An Extension of C Preprocessor Directives for Device Programming

Kuan Jen Lin, Jian Lung Chen, and Chuang Hsiang Huang

Department of Electronic Engineering, Fu Jen Catholic University, Taiwan

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

ABSTRACT

Writing device driver has always been tedious and error-prone. Traditionally, C programmers exploit preprocessor directives to facilitate the driver development. In this paper, we follow this programming style and extend preprocessor directives to improve the development. The extension makes the C driver code more readable and concise. Furthermore, our complier provides stronger type-checking capability than original C compiler. Current assessment for an embedded Linux environment shows favorable results.

KEY WORDS: Device driver, Programming language, Preprocessor, Linux, Open source.

1. INTRODUCTION

A device driver is a software layer to talk to peripheral hardware device. Writing device drivers needs knowledge of target hardware (understanding device specification) and involves many low-level instructions such as direct memory access and bit-operation [10]. Traditionally, the C language is the most commonly used to write device drivers due to its support of such kinds of instructions. However, such operations in C are not checked for type-correctness. Moreover, they are fairly unreadable. Because of lacking appropriate assistant tool and programming style, writing device drivers has always been a tedious and error-prone task. Furthermore, for modern multi-tasking OSs like Windows and Linux, the driver is run in kernel-mode as a portion of OS. This makes it hard to debug and one small error, like some bit of a device register being set wrong, may cause the whole system to fail. As reported in [1], device drivers have been noted as a major source of faults in operating system code. Hence, the reliability of device drivers is critical to system stability.

To facilitate the development process and improve the reliability, a variety of approaches have been suggested from both industrial and academic communities. Current commercial assistant tools like Jungo's WinDriver and Bsquare's WinDk provide a graphical user interface for specifying the main features of a driver. They can automatically generate a code skeleton which is comprised of coarse-grained functions and libraries which wrap kernel functions. Most of research works in literature attempt to automatically generate fine-grained driver code from Domain-Specific Languages (DSLs). A DSL is a programming language tailored for a specific application and provides more expressive power over the application domain. The language GAL developed by Thibault et al. [9] is designed for specifying X Windows video driver. Though the driver code is reduced about 90%, GAL covers a very restricted domain. The language Devil developed by Mérillon et al. [6] is designed for more general classes of devices. Its specification is complied into a set of C procedures (or in-line function) for accessing registers and manipulating data. These procedures are called in a traditional C driver code and prevent programmers from directly dealing with low-level codes. The languages Dveil+ [11] and NDL [3] both follow the Devil and attempt to propose a complete language to replace General Programming Languages (GPL) like C for writing device drivers. Although their languages have higher level ab-

straction of device behaviors, they incur less flexible programming capability than GPLs. Interface HW/SW co-synthesis works [2, 7, 12] also address the automatic generation of driver codes for dedicated device behaviors that can be interpreted by their modeling.

Traditionally, C programmers exploit preprocessor directives to simplify the driver development [4]. In this paper, we follow this programming style and propose a set of Extended C Preprocessor (ECP) directives to further facilitate the development. The extension makes the C driver code more readable and concise. Furthermore, our complier provides stronger type-checking capability than original C compiler. Current assessment for an embedded Linux environment will show favorable results.

The approach presented in this paper is a part of our design framework methodology described in our earlier publication [5]. A modified framework will be overviewed in the next section. Section 3 will introduce a device model and *device variable* to abstract register data. Section 4 will present the grammar of ECP. Preliminary assessment will be shown in Section 5. Final section concludes the paper and presents some line of future work.

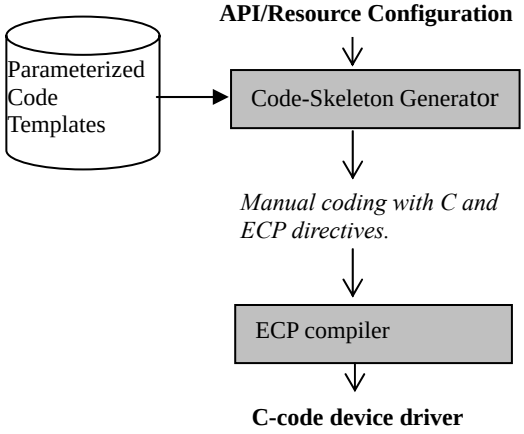

Fig.1: The design framework for Linux driver.

2. DESIGN FRAMEWORK

Fig. 1 shows our design framework that defines a design flow and supports assistant tools for Linux driver programming. It consists of a two stage design process. The inputs are user-specified configurations for API calls as well as system resources (such as DMA channel and interrupt vector) associated with the device. API is an abstraction of functions provided by the peripheral device, through that operation system (on behalf of application programs) can communicate with devices. Linux OS has defined a standard API function prototype for each API routine. We prepare a set of parameterized code templates for these routines. The templates are developed on a driver model which interprets device behavior in terms of control flow and data flow. The parameters are used to define what a subset of API functions provided and what operation mode used in both control and data flows. The program *skeleton generator* parses the configuration file to extract the parameter values. Then it builds the driver code skeleton from the set of code templates. Basically, a skeleton is comprised of a set of the API routines, together with initialization and interrupt routines. All the routines contain not only the function interface but also the required statements to serve operations in both control and data flows. The details can be found in [13].

In the second stage, one can start from the generated code skeleton and continue to complete the driver with C language and our proposed ECP directives. It has always been a popular programming style to write device driver with the help of C preprocessor [5]. Our extension is expressive uniquely over the specific features of device drivers, which will be elaborated in later sections. The program *ECP compiler* translates preprocessor statements to C codes. The final output is a device driver in C language, which then can be fed into *gcc* to get executable codes.

3. DEVICE VARIABLES

The main tasks of a device driver are to configure the device operation mode and observe its status to manage the data transfer. The real physical part interacting with the driver is a dedicated controller (an IC or IP) that manages

the peripheral device, as shown in Fig. 2. In the IO controller, the registers can be viewed as a programmable interface to the driver and the kernel part performs the functionality provided by the device. Through the registers, the driver configures the functions of the device and observes its statuses. For example, a typical UART controller contains a set of registers for setting data format and transfer rate (baud rate), as shown in Fig. 3. The data format is determined by the values of PMD, STB and WL, each of which occupies a bit range within the register ULCON0. They are referred to as *device variables.* To access and manipulate device variables involve low-level instructions such as direct memory access and bit-operation. Such operations in C are not checked for type-correctness and are fairly unreadable. ECP provides more readable and concise expressions to allow them to be read or written like any variable in C. For example, if we want to set the number of stop bits per frame to be 2 (i.e. STB=1) while keeping other variables unchanged, a typical sequence of C codes might be written as follows:

```
temp = *(ULCON0);
temp = temp & 0xFFFFFFFB | (1 <<2);
*(ULCON0) = temp;
```

Using ECP directives, you can write a simple assignment in C to get equivalent result.

```
STB = 2;
```

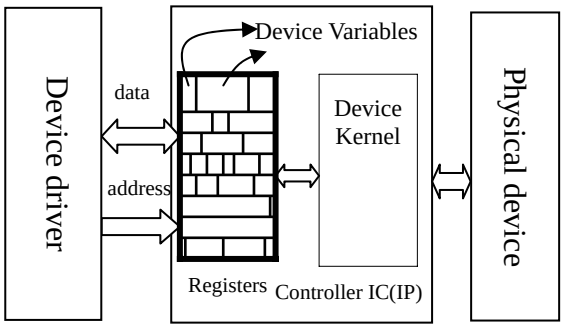

Fig. 2: A peripheral device model.

Group Variable

To set a function or get a status often involves several relevant device variables. We call the set of variables as *group variable*. For example, if we want to set a data format for UART communication, it is often to set PMD, STB and WL simultaneously. We can let a variable group "UARTFRAME" be composed of PMD, STB and WL. The ECP directives allow us to use the following statement to set the group.

```
#set(UARTFRAME [ODD, TwoS, Bit8])
```

And use the following statement to get the group.

```
#get(UARTFRAME[ x, y, z])
```

Concatenated Variable

A device variable could comprise several fragments residing in different registers. For example, the setting of baud rate is determined by three values: the *X* in ULCON0 register (it selects external clock MCLK or internal clock UCLK as a reference clock), the CNT1 and CNT0 in UBRDIV0 register. The calculation is shown in Fig. 3. If we concatenate the three variables to be a new variable *BaudRate,* a simple assignment can be used to set its vale as follows:

```
BaudRate   = U2400;
```

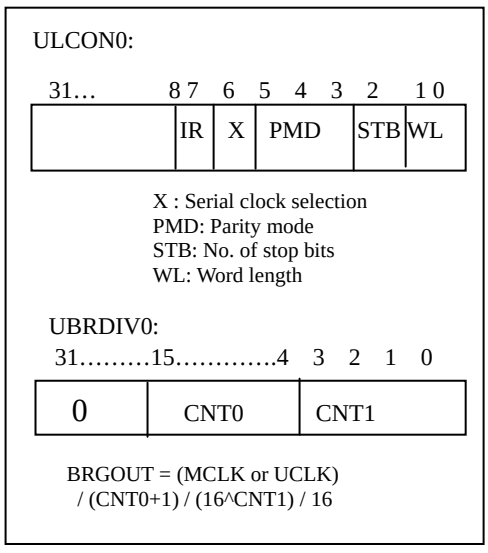

Fig. 3: UART registers (a real case from Samsung 3C4510B SOC).

```
driver →device (device | getfun | setfun | flush)*
device → #dev   dev_line   dev_content   #enddev
dev_line →   dev_id ( reg_bitwidth )   iobase_addr
dev_content →   (reg_statement | convar_statement | gupvar_statement)+
reg_statement →   #reg   reg_line   reg_content   #endreg
reg_line →   reg_id ((R|W|RW))  offset   (bank_id[index]) ?
reg_content → ( var_statement)+
var_statement →   #var   var_line   ( (var_content)+   #endvar )?
var_line →   ( - | var_id) reg_id [ bit_position ]
var_content → val_id   :=   num
convar_statement →   #convar   convar_line ( (convar_content)+ #endconvar)?
convar_line → convar_id ( covar_member (,convar_member)* )
convar_member →   var_id
                | convar_id
                | reg_id [ regbit_num ]
convar_content → conval_id   :=   ( num | val_id | conval_id)(, ( num | val_id | conval_id) )*
gupvar_statement → #gupvar   gupvar_id ( gupvar_membert (,gupvar_member)* )
gupvar_member → ( var_id | convar_id )
getfun → #get ( gupvar_id [ c_var_id (, c_var_id )* ] )
setfun → #set ( gupvar_id [ gupvar_para (, gupvar_para)* ] )
flush   → #flush
```

Fig. 4: The main portion of ECP grammar.

4. ECP DIRECTIVES

This section will describe the main portion of ECP grammar and take statement examples to show its application. Fig. 4 summarizes the grammar and Fig. 5 gives statements in ECP directives to specify a UART device shown in Fig. 3.

The syntax for device declaration begins with the nonterminal *device*, as shown in Fig. 4. The directive **#dev** is used to define the name of a device, the bit-width of its registers and its base address (referring to line 1 in Fig. 5). The syntax for register declaration begins with nonterminal *reg_statement*. The directive **#reg** (line 2) is used to define the name, RW attribute, the address offset and an optional bank-register ID. The RW attribute specifies the register to be read-only, write-only or read-write. The offset plus the device's based address equals the physical address of the register. In some devices, several sets of registers are mapped to the same memory addresses. Each set usually is called a bank. A bank register is used to select which set to be used. The UART device shown here does not use such addressing. The lines between **#reg** and **#endreg** define all the device variables within the register. Each variable definition begins with directive **#var**. The statement in line 4 of Fig. 5, defines "PMD" to represent the value in bit5~bit3 in register ULCON0. Following the declaration, an enumeration of identifiers that represent all the permissible values of the variable can be defined (line 5~10). Such a structure can be thought of as enumerated type in C. Only these values in the type can be assigned to the device variable. Our ECP compiler performs a type-checking to avoid other values. The facility provides more safety type-checking than C language. If no type definition exists, the variable accepts any positive integer less than 2^N, where N is its bit length.

After the variable declaration, we can write a simple assignment to access the variable. The following statement assigns "TwoS" to variable STB to configure the UART to use two stop bits in a date frame:

STB = TwoS;

ECP compiler will convert the statement into the following:

```
*(0x03ffd000) = *(0x03ffd000) &
        0xFFFFFFFB | (1 <<2);
```

The part between **#convar** and **#endconvar** (line 26~31) defines a concatenated variable *Baudrate* and its enumerated data type, only a portion shown here. We can use the following simple expression to set value.

BaudRate = U2400;

ECP complier converts the statement into:

```
*(0x03ffd000) = *(0x03ffd000) &
    0xFFFFFFFB | (1 <<6);
*(0x03ffd014) =867 << 4 | 1 ;
```

The directive **#gupvar** is used to define a group variable. As shown in line 32, the group *UARTFRAME* is composed of *PMD*, *STB* and *WL*. Then we can use the following statement to set their values:

#set(UARTFRAME [ODD, TwoS, Bit8])

This statement is converted into:

```
*(0x03ffd000)= *(0x03ffd0000) &
    0xFFFFFFFC0 | (( 4<<3) | (1 << 2 ) | 3);
```

Reading the group at a time is specified as follows:

#get(UARTFRAME[x, y, z])

This statement is converted into:

```
temp = *(0x03ffd000);
x = (temp & 0x38) >> 3;
y = (temp & 0x04) >> 2;
z = (temp & 0x03);
```

The statements accessing device variables in the same register can be grouped into one instruction if their access orders do not affect the device behavior. The directive **#flush** is used to tell the compiler that all the statements preceding it must be completed before doing succeeding statements.

```
1   #dev   UART(32)   0x03FF0000
2   #reg   ULCON0(RW)   0xD000
3   #var  X      ULCON0[6]
4   #var  PMD ULCON0[5:3]
5         Disable:= '0**'
6         Odd   := '100'
7         Even  := '101'
8         As1   := '110'
9         As0   := '111'
10  #endvar
11  #var   STB  ULCON0[2]
12        OneS  := 0
13        TwoS  := 1
14  #endvar
15  #var   WL   ULCON0[1:0]
16        Bit5   := 0
17        Bit6   := 1
18        Bit7   := 2
19        Bit8   := 3
20  #endvar
21  #endreg
22  #reg  UBRDIV0(RW)    0xD014
23  #var  CNT0        UBRDIV0[15:4]
24  #var  CNT1        UBRDIV0[3:0]
25  #endreg
26  #convar BaudRate(X,CNT0,CNT1)
28        U1200 := 1,1735,1
29        U2400 := 1,867,1
30    …….
31  #endconvar
32  #gupvar UARTFRAME(PMD, STB, WL)
33  ……..
34  #enddev
```

Fig. 5: A portion of the device declaration for UART.

5. ASSESSMENT

The ECP compiler has been written in C using Flex and Berkeley Yacc to generate a front-end parser. To assess the utility of ECP directives, we have implemented four drivers, as listed in Table 1. The first two are designed for a Samsung 3C4510B-based platform, which runs under uClinux. The UART_DMA is a UART device with DMA transfer mode. The LCD is a 16 character by 2 line device. The last two are designed for an X-Hyper250B platform, which

runs under Linux 2.4.18 kernel. The RTC (Real Time Clock) device is used to configure a clock source with a wide range of frequencies. The I2C device enables the processor to communicate with I2C peripherals. Table 1 shows the result in terms of lines of code. All original codes are designed in C. The number of lines of C preprocessor (CPP) is counted for only the part dealing with the target devices. The number of lines of configuration file used to specify API and system resource is given in "CF" column. The number of lines in ECP directives is given in "ECP" column. The "C" column in ours indicates how many lines of C code are required to finish the driver, not including the code produced by the skeleton generator and ECP compiler. The required numbers in average are significantly shorter than the original ones.

Table 1: Code comparison

Device	Original		Ours		
	C	CPP	CF	ECP	C
UART_DMA	43	19	6	220	2
LCD	107	29	5	64	65
RTC	424	26	23	24	317
I2C	666	31	14	90	617

6. CONCLUSION

Writing device driver has always been tedious and error-prone. In this paper, we propose en extended set of C preprocessor directives to facilitate the driver development. The extension makes the C driver code more readable and concise. Furthermore, our complier provides stronger type-checking capability than original C compiler. Current assessment for an embedded Linux environment shows favorable results. We are studying more real device drivers and wish to give more comparisons to demonstrate the effectiveness of our design framework.

7. ACKNOWLEDGMENTS

This work was supported by the National Science Council, Taiwan, R.O.C. under grant NSC 92-2213-E-030-022.

8. REFERENCES

[1] A. Chou et al., "An Empirical Study of Operation System Errors," *Proceedings of the 18th ACM Symposium on Operation System Principles,* pp. 73-88, 2001.

[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] C. L. Conway and S. A. Edwards, "NDL: A Domain-Specific Language for Device Drivers," *Proceedings of 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*.

[4] A. Mansky, "Using The C Preprocessor For Device Control," *C/C++ Users Journal*, Dec. 1990.

[5] K. J. Lin, S. W. Chen and J. L. Chen," A Design Framework for Embedded Linux Drivers," NCS, Taiwan, 2003.

[6] 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.

[7] 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.

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

[9] S. A. Thibault, 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.

[10] E. Tuggle, "Writing Device Drivers," *Embedded Systems Programming*, pp. 42-65, Jan 1993.

[11] Q. L. Zhang, M. Y. Zhu and S. Y. Chen, "Automatic Generation of Device Drivers," ACM SIGPLAN Notices, pp.60-69, June 2003.

[12] Shaojie Wang, Malik S., Bergamaschi, R.A, "Modeling and Integration of Peripheral Devices in Embedded Systems," *DTAE*, pp.136-141, 2003.

[13] J. L. Chen, "Assistant Tools and Language for Device Programming," MS. Thesis, Fu Jen Catholic University, 2004.