Reverse-Engineering a Closed-Box Hardware and Software Linux Embedded System

Rafael Zurita, Rodolfo del Castillo, Miriam Lechner, Eduardo Grosclaude

Departamento Ingeniería de Computadoras - Facultad de Informática
Universidad Nacional del Comahue
{rafa,rdc,mtl,oso}@fi.uncoma.edu.ar

Abstract— **Linux embedded systems are often found in closed-box products. The present work documents the process by which such a closed-box, off-the-shelf product, has been analysed to study how it is built, how it is powered by Linux, and how its software can be updated or modified to adapt the device to other usages. The final result, after the described procedure is applied, is a generic, embedded Linux development board, suitable for arbitrary purposes. Our practical experiments targeted an Encore ENTC1000 product marketed as a thin client device for remote desktops.**

I. Introduction

Dozens of embedded devices such as toasters, mobile phones, refrigerators, vacuum cleaners, TV decoders, clocks, home network routers, game consoles, are used in daily life. Most of these, unlike domestic PCs, are bound to perform a single function. This single function makes them qualify as embedded systems [REF].

However, these household items are no longer as simple as they were once. Some years ago, most of these devices were just mechanical and electrical artifacts. Today's appliances integrate electronic circuits, sensors, microcontrollers, microprocessors, and are capable to perform a sophisticated array of functionalities.

Such complexity comes along with the need to have the device run an operating system, as several functionalities need to be concurrently managed. Vendors opt for Linux in many cases due to low cost and easy customisation [REF ELECTROLUX BRAZIL ANDROID ROUTER].

In our region, acquisition of embedded Linux systems for research or development is a complex and expensive prospect. If we were to build, say, a sophisticated robotic device, then our best strategy would be reusing as many parts as possible from other previous work, and choosing off-the-shelf, ready-made hardware and software components to be repurposed.

Many embedded devices are available in our regional market for study or customization. Most of them come with the required hardware to run an embedded Linux system, although they not always make this claim. Typical devices capable to run a Linux system are TP-Link home routers or Android-equipped mobile phones.

The ENTC-1000 embedded system is a proprietary product of Encore Electronics Inc. [5], working as a remote desktop terminal supporting RDP (Remote Desktop Protocol) and XWindow protocols. Commercial brochures and product documentation provide minimal specs, i.e. the device is marketed as a Linux 2.4 embedded system[1]. However, there is no additional documentation to architecture or software internals. Encore Electronics Inc. does not provide the source code to the installed software, nor any compiling and installing instructions as required by the GPL licence Linux is released under[2].

As the ENTC-1000 device is a modern embedded system, available for purchase in our country, understanding its architecture and behavior is of interest to our Department. Knowing its internals will allow us an inexpensive way to experiment with embedded Linux for academic purposes.

II. Legal Background

When repurposing a closed-box hardware and software system, the knowledge we need about its internals can be obtained either by having a complete written documentation of the target device, or by directly experimenting with it. This experimentation is called reverse engineering and can be driven by several activities and tools. This is the method we used for this case study.

As reverse engineering happens to be illegal in some countries, a question is raised about the legal status of our procedure. In our particular case, the goal in using reverse engineering is exactly to reuse the hardware for a different purpose. This goal matches the terms in the act *Ley Argentina N° 24.240 de Defensa del Consumidor*, which allows this action provided the product has been purchased legally.

III. Architecture of an Embedded Linux System

A. Hardware Architecture

We must know first the hardware architecture in order to understand the behavior of an embedded Linux device. In a modern embedded system, the main component is a microcontroller or microprocessor (CPU). Nowadays, many systems integrate a system-on-a-chip (SOC) device containing the CPU and most of an embedded system's components, in a single chip [3].

We also need to know about other components existing in the system. Coming after the CPU, most important components

[1]This kernel version was released in 2001.

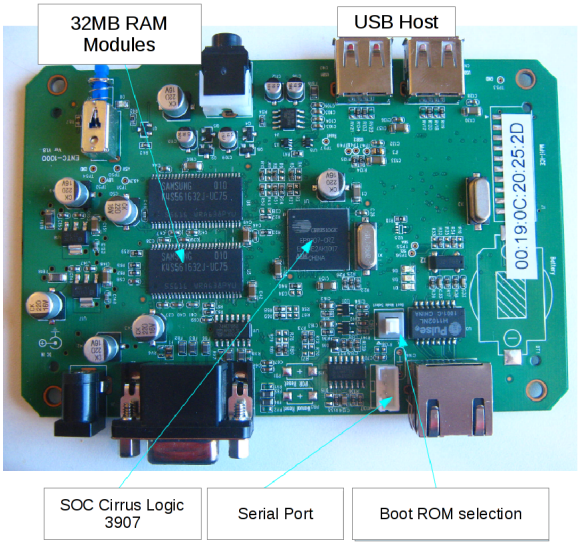

Figure 1. Encore ENTC-1000 board

are memory modules (RAM and Flash types) and communication modules (such as serial or USB). These devices, together with the CPU, give an overview to what the boot process or operational regime of the system will be like.

In practice, the architecture of an embedded system can be known by detaching the circuit board from the enclosing case. This enables us to visually inspect and list the main system components. Most of the time, from this listing we can get the needed ID of components or chips so as to reach its public documentation. This documentation is known as data sheets or, sometimes, programming manuals.

As we disassembled the ENTC-1000 (see Figure 1), we identified a Cirrus EP3907A SOC as the main component. This chip encloses an ARM920T main processor running at 200MHz. The board also holds two 32MB Samsung SDRAM memory modules and one 16MB Intel Flash embedded module.

Although Encore does not public any documentation, Cirrus Logic does. Cirrus has released detailed documentation of the Cirrus 3907 SOC [6]. Cirrus Logic also distributes programming software and Linux source code for experimentation. Our search was completed by finding documentation for experimental lab boards published by Cirrus Logic. These design papers are very useful to firms employing Cirrus Logic's SOC, as they can take them as a reference design for their own developments.

B. *Software architecture*

On the software side, the architecture typically consists of three essential components:

- A bootloader for an embedded system, usually Das U-Boot [4]. Other bootloaders are closed source boot managers.
- A Linux kernel, or, when the system lacks a memory management unit (MMU), a uClinux (also called μClinux) kernel [**?**].
- a minimalistic filesystem containing, at the very least, a C library (usually uClibc or eglibc) and a busybox shell [**?**]. Busybox is a small executable program optimized for embedded systems. Busybox is capable of performing roughly the same work as many basic UNIX utilites like ls, cp, etc.

Sometimes a fourth software component set is found in the system. These are the vendor's own applications and hardware drivers. Vendors' applications are seldom of interest for our research; however, drivers are of prime importance. If closed source drivers are present in the system, then the work of preparing an embedded Linux system supporting the whole hardware will be utterly difficult, due to the lack of access to source code. In our experience with ENTC-1000 we did not find any closed source drivers, so this issue is beyond the scope of this article.

The work done at this stage was preparing two complete replacement embedded Linux systems: Amstrong and emDebian. The documentation for preparing embedded Linux systems is broadly and publicly available (REFERENCIAS). There are dozens of Linux distributions for embedded systems, or development environments for embedded systems. With these tools, preparing a system targeted to a specific device is an easier task. Some of these projects are Openwrt, Buildroot or OpenEmbedded (REFERENCIAS).

Once we have the knowledge of how to build (i.e. compile, configure) and operate the software components for the target architecture, we must determine whether the original firmware can be modified.

IV. Software access

A. *Serial interface*

There are several ways to access an embedded Linux system. The most usual way is through a serial console. In Linux systems, the serial console allows to look at the kernel error messages and to interact with the boot manager.

Most embedded systems have a UART (Universal Asynchronous Receiver-Transmitter) controller, which is used mainly for debugging and development, but is also used by Linux as a predetermined serial interface. The UART translates information in parallel form, coming from the system bus, to data in serial form, so as to be transmitted through different ports. In embedded systems, usually the UART controller is a part of the CPU, so that, even when there is no exposed connector in the board, the controller is there.

During our work with ENTC-1000, the pins of the UART were not identified or labeled, but present. We identified the ground (GND) and direct current (VCC) pins by means of a multimeter and an oscilloscope. Then, using the GND pin, the rest of the exposed pins in the board were tested to identify transmission (TX) and reception (RX) lines.

To establish that the UART lines we had found were the right ones, we connected an RS-232 level converter to them, and then to a PC. We finally used the minicom communications program to obtain some visual information. This information

was, however, meaningless as the operating serial transmission speed for the original firmware was unknown.

B. Taking control

Once we knew the software components, we conducted access tests to the original system, at least at three levels in the architecture :

- Through TCP/IP connections, as soon as the original firmware is operating;
- by connecting to the boot manager;
- by accessing the boot ROM software.

The goal at the first two stages is to get root access to the closed Linux system. For the boot manager, we seek to access the minimal command interpreter offered by some boot managers. The first two test cases did not yield useful results. We went on deeper in the architecture, trying to reach for operating stages earlier to the execution of the boot manager.

Using the available hardware architecture and programming documentation, we analyzed the hardware reset process and the way the ENTC-1000 internal CPU begins executing instructions.

Communication with the CPU

As stated in the documentation at hand, Cirrus Logic EP9307 SOC has a Boot ROM memory which supports taking the UART as a program source to initialize the processor. On system reset, the ARM920t CPU begins the execution of code at address 0. The SOC honors the "Hardware Configuration" controls to select the device that appears at address 0. We found a switch labeled "boot mode select" on the ENTC-1000 board, with which we experimented to select Boot ROM, thus mapping it to address 0 to be executed.

Upon a system reset, Cirrus Logic EP9307 will work together with the Boot ROM software in a mode called "serial download" which allows us to send some code bytes to be executed by the CPU. The sequence of steps is as follows:

1) Initialize UART1 to 9600 baud, 8 bits, no parity, 1 stop bit
2) Output a "<" character over the serial connection
3) Read 2048 characters from UART1
   - If our serial peer does not send 2048 bytes, go on with boot process using internal Flash memory
   - If serial peer sent 2048 bytes, store them in internal buffer, output ">" through serial link to indicate successful read
4) Turn on Green LED
5) Jump to the start of the internal Boot Buffer

After several reset tests, the minicom communication program on the PC got the "<" character. This confirmed some hypotheses: the wired connections between PC and board were functional, and the Boot ROM code was being executed. Afer that, it was able to download a specific program as the first code to be executed by the ARM core.

We took a small sample program in ARM assembly language from Cirrus Logic as a start. Once assembled, the resulting object version for this program is smaller than 2048 bytes. By incrementally modifying the program source[2], we were able to discover the missing information about the architecture.

We were mainly interested in the addresses where the software components (bootloader, Linux kernel and the root file system) could be found inside the Flash memory. Hence, when executed, our program looks for the physical addresses of the internal Flash memory. To do so, the program reads consecutive memory locations, starting from the physical address 0x60000000, looking for the string "CRUS". We chose the 0x60000000 address based on possible values listed on the SOC documentation. The "CRUS" token is defined as the keyword for the CPU to identify the boot address when probing different addresses for booting code.

We loaded our small probe program through the serial download feature of the Boot ROM process. The "CRUS" token was found on the Flash memory at the address 0x60001000. As the CPU, during the boot process, will try to execute any code written after that memory location, it was a proper place to locate the bootloader component software.

Our last goal during this stage was to find the last valid physical address for RAM memory. After modifying our probe program to do some few trial and error tests, the program found that limit, and we modified the U-Boot bootloader RAM settings accordingly.

C. Booting an Alternative Linux Embedded System

Knowing how to execute a small program after a hardware reset event, and having the RAM and Flash addresses, we were ready to deploy an alternative Linux embedded system over the device. Our first step to accomplish this was to prepare a small bootloader to be sent into RAM through the serial line. We chose U-Boot, configured for this particular device.

After we successfully executed the bootloader, and were logged into its minimal -but powerful enough- command line, we had to decide how to re-layout software within the Flash memory, hence modifying the original software, to achieve our goal.

Using U-Boot TFTP [3] utilities and networking capability, we transferred a Linux 3.X kernel [4] to a particular address in RAM. Now the two main pieces of software (bootloader and kernel) required to boot a system were ready on RAM.

The next step was to copy RAM content over the Flash. For this we used a copy (cp.b) command available in U-Boot command line, the physical RAM addresses where U-Boot and the Linux kernel were loaded, and the start address for Flash memory (found when probing for the "CRUS" token as described in the previous section).

Finally, it is worth mentioning that we modified the U-Boot configuration in Flash so as to load and boot automatically

[2]While taking into account the size restrictions imposed by the serial downloader feature of the Boot ROM code.

[3]Trivial File Transfer Protocol

[4]Latest available at the time the article was written is 3.8.8, original was 2.4

from the address where the Linux kernel was saved on Flash. The Linux kernel we built was prepared to load the root file system (the last remaining major piece of software needed for a functional system) from a usb storage device. As a result of this process, we modified the Encore ENTC-1000 firmware, thus offering at least two new interesting features:

- A modern and customizable bootloader, accesible through a serial line.
- An updated Linux kernel, prepared to boot different Linux operating systems from a USB port.

This functionalities lead us to a wide range of options to reuse the Encore ENTC-1000 device. From here we are ready to start our programming tests based on Linux embedded systems, without being required to go through the troublesome process described above. The new device is just boot-and-use.

V. Conclusions and Future Work

We used reverse engineering in order to understand the architecture, functioning and programming of a modern Linux embedded system.

Our knowledge has been increased through the experience and documentation process itself. The resulting system allows for academic experimentation based on embedded Linux, running on a commercial embedded hardware which can be repurposed for specific needs.

We chose the Encore ENTC-1000 embedded system for this experience as it met our expectations about device cost and capabilities. However, at first sight, our chances to be able to research and reuse the system were not clear.

As a result of our experience, we have been able to understand its internal functioning, and moreover, the device has been reused to provide the following functionalities in a production environment:

- Print server for USB printers without networking capability.
- DHCP server for small internal subnets at University.
- Brain for research robot, tied to a microcontroller and a set of sensors.
- Embedded Linux system for learning and experimentation.
- DNS experimental server.

As future work, we consider the fact that there may be still some time ahead until the acquisition of specialized hardware for embedded systems can be made affordable to public educational institutions. Until that moment can be reached, we aim to build standardized guidelines required for a methodology allowing the reuse of commercial Linux embedded systems. Even when such hardware is available for development, reverse engineering experience on comercial Linux embedded systems can be thought as a method to encourage learning about the architecture and functioning of embedded systems.

References

[1] S. Heath, *Embedded Systems Design*, Second Edition. Newnes, 2002. ISBN-13: 978-0750655460

[2] Free Software Foundation, *GNU General Public License*. http://www.gnu.org/licenses/gpl.html

[3] S.B. Furber, *ARM System-on-chip Architecture*, Second Edition. Addison-Wesley Professional, 2000. ISBN: 8131708403

[4] Denx, *U-Boot bootloader*. http://www.denx.de/wiki/U-Boot/

[5] Encore Electronics, *Encore Thin Client ENTC-1000*. http://www.encore-usa.com/ar/cat/Thin-Client/Encore-Thin-Client

[6] Cirrus Logic, *EP93xx User's Guide*. http://www.cirrus.com/en/pubs/manual/EP93xx_Users_Guide_UM1.pdf