연재4회 Embedded Systems에의 RTOS포팅

Motorola 계열로의 포팅 실제(68K Core)

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

전회에 이어 이번엔 Motorola 68K 시리즈의 한 칩인 MC68302로의 임베디드 RTOS 포팅이다. 전회에서는 PC를 타깃 시스템으로 하여 RTOS를 포팅하고 동작을 관찰했었다. 이번에는 임베디드 시스템에 RTOS를 올리는 실전에 들어가겠다. 전에 PC에서의 테스트와 비교하면 이해하기도 훨씬 쉬울 뿐 아니라 Processor Dependent한 부분을 바꾸는 작업이 정확히 어떠한 것들을 필요로 하는지 알게 될 것이다. 포팅의 배경과 절차에 대해서는 전회에서 설명하였으므로 이번 회부터는 바로 타깃 프로세서, 시스템에 대한 설명을 시작으로 환경 설정, 포팅 과정을 실행 해보자.

포팅 사전 준비

먼저 포팅할 타깃에 대해 알아보자.

전에도 언급했듯이 OS 포팅은 새로운 기능을 하도록 프로그램을 짜는 것이 아니라 같은 기능을 다른 시스템에서 할 수 있도록 맞추는 작업임을 생각해 본다면 바뀌게 되는 이 시스템에 대해 아주 자세히 알아야 된다는 것을 직감할 수 있을 것이다.

여기서는 타깃에 대해 개략적인 것만을 설명하겠지만 적극 활용하고자 하는 사람은 꼭 CPU 유저 메뉴얼이나 관련 서적 등을 통해 기본 지식을 습득하기 바란다.

1. MC68302란

MC68302는 Motorola 사에서 제공하는 Integrated Multi-Protocol Processor(IMP)로서, M68000 core에 통신을 위한 RISC 프로세서를 따로 가지고 있는 구조이다. 통신을 위한 프로세서가 따로 있는 데다가 MC68302 자체에서 ISDN, Terminal Adaptor등의 일반적인 산업용 통신 규약을 지원하기 때문에 데이터 집중화 장치, 모뎀, 게이트 웨이, 등을 만드는 데에 유용하게 사용할 수 있다. 어쩌면 통신 전문 기업 Motorola의 CPU이니 당연한 건지도 모르겠다.

2. MC68302의 구조

이제 타깃 프로세서의 구조를 살펴보자. 그림 1에 프로세서의 전체 구성도가 그려져 있다.

이 구조를 자세히 살펴보면 크게 두 부분의 시스템 블록과 이 둘을 연결하고 있는 연결 고리를 볼 수 있을 것이다. 위쪽 파트는 거의 일반적인 형태의 프로세서의 내부 구조로 되어 있는데 CPU Core로 M68K를 사용하고 있다. 특이하게도 아래쪽 파트를 살펴보면 RISK 프로세서가 하나 더 보일 것이다.

즉, 이 칩은 2개의 CPU Core를 가지고 있는 셈이다. 아래쪽 통신 전용 컨트롤러인 RISC 프로세서는 통신 전용 블

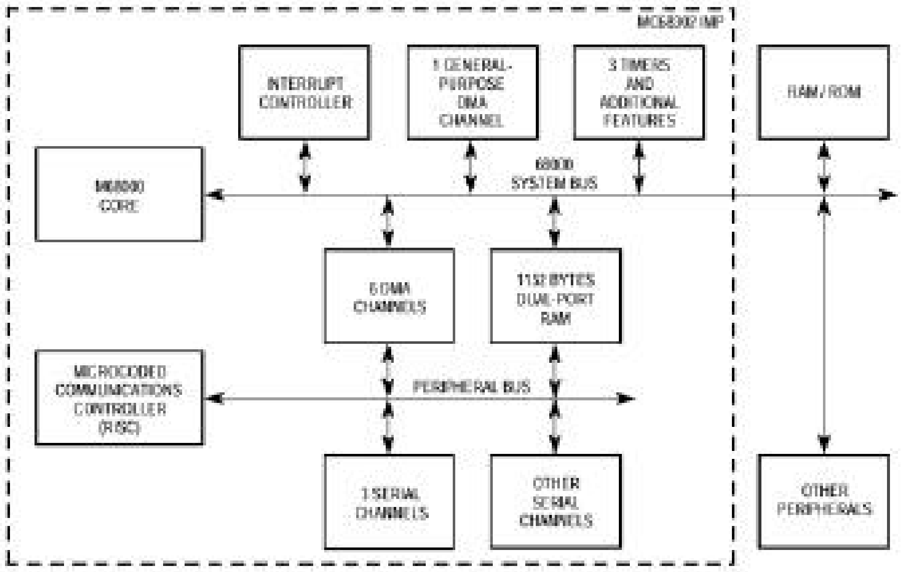


그림 1. MC68302 Block Diagram

록인 SMC, SCC(후에 설명)들의 통신 기능만을 담당하는 역할을 하고 이러한 독립적인 구조가 강력한 통신 기능을 제공하는 기반이 된다. 이러한 형태임을 염두해 두고 분석해 보도록 하자.

(1) M68000 core

MC68302의 Main CPU Core인 M68000 core는 8/16 Bit Bus System을 지원하며 user, supervisor programming mode가 있다.

User mode는 보통 응용 프로그램을 실행하는 mode이고, supervisor mode는 OS나 system programming 수행을 위해 user mode보다 몇 가지 특수한 명령어와 권한이 주어진 mode라고 생각하면 된다.

이 Core에는 일반적으로 사용되는 Resister (16, 32bit Address Register, 8, 16, 32bit Data Register), 32 bit Program Counter(PC), 8 bit Condition Code Register(CCR), User Stack Pointer(USP), Supervisor Stack Pointer(SSP), Status Register가 포함되어 있다.

이 중 User Stack Pointer와 PC, Condition Code Register는 user mode에서도 사용이 가능하나, Supervisor Stack Pointer와 Status Register는 supervisor mode에서만 사용하는 것이 가능하다.

이러한 레지스터 정보를 자세히 알아야 Context Switch 시에 필요한 것들을 이해할 수 있다.

Data type으로는 bit, digit (4bit), byte (8bit), word (16bit), long word (32bit) 의 다섯 가지를 기본으로 지원하며 address mode로는 Register Direct, Register Indirect, Absolute, Immediate, Program Count Relative, Implied 의 6가지를 사용할 수 있다.

또한 예외상황(인터럽트나 오류 등)이 발생했을 때 다음의 과정을 거쳐 이를 처리한다. 일단 임시로 SR을 복사한 후 SR을 예외처리 상황으로 설정한다.

그 후 Exception vector를 설정한 후 현재의 process context를 저장하고 최종적으로 새 context를 불러들여 수행하는 식으로 이루어진다. Exception vector는 예외처리를 수행하기 위한 루틴의 주소를 기술하고 있는 table이다.

한편 MC68302는 표준 M68000 core에는 없는 RMC signal을 제공하며, 이는 주로 bus lock으로 쓰인다. 또 M68000에는 있지만 MC68302에서는 사용하지 않는 signal이 두개가 있는데 이는 valid memory address (VMA)와 enable (E)이다.

MC68302의 모든 레지스터는 메모리 맵핑이 되어 있는데 Exception vector table, entry들이 $0부터 맵핑되며, 그 외에 dual-port system RAM과 parameter RAM, 내부 register의 4K의 block이 base+0$에 맵핑된다.

MC68302에는 주변장치로부터의 내부 레지스터로의 접근이 일어나는 것을 알려주는 IAC 핀이 있으며, 이벤트 발생 시 이들을 위한 Channel Status Register (CSR), Interrupt Pending Register (IPR), Interrupt Service Register (ISR), Timer Event Register (TER), Serial Communication Controller Event Register (SCCE) 등 몇 개의 특별 레지스터가 존재한다. 이를 통해 이벤트 발생 여부를 인식할 수 있게 된다.

(2) System Integration Block

SIB에는 그림1에서 볼 수 있듯이 interrupt controller, IDMA, timer, parallel I/O, clock generator등이 존재한다.

M68302에는 7개의 Direct Memory Access (DMA) channel이 존재하는데, 그 중 6개는 Serial DMA (SDMA)로써 세 개의 Serial Communications Controller(SCC)에 두개씩 존재하고 나머지 하나는 Independent DMA (IDMA)이다.

IDMA에는 6개의 레지스터가 있는데 이들은 모두 특정한 조작을 위해 지정되어 있으며, 각각을 설정하여 줌으로서 여러 상태를 만들 수 있다. 이를테면 채널 모드 레지스터 (CMR)의 RST Bit를 셋함으로서 소프트웨어 리셋을 실행시킬 수 있다.

MC68302의 인터럽트에는 7개의 우선순위 레벨이 존재한다. 이를 처리하기 위해, normal과 dedicated의 두 개의 mode가 존재하는데, normal mode에서는 6개의 user interrupt mode, dedicated mode에서는 3개의 user mode가 있으며 주변 장치 (외부 interrupt)는 level 4로 처리된다.

또 18개의 완전히 중첩될 수 있는 interrupt source가

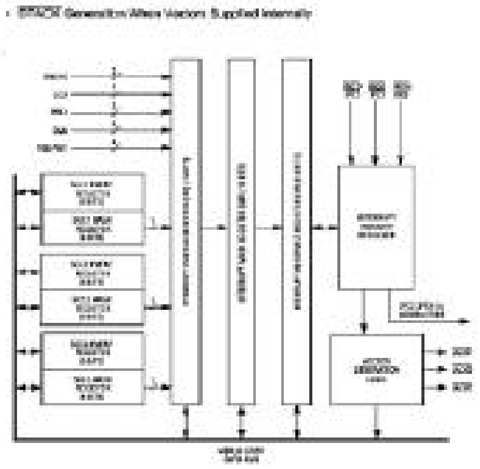


그림 2. Interrupt Signal handler

있고 각각의 source마다 유일한 vector number가 있다.

Interrupt controller는 interrupt event를 모아 interrupt pending register (IPR), interrupt mask register (IMR), interrupt in-service register (ISR)로 interrupt 요구를 검사한 후, 가장 높은 우선 순위의 것을 core에 건네준다.

실제 인터럽트가 발생하는 상황을 간단히 살펴보면 내부에서 하나 이상의 인터럽트가 발생하면, IPR에 그 인터럽트에 해당하는 bit가 셋된다. 그 후 IPR을 IMR과 비교하여, IMR에 셋이 되어있는 bit의 인터럽트는 빼고, 나머지만을 ISR에 전달해 주게 된다.

Interrupt priority resolver는 이렇게 ISR에서 전달받은 인터럽트와 현재 서비스중인 인터럽트의 우선순위를 비교하여 그 중 가장 우선순위가 높은 인터럽트 요구에 해당하는 벡터를 생성시켜 그것을 M68000 core에 전달한다.

외부 인터럽트는 mask할 수 없다는 점 외에는 내부 인터럽트와 같이 처리된다.

Core는 현재 instruction이 끝난 후에 그에 답하며 그 후 interrupt controller는 interrupt acknowledge cycle을 수행하며 Core에게 interrupt request를 알려준다. Core는 그 vector를 읽고 exception vector table에서 interrupt handler의 주소를 찾아 그 주소의 내용을

실행한다. 이 부분은 OS 포팅시에 인터럽트 처리부분에서 중요하게 쓰일 것이다.

MC68302가 마이크로컨트롤러의 역할을 수행하는데 필요한 A, B 두 개의 병렬 입출력 포트를 살펴보자. Port는 port A<B> control register (PA<B>CNT), port A <B> Data direction register (PA<B>DDR), port A <B> data register (PA<B>DAT)등을 통해 제어할 수 있다. Port A는 16 핀이며, port B는 12 핀이고 PB8 PB11은 일반 입출력 목적이 아닌 외부 인터럽트용으로 사용할 수도 있다.

1176Byte의 Dual-Port RAM중 576Byte는 시스템 RAM이고 나머지 576Byte는 parameter RAM이다. Parameter RAM은 SCC, SMC등의 채널을 위한 Buffer Descriptor를 포함하며 이 용도로 사용되지 않을 때는 일반 메모리로 사용된다.

각종 내부 주변 장치들을 위한 레지스터들은 internal register에 위치한다.

Timer에는 두 개의 일반적인 timer와 watchdog timer 한 개가 있다. Watchdog timer는 0부터 reference값까지 증가하여 reference값에 도달하면 WDOG신호를 발생시킨다.

Task가 실행되다가 어떤 일이 있어서 시스템이 제 시간에 프로그램을 실행하지 못 했을 경우나 프로그램에서 적절한 때에 watchdog timer의 값을 reset시켜주지 않았을 때 WDOG신호가 발생되며, 이는 오류로 간주되어 시스템의 오류처리 루틴을 실행시킨다. 또한 OS의 Clock Tick도 결국은 CPU의 timer에서 공급해 주어야 사용이 가능하다.

또 외부 인터페이스를 위해 외부 chip select signal이 있어 여러 개의 memory device를 한 개의 MC68302에 장착한 후 선택하여 사용할 수 있다.

이 외에 System control을 위해 system control register (SCR)이 있다. 이는 system status, control bit, bus arbiter control bit, hardware watchdog control bit, low-power control bit, freeze select bit로 구성되어 있다.

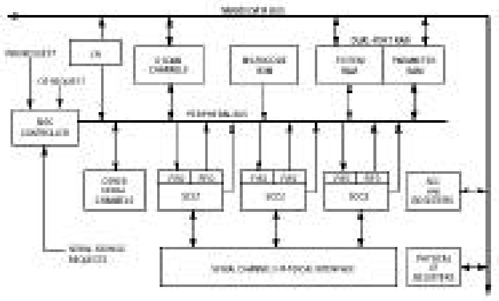


그림 3. CP Architecture

또 dynamic ram (DRAM) refresh control을 위한 신호들이 존재한다.

OS에서 필요로 하는 타깃 시스템 정보들은 여기까지이다. 그러나 통신 기능은 이 칩의 핵심적인 응용 기능이고, 비록 OS에서는 상관없겠지만 OS를 테스트 할 응용 프로그램을 만들 때에 거의 대상이 되므로 다음의 통신 파트에 대해서 간단히 알아보도록 하자.

(3) Communication Processor

CP에는 Main Controller (MC), 6개의 SDMA, Command Register (CR), Serial Channel, 3개의 Serial Communication Controller (SCC), Serial Communication Port (SCP), 2개의 Serial Management Controller (SMC) 가 포함되어 있다.

Main controller는 실제 프로그램을 수행하는 core와는 분리된, 통신만을 위한 프로세서이다. M68000 Core는 8 bit의 CR을 이용하여 CP에 명령을 내릴 수 있으며, SDMA channel은 각 SCC당 두 개씩 할당(전송, 수송)되어 외부 메모리와 연계할 경우 사용되며, RISC controller는 매 사이클마다 M68000 core와 충돌을 피하여 Dual-Port RAM을 접근한다.

3개의 SCC를 통한 physical interface로 MC68302에 가장 일반적으로 쓰이는 것은 Non-Multiplexed Serial Interface (NMSI)이다.

또한 Pulse Code Modulation Highway(PCM), Interchip Digital Link (IDL), General Circuit Interface (GCI)를 지원한다.

각 인터페이스는 테스트를 위해 echo(받은 신호를 다시 돌려주는 기능)와 loopback(전달된 신호를 다시 받게 하는 기능) mode를 지원한다.

SCC는 HDLC/SDLC, UART, BISYNC, DDCMP, V.110, 몇 개의 Transparent mode 등 다양한 통신 프로토콜을 지원한다. 이는 모두 IDL, GCI, PCM, NMSI의 네 가지 physical layer interface에 모두 구현될 수 있다.

SCP는 동기를 필요로 하는 채널을 지원하기 위해 사용되며 receive, transmit, clock의 세 선으로 구성되어 있다. SCP의 송수신부는 같은 clock에 의해 동작한다.

여기에서는 여러 MC68302가 지원하는 프로토콜 중 앞으로 OS 테스트에 사용할 UART에 대해서만 간단히 살펴보겠다. UART를 비동기 저속의 character 지향 통신 방식으로 사용할 것이다.

이 UART에는 여덟 개의 Control Character Comparison Register, 두 개의 Address Comparison Register, 네 개의 16 bit Error Counter가 있다. 또 7, 8 bit data, even/odd parity bit를 생성하고 검사할 수 있고 frame error, noise error, break, IDLE 신호를 검사할 수 있다는 특징이 있다.

UART의 통신 방식은 모두 잘 알고 있으리라고 보고 이의 구체적인 설명은 생략한다. 이제 타깃 프로세서에 대해 기본적인 설명은 마쳤다.

이 연재에서는 지면 상 OS 포팅에 관련된 부분만 설명할 수밖에 없다. 자세한 사항들은 메뉴얼을 참고하기 바란다.

3. 컴파일러

이제 남문인 컴파일러 및 어셈블러를 구하는 일이 남았다. 보통 컴파일러(앞으로는 컴파일러와 어셈블러를 통칭 컴파일러로 표기하겠다)는 특정 프로세서의 제품군을 대상으로 만들어진다.

Core는 같지만 내장된 주변 기능이 다른 제품군 내의 여러 프로세서를 위해 컴파일 알고리즘은 같게 한다. 단 개별적으로 특징적인 부분을 따로 Configuration 파일로 정의해 놓는다.

이번에 사용하는 MC68302의 경우도 M68K의 Core를 사용하므로 보통 68K용 컴파일러를 사용하고 확장 주변 장치에 대한 정보를 담고 있는 파일을 연결시켜 사용하면 된다.

(1) 컴파일러 구하기

컴파일러를 찾고 싶다면 칩을 만드는 회사의 홈페이지를 참조하면 직접 제공하는 경우도 있고 Third Party에 컴파일러 제작 회사명을 연결해 놓기도 한다.

보통의 컴파일러는 일정 비용을 지불해야 사용할 수 있는 것들이 많으며 Evaluation Version으로 일정 기간만 사용할 수 있거나 컴파일된 결과 파일의 크기를 제한하는 것의 형태로 공개하기도 한다.

한편 무료로 사용할 수 있는 것들 중에 대표적인 것으로 GCC를 꼽을 수 있는데 이를 Cross-Compile하여 사용하면 된다. 이번에는 Microtec Research, Inc.(Mentor Graphics)라는 회사의 제품인 MCC68K라는 비교적 Motorola Processor에 많이 사용되는 컴파일러를 사용하여 테스트 할 것이다.

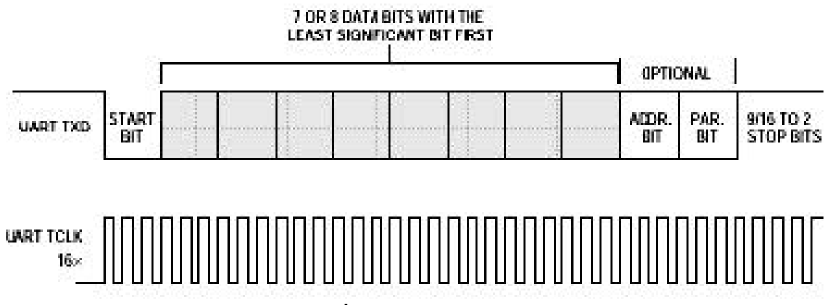


그림 4. UART Wave Form

(2) 컴파일러 설정

Microtac 68k 컴파일러 패키지에 포함된 자원들을 살펴보자. 이 패키지에 포함되는 것들 중 필수적으로 사용해야 하는 것만 살펴보면

· ASM68K.EXE - M68K용 어셈블리어를 사용할 수 있도록 하기 위한 어셈블러 프로그램

· MCC68K.EXE - C,C++파일을 컴파일하기 위한 컴파일러 프로그램

· LNK68K.EXE - 컴파일, 어셈블된 각 파일들을 묶기 위해 또 메모리에 위치시키기 위해 사용되는 프로그램

· LIB68K.EXE - 컴파일러에 필요한 각종 라이브러리를 실행하기 위해 사용되는 부가 프로그램

이 이외에 프로세서의 조정(CFE 68K), 최적화(CFE 68K) 등등 여러 가지 프로그램들을 사용할 수 있다.

이번에는 기본적인 것들만을 사용한 것이며 간략히 사용 방법을 기술해놓았다. 매뉴얼을 참고하여 자신의 시스템 상태에 가장 잘 맞도록 설정하는 것은 프로그래머의 몫이다.

```
--    사용법    --
① mcc68k는 후에 기술하겠다.
② asm68k [-I | -L][-b][-h][-V][-D sym[=val]]
[-I pathname][-oobj_file][-H sym_file][-f opt[,
opt]][-p processor][-Q opt]<src_file>
③ Ink68k [-V][-M][-m][-h][-r][-c 층_file][-C
Lnk_층][-u  name][-H  link_sym_file][-p
processor][-Q opt][input_file]...[>map_file]
```

현재 컴파일러를 사용하는 방법을 기술하면 다음과 같다. 어셈블러 파일 A.src와 C언어파일 B.c, C.c가 있다고 가정하자.

보통 어셈블러는 Hardware Dependent한 부분(주로 레지스터 설정 작업이나 빠른 명령 수행이 필요한 부분)을 위한 파일이고, C 파일은 일반적인 응용 프로그램인 경우가 많다.

어쨌든 이들을 위한 컴파일 방법은 다음과 같다.

1) A.src 파일은 프로세서의 모든 레지스터가 복잡한 내부 주소를 갖기 때문에 이를 대표적인 이름으로 쉽게 나타낸 파일을 첨부하게 된다.

이 파일을 어셈블하기 위해 ASM68K를 사용하여 A.src 파일을 A.obj 파일로 만든다.

```
----A.src----
include macros,inc
include MC68302.inc


....
END
```

2) xx.C 파일들은 MCC68k를 이용하여 xx.obj파일로 만든다.

```
----main.c----
#include" MC68302.h"
typedef unsignd short u_short;

void main()
{
......
 }
```

3) 이제 각 파일별로 만들어진 .obj 파일들은 Link and locate과정을 거치게 된다. 파일들간의 정보를 서로 연결하고 ROM 영역에 위치시키기 위한 Locate를 수행하는 것이다. 이 때 필요로 하는 파일이 있는데 기본적으로는 mc86302.cmd 파일이다. 이 파일에는 프로세서 정보, 이미지 파일의 위치 설정 등이 표기되어 있다.

링커는 이런 설정 정보와 필요한 .obj파일, .Lib파일들을 불러들여 작업한다.

```
----mc68302.cmd----
CHIP 68000
```

```
listmap PUBLICS
BASE $400
sec code=$800000
sec vars=$8500
format s
*order code, 0, 1, 2, literals, strings, const
ROM sections
*order vars, zerovars, heap
*RAM sections
load a.obj
load MAIN.obj
load xxx.obj
load d:\mcc68k\68k\mcc68kzb.lib
END
```

이를 수행하면 .hex 파일이 생성되어 실행이미지가 만들어진다. 이러한 컴파일 흐름을 가지고 OS를 컴파일 할 것이다.

시스템 구성

이제 Motorola MC68302를 중심으로 메모리, Serial 인터페이스를 구성하여 볼 것이다. 이것이 가장 기본적인 형태가 된다.

1.회로 설계

MC68302는 외부 Data 버스가 16비트로 구성되어 있으므로 외부 인터페이스를 ROM, RAM을 16Bit Data 버스용이나 8bit 용 2개를 사용하여 연결하면 가장 좋은 효율을 나타낼 수 있다. 그러나 초기 개발 시에 2개의 ROM을 사용하여 개발하려면 Emulator나 JTAG 장치를 이용하지 않는 이상 많은 수고가 따른다.

그래서 이번 디자인에서는 ROM Emulator를 1개만 사용하고 개발을 편리하게 하기 위해 외부 인터페이스를 8Bit로 설정한다. MC68302 버스 인터페이스는 8, 16 bit 용이 확연히 다르므로 주의해야 한다.

연결된 사항을 살펴보면 외부 입력 클록을 그대로 사용하고 각종 시스템 설정을 위한 Pull-up, Pull-down을 하였고, ROM, RAM의 메모리 인터페이스와 UART를 사용하기 위해 SMC에 TXD, RXD를 RS-232로 연결하였다.

하나의 프로세서와 프로그램 메모리, 메인 메모리만 연결하면 가장 기본적인 기능을 할 수 있으므로 이쯤에서 시스템 설계는 접도록 하겠다.

여기에 메모리를 확장하게나 추가적으로 각종 디바이스를 붙이는 작업은 개발자가 원하는 방식대로 만들면 된다. 여기서는 Serial Port와 병렬 포트를 사용하는 LED가 이런 추가적인 사항이 되겠다.

현재로써는 OS 포팅이 중점적인 사항이므로 가장 기본적인 애플리케이션만 구성한다.

2. 초기화 코드

Start-Up 코드, Boot 코드라고 불리는 프로세서 초기화 코드는 OS의 Hardware Dependent code와는 별개의 사항이다.

일단 시스템이 부팅되어야 OS가 움직이든 애플리케이션 프로그램이 실행되든 할 것이기 때문이다. 이 코드는 프로세서가 처음 동작을 시작하는 어드레스의 영역에 위치하여 수행되어야 한다. 주로 시스템이 구성된 상황에 맞춰 내부 특별 레지스터를 설정하는 역할을 한다. 이 부분의 코드는 ASM으로 작성되어 있으며, 컴파일러가 해석할 수 있는 구조로 짜여지므로 컴파일러에서 제공되는 매뉴얼 등을 참고하여 작성되어야 한다. 여기서는 Source Code의 MC68302.ASM이 이에 관계된 부분이다

이 부분에서 해야 할 일을 순차적으로 분석하면 다음과 같다.

(1) 스택 포인터(SP)의 메모리 상에서의 위치 설정 (.first:)
(2) 프로그램 카운터(PC)값의 초기화 설정
(3) 인터럽트 및 트랩의 벡터 어드레스에 관련 함수 설정
(4) 하드웨어 리셋 시의 루틴 설정(.hwl_warm_start:)
1) Base Address (BAR)
: Internal Memory Base 설정

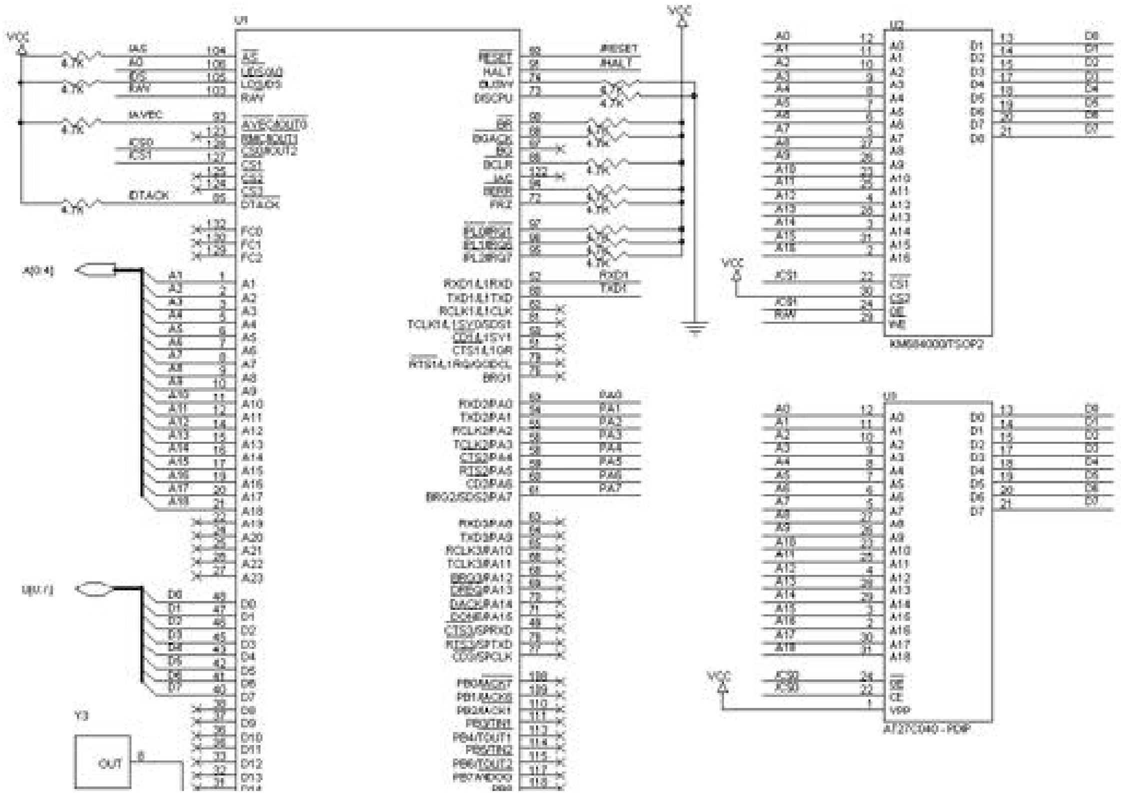


그림 5-1. System Scheametic(Main)

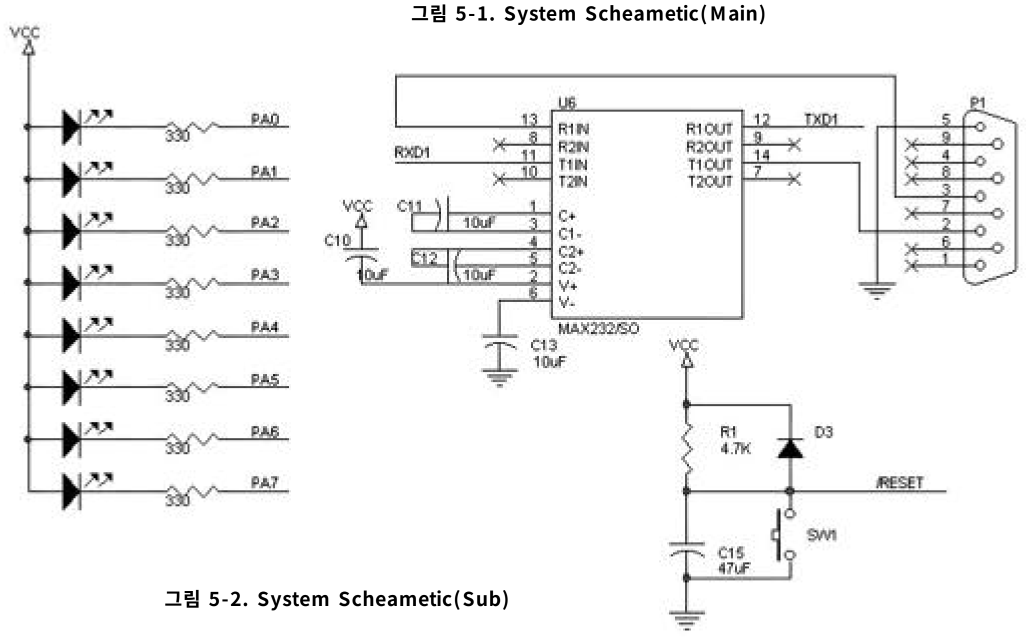


그림 5-2. System Scheametic(Sub)

2) System Control Register(SCR)

: 시스템 기본 설정

3) Watchdog Reference Register (WRR)

: Watchdog Disable

4) Global Interrupt Mode Register(GIMR), Interrupt Pending Register(IPR), Interrupt Mask Register(IMR) : IRQ Service 일단 정지

5) CS0-REGISTERS(BR0, OR0) 설정

: ROM을 위한 접근 속도 메모리 위치 등의 결정

6) Interrupt Disable로 설정

(5) 소프트 웨어 리셋 시의 루틴 설정(.hwl_cold_start:, .copy_code)

1) ROM에서 Internal DPRAM(Dual Port RAM)으로 설정 값 복사

(6) 실제적인 시스템 재 설정(go_parram :)

1) ROM을 위한 CS0-REGISTERS(BR0, OR0) 재 설정

2) RAM을 위한 CS0-REGISTERS(BR1, OR1) 재 설정

3) 기타 필요한 칩의 메모리 영역, 속도 등을 재설정

(7) C의 Main()루틴으로의 점프(jsr _main)

(8) 기타 추가적인 기능으로 인터럽트 가, 부를 위한 루틴(_enable_int:, _disable_int:,)

이 Startup 코드를 이용하여 부팅 후의 프로그램이 실제적으로 시스템에 접근할 수 있는 환경을 만든다. 또한 C언어로 작성된 Main함수로의 연결 고리를 만드는 역할이 가장 중요한 일이 되겠다.

시스템 부팅 시 처음에 수행되는 이 코드가 있으므로 해서 C언어 사용 등의 확장성과 이식성을 보장받을 수 있다고 생각해도 무방하다.

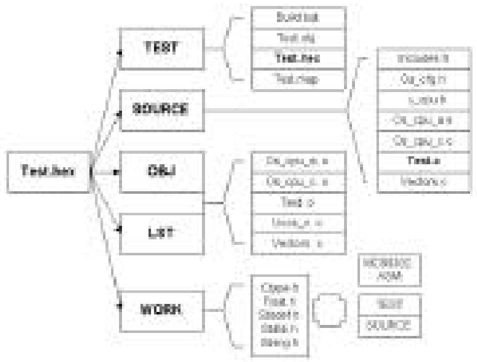


그림 6. 파일구성도

--정보--

· **Motorola Semiconductor homepage**

http://www.mot-sps.com

이곳에서 MC68302의 자료뿐만 아니라 각종 개발 시 필요한 도서, CD, DataSheet들을 제공하여 준다.

· **uC/OS-II homepage**

http://www.ucos-ii.com

이곳은 uC/OS가 M68K에 포팅된 자료를 공개한 곳이다. 이곳에서 포팅자료 일부를 수집할 수 있다.

· **Microtek Compiler**

http://www.mentor.com/embedded/compilers/index.html

컴파일러를 Evaluation Version으로 구해볼 수 있다. 사용자 등록 후 각종 컴파일러, 링커, 어셈블러들을 찾아본다.

OS 포팅

전 회의 OS 포팅에서와 같이 Hardware Dependent Code에 대해서만 변경을 하면 이 MC68302에도 uCOS가 포팅 될 수 있다.

1. Hardware Dependant Code

이제 몇 가지의 파일을 변경해 보자.

(1) OS_CPU.H

· 데이터 타입은 68K가 uC/OS의 모든 타입을 지원하므로 그대로 사용하여도 된다.

· 인터럽트 Disable, Enable을 위한 ASM코드를 68K에 맞게 수정한다.

```
#if        OS_CRITICAL_METHOD == 1
#define     OS_ENTER_CRITICAL()     asm("
ORI  #$0700,SR\n")#define
OS_EXIT_CRITICAL()          asm("      AND
#$0F800,SR\n")
#endif


 #          i              f
OS_CRITICAL_METHOD == 2
#define     OS_ENTER_CRITICAL()     asm("
MOVE    SR,-(A7)\n    ORI  #$0700,SR\n")
#define      OS_EXIT_CRITICAL()         asm("
MOVE    (A7)+,SR\n")
#endif
```

· Task 전환을 위해 OS_TASK_SW()함수를 호출하면 트랩이 발생하고 이 때 점프할 곳을 설정하는 데 이를 위해 Vector(Trap #15)에 등록한다. 실제로 이로 인해 불리게 되는 함수는 ASM코드 내의 OSCtxSw 함수이다.

```
#define     OS_TASK_SW()                  asm("
TRAP    #15\n")
```

· 스택의 자라는 방향을 거꾸로 자라게 한다.

```
#define   OS_STK_GROWTH           1
```

· 인터럽트 Enable, Disable을 위한 코드를 정의한다.

```
#define     CPU_INT_DIS()                 asm("
ORI      #$0700,SR\n")
#define    CPU_INT_EN()             asm("   AND
#$0F800,SR\n")
```

(2) OS_CPU_A.ASM

·_OSStartHighRdy: OSSrart()시 최상위 Task를 결정하는 루틴으로 그 Task의 Pointer를 새로 읽어들이고 Stack 포인터도 읽어드린다. CPU 레지스터 값을 저장하고 복귀한다.

```
JSR            _OSTaskSwHook
ADDQ.B        #1,_OSRunning
MOVE.L        (_OSTCBHighRdy),A1
MOVE.L            (A1),A7              MOVEM.L
(A7)+,A0-A6/D0-D7
RTE
```

·_OSCtxSw : Task 전환 시 관련 레지스터 등의 값을 저장하고 새로운 Task의 레지스터 정보를 읽어 설정하는 루틴이다. 레지스터 저장, TCB 전환, 새 Task의 레지스터 입수 등의 과정을 거친다. 이 Context Switch 루틴에서 이 CPU에 중요한 레지스터들이 저장되고 있는 것이 보일 것이다.

·_OSIntCtxSw : Context switching 중 인터럽트에 의해 현재의 수행 Task를 저장하고 처리하는 루틴으로 구성되어 있다.

```
MOVEM.L      A0-A6/D0-D7,-(A7
MOVE.L       (_OSTCBCur),
MOVE.L  A7,(A1)
JSR            __OSTaskSwHook
MOVE.L                      (_OSTCBHighRdy),A1

MOVE.L                               A1,(_OSTCBCur)
    MOVE.L       (A1),
MOVE.B       (_OSPrioHighRdy),(_OSPrioCur)
MOVEM.L                        (A7)+,A0-A6/D0-D7

RTE
```

·_OSTickISR : Time Tick을 위해 Time 인터럽트가 걸렸을 경우 인터럽트 벡터에 의해 이 함수를 호출하게 된다. 인터럽트이므로 Nesting 하고 상태를 저장하고 Time Tick 값을 증가 시킨 후 인터럽트를 처리하고 다시 수행할 Task의 값을 읽어 온다.

(3) OS_CPU_C.C

· void *OSTaskStkInit()만이 수행되는 중요한 함

```
ADDA        #18,A7
MOVE.L          (_OSTCBCur),A1    MOVE.L
A7,(A1)
JSR         _OSTaskSwHook
MOVE.L                    (_OSTCBHighRdy),A1

MOVE.L                          A1,(_OSTCBCur)

MOVE.B      (_OSPrioHighRdy),(_OSPrioCur)
MOVE.L      (A1), A7
MOVEM.L      (A7)+,A0-A6/D0-
RTE
```

수이다. 모두 CPU에 중요한 A0 A6, D0~D7, 스택, 복귀 주소 등의 설정에 관여한다.

```
ADDQ.B      #1,_OSIntNesting
MOVEM.L   A0-A6/D0-D7,-(A7)
JSR       _OSTimeTick
JSR       _OSIntExit
MOVEM.L      (A7)+,A0-A6/D0-D7
RTE
```

여기까지 Hardware Dependent 한 코드의 변경 사항을 살펴보았다. 이 각 파일들과 전 회에서 언급한 같은 파일들을 비교해보면 변경되어야 할 것들이 어떠한 것들인지 파악할 수 있을 것이다. 주로 CPU의 Architecture에 따라 특징적인 Interrupt, Timer, Register 들임을 직관할 수 있을 것이다.

이처럼 거의 모든 OS Porting이 이런 식이다.' 핵심적인 OS의 기능은 변경하지 않는다. 다만 CPU와의 연결 고리만을 변경한다 가 기본 방침인 것이다. 이제 주요 핵심부분을 수정하였으니 응용 프로그램을 위해 조정해야 할 파일들을 살펴보자.

2. Configuration Code

· OS_CGF.H : 응용 프로그램마다의 자원 이용 정도가 달라지므로 이를 위해 설정해주는 파일이다.

여기에서는 사용될 최대 개수의 Task수, 가장 높은, 낮은 우선 순위 레벨, IDLE, STAT Task의 설정, 각종 중요 함수의 사용 여부 결정 등을 표시한 파일이다.

· INCLUDE.H : 응용 프로그램에서 사용하는 OS 함수들이나 라이브러리 함수들을 연결 시키는 파일이다.

· MC68302.H : MC68302 레지스터, 각종 설정 값들을 정의하여 응용 프로그램에서 쉽게 사용할 수 있도록 한다.

· VECTOR.C : 외부 인터럽트 및 내부 인터럽트 등을 수행할 때 해당 인터럽트의 수행 코드가 어디에 있는지를 지시하는 역할을 할 코드이다.

모두 필요에 따라 결정하여 사용하면 되며 파일의 경로 등을 설정한다.

```
INT32U    *pstk32;
INT16U    *pstk16;
opt         =             opt;
pstk32         =  (INT32U  *)((INT32U)ptos  &
0x  F  F  7  F  F  F  F  C  L  )  ;
* - - p s t k 3 2 = ( I N T 3 2 U ) p d a t a ;
*--pstk32  =  (INT32U)task
pstk16       =  (INT16U  *)pstk32
*--pstk16    =    (INT16U)(0x0080    +    4    *
OS_TRAP_NBR);
pstk32       =  (INT32U  *)pstk16
* - - p s t k 3 2 = ( I N T 3 2 U ) t a s k ;
pstk16       =  (INT16U  *)pstk32
*--pstk16  =  (INT16U)OS_INITIAL_SR;
p s t k 3 2 = ( I N T 3 2 U * ) p s t k 1 6 ;
*--pstk32  =  (INT32U)0x00A600A6L;
*--pstk32  =  (INT32U)0x00A500A5L;
*--pstk32  =  (INT32U)0x00A400A4L;
*--pstk32  =  (INT32U)0x00A300A3L;
*--pstk32  =  (INT32U)0x00A200A2L;
*--pstk32  =  (INT32U)0x00A100A1L;
*--pstk32  =  (INT32U)0x00A000A0L;
*--pstk32  =  (INT32U)0x00D700D7L;
*--pstk32  =  (INT32U)0x00D600D6L;
*--pstk32  =  (INT32U)0x00D500D5L;
*--pstk32  =  (INT32U)0x00D400D4L;
*--pstk32  =  (INT32U)0x00D300D3L;
*--pstk32  =  (INT32U)0x00D200D2L;
*--pstk32  =  (INT32U)0x00D100D1L;
*--pstk32  =  (INT32U)0x00D000D0L;
return  ((void  *)pstk32);
```

```
void main(void)
[illegible]
      [illegible]
      OSInit();
      OSTaskCreate(StartTask,     (void     *)0,
(void *)&AppStartTaskStk[[illegible]], [illegible]);
      OSStart();
      while(1){
               for(i=0; i<[illegible]; i++) {}
              *[illegible] = [illegible];
             [illegible]
          for(i=0; i<[illegible]; i++) {}
           *[illegible] = [illegible];
   }
   }
   void [illegible](void) {
   [illegible]
   *[illegible] = [illegible];
   *[illegible] = [illegible];
   *[illegible] = [illegible];
   [illegible]
   [illegible]
   [illegible]
   }

static void StartTask(void *pdata) {
   int i;
   pdata = pdata;
   [illegible]

   OSStatInit();
   OSTaskCreate(AppTask1,     (void     *)0,
   (void *)&AppTask1Stk[[illegible]], [illegible]);
   OSTaskCreate(AppTask2,     (void     *)0,
   (void *)&AppTask2Stk[[illegible]], [illegible]);
   OSTaskCreate(AppTask3,     (void     *)0,
   (void *)&AppTask3Stk[[illegible]], [illegible]);
   [illegible]
```

```
   while (1) {
          OSTimeDly(1);
          [illegible]
   }
   }

   static void [illegible](void) {
   [illegible]
   OS_ENTER_CRITICAL();
   [illegible]
   [illegible]
   [illegible]
   [illegible]
   [illegible]
   [illegible]
   OS_EXIT_CRITICAL();
   }
   static void AppTask1(void *pdata) {
   [illegible]
   pdata = pdata;
   while (1) {
      for(i=0; i<[illegible]; i++) {
   }
      *[illegible] = [illegible];
     for(i=0; i<[illegible]; i++) {
      }
   *[illegible] = [illegible];
    OSTimeDly(1);
    [illegible]
   }
   }

   static void AppTask2(void *pdata);
}

   static void AppTask3(void *pdata);
}
```

실험하기

이제 실제 프로그램 하나를 이용하여 실험을 해보자. 실험을 위해 예제 코드를 하나 만들었다.

전 회의 실험과 비슷하다. uCOS는 OS와 Task가 함께 컴파일 되는 형태이다.

그러므로 Main() 함수의 호출이 있는 곳이 전체 프로그램의 시작 부분이 된다.

프로그램을 살펴보면 Main()함수 내에서 Init()함수를 호출하여 OS 시작 전에 설정해야 할 항목들을 설정하였다. 이 실험에서는 LED를 켜기 위한 병렬 포트와 Serial 통신을 위한 UART를 사용할 것이므로 이에 대한 설정을 한 것이다.

여기서 UART 설정을 하기 위해 serial_init()함수를 호출하는데 이를 위해 serial.c라는 파일을 만들었다. 이는 일종의 라이브러리로써 68302에서 SCC를 통하여 RS-232시리얼 통신을 수행할 수 있는 함수들이 구성되어 있다.

전의 PC에서는 PC.H와 같은 역할을 하는 파일이다. 이 파일의 함수인 시리얼 초기화, 전송, 수신들은 응용 프로그램에서 사용되므로 직접 작성해야 될 듯하다.

이처럼 Low Level 함수는 애플리케이션 프로그램처럼 작성되어야 한다. 이후 OSInit() 함수를 호출하여 OS를 준비하고 OSTaskCreate()함수를 사용하여 첫 번째 Task를 등록시킨다.

```
--- make.bat ---

asm68k.exe -l >startup.lst startup.src
asm68k -l >oscpua.lst oscpua.srcmcc68k.exe
-c oscpuc.c >oscpuc.lst
mcc68k.exe -c vectors.c >vectors.lst
mcc68k.exe -c test.c >test.lst
mcc68k.exe -c ucos_ii.c >ucos_ii.lst
lnk68k.exe -c test.cmd -o test.hex -m -f S
>test.map
```

이 첫 번째 Task에서는 앞으로 StartTask() 함수를 호출하게 되는데 이 함수에서는 여기서 3개의 Task를 새로 등록하고 OS Tick을 위한 Timer를 기동시킨다. 또 무한 루프를 형성해서 Serial Port를 통해서 특정 문자 메시지를 주기적으로 보내는 작업을 할 것이다.

Timer 기동에 관련된 함수는 Apptickinit()이고 이 함수에서는 Critical Section에서 Interrupt를 disable한 상태에서 Timer 주기 값들을 설정하고 실행시켰다.

앞으로 새로 등록된 3개의 Task에서는 각각 Task1, Task2, Task3 함수(AppTask1, AppTask2, AppTask3)를 수행 시간이 되면 호출하게 될 것이다. Task1에서는 병렬 포트를 통해서 데이터를 바꿔 뿌리는

```
-- Test.cmd --

CHIP 68000
listmap PUBLICS
BASE     $400
sec    code=$800000
sec    vars=$7500
format s
*order code,literals,strings,const
* ROM sections
*order vars,zerovars,heap
* RAM sections
load startup.obj
load oscpua.obj
load vectors.obj
load oscpuc.objload test.obj
load ucos_ii.obj
*load cMAIN.obj
*load port.obj
load
mcc68kab.lib


END
```

작업이 수행되고 있음을 보여주고 있다. 지금 빈 공간으로 남겨둔 다른 Task 함수에도 원하는 기능의 코드를 넣으면 실시간 작업을 수행하게 될 것이다.

이렇게 간단히 Multitasking이 구현되었다. 이렇듯 운영체제를 이용하여 설계된 시스템에서는 개발자에게 부담을 많이 줄여주게 된다.

이제 실제 시스템에 이 완성된 프로그램을 컴파일 하여 넣고 실행시키는 과정을 알아보자.

이번에는 bat 파일로 어셈블, 컴파일, 링크를 엮어 실행하겠다. 자세한 각종 컴파일 option은 해당 Site를 통해서도 알 수 있다.

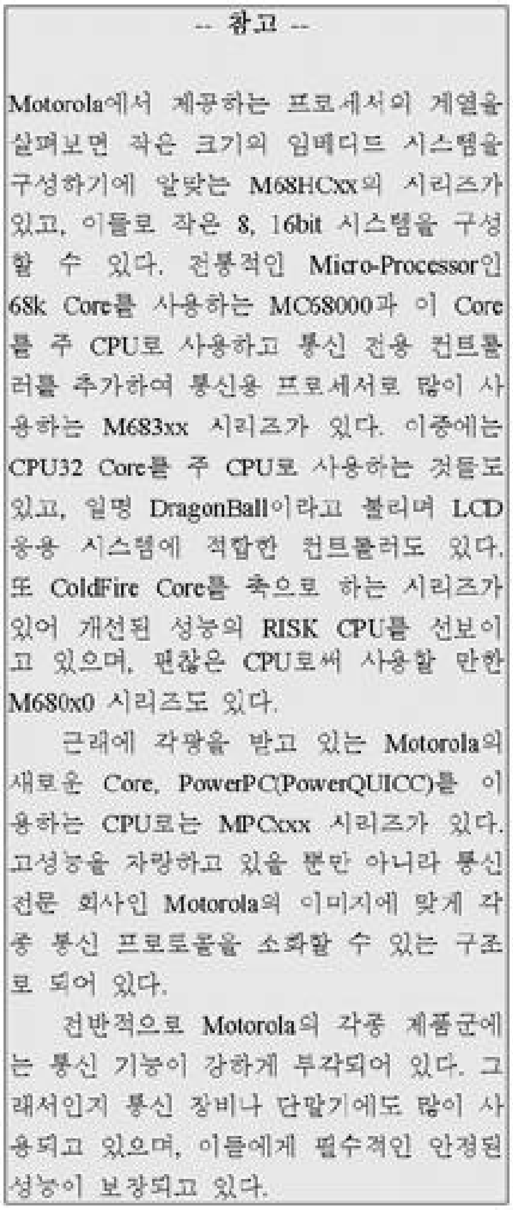
Bat파일을 살펴보면 각각의 ASM 파일과 C파일을 각기 컴파일 하여 링크시키고 있다.

이는 앞의 컴파일 예와 동일하다. 또 링크에서 test.cmd파일의 정보를 이용하여 링크를 하게 되므로 이에 대한 자세한 정보를 아래 기술하였다. 이 Command파일을 만드는 것도 컴파일러 Manual에 자세히 수록되어 있다. 주로 컴파일 할 프로세서의 Core, 탑재될 메모리 위치, 관련 Object 파일명, Library 파일명들이다.

이렇게 해서 생성된 TEST.HEX 파일은 Serial Port를 통해서 ROM Emulator로 전송되며, 이제 이 ROM 역할을 하는 장치로부터 CPU가 명령어를 받아들여 기능을 수행할 것이다.

마치며

이번 호에서는 MC68302가 탑재된 실제 임베디드 시스템을 제작하여 RTOS 포팅을 시도하였다. 전 회의 PC에서의 포팅과 비교하면 차이점을 찾을 수 있을 것이다. 그 차이점이 보통 포팅에서 필요로 하는 것들이다. 이들을 알기 위해서는 많은 시행착오를 겪어야 할지 모르지만 수행해 가는 방향을 잘 잡고, 결과를 예상하여 실제와 틀린 점을 찾아가는 것이 가장 좋은 방법이 될 것이다. 다음 회에서는 요즘 임베디드 시스템에 많이 적용되고 있는 ARM Core를 사용한 Intel의 StrongARM 보드를 이용하여 포팅을 해 볼 것이다. EL



-- 참고 --

Motorola에서 제공하는 프로세서의 계열을 살펴보면 작은 크기의 임베디드 시스템을 구성하기에 알맞는 M68HCxx의 시리즈가 있고, 이들로 작은 8, 16bit 시스템을 구성할 수 있다. 전통적인 Micro-Processor인 68k Core를 사용하는 MC68000과 이 Core를 주 CPU로 사용하고 통신 전용 컨트롤러를 추가하여 통신용 프로세서로 많이 사용하는 M683xx 시리즈가 있다. 이중에는 CPU32 Core를 주 CPU로 사용하는 것들도 있고, 일명 DragonBall이라고 불리며 LCD 응용 시스템에 적합한 컨트롤러도 있다. 또 ColdFire Core를 축으로 하는 시리즈가 있어 개선된 성능의 RISK CPU를 선보이고 있으며, 괜찮은 CPU로써 사용할 만한 M680x0 시리즈도 있다.

근래에 각광을 받고 있는 Motorola의 새로운 Core, PowerPC(PowerQUICC)를 이용하는 CPU로는 MPCxxx 시리즈가 있다. 고성능을 자랑하고 있을 뿐만 아니라 통신 전문 회사인 Motorola의 이미지에 맞게 각종 통신 프로토콜을 소화할 수 있는 구조로 되어 있다.

전반적으로 Motorola의 각종 제품군에는 통신 기능이 강하게 부각되어 있다. 그래서인지 통신 장비나 단말기에도 많이 사용되고 있으며, 이들에게 필수적인 안정된 성능이 보장되고 있다.