J · BOSN 실시간 운영체제 ③

동기화 및 통신 모델, 서버 모델

J · BOSN 실시간 운영체제는 고성능의 멀티-태스킹(Multi-tasking)을 지원하고 매우 강력한 실시간 응용프로그램을 개발할 수 있는 바탕을 제공하여 준다. J · BOSN 실시간 운영체제의 개념을 알아본 지난호에 이어, 이번호에는 마지막으로 매크로 커널, 디바이스 드라이버, 응용 프로그램 등과 태스크/쓰레드 기능에 대하여 중점적으로 알아본다.

자료제공: 아이보슨시스템즈
www.jbosn.com

J · BOSN 실시간 운영체제

1회 J · BOSN 실시간 운영체제란 무엇인가 I
2회 J · BOSN 실시간 운영체제란 무엇인가 II
3회 J · BOSN의 동기화 및 통신 모델, 서버 모델

소개

J · BOSN 운영체제의 동기화 및 통신 모델을 간략히 살펴보겠다. 제공되는 모든 동기화 및 통신 모델은 빠른 응답시간과 동작의 에러를 최소화하기 위하여 간단 명료한 모델들로 작성 되어 서로간의 통합이나 의존성을 제거하였다. 결론적으로 다른 운용체제에서 발견할 수 있는 매우 복잡하고 서로간의 의존성이 큰 동기화는 찾아볼 수 없지만, 주어진 기능만으로도 개발자들에게 충분하다고 판단된다. 오히려 복잡하고 의존적인 기능들을 모두 이해하여 정상적으로 사용하려할 때, 많은 오류가 발생할 수 있을 것이다.

공유 메모리에 대한 접근을 배타적으로 하지 않을 경우에 여러 개의 쓰레드가 공유 메모리에 접근할 때 문제가 발생하게 된다. 상호배재 알고리즘은 하나의 쓰레드가 크리티컬 섹션을 수행하고 있을 경우에 다른 쓰레드의 크리티컬 섹션이 수행되지 않도록 보장해 주는 알고리즘이다.

공유자원에 대한 접근을 동기화하는 한 가지 방법은 클라이언트-서버모델을 사용하는 것이다. 이 모

델에서는 자원서버(resource server)가 동기화를 담당한다. 공유 자원을 접근하고자 하는 쓰레드는 반드시 자원 서버에 요청을 보내서 자원에 대한 접근 권한을 얻어야 한다. 자원 서버는 이미 설정된 규칙이나 휴리스택(heuristics)에 따라 요청을 수락할 지를 결정한다. 이 방법은 jBOSN 운영체제의 자원관리의 기본을 이루고 있다.

클라이언트-서버 모델은 자원 동기화를 간단하게 해주기는 하지만 자원 서버가 시스템의 병목현상의 원인이 될 수 있다. 이것을 해결하기 위하여 동기화 및 자원관리의 기본적인 도구들은 모두 나노커널이 지원을 하고 동기화 서버/시스템서버는 지원되는 기능을 조합하는 일만을 담당하도록 설계가 되었다. 세마포어, 뮤텍스 등의 동기화기법 등도 이러한 방법으로 구현돼 있다.

동기화란

실시간 시스템용 소프트웨어는 시스템 효율을 극대화하기 위하여 멀티태스킹을 이용한다. 응용프로그램 설계에는 보통 여러 개의 쓰레드가 병행하여 동작하는데 이들 객체간의 작업들을 조율하기 위해서는 쓰레드간 동기화와 통신이 필요하다.

동기화는 자원동기화(resource synchronization)와 동작동기화(activity synchronization)의 두 가지로 나누어 생각할 수 있다. 자원동기화는 여러 쓰레드가 공유자원에 접근 할 때 접근이 안전한지를 검사해 주거나, 안전하게 접근할 수 있는 방법론을 제시하여 준다. 동작동기화는 쓰레드가 특정 상태에 있거나, 특정상태가 될 때를 알려주는 방법론을 제시하여 준다.

자원동기화는 여러 개의 쓰레드가 동시에 같은 자원(공유자원)을 접근할 때 자원의 무결성을 위하여 자원의 접근을 동기화 시켜준다. 제공되는 기능 중 세마포어, 뮤텍스, 크리티컬섹션이 이러한 기능을 제공한다. 예를 들면, 2개의 쓰레드가 하나의 메모리를 공유하고 있다고 가정해보자. 하나의 쓰레드는 공유메모리에 데이터를 쓰고, 다른 쓰레드는 공유메모리에서 데이터를 읽는 동작을 한다. 쓰는 동작을 하는 쓰레드가 데이터를 쓰고 있는 중간에 읽는 쓰레드가 데이터를 읽어가면 데이터의 완결성이 없고 이전의 데이터와 현재의 데이터가 혼합된다.

동작동기화는 다중 쓰레드로 작성된 프로그램이 제대로 동작을 하려면 쓰레드의 동작을 다른 쓰레드의 동작의 결과의 조건이나 상태에 맞추어서 동기화를 시켜줘야 한다. 이것이 동작 동기화는 조건 동기화 또는 순차동기화라도 불리기도 하는 이유이다.

동작동기화는 동기적일 수도 있고 비동기적일 수도 있다. 제공되는 동기화 기능중 이벤트나 메시지 시그널이 대표적이다. 하나의 쓰레드는 여러 개의 쓰레드가 실행한 결과를 입력으로 받아서 일을 하는 예가 많다. 이러한 동작을 구현하기 위해서 동작동기화를 사용한다.

동기화를 위하여 제공되는 모델은 세마포어(semaphore), 뮤텍스(mutex), 이벤트(event), 크리티컬섹션(criticalsection), 조건변수(conditional variables), 메시지시그널(message signal) 등이 있다. 세마포어는 이진 세마포어와 카운트세마포어를 구현할 수 있고, 뮤텍스는 재귀적 잠금과 소유권기능을 가지고 있다.

통신을 위하여 제공되는 모델은 메시지버스(message bus)가 존재한다. 통신을 위한 모델을 최소한으로 제한한 것은 j-BOSN 운영체제는 단일 메모리 공간에서 동작을 하고 있으며, 쓰레드간의 의존성을 높였고, 태스크간의 의존성을 최대한으로 차단한다는 가정하에서 설계를 하였고, 데이터 통신으로 인한 메모리 내용의 이동을 최대한 제한하여 시스템의 전체 성능을 높이기 위해서이다. 데이터 통신을 필요로할 때는 동기화 모델을 사용하여 자원(특히 메모리)의 동기화 기법을 사용하는 것을 추천한다.

또 다른 측면에서 보면, 멀티태스킹을 지원하는 시스템에서 쓰레드는 다른 쓰레드들과 통신을 사용하여 상호협력을 하면서 하나의 작업을 수행한다. 이 때 통신이란 쓰레드간의 데이터를 주고받는 행위를 말한다. 데이터란 특정 상태를 나타내는 신호나 실제 값을 나타내는 데이터 두 가지가 있다. 통신을

하는 방법은 신호를 전달하는 메시지 시그널이나 이벤트 등의 방법이나 실제 데이터를 전달하는 메시지버스 방법 등이 있을 수 있다. 그러나 실제 구현에서는 제한된 방법을 사용하는 것보다 다양한 동기화 기법을 사용하여 데이터 통신을 구현하는 경우가 많다.

결론적으로 이상의 기능들을 사용하면 데이터 전송, 쓰레드의 현재 상태를 알려주기, 다른 쓰레드 실행의 간접제어, 다른 쓰레드와 동작동기화를 이룰 수 있으며 또한 자원을 공유하기 위하여 여러 방법을 스스로 구현할 수 있다.

세마포어(semaphore)

여러 쓰레드들이 동시에 동작하는 멀티태스킹 기능은 필수적으로 쓰레드간의 자원을 접근을 할 때 경쟁을 하게 된다. 이때 하나의 자원을 독점적으로 접근할 권한을 부여하는 방법이 있어야만 쓰레드간의 자원 사용에서 충돌을 피할 수 있다. J-BOSN 운영체제는 세마포어를 제공하여 이런 기능을 수행할 수 있게 한다.

세마포어는 실행중인 여러 쓰레드가 동기화 또는 상호배재를 목적으로 획득하거나 반환할 수 있는 커널 자원이다. 세마포어는 어떤 일을 수행할 때나 공유자원을 접근할 필요한 열쇠와 비슷하다. 태스크가 세마포어를 획득할 경우 원하는 동작을 수행하거나 공유자원에 접근할 수 있지만, 획득을 실패할 경우 세마포어가 획득할 때까지 대기행렬에서 기다린다. 세마포어의 종류는 바이너리 세마포어와 카운팅 세마포어로 나눌 수 있는데, J-BOSN 운영체제는 두 가지를 하나의 방법으로 구현을 할 수 있게 제공한다. 세마포어의 내부 카운터의 값이 '0' 일 경우 세마포어는 사용불가능하고 이외의 값을 가지고 있을 때는 그 값만큼 사용할 수 있다. 제한조건으로 최대로 증가할 수 있는 값(상한선)을 설정하여 무한히 증가하는 것을 막을 수 있게 되어 있다.

세마포어의 내부에 최대값을 '1' 로 설정을 하면 바이너리 세마포어를 구현할 수 있고, '1' 보다 큰 값을 설정하면 카운팅 세마포어를 구현할 수 있다. 세마포어의 카운터 값은 현재 사용가능한 자원의 수를 나타내고 있으므로 세마포어를 설정할 때, 동시에 사용가능한 자원의 수를 최대값으로 설정을 하면 된다.

세마포어 자원을 얻을 때 카운터값을 1씩 감소하고, 세마포어 자원을 반환할 때 1씩 증가한다. 카운터값이 '0' 일 때는 더 이상 자원이 없으므로 자원이 반환될 때까지 대기행렬에서 기다린다. 세마포어는 소유권이 없으므로 세마포어 핸들을 알고 있는 모든 쓰레드에서 접근이 가능하다.

뮤텍스(mutex)

뮤텍스는 아주 특수한 제한조건을 가지고 있는 바이너리 세마포어의 일종이다. 일반적인 세마포어는 자원의 사용가능 개수를 나타내는 것과는 달리, 뮤텍스는 자원의 잠금과 풀림을 나타낸다. 뮤텍스는 자원의 잠금을 나타내기 때문에 획득을 하는 즉시 잠김 상태로 천이를 하게 되고, 반환을 하면 바로 풀림상태로 천이를 한다.

뮤텍스를 획득하면 세마포어와는 달리 소유권 동시에 획득하기 때문에 다른 쓰레드에서 뮤텍스에 접근하는 것 자체가 불가능하게 된다. 따라서 획득한 쓰레드가 직접 뮤텍스를 반환하는 것을 기다려야 한다. 이러한 기능은 다른 쓰레드에 의해서 동기화가 방해를 받지 않고 안정적으로 동작하는 것을 보장하여 준다.

다른 기능으로는 재귀적 잠금을 지원하는 것이다. 재귀적 잠금이란 뮤텍스를 획득한 쓰레드는 반복적으로 해당 뮤텍스를 획득할 수 있다. 뮤텍스는 이 경우 반복적으로 획득된 숫자를 계산하고 있다가 뮤텍스를 반환할 때 숫자를 감소시킨다. 획득된 숫자만큼 반환이 되었을 때 뮤텍스 자원을 완전히 반환이 되어 소유권을 내놓게 된다. 이 기능은 뮤텍스를 획득하는 기능을 가지고 있는 함수가 반복적으로 호출이 될 경우에 매우 유용하게 사용될 수 있다. 세마포어처럼 상한선을 가지고 있지는 않다.

크리티컬 섹션(criticalsection)

크리티컬 섹션은 공유자원을 접근하는 코드의 집합을 지칭하는 말이다. 서로 다른 쓰레드에서 공유자원을 접근할 수 있는데, 다른 동기화 방법과는 달리 실행되는 코드집합에 진입하는 것을 제한하는 방법이다. 제일 먼저 크리티컬 섹션에 진입을 하는 쓰레드는 다른 쓰레드의 방해를 받지 않고 공유자원을 독점적으로 사용하는 코드집합을 수행한 후에 크리티컬 섹션에서 나오게 된다. 다른 쓰레드는 독점적인 사용이 끝날 때까지 대기한다. 하지만 이 경우 우선순위 역전현상과 크리티컬 섹션의 크기와 진입한 쓰레드의 수행 속도에 따라서 시간적으로 매우 중요한 쓰레드가 실행이 되지 못하는 경우도 발생한다. 왜냐하면 크리티컬 섹션을 수행하지 않는 다른 쓰레드와는 경쟁관계에 놓이지 않기 때문에 다른 쓰레드는 수행에 방해를 받지 않아 CPU 자원을 빼앗아 갈 수 있다. 기본적으로 j-BOSN 운영체제는 크리티컬 섹션을 제어하는 기능을 제공하고 있지만 시간적으로 매우 민감한 코드는 좀 더 확실한 방법을 사용하기를 권장한다.

많은 코드를 살펴보면 크리티컬 섹션을 보호하기 위해서 인터럽트 잠금이나 선점 잠금과 같은 상호배재 기능을 사용하는 것을 볼 수 있다. 이 경우 크리티컬 섹션을 매우 짧게 작성하지 않으면 시스템의 성능에 치명적인 부담이 될 수 있다. 하지만 시간적으로 매우 중요한 쓰레드는 위에 언급된 것을 희생하더라도 다른 외부사항의 방해없이 시간적인 요구사항을 만족시켜 줄 수 있다. 사용자는 어떤 것을 중시하느냐에 따라 크리티컬 섹션을 구현하는 방법을 선택할 수 있다.

이벤트(event)

이벤트는 쓰레드에서 다른 쓰레드와의 실행 동기화를 맞추기 위해서 사용하는 기능이다. 다른 동기화 기능들은 공유자원의 사용에 대한 것이나 이벤트는 쓰레드가 실행할 때 자신의 실행 상황을 다른 쓰레드에게 알려주는 기능을 한다. 다른 쓰레드는 이벤트자원으로부터 다른 쓰레드가 이미 알려준 정보를 보고 상황을 추측할 수 있다. 부가적으로 이벤트를 발생시킬 때 데이터를 전송시킬 수도 있는데, 이 데이터는 보호를 받지는 못하고 다른 데이터가 새로 전송이 되면, 곧바로 새로운 값이 써지게 된다. 결론적으로 가장 최근에 전송된 데이터가 남아 있을 것이다. 쓰레드가 이 값을 읽더라도 지워지지 않고 남아있다.

이벤트는 쓰레드가 실행을 할 조건으로 다른 쓰레드가 정하여진 상태로 진입하는 것을 구현할 때 매우 유용하다. 또한 데이터를 전송할 수 있어서 결과값을 전송 받을 수도 있다.

조건 변수(conditional variable)

조건변수는 쓰레드에서 다른 쓰레드와 실행 동기화를 맞추기 위해서 사용하는 기능이다. 다른 동기화와 다른 점은 조건변수를 기다리는 쓰레드는 해당하는 조건을 만족하도록 변수가 변할 때까지 기다리는 것이다. 이 때 조건은 BITMAP-AND 연산을 수행하는 경우를 구현했다. 즉, 변수가 변하기를 기다리는 조건으로 여러 비트를 할당을 하고, 만일 여러 해당되는 비트중 하나만이라도 세팅(Set) 되더라도 자원을 획득한 것으로 인식을 하는 것이다. 이러한 기능은 여러 개의 자원 중 최소한 하나를 기다릴 때 매우 유용하게 사용될 수 있다.

메시지 큐(message queue)

메시지 큐는 쓰레드에서 다른 쓰레드로 데이터를 전달하기 위하여 사용하는 통신기능이다. 다른 동기화와 다른 점은 대용량의 데이터를 전달할 수 있다는 것이다. JBOSN RTOS의 메시지큐는 private buffer를 커널의 내부에 생성을 하여 데이터를 보내는 쓰레드에 존재하는 데이터를 커널내부에 존재하는 private buffer에 복사한다. 또한 데이틀 받는 쓰레드는 데이트를 커널 내부의 private buffer에서 복사를 해서 사용하게 된다. 따라서, 대용량의 데이터를 보낼 때는 메모리의 복

사로 인한 지연을 고려하여 사용해 한다. 커널 내부의 private buffer는 페이지단위로 할당이 되어 사용되므로 메시지 큐의 생성시 이 점을 고려하여 메시지 큐를 생성하여야 한다.

메시지시그널(message signal)

메시지 시그널은 j-BOSN 운영체제가 메시지버스를 통하여 비트맵 시그널을 전송하는 방법이다. 이 기능은 커널의 다양한 기능을 메시지버스에 구현하려는 시도에서 탄생했다. 쓰레드들은 이 기능을 통하여 특별한 기능을 요청하거나 자신의 상태를 알려줄 수 있다. 이벤트는 수신측의 상황에 따라 수신을 반드시 하지 않아도 되지만, 메시지 시그널은 수신이 보장이 되므로 확실한 전송을 할 수 있다.

이 기능은 마이크로 커널이나 커널 레이어에서 실행되는 서비스 서버들이 자주 이용하는 방법으로 사용자는 매우 주의를 하여 사용하여야 한다.

서버 모델

소개

이제 J · BOSN 운영체제의 서버의 작성 모델을 간략히 살펴보겠다. 제공되는 모든 서버는 설명되는 모델에 기반하여 작성되어 있다. 디바이스 드라이버 또한 서버로서 동작을 하기 때문에 아래의 모델에 기준하여 작성된다.

서버는 기본적으로 클라이언트(client)의 요청을 서비스해주는 역할을 하므로 기본적으로 갖추어야 요건은 요청을 받을 통신 채널과 서비스를 제공하는 채널을 갖추어야 하고 또한 여러 개의 클라이언트가 서비스를 요청할 경우 요청을 관리할 수 있어야 한다. J · BOSN은 이러한 기능을 메시지 통신을 통하여 제공하고 있고, 메시지 통신의 양단에 포트(port)라는 개념을 도입하여 해결했다. 비유를 하자면, 컨테이너화물을 배에 선적하여 원하는 목적지로 보내는 것을 생각해 보자. 예를 들어, LA에서 부산으로 화물을 보낸다고 해보자. LA는 LA항구에 정박한 화물선중 부산항을 거쳐가는 화물선을 찾아서 컨테이너를 선적할 것이다. 그러나, 해당하는 화물이 어느 항구에서 내려야하는 것인지는 화물선입장에서는 알 수 없다. 따라서, 컨테이너는 하역하여야할 항구에 해당하는 일렬번호를 부여받아야 한다. 이 일렬번호를 보고 화물선을 해당하는 컨테이너를 부산항에 내려줄 것이다. 즉, LA항이나 부산항이 'port'에 해당하고, 일렬번호가 바로 'channel key'에 해당한다.

또 다른 예를 들면, 전화를 사용하려면 통신을 할 수 있도록 통신 선로를 연결하고 양 끝단에 전화기를 설치하는 것과 같다. 그러나 양 끝단은 항상 전화를 거는 사람이나, 받아주는 사람이 있어야 한다. 즉, 메시지는 클라이언트가 지정한 포트를 출발하여 서버에 직접 전달되지 않고 서버가 지정한 포트로 전달이 되며, 서버는 이곳에서 서비스 요청을 얻어 서비스를 제공하게 된다. 클라이언트 또한 자신의 요청에 대한 서비스가 도착할 때까지 미리 지정한 클라이언트 포트에서 기다리고 있다.

서버의 포트는 클라이언트에게 알려져 있지 않아 클라이언트가 서버를 접근할 방법이 없다. 이를 해결하기 위해서 서버는 클라이언트의 접근을 허용하기 위한 통신채널 키(channel key)를 제공하게 된다. 이 채널 키를 이용하여 클라이언트는 서버에 요청을 보내고 메시지 통신 채널은 이 채널 키를 해석하여 메시지를 서버에 전달을 하여준다. 결론적으로 클라이언트는 이미 공개되거나 자신이 얻은 채널키(channel key)를 통하여 서버에 접근을 할 수 가 있고, 서버는 자신이 가지는 포트를 통하여 서비스 요청을 기다리며, 서비스는 메시지가 처음 출발한 포트로 전달된다.

이러한 기능은 **그림 1**과 같은 형태로 구현이 된다.

그림 1은 클라이언트가 서비스를 요청하는 순서를 나타낸 것이다. 첫 번째로 서버가 서버 포트를 통하여 메시지를 기다리고, 둘째로 클라이언트는 공개키(channel key)를 이용하

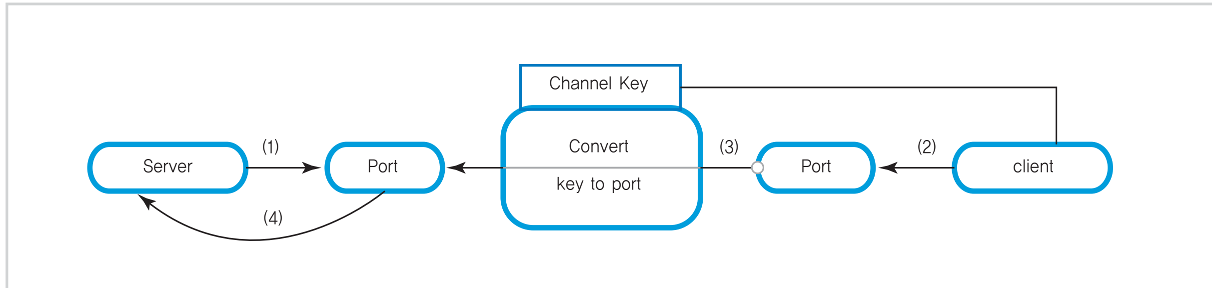

그림 1. 메시지를 이용한 클라이언트의 서비스 요청 구조

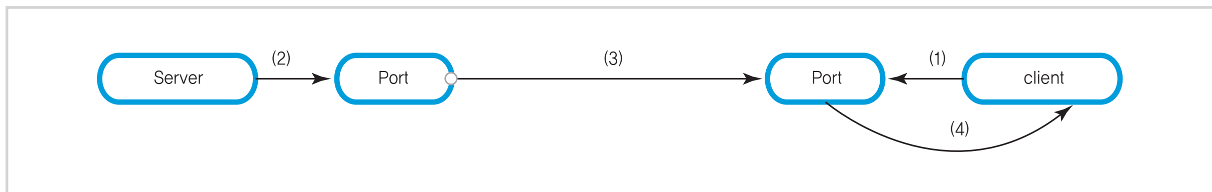

그림 2. 메시지를 이용한 서버의 서비스 응답 구조

여 자신의 포트를 통해 서비스 요청 메시지를 전달한다. 셋째로 이 공개키는 포트로 해석되어 서버의 포트에 메시지를 전달한다. 넷째로, 서버는 서버 포트를 통해 메시지를 수신하게 된다.

그림 2는 서버가 요청받은 서비스에 대한 응답을 하는 순서이다. 첫째로 클라이언트는 서비스 요청을 한 후에 자신의 포트에서 응답이 오는 것을 기다리게 된다. 둘째로 서버는 응답을 서버 포트를 통하여 서비스를 요청한 포트로 응답 메시지를 전송하게 된다. 셋째로 응답 메시지는 서버 포트에서 클라이언트 포트로 전송이 되고, 넷째로 클라이언트는 자신의 포트로부터 메시지를 수신하게 된다.

메시지 구조

위의 메시지 통신 체계는 J · BOSN 운영체제 나노커널의 "Message Management" 모듈에서 전적으로 책임을 지고 서비스를 해준다. 이제 이를 이용하여 서버와 클라이언트 모델을 작성해 보자. 메시지 전달의 과정은 복잡하지만 실제 사용자가 사용하기는 매우 단순한 구조로 되어 있어 작성하기는 쉽다.

사용자가 고려해야할 것은 메시지의 데이터 구조와 메시지 통신을 위한 함수뿐이다. 서버와 클라이언트도 이 두가지만 가지고 작성된다.

메시지 구조체는 **그림** 3에서 보듯이 크게 2개로 나눠진다. 우선 줄1부터 줄3까지는 공통으로 사용되는 메시지의 데이터 구조이다. 이 변수를 각각 어떻게 사용하던지 문제가 되지는 않는다. 단지 변수 "Type"에는 서비스를 받고자 하는 서비스 종류를 정하는 것으로 사용하고 있고, 나머지 변수는 서비스 요청시 필요한 변수를 전달하는 것에 사용하고 있다. 메시지가 서버에서 클라이언트로 응답으로 사용될 때는 "Type" 변수는 서비스의 성공, 실패 등의 리턴 값을 담는 공간으로 사용되고 있다. 줄4 부터 줄7까지의 JB_MESSAGE는 클라이언트가 서버에게 요청을 할 때 사용되는 것이다. 줄8 부터 줄13까지의 JB_RECVMESSAGE는 서버가 클라이언트의 요청을 받아들일 때 사용되는 구조체이다. 특히 변수 "h_Sender" 는 서비스를 요청한 클라이언트의 정보가 들어가는 중요한 변수이다. 따라서 서버는 이 변수를 잘 보관하여 다음에 서비스 응답을 할 때 사용할 수 있다. 변수 "Handle"은 디바이스 드라

```
1. /// Message definition
2. #define JB_MESSAGE_CONTENTS   \
3.    UINT32 Type;               \
4.    UINT32 Param;              \
5.
6. typedef struct _JB_MESSAGE
7. {
8.    JB_MESSAGE_CONTENTS
9. } JB_MESSAGE, *JB_PMESSAGE;
10.
11. typedef struct _JB_RECVMESSAGE
12. {
13.    JB_MESSAGE_CONTENTS
14.
15.    HANDLE         hSender;
16.    HANDLE         Handle;     // NOT FOR USER: only for device
handle on io_subsystem
17. } JB_RECVMESSAGE, *JB_PRECVMESSAGE;;
```

그림 3. 메시지 구조체

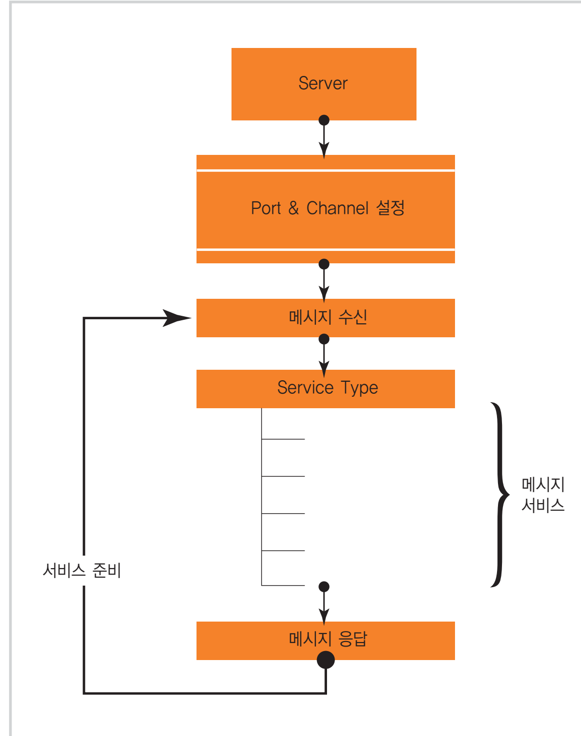

그림 4. 서버의 구조–블록다이어그램

이버 서버에만 해당하는 변수로서 디바이스 드라이버 서버는 이 변수를 보고 자신이 접근을 허가한 쓰레드로부터 요청이 들어온 것인지 아닌지를 판별할 수 있다.

서버 구조

그림 4는 서버의 구조를 블록다이어그램으로 나타내고 있는 예제이다. 필수적으로 있어야 하는 부분만을 나타내고 있다. 특히 "Port & Channel 설정" 부분은 통신을 위하여 반드시 필요한 부분이다. 각 부분의 보다 자세한 내용은 아래의 코드예에서 참조하기 바란다.

그림 5는 서버의 코드의 예를 나타내고 있다. 줄2에서 서버의 프로토타입(prototype)은 VOID server(UINT32)로 구성이 되어 있어 서버를 생성시킬 때 argument를 전달 할 수 있다. 줄4는 메시지 구조체에 해당하는 선언을 하고, 줄5는 서버의 포트의 핸들을 선언한다. 줄6에서 서버의 포트를 할당을 받고 줄7에서 메시지를 수신할 수 있는 모드로 포트를 설정한다. 줄8에서는 할당받은 포트를 이용하여 서버의 공개 채널 키를 할당받고 외부에 공개를 한다. 이제 모든 객체의 할당을 받아 서비스 요청 메시지를 기다리고 서비스를 제공할 수 있게 되었다. 줄12에서 메시지를 수신하려 준비를 한다. 줄13, 줄14는 정상적인 메시지 수신인가를 검사한다. 만일 정상적인 메시지 수신이 아니고 에러가 발생할 경우는 다시 메시지 수신모드로 전환이 된다. 메시지가 정상적으로 수신이 되었을 경우 줄15에서 메시지 타입(Type) 항목을 보고 어떤 서비스를 요청하였는가를 결정한다. 그리고 그에 대한 서비스를 실행을 하고 줄24에서처럼 그에 대한 응답을 메시지 변수에 들어있는 타입에 넣어서 줄28처럼 클라이언트에 보내게 된다. 주의할 점은 메시지 변수에 들어있는 변수중 "hSender"는 수정해서는 안된다. "hSender"는 메시지를 보

```
1. PUBLIC BOOL
2. Server (UINT32 Arg)
3. {
4. JB_RECVMESSAGE          Message;
5. HANDLE                  hPort;
6. hPort = PortGet ();
7. PortSetType( hPort, ORTTYPE_MESSAGE|PORTTYPE_INTER-
RUPT|PORTTYPE_SIGNAL );
8. Channel = ChannelGet (hPort);

  // if this server is device deiver,
  // then, here, register the device driver

9. while (1)
10. {
11.   UINT32 Ret;
12.   Ret = MessageReceive (hPort, &Message);
13.   if (Ret != ERR_SUCCESS)
14.     continue;
15.   switch (Message.Type)
16.   {
17.     case MESSAGE_TYPE_SIGNAL:
18.       // do work, here
19.       continue;
20.     case MESSAGE_TYPE_INTERRUPT:
21.       // do work here
22.       continue;
23.     case MESSAGE_TYPE_DEVICE_READ:
24.       // do work here
        Message.Type = 0;
25.     break;
26.   }
27.   // return the results value. The return value is Message.Type
28.   MessageReply(&Message);
29. }
```

그림 5. 서버의 구조 – 코드

내 쓰레드를 나타내고 있기 때문에 이 변수를 수정하면 서비스를 응답해줄 포트를 잃어버리게 된다. 또 하나는 줄17이나

```
1. JB_MESSAGE    Message;

2. Message.Type = MESSAGE_TYPE_DEVICE_CLOSE;
3. MessageSendRecv( h_Key, &Message);

4. return Message.Type;
```

그림 6. 클라이언트의 구조

줄21의 두 경우에는 응답을 할 클라이언트가 없는 경우이므로 절대로 줄28의 메시지 응답을 불러서는 안된다. 서비스를 요청할 때, 전달된 타입이나 변수들이 잘못되었거나 서비스가 실패를 하면 타입 변수에 실패했다는 값을 설정하고 메시지를 응답한다. 이때, 시스템 콜인 "ThreadSetLastError"를 사용하여 실패한 이유를 클라이언트에게 알려야 한다.

클라이언트 구조

그림 6은 서비스를 요청하는 클라이언트의 구조를 나타내고 있는 예제이다. 실제로 이러한 구조는 라이브러리 형태로 만들어져 사용자에게 배포가 된다. 줄1 에서는 클라이언트용 메시지 구조체를 정의하고 줄2에서는 서비스를 요청할 서비스 타입을 결정한다. 이 예제에는 다른 변수를 사용하고 있지 않다. 줄3에서는 공개된 h_Key를 사용하여 변수를 전달한다. 줄5는 응답으로 온 메시지에서 타입 변수를 검사하면 서비스의 성패를 알 수 있다.

만일 실패하였을 경우 "ThreadGetLastError" 시스템 콜을 사용하여 실패한 이유를 알 수 있다. 지금까지 서버와 클라이언트의 제작 방법에 대하여 알아보았다. 〈연재끝〉 RealTime