템플릿을 이용한 디바이스 드라이버 자동생성 시스템 설계

준회원 김 현 철*, 정회원 이 서 훈*, 황 선 영*

Design of an Automatic Generation System of Device Drivers Using Templates

Hyoun-Chul Kim* *Associate Member*,
Ser-Hoon Lee*, Sun-Young Hwang* *Regular Members*

요 약

어플리케이션에 맞춤화된 다양한 임베디드 시스템은 리소스의 효율적인 관리를 위해 임베디드 OS의 종류와 버전에 맞는 디바이스 드라이버가 요구된다. 본 논문에서는 동일한 OS의 새로운 버전에 대해 확장성이 용이한 디바이스 드라이버 자동생성 시스템을 제시한다. 제안한 시스템에서는 특정 OS 고유의 디바이스 드라이버 구조를 템플릿으로 작성한 후 라이브러리화하며, 라이브러리에 저장된 템플릿을 기본골격으로 하여 시스템의 특성에 따른 코드를 추가하는 방법으로 디바이스 드라이버를 생성한다. 생성된 디바이스 드라이버를 커널에 등록하여 데이터 전송 시간을 비교한 결과 매뉴얼로 설계한 디바이스 드라이버에 비해 자동생성된 TFT-LCD 드라이버, USB 인터페이스 키보드 마우스 드라이버, 그리고 AC'97 컨트롤러 드라이버가 각각 경미한 증가를 보였다. 생성된 드라이버를 커널 컴파일한 후의 코드 사이즈도 각각 경미한 증가를 보였다.

Key Words : Embedded system, Device driver, Automatic generation

ABSTRACT

Applications running under embedded systems require various device drivers designed for different types and versions of the OS to manage resources effectively. In this paper, an automated device driver generator system which can generate the device drivers to be used in newer versions the target OS is proposed. In the proposed system, the structures of device drivers of specific OS are designed in the templates and stored in a library, and the target device drivers are generated by adding codes to the stored templates. Once device drivers are generated, they are registered into the kernel. The experimental results show that data transfer time has been slightly increased when compared against manually created drivers for TFT-LCD driver, USB interface keyboard/mouse driver, and AC'97 controller drivers. The code size for the generated drivers after compilation has also been increased slightly when compared with manually designed device drivers.

※ 본 연구결과물 (예: 논문, 특허, 표준, IP, SoC, IT 시스템 등)은 2008년도 「서울시 산학연 협력사업」의 「나노 IP/SoC 설계기술혁신사업단」의 지원으로 이루어졌습니다. 본 연구에 사용된 CAD tool은 IDEC 프로그램에 의해 지원되었음.
* 서강대학교 전자공학과 CAD & ES. 연구실
논문번호 : KICS2007-12-544, 접수일자 : 2007년 12월 6일, 최종논문접수일자 : 2008년 9월 2일

Ⅰ. 서 론

반도체 산업과 집적회로 기술의 발전으로 모바일 기기에 사용되는 특정 어플리케이션에 최적화된 임베디드 시스템의 개발이 활발히 진행되고 있다. 초기 임베디드 시스템은 어플리케이션이 단순하여 마이크로프로세서와 간단한 펌웨어를 내장하여 수행하도록 하였으나 어플리케이션의 복잡도가 증가함에 따라 임베디드 시스템은 단순히 펌웨어 수준의 소프트웨어가 아니라 하드웨어 리소스 관리, 태스크의 실시간 스케쥴링 및 멀티 태스킹 등을 수행하기 위한 운영체제 기반의 소프트웨어를 요구하게 되었다. 임베디드 시스템은 내장할 수 있는 메모리의 한계로 인해 범용 OS를 사용하지 못하고 저용량의 경량화된 OS를 사용해야 한다. 경량화된 OS는 범용 OS의 커널을 기반으로 임베디드 시스템에서 사용되는 하드웨어를 구동하기 위한 기능을 포함하여 구성된다[1]. 임베디드 시스템은 수행하는 어플리케이션에 특화되어 설계되기 때문에 사용되는 OS의 종류와 버전도 어플리케이션에 따라 다양하게 존재한다. OS는 하드웨어를 효율적으로 컨트롤하기 위해 디바이스 드라이버를 필요로 하며 맞춤화된 다양한 임베디드 OS의 종류와 버전에 맞는 디바이스 드라이버가 요구된다[2]. 디바이스 드라이버는 하드웨어와 OS에 의존적이기 때문에 개발자가 디바이스 드라이버를 개발하기 위해서는 하드웨어와 OS에 대한 정확한 이해를 바탕으로 설계자가 직접 설계해야하므로 시스템 개발에서 많은 개발기간과 비용을 차지한다[3]. 디바이스 드라이버의 설계비용을 감소시키기 위해 임베디드 시스템에서 사용되는 OS에 맞는 디바이스 드라이버 자동생성 시스템에 대한 연구가 진행되어왔다[4]. 디바이스 드라이버 자동생성 시스템에는 한국전자통신연구원에서 개발한 Quick Driver[5], Jungo사의 WinDriver[6], Compuware사의 DriverStudio[7], 그리고 마이크로 소프트에서 개발한 DDK[8] 등이 개발되었다. 기존의 디바이스 드라이버 개발도구들은 하나의 OS만을 지원하며 사용 가능한 디바이스 드라이버가 아닌 디바이스 드라이버의 기본골격만을 생성하므로 다양한 OS에 따라 사용가능한 디바이스 드라이버를 자동생성해 주는 시스템이 요구된다. 디바이스 드라이버 자동생성을 위해 OS의 버전이 향상되더라도 기존의 디바이스 드라이버의 생성 모듈을 재사용하여 향상된 OS의 디바이스 드라이버를 생성한다. 디바이스 드라이버 생성 모듈의 재사용은 소프트웨어 재사용 기법을 사용하여 구현한다[9][10]. 소프트웨어 재사용 기법을 적용하여 하나의 OS에서 사용되어지는 디바이스 드라이버의 기본골격을 OS의 향상된 버전이나 유닉스와 리눅스같은 유사한 계통의 다른 OS에서 재사용하게 함으로써 OS마다 각각 기본골격을 만들어줘야 하는 한계를 극복하였다.

본 논문에서는 동일한 OS의 새로운 버전이나 유사한 계통의 OS에 대해 확장성이 용이한 디바이스 드라이버 자동생성 시스템을 제안한다. 형식언어(Formal Specification Language)의 개념을 도입하여 OS 고유의 디바이스 드라이버 구조를 형식언어로 기술한 템플릿으로 작성한 후 라이브러리화한다. 여러 OS에서 사용되는 동일한 기능을 갖는 하드웨어의 디바이스 드라이버 자동생성을 지원하기 위해 라이브러리에 저장된 디바이스 드라이버의 기본골격을 다른 OS에서 재사용하여 디바이스 드라이버를 생성한다. 본 논문의 Ⅱ절에서는 디바이스 드라이버 개발 시스템에 대한 관련연구에 대해 설명하고, Ⅲ절에서는 제안된 디바이스 드라이버 자동생성 시스템에 대해 설명한다. Ⅳ절에서는 생성된 디바이스 드라이버의 실행 화면을 보이고 자동 생성된 디바이스 드라이버와 매뉴얼 설계한 디바이스 드라이버의 성능 비교를 보인다. 마지막으로 Ⅴ절에서는 결과 및 추후 과제를 제시한다.

Ⅱ. 관련연구

디바이스 드라이버의 개발은 OS와 하드웨어의 이해가 선행되어야 하며 설계와 OS에 포팅하는 작업의 어려움으로 인해 시스템 설계 시 많은 설계비용이 소요되는 작업이다[11][12]. 뿐만 아니라, 디바이스 드라이버는 시스템에 사용되는 OS에 따라 재설계해야 하는 오버헤드를 가진다. 그림 1은 기존의

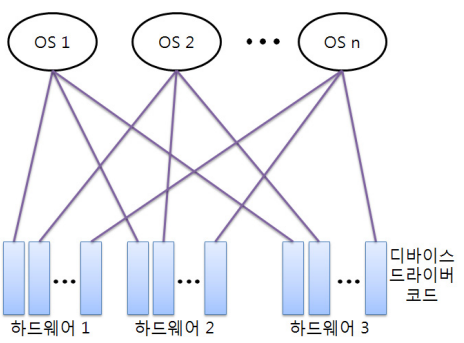

그림 1. 기존의 디바이스 드라이버 개발 모델

디바이스 드라이버 개발 모델을 보인다.

그림 1에서 볼 수 있듯이 하나의 하드웨어가 여러 OS에서 사용되는 경우 사용되는 OS에 따라 디바이스 드라이버를 각각 제작해야하며 동일한 기능이라도 하드웨어가 다르면 각기 다른 디바이스 드라이버를 제작해야 한다. 이러한 특성으로 인해 디바이스 드라이버의 설계는 개발기간과 비용이 많이 요구되므로 설계비용의 감소를 위해 디바이스 드라이버 자동생성기법이 연구되어왔다[5]-[8]. Quick Driver는 리눅스 기반의 디바이스 드라이버 개발을 지원하는 툴로서 디바이스 동작특성인 버스 타입, 캐릭터 타입, 블록타입 그리고 네트워크 타입에 따라 타입별 디바이스 드라이버가 유사한 패턴을 갖고 있다는 점에 착안하여 개발되었다[5][13]. 디바이스 드라이버에서 필요로 하는 주 넘버, 파일 연산 정보 같은 하드웨어 기본정보와 인터페이스 회로 정보, 인터럽트 사용유무, 타이머 사용유무 등의 세부 정보들을 입력하여 디바이스 드라이버의 기본골격 소스를 생성한 후, 기본골격으로 구성된 불완전한 디바이스 드라이버 코드의 안정화를 위해 소프트웨어 테스트와 정적 검증기능을 지원한다. 디바이스 드라이버를 자동생성 할 수 있는 시스템이지만 현재 지원되는 OS는 리눅스 OS에 한정되어 있다. Jungo사에서 개발하여 보급하고 있는 WinDriver[6]는 현재 사용되는 대부분의 OS 종류에 따라 프로그램을 개발하여 보급하고 있으며 PCI, PCMCIA, ISA, ISA PnP, EISA, CompactPCI 등의 버스 구조와 USB용 디바이스 드라이버 자동생성을 지원한다. WinDriver에서는 별도의 GUI를 제공하지 않고 제공하는 컴파일러 중에서 개발자가 원하는 컴파일러를 선택하여 디바이스 드라이버를 자동생성하고 컴파일러와 연동한다. WinDriver는 OS종류와 버전별로 디바이스 드라이버를 각각 개발하여 보급하고 있어 대부분의 OS를 지원하지만 새로운 OS가 개발되었을 경우 해당 OS를 지원하는 디바이스 드라이버 자동생성 시스템 전체를 추가로 개발해야하는 단점을 갖고 있다. DriverStudio[7]는 윈도우용 디바이스 드라이버 자동생성 시스템으로 DriverWorks 컴포넌트 내에 드라이버 코드 라이브러리를 가지고 있어 하드웨어 스펙정보를 입력하면 디바이스 드라이버 코드가 자동생성된다. 뿐만 아니라, 가상 장치 디바이스, 네트워크 인터페이스, 윈도우용 하드웨어 컨트롤러 등을 지원하여 윈도우에서 사용되는 다양한 하드웨어의 디바이스 드라이버 생성과 디버깅 환경을 지원한다. DriverStudio는 윈도우용 디바이스 드라이버 자동생성에 적합한 시스템이나 디바이스 드라이버 자동생성 시 표준 코드 외에 드라이버의 구동을 정의하는데 필요한 코드를 추가해야 하는 오버헤드와 윈도우 이외의 다른 OS는 지원하지 않는 단점이 있다. 마이크로 소프트사에서 개발한 DDK[8]는 윈도우 디바이스 드라이버 개발을 지원하기 위해 윈도우 버전에 맞춰 출시된 윈도우 디바이스 드라이버 생성 시스템이다. DDK는 드라이버 개발에 필요한 컴파일러, 헤더파일 라이브러리, 샘플코드, 그리고 타겟 디버거 같은 유틸리티로 구성된다. DDK는 개발하는 하드웨어와 관련된 샘플코드와 마이크로 소프트의 DDK관련 서적을 참고하여 디바이스 드라이버를 작성해야하므로 개발자가 디바이스 드라이버에 관련된 모든 사항을 이해하고 있어야 한다. 타겟 디버거는 DDK를 통해 디바이스 드라이버를 개발한 후 타겟 OS에 설치하여 디바이스 드라이버를 디버깅할 수 있다. 그러나, DDK는 윈도우에 한정되어 지원되며 타겟 윈도우 버전에 특화되어 있다. 그 외에도 VDM-SL (the Vienna Development Method Specification language)[14][15]을 사용하여 디바이스 드라이버 구조의 추상화 모델을 상위부터 단계별로 모델링하여 작성하는 방법이 있다. VDM-SL을 통해 작성된 디바이스 드라이버 구조는 생성하려는 하드웨어의 구조를 입력하여 기본골격을 자동생성 할 수 있지만, 인터럽트 서비스 루틴과 타이밍 컨트롤러를 지원해주지 못한다[16].

Ⅲ. 디바이스 드라이버 자동생성 시스템

본 절에서는 본 논문에서 제안하는 디바이스 드라이버 자동생성 시스템에 대해 기술한다. 먼저 시스템 개관을 보이고 자동생성기의 전체의 흐름에 따라 각 단계별 수행을 보인다.

3.1 시스템 개관

디바이스 드라이버의 기능은 하드웨어 타입에 따라 결정된다. 타입이 같은 하드웨어는 개발자가 다른 경우에도 디바이스 드라이버의 구조가 비슷하여 기본골격을 이용해 디바이스 드라이버를 제작할 수 있다[17]. OS마다 동일한 기능을 하는 하드웨어의 디바이스 드라이버 작성 방법은 다르지만, 디바이스 드라이버를 작성하는데 필요로 하는 하드웨어 타입 정보, 인터럽트와 타이밍 컨트롤러 사용유무, 버퍼 사용유무, 버스 정보 등의 하드웨어 스펙정보는 크게 다르지 않다[17]. 하드웨어 종류에 따른 기본골격,

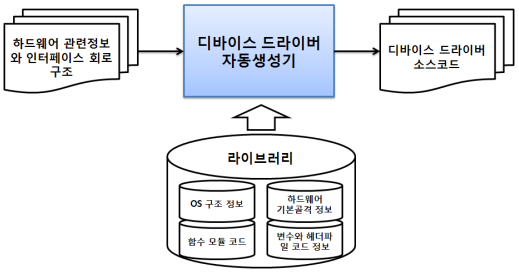

그림 2. 디바이스 드라이버 자동생성 시스템

OS 정보, 하드웨어 스펙정보, 입출력 인터페이스 회로 정보 등을 입력으로 하여 디바이스 드라이버 자동생성 시스템은 OS에 따른 특정 디바이스의 드라이버를 생성한다.

디바이스 드라이버는 OS 종류에 따라 작성 방식이 다르며, 동일한 OS의 경우에도 버전에 따라 디바이스 드라이버에서 사용되는 함수의 형태나 지원되는 헤더파일의 종류가 상이하므로 디바이스 드라이버 자동생성 시스템은 OS의 종류와 버전정보를 입력받는다. 또한, 유사한 동작 형태의 하드웨어 디바이스 드라이버는 기본 골격이 동일하므로 기본 골격을 선택해 주어야 하며 동일한 기능의 하드웨어의 경우에도 하드웨어의 스펙과 인터페이스 회로 구조에 따라 디바이스 드라이버가 달라지므로 하드웨어의 스펙정보와 인터페이스 회로 구조를 입력 파라미터를 통해 입력해 주어야 한다. 그림 2는 입력된 정보를 이용하여 디바이스 드라이버를 자동생성하는 시스템의 구조를 보인다.

본 논문에서 제안하는 디바이스 드라이버 자동생성 시스템은 하드웨어 스펙정보와 인터페이스 회로 구조를 입력으로 받아서 입력된 파라미터에 따라 라이브러리에 저장되어있는 정보를 기반으로 디바이스 드라이버 소스코드를 자동생성한다.

3.2 자동생성기 흐름도

본 절에서는 본 논문에서 제안한 디바이스 드라이버 자동생성기의 내부 흐름도를 보인다. 그림 3은 제안한 디바이스 드라이버 자동생성기의 흐름을 보인다.

디바이스 드라이버 자동생성기는 GUI로 구성된 입력기를 통해 입력 파라미터를 입력받는다. 입력된 파라미터는 자동생성기 내부에서 이용할 수 있도록 변환되고 변환된 데이터를 통해 라이브러리에서 OS와 하드웨어의 종류에 맞는 기본골격코드가 선택된다. 기본골격이 선택되면 기본골격코드에 추가되는

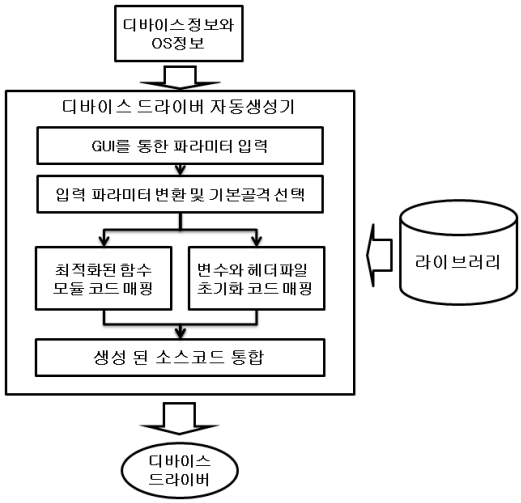

그림 3. 디바이스 드라이버 자동생성기 흐름도

함수 모듈 코드와 변수, 헤더파일 초기화 코드가 라이브러리를 참조하여 생성된다. 생성된 소스코드들은 디바이스 드라이버 양식에 맞게 통합되어 사용가능한 디바이스 드라이버 코드로 출력된다.

3.3 입력 파라미터

디바이스 드라이버는 동일한 하드웨어의 경우 특정 OS에서 설계자가 상이하여도 형태의 유사성이 있으며, 동일 기능 갖는 상이한 하드웨어의 경우 데이터 입출력 및 인터페이스를 담당하는 부분을 제외한 기본 구조는 유사성을 가진다. 디바이스 드라이버 자동생성 시스템은 디바이스 동작의 유사성을 기반으로 기본 골격을 선택한 후, 하드웨어 정보를 갖는 파라미터를 입력받아 기본 골격에 코드를 추가하여 특정 디바이스에 적합한 드라이브를 생성한다. 파라미터는 하드웨어의 스펙정보와 입출력 인터페이스 회로정보를 포함하며 입력파라미터는 OS의 정보와 기본골격을 라이브러리에서 선택하기 위한 정보, 함수 모듈 코드와 변수, 그리고 헤더파일 초기화 코드를 선택하기 위한 정보로 구성된다.

3.4 기본골격

기본골격은 하나의 OS에서 구동되는 디바이스 드라이버들이 open/release 등 최소한의 기본 연산을 할 수 있도록 구현된 코드로써 변경되지 않고 하드웨어의 동작 특성 및 OS 종류에 따라 공통적으로 사용된다. 기본골격은 OS 정보 및 디바이스 드라이버 타입 등을 통해 선택되며 자동생성 시스템 개발 시 여러 디바이스 드라이버의 분석을 통해

```
//모듈 매크로 선언 코드

①static int [device's name]_open( struct inode *inode, struct file *filp )
{
    /*open 함수 기본 내부 연산 코드*/
}

②static int [device's name]_release( struct inode *inode, struct file *filp )
{
    /*release 함수 기본 내부 연산 코드*/
}

③struct file_operations fops = {
    open    : [device's name]_open,
    release  : [device's name]_release,
                    ⋮
};

④static int __init init( void )
{
    /*init 함수 기본 내부 연산 코드*/
}

⑤static void __exit exit( void )
{
    /*exit함수 기본 내부 연산 코드*/
}
//모듈 매크로 선언 코드
```

그림 4. 리눅스 기반의 문자 디바이스 드라이버 기본골격 구조

```
1  :<Linux>
2  :<main-function_description>
3  :        <file_operation_function>
4  :                <open_function>
5  :        static int [device's name]_open( [argument list] )
6  :        {
7  :                MOD_INC_USE_COUNT;
8  :                return 0;
9  :        }
10 :                </open_function>
11 :                <release_function>
12 :        static int [device's name]_release( [argument list] )
13 :        {
14 :                MOD_DEC_USE_COUNT;
15 :                return 0;
16 :        }
17 :                </release_function>
18 :        </file_operation_function>
19 :</main-function_description>
20 :<operation_description>
21 :        <file_operation>
22 :        struct file_operations fops = {
23 :                open    : [device's name]_open,
24 :                release  : [device's name]_release,
25 :        </file_operation>
26 :</operation_description>
27 :</Linux>
```

그림 5. 리눅스 디바이스 드라이버 구조의 기본골격 기술 템플릿

라이브러리로 구축된다. 디바이스 드라이버 기본골격 구조는 그림 4와 같이 구성된다.

①의 함수는 디바이스 드라이버가 커널에 등록된 후 응용프로그램이 하드웨어를 사용하고자할 때 처음 수행되는 함수이다. ②의 함수는 응용프로그램이 하드웨어의 사용을 종료하면 수행되는 함수이다. ③의 'file_operations' 구조체는 문자 디바이스 드라이버일 경우에 디바이스 드라이버와 응용프로그램의 연산을 위해 정의되는 부분이다. 이 구조체는 디바이스 드라이버의 내부에서 사용되는 연산관련 함수만을 정의한다. ④의 함수는 디바이스 드라이버가 커널에 등록될 때 수행되는 함수로 하드웨어가 사용되기 전에 미리 하드웨어의 특징을 커널에 추가하는 부분이다. ⑤의 함수는 디바이스 드라이버가 커널에서 해제될 때 수행되며, 커널에 등록될 때 정의된 연산들을 해제하는 함수이다.

위의 기본골격은 그림5와 같은 템플릿으로 작성되어 라이브러리에 저장하게 된다. 템플릿을 이용해서 기본골격을 라이브러리화하면 각 템플릿별로 기본골격이 분리 저장되어 구조적으로 관리가 가능하며 동일한 OS의 상이한 버전이나 유사한 계통의 OS의 경우 이전 OS에서 사용하는 분리되어 저장된 기본골격을 재사용함으로써 기본골격을 추가로 제작하여 라이브러리화하는 작업을 줄일 수 있다. 그림 5는 그림4와 같은 기본 골격을 생성하는 라이브러리에 저장된 템플릿을 보인다.

기본골격의 각 함수의 이름은 템플릿의 명칭을 참조하며 각 함수에 대한 내용을 입력할 때 '<템플릿명칭>', '</템플릿명칭>'로 시작과 끝을 구분하여 입력한다. 1번 라인의 '<Linux>' 템플릿은 OS의 이름을 나타내며 기본골격의 시작을 나타낸다. 2번 라인에서 보이는 '<main-function_description>' 템플릿의 하위 템플릿인 3번 라인의 '<file_operation>' 템플릿의 내부에 파일 연산에 관련된 함수 기본골격을 입력한다. '<file_operation>' 템플릿의 하위 템플릿인 4번 라인의 '<open_function>' 템플릿에 그림 4의 ①번 함수를 생성하는 내용을 입력했고 11번 라인의 '<release_function>' 템플릿에 그림 4의 ②번 함수를 생성하는 내용을 입력했다. 그림 5의 20번 라인에서 보이는 '<operation_description>' 템플릿 내부의 21번 라인의 '<file_operation>' 템플릿은 그림 4의 ③번 'file_operation' 구조체 기본골격 부분과 같이 생성할 수 있게 한다. 각 함수를 생성하는 부분의 템플릿은 계층적으로 저장되어 3.6장에서 기술되는 디바이스 드라이버 구조 템플릿에 파라미터의 입력값과 함께 통합되어 디바이스 드라이버로 생성된다.

3.5 코드 매핑

입력된 파라미터에 따라 기본골격이 선택되면 기본골격에 작성되어있지 않는 코드를 추가해야한다. 입력 파라미터의 값에 따라 커널 타이머, 인터럽트,

DMA 채널 등의 함수가 추가되어야 하며, 기본골격에 작성되어 있는 함수의 경우도 함수 내부에 코드가 추가된다. 추가되는 함수 모듈 코드는 디바이스 드라이버에서 주요 연산을 하는 코드이며 OS에서 제공하는 디바이스 드라이버 문법에 따라 구성된다. 모듈 코드는 OS마다 따로 구성하여 라이브러리에 저장된다. 함수 모듈 코드는 함수의 기능에 최적화되도록 불필요한 코드를 없애고 유사한 여러 종류의 구현 방식을 하나로 통일하여 함수의 필수적인 연산만으로 라이브러리를 구성한다.

기본골격에 함수 모듈 코드를 생성하면 함수 모듈 코드에서 사용되는 추가적인 변수와 헤더파일 초기화 코드가 필요하다. 디바이스 드라이버 기본골격을 구성할 때 필수적으로 사용되는 헤더파일은 헤더파일 테이블에 기본적으로 선택되고 기본골격에서 쓰이는 변수 초기화 코드는 기본적으로 변수 초기화 코드 링크드 리스트에 포함된다. 추가되는 헤더파일은 OS에서 구현되어 있으며 선언을 통해 사용하므로 라이브러리에 구축되어 있는 테이블을 통해 필요한 헤더파일이 파악되어 추가되어지며, 함수 모듈 코드에서 사용되는 추가적인 변수는 변수 초기화 코드 링크드 리스트에 추가되어 헤더파일 선언 부분과 변수 초기화 부분의 코드를 자동생성한다.

3.6 라이브러리

라이브러리는 OS의 디바이스 드라이버 구조 부분, 기본골격 부분, 함수 모듈 코드 부분, 그리고 변수와 헤더파일 정보 부분으로 구성된다. 본 논문에서 제안하는 디바이스 드라이버 자동생성 시스템은 기존의 디바이스 드라이버 자동생성 시스템에서 새로운 OS를 추가할 경우 모든 하드웨어들의 디바이스 드라이버 생성모듈 전체를 새롭게 추가해야 하는 점을 보완하기 위해서 OS의 디바이스 드라이버 구조를 템플릿으로 작성한 후에 작성된 템플릿의 형식에 맞춰 기본골격을 라이브러리화한다. 템플릿을 이용한 기본골격의 라이브러리화를 통해 기존에 지원되는 OS의 새로운 버전이나 유사한 계통의 OS의 경우 기본골격의 추가입력이 없이 지원되는 OS의 기본골격으로 디바이스 드라이버의 자동생성이 가능하다. 기본골격은 라이브러리에 저장된 OS의 디바이스 드라이버 구조 템플릿을 태그 형식으로 시작과 끝에 사용하여 입력할 수 있다. 함수 모듈 코드는 여러 종류의 디바이스 드라이버 코드 분석을 통해 기본골격에 추가되는 형태로 라이브러리

```
#OS-frame //템플릿 시작
<"Linux"> //OS name
<"main-header"> //main-header file입력부분
<"sub-header"> //필요에 따라 추가되는 header file 입력부분
//OS dependent code 부분
$"MODULE_AUTHOR("SAGOD");"$
$"MODULE_DESCRIPTION("Device Driver");"$
$"MODULE_LICENSE("GPL");"$

<"variable_declaration"> //변수 입력부분
//Timer, Interrupt, 추가적으로 필요한 function 입력부분
<"user-function_description">
    <user-function_description "Kernel_timer_function">
    <user-function_description "Interrupt_handling_function">
                        ⋮
//Device type에 따른 basic function 입력부분
<"main-function_description">
    #"1" //main-function description case 1
    <main-function_description "file_operation_function">
    #~
    #"2" //main-function description case 2
    <main-function_description "block_operation_function">
    #~
                        ⋮
```

그림 6. 리눅스의 디바이스 드라이버 구조 템플릿

에 구성하고 기본골격코드와 함수 모듈 코드에서 사용되는 변수와 헤더파일 코드를 라이브러리로 구성한다. 그림 6은 리눅스의 디바이스 드라이버 구조를 템플릿으로 작성한 것을 보인다.

디바이스 드라이버 구조의 입력은 그림 6에서 보이는 것과 같이 시작에 '#OS-frame'을 입력하여 시작하고 '<','>' 사이에 쌍 따옴표를 사용하여 원하는 각 부분에 구조의 명칭을 기술한다. 각 부분의 기술된 명칭은 그림5에서 기술한 템플릿과 이름으로 매치되어 해당 코드를 삽입하게 된다. 디바이스 드라이버 구조의 마지막은 '#OS-frame-end'를 입력하여 마무리한다. 하나의 템플릿의 하단 템플릿을 입력하는 방식은 '<' 다음에 상위 템플릿의 명칭을 입력하고 두 개의 쌍 따옴표 사이에 하위 템플릿의 명칭을 입력한 뒤에 '>'를 붙여서 입력한다. 템플릿이 여러 가지 타입으로 사용되어지는 경우 각 타입에 해당하는 템플릿의 이전 라인에 '#'를 입력하고 두 개의 쌍 따옴표 사이에 타입의 번호를 입력하여 시작한다. 각 타입의 끝은 입력을 끝낸 템플릿의 다음 라인에 '#~'를 입력하여 끝낸다. 한 OS에서만 쓰이는 OS 의존적인 코드는 원하는 위치에 두 개의 '$'표시를 사용하여 그 사이에 쌍 따옴표를 하고 입력한다. 주석은 '//'를 입력한 뒤에 그 라인 마지막까지 입력한다.

Ⅳ. GUI 환경 및 실험결과

본 절에서는 본 논문에서 제안한 자동생성 시스템의 GUI 환경과 실험 결과에 대해 설명한다. 먼저 GUI 환경을 기술하고, 매뉴얼 설계한 디바이스 드라이버와 자동생성기를 통해 생성한 디바이스 드라이버의 성능을 비교한 실험 결과를 보인다.

4.1 GUI 환경

제안된 시스템은 사용자의 편의를 위해 윈도우 환경의 MFC를 이용하여 GUI 환경으로 구축하였다. GUI를 통한 파라미터 입력 환경은 OS 정보 선택 및 드라이버 이름을 입력하는 OS 입력 윈도우, 디바이스 드라이버 생성을 위한 일반 데이터를 입력할 수 있고 하드웨어 특징 정보를 입력할 수 있는 입력 데이터 윈도우, 새로운 OS의 디바이스 드라이버 구조를 라이브러리에 저장하는 OS 구조 저장 윈도우, 템플릿에 맞춰서 기본골격을 라이브러리에 저장하는 기본골격 저장 윈도우, 그리고 하드웨어와의 실제 데이터 매핑정보 및 인터페이스 구조를 입력하는 인터페이스 연산 데이터 입력 윈도우로 이루어진다. 디바이스 드라이버 자동생성 후에 통합된 디바이스 드라이버 코드는 수정 가능하도록 에디트 기능을 갖는 윈도우에 디바이스 드라이버 코드를 생성한다.

4.2 실험 결과

구현된 시스템의 성능 검증을 위해 디바이스 드라이버는 속도, 코드사이즈, 올바른 기능의 세 가지 방법을 이용하여 매뉴얼에 따라 개발자가 설계한 디바이스 드라이버와 자동생성 시스템을 통해 생성된 디바이스 드라이버의 성능을 비교하였다. 실험을 위하여 삼성 3.5인치 TFT-LCD의 특징 정보와 USB 인터페이스 키보드, 마우스의 특징 정보 및 AC'97 컨트롤러의 특징정보를 디바이스 드라이버 자동생성기의 입력 윈도우에 입력하였다. ARM 프로세서용 리눅스 버전 2.4.19의 디바이스 드라이버 문법에 맞는 TFT-LCD, USB인터페이스 키보드, 마우스 드라이버와 AC'97 컨트롤러 드라이버의 기본골격 코드를 디바이스 드라이버 템플릿의 형식에 맞춰서 자동생성기의 라이브러리에 저장하였다. 디바이스 드라이버 자동생성기를 통해 라이브러리에 서로 다른 버전으로 입력된 기본골격으로 ARM 프로세서용 리눅스 버전 2.4.19에서 구동하는 TFT-LCD 디바이스 드라이버와 USB 인터페이스

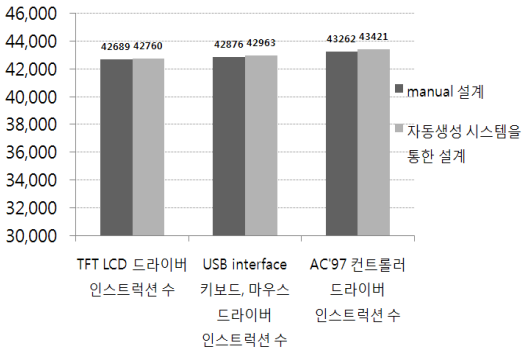

그림 7. 컴파일된 디바이스 드라이버의 인스트럭션 수

키보드, 마우스 드라이버, AC'97 컨트롤러 드라이버를 자동 생성하였다. 그림 7은 매뉴얼 설계된 디바이스 드라이버와 자동 생성된 디바이스 드라이버의 커널 컴파일한 후 인스트럭션 수를 보인다.

매뉴얼 설계한 디바이스 드라이버에 비해 자동 생성된 TFT-LCD 디바이스 드라이버와 USB 인터페이스 키보드, 마우스 드라이버, AC'97 컨트롤러 드라이버는 각각 0.17%, 0.20%, 0.36%의 코드 사이즈 증가를 보인다. 범용적인 기본골격과 함수 모듈 코드를 사용하여 디바이스 드라이버를 생성하므로 매뉴얼 설계에 비해 코드 사이즈는 증가를 보이지만, 불필요한 부분을 없애고 여러 구조를 통일한 함수 모듈 코드를 통한 디바이스 드라이버 생성으로 코드 사이즈에서 매뉴얼 설계에 비해 큰 차이를 보이지 않는다. 표 1은 TFT-LCD 디바이스 드라이버와 USB 인터페이스 키보드, 마우스 드라이버, AC'97 컨트롤러 드라이버의 쓰기 연산시간을 보인다.

매뉴얼 설계된 디바이스 드라이버에 비해 자동 생성된 TFT-LCD 디바이스 드라이버, USB 인터페이스 키보드 그리고 마우스 드라이버는 각각 1.14%, 1.21%, 0.74%의 쓰기 연산 속도 증가를 보인다. 매뉴얼 설계

표 1. 디바이스 드라이버 쓰기 연산시간

	매뉴얼 설계된 드라이버 쓰기 연산 시간	자동 생성된 드라이버 쓰기 연산 시간	비고
TFT-LCD	59,45μs	60,13μs	+1.14%
USB 인터페이스 키보드, 마우스	7.43μs	7.52μs	+1.21%
AC'97 컨트롤러	8.06μs	8.12μs	+0.74%

된 디바이스 드라이버의 경우 ioctl()나 DMA 채널 중에서 하나를 이용해 데이터 전송을 하였으나 자동 생성된 디바이스 드라이버의 경우 함수 모듈 코드 중에서 ioctl()에 정의 되는 부분과 mmap()에 정의 되는 부분의 역할이 달라서 ioctl()과 DMA 채널을 모두 이용해 쓰기 연산을 하여 속도 증가를 보였다. OS 레벨의 응용프로그램에서 드라이버 파일을 오픈하여 쓰기 할 경우 발생할 수 있는 지연시간을 고려하면 실제 드라이버 성능에는 차이가 없는 것을 확인할 수 있다.

본 논문에서 제안한 자동생성 시스템을 이용한 설계는 해당되는 디바이스에 대한 정보를 이해하는데 걸리는 시간을 제외한 나머지 시간의 대부분을 절약할 수 있어 매뉴얼 설계에 비해 디바이스 드라이버 개발시간을 크게 줄일 수 있고, 기존의 디바이스 드라이버 제작 도구에 비해 기본골격의 재사용으로 라이브러리를 갱신하는데 들어가는 시간과 비용을 줄일 수 있다. 디바이스 드라이버 자동생성 시스템에 의해 자동 생성된 TFT-LCD 드라이버와 USB 인터페이스 키보드, 마우스 드라이버, AC'97 컨트롤러 드라이버 모두 개발자가 매뉴얼에 따라 설계한 디바이스 드라이버와 비교하여 오류 없이 동일한 결과를 보이는 것을 확인하였으며, 디바이스 드라이버 자동생성 시스템을 통해 자동 생성된 디바이스 드라이버가 매뉴얼 설계된 디바이스 드라이버에 비해 성능이 대등함을 확인하였다.

V. 결론 및 추후과제

본 논문에서는 디바이스 드라이버 자동생성 시스템에 대해 제안하였다. 제안한 디바이스 드라이버 자동생성기는 특정 OS에 대한 디바이스 드라이버가 라이브러리화 되어있는 경우 새로운 버전이 추가되거나 유사한 계통의 OS가 새로이 추가되더라도 하드웨어의 기본골격을 추가할 필요가 없이 기존의 하드웨어의 기본골격을 이용하여 디바이스 드라이버를 자동생성할 수 있다. 디바이스 드라이버 자동생성 시스템은 GUI환경을 통해 입력 데이터를 쉽게 입력할 수 있으며, 생성된 디바이스 드라이버는 에디트 윈도우에 출력되어 수정과 저장을 할 수 있다. 자동생성 시스템은 자동생성을 위한 파라미터로 OS 정보, 하드웨어 특징 정보, 인터페이스 정보 등을 입력받는다. 파라미터를 통해 라이브러리에 저장된 특정 OS의 디바이스 드라이버 구조 템플릿의 형태로 입력된 기본골격을 선택한다. 디바이스 드라이버의 기본골격이 선택되면 하드웨어 특징 정보와 인터페이스 회로 구조 등을 통해 드라이버 코드에 추가할 새로운 함수 모듈 코드와 기본골격 내부 코드들이 생성되어 추가된다. 디바이스 드라이버에 필요한 함수 모듈 코드가 생성 추가되면 전체 코드에서 사용되는 변수와 모듈 매개변수 선언 코드를 생성하고 헤더파일 테이블을 통해 필요한 헤더파일을 선택하여 변수와 헤더파일을 선언과 초기화하는 코드가 추가된다. 제안된 방법으로 TFT-LCD 디바이스 드라이버와 USB 인터페이스 키보드, 마우스 드라이버, AC'97 컨트롤러 드라이버를 자동생성하여 성능을 측정한 결과 디바이스 드라이버 자동생성 시스템을 통해 자동 생성된 디바이스 드라이버는 매뉴얼 설계된 디바이스 드라이버와 대등한 성능을 보인다.

현재 구현한 디바이스 드라이버 자동생성 시스템은 라이브러리가 구축된 상태에서 GUI를 통해 사용자가 요구되는 입력 파라미터를 직접 입력하여 디바이스 드라이버 코드를 자동생성하는 방식이다. 라이브러리에서 지원하지 않는 하드웨어의 경우, 유사한 하드웨어의 디바이스 드라이버를 자동생성하여 수정하는 방법을 사용할 수 있다. 이 경우 생성한 디바이스 드라이버는 수정 후에 커널 컴파일을 통해 디버깅 과정을 거쳐야 하므로 자동생성한 디바이스 드라이버를 타겟 하드웨어와 직접 연결하여 디버깅할 수 있는 모듈의 생성이 차후 요구된다.

참 고 문 헌

〔1〕 김선자, 김홍남, 김채규, "정보가전용 임베디드 운영체제 기술" 한국통신학회지(정보와 통신), 제 18권, 제12호, pp.72-81, 2001년 12월.

〔2〕 A. Rubini, J. Corbet, and G. Kroah- Hartman, Linux Device Drivers, 3rd Edition, O'Reilly Pub., 2005.

〔3〕 T. Katayama, K. Saisho, and A. Fukuda, "Proposal of a Support System for Device Driver Generation", in Proc. Software Engineering Conf., pp.494-497, Dec. 1999.

〔4〕 황선영, 김현철, 이서훈, "인터페이스 회로와 디바이스 드라이버 통합 자동생성 시스템 설계", 한국통신학회 논문지, 제 32권, 제6호, pp.325-333, 2007년 6월.

〔5〕 Y. Ma and C. Lim, "Test System for Device Drivers of Embedded Systems", in Proc. ICACT2006, Vol.1, pp.550-552, Feb. 2006.

〔6〕 Jungo,Windriver,http://www.jungo.com/ support/ support_ windriver.html
〔7〕 Compuware,DriverStudio,http://www. compuware.com/ products/driverstudio/default.htm
〔8〕 마이크로 소프트, Windows Driver Development kit, http://www.microsoft.com/whdc/ devtools/ddk/default.mspx
〔9〕 Krueger, "Software reuse", ACM Computing Surveys(CSUR), Vol.24, Issue 2, pp.131-184, Jun. 1992.
〔10〕 C. McClure, The Three Rs of Software Automation: Re-engineering, Repository, Reusability, Prentice-Hall Pub., 1992.
〔11〕 E. Tuggle, "Introduction to Device Driver Design", in Proc. 5th Annual Embedded Sys. Conf., Vol.2, pp.455-468, 1993.
〔12〕 D. Jensen, J. Madsen, and S. Pedersen, "The Importance of Interfaces: A HW/SW Codesign Case Study", in Proc. 5th Int. Workshop on Hw/Sw Codesign CODES'97, pp.87-91, March 1997.
〔13〕 Y. Choi, W. Kwon, and H. Kim, "Code Generation for Linux Device Driver", in Proc. ICACT2006, Vol.1, pp.734-737, Feb. 2006.
〔14〕 M. Zhu, L. Luo, and G. Xiong, "High-Availability in σ-CORE: A Formal Derivation", Dedicated Systems Magazine, Vol.3, pp.38-46, July 2001.
〔15〕 M. Zhu, L. Luo, and G. Xiong, "The Minimal Model of Operating Systems", ACM SIGOPS Operating Systems Review, Vol.35, Issue 3, pp.22-29, July 2001.
〔16〕 Q. Zhang, M. Zhu, and S. Chen, "Automatic Generation of Device Drivers", ACM SIGPLAN Not., Vol.38, Issue 6, pp.60-69, June 2003.
〔17〕 T. Katayama, K. Saisho, and A. Fukuda, "Prototype of the Device Driver Generation System for UNIX-like Operating Systems", in Proc. Int. Symp. Principles of Software Evolution, pp.302-310, Nov. 2000.

김 현 철 (Hyoun-Chul Kim) 준회원

2006년 2월 조선대학교 컴퓨터공학과 졸업
2008년 2월 서강대학교 전자공학과 석사
2008년 3월~현재 LG 전자 연구원
<관심분야> Automatic generation of device driver

이 서 훈 (Ser-Hoon Lee) 정회원

2003년 2월 서강대학교 전자 공학과 졸업
2005년 2월 서강대학교 전자 공학과 석사
2005년 3월~현재 서강대학교 전자공학과 대학원 CAD & Embedded Systems 연구실 박사과정
<관심분야> SoC 설계, IP Interface 자동설계기법

황 선 영 (Sun-Young Hwang) 정회원

1976년 2월 서울대학교 전자공학과 졸업
1978년 2월 한국 과학원 전기및 전자공학과 공학석사
1986년 10월 미국 Stanford 대학 전자공학 박사
1976년~1981년 삼성반도체 주식회사연구원, 팀장
1986년~1989년 Stanford 대학 Center for Integrated System 연구소 책임 연구원 Fairchild Semiconductor Palo Alto Research Center 기술 자문
1989년~1992년 삼성전자(주) 반도체 기술 자문
1989년 3월~현재 서강대학교 전자공학과 교수
<관심분야> SoC 설계 및 framework 구성, CAD시스템, Com. Architecture 및 DSP System Design 등