A Design Framework for Embedded Linux Drivers

Kuan Jen Lin, Shin Wen Chen and Jian Lung Chen

Department of Electronic Engineering, Fu Jen Catholic University, Taiwan

E-mail: kjlin@mails.fju.edu.tw

ABSTRACT

A device driver is a software layer to talk to peripheral hardware device. Because of the need for hardware knowledge and the lack of appropriate assistant tool, writing device drivers has always been tedious and error-prone. This paper proposes a design framework to facilitate the driver development for embedded Linux OS. The framework provides a structural design approach and automatic tools for design. It consists of a two stage design process. The first stage is to automatically generate a driver code skeleton from user-specified configurations. In the second stage, a user can complete the driver code using C language and our proposed Embedded Driver-Specific Language (EDSL). The driver code written in EDSL not only is more concise and easier verified, but also can be automatically synthesized to C codes. A preliminary implementation of the design framework has been applied to develop device drivers for an ARM7-based platform and obtained favorable results.

KEY WORDS: Embedded Linux, Device driver, Design framework, Software synthesis, Open source.

1. INTRODUCTION

A device driver is a software layer to talk to peripheral hardware device. It is a necessary part to release a new peripheral device and IC. Nowadays, the rapid progress in assistant tools has significantly reduced the development time for hardware parts, while the development of device drivers has made little progress. The availability of device driver often delays the release of a new device or IC. For Linux users, the problem is even more troublesome since the driver support for Linux generally is scheduled after for Microsoft OSs. According to a recent survey [7], the availability of device drivers becomes the most concern for designers of embedded systems to adopt Linux as their OS.

It is a tedious and error-prone task to write device drivers because it needs knowledge of target device hardware and technique of kernel-mode programming. Nevertheless, a device driver generally needs to be rewritten for different OSs and different processors [14]. As a result, the assistant tool and effective design approach to facilitate the development process have always been highly desirable. Current commercial assistant tools like Jungo’s WinDriver [15] and Bsquare’s WinDk [16] (basically specific to Microsoft Windows OS) provide a graphical user interface for specifying the main features of a driver and can automatically generate a code skeleton that is comprised of coarse-grained functions. Another facility of these tools is to provide libraries which wrap kernel functions and allow users to access hardware resource via user-mode programming. However, the approach incurs performance penalty and more difficulty to handle shared devices.

Most of research works in literature attempt to automatically generate fine-grained driver code from a high level specification and/or maximize the source-portability across different platforms. Various Domain-Specific Languages (DSL), like *Devil* and *ProGram*, are proposed as high level specifications to design device drivers [9, 11, 13]. A DSL is a programming language tailored for a specific application and provides more expressive power over the application domain. The generally claimed benefits of using DSL approach include higher-level abstraction, increased productivities, and better error reporting. However, as pointed out in [3], DSL also incurs a number of disadvantages such as the costs of education for users and the potential loss efficiency when compared with general programming language. The work [2] dealing with the interface HW/SW co-synthesis also addresses the task of automatically generating device drivers from a high level specification. However, the generated software driver is not in the context of an OS, but rather to emulate the signaling of bus interface in software. As for the portability, the industrial ongoing project UDI [17] separates the OS-dependent part and device-dependent part from a device driver and normalizes the interface between them and the rest of driver, which essentially is OS independent and device-vendor independent. The system proposed by Katayama et al. does a similar work, though only Unix-like OSs considered [8]. Our current work aims at supporting embedded Linux (uClinux) [19] running at ARM-7 platform only. The portability issue will not be studied in this paper.

This paper proposes a new design framework to facilitate the development of Linux device drivers. The framework provides a structural design approach and automatic tools for design. It consists of a two stage design process. The first stage is to automatically generate a driver code skeleton according to user-specified configuration. The driver skeleton is composed of a set of parameterized templates which contain coarse-grained functions. The second stage is based on our proposed Embedded Driver-Specific Language (EDSL) which offers specific expressive power over the driver domain and allows one to complete the driver code from the skeleton. Specifically, the EDSL is a kind of DSL, but it is embedded in the C language. The programming scheme retains the expressive power of C language for users as well as offers expressive power over the domain of device drivers. The driver code written in EDSL not only is more concise and easier verified, but also can be automatically synthesized to C codes. A preliminary implementation of the design framework has been applied to design device drivers for an ARM7-based platform and obtained favorable results.

The reset of the paper is organized in the following way. The next section overviews our design framework and introduces the input specification to our framework. Then the two primary components, skeleton-generator and EDSL approach, will be described in detail in Section 3 and Section 4 respectively. Preliminary assessment for our approach will be shown in Section 5. Final section concludes the paper and presents some line of future work.

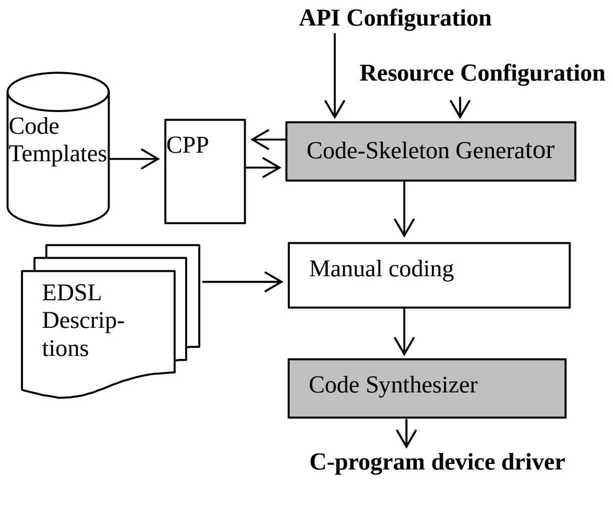

Fig.1: The design framework for embedded Linux driver.

2. SYSTEM OVERVIEW

Fig. 1 shows our design framework for embedded Linux device driver. The framework consists of a two stage design process. The first-stage process accepts API (Application Program Interface) and resource configuration files as input specifications. API is an abstraction of functions provided by the peripheral device, through that operation system (on behalf of application programs) can communicate with devices. Fig. 2 shows the operation scheme. The key idea behind the first-stage process is based on that Linux OS has defined a set of standard API function prototypes (such as *read*, *open* and *ioctl*) for various kinds of device [12], that is, character, block and network device. A practical driver is comprised of a subset of the API set. We prepare a code template for each API function of the character-type device (currently block and network devices having not been considered), which consists of optional codes and substitutable string variables. The API configuration file determines what subset of APIs and what specific features to be provided by a driver. An API function performed generally involves certain hardware resources, like IO registers, interrupt and DMA. Their features are defined in the resource configuration file.

API and resource configurations are parsed by the *skeleton generator* and relevant parameters derived are passed to a *C-preprocessor* to instantiate code templates. These templates are written in C language with preprocessing directives. Though there are more powerful preprocessing languages such as *perl* and *Ksh*,

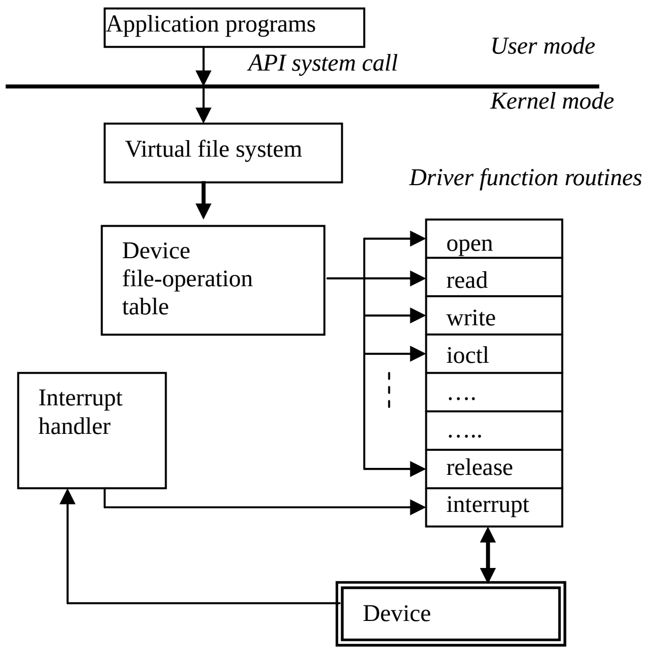

Fig. 2: The API system call and the Linux device driver.

NonBlock	; indicates whether nonblocking mode is used..
UseBuf	; indicates whether a buffer is used to hold input data.
BufSize	; the size of read buffer
BufFull	; indicates the approach used when the buffer is full (types: CirCular/Signal/Discard)
Access	; indicates the operation type to read data (e.g. directly read, read via DMA)

Fig. 3: A list of parameters used to specify the *read()* function.

C-preprocessor, which can perform string substitution and conditional statements, is enough for our need. All the generated codes templates are assembled to build a skeletal device driver. In addition, a header file and a Makefile are also generated in the process.

In the second stage, one can start from the code skeleton and continue to complete the driver with C language and our proposed EDSL. The EDSL is a small language and is expressive uniquely over the specific features of device drivers. The programming scheme attempts to retain the expressive power of C language for users as well as offers expressive power over the driver features. Presently, the EDSL provides three kinds of descriptive capability: *accessing registers*, *software-driven signaling* and *describing state machines.* They will be further detailed in a later section. After added with C statements and EDSL statements, the driver code is then fed into the *code synthesizer*. It transforms the EDSL statements to C codes and tries to optimize the code size and/or performance. The final output is a device driver in C language.

3. SKELETON GENERATOR

The skeleton generator accepts API and resource configurations as input specifications and outputs a skeletal driver that is built from a set of predefined code templates. This section firstly introduces the syntax and semantics of the configuration files and then describes the operation of the skeleton generator.

The API configuration file determines not only what kinds of API the driver supports but also what functional facilities every API provides. Roughly, the syntax of the API configuration can be stated as follows:

<file> :== { <param- line> } EOF

< param-line > :== <ApiName> { ',' <param-item> } '\n'

<param- item> :== <NAME '=' VALUE >

Each of Linux API functions has a unique ApiName, for instance, ReadApi for read() function. Associated with each ApiName, there are certain parameters to differentiate various operation modes and services that the API function provides. For example, a read() function can be either blocking or non-blocking and use buffer or not. Fig. 3 lists a set of parameters associated with the read() function and Fig. 4 shows a simple example file. It specifies that the read( ) function does not support nonblocking mode and uses a circular buffer with the size of 2048 bytes to hold data, which is moved to memory via a DMA operation. This configuration file also defines two *ioctl* commands. An *ioctl* command is defined by three parameters, namely *Cmd* (Command name) *name, Dir* (access direction) and *Argu* (the type of argument).

```
ReadApi    NonBlock=No, UseBuf = Yes,
ReadApi    BufSize = 2048, Access = DMA
IoctlApi Cmd=UART_BAUDRATE, Dir=WR, Argu=int
IoctlApi Cmd=UART_STATUS, Dir=RD, Argu=char
```

Fig 4. A sample of API configuration file.

The resource configuration of a device involves device registers, interrupt, DMA and IO memory. The device registers can be viewed as a programming interface for device function. Generally, a device register is partitioned into several bit fields each of which represents a certain attribute (i.e. status or configuration) of the device. In our framework, each bit field is called as a *device attribute* and each attribute can be accessed individually in EDSL statements. The configuration file defines the physical address of device register and the bit-range for each device attribute. Fig. 5 shows an example of resource configuration file for a UART device. The syntax of the file is the same as API configuration. In the example, the name of device is specified as “UART” and the physical base address 0x3ff0000. The device has a register which resides at the offset 0xD0000 and named as “UART_ULCON0”. The register contains five device attributes (IR, SC, PMD, STB and WL) whose bit positions are at bit 7, bit 6, bit 5~3, bit 2 and bit 1~0 respectively. Other bits are not used and denoted as “U”. Besides of register declarations, the features of interrupt and DMA used by the device are also specified in the file, which shows that the interrupt number used by the UART device is 7 and its type is “slow” (means its served priority is low). Furthermore, its corresponding service routine uses bottom-half mechanism [12].

```
Device Name=UART, IOBASE=0x03ff0000
INT IntNum = 7, Type=Slow, UseBH=Yes
DMA DmaChan=0
REG Name=ULCON0, Addr=0xD0000, \
   24:1:1:3:1:2=U:IR:SC:PMD:STB:WL
REG Name=BRDDIV0, Addr=0xD014, \
16:12:4=U:CNT0:CNT1
```

Fig 5. A sample of resource configuration file.

The first task of the skeleton generator is to parse the API and resource configurations and to pass the derived parameter values to the template instantiater (i.e. C preprocessor). The grammar of these two configuration files is rather simple (i.e. regular language). We have implemented its parser in C language. The parameter-value pairs parsed are given to the template instantiater. Each of standard API functions and device initialization procedure has a corresponding code template. These templates are written in C language with preprocessing directives. Specifically, it consists of optional codes and substitutable string variables. Fig. 6 is a template example of device initialization procedure. The template uses parameter "DEVNAME" to do several string substitutions and parameter "INT" to determine whether the codes about interrupt are included. According to the resource configuration in Fig. 5, the parser will issue the command, "gcc –E –DDEVNAME=uart –DINT –c template.c", to generate the needed code template. The second task of the skeleton generator is to collect all generated code templates into a file and to produce relevant header file and Makefile for the driver. The header file contains all the symbol definitions and function prototypes. For examples, INTNUM is defined as "7" and ULCON0 is defined as "IOBASE+0xD0000" in the header file.

4. EMBEDDED DRIVER-SPECIFIC LANGUAGE

Starting from driver skeleton, users can complete the driver in C language and EDSL. The EDSL is a kind of DSL, but it is embedded in the C language. The programming scheme retains the expressive power of C language for users as well as offers expressive power over the domain of device drivers. The descriptive capability of the EDSL provided is associated with the functional partition of a device, as shown in Fig. 7. It shows that a driver can be separated into three parts: bus interface, device attributes and device core. A bus is made up of both an electrical interface and a programming interface. EDSL currently provides capability to emulate electrical interface in software. The attributes (stored in registers) abstract the status and the configuration setting of the device. EDSL allows a user to manipulate them individually. The device core is responsible for performing device functions. EDSL currently provides state-machine description to help a driver to maintain the device resource. The followings give examples to illustrate their descriptions.

```
#define DEV_INIT(s) DEV_INITX(s)
#define INTISR(s) INTISRX(s)
#define FOPS(s) FOPSX(s)
#define STR(s) STRX(s)
#define DEV_INITX(s) int s## _init( void)
#define INTISRX(s)    s## _interrupt
#define FOPSX(s) &## s## _fpos
#define STRX(s) #s
int major = 0;
DEV_INIT(DEVNAME)
{
        int      result;
#ifdef INT
        if(request_irq(INTNUM, INTISR(DEVNAME), 0,
STR(DEVNAME), NULL)) {
             printk(STR(DEVNAME) " IRQ %d is not free.\n",
INTNUM);
                return -EIO;
        }
#endif
        /* Register a character device */
        if ((result = register_chrdev(major, STR(DEVNAME),
FOPS(DEVNAME))) < 0) {
                printk(KERN_WARNING STR(DEVNAME)
"can't get major %d\n",major);
                return result;
        }
}
```

Fig 6. A code-template of device_init( )

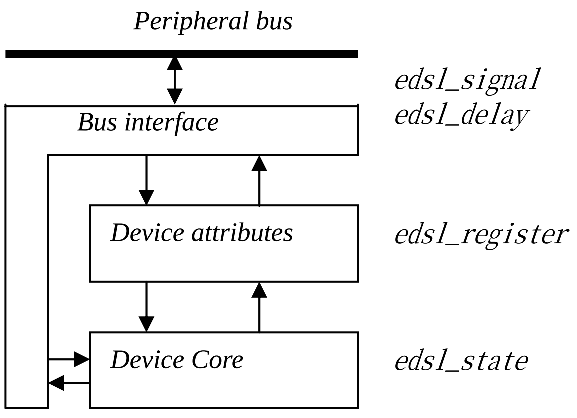

Fig. 7: A functional partition of a device and the associated EDSL facilities for each part.

Accessing registers: The registers of a device can be viewed as a programming interface of the device function. Through accessing registers, a driver can configure the device, read device status and do data transaction. As stated in a previous section, a register may contain more than one device attribute and each one can be manipulated individually. The manipulation of device attribute requires bit mask and shift operations which are error-prone in a general programming language such as C. A driver code generally consists of many such bit operations, which can represent up to 30% of driver code [9]. The EDSL allows users to specify the access of device attribute individually, and the bit operation and type consistency checking will be performed implicitly in our framework. The followings show a code segment of EDSL to access device attributes (referring to Fig. 4 and Fig. 5 configurations):

```
edsl_register(ULCON0_WL=0x2,
ULCON0_STB=StbLength,
ULCON0_OTH=UNCHAN);
edsl_register(baudrate = BRDDIV0_CONT0);
```

The above statements can be transformed to the followings:

```
        temp = *(ULCON0);
        temp = temp & 0xFFFFFFF8; /* Other
bits are unchanged */
        temp = temp | 0x2;
        temp = temp | (stbLength << 2) ;
       *(ULCON0) = temp;
       baudrate = (*(BRDDIV0) & 0x0000FFF0 )
>> 4;
```

The above transformation is performed by the code-synthesizer in the framework. Some kinds of optimization can be further explored in the transformation, such as the combining consecutive accesses of the same register into single access and explicitly register caching. They have not been included in our current framework.

Software-driven signaling: Embedded computing systems seldom adopt standard IO buses, such as PCI or ISA for PC systems, to communicate with hardware devices. The communication interfacing can be maintained by aided hardware or software. We call the emulation of bus interface in software to be *software-driven signaling.* It is applicable for low speed and low cost applications. The following EDSL statement requests a CPU to generate an active high signal having the duration of at least 200ms through the IO port ADC_RESET.

```
edsl_signal( ADC_RESET, Active = HIGH, DUR
= 200ms );
```

To realize the above action, the driver needs the invocation of timer and prepares associated handler. Our framework will automatically generate these codes. Furthermore, the insertion of delay between consecutive register accesses sometimes is required for low speed IO devices. This can be achieved by busy-waiting or invoking a timer. The edsl_delay() directive is proposed to specify the behavior.

Describing state machines: State machines are usually used to model protocol-based network layered devices and to manage shared resources [10]. EDSL provides a directive to describe a state table, as shown below:

```
edsl_state(S0, x >= 3 && y == 4, S1, do_play);
edsl_state(S1, x<= 2 || y != 4, S0, do_exit);
```

The four fields of the edsl_state statement represent *current state, conditions*, *next state* and *action performed before changing to next state*, respectively. Currently, we transform the state table to C codes using procedure scheme in [5]. In recent years, optimal code synthesis for control-dominated machine attracts much attention in EDA research community [1, 4, 6]. Partial results of them can be directly applied to our framework. However, for certain kinds of devices, a driver needs to distribute state management among different API functions, which needs additional techniques to optimize the codes. The optimization techniques have not been included in our framework.

5. ASSESSMENT

To assess the proposed design framework, a preliminary implementation has been applied to develop device drivers for an ARM7-based platform from Wiscore corporation [18]. The platform contains an SOC (Samsung 3C4510B) that includes an ARM7 as CPU and most of peripheral devices, such as timer, DMAC and UART. We connect a LCD to the platform for experiment. Two drivers under uClinux OS for UART and LCD respectively have been designed for the assessment. The UART device is programmed to issue a DMA request each time when it receives one byte data. Its driver should collect data in a buffer and provide blocking read. The driver for LCD involves many *ioctl* calls to set character attributes and needs delay-insertion statements because the response of LCD is rather slow. Both drivers heavily rely on register access. In addition, our code template for read( ) function contains the codes for DMA transfer. Therefore, about 80% of both driver codes can be covered by the generated driver skeleton and EDSL statements.

6. CONCLUSION AND FUTURE WORK

We have presented a new design framework to facilitate the development of embedded Linux device drivers. The framework provides a two stage design process and automatic tools for design. The first stage is to automatically generate a driver skeleton based on a set of parameterizable templates. The second stage is to allow users to complete the driver with the new proposed EDSL. The driver code written in EDSL has been shown more concise and easier verified. Furthermore, the EDSL codes can be automatically synthesized to C language. A preliminary implementation of the design framework has been applied to design device drivers for an ARM7-based embedded platform and obtained favorable results.

The future works include the exploration of optimal code synthesis for EDSL and the implementation of an integrated GUI development environment based on our design framework.

7. ACKNOWLEDGMENTS

This work was supported by the National Science Council, Taiwan, R.O.C. under grant NSC 91-2213-E-030-017.

8. REFERENCES

[1] Felice Balarin et. al., “Synthesis of Software Programs for Embedded Control Applications,” *IEEE Trans. on CAD*, vol. 18, no. 6, pp. 834-849, June, 1999.

[2] P. Chou, R. B. Ortega and G. Borriello, “Interface Co-synthesis Techniques for Embedded Systems,” *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 280-287, 1995.

[3] A. Deursen et al., “Domain-Specific Languages: An Annotated Bibliography,” *ACM SIGSOFT Software Engineering Notes*, Nov. 2001.

[4] S. Edwards, “Compling Esteral into Sequential Code,” In *Proc. of DAC*, 2000.

[5] Paul Fischer, “State Machines In C,” *C/C++ Users Journal*, Dec, 1990

[6] Y. Jiang and R. K. Brayton, “Software Synthesis from Synchronous Specifications Using Logic Simulation Techniques,” In *Proc. of DAC*, pp. 319-324, 2002.

[7] R. Lehrhaum, “The State of Embedded Linux,” presented in *Embedded Linux Expo & Conference*, June 2001.

[8] T. Katayama, K. Saisho and A. Fukuda, “Prototype of the Device Driver Generation System for UNIX-like Operating Systems,” *Proceedings of International Symposium on Principles of Software Evolution*, pp. 302 –310, 2000.

[9] F. Merillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller, “A DSL Approach to Improvee Productivity and Safety in Device Drivers Development,” *Proceedings of the 15th International Conference on Automated Software Engineering*, 2000.

[10] Thomas Nelson, “The Device Driver as State Machine,” *C/C++ Users Journal*, March, 1992.

[11] M. O’Nils and A. Jantsch , “Operating system sensitive device driver synthesis from implementation independent protocol specification,” *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pp. 563-567, 1999.

[12] A. Rubini and J. Corbet, *Linux Device Driver*, 2nd Edition O’Reilly, 2001.

[13] S. A. Thibaut, R. Marlet and C. Consel, “Domain-Specific Language: From Design to Implementation Application to Video Device Drivers Generation,” *IEEE Tran. On Software Engineering*, pp. 363-377, May/June 1999.

[14] E. Tuggle, “Writing Device Drivers,” *Embedded Systems Programming*, pp. 42-65, Jan 1993.

[15] Jungo Ltd, *WinDriver V5 Users Guide*, URL: http://www.jungo.com

[16] Bsquare, *WinDk Users Manual*, URL: http://www.bsquare.com.

[17] .UDI Core Specification, http://www.project-udi.org.

[18] http://www.wiscore.com.tw

[19] http://www.uclinux.org.