Partha Pratim Ray

Automated bug localization in embedded softwares

A new paradigm through holistic approaches

LAP LAMBERT Academic Publishing

Automated bug localization in embedded softwares is aimed at developing an understanding of the holistic concepts and techniques of embedded software debugging. This book discusses general concepts of embedded systems incorporating embedded systems as its core to be set free with bug localization. This book contains general concepts of software bugs along with their types and handling mechanisms. Debugging concepts are given very precisely. Discussion on embedded software characteristics, architecture and models of computation are the great features of this book. This text has choose Busybox as the test bed of practical implementations. This book organizes detailed discussions on the deployments of concepts and methodologies regarding memory error detection, invariant analysis and object state incorporated debugging. The organization of the text material differs greatly from the organization of a book; the main idea being the reader understand the concepts involved in embedded software debugging very easily. This text material will be useful to research grads as well as practicing engineers.

Partha Pratim Ray

Mr. Partha Pratim Ray is an Assistant Professor of Department of Computer Science at Surendra Institute of Engineering and Management, Siliguri. He received his M Tech in Electronics from West Bengal University of Technology, Kolkata. His areas of interest are Embedded System Design and Wireless Sensor Network. He is a member of IEEE.

978-3-8484-4439-7

Partha Pratim Ray

Automated bug localization in embedded softwares

A new paradigm through holistic approaches

LAP LAMBERT Academic Publishing

Impressum/Imprint (nur für Deutschland/only for Germany)
Bibliografische Information der Deutschen Nationalbibliothek: Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.
Alle in diesem Buch genannten Marken und Produktnamen unterliegen warenzeichen-, marken- oder patentrechtlichem Schutz bzw. sind Warenzeichen oder eingetragene Warenzeichen der jeweiligen Inhaber. Die Wiedergabe von Marken, Produktnamen, Gebrauchsnamen, Handelsnamen, Warenbezeichnungen u.s.w. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutzgesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Coverbild: www.ingimage.com

Verlag: LAP LAMBERT Academic Publishing GmbH & Co. KG
Heinrich-Böcking-Str. 6-8, 66121 Saarbrücken, Deutschland
Telefon +49 681 3720-310, Telefax +49 681 3720-3109
Email: info@lap-publishing.com

Approved by: Haldia, West Bengal University of Technology, Thesis, 2011

Herstellung in Deutschland:
Schaltungsdienst Lange o.H.G., Berlin
Books on Demand GmbH, Norderstedt
Reha GmbH, Saarbrücken
Amazon Distribution GmbH, Leipzig
ISBN: 978-3-8484-4439-7

Imprint (only for USA, GB)
Bibliographic information published by the Deutsche Nationalbibliothek: The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.
Any brand names and product names mentioned in this book are subject to trademark, brand or patent protection and are trademarks or registered trademarks of their respective holders. The use of brand names, product names, common names, trade names, product descriptions etc. even without a particular marking in this works is in no way to be construed to mean that such names may be regarded as unrestricted in respect of trademark and brand protection legislation and could thus be used by anyone.

Cover image: www.ingimage.com

Publisher: LAP LAMBERT Academic Publishing GmbH & Co. KG
Heinrich-Böcking-Str. 6-8, 66121 Saarbrücken, Germany
Phone +49 681 3720-310, Fax +49 681 3720-3109
Email: info@lap-publishing.com

Printed in the U.S.A.
Printed in the U.K. by (see last page)
ISBN: 978-3-8484-4439-7

Acknowledgements

I thank Dr. Ansuman Banerjee, Assistant Professor of the Advanced Computing and Microelectronics Computing Unit of Indian Statistical Institute, Kolkata. He was extremely helpful in providing the keen guidance and expertise I needed in order to complete this work, especially with regard to the debugging aspect of embedded softwares. He was never ceasing in his belief in me, always providing clear explanations when I was lost, constantly driving me with energy when I was tired, and always, giving me his time, in spite of anything else that was going on.

I would like to express my deep felt gratitude to my adviser, Mr. Banibrata Bag, Assis- tant Professor of the Electronics and Communication Engineering department at Haldia Institute of Technology, for his advice, encouragement, enduring patience and constant support.

I also wish to thank Professor Malay Kumar Pandit and Associate Professor Mr. Asim Kumar Jana of the Electronics and Communication Engineering department, both at the Haldia Institute of Technology. Their suggestions and comments were invaluable to the completion of this work. Dr. Malay Kumar Pandit graciously volunteered to act as my thesis coordinator.

Additionally, I want to thank the faculties and staff of Electronics and Communication Engineering department for all their help to complete my degree and prepare for a career as a enthusiastic science-seeker. This includes (but certainly is not limited to) the following individuals:

Dr. Sunandan Bhunia

He made it possible for me to have many wonderful experiences I enjoyed as a student, including the opportunity to get in touch with the external sources of knowledge base such as ISI, Kolkata. His believe and immense support on me and my work encouraged to move ahead in research work. I want to greet a very special thank to him.

Mr. Sourav Kumar Das

His constant help and faith in Embedded Systems Laboratory during last few months provided me great support in completing my work.

And finally, I must thank my dear friends, father and mother, well wishers and obviously Poulami Majumder, for putting up with me during the development of this work with continuing, loving support and no complaint. I do not have the words to express all my feelings here, only that I love you, all!

Place: Haldia

Date: 29th April, 2011

Partha Pratim Ray

Abstract

Debugging denotes the process of detecting root causes of unexpected observable behaviors in programs, such as a program crash, an unexpected output value being produced or an assertion violation. Debugging of program errors is a difficult task and often takes a significant amount of time in the software development life cycle. In the context of embedded software, the probability of bugs is quite high. Due to requirements of low code size and less resource consumption, embedded softwares typically do away with a lot of sanity checks during development time. This leads to high chance of errors being uncovered in the production code at run time.

Debugging embedded softwares is a common problem. Several techniques and tools do exist for solving this problem. However, various kinds of bugs remain untouched regarding unauthorized and inefficient memory related issues and few bugs may also reside inside the states of the software code. No feasible algorithm or technique has ever been developed that will solve all the particular cases of this problem accurately.

It turns out, though, that the width of the debugging metrics are quite important in a large number of the embedded systems having various buggy issues. This becomes readily apparent when we learn that the bugs typically come from inaccurate debugging technique.

Any measurement of a buggy metric is limited by the precision and accuracy of the debugging technique. To be of practical use, the debugging process must be reasonably accurate. This implies that, for most of the parts, the actual values associated with measurements lie within relatively narrow interval parameters or metrics such as memory, invariant and object state. Indeed, manufacturers often guarantee the error of their softwares to be very small.

Thus, we desire to look only at narrow interval parameters when considering the development of the methodology for debugging embedded softwares. As no such effective techniques exist that can solve such software problems, developing such methodologies seems indeed promising. Therefore, the goal of this thesis is to answer the following question:

Can any automated technique or methodology be developed for the general problem of debugging embedded softwares with narrow interval parameters?

We show here that this problem of debugging embedded softwares with narrow interval coefficients, is quite possible; thus, we do consider it possible to develop a feasible technique that will solve all particular cases of this problem automatically.

Table of Contents

Chapter 1

Introduction

Embedded softwares are the heart of embedded systems. The development of embedded softwares need to undergo the process of general software development cycle, resulting in code development and debugging as important activities. Embedded softwares typically do away with lot of sanity checks during development time. This leads to high chance of errors being uncovered in the production code at run time.

In this thesis we propose a methodology for debugging errors in embedded softwares. As a case study we have used BusyBox, de-facto standard for embedded Linux in embedded systems and a few pseudo codes resembling a few blocks of embedded BusyBox. Our first methodology works on top of Valgrind, a popular memory error detector. Our second work deals with invariant analysis of some parts of BusyBox by Daikon, an invariant analyzer. As our final contribution we developed a technique which incorporates object state information into class dependence graph (cldg) to detect bugs in object oriented programs for embedded softwares.

Problem Statement : Different kinds of debugging techniques are available for debugging embedded softwares. Specific types of errors that result from memory mishandling, unauthorized memory access etc. are very much crucial to embedded softwares in respect of bug intrusion, that must be detected well at the time of development. Bugs can penetrate into softwares through other ways also. But the existing approaches to detect the above said bugs, are unable to execute its job satisfactorily. Hence the need for developing new debugging methodologies aroused.

Organization : This thesis is organized as follows. Chapter 2 presents literature review regarding various aspects of bug, debug, embedded systems and embedded software and their features. Chapter 3 presents the related work on this matter. Chapter 4 presents the memory error detection. Chapter 5 presents invariant analysis. Chapter 6 presents object state incorporated debugging technique. Chapter 7 presents future work and concluding remarks.

Chapter 2

Literature Review

2.1 Software Bug

A software bug [40] is an error, flaw, mistake, undocumented feature that prevents it from behaving as intended (e.g. producing an incorrect result). Most bugs arise due to mistakesand errors made by the programmer, either in the program source code or its design, and a few are caused by compilers producing incorrect code. A program that contains a large number of bugs, and or bugs that seriously interfere with its functionality, is said to be buggy. Reports detailing bugs in a program are commonly known as bug reports, fault reports or problem reports.

2.1.1 Types of Bugs

1. Type I error [39], also known as an error of the first kind, an α error, or a false positive: the error of rejecting a null hypothesis when it is actually true. It occurs when we are observing a difference when in truth there is none, thus indicating a test of poor specifiction. Type I error can be viewed as the error of excessive credulity [46] or even hallucination.

2. Type II error [39], also known as an error of the second kind, a β error, or a false negative: the error of failing to reject a null hypothesis when in fact we should have rejected it. In other words, this is the error of failing to observe a difference when in truth there is one, thus indicating a test of poor sensitivity. Type II error can be viewed as the error of excessive skepticism [47] or myopia.

3. Here we present the bugs those can be branched from the two above bugs and their causes [40]:

    - Conceptual error, the causes are: code is syntactically correct, but the programmer or designer intended it to do something else.
    - Arithmetic bugs,the causes are: division by zero, arithmetic overflow or underflow, loss of arithmetic precision due to rounding or numerically unstable algorithms.
    - Logic bugs, the causes are: infinite loops and infinite recursion, off by one error, counting one too many or too few when looping.
    - Syntax bugs, the causes are: use of the wrong operator,
    - Resource bugs, the causes are: null pointer dereference, using an uninitialized variable, access violations, resource leaks; where a finite system resource such as memory or file handles are exhausted by repeated allocation without release, buffer overflow, in which a program tries to store data past the end of allocated storage, This may or may not lead to an access violation or storage violation.
    - Multi-threading programming bugs, the causes are: deadlock, race condition, concurrency errors in critical sections, mutual exclusions and other features of concurent processing etc.
    - Interfacing bug, the causes are: incorrect API usage, incorrect protocol implementation, incorrect hardware handling.
    - Teamworking bugs, the causes are: unpropagated updates; e.g. programmer changes "*myAdd*" but forgets to change "*mySubtract*", which uses the same algorithm.

2.1.2 Bug Handling

It [40] is common practice for software to be released with known bugs that are considered non-critical, that is, that do not affect most users' main experience with the product. While software products may, by definition, contain any number of unknown bugs, measurements during testing can provide an estimate of the number of likely bugs remaining; this becomes more reliable the longer a product is tested and developed ("if we had 200 bugs last version, we should have 100 this version").

It is often considered impossible to write completely bug-free software of any real complexity. So bugs are categorized by severity, and low-severity non-critical bugs are tolerated, as they do not affect the proper operation of the system for most users.

Like any other part of engineering management, bug management must be conducted carefully and intelligently because "what gets measured gets done" [38] and managing purely by bug counts can have unintended consequences.

2.2 Debugging

Debugging [41] is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware, thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another.

As software and electronic systems have become generally more complex, the various common debugging techniques have expanded with more methods to detect anomalies, assess impact, and schedule software patches or full updates to a system. The words anomaly and discrepancy can be used, as being more neutral terms, to avoid the words error and defect or bug where there might be an implication that all so-called errors, defects or bugs must be fixed [41].

Debugging ranges, in complexity, from fixing simple errors to perform lengthy and tiresome tasks of data collection, analysis, and scheduling updates. The debugging skill of the programmer can be a major factor in the ability to debug a problem, but the difficulty of software debugging varies greatly with the complexity of the system, and also depends, to some extent, on the programming language(s) used and the available tools, such as debuggers. Debuggers are software tools which enable the programmer to monitor the execution of a program, stop it, re-start it, set breakpoints, and change values in memory.

Generally [41], high-level programming languages, such as Java, make debugging easier, because they have features such as exception handling that make real sources of erratic behavior easier to spot. In programming languages such as C or assembly, bugs may cause silent problems such as memory corruption, and it is often difficult to see where the initial problem happened. In those cases, memory debugger tools may be needed.

2.3 Embedded System

An embedded system [24] is a computer system designed to perform one or a few dedicated functions [25] [26] often with real-time computing constraints. It is embedded as part of a complete device often including hardware and mechanical parts. By contrast, a general-purpose computer, such as a personal computer is designed to be flexible and to meet a wide range of end-user needs. Embedded systems control many devices in common use today [27]. A few characteristics of embedded systems are [24]:

- Embedded systems are designed to do some specific task, rather than be a general-purpose computer for multiple tasks.
- Embedded systems are not always standalone devices, it may consist of small, computerized parts within a larger device that serves a more general purpose.
- The program instructions written for embedded systems are referred to as firmware, and are stored in read-only memory or flash memory chips.

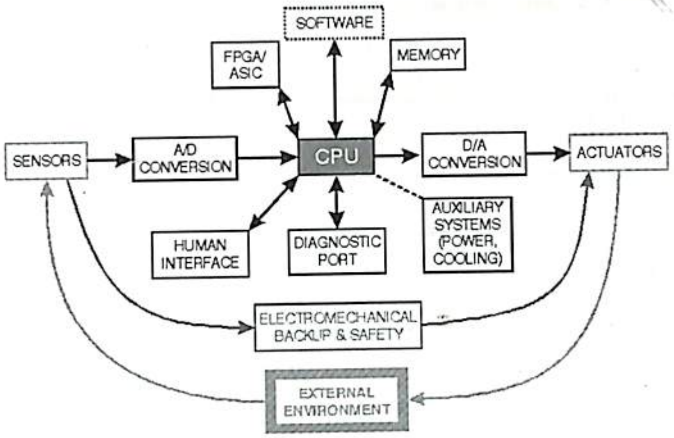

Figure 2.1: An embedded system encompasses the CPU as well as many other resources.

2.3.1 Embedded Processors

Embedded processors can be broken into two broad categories: ordinary microprocessors (μP) and microcontrollers (μC), which have many more peripherals on chip, reducing cost and size. There are Von Neumann as well as various degrees of Harvard architectures, RISC as well as non-RISC and VLIW types of processors are available in market. Few of these are: 65816, 65C02, 68HC08, 68HC11, 68k, 78K0R/78K0, 8051, ARM, AVR, AVR32, Blackfin, C167, Coldfire, COP8, Cortus APS3, eZ8, eZ80, FR-V, H8, HT48, M16C, M32C, MIPS, MSP430, PIC, PowerPC, R8C, RL78, SHARC, SPARC, ST6, SuperH, TLCS-47, TLCS-870, TLCS-900, Tricore, V850, x86, XE8000, Z80, AsAP etc.

2.3.2 Features of Embedded Systems

The figure 2.1 illustrates the peripheral environment of an embedded processor as well as a construction of a basic embedded system. Embedded systems are very much accustomed to various need to be guaranteed at the time of development. Few of them are illustrated below [23].

- **Real Time** :Real time system operation means that the correctness of a computation depends, in part, on the time at which it is delivered. In many cases the system design must take into account worst case performance.
- **Small size, low weight** :Many embedded computers are physically located within some larger artifact. Therefore, their form factor may be dictated by aesthetics, form factors existing in pre-electronic versions, or having to fit into interstices among mechanical components.
- **Safe and Reliable** :Some systems have obvious risks associated with failure. In mission-critical applications such as aircraft flight control, severe personal injury or equipment damage could result from a failure of the embedded computer.
- **Harsh environment** :Many embedded systems do not operate in a controlled environment. Excessive heat is often a problem, as in applications involving combustion.
- **Cost sensitivity** :Even though embedded computers have stringent requirements, cost is almost always an issue.
- **Less power requirement** :Low power management is crucial in this case.

2.4 Embedded Software

Embedded software is computer software which plays an integral role in the electronics it is supplied with. Embedded software's principal role is the interaction with the physical world. Embedded software is '*built in*' to the electronics in cars, telephones, audio equipment, robots, appliances, toys, security systems, pacemakers, televisions and digital watches, for example. One such open source example of embedded software is BusyBox [22] from Linux distribution.

Embedded Operating System : An embedded operating system is an operating system for embedded computer systems. These operating systems are designed to be compact, efficient, and reliable, forsaking many functions that non-embedded computer operating systems provide, and which may not be used by the specialized applications they run.

2.4.1 Characteristics of Embedded Software

An arrogant view of embedded software is that it is just software on small computers. This view is naive. Timeliness, concurrency, liveness, reactivity, and heterogeneity need to be an integral part of the programming abstractions. They are essential to the correctness of a program [37].

1. **Timeliness** :Time has been systematically removed from theories of computation. "Pure" computation does not take time, and has nothing to do with time.

2. **Concurrency** :Embedded systems rarely interact with only a single physical process. They must simultaneously react to stimulus from a network and from a variety of sensors, and at the same time, retain timely control over actuators. This implies that embedded software is concurrent.

3. **Interfaces** :Software engineering has experienced major improvements over the last decade or so through the widespread use of object oriented design which is a component technology, in the sense that a large complicated design is composed of pieces that expose interfaces that abstract their own complexity.

4. **Heterogeneity** :Heterogeneity is an intrinsic part of computation in embedded systems. It mixes computational styles and implementation technologies.

5. **Reactivity** :Reactive systems are those that react continuously to their environment at the speed of the environment. Harel and Pnueli [33] and Berry [34] contrast them with interactive systems, which react with the environment at their own speed, and

transformational systems, which simply take a body of input data and transform it into a body of output data.

2.4.2 Embedded Software Architecture

Embedded software contains some feature of important measure, such as [24]:

- Simple control loop :In this design, the software simply has a loop. The loop calls subroutines, each of which manages a part of the hardware or software.
- Interrupt controlled system :Some embedded systems are predominantly interrupt controlled. This means that tasks performed by the system are triggered by different kinds of events.
- Cooperative multitasking :A non-preemptive multitasking system is very similar to the simple control loop scheme, except that the loop is hidden in an API.
- Preemptive multitasking :In this type of system, a low-level piece of code switches between tasks or threads based on a timer (connected to an interrupt).
- Micro and Exo-kernel :A microkernel is a logical step up from a real-time OS. The usual arrangement is that the operating system kernel allocates memory and switches the CPU to different threads of execution. In general, microkernels succeed when the task switching and intertask communication is fast, and fail when they are slow. Exo-kernels communicate efficiently by normal subroutine calls.
- Monolithic kernels :In this case, a relatively large kernel with sophisticated capabilities is adapted to suit an embedded environment.
- Exotic custom operating systems :A small fraction of embedded systems require safe, timely, reliable or efficient behavior unobtainable with the one of the above architectures.

2.4.3 Models of Computation

There are many models of computation, each dealing with concurrency and time in different ways. Here we outline some of the most useful models for embedded software [37].

1. Dataflow : In dataflow models, actors are atomic (indivisible) computations that are triggered by the availability of input data. Connections between actors represent the flow of data from a producer actor to a consumer actor.

2. Time Triggered :Some systems with timed events are driven by clocks, which are signals with events that are repeated indefinitely with a fixed period.

3. Synchronous/Reactive :In the synchronous/reactive (SR) model of computation, connections between components represent data values that are aligned with global clock ticks, as with time-triggered approaches [37].

4. Rendezvous :In synchronous message passing, the components are processes, and processes communicate in atomic, instantaneous actions called rendezvous. If two processes are to communicate, and one reaches the point first at which it is ready to communicate, then it stalls until the other process is ready to communicate.

5. Finite State Machines :All [37] of the models of computation considered so far are concurrent. It is often useful to combine these concurrent models hierarchically with finite-state machines (FSMs) to get modal models. FSMs are different from any of the models we have considered so far in that they are strictly sequential. A component in this model is called a state or mode, and exactly one state is active at a time. The connections between states represent transitions, or transfer of control between states. Execution is a strictly ordered sequence of state transitions. Transition systems are a more general version, in that a given component may represent more than one system state (and there may be an infinite number of components).

2.5 BusyBox

In this text we have experimented on BusyBox [22], a de-facto standard for Linux in emebedded systems. BusyBox is a fairly comprehensive set of programs needed to run a Linux system. Moreover it is the de-facto standard for embedded Linux systems, providing many standard Linux utilities, but having small code size (size of executables), than the GNU Core Utilities. BusyBox provides compact replacements for many traditional full-blown utilities found on most desktop and embedded Linux distributions. Examples include the le utilities such as ls, cat, cp, dir, head, and tail. BusyBox also provides support for more complex operations, such as ifcong, netstat, route, and other network utilities. BusyBox is remarkably easy to configure, compile, and use, and it has the potential to significantly reduce the overall system resources required to support a wide collection of common Linux utilities. BusyBox in general case can be built on any architecture supported by gcc.

BusyBox is modular and highly configurable, and can be tailored to suit particular requirements. The package includes a configuration utility similar to that used to configure the Linux kernel. The commands in BusyBox are generally simpler implementations than their full-blown counterparts. In some cases, only a subset of the usual command line options is supported. In practice, however, the BusyBox subset of command functionality is more than sufficient for most general embedded requirements.

The BusyBox bundle functions as a single executable where the different utilities are actually passed on at the command line for separate invocation. It is not possible to build the individual utilities separately and run them stand alone. For example, for running the arp utility, we need to invoke BusyBox as busybox arp-Ainet and record the execution trace. Since we work on the binary level, the buggy implementation for us is the BusyBox binary, which has a large code base (about 121000 lines of code).

2.5.1 Overall View of BusyBox

BusyBox contains seveal standrad Linux utilities in it. Figure 2.2 shows that. Several Projects and products that do use BusyBox are given below in figures 2.3 and 2.4 respectively [22].

2.5.2 Usage

BusyBox is a multi-call binary [22]. A multi-call binary is an executable program that performs the same job as more than one utility program. That means there is just a single BusyBox binary, but that single binary acts like a large number of utilities. This allows BusyBox to be smaller since all the built-in utility programs (applets) can share code for many common operations. We can also invoke BusyBox by issuing a command as an argument on the command line. For example, entering

/bin/busybox ls

will also cause BusyBox to behave as 'ls'.

2.5.3 Bugs in BusyBox

KLEE [17] has reported some bugs in BusyBox 1.4.2. by a test generation method. Klee had checked all 279 BusyBox tools in series to demonstrate its applicability to bug finding. It has been seen that 21 bugs are present in BusyBox. We tried them on BusyBox version 1.4.2 and found (version 1.16.0) 6 of them persists namely, arp -Ainet, tr [, top d, printf %Lu, ls -co, install -m.

[, [[, acpid, addgroup, adduser, adjtimex, ar, arp, arping, ash, awk, basename, beep, blkid, brctl, bunzip2, bzcat, bzip2, cal, cat, catv, chat, chattr, chgrp, chmod, chown, chpasswd, chpst, chroot, chrt, chvt, cksum, clear, cmp, comm, cp, cpio, crond, crontab, cryptpw, cttyhack, cut, date, dc, dd, deallocvt, delgroup, deluser, depmod, devmem, df, dhcprelay, diff, dirname, dmesg, dnsd, dnsdomainname, dos2unix, du, dumpkmap, dumpleases, echo, ed, egrep, eject, env, envdir, envuidgid, ether-wake, expand, expr, fakeidentd, false, fbset, fbsplash, fdflush, fdformat, fdisk, fgrep, find, findfs, flash_eraseall, flash_lock, halt, hd, hdparm, head, hexdump, hostid, hostname, httpd, hush, hwclock, id, ifconfig, ifdown, ifenslave, ifplugd, ifup, inetd, init, inotifyd, logread, losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lspci, lsusb, lzmacat, lzop, lzopcat, makedevs, makemime, man, md5sum, mdev, patch, pgrep, pidof, ping, ping6, pipe_progress, pivot_root, pkill, popmaildir, poweroff, printenv, printf, ps, pscan, pwd, raidautorun, rdate, rdev, readahead, readlink, readprofile, realpath, reboot, reformime, renice, reset, resize, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run-parts, runlevel, runsv, runsvdir, rx, script, scriptreplay, sed, sendmail, seq, setarch, setconsole, setfont, setkeycodes, setlogcons, setsid, setuidgid, sh, sha1sum, sha256sum, sha512sum, showkey, slattach, sleep, softlimit, sort, split, ostart-stop-daemon, stat, strings, stty, su, sulogin, sum, sv, svlogd, swapoff, swapon, switch_root, sync, sysctl, syslogd, tac, tail, tar, taskset, tcpsvd, tee, telnet, telnetd, test, tftp, tftpd, time, uudecode, uuencode, vconfig, vi, vlock, volname, wall, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat, zcip, msh, mt, mv etc.

Figure 2.2: The commands incorporated in BusyBox version-1.16.0.

* buildroot--A configurable means for building users' own busybox/ uClibc based system systems, maintained by the uClibc developers
* OpenWrt--A Linux distribution for embedded devices, based on buildroot
* Tiny Core Linux--A very small minimal Linux GUI Desktop
* PTXdist--Another configurable means for building your own busybox based systems
* Debian installer (boot floppies) project
* Red Hat installer
* Slackware Installer
* Gentoo Linux install/boot CDs
* The Mandriva installer
* Linux Embedded Appliance Firewall--The sucessor of the Linux Router Project, supporting all sorts of embedded Linux gateways, routers, wireless routers, and firewalls
* Build Your Linux Disk
* AdTran - VPN/firewall VPN Linux Distribution
* mkCDrec - make CD-ROM recovery
* Partition Image
* Familiar Linux--A linux distribution for handheld computers
* Netstation
* GNU/Fiwix Operating System
* TimeSys real-time Linux
* MoviX--Boots from CD and automatically plays every video file on the CD
* Salvare--More Linux than tomsrtbt but less than Knoppix, aims to provide a useful workstation as well as a rescue disk and many more.

Figure 2.3: List of projects that use the BusyBox.

* Galaxy MGB4500 Raid Pro NAS
* StoryBox Ultimate uses BusyBox v1.1.3
* Western Digital's ShareSpace network attached storage device
* RD129 embedded board from ELPA
* EMTEC MovieCube R700 uses BusyBox 1.1.3
* The Kerbango Internet Radio
* LinuxMagic VPN Firewall
* Cyclades-TS and other Cyclades products
* Linksys WRT54G - Wireless-G Broadband Router
* Dell TrueMobile 1184
* NetGear WG602 wireless router with sources here
* ASUS WL-300g Wireless LAN Access Point with source here
* Belkin 54g Wireless DSL/Cable Gateway Router with source here
* Acronis PartitionExpert 2003
* U.S. Robotics Sureconnect 4-port ADSL router with source here
* ActionTec GT701-WG Wireless Gateway/DSL Modem with source here
* DLink--Model GSL-G604T, DSL-300T
* Siemens SE515 DSL router
* Free Remote Windows Terminal
* ZyXEL Routers etc.

Figure 2.4: List of products that use the BusyBox.