Using GNU/Linux tools for creating ARM9-based embedded applications

Nikolay Rumenov Kakanakov and Mitko Petrov Shopov

Abstract – **The paper aims to provide a reference manual for using Linux/GNU tools in developing embedded Internet-ready applications for ARM9 based systems. It provides steps-by-step guide for configuring the bootloader, Linux kernel, toolchain and developing environment. As an example ARM920T based development board (EP9302) is used, running 2.6.32 kernel and embedded application that reads data from sensor on serial port and provides it as Web service.**

Keywords – **ARM9, Emdebian, toolchain, Eclipse**

I. Introduction

In modern digital and electronic society embedded systems are ubiquitous. New embedded world is built on new dynamic, distributed and collaborative architectures and environment. That leads to new requirements to the hardware and software of the embedded systems – low power consumption, short response time, high availability, seamless human-machine interfaces, etc. Some of the embedded application must cover this requirements at any price while other (e.g. home and customer electronics) at low price. Furthermore the design and development time should be small enough to cover the needs of the market. A way to cover that need of small time-to-market and low price is to use of-the-shelf software architectures, operating systems, development environments that are familiar to the engineers. The solution in that field, not new but gathering popularity with the last hardware achievements, is the application of embedded Linux boards and solutions [1] [2].

The reason for the popularity of Linux in the embedded applications is its layered structure, integrated security, native writing in C/C++, clearest TCP/IP stack, many protocols and drivers distributed with the OS. Following the open source ideology, embedded Linux engineers can share thoughts, ideas, technologies and even device drivers but keeping the company specific parts closed. Furthermore, all popular development tools that Linux programmers are familiar with can be easily configured as embedded IDEs and reduce the difference between the host and target systems [2] [3].

As an example application, a system for remote energy management is used. It is implemented on EP9302 embedded board from Olimex [7]. It reads the values from energy management sensor via serial port, stores them on USB flash drive and provides their graphical representation as a Web service [4].

The rest of the paper provides some materials on design and development of Linux embedded systems. It discusses the hardware-specific tools (e.g. bootloader); choosing, configuring and building an embedded Linux kernel; and creating user application for the embedded system.

N. Kakanakov is is with the Department of Computer Systems and Technologies, Faculty of Electronics and Automation, Technical University - Sofia, Plovdiv branch, 25, Tsanko Djustabanov str., 4000 Plovdiv, Bulgaria, e-mail: kakanak@tu-plovdiv.bg

M. Shopov is with the Department of Computer Systems and Technologies, Faculty of Electronics and Automation, Technical University - Sofia, Plovdiv branch, 25, Tsanko Djustabanov str., 4000 Plovdiv, Bulgaria, e-mail: mshopov@tu-plovdiv.bg

II. Choosing the Appropriate Architecture

When choosing the hardware platform and vendor for an embedded Linux application, the designer should have in mind not only the general parameters but is the hardware (architecture, CPU, peripheral modules) are supported by Linux kernel. In general terms Linux is the kernel and sometimes support libraries for creating applications (libc, binutils, gcc). Some hardware vendors provide specially built kernels for their platforms. In most cases it is not the best idea as the Linux packages and drivers are ever evolving together with the standard kernel. If a custom kernel is used, the ability to update the libraries and tools will be difficult. The better way is to choose a hardware architectures supported by the standard kernel. The most promising hardware architecture for Embedded Linux systems is the ARM core. It is used in many everyday tools like MP3 players, mobile phones, DVD/Satellite players, routers, access points.

A. ARM9 Family

Many hardware vendors develop products using the ARM9 core and especially ARM920T. It is ARM9 core with cache, protection unit, and Memory Management Unit. It is a Harvard cache architecture processor that is targeted at multiprogrammed applications where full memory management, high performance, and low power are all-important. The separate instruction and data caches in this design are 16KB each in size, with an 8-word line length. The ARM920T processor implements an enhanced ARMv4 architecture with MMU to provide translation and access permission checks for addresses [6].

As ARM9 is just a IP core and architecture, there are many vendors that produce processor chips and embedded boards based on this core. Some of most famous ones are: Freescale Semiconductor (i.MX and Kinetis series), Marvell (PXA family), ST Microelectronics (SPEAr eMPUs, STM32, STR7, and STR9), Texas Instruments (DaVinci, Sitara™, and Stellaris family), Xilinx (Zynq-7000 Extensible Processing Platform). Among this

mastodon vendors there are some popular chips with low price Cirrus Logic EP93xx series.

Several vendors provide embedded boards based on Cirrus Logic EP93xx ARM chips. One that is easy to find on the Bulgarian market is Olimex Ltd. Plovdiv. Their board is built with Ethernet controller, 2 USB and 2 serial, SD/MMC, IrDA and JTAG interfaces [7].

B. Bootloader

To enable an embedded hardware for an operating system like Linux, a bootloader is needed. This is a special part of the embedded code that prepares the hardware for initial boot and provides commands and environment to load the operating system kernel. To be flexible and fully functional, the bootloader must provide some user interface for configuration, initialization script for main peripheral devices (e.g. serial, USB, network, Flash memory) and different mechanisms to find/load the kernel. The most popular bootloaders for such hardware platforms are U-Boot and RedBoot. The former is preferred in applications that want to strictly apply GPL and has more limited resources and is almost a standard for ARM based devices. The latter is developed by RedHat and was created to boot-up e-Cos operating system. It gains its popularity due to its flexibility and many supported platforms and operating systems (e.g. e-Cos, QMX, Linux, Unix, BSD, Embedded Windows) [1].

In this paper the RedBoot will be used as an example because most of EP93xx boards are shipped with preloaded RedBoot binary.

RedBoot provides commands for configuring the initial environment, load and execute the operating system kernel (or eventually call other OS specific loader). In the case of Linux, the environment configuration depends on the chosen way of loading the kernel. The most popular ways to load the kernel is from chip's flash memory, from *tftp*, or from *http*. The first way is best suited for working boards, while the latter two are suitable for the development time. For using them some network configuration should be made for the RedBoot environment [12].

C. RedBoot binaries. Building Linux Kernel.

The RedBoot environment not only provide mechanisms for the developers to contact the board, load kernel images and start the OS. It also provides an environment to execute applications. This applications should be built as an e-Cos application, for freestanding environment (no standard libraries) and should be statically linked. This special feature given to the compiler and linker are needed because the RedBoot environment do not load any additional libraries. Binaries created this way must be started giving the entry point manually as long as there is no library for searching the main function.

Embedded systems usually come with strict requirements for memory and *cpu* usage. While one can use a general Linux kernel, it is more appropriate to build a custom kernel with respect to specific hardware and application requirements. To build a custom kernel one should first obtain a copy of the Linux kernel source tree [11]. The next step is to apply all the necessary kernel patches specific for the hardware platform you build the kernel for. All the necessary modules could be build-in with the kernel or could be build separately as loadable kernel modules. With the first approach all of the functionality will constantly reside in memory, even if not used. That will waste valuable memory resources. On the other side loadable modules require times to be loaded and are not applicable in some embedded applications. To start customizing your kernel you could use one of the available configuration tools, with the most used being *make menuconfig*. Here you should remember to specify the architecture you are building for. To build the kernel a cross-compiler for ARM should be used. The result is a custom and optimized Linux kernel. It could be in compressed and not compressed image. The compressed image could reduce memory footprint when image is saved on internal flash memory, but will require more time before loading after reset.

III. Choosing the Linux Distribution

The main view of the Linux OS is a kernel and a number of packages. As long as the kernel development is controlled by single public body [11], the packages are developed in many independent branches, called distributions. To build an embedded Linux application, one needs to obtain a kernel (and eventually patch it for specific hardware) and then choose the distribution that best suits the application to obtain the packages from. Many general purpose distributions now support ARM target but in some cases it is better to look for embedded distributions. There are many commercial and open source embedded Linux projects, but in current paper the Embedded Debian Project (Emdebian) is chosen. It is based on the Debian distribution but some of the packages are optimized in sizes and memory use and cross-toolchains are added. Official support is available for *i386*, *amd64*, *powerpc*, *armel*, *mips*, *mipsel* [8].

Using EP93xx board for embedded Linux allows several different paces to position the root file system. For very small size applications the best way is to build a archived ramdisk and store it in the internal flash. When booting up the board it extracts the ramdisk in the operating memory and uses it as root file system, providing very fast access to the storage during the work of the system, but lacking of non-volatile data storage. For the development time the best ways are to use Network File System or USB drive, as they can be edited easily from the host computer. For these features to work the kernel should be compiled with NSF and SATA support. The mounted USB drive is recognized as SATA drive from the kernel. For real applications the best way is to use the SD/MMC card interface. For this purpose a special patch [9] must be applied to the kernel to support MMC block devices and MMC drive as root file system. For some version of the hardware (e.g. EP9312, EP9315) there is and IDE interface and hard drive can be used for the root file system. It is best if the application

should store big amounts of data for a long time without restrictions to the consumed energy.

To use Emdebian on the embedded board means simply to build root file system for the device using Emdebian cross development tools and package repositories. The creation of the root file system is relatively easy process, unless using the cross development. Emdebian provides two ways for creating root file system – *debootstrap* and *multistrap*. *Multistrap* is a tool that starts off doing essentially same job as *debootstrap* but uses an entirely different method based on *apt* and extended to provide support for multiple repositories, using a configuration file to specify the relevant suites, architecture, extra packages and the mirror to use for each bootstrap. *Multistrap* works from pre-built binary packages and so creating a root file system requires that the full set of packages are available in a repository that *multistrap* can use. If different functional changes from those used in Emdebian are needed, the better tool is standard *debootstrap*. Using *debootstrap* has two main stages. The first stage is downloading the packages from the repository that contains for the target architectures and unpack them. For cross platform purposes the option '*foreign'* is used. It makes the initial unpack phase of bootstrapping only, and a copy of debootstrap sufficient for completing the bootstrap process will be installed in the target file system. Other important option is '*arch*', which specifies the target architecture (i386, arm, mips, etc.). After creating the initial phase, second stage of actual building the packages should be run on the target architecture or with cross-build and chroot environment.

IV. Creating Applications

For developing application software for the embedded Linux systems, one may use different integrated development environments. Most of the environments are specially built and/or configured for the targeted board from its vendor. As long as full featured Linux runs on the board, the application software can be built there. It works well for small projects but when some bigger project is developed the cross-platform compilers, linkers, builders and development environments are needed. For most ARM-based embedded platforms the most popular integrated development environment is probably Eclipse project. It is a Java based environment supporting multiple languages and application fields. It has a special sub-project CDT (C/C++ Development Tooling) [10]. In fact CDT provides an environment for writing source code with syntax highlighting, content advisor, and platform for integration of multiple plug-ins and external build systems. For the CDT project, the GNU Build Systems (so called *automake*) is used. It includes compilers, assemblers, linkers, loaders and auto-configuration tools for creating make files. The reason of the hegemony of Eclipse CDT in embedded applications development is that the build system is called externally using the *make* tool and can be substituted with other build systems. Thus, using cross-platform tools is relatively easy – installing the appropriate toolchain and configure the Eclipse environment to use it. Assuming that development of embedded software differs a little from other C/C++ projects, some plug-ins can be added to support and facilitate the embedded developers.

In software development the term toolchain is often used to represent a set of tools used in a chain to produce executables. Most of the toolchains are based on tools developed by the GNU open-source project. Gnu toolchains consist of cross-compiler, linker, C-libraries and cross-debugger. Several ARM cross-toolchains are available for use by embedded developers. Some of them are distributed as source code that could be compiled while other comes in a pre-build binaries. Some of the popular ARM cross-toolchains are GnuARM, Emdebian, Code Sourcery, Yagarto, and WinARM. To integrate a toolchain in the standard build process of Eclipse CDT an appropriate plug-in is needed. One can use an existing plug-in or alternatively can develop its own. Eclipse plug-ins that support the most popular ARM cross-toolchains are widely availability for free [5].

ARM cross-debugger could be used to remotely debug an application using serial or Ethernet connection with the remote target. Eclipse CDT provides a graphical interface to the cross-debugger, but is not capable to start the remote gdb server on the target hardware. For that purpose openOCD should be used. OpenOCD JTAG is a open-source on-chip debug solution for targets based on the ARM7, ARM9, Cortex-M3 and XSCALE families via JTAG port. In addition internal and external FLASH memory programming is supported.

A. Example application.

The example application is designed and developed as a part of project for creating Web-based system for measurement and control of electric power systems. It is supposed to be connected to energy sensors via special serial protocol and then store the collected data to provide it on demand as Web service or other network service. Creating this application was started from choosing hardware platform. The one that best suits the project as a price, availability and potential for fast development of Web applications is EP9302 from Olimex. It is based on ARM920T core that provide good working with Linux. After some tests with this board bootloader, Linux distribution and cross-development toolchain were chosen (RedBoot, kernel 2.6.24, Emdebian distribution with its ARM toolchain). This section presents steps for preparing the bootscript for RebBoot, buildnig and loading the kernel, creating root file system on USB drive, preparing the development environment, and creating the actual user-space application.

For the example application we have decided to build a custom Linux kernel. That way it was possible to exclude all unnecessary modules. The build process started with obtaining the source tree (linux-2.6.32). Next, all required patches have been applied. In the example application the MMC/SD interface could be used for data logging, so we have added MMC/SD support by obtaining the patch linux-2.6.24-rc8_ep93xx_mmc.patch.gz [9], slightly modified to match 2.6.32 kernel version.

$patch p1 < linux-2.6.32_mmc.patch

Then the kernel was configured with the help of make menuconfig and builded with the emdebian cross-compiler.

$ make ARCH=arm CROSS_COMPILE=arm-linux- zImage

In the example application the kernel is loaded from http server. It allows changing the configuration and integrated module drivers without reconfiguring the board. First, IP address (and eventually MAC address), subnet mask, default gateway and default server should be configured. Then the *load* command should be executed with options showing: (-b) the memory location to save the image on the board; (-m) transport protocol; (-h) server address; and the name of the kernel image.

#load -b 0x800000 -m http -h 192.168.2.14 zImage_2.6.32

After loading the image into memory it can be executed using *exec* command. This command (using the -c option) can forward some options to the executable. In the case of executing kernel, these options include: where is the root files system; to use serial console on specific serial device; IP configuration; serial port baudrate.

#exec -c "concole=/dev/ttyAM0 root=/dev/sda1 rootdelay=5"

For the example application the root file system is created on USB flash drive using *debootstrap*. The main packages are downloaded form the Emdebian *Grip* distribution . Grip is based on light version of Squeeze with as few functional changes as possible and highest level of binary and functional compatibility. It is intended to be a native build environment and cross runtime environment with the ability to mix and match Emdebian and Debian packages with minimal effort. The first stage of bootstrapping is made on the host platform, providing the appropriate architecture, *foregn* option, target directory to download the packages and repository URL.

$debootstrap --arch=armel --foreign squeeze grip/ http://www.emdebian.org/grip/

After this step, the file system is successfully downloaded, but needs to be configured. First, information about *proc* file system should be added to the *fstab*. Then in the dev/ folder at least two nodes should be created – for console and for the serial port (which will be used by console). And last (optional) add URL of the repository to the *apt* configuration.

$echo "proc /proc proc none 0 0" >>etc/fstab
$mknod dev/console c 5 1
$mknod dev/ttyAM0 c 204 64

After these steps, the file systems is downloaded and configured and is ready to be uploaded to the flash drive.

The second stage should be executed on the target device. Using the RedBoot commands, the kernel image should be loaded. After loading it must be executed, but with special option *init=* to tell the kernel what to start in init phase, because the root file system is still not ready.

'console=ttyAM0 root=/dev/sda1 init=/bin/sh rootdelay=10'

After this the system will boot-up staring shell. In this shell the path to binary executables should be exported and proc file system mounted. Then the second stage should be started.

sh-3.2# /debootstrap/debootstrap --second-stage

After waiting a relatively long time, the system will be installed and some adjustments to the configuration files can be made (e.g. networking, inittab, passwords and users). Then the system should be restarted omitting the *init=* option. The newly booted system can be upgraded to the up-to-date version using *apt.*

The actual user-space application starts two threads – using POSIX *pthread* library. The first thread queries the sensor on serial port using *termios* library, while the second implements standalone Web service using gSOAP Web-service toolkit. More information about this application is presented in [4]. The application is created using Eclipse CDT integrated development environment with GnuARM plug-in and is built using arm toolchain from Emdebian.

V. Conclusions

The presented paper can be used as a reference manual for creating embedded Linux applications from the very start. It covers the selection of hardware, configuring the bootloader and kernel/firmware, and creating the actual applications with commonly used toolchains and development environment. An example application is used to demonstrate the efficiency of the presented concept.

Using Linux as embedded operating system provides the developers with many ready of-the-shelf applications, drivers, tools. It also takes the embedded applications one step closer to the Web of Things concept, as the Linux environment is best suited for networking and Web services. Embedded Linux further enables multiple flexible home and mobile applications that will be the essential part of the future Internet society.

VI. Acknowledgments

The presented work is supported by Technical University of Sofia, project "102ни200-3/2010", entitled "Investigation of technologies for development of Web-based systems for measurement and control of electric power systems" and partly supported by project "BG 051PO001-3.3.04/13" of European social fund 2007-2013.

References

[1] K. Yaghmour, J. Masters, G. Ben-Yossef, P. Gerum, "Building Embedded Linux Systems", O'Reilly Media 2008.
[2] Hallinan Christopher, "Embedded Linux primer : a practical real-world approach", Pearson Education 2011, ISBN-13: 978-0-137-01783-6.
[3] "Embedded Linux Best Practices," white paper, Katalix Systems, Dec. 2006.
[4] N. Kakanakov, G. Spasov, "Web enabled system for remote energy management," in Journal of Electronics, vol. 4, no.2, pp. 95- 98, 2010, ISSN 1313-1842, 2010.
[5] R. Stigge, "Embedded Linux Development with Debian for ARM", Philosys GmbH, Oct. 2007, http://www.philosys.de/
[6] ARM homepage: http://www.arm.com
[7] Olimex Ltd homepage: http://www.olimex.com/dev/
[8] Emdebian homepage: http://www.emdebian.org/
[9] Peter Ivanov's site: http://dev.ivanov.eu/
[10] Eclipse project: http://www.eclipse.org/cdt/
[11] Linux kernel homepage: http://www.kernel.org
[12] RedBoot User's Guide: http://ecos.sourceware.org/docs-latest/redboot/redboot-guide.html
[13] DSNet V-Lab forum for EP9302: http://dsnet.tu-plovdiv.bg/forums/viewforum.php?f=23