Process-Based 운영체제 기반 Embedded S/W 의 Thread-Based 운영체제 기반으로의 변환을 위한 OS Abstraction Layer 의 구현

이종인, 신정민, 한국현, 권재욱
삼성전자 디지털 미디어 연구소 SW Platform Lab

Implementation of OSAL to Migrate Embedded S/W from Process-Based System to Thread-Based System

*Jong In Lee, Jungmin Shin, Kuk-Hyun Han, Jaewook Kwon
Software Platform Lab Digital Media R&D Center Samsung Electronics
*chadlee@samsung.com

요 약

정보 가전 분야에 있어서 급속한 기술 발전으로 인해 하루가 다르게 새로운 기능이 추가됨에 따라 embedded system S/W 의 복잡도 또한 증가하고 있고 이를 개발하고 유지보수 하는데 있어서도 막대한 비용과 노력이 요구되고 있다. 이로 인해 embedded S/W 에도 재사용성의 요구가 높아지고 있지만 embedded system 의 경우 제품의 특징에 따라 system 의 process model 을 변경해야 하는 경우가 있는데 이러한 경우 OS 의 변경으로 인해 S/W 의 재사용성이 크게 낮아진다. 즉, process-based model 에서 thread-based model 로 변경하는 경우 OS 변경으로 인해 전체 S/W 의 변경이 요구되어 개발의 효율이 떨어지게 되는 것이다. 이에 본 논문에서는 OS 변경으로 인한 전체 S/W 의 변경을 최소화 하고자 OS abstraction layer 를 도입하고 process-based model 에서 thread-model 로의 변환 시 발생하는 변경 요소인 process 와 thread 간의 변환에 따른 memory 및 scheduling 관련 사항, IPC (Inter-Process Communication) 관련 사항, device I/O 관련 사항 중, 상위 S/W 의 재사용성에 직접적인 영향을 미치는 IPC 에 대해 OSAL 을 통한 process/thread model 공통 interface 를 제안하고자 한다. 특히, 고려해야 할 사항이 많은 components 인 semaphore, shared memory, socket communication 등의 OSAL 구현 방법을 Linux to VxWorks 의 경우에 대해 제안하고 이를 이용한 DTV 시스템의 구현을 통해 타당성을 검증한다.

I. 서 론

임베디드 시스템의 기능이 다양해지고 규모가 커짐에 따라 운영체제(OS)의 사용은 필수적이 되었고, OS 선정 또한 시스템의 특성에 적합하게 다양화되고 있다[1]. 이에 임베디드 시스템 구현에 있어서 OS 선정은 시스템 구현의 시작일 뿐만 아니라 시스템의 성능을 결정하는 중요한 요소이다. 하지만 각 시스템 및 OS 가 갖는 특성으로 인해, 특정 시스템에 특정 OS 가 적합하다는 식의 일률적인 규칙을 적용하기는 어렵다. 이로 인해 프로세스 모델을 변경해야 하는 경우가 발생하게 되는데 이 때 OS 변경으로 인해 전체 S/W 의 재사용성이 크게 떨어지게 된다. 프로세스 기반 모델은 응용 프로그램이 process 의 단위로 수행되며 각 process 들은 서로의 메모리가 보호되어 있어서 application 개발이나 모듈의 추가, 변경이 용이하고, 안정된 시스템의 개발이 가능하여 대규모 시스템의 개발에 적합한 특징을 갖는데[2], 반면 쓰레드 기반 모델은 응용 S/W 즉, thread 들이 address 공간을 공유하는 방식으로 공통의 작업영역(memory)을 자유롭게 접근할 수 있을 뿐 아니라 OS 의 크기가 작고, 구현이 용이하고 빠르다는 특징을 갖기 때문이다[3]. 특히 DTV 와 같은 경우는 임베디드 시스템의 규모가 커지고, 각 CE(Consumer Electronics) 제품 군의 특성이 복잡해짐에 따라 다양한 OS 의 적용이 시도되게 되는데, 이러한 경우 OS 의 변경으로 인하여 수정이 필요 없는 응용 프로그램 등을 포함한 전체 S/W 의 변경 및 재구현이 불가피한 경우가 발생한다[4]. 이를 방지하기 위해서는 OS 변경에 따른 수정 부분 최대한 abstraction 하여 빠른 시간 내에 시스템의 재구성이 가능하도록 하는 embedded S/W 구조가 필요하게 된다.

본 논문에서는 특히 상위 S/W 의 변경이 많이 요구되는 process-model 에서 thread-based model 로의 변환 시 필요한 OSAL 을 구현하는 데 있어서 고려 요소인 process-thread 변환에 따른 memory 및 scheduling 관련 사항, IPC(Inter-Process Communication) 관련 사항, device I/O 관련 사항 가운데 상위 S/W 의 재사용을 낮추는 직접적인 영향을 갖는 IPC 에 대해 OSAL 을 구현하는데 있어서 OSAL 을 통한 process/thread model 공통 interface 를 제안하고자 한다. 특히, semaphore, shared memory, socket communication 등 process model 변환 시 고려해야 할 사항이 많은 components 의 OSAL 구현 방법을 제안하고 Linux to VxWorks 의 경우에 대

해 적용을 하고, 이를 이용한 DTV 시스템의 구현을 통해 검증하고자 한다.

2 장에서는 OS 변경에 OSAL 을 갖는 임베디드 S/W 구조 및 그 방법론을, 3 장에서는 제안된 S/W 구조 및 방안의 구체적인 적용 예로서 Linux 에서 VxWorks 로의 변환과정을 기술한다. 4 장에서는 제안된 S/W 구조 및 방안을 실제 디지털 TV 시스템에 적용, 검증하고, 마지막으로 5 장에서는 결론과 향후 과제를 제시한다.

II. Embedded System S/W Hierarchy

1. Conventional Hierarchy

임베디드 시스템은 일반적으로 그림 1 (a)와 같이 하위 임베디드 시스템 H/W 계층부터 상위 응용 프로그램 계층까지의 구조를 갖는다. 여기서 H/W Abstraction Layer (HAL)를 두는 것은 H/W 변경에 따라 모든 S/W 계층을 변경하지 않아도 되도록 하기 위해 필수적이다. 즉, HAL 이상의 계층의 S/W 는 재사용이 가능하도록 하기 위한 설계구조이다[5].

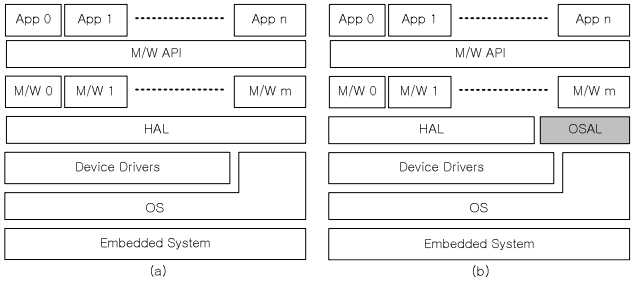

그림 1. Embedded S/W Hierarchy (a) Conventional (b) OS abstracted

2. OS Abstracted System

OS 와의 직접적인 연관성을 배제하기 위해 그림 1 (b)와 같이 S/W 구조에 OS Abstraction Layer (OSAL)을 추가할 수 있다. OSAL 은 OS 의 변경에 대하여 상위 S/W 에 대한 재사용성을 제공한다. OSAL 에서 각 OS 의 변환요소에 대해 추상화한 공통적인 API 를 제공함으로써 application 및 middleware(M/W)의 변경 없이도 OS 를 변환하여 사용하는 것이 가능하다. 즉, 그림 3 과 같이 각 OS 에서 제공되는 kernel component 별로 공통적인 API 를 제공하여 상위 응용 프로그램에서 이를 이용하여 OS 에 접근하도록 하는 것이다.

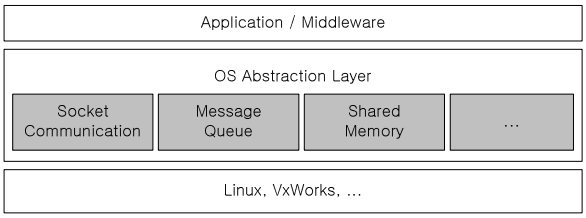

그림 2. OS Abstraction Layer

이러한 추상화 계층을 통해 S/W 를 구현하면 OS 가 바뀌어도 검증된 S/W 모듈(module)은 수정하지 않고 추상화 계층의 수정만을 통해 전체 S/W 의 변환(migration)이 가능하다. 이와 더불어 OS 에서 해당 기능 자체를 제공하지 않는 경우에는 추상화 계층만의 수정으로는 OS 변경이 불가능하므로 해당 기능을 제공하는 OS 의 API 형태를 유지하여 해당 기능을 추가 구현하는 과정도 필요하다. 특히 프로세스 기반 모델을 쓰레드 기반 모델로 변경하는 경우, 프로세스간의 통신과 같이 쓰레드 기반 OS 에서 제공되지 않는 기능에 대해 구현해야 할 필요성이 있다.

III. Kernel Components Interface: Linux to VxWorks

프로세스 기반 OS 로 구현된 임베디드 S/W 를 쓰레드 기반 OS 에서도 사용할 수 있도록 구현하기 위해서는 프로세스 기반 OS 의존적인 구현 부분을 쓰레드 기반 OS 의존적으로 변경 구현해야 한다. 이러한 OS 의존적인 부분들을 OSAL 로 구현함으로써 대부분의 S/W 모듈들은 재활용이 가능해진다. 본 장에서는 프로세스 기반 OS 와 쓰레드 기반 OS 로서의 Linux 와 VxWorks 에 대한 실제 OSAL 구현 방법을 기술한다.

1. Linux vs. VxWorks

임베디드 S/W 의 OS 에 대한 호환성을 유지시키기 위한 OSAL 을 구현하기 위해서는 임베디드 S/W 관점에서 본 OS 간의 차이점을 명확히 이해해야 한다. 표 1 은 임베디드 S/W 에서 활용하는 OS 의존적인 요소들 중 주요 항목에 대해 Linux 와 VxWorks 에 대한 비교 결과다. Linux 와 VxWorks 의 가장 큰 차이점은 프로세스 개념의 지원 여부이다. Linux 는 프로세스를 지원하기 때문에 임베디드 S/W 를 개발할 경우 여러 개의 프로세스로 나누어 구현할 수 있지만, VxWorks 의 경우에는 프로세스 개념을 적용하여 임베디드 S/W 를 구현하는 것이 불가능하다. 또한, 이와 관련하여 Linux 에서만 프로세스 기반 공유메모리(shared memory) 기능을 제공한다. 세마포어(semaphore) 기능은 두 OS 모두 지원하지만, 프로세스를 기반으로 하는 세마포어와 쓰레드를 기반으로 하는 세마포어는 인터페이스(interface) 측면에서 차이가 있다. 태스크(task)간 통신의 경우, 프로세스간의 소켓(socket) 통신은 VxWorks 에서는 지원하지 않는다.

	Linux	VxWorks
Process	지원	미지원
POSIX	지원	지원
Shared memory	Process-based	Thread-based
Inter-task communication	Socket, Message queue	Message queue
Semaphore	Process-based	Thread-based
Task Scheduling	FIFO / Round-robin	Priority-based preemptive / Round-robin

표 1. Linux 와 VxWorks 의 커널 지원 요소 비교

2. Linux to VxWorks

본 절에서는 프로세스의 지원 여부에 따른 OS 의존적인 요소에 대한 인터페이스를 유지시킬 수 있는 OSAL 의 구현 방법 중 앞서 서론에서 언급한 바와 같이 IPC interface 구현에 대해서 다루기로 하고 특히 Linux to VxWorks 와 같이 process 의 변환

시 공통 interface 를 유지하기 어려운 semaphore, shared memory, socket communication 에 대해 다루도록 한다.

2.1 세마포어 (Semaphore)

OSAL 구현이 필요한 커널 요소 중 세마포어와 같이 두 가지의 OS 에서 모두 지원되는 API 가 존재하는 경우 이를 공통적으로 사용할 수 있도록 그림 3 과 같이 세마포어 인터페이스를 공통 API 형태로 구현한다. 세마포어 인터페이스는 동일한 API 를 갖지만, OS 에서 제공하는 세마포어 API 를 호출하게 된다. 이때, 프로세스를 지원하지 않는 VxWorks 의 세마포어 API 의 경우에는 프로세스간의 공유 키 값에 대한 관리가 이루어지지 않는다. 따라서, 공유 키 값에 대한 structure 를 정의하고, structure 를 전역적으로 관리하는 전역 리스트(list) 변수의 선언이 필요하다.

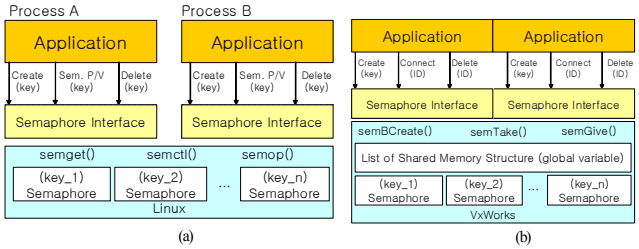

그림 3. Semaphore Interface (a) Linux (b) VxWorks

2.2 공유 메모리 (Shared Memory)

그림 4 (a)는 Linux 에서 제공되는 공유 메모리 인터페이스를 도시한 것이다. VxWorks 의 경우 프로세스 기반 공유 메모리 기능을 제공하지 않기 때문에, 공유 메모리 인터페이스를 그대로 유지시키기 위한 방법이 요구된다. 이와 같은 경우에는 VxWorks 에서도 마치 프로세스 기반 공유 메모리 기능이 동작하는 것처럼 구현해 주는 방법이 가장 효율적이다.

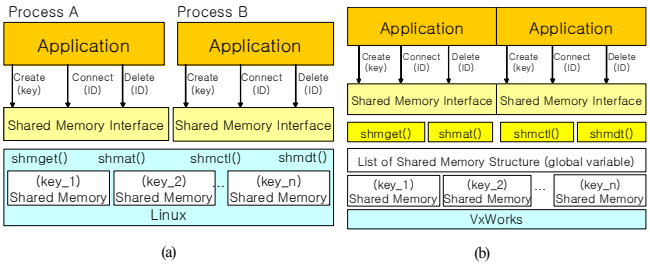

그림 4. Shared Memory Interface (a) Linux (b) VxWorks

그림 4 (b)는 VxWorks 에서의 공유 메모리 인터페이스의 구현을 도시한 것이다. Linux 에서 제공되는 프로세스 기반 공유 메모리 API 와 동일한 API 를 VxWorks 에서도 사용할 수 있도록 추가 구현한다. 이때, 세마포어와 마찬가지로 프로세스간의 공유 키 값의 개념을 유지시키기 위하여 공유 메모리에 대한 structure 의 선언 및 structure 의 전역 리스트 관리를 추가 구현한다. 본 논문에서 제안한 방안으로 구현한 경우, 응용 프로그램에서는 공유 메모리 통신이 이루어지고 있는 것처럼 보인다. 즉, OS 차원에서 공유 메모리를 지원하지 않더라도 상위 계층의 S/W 는 전혀 수정하지 않고, 공유 메모리의 기능을 그대로 유지할 수 있게 된다.

2.3 소켓 인터페이스(Socket Interface)

Linux 와 같이 프로세스를 지원하는 OS 를 기반으로 개발된 시스템 S/W 에서 프로세스간 name-based 소켓 통신을 수행하는 모듈을 VxWorks 와 같이 프로세스를 지원하지 않는 OS, 즉, 쓰레드 기반으로 변경하기 위해서는 모듈 자체의 통신 방식을 수정해야 한다. 프로세스를 지원하지 않는 OS 에서는 name-based 소켓 통신 기능을 지원하지 않기 때문에 통신 방식 자체를 message queue 방식 등으로 변경해야 하는데, 이는 곧 전체 S/W 를 수정해야 하는 과정을 거쳐야 한다는 것을 의미한다. 이러한 변경 과정을 피하기 위해 name-based 소켓 통신 기능을 제공하지 않는 OS 에서도 마치 name-based 소켓 통신이 동작하는 것처럼 구현하는 방법으로, name-based 소켓 통신 인터페이스는 그대로 유지한 채 소켓 인터페이스 내부의 구현을 변경한다.

소켓 통신은 그림 5 (a)와 같이 server 와 client 로 나뉘며, server 는 원하는 이름(eg. "NAME")을 인자로 갖는 create 함수를 통해 생성된다. server 가 생성이 되면 client 의 접속을 위해 대기한다. client 에서 "NAME"의 이름으로 connect 함수를 이용하여 server 에 접속하면, server 에서 client 로의 소켓 통신 채널이 형성되게 된다. 소켓 통신 채널이 형성된 이후에는, server 또는 client 가 각각 원하는 시점에서 send 함수를 이용하여 packet 을 전송 할 수 있다. 이때 client 또는 server 는 각각 receive 함수를 이용하여 packet 을 수신하게 된다. 프로세스를 지원하지 않은 OS 는 상위의 S/W 에서 동일한 인터페이스를 유지하며 name-based 소켓 통신이 동작하는 것처럼 보이도록 구현하기 위해서 그림 5 (b)와 같이 소켓 인터페이스 내부에 메시지 큐(message queue) 인터페이스를 구현한다.

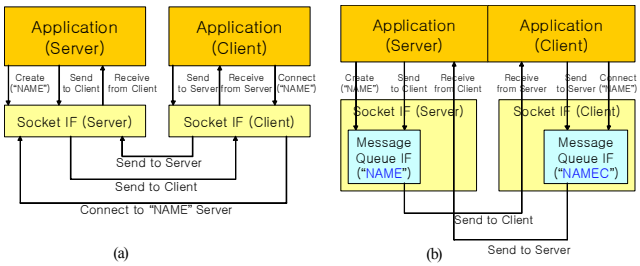

그림 5. Name-based Socket 통신 Interface (a) Linux (b) VxWorks

server 에서 소켓 인터페이스를 "NAME"이름으로 생성하게 되면, 소켓 인터페이스 내부적으로 "NAME"이름을 갖는 메세지큐를 생성한다. 메시지 큐는 단방향 통신 채널을 제공하므로, "NAME"이라는 이름으로 생성된 메시지 큐는 server 에서 client 로의 통신 채널만을 제공하게 된다. client 에서 "NAME"이라는 이름으로 접속을 시도할 경우, 소켓(client) 인터페이스 내부에서 예를 들어 "NAMEC"와 같은 이름으로 일부 수정하여 "NAMEC"라는 이름을 갖는 새로운 메시지 큐를 생성한다. 즉, client to server 통신 채널을 생성하는 것이다. 이렇게 양방향의

통신 채널을 2 개의 메시지 큐를 이용하여 생성하고 나면, server 에서 client 로 패킷을 전송할 경우에는 "NAME" 메시지 큐를 이용하고, client 에서 server 로 패킷을 전송할 경우에는 "NAMEC" 메시지 큐를 이용하면 된다.

상기와 같이 본 논문에서 제안한 방법으로 구현할 경우에는 응용 프로그램 입장에서는 name-based 소켓 통신이 이루어지고 있는 것처럼 보인다. 즉, OS 차원에서 name-based 소켓 통신을 지원하지 않더라도, S/W 는 전혀 수정하지 않고 name-based 소켓 통신을 그대로 유지할 수 있게 된다.

IV. 실험 결과

본 논문에서는 임베디드 시스템의 OS 변환 사례 적용을 위해 CE 계열 중 비교적 규모가 큰 DTV 를 target system 으로 선정하였고, Linux 와 VxWorks 두 OS 간의 변환에 대해 본 논문에서 제안한 방안을 적용한 OSAL 의 구현과 이에 따른 전체 시스템 동작의 시험 평가를 진행하였다. 디지털 TV 시스템의 경우 기능과 규모의 특성으로 인하여 프로세스 기반 OS 와 쓰레드 기반 OS 의 적용이 병행되고 있으며, 디지털 TV 제품군의 제품별 특성에 맞게 OS 의 변환이 수시로 발생할 가능성이 많다. 본 연구에서는 OSAL 부분의 독립적인 검증을 위하여 H/W 부분을 추상화한 simulator 를 구현하여 시험을 하였을 뿐만 아니라, 실제 디지털 TV 시스템의 기본 기능 구현을 통하여 본 논문에서 제안한 방법에 대해 검증하였다.

1. Simulated System

3 장에서 제안한 변환 방법의 실제 구현 및 시험을 위하여 DTV-simulated 시스템을 구현하였으며, 디지털 TV 기능에 해당하는 H/W 및 HAL 부분을 DTV 시뮬레이터로 대체하고 상위의 메시지 통신 및 응용 프로그램을 구동시켜 상기 기능을 시험하였다. 즉, H/W 적인 부분을 배제하고 OS 의존적인 부분만을 검증하고자 그림 6 (a)와 같이 가상 플랫폼(Virtual Platform)으로 구성된 시뮬레이터상에서 시험을 수행하였다.

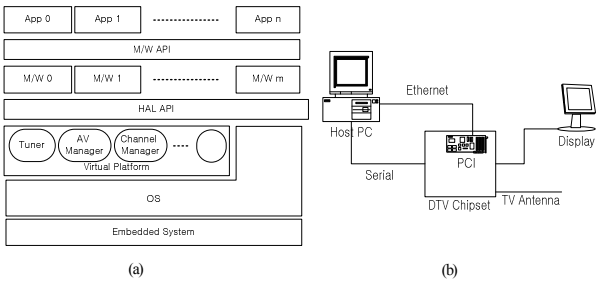

그림 6. 통합 Test 환경 (a) Simulator S/W 구조 (b) 개발 환경

2. DTV System

4.1 절에서 검증된 OSAL 및 S/W 를 이용하여 그림 6 (b)와 같이 DTV 시스템을 구현하였다. DTV chipset 의 core 는 MIPS32 이고, host PC 와 target 과의 통신은 serial 통신 및 ethernet 을 사용하였다. 공중파 DTV 방송을 수신하여 MPEG2 decoding 처리 후 화면을 출력하는 기능과 채널 업/다운, 볼륨 업/다운 등의 기본 기능, 그리고 배너(banner) 및 메뉴를 통한 그래픽 화면 출력 등의 DTV 응용 프로그램 구현 실험을 통하여 본 논문에서 제안된 OSAL 의 기능이 정상적으로 동작함을 검증하였다.

이와 같이 제안된 OSAL 을 이용하여 Linux to VxWorks 변환 시 기능 구현 및 성능 상 이상 없이 동작함을 검증하였고, 이 때 상위 S/W 중 IPC 를 사용하는 모듈에 대해서는 Linux 상의 S/W 를 VxWorks 상에서 100% 재사용이 가능하였다.

V. 결론 및 향후 연구

CE 제품을 비롯한 많은 임베디드 시스템들은 기능이 다양해지고 있을 뿐만 아니라, 제품별 목표 시장(target market)에 따른 다양한 제품군들이 필요하게 되었다. 이에 임베디드 시스템은 H/W 와 응용 프로그램뿐만 아니라 OS 자체의 변경도 요구되어지고 있는 추세이다. 이에 본 논문에서는 OS 변경에 따른 전체 임베디드 S/W 의 변경 부분을 최소로 하기 위하여 OSAL 을 적용한 구조와 이에 대한 구현 방법론을 제안하였다. 또한, 실제 OS 변환 이슈(issue)가 많은 Linux 에서 VxWorks 로의 변환에 대하여 주요 커널 요소별로 변환 내용을 기술하였다. 이러한 제안된 사항들을 실제 디지털 TV 에 적용하여 구현한 결과, 상위 응용 프로그램의 변경 없이 각각의 OS 에서 구동되는 공통 S/W 스택(stack)을 확보하였다. 시뮬레이터를 통하여 각 변환 요소를 검증했을 뿐만 아니라, 디지털 TV 시스템에의 적용을 통하여 성능 부분에 있어서 본 논문에서 제안한 방법의 무결성을 검증하였다. 이러한 제안된 구조에 기반한 개발을 통하여 용이한 OS 변환 및 효율적인 유지보수뿐만 아니라, 시장의 요구에 대한 time-to-market 의 확보도 가능하게 된다.

마지막으로, OS 의존성을 배제시킨 임베디드 S/W 를 구현하기 위해 반드시 고려되어야 할 사항 중의 하나인 태스크 스케줄링 정책에 대해서는 추후 과제로 진행하고자 한다.

참고문헌

[1] Q. Li, C. Yao, Real-Time Concepts for Embedded Systems, 2003.

[2] B. Graaf, M. Lormans, H. Toetenel, "Embedded Software Engineering: The State of The Practice," Software, IEEE, Vol. 20, No. 6, Nov. 2003, pp. 61-69.

[3] I-L. Yen, J. Goluguri, F. Bastani, L. Khan, J. Linn, "A Component-Based Approach for Embedded Software Development," Proc. Fifth IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing, 2002, pp. 402-410.

[4] L. Zhu, F. Wang, "Component-Based Constructing Approach for Application Specific Embedded Operating Systems," Proc. Intelligent Transportation Systems, Vol. 2, 2003, pp. 1338-1343.

[5] D. Stepner, N. Rajan, D. Hui, "Embedded App. Design Using A Real-Time OS," Proc. Conf. on Design Automation, 1999, pp. 151-156.