인터페이스 회로와 디바이스 드라이버 통합 자동생성 시스템 설계

정회원 황 선 영*, 준회원 김 현 철*, 이 서 훈*,

Design of an Integrated Interface Circuit and Device Driver Generation System

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

요 약

설계된 HW IP를 응용수준에서 제어하기 위해 OS상에서의 디바이스 드라이버가 요구된다. 디바이스 드라이버의 개발은 하드웨어와 OS에 대해 시스템 개발자의 정확한 이해가 필요하며 하드웨어 개발 기간과 비용의 많은 부분을 차지한다. 본 논문에서는 OS정보, 하드웨어 특징정보를 이용하여 OS에 따른 디바이스 드라이버를 인터페이스 회로와 함께 자동 생성하는 시스템의 구축에 대해 제시한다. 제안한 시스템에서는 효율적인 디바이스 드라이버 자동생성을 위해 디바이스 드라이버의 기본골격과 함수 모듈 코드, 헤더파일 테이블 등을 라이브러리로 구축하여 입력 데이터에 따라 선택되어 디바이스 드라이버가 자동생성 되도록 하였다. 제안된 방법으로 ARM922T 코어에 삼성 3.5인치 TFT-LCD를 장착하여 커널버전 ARM-Linux 2.4.19를 탑재한 후 디바이스 드라이버를 자동 생성하여 커널에 등록한 뒤 하드웨어에 write 연산을 실행하는데 걸린 시간을 비교한 결과 매뉴얼로 설계한 디바이스 드라이버에 비해 1.12%의 감소를 보였다. 커널 컴파일 후의 코드 사이즈는 0.17%의 증가를 보였다. 생성된 디바이스 드라이버는 응용프로그램 레벨에서 하드웨어를 제어할 때 발생하는 지연시간을 고려하면 실제 성능의 차이가 없음을 보인다. 본 논문에서 제안한 시스템을 사용하여 시스템 개발기간을 단축할 수 있다.

Key Words : Device driver, Interface circuit, Generation system, IP, Operating system

ABSTRACT

An OS requires the device driver to control hardware IPs at application level. Development of a device driver requires specific acknowledge for target hardware and OS. In this paper, we present a system which generates a device driver together with an interface circuit. In the proposed system, an efficient device driver is generated by selecting a basic device driver skeleton, a function module code, and a header file table from the pre-constructed library and an interface circuit is constructed such that the generated device driver operates correctly. The proposed system is evaluated by generating a TFT-LCD device driver on the ARM922T core with 3.5 inch Samsung TFT-LCD in ARM-Linux environment. Experiment result shows that the writing time on the LCD is decreased by 1.12% and the compiled code size is increased by 0.17% compared to the manually generated one. The automatically generated device driver has no performance degradation in the latency of hardware control at the application program level. The system development time can be reduced using the proposed device driver generation system.

※ 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT 연구센터 육성지원사업의 연구결과로 수행되었음.
* 서강대학교 전자공학과 대학원 CAD & ES연구실
논문번호 : KICS2007-03-133, 접수일자 : 2007년 3월 22일, 최종논문접수일자 : 2007년 6월 13일

Ⅰ. 서 론

반도체 산업과 집적회로 기술의 발전으로 회로 집적도가 높아짐에 따라 과거에 PCB위에 시스템을 구현하던 기술이 현재는 하나의 칩에 집적화 할 수 있는 SoC로 발전하였다. SoC는 프로세서와 하드웨어, 소프트웨어, 복잡한 버스 모듈, clock, power 관리모듈, 테스트 모듈, 그리고 버스 등을 하나의 칩에 포함하므로 많은 개발 기간과 비용을 요구한다[1]. 반도체 칩은 빠른 time-to-market 요구에 만족하기 위해 기능 블록을 처음부터 설계하지 않고 기존에 설계되고 검증되어 있는 IP (Intellectual Property)를 사용하여 칩의 개발기간을 단축하는 방식이 채택되고 있다. 이미 설계 검증된 IP는 특정한 모듈과의 데이터 통신을 위한 특정한 프로토콜을 갖고 있어 다른 모듈과의 통신을 위해서는 모듈간의 서로 다른 프로토콜을 맞춰주는 인터페이스를 필요로 한다[2,3,4]. 인터페이스 회로는 데이터 전송 시간을 지연하게 되므로 저성능 인터페이스 회로는 시스템 성능의 저하를 가져올 수 있다. 이러한 성능 저하를 줄이기 위해 인터페이스 회로는 매뉴얼에 따른 설계 방식이 주를 이룬다. 인터페이스 회로의 매뉴얼 설계의 경우 많은 설계 시간과 비용을 요구하므로 인터페이스 회로 자동생성 시스템은 매뉴얼을 통한 인터페이스 회로 생성에 비해 설계 시간을 절약할 수 있다[5,6].

설계된 하드웨어 IP를 응용프로그램 수준에서 동작시키기 위해 OS에 따라 다른 디바이스 드라이버를 개발해야 한다. 디바이스 드라이버는 하드웨어와 OS 및 응용프로그램 사이의 연결고리 역할을 하면서 응용 프로그램이 하드웨어에서 제공하는 기능의 효율적인 사용을 위해 하드웨어와 소프트웨어 사이에 제어와 상호 동작을 하도록 일관된 인터페이스를 제공하는 소프트웨어이다[7]. 디바이스 드라이버는 하드웨어와 OS에 의존적으로 드라이버 개발자가 하드웨어와 OS에 대한 정확히 이해가 필요하며 디바이스 드라이버 제작은 하드웨어 개발기간과 비용의 많은 부분을 차지한다[8]. 디바이스 드라이버 자동 생성 시스템의 개발은 디바이스 드라이버의 개발기간과 비용을 단축시켜 하드웨어 제작 기간과 제작비용의 절약을 통해 전체 시스템의 비용을 줄이는 효과를 보인다. 본 논문에서 구현된 인터페이스 회로와 디바이스 드라이버 통합 자동 생성 시스템을 통해 SoC 시스템 개발 기간과 비용을 줄일 수 있다.

한국전자통신연구원에서 개발한 Quick driver[9], Jungo사의 WinDriver[10], Compuware의 DriverStudio[11], 그리고 Microsoft에서 개발한 DDK[12] 등이 개발 발표되었다. 이 프로그램들은 완성도 높은 디바이스 드라이버를 자동 생성하지만 인터페이스 회로의 자동생성에 대한 고려가 없어 개발자가 인터페이스 회로를 설계하며 시스템의 동작을 확인하여야 하는 co-design 과정을 거쳐야 한다.

본 논문에서는 인터페이스 회로와 디바이스 드라이버의 통합 생성 시스템의 설계와 구축 결과를 제시한다. 2절에서는 관련연구에 대해 설명하고, 3절에서는 제안된 인터페이스 회로와 디바이스 드라이버 통합 생성 시스템에 대해 설명한다. 4절에서는 입력 GUI 환경의 소개와 자동 생성된 디바이스 드라이버와 매뉴얼 설계한 디바이스 드라이버의 성능 비교를 보인다. 마지막으로 5절에서는 결과 및 추후 과제를 제시한다.

Ⅱ. 관련 연구

디바이스 드라이버 자동 생성 시스템으로 한국전자통신연구원에서 개발한 Quick driver[9], Jungo사에서 개발한 WinDriver[10], Compuware에서 개발한 DriverStudio[11], 그리고 Microsoft에서 개발한 DDK[12] 등이 발표되었다. Quick driver는 리눅스 기반의 디바이스 드라이버 개발을 지원하는 개발 툴로서 PCI, USB, PCMCIA 등의 버스 타입과 문자형, 블록형, 네트워크형의 디바이스 종류에 따라 디바이스 드라이버가 유사한 패턴을 갖고 있다는 점에 착안하여 개발되었다. 디바이스 정보와 함께 드라이버에 관한 세부 정보들을 입력해서 디바이스 드라이버의 골격 소스를 생성한다. 또한, 불완전하게 생성된 디바이스 드라이버 소스코드의 질을 향상시키기 위해서 소프트웨어 테스팅과 정적 검증기능을 지원한다[9,13]. 디바이스 드라이버 생성시스템으로 완성도가 높은 시스템이나 아직은 Linux OS에 한정되어 있다. Jungo사에서 개발하여 보급하고 있는 WinDriver[10]는 현재 사용되는 대부분의 OS 종류에 따라 프로그램을 개발하여 보급하고 있으며 PCI, PCMCIA, ISA, ISA PnP, EISA, CompactPCI 등의 Bus architecture와 USB를 지원한다. WinDriver에서는 별도의 GUI를 제공하지 않고 개발자의 환경에서 제공하는 컴파일러 중에서 개발자가 원하는 컴파일러를 선택하여 연동하고 작성할 수 있도록 개발되었다. WinDriver는 디바이스 드라이버의 신

속한 개발을 위해 드라이버 골격 코드를 자동생성하나, OS종류와 버전별로 프로그램을 각각 개발하여 보급하고 있어 대부분의 OS를 지원한다. 새로운 구조의 하드웨어나 시스템이 개발되었을 경우 모든 디바이스 드라이버 개발 프로그램을 변경하거나 새롭게 개발해야하는 단점을 갖고 있다. 이 외에도 Compuware의 DriverStudio[11], Microsoft의 DDK[12] 등이 있으며 이 디바이스 드라이버 개발 툴들의 경우 지원하는 OS가 모두 Windows에 국한되어져 있으며 다양한 OS를 위한 디바이스 드라이버 개발은 개발자가 별도로 제작해야한다. 현재 개발된 관련 프로그램의 경우 인터페이스 회로에 대한 고려가 없어 디바이스 드라이버 개발자가 인터페이스 회로를 구현해야하며, 임베디드 시스템의 OS는 일반 OS와 디바이스 드라이버의 차이가 있어 임베디드 OS를 지원하지 않는 개발 프로그램의 경우 디바이스 드라이버 생성 후 추가적인 수정이 필요하다.

Ⅲ. 제안된 인터페이스 회로와 디바이스 드라이버 통합 생성 시스템

여기서는 제안된 인터페이스 회로와 디바이스 드라이버 통합 생성 시스템에 대해 기술한다. 먼저 통합 생성 시스템을 보이고, 인터페이스 회로 자동생성 시스템과 디바이스 드라이버 자동생성 시스템에 대해 보인다. 통합 생성 시스템은 인터페이스와 디바이스 드라이버를 자동 생성한다. 디바이스 드라이버 자동생성기는 OS 정보와 인터페이스 자동생성기를 통해 생성된 인터페이스 회로와 하드웨어 정보 등을 입력으로 받아서 디바이스 드라이버를 자동생성한다.

3.1 인터페이스 자동생성 시스템

하드웨어간의 인터페이스 자동생성기는 GUI를 통해 인터페이스 회로 설계 스펙을 입력하고 IP 프로토콜 입력기를 사용하여 IP의 프로토콜을 입력하면 인터페이스 회로 생성기에 입력으로 들어갈 문자열을 생성해 준다[6]. IP의 스펙 및 프로토콜에 대한 기술은 문자열로 인터페이스 회로 생성기의 입력으로 들어가게 되어 IP의 프로토콜과 듀얼 구조인 FSM을 생성하게 되고, 라이브러리에 저장된 표준화된 버스 프로토콜에 듀얼 구조인 FSM과 버퍼를 사용하여 인터페이스 회로를 생성해 낸다. 결과 인터페이스 모듈은 IP의 듀얼 FSM, 버스의 듀얼 FSM, 그리고 버퍼를 연결한 구조로 RTL 수준의

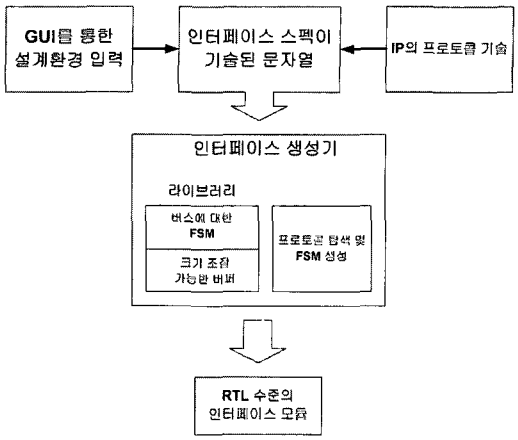


그림 1. 인터페이스 생성기 흐름도.

HDL 코드를 출력하게 된다[6]. 그림 1은 사용된 하드웨어간의 인터페이스 회로 자동생성기의 흐름을 보인다.

3.2 디바이스 드라이버 자동생성 시스템

여기서는 디바이스 드라이버 자동생성 시스템을 제시한다. 먼저 자동생성기 구조를 보이고, 기본골격, 입력 데이터, 함수 모듈 코드, 변수 및 헤더파일 초기화 코드, 라이브러리 등을 보인다.

3.2.1 자동생성 시스템 개관

디바이스 드라이버의 기능은 하드웨어 타입에 따라 결정된다. 타입과 기능이 같은 하드웨어 중에서 다른 칩을 사용하는 하드웨어의 경우에도 디바이스 드라이버가 작성되는 방법은 비슷하다[14]. 최소한의 기능을 하는 디바이스 드라이버를 이용해 동일한 기능의 하드웨어들의 디바이스 드라이버를 제작할 수 있다. 동일한 하드웨어라도 OS다다 디바이스 드라이버 작성 방법은 다르지만, OS에 따른 디바이스 드라이버를 작성하는데 필요로 하는 하드웨어에 대한 특징적인 정보는 크게 다르지 않다. 리눅스에서 디바이스 드라이버를 개발하는데 필요한 정보는 리눅스 디바이스 드라이버의 기본적인 구조와 하드웨어의 특징적인 정보들이다. Windows의 경우에도 필요한 정보는 Windows 디바이스 드라이버의 기본적인 구조와 하드웨어의 특징적인 정보이며 이들 정보는 최소한의 기능을 갖는 디바이스 드라이버의 경우 유사하다[14].

디바이스 드라이버 자동생성 시스템에서 필요한 정보들은 하드웨어 종류에 따른 기본 골격, OS 정보, 그리고 하드웨어의 특징 정보와 인터페이스 회로

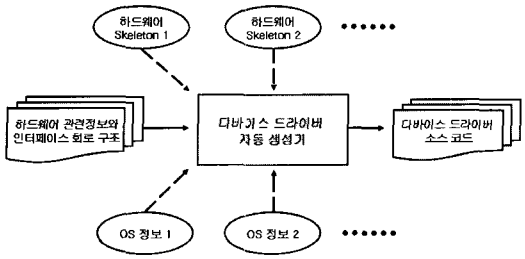


그림 2. 디바이스 드라이버 자동생성 시스템.

정보 등이다. 동일한 기능의 하드웨어의 경우에 매우 유사한 디바이스 드라이버 구조를 갖고 있으므로 유사한 하드웨어의 구조를 포함한 하드웨어의 기본 골격을 자동생성 시스템에 포함시켜야 한다. OS는 종류에 따라 디바이스 드라이버의 작성 방식이 다르며, 동일한 OS의 경우에도 버전에 따라 디바이스 드라이버 작성 방식에 차이가 있어 버전정보도 자동생성 시스템에 포함 시켜야 한다. 동일한 기능의 하드웨어의 경우에도 하드웨어의 특징과 인터페이스 회로 구조에 따라 디바이스 드라이버가 달라지므로 하드웨어의 특징정보와 인터페이스 회로 구조를 시스템에 포함시켜서 선택하거나 입력 데이터를 통해 입력해 주어야 한다. 그림 2는 디바이스 드라이버 자동생성 알고리즘을 적용한 디바이스 드라이버 자동생성 시스템 구조를 보인다.

기존의 디바이스 드라이버 개발 프로그램의 경우 단일 OS에 해당하는 드라이버 코드가 생성이 되며, 다른 OS의 디바이스 드라이버를 개발하려면 별도의 시간이 추가로 소모되고 새로운 하드웨어가 개발될 경우 모든 OS에 맞는 디바이스 드라이버를 새로 개발해야하는 단점이 있다. 본 논문에서 제안하는 디바이스 드라이버 자동생성 시스템은 동일한 하드웨어의 경우 서로 다른 OS라 할지라도 추가로 필요한 디바이스 특징 정보가 매우 유사하다는 점을 착안하여 OS종류와 하드웨어별로 드라이버의 기본 골격을 라이브러리에 미리 입력해 놓고 추가로 필요한 정보들의 입력을 통해 원하는 OS에 해당하는 디바이스 드라이버를 생성한다.

3.2.2 자동생성기 흐름도

디바이스 드라이버 자동생성기는 GUI로 구성된 입력기를 통해 입력 데이터를 입력하고, 입력된 데이터는 자동생성기 내부에 맞게 입력 데이터 파서를 통해 파싱된다. 파싱된 데이터를 통해 기본골격 코드가 선택되고 기본골격코드에 필요한 최적화된 함수 모듈 코드 및 변수와 헤더파일 초기화 코드가

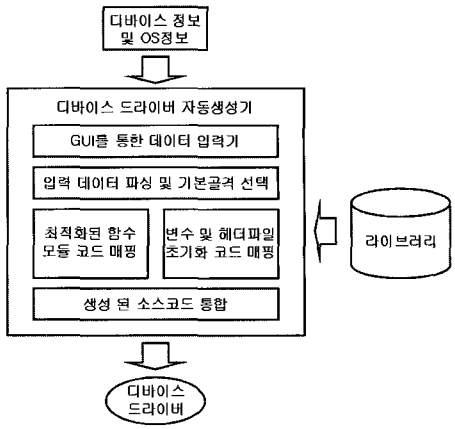


그림 3. 디바이스 드라이버 자동생성기 흐름도.

생성된다. 생성된 소스코드들은 마지막으로 디바이스 드라이버 양식에 맞게 통합되어 출력된다. 그림 3은 제안한 디바이스 드라이버 자동생성기의 흐름을 보인다.

3.2.3 기본골격

기본골격은 하나의 OS에서 구동되는 디바이스 드라이버들이 기본적으로 open/release 등의 최소한의 기본 연산을 할 수 있도록 구현된 코드이며 변경되지 않고 공통적으로 사용되는 코드이다. 기본골격은 OS 정보 및 디바이스 드라이버 타입 등을 통해 선택되며 여러 디바이스 드라이버의 분석을 통해 기본골격을 라이브러리로 구축하고 입력되는 데이터에 따라 기본골격을 선택하여 출력한다. 그림 4는 임베디드 리눅스 환경의 문자 디바이스 드라이버의 기본 골격 구조를 보인다.

```
//모듈 매크로 선언 코드

static int open( struct inode *inode, struct file *filp)
{
    /*open함수 기본 내부 연산 코드*/
}

static int release( struct inode *inode, struct file *filp)
{
    /*release함수 기본 내부 연산 코드*/
}

struct file_operations fops = {
  open    : <device's name>_open,
  release: <device's name>_release,
          ⋮
};

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

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

그림 4. 임베디드 리눅스 기반의 문자 디바이스 드라이버 기본 골격 구조.

3.2.4 입력 데이터

입력 데이터는 기본골격을 라이브러리에서 선택하기 위한 정보들과 함수 모듈 코드를 선택하기 위한 정보들로 구성된다. 기본골격을 선택할 수 있는 입력 데이터에는 OS 정보 및 디바이스 드라이버 타입 등이 있으며 이 정보는 기본 골격뿐만 아니라 함수 모듈 코드의 종류를 선택하는 정보로도 이용된다. 하나의 OS에서 구동되는 디바이스 드라이버들에서 기본 골격을 제외하고 변화될 수 있는 코드들을 분석하면 함수 모듈 코드를 생성 및 추가할 수 있는 선택 요소들을 구할 수 있으며 임베디드 리눅스의 경우 커널 타이머, 인터럽트, 파라미터 정보 등이 선택 요소가 된다.

3.2.5 최적화된 함수 모듈 코드

함수 모듈 코드는 디바이스 드라이버에서 주요 연산을 하는 코드이며 OS에서 제공하는 디바이스 드라이버 문법에 따라 구성된다. OS마다 디바이스 드라이버의 문법이 틀리며 함수 모듈 코드도 OS마다 따로 구성하여 라이브러리에 저장된다. 함수 모듈 코드는 함수의 기능에 최적화 되도록 불필요한 코드를 없애고 유사한 여러 종류의 구현 방식을 하나로 통일하여 라이브러리를 구성한다. 기본골격이 선택된 후 입력 데이터의 내용에 따라 필요한 함수 모듈 코드가 선택되며 인터럽트나 커널 타이머 같은 새로운 함수 모듈 코드가 기본골격에 추가된다. 그림 5는 임베디드 리눅스 환경에서 구동되는 TFT-LCD 디바이스 드라이버의 함수 코드를 보인다. 그림 5(a)는 디바이스 드라이버의 기본골격 코드의 일부분이며, 그림 5(b)는 'init' 함수의 기본골격 코드에 인터럽트가 추가되는 코드를 보인다.

3.2.6 변수 및 헤더파일 초기화 코드

디바이스 드라이버에서 사용되는 변수에는 모듈 매개 변수와 프로그래밍 언어에서 사용되는 문법과 동일한 방법으로 사용되는 일반 변수가 있다. 모듈 매개 변수는 모듈 매크로를 통해 선언이 되며 디바이스 드라이버 모듈 등록 시 지정하여 사용하고 입력데이터를 통해 생성된다. 일반 변수의 경우 기본골격에서 사용되는 변수 외에도 함수 모듈 코드를 기본골격에 추가할 경우 새롭게 사용되는 변수가 생기며, 변수의 사용을 위해 변수의 선언 및 초기화 코드의 추가가 필요하다. 함수 모듈 코드를 통해 추가되는 변수는 linked-list를 통해 추가되어 관리되고 최종적으로 생성된 드라이버 코드 통합 시에 출력

```
static int __init lcd_init(void)
{
    int result;
    //문자 디바이스 드라이버 등록
    result = register_chrdev( lcd_major, "lcd", &lcd_fops);
    if( result < 0 )
    {  //major number를 재대로 할당 받지 않았으면 에러 메시지 출력
        printk(KERN_INFO "lcd : Can't get major number₩n");
        return result;
    }
    //드라이버 미등록시 에러 출력.
    printk("lcd : registerd TFT-LCD driver₩n");
    return 0;

}
```

(a)

```
static int __init lcd_init(void)
{
    int result;
    //인터럽트 공유등록
    if( request_irq( LCD_IRQ, lcd_interrupt, 0, "LCD IRQ", NULL ) != 0 )
    {  //인터럽트가 정상적으로 등록되지 않으면 에러 메시지 출력
        printk("Can't request IRQ 0x%x₩n", LCD_IRQ );
        return -1;
    }
    //문자 디바이스 드라이버 등록
    result = register_chrdev( lcd_major, "lcd", &lcd_fops);
    if( result < 0 )
    {  //major number를 재대로 할당 받지 않았으면 에러 메시지 출력
        printk(KERN_INFO "lcd : Can't get major number₩n");
        //인터럽트 공유해제
        free_irq(LCD_IRQ, NULL);
        return result;
    }
    //드라이버 미등록시 에러 출력
    printk("lcd : registerd TFT-LCD driver₩n");
    return 0;

}
```

(b)

그림 5. LCD 드라이버 'init' 함수 코드. (a) LCD 드라이버 기본 골격 'init' 함수 코드, (b) 인터럽트를 사용 코드를 추가한 'init' 함수 코드.

된다. 디바이스 드라이버 기본골격을 구성할 때 필수적으로 포함되는 헤더파일 외에도 함수 모듈 코드가 생성되면서 함수의 사용을 위해 헤더파일의 추가가 필요하다. 헤더파일은 이미 OS에서 구현되어 선언을 통해 사용하므로 라이브러리에 구축되어 있는 테이블을 통해 추가되는 헤더파일을 파악하고 테이블의 'Enable' flag를 '1'로 변환하여 함수 모듈 코드가 모두 생성되면 헤더파일 선언 코드를 생성되는 디바이스 드라이버 코드에 추가한다. 그림 6은 함수 모듈 코드를 통해 변수와 헤더파일의 선언 및 초기화 코드가 추가되는 전체 구조를 보인다.

3.2.7 라이브러리

디바이스 드라이버를 자동생성 하는데 필요로 하는 데이터들을 라이브러리로 구성하기 위해서는 여러 디바이스 드라이버 코드의 분석이 필요하다. 여러 종류의 디바이스 드라이버 코드 분석을 통해 기본

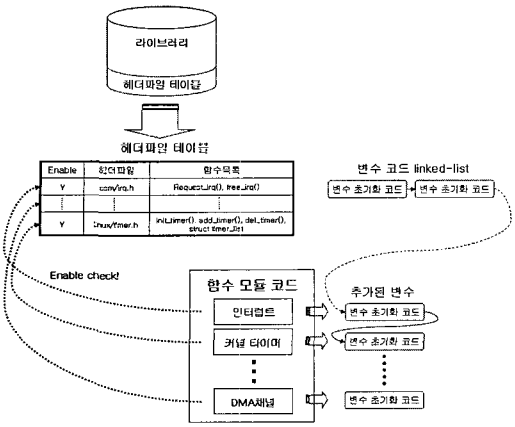


그림 6. 헤더파일 및 변수 초기화 코드 추가 구조도

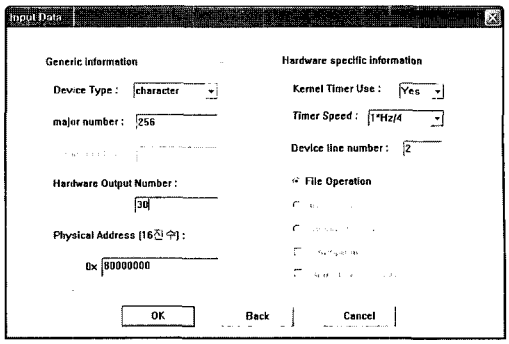
골격코드 및 형태가 크게 변하지 않는 함수 모듈 코드를 라이브러리로 구성한다. 라이브러리는 기본 골격, 함수 모듈 코드, 헤더파일 테이블 등의 세 부분으로 구성되어 있다. OS 종류와 하드웨어 타입에 따라 공통으로 사용할 수 있는 디바이스 드라이버 기본골격이 미리 라이브러리에 구성된다. 기본골격 코드가 생성된 후에 추가되는 함수 모듈 코드는 변화될 수 있는 부분을 제외하고 OS 종류에 따라 다른 디바이스 드라이버 문법에 맞도록 구성된다. 헤더파일은 디바이스 드라이버에서 사용되는 모든 헤더파일을 사용되어지는 함수와 종류에 따라 테이블로 구성되어 라이브러리에 저장된다. 테이블의 구조는 헤더파일명과 사용되는 함수, 그리고 'Enable' flag로 이루어진다.

Ⅳ. GUI 환경 구축 및 실험 결과

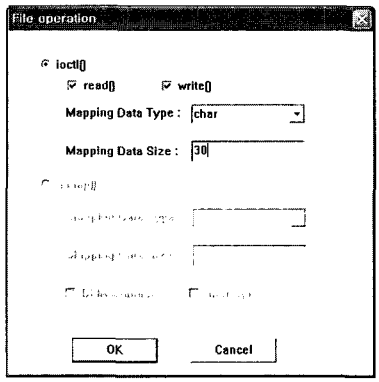
본 절에서는 입력 GUI 환경에 대해서 기술하고 디바이스 드라이버 자동생성기를 통해 자동 생성된 디바이스 드라이버의 속도와 코드 사이즈 등을 매뉴얼 설계한 디바이스 드라이버 코드와 비교한 결과를 보인다.

4.1 GUI 환경

구현된 시스템의 입력은 Windows 환경의 MFC를 이용하여 GUI 환경으로 구축하였다. GUI를 통한 데이터 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 윈도우, 그리고 하드웨어와의 실제 데이터 매핑정보 및 인터페이스 구조를 입력하는 파일 연산 데이터 입력 윈도우로



(a)



(b)

그림 7. GUI 입력 환경. (a) 일반 데이터 및 특징 데이터 입력 윈도우, (b) 파일 연산 데이터 입력 윈도우.

이루어진다. 디바이스 드라이버 자동생성 후에 통합된 디바이스 드라이버 코드는 수정 가능하도록 에디트 기능을 갖는 윈도우에 디바이스 드라이버 코드를 생성시킨다. 그림 7은 구현한 디바이스 드라이버 자동생성기의 GUI 입력 환경을 보인다.

4.2 실험 결과

구현된 시스템의 성능 검증을 위해 속도, 코드사이즈, 올바른 실행의 세 가지 방법을 이용하였으며, 매뉴얼에 따라 개발자가 설계한 디바이스 드라이버와 자동생성 시스템을 통해 생성된 디바이스 드라이버의 성능을 비교하였다.

실험을 위하여 삼성 3.5인치 TFT-LCD의 특징 정보를 디바이스 드라이버 자동생성기의 입력 데이터 윈도우에 입력하였고, ARM-Linux 기반의 커널 버전 2.4.19의 디바이스 드라이버 문법에 맞는 TFT-LCD 기본골격 코드를 자동생성기 라이브러리에 입력하였다. 디바이스 드라이버 자동생성기를 통해 ARM-Linux 기반의 커널 버전 2.4.19에서 구동하는 TFT-LCD 디바이스 드라이버를 자동 생성하였다. 매뉴얼 설계한 디바이스 드라이버에 비해 자

동 생성된 디바이스 드라이버는 0.17%의 코드 사이즈 증가를 보인다. 범용적인 기본골격과 함수 모듈 코드를 사용하여 디바이스 드라이버를 생성하므로 매뉴얼 설계에 비해 코드 사이즈는 증가를 보이지만, 최적화한 함수 모듈 코드를 통한 디바이스 드라이버 생성으로 코드 사이즈에서 매뉴얼 설계에 비해 큰 차이를 보이지 않는다. 매뉴얼 설계된 디바이스 드라이버에 비해 자동 생성된 디바이스 드라이버는 1.12%의 write 연산 속도 감소를 보인다. 매뉴얼 설계된 디바이스 드라이버의 경우 DMA를 이용해 데이터 전송을 하였으나 자동 생성된 디바이스 드라이버의 경우 ioctl()과 DMA를 모두 이용해 write 연산을 하여 1.12%의 속도 감소를 보였으며 OS 레벨의 응용프로그램에서 드라이버 파일을 open 하여 write 할 경우 발생할 수 있는 지연시간을 고려하면 실제 드라이버 성능에는 차이가 없는 것을 확인할 수 있다. 표 1은 개발자에 의해 매뉴얼 설계된 디바이스 드라이버와 자동 생성된 디바이스 드라이버를 커널 컴파일한 후 인스트럭션 수와 등록 후 write 연산시간을 보인다.

표 1. 컴파일된 디바이스 드라이버의 인스트럭션 수와 write 연산시간.

	매뉴얼 설계된 디바이스 드라이버	제안된 시스템이 생성한 디바이스 드라이버	비 고
TFT-LCD용 디바이스 드라이버 인스트럭션 수	42,689	42,760	+0.17%
TFT-LCD write 연산 시간	59,45μs	60,13μs	-1.12%

자동생성 시스템을 이용한 설계는 해당되는 디바이스에 대한 정보를 이해하는데 걸리는 시간을 제외한 나머지 시간의 대부분을 절약할 수 있어 manual 설계에 비해 time-to-market을 크게 줄일 수 있다.

그림 8은 ARM922T 코어에 삼성 3.5인치 TFT-LCD를 장착하였으며 코어에 ARM-Linux 2.4.19를 탑재한 후 자동 생성된 디바이스 드라이버를 커널에 등록하여 구동한 화면을 보인다.

디바이스 드라이버 자동생성 시스템에 의해 자동 생성된 TFT-LCD 디바이스 드라이버는 개발자가 매뉴얼에 따라 설계한 디바이스 드라이버와 비교하여

그림 8. 자동 생성된 디바이스 드라이버 구동 화면.

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 구현한 인터페이스회로와 디바이스 드라이버의 통합 생성 시스템에 대해 제시하였다. 디바이스 드라이버 자동생성기는 기존의 디바이스 드라이버 개발 프로그램들과 다르게 OS Platform에 따라 디바이스 드라이버를 생성할 수 있는 알고리듬에 의해 구현되었으며, 인터페이스 회로와 디바이스 드라이버를 동시에 효율적으로 개발할 수 있다. 디바이스 자동생성 시스템은 GUI환경을 통해 입력 데이터를 쉽게 입력할 수 있으며, 생성된 디바이스 드라이버는 에디트 윈도우에 출력되어 수정과 저장할 수 있다. 자동생성 시스템은 자동생성을 위해 입력 데이터로 OS정보, 하드웨어 특징 정보 등을 입력하여 라이브러리를 통해 디바이스 드라이버 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 드라이버를 자동 생성하여 성능을 측정한 결과 인터페이스 회로와 디바이스 드라이버 통합 자동생성 시스템 설계 중 디바이스 드라이버 자동 생성은 매뉴얼 설계와 성능이 대등하며 매뉴얼 설계보다 개발기간을 단축하는 효과를 보였다.

현재 구현한 디바이스 드라이버 자동생성 시스템은 GUI를 통한 사용자가 요구되는 입력 데이터를 직접 입력하여 드라이버 코드를 자동 생성하는 방식이다. 하드웨어간 인터페이스 자동생성 시스템과 디바이스 드라이버 자동생성 시스템의 효율적인 연계를 위해 하드웨어간 인터페이스 자동생성 시스템을 통해 자동 생성된 인터페이스가 RTL 레벨의 HDL 코드이므로 추후에는 GUI를 통한 사용자 입력대신 HDL 코드를 통한 입력이 가능하도록 HDL 코드 파싱으로 디바이스를 분석할 수 있는 모듈의 추가가 요구된다.

감사의 글

본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT 연구센터 육성지원사업의 연구결과로 수행되었음.

참 고 문 헌

[1] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip Verification : Methodology and Techniques*, Kluwer Academic Pub., 2001.

[2] R. Ortega, L. Lavagno, and G. Borriello, "Models and Methods for HW/SW Intellectual Property Interfacing", in Proc. int. Symp. System- level Synthesis, Hsinchu, Taiwan, pp. 397-432, July 1998.

[3] A. Wenban, J. O'Leary, and G. Brown, "Codesign of Communication Protocols", Computer, *IEEE*, Vol. 26 No. 12, pp. 46-52, Dec. 1993.

[4] P. Chou, B. Ortega, and G. Borriello, "Interface Co-Synthesis Techniques for Embedded System", in Proc. *ICCAD*, San Jose, CA, pp. 280-287, Nov. 1995.

[5] 이서훈, 강경구, 황선영, "Protocol Mapping을 이용한 인터페이스 자동생성 기법 연구" *한국통신학회 논문지*, 제 31권, 8A호, pp. 820-829, 2006년 7월.

[6] 이서훈, 문종욱, 황선영, "FSM을 이용한 표준화된 버스와 IP간의 인터페이스 회로 자동생성에 관한 연구" *한국통신학회 논문지*, 제 30권, 2A호, pp. 137-146, 2005년 2월.

[7] A. Rubini, J. Corbet, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd Edition, O'Reilly Pub, 2005.

[8] 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.

[9] Y. Ma and C. Lim, "Test System for Device Drivers of Embedded Systems" in Proc. *ICACT*, Phoenix park, Gangwon-Do, Korea, Vol. 1, pp. 550-552, Feb. 2006.

[10] Jungo, Windriver, http://www.jungo.com/support/support_windriver.html

[11] Compuware, DriverStudio, http://www.compuware.com/products/driverstudio/default.htm

[12] Microsoft, Windows Driver Development kit, http://www.microsoft.com/whdc/devtools/ddk/default.mspx

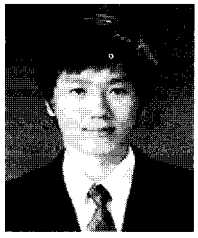
[13] Y. Choi, W. Kwon, and H. Kim, "Code Generation for Linux Device Driver" in Proc. *ICACT*, Phoenix park, Gangwon-Do, Korea, Vol. 1, pp. 734-737, Feb. 2006.

[14] 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, Kanazawa, Japan, pp. 302-310, Nov. 2000.

황 선 영 (Sun-Young Hwang) 정회원

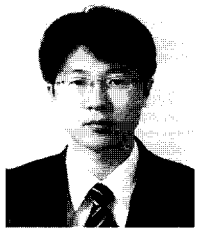
1976년 2월 서울 대학교 전자공학과 졸업
1978년 2월 한국 과학원 전기 및 전자공학과 공학석사 취득
1986년 10월 미국 Stanford 대학 전자공학 박사학위 취득
1976~1981 삼성 반도체 주식회사 연구원, 팀장
1986~1989 Stanford 대학 Center for Integrated System 연구소 책임 연구원
Fairchild Semiconductor Palo Alto Reaserch Center 기술 자문
1989~1992 삼성전자(주) 반도체 기술 자문
1989년 3월~현재 서강대학교 전자공학과 교수
<관심분야> SoC 설계 및 framework 구성, CAD 시스템, Computer Architecture 및 Embedded System Design 등

김 현 철 (Hyoun-Chul Kim) 준회원



2006년 2월 조선대학교 컴퓨터공학과 졸업
2006년 3월~현재 서강대학교 전자공학과 대학원 CAD & Embedded Systems 연구실
<관심분야> Automatic generation of device driver

이 서 훈 (Ser-Hoon Lee) 준회원



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