J · BOSN RTOS

J · BOSN 실시간 운영체제 포팅 가이드

J · BOSN 실시간 운영체제는 고성능의 멀티-태스킹을 지원하고 매우 강력한 실시간 응용프로그램을 개발할 수 있는 바탕을 제공하여 준다. 특히 운영체제의 기능이 모듈화 되어 있어 사용자가 개발하는 실시간 시스템에 맞게 운영체제를 재구성할 수 가 있다. 지난호에 이어지는 〈J · BOSN 실시간 운영체제를 이용하여 포팅하는 방법〉에는 실제 하드웨어에 연결하는 방법과 드라이버 탑재 기술을 자세히 소개한다.

자료제공: 아이보슨시스템즈
www.jbosn.com

포팅 가이드(Porting Guide)

①J · BOSN 실시간 운영체제 개발 시스템 패키지 구조, 이식
②하드웨어 연결 및 모듈 탑재, 드라이버 탑재

개발 시스템에 탑재하기

JBOSN 운영체제의 하드웨어 연결

이제부터는 HAL 레이어를 작성하여 JBOSN 운영체제의 나노커널을 개발 시스템에 탑재하는 방법을 소개한다.

"PLATFORM/[BSP]/PORT/HAL" 디렉토리 밑에 있는 파일은 개발 시스템의 전체적인 설정과 나노커널의 하드웨어 제어에 관한 가장 기본적인 함수와 변수를 포함하고 있다. 매우 간단하고 쉽게 작성할 수 있기 때문에 JBOSN 운영체제를 개발 시스템에 매우 쉽게 이식할 수 있다. 개발 시스템이 처음 시작을 할 때, 기본적인 운영체제의 설정을 마치고 하드웨어와 관련된 기본설정을 하기 위하여 여기에 있는 변수를 참조하게 되고, 함수를 호출하게 된다. 또한 이곳에서 기본적으로 사용할 디바이스 드라이버와 필요한 최소한의 시스템 모듈을 선택, 조립, 연결시키게 된다. 다음은 "PLATFORM/[BSP]/PORT/HAL" 디렉토리에 반드시 있어야할 파일들이다.

• hal_locore.S : 시스템의 초기시작 및 운영체제의 configuration을 설정하고 Exception Table을 작성한다.
• hal_interrupt.c : 인터럽트에 관련된 사항을 설정한다.
• hal_main.c : 사용하려는 시스템 모듈을 선택, 조립, 연결시킨다.
• hal_time.c : 시간과 관련된 하드웨어의 초기화와 설정을 한다.
• idle.c : 시스템의 전력관리가 실행된다.(시스템모드를 변경한다)
• panic.c : 시스템이 동작이상이 생겼을 때, 호출된다.
• console.c : 기본적인 디버깅과 panic메시지를 볼 수 있는 개발플랫폼과의 채널을 설정한다.
• hal_kernelLoad.c : JBOSN RTOS의 커널을 로딩한다.(option)

hal_locore.S

이 파일은 시스템이 제일 처음 기동되었을 때 시작하는 부분으로 모두 어셈블리어로 작성이 되어 있습니다. 첫 번째 기능은 먼저 'exception table'을 작성하여 나노커널의 exception과 연결을 시켜준다. 연결시킨다는 것은 크게 두 가지로 해석을 할 수 있습니다. 첫째로, 나노커널이 배치된 메모리 영역에서 '나노커널'에 존재하는 'exception table'을 순서적으로 매칭하여 호출을 하는 방법이 있을 수 있다. 두 번째로, 나노커널에 존재하는 'exception table'을 하드웨어가 요구하는 'exception table' 영역으로 복사하는 방법이 있다.

두 번째는, 시스템을 초기화 해주는 부분이다. 보통은 JBOOT를 사용하면, JBOOT에서 일반적인 초기화가 수행되었을 것이다. 그러나 JBOOT가 사용되지 않았다면 이 부분에서 시스템의 초기화를 반드시 해주어야 한다.

세 번째는, 현재 HAL 레이어를 적당한 메모리영역으로 재배치를 수행하여, 다음의 명령을 수행할 완전한 준비를 수행한다. 여기서 주의할 점은 반드시 HAL 레이어에서 사용할 메모리는 스스로 초기화 해주어야 한다. 예를 들어, 코드가 수행할 영역, 데이터가 존재할 영역으로 재배치한 후, BSS 영역은 반드시 초기화가 필요하다.

네 번째로, 잠시 동안 사용할 임시스택을 설정한다. 보통은 다른 모듈이 사용하지 않는 영역 중에서, RAM 중에 최상위 주소로 설정하는 것이 일반적이다.

다섯 번째로, 디버깅이나 panic을 위해서 시리얼통신을 사용한다면, 함수 'PanicInit'를 호출한다.

여섯 번째로, JBOSN RTOS의 커널들을 로딩하여 메모리에 탑재한다(option).

일곱 번째로, 나노커널에 전달할 시스템 설정을 레지스터로 로딩한 후에 나노커널이 시작하는 부분으로 제어권을 넘겨준다. 통상적으로 나노커널이 필요한 정보(함수 호출의 인자들(arguments))는 아래와 같다.

Argument 1: NKERNEL_BOOT pointer (hal_main.c에 정의되어 있다)

Argument 2: IDLE thread configuration pointer (idle.c에 정의되어 있다)

Argument 3: 커널의 서버 configuration pointer (kernel_main.c에 정의되어 있다)

Argument 4: 나노커널이 사용할 스택의 맨 윗부분의 주소

hal_interrupt.c

이어서 오는 모든 파일은 나노커널의 하드웨어와 관련된 함수들이다. 이 함수들이 나노커널과 하드웨어를 연결시켜주는 역할을 하게 된다.

이 파일은 J•BOSN 운영체제의 인터럽트관리 시스템 중에 하드웨어관련 부분이 정의되어 있다. 이 파일에 있는 함수를 이용하여 J•BOSN 운영체제의 인터럽트와 관련된 부분이 통합/일관성 있게 관리된다.

또한 나노커널은 여기서 정의된 함수를 이용하여 디바이스 드라이버에서 요청한 인터럽트 관련 호출을 수행하게 된다.

이곳에 구현된 루틴은 최대한 간략하고 신속하게 실행되도록 작성이 되어야 하고, 디바이스 드라이버의 작성을 염두해 두고 기능을 조심스럽게 구현해야 한다. 작성되는 함수들은 J•BOSN 운영체제의 I/O 시스템과는 전혀 무관하고 오직 디바이스 드라이버와 관련이 되어 있다. 따라서 함수를 작성할 때 J•BOSN 운영체제의 I/O 시스템에 미치는 영향을 고려하지 않아도 되고 오직 작성될 드라이버와의 관계만을 염두하여 작성하면 된다. 결론적으로 이곳의 문제가 다른 운영체제들처럼 시스템 전체로 전파되지 않고 대응되는 드라이버에만 영향을 미치기 때문에 안정적으로 J•BOSN 운영체제를 운영할 수 있고, 또한 뛰어난 안정성을 보장해 준다. 단 여기에서 사용되는 모든 인터럽트 번호는 하드웨어적으로 주어진 번호를 사용하지 않고 사용자가 개별적으로 정한 인터럽트 번호(가상 인터럽트 번호)를 사용할 수 있게 되어 있다. 가상 인터럽트 번호는 사용자가 직접 정하여 'PORT/INCLUDE/hal_interrupt.h' 화일에 정의를 하면 된다. 'hal_interrupt.c' 에서 제공되어야 할 기본적인 함수는 다음과 같다.

- PUBLIC VOID HAL_InterruptInit (VOID)
- PUBLIC VOID HAL_InterruptEnable (UINT32 nInterrupt)
- PUBLIC VOID HAL_InterruptDisable (UINT32 nInterrupt)
- PUBLIC VOID HAL_InterruptDone (UINT32 nInterrupt)
- PUBLIC UINT32 HAL_InterruptHandler (VOID)

함수 'HAL_InterruptInit' 에는 하드웨어의 인터럽트 관련된 부분을 초기화한다. 이 함수는 J•BOSN 운영체제가 시작하여 하드웨어 초기화를 하는 것 중 가장 빨리 호출된다. 모든 인터럽트는 반드시 차단이 되어야 하고, 또한 현재 사용하려는 방향으로 인터럽트 관련 하드웨어가 초기화 돼야 한다.

'HAL_InterruptEnable' 함수는 드라이버에서 해당하는 인터럽트의 처리를 하기위한 모든 준비를 완료하여 인터럽트를 개방하려고 호출을 한다. 이 함수에는 한 개의 변수가 주어지는데 'nInterrupt' 는 개방하려는 사용자가 이미 지정한 가상 인터럽트의 번호이다. 이 함수는 해당 인터럽트가 발생할 수 있도록 설정하는데, 가능하면 인터럽트 콘트롤러만을 제어하는 것을 추천하며, 각 디바이스의 인터럽트에 관련된 부분은 디바이스 드라이버에서 제어를 하는 것이 바람직하다.

'HAL_InterruptDisable' 함수는 드라이버의 문제로 인하여 해당하는 인터럽트의 처리를 더 이상 하지 못하는 상황에서 호출이 되거나 드라이버의 준비가 아직 되지 않았다고 생각되었을 때 시스템이 자동으로 이 함수를 호출한다. 이 함수에는 한 개의 변수가 주어지는데 'nInterrupt' 는 차단하려는 사용자가 이미 지정한 가상인터럽트 번호이고 해당하는 인터럽트는 더 이상 발생하지 않도록 제어해야 한다. 인터럽트가 더 이상 발생하지 않도록 해당 디바이스를 제어하는 것 보다 인터럽틀 컨트롤러를 제어하여 인터럽트가 전달되지 못하도록 하는 것이 바람직하다.

'HAL_InterruptDone' 함수는 드라이버가 관련된 인터럽트의 발생을 통보 받고 나서 해당 인터럽트에 대한 서비스를 모두 마친 다음 다시 인터럽트가 발생하더라도 디바이스의 동작이나 드라이버의 수행에 아무런 문제가 없다고 판단이 되는 시점에 호출을 하게 된다.

이 함수는 한 개의 변수를 받게 되는데 'nInterrupt' 는 다시 개방되어야 하는 가상 인터럽트 번호를 나타내고 해당하는 가상의 인터럽트는 다시 발생하여 드라이버에 서비스를 요청할 수 있다. 이 함수는 'HAL_interruptHandler' 와 대응하여 동작을 하게 되므로 'HAL_InterruptHandler' 의 해당 가상 인터럽트 번호의 처리 루틴과 상호 대응되게 작성이 되어야 한다.

'HAL_InterruptHandler' 함수는 하드웨어적으로 발생하

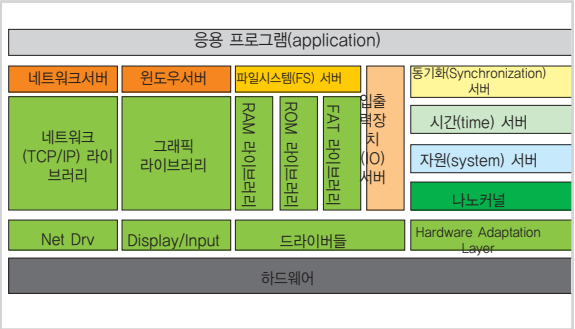

그림 1. JBOSN 라이브러리 개념도

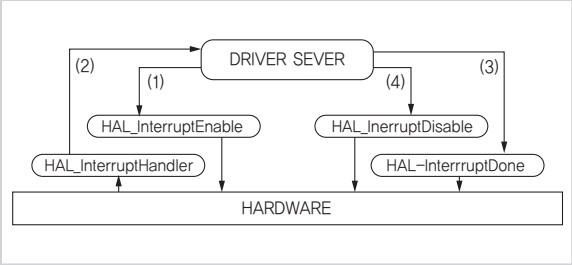

그림 2. 인터럽트의 전달 순서 및 제어 구조

는 하드웨어 인터럽트를 사용자가 정한 가상 인터럽트 번호로 변화시켜 주고 동시에 인터럽트를 발생시킨 디바이스의 동작성에 영향을 미치지 않는 범위내에서 해당하는 인터럽트가 더 이상 발생하지 못하도록 만든다. 이 함수가 인터럽트 관리의 핵심이고 또한 수행이 될 때 모든 인터럽트의 발생이 제한을 받게 되므로 최소한의 필요한 동작만을 하도록 권장한다. 만일 이곳의 루틴이 길어질 경우 시스템 전체의 interrupt latency가 길어지게 되고 주어진 Task의 실시간 처리에 영향을 미치게 된다. HAL_interruptHandler는 나노커널이 하드웨어 인터럽트를 받았을 때, 발생한 하드웨어 인터럽트를 가상 인터럽트 번호로 변환을 하기 위해서 호출이 된다. 주의할 점은 하드웨어적으로 발생한 인터럽트가 없다는 '0'을 리턴하기 까지 지속적으로 호출이 된다.

그림 2는 JBOSN 운영체제에서 인터럽트의 전달과 제어가 어떻게 진행되는지를 보여주고 있다. 드라이버는 'HAL_InerruptEnable' 함수를 호출하여 인터럽트를 개방시키고 해당하는 하드웨어 디바이스는 서비스를 요청하기 위하여 인터럽트를 발생시킨다. 이 하드웨어 인터럽트는 'HAL_InterruptHandler' 함수에 전달이 되어서 가상 인터럽트 번호로 변환이 되어 드라이버에 전달되고 해당하는 하드웨어 인터럽트는 차단된다. 드라이버는 요청된 서비스를 수행한 뒤 'HAL_InterruptDone' 함수를 호출하여 다시 인터럽트를 개방시키게 된다. 어떤 문제가 발생하거나 필요한 경우에 따라 인터럽트 요청을 서비스할 수 없을 경우에 'HAL_InterruptDisable' 함수를 호출하여 인터럽트의 발생을 중지시킬 수 있다.

hal_time.c

이 파일은 시간과 관련된 시스템 서비스의 하드웨어 관련 부분이 정의 되어있다. 이곳의 함수를 이용하여 J•BOSN 운영체제는 시간적인 동기를 맞출수 있고 수행되고 있는 Thread들의 수행시간을 적절히 제어를 할 수 있다. 또한 수행되는 Thread들의 시간과 관련된 서비스요청을 처리할 수 있게 된다.

이 파일에서 제공되어야 할 기본적인 함수는 다음과 같다.

- PUBLIC VOID HAL_ClockInit (VOID)
- PUBLIC VOID HAL_TimerInterruptHandler (PUINT64 pMsCount)
- PUBLIC VOID HAL_GetRealTime(PSYSTEMTIME pSystemTime)
- PUBLIC VOID HAL_SetRealTime(PSYSTEMTIME pSystemTime)

'HAL_ClockInit' 함수는 하드웨어의 시간과 관련된 부분을 초기화한다. 이 함수는 J•BOSN 이 시작하여 하드웨어 초기화를 모두 마치고 마지막으로 호출이 된다. 시간과 관련하여 RTC(real-time clock)과 J•BOSN 운영체제가 동작을 하기 위한 Tick Timer를 반드시 설정을 하여야 한다. Tick

Timer라는 것은 Thread의 미리 정한 시간(Quantum)과 우선순위를 중심으로 서로간의 수행시간을 적절히 제어 할 수 있도록 시간에 대한 정보를 J•BOSN 운영체제에 전달하는 역할을 담당한다. 또한 시간과 관련된 서비스 요청을 처리할 수 있도록 한다. 하지만 이곳에서 인터럽트에 관련된 부분은 설정하지 않고 단지 RTC를 초기값으로 설정하고(시스템 설계자의 자체 초기값으로 J•BOSN 운영체제는 알지 못한다), 또한 Tick Timer를 시스템 설계자가 미리 정한 값으로 설정하여 미리 정한 시간마다 Tick Timer Interrupt가 발생할 수 있도록 초기화 한다(시간에 관련된 초기화 값은 PORT/PLATFORM/INCLUDE/hal_timer.h 에 정의합니다).

'HAL_TimerInterruptHandler' 함수는 TickTimer 인터럽트가 발생을 하고 하드웨어에 관련된 부분을 재설정할 수 있도록 하기 위해서 제공되는 함수이다. 이 함수에서는 특히 다음 TickTime Interrrupt가 다시 발생할 수 있도록 Tick Timer 하드웨어를 재설정하고 또한 J•BOSN 운영체제가 시간을 관리하는 변수인 pMsCount에서 지정한 변수의 값을 재설정을 하여야 한다. pMsCount는 64bit의 변수포인터로서 시스템이 시작하여 TickTimer가 동작한 후로부터 지금까지의 시간을 milli-second 단위로 기록하는 변수이다. 이 변수는 유일하게 이곳에서 재설정이 되므로 시스템 설계자는 다른 시스템의 동작성을 고려하여 어떠한 경우에서도 정확하게 시간을 설정을 하여야 한다. 만일 시간의 설정에 오류가 발생하면 J•BOSN 운영체제가 정상적으로 동작하지 않을 수 있다. 이 함수 자체적으로 TickTimer에 관련된 인터럽트에 관련된 설정을 할 수 있겠지만 인터럽트는 'hal_Inerrupt.c' 파일에서 설정하는 것을 권장한다. 또한 함수가 실행이 될 때는 모든 인터럽트가 차단되기 때문에 가능한 빠른 수행을 할 수 있도록 작성을 하여야 한다.

'HAL_GetRealTime'과 'HAL_SetRealTime' 함수는 RTC와 관련된 함수로서 현재의 시간을 설정하거나 읽어낼 수 있도록 한다.

hal_main.c

이 파일은 HAL레이어의 메인루틴에 해당한다. J•BOSN 운영체제가 실행되는 개발 시스템의 하드웨어의 초기화 및 설정을 하고, 개발 시스템에서 필요한 J•BOSN 운영체제의 모듈을 결정하고, 수행을 시키도록 나노커널에 명령을 한며, 작성된 드라이버를 시작하는 등, 개발 시스템의 설정을 완성하는 역할을 한다. 결론적으로 이 함수는 개발된 실시간 시스템의 하드웨어와 소프트웨어의 초기화 및 설정을 주관하는 핵심적인 파일이라고 할 수 있다.

이 파일에서 제공되어야 할 기본적인 함수 및 구조체 변수는 다음과 같다.

- PUBLIC VOID HAL_Init (VOID)
- PUBLIC VOID HAL_main (VOID)
- NKERNEL_BOOT HalBoot

'HAL_init' 함수는 타이머와 관련된 하드웨어 설정만을 제외한 모든 하드웨어 초기화(HAL_InterruptInit 함수, 인터럽트 초기화를 포함)와 설정을 담당한다. 아직 J•BOSN 운영체제가 완전히 초기화 되지 않은 상태이므로 J•BOSN 운영체제가 제공하는 서비스는 사용하여서는 안되고 디바이스의 설정과 타이머 설정을 제외한 모든 하드웨어의 초기화 및 설정을 이곳에서 하여야 한다. 인터럽트는 HAL_InterruptInit를 호출하여 반드시 이곳에서 초기화가 되어야 한다.

'HAL_main' 함수는 개발 시스템에서 필요한 J•BOSN 운영체제의 모듈을 선택하여 초기화하며, 필요한 디바이스 드라이버를 설정, 시작하는 부분을 담당하게 된다. 이 함수는 J•BOSN 운영체제의 모듈이 정상적으로 동작할 수 있는 모든 하드웨어 초기화 및 설정이 마무리 된 후 나노커널에서 호출을 한다. 즉, J•BOSN 운영체제에서 처음 실행되는 메인 함수이며 오직 J•BOSN 운영체제의 초기화와 설정만을 담당하게 된다. 나노커널이 수행하는 초기화 순서는 다음과 같

다. 가장 먼저 나노커널에서 필요한 내부적인 초기화를 먼저 수행하여 나노커널을 초기화한다. 이제 J•BOSN 운영체제는 최소한의 사양으로 동작을 시작할 수 있으며 다른 J•BOSN 운영체제의 모듈들이 수행될 수 있는 바탕을 제공해 준다. 이후 곧바로 HAL_ClockInit 함수를 호출하여 TickTimer가 발생할 수 있는 조건을 만들어 주면, TickTimer 인터럽트가 발생하여 시스템은 정상적으로 시간, Thread 관리 등을 시작한다.

다음으로 구조체 변수 ukernelConfig('PORT/ukernel.c'에 정의)에 정의된 정보를 바탕으로 마이크로커널에서 제공되는 세 가지 서버(Time server, synchronization server, system server)를 시작하여 세 가지 서버의 서비스를 제공합니다. 이것으로 마이크로커널까지 수행됐다.

매크로커널에서 제공하는 여러 가지 서버들은 사용자가 개발시스템의 요구조건에 따라서 선택하여 수행할 수 있다. 따라서 나노커널의 초기화에서는 곧바로 'HAL_main' 함수를 호출하여 매크로커널을 수행하는 것을 사용자의 선택사항으로 넘기게 된다. 따라서 사용자는 'HAL_main' 함수에서 반드시 매크로커널에 해당하는 서버를 가장먼저 수행을 시켜주어야 한다. 예를 들면, 디바이스 드라이버를 사용하려면 반드시 J•BOSN 운영체제의 I/O 서브시스템을 관리하는 'I/O 서버'를 시작해 주어야 합니다. 물론 파일시스템 중 Rom파일시스템, Ram파일시스템을 제외한 블록디바이스를 사용하려 하여도 I/O 시스템은 초기화 되어야 한다.

이제 사용하려는 드라이버 등을 등록할 수 있어 시스템에 연결할 수 있다. 드라이버를 I/O 서브시스템에 등록시키려면 '디바이스레지스터' 함수를 사용하여 등록을 할 수 있다. 그러나, J•BOSN 운영체제에서 제공하는 표준 드라이버 모델은 각각 드라이버를 등록시키는 다른 방법을 가지고 있다. 시리얼 통신과 같은 스트림형 디바이스 드라이버는 'Device RegisterWave' 함수를 이용하여 등록을 할 수 있고, 소리와 관련된 드라이버는 'DeviceRegisterWave' 함수를 이용하여 등록을 할 수 있다. 물론 이 두 모델 역시 'DeviceRegister'라는 함수를 사용하여 디바이스 드라이버를 등록하게 된다. 등록된 디바이스 드라이버는 'I/O 서버'를 통하여 J•BOSN의 애플리케이션이나 다른 서버에 의해서 접근을 허용하게 된다. 하지만 'DeviceRegister'라는 함수는 디바이스 드라이버 자신이 초기화를 마친 다음 직접 호출하도록 설계했다. 결과적으로 'HAL_main' 함수에서는 단지 사용하려는 디바이스를 관리해 주는 드라이버 thread를 생성하여 주기만 하면 된다. J•BOSN 운용체제에서는 모든 디바이스 드라이버는 특수한 경우를 제외하고 모두 서버의 형태로 작성이 되므로 가능한 것이다.

디바이스 드라이버를 모두 등록을 한 후에 블록 디바이스 드라이버를 등록하여 파일시스템을 사용하려면, 먼저 블록 디바이스드라이버를 위와 같은 방법으로 'I/O 서버'에 등록을 한 후에 현재의 개발 시스템에서 사용하려는 파일시스템 타입(예: FATFS, ROMFS, RAMFS)에 따라 함수 'UseFatFS', 'UseRomFS' 그리고 'UseRamFS'를 호출하면 된다. 물론 RomFS와 RamFS는 디바디스드라이버를 설정할 필요 없이 사용할 수 있다. 그리고 'FsSetRoot' 함수를 호출하여 파일시스템의 Root 디바이스를 반드시 등록하여 주어야 한다. 다른 블록디바이스를 파일시스템에 연결을 하고 싶다면, 함수 'FsAttachDevice'를 호출하여 기존의 파일시스템에 마운트를 할 수 있다.

J•BOSN 운영체제의 설정을 마치고 나면 이제 애플리케이션을 시작할 수 있는 조건이 완료되어 애플리케이션을 시작하는 모드로 전환이 가능하다. 나노커널은 어플리케이션을 시작하기 위해서 'APP' 디렉토리에서 정의한 애플리케이션을 시작하게 된다.

'HalBoot' 구조체 변수는 위에서 수행한 모든 HAL 레이어에 관련된 기능을 나노커널에게 전달하는 내용을 담고 있습니다. 다음은 'HALBOOT'의 구조체이다. ("INCLUDE/nano_boot.h"에 정의됨)

```
typedef struct _NKERNEL_BOOT
{

UINT32 MaxUserChannel;          : 최대채널수
UINT32 FreeMemoryStart;         : 메모리시작
UINT32 FreeMemoryEnd;           : 메모리 끝
VOID    (*HAL_Init)( VOID );
VOID    (*HAL_ClockInit)( VOID );
VOID    (*HAL_InterruptEnable)(UINT32);
VOID    (*HAL_InterruptDisable)(UINT32);
VOID    (*HAL_InterruptDone)(UINT32);
VOID    (*HAL_InterruptHandler)( VOID );
UINT32 (*HAL_TimerInterruptHandler)( VOLATILE PUINT64 pMsCount );
VOID  (*HAL_SetRealTime)(PSYSTEMTIME pSystemTime);
VOID    (*HAL_GetRealTime)(   PSYSTEMTIME pSystemTime );
UINT32 (*HAL_IoControl)( UINT32 uCmd, PVOID pInBuf,UINT32 nInBufSize, PVOID pOutBuf, UINT32 nOutBufSize );
VOID    (*HAL_PowerOff)( VOID );
VOID    (*HAL_DebugWriteByte)( BYTE Ch );
INT32   (*HAL_DebugReadByte)( VOID );
 VOID   (*HAL_main)(VOID);
INT32   (*main)(VOID);              :애플리케이션시작 주소
}NKERNEL_BOOT, *NKERNEL_PBOOT;
```

idle.c

이 파일은 J • BOSN 운영체제가 주어진 Thread를 수행할 때 더 이상 수행할 Thread 가 없거나 요청된 자원의 부족으로 잠시 동안 서비스할 것이 없을 경우에 이 함수를 호출하게 된다. 'idle thread'의 메인 함수와 같다. 이 파일을 작성할 때는 J • BOSN 운영체제에서 제공되는 어떠한 서비스도 요청을 하여서는 안되고, 오직 하드웨어에 관련된 일만을 추천한다.

이 파일에서 제공되어야 할 기본적인 함수와 구조체 변수는 다음과 같다.

- PUBLIC VOID Idle_main (VOID)
- THREADCONFIG IdleConfig
- PUBLIC VOID HAL_PowerOff (VOID)

'Idle_main' 함수는 J • BOSN 운영체제가 서비스를 요청받지 못하였거나 요청 받은 서비스의 자원이 아직 준비가 되지 않았을 때 호출된다. 이 함수는 무한 루프를 가지고 있으며, 이 루프에서 빠져 나오지 않도록 하여야 한다. 이 함수가 호출될 때는 시스템에게 주어진 일이 없고 대기 하는 시간이므로 시스템 설계자는 시스템이 대기 모드로 전환이 되도록 하드웨어 관리를 해주면 시스템의 전력이나 기타 문제를 쉽게 해결할 수 있게 된다.

매우 빈번히 이 함수가 호출될 수 있기 때문에 시스템을 대기모드로 설정을 하였다면, 복귀하는데 걸리는 시간을 충분히 계산하여야 원하는 시스템을 설계할 수 있을 것이다. 특히 이곳은 하드웨어의 인터럽트에 의해서만이 J • BOSN 운영체제가 제어권을 회수할 수 있으므로 인터럽트는 최소한 하나는 개방되어야 한다. 여기서는 시간과 관련된 자원을 관리하기 위하여 TickTimer를 권장하지만, 다른 것도 물론 사용될 수 있다.

구조체 변수 'IdleConfig'는 Idle_main 함수를 메인으로 idle thread를 생성할 수 있는 설정을 담고 있다. 이 변수는 'hal_locore.S'에서 나노커널로 전달이 되어야 한다. ('hal_locore.S' 참조)

'HAL_PowerOff' 함수는 전원관리에서 'SystemPowerOff' 함수가 콜되었을 때, 최종적으로 하드웨어를 수면상태로 보내기 위해서 사용된다. 이 함수는 드라이버의 전원관리에는 사용되지 않고, 단순히 CPU의 전원관리에만 사용하는 것을 권장한다.

console.c

이 파일은 시스템의 console 역할을 담당하는 기능의 하드웨어 관련 부분이 정의 되어 있습니다. 이곳의 함수를 이용하여 J•BOSN 운영체제의 panic에 관련된 부분과 디버깅 메시지가 출력된다. 또한 구현되어야 할 함수들의 입출력은 반드시 J•BOSN 운영체제의 I/O 시스템을 이용하지 않아야 하고, 특히 인터럽트에 관련한 기능을 사용하지 않고 구현이 되어야 한다. 필수적으로 기본적인 함수는 다음과 같다.

- PUBLIC VOID ConsoleInit (VOID)
- PUBLIC INT32 HAL_DebugReadByte(VOID)
- PUBLIC VOID HAL_DebugWriteByte(BYTE Ch)

'ConsoleInit' 에는 현재 Console용으로 사용하려는 하드웨어의 초기화를 담당한다. 이 함수는 J•BOSN이 시작하여 패닉 시스템을 초기화 할 때 호출이 됩니다.

'HAL_DebugReadByte' 에는 변수가 주어지지 않는다. 읽는 중간에 하드웨어적인 에러가 발생하거나, 데이터가 존재하지 않으면, 에러를 리턴하고, 데이터가 존재하면 해당하는 데이터를 리턴하면 된다.

'DebugWriteByte' 는 한 개의 변수가 주어지는데, 'Ch' 은 출력을 하려는 문자를 나타낸다. 이 문자는 출력 중 하드웨어의 에러가 발생할 경우에도 긴급히 에러를 수정하고 반드시 문자을 출력해야 한다. 또한 LF(line feed) 가 포함되어 있을 경우에는 CR(carrage return)을 이어서 출력을 시켜 주어야 합니다. 리턴되는 값은 없다.

JBOSN운영체제의 모듈 탑재

JBOSN 운영체제는 여러 가지의 서버모듈을 가지고 있다. 사용자는 필요한 모듈을 선택하여 개발시스템에 탑재를 할 수 있다. 'WORKSPACE/[MODEL]/config/ukernel_main.c' 는 구조체변수 배열인 ukernelConfig[]를 정의하여 필요한 마이크로커널에 해당하는 서버를 시작하도록 나노커널에 설정데이터를 전달할 수 있다. 이 변수는 'PLATFORM/[BSP]/PORT/HAL/hal_locore.S' 를 통하여 나노커널에 전달이 됩니다. 'WORKSPACE/[MODEL]/config/mkernel_main.c' 는 함수 'MacroKernelStart' 를 정의하여 매크로커널에 해당하는 서버를 선택하여 개발 시스템에 탑재를 할 수 있다. 이 함수는 'HAL_main' 함수에서 호출된다.

JBOSN 운영체제의 드라이버 탑재

'PLATFORM/[BSP]/PORT/DRIVER' 디렉토리에 개발 시스템에서 필요한 드라이버를 작성하여 탑재한다. 이 드라이버 thread는 'HAL_main' 함수에서 생성되도록 호출된다.

이제 시스템 설계자는 J•BOSN 운영체제의 하드웨어에 관련된 모든 기능을 설계하는 작업을 마쳤다. 이제 실제 사용자가 원하는 시스템을 구현하는 J•BOSN 운영체제의 애플리케이션을 작성해 보자.

JBOSN운영체제의 PORT디렉토리 링크옵션

'PORT' 디렉토리는 하드웨어와 JBOSN 운영체제의 모든 연결을 확립하는 곳이다. 또한, 개발 시스템의 설정을 결정하는 곳이다. 특히 프로그램과 운영체제의 메모리상 배치와 사용하고 싶은 변수, 버퍼 등을 설정하는 곳이다. 아래의 ld 스크립트(표 1)는 하나의 예를 보여준다.

전체 운영체제의 이미지를 개발 시스템의 메모리 사용의 용도에 따라 재배치를 하는 부분이며, 운영체제를 시작시켜주는 부분이다. 경우에 따라서는 사용하려는 메모리의 초기화를 설정하여 주기도 한다. 운영체제의 메모리 사용에 관한 규칙은 링커의 설정에 해당하는 링크 스크립트의 설정에 맞게 제작되어야 한다. JBOSN 운영체제는 'PORT/platform.lds' 라는 링크 스크립트를 가지고 있다. 다음은 그 예를 살펴보자.

```
/*
 * ld script to make J-BOSN Operating System memory layout.
 * refer the relcation part of "locore.S"
 * Written by iBOSN systems
 */

/* our program's entry point; not useful for much except to make sure the S7 record is proper,
 *  because the reset vector actually defines the entrypoint in most embedded systems
 */

/* a list of files to link (others are supplied on the command line) */

OUTPUT_ARCH(arm)
ENTRY(stext)

/* list of our memory sections */

/* NKERNEL       : 8KB    */
/* UKERNEL_TIME   : 4KB    */
/* UKERNEL_SYSTEM : 4KB    */
/* UKERNEL_SYNC   : 8KB    */
/* MKERNEL_ID     : 4KB    */
/* MKERNEL_FS     : 8KB    */
/* MKERNEL_WN     : 24KB   */
/* MKERNEL_NW     : 24KB   */

MEMORY
{
  Kernel    : o = 0x20000000 ,  l =   16K
  MMU_buf:o= 0x20000000 + 16K,                    l =   16K
  DMA_buf :  o = 0x20000000 + 16K + 16K,          l =   612K
  DMA_NET:  o = 0x20000000 + 16K + 16K + 612K,    l =   268K

  display_f   :  o = 0x20000000 + 16K + 16K + 612K + 268K,    l =   228K
   display_i   :  o = 0x20000000 + 16K + 16K + 612K + 268K + 228K,    l = 228K
    ram          :  o = 0x20000000 + 16K + 16K + 612K + 268K + 228K + 228K,  l = 32M - (4K + 16K + 612K + 268K + 228K + 228K)
}
```

```
/* section description */
SECTIONS
{
  .init :
  {
    _startof_platform = .;

        _text = .;          /* Text and read-only data                  */
    hal_locore.o(.text.init)
    _end_init = .;

  } > ram

  /* TEXT */
.text :
{
   . = ALIGN(4);
*(.text)
        *(.text.*)
        *(.fixup)
        *(.gnu.warning)
        *(.text.lock)       /* out-of-line lock text */
      *(.glue_7)
      *(.glue_7t)
        *(.rodata)
        *(.rodata.*)
. = ALIGN(4);
    _etext = ALIGN(4);      /* End of text section*/
} > ram

  /* DATA */
.data : AT(ADDR(.text) + (SIZEOF(.text) + 3)&~(3) )
{
/*
    * and the usual data section
    */
    . = ALIGN(4);
    _sdata = .;
        *(.data)
        . = ALIGN(4);
        _edata = .;
} > ram

/* BSS */
.bss : AT(ADDR(.text) + (SIZEOF(.text) + 3)&~(3) + (SIZEOF(.data)+3)&~(3)  )
{
__bss_start = .;
    . = ALIGN(4);
*(.bss)
*(COMMON)
    _end = ALIGN(4);
} > ram
  _startof_ram = .;
  _endofram = 0x20000000 + 32M;
}
```

표 1. 링크스트립트 예

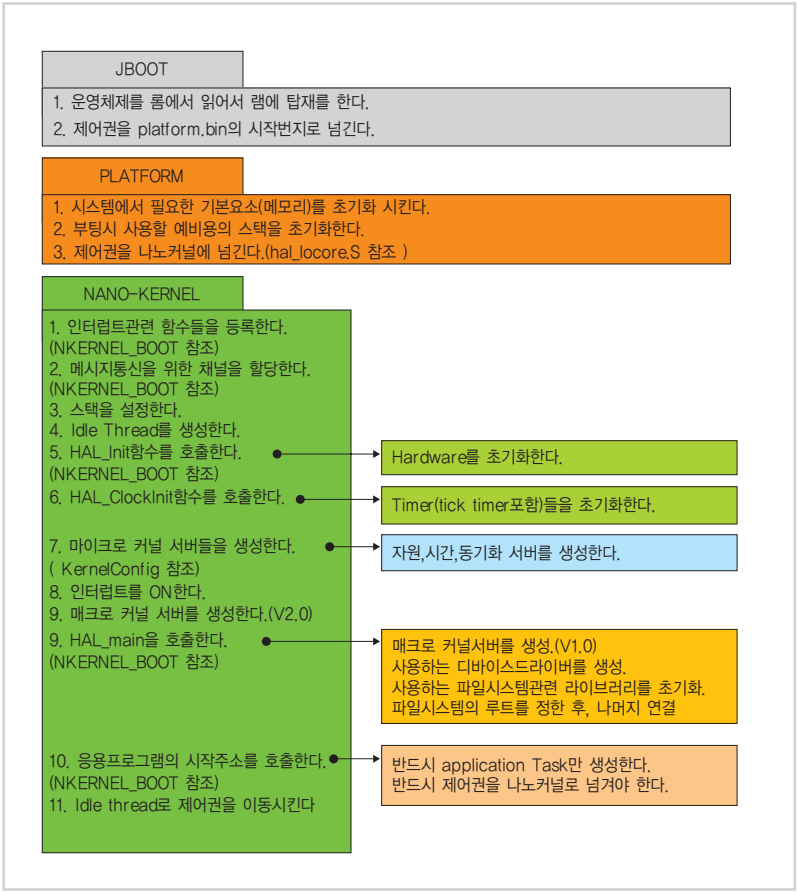

그림 3. J • BOSN 부팅 순서도

메모리 섹션에서 메모리의 사용을 설정을 하여 버퍼를 지정하고 롬과 램영역을 설정하며, 섹션에서 생성된 코드의 배치를 설정한다. 커널 라이브러리에서 사용하는 다음의 변수를 반드시 설정을 하여 주어야 한다. _startofram, _endofram 등은 커널에서 메모리를 설정할 때 사용하고 있는 변수로서 반드시 설정을 하여 주어야 하며, 그 외의 설정은 사용자의 요구에 맞게 설정한다. 예제에서는 _etext, _sdata, _edata, _end 등이 'hal_locore.S' 에서 사용되고 있으므로 알맞게 설정이 되어야 한다. 이 링크스크립트에 의해 생성된 이미지는 롬영역이 주소 0x40000000에서 시작하여 크기는 1MBYTE로 설정이 되고 램영역은 주소 0x20000000에서 시작하여 크기는 32MBYTE로 설정이 된다. 또한 이미지가 수행이 될 때, 실행코드는 롬에 적재가 되고 데이터는 램에 적재가 되어야 한다. 따라서 램에 적재될 데이터를 롬에서 램영역으로 복사가 이루어져야 한다. 이 역할을 'hal_locore..S' 에서 이 동작을 수행한다.

J • BOSN 시스템 부팅 순서

J • BOSN 운영체제는 '그림 3. JBOSN 부팅순서도' 와 같은 순서로 부팅하게 된다. 시스템 작성자와 응용프로그램 작성자는 부팅순서도를 참조하여 올바른 플랫폼을 작성해야 한다.

응용프로그램 제작

소개

이 장에서는 J • BOSN 운영체제가 완벽히 이식된 개발시스템에 애플리케이션을 탑재하는 방법에 대하여 살펴본다. 여기에서 설명한 모든 것은 'APP' 디렉토리를 작성, 완성하는 것이다.

'APP' 디렉토리 밑에 있는 파일은 J • BOSN 운영체제의 애플리케이션이 시작되는 곳이다.

다음은 'APP' 디렉토리에 있는 필수파일들이다.

- app_locoare.S : 애플리케이션영역을 초기화합니다.(option)
- main.c : 필요한 애플리케이션을 생성합니다.

'main.c' 에는 최소한 'main' 이라는 함수 하나만 존재하면 됩니다. 'main' 함수는 J • BOSN 운영체제의 초기화가 모두

끝나고 개발 시스템의 동작이 시작된 후에 애플리케이션을 수행하기 위하여 애플리케이션을 생성을 목적으로 나노커널이 호출하는 함수이다. 이 함수는 애플리케이션은 아니고 단지 원하는 애플리케이션을 생성시키는 하나의 함수에 불과하다. 따라서 'TaskCreate' 서비스를 이용하여 애플리케이션을 생성만하고 다른 서비스를 이용하지 말고 바로 수행을 마쳐야 한다. 만일 그렇지 않다면 J • BOSN 운영체제가 정상적으로 동작하지 않을 수 있다. 이 함수에서 생성된 Task는 최초의 애플리케이션 Task가 되어 곧바로 수행을 시작한다. 이후의 모든 동작을 이곳에서 생성된 Task에서만 수행된다. 애플리케이션 개발자는 이 초기 애플리케이션 Task에서 시작하여 'TaskCreate'나 'ThreadCreate'를 사용하여 모든 애플리케이션을 작성하여 원하는 작업을 수행해야 한다. 참고로, 최초로 생성된 Task도 하나의 Thread로서 동작하기 때문에 아무런 일을 하지 않고 수행을 마치면 곧바로 Thread는 종료되어 J • BOSN 운영체제에는 아무런 애플리케이션이 없게 되고 또한 생성될 수도 없게 된다. Embedded World

〈연재끝〉

```
//=========================================

#define APPMAINSTACK_SIZE (4096)
PUBLIC UINT32    AppMainStack[APPMAINSTACK_SIZE>>2];

STATIC INT32     AppMain( UINT32 Arg );
PUBLIC TASKCONFIG  AppMainTaskConfig =
                {
                   NULL,
                   AppMain,
                   (PBYTE)AppMainStack,
                   APPMAINSTACK_SIZE,
                   THREAD_PRIORITY_USER_HIGH + 2
                };
```

```
//=========================================

INT32
main( VOID )
{
  HANDLE hTask;
  // start the applications...
  kprintf("JBOSN Start+\n");

  hTask = TaskCreate( &AppMainTaskConfig );
  kprintf ("JBOSN Start-\n");

  return hTask;
}

STATIC INT32
AppMain( UINT32 Arg )
{
  kprintf( "Start AppMain\n");

  for( ; ; )
    kprintf("HELLO WORLD\n");

  return 0;
}
```