상용 운영체제 기반 이중화 시스템 설계

정회원 김 종 호*, 이 제 헌***, 임 형 택**, 방 경 은**,
이 숙 진***, 임 순 용***, 양 승 민**

Design of Duplicate System based on Commercial OS

J. H. Kim*, J. H. Rhee***, H. T. Lim**, K. E. Bang**,
S. J. Lee***, S. Y. Lim***, S. M. Yang** ***Regular Members***

요 약

중요한 일을 처리하는 제어 시스템에서 고장이 발생하여 제어가 중단될 경우 큰 경제적 손실이 발생한다. 이런 경우에 대비하여 고 신뢰도 보장을 위한 시스템의 이중화가 반드시 필요하다. 기존 이중화에서는 이중화 기능을 갖는 전용 운영체제를 직접 만들었는데, 이는 많은 개발 비용과 유지보수 비용을 필요로 한다. 이중화 기능을 사용자 수준에서 지원하고 상용 운영체제를 이용하면 운영체제와 개발환경을 만드는데 필요한 비용과 시간을 줄일 수 있다.

이에 본 논문에서는 상용 운영체제를 이용한 시스템의 이중화 방법을 제시한다. 본 논문에서 설명하는 시스템은 제어국 시스템으로서 동시쓰기 메모리를 이용하는 이중화 구조이며, VxWorks를 운영체제로 사용한다. 이중화를 지원하는 태스크들이 선점되지 않고 즉시 수행될 수 있도록 하는 방법, 스탠바이가 부팅된 후 액티브와 메모리를 동기화하는 방법, 사용자가 동시쓰기 메모리를 쉽게 사용할 수 있도록 VxWorks의 파티션을 이용하는 방법을 제시한다. 그리고, 액티브의 하드웨어와 소프트웨어가 하드웨어 고장을 감지했을 때의 절체 방법과 스탠바이가 액티브의 고장을 감지했을 때의 절체 방법을 설명한다.

ABSTRACT

If the control system that works important job fails, economical loss occurred. Hence, to guarantee high reliability, it must be duplicated. In the case of traditional duplication mechanism, dedicated operating system with duplication functions were built. This required much development and maintenance cost. They can be saved, if we use commercial operating system and its development environment.

This paper proposes a duplication mechanism for the system based on commercial OS. The system that explained in this paper is BSC(Base Station Controller). The duplicated BSC system uses concurrent write memory for synchronization and VxWorks as an operating system. We propose how the task supporting duplication functions is executed without delay and preemption, how to synchronize standby' s memory with active' s, and how to use concurrent-write memory easily with VxWorks' s partition. We also describe the takeover procedure when the active detects its hardware fault and when the standby recognizes the failure of the active.

* (주) 팬택 기술연구소 (noopykim@hanmail.net)
** 숭실대학교 컴퓨터학과
*** 한국전자통신연구원
논문번호 : 00055-0214, 접수일자 : 2000년 2월 14일
※ 이 연구는 1999년도 한국전자통신연구원 위탁연구과제로 이루어졌음.

I. 서 론

제어시스템에 고장이 발생할 경우 경제적 손실은 물론 인명피해까지 초래할 수 있기 때문에 신뢰도의 보장은 필수적이다.

제어국과 같은 시스템은 같은 시스템을 두 개 준비하여 하나가 고장이 났을 때 다른 하나가 그 일을 계속 하도록 하는 이중화(Duplication) 방법으로 고 신뢰도를 보장한다. 기존 이중화 시스템들에는 직접 개발한 전용 하드웨어와 운영체제가 사용되었으며 운영체제가 이중화 기능을 지원하였다[5][6][7]. 이 경우 하드웨어와 운영체제뿐만 아니라 개발환경까지도 개발하여야 하므로 많은 인력과 시간이 소비된다. 또한, 결함허용 기법이나 시스템 설계가 수정되면 하드웨어나 소프트웨어를 일일이 다시 개발해야 하므로 유지보수가 어렵다. 특히, 이중화 기능을 운영체제가 제공하는 경우 시스템 구성이나 환경에 따라 결함허용 방법을 수정하기 어렵다.

근래에는 상용(COTS 또는 Commercial Off-The-Shelf) 하드웨어와 소프트웨어를 이용하여 개발 및 유지보수 비용을 줄이면서도 고 신뢰도를 보장하는 방법들이 연구되고 있다. NASA Jet Propulsion Laboratory의 REE(Remote Exploration and Experimentation) 과제[4]의 경우 상용 하드웨어와 소프트웨어를 이용하여 우주 비행선에 적합한 시스템을 설계하고 개발하는 것을 목표로 하고 있다. Chameleon[4]은 이기종 시스템들로 구성된 분산 환경에서 상용 하드웨어와 소프트웨어를 이용하여 고 신뢰도를 보장하기 위하여 제안되었다. Chameleon은 다양한 결함허용 환경에 맞게 커스터마이즈가 용이하고 위치 투명성을 제공한다. TFT(Transparent Fault Tolerance)[3]는 hot standby spare 결함허용 기능을 미들웨어 수준에서 제공한다. 이중화 기능이 미들웨어에서 지원됨으로써 운영체제의 결함을 허용할 수 있고, 좋은 상용 운영체제와 상용 개발 환경을 사용할 수 있다. 고 신뢰도를 필요로 하는 시스템들에서도 더욱 시기 적절한 판매(time-to-market)가 중요해지고 제품의 생명주기가 짧아질 것이므로 개발 기간을 최소화할 수 있도록 상용 하드웨어와 소프트웨어를 이용하는 방법에 대한 연구가 반드시 필요하다.

이에 저자는 상용 운영체제 기반의 이중화 방법을 연구하였다[1][2]. 그림 1과 같이 운영체제에 내장되어 있던 이중화 지원 기능은 이중화 지원 태스크와 이중화 지원 인터럽트 서비스 루틴에 의해 수행된다. 이 방법은 안정적인 상용 운영체제와 좋은 개발환경을 이용하므로 개발 기간과 개발 비용을 감소시킬 수 있고, 시스템 구성이나 환경이 변했을 때 쉽게 결함허용 기법을 수정할 수 있다. 본 논문의 이중화 방법은 액티브와 스탠바이 간의 빠른 동기와 절체를 위하여 동시쓰기 메모리를 이용한다. 상용 운영체제로는 VxWorks를 사용하였다.

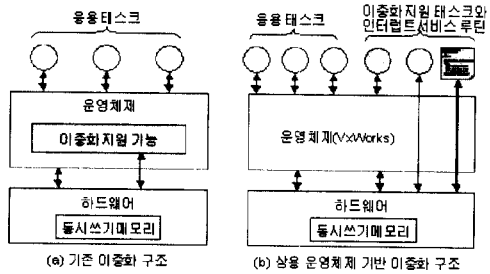


그림 1. 이중화 구조의 비교

상용 운영체제 기반 이중화 방법에서는 다음과 같은 두 가지 문제점이 해결되어야 한다. 첫째, 하드웨어가 고장났을 때 높은 우선순위를 갖는 태스크나 인터럽트에 의해 절체 작업이 바로 수행되지 않거나 수행 중에 중단이 되면 시스템은 정상적으로 동작할 수 없다. 둘째, 동시쓰기 메모리 기반 이중화 구조에서 액티브의 메모리를 스탠바이에 복사하기 위하여 액티브의 동시쓰기 메모리 전체를 읽었다 쓴다. 이를 위하여 이중화 지원 태스크는 다른 태스크나 커널이 사용하고 있는 메모리 영역을 접근할 수 있어야 한다.

본 논문에서는 VxWorks를 이용하여 절체 작업이 필요할 때 즉시 수행되고 다른 작업에 의하여 선점되지 않는 방법과 액티브의 이중화 지원 태스크가 동시쓰기 메모리에 직접 접근하여 스탠바이와 액티브를 동기화하는 방법을 제시한다. 액티브와 스탠바이 간에 동기화되어야 하는 프로그램과 태스크는 동시쓰기 메모리에 존재해야 한다. 이를 위하여 사용자가 손쉽게 동시쓰기 메모리에 프로그램을 적재하고 태스크를 생성하기 위한 방법을 제시한다. 그리고, 이 방법들을 이용하여 액티브에 의한 절체 방법과 스탠바이에 의한 절체 방법을 설명한다.

기존 이중화 기법에 대하여 2장에서 설명하고, 3장에서는 본 논문에서 사용하고 있는 이중화 시스템구조에 대해 설명한다. 4장에서는 VxWorks상에서 이중화를 지원하기 위한 방법을 제시하고, 5장에

서는 동시쓰기 메모리의 사용 방법에 대하여 설명한다. 액티브에 의한 절체 과정과 스탠바이에 의한 절체 과정을 6장에서 설명한 후 7장에서 결론 및 향후 연구 과제를 제시하고 논문을 마친다.

Ⅱ. 이중화 기법

이중화 방식은 액티브와 스탠바이 간의 동기화 방법에 따라 동기식 이중화 방식과 비동기식 이중화 방식으로 구분된다. 동기식 이중화 방식에서 액티브와 스탠바이는 동일한 명령어를 동일한 순서로 실행한다. 비동기식 이중화 방식에서는 액티브만 일을 하고 스탠바이는 액티브의 상태를 받아서 액티브와 일치시킨다.

동기식 이중화 방식은 액티브가 고장났을 때 신속히 복구할 수 있으며 동기화 수준에 따라 마이크로 수준, 명령어 수준, 프로세스 인스턴스 수준으로 구분된다. 마이크로 수준과 명령어 수준 방식은 마이크로 명령어(micro-instruction)나 명령어마다 동기화를 하므로 오버헤드가 크다. 프로세스 인스턴스 수준 방식은 하드웨어 성능 저하는 적은 대신 소프트웨어 부하로 인한 성능 저하가 발생한다.

비동기식 이중화 방식에서는 액티브와 스탠바이의 상태를 동일하게 유지하는 것이 매우 중요하다. 비동기식 이중화 방식은 동기화를 위하여 액티브가 자신의 변경된 상태를 스탠바이에게 전달하는 방식에 따라 메시지전달 방식과 동시쓰기 방식으로 구분된다. 메시지전달 방식을 사용하는 시스템으로는 TDX-1[7]이 있는데 전달할 정보량이 많아질 경우 소프트웨어 오버헤드가 매우 크다. 동시쓰기 방식을 사용하는 시스템으로는 TDX-10[7], No.5 ESS[6]가 있는데, 액티브의 처리결과가 하드웨어에 의하여 액티브와 스탠바이의 메모리에 동시에 쓰여지므로 동기화에 의한 오버헤드가 거의 없고, 절체 시에 레지스터만 복사하면 되므로 절체 시간이 매우 짧다.

동시쓰기(concurrent write) 기반의 이중화 구조는 액티브와 스탠바이, 그리고 동시쓰기 메모리로 구성된다. 액티브는 작업을 수행하는 노드이고, 스탠바이는 작업을 수행하지는 않지만 액티브가 고장나면 즉시 작업을 계속 수행할 수 있도록 대기하고 있는 노드이다. 그림 2와 같이 동시쓰기 메모리는 액티브가 자신의 메모리에 쓰면 스탠바이의 메모리에도 동시에 동일한 주소에 쓰이는 것이다. 동시쓰기는 한 방향으로만 이루어지며 스탠바이는 자신의 동시쓰기 메모리에 쓰기 작업을 할 수 없다. 동시쓰기 메모리를 사용하는 경우 동기화에 의한 오버헤드가 매우 적고 액티브와 스탠바이가 항상 동일한 상태로 유지되므로 절체 시간이 매우 짧다. 5ESS나 TDX 계열의 교환기에서 동시쓰기 메모리가 사용되고 있다.

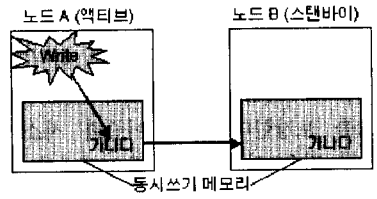


그림 2. 동시쓰기 메모리

Ⅲ. 이중화 시스템의 구조 및 특징

본 논문의 이중화 시스템은 그림 3과 같은 구조를 갖는다. 하드웨어로는 동시쓰기 메모리와 마이크로 프로세서, 그리고 하드웨어 고장을 마이크로 프로세서에게 알려주는 인터럽트 프로세서를 사용한다. 운영체제로는 내장 실시간 시스템에 적합한 VxWorks를 사용한다. 이 운영체제의 태스크는 커널과 함께 수퍼바이저 모드에서 동작한다. 즉, 범용 운영체제와는 달리 커널 모드와 사용자 모드의 구분이 없다.

시스템 태스크는 VxWorks 운영체제에 포함된 태스크로써 운영체제를 초기화하는 태스크, 로깅하는 태스크, 예외를 처리하는 태스크, 네트워크에 관련된 일을 하는 태스크, 쉘 태스크 등이 있다. 본 논문에서는 시스템 태스크가 아닌 모든 태스크들을 사용자 태스크라 정의하며, 사용자 태스크는 응용 태스크와 이중화 지원 태스크로 구분한다. VxWorks 스케줄러는 시스템 태스크와 사용자 태스크를 오로지 우선순위에 의해서만 스케줄링한다. 즉, 시스템 태스크라 할지라도 사용자 태스크보다 우선순위가 낮다면 사용자 태스크에 의해 선점될 수 있다. 태스크의 우선순위는 0 부터 255 까지의 레벨이 있는데 레벨 0이 가장 높고, 레벨 255가 가장 낮은 우선순위이다. 라운드로빈 스케줄링을 선택하면 우선순위가 같은 태스크들을 타임 퀀텀만큼 돌아가면서 수행시킨다. 우선순위가 0인 태스크는 인터럽트가 발생했을 때에만 선점될 수 있다. VxWorks는 사용자가 작성한 코드를 인터럽트에 연결할 수 있는데 이를 사용자정의 인터럽트 서비스 루틴(ISR또는 Interrupt Service Routine)이라 한다. 그림 3처럼 이

중화 지원 기능은 태스크와 사용자정의 ISR에 의해 수행된다.

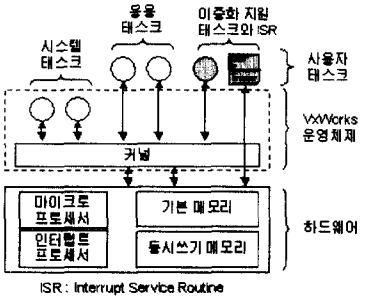


그림 3. 이중화 시스템의 구조

동시쓰기 메모리의 크기와 동시쓰기 메모리에 쓸 내용에 따라서 메모리의 구성은 매우 다양하다. 예를 들면, 메모리 전체를 동시쓰기 메모리로 구성하거나 일부만 동시쓰기 메모리로 구성할 수도 있다. 동시쓰기 메모리에 어떤 것들이 기록되게 할 것인지에 따라서도 다양하게 구성할 수 있다. 예를 들면, 동시쓰기 메모리에 응용 프로그램과 응용 태스크만 존재하게 하는 구성 방법, 동시쓰기 메모리에 운영체제의 일부나 전체까지 두는 구성 방법 등이 있을 수 있다. 따라서, 동시쓰기 기반의 이중화도 동시쓰기 메모리 구성에 따라 이중화 구현 방법이 달라지게 된다. 본 논문에서는 다양한 메모리 구성 방법 중 기본 메모리와 동시쓰기 메모리를 사용하는 메모리 구성을 사용한다. 기본 메모리에는 운영체제와 이중화 관련 프로그램이 적재되며 동시쓰기 메모리에는 응용 프로그램의 코드와 응용 태스크의 TCB(Task Control Block)와 스택이 저장된다.

Ⅳ. VxWorks 상에서의 이중화 지원방법

운영체제에 포함되던 이중화 지원 기능이 태스크에 의해 수행될 때 다른 인터럽트나 태스크에 의해 선점되지 않고 즉시 수행될 수 있도록 높은 우선순위로 수행되는 방법을 제시한다. 절체를 담당하는 태스크가 절체를 하는 중에 다른 태스크에게 선점되면 안되기 때문이다. 그리고, 스탠바이가 초기화될 때 이중화 지원 태스크가 액티브의 메모리와 스탠바이의 메모리를 동기화시키기 위한 방법도 제시한다. 이를 위하여 다른 응용 태스크나 시스템 태스크가 사용하는 메모리를 이중화 지원 태스크가 접근할 수 있어야 한다.

1. 이중화 지원 태스크의 높은 우선순위

VxWorks는 사용자가 작성한 프로그램을 특정 인터럽트에 연결할 수 있도록 intConnect()라는 함수를 제공한다. intConnect()를 이용하여 절체 ISR을 하드웨어 고장 인터럽트에 연결한다. 인터럽트 프로세서가 하드웨어 고장을 감지하면 바로 인터럽트를 발생시키고, 미리 등록한 절체 ISR이 수행되면서 절체가 시작된다. 그런데, VxWorks의 ISR은 호출자를 블록시킬 가능성이 있는 루틴을 호출하면 안된다는 제약사항 때문에 malloc()과 같은 메모리를 할당하는 루틴이나 메시지를 전송하는 루틴을 호출할 수 없다. 액티브가 스탠바이에게 메시지 전송을 통해서 절체를 알리는 경우 절체 ISR은 이를 수행할 수 없다. 절체 ISR은 이중화 지원 태스크의 하나인 절체 태스크를 실행시킨다. 절체 태스크가 실행 중일 때 다른 태스크나 인터럽트에 의하여 실행이 중지되거나 지연되면 안된다. VxWorks는 태스크가 인터럽트에 의해 중지되지 않게 하기 위하여 intLock()과 intUnlock()을 제공하고 태스크가 우선순위가 높은 다른 태스크에 의해 선점되지 않게 하기 위하여 taskLock()과 taskUnlock()을 제공한다. 따라서, 절체 태스크는 먼저 intLock()과 taskLock()을 호출하여 자신이 어떤 경우에도 선점되지 않게 한 후 절체 작업을 수행한다. 그런데, 태스크를 새로 생성하는 작업도 malloc()을 이용하므로 절체 ISR은 태스크를 생성하는 taskSpawn()이나 taskInit()을 호출할 수 없다(taskSpawn()은 태스크를 생성한 후 바로 실행시키는 함수이며, taskInit()은 태스크를 생성하기만 하고 실행시키지는 않는 함수이다.). 따라서, 시스템 초기화 시에 미리 절체 태스크를 생성한 후 블록시킨다(이는 VxWorks의 taskInit()을 이용한다). 그런 다음, 절체 ISR은 taskActivate()를 이용하여 블록되어 있는 절체 태스크를 활성화시킨다.

ISR의 제약사항에 해당하지 않는 방법으로 액티브가 스탠바이에게 절체를 알리는 경우에는 절체 ISR이 직접 절체를 요청한다. 즉, 절체 태스크를 활성화시키고 태스크가 선점되지 않도록 intLock()과 taskLock()을 호출할 필요가 없다.

2. 동시쓰기 메모리의 동기화

동시쓰기 메모리를 사용하는 이중화 구조에서 스탠바이가 부팅되어 정상적으로 스탠바이의 역할을 하려면 자신의 동시쓰기 메모리를 액티브의 동시쓰기 메모리와 동기화시켜야 한다. 이를 위하여 액티

브의 이중화 지원 태스크는 자신의 모든 동시쓰기 메모리를 읽었다 쓴다. 동시쓰기 메모리이기 때문에 액티브의 메모리가 스탠바이의 메모리로 복사되어 메모리 동기화가 된다. 이를 위하여 동기화를 수행하는 memSync 태스크는 다른 태스크나 커널이 사용하고 있는 메모리 영역을 접근할 수 있어야 한다. 그런데, VxWorks의 태스크는 모두 하나의 주소공간에서 수행되고 수퍼바이저 모드에서 수행되므로 memSync 태스크는 커널이나 다른 태스크가 사용중인 메모리 영역에도 접근할 수 있다.

memSync는 액티브에 있는 이중화 지원 태스크의 하나로써 메모리를 동기화하는 태스크이다. 이 태스크가 메모리의 한 바이트를 읽고 쓰는 사이에 선점되면 안된다. 예를 들면, 이 태스크가 1000 번지에 저장된 값('a'라고 가정)을 읽은 다음 선점되었다. 이 때 호처리 태스크가 1000 번지에 'b'를 기록한 후 다시 memSync 태스크가 실행되면 memSync 태스크는 1000 번지에 'a'를 기록하게 되므로 호 처리 태스크가 최근에 갱신한 값이 사라지게 된다. 따라서, memSync 태스크는 먼저 intLock()과 taskLock()을 호출하여 선점되지 않게 한 후에 메모리를 읽고 쓴다.

memSync 태스크가 한꺼번에 모든 메모리를 다 읽고 쓰게 되면, 그동안 다른 태스크들이 수행되지 못하게 되어 호 처리를 할 수 없게 된다. 따라서, 이 태스크는 한 번 수행될 때 일정한 크기만큼만 읽고 쓰기를 한 후 intUnlock()과 taskUnlock()을 호출하여 인터럽트 처리나 다른 태스크들이 수행될 수 있게 한다. 이 태스크가 한 번에 읽고 쓸 메모리의 크기는 시스템의 성능, 시스템의 부하, 메모리의 속도 등을 고려하여 결정하여야 한다.

스탠바이의 메모리가 액티브와 동기화되어야 스탠바이가 자신의 역할을 할 수 있으므로 액티브의 호 처리에 방해되지 않는 한도에서 가능한 빨리 메모리 동기화가 될 수 있어야 하며 메모리 동기화가 완료되는 시간을 예측할 수 있어야 한다. 이를 위하여 memSync 태스크는 높은 우선순위를 가지며 타이머에 의해 주기적으로 활성화된다. 이 태스크가 타이머에 의해 수행되는 주기와 한 번 실행될 때 동기화하는 메모리의 크기가 정해지면 전체 메모리를 동기화하는데 걸리는 시간을 알 수 있다.

그림 4는 memSync 태스크가 메모리를 동기화시키는 과정을 도식화 한 것이다. 시스템은 전원이 들어오면 부팅과정을 거친 후 BEMS (Basestation Element Management Subsystem) 접속을 통해 시스템에 필요한 자료들을 가져온다. 그런 다음 스탠바이는 액티브의 memSync 태스크에게 메모리를 동기화시켜줄 것을 요청한다. memSync 태스크가 동시쓰기 메모리를 모두 읽었다 쓴 후에 동기화가 완료되었음을 스탠바이에게 알려주면 스탠바이는 정상적으로 스탠바이의 역할을 시작한다.

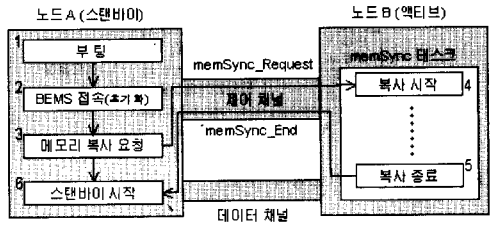


그림 4. 메모리 동기화

동시쓰기 메모리 전체에 대해서 동기화가 이루어지기 전에 이미 동기화된 부분에 액티브가 자료를 쓰면 역시 스탠바이의 동일 영역에 써지므로 액티브와 스탠바이의 동시쓰기 메모리는 일치된 상태가 유지된다. 따라서, 메모리 동기화 작업은 다른 작업들과 병행하여 수행하여도 문제가 없다.

스탠바이가 액티브에게 메모리 동기화를 요청하면 액티브에서는 1) taskDelay()를 이용하는 방법, 2) nanosleep()을 이용하는 방법, 3) watchdog 타이머를 이용하는 방법 중 한가지 방법으로 메모리를 스탠바이에게 복사해 준다. taskDelay()는 원하는 클럭 틱만큼, nanosleep()은 나노초만큼 태스크를 쉬게 한다. 따라서, 1 MB의 메모리를 동기화시킨 후 taskDelay()를 호출하면 memSync 태스크는 잠을 자게 되고 그동안 호처리 태스크가 실행될 수 있다. 다시 일정한 시간이 지나면 또 이 태스크가 실행되어 그 다음 1 MB를 동기화시킨다. 이 태스크의 우선순위를 호 처리 태스크보다 높게 하면 이 태스크가 깨어났을 때 반드시 호처리 태스크들보다는 먼저 수행된다. watchdog 타이머는 주기적으로 지정하는 함수를 실행시켜준다. 그리고, 지정된 함수는 시스템 클럭 인터럽트 레벨에서 수행된다. 함수가 ISR이기 때문에 반응이 빠르지만, 인터럽트 레벨에서 수행되므로 ISR의 제약조건에 따라서 할 수 있는 일에 제약이 있다.

메모리 동기화가 끝나면 액티브는 memSync_End 메시지를 스탠바이에게 보내어 메모리 동기화를 마치고, 스탠바이는 정상적인 동작을 시작한다. 다음은 스탠바이의 동시쓰기 메모리를 액티브와 동기화

하는 네 가지 방법이다.

V. 동시쓰기 메모리 사용을 위한 지원

액티브가 하던 일을 스탠바이가 그대로 이어받아 수행하기 위해서는 액티브와 스탠바이에 적재되어 있는 응용 프로그램의 위치가 동일해야 한다. 스탠바이였던 시스템이 액티브가 되면 액티브에서 수행되다가 중단된 태스크들을 프로그램 카운터가 가리키는 곳부터 다시 수행시킨다. 그런데, 액티브에서 수행되던 태스크가 참조하던 응용 프로그램의 위치와 스탠바이에 적재되어 있는 응용 프로그램의 위치가 다르면, 스탠바이의 태스크는 수행이 중단되었던 부분부터 재시작할 수가 없게 된다. 프로그램 카운터가 가리키는 곳에 있는 코드가 액티브일 때와 스탠바이일 때가 다르기 때문이다. 따라서, 동기화되어야 하는 응용 프로그램은 액티브와 스탠바이의 동일한 위치에 적재되어야 한다. 이를 위하여 본 논문에서는 응용 프로그램을 동시쓰기 메모리에 적재하도록 하였다. 이 경우 액티브가 동시쓰기 메모리에 응용 프로그램을 적재하기만 하면 자동적으로 스탠바이의 동일한 위치에 응용 프로그램이 적재된다.

응용 태스크의 문맥에는 TCB (Task Control Block)와 스택이 있는데, 이들은 액티브와 스탠바이 간에 항상 동기화되어야 한다. 그렇지 않으면, 스탠바이는 액티브가 하던 일을 이어받아서 할 수 없다. 응용 태스크의 동기화를 위하여 응용 태스크의 문맥은 동시쓰기 메모리에 존재해야 한다.

따라서, 액티브는 동시쓰기 메모리에 응용 프로그램을 적재하고 태스크의 문맥을 생성해야 한다. 동시쓰기 메모리의 사용을 편리하게 해줄 수 있도록 VxWorks의 파티션을 이용하였다. 본 절에서는 동시쓰기 메모리 파티션에 대해서 설명하고, 이 파티션을 이용하여 동시쓰기 메모리에 프로그램을 적재하는 방법과 태스크를 생성하는 방법을 제시한다.

1. 동시쓰기 메모리 파티션

동시쓰기 메모리에 응용 프로그램을 적재하거나 태스크를 생성할 때 절대주소를 사용한다면 사용자가 동시쓰기 메모리에 대한 관리를 직접해야 하므로 번거롭다. VxWorks는 메모리에 파티션을 생성할 수 있도록 지원하는데, 동시쓰기 메모리를 하나의 파티션으로 생성하면 사용자는 파티션의 번호만으로 동시쓰기 메모리에서 필요한 만큼 메모리를 할당받을 수 있으므로 편리하다. 이에 본 논문에서는 동시쓰기 메모리를 하나의 파티션으로 구성하여 사용한다.

VxWorks는 메모리에 파티션을 생성해주는 memPartCreate() 함수와 파티션에 메모리를 할당해주는 memPartAlloc() 함수를 제공한다. memPartCreate() 함수는 명시한 주소부터 원하는 크기만큼 파티션을 생성한 후 파티션 번호를 리턴한다. memPartAlloc() 함수는 파티션 번호와 필요한 메모리의 크기를 요청하면 파티션에 메모리를 할당하고 그 주소를 리턴한다.

2. 동시쓰기 메모리에 프로그램 적재

본 논문에서는 동시쓰기 메모리 파티션에 응용 프로그램을 적재하는 loadModuleAtPart() 함수를 만들었다. VxWorks는 응용 프로그램을 원하는 곳에 적재할 수 있도록 loadModuleAt() 함수를 제공하고 있으나, 이 함수는 절대 주소를 이용해야 하므로 사용자가 사용하기에는 불편하다. loadModuleAtPart() 함수는 파일디스크립터가 가리키는 모듈을 동시쓰기 메모리 파티션에 적재하므로 사용자는 메모리를 관리할 필요가 없다. 이 함수의 인자로는 파일디스크립터와 심볼 플래그, 그리고 파티션 번호가 있다.

```
MODULE_ID loadModuleAtPart(int fd, int symFlag, PART_ID partID)
{
  MODULE_ID module_id;
  int stext, sdata, sbss;

  /* 로드 될 모듈의 헤더를 읽어서 text, data, bss size 알아내는 부분 */
  size(fd, &stext, &sdata, &sbss);

  /* 동시쓰기 메모리 영역에 공간을 할당 받는다. */
  pText = memPartAlloc(partID, stext); /* address of text segment */
  pData = memPartAlloc(partID, sdata); /* address of data segment */
  pBss = memPartAlloc(partID, sbss); /* address of bss segment */

  /* 할당받은 주소를 loadModuleAt() 에 넘겨준다. */
  module_id = loadModuleAt(fd, symFlag, &pText, &pData, &pBss);

  return module_id;
}
```

그림 5. loadModuleAtPart()

그림 5는 loadModuleAtPart() 함수의 알고리즘이다. 하나의 프로그램 모듈은 텍스트, 데이터, BSS 세그먼트로 나뉘어서 메모리에 적재된다. 먼저, size() 함수를 이용하여 적재할 프로그램 모듈의 텍스트, 데이터, BSS의 크기를 알아낸 후 memPartAlloc() 함수를 이용하여 텍스트, 데이터,

BSS 세그먼트를 위한 공간을 동시쓰기 메모리 파티션에 할당받고 세 가지 세그먼트에 대한 절대 주소를 리턴 받는다. VxWorks의 loadModuleAt() 함수에게 세 가지 세그먼트의 절대 주소와 함께 파일 디스크립터와 심볼플래그를 전달하면 동시쓰기 메모리에 프로그램 모듈의 적재가 완료된다. 프로그램이 적재되면 심볼들은 호스트의 시스템 심볼테이블에 추가된다. 심볼플래그는 모듈에 정의된 변수를 호스트의 시스템 추가하거나 파일을 링크할 때 심볼테이블에서 어떻게 관리할지 알려주는 플래그이다.

3. 동시쓰기메모리 파티션에 태스크 생성

원하는 위치에 태스크의 스택과 TCB를 생성할 수 있도록 VxWorks는 taskInit() 함수를 제공한다. taskInit() 함수를 이용하는 경우 사용자는 동시쓰기 메모리 영역에 대한 관리를 해야 하므로 복잡하다. 본 논문에서는 쉽게 동시쓰기 메모리 파티션에 태스크를 생성할 수 있도록 taskCreateAtPart() 함수를 만들었다.

그림 6은 taskCreateAtPart() 함수이다. memPartAlloc() 함수를 이용하여 동시쓰기 메모리 파티션에 TCB와 스택을 위한 메모리를 할당받고 그 주소를 얻는다. WIND_TCB는 VxWorks의 TCB 타입이다. TCB와 스택의 주소와 함께 우선순위와 인자 등을 taskInit() 함수에 주면 taskInit() 함수는 태스크를 생성한다. 태스크의 ID는 TCB의 시작주소를 정수형으로 변환한 값이다. taskInit() 함수는 태스크를 생성만 하고 실행하지는 않는다. 나중에 taskActivate() 함수를 이용하여 태스크를 활성화시켜야 한다.

```
int taskCreateAtPart(
  PART_ID partitionCW, char *taskName, int priority, ...,
  int stackSize, FUNCPTR entryPt, int arg1, ...)
{
  WIND_TCB *tcbLoc; /* TCB의 주소 */
  char *stackLoc;    /*스택의 주소 */
  /* 태스크의 TCB와 스택에 대한 메모리를 할당 */
  tcbLoc = (WIND_TCB *)memPartAlloc(partitionCW, sizeof WIND_TCB);
  stackLoc = (char *)memPartAlloc(partitionCW, stackSize);

  /* 태스크 생성 */
  taskInit(tcbloc, taskName, priority, ...,
        stackLoc, stackSize, entryPt, arg1, ... );
  return (int)tLoc;
}
```

그림 6. taskCreateAtPart()

4. 응용프로그램의 활용 예

usrConfig.c의 UsrRoot()는 시스템 초기화 루틴으로써 이 곳에 응용 모듈의 실행에 대한 코드를 추가한다. 그림 7은 usrRoot() 루틴에서 응용 모듈을 실행시키는 코드를 추가한 것으로써 App모듈에 있는 LoadApp() 함수를 실행시키는 태스크를 생성하고 있다. App는 App.c의 실행화일이다.

```
usrRoot()
{
  ...
  /* spawn LoadFunc if selected */
  #if defined(INCLUDE_LOADFUNC)

  TaskSpawn ("App", 20, 0, 2000, (FUNCPTR)LoadApp,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
  #endif
  ...
}
```

그림 7. 시스템 초기화 루틴

제어국의 액티브, 스탠바이 시스템은 하드디스크가 없고 메모리만 존재한다. 따라서 응용을 실행하기 위해서는 호스트에 존재하는 응용 프로그램 모듈을 제어국의 메모리로 가져와야 한다. VxWorks의 nfsMount()를 사용하면 네트워 디바이스를 생성하여 호스트의 지정한 디렉토리를 제어국 시스템 내부의 디렉토리로 마운트하여 사용할 수 있다. 네

```
/* App.c */
PART_ID partitionCW;

LoadApp()
{
  int fd;
  int tl;
  partitionCW = memPartCreate(33000, 10000); /*파티션 생성 */
  ...
  /* 호스트에 있는 응용 프로그램 모듈을 적재할 준비를 */
  hostAdd(wrs, "90.0.0.2");
  nfsMount("wrs", "/d0/", "/myd0/");
  fd = open("a", O_RDONLY);

  /*응용 프로그램 모듈을 적재 */
  loadModuleAtPart(fd, LOAD_NO_SYMBOLS, partitionCW);
  close(fd);
  nfsUnMount("wrs");
  hostDelete(wrs, "90.0.0.2");
  ...
  /* 응용 태스크를 생성 */
  tl = taskCreateAtPart(partitionCW, taskA, 50, ...,
              100, DoSomething, 1, ... );
  /* 응용 태스크를 활성화 */
  taskActivate(tl);
}
```

그림 8. App.c의LoadApp() 함수

트웍 디바이스가 설정되면, 호스트의 파일 시스템 서버에 존재하는 파일을 마치 내부에 존재하는 것처럼 다룰 수 있다. 이 함수를 사용하기 전에 hostAdd()를 호출하여 호스트를 설정한다.

그림 8은 응용 프로그램 App.c의 LoadApp() 함수이다. 동시쓰기 메모리 파티션을 생성한 후 호스트로부터 프로그램 모듈을 다운로드하기 위하여 hostAdd()와 nftMount() 함수를 호출한다. 그런 다음, 응용 프로그램 모듈을 개방하고 loadModuleAtPart를 호출하여 응용 프로그램 모듈을 동시쓰기 메모리 파티션에 적재한다. 마운트된 디바이스를 제거하고 호스트 테이블에 등록된 호스트를 제거한다. 동시쓰기 메모리 파티션의 응용 프로그램 모듈에 있는 DoSomething() 함수를 실행하는 태스크를 생성한다. 마지막으로 taskActivate()를 이용하여 생성된 태스크가 실행될 수 있게 만든다.

VI. 절체방법

절체를 결정하는 주체에 따라 액티브에 의한 절체와 스탠바이에 의한 절체가 있다. 액티브에 의한 절체는 액티브가 자신의 고장을 감지하고 스탠바이에게 절체를 요청한다. 스탠바이에 의한 절체는 스탠바이가 액티브를 진단한 결과 액티브가 죽었다고 판단되면 자신을 액티브로 전환한다.

액티브가 스탠바이에게 절체를 알리거나 스탠바이가 액티브를 진단하기 위하여 액티브와 스탠바이는 서로 통신을 해야 한다. 본 논문의 이중화 시스템 구조에서는 세 가지 통신 방법이 가능하다. 첫째, 시리얼 라인이나 이더넷과 같은 통신 매체를 통하여 액티브와 스탠바이가 메시지를 주고받는 방법이다. 둘째, 액티브와 스탠바이에서 동시에 인터럽트가 발생하게 하는 하드웨어 장치를 이용한다. 즉, 액티브가 특정 인터럽트를 걸면 스탠바이에도 동시에 인터럽트가 발생하게 된다. 마지막으로 동시쓰기 메모리를 이용한 방법이 있다. 액티브가 동시쓰기 메모리에 쓰는 모든 것이 스탠바이의 동시쓰기 메모리에도 씌여지므로 액티브가 스탠바이에게 전달할 정보를 동시쓰기 메모리에 쓰면 스탠바이는 그 정보를 받을 수 있다.

1. 액티브에 의한 절체

액티브의 인터럽트 프로세서나 고장 감지 태스크는 액티브의 하드웨어 고장을 감지하고 절체할지 여부를 결정한다. 절체를 결정을 하였으면 액티브는 스탠바이에게 절체를 해야 한다고 알려준다. 스탠바이는 절체 요청을 통보 받고 액티브로 전환한다.

액티브의 dgByActive(diagnosis by active) 태스크는 하드웨어 요소의 고장을 감지하기 위하여 이들을 주기적으로 진단하고 하드웨어 고장이 감지되면 스탠바이의 toContinue 태스크에게 절체를 요청한다. 액티브는 각 하드웨어 고장 인터럽트에 대해서 시스템 초기화 시에 takeover ISR을 미리 등록한다. 인터럽트 프로세서가 하드웨어의 고장을 감지하여 인터럽트를 발생시키면 실행 중이던 태스크가 중단되고 takeover ISR이 실행된다. takeover ISR도 스탠바이의 toContinue 태스크에게 절체를 요청한다. toContinue (Takeover Continue) 태스크는 스탠바이를 액티브로 전환하는 작업을 수행한다. 이를 위하여 스탠바이에서만 실행되는 태스크들은 종료시키고 액티브로써 동작하기 위하여 필요한 태스크들을 활성화시킨다. 그리고, 동시쓰기 메모리의 방향을 바꾼다. 이 작업들은 중단되면 안되는 중요한 작업이므로 인터럽트나 다른 태스크에게 제어를 빼앗기지 않아야 한다. 이를 위하여 먼저 intLock() 과 taskLock()을 호출한다. 스탠바이에서 액티브로의 전환 작업이 끝나면 taskUnlock()과 intUnlock() 함수를 호출하여 인터럽트가 처리되고 태스크들이 정상적으로 실행될 수 있게 한다.

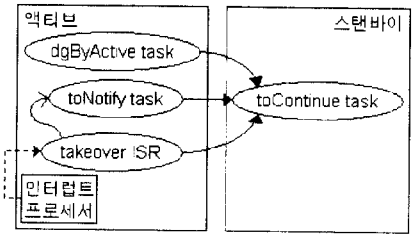


그림 9. 액티브에 의한 절체 과정

takeover ISR이나 dgByActive 태스크는 세 가지 액티브/스탠바이 통신 방법을 하나 이상 이용하여 스탠바이의 toContinue 태스크에게 절체요청을 전달할 수 있다. 메시지 전달을 이용한 방법에서는 액티브의 dgByActive와 toNotify 태스크가 스탠바이의 toContinue 태스크에게 절체요청 메시지를 전송한다. ISR은 메시지를 전송할 수 없으므로 takeover ISR은 toNotify 태스크를 활성화시켜서 대신 절체요청 메시지를 스탠바이에게 전달하게 한다. toNotify 태스크는 시스템 초기화 시에 생성만 되고 실행되지는 않게 대기시킨다.

인터럽트를 이용한 방법에서는 takeover ISR이나 dgByActive 태스크가 절체요청 인터럽트를 발생시킨다. 그러면, 스탠바이에서도 동일한 인터럽트가 발생되고 이 인터럽트에 연결되어 있는 ISR이 실행된다. 이 ISR은 바로 toContinue 태스크를 활성화시켜서 스탠바이가 액티브로 전환될 수 있게 한다. 이 방법은 메시지로 전달하는 것 보다 신속하게 절체를 알릴 수 있는 장점이 있다

동시쓰기 메모리를 이용한 방법은 동시쓰기 메모리에 SB(Status Bit)를, 기본 메모리에 Local_SB를 두어 SB와 Local_SB가 같으면 액티브를 수행하도록 하는 방법이다. Local_SB의 내용은 변함이 없고 액티브가 사용하는 동시쓰기 메모리의 SB를 스위치시킴으로써 스탠바이에게 절체를 알린다. 이 경우 스탠바이의 toContinue 태스크는 주기적으로 SB와 Local_SB를 감시하여야 한다. 그림 10은 절체되는 과정을 보여준다. 1) 처음에 동시쓰기 메모리의 SB는 1이라고 가정한다. 2) 시스템 A가 자신이 액티브가 되어야 한다고 판단하면 현재 동시쓰기 메모리의 SB를 자신의 Local_SB로 가져온다. 3) 시스템 B는 자신이 스탠바이가 되어야 한다고 판단하고 현재 동시쓰기 메모리의 SB의 Not 값을 Local_SB로 가져온다. 4) 시스템 A에 고장이 발생하면 takeover ISR이나 dbByActive 태스크는 시스템 B에게 절체를 알리기 위해 동시쓰기 메모리의 SB를 스위치시킨다. 5) 주기적으로 시스템 B의 SB와 Local_SB를 감시 중인 toContinue 태스크가 SB와 Local_SB가 같다는 것을 감지한 후 시스템 B를 액티브로 전환시킨다. 6) 시스템 A가 재부팅되었을 때에는 스탠바이가 되어야 하고 SB가 0이므로 Local_SB는 1이 된다

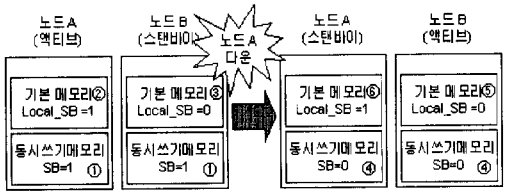


그림 10. 동시쓰기메모리를 이용하여 절체를 알리는 방법

2. 스탠바이에 의한 절체

스탠바이는 주기적으로 액티브가 정상인지 확인해야 한다. 확인 결과 액티브가 정상이 아니라면 스탠바이가 절체를 결정한다. 스탠바이의 dgByStandby(diagnosis by standby) 태스크가 액티브에게 주기적으로 신호를 보내 응답이 있는지를 확인하거나 액티브가 주기적으로 보내는 신호를 확인한다. 액티브가 일정 시간동안 아무런 반응이 없으면 dgByStandby 태스크는 액티브가 정상이 아니라고 판단하고 toContinue 태스크를 활성화시켜 스탠바이를 액티브로 전환시킨다. 그런데, 액티브는 정상적으로 동작하고 있음에도 액티브/스탠바이 통신 매체의 고장에 의해서 응답이 없을 수도 있다. 따라서, 스탠바이는 여러 가지 방법으로 액티브의 정상작동 여부를 확인하여야 한다. 그림 11은 액티브/스탠바이의 세 가지 통신 방법을 이용하여 액티브의 고장을 알아내는 과정을 보여준다.

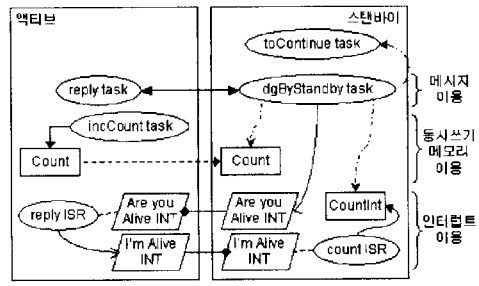


그림 11. 스탠바이에 의한 절체결정

스탠바이는 세 가지 액티브/스탠바이 통신 방법을 이용하여 액티브로부터 응답이 있는지를 확인한다. 시리얼 라인이나 이더넷을 통한 태스크간 통신 프리미티브를 이용하여 스탠바이는 액티브의 태스크에게 주기적으로 "Are you alive" 메시지를 보내고 액티브로부터의 "I'm alive" 메시지를 받는다.

인터럽트를 이용하여 액티브의 고장을 알아내기 위해서는 액티브와 스탠바이에서 동시에 발생하는 "Are you alive"와 "I'm alive" 인터럽트를 이용한다. 액티브는 "Are you alive" 인터럽트에 reply ISR을 연결하고, 스탠바이는 "I'm alive" 인터럽트에 count ISR을 연결한다. reply ISR은 "I'm alive" 인터럽트를 발생시켜서 스탠바이에게 액티브가 잘 작동하고 있다는 것을 알린다. count ISR은 실행될 때마다 CountInt 변수의 값을 증가시킨다. 스탠바이의 dgByStandby 태스크는 주기적으로 "Are you alive" 인터럽트를 발생시킨다. 그러면, 액티브의 reply ISR이 "I'm alive" 인터럽트를 발생시키고 스탠바이의 count ISR은 CountInt 변수의 값을 증가시킨다. dbByStandby 태스크는 CountInt 변수의 값이 주기적으로 증가하고 있는지를 검사한다. 액티브가 다운된 경우에는 reply ISR이 응답하지 못하므

로 CountInt 변수의 값이 증가하지 않게 된다.

동시쓰기 메모리를 이용한 방법으로는 동시쓰기 메모리의 Count 변수를 이용한다. 액티브의 incCount 태스크는 Count 변수의 값을 주기적으로 증가시킨다. Count 변수는 동시쓰기 메모리에 있으므로 스탠바이의 동일한 주소에 있는 Count 변수도 같이 증가한다. 스탠바이의 dgByStandby 태스크는 Count 변수가 주기적으로 증가하지 않으면 액티브가 고장난 것으로 판단할 수 있다.

Ⅶ. 결 론

기존 이중화 시스템은 이중화를 지원하는 운영체제를 직접 만들어 사용하였다. 이 경우 하드웨어 구성이나 이중화 기능이 변했을 경우 운영체제를 직접 수정해야 하는 어려움이 있으며, 개발환경도 직접 개발해야 하므로 많은 비용과 시간이 필요하다. 상용 운영체제를 이용하면 상용 운영체제가 제공하는 좋은 개발환경을 그대로 이용할 수 있으며 하드웨어 구성이나 이중화 기능이 변하더라도 더욱 쉽게 수정할 수 있다는 장점이 있다.

본 논문은 동시쓰기 메모리와 상용 운영체제인 VxWorks를 사용하는 제어국에서 이중화를 지원하기 위한 방법을 제안하였다. 이중화를 지원하는 태스크들이 높은 우선순위로 선점되지 않고 즉시 수행될 수 있도록 하는 방법과 스탠바이가 부팅된 후 액티브와 메모리를 동기화하는 방법을 제시하였다. 그리고, 사용자가 동시쓰기 메모리를 쉽게 사용할 수 있도록 VxWorks의 파티션을 이용하는 설명하였다. 마지막으로 액티브의 하드웨어와 소프트웨어가 하드웨어 고장을 감지했을 때의 절체 방법과 스탠바이가 액티브의 고장을 감지했을 때의 절체 방법을 설명하였다.

향후 과제로는 두 시스템이 부팅되었을 때 서로 협상을 하여 액티브와 스탠바이를 결정하는 방법, 액티브가 결함을 감지하는 방법, 스탠바이가 액티브가 고장났음을 판단하는 알고리즘, 스탠바이가 언제든지 액티브로 전환되어 정상적으로 동작할 수 있는지 여부를 확인하는 방법 등이 연구되어야 한다. 그리고, 동시쓰기 메모리를 사용하지 않는 구조에서 이중화를 지원하는 방법에 대한 연구도 필요하다.

참 고 문 헌

[1] J. H. Kim, H. T. Lim, J. H. Rhee, S. J. Lee, J. H. Ahn, and S. M. Yang, "COTS based duplication of BSC," *Proc. Of 4th CIC* (CDMA International Conference), pp.481-484, Sep. 1999.

[2] 김종호, 임형택, 방경은, 이제헌, 안지환, 양승민, "상용 운영체제 기반의 제어국 이중화구조 설계," *99년 제4회 통신 소프트웨어 학술대회*, pp.137-141, 1999.

[3] Thomas C. Bressoud, "TFT: A Software System for Application Transparent Fault Tolerance," *Proc. of 28th International Symposium on Fault-Tolerant Computing (FTCS)*, June 1998.

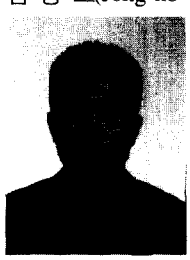
[4] Z.T. Kalbarczyk, R.K. Iyer, and S. Bagchi, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 6, pp.560-579, June 1999.

[5] Ozalp Babaoglu, "Fault-tolerant computing based on Mach," *Technical report, Cornell University*, Dept. of Computer Science, Number TR 89-1032, Aug. 1989.

[6] D. P. Siewiorek and R. S. Swarz, Reliable Computer Systems: Design and Evaluation, 2nd ed., *Digital Press*, 1992.

[7] CROS, http://circle.etri.re.kr/research/cros/cros.html

김 종 호(Jong-ho Kim) 준회원



1998년 2월 : 경원대학교 전자계산학과 졸업
2000년 2월 : 숭실대학교 컴퓨터학과 석사
1999년 12월~현재 : (주) 팬택 기술연구소 연구원

<주관심 분야> 무선통신, 실시간시스템.

이 제 헌(J. H. Rhee) 정회원

1993년 2월 : 숭실대학교 전자계산학과
1995년 8월 : 숭실대학원 전자계산학과 석사
1995년 9월~현재 : 한국전자통신연구원 연구원

임 형 택(Hyung-taek Lim) 준회원

1994년 2월 : 숭실대학교 전산과 졸업
1996년 2월 : 숭실대학교 전산과 석사
1996년 3월~현재 : 숭실대학교 컴퓨터학과 박사과정

<주관심 분야> 결함허용, 실시간시스템

방 경 은(Kyeong-eun Bang) 준회원

1999년 2월 : 숭실대학교 컴퓨터학부 졸업
1999년 3월~현재 : 숭실대학교 컴퓨터학과 석사과정
<주관심 분야> 결함허용, 실시간시스템

이 숙 진(S. J. Lee) 정회원

1990.2 : 경북대학교 전자공학과
1990년 3월~현재 : 한국전자통신연구원 선임연구원

임 순 용(S. Y. Lim) 정회원

1981년 2월 : 인하대학교 전자공학과
1983년 2월 : 인하대학원 전자공학과 석사
1983년 3월~현재 : 한국전자통신연구원 책임연구원

양 승 민(S. M. Yang) 정회원

1978년 2월 : 서울대학교 전자공학과 졸업
1978년 2월~1981년 7월 : 삼성전자(주)
1983년 4월 : Univ. of South Florida 전산학 석사
1986년 12월 : Univ. of South Florida 전산학 박사
1986년 8월~1987년 8월 : Univ. of South Florida 조교수
1988년 1월~1993년 1월 : Univ. of Texas at Arlington 조교수
1996년~1998년 : 국회도서관 정보처리국장
1992년 12월~현재 : 숭실대학교 컴퓨터학부 부교수