kboot – A Boot Loader Based on Kexec

Werner Almesberger
werner@almesberger.net

September 26, 2005

Abstract

Compared to the "consoles" found on traditional Unix workstations and mini-computers, the Linux boot process is feature-poor, and the addition of new functionality to boot loaders often results in massive code duplication. With the availability of kexec, this situation can be improved.

kboot is a proof-of-concept implementation of a Linux boot loader based on kexec. kboot uses a boot loader like LILO or GRUB to load a regular Linux kernel as its first stage. Then, the full capabilities of the kernel can be used to locate and to access the kernel to be booted, perform limited diagnostics and repair, etc.

1 Oh no, not another boot loader !

There is already no shortage of boot loaders for Linux, so why another one ? The motivation for writing kboot is simply that the boot process of Linux is still not as good as it could be, and that recent technological advances have made it comparably easy to do better.

Looking at traditional Unix servers and workstations, one often finds very powerful boot environments, offering a broad choice of possible sources for the kernel and other system files to load. It is also quite common to find various tools for hardware diagnosis and system software repair. On Linux, many boot loaders are much more limited than this.

Even boot loaders that provide several of these advanced features, like GRUB, suffer from the problem that they need to replicate functionality or at least code found elsewhere, which creates an ever increasing maintenance burden. Similarly, any drivers or protocols the boot loader incorporates, will have to be maintained in its context.

New boot loader functionality is not only required because administrators demand more powerful tools, but also because technological progress leads to more and more complex mechanisms for accessing storage and other devices, which a boot loader eventually should be able to support.

It is easy to see that a regular Linux system happens to support a superset of all the functionality described above.

With the addition of the kexec system call to the 2.6.13 mainline Linux kernel, we now have an instrument that allows us to build boot loaders with a fully featured Linux system, tailored according to needs and resources.

Kboot is a proof-of-concept implementation of such a boot loader. It demonstrates that new functionality can be merged from the vast code base available for Linux with great ease, and without incurring any significant maintenance overhead. This way, it can also serve as a platform for the development of new boot concepts.

The project's home page is at `http://kboot.sourceforge.net/`

The remainder of this section gives a high-level view of the role of a boot loader in general, and what kboot aims to accomplish. Additional technical details about the boot process, including tasks performed by the Linux kernel when bringing up user space, can be found in [1].

Section 2 briefly describes Eric Biederman's kexec [2], which plays a key role in the operation of kboot. Section 3 introduces kboot proper, explains its structure, and discusses its application. Section 4 gives an outlook on future work, and we conclude with section 5.

1.1 What a boot loader does

After being loaded by the system's firmware, a boot loader spends a few moments making itself comfortable on the system. This includes loading additional parts, moving itself to other memory regions, and establishing access to devices.

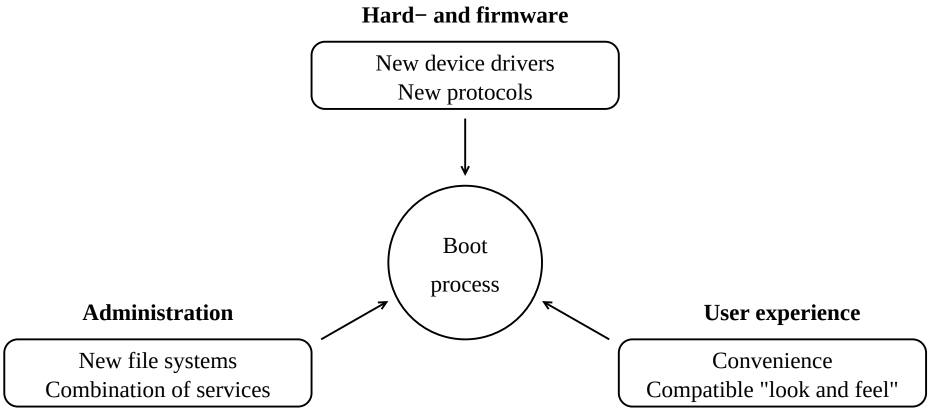

Figure 1: The boot process exists in a world full of changes and faces requirements from many directions. All this leads to the need to continuously grow in functionality.

After that, it typically tries to interact with the user. This interaction can range from checking whether the user is trying to get the boot loader's attention by pressing some key, through a command line or a simple full-screen menu, to a lavish graphical user interface.

Whatever the interface may be, in the end its main purpose is to allow the user to select, perhaps along with some other options, which operating system or kernel will be booted. Once this choice is made, the boot loader proceeds to load the corresponding data into memory, does some additional setup, e.g., to pass parameters to the operating system it is booting, and transfers control to the entry point of the code it has loaded.

In the case of Linux, two items deserve special mention: the boot parameter line and the initial RAM disk.

The boot parameter line was at its inception intended primarily as a means for passing a “boot into single user mode” flag to the kernel, but this got a little out of hand, and it is nowadays often used to pass dozens if not hundreds of bytes of essential configuration data to the kernel, such as the location of the root file system, instructions for how certain drivers should initialize themselves (e.g., whether it is safe for the IDE driver to try to use DMA or not), and the selection of items included in a generic kernel (e.g., disabling ACPI support).

Since a kernel would often not even boot without the correct set of boot parameters, a boot loader must store them in its configuration, and pass them to the kernel without requiring user action. At the same time, users should of course be able to manually set and override such parameters.

The initial RAM disk (initrd), which at the time of writing is gradually being replaced by the initial RAM file system (initramfs), provides an early user space, which is put into memory by the boot loader, and is thus available even before the kernel is fully capable to interact with its surroundings. This early user space is used for extended setup operations, such as the loading of driver modules.

Given that the use of initrd is an integral part of many Linux distributions, any general-purpose Linux boot loader must support this functionality.

1.2 What a boot loader should be like

A boot loader has much in common with the operating system it is loading: it shares the same hardware, exists in the same administrative context, and is seen by the same users. From all these directions originate requirements on the boot process, as illustrated in figure 1.

The boot loader has to be able to access at least the hardware that leads to the locations from which data has to be loaded. This does not only include physical resources, but also any protocols that are used to communicate with devices. Firmware sometimes provides a set of functions to perform such accesses, but new hardware or protocol extensions often require support that goes beyond this.

Above basic access mechanisms lies the domain of services the administrator can combine more or less freely. This begins with file system formats, and gets particularly interesting

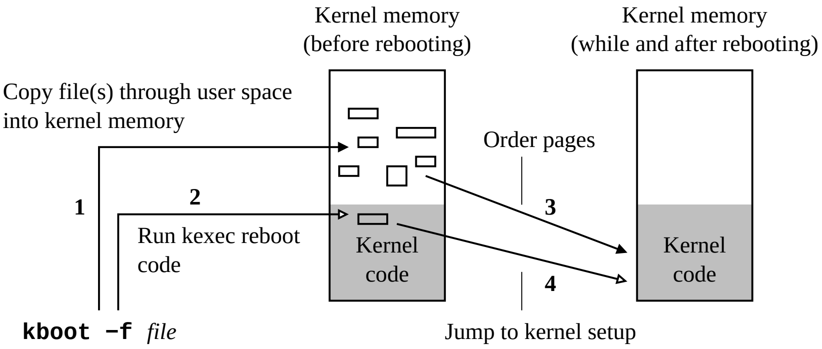

Figure 2: Simplified boot sequence of kexec.

when using networks. For example, there is nothing inherently wrong in wanting to boot kernels that happen to be stored in RPM files on an NFS server, which is reached through an IPsec link.

Last but not least, whenever users have to perform non-trivial tasks with the boot loader, they will prefer a context similar to what they are used to from normal interaction with the system. For instance, path names starting at the root of a file system hierarchy tend to be easier to remember than device-local names prefixed with a disk and partition number.

In addition to all this, it is often desirable if small repair work on an unbootable system can be done from the boot loader, without having to find or prepare a system recovery medium, or similar.

The bottom line is that a general-purpose boot loader will always grow in functionality along the lines of what the full operating system can support.

1.3 The story so far

The two principal boot loaders for Linux on the i386 platform, LILO and GRUB, illustrate this trend nicely.

LILO was designed with the goal in mind of being able to load kernels from any file system the kernel may support. Other functionality has been added over time, but growth has been limited by the author's choice of implementing the entire boot loader in assembler.[1]

GRUB appeared several years later and was written in C from the beginning, which helped it to absorb additional functionality more quickly. For instance, GRUB can directly read a large number of different file system formats, without having to rely on external help, such as the map file used by LILO. GRUB also offers limited networking support.

Unfortunately, GRUB still requires that any new functionality, be it drivers, file systems, file formats, network protocols, or anything else, is integrated into GRUB's own environment. This somewhat slows initial incorporation of new features, and, worse yet, leads to an increasing amount of code that has to be maintained in parallel with its counterpart in regular Linux.

In an ideal boot loader, the difference between the environment found on a regular Linux system and that in the boot loader would be reduced to a point where integration of new features, and their subsequent maintenance, is trivial. Furthermore, reducing the barrier for working on the boot loader should also encourage customization for specific environments, and more experimental uses.

The author has proposed the use of the Linux kernel as the main element of a boot loader in [1]. Since then, five years have passed, some of the technology has first changed, then matured, and with the integration of the key element required for all this into the mainstream kernel, work on this new kind of boot loader could start in earnest.

[1] LILO was written in 1992. At that time, 32-bit real mode of the i386 processor was not generally known, and the author therefore had to choose between programming in the 16-bit mode in which the i386 starts, or implementing a fully-featured 32-bit protected mode environment, complete with real-mode callbacks to invoke BIOS functions. After choosing the less intrusive of the two approaches, there was the problem that no suitable and reasonably widely deployed free C compiler was available. Hence the decision to write LILO in assembler.

2 Booting kernels with kexec

One prediction in [1] came true almost immediately, namely that major changes to the bootimg mechanism described there were quite probable: when Eric Biederman released kexec, it swiftly replaced bootimg, being technologically superior and also better maintained.

Unfortunately, adoption of kexec into the mainstream kernel took much longer than anyone expected, in part also because it underwent design changes to better support the very elegant kdump crash dump mechanism [3], and it was only with the 2.6.13 kernel that it was finally accepted.

2.1 Operation

This is a brief overview of the fundamental aspects of how kexec operates. More details can be found in [4], [5], and also [3].

As shown in figure 2, the user space tool `kexec` first loads the code of the new kernel plus any additional data, such as an initial RAM disk, into user space memory, and then invokes the `kexec_load` system call to copy it into kernel memory (1). During the loading, the user space tool can also add or omit data (e.g., setup code), and perform format conversions (e.g., when reading from an ELF file).

After that, a `reboot` system call is made to boot the new kernel (2). The reboot code tries to shut down all devices, such that they are in a defined and inactive state, from which they can be instantly reactivated after the reboot.

Since data pages containing the new kernel have been loaded to arbitrary physical locations and could not occupy the same space as the code of the old kernel before the reboot anyway, they have to be moved to their final destination (3).

Finally, the reboot code jumps to the entry point of the setup code of the new kernel. That kernel then goes through its initialization, brings up drivers, etc.

2.2 Debugging

The weak spot of kexec are the drivers: some drivers may simply ignore the request to shut down, others may be overzealous, and deactivate the device in question completely, and some may leave the device in a state from which it cannot be brought back to life, be this either because the state itself is incorrect or irrecoverable, or because the driver simply does not know how to resume from this specific state.

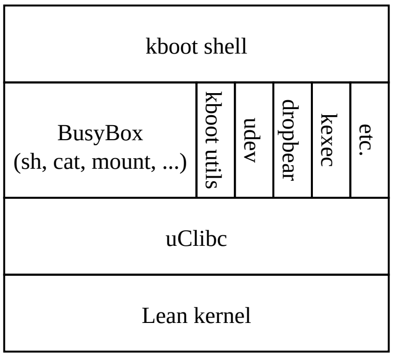

Figure 3: The software stack of the kboot environment.

Many of these problems have not become visible yet, because those drivers have not been subjected to this specific shutdown and reboot sequence so far.

The developers of kexec and kdump have made a great effort to make kexec work with a large set of hardware, but given the sheer number of drivers in the kernel and also in parallel trees, there are doubtlessly many more problems still awaiting discovery.

Since kboot is the first application of kexec that should attract interest from more than a relatively small group of developers, many of the expected driver conflicts will surface in the form of boot failures occurring under kboot.

3 Putting it all together

Kboot bundles the components needed for a boot loader, and provides the “glue” to hold them together. For this, it needs very little code: only roughly 3’000 lines, as of version 4. Already LILO exceeds this by one order of magnitude, and GRUB further doubles LILO’s figure.[2]

Of course, during its build process, kboot pulls in various large packages, among them the entire GCC tool chain, a C library, BusyBox, assorted other utilities, and the Linux kernel itself. In this regard, kboot resembles more a distribution like Gentoo or OpenEmbedded, which

[2] These numbers were obtained by quite unscientifically running `wc -l` on a somewhat arbitrary set of the files in the respective source trees.

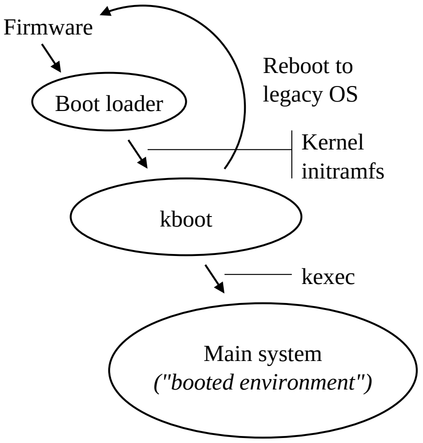

Figure 4: The boot sequence when using kboot.

consist mainly of meta-information about packages maintained by other parties.

3.1 The boot environment

Figure 3 shows the software packages that constitute the kboot environment. Its basis is a Linux kernel. This kernel only needs to support the devices, file systems, and protocols that will be used by kboot, and can therefore be considerably smaller than a fully-featured production kernel for the same machine.

In order to save space, kboot uses uClibc [6] instead of the much larger glibc. Unfortunately, properly supporting a library different from the one on the host system requires building a dedicated version of GCC. Since uClibc is sensitive to the compiler version, kboot also builds a local copy of GCC for the host. To be on the safe side, it also builds binutils.

After this tour de force, kboot builds the applications for its user space, which include BusyBox [7], udev [8], the kexec tools [2], and dropbear [9]. BusyBox provides a great many common programs, ranging from a Bourne shell, through system tools like “mount”, to a complete set of networking utilities, including “wget” and a DHCP client. Udev is responsible for the creation of device files in `/dev`. It is a user space replacement for the kernel-based devfs. The kexec tools provide the user space interface to kexec.

Last but not least, dropbear, an SSH server and client package, is included to demonstrate the flexibility afforded by this design. This also offers a simple remote access to the boot prompt, without the need to set up a serial console for just this purpose.

3.2 The boot sequence

The boot sequence, shown in figure 4, is as follows: first, the firmware loads and starts the first-stage boot loader. This would typically be a program like GRUB or LILO, but it could also be something more specialized, e.g., a loader for on-board Flash memory. This boot loader then immediately proceeds to load kboot's Linux kernel and kboot's initramfs.

The kernel goes through the usual initialization and then starts the kboot shell, which updates its configuration files (see section 3.5), may bring up networking, and then interacts with the user.

If the user chooses, either actively or through a timeout, to start a Linux system, kboot then uses kexec to load the kernel and maybe also an initial RAM disk.

Although not yet implemented at the time of writing, kboot will also be able to boot legacy operating systems. The plan is to initially avoid the quagmire of restoring the firmware environment to the point that the system can be booted from it, but to hand the boot request back to the first stage boot loader (e.g., with `lilo -R` or `grub-set-default`), and to reboot through the firmware.

3.3 The boot shell

At the time of writing, the boot shell is fairly simple. After initializing the boot environment, it offers a command line with editing, command and file name completion, and a history function for the current session.

The following types of items can be entered:

- Names of variables containing a command. These variables are usually defined in the kboot configuration file, but can also be set during a kboot session.[3] The variable is expanded, and the shell then processes the command. This is a slight generalization of the `label` in LILO, or the `title` in GRUB.

- The path to a file containing a bootable kernel. Path names are generalized in kboot, and also allow direct access to devices and some network resources. They

[3] In the latter case, they are lost when the session ends.

Syntax	**Example**	**Description**
variable	`my_kernel`	Command stored in a variable
/path	`/boot/bzImage-2.6.13.2`	Absolute path in booted environment
//path	`cat //etc/fstab`	Absolute path in kboot environment
path	`cd linux-2.6.14`	Relative path in current environment
device	`hda7`	Device containing a boot sector
`/dev/`*device*	`/dev/hda7`	Device file of device with boot sector
device`:`*/path*	`hda1:/bzImage`	File or directory on a device
device`:`*path*	`hda1:bzImage`	(implicit `/dev/`)
`/dev/`*device*`:`*/path*	`/dev/sda6:/foo/bar`	File or directory on a device
`/dev/`*device*`:`*path*	`/dev/sda6:foo/bar`	(explicit `/dev/`)
host`:`*/path*	`server:/home/k/bzImage-a`	File or directory on an NFS server
`http://`*host/path*	`http://server/foo`	File on an HTTP server
`ftp://`*host/path*	`ftp://server/foo/bar`	File on an FTP server

Table 1: Types of path names recognized by kboot.

are described in more detail in the next section. When such a path name is entered, kboot tries to boot the file through kexec.

- The name of a block device containing the boot sector of a legacy operating system, or the path to the corresponding device file.
- An internal command of the kboot shell. It currently supports `cd` and `pwd`, with the usual semantics.
- A shell command. The kboot shell performs path name substitution, and then runs the command. If the command uses an executable from the booted environment, it is run with chroot, since the shared libraries available in the kboot environment are almost certainly incompatible with the expectations of the executable.

With the exception of a few helper programs, like the command line editor, the kboot shell is implemented as a shell script.

3.4 Generalized path names

Kboot automatically mounts file systems of the booted environment, on explicitly specified block devices, and – if networking is enabled – also from NFS servers. Furthermore, it can copy and then boot files from HTTP and FTP servers.

For all this, it uses a generalized path name syntax that reflects the most common forms of specifying the respective resources. E.g., for NFS, the *host*`:`*path* syntax is used, for HTTP, it is a URL, and paths on the booted environment look just like normal Unix path names. Table 1 shows the various forms of path names.

Absolute paths in the kboot environment are an exception: they begin with two slashes instead of one.

We currently assume that there is one principal booted system environment, which defines the "normal" file system hierarchy on the machine in question. Support for systems with multiple booted environments is planned for future versions of kboot.

3.5 Configuration files

When kboot starts, it only has access to the configuration files stored in its initramfs. These were gathered at build time, either from the user (who placed them in kboot's `config/` directory), or from the current configuration of the build host.

This set of files includes kboot's own configuration `/etc/kboot.conf`, `/etc/fstab`, and `/etc/hosts`. The kboot build process also adds a file `/etc/kboot-features` containing settings needed for the initialization of the kboot shell.

Kboot can now either use these files, or it can, at the user's discretion, try to mount the file system containing the `/etc` directory of the booted environment, and obtain more recent copies of them.

The decision of whether kboot will use its own copies, or attempt an update first, is made at build time. It can be superseded at boot time by passing the kernel parameter `kboot=local`.

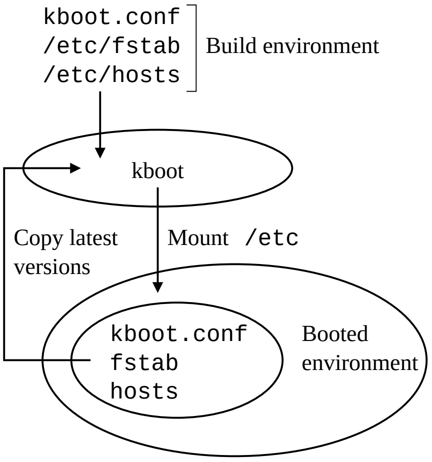

Figure 5: Configuration files used by kboot.

3.6 When not to use kboot

While kboot it designed to be a flexible and extensible solution, there are areas where this type of boot loader architecture does not fit.

If only very little persistent storage is available, which is a common situation in small embedded systems, or if large enough storage devices would be available, but cannot be made an integral part of the boot process, e.g., removable or unreliable media, only a boot loader optimized for tiny size may be suitable.

Similarly, if boot time is critical, the time spent loading and initializing an extra kernel may be too much. The boot time of regular desktop or server type machines already greatly exceeds the minimum boot time of a kernel, which embedded system developers aim to bring well below one second [10], so loading another kernel does not add significant overhead, particularly if the streamlining proposed below is applied.

Finally, the large hidden code base of kboot is unsuitable if high demands on system reliability, at least until the point when the kernel is loaded, require that the number of software components be kept to a minimum.

3.7 Extending kboot

The most important aspect of kboot is not the set of features it already offers, but that it makes it easy to add new ones.

New device drivers, low-level protocols (e.g., USB), file systems, network protocols, etc., are usually directly supported by the kernel, and need no or only little additional support from user space. So kboot can be brought up to date with the state of the art by a simple kernel upgrade.

Most of the basic system software runs out of the box on virtually all platforms supported by Linux, and particularly distributions for embedded systems provide patches that help with the occasional compatibility glitches. They also maintain compact alternatives to packages where size may be an issue.

Similarly, given that kboot basically provides a regular Linux user space, the addition of new ornaments and improvements to the user interface, which is an area with a continuous demand for development, should be easy.

When porting kboot to a new platform, the foremost – and also technically most demanding – issue is getting kexec to run. Once this is accomplished, interaction with the boot loader has to be adapted, if such interaction is needed. Finally, any administrative tools that are specific to this platform need to be added to the kboot environment.

4 Future work

At the time of writing, kboot is still a very young program, and has only been tested by a small number of people. As more user feedback arrives, new lines of development will open. This section gives an overview of currently planned activities and improvements.

4.1 Reducing kernel delays

The Linux kernel spends a fair amount of time looking for devices. In particular, IDE or SCSI bus scans can try the patience of the user, because they repeat similar scans already done by the firmware. The use of kboot now adds another round of the same.

A straightforward mechanism that should help to alleviate such delays would be to predict their outcome, and to stop the scan as soon as the list of discovered devices matches the prediction. Such a prediction could be made by kboot, based on information obtained from the kernel it is running under, and be passed as a boot parameter to be interpreted by the kernel being booted.

Once this is in place, one could also envision configuring such a prediction at the first stage boot loader, and passing it directly to the first kernel. This way, slow device scans that are known to always yield the same result could be completely avoided.

4.2 Using a real distribution

The extensibility of kboot can be further increased by replacing its build process, which is very similar to that of buildroot [11], with the use of a modular distribution with a large set of maintained packages. In particular OpenEmbedded [12] looks very promising.

The reasons for not reusing an existing build process already from the beginning were mainly that kboot needs tight control over the configuration process (to reuse kernel configuration, and to propagate information from there to other components) and package versions (in order to know what users will actually be building), the sometimes large set of prerequisites, and also problems encountered during trials.

4.3 Modular configuration

Adding new functionality to the kboot environment usually requires an extension of the build process and changes to the kboot shell. For common tasks, such as the addition of a new type of path names, it would be desirable to be able to just drop a small description file into the build system, which would then interface with the rest of kboot over a well-defined interface.

Regarding modules: at the time of writing, kboot does not support loadable kernel modules.

5 Conclusions

Kboot shows that a versatile boot loader can be built with relative little effort, if using a Linux kernel supporting kexec and a set of programs designed with the space constraints of embedded systems in mind.

By making it considerably easier to synchronize the boot process with regular Linux development, this kind of boot loader architecture should facilitate more timely support for new functionality, and encourage developers to explore new ideas whose implementation would have been considered too tedious or too arcane in the past.

References

[1] Almesberger, Werner. *Booting Linux: The History and the Future*, Proceedings of the Ottawa Linux Symposium 2000, July 2000. `http://www.almesberger.net/cv/papers/ols2k-9.ps`

[2] Biederman, Eric W. Kexec tools and patches. `http://www.xmission.com/~ebiederm/files/kexec/`

[3] Goyal, Vivek; Biederman, Eric W.; Nellitheertha, Hariprasad. *Kdump, A Kexec-based Kernel Crash Dumping Mechanism*, Proceedings of the Ottawa Linux Symposium 2005, vol. 1, pp. 169–180, July 2005. `http://www.linuxsymposium.org/2005/linuxsymposium_procv1.pdf`

[4] Pfiffer, Andy. *Reducing System Reboot Time with kexec*, April 2003. `http://www.osdl.org/archive/andyp/kexec/whitepaper/kexec.pdf`

[5] Nellitheertha, Hariprasad. *Reboot Linux Faster using kexec*, May 2004. `http://www-128.ibm.com/developerworks/linux/library/l-kexec.html`

[6] Andersen, Erik. *uClibc*. `http://www.uclibc.org/`

[7] Andersen, Erik. *BUSYBOX*. `http://busybox.net/`

[8] Kroah-Hartman, Greg; et al. *udev*. `http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html`

[9] Johnston, Matt. *Dropbear SSH server and client*. `http://matt.ucc.asn.au/dropbear/dropbear.html`

[10] CE Linux Forum. *BootupTimeResources*, CE Linux Public Wiki. `http://tree.celinuxforum.org/pubwiki/moin.cgi/BootupTimeResources`

[11] Andersen, Erik. *BUILDROOT*. `http://buildroot.uclibc.org/`

[12] *OpenEmbedded*. `http://oe.handhelds.org/`