DriverGen: Automating the Generation of Serial Device Drivers

Jiannan Zhai[1](✉), Yuheng Du[2], Shiree Hughes[1], and Jason O. Hallstrom[1]

[1] Institute for Sensing and Embedded Network Systems Engineering, Florida Atlantic University, 777 Glades Road, Boca Raton, FL 33431, USA
{jzhai,shughes2015,jhallstrom}@fau.edu

[2] School of Computing, Clemson University, Clemson, SC 29634, USA
yuhengd@clemson.edu

Abstract. Microprocessors operate most serial devices in the same way, issuing commands and parsing corresponding responses. Writing the device drivers for these peripherals is a repetitive task. Moreover, measuring the response time of each command can be time-consuming and error prone. In this paper, we present DriverGen, a configuration-based tool developed to provide accurate response time measurement and automated serial device driver generation. DriverGen (i) simulates the command execution sequence of a microprocessor using a Java program running on a desktop, (ii) measures the response time of the target device to each command, and (iii) generates a device driver based on the received responses and measured response times. To evaluate DriverGen, three case studies are considered.

1 Introduction

Our work is motivated by the recurrent structure of most serial device drivers and the importance of accurate timing. The main contributions of our work are as follows: (i) We present a serial device driver configuration language that generalizes the specification of a serial device driver. (ii) We present an approach that measures response times with precision on the order of 10 s of microseconds by monitoring data signals in the communication interface. (iii) We implement DriverGen, a configuration-based tool developed to accurately measure response times, and to automatically generate the specified serial device driver. (iv) Finally, we evaluate DriverGen, considering the performance of generated drivers for three serial devices.

2 Related Work

Automated driver synthesis is discussed in [8]. Ratter proposes synthesis as a method to ensure correct driver construction. A state machine is generated automatically using specifications for both the device and the (desktop) operating system, and ultimately supports the generation of a driver for the device in C. We similarly provide the ability to generate a microprocessor driver for device communication. When generating a driver for a microprocessor, we experience

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2016
B. Mandler et al. (Eds.): IoT 360° 2015, Part II, LNICST 170, pp. 325–330, 2016.
DOI: 10.1007/978-3-319-47075-7_37

the added challenges of memory and power constraints, timing precision, and a single-threaded operating system. Our driver must be efficient with respect to both memory usage and power consumption.

Another method for automating device driver generation is Termite [9]. Termite acts as an interface between the OS and a target device. It uses a formal specification of the device to generate a set of OS-independent commands. It allows the device creator to focus on the device, and the OS expert to focus on the OS, and still create a communication link between the two. Similarly, we create a method to automatically generate drivers for serial devices, eliminating the need for developers to manually write the drivers.

In [6], O'Nils et al. show that by using synthesis, development time can be reduced by as much as 98 %. Their method uses ProGram, a specification language, to model the behavior of a device based on sequences of permissible events. Three inputs are required to synthesize the device driver from its behavior: architecture independent protocols, a specification of the processor and bus interface, and a specification of the target operating system.

An important requirement of automatically generated code is that the quality must be equal to or surpass that of hand-written code. In [7], O'Nils et al. argue that their tool produces a quality driver (generated in C) that is comparable to handwritten code. This tool requires a protocol specification for both the device and the operating system.

3 System Design/Implementation

DriverGen is based on the observation that all serial device drivers work in almost the same way. Our system simulates the execution of each command and generates the target device driver based on the execution results. Each command sequence is implemented as a function in the driver. To match the response pattern and save the desired information, we implement regular expression libraries in Java and C, used by DriverGen and the generated drivers, respectively. To determine if the target device is responding, or the response is finished, timeouts are used, making accurate timing important. DriverGen monitors the UART communication signals to measure the response time of a device to each command (response time), and the time between bytes in the response (inter-byte times).

3.1 Hardware Setup

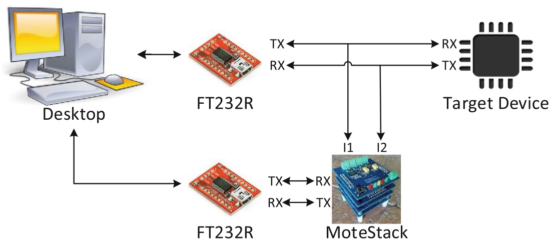


Fig. 1. Hardware setup

The DriverGen hardware, shown in Fig. 1, consists of a desktop running a Java program, two FT232R chips, and a MoteStack [3]. The FT232R chips are used by the desktop to communicate with the target device and a

MoteStack, respectively. The MoteStack is used to monitor the UART data signals to measure the response and inter-byte times.

3.2 Driver Configuration

DriverGen runs based on a driver configuration file that is used to configure UART communication, control execution of each command, and generate the target driver. The configuration parameters specify (i) basic driver information, such as driver name, version; (ii) global definitions, such as response timeout, which specifies the maximum time before the first response byte should be received; and (iii) function details, such as function names, the commands to be sent to the target device, the responses expected, and other information.

3.3 System Architecture

The DriverGen system consists of three modules. The *Parser* module is used to read, parse, and validate a driver configuration. The *Executor* module is used to execute the functions specified in the configuration, and to control the MoteStack to measure response times and inter-byte times. The *Generator* module is used to generate the driver source code based on the configuration parameters and execution results.

4 Evaluation

We now present our evaluation of the driver generation approach. We introduce three serial devices and corresponding applications previously developed to operate with functionally equivalent, time-tested, handwritten drivers. We validate the correctness of each generated driver via substitution within the corresponding application. Finally, we consider the relative performance of the drivers, both in terms of space and execution speed.

In our experiments, the drivers and applications are implemented based on the AVR platform. To evaluate the WiFi and cellular devices, a standard x86 server is used to collect data sent from the devices.

4.1 Test Devices and Applications

Three serial devices are used to evaluate our approach. The WH2004A is an LCD device that executes commands to display characters. The RN131 is a standalone embedded WiFi device with built-in TCP/IP support. The GM862 is a quad-band GSM/GPRS cellular modem with built-in TCP/IP, FTP, and SMTP support.

To evaluate the generated driver for the WH2004A, an application which detects a door trespassing event and displays the event counts on the LCD is used. Since the WH2004A does not respond to incoming commands, the evaluation is focused on correctness only. The generated driver displayed the event counts without any errors for 100 door trespassing events.

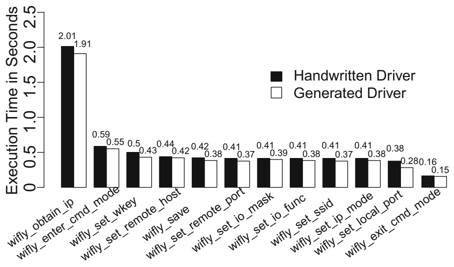
To evaluate the generated drivers for the RN131 and GM862, two test applications were used. The applications sense data from a group of sensors every 10 and 120 s, respectively, and record the execution time of each function. Sensor readings and execution times are then sent to a server. Each application is configured to perform 1000 transmission rounds in each test, and the average is used. Based on stored messages in the database, both drivers work as expected.

4.2 Performance Evaluation

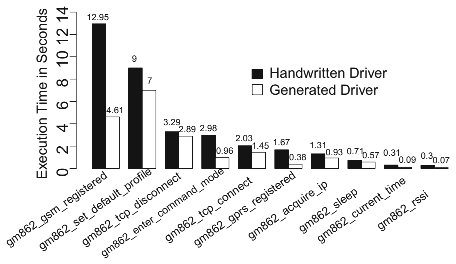
We next evaluate the performance of the generated drivers relative to the handwritten drivers, both in terms of space and execution speed. We focus on the WiFi and cellular devices.

Execution Speed. We first evaluate the execution speed of the generated drivers by sending 1000 850-byte messages to the server and tracking the execution time of each associated function. Figures 2a and b summarize the speed of the generated driver functions for the RN131 and the GM862 compared to the handwritten drivers. The x-axis represents the driver functions, and the y-axis represents the average cumulative execution time, in seconds, in a single transmission round. The functions are ordered by execution time, in decreasing order from left to right. As the figures illustrate, the generated drivers run faster than the handwritten drivers across all functions. The speed-up is achieved by reducing the time spent waiting for each response. The cumulative speed-up is proportional to the number of executions of each function in a transmission round. For example, in each round, the `gm862_gsm_registered` function executes approximately 40 times before detecting a valid network registration. Therefore, it shows a high speed-up in Fig. 2b. For the GM862, the overall execution time in each round is 48.50 s for the generated driver, compared to 59.60 s for the handwritten driver. For the RN131, the overall execution time in each round is 11.99 s for the generated driver, and 14.68 s for the handwritten driver.

Memory Usage. We next evaluate the memory overhead introduced by the generated drivers. *Avr-size* is used to collect the memory data. Figure 3a summarizes the drivers' program memory (ROM) usage. The x-axis represents the drivers, and the y-axis represents size, in bytes. The hashed area represents ROM overhead introduced by the regular expression library. The ROM overhead is approximately 3400 bytes for both drivers. Figure 3b summarizes the drivers' data memory (RAM) usage. Again, the x-axis represents the drivers, and the y-axis represents size, in bytes. The hashed area represents RAM overhead introduced by the regular expressions used in the generated driver. The RAM overhead is closely related to the number of regular expressions used. Since the GM862 requires more regular expressions, the overhead for the GM862 is slightly larger than the WiFi chip, at 503 bytes.

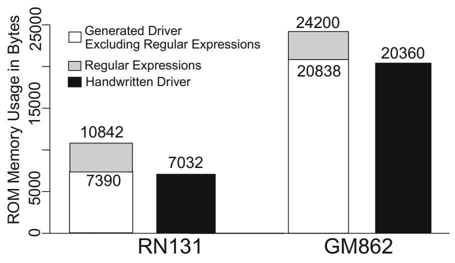


(a) RN131

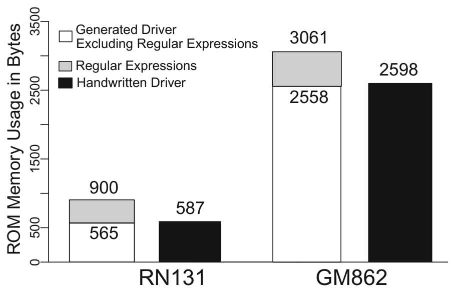


(b) GM862

Fig. 2. Driver function execution time



(a) Driver ROM Usage



(b) Driver RAM Usage

Fig. 3. Memory usage

5 Conclusion

We described a configuration-based system to automatically generate serial device drivers and accurately measure the timeout characteristics associated with each driver command. Results show that the generated drivers perform as expected, introducing modest memory overhead. Importantly, the execution time of each command is reduced compared to the handwritten drivers. As a result, driver performance is increased, and improved energy efficiency is achieved.

Acknowledgments. This work is supported by the NSF through awards CNS-1541917 and CNS-1545705.

References

1. CESANTA. SLRE: super light regular expression library, September 2013. slre.sourceforg.net/
2. Chou, P., Ortega, R., Borriello, G.: Synthesis fo the hardware/software interface in microcontroller-based systems. In: Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1992, pp. 488–495. IEEE Computer Society Press, Los Alamitos (1992)

3. Eidson, G.W., Esswein, S.T., Gemmill, J.B., Hallstrom, J.O., Howard, T.R., Lawrence, J.K., Post, C.J., Sawyer, C.B., Wang, K.C., White, D.L.: The south carolina digital watershed: end-to-end support forreal-time management of water resources. IJDSN, 1 (2010)
4. Li, J., Xie, F., Ball, T., Levin, V., McGravey, C.: Formalizing hardware/software interface specifications. In: Procceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, pp. 143–152. IEEE Computer Society, Washington (2011)
5. Locke, J.: Jakarta regexp Java regular expression package, April 2011. jakarta.apache.org/regexp/
6. O'Nils, M., Jantsch, A.: Operating system sensitive device driver synthesis from implementation independent protocol specification. In: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, pp. 562–567 (1999)
7. O'Nils, M., Jantsch, A.: Device driver and DMA controller synthesis from HW/SW communication protocol specifications. Des. Autom. Embed. Syst. **6**(2), 177–205 (2001)
8. Ratter, A.: Automatic device driver synthesis from device specifications. The University of New South Wales, November 2012
9. Ryzhyk, L., Chubb, P., Kuz, I., Le Sueur, E., Heiser, G.: Automatic device driver synthesis with termite. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 2009, pp. 73–86. ACM, New York (2009)
10. Shier, P., Garban, P.L., Oney, A.: System and method for validaitng communication specification conformance between a device driver and a hardware device. US2005246722 (2005)