임베디드 시스템과 Real-Time OS

진태석
동서대학교 메카트로닉스공학과 교수
jints@dongseo.ac.kr

1. 기술 개요

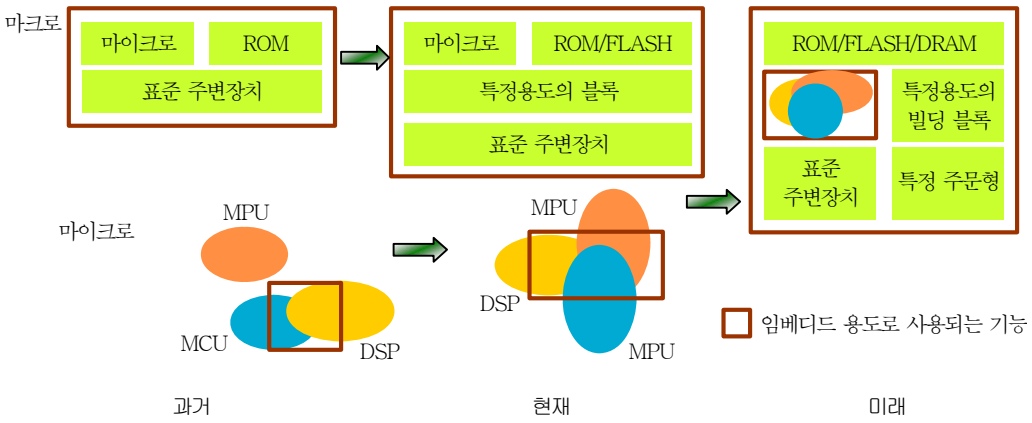
가까이 있지만 멀게만 느껴지는 분야가 바로 임베디드 시스템(Embedded System)이라 할 수 있을 것이다. 왜냐하면 우리 생활과 가장 밀접한 관계에서 이용되는 여러 제품들이 바로 임베디드 시스템으로 구현되어 실제로 활용이 되고 있으나 기술적인 구현이나 개발에 있어서는 보다 전문적이고 어려운 분야로 여겨지기 때문이다. 이러한 임베디드 시스템에 대한 전반적인 내용과 실시간 운영체계(RTOS)와 임베디드 시스템이 어떠한 관계를 가지고 구현되는지를 알아보도록 한다.

가. Embedded system 의 의미

전기, 전자, 컴퓨터 기술들이 발달하면서 이러한 기술들을 이용한 다양한 기기들이 우리의 생활 주변에 들어오게 되었다. PC 를 제외하더라도 일상생활에서 사용되고 있는 TV, 냉장고, 세탁기, 전자레인지 같은 전자 가전제품뿐 아니라 우리가 가지고 다니는 핸드폰, PDA 그리고 요즘 주택가에서 불고 있는 사이버 아파트에서 보면 홈 관리 시스템, 홈네트워크 게이트웨이 장치, 밖으로 살펴보면 교통관리 시스템, 주차 관리 시스템, 홈 관리 시스템, 엘리베이터 시스템, 현금 지급기(ATM), 항공 관제 시스템, 우주선 제어장치, 군사용 제어 장치 등 셀수도 없이 많은 기술, 제품들이 우리 생활과 밀접하게 붙어서 도움을 주고 있다.

위에서 열거한 것들의 개발 환경을 생각해 본다면 우리는 임베디드 시스템에 대한 개념을 잡는데 상당히 도움이 될 것이다. 즉 임베디드 시스템(Embedded System)이란 미리 정해진 특정 기능을 수행하기 위해서 컴퓨터 하드웨어와 소프트웨어가 조합된 전자 제어 시스템을 말하며,

* 본 컬럼은 동서대학교 메카트로닉스공학과에서 작성한 내용입니다. 본 내용과 관련된 사항은 동서대학교 메카트로닉스공학과 진태석 교수(☎ 062-230-7107)에게 문의하시기 바랍니다.
**본 내용은 필자의 주관적인 의견이며 IITA 의 공식적인 입장이 아님을 밝힙니다.



(그림 1) 임베디드 아키텍처의 변천

필요에 따라서는 일부 기계(mechanical parts)가 포함될 수 있다. 좀더 자세히 이야기를 하면 우리 생활에서 쓰이는 각종 전자기기, 가전제품, 제어장치 중에서 단순히 회로로만 구성된 것이 아니라 마이크로 프로세서가 내장되어 있고, 그 마이크로 프로세서를 구동하여 특정한 기능을 수행하도록 프로그램이 내장되어 있는 시스템을 가리키는 것이다. 세탁기를 예로 들었을 때 예전의 것은 세탁과 탈수의 기능만을 가지는 단순한 기기였는데 요즘 나오는 세탁기는 옷감 종류부터 시작해서 세탁할 옷의 양, 물의 선택 등을 고려해서 세탁을 할 수 있도록 되어 있다. 이전의 시스템으로는 하기 힘든 것을 마이크로 프로세서와 그에 따른 제어 프로그램이 내장된 임베디드 시스템이 하는 것이다.

나. Embedded System 의 역사와 의미

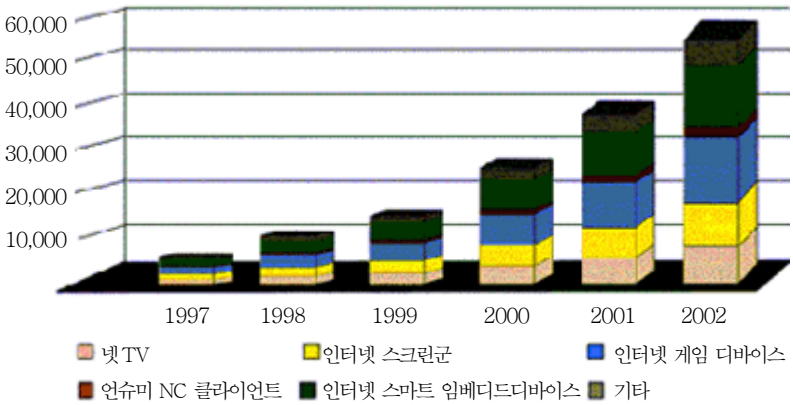
제어 장비 또는 임베디드 시스템으로 쓰이는 컴퓨터는 컴퓨터 자체만큼이나 오래되었으며, 통신 분야에서는 1960 년대 후반 전기-기계식 전화 교환기와 내장 프로그램 제어 시스템을 제어하는데 이러한 시스템이 쓰였다. 그때에는 컴퓨터라는 단어가 보편화되지 않아서 Stored Program 이라는 의미는 프로그램이 사용하고 있는 메모리와 경로 정보를 칭하였고, 이러한 컴퓨터는 각 응용 프로그램에 맞게 설계되었다. 현재에는 PC 와 같은 현재의 표준 때문에 특정 목적의 명령, 메인 컴퓨팅 엔진에 통합된 I/O 장치들은 어색하게 보일 수도 있다.

소프트웨어도 하드웨어에 따라 발전하였고, 처음에는 소프트웨어를 만들고 테스트하는데 단순한 프로그램 개발 도구만을 사용할 수 있었다. 각 프로젝트의 런타임 소프트웨어는 보통 밑바닥부터 새로 제작되었다. 이러한 소프트웨어는 항상 어셈블리 언어 또는 매크로 언어로 작성되었는데, 왜냐하면 컴파일러에 버그가 많았고 쓸만한 디버거가 없었기 때문이었다.

임베디드 시스템의 운영체계에서 표준화된 대량생산이 등장한 것은 70 년대 후반이며, 이들 대다수는 어셈블리 언어로 제작되었고, 개발 대상으로 하는 마이크로프로세서에서만 사용할 수 있었다. 그러므로 마이크로프로세서가 구식이 되면, 그 운영체계도 구식이 되었다.

C 언어가 나타난 그때부터 운영체계를 효율적, 안정적이고 포터블(Portable)한 방법으로 작성할 수 있었다. 이는 현재의 마이크로프로세서가 구식이 되었을 때, 그때까지 들인 소프트웨어에 대한 투자를 보호할 수 있었기 때문이다. 이는 마케팅 관점에서 무척 반가운 이야기로 들렸으며, 결국 C 언어로 작성된 운영체계는 표준이 되었고 오늘날까지도 남아 있다. 다시 말해서 소프트웨어의 재사용이 이루어지게 되었고 오늘날까지 이어지고 있는 것이다.

그럼 이러한 역사를 토대로 임베디드 시스템이 무엇인가를 알아보도록 한다. 데이터베이스나 응용 프로그램의 경우는 특별히 '내장'되었다는 표현을 쓰지 않지만, 엘리베이터나 텔레비전 등의 경우에는 내장된 시스템이라는 용어를 쓰고 있는데, 이런 분야에서 임베디드 시스템(Embedded System)이라는 말이 거론될 수 있다.

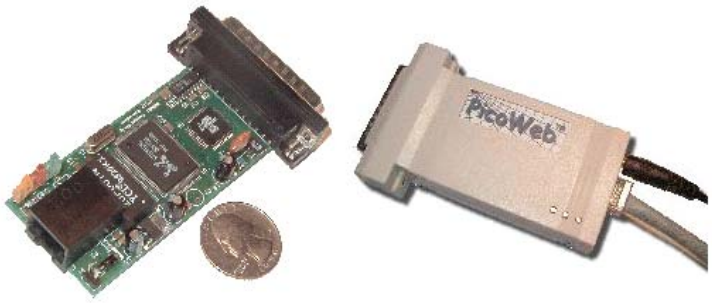


<자료>: IDC

(그림 2) 임베디드 시스템의 인터넷 응용분야 전망

우리 주변에는 인공지능 혹은 퍼지라는 기능을 채택한 전자제품이 많이 나오고 있다. 퍼지 세탁기, 인공지능 전기밥솥 등의 대표적인 제품이라고 할 수 있는데, 단지 세탁을 하고 밥을 하거나 얼음을 얼리는 등의 기능이라면 특별히 인공지능이란 표현을 쓰지 않아도 될 것이다. 여기에는 세탁물의 종류에 따라, 혹은 밥을 지을 때 들어가는 재료에 따라, 냉장실 또는 냉동실에 들어가는 재료에 따라 기기가 자동/수동으로 반응하기 때문에 인공지능이란 표현을 쓴 것이다.

그런데 여기서 중요한 것은 이런 모든 기능을 회로만으로 구성해서 구현한다는 것은 사실상 불가능하다는 점이다. 적당한 제어용 CPU 가 있고, 또 그 기기의 기능에 맞는 프로그램이 탑재



(그림 3) Atmel's AT90S8515 마이로컨트롤러를 사용한 PicoWeb Server

되어 있어 그 프로그램을 통해야만 기능을 구현할 수 있는 것이다. 바로 이런 것이 임베디드 운영체계라 할 수 있다.

임베디드 시스템에 대한 예를 일상에서만 찾아보았지만, 실제로 임베디드 시스템이 요구되는 곳은 무궁무진하다. 특히 공장 자동화나 가정 자동화와 같이 자동화 분야에서는 필수적인 요소로 부각되고 있다. 즉 임베디드 시스템이란 기계 또는 전자 장치의 두뇌 역할을 하는 마이크로프로세서를 장착해 설계함으로써 효과적인 제어를 할 수 있도록 하는 시스템을 의미한다.

'임베디드 시스템이 바로 이런 것이다'라는 것을 보여줄 만한 예로서 바로 작은 웹서버를 만들수 있다는 가능성을 보여준 사이트 http://www.picoweb.net 이다.

2. Real-Time OS 의 기본 개념

가. task 와 multitasking

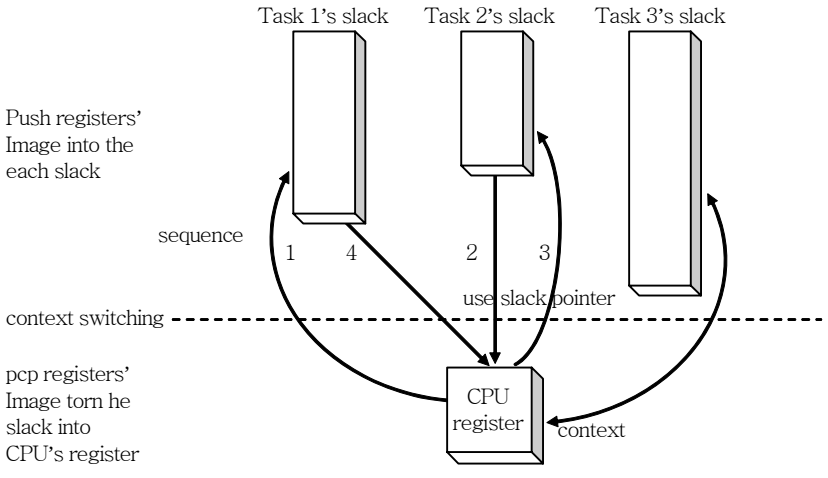
UNIX 를 써본 적이 있는 사람은 여러 사람이 한꺼번에 login 을 해서 동시에 자신의 프로그램들을 수행할 수 있음을 알 것이다. 굳이 UNIX 를 예로 들지 않더라도 Windows 95 시스템인 경우만 해도 Sound card 를 통해서 음악을 들으며 C compiler 는 compile 을 수행하면서 프린터로 문서를 출력할 수 있음을 알고 있다. 이때 음악 연주 프로그램, compiler program, printer spooler 를 각각의 태스크라고 한다. 그리고 이처럼 여러 개의 태스크를 동시에(여기에서 말하는 의미는 어떤 기간을 말하는 것으로 어느 한 시점으로 보면 아님, multiprocessor system 과 비교!) 실행할 수 있는 것을 멀티태스킹이라 한다.

Embedded system 에서도 처리할 작업이 많아지면서 멀티태스킹을 적용하게 되었으나 앞에서 말한 것과는 차이가 있다. 일반적인 컴퓨터에서는 각 태스크들이 대부분 무관한 프로그램들이지만 embedded system 에서의 태스크들은 하나의 큰 응용 프로그램을 논리적으로 나눈 것

이다. 즉, 어떤 응용 프로그램이 수행되기 위해서는 여러 기능들이 동시에(물리적으로는 아님) 수행되어야 할 필요가 있고 이를 순차적으로 프로그램을 하기 어렵기 때문에 사용자가 프로그램을 쉽게 하기 위해서 도입한 개념이 바로 멀티태스킹이다. 하나의 응용 프로그램을 논리적으로 나누었기 때문에 기능상 매우 밀접한 관계가 있기 때문에 태스크 사이에 이루어지는 작업들이 매우 많다. 이처럼 일반 시스템과 embedded system 에서의 멀티태스킹이 다른 의미를 갖지만 아래에서 설명하는 것은 OS 에서 논의되는 일반적인 개념으로 어떤 시스템에도 적용할 수 있는 것이다.

나. context switching

우선, task1 에서 하던 작업이 중단되고 task2 가 수행되려 한다고 가정하자(이렇게 되는 경우는 매우 많고 3 장의 task transition 에서 더 설명될 것이다.). task1 이 완전히 종료가 된 상태라면 별 문제 없겠지만 중단된 것이기에 task2 가 수행을 마치거나 중단되는 경우에 다시 task1 을 수행하게 된다면 이전에 수행되던 상태를 알아야 할 것이다(이는 함수에서 함수를 다시 호출하는 경우와 비슷하다.). 따라서 task2 가 수행되기 이전에 task1 의 상태를 저장해 둘 필요가 있다. Computer system 에서 상태라는 것은 레지스터, 메모리, 하드 디스크 등에 저장된다. 이때 메모리 같은 자원은 태스크마다 나누어 쓰면 별 문제가 없을 수 있지만 CPU 의 레지스터는 공유하는 자원이므로 구역을 나누어 쓸 수 없다. 따라서 상태를 저장해야 하는 가장 중요한 자원은 레지스터이다.



(그림 4) 멀티테스킹과 Context switching

그래서 task1 이 중단되기 전에 레지스터 값들을 task1 의 스택에 보관을 하고 task2 가 수행되고 다시 task1 이 수행될 시점이 오면 스택에 저장되었던 값들을 레지스터로 옮겨와서 task1 을 다시 수행하게 된다. 이를 context switching 이라 한다.

Context switching 이 이루어지는 동안은 실제 응용 프로그램에서 원하는 작업은 이루어지지 않기 때문에 오버헤드라고 볼 수 있다. 이것은 저장할 레지스터의 개수에 따라, 쓰는 프로세서 등에 따라 달라진다. Context switching 에 걸리는 시간이 짧으면 짧을수록 좀 더 효율적인 OS 라고 볼 수 있기 때문에 예전에 real-time OS 를 비교할 때는(현재도 그러하지만) 이 시간이 얼마나 짧은가가 매우 중요한 척도였지만 프로세서의 성능이 점점 좋아지면서 OS 간에 차이가 줄어들고 있다.

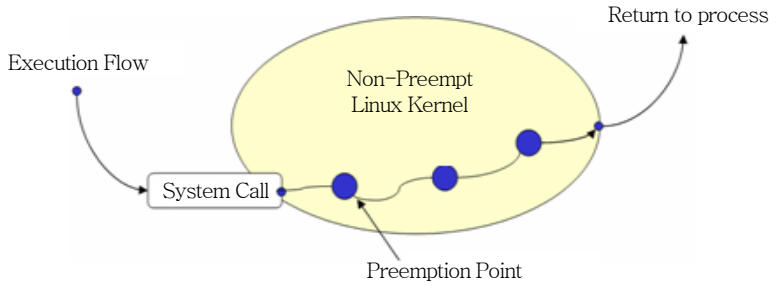
다. Kernel

OS 란 앞에서도 언급했지만 컴퓨터 자원을 효율적이고 쉽게 쓰도록 하기 위한 소프트웨어이다. OS 의 기능 중에서도 핵심이 되는 부분을 커널이라고 하고 여기에서는 앞에서 설명한 context switching, task scheduling, memory management 등의 작업을 한다.

(1) Scheduler

멀티태스킹 환경에서 다음에 어떤 태스크가 수행될 것인지 결정하는 책임을 맡는다. 커널이라고 부르는 부분에서도 핵심적인 부분이라고 할 수 있는 것이 바로 스케줄러로서 디스패처라고도 한다.

일반 멀티태스킹 OS 에는 여러 가지 스케줄링 기법에 있으며 real-time OS 에도 이에 맡는 여러 가지 스케줄링 기법들이 제안되었다. 그러나 구현상의 문제 등으로 실제로 대부분의 real-time OS 에서 쓰는 스케줄링 기법은 priority-based scheduling 이다. 또한 같은 프라이어러티에 대해서는 FIFO 나 round-robin 방법이 적용된다.



(그림 5) Preemptive Kernel Approach

(2) Non-Preemptive Kernel

우리말로 비-선점형 커넬이라고 한다. 쉽게 말해서 커넬의 능력이 크지 못하다고 할 수 있다. 즉, 어떤 태스크가 수행되고 있을 때 커넬이 중간에서 그 태스크의 수행을 중지시키고 다른 태스크를 수행시킬 수 있는 능력이 없다. 다른 말로 cooperative multitasking 이라고도 한다. 그래서 preemptive kernel 보다 design 하기 쉽다.

Preemptive 되는 경우는 오직 ISR(Interrupt Service Routine)에 의해서이다. 이때도 역시 ISR 에서 interrupted task 로 제어권을 수동으로 넘겨줘야 한다.

장점으로는 interrupt latency 가 비교적 짧다는 것이고 또 하나 reentrant function 을 쓸 수 있다는 것이다. 그러나 단점은 태스크가 제어권을 넘기지 않으면 커넬이 아무 지시도 내릴 수 없기 때문에 태스크들이 매우 잘 계획이 되어야 한다는 것이다. 따라서 프라이어러티가 높은 태스크도 무한정 기다릴 수 있기 때문에 real-time system 에서는 일반적으로 잘 쓰이지 않는다.

우리가 알고 있는 OS 중에서는 Windows 3.1 이 non-preemptive 에 속한다(Windows 3.1 은 OS 라고 말할 수 없는 성격이긴 하지만).

(3) Preemptive Kernel

우리말로 선점형 커넬이라고 한다. 이 커넬은 어떤 태스크를 중단시키고 다른 태스크를 수행시킬 능력이 있는 것이다. 따라서 앞의 non-preemptive kernel 보다는 구현하기 복잡하다. 또한 interrupt latency 가 조금 길어진다는 단점이 있다.

그러나 이런 단점에도 불구하고 대부분의 real-time OS 에서 채택하는 이유는 프라이어러티가 높은 태스크가 먼저 수행될 수 있다는 점 때문이다. 즉, 가장 프라이어러티가 높은 태스크는 가장 먼저 수행될 수 있다는 deterministic 한 성격을 지닌다.

앞의 non-preemptive kernel 에서는 ISR 에서 interrupted task 로 제어권을 넘겨줘야 했지만 이 경우에 ISR 이 interrupted task 보다 높은 priority task 를 호출한다면 ISR 이 끝나고 interrupted task 로 진행되는 것이 아니고 priority task 로 진행이 되고 interrupted task 는 suspend 상태가 되는 것이다. 앞으로 설명할 OS 는 이 preemptive kernel 이다. 우리가 알고 있는 대부분의 OS(DOS, Windows 3.1)을 제외하고 모두 이 preemptive kernel 에 해당한다.

라. critical section

다른 태스크에 의해서 중단되어서는 안되는 일련의 명령 혹은 코드의 블럭을 말한다. 예를 들어 shared memory 를 접근하는 부분에서는 중단이 되는 경우에는 그 결과를 보장할 수 없다. 이를 해결하기 위해서는 아래에서 설명하는 mutual exclusion 이 보장되어야 한다.

마. mutual exclusion

예를 들어서 두개의 태스크가 동시에 프린터를 쓰려고 한다고 가정하자. 만약 task1 이 프린트 하는 도중에 task2 가 실행되고 task2 가 실행되는 도중에 다시 task1 이 실행된다면 나온 결과는 task1 이나 task2 모두 원하지 않는 결과일 것이다. 이때 mutual exclusion 이 되어야 할 필요가 있다. 또한 위에서 언급한 shared memory 도 마찬가지이다. 즉 shared resource 를 쓰는 경우에는 mutual exclusion 이 보장되어야 하는 것이다.

3. Real-time System Design

가. 어떤 Real-time OS 를 쓸 것인가?

Real-time OS 를 쓰기로 결정했다면 과연 어떤 real-time OS 를 쓸 것인지 상당히 고민이 될 것이다. 현재 나와있는 real-time OS 제품들이 그 수가 수십 개에 달하기 때문이다.

초기에는 real-time OS 를 선정할 때에는 latency 를 얼마나 최소화할 수 있는가 external interrupt, kernel service, task switching 과 관련된 latency 를 최소화였으나 오늘날에 micro-processor 가 발전하면서 latency 최소화의 중요성이 점차 약해지고 있다.

이제는 이런 것보다 더 다양한 면에서 여러 real-time OS 를 비교할 수 있는 기준이 필요하다. 그렇다면 우선 real-time OS 를 쓰는 이유는 생각해 봐야 할 것이다.

이상적으로 말한다면 성능을 저하시키지 않고 메모리를 많이 요구하지 않으면서 코드를 복잡하게 하지 않으면서 적절한 system service call 을 제공하고 task scheduling 에서도 유연성을 줄 수 있어야 한다. 즉, 간단히 말해서 효율적이면서 쉽게 개발할 수 있도록 해야 한다는 것이다. 우선 이런 점을 만족시켜야 할 것이지만 실제로 선정을 할 때에는 이런 기능적인 측면 외에도 다른 면들도 고려해야 한다. 이제부터 한 가지씩 살펴보기로 하자.

(1) 원하는 프로세서를 지원하는가? 또 새로운 프로세서에 대한 portability 는?

우선 무엇보다도 시스템에서 쓰고자 하는 프로세서를 지원해야 할 것이다. 이미 나온 지가 오래된 프로세서에 대해서는 지원하는 RTOS 들이 꽤 있을 것이지만 최근에 나온 프로세서에 대해서는 지원하는 OS 들이 많지 않을 수도 있다. 또한 개발한 시스템을 업그레이드한다고 가정해 보자. 이때 새로운 프로세서를 쓰게 될 수도 있다. 이전의 시스템에서 쓰던 RTOS 가 새로운 프로세서를 지원하지 못한다면 다른 RTOS 를 쓸 수밖에 없다. 이를 해결하기 위해서는 2 가지 방법이 있을 수 있는데 하나는 처음부터 새 프로세서에 대한 지원이 빠른 RTOS 를 선정하는 방법이고 다른 하나는 TRON 이나 POSIX 와 같이 표준을 지원하는 RTOS 를 쓰는 방법이다. 이때

는 표준의 API 를 쓰게 되므로 새 프로세서를 지원하는 다른 RTOS 와 같은 API 를 제공한다면 이 역시 포팅이 쉽게 된다. 그러나 아직 이 표준은 완벽하지 못하며(이에 대해서는 뒤에서 다시 설명한다) 다른 RTOS 가 API 를 지원해야 한다는 문제가 있다. 따라서 가장 좋은 방법은 새 프로세서에 대한 portability 가 좋은 RTOS 를 선정하는 것이다. 새로운 프로세서가 나왔을 때 RTOS 가 어떻게 포트되는가를 조사해보면 그 다음 새로운 프로세서가 나왔을 때 그 RTOS vendor 가 빨리 포트할 수 있을지 도움이 될 것이다. 예전에는 RTOS 가 애플리케이션에 부가하는 오버헤드가 작고 external event 에 대해 빠르고 결정된 반응을 하기 위해서 어셈블리 언어로 작성되었다. 그러나 새로운 프로세서에 포팅이 어려웠기 때문에 최근에는 C 언어로 바뀌왔다. 이는 또한 RISC 프로세서처럼 좋은 compiler 에 의존하는 경우를 생각해 본다면 자연스러운 일이다. 대개 time-critical scheduling 과 task switching routine 은 어셈블리 언어로 작성하고 그 외의 부분은 모두 C 로 작성한다.

(2) Scalability

메모리가 제한이 되어 있는 시스템인 경우에 특히 RISC OS 의 크기도 줄이려는 경우가 있다. 이때 어떤 OS 들은 매우 기본적인 일만 하도록 하는 커넬을 작성한 후 필요에 따라서 추가하도록 한다. 이때 추가가 쉽고 효율적이어야 한다는 것이다.

(2) Multiprocessor support

최근에는 real-time OS 를 embedded system 뿐 아니라 multiprocessor system 에서도 많이 쓰고 있다. 기존의 OS 는 단일 프로세서에서만 작동하도록 되어 있기 때문에 프로세서가 여러개 있는 경우 이를 제어할 OS 가 필요하고 여기에 real-time OS 가 쓰이고 있다. 따라서 multiprocessor 시스템을 설계할 때엔 어떤 real-time OS 가 지원되는지 살펴봐야 할 것이다.

(3) Extended services, Vertical Application

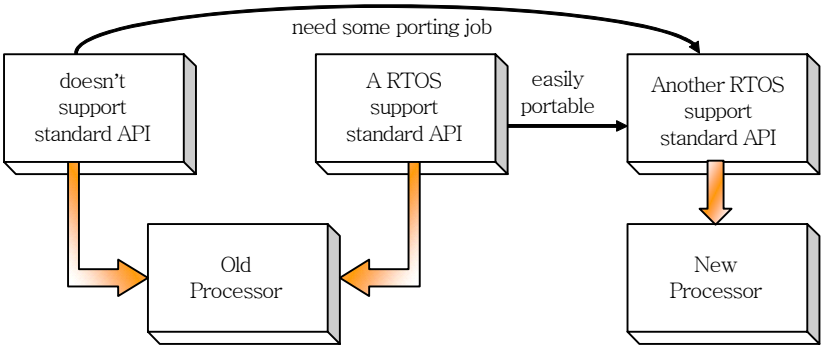
실제 응용프로그램을 만들다 보면 여러 가지 서비스가 필요하다. 예를 들어네트워크과 연계되는 시스템을 설계한다고 할 때 네트워크 프로토콜이 필요할 것이며 가장 대표적으로 TCP/IP, SNMP, ISDN 등이 있다. 또한 File system 을 붙이는 시스템도 있을 것이다.

이런 것들을 위해서 real-time OS vendor 들이 이런 표준들에 대해서는 자신들이 소프트웨어를 만들어서 판매를 한다.

가격은 좀 비싼 편이지만 자체 개발비와 개발기간 등을 고려해 본다면 자신들의 시스템에 적용할 수 있는 것은 사고 나머지를 개발하는 것도 좋은 방법이다.

(4) standards/POSIX compliance

POSIX 는 앞에서도 말한 바 있지만 standard API 이므로 이를 따르는 RTOS 들 사이에서는 더욱 포팅이 쉬워진다. 다음의 예를 보면 왜 표준의 API 를 따르는 것이 좋은 지 이해가 될 것이다.



(그림 6) 표준 API 를 쓸 때의 장점

<참 고 문 헌>

[1] 실시간 운영체제와 임베디드 시스템(www.dpc.or.kr/dbworld/document/200001/tech.html)
[2] www.busanssm.com/~rtos21/main/jaewook_frame.htm
[3] An Introduction to the Real-time OS(REAL TIME KOREA-Accelerated Technology Inc. Korea Representative), 2006.
[4] Operating System Concepts - Silberschatz, Galvin 공저 홍릉과학출판사, 2003.
[5] JANE W. S. LIU, "real-time systems", Prentice hall, 2000
[6] JEAN J. LABROSSE, "MicroC/OS-Ⅱ the Real-Time Kernel", 2nd edition, CMPBooks, 1992.
[7] Steve Furber, "ARM System Architecture", ADDISON-WESLEY, 1996.
[8] David Seal, "ARM Architecture Reference Manual", Addison-Wesley, 2002.
[9] "ARM7 TDMI Data Sheet", ARM DDI 0029E(www.arm.com)
[10] "ARM system-on-chip Architecture, 2nd Edition", Addison-Wesley, 2000.
[11] David Cormie, "The ARM11 Microarchitecture", 2007.
[12] www.arm.com/support/White_Papers
[13] www.picoweb.net