재사용성 및 신뢰성을 고려한 리눅스기반 임베디드 디바이스 드라이버 개발 기법

최 재 현[†] · 이 우 진[††] · 정 기 원[†††]

요 약

최근 임베디드 시스템의 급격한 발전으로 인해, 임베디드 시스템의 활용 범위와 복잡도는 증가하고 있지만 임베디드 시스템의 개발생산성과 품질은 그러한 현실에 부합하지 못하고 있다. 이것은 임베디드 시스템 개발, 특히 소프트웨어의 개발에 있어서는 특정 하드웨어 환경에 종속적인 개발방식에서 비롯된 것이며 임베디드 시스템의 개발생산성 및 품질을 향상시키기 위해서는 이러한 개발방식에 대한 개선이 시급하다. 이러한 개발방식의 개선을 위한 기존 연구들에서는 별도의 인터페이스 명세언어나 복잡한 템플릿을 이용하여 디바이스 드라이버를 자동으로 생성하도록 하고 있지만 명세기법의 높은 복잡도나 부족한 유연성으로 인해 실제로 적용하기에는 많은 노력이 필요하다. 이에 따라 본 논문에서는 재사용 가능한 디바이스 드라이버 개발을 위해 가벼운 템플릿을 정의하고, 모델링 측면을 감안하여 XML로 표현되는 하드웨어 정보를 결합하여 디바이스 드라이버를 생성하는 방안을 제시한다. 이것은 개발자가 손쉽고 적용 가능한 템플릿으로 디바이스 드라이버를 정의하고, XML로 명세된 특정 하드웨어 정보를 결합하여 실제 드라이버를 생성함으로써 높은 개발생산성을 보장하며 발전적으로 설계 단계에서의 검증을 통한 품질향상도 기대할 수 있다.

키워드 : 임베디드 시스템, 재사용, 디바이스 드라이버, 템플릿

A Technique of Reusable and Reliable Device Driver Development For Embedded-Linux System

Jaehyun Choi[†] · Woojin Lee[††] · Kiwon Chon[†††]

ABSTRACT

According to the development of embedded systems, they are used in various fields. However, the productivity and the quality of embedded systems are not acceptable because of the hardware environment-oriented development. Severalworks has proposed new techniques using a formal specification method or complex template, but due to high complexity and low flexibility of the method, it is difficult to be used. Accordingly, this paper presents a new approach to develop embedded device drivers which are kinds of embedded system software. In our approach, we define a light device driver template for ensuring high flexibility and productivity and generate the device driver by incorporating with embedded system information represented by XML. By using XML, we consider the design phase in device driver development process for the quality of the driver. Our approach, improves the reusability of embedded device driver and finally, it will improve productivity and quality of embedded system.

Key Words : Embedded System, Reuse, Device Driver, Template

1. 서 론

산업용 기기를 제어하기 위해 사용되던 임베디드 시스템은 공장 자동화 및 가정 자동화에 필요한 자동제어 시스템을 비롯해 각종 디지털 정보가전 기기, 자동 센서장비 등의 기능을 다양화하고 부가가치를 높일 수 있는 핵심 인프라로 자리매김 하고 있다. 최근에는 임베디드 시스템의 멀티미디어 처리, 다중작업 및 실시간 처리 능력이 강화되고 유무선 통신 및 네트워크와의 접목으로 전통적인 제조, 유통, 금융, 서비스 산업뿐만 아니라 항공, 우주, 국방, 의료, 멀티미디어 통신 및 에너지 개발 등의 첨단 분야에 이르기까지 그 사용 범위와 영향력이 점점 커지고 있다[1].

하지만, 이러한 임베디드 시스템에서 동작하는 소프트웨어는 범용으로 개발되기 보다는 특정 임베디드 시스템 환경을 대상으로 하여 개발되고, 자체 개발능력을 갖춘 수백 개의 소형 업체들에 의해 독자적으로 개발되기 때문에 동일 기능을 수행하는 임베디드 소프트웨어라 할지라도 특정 임베디드 시스템 환경에 따라 별도로 제작되어야 하는 어려움

※ 본 연구는 숭실대학교 교내연구비 지원으로 이루어졌음.
† 준 회 원 : 숭실대학교 컴퓨터학과 대학원 석사과정
†† 준 회 원 : 숭실대학교 컴퓨터학과 대학원 박사과정
††† 종신회원 : 숭실대학교 컴퓨터학부 교수
논문접수 : 2005년 6월 10일, 심사완료 : 2005년 11월 15일

이 있다. 이러한 상황은 임베디드 소프트웨어의 개발생산성을 떨어뜨리고 소프트웨어에 대한 유지보수성 및 품질 저하에 큰 영향을 끼치고 있다. 특히, 임베디드 소프트웨어인 디바이스 드라이버는 임베디드 시스템 환경상의 특정 하드웨어의 운용을 위한 제어 소프트웨어로써 동일 장치를 대상으로 동일 기능의 제공을 목적으로 개발되지만 대상 임베디드 환경에 따라 별도로 개발되고 있어 임베디드 시스템의 개발생산성 저하의 주요 원인이 되고 있다. 기존의 연구들에서 이를 해결하기 위해 별도의 인터페이스 명세언어나 복잡한 템플릿을 이용하여 디바이스 드라이버를 자동으로 생성하는 연구들이 진행되고 있지만 명세기법이 높은 복잡도를 가지거나 유연성의 부족으로 인해 디바이스 드라이버를 완전히 표현하기에는 다소 부족한 점이 있다.

따라서 본 논문에서는, 임베디드 시스템 개발의 총체적인 개발생산성 및 품질 향상을 위해 리눅스 기반의 임베디드 시스템에서 디바이스 드라이버의 개발생산성 향상 및 사용 편리성과 유연성을 고려한 임베디드 디바이스 드라이버 개발 방법을 제안한다. 이는 상이한 임베디드 시스템 환경상에 존재하는 동일 하드웨어의 운용을 위한 디바이스 드라이버 개발을 위한 디바이스 드라이버 템플릿을 정의하고 여기에 XML로 표현되는 특정 임베디드 시스템 정보를 결합하고 검증함으로써 목적 환경에서 동작 가능한 디바이스 드라이버를 자동으로 생성하게 한다.

2. 관련 연구

2.1 디바이스 드라이버

장치 제어기 또는 구동 드라이버로서 정의 된다. 하드웨어와 운영체제 응용프로그램의 연결 고리가 되는 프로그램으로 하드웨어 구성 요소가 운영체제하에서 작동하기 위해 필요한 모듈이다. 이러한 모듈은 근본적으로 특권(Privilege)층에서 실행되고, 메모리에 상주하며 저급 하드웨어 처리 루틴을 가진 공유 라이브러리이다[2]. 이러한 디바이스 드라이버는 하드웨어 디바이스의 통신을 통해 디바이스를 제어하며 이 때 디바이스 드라이버는 하드웨어의 레지스터에 명령어를 전달하고 하드웨어는 그 명령어를 해석하여 특정 작업을 수행하게 된다. 여기에서는 사용되는 레지스터의 지정방식에 따라 디바이스 드라이버의 입출력 방식이 2가지로 분류된다.

▪ I/O 맵 전달

입출력을 위한 특정한 명령어를 지정하고는 이 명령어를 이용하여 하드웨어 레지스터에 데이터를 쓰거나 하드웨어 레지스터로부터 데이터를 읽는 방식이다. 인텔의 CPU들이 이 정책을 쓰고 있다. 단점으로는 입출력 명령어는 메모리에 대한 명령어보다는 단순하므로 대량의 데이터를 빠르게 전송하지 못한다는 점이다. 유닉스에서는 커널만이 입출력 명령어를 사용할 수 있는데, 이것은 응용 프로그램에서 직접 하드웨어를 제어하는 경우에 발생할 수 있는 보안 문제를 고려한 것이다.

▪ 메모리 맵 전달

메모리의 특정 번지를 입출력을 위하여 유보해 두고 이 부분에 데이터를 쓰고/읽으면 하드웨어 레지스터에 읽고/쓰는 방식이다. 모토롤라의 CPU 들이 이러한 방식을 취한다. 특정한 입출력 명령이 존재하지 않고 메모리에 읽고 쓰는 load/store 명령으로 하드웨어에 대한 입출력이 가능한 장점이 있다. 이러한 메모리 맵 전달 방식은 구현자체가 간단하고 입출력을 빠르게 수행할 수 있다. 따라서, 일반적으로 제한된 자원을 가지고 빠른 응답시간을 요구하는 임베디드 시스템에 가장 적합한 방식으로 임베디드 시스템의 구현에 많이 사용되고 있는 방식이다.

본 논문에서, 제시하고 있는 디바이스 드라이버 개발 기법은 이러한 메모리 맵 전달방식의 디바이스 통신을 하고 있는 임베디드 시스템을 대상으로 한다.

2.2 디바이스 드라이버의 개발

최근 유비쿼터스 사회로의 진전이 이루어지면서 임베디드 시스템의 구축 및 개발에 관한 연구가 활발히 진행되고 있다. 하지만, 이러한 임베디드 시스템의 개발은 특정 하드웨어 플랫폼에 종속적으로 개발되고 있어 개발생산성 및 효율성에 다소 문제점이 있다. 이에 따라, 임베디드 시스템 개발생산성 및 효율성을 개선을 위한 많은 연구가 요구되고 있고 또한 실제로 이 부분에 있어서 많은 연구가 수행되고 있다. 이러한 연구들은 대부분 하드웨어 독립적인 부분과 종속적인 부분을 구분하고 각각을 표현하기 위한 방식을 정의함으로써 편리한 디바이스 드라이버의 개발을 목표로 하고 있다. Devil+ 언어를 이용한 디바이스 드라이버의 개발[3-6]은 임베디드 시스템의 개발을 위해 Devil+ 언어로 디바이스 드라이버를 명세하고 별도의 Devil+ 언어 컴파일러인 Delta Devil+ 를 통해 디바이스 드라이버를 생성한다. 하드웨어 인터페이스 정의 언어인 Devil 을 바탕으로 하고 있는 이러한 개발방식은, 임베디드 디바이스 드라이버의 개발에 있어서 외부적인 언어를 사용한다는 점에 있어서 기존의 임베디드 시스템 개발자들에게 별도의 교육이나 추가적인 노력이 있어야 하는 단점이 있어 실제로 적용되기에는 어려운 점이 있다. 또한, 디바이스 드라이버의 개발이 Devil 언어에 의존적이므로 Devil 언어로 표현될 수 없는 복잡한 임베디드 시스템 개발 시 시스템 적용이 불가능한 점도 단점으로 지적된다.

디바이스를 정규 명세 기법을 통해 명세하고 이를 통해 디바이스 드라이버를 개발하는 기법[7, 8]들은 템플릿이나 기타 명세언어를 통해 디바이스를 표현하고, 이와 별도로 명세되는 하드웨어 정보를 통해 디바이스 드라이버를 생성한다. 하지만 이러한 기법들은 역시, 명세기법의 복잡성으로 인해 실제로 적용하기에 어려운 점이 있고, 또한 생성에만 초점이 맞추어져 있어 올바른 디바이스 드라이버 생성을 위한 설계단계가 고려되지 않고 있다. 디바이스 드라이버는 하드웨어와 밀접한 관련을 지닌 소프트웨어 이므로, 사실상 자동으로 생성된 디바이스 드라이버가 올바르게 동작한다는 것을 보장하기는 힘들다. 따라서, 본 연구에서는 우선, 생성된 코드 및 명세 기법을 최대한 소스레벨로 끌어내림으로

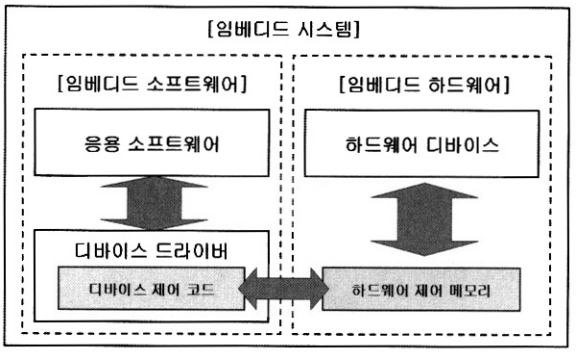
해서 디바이스 드라이버를 템플릿을 사용하여 정의하는 과정에서 개발자가 편리하고 유연하게 사용할 수 있도록 하였으며, 동시에 이러한 템플릿 명세기법의 복잡도를 최소화함으로 인해 추가적인 노력의 소요를 방지하여 개발과정에서 쉽게 적용할 수 있도록 하였다. 또한 이렇게 템플릿으로 표현된 디바이스 드라이버와 하드웨어 정보와의 결합과정에 있어서도 RT-UML, Petri net과 같은 모델링 언어가 사용될 수 있는 설계 단계와의 호환성을 고려해 XML 을 활용하였다. 이렇게 디바이스 드라이버 개발에 있어서 XML 문서를 사용하는 것은 추후에 임베디드 시스템 개발과정에서 RT-UML과 같은 모델링 언어를 사용해 시스템을 모델링 하고 이를 바탕으로 디바이스 드라이버를 생성하는 것을 가능하게 한다. 그리고 이러한 디바이스 드라이버 개발과정에서 설계 단계를 고려하는 것은 설계 단계에서의 모델 검증을 가능하게 하고 또한 시뮬레이션 또는 래피드 프로토타이핑과 같은 시험기법을 쉽게 적용할 수 있어 임베디드 시스템 전체의 개발생산성 및 품질을 향상 시킬 수 있다. 따라서 본 연구에서는 이러한 임베디드 시스템 개발 프레임워크 중에서 XML 정보로 표현된 하드웨어 정보와 디바이스 템플릿을 사용하여 재사용 가능한 디바이스 드라이버를 생성하도록 하는 방안의 제시를 목적으로 하며 추후에 이러한 임베디드 시스템 정보를 RT-UML과 같은 모델링 언어를 사용한 모델로부터 추출하는 기법에 대한 연구를 진행할 것이다.

3. 재사용성 및 신뢰성을 고려한 리눅스 기반 임베디드 디바이스 드라이버 개발 기법

3.1 디바이스 드라이버 템플릿

메모리 맵 전달 방식을 사용하는 임베디드 시스템상의 디바이스 드라이버에는 특정 하드웨어에 제어명령을 전달하기 위해 동작하는 제어코드가 존재한다. 이 제어코드는 실제 임베디드 시스템 환경에서 특정 하드웨어 제어를 위해 사전에 할당되어 있는 메모리 주소에 제어명령을 전달함으로써 하드웨어의 제어를 수행한다. (그림 1)은 이러한 과정을 표현하고 있다.

즉 디바이스 드라이버는, 대상 하드웨어를 모니터링 하기 위해, 해당 주소에 있는 데이터를 참조하고 이를 제어하기 위해서 해당 주소에 특정 데이터를 전달한다. 그러나, 동일한 하드웨어라 할지라도 이를 제어하기 위한 메모리 주소는 임베디드 시스템 환경에 따라 차이가 존재하기 때문에 그러한 환경에 따라 디바이스 드라이버를 재 작성해야 하는 문제점이 존재한다. 따라서 본 논문에서는 이를 해결하기 위해 가상 참조자를 사용하는 디바이스 드라이버 템플릿을 정의하고 사용하는 개발 기법을 제안한다. 여기서 사용되는 가상 참조자는 디바이스 드라이버의 정의에 있어서 하드웨어 종속적인 정보를 표현하는 것으로 XML로 표현된 임베디드 시스템 정보와의 결합과정에서 실제 하드웨어 정보로 변환되게 된다. 즉 임베디드 디바이스 드라이버를, 템플릿을 사용하여 임베디드 시스템 독립적인 형태로 표현하고 임베디드 시스템 종속적인 정보와의 결합을 통해 실제적으로 동작 가능한 디바이스 드라이버를 생성하도록 하는 것이다.



(그림 1) 디바이스 드라이버의 하드웨어 제어

본 연구에서 정의된 템플릿은 기존의 연구에서 정의되었던 템플릿과는 달리 임베디드 개발자가 실제에 가장 쉽고 빠르게 적용 가능하도록 소스 레벨의 가장 가벼운 템플릿으로 정의하였다. 이것은 명세기법의 복잡도나 불완전성이 임베디드 시스템 개발생산성이나 품질에 영향을 최소화하기 위한 고려이다. 또한, 이것은 임베디드 시스템이 실시간 시스템으로 특히 타이밍(Timing)이나 예외처리에 있어서 높은 복잡성을 가지기 때문에 이러한 복잡성을 개발자가 세밀하게 다룰 수 있도록 하기 위한 고려이기도 하다. 디바이스 드라이버 명세를 위한 템플릿 규칙은 <표 1>과 같다.

〈표 1〉 디바이스 드라이버 템플릿 규칙

예약어	사용목적
$[DeviceName]$	제어할 장치의 이름을 기술한다. 디바이스를 명시하기 위해 사용한다.
$[DriverName]$	생성되는 디바이스 드라이버의 이름을 기술한다. 생성되는 소스코드의 파일명 및 구분을 위해 사용한다.
$[HeaderFiles]$	필요한 헤더파일명들을 나열한다. 드라이버 구현에 필요한 헤더파일들을 명시하기 위해 사용한다.
$[Defines]$	상수 선언들을 나열한다. 드라이버 구현에 사용될 상수들을 선언하기 위해 사용한다.
$[External References]$	드라이버에 필요한 외부참조들을 정의한다. 드라이버 구현을 위해 필요한 외부 변수나 참조자 들을 명시하기 위해 사용한다.
$[Memory References]$	메모리 정보를 위한 가상 참조들을 정의한다. 이후에 XML로 표현된 임베디드 시스템 정보에 의해 실제 정보로 변환된다.
$[GlobalVariables]$	드라이버 구현에 필요한 전역변수를 정의한다.
$[Initialize Module]$	디바이스 드라이버 초기화 시에 필요한 작업을 정의한다. 디바이스 드라이버의 커널 적재 시에 필요한 작업들을 위해 사용한다.
$[CleanupModule]$	디바이스 드라이버의 제거시의 작업을 정의한다. 디바이스 드라이버가 커널에서 제거될 시에 필요한 작업들을 명시하기 위해 사용한다.
$[InterfacesDefine]$	디바이스 드라이버에 구현될 인터페이스 및 함수 매핑을 기술한다. 디바이스 드라이버의 입출력 담당을 위한 함수 선언과 매핑을 정의하기 위해 사용한다.

〈표 1〉 디바이스 드라이버 템플릿 규칙(계속)

예약어	사용목적
$[Func]$[argument][return]	함수를 정의한다. 필요한 매개변수와 반환값이 정의된다. 이것은, 디바이스 드라이버의 기능을 수행하기 위한 함수들을 정의하기 위해 사용한다. 정의된 함수의 참조를 위해서는 $$Func$$ 형태의 참 지시문을 사용한다.

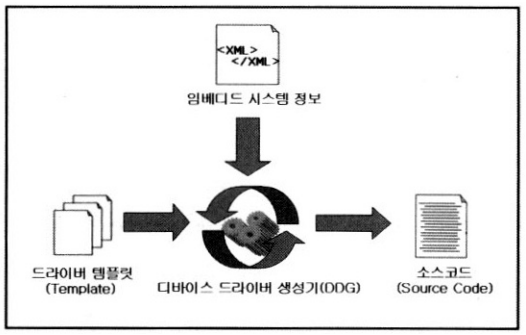
3.2 디바이스 드라이버의 생성

디바이스 드라이버 템플릿으로 표현된 디바이스 드라이버는 각각의 특정 임베디드 시스템 정보와의 결합을 통하여 실제적인 디바이스 드라이버로 생성된다. (그림 2)는 이러한 디바이스 드라이버의 생성과정을 표현하고 있다.

템플릿 형태로 정의된 디바이스 드라이버는, 디바이스 드라이버 생성기(DDG)에 의해 분석되고 임베디드 시스템 정보와 결합되어 실제 디바이스 드라이버의 소스코드로 변환된다. 이러한 임베디드 시스템 정보는, 임베디드 시스템의 내부적인 메모리 정보나 시스템정보를 담고 있는 것으로 XML문서 형태로 표현된다. 이러한 임베디드 시스템정보는 본 연구에서는 개발자가 직접 정의하여야 하나 추후 연구를 통해서는 RT-UML과 같은 모델링 언어로 설계된 모델에서 자동 생성되도록 한다.

(그림 3)는 임베디드 시스템 정보를 위한 XML 문서의 스키마이다.

임베디드 시스템 정보를 위한 XML 스키마는 Embedded System 태그를 루트 원소로써 가지며 디바이스 드라이버의 변환을 위한 정보를 포함하고 있다. 현재의 XML 스키마에는 디바이스 드라이버에서 사용되는 정보만을 표현하기 위한 스키마만 정의되어있다. 이러한 제어 메모리 정보를 표현하기 위한 스키마에는 디바이스 이름과 대상 임베디드 시스템 상에서의 디바이스 드라이버가 사용하는 제어 메모리 주소 및 접근권한이 명시된다. 메모리 주소는 0xFFFFFFFF와 같은 16진수 표현으로 명시되며, 메모리 참조자는 C언어의 변수 선언 규칙과 동일하게 정의된다. <표 2>는 디바이스 드라이버 템플릿에서 디바이스 제어를 위한 제어 메모리 정보를 제공하기 위한 XML 태그와 그에 대한 설명을 보여준다.



(그림 2) 디바이스 드라이버의 생성

〈표 2〉 디바이스 제어 메모리 매핑을 위한 XML 태그

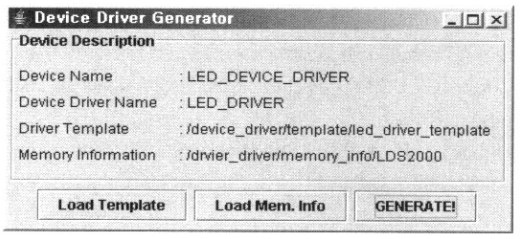
태그	설명
<reference>	템플릿에 정의된 메모리 참조자의 이름을 명시하기 위해 사용한다. 메모리 참조자는 C언어의 변수선언 규칙과 동일한 형식으로 정의된다. <reference> LED_ADDRESS_REF </reference>
<access>	메모리 참조자의 메모리 접근 권한을 명시하기 위해 사용한다. <access> ReadWrite </access>
<address>	템플릿에 정의된 메모리 참조자와 실제 임베디드 시스템에 매핑할 주소를 명시하기 위해 사용한다. 0xFF4E1035와 같은 16진수 형태의 4바이트 표기가 사용된다.<address>0x08300000</address>

```
<?xml version="1.0" encoding="euc-kr"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:simpleType name="accessType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="ReadOnly"/>
            <xsd:enumeration value="WriteOnly"/>
            <xsd:enumeration value="ReadWrite"/>
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType name="nameType">
        <xsd:restriction base="xsd:string">
            <xsd:pattern value="[A-Za-z_][A-Za-z0-9_]+"/>
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType name="addressType">
        <xsd:restriction base="xsd:string">
            <xsd:pattern value="0x[0-9A-Fa-f]{8}"/>
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:group name="MemoryMapping">
        <xsd:sequence>
            <xsd:element name="reference" type="nameType"/>
            <xsd:element name="access" type="accessType"/>
            <xsd:element name="address" type="addressType"/>
        </xsd:sequence>
    </xsd:group>
    <xsd:complexType name="DeviceDriverMapping">
        <xsd:sequence>
            <xsd:element name="DeviceName" minOccurs="1" maxOccurs="1" type="nameType"/>
            <xsd:group ref="MemoryMapping" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="SystemInfo">
        <xsd:all>
            <xsd:element name="DeviceMapping" type="DeviceDriverMapping" minOccurs="0" maxOccurs="1"/>
        </xsd:all>
    </xsd:complexType>
    <xsd:element name="EmbeddedSystem" type="SystemInfo"/>
</xsd:schema>
```

(그림 3) 임베디드 시스템 정보를 위한 XML 스키마

4. 구현 및 적용

본 논문에서 제시한 디바이스 드라이버 개발 기법을 적용하여 LED 디바이스 드라이버를 위한 템플릿을 정의한 후, 디바이스 드라이버 생성기를 이용하여 해당 디바이스 드라이버를 생성하였다. 디바이스 드라이버 생성기는 자바환경(J2SDK 1.4.2_08)에서 구현되었으며 (그림 4)은 구현된 디바이스 드라이버 생성기의 사용자 인터페이스를 나타낸다.



(그림 4) 디바이스 드라이버 생성기의 사용자 인터페이스

(그림 5)은 테스트에 사용된 LED 디바이스 드라이버 템플릿의 일부이다. 먼저, 대상 디바이스의 이름과 드라이버 이름을 정의하고 메모리 참조자와 초기화 함수 및 LED 장치를 위한 메모리 매핑 함수를 정의하고 있다. LED 장치를 위한 매핑 함수에서는 앞서 정의된 가상 참조자가 사용되고 있으며 이 함수를 통하여 임베디스 시스템 내의 LED 장치에게 제어 명령을 전달하거나 해당 장치를 모니터링 하는 메모리 주소를 프로그램에서 사용할 수 있게 된다.

(그림 5)의 템플릿을 사용하여 시험용 임베디드 장비인 Corebell 사의 LDS2000[10]과 Hybus 사의 X-Hyper255B [11]의 LED 디바이스 드라이버를 생성하였다. 이 과정에서 사용된 임베디드 시스템에 대한 정보는 (그림 6) 및 (그림 7)과 같다.

```
$[DeviceName]$
LED_DEVICE_DRIVER

$[DriverName]$
LED_DRIVER

$[HeaderFiles]$
<linux/kernel.h>
<linux/module.h>
<linux/modversions.h>
        ...

$[MemoryReferences]$
@LED_DEVICE_ADDRESS@
        ...

$[InitializeModule]$
int error;
if ((major = register_chrdev(0, DEVICE_NAME, &led_fops)) < 0) {
   printk("error : initialize failed");
   return major;
}
printk("initialized : major number = %dWn", major);
        ...

$[InterfaceDefine]$
struct file_operations led_fops = {
    open    : $$led_open$$,
    release : $$led_release$$,
    write   : $$led_write$$,
};
        ...

$[led_memory_mapping]$[][int]
led_ptr  = ioremap((unsigned int)@LED_DEVICE_ADDRESS@,(unsigned int)(1024));
return 0;
        ...
```

(그림 5) 템플릿으로 표현된 LED 디바이스 드라이버

```
<?xml version="1.0" encoding="UTF-8"?>
<EmbeddedSystem xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="EmbeddedSystem.xsd">
    <SystemName>LDS2000</SystemName>
    <DeviceMapping>
        <DeviceName>LED_DEVICE_DRIVER</DeviceName>
        <reference>LED_DEVICE_ADDRESS</reference>
        <access>ReadWrite</access>
        <address>0x08000008</address>
    </DeviceMapping>
</EmbeddedSystem>
```

(그림 6) LDS2000을 위한 임베디드 시스템 정보

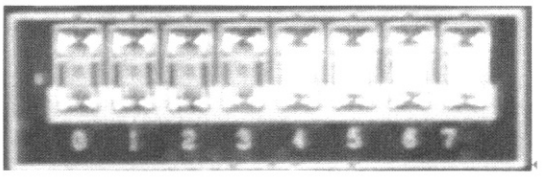
```
<?xml version="1.0" encoding="UTF-8"?>
<EmbeddedSystem xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="EmbeddedSystem.xsd">
    <SystemName>X_Hyper255B</SystemName>
    <DeviceMapping>
        <DeviceName>LED_DEVICE_DRIVER</DeviceName>
        <reference>LED_DEVICE_ADDRESS</reference>
        <access>ReadWrite</access>
        <address>0x08300000</address>
    </DeviceMapping>
</EmbeddedSystem>
```

(그림 7) X-Hyper255B를 위한 임베디드 시스템 정보

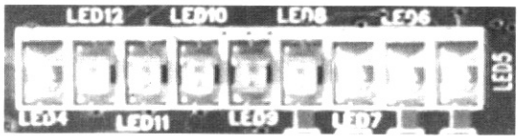
템플릿과 제어 메모리 정보를 이용하여 생성된 디바이스 드라이버 소스 코드는 CoreBell LDS2000과 Hybus X-Hyper255B에서 제공한 각각의 크로스 컴파일러에 의해 컴파일 된 후, 각각의 시스템에 탑재되어 있는 LED에 대한 올바른 제어를 수행하였다. (그림 8)은 Hybus X-Hyper 255B에 대해 생성된 소스 코드의 일부이며 (그림 9)과 (그림 10)는 생성된 소스코드를 사용한 실제 LED 디바이스의 동작을 보여주고 있다.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
        ...

#define DEVICE_NAME  "LED_DEVICE_DRIVER"
        ...

static int     LED_open(struct inode *inode,struct file *filp);
static int     LED_release(struct inode *inode,struct file *filp);
static ssize_t LED_write(struct file *filp,const char *buffer,size_t length,loff_t *offset);
        ...

int init_module(void)
{
   int error;
   if ((major = register_chrdev(0, DEVICE_NAME, &led_fops)) < 0) {
      printk("error : initialize failed");
      return major;
   }
   printk("initialized : major number = %dWn", major);
}
        ...

struct file_operations led_fops = {
    open    : led_open,
    release : led_release,
    write   : led_write,
};
        ...

int LED_memory_mapping()
{
  LED_PTR  = ioremap((unsigned int)0x08300000,(unsigned int)(1024));
  return 0;
}
        ...
```

(그림 8) 생성된 디바이스 드라이버 소스 코드



(그림 9) Corebell LDS2000의 LED 동작



(그림10) Hybus X-Hyper255B의 LED 동작

5. 결론 및 향후 연구

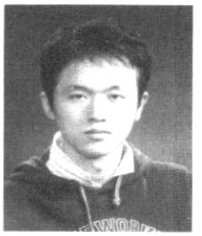
본 논문에서는, 서로 다른 임베디드 플랫폼에서 동일 장치의 제어를 목적으로 하는 디바이스 드라이버를 템플릿을 이용하여 정의하고, 여기에 XML기반의 제어 메모리 정보를 결합하여 특정 임베디드 시스템 환경에서 동작하는 디바이스 드라이버를 구현하는 방법을 제시하였다. 그리고 이것을, 두 개의 상이한 임베디드 시스템 환경에 대하여 적용하여 검증하였다.

템플릿을 이용한 디바이스 드라이버 개발 기법은 여러 임베디드 시스템 상의 동일 디바이스에 대한 디바이스 드라이버를 템플릿을 통해 시스템 독립적으로 표현하고 개발하도록 함으로써, 임베디드 시스템 구성에 따른 디바이스 드라이버의 중복 개발을 방지하여 개발생산성을 향상시킨다. 뿐만 아니라, 시스템 독립적으로 표현된 템플릿을 통해 여러 임베디드 시스템상의 모든 디바이스 드라이버를 편리하게 관리 할 수 있게 하여, 디바이스 드라이버의 높은 유지보수성을 보장한다.

하지만, 보다 완전한 디바이스 드라이버의 개발을 위해서는 디바이스 드라이버의 신뢰성과 같은 품질 문제 또한 고려되어야 한다. 따라서, 추후 연구에서는, 본 연구에서 제시한 템플릿 기반의 개발 기법과 RT-UML 및 Petri net과 같은 모델링 기법을 접목하여 디바이스 드라이버의 설계에서부터 개발에 이르는 과정을 체계적으로 정의 및 구현할 것이다. 이를 위해 템플릿 기반의 개발 기법에서 사용한 XML 기반의 임베디드 시스템 정보는 모델링 언어로 표현된 디바이스 드라이버 모델로부터 추출되고 사용될 것이며 그 전에 만들어진 모델을 설계 단계에서 시험하고 검증함으로써 디바이스 드라이버의 신뢰성 및 품질향상을 가져올 수 있다.

참 고 문 헌

[1] VDC, The Embedded Software Strategic Market Intelligence Program 2002/2003, Volume II, 2003.
[2] Alessandro Rubini, "Linux Device Drivers," O'Reilly, 2000
[3] Fabrice M´erillon, Laurent R´eveill'ere, Charles Consel, Renaud Marlet, Gilles Muller, "Devil: An IDL for Hardware Programming," In OSDI 2000, pages 17-30, San Diego, October, 2000.
[4] Laurent R´eveill'ere, F. M´erillon, C. Consel, R. Marlet, and G. Muller, "The Devil Language release 0.4,". August 24, 2000.
[5] Lei Luo, Ming-Yuan Zhu, Qing-Li Zhang, "A formal semantic definition of DEVIL," SIGPLAN Notices 38(4) : 47-56 (2003)
[6] Yingxi Yu, Mingyuan Zhu, Shuoying Chen, "New Approach for Device Driver Development - Devil+ Language," Lecture Notes in Computer Science, Volume 3605/2005, Page(s): 418
[7] Katayama, T.; Saisho, K.; Fukuda, A, "Prototype of the device driver generation system for UNIX-like operating systems," Principles of Software Evolution, 2000. Proceedings. International Symposium on 1-2 Nov 2000 Page(s):302-310]
[8] Tetsuro Katayama, Keizo Saisho, and Akira Fukuda, "A Method for Automatic Generation of Device Drivers with a Formal Specification Language," Proc. Int. Workshop on Principles of Software Evolution, pp.183-187, Apr., 1998.
[9] http://idaemon.com.ne.kr/Linux/Linux_Kernel/tlk8.html
[10] Corebell, LDS2000 Specification, http://www.corebell.com/product/01.asp
[10] Hybus, X-Hyper255B Specification, http://www.hybus.net/sub02/sub02_xscale02.asp



최 재 현

e-mail : uniker80@empal.com
2004년 숭실대학교 컴퓨터학부(공학사)
2004년~현재 숭실대학교 대학원 컴퓨터학과 석사과정
관심분야 : 임베디드 시스템, RFID/USN, 유비쿼터스 컴퓨팅, 소프트웨어 테스팅, Persistent Software Attributes

이 우 진

e-mail : bluewj@empal.com
2000년 숭실대학교 컴퓨터학부(공학사)
2002년 숭실대학교 대학원 컴퓨터학과 (공학석사)
2002년~현재 숭실대학교 대학원 컴퓨터학과 박사과정
관심분야 : 웹 어플리케이션, 웹 서비스, 유비쿼터스 컴퓨팅, 모바일 컴퓨팅, 홈 네트워크, 임베디드 시스템

정 기 원

e-mail : chong@ssu.ac.kr
1967년 서울대학교 전기공학과(공학사)
1981년 미국 알라바마주립대(헌츠빌) 전산학과(석사)
1983년 미국 텍사스주립대(알링턴) 전산학과(박사)
1971년~1975년 한국과학기술연구소 연구원
1975년~1990년 국방과학연구소 책임연구원
2002년~2003년 한국전자거래학회 회장
1990년~현재 숭실대학교 컴퓨터학부 교수
2001년~현재 IT감리포럼 회장
관심분야 : 소프트웨어공학, 소프트웨어프로세스, 정보시스템감리, 전자거래(CALS/EC), 유비쿼터스 컴퓨팅