번 호부터는 실제로 RTOS(uC/OS-ii)가 Target 시스템에서 운영되도록 하기 위해 포팅이라는 작업을 할 것이다. 이를 위해 준비해야 할 제반 사항들을 알아보고 포팅과 간단한 테스트를 해보도록 하자.

포팅이란

포팅(Porting)이란 용어를 명확히 정의하기는 쉽지 않다. 그래서 나름대로 이를 정의해 보면 넓은 의미에서의 포팅이란 한 시스템에서 구현되고 있는 기능을 다른 환경의 시스템에서도 같은 기능을 수행하도록 조정해 주는 작업이라 하겠다.

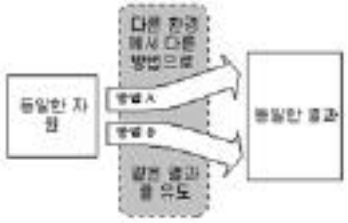

그림 1. 넓은 의미의 포팅

집에서 전기 밥솥에 쌀과 물을 넣고 버튼을 눌러 주면 밥이 만들어진다. 그러나 산에 캠핑을 간 경우 밥을 만들기 위해서는 장작과 솥이 필요하다(물론 야영장에서…). 기본적으로 쌀과 물로 밥을 만든다는 점에서 본다면 밥짓는 작업은 둘 다 같은 것이다. 그러나 전기와 밥솥 대신, 장작과 냄비를 사용한다면 방법론이 틀린 것이 된다. 즉, 같은 결과를 얻기 위해 다른 환경에서 방법을 다르게 하는 것을 포팅이라 한다.

이제 시스템 프로그래머들에게 잘 와 닿는 좁은 의미의 포팅을 이야기해 보자. 한 예로, 가장 많이 쓰이는 마이크로컨트롤러 8051의 포트에 LED가 연결되어 있고, 이 LED를 멋지게 켜는 프로그램을 짰다고 정하자. 그리고 이를 코딩이라고 정의하자. 여기서 Motorola 68K의 CPU에 이 프로그램의 이미지 파일(HEX, BIN)을 그대로 수행시킨다고 해서 동작할 것인가. 당연히 연결된 LED 구조나 내부 아키텍쳐가 틀린 데 동작할 리가 없다고 생각할 것이다. 그렇다고 이 새로운 플랫폼(시스템)을 위해 코드를 새로 다짜기를 원하지는 않을 것이다. 이전에 잘 수행되던 코드가 ASM(Assembler) 전용으로 짜여 있고 계열이 틀린 시스템이라면 거의 전반적으로 코드를 수정해야 할 것이다. 그러나 LED 켜는 알고리즘은 C같은 상위 언어로, Port 제어는 ASM같은 하위 언어로 짜여져 있다면 해당 플랫폼에 맞는 ASM 코드만을 변경함으로써 같은 기능을 쉽게 구현할 수 있을 것이다. C 언어의 이식성이란 이런 경우를 두고 하는 말이다. 좁은 의미에서의 포팅을 말하자면 이렇게 한 시스템에서 사용되는 프로그램을 다른 시스템에서도 사용할 수 있도록 시스템에 의존적인 코드만을 바꾸는 작업을 말한다. 이전 회에서 살펴보았듯이 OS를 이루는 대부분 코드들은 이식 가능한 C언어로 구현되어 있다. 다만 CPU마다 다른 구조의 아키텍쳐(레지스터, 인터럽트, 타이머 등)를 가지고 있으므로 이 부분을 사용하는 코드들을 바꾸어 내면 OS를 다른 시스템에서도 동작시킬 수 있게 된다. 이

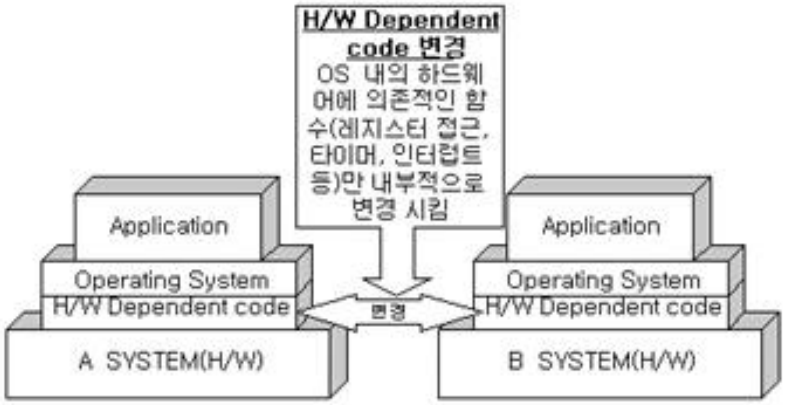

그림 2. OS에서의 포팅

OS 포팅이 되면 OS 위에서 동작하는 애플리케이션 프로그램은 플랫폼에 상관없이 실행될 수 있다. 근래에 많이 사용되는 JAVA의 특징이 플랫폼에 무관한 코딩 스타일을 지원한다는 것인데, 이를 가능하게 해주는 것도 이미 포팅과 같은 작업이 이루어졌기 때문일 것이다.

이런 전반적인 작업을 모두 소화해야 하기 때문에 OS 포팅을 위해서는 시스템의 하드웨어 구조(외부, 내부 모두)와 컴파일러, 프로그래밍에 깊은 지식이 필요하다. 이제 이 포팅을 위해서 필요한 것들을 하나하나 짚어보자.

포팅 작업을 위한 사전 준비

포팅 작업을 위해 갖춰야 할 것들을 살펴보자.

1. 플랫폼

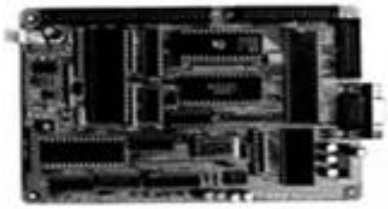

그림 3. 임베디드 플랫폼

기본적으로 Target 시스템이 있어야 한다. Target 시스템은 OS를 실을 수 있는 약간의 메모리와 동작시킬 CPU, 기타 원하는 작업을 위해 필요한 디바이스들도 구성되어 있는 어떠한 시스템이라도 상관없다. 앞으로 x86, 68K, ARM7 시스템에 포팅하는 다양한 경험을 하게 될 것이다. 특히 PC는 x86계열을 테스트하는데 가장 간편하고, 빠르게 접근할 수 있는 Target 시스템이 될 수 있다.

2. Host 측 필요장비

Target 시스템이 x86 계열의 PC라면 개발과 테스트가 동시에 같은 플랫폼 상에서 이루어지므로 컴파일러, 어셈블러들과 OS 코드, 애플리케이션 코드만 있다면 거의 모든 것이 다 갖추어진 셈이다. 만일 PC 외부로의 제어가 필요하다면 추가로 디바이스를 연결하여 설정하면 되겠지만 PC가 갖추고 있는 디바이스만으로도 충분히 테스트해 볼 수 있을 것이다. 그러나 Target이 독립적인 다른 시스템이라면 개발을 수행하는 PC 즉, Host와 Target 서로를 연결하여 줄 무언가가 필요하게 된다. 일반적으로 개발(코딩, 컴파일, 디버깅 등)은 PC측에서 이루어지고, 이 작업 결과는 Target에 전달되는 형태로 진행되는데 이 때 필요로 하는 것들이 Host측에서 제공해야 할 것들이다.

PC는 x86계열이므로 Target이 다른 계열의 시스템이라면 컴파일러, 어셈블러들이 PC상에서 동작하여 이미지를 생성하고 이것이 다른 종류의 CPU에서 인식될 수 있도록 Cross-Compile환경을 구축하든지 PC용 해당 컴파일러를 구하여 사용하여야 한다.

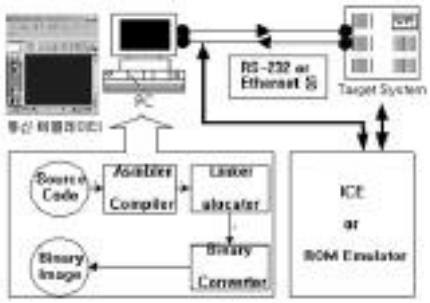

그림 4. 개발 환경 구성

보통 컴파일 환경 구축과 Binary Utility를 구하는 것이 시작에 있어서 가장 큰 걸림돌이 되기도 한다. 그래서 첫 작업은 컴파일러를 구하는 것이 된다. 보통 컴파일러들은 PC 도스환경에서 작동하도록 되어 있는 경우가 많으며 근래에 윈도용으로 재개발되어 사용이 편리하게 만들어진 컴파일러도 종종 보인다. 또 근래의 추세에 따르면 리눅스(Linux)나 유닉스(Unix)환경에서 GNU 컴파일러들을 사용하여 Cross-Compile 환경을 구축하기도 한다. 많은 방법이 있으니 개발자의 구미에 맞는 컴파일러를 구해 사용하면 된다. 단, 각 컴파일러나 어셈블러마다의 지원 환경이 다르므로 사전 조사를 철저히 할 필요는 있다.

컴파일러가 구비되면 Binary Utility 프로그램을 구해보자. Binary Image란 실행파일을 말하는 것으로 도스나 윈도의 예를 들면 프로그램을 실행시키기 위해서 사용되는 확장자가 .exe인 파일들이다. 이런 파일이 실행 이미지 파일이며 CPU가 인식하는 기계어 묶음이라고 생각하면 된다. Target 시스템에서 사용할 이미지 파일을 만들기 위해서는 Cross-Compile 이외에 몇 가지 부수적인 작업이 필요한데, 개발 환경에서 만들어진 이미지 파일을 Target에서 동작하는 Binary Image의 형태와 주소를 갖도록 바꾸는 것들이다. 이런 일들을수행하는 프로그램들을 통상 Binutil이라고 부른다. exe, bin, hex, elf 등의 파일 형태를 서로 바꾸는 프로그램들은 통신, 인터넷 등에서 구할 수 있으며, 상용 개발툴에 포함되어 있기도 하다.

통신 에뮬레이터를 살펴보자. 하드웨어 개발자들이 가장 많이 사용하는 방식은 Target 보드의 Serial Port를 RS-232통신 방식으로 개통하고 이를 이용하여 PC측에서 이미지 파일을 다운 로드하거나 디버깅을 하는 데 사용하는 것이다. Host(PC) 측에서는 시리얼 데이터를 처리하여 보여줄 수 있는 프로그램이 필요한데 PC통신에 사용하는 데이터맨이나 이야기 등의 통신 프로그램을 모뎀 환경으로 설정하여 이용하면 쉽다. 컴파일러가 도스용인 경우 위의 윈도용 통신프로그램과 오가기에 귀찮을 수도 있는데 이 경우 도스용 시리얼 통신 프로그램을 구해보도록 한다. 리눅스를 이용하여 Crose-Compile한 결과를 Target에 직접 다운 로드하기는 좀 어렵지만 리눅스용 통신 에뮬레이터를 사용할 수도 있겠다.

3. Target 측 필요 장비

위의 통신 에뮬레이터는 단지 RS-232 신호를 정보로 바꾸는 역할만을 할 뿐이다. 실제로 실행 이미지를 Target 시스템의 메모리 상으로나 실행 영역으로 옮기기 위해서 이를 관장하는 프로그램이 있어야 한다. 그러므로 Target과 Host 사이의 정보교환을 위해서 우리는 Target 측에 사용될 ROM 에뮬레이터나 모니터 프로그램을 개발하여야 한다. 기존에 개발된 시스템과 모니터 프로그램이 있다면 이를 이용하여 위의 번거로운 작업을 줄일 수 있다.

그림 5. ROM Emulator

롬 에뮬레이터란 PC측에서 롬 에뮬레이터에 Target 시스템에서 실행될 이미지를 전해주고 롬 에뮬레이터가 시스템에서 롬의 역할을 하도록 하는 장비이다. Start-up, OS 초기화 코드를 테스트하거나 프로그램 개발 전반에 이용할 수 있는 유용한 장비이다. 판매되는 제품도 있으나 일단 롬

에뮬레이터의 메모리에 롬 코드를 써넣고 타깃으로부터 롬에 접근할 수 있도록 하는 기능만 있으면 되므로 직접 만들어 사용하여도 될듯하다.

요즘 거의 모든 CPU나 컨트롤러에는 JTAG 장치라는 것이 있는데 이를 이용하여 CPU내부의 정보를 알아내거나 데이터를 전송하는데 사용할 수 있다. 사용 경험에 의하면 좀 느리다는 단점과 개성이 강한 개발자들의 구미에 맞지 않을 수도 있다는 것이다.

그림 6. ICE Set

한편, 개발을 위해 상용 Target 시스템을 산 경우 다운로드, 디버깅 환경이 갖추어진 개발도구를 제공하기도 하며 ICE(In Circuit Emulator) 등의 개발 툴을 PC에 연결하여 사용해도 된다. 근래에 Target 시스템이 네트워크를 지원하는 경우가 많아져 통신 포트로 Serial 대신 Ethernet을 사용하는 경우도 있다. 자신에게 가장 잘 맞는 개발 환경은 스스로 꾸준히 갖춰나가면 된다.

하드웨어가 검증되지 않은 상태라면 하드웨어 디버깅을 위해서 로직 어날라이저(Logic Analyzer)나 오실로스코프(Oscilloscope) 등의 측정 장비가 추가로 필요하겠지만 이제 기본적인 환경의 설정은 끝났다고 본다. 이제 한 Target 시스템에 실제로 포팅을 해보도록 하자.

x86 계열로의 포팅

x86계열이라 함은 x86은 80286부터 시작된 인텔 마이크로프로세서 시리즈의 총칭이다. 이 계열의 프로세서들은 1982년에 80286이 나온 이후로 줄곧 PC에 장착되어 오고 있다. x86 마이크로프로세서는 최근의 MMX 펜티엄 프로세서를 비롯하여, 386DX/SX/SL와 486DX/SX/DX2/SL/DX4 등을 포함하며, 인텔의 이전 시리즈인 8086/8088을 대체하는 모델이다(186의 core도 포함된다). 요즘에는 이 인텔 아키텍쳐와 호환이 되는 CPU들이 많이 등장하므로 외부적으로는 이들도 이 계열에 포함시킬 수 있을 것이다.

CPU의 버전이 올라가면서 속도와 성능은 물론 구조에서도 많은 변화가 있었지만, 애플리케이션 프로그램의 호환성을 지키기 위해 상위 프로세서를 개발할 때 하위 프로세서에서 동작하는 프로그램은 모두 동작할 수 있도록 설계하고 있다. 임베디드 시스템 개발자들에게는 386급 성능의 CPU Core가 탑재된 인텔사의 386EX, AMD사의 486급 ELAN, NEC의 V Series 등이 더욱 친숙할 것이다.

표 1. Intel Processor Series

마이크로프로세서	개략적 성 능 (MIPS)	레지스터 (bits)	버스 (bits)
8086	0.5	16	16
8088	0.5	16	8
80286	1.5	16	16
80386DX	10	32	32
80386SX	2.5	32	16
80386SL	5	32	32
80486DX	30	32	32
80486SX/SL	20	32	32
80486DX2 OverDrive	50	32	32
486DX4	70	32	32
Pentium 계열	125	32	64

이런 막강한 호환성을 이용하면 가장 친숙한 환경인 PC에서 개발한 코드로 x86계열 CPU를 무리 없이 돌릴 수 있

는 것이다. x86 임베디드 프로세서의 경우 칩의 코어는 같지만 여기에 여러 부가 블록들을 첨가한 것이 많아 레지스터 설정에서 많이 다르므로 Start-Up 코드, 레지스터 관련 설정 파일을 사용해야 된다. 그러나 프로그램측면에서 볼 때 동일한 코딩스타일과 어셈블러가 변하지 않으므로 많은 차이가 있지는 않다.

이번 x86계열에서 하는 포팅에서는 Target 시스템을 PC로 할 것이다. 가장 쉽게 테스트할 수 있는 시스템이고, 포팅이라는 것이 한 시스템에서 잘 실행되는 것을 다른 시스템에서도 잘 운영되도록 하는 것이므로 사용자에게 가장 친숙한 환경을 처음에 시도하는 것이 좋다. 도스 환경에서 uC/OS를 실행할 것이다. OS위에 OS를 실험하는 것이 이상하게 느껴질지도 모르겠지만 단순하게 생각하면 uC/OS는 그의 응용 프로그램과 같이 컴파일되어 실행 파일의 형태가 될 것이므로 도스의 입장에서 이는 단지 하나의 실행 파일이 될 뿐이고, uC/OS의 입장에서는 도스를 거쳐 시스템 자원을 사용하는 것이므로 아무런 문제가 없다. 도스는 Single TASK를 수행하는 OS이므로 이 위에서 실행시킬 uC/OS와 그 애플리케이션 프로그램들은 실시간으로 각자의 TASK가 실행되는 즉, 멀티태스킹이 되는 것을 보여줄 수 있게 되는 것이다. PC에서의 수행이 성공적이라면 다음 연재에서의 Target 시스템인 68K나 ARM으로의 이식과 비교해 그 방법을 더욱 쉽게 알 수 있게 될 것이다.

1. x86 포팅 준비 작업

그림 7. BC 3.1 IDE

컴파일러는 시스템 프로그래머라면 누구나 한 번쯤은 사용해 보았을 도스용 C 컴파일러 Boland C 3.1을 사용한다. 이 BC3.1 패키지에는 C, C++ 컴파일러와 Turbo 어셈블러가 내장되어 있어 uC/OS 코드 내의 어셈블리어와 C를 컴파일을 모두 사용할 수 있다. 또한 실행이미지를 .exe 파일로 생성하여 PC-DOS상에서 실행되므로 따로 bin-util을 사용할 필요도 없다.

타깃 보드가 같은 플랫폼이므로 이들을 잇는 통신 프로그램은 사용할 필요가 없다. 보통 BC3.1을 사용할 때 통합 환경을 이용하여(F9,Ctrl-F9) 컴파일을 하는데, 일반 PC-도스용 프로그램 개발자라면 별 문제 없겠지만, 시스템 프로그래머의 입장에서는 결코 바람직한 일이 아니다. 후에 각종 시스템 상황에 맞게 컴파일 옵션을 수정해야 할 상황이 발생할 것이므로 미리 하나하나 체크해 보고 Command Line 상에서 컴파일해 보는 것이 나중에 다른 x86 시스템을 개발할 때 도움이 될 것이다.

2. 컴파일러 환경 설정

그림은 BC3.1의 통합 환경(Integrated Development Environment)의 모습을 나타내고 있다. 이를 사용할 경우 Option->Compiler->Code Generation…과 Advanced Code Generation… 항목에서 설정하여 줄 중요한 변수가 있다. Memory Model과 Floating Point 그리고 Instruction Set들이다.

메모리 모델이란 컴파일 시 CS(Code Segment), DS(Data Segment), SS(Stack Segment), ES(Extra Segment)로 구성된 64Kbyte 메모리 영역을 어떻게 사용할 것인가를 정의하는 것으로 small, medium, compact, large 등 4가지가 있다. 메모리 모델을 정확히 이해하기 위해서는 내부 레지스터, 세그먼트, 포인터들에 대해 자세히 알아봐야 하겠지만 지면 관계상 대강 다음 표와 같은 메모리 사용 관계 정도로 알아두고 더 자세한 사항들은 웹이나 x86 아키텍쳐를 심도있게 다룬 서적들을 참고하도록 한다.

이중에서 Large 모델로 테스트할 것이다.

한편 Floating Point는 산술 연산 등에서 사용하는

표 2. Memory Model

		Code size	
		64K	16MB/4GB
Data Size	64K	Small (no overlap: total = 128K)	Medium (small data, large code)
	16MB/4GB	Compact (large data, small code)	Large (large data, large code)

Floating Point 연산 설정 옵션이다. 우리가 흔히 코프로세서라고 부르곤 하는 8087, 287, 387 등과 같은 칩들의 정식 명칭은 FPU(Floating Point Unit) 혹은 NDX (Numeric Processor Extention)으로써 이 칩을 추가하여 수치연산 기능이 확장된 시스템을 구성할 수 있게 된다. CPU 내부에는 부동 소수점 연산을 할 수 있는 장치가 되어있지 않기 때문에 이를 추가적으로 사용하여야 하겠지만 우리가 옵션으로 설정한 에뮬레이터(emulator)라는 것을 이용하여 부동 소수점 연산 장치 없이 연산을 할 수 있도록 한다. 이 경우 자동으로 이러한 에뮬레이터 루틴을 exe 화일에 링크시켜 주는데 대신 간단한 프로그램도 실행 이미지 크기가 커지는 단점이 있다.

마지막으로 Instruction Set의 경우 80186용 instruction을 이용하도록 할 것이다. 호환성에 의해 상위 모든 CPU에서 인식될 수 있기 때문에 이것으로 설정하면 x86 계열의 다른 CPU에서도 컴파일 에러가 나지 않게 된다.

기타 Optimization 항목들도 한 번 정도 살펴보면 좋을 것이다.

3. OS 코드

이제 프로그램을 짜서 컴파일하는 일만 남았다. OS 코드와 애플리케이션 코드를 하나로 묶어 컴파일 할 것이다. 앞에서 언급했듯이" MicroC/OS-II, The Real-Time Kernel"이라는 책의 부록이나 PC통신 동호회(예:하이텔 디지털 동호회: go dig) 자료실에서 OS 소스 코드를 구할 수 있다. 이 자료에는 uC/OS-II를 구성하는 소스 코드이외에 x86 시스템의 Hardware Dependent Code와 예제로써 x86 large 모델 소스도 포함되어 있다.

컴파일하기 전에 프로그램 소스의 구조를 살펴보면, 애플리케이션 프로그램은 여러 개의 TASK 프로그램들의 집합으로써 각각의 TASK는 독립적으로 운영되도록 프로그램된 함수들의 집합이다. 각 TASK는 함수로 등록되어 TASK Running시 해당 함수가 실행된다. 프로세서 독립적인 코드 리스트는 전회에서 살펴본 여러 함수들을 종류별로 나누어 놓은 파일들의 집합이다.(OS_CORE.C, OS_Q.C, OS_MBOX.C OS_SEM.C, OS_MEM.C, OS_TASK.C, OS_TIME.C, uCOS_II.C, uCOS_II.H)스케줄러, IPC, TASK, Timer 관리를 위한 OS의 일부들로서 C언어로 작성되었으며 하드웨어에 상관없이 수행되는 일종의 알고리즘과 같은 것들이다. 애플리케이션 프로그램과 OS 코드를 같이 컴파일하게 되므로 이들 사이

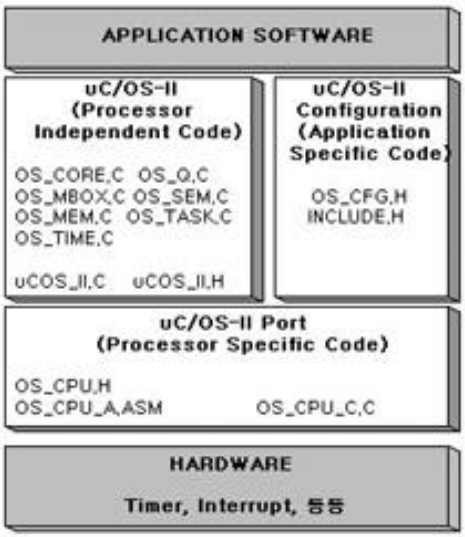

그림 8. uC/OS S/W 아키텍쳐

의 함수 연결이라든지 TASK가 OS를 이용할 때의 제약들을 정의하기 위해 Configuration이 필요하다(OS_CFG.H, INCLUDE.H).

한편 OS에서 이용하는 하드웨어 자원들(Timer, Interrupt, Memory 등)의 경우 이들을 접근하는데 시스템마다 환경 설정 및 접근 방법이 다르므로 시스템에 맞추어 설정해야 한다.

보통 어셈블러 같은 저급 언어로 되어 있으며, Inline ASM을 사용하여 C와 혼용해서 사용하기도 한다(OS_CPU.H, OS_CPU_A.ASM, OS_CPU_C.C).

사실 이 Processor Dependent Code를 잘 바꾸어 주는 것이 운영체제 포팅의 전부라 할 수 있다. uC/OS에서는 위의 세 파일에서 인터럽트, 타이머, Context Switch 등의 기능을 구현하고 있다(ASM 파일이 Context Switch 등의 하위 중요 함수를 담고 있다).

4. Hardware Dependent Code 수정

하드웨어가 달라짐에 따라 변경되어야 하는 코드

· OS_CPU.H : 시스템에서 사용할 데이터 타입을 정의하고 Critical Section에서 사용할 인터럽트 사용에 대해 정의한다.

· OS_CPU_A.ASM : _OSStartHighRdy, _OSCtxSw, _OSIntCtxSw, _OSTickISR 등 최하위 기능 함수들을 정의한다. 프로세서에 따라 달라지는 Context Switching, Timer 등을 수정하여야 한다.

· OS_CPU_C.C : TASK가 생성되어 초기화될 때 스택 영역을 설정, 초기화하고 레지스터 영역을 잡는 OSTaskStkInit함수를 포함하고 있다. 또한 xxxHOOK함수를 이용하여 전처리를 수행할 수 있도록 빈 블록이 제공된다. 보통 확인을 위한 코드를 이곳에 넣으면 쉽게 구현할 수 있다.

이제 바꾸어야 할 파일과 함수들을 파악했으므로 이들을 현 시스템에 맞게 수정하여야 한다. 이번에는 PC가 Target 시스템이므로 x86 아키텍쳐에 의해 이들 코드가 어떻게 만들어졌는지만 잘 관찰하면 된다. 다음 다른 아키텍쳐로의 포팅 시에 비교 대상이 될 것이다.

(1) OS_CPU.H

설정 사항은 다음과 같다.

소스 1. OS_CPU.H 파일

```
typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;
typedef signed   char  INT8S;
typedef unsigned int   INT16U;
typedef signed   int   INT16S;
typedef unsigned long  INT32U;
typedef signed   long  INT32S;
typedef float          FP32
typedef double         FP64;
typedef unsigned int   OS_STK;
#define BYTE           INT8S
#define UBYTE          INT8U
#define WORD           INT16S
#define UWORD          INT16U
#define LONG           INT32S
#define ULONG          INT32U
```

```
#define OS_CRITICAL_METHOD    2

#if     OS_CRITICAL_METHOD == 1
#define OS_ENTER_CRITICAL()  asm  CLI
#define OS_EXIT_CRITICAL()   asm  STI
#endif

#if       OS_CRITICAL_METHOD == 2
#define OS_ENTER_CRITICAL() asm {PUSHF;
CLI}
#define OS_EXIT_CRITICAL()  asm  POPF
#endif

#define  OS_STK_GROWTH        1
#define  uCOS                 0x80

#define  OS_TASK_SW() asm  INT uCOS
OS_CPU_EXT  INT8U  OSTickDOSCtr;
```

· Data Types : 각기 다른 길이의 데이터형을 갖는 프로세서로 포팅을 위해 데이터형을 일관하게 정의한다.

· Critical Sections : Context Switching과 같은 Critical Section에서 동작할 작업의 전후에 하는 인터럽트 처리 작업을 Inline ASM으로 정의하여 사용한다. 이 ASM 코드는 x86계열에서 인터럽트와 스택에 관여하는 명령어이다.

· Stack Growth : x86에서는 상위에서 하위 메모리로 증가하므로 1로 설정한다.

· Vector Number : Context Switching을 위한 인터럽트 벡터 번호를 설정한다.

· TimeTick : DOS에서 사용하는 Timer 인터럽트를 사용한다.

(2)OS_CPU_A.ASM

어셈블러 코드로 이루어져 있으며 멀티태스킹에 가장 중요한 Context Switching 함수를 가지고 있다. 앞으로 나오는 함수들을 이해하려면 매크로 어셈블러를 참고하면 된다. 또한 이 코드이외에 하드웨어와 밀접한 관계에 있는 함수들을 이해하기 위해서는 x86 아키텍쳐에 대해 정확히 알고 있어야 한다. x86 아키텍쳐는 CPU 중 대표적인 예로 많이 사용하는 것이어서 관련된 정보가 많을 것으로 여겨 여기서는 자세히 다루진 않으나 포팅에 필요한 부분은 따로 언급할 것이다.

그림 9는 8086의 내부 연산구조를 나타내고 있다. x86도 이의 기본 구조를 따르므로 레지스터 구조만 살펴보자. 표 3에서 보는 바와 같이 14개의 내부 중요 레지스터들이 있다. 한 TASK가 실행되고 있을 때 이들 레지스터들을 사용하고 있으므로 다른 TASK로 전환할 때에는 이들 레지스터들을 저장하고 새로운 TASK에 해당하는 각 레지스터 값들을 불러들여야 한다. 이것이 Context Switch의 간단한 의미이다. 이에 관계된 함수들을 살펴보자.

표 3. 80x86 Real-mode Register Map

15...8	7...0	NAME
AH	AL	General-Purpose Registers
BH	BL	
CH	CL	
DH	DL	
BP		Pointers
SP		
SI		Index Registers
DI		
IP		Instruction Pointer
SW		Status Word
CS		Segment Registers
SS		
DS		
ES		

· OSStartHighRdy() : 이 함수는 OSStart()에서 호출이 되고 최우선순위 TASK를 준비시킨다. 그에 앞서 OSInit(), OSTaskCreate()나 OSTaskCreateExt()가 실행된다.

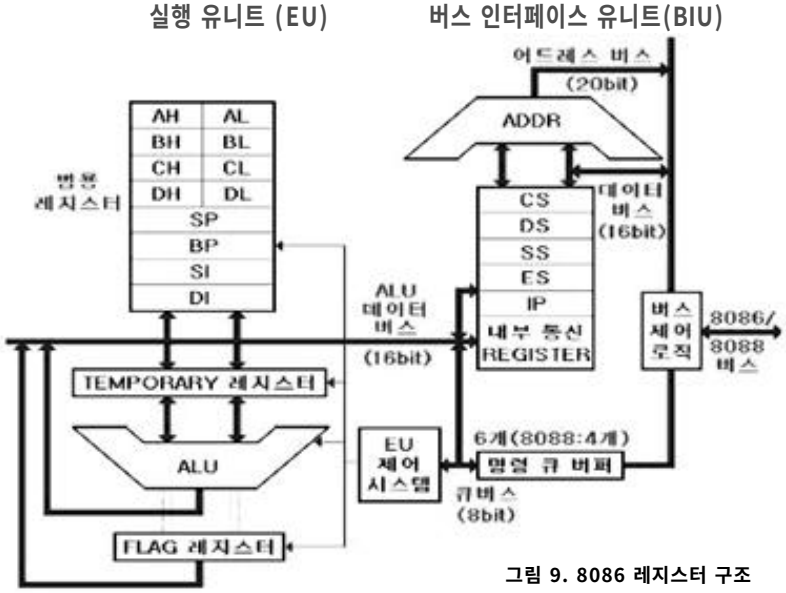

그림 9. 8086 레지스터 구조

소스 2. OSStartHighRdy 함수

```
_OSStartHighRdy  PROC FAR

 MOV    AX, SEG _OSTCBHighRdy
 MOV    DS, AX

  CALL   FAR PTR _OSTaskSwHook

  INC    BYTE PTR DS:_OSRunning

 LES              BX,    DWORD    PTR
DS:_OSTCBHighRdy
 MOV    SS, ES:[BX+2]
 MOV    SP, ES:[BX+0]
 POP    DS
 POP    ES
 POPA
IRET
_OSStartHighRdy  ENDP
```

위의 코드를 보면 OSTCBHighRdy함수는 다음에 실행되기 위해 대기 중인 TCB의 세그먼트를 데이터 세그먼트에 설정하고 OSTaskSwHook 함수를 호출한다. 이 함수는 OS_CPU_C.C에 정의되어 있는 빈 함수로 이곳에 사용자가 특별히 수행하고 싶은 루틴을 넣을 수 있다. 이후 OSRunning을 1로 설정하고 TASK의 스택 포인터를 읽어들이는 과정을 통하여 현재 선택된 TASK의 각종 레지스터 값들을 설정한다. POPA, PUSHA 등은 매크로 함수로써 여러 레지스터들을 저장, 읽어 들이는 명령어이다.

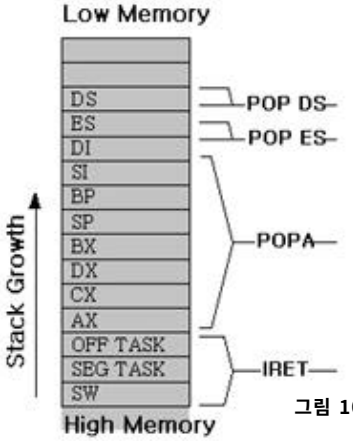

그림 10. Stack Frame

스택에는 그림 10의 데이터 구조와 같이 저장되고 읽혀져 TASK를 수행하는데 사용된다.

이제 가장 중요한 함수 OSCtxSw()와 OSIntCtxSw()를 살펴보자.

· OSCtxSw() : TASK간의 전이가 실제로 일어나는 곳으로 높은 우선순위의 작업으로 교체된다.

· OSIntCtxSw() : ISR(Interrupt Service Routine)의 OSIntExit()에서 호출되고 가장 높은 우선순위의 Task를 취하여 Context Switch를 한다.

다음의 TASK-Level Context Switch와 Interrupt-

소스 3. OSCtxSw, OSIntCtxSw 함수

```
_OSCtxSw PROC   FAR
  PUSHA
  PUSH   ES
  PUSH   DS

  MOV    AX, SEG _OSTCBCur
  MOV    DS, AX

  LES    BX, DWORD PTR DS:_OSTCBCur
  MOV    ES:[BX+2], SS
  MOV    ES:[BX+0], SP

  CALL   FAR PTR _OSTaskSwHook

  MOV              AX,    WORD    PTR
DS:_OSTCBHighRdy+2
  MOV              DX,    WORD    PTR
DS:_OSTCBHighRdy
  MOV    WORD PTR DS:_OSTCBCur+2, AX
  MOV    WORD PTR DS:_OSTCBCur, DX

  MOV    AL, BYTE PTR DS:_OSPrioHighRdy
  MOV    BYTE PTR DS:_OSPrioCur, AL
  LES              BX,    DWORD    PTR
DS:_OSTCBHighRdy
  MOV    SS, ES:[BX+2]
  MOV    SP, ES:[BX]

  POP    DS
  POP    ES
  POPA
  IRET
_OSCtxSw    ENDP
```

```
_OSIntCtxSw PROC FAR
ADD    SP,10
_OSIntCtxSw ENDP
```

Level Context Switch 함수에서는 전반부를 제외하고 코드 내용이 같다. 먼저 OSCtxSw에서는 처음에 DS, ES, DI,SI, BP, SP, AX, BX, CX, DX의 레지스터를 저장한다. TASK 전환을 위해서 일단 연산중인 정보를 저장해야 하기 때문이다. 반면 OSIntCtxSw에서는 이미 ISR에서 레지스터들을 저장하였으므로 이의 과정이 필요치 않다. 대신 TASK의 Context만 포함하는 위치를 지정하기 위해 스택포인터를 이동시켜야 한다.

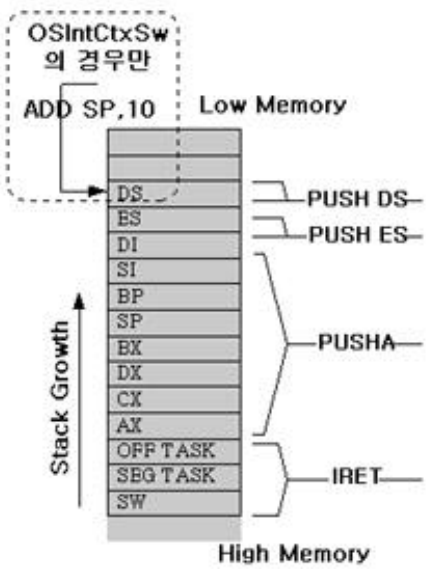

그림 11. OSCtxSw, OsintCtxSw의 초기부분

이후 두 함수 모두 공통적인 구조를 가지는데 SS와 SP를 OS_TCB에 저장한 후 OSTCBHook을 호출하고 OSTCBCur라는 현재 TCB(Task Control Block)의 포인터를 다음에 수행할 TASK의 포인터에 맞추고 OSPrioCur(현재 TASK의 우선순위)도 재설정한다. 이제 TCB에서 SS와 SP를 읽어 내고 새로이 수행될 TASK의 각종 레지스터를 읽어 설정한다. 이로써 Context Switch가 이루어지게 된다.

정리하자면 Context Switch라는 것은 멀티태스킹이 가능한 시스템에서 한 TASK가 CPU를 점유하여 사용하다가 다른 TASK에게 작업의 권한을 넘겨줄 때 연산에 사용 중이던 각종 레지스터 정보를 스택에 저장하고 새로이 등장한 TASK의 기존 레지스터 정보를 불러들이는 작업을 수행하는 것을 의미한다. 다른 계열의 시스템에서도 역시 사용되는 레지스터 항목들의 저장과 읽어 들이는 일을 수행하면 된다.

· OSTickISR() : 멀티태스킹 후에 ticker interrupt는 가장 중요해진다. Tick은 OS의 움직이는 심장으로써 시스템의 시계 역할을 한다.

표 4. PC Interrupt Vector Table

이전 DOS 사용		uC/OS의 DOS 사용	
0x05	...	0x05	...
0x06		0x06	
0x07		0x07	
0x08	DOS Tick Handler(18.20648Hz)	0x08	OSTickISR() (200Hz)
0x09	...	0x09	...
		0x80	OSCtxSw()
		0x81	DOS Tick Handler

첫 TASK가 실행되면 OSTickISR()가 수행되며, 이 시스템에서는 도스위에서 움직일 것이므로 도스의 인터럽

소스 4. OSTickISR 함수

```
_OSTickISR  PROC   FAR
    PUSHA
    PUSH   ES
    PUSH   DS
    MOV    AX, SEG _OSTickDOSCtr
    MOV    DS, AX
    INC    BYTE PTR _OSIntNesting
    DEC    BYTE PTR DS:_OSTickDOSCtr
    CMP    BYTE PTR DS:_OSTickDOSCtr, 0
    JNE    SHORT _OSTickISR1
    MOV    BYTE PTR DS:_OSTickDOSCtr, 11
    INT    081H
    JMP    SHORT _OSTickISR2

_OSTickISR1:
```

```
    MOV    AL, 20H
    MOV    DX, 20H
    OUT    DX, AL

_OSTickISR2:
    CALL   FAR PTR _OSTimeTick
    CALL   FAR PTR _OSIntExit
    POP    DS
    POP    ES
    POPA
    IRET

_OSTickISR  ENDP
```

트를 사용한다. 단 도스의 Tick 주기가 너무 빠르므로 이를 200Hz정도로 맞추기 위해서 11배정도 늦추는 작업을 수행하고 있다.

이 코드를 살펴보면, 전과 같이 일단 CPU에서 사용하던 레지스터들을 저장하고 Interrupt Nesting을 시킨 후 OSTimeTick을 호출하여 Waiting, Pending된 TASK의 값을 업데이트하고 있다.

이 후 도스가 8059 인터럽트 칩을 사용하여 제어하는 Tick을 11회 루프시키고 인터럽트에서 빠져나와 다시 전의 레지스터들을 복원시킨다.

여기서 주목해야 할 것들이 있다.

x86 계열의 인터럽트 처리 방식을 보면 어드레스의 첫 부분을 인터럽트 벡터 핸들러 등록 부분으로 사용한다는 점이다. 즉, OSTickISR과 OSCtxSw 함수들은 특정 하드웨어 인터럽트(Timer 등) 발생시 이를 처리해주는 인터럽트 핸들러로 직접 등록하고 있음을 주의깊게 살펴보아야 할 것이다.

정리하자면 시스템의 타이머 인터럽트를 사용하여 주기마다 인터럽트가 발생되고 이의 핸들러로 직접 OSTickISR이 사용된다.

이 함수에서는 OS에서 사용하는 Tick값을 증가시키는 역할을 하게 되고, 이 경우 PC의 자원을 이용하기 위해 도스의 주기적 시스템 타이머를 빌어 사용하는 순서를 밟고 있다.

(3) OS_CPU_C.C

이 파일에서 중요한 함수로는 TASK의 스택을 초기화시키는 함수가 있고 그 이외의 함수들은 OSxxxHook들로 사용자가 코드를 추가할 수 있는 함수들이므로 설명을 생략한다.

· OSTaskStkInit() : OSTaskCreate()와 OSTaskCreateExt()에 의해 불려지고 TASK의 스택 프레임을 초기화한다. TASK가 생성되면 TASK의 시작번지(task), 포인터(pdata), TASK의 Top of Stack(ptos)와 TASK의 우선순위(prio)를 OSTaskCreate() 또는 OSTaskCreateExt()에 돌려주는 함수이다.

소스 5. OSTaskStkInit 함수

```
void *OSTaskStkInit (void (*task)(void
*pd), void *pdata, void *ptos, INT16U
opt)
{
    INT16U *stk;
    opt    = opt;
    stk    = (INT16U *)ptos;
    *stk-- = (INT16U)FP_SEG(pdata);
    *stk-- = (INT16U)FP_OFF(pdata);
    *stk-- = (INT16U)FP_SEG(task);
    *stk-- = (INT16U)FP_OFF(task);
    *stk-- = (INT16U)0x0202;
    *stk-- = (INT16U)FP_SEG(task);
    *stk-- = (INT16U)FP_OFF(task);
    *stk-- = (INT16U)0xAAAA;
    *stk-- = (INT16U)0xCCCC;
    *stk-- = (INT16U)0xDDDD;
    *stk-- = (INT16U)0xBBBB;
    *stk-- = (INT16U)0x0000;
   *stk-- = (INT16U)0x1111
   *stk-- = (INT16U)0x2222;
   *stk-- = (INT16U)0x3333;
   *stk-- = (INT16U)0x4444;
   *stk   = _DS;
   return ((void *)stk);
}
```

위의 초기화 과정을 통하여 스택의 영역을 확보하고 임의의 값을 설정하는데, 그 결과 다음과 같이 스택영역에 한 TASK를 위한 구조적 메모리 영역이 설정되게 된다.

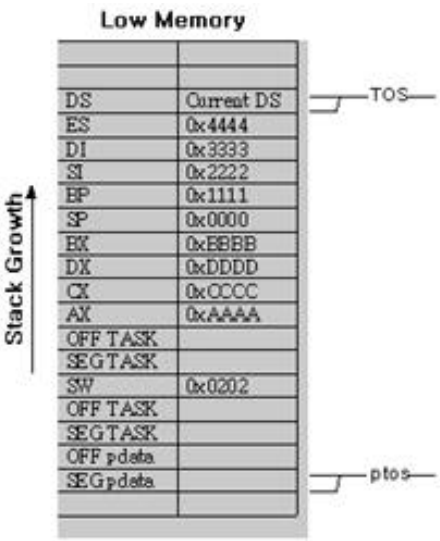

그림 12. 스택 초기화 모습

TASK 생성과 컴파일

이제 하드웨어에 의해 수정하여야 할 코드들은 거의 수정하였다. 이제 테스트를 위해 실험용 TASK들을 만들어 OS와 연결시키고 컴파일해 보자.

1. 실험용 TASK 생성

이제 아주 간단한 TASK 몇 개를 만들어 테스트를 해 볼 것이다. 일단 최상위 프로그램의 이름을" Maintest.c"라고 하자.

소스 6-1. Maintest.c 파일 전반부

```
#include "includes.h"
#define   TASK_STK_SIZE 512
#define   N_TASKS       10

OS_STK
TaskStk[N_TASKS][TASK_STK_SIZE];
OS_STK TaskStartStk[TASK_STK_SIZE];
char   TaskData[N_TASKS];
OS_EVENT *RandomSem;

void   Task(void *data);
void   TaskStart(void *data);
```

이 파일을 살펴보면 처음 Include.h라는 파일을 통해서 컴파일러가 사용 가능한 C 라이브러리를 설정하고, OS관련 소스를 포함시킨다. 또한 TASK 프로그램을 지원할 드라이버 레벨의 함수들을 정의한 파일, 하드웨어 관련 함수들도 연결한다. 이후 각 TASK의 스택관련 배열을 설정하고 사용될 함수를 정의한다.

소스 6-2. Maintest.c Main함수

```
void main (void)
{
    OSInit();
    PC_DOSSaveReturn();
    PC_VectSet(uCOS, OSCtxSw);
    RandomSem = OSSemCreate(1);
    OSTaskCreate(TaskStart, (void *)0, (void
*)&TaskStartStk[TASK_STK_SIZE - 1], 0);
    OSStart();
}
```

프로그램의 첫 시작 부분이다. uC/OS의 사용방법을 정의하는 것이나 마찬가지이므로 주의깊게 살펴보아야 한다. OSInit()함수를 통해 uC/OS를 초기화시키는 것이 첫 번째이다. 다음으로 PC_DOSSaveReturn()함수를 호출하여 도스에서 사용하는 타이머의 인터럽트 벡터에 인터럽트 핸들러를 등록시킨다. PC_xxx()라는 함수는 이 OS를 제공하는 디렉토리 내에서 특별히 x86 PC 시스템을 사용하기 위해 만든 드라이버 레벨의 함수들로써 모니터 화면과 Timer를 사용할 수 있도록 짜여져 있다. 이들은 PC.C, PC.H라는 파일에 정의되어 있다. 임베디드 시스템에서 간혹 LCD등의 Display를 사용하는데 이들의 저 수준 제어 함수를 만들고 이를 일반 TASK에서 사용하면 같은 방식이 된다.

이후 OSCtxSw()함수의 인터럽트 핸들러 등록을 시도하고 세마포어를 하나 사용할 수 있도록 생성하였다. 드디어 OSTaskCreate()함수를 사용하여 하나의 TASK를 등록하였다. 그리고 Osstart()함수를 실행하면 uC/OS가 x86시스템 위에서 멀티태스킹과 실시간 처리 TASK를 하기 시작한다. 위의 예에서 보면 이 TASK의 우선순위는 0으로 가장 높게 되어 있고, 실행될 때 실제로 TaskStart

()라는 함수의 내용을 수행하게 된다.

여기서 하나 더 고려하자면 OS의 심장인 Tick이 카운드되고 있지 않다. 일반적인 구조로 볼 때 Main()함수에서는 위와 같이 하나의 TASK를 등록한 후 그 TASK 내에서 Tick을 설치하여 기동시키고, 다시 또 다른 TASK를 등록하는 방법을 자주 사용하고 있다. 이를 구현해보자.

소스 6-3. Maintest.c 파일 TaskStart함수

```
void TaskStart (void *data)
{
    UBYTE  i;
    WORD   key;

     OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);
    PC_SetTickRate(OS_TICKS_PER_SEC);
    OS_EXIT_CRITICAL();

     OSTaskCreate(Task,(void)&TaskData[i],(voi
d *)&TaskStk[i][TASK_STK_SIZE - 1], i + 1);

  for (;;) {
       if (PC_GetKey(&key) == TRUE) {
            if (key == 0x1B) {
                PC_DOSReturn();
            }
       }
       OSTimeDlyHMSM(0, 0, 1, 0);
  }
}
```

이 함수는 처음으로 등록된 함수이고 여기서 주기 타이머 인터럽트를 등록시키고 기동한다. 그 후 TASK를 생성하게 되는데 그 TASK의 이름이 Task라고 되어 있으며 우선순위는 0보다는 낮게 잡힐 것이다. 이 함수 역시 무한 루프를 돌며 수행되고, 주기적으로 키 값을 검사하고 있다.

실제 실행 코드가 있는 Task() 함수에서는 세마포어가 사용되어 외부에서 전달된 신호를 지속적으로 감시하고 있으며, 주기적으로 화면에 무언가를 찍는다. 이러한 TASK

소스 6-4. Maintest.c 파일 Task함수

```
void Task (void *data)
{
    UBYTE x;
    UBYTE y;
    UBYTE err;

    for (;;) {
        OSSemPend(RandomSem, 0, &err);
        x = random(80);
        y = random(16);
        OSSemPost(RandomSem);

  PC_DispChar(x,  y  +  5,  *(char  *)data,
DISP_FGND_LIGHT_GRAY);
     OSTimeDly(1);
    }
}
```

는 56개 이내에서 만들 수 있으며 이들은 각각 우선순위를 지켜가며 독립적으로 수행되고 있다.

이러한 TASK 구조의 장점은 다른 이벤트를 고려할 필요없는 독립적 프로그램 구조를 이루어 프로그램이 간단히 작성될 수 있다는 점이다.

2. 컴파일과 환경 설정

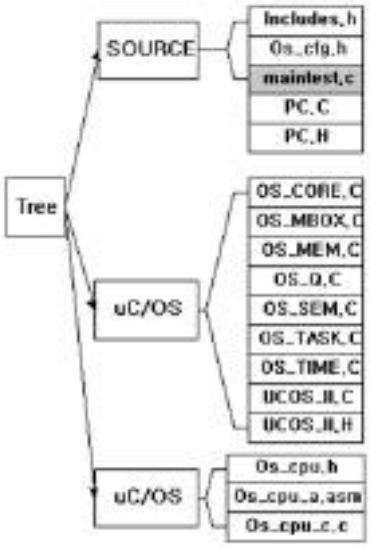

그림 13. 컴파일할 파일 구조

OS_CFG.H파일에서는 TASK관련 함수의 사용여부와 IPC 사용여부를 결정하고 TASK수나 최고 우선 순위의 한계를 명시하는 일을 한다.

UCOS_II.C, .H 파일은 OS_xxx.C파일들의 컴파일 경로와 각 부분의 사용 여부에 따른 구조체의 설정을 조정하는 일을 하므로 이들은 TASK 일의 종류에 따라 다르게 설정될 것이다.

이제 지금까지 만들어진 파일을 가지고 컴파일을 해보자. PC환경에서는 컴파일에 대한 제약이 크지 않으므로 통합환경을 이용해도 된다.

BC3.1의 통합환경에서 Project -> Open Project… 에서 프로젝트 이름을 설정하고 Project -> Add Item… 을 통해 연관된 파일을 링크시킨다.

등록할 파일들은 앞의 예의 경우 MAINTEST.C OS_CPU_C.C, OS_CPU_A.ASM, PC.C, UCOS_II.C 파일들이다. 이들의 경로가 제대로 설정되지 않았다면 에러를 발생할 것이며, Library, Include 파일들의 경로

```
.AUTODEPEND

.PATH.obj = OBJ

CC = bcc +MAINTEST.CFG
TASM = TASM
TLIB = tlib
TLINK = tlink
LIBPATH = C:₩bc31₩LIB
INCLUDEPATH = C:₩bc31₩INCLUDE

.c.obj:
  $(CC) -c {$< }

.cpp.obj:
  $(CC) -c {$< }
EXE_dependencies =  ₩
 os_cpu_a.obj ₩
 os_cpu_c.obj ₩
 MAINTEST.obj ₩
 pc.obj ₩
 ucos_ii.obj
```

```
obj₩MAINTEST.exe:
 MAINTEST.cfg $(EXE_dependencies)
  $(TLINK)      /v/s/c/P-/L$(LIBPATH)
@&&|
c0l.obj+
obj₩os_cpu_a.obj+
obj₩os_cpu_c.obj+
obj₩MAINTEST.obj+
obj₩pc.obj+
obj₩ucos_ii.obj
obj₩MAINTEST,obj₩MAINTEST
emu.lib+
mathl.lib+
cl.lib
|
*
os_cpu_a.obj:
MAINTEST.cfg ..₩x86l₩os_cpu_a.asm
$(TASM)          /MX          /Zi
/O ..₩X86L₩OS_CPU_A.ASM,OBJ₩OS
_CPU_A.OBJ

os_cpu_c.obj:
MAINTEST.cfg ..₩x86l₩os_cpu_c.c
                          $(CC) -c ..₩

MAINTEST.obj:        MAINTEST.cfg
MAINTEST.c

pc.obj: MAINTEST.cfg ..  ₩source₩pc.c
$(CC) -c ..  ₩source₩pc.c

ucos_ii.obj:
MAINTEST.cfg ..₩source₩ucos_ii.c
$(CC) -c ..₩source₩ucos_ii.c

MAINTEST.cfg: makefile.mak
  copy &&|
-ml -1 -v -G -O -Og -Oe -Om -Ov -Ol -
Ob -Op -Oi -Z -k -vi- -wpro -weas -
wpre -nOBJ
-I$(INCLUDEPATH)
-L$(LIBPATH)
-P-.C
```

도 맞아야 한다.

이외의 경로 설정은 Include.h, UCOS_II.C에 수록되어 연결될 것이다. 통합환경에서 이들의 컴파일은 F9에 의해 Maintest.exe 파일을 만드는 것으로 도스용 실시간 처리 프로그램의 작성이 모두 끝나게 된다.

앞에서도 언급했지만, 이런 편한 통합 환경은 시스템 프로그래머에게는 좋지 않은 습관이 될 수 있으므로 모든 환경 옵션을 알 수 있는 Command Line 컴파일을 권장한다.

보통 .bat 파일이나 Makefile을 만들어 사용하는데 이번 컴파일의 Makefile을 예로 제시하고 이번 호를 마치겠다. Makefile의 자세한 예는 다음 호에서 다룰 임베디드용 프로세서 컴파일 시에 더 자세히 다루도록 하겠다.

Boland C를 컴파일 하는데 이렇게 복잡한 과정을 거치는지 몰랐을 독자들도 있을 것이다.

일반 임베디드 CPU 컴파일처럼 이러한 것들을 알아두기는 해야 할 것 같다(사실 Makefile이 bat 파일보다 복잡하지만 기능은 더 막강하다).

마치며

이번 호에서는 PC를 Target 시스템으로 하여 uC/OS를 테스트 해보았다. 처음 이러한 OS를 접할 때의 어려움을 생각하여 처음은 쉽게 접근할 수 있도록 PC를 사용하였다. 이번 호는 uC/OS를 68K, ARM7 임베디드 시스템으로 포팅할 다음 호 연재의 사전 답사로 생각하자.

그리고 기회가 닿는다면 386EX와 같은 x86 임베디드 컨트롤러에 포팅하는 기회를 가져보도록 하겠다.

다음 호에서는 68K Core에 Communication Processor가 장착된 임베디드 컨트롤러인 MC68302에 uC/OS를 포팅하는 과정을 살펴볼 것이다.

-- 참고 사항 --

일반적으로 실시간 운영체제를 포팅해 줄 때 가장 중요한 것은 memory, interrupt, timer부분 등이다. 메모리에서는 주로 Kernel이 쓸 stack을 잡아주는데 C프로그램을 작성하는 경우 code(각종 초기화 변수, 초기 stack, heap)와 data(프로그램 수행시의 변수 등) 영역의 구분에 주의를 기울여야 한다. 컴파일 후 링크 시에 설정한다. stack영역이 확보되면 다시 Kernel용과 Task용으로 분리한다. 이것이 핵심적인 작업 중의 하나이다.

Interrupt의 경우 Signal이 발생되면 일반적으로 인터럽트 벡터를 이용하여 관련 처리를 수행한다. Timer는 시스템에서 쓰는 Timer마다 다르며 이의 초기화가 잘 이루어지지 않으면 운영체제에서 제공하는 함수 중에서 Time에 관련된 작업을 수행할 수 없게 되어 정확한 시간을 요구하는 실시간 시스템의 의미가 깨지게 될 뿐만 아니라 OS의 운영 자체가 불가능해 지기도 한다.

uC/OS-II를 실행하기 위해서는 프로세서가 재진입이 가능한 코드를 생성하는 컴파일러가 필요하며 C에서 인터럽트를 동작이 가능하고, 프로세서는 인터럽트를 지원하거나 비슷한 기능을 수행할 수 있어야 하며, 일정한 Timer 기능의 인터럽트를 제공할 수 있어야 한다. 프로세서는 많은 데이터를 수용할 수 있는 하드웨어적, 소프트웨어적 스택을 지원해야 하며, 스택이나 메모리에서 스택 포인터와 다른 CPU 레지스터를 불러오고 저장하는 Instruction을 갖아야 한다. Data Types의 설정이나 중요한 부분에 접근하기 위한 매크로로 사용되는 인터럽트 금지와 해제의 구현에 주의하여야 한다.