Integrating Open Source Tools for Developing Embedded Linux Applications

Raul Fernandes Herbster[1], **Hyggo Almeida**[1], **Angelo Perkusich**[1], **Dalton Guerrero**[1] [*]

[1]Embedded Systems and Pervasive Computing Laboratory
Electrical Engineering and Informatics Center
Federal University of Campina Grande

{raul,dalton}@dsc.ufcg.edu.br,{hyggo,perkusic}@dee.ufcg.edu.br

Abstract. *The development of embedded Linux applications have been supported by several open source tools. In order to make the programming activity less complex and more productive, such tools should be easy to install, configure and use. However, using the available open source tools, developers still have to understand and use different kinds of user interfaces and environments. Time and effort are spent on configuration issues rather than on programming. In this paper we present how to integrate open source development tools for programming embedded Linux applications. We introduce an Eclipse plug-in to support the development of embedded Linux applications using a single and integrated environment.*

1. Introduction

In the last years the Linux operating system has been successfully applied to develop many different embedded devices [Geer 2004]. In fact as reported in [Lafer 2006], Linux was the most cited operating system for current embedded systems development. The widespread interest and special attention generated by Linux success on several embedded applications has encouraged researches and development projects focused on *embedded Linux*. More and more solutions based on embedded Linux are being required, specially open source solutions [Wong 2006], and hence there is an increasing need for programmers for the embedded Linux.

Besides programmers, good tools for supporting the programming activity are also essential. Diversity and quality of tools are important factors that make the technology largely adopted by the community of programmers. There is a huge variety of techniques and tools for developing embedded Linux applications, supporting different phases of the development process. Each one has its own user interface and needs different kinds of input and produces outputs in special ways. The correct management of such tools makes the programming activity harder and more complex.

To solve this problem, the integration of programming tools is critical. Integrating environments which were not designed to communicate with other external tools demands extra work, because no interface is defined for such an integration. Another issue is the lack of documentation – several open source tools do not have good documentation. This makes necessary to understand their source code to discover how to integrate them.

In this paper we present how to integrate two open source tools - Eclipse CDT [Eclipse.org 2006a] and Scratchbox [Scratchbox 2006] - for programming embedded Linux applications. We introduce an Eclipse plug-in to support the development of

[*]The authors would like to thank Nokia do Brasil and Instituto Nokia de Tecnologia for the support to develop this work.

such applications using an integrated environment, reducing time and increasing productivity in embedded Linux applications development.

The remainder of this paper is organized as follows: Section 2. describes some features of Embedded Linux; Section 3. presents the most used development tools for embedded Linux applications; our approach for integrating such tools are described in Section 4.; and conclusions are presented in Section 5.

2. Embedded Linux

An *embedded Linux system* is an embedded system based on the Linux kernel, that uses or not any specific library [Yahgmour 2003]. The recent success of Linux based applications and solutions have been attracted users, media and business interest. Linux is becoming the preferred operating systems for embedded applications, placed everywhere in our lives, from mobile phones to medical equipment [Yahgmour 2003, Lafer 2006].

In the year 2000, the *Embedded Systems Programming* magazine conducted a survey including 547 subscribers. The result was that 38% of readers were considering Linux as the operating system for their design [Yahgmour 2003]. According to Venture Development Corporation, the embedded Linux market will top $100 million this year [DeviceLinux.com 2006].

There are several reasons for using Linux over other embedded operating systems [Yahgmour 2003]:

- Quality and reliability of code - Many programmers agree that Linux kernel and most projects used in a Linux system have quality and reliability of code. From these aspects, one can expect some characteristics that make the code easier to fix, extend and maintain.
- Availability of code - Linux source code and build tools are available without access restrictions. The advantages are the possibility of fixing the code without exterior help and the capability to understand its operation by observing its execution.
- Hardware support - Linux supports different kinds of hardware platforms and devices. Many drivers are created and maintained by the Linux community itself.
- Cost - The open source license of Linux reduces its cost.
- Available tools - There is a huge variety of tools for Linux. One can find easily a free application that is looking for.
- Community support - This is perhaps the strongest feature of Linux. Forums and mail lists are the best place to find this community support, and the level of expertise found there surpasses the expensive support offered by proprietary operating systems vendors.
- Communication protocol and software standards - Linux provides broad communication protocols and software support. Integrating Linux with others systems over a network is not a difficult task.

3. Development Tools

Embedded systems have some constraints that must be carefully analyzed while designing applications for them [Yahgmour 2003]: memory usage and disposal, power consumption, cost, restricted user interfaces and mobility issues. These aspects point out that developing applications for embedded systems require more attention than designing applications for desktop.

The development processes for embedded applications must take these aspects into account. An extra time is spent for designing these applications, because the inception phase must be carefully analyzed and described [Wolf 2001]. Besides, as programmers of other platforms, embedded Linux developers need development tools, like compilers, text editors, linkers, debuggers, and integrated development environments to improve productivity [Yahgmour 2003]. The major difference between conventional Linux development is that embedded Linux is based on *cross-platform development*. That is, embedded systems tools run on a host development platform and the developed applications run on another one, namely the target platform.

3.1. Cross-Platform Development Toolchain

Compiling the code on a host machine with larger computational resources is faster than in the target embedded system platform. Usually, the target platform has more restricted computational resources, and, generally, less computing power than a desktop or a workstation machine. The objective of a building process is to generate an executable that runs on the target platform. The standard approach is to develop the software on an environment specially configured for building such software for a specific target device. Everything programmers need, like compilers, libraries and binary utilities are specific for the target they work with.

A cross-platform development toolchain is a collection of essential tools and libraries for building specific-platform applications. The host machine, generally a desktop, builds the application to a target device - an ARM [ARM 2005] based platform, for example.

3.1.1. Standalone Toolchain Build

Building a cross-platform development toolchain from scratch is, in most cases, a painful work. It takes long hours for compiling and performing all the process. In what follows we describe concisely describe the overall build process. Details related to the cross-platform development toolchain is can be found in [Yahgmour 2003] and [Hollabaugh 2003]. To build a basic toolchain, the following GNU [GNU 2006] tools are used:

binutils Binaries utilities [Binutils 2006], which is a collection of binary tools, including linkers, assemblers and much more.

GCC GNU Compiler Collection [GCC 2005], that includes front-ends for C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages.

glic The GNU C Library [glibc 2006]. It is used as the C library in the GNU system and most systems with the Linux kernel. C library is important because it defines the "system calls" and other basic facilities such `malloc`, `scanf`, `exit` and others. Other C library variants can be also used [uClibc 2006].

GDB The GNU debugger [GDB 2005] is used to debug executables. It is the most used GDB for C/C++ programmers and it has useful features, like memory and registers display, trace, and others.

There are also patches for specific platforms, such as ARM based ones, that must be also installed. To start with, one has to select the component versions that will be used. To work properly, it is important to choose the correct version of each component, that is, select binutils version, a gcc version and a glibc version. It is recommended to use old and stable packages, because some new packages require other packages to provide certain capabilities. In [Kegel 2006] and [Yahgmour 2003], the programmer can find information on stable functional packages version combination.

With the appropriate tools already in place, the programmer must build the toolchain. This process takes hours because all the source of components must be compiled for the target platform chosen. Buildroot [Buildroot 2006] and Crosstool [Kegel 2006] are tools for helping to build toolchains by downloading sources and applying patches.

After building all the components, the programmer has a fully functional cross-development toolchain, which can be used as a native GNU toolchain, with a small difference: the target name is added to every command that is used. For example, instead of using `gcc` for the target, the programmer will invoke `arm-linux-gcc` for ARM target.

3.1.2. Using Ready-To-Use Cross-Platform Development Toolchain

In fact, creating a customized cross-platform development toolchain to build applications for specific targets is a complex task, which requires time and patience to configure all components. It is important to understand the dependencies between the different packages and a lot of time is spent for configuring the environment that will be used.

Unfortunately, in order to change to another target, the programmer has to perform all tasks again. For example, if the application must run on an ARM platform, the cross-platform development toolchain must be built for ARM as target. Then, if the target platform is changed to Sparc [ULTRASparc 2005], the cross-platform has to be built again.

There are ready-to-use cross-platform development toolchains that are already built for specific platforms, saving precious time and effort of the programmer. Some of this pre-compiled toolchains can be found at [CodeSourcery 2006].

3.2. Scratchbox Toolchain

Scratchbox [Scratchbox 2006] is a compilation and configuration environment for building Linux software and entire Linux distributions [Mankinen and Rahkonen 2004]. Created by Movial [Movial 2006] as an open source project, Scratchbox offers to developers an environment that works and looks like the target environment even before the target environment becomes available [Mankinen and Rahkonen 2004].

Scratchbox project has a lot of tools for cross-compiling software for a large number of architectures, like ARM and Sparc. For using Scratchbox, the programmer must define the TARGET with the aid of a wizard, without worrying about the correct versions of binaries, C libraries, compiler and other elements to be installed. Thus, the code is compiled using GCC commands as whether the code were being compiled to a Linux desktop machine.

Since design, coding, testing, integrating and documentation can be performed on the Scratchbox, the developers can start working even if the platform is not finished yet. They must have only the specific target configured properly. The lack of a physical platform does not affect the development task.

3.3. Eclipse C/C++ Development Tools

Eclipse [Eclipse.org 2006b] is an extensible platform for tool development and integration used by millions of developers around the world [Clayberg and Rubel 2004]. Through the Eclipse platform, the integration of tools from several different vendors is possible on Linux, Windows and other operating systems.

Eclipse Platform has mechanisms that allows developers to extend its functionalities, adding new tools to the platform. These mechanisms, called plug-ins, allow developers programming in different languages like Java, C/C++ and Phyton. The C/C++

Development Tools (CDT) [Eclipse.org 2006a] provide support for Linux platform and for GNU tools (GCC, GDB, Make), but it is also possible to use versions of GCC for other platforms, like MingW for Windows [MingW 2006].

For programming, CDT offers useful features, like syntax highlighting, search engine, error parsing, wizards for creating projects, source and head files and much more. The debugger environment, which runs over GDB, displays variables, memory map, and registers. It is also possible to debug multi-threaded programs. Eclipse Platform, and hence CDT, offers several extension points so that developers can develop plug-ins to add new functionalities to the CDT environment.

4. Integrating Scratchbox and CDT

Using both Scratchbox and CDT can reduce configuration, programming and compilation time. However, productivity problems remain critical. It occurs because, in the case of embedded systems, CDT supports only the codification activity. Compilation, debugging, and execution activity must be performed using Scratchbox. The developer spends a large amount of time switching environments, using the output from CDT as an input for the Scratchbox. Besides, programmers have also to correctly manage and configure each tool.

In order to provide an integrated environment for developing embedded Linux applications we propose the ESBox Plug-in for the Eclipse platform. ESBox is an extension of the CDT plug-in, strongly integrated with Scratchbox, reducing time and effort for configuration and increasing the productivity on Linux-based embedded systems development.

4.1. ESBox Plug-in Architecture

The integration of Scratchbox and CDT occurs by means of service calls. In a nutshell, Scratchbox is a command line environment and the service calls are also made through command line. To perform the invocation of Scratchbox services from CDT, Scratchbox commands are wrapped into processes instances and all communication with Scratchbox are made through these processes. Input, output and error streams of such processes are redirected to GUI elements, such as lists and errors messages. CDT features, like wizards and resources property pages, are implemented via extension points mechanisms.

ESbox architecture is detailed in Figure 1. Both Eclipse and Scratchbox run over Linux. ESbox has two modules: the framework, which calls Scratchbox services and manipulate the streams of the processes properly; the plug-in, which changes data with framework, requests Scratchbox services, displays information from Scratchbox to the programmer and implements features for helping editing, launching and debugging. Tools which use Scratchbox, like Maemo [Maemo.org 2006], can also be plugged into ESbox.

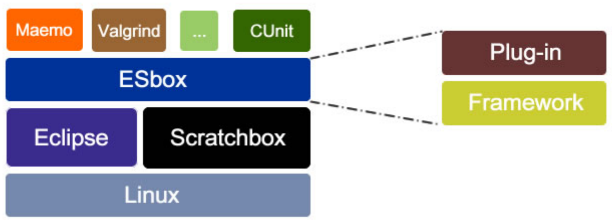


Figure 1. ESbox Plug-in Architecture

Large experience in Scratchbox is not essential. The programmer has to know only some basic concepts, because ESbox offers a middle layer between Scratchbox and the programmer. ESbox supplies the need of use command line and archaic text editors.

The version 1.0 of ESbox was developed for Eclipse 3.1, CDT 3.0 and Linux (Ubuntu [Ubuntu 2006] is the used distribution) and supports Scratchbox versions 0.9.8 or 1.x.x. ESbox is licensed under GNU General Public License (GPL). It includes a C/C++ code editor that provides features like code completion, search engine, error parsing and syntax highlighting; a front-end GDB debugger; remote debug on the target; a launcher; a Scratchbox target manager; and wizards to create managed or standard C/C++ projects inside Scratchbox.

4.2. ESBox Functional Scenario

First of all, the developer has to define the Scratchbox target to be used. The management of targets are performed at ESbox environment, using GUI wizards to create and delete targets, if it is necessary. ESbox offers to programmers different kinds of project: standard or managed. Standards projects are built using makefiles written by the programmer, whereas automatic generated makefiles are used to compile the managed projects.

The developer uses a wizard to create and configure projects, defining error parsers and default make commands. After creating the project, the programmer can create resources, like source files and paths, header files and makefiles. Useful features for coding and a rich GUI to manage the project are available for the programmers.

After editing the source code, the project can be built automatically or by using the makefile previously written. Errors or warnings generated during compiling process are marked at source code, so the programmer can identify where the problems are. The binary output is displayed at the *project structure* view and it can be launched using some clicks of the mouse. Finally, after launching the executable, the console displays the application input and output. All this process occurs under a GUI environment and the programmers does not have to interact directly with Scratchbox.

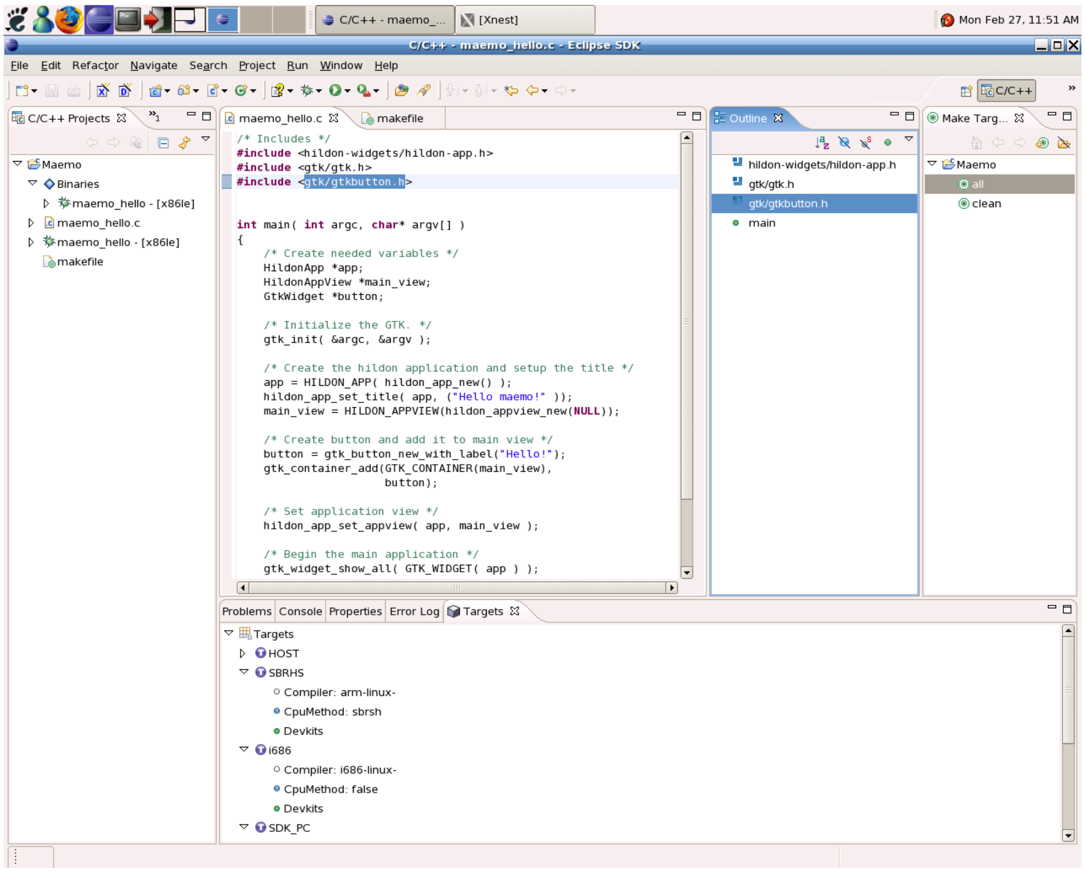
If the programmer wants to debug the application, instead of launching the binary, the programmer just choices “debug application” and debug perspective, with a set of views that displays registers, memory map, state of variables and stack and more are available to the programmer for debugging step-by-step. In Figure 2 a screen shot of the tool running is shown. A more detailed description of the ESBox project can be found at http://tesla.dee.ufcg.edu.br/~omapsdk/.

5. Conclusion

In most cases programming Embedded Linux applications using open source tools can be a painful work. This is mainly because such tools have complex installation and configuration processes. Nowadays, tools such as Scratchbox can simplify the programming activity by providing mechanisms for easily configuring and emulating the target platform. On the other hand, Eclipse based programming tools, such as CDT plug-in, have been largely used.

However, there is no automatic integration between such tools. And, therefore, the developer must spend a large amount of time switching environments, using the output from CDT as an input for the Scratchbox. Moreover, configuration and management of such tools require a lot of time and effort.

In this paper we presented the integration of two open source tools, Eclipse CDT and Scratchbox, for programming embedded Linux applications. We introduced the im-



```
/* Includes */
#include <hildon-widgets/hildon-app.h>
#include <gtk/gtk.h>
#include <gtk/gtkbutton.h>


int main( int argc, char* argv[] )
{
    /* Create needed variables */
    HildonApp *app;
    HildonAppView *main_view;
    GtkWidget *button;

    /* Initialize the GTK. */
    gtk_init( &argc, &argv );

    /* Create the hildon application and setup the title */
    app = HILDON_APP( hildon_app_new() );
    hildon_app_set_title( app, ("Hello maemo!" ));
    main_view = HILDON_APPVIEW(hildon_appview_new(NULL));

    /* Create button and add it to main view */
    button = gtk_button_new_with_label("Hello!");
    gtk_container_add(GTK_CONTAINER(main_view),
                      button);

    /* Set application view */
    hildon_app_set_appview( app, main_view );

    /* Begin the main application */
    gtk_widget_show_all( GTK_WIDGET( app ) );
```

```
Targets
  HOST
  SBRHS
    Compiler: arm-linux-
    CpuMethod: sbrsh
    Devkits
  i686
    Compiler: i686-linux-
    CpuMethod: false
    Devkits
  SDK_PC
```

Figure 2. Screen Shot of the Tool Running

plementation of a plug-in called ESBox, which extends the CDT functionality to integrate it with Scratchbox, reducing configuration time and thus increasing productivity.

Major problems faced during the development were related to the lack of documentation for extending the Eclipse CDT plug-in. There are no helps, tutorials, articles or books explaining how to extend CDT functionalities. The ESBox development team had to understand the source code to implement the CDT extensions.

Today, some students of Embedded Systems and Pervasive Computing Laboratory are using ESbox for developing an API over Flute [Flute 2006] and all reports generated by this team are analyzed to verify which features need to be implemented at next releases of ESbox. As future works we plan to implement productivity and quality plug-ins, like a design checker and a C/C++ unit test tool. A plug-in for Maemo platform [Maemo.org 2006] is also a future feature of ESbox.

References

ARM (2005). *ARM Processors*. http://www.arm.com. Last access on 12/28/2005.

Binutils (2006). The gnu binutils. http://www.gnu.org/software/binutils/. Last access on 02/17/2006.

Buildroot (2006). Buildroot. http://buildroot.uclibc.org/. Last access on 02/14/2006.

Clayberg, E. and Rubel, D. (2004). *Eclipse: Building Commercial-Quality Plug-ins*. Addison Wesley, Boston, USA.

CodeSourcery (2006). *GNU Toolchain for ARM Processors.* http://www.codesourcery.com/gnu_toolchains/arm/. Last access on 02/16/2006.

DeviceLinux.com (2006). *Embedded Linux market growing significantly, research firm says.* http://www.linux.org. Last access on 01/30/2006.

Eclipse.org (2006a). *Eclipse CDT.* http://www.eclipse.org/cdt. Last access on 02/14/2006.

Eclipse.org (2006b). *Eclipse Platform.* http://www.eclipse.org. Last access on 02/14/2006.

Flute (2006). *FLUTE - File Delivery over Unidirectional Transport.* http://rfc3926.x42.com/. Last access on 03/17/2006.

GCC (2005). *GCC - GNU Compiler Collection.* http://gcc.gnu.org. Last access on 12/22/2005.

GDB (2005). *The GNU Project Debugger.* http://www.gnu.org/software/gdb. Last access on 12/22/2005.

Geer, D. (2004). Survey: Embedded linux ahead of the pack. *Distributed Systems Online*, 5(10).

glibc (2006). Gnu c library. http://www.gnu.org/software/libc/. Last access on 02/17/2006.

GNU (2006). *The GNU Operating System.* http://www.gnu.org/. Last access on 02/16/2006.

Hollabaugh, C. (2003). *Embedded Linux: Hardware, Software, and Interfacing.* Addison Wesley.

Kegel (2006). Crosstool. http://www.kegel.com/crosstool/. Last access on 02/14/2006.

Lafer, C. (2006). Open source in the embedded market: Linux and much more. *Electronic Design, http://www.elecdesign.com/Articles/Index.cfm?AD=1&ArticleID=11733, Last access 02/27/2006*, (ED Online ID #11733).

Maemo.org (2006). *Maemo Platform.* http://www.maemo.org. Last access on 02/14/2006.

Mankinen, V. and Rahkonen, V. (2004). *Cross-Compiling Tutorial with Scratchbox.* http://www.scratchbox.org/documentation/docbook/tutorial.html.

MingW (2006). *MingW - Minimalist GNU for Windows .* http://www.mingw.org. Last access on 02/16/2006.

Movial (2006). Movial. http://www.movial.fi. Last access on 02/18/2006.

Scratchbox (2006). *Scratchbox Toolchain.* http://www.scratchbox.org. Last access on 02/14/2006.

Ubuntu (2006). *Ubuntu Linux Distribution.* http://www.ubuntu.com/. Last access on 01/03/2006.

uClibc (2006). *uClibc.* http://www.uclibc.org/. Last access on 02/18/2006.

ULTRASparc (2005). *ULTRASparc Processors.* http://www.sun.com/processors. Last access on 12/28/2005.

Wolf, W. (2001). *Computer as Components: principles of embedded computing system design.* Morgan Kaufmann, San Francisco, California, USA.

Wong, W. (2006). Embedded software: An open-source territory. *Electronic Design, http://www.elecdesign.com/Articles/Index.cfm?AD=1&ArticleID=11733, Last access 02/27/2006*, (ED Online ID #11735).

Yahgmour, K. (2003). *Building Embedded Linux Systems.* O'Reilly, California, USA.