ABSTRACT

COLON, MICAH. A New Operating System and Application Programming Interface for the EvBot Robot Platform. (Under the direction of Dr. Edward Grant.)

The research presented in this thesis describes the development of the Linux distribution and a new control architecture for robots. The reasons Linux was chosen are enumerated and a description of the build system and setup used to generate the distribution, with support for multiple platforms, is discussed. The Evbot Abstraction Layer (EAL), a new robot control architecture and framework is described, and the simple API is detailed.

A New Operating System and Application Programming Interface for the EvBot Robot Platform

by
Micah Colon

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2010

APPROVED BY:

Dr. Alexander Dean	Dr. Troy Nagle

Dr. Edward Grant
Chair of Advisory Committee

BIOGRAPHY

Micah Colon graduated with Bachelor of Science degrees in Computer Science and Computer Engineering from North Carolina State University in December 2000. He remained as an employee with the university and eventually pursued a Masters in Computer Engineering part time while working full time. His research interests include robotics, embedded systems programming, and artificial intelligence.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALFS = Automated Linux From Scratch

API = Application Programming Interface

CBSE = Component-Based Software Engineering

CORBA = Common Object Request Broker Architecture

CPU = Central Processing Unit

CRIM = Center for Robotics and Intelligent Machines

DHCP = Dynamic Host Configuration Protocol

DoC = Disk on Chip

DSP = Digital Signal Processor

EAL = EvBot Abstraction Layer

GNU = GNU's Not Unix

GRUB = GRand Unified Bootloader

IDE = Integrated Drive Electronics

LFS = Linux From Scratch

LILO = LInux LOader

MIRO = MIddleware for RObots

OS = Operating System

PCMCIA = Personal Computer Memory Card International Association

RAM = Random Access Memor

SCSI = Small Computer System Interface

SDHC = Secure Digital High Capacity

SDK = Software Development Kit

SSH = Secure SHell

TCP/IP = Transport Control Protocol/Internet Protocol

URI = Uniform Resource Identifier

USB = Universal Serial Bus

Chapter 1

Introduction

1.1 Motivation

The autonomous mobile evolutionary robot platforms at NC State University, originally developed by John Galeotti and further refined by Leonardo Mattos are still used for experiments and research in robotics in the Center for Robotics and Intelligent Machines (CRIM). Continued support for these platforms is becoming more difficult as the operating system and control methods have become increasingly outdated and incompatible with current needs. Updated support for newer sensor and acutator hardware is required, as well as a new method of accessing and using these platforms.

1.2 Thesis Outline

The research presented in this thesis addresses the needs of a current operating system and new control methods. Chapter 2 presents an overview of the literature for the different control methods currently employed as well as the operating systems in use

and the motivation for their adoption. In Chapter 3 the operating system needs are evaluated for support of existing hardware and future updates, as well as the longevity desired for support and applicability. The considerations and desired feature set are presented and the justifications for the continued use of Linux on the EvBots. Several methods are explored to find a repeatable system of creating the needed operating system and associated development tools and software, customized specifically for the needs of the lab. Chapter 4 presents the details of the Evbot Abstraction Layer (EAL), the control system developed. The EAL framework separates the need for the user to be aware of the details of the hardware on the robot, yet still able to access the full array of sensors and actuators through a simple interface. The EAL (EvBot Abstraction Layer) framework has the added design benefit of allowing the same control method to be used with different hardware without additional modification. This allows research to be conducted with multiple platforms or sensors and actuators to be upgraded with little to no impact to control algorithms. With only 17 total system calls, the EAL framework is also very easy to learn and use.

Experiments conducted to test the flexibility of the OS build system and the use of the new control system are described in chapters 5 and 6. Lastly, in chapter 7, the benefits of the operating system, the associated build system, and the new control system are discussed. Additionaly, some of the decisions made in the current implemention of the control system have were discovered to have ramifications for future use and some possible choices for future work are presented.

1.3 Thesis Goals

The objectives of this thesis are to describe:

- Design decision for long term support of an operating system for the EvBots
- Design and implementation of a new control method, EAL.
- Demonstration of the flexibility of the OS and the associated build system.
- Demonstration of the simple yet effective control of robots using EAL.

Chapter 2

Literature Review

The design and building of robots has created numerous hardware and software platforms over the years, from the very simple to the extremely complex [16] [18] [28]. Much work has gone into the control systems and architectures of these robots. Increasingly, the focus has centered on making these systems re-usable. [37] [14]

There are several characteristics that have been noted as highly desirable in the design of robot architectures. Programmability and modularity are useful to ensure that a robot is applicable to more than one task [33]. A level of hardware abstraction provides portability, as does extendibility of the architecture, allowing new components to be added over time. Additionally, the complexity of the architecture should be reduced to enable simple itegration of new components and ease debugging. [22].

A number of robot architectures have implemented *middleware* to meet the characteristics noted above. Player is a highly portable and reusable robot code base but does not fully define network capabilities [37]. Orca is another *middleware* designed with a Component-Based Software Engineering (CBSE) approach that emphasizes modularity

and extendibility [15], though some have found that the hardware requirements are large enough to limit it to off-board processing [27]. MIRO (MIddlware for RObots) is another framework for mobile robot control which uses CORBA (Common Object Request Broker Architecture) to facilitate distributed communication in a network environment [36]. The use of CORBA has been seen in other *middleware* frameworks as well [13] [29] [39].

CORBA is a collection of specifications created by Object Management Group (OMG), defining a communication standard for programs. It provides portable distributed applications across heterogeneous systems in a network environment [6]. Gowdy [24] found, in his review of networking layers, that CORBA could solve almost any issue, current or future, that a developer might encounter, but is so comprehensive that it is very difficult to master. A similar review of several networking layers for robotics use found that CORBA was the "largest consumer of computing resources" but also the "most flexible, modular, portable, and extensible" [17]. It was also noted that it took nearly a year for developers to become familiar with most aspects of the COBRA/MIRO system they chose.

Researchers have developed a variety of full robotic software architectures, including development systems and simulation environments. Some of these have built on the *middleware* provided by MIRO [34] while others have used some of the advanced functionality of CORBA to allow dynamic addition of components to the system [20]. Other researchers designed a robotic platform using the QNX operating system, but it was limited to a single platform [30]. One system architecture managed approximately 70% code reuse when moving to a different hardware platform by applying their hardware abstraction layer, but focused primarily on very tiny embedded systems [27]. One unifying theme for all of these systems is that they are all complex.

A layered approach, allowing for complex access for expert users while still providing simplicity for new users is explored in robots designed to be used for both teaching and research [26]. A related layered approach in a robotic framework are used in a Java-based system [32], though it and the strongly typed programming framework used in ROCI [19] are intended for cooperative multi-robot systems. Despite attempts to remain simple, these architectures still remain fairly complicated.

At least one system has been seen that uses QNX as the operating system and there are others that use Windows XP [12]. A larger majority of the robotic platforms looked at, however, chose to use Linux as their primary operating system [21] [23] [31] [26]. The OpenHRP [28] architecture uses Linux with real-time extensions. Winfield [38] specifically describes his reasons for using the Linux operating system, including the modest hardware requirements.

The literature has shown that a modular robotic architecture which provides hardware abstraction and extendibility is both a good goal and an achievable one. However, making such a robotic architecture that is highly programmable, portable, and yet remains simple to use does not appear to have been achieved.

Chapter 3

EvBot Operating System

The autonomous mobile Evolutionary Robot platform at NC State University has proven to be a useful hardware and software platform, that has enabled researchers to investigate a variety of mobile robot areas [23] [31]. However, as more use was made of these platforms, some of the initial systems and design decisions have become out-dated, or found limiting to certain degrees. One of the major limitations has been the hardware of the EvBots, which has been rapidly rendered obsolete. Another important part of the EvBots that needs revision periodically is the operating system. This chapter researches requirements of, and creation of, a new operating system for the EvBots.

3.1 History, EvBots, and Operating Systems

The EvBot I operating system was based primarily on the size constraints of the flash PCMCIA cards used as primary storage. The largest use of space was Matlab, which was used extensively for user programs. This left only a fraction of the total capacity of the

memory for the operating system and user data. Based on evaluations at the time, Linux was chosen, and a modified distribution with further customizations was used. [23]

The EvBot II revision updated the hardware, focused primarily on extending the sensor capabilities, but did not change the operating system. The most significant change was an increase in the size of the flash PCMCIA cards. [31]

Currently work is underway on producing an entirely new hardware base for the EvBots. This new mobile robot platform, the EvBot III, requires a new operating system. The operating system described in this chapter will be used with the EvBot III when the hardware is ready, but will also be compatible with the older hardware platforms.

3.2 Goals and Considerations

There are many things to consider when designing an operating system, especially when dealing with limitations associated with hardware and resources. Some things, such as lack of hardware support, immediately rule out certain possibilities such as DOS, or older, smaller operating systems. Other considerations, however, can be met, in varying degrees, by any number of operating system choices including various flavors of Microsoft Windows, multiple distributions of Linux, and others [25]. A proper set of requirements and preferences is instrumental in making a final choice. Guidelines are given below.

3.2.1 General Guidelines

After looking at previous iterations of the EvBots and discussing the needs and preferences, the following set of four general guidelines evolved. There is a need for:

- A current, working, modifiable operating system
- Consistency across hardware platforms
- Stability
- Free or Open Source

An operating system that does not work is obviously useless. However, a competely out-dated operating system that no longer supports user needs is similarly useless. Being able to modify the operating system does present some potential security issues, but this is acceptable given the intended limited scope. Having the ability to keep things current is of much greater value for the mobile robot platforms. The new EvBot III is a completely new mechatronic system hardware design. As such, changes and modifications will be required in the short-term, and long-term, as needed.

Consistency across platforms is also important. Within current hardware platforms there are differences and there will be further changes in the future. While there may need to be some changes in the operating system to account for these mechatronic system changes, the user should not generally be aware of any changes, except where absolutely necessary. This should hold true for all general mechatronic systems upgrades, though not necessarily for backwards compatibility.

For a research platform, stability is definitely a desireable feature. There are some operating systems, historic and current, which have been plagued by well documented instability [35]. There are also those which fall into the other category of extremely stable, and of course those in between. Some stability may be sacrificed for other aspects such as specific hardware support, but in general the system should be very stable and reliable.

Finally, the operating system should be completely free and/or open source. It is a common misconception that open source means free software. This is not always the case. Open source specifies only that all of the source code for the system is, depending on the specific licensing, modifiable and distributable. Here, the option of sharing the resulting operating system with other researchers should be achieved without worry or cost.

3.2.2 Considerations

There are a few things that must be taken into consideration when choosing and customizing the operating system for mobile robots. For our purposes, the main considerations are the computer system hardware limitations of the EvBots themselves and also the ability to perform software development.

The EvBots are relatively computer systems hardware limited, especially when compared to the computing power of current desktop systems. Hardware limitations include financial constraints during their development, and the design constraints of size and power consumption. Of particular importance, with the EvBot I & II, was the startup from the DiskOnChip module, coupled with the main drive being a flash PCMCIA card. This specific hardware configuration presents booting challenges that must be addressed here. In addition, the slower CPU (in comparison to desktop systems) encourages the design to place as few demands on the processor as possible, leaving it to be utilized fully by user programs.

The limited hardware of the EvBots, in addition to their intended mobile nature, makes software development something of a challenge. There are two main lines of development currently targeted to address code development limitations: compiled code

and Matlab/Octave scripts. For speed, compiled code is often preferable, although it can be more difficult to write and debug. As a counterpoint, Matlab or Octave (an open source alternative to Matlab) provides some flexibility and possibly faster algorithm development. The new operating system should be able to support both these modes of software development.

3.2.3 Desired Feature Set

Reflecting on the general guidelines and the considerations discussed, a specific set of capabilities for the operating system was determined. These capabilities are expected to be heavily utilized during both the development of software, and possibly hardware and also during final testing and use of the EvBots.

The following features are desired:

- Minimal hard disk use

  Compared to memory, hard drive access is slow. In addition, as the primary drive capacity is flash memory, the lifetime of the primary drive can be extended if excessive usage can be reduced.

- USB auto-mounting and auto-configuration

  There should be no user involvement in recognizing, mounting, and making available to the user known USB devices. The ability to update the OS as noted earlier would allow for updates to the database (and drivers) of 'known' USB devices.

- Networking and dynamic configuration

  As a well known standard, general TCP/IP networking should be available, providing the hardware capability exists. This should be dynamically configured, most likely with DHCP, to allow for use and testing outside of a known enviroment (such as the labratory).

- Remote access capabilities

  In conjunction with the networking capabilities, the EvBots should be accessible from a remote workstation. The primary method is intended to be a remote shell using SSH.

- DoC startup

  As noted earlier, the hardware of the current EvBots (version II), has constraints, requiring that they start from a DiskOnChip (DoC) module. Being able to do so and then transfer to a different primary 'drive' is required.

- Logging

  System events which may help in debugging should be recorded and logged. In general, keeping these logs between boot cycles of the EvBots is not required. However, some logs may need to be persistent. Balancing this against the minimal hard disk use may be required.

3.3 Design Choices

One of the guidelines for the operating system was that it be open source or free, as discussed previously in section 3.2.1, and modifiable. Research into open source systems

and their capabilities leads to the choice of Linux for the OS [11]. The decision to use Linux, however, requires a further decision making process to determine how Linux will be implemented on the EvBot III. There were several options explored, with respective advantages and disadvantages.

3.3.1 Stripping an Existing Distribution

There are over a hundred easily available distributions of Linux with many different goals, hardware requirements, or intended users [8]. Of the many available distributions, very few meet the minimal hard drive space limitations imposed by the EvBot computer system hardware, even with their most basic installation. To use almost any Linux distribution it would have to be stripped of any excess not required for the EvBot use.

One advantages to using an existing Linux distribution is the generally large selection of available, generally well tested packages, which can easily be installed or un-installed. Additionally, development can take place on an installation of the same distribution (of the same hardware architecture).

On the negative side, some distributions do not provide support for older architectures and distributions occasionally change management or stop being developed. Significantly, Linux distributions are often targeted for desktop systems. A lot of work must be done to get a distribution small enough to fit on the EvBots, while still working properly.

3.3.2 Buildroot with uClibc

One of the largest contributors to overall size in a scaled down Linux operating system is often the main system function libraries, *libc* and some supporting libraries. These libraries provide shared versions of functions which are used by the main GNU utilities

Figure 3.1: Some Linux Distributions

that comprise the core of the Linux system. For smaller systems with limited space, an alternative to the larger GNU C Library (glibc) is available, known as uClibc. uClibc provides most of the functionality of *glibc*, but is much smaller, making it ideal for many embedded systems. [10]

A related project is *buildroot*, a build system that helps in building a small system around a *uClibc* core. It is highly configurable and relatively easy to use. This build system is one of the one of the reasons that makes this option rather attractive, as a major amount of the work is done. With appropriate tweaking, the entire build of the

final OS is mostly automated, and the resulting, fully functional, OS is very small. It is very easy to build a custom system with only the required software included.

The main disadvantages to using this type of system and resulting OS are the need for a specific development system or environment and occasionally, incompatibilities between glibc and uClibc.

3.3.3 Linux From Scratch

At the most basic, Linux from scratch is the process of compiling all the required libraries and programs from sources in a very specific order, resulting in a bare-bones Linux system. There are several guides and on-line books available which can guide a user through the process in great detail. The one referenced in this endeavor is named simply, Linux From Scratch. [7]

Even with a fast machine, the process of following all the directions can be tedious and time consuming. There have been some efforts to automate Linux From Scratch (LFS) builds, known as ALFS (Automated Linux From Scratch), which make the process much easier. [1]

One main advantage of building and using an LFS system is that only what is needed is added to the system. In addition, since the system uses glibc, development can occur on almost any system with a compatible version of glibc (assuming any additional libraries needed exist on both systems as well).

Unfortunately, the available 'build systems', to use the term loosely, are not very flexible or customizable, making any changes or additions more difficult. Additionally, a significant amount of work is involved in reducing the size of glibc and other libraries on the system to meet size constraints.

Figure 3.2: The Different Build Systems

3.3.4 OpenEmbedded

OpenEmbedded is a build framework for embedded Linux, designed to accomodate multiple hardware platforms [9]. Essentially a massive collection of scripts, OpenEmbedded, once configured from a central file, will perform all the steps needed to build a full operating system from scratch. Additionally, OpenEmbedded can make use of parallel computing using multiple processors or distributed compiler systems to reduce system development time.

There are similarities between OpenEmbedded and buildroot and some significant differences. The most notable difference, however, is that in OpenEmbedded, there are several *libc* options available, including *glibc*, *uClibc*, and *eglibc*. Buildroot is limited to *uClibc* only.

With only a change to specify the destination platform, OpenEmbedded can be used to build systems for different platforms. Additionally, OpenEmbedded supports thousands of packages and can be configured as sparsely or as feature rich as required. It can also build software development kits (SDK) for cross-compiling on a development machine as well as native SDKs for the target.

3.4 Design Implementation

The implementation of the EvBot Linux OS has currently been through three development cycles. Some limitations, discovered only at the conclusion of the first cycle of development, required re-evaluation. The second cycle of development provided a fully working system but the process was not maintainable. The third development cycle produced a similar system to that from the second cycle of development, but in an automated and easily reproducible manner.

3.4.1 Initial Buildroot/uClibc Build

Initially, the build system and small size of a buildroot/uClibc system made it the most attractive option. Getting a basic working system together was fairly easy. Likewise, further customizations and setup scripts to get things booting properly were generally straight-forward, if requiring a fair amount of testing to get things working as needed.

Problems arose when attempting to compile and install Octave on the system. Inital attempts to integrate this into the build system were not successful, so additional tools were installed on the developed OS in order to build Octave. Eventually, Octave completed compilation without complaint. However, testing revealed that there was an obvious issue, as the only answer returned was 'h.' Further attempts of variations in the methods used to compile Octave failed to change this behavior in the resulting binary. It is believed this was a library incompatibility due to the use of uClibc.

A very old version of Matlab had been in use on the previous operating system available on the EvBot II. With some work, this was copied to the new system and library paths set up in such a manner that it would operate. While this goes against the

Table 3.1: Failed Octave Example Results

Octave Command	Result
2 + 2	h
3 / 2	h
a = 42	h

guideline of open source, it is a tool used, and not an integral part of the OS. There was also some demand for a system that could be used immediately by various members of the lab for projects involving the EvBots.

A development environment which can be used on faster desktop systems is provided for this version of the EvBot OS. This consists of both an image that can be used in QEMU (an open source processor emulator) to boot a virtual machine as well as a copy of the root file system. The root filesystem can be utilized using the Linux *chroot* command, allowing a user to work within the constraints of that specific file system. Details are provided in appendix A.

3.4.2 LFS Build

With the eventual difficulties encountered using uClibc, later development switched to a LFS *glibc* based system. Fortunately, some of the work and tweaking involved in the previous iteration was applicable in this phase of the research.

One of the most useful tools available to developers of small systems is a piece of software called BusyBox. Hailed as the "Swiss Army Knife of Embedded Linux," BusyBox provides small versions of many common UNIX utilities, bundled into a single binary. While not always providing all of the options of the official versions, the primary capa-

bilities exist [4]. In an effort to keep size down, use was made of BusyBox for most of the core system, using full versions of utilities when needed. BusyBox has some association with the uClibc and buildroot projects and was also used in the uClibc/buildroot version of the EvBot OS.

Using some portions of the automated LFS tools, some additional scripts written, and a considerable amount of hands-on development, a base system was assembled. Going through many of the same issues previously, Octave was successfully compiled. Now, Octave worked as expected, producing valid results. The total size of the system is larger than the uClibc/buildroot system, but still within the limitations imposed by the Evbot hardware.

3.4.3 OpenEmbedded Build

Despite having a working system using LFS, the process involved was very cumbersome, even using scripts to automate as much as possible. Use of an ARM processor development board (the BeagleBoard [2]) led to direct exposure to projects using OpenEmbedded. Further investigation proved that it would be a good option to build the EvBotOS and all associated SDKs.

Using the knowledge gained from the previous development cycles for the EvBotOS, configurations were created for all the necessary libraries and packages (including Octave). There were several issues encountered again when compiling Octave, but these were determined to come from an incomplete tool-chain (missing a Fortran compiler). When that issue was resolved, a complete system, as well as an SDK, was created by issuing a single command.

The flexibility and convenience of OpenEmbedded was further proved by building the same system for a different hardware platform (namely the BeagleBoard mentioned previously). There was only one issue encountered during that process, and that due to an incomplete script for building Octave.

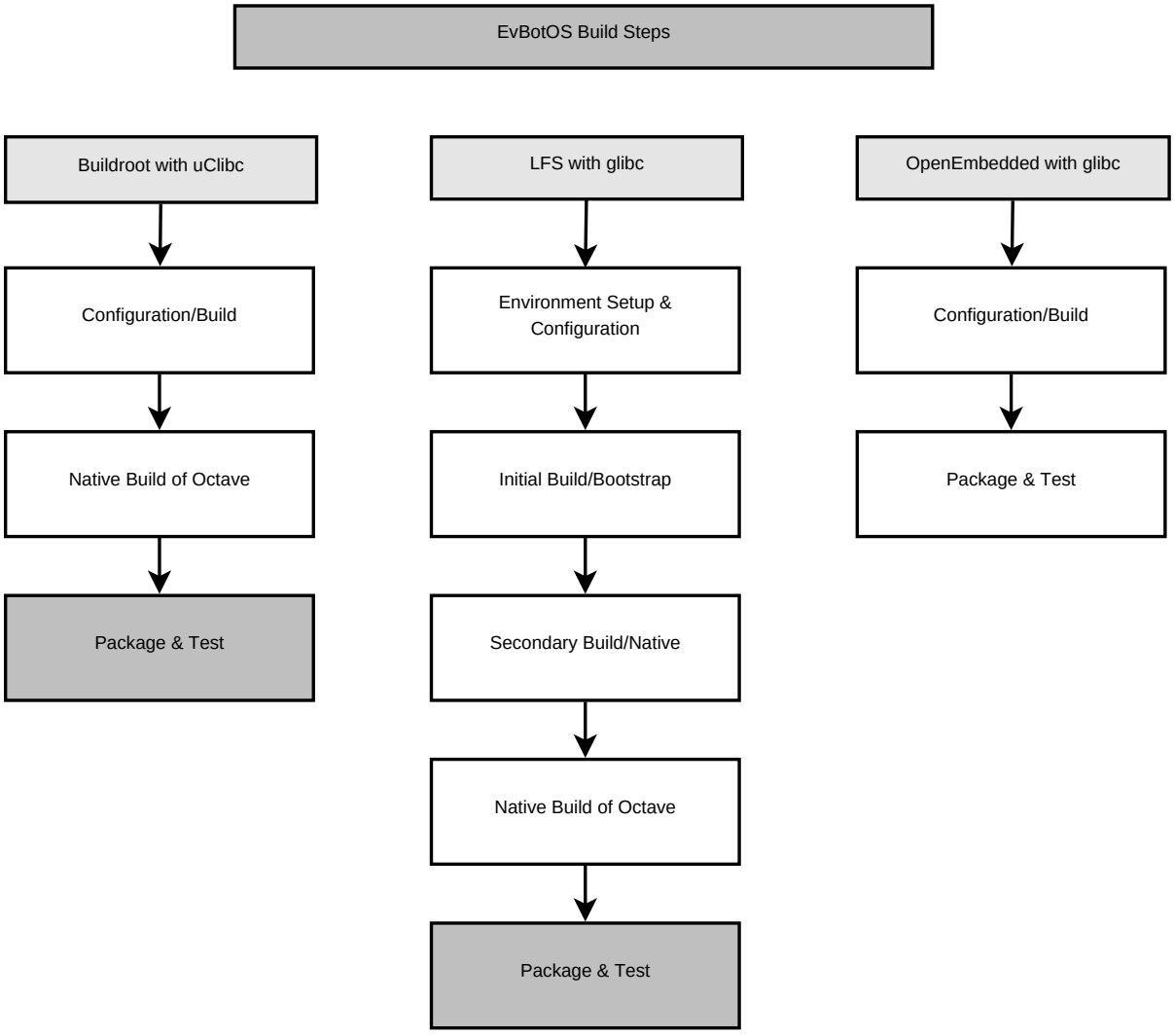

Figure 3.3: EvBotOS Build Steps

3.4.4 General Issues

A few general issues that either affected the development or will affect use of the EvBot OS are important to note.

3.4.4.1 Cross Compiling

Most development for the EvBot OS occured on newer and faster desktop machines running several different distributions and versions of Linux. In order for the resulting OS to run on the EvBot hardware, cross-compiling played a major role. Cross-compiling is the method by which a compiler is capable of creating output binaries for a specified architecture while running on a completely different architecture. Reasons behind this have been alluded to or mentioned previously (see section 3.2.2), but center on the fact that effective development can be difficult or impossible with the limitations of the current EvBot hardware.

Cross compiling also plays a role in software development for the EvBot OS. Direct development on the EvBots themselves is less desirable, due to longer compilation times, repeated writes to flash drives, and additional aspects of both software and hardware limitations. This does not mean that direct development on the EvBots themselves is strictly ruled out, but that it is intended as a final debugging or possibly emergency fix measure.

3.4.4.2 PCMCIA Support

Most use and handling of PCMCIA devices for standard systems generally occurs later in the boot process, after most of the main system drivers have been loaded. With the hardware on the EvBot II, access and use of PCMCIA devices is required early in the

boot process, as the main drive is a PCMCIA flash card. Crafting an appropriate kernel and start-up scripts required a fair amount of testing and tweaking to get everything running smoothly.

3.4.4.3 Additional Hardware Drivers and Software

Some hardware intended to be used with the EvBots required additional work to be supported. Drivers and occasionally supporting software had to be added, as default support was not available in the chosen kernel. The hardware requiring the most effort was USB cameras, not only in terms of drivers, but also the utilities necessary to access and make use of the cameras. With newer kernels, much of this support will be included, though upgrading to newer hardware may continue to cause issues.

3.4.4.4 GRUB versus LILO

Two of the common bootloaders (which start the process of getting the operating system loaded) are LILO and GRUB. LILO (**LI**nux **LO**ader) is an older boot loader that was used with many distributions before GRUB became popular. GRUB (**GR**and **U**nified **B**ootloader) is currently a very popular choice, with some impressive features.

One of the reasons GRUB is popular is because it can reference a configuration file on disk at runtime to modify its behavior. This contrasts sharply with LILO, which requires any modifications to its configuration to be written to the boot sector of a drive (or other appropriate sector, the boot sector being the most common). Without proper care, an improper configuration file for LILO, written to the boot sector, can cause the system to fail. Recovery from such an event, while not impossible, does require some significant

effort. GRUB, on the other hand, provides a small shell when it encounters configuration file issues which a user can use to recover or boot the system easily.

While GRUB is indeed very nice, there are some instances when dealing with specific hardware that it will not work. One particular instance is the DiskOnChip on the EvBot II; to get things working properly LILO had to be used.

3.4.5 Changes During Development

Technology continues to develop over time, rapidly making hardware obsolete. During the development of the EvBot OS there have been significant changes affecting future use. Increased capacity and dramatically lowered prices for flash drives reduce the hard drive constraints imposed on the overall operating system size. In addition, the current increased use of solid state devices as hard drives in many systems affects the imposed limitations on hard drive use. While usage on the current hardware platforms still impose these constraints, future use of the OS on newer hardware may not.

3.5 Final Results

Using OpenEmbedded and a minimal configuration, a working system with Octave and support for all the needed hardware was achieved. It is easily re-creatable, can be modified, and manages to fit within 65MB without any additional effort needed to minimize size. A software development kit is available on both the build system environment and a development image for the target system (which still remains less than 175MB). Additionally, the same configuration can be built for other hardware platforms with limited change to the build setup.

Chapter 4

EvBot Abstraction Layer

The EvBot Abstraction Layer (EAL) concept was developed over a series of discussions with various members of the CRIM research group at North Carolina State University, involved in various projects using the common EvBot platform. Some of the major requirements or desires that arose from these discussions were:

- An interest in making a modular system, allowing a user to plug together the necessary modules, and add a controller or 'brain' to make it work.
- A desire for standardized interfaces to similar hardware to avoid re-coding controllers if or when hardware changed.
- The need to simulate EvBots without controller modification (or requiring very minimal modification).

The EvBot Abstraction Layer attempts to satisfy all of these requirements and desires.

4.1 EAL Design and Overview

The EvBot Abstraction Layer is composed of two 'ends': that of the controller and that of the resources. The controller is the realm of the user, the AI, or whatever control architechture is in charge of the EvBot. The other end, that of the resources, is the connection to physical or simulated hardware. What lies between these two ends is the EAL framework, facilitating communication and simplifying access. The EvBot Abstraction Layer is designed to be flexible enough to handle most needs, at both the controller level and the resource level.

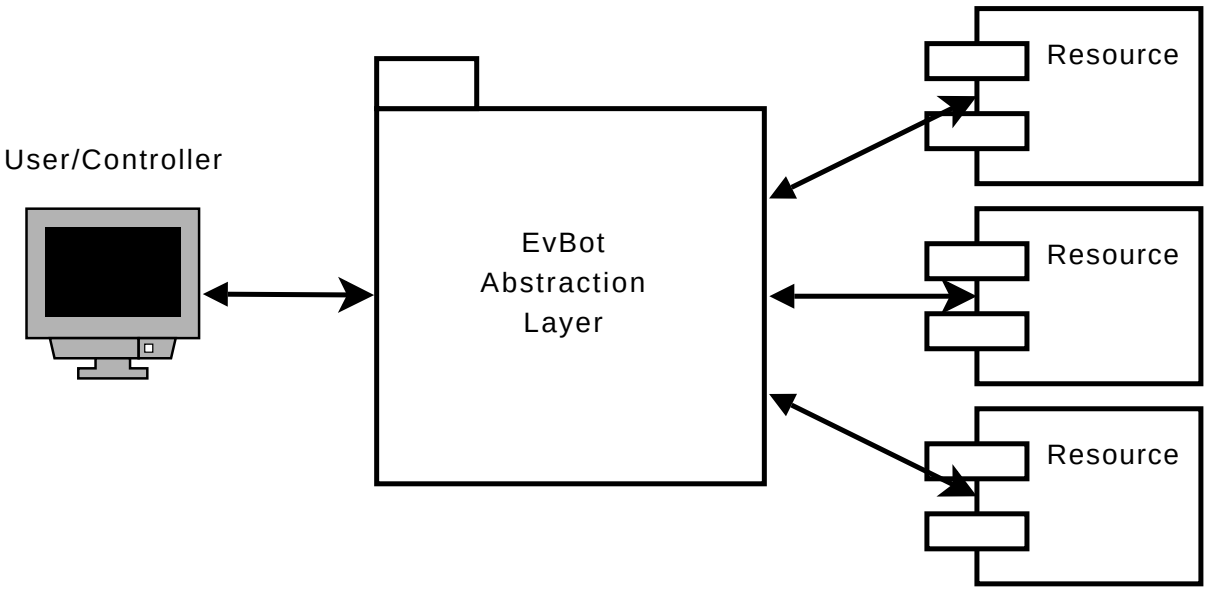

Figure 4.1: EvBot Abstraction Layer Overview

In this chapter we will present some details of the EvBot Abstraction Layer (EAL), working our way from resources through the framework and then to the user API. We will discuss some additions which allow EAL to be used over a network, without change to the controller, and touch briefly on some important aspects of data transfer and use within EAL. Finally, we will present sample code that puts together a small working system using EAL.

4.2 The Hardware Connection: EAL Resources and Plugins

The back end of EAL is where all connections to hardware, computational modules, or simulation takes place. Resources are the access points to specific EAL API calls while plugins are a collection of one or more resources with a minimal library wrapper. It is this wrapper which allows EAL to know what resources a plugin provides as well as get access to the resources within a plugin.

4.2.1 Resources

Resources are the EAL interface link, or wrapper, which manage the work of translating EAL API commands into access of machine hardware, simulation, or any other device or possibly software. There are a few rules which govern the way EAL communicates with resources as well as some guidelines which must be followed in order for EAL to be able to use a provided resource.

Resources are broken down into four broad types: Sensor, Actuator, Communication, Other. Sensors types provide data, Actuators manipulate something, Communication

types provide a path for general data flow, and Other is reserved for anything that falls outside the scope of the other types. A resource must define itself to be one of these types, or a combination thereof, depending on what it provides.

Table 4.1: Resource types and purpose

Sensor	Get an input
Actuator	Manipulate something
Communication	General communication
Other	Doesn't fit other categories

The type, or types, a resource is defined as does affect which API calls would be used with a resource. For example, a resource providing access to a simple bump sensor, specified as only a Sensor type would be able to get a value, as it is reactive only. A modified bump sensor which is extended on an arm to test some distance ahead would be both a Sensor and an Actuator, while a lever simply sweeping back and forth on command would likely be specified as only an Actuator.

4.2.1.1 EAL Resource API

The most basic resource provides these methods:

- a resource class constructor
- a resource class destructor
- create<Resource Name>()
- destroy<Resource Name>()

Obviously, providing just these methods isn't very useful, as it only allows an instance of the resource to be created and destroyed. Resources also provide some or all of the following methods from the EAL user API, with appropriate options (not shown here), described in more detail in section 4.4:

- getStatus()
- getValue()
- setValue()
- sendReq()
- sendCmd()
- readComm()
- writeComm()
- getConfig()
- setConfig()

4.2.1.2 create<Resource Name> and destroy<Resource Name>

There is a naming constraint for the create and destroy calls, which requires that the functions be the words 'create' or 'destroy' followed by the resource name. This allows other parts of EAL to create and destroy instances of this resource only knowing its name. Providing the name of resources to the rest of EAL is the primary function of plugins.

4.2.2 Plugins

An EAL plugin is a library of one or more resources. The plugin provides the EAL framework with the list of resources it contains, which the framework uses to access individual resources. The EAL framework requires that a resource *must* be placed in a plugin in order to be accessible.

4.2.2.1 EAL Plugin API

- connect()
- disconnect()
- listResources()

The creator of a plugin is responsible for specifying which resources the plugin provides. If additional resources are included but not explicitly named, EAL will not be able to use them. It is not possible to have two resources with the same name in the same plugin. However, there is not a requirement that all resource names be unique across all plugins. This case is discussed in section 4.3.2.

4.3 A Middle Skeleton: The EAL Framework

The EAL framework is the portal through which all messages between the user and resources passes. All loading of plugins and handling of associated resources takes place inside the EAL framework. Any configuration changes, setup or EAL behavior changes, have their basis here. The interfaces in and out of the EAL framework are handled by the EALResourceManager annd EALPluginManager, while the EALConfigManager handles configuration issues.

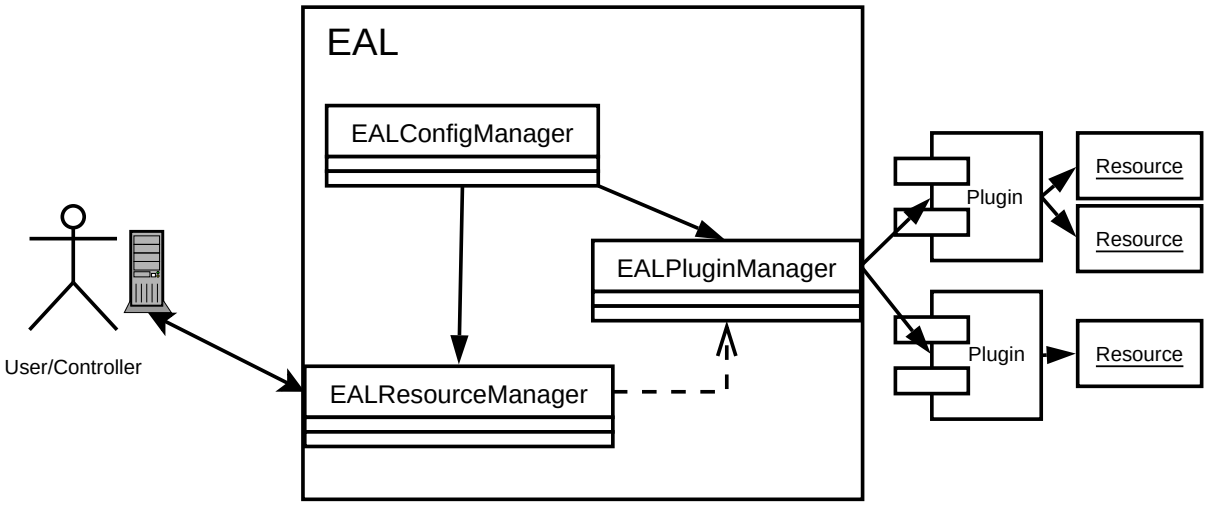

Figure 4.2: EAL Organization

4.3.1 EALConfigManager

The purpose of the EALConfigManager is to read all configuration files, handle any configuration changes, and make all necessary information available to other parts of EAL. At this point, the EALConfigManager is little more than a defined interface, as the methods simply return hard-coded values instead of accessing dynamic information. Further design or extension of EAL may expand the EALConfigManager capabilities.

4.3.2 EALPluginManager

The EALPluginManager is in charge of all the dynamic library issues associated with plugins and resources, making use of the EALConfigManager to determine appropriate directories and other configurations. It communicates with the EALResourceManager to provide requested resources from plugins and is ultimately responsible for returning those resources, when the EALResourceManager is done with them.

One of the features of the EALPluginManager is resource renaming. If two plugins provide resources with the same name, resources will be renamed within the EAL framework to make them distinct to the user. This can possibly affect user code when plugins are added, as the renaming occurs as resources are found. Renaming is handled through the concatenation of the plugin name, so intelligent handling of similarly named resources is feasible.

4.3.3 EALResourceManager

The EALResourceManager keeps track of all the resources requested or in use, makes requests to the EALPluginManager for resources as needed, and, with some sanity checks, passes on accesses to the specified resources.

As a part of of keeping track of resources, the EALResourceManager assigns resources numeric identifiers. All EAL methods that deal directly with a known resource require this numeric resource identifier, unless the method explicitly uses the resource name. This identifier, of type `EALResourceId_t`, is assigned when a resource is initially requested.

4.4 The Front End: The EAL User API

It is the user API that is of the most interest to a user of EAL, provided a system with existing plugins and associated resources exists that meets their needs. The API methods fall into four categories: connection, resource knowledge, structured resource access, and direct resource access.

All EAL methods which deal specifically with a single known resource require a numeric identifier, the type specified as `EALResourceId_t`, unless the method explicitly provides access using the resource name. This is the identifier that the EALResourceManager assigns when

Table 4.2: The EAL API by Category

Connection	`EALconnect()` `EALdisconnect()`

Resource Knowledge	`getResourceList()`
	`findResource()`
	`resourceInfo()`
	`resourceInfobyName()`
	`getStatus()`

Structured Resource Access	`getValue()`
	`setValue()`
	`sendReq()`
	`sendCmd()`
	`readComm()`
	`writeComm()`
	`getConfig()`
	`setConfig()`

Direct Resource Access	`getResource()` `returnResource()`

resources are initially requested. There is no guarantee that this identifier will be the same for different connections to the EAL framework.

4.4.1 Connection

It is necessary to connect to EAL before any access is possible; users should also close the connection when done using the interface.

- `EAL* EALconnect()`
- `void EALdisconnect( EAL *eal )`

4.4.2 Resource Knowledge

The resource knowledge methods allow the user to determine what resources are available, get more information about resources, find a resource identifier, or get the status of a resource.

`EALResourceId_t`, as discussed in section 4.3.3, is a numeric identifier and `EALResourceInfo_t` is a structure containing more details about a resource.

- `list<string>* getResourceList()`
- `EALResourceId_t findResource( const char *resourceName )`
- `EALResourceInfo_t* resourceInfo( EALResourceId_t id )`
- `EALResourceInfo_t* resourceInfobyName( const char *resourceName )`
- `getStatus( EALResourceId_t id )`

4.4.3 Structured Resource Access

Structured resource access methods are used to get data to and from resources. Not all resources will provide *all* of these methods, instead, providing the ones that make the most sense for the resource.

`EALData_t` is a data encapsulation structure used in EAL, discussed in further detail in section 4.6.

- `EALData_t getValue( EALResourceId_t id )`
- `int setValue( EALResourceId_t id, EALData_t data )`
- `EALData_t data sendReq( EALResourceId_t id, const char *req )`
- `int sendCmd( EALResourceId_t id, const char *cmd, EALData_t data )`

- `int readComm( EALResourceId_t id, EALData_t data, unsigned int length )`
- `int writeComm( EALResourceId_t id, EALData_t data, unsigned int length )`
- `EALData_t data getConfig( EALResourceId_t id, const char *config )`
- `int setConfig( EALResourceId_t id, const char *req, EALData_t data )`

Some resources, simple sensors or actuators, do not require or allow anything other than "get this" or "do that." The `getValue()` and `setValue()` methods are the simplest resource access calls.

The `sendReq()` and `sendCmd()` methods offer the most flexibility of the structured resource access methods. Most needs can probably be handled through these methods, if the resource is complicated enough to require them.

`readComm()` and `writeComm()` are provided, as their names imply, primarily for communication resources. Reading and writing to a serial port is a likely candidate.

Finally, not all resources can, or should, be used without some user specified configuration. The `getConfig()` and `setConfig()` methods are essentially the same as the flexible `sendReq()` and `sendCmd()` methods, but use a different naming scheme to enhance readability of user and resource code.

4.4.3.1 Direct Resource Access

The direct resource access methods allow the user to access a resource object directly. This provides a way for the user to access available resources without the overhead of methods passing through the EAL framework or use methods provided by a resource that cannot be accessed by the EAL interface. It can also be used for debugging purposes for developers of resources as they implement the EAL API.

- `EALResource* getResource( const char* resourceName )`
- `int returnResource( EALResource **resource )`

The EALResourceManager keeps track of which resources are 'checked' out with the `getResource()` method, much like a librarian. It is important to note that any resource that is accessed using the `getResource()` method must be returned to the EAL framework using the `returnResource()` method or the program will be unable properly to disconnect from EAL.

4.5 Moving Further Away: The EAL Network Layer

Early in the design of EAL, network was an integral part of the framework. Over time, the design changed as various implementation details were worked out and network was moved out of the core framework. Instead, network access is provided as a layer between the user and the EAL framework, using the standard client/server model.

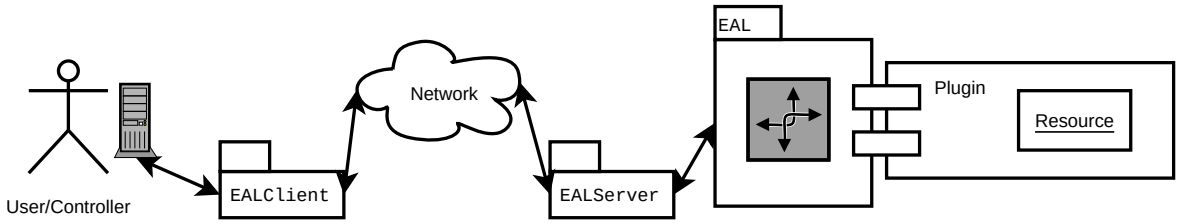

Figure 4.3: The EAL Network Layer

The EALClient and EALServer working together make the existence of the network as invisible to the user as possible. In general, this succeeds. There are only two changes to the API for the user.

1. EALconnect() adds two parameters, `server` and `port`. Respectively, these indicate the hostname of the server and the port number of the EALServer to connect to.

2. The `getResource()` and `returnResource()` methods are not available when connecting to the EAL framework using the network. This includes connections established on the same machine which use the EALClient/EALServer connection.

The overhead involved in making the network layer transparent to the user, when using the main API methods, is significant enough that even simple programs will notice the delay. However, the additional processing power of a remote controller may be worth the delays, depending on the application.

4.6 EAL Data Encapsulation

In order for EAL to accomodate different resources and the near infinite variety of data types from both resources and users, it must generalize the flow of data in some way. The simplest way of doing this would be to use the `void *` construct of C and C++. This works fairly well in many instances, with one end of data flow interpreting the data as needed. However, it is no longer adequate with the addition of network layers or multiple unique accesses to the same resource.

The main drawback to using `void *` is that the size of the data is unknown. This is especially important when it comes to the network layer as all of the data must be transfered successfully from one side of the connection to another. To get around this, as well as some other issues, EAL utilizes some data constructs.

The primary construct is a specific data structure, `EALDataInfo_t`, which incorporates the data payload plus some additional information specifying what type of data is in the payload. Using this data a user (or resource) can determine the size and type of the data, or act as needed. A constructor function is provided that makes working with this data structure easier.

The general data construct is a generic type, `EALData_t`. This is the main data type used for general data transfers in the EAL framework. To avoid copying the overhead of a large data structure through successive function calls, this is defined as a pointer to a `EALDataInfo_t` structure.

4.7 Getting Dirty: Sample Code using the EAL API

The following code examples include a sample resource, a plugin with that resource, and some user code that uses EAL to access the sample resource.

4.7.1 Sample Resource Header

The sample resource header provides the class definition for `SampleResource`, which we will define as both Sensor and Actuator types. It uses the simple `getValue()` and `setValue()` EALResource methods.

Not all of the EALResource methods have to be defined in `SampleResource` because it is using `EALBaseResource`, which implements methods for them. The implemented methods in `EALBaseResource` return null or invalid, depending on the return type.

Table 4.3: SampleResource header

```
// SampleResource.h

#include <EALBaseResource.h>

class SampleResource: public EALBaseResource {

  private:

    int sampleValue;

  public:

    SampleResource();

    virtual EALData_t getValue();
    virtual int setValue( EALData_t value );
};
```

4.7.2 Sample Resource Code

The resource code implements the class `SampleResource` and also includes the necessary resource functions `createSampleResource()` and `destroySampleResource()` as required (see section 4.2.1.2).

You can see that the create and destroy functions are defined as well as the resource constructor. The type of the resource is set to be both SENSOR and ACTUATOR and the initial value sampleResource is set to zero. Additionally, the `getValue()` and `setValue()` methods are defined.

Table 4.4: Beginning of SampleResource Code

```
// SampleResource.cpp

#include "SampleResource.h"
#include <iostream>
using std::cerr;

extern "C" EALResource* createSampleResource() {
    return new SampleResource;
}

extern "C" void destroySampleResource(EALResource* p) {
    delete p;
}

SampleResource::SampleResource() {
  info_.type = SENSOR | ACTUATOR;
  sampleValue = 0;
}
```

Table 4.5: Defining `getValue()`

```
EALData_t
SampleResource::getValue() {
  int* retValue;

  try {
    retValue = new int;
  }
  catch( bad_alloc& ) {
    cerr << "Could not allocate memory.\n";
    return NULL;
  }

  *retValue = sampleValue;
  return new EALDataInfo_t(INT, retValue);
}
```

Table 4.6: Defining `setValue()`

```
int
SampleResource::setValue( EALData_t value ) {

  if( value ) {
    if( value->type == INT ) {
      // type cast the data pointer from void* to int*,
      // then dereference
      sampleValue = *((int*)(value->data));
      return 1;
    }
  }

  // failure
  return -1;
}
```

4.7.3 Sample Plugin

Table 4.7: Sample Resource Plugin

```
// pluginSampleResource.cpp

#include <EALPlugin.h>

extern "C" EALPlugin* connect() {
  return new EALPlugin("SampleResource", NULL);

  // The alternate form is to specify the number
  // of resources without the final NULL parameter:
  //
  //  return new EALPlugin(1, "SampleResource");
  //
}


extern "C" void disconnect( EALPlugin* p ) {
  delete p;
}
```

4.7.4 Sample User Code

```
// sampleUserCode.cpp

#include <EAL.h>
#include <iostream>
#include <list>
#include <string>
using namespace std;

int main( int argc, char *argv[] ) {

  EAL* eal;
  EALResourceId_t id;
  EALData_t value;
  list<string>* resourceList;

  eal = EALconnect();
  if( !eal ) {
    cerr << "Could not connect to EAL\n";
    exit(0);
  }

  resourceList = eal->getResourceList();
  while( !resourceList->empty() ) {
    cout << "Available resource: " << resourceList->front() << "\n";
    resourceList->pop_front();
  }

  id = eal->findResource("SampleResource");
  if( id < 0 ) {
    cout << "Could not find SampleResource\n";
  }
  else {
    value = eal->getValue( id );
    if( value ) {
      cout << "SampleResource value = " << *((int*)(value->data)) << "\n";
      *((int*)value->data) = 42;
      eal->setValue( id, value );
      delete value;
    }

    value = eal->getValue( id );
    if( value ) {
      cout << "SampleResource value is now = " << *((int*)(value->data)) << "\n";
      delete value;
    }
  }

  EALdisconnect( eal );

  return 1;
}
```

Chapter 5

Operating System Results

An operating system meeting the feature set originally described in section 3.2.3 was successfully created using OpenEmbedded. The ability of the build system to do the same for multiple hardware platforms was tested using several systems. Additionally, verification of Octave as a Matlab replacement was conducted and some hardware performance benchmarks were conducted using Octave.

5.1 OS Build System

The build system was tested by building a full development image for three different hardware platforms. These three platforms are a PC/104 MZ104, a PC/104 SBC1496, and a BeagleBoard.

5.1.1 Hardware

5.1.1.1 PC/104 MZ104

The PC/104 MZ104 from Tri-M is a Pentium/586 class motherboard with integrated keyboard, mouse, floppy, hard disk, serial, parallel interfaces, and USB 1.1 interfaces. The CPU runs at 100MHz and the there is 64MB RAM. The MZ104 was also stacked with a PCMCIA PC/104 module to add support for wireless networking and a mass storage device for the operating system. The networking is provided by a Cisco Airo 340 PCMCIA card and the the flash drive is a 96MB Kingston Technology PCMCIA card.

For the tests, an additional PC/104 module providing IDE access to a CompactFlash card was added to the stack, allowing for full testing before the DoC was modified to support the new OS.

Figure 5.1: The MZ104 Board

5.1.1.2 PC/104 SBC1496

The PC/104 SBC1496 motherboard is made by Micro/Sys and contains an Atlas 486DX processor running at 133MHz, integrated keyboard, mouse, floppy, and serial interfaces. It provides six USB ports, two support USB 1.1 and the remaining four support version USB 2.0. Mass storage is provided via a CompactFlash connector. Memory is limited to 64MB of onboard RAM. Networking is provided by a USB network adapter.

Figure 5.2: The SB1496 Board

5.1.1.3 BeagleBoard

The BeagleBoard is a small development board with a Texas Instruments OMAP3530 processor, including an ARM Cortex-A8 core running at 500MHz and an integrated

TMS320C64x+ DSP. The board includes 128MB RAM and support for SDHC mass storage, as well as integrated USB 2.0 support. The board also has 256MB NAND flash, but it was not utilized during these tests. Networking is provided by a USB network adapter.

Figure 5.3: The BeagleBoard

5.1.2 Build Setup and Configuration

The setup for all three hardware platforms is almost identical, with the most difference existing between the arm processor BeagleBoard and the x86 processors of the MZ104 and SBC1496. The primary changes required are platform specification, kernel configuration and image definition.

5.1.2.1 Platform Specification

OpenEmbedded requires a configuration file that describes the platform being targeted for the build and the build method chosen. The minimum requirements are the processor and the OS type (specifically, Linux with a choice of *glibc*, *uClibc*, or *eglibc*). As noted in section 3.3.4, OpenEmbedded can make use of multiple processors. The configuration file is where the number of processors to use is specified, though it should be limited to the number of processors actually available.

The specific processor and generic x86 hardware configurations were chosen for the MZ104 and SBC1496 platforms. The BeagleBoard is a known hardware configuration in OpenEmbedded so the default configuration for it was used.

5.1.2.2 Kernel Configuration

Building a proper Linux kernel that contains all the necessary support for a particular hardware platform is relatively easy if you choose to include all the drivers available. However, this bloats the kernel and increases build times. The other extreme of kernel building is to include just the bare minimum of drivers needed for the intended hardware.

For the tests, a mostly generic kernel configuration was chosen, removing support for some broad hardware categories (such as SCSI hardware). Generic IDE support was marked to be built in and a broad category of networking and USB drivers were marked to be built as modules. PCMCIA support was added for the MZ104.

The BeagleBoard is supported in OpenEmbedded so no additional kernel configuration was required.

5.1.2.3 Image Definition

The image definition is where the software that is to be installed on the target system is specified. There is a *base-image* definition that includes everyting necessary to boot the system and remote access it via ssh. This definition was expanded to include any other necessary system tools as well as the development packages.

The *base-image* definition does not expect the primary network device to be wireless, so the *wireless-tools* package needed to be added (including the command iwconfig). Additionally, PCMCIA devices are getting older, and the required *pcmciautils* package was also required for the MZ104 definition. Other packages specified added octave and several image libraries (jpeg, png, tiff) to the final OS image.

5.1.3 Build Times

Build times will vary depending on the additional packages requested and their availability, as well as the speed of the machine doing the build. All recipes specify their source file location via a URI, and are downloaded when needed. For slow connections this can take some time, but once everything is downloaded, this does not contribute any additional time for subsequent builds.

Two machines were used for testing. The first is a dual processor desktop with two Intel Pentium 4 CPUs running at 3.00GHz, with 4GB of RAM. The second is a dual processor server with two Intel Xeon quad-core CPUs running at 2.66GHz (for a total of eight cores), with 16GB of RAM. Table 5.1 lists build times of the base image for comparison.

Table 5.1: Build Times for Base Image (time in hour:min:sec)

	Two 3.00GHz cores	Eight 2.66GHz cores
MZ104	7:38:51	1:13:00
SBC1496	7:42:41	1:13:29
BeagleBoard	7:50:27	1:18:38

5.2 Octave Benchmarks

One of the prime goals of the operating system build was to have Octave compiled and running. While this was accomplished, some testing of Octave was required to determine whether it was a valid replacement of Matlab. A benchmark written for Matlab and clones by Derek O'Connor [3] was modified slightly for this purpose and used as both a test to compare Matlab and Octave as well as a general performance guideline.

5.2.1 Matlab vs. Octave

The benchmark was run by both Matlab and Octave on the same MZ104 platform using the new Linux operating system and a previous version that included the Matlab executables. As can be seen in Table 5.2, Octave did marginally better for almost all the tests, but was slightly slower at the beginning of the benchmark. Unfortunately Matlab was unable to complete the benchmark running on the MZ104 as it ran out of memory for the last two tests. Octave was able to complete the full benchmark, with a total run time of 2.453 hours. For comparison, a 2GHz desktop machine completed the same test in 26.51 seconds.

Table 5.2: Matlab vs. Octave on MZ104 (time in seconds)

	Matlab	**Octave**
1000x1000 Creation	3.23	5.61
Multiplication	927.22	1157.30
LU Decomposition	398.77	380.58
Inverse	1190.38	1142.08
Singular Value Decomposition	1537.18	1488.04
QR Factorization	699.50	680.46
Eigenvalues and Eigenvectores	NA	3574.38
Compute Rank	NA	403.51

5.2.2 Octave Performance Measurement

The same Octave benchmark was run on the SBC1496 and the BeagleBoard as well. Table 5.3 lists the results. Considering the difference in processors and speeds, the results are fairly predictable, with the SBC1496 486DX processor, though running at a higher clock rate than the MZ104, achieving the slowest results and the BeagleBoard the best results.

Table 5.3: Octave on MZ104, SB1496, and BeagleBoard (time in seconds)

	MZ104	**SBC1496**	**BeagleBoard**
1000x1000 Creation	5.61	6.75	0.51
Multiplication	1157.30	1460.95	272.19
LU Decomposition	380.58	462.66	105.51
Inverse	1142.08	1412.69	317.17
Singular Value Decomposition	1488.04	1851.99	420.96
QR Factorization	680.46	850.86	203.85
Eigenvalues and Eigenvectores	3574.38	4433.89	1103.65
Compute Rank	403.51	498.84	107.41
Total Time	8831.96	10978.63	2531.25

Chapter 6

EAL Experiments

Some experiments were conducted to prove that EAL can meet the design requirements. These experiments tested the coding ease to control a single robot as well as the ability of EAL to control multiple robotic platforms with different hardware in a similar manner. Additionally, experiments were conducted to indicate the success of EAL as a framework for distributed sensor systems.

6.1 Robotic Hardware

6.1.1 EvBot II

The EvBot II hardware is built on a modified base of a radio controlled vehicle [31] with a tank-like track system. It is capable of forward and backward motion and spins in either direction. The main controller for the system is the MZ104 described in section 5.1.1.1 and has an optional USB camera. For the experiments, the system used the new EvBot Linux operating system.

Figure 6.1: The EvBot II

6.1.2 EvBot III

The EvBot III hardware is a completely custom hardware platform with a two wheeled base, each capable of independent motion. The EvBot III has a top section that can rotate independantly from the wheeled base. The base control system uses the BeagleBoard mentioned in 5.1.1.3, but does not use the EvBot Linux operating system. Primary communication with the base from the turret is through an 802.11b wireless network. [5]

There is no current vision system for the EvBot III but it can be managed easily by the use of a small laptop computer with an integrated webcam. For some experiments a Samsung NC10 running Ubuntu Linux was used for this purpose.

Figure 6.2: The EvBot III

6.1.3 Mote Roomba

The Mote Roomba is a modified iRobot Roomba (model 4100), controlled with serial commands (per the iRobot Serial Command Interface) issued by a connected Berkley Mote IV. The mote on the Roomba accepts radio packets from a secondary mote connected to the controlling machine via USB. Commands issued to the mote from the controlling machine are passed to the mote on the Roomba and the requested command is passed to the Roomba.

Due to the signal strength of the radios on the motes, the distance between the controlling machine and the Roomba should not exceed 100 feet for reliable communication. However, the distance during experiments did not exceed 50 feet so this was not an issue.

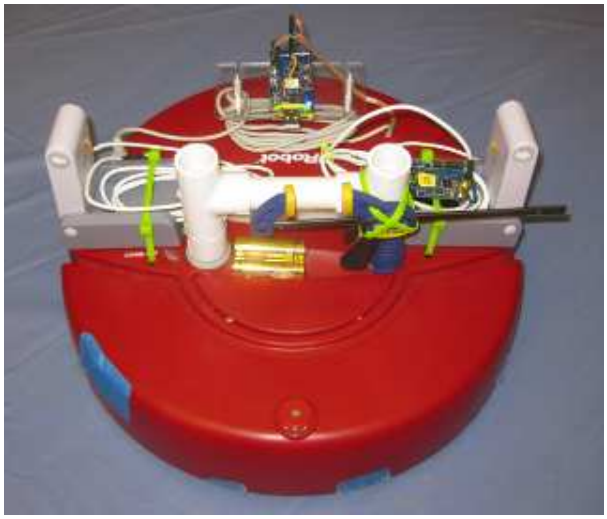

Figure 6.3: The Mote Roomba

6.2 Experiments

6.2.1 Drive Test

The Drive test is a simple test of the ability to drive the base platform around a square. The only commands issues to the base plugin are "left", "right", "go", "back", and "stop" (coded for simplicity as "l", "r", "g", "b", and "s"). The simplicity of this test requires that the "Base" plugin commands drive the base the same distance forward and backward and rotates the base 90 degrees either right or left. The code is listed in Table 6.1.

For a perfect drive system on the base, this would drive the robot in a square in a clock-wise direction and then retrace the exact square moving backwards. However, as can be seen in Figure 6.4 provided, the control of the bases is not that precise.

Table 6.1: EAL Code for Driving a Square

```
// Drive the robot in a square forward and backward
#include <iostream>
#include <EAL.h>

using namespace std;

int main( int argc, char* argv[] ) {

  EAL* client = 0;
  EALResourceId_t id = -1;
  int i = 0;

  client = EALconnect();
  if( client == NULL ) {
    cout << "Error connecting to EAL" << endl;
    return 1;
  }

  id = client->findResource( "Base" );

  for( i=0; i < 4; i++ ) {
    client.sendCmd( id, "g", NULL );
    client.sendCmd( id, "r", NULL );
  }
  client.sendCmd( id, "s", NULL );

  for( i=0; i < 4; i++ ) {
    client.sendCmd( id, "l", NULL );
    client.sendCmd( id, "b", NULL );
  }
  client.sendCmd( id, "s", NULL );

  EALdisconnect( client );

  return(0);
}
```

6.2.2 Image Capture with Remote Sensor

To test the network capabilities of the EAL framework with more complicated data a Camera plugin was created that interfaced with the webcam on the EvBot II. The "Camera" plugin, only provides `getValue()` method, which returns a single captured frame. The test captures a series of images in sequence as the robot turns in place, as

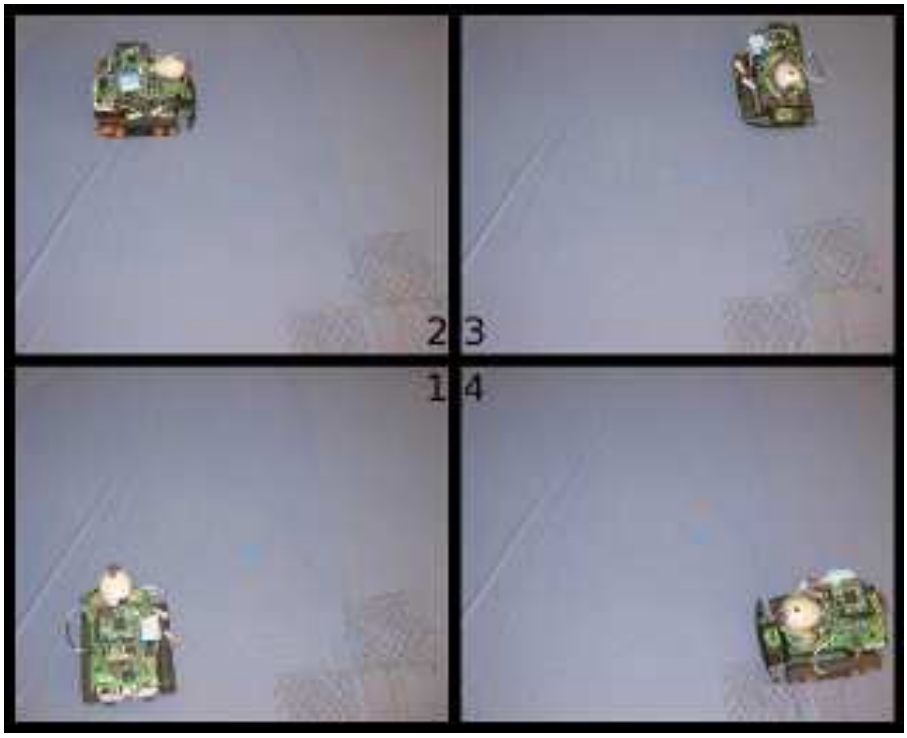

Figure 6.4: Square Drive Motion

might be used for panoramic stitching of images, mapping, or some form of detection algorithm. Enhancements have been made to the "Base" plugin to allow for smaller rotations than 90 degrees. The `saveFrame()` function, externally defined, saves the frame to the specified filename, with a numeric and format suffix, such as "/tmp/frame.0.jpg". Code is listed in Table 6.2.

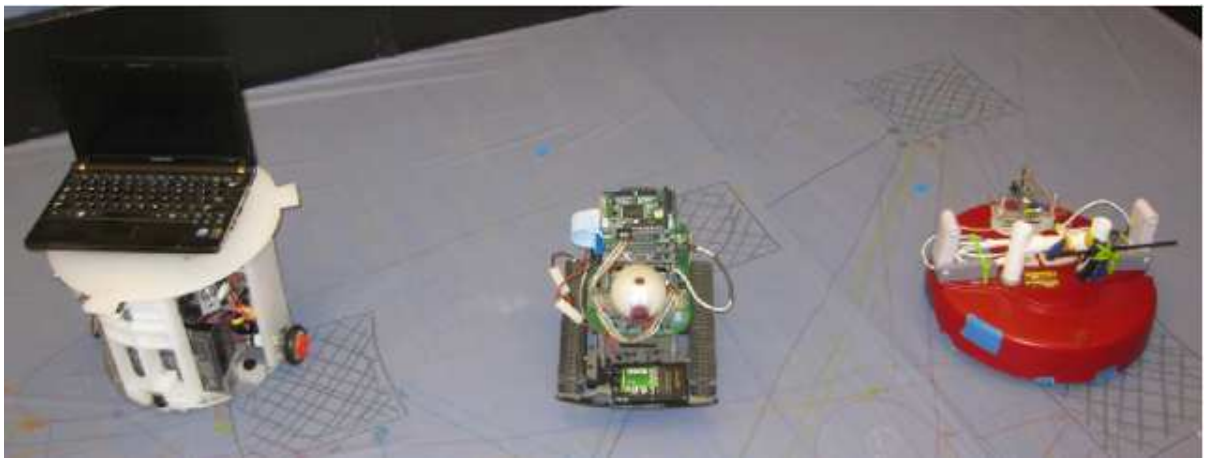

Figure 6.5: The Robot Dance

6.2.3 Multiple Robot Control

To illustrate controlling multiple robots from the same control code, the test has the robots follow a short sequence of 'dance' moves. Moves are issues from the same controller and time is provided for all robots to complete their step before moving on to the next one. Code is listed in Table 6.3.

Table 6.2: EAL Code for Camera Capture While Rotating

```
// Rotate the robot taking pictures periodically
#include <iostream>
#include <stdlib.h>
#include <EAL.h>
#include "frame.h"

using namespace std;

int main( int argc, char* argv[] ) {

  EAL* client = 0;
  EALResourceId_t baseId = -1;
  EALResourceId_t cameraId = -1;
  EALData_t frame;

  // argv[1] = hostname
  // argv[2] = hostname
  if( argc < 2 ) {
    return 0;
  }

  client = EALconnect( argv[1], atoi(argv[2]) );
  if( client == NULL ) {
    cout << "Error connecting to EAL" << endl;
    return 1;
  }

  baseId = client->findResource( "Base" );
  cameraId = client->findResource( "Camera" );

  for( int i=0; i < 7; i++ ) {
    client.sendCmd( baseId, "r45", NULL );
    frame = client.getValue( cameraId );
    saveFrame( "/tmp/frame", i );
  }

  EALdisconnect( client );

  return(0);
}
```

Table 6.3: EAL Code for Robot Dance

```
// Group Robot Dance
#include <iostream>
#include <stdlib.h>
#include <EAL.h>

using namespace std;

int main( int argc, char* argv[] ) {

  EAL* client[3];
  EALResourceId_t id[3];
  int i = 0;
  const char* dance[] = { "r45", "l90", "r90",
                          "l10", "r10", "l10", "r360" };

  // arguments are hostnames (up to three)
  if( argc < 2 )
    return 0;

  // connect to robots
  for( i=0; i < argc-1; i++ ) {
    client[i] = EALconnect( argv[i+1], 50000 );
    if( client[i] ) id[i] = client[i]->findResource("Base");
  }

  // do the dance
  for( int step=0; step < 7; step++ ) {
    for( i=0; i < argc-1; i++ ) {
      if( client[i] == NULL ) continue;
      client[i]->sendCmd( id[i], dance[step], NULL );
    }
    sleep(3);    // for slower robots
  }

  // disconnect each client
  for( i=0; i < argc-1; i++ ) {
    if( client[i] )
      EALdisconnect( client[i] );
  }

  return(0);
}
```

Chapter 7

Conclusion and Further Development

7.1 Concluding Remarks

The goals of the research presented in this thesis were to provide a current, open source, and updatable operating system and a modular and standardized programming interface for the Evolutionary Robotic Platforms.

Linux was chosen as the operating system for its stability, consistency across platforms, the fact that it is free and open source, and it can be updated and modified as needed. Using the OpenEmbedded build system, the resulting Linux distribution is easily customized and portable to multiple hardware platforms. With sufficiently powerful hardware it can be updated with packages and additional software in a few minutes or recreated completely from source in a little more than an hour. Software development environments are created during the OS build and are available for both the develop-

ment platform and natively on the robot. All parts of the operating system, supporting software, and the build system are free and open source so it can be shared and modified as desired.

The EvBot Abstraction Layer (EAL) framework and API provides a simple and consistent interface to a variety of hardware and software resources. The modular system of plugins provides for easy addition and removal of hardware or software systems support with little impact on user control code. With only seventeen methods to learn, and possibly only the need to use three or four, users can quickly get their code working with little to no knowledge of hardware. The network layer of EAL makes control code development even faster by allowing algorithms to be developed and debugged on a desktop machine while still acessing the sensors and actuators on the robot using the same interface.

The research work presented in this thesis has resulted in a usable Linux operating system for use by mobile robots as well as an application programming interface and framework that allows for simple control code and great flexibility in sensor and actuator manipulation.

7.2 Further Development

The operating system will occasionally need updating to support new hardware, performance enhancments, and other customizations. With the on-going support of OpenEmbedded, support for the latest kernels and operating system software should be available for the forseeable future. However, there will be a need for support of those custom

packages or additions, such as additonal Octave toolboxes. It is also possible, however, to use the native development system to provide these directly.

The current implementation of the flexibility and network capability of EAL does have some issues that may affect its scalability and use in some instances.

The early design of EAL did not consider the possibilty of controlling multiple robots. The code is very linear most sections are not thread safe at all. Reworking the framework of EAL as a leaner threaded system would add speed and flexibility when dealing with multiple systems that may be delayed for any number of reasons. Additionally, it would also likely increase the performance and applicability for local applications, especially those dealing with realtime issues.

The networking ability of EAL could also use some further development. While robust enough for current use, the previously mentioned lack of threading and depedence of connection oriented traffice (TCP) is not highly scalable. Addressing those issues as well as some continued development involving compression of data streams and encryption would benefit the robustness of the framework.

REFERENCES

[1] Automated linux from scratch. http://www.linuxfromscratch.org/alfs/.

[2] Beagleboard. http://beagleboard.org/hardware/.

[3] Benchmarks – Matlab & Clones. http://www.derekroconnor.net/Software/Benchmarks.htm.

[4] Busybox. http://www.busybox.net/.

[5] Communication with Matt Craver Regarding EvBot III Hardware.

[6] Corba faq. http://www.omg.org/gettingstarted/corbafaq.htm.

[7] Linux from scratch. http://www.linuxfromscratch.org/.

[8] List of linux distributions on ibiblio.org. http://distro.ibiblio.org/pub/linux/distributions/.

[9] Openembedded. http://wiki.openembedded.net/index.php/Main_Page.

[10] uclibc. http://www.uclibc.org/.

[11] Wikipedia Entry: Comparison of Open Source Operating System. http://en.wikipedia.org/wiki/Comparison_of_open_source_operating_system%s.

[12] Erin Anderson, William Cox, Mike Faircloth, Sterling Greene, and Frankie Myers. SEAWOLF II: Autonomous Underwater Vehicle. Technical report, North Carolina State Univesity, Raleigh, NC, 2006.

[13] Noriaki Ando, Takashi Suehiro, Kosei Kitagaki, Tetsuo Kotoku, and Woo-Keun Yoon. RT-Middleware: Distributed Component Middleware for RT (RobotTechnology). 2005.

[14] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback. Brooks, a.; kaupp, t.; makarenko, a.; williams, s.; oreback, a. In *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 163–168. IEEE, August 2005. unread.

[15] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Oreback. chapter Orca: a Component Model and Repository. 2006.

[16] Rodney A. Brooks. Artificial life and real robots. In *Proceedings of the First European Conference on Artificial Life*, pages 3–10. MIT Press, 1992.

[17] Greg Broten, Simon Monckton, Jared Giesbrecht, and Jack Collier. Software Systems for Robotics: An Applied Research Perspective. *Advanced Robotic Systems, International Journal of*, 3(1):011–016, 2006. unread.

[18] G. Caprari, K. O. Arras, and R. Siegwart. The autonomous miniature robot alice: from prototypes to applications. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 793–798. IEEE Press, 2000.

[19] L. Chaimowicz, A. Cowley, V. Sabella, and CJ Taylor. ROCI: a distributed framework for multi-robot perception and control. In *Intelligent Robots and Systems., Proceedings, IEEE/RSJ International Conference on*, volume 1, 2003.

[20] Eric Colon, Hichem Sahli, and Yvan Baudoin. CoRoBa, a multi mobile robot control and simulation framework. *Advanced Robotic Systems, International Journal of*, 3(1):073–078, 2006.

[21] William Cox, Mike Faircloth, Sterling Greene, Frankie Myers, Jim Simpson, and Ryan Sturmer. SEAWOLF I: Autonomous Underwater Vehicle Platform. Technical report, North Carolina State Univesity, Raleigh, NC, 2005.

[22] S. Fleury, M. Herrb, R. Chatila, and T. CNRS. Design of a modular architecture for autonomous robot. In *Robotics and Automation., Proceedings, IEEE International Conference on*, pages 3508–3513, 1994.

[23] John M. Galeotti. The Evbot: A Small Autonomous Mobile Robot for the Study of Evolutionary Algorithms in Distributed Robotics. Master's thesis, North Carolina State University, 2002.

[24] Jay Gowdy. A Qualitative Comparison of Interprocess Communications Toolkits for Robotics. Technical Report CMU-RI-TR-00-16, Robotics Institute, Pittsburgh, PA, June 2000.

[25] Steve R. Hastings. Lindows 4.0. *Linux Journal*, December 2003.

[26] Patrick Hohmann, Uwe Gerecke, and Bernardo Wagner. A Scalable Processing Box for Systems Engineering Teaching with Robotics. In *Systems Engineering., International Conference on*, Coventry, UK, September 2003.

[27] Dayang N. A. Jawawi, Rosbi Mamat, and Safaai Deris. A Component-Oriented Programming for Embedded Mobile Robot Software. *Advanced Robotic Systems, International Journal of*, 4(3):371–380, 2007.

[28] F. Kanehiro, H. Hirukawa, and S. Kajita. OpenHRP: Open Architecture Humanoid Robotics Platform. *The International Journal of Robotics Research*, 23(2):155–165, 2004.

[29] S. Knoop, S. Vacek, R. Zollner, C. Au, and R. Dillmann. A CORBA-based distributed software architecture for control of service robots. *Intelligent Robots and Systems., Proceedings. IEEE/RSJ International Conference on*, 4, 2004.

[30] M.S. Loffler, V. Chitrakaran, and D.M. Dawson. Design and Implementation of the Robotic Platform. *Intelligent and Robotic Systems., Journal of*, 39(1):105–129, 2004.

[31] Leonardo Serra Mattos. The Evbot II: An Enhanced Evolutionary Robotics Platform Equipped with Integrated Sensing for Control. Master's thesis, North Carolina State University, 2002.

[32] P. Nebot and E. Cervera. A framework for the development of cooperative robotic applications. *Advanced Robotics., Proceedings, 12th International Conference on*, pages 901–906, 2005.

[33] A. Orebäck and H.I. Christensen. Evaluation of Architectures for Mobile Robotics. *Autonomous Robots*, 14(1):33–49, 2003.

[34] G. Steinbauer, G. Fraser, A. Mühlenfeld, and F. Wotawa. A Modular Architecture for a Multi-Purpose Mobile Robot. *Innovations in applied artificial intelligence., Proceedings of the 17th international conference on*, pages 1007–1015, 2004.

[35] Dan Tynan. The 25 Worst Tech Products of All Time. *PC World*, May 2006.

[36] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar. Miro-middleware for mobile robot applications. *Robotics and Automation, IEEE Transactions on*, 18(4):493–497, 2002.

[37] RT Vaughan, BP Gerkey, and A. Howard. On device abstractions for portable, reusable robot code. *Intelligent Robots and Systems., Proceedings. IEEE/RSJ International Conference on*, 3, 2003.

[38] Alan F.T. Winfield. Linux: an Embedded Operating System for Mobile Robots. In *Developing Embedded Real-Time Systems., IEE Embedded and Real-time Systems Professional Network colloquiumon*. IEE, May 2003.

[39] E. Woo, BA MacDonald, F. Trepanier, E.D.S.N.Z. Ltd, and N.Z. Auckland. Distributed mobile robot application infrastructure. *Intelligent Robots and Systems., Proceedings. IEEE/RSJ International Conference on*, 2, 2003.

APPENDICES

Appendix A

OpenEmbedded and the EvBot Operating System

The following is a brief guide to using OpenEmbedded, with all details provided referring to the particular setup and repository provided with the media for this document. Full details for using OpenEmbedded, including the required packages and setup for a build machine, can be found at *http://www.openembedded.org*. Additional scripts, setup, and getting the filesystem onto appropriate media for use on the EvBot platform is also described.

A.1 Introduction to OpenEmbedded

OpenEmbedded is a complete build framework for embedded Linux, supporting many different architectures, thousands of packages (with updates, changes, and additions being worked on all the time), and allows for near infinite customization. However, though any

level of complexity is possible, a fully functional build system and customized distribution can be achieved with only a few short steps.

1. Get the prerequisite packages installed on the build system.
2. Download BitBake.
3. Checkout the SVN repository of OpenEmbedded.
4. Add any local or distribution customizations.
5. Create an appropriate configuration file.
6. Build a boot image.

Specific instructions (and possibly updated versions of tools) for the distribution of Linux used on the build system can be found on the OpenEmbedded website. OpenEmbedded was used, for the research in this thesis, on installations of Fedora 12 and RedHat Enterprise Linux (RHEL) 5. It is recommended to follow the instructions provided on the website for your distribution to install the prerequisite packages.

The version of BitBake and a repository of OpenEmbedded used for this research presented are provided with the media of this document.

A.2 File Structure Overview

OpenEmbedded can be setup almost anywhere in a file system as long as there are not spaces in any directory of the path. The example provided here is installed in `/home/OE`. It is important to note that a full build can require 20GB of space or more for a single

platform, so wherever is chosen must have adequate space available. For all subsequent examples, the following paths and directories are assumed.

```
/home/OE/
      `-|-- bitbake
        |-- build
        |-- local
        |-- openembedded
        `-- sources
```

The `bitbake` directory contains the BitBake package, which does not need be installed on the system. The `build` directory is where all the compilation, staging, and cross-compiling tools are created. The `local` directory is where additional packages and local distribution customizations are specified. Finally, the `openembedded` and `sources` directories contain a copy of the OpenEmbedded repository and all the sources packages used to build the distribution, respectively.

A copy of this directory structure with the contents of all the directories can be found on the included media. Please refer to appendix C for more details.

A.3 Distribution Configuration

OpenEmbedded provides for a local overlay structure where all user configuration files are located, taking precedence over any main OpenEmbedded configurations. This local overlay structure is provided in `/home/OE/local` and mirrors some of the structure found in `/home/OE/openembedded`. Only those files that should be used in place of the ones found in the main OpenEmbedded source are required.

The two main files of importance are `/home/OE/local/site.conf` and `/home/OE/local/local.conf`. The site.conf configuration file is what allows the local overlay to work, adding paths for local configuration files. The local.conf is where the distribution is configured, specifying the target platform and build environment. Both these files are used by the BitBake tool described in section A.4.

A.4 Using BitBake

BitBake is the tool that makes all of OpenEmbedded possible. The repository of OpenEmbedded is just a large collection of configuration files and recipes for building software which BitBake can read to run all the necessary commands required to build the distribution.

A.4.1 Environment Setup

To use BitBake, the environment must be setup to be able to find the bitbake tools and the OpenEmbedded configuration files. This requires setting the to environment variables `PATH`, and `BBPATH`. The following lines can be added either to your starting shell scripts or must be entered on the command line before beginning to use OpenEmbedded.

For bash shell users:

```
export PATH=/home/OE/bitbake/bin:$PATH
export BBPATH=/home/OE/local:/home/OE/build:/home/OE/openembedded
```

For tcsh shell users:

```
setenv PATH /home/OE/bitbake/bin:$PATH
setenv BBPATH /home/OE/local:/home/OE/build:/home/OE/openembedded
```

A.4.2 BitBake Commands

All BitBake commands should be executed in the `build` directory, as all OpenEmbedded files will be created under the current working directory at the time of execution.

The use of BitBake is `bitbake recipe`, where recipe is either a package or image recipe. This will scan the configuration files, determine any prerequisites, and build everything required for the recipe. In the case of image recipes, this includes all of the build toolchain, staging directories and files, all target packages, and finally, the image root directory and requested filesystem images. See Table A.1 for some useful BitBake options.

The full manual for BitBake can be found at *http://bitbake.berlios.de/manual/*.

A.5 After Successful Compilation

After a successful compilation run of OpenEmbedded, there will be a tar file under the `/home/OE/build/try/deploy` directory of the roof file system. The following steps describe how to take the roof file system and put it on a CompactFlash or SHDC card and

Table A.1: Useful BitBake Options

Option	Meaning
-k	Continue as much as possible after an error.
-c	A specific task to execute (configure, build, package, etc.)
-b	Specify a particular recipe file.
-D	Output debugging information.
-h	List basic BitBake help and usage.

make it bootable. The following steps were performed on a Fedora 12 Linux installation, with the CompactFlash card showing up as */dev/sdc*.

1. Create the filesystem. The contents of the "part_commands.txt" file (see Table A.2) is a list of inputs that would normally be part of an interactive experience with fdisk. The file deletes all primary partitions (if they exist), creates a 10MB boot partition for GRUB, and makes one big partition with the rest of the drive.

   ```
   fdisk /dev/sdc < part_commands.txt
   mkfs.ext2 /dev/sdc1
   mkfs.ext2 /dev/sdc2
   sync
   ```

2. Install GRUB on the first partition.

   This is valid for an x86 processor platform only.

   ```
   mkdir /mnt/tmp
   mount /dev/sdc1 /mnt/tmp
   mkdir /mnt/tmp/grub
   ```

Table A.2: Partition Commands

```
d
1
d
2
d
3
d
4
d
n
p
1
1
+10MB
n
p
2


w
```

```
cp /usr/share/grub/i386-redhat/* /mnt/tmp/grub
grub-install /dev/sdc
sync
umount /mnt/tmp
```

3. Copy the root filesystem to the second partition.

```
mount /dev/sdc2 /mnt/tmp
cd /mnt/tmp
tar xvf FILENAME\_TO\_ROOTFS\_TAR\_FILE\_HERE
sync
umount /mnt/tmp
```

4. Eject CompactFlash, install in EvBot, and boot.

Appendix B

EAL Programmer's Guide

This guide is intended to be a useful tool to programmers who are using the EvBot Abstraction Layer, either providing plugins or using the user interface in their code. For either use, the discussion and detail regarding EAL data types should be referenced, as an understanding of that is crucial. Detailed descriptions of the EAL and EALPlugin APIs are provided, along with notes pointing out any special cases or things of which to be aware.

B.1 EAL Data Types

B.1.1 `EALDataInfo_t`

The EAL framework does not place limits on the type of data that can be passed between resources and the end user. However, to accomodate the near infinite possibilities, it does define the method by which this data passes through the EAL framework. The primary way this is achieved is encapsulating everything within a `EALDataInfo_t` structure.

The EALDataInfo_t structure has the following fields: type, length, data, isPtr, isUnsigned, isLong, and isShort.

Table B.1: EALDataInfo_t fields

EALDataType_t	type
unsigned int	length
void *	data
bool	isPtr
bool	isUnsigned
bool	isShort
bool	isLong

The type field defines the data type of the payload data, being one of the standard types of int, char, float, double, bool, or void. These are specified using the corresponding enumerated values from EALDataType_t, INT, CHAR, FLOAT, DOUBLE, BOOL, and VOID.

The length field defines the length of the data, in total number of bytes. This field is particularly important to have correct, or the full data may not be copied using certain commands (especially true for networked use of EAL).

The data is a pointer to the starting address of the data, with a total known length of length size bytes.

The boolean modifiers are provided for the user to specify any base type modifiers, recording whether the data is a pointer, unsigned, long, or short. For some base types,

such as `char`, not all the fields will apply (it is not possible to specify a `long char`). These fields default to false if they are not specified at the creation of the `EALDataInfo_t` structure instance.

B.1.2 Constructor Function

To make it easier to work with `EALDataInfo_t` structures, a constructor function is provided. The constructor is defined as:

```
EALDataInfo_t( EALDataType_t type_ = INT,
               void *data_ = 0,
               unsigned int length_ = 0,
               bool isPtr_ = false,
               bool isUnsigned_ = false,
               bool isShort_ = false,
               bool isLong_ = false );
```

To avoid scope issues with the compiler, the passed value names mirror the data fields of the `EALDataInfo_t`. As can be seen, all parameters have default values, making use of the constructor simpler.

It should be noted that the values `ISPTR`, `ISUNSIGNED`, `ISLONG`, and `ISSHORT` are defined as boolean `true`, for use with the constructor function for greater code clarity.

B.1.2.1 Examples for Base Types

For base types that are not a pointer, the length does not have to be specified, as can be seen in the examples here for creating `EALDataInfo_t` structures for an integer and a character.

```
EALDataInfo_t *intData = new EALDataInfo_t(INT, new int);
*(intData->data) = 42;

EALDataInfo_t *charData = new EALDataInfo_t(CHAR, new char);
*(intData->data) = 's';
```

B.1.2.2 Examples for More Complicated Data

Creation of `EALDataInfo_t` structures for more involved data, such as strings or arrays of integers is a little more complicated. In these cases, the length of the data must be specified, as well as the modifier `isPtr`.

The following examples shows the creation of an array of integers as well as a string (character array).

```
// defining an integer array and setting initial values
EALDataInfo_t *intArrayData = new EALDataInfo_t(INT, new int[10],
                                                sizeof(int)*10, ISPTR);
for(int i=0; i<10; i++) {
  *(intArrayData->data)[i] = i;
}


// defining a character array, value set to "hello"
EALDataInfo_t *charArrayData = new EALDataInfo_t(CHAR, new char[10],
                                                 sizeof(char)*10, ISPTR);
strcpy(*(charArrayData->data), "hello");
```

Finally, an example for which all parameters of the constructor function are specified, a `long integer`. There are two ways in which this can be defined, actually, since it is not an array. Both are presented.

```
// specifying all parameters to the constructor function
EALDataInfo_t *sampleLongInteger =
               new EALDataInfo_t(INT, new long int, sizeof(long int),
                                 !ISPTR, !ISUNSIGNED, !ISSHORT, ISLONG);
*(sampleLongInteger->data) = 1000000;

// alternatively
EALDataInfo_t *sampleLongInteger =
               new EALDataInfo_t(INT, new long int, sizeof(long int));
sampleLongInteger->isLong = true;
*(sampleLongInteger->data) = 1000000;
```

B.1.3 EALData_t

The data construct used for data transfer in most of the EAL framework is `EALData_t`. This is defined as a pointer to a `EALDataInfo_t`, which is done to avoid the overhead of copying a large data structure through successive function calls.

B.2 EAL Programmer Reference

B.2.1 Connection Functions

B.2.1.1 EALconnect

`EAL* EALconnect( )` `EAL* EALconnect( const char* host, int port )`	
Parameters	**host** - The hostname to connect to EAL server. **port** - The port number to connect to EAL server.
Returns	A pointer to an `EAL` framework or NULL.

The function attempts to connect to an `EAL` instance, either one created locally or on accessed through an `EALClient` interface.

B.2.1.2 EALdisconnect

`void EALdisconnect( EAL* eal )`	
Parameter	**eal** - The `EAL` framework to disconnect.
Returns	None

The function disconnects (and in the local case, closes and cleans up) the passed `EAL` framework.

B.2.2 Resource Knowledge Methods

B.2.2.1 getResourceList

`list<string>* getResourceList( )`	
Parameters	None
Returns	A list of resources available.

This method can be used to determine what resources are currently available.

B.2.2.2 findResource

`EALResourceId_t findResource( const char* resourceName )`	
Parameters	**resourceName** - The name of the requested resource.
Returns	Resource id of the requested resource if it exists, -1 if it was not found or there was an error.

This method can be used to get the id of a named resource, if it exists.

B.2.2.3 resourceInfo

`EALResourceInfo_t* resourceInfo( EALResourceId_t id )`	
Parameters	**id** - The id of the resource in question.
Returns	The resource info provided by the resource or NULL if there was an error.

This method can be used to get resource information from a specified resource, with a known id.

B.2.2.4 resourceInfobyName

`EALResourceInfo_t resourceInfobyName( const char* resourceName )`	
Parameters	**resourceName** - The name of the resource in question.
Returns	The resource info provided by the resource or NULL if there was an error.

This method can be used to get resource information from a named resource, if it exists. Lack of a response does not indicate that the resource does not exist. **findResource** should be used for that purpose.

B.2.2.5 getStatus

`int getStatus( EALResourceId_t id )`	
Parameters	**id** - The id of the resource in question.
Returns	The status value returned by the resource, -1 if the resource cannot be found.

This method is used to poll the resource status. The value and the meaning is dependent on the resource programming.

B.2.3 Resource Access

B.2.3.1 getValue

`EALData_t getValue( EALResourceId_t id )`	
Parameter	**id** - The id of the resource.
Returns	The data returned by the resource. NULL if the resource cannot be found.

This method can be used to request data from the resource. The data will be packed in the `EALData_t` structure.

B.2.3.2 setValue

`int setValue( EALResourceId_t id, EALData_t data )`	
Parameters	**id** - The id of the resource. **data** - The data to be 'set'.
Returns	The status value returned by the resource. -1 if the resource cannot be found.

This method can be used to pass data to a resource.

B.2.3.3 sendReq

`EALData_t sendReq( EALResourceId_t id, const char* req )`	
Parameters	**id** - The id of the resource. **req** - The specified request.
Returns	The data returned from the resource. NULL if the id cannot be found.

This method is used to send requests to a resource for data, used primarily for more complicated resources.

B.2.3.4 sendCmd

`int sendCmd( EALResourceId_t id, const char* cmd, EALData_t data )`	
Parameters	**id** - The id of the resource. **cmd** - The command to the resource. **data** - Additional data required for the command (can be NULL).
Returns	The status value returned by the resource. -1 if the resource cannot be found.

This method is used to send commands to the resource along with any necessary data. As with `sendReq`, this is primarily used with more complicated resources.

B.2.3.5 readComm

`int readComm( EALResourceId_t id, EALData_t data, unsigned int length )`	
Parameters	**id** - The id of the resource. **data** - The data location for read bytes. **length** - The number of units (bytes) to read.
Returns	The status value returned by the resource, the number of units (bytes) successfully read. -1 if the resource cannot be found.

This method is used for reading from a stream of data of a resource. The data read is returned in the `EALData_t` passed to the method and should be created prior to the call with sufficient memory allocation to hold the specified `length` of data.

B.2.3.6 writeComm

`int writeComm( EALResourceId_t id, EALData_t data, unsigned int length )`	
Parameters	**id** - The id of the resource. **data** - The data to be written. **length** - The number of units (bytes) to write.
Returns	The status value returned by the resource, the number of units (bytes) successfully written. -1 if the resource cannot be found.

This method is used to write to a stream of data of a resource. The data to be written is read from the passed `EALData_t` structure and should contain at least `length` amount of data to be written.

B.2.3.7 getConfig

`EALData_t getConfig( EALResourceId_t id, const char* config )`	
Parameters	**id** - The id of the resource. **config** - The configuration requested.
Returns	The configuration data requested. NULL if no valid resource is found.

This method is used to get configuration data from a resource.

B.2.3.8 setConfig

`int setConfig( EALResourceId_t id, const char* config, EALData_t data )`	
Parameters	**id** - The id of the resource. **config** - The configuration being set. **data** - Additional data for the command (can be NULL).
Returns	The status value returned by the resource. -1 if the resource cannot be found.

This method is used to set configuration data for the specified resource.

B.2.4 Direct Access

B.2.4.1 getResource

`EALResource* getResource( const char* resourceName )`	
Parameter	**resourceName** - The name of the requested resource.
Returns	A resource pointer to the requested resource if the resource can be found.

This method can only be used with a local connection to an EAL framework, as it provides direct access to an EALResource pointer. This method is provided for debugging of resources or extremely special situations where the normal EAL API adds too much overhead to meet timing requirements.

All resources accessed with this method must be returned with the `returnResource` method before calling `EALdisconnect`. It is possible to call this more than once on a single resource, but a matching number of `returnResource` calls must be made.

B.2.4.2 returnResource

`int returnResource( EALResource* *resource )`	
Parameter	**resource** - The address of a resource pointer.
Returns	1 on success -1 on failure

This method is used in conjunction with `getResource` to successfully return a previously requested EAL resource pointer. Note that the parameter returned is the **address** of the pointer variable used for the resource pointer.

B.3 EAL Plugin Programmer Reference

An EAL plugin is just a dynamicly loaded library of resources with two accessor functions specified. The two accessor functions are `connect` and `disconnect` and **must** be specified, or the EAL Plugin Manager will be unable to access the contents of the plugin.

B.3.1 Plugin Accessor Functions

The `extern "C"` is required for the `connect` and `disconnect` functions to be found by the dynamic library access calls.

B.3.1.1 connect

`extern "C" EALPlugin* connect( )`	
Parameter	None
Returns	An EAL plugin pointer with information about this plugin.

The `connect` function's sole purpose is to provide a common access point to an EAL plugin that the EAL Plugin Manager can use. The only thing this function should do is return a EALPlugin pointer, created with the C++ dynamic memory allocator `new`. See section B.3.2 for more details of `EALPlugin` and section B.3.3 for sample code and compiling instructions.

B.3.1.2 disconnect

`extern "C" void disconnect( EALPlugin* plugin )`	
Parameter	**plugin** - The plugin requested to be disconnected.
Returns	None

The disconnect function, called when the EAL Plugin Manager is finished with a plugin, cleans up the memory associated with the creation of the plugin in `connect`.

B.3.2 EAL Plugin Creation

`EALPlugin( const char* resourceName, ..., NULL )` `EALPlugin( int count, const char* resourceName, ... )`	
Parameters	**resourceName** - The name of a resource provided by the plugin. **count** - The specified number of resources provided.
Returns	When used in conjunction with `new`, returns an EALPlugin*

The EALPlugin constructor has one purpose, and that is to accept the names of the resources the plugin provides. Any number of resources can be compiled into the library of the plugin, but if their names are not known, the EAL Plugin Manager cannot provide access to them. The two forms of the contructor allow the user to specify the names of resources within a plugin, terminating the comma seperated list with a NULL entry, while the second requires the count of resources, followed by at least that number of resource names.

The first form allows the programmer to easily add resources to a plugin without carefully counting the number of resources. The second form is limiting, ensuring during debug stages that the full number of intended resources is specified. Any additional resource names provided over the number specified by `count` will be ignored, whether they are compiled into the plugin or not.

B.3.3 Sample Plugin Code and Compiling Instructions

B.3.3.1 Sample Code

```
// pluginSampleResource.cpp

#include <EALPlugin.h>

extern "C" EALPlugin* connect() {
  return new EALPlugin("SampleResource", NULL);

  // The alternate form is to specify the number
  // of resources without the final NULL parameter:
  //
  //  return new EALPlugin(1, "SampleResource");
  //
}


extern "C" void disconnect( EALPlugin* p ) {
  delete p;
}
```

The code shown provides a sample plugin file, which provides access to the resource `sampleResource`.

B.3.3.2 Compiling Instructions

```
g++ -W -fPIC -shared -Wl,-soname,SampleResource.plugin.1 \
    -o SampleResource.plugin.1 \
    pluginSampleResource.o EALResource.o SampleResource.o
```

For this example it is assumed that the compiler is *GNU g++* and all the required header, source, and object files can be found. Of importance are the `-fPIC`, `-shared`, and `-Wl,-soname,SampleResource.plugin.1`. The full compilation and assembly instruction is shown below.

Appendix C

Media Description

The directory structure for the attached media:

```
/
‘-+-- EAL
  |    ‘-+-- bin
  |      +-- eal
  |      +-- includes
  |      +-- Makefile
  |      +-- plugins
  |      +-- server
  |      ‘-- test
  +-- OE
  |     ‘-+-- bitbake
  |       +-- build
  |       +-- local
  |       +-- openembedded
  |       ‘-- sources
  ‘-- tarballs
```

The `EAL` directory contains all the source files for EAL and associated test programs, plugins, and Makefiles. The top Makefile can be used with "make all" or "make install" to compile everything.

The `OE` directory contains the OpenEmbedded build system along with the local overlay and all the downloaded sources needed to build the EvBotOS Linux distribution.

The `tarballs` directory contains tarred and gzipped copies of both the EAL and OE directories.

A copy of this document is provided in the root directory.