Insights into Transforming a Linux Wireless Device Driver to Run on a Bare Machine

William Agosto-Padilla, Ramesh Karne and Alexander Wijesinha

Department of Computer & Information Sciences, Towson University, Towson, U.S.A.

Keywords: Linux, 802.11 Wireless, Device Driver, Bare Machine, Transformation.

Abstract: Developing device drivers is often tedious and error-prone. Drivers for bare machine applications that run without an OS have been previously written by directly using specification documents. Transformation is an alternate approach that attempts to develop a bare machine driver by using existing code for an OS driver. We investigate the transformation of a Broadcom Linux 802.11n wireless device driver to a bare machine driver. As a first step towards understanding the transformation, we study the functions and OS dependencies of the Linux driver. The driver module is analyzed by examining its interaction with other modules, providing details of its key elements and code sizes, and by comparing code sizes with its counterpart Windows driver. We also discuss design issues that will be useful for developing device drivers that are independent of any operating system, kernel, or embedded system.

1 INTRODUCTION

Bare machine computing applications (Karne et al., 2005; Karne et al., 2013a) run on the hardware without any operating system (OS) support. Device drivers they use must therefore be written so that they are independent of any OS. Many bare machine drivers have been written by directly using the relevant standards or specification documents. They include a USB device driver, 3Com/Intel network interface card drivers, and an audio card driver (Karne et al., 2013b; He et al., 2008; Ford et al., 2009; Khaksari et al., 2007; Appiah-Kubi et al., 2012; Rawal et al., 2011). We investigate an alternate approach for developing bare machine drivers that transforms an existing OS driver to run on a bare machine. It is possible to use a driver for one OS with another OS. For example, NDISwrapper (NDISwrapper) enables Linux to use a Windows driver.

Transforming an OS driver to a bare driver requires that OS calls in the driver be eliminated or replaced with bare interfaces. We have previously transformed a Windows SQLite database engine to run on a bare machine (Okafor et al. 2013). However, transforming a typical OS-based driver such as a network driver to run without OS support appears to be harder. To gain insight into the driver

transformation problem, we have undertaken a study of a popular wireless Linux driver: 802.11n Broadcom b43 (b43-Linux Wireless). This particular wireless driver was selected for transformation since its source code is readily available. Source code for wireless drivers is often proprietary; as a result, some Linux drivers have been reverse-engineered from their Windows equivalents (Broadcom Wireless-Arch Wiki; Corbet, 2011).

An important goal in driver transformation is to avoid understanding low-level functionality and details of the driver source code as much as possible. Yet, it is first necessary to determine how the driver module (b43.ko in this case) communicates with external modules and the Linux kernel. In this paper, we analyze and classify system-related aspects of the Linux b43xx wireless driver in order to gain insight into this question. The rest of the paper is organized as follows. The Linux wireless driver module system view is presented in Section 2. The key code details and structures are illustrated in Section 3. The module interactions are described in Section 4. The OS/device driver interfaces and system calls are shown in Section 5. Related work is discussed in Section 6, and driver design issues are outlined in Section 7. The conclusions are given in Section 8.

2 SYSTEM VIEW

Linux OS modules are based on object code interfaces instead of a binary executable. That is, a module is not linked until its run time, or is dynamically inserted into the kernel. While the module approach is very convenient and efficient for kernel management, it is not suited for bare machine transformation. The dependencies of a Linux driver module are somewhat complex and it is not possible to easily decouple the module from the Linux kernel. In general, the kernel provides a variety of services including process management, memory management, module management, dynamic linking and error checking. There is no kernel in a bare machine and a bare application manages memory and implements its own task scheduler.

Fig. 1 shows a system view of the Broadcom b43 wireless driver for Ubuntu 3.9.3. In this figure, we have labeled the key interfaces as #KSC and #EMC and the data symbols as #EDO for convenience. The significance of the numbers shown alongside the edges is explained later. The wireless module b43.ko, which incorporates essential driver functionality, has many dependencies on other modules and the kernel. In particular, b43.ko interacts with BCMA (Broadcom Microcontroller Architecture), SSB (Sonics Silicon Backplane), skbuf (socket buffer structure), LED (Light Emitting Diode), DMA (direct memory access), PRINT (print facilities) and IEEE802.11 (wireless standard). In addition, b43.ko also uses some system calls that are provided by the OS.

#KSC: only kernel or system calls
#EMC: external module calls
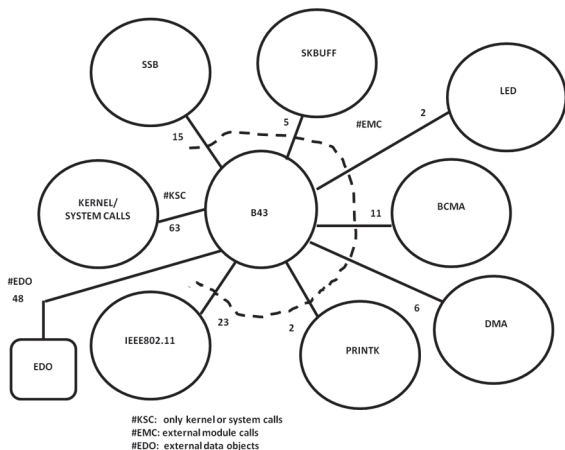#EDO: external data objects

Figure 1: System View (Linux Module).

The b43.ko module exports symbol names (functions and variables) in order to provide a global scope so that other modules can use these interfaces.

The module also has certain parameters defined externally and passed to the module when it is instantiated. These variables can change the internal behavior of the module using #ifdef statements.

Table 1: B43 Source Files.

bus.c	Bus Abstraction
debugfs.c	Debug
dma.c	DMA
leds.c	LED Control
lo.c	Local Oscillator
main.c	Main
pcmcia.c	PCMCIA Card Interface
phy_a.c	PHY -a
phy_common.c	PHY common
phy_g.c	PHY -g
phy_ht.c	PHY -h
phy_lcn.c	PHY -l
phy_lp.c	PHY -lp
phy_n.c	PHY -n
pio.c	Parallel I/O
radio_2055.c	Radio Dev Data Tables
radio_2056.c	Radio Dev Data Tables
radio_2057.c	Radio Dev Data Tables
radio_2059.c	Radio Dev Data Tables
rfkill.c	Radio Enable
sdio.c	Secure Data Card Int
sysfs.c	Virtual File System
tables.c	Radio Device Data Tables
tables_lpphy.c	Radio Device Data Tables
tables_nphy.c	Radio Device Data Tables
tables_phy_ht.c	Radio Device Data Tables
tables_phy_lcn.c	Radio Device Data Tables
wa.c	Work arounds
xmit.c	TX/RX Functions

In order to test the b43.ko module, we need the whole system running as a single monolithic executable. A variety of techniques were used to test the module including PRINTK statements, log traces, Objdump, KGDB, and intercepting interrupts using sys_call_table.

3 CODE VIEW

The "linux-source-3.9.3" version for Ubuntu 12.10 version has 29 "C" source code files and "30" header files. Each source file has a corresponding header file and the extra header file b43.h consists of data definitions, constants and inline functions for the b43 module. In addition, there are 41 header files for libraries and system calls. The b43 source files are classified into groups based on their functionality in Table 1. The total source code size is 45,483 lines

and the header code size is 7,446 lines, amounting to a total of 52,929 lines. In Fig. 2, the code percentages for the b43 code groups identified in Table 1 are shown. It can be seen that this device driver covers many features and options that are not required for a typical usage of the driver. For example, LED Control, Debug, and PCMCIA were not used when we tested the module with a desktop PC running Ubuntu. When we eliminate code sizes that are not relevant to a typical application, the total code size becomes 35,943 lines (that is 68% of the original code).

Figure 2: Percentage of Code Sizes per Group.

We also analyzed the b43.ko module using the Source Insight tool (sourceinsight.com). The module summary it gave for b43.ko is shown in Table 2. It indicates the number of functions (899), function prototypes (220) and external variables (43). Constants, functions and structure members are seen to be the largest contributors to b43. The b43.h header file contains many data structures, definitions and system header files. The components of this header file and their relationships can be analyzed by using the Doxygen tool (stack.nl) as shown in Fig. 3.

Table 2: B43 Module Dissection.

Summary of Items on b43 driver
Structures (109) Macros (68)
Unions (10) Function Prototypes (220)
Enumerations (24) Method Prototypes (23)
Constants (4275) Structure Members (891)
Enum Constants (118)
Variables (257)
External Variables (43)
Functions (899)

There are 18 system header files in this header

including 802.11mac.h, ssb.h, and kernel.h. The relation between the 802.11 physical layer header files phy_a.h, phy_g.h, phy_common.h, and nl802.11.h is shown on the right side of the figure.

Figure 3: b43.h Header Structure.

The 899 functions found in Table 2 were also classified based on their functional unit as defined in Table 1. The number of functions in each unit is shown in Fig. 4.

Figure 4: Percentage of Functions per Group.

In this classification, the largest unit is phy (42%). If we ignore functions that are not used in our desktop PC and b43.ko wireless module connection with security turned off, we only need 583 functions (which are 65% of total functions

implemented in the module). We also generated a log file with PRINTK statements in each function and found that only functions in the files dma.c, leds.c, main.c, phy_common.c, phy_n.c, xmit.c, and b43.h (inline) were activated. These functions include a total of 440 out of 899, which is 49% of the total functions implemented in the module. This implies that the b43.ko module has about 50% of functions that may not be used at a given time for a given setup.

We have also compared the Linux wireless driver with a Windows version of its counterpart (WMP300Nv1.sys). The source code for this Windows driver is not available.

Table 3 shows a comparison of assembly source lines of code and number of "call" statements in the Windows and Linux drivers. The Linux driver assembly code is almost three times smaller as it is a module and requires other interfaces at execution. The number of CALL statements in the code includes local calls and external interfaces.

Table 3: (LoC, # of Calls) for Linux and Windows.

Driver	Assembly LoC	Number of CALLs
Linux	53,863	7340
Windows	160,490	9798

4 EXTERNAL INTERACTIONS

While the code views shown in the previous section show the structure of the b43.ko module, they do not reflect module dependencies and kernel interactions. There are many tools that can be used to identify the external interactions and OS related calls for this module. For example, "objdump" (available in Linux and Windows) has a variety of options that give useful information about the object module. Similarly, GNU's "nm" utility provides undefined symbols (UND) that indicate system calls and other library interfaces. In addition, the Linux command "/proc/kallsyms" in Linux lists all symbols in the kernel.

The parameters passed to the module can be obtained using: "ls –l /sys/module/b43/parameters". In this module there are nine parameters that can be passed to the module (bad_frames_preempt, btcoex, fwpostfix, hwpctl, hwtkip, nohwcrypt, pio, qos, and verbose). We removed the module from the kernel using "sudo rmmod b43", and checked the exported symbols using "cat /proc/kallsyms | grep b43_".

Surprisingly, we found that three symbols (b43_pci_bridge_driver, b43_pci_bridge_tbl, and b43_pci_ssb_bridge_exit) still remained in the "/proc/kallsyms" file after removal of the module. It is possible that these three symbols may have been defined by other modules using the b43_ label since there are no such symbols referred to in the b43.ko module. We inserted the module into the kernel using "sudo modprobe b43" and checked the exported symbols again. This time, there were 771 symbols in the exported table. These symbols consist of 471 functions, the rest being data symbols. The 471 functions are exported from the b43.ko module for other modules to use. We did not find any EXPORT statements in the b43 source code. So it appears that the command "modprobe" generated these EXPORT symbols during the insert or dynamic link time.

To understand the 471 functions above and to get more details on b43.ko, we used "objdump" and "nm". The command "objdump –t b43.ko > objdump.txt" gives all symbols in the b43.ko module. There were 998 lines captured in the objdump.txt file. Likewise, the command "grep g objdump.txt > globals.txt" gives all global symbols in the b43.ko module, which amounted to 195 symbols. The command "grep .text globals.txt > globalfun.txt" showed that there are 147 functions in the module. The rest of the global symbols (195-147 = 48) are global data parameters that consist of the .data and .rodata sections of the module (#EDO in Fig. 1). The command "grep UND objdump.txt > syscalls.txt" identified 127 system calls that were needed in the module. Similar results were also obtained by using the "nm" command. The command "nm –g b43.ko > nmext.txt" resulted in 322 global symbols. The command "grep U nmext.txt > nmsyscalls.txt" also showed that there are 127 system calls in the module, which served to confirm this number. The 127 external system calls include all the interfaces shown in Fig. 1 except #EDO. The equivalents of these 127 kernel/system calls (#KSC and #EMC interfaces in Fig. 1) must be provided to transform the Linux b43 wireless driver into a wireless driver that can run on a bare machine.

We also collected more data for the BCM43xx Windows wireless driver WMP300Nv1.sys, which is not open source. Fig. 5 shows its system view and relation to HAL (hardware abstraction layer), NDIS (network driver interface specification) and Kernel. All these components are DLL elements that work with the wireless driver. The HAL component has 3 interfaces: acquire spin lock, release spin lock and stall processor. NDIS has 41 interfaces, which are

wrapper interfaces enabling the driver to run in a Linux environment. There are 29 kernel interfaces, which are similar to the #KSC interfaces in Fig. 1. The Windows driver has 73 external interfaces or system calls compared to 127 for the Linux driver. More interfaces are needed in Linux, although it is a module and not an executable. We determined that 8 interfaces here are similar to #KSC in Fig. 1, while the rest are unique to Windows. There are less system calls in the Windows driver compared to the Linux driver since Windows encapsulates more functions into its driver. The DLL approach in Windows is based on executable modules instead of object modules, enabling the driver to be more self-contained.
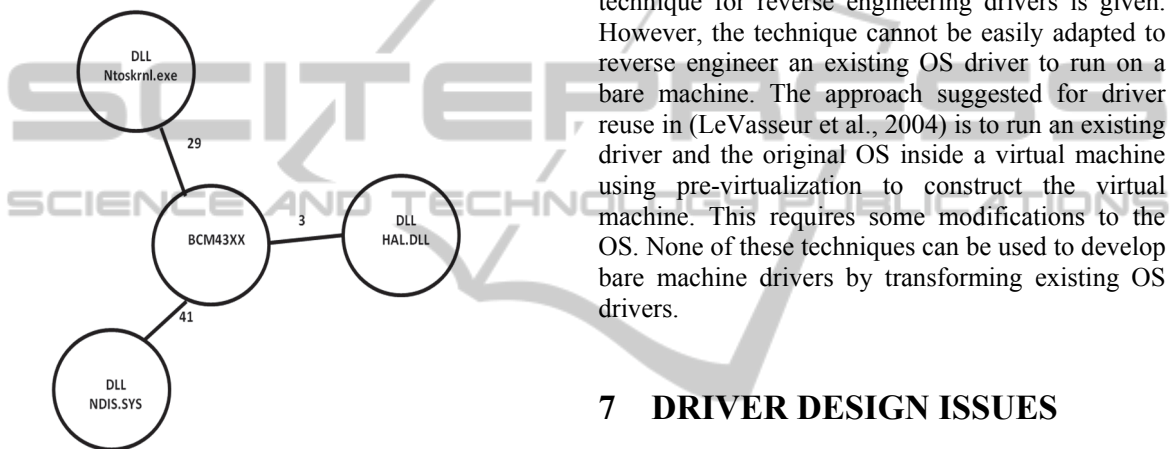
Figure 5: System View (Windows Driver).

5 KERNEL INTERACTIONS

The #KSC, #EMC interfaces and #EDO data symbols identified in the previous section enable the b43.ko module to interact with the Linux kernel and other modules. The 127 system interfaces identified for b43 consist of 63 #KSC and 64 other interfaces (15 SSB, 23 IEEE802.11, 5 SKBUFF, 6 DMA, 2 LED, 11 BCMA, and 2 LED) as shown in Fig. 1. Similarly, the 48 EDO interfaces consist of 4 .data objects, 43 .rodata (read-only data) objects, and 1 .gnu.linkonce.this_module object. These objects represent local and remote data objects.

6 RELATED WORK

Many studies have been conducted to understand the design and internals of device drivers. In (Kadav and Swift 2012), an in-depth study of Linux device drivers is undertaken. Static analysis tools are used to analyze driver code and determine how drivers interact with the OS and the hardware. The study does not consider a particular driver or wireless driver. In (Amar et al., n.d.), driver development using hardware abstraction and APIs for hardware and software interfaces are discussed. While their Device Object Model is useful for separating OS and device-specific components of drivers, it does not provide any insights for transforming an OS driver so that it becomes OS independent. The focus of (Boyd-Wickizer and Zeldovich 2010) is a system that enables Linux drivers to run in user space so that they are able to limit the impact of attacks on drivers. In (Chipounov and Candea 2006), a technique for reverse engineering drivers is given. However, the technique cannot be easily adapted to reverse engineer an existing OS driver to run on a bare machine. The approach suggested for driver reuse in (LeVasseur et al., 2004) is to run an existing driver and the original OS inside a virtual machine using pre-virtualization to construct the virtual machine. This requires some modifications to the OS. None of these techniques can be used to develop bare machine drivers by transforming existing OS drivers.

7 DRIVER DESIGN ISSUES

The b43.ko module is designed to operate with the Linux OS. Instead of using an NDIS-like wrapper, an alternate approach that will enable a b43 driver for one OS to be used with another OS and on a bare machine is to provide appropriate and equivalent interfaces to #KSC, #EMC, and #EDO. Ideally, the wireless device driver specification can be an abstract data model that is analogous to other standards such as the USB standard or SCSI standard. The USB standard is particularly convenient as many devices already use USB interfaces. This will allow all wireless commands to be wrapped inside a USB command payload and the driver to execute these commands while hiding the complexity.

A wireless device driver can then be designed as a self-contained object that provides a high-level API to applications including the kernel. This API does not micro-manage the wireless driver. That is, its API can be directly invoked by an application programmer, and there is no kernel or embedded system running in the machine. Also, providing a standard set of operations such as Initialize(), Reset(), Read(), Write() and Configure() will

simplify the design of both wireless and wired network interface device drivers.

Another issue related to Linux and b43.ko interdependencies is the complexity of the module architecture. The b43.ko module has a large number of external interfaces making it difficult to decouple the driver module from its operating environment. By modeling a wireless device driver as a self-contained object it becomes possible to eliminate all other module interactions and provide a standard interface directly to the application programmer. This approach was used to develop device drivers for a variety of bare machine applications. These drivers require no OS, kernel, or embedded system to support their operation.

The b43.ko module also incorporates general functionality that is rarely used in a typical operating environment. Such functionality makes the module itself complex and large in size. This in turn makes it harder to understand the driver, port it, and test its operations. Bare machine drivers can solve these problems; however, more research is required to understand how to transform existing OS drivers so that their OS dependencies are removed.

8 CONCLUSIONS

We studied the b43.ko Linux wireless driver module and discussed its system view and code view, and also its external interactions with other modules. The code and functions used in the module were classified based on their type of operation and functionality. Design issues identified with the design of b43.ko provide a starting point for future development of device drivers that are independent of any OS. The driver details provided in the paper will be useful for developing techniques that can transform OS drivers to bare machine drivers.

REFERENCES

Appiah-Kubi, P., Karne, R. K., Wijesinha, A. L., 2012. A bare PC TLS webmail server. In *ICNC*.

Amar, A., Joshi, S., Wallwork, D., Generic driver model using hardware abstraction and standard APIs. Available from: <http://www.design-reuse.com/articles/18584/generic-driver-model.html>. [12 Dec 2014].

Boyd-Wickizer, S., Zeldovich, N., 2010. Tolerating malicious device drivers in Linux. In USENIX ATC.

Broadcom Wireless-Arch Wiki. Available from: <http://wiki.archlinux.org/index.php/broadcom_wireless>. [12 Dec 2014].

b43-Linux Wireless. Available from: <http://wireless.kernel.org/en/users/Drivers/b43>. [12 Dec 2014].

Chipounov, V., Candea, G., 2006. Reverse engineering of binary device drivers with RevNIC. In *Eurosys*.

Corbet, J., 2011. Broadcom's wireless drivers, one year later. Available from: <http://lwn.net/Articles/456762/>. [12 Dec 2014].

Doxygen. Available from: <http://www.stack.nl/~dimitri/doxygen/>. [12 Dec 2014].

Ford, G. H., Karne, R. K., Wijesinha, A. L., Appiah-Kubi, P., 2009. The design and implementation of a bare PC email server. In *COMPSAC*.

He, L., Karne, R. K., Wijesinha, A. L., 2008. Design and performance of a bare PC web server. *International Journal of Computers and Their Applications (IJCA)*.

Kadav, A., Swift, M. M., 2012. Understanding modern device drivers. In *ASPLOS XVII*.

Karne, R. K., Jaganathan, K. V., Ahmed, T., Rosa, N., 2005. Dispersed Operating System Computing (DOSC). In *Onward Track OOPSLA*.

Karne, R. K., Wijesinha, A. L., Okafor, U., Appiah-Kubi, P., 2013a. Eliminating the operating system via the bare machine computing paradigm. In *Future Computing*.

Karne, R. K., Liang, S., Wijesinha, A. L., Appiah-Kubi, P., 2013b. A bare PC mass storage USB driver. *International Journal of Computers and Their Applications (IJCA)*.

Khaksari, G. H., Wijesinha, A. L., Karne, R. K., He, L., Girumala, S., 2007. A peer-to-peer bare PC VoIP application. In *CCNC*.

LeVasseur, J., Uhlig, V., Stoess, J., Gotz, S., 2004. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI*.

NDISwrapper. Available from: <http://en.wikipedia.org/wiki/NDISwrapper>. [12 Dec 2014].

Okafor, U., Karne, R. K., Wijesinha, A. L., Appiah-Kubi, P., 2013. A methodology to transform OS based applications to a bare machine application. In *TrustCom/ISPA/IUCC*.

Rawal, B., Karne, R., Wijesinha, A. L., 2011. Mini web server clusters based on HTTP request splitting. In *HPCC*.

Source Insight Program Editor and Analyzer. Available from: <http://www.sourceinsight.com>. [12 Dec 2014].