The Components of Sioux Machine2World Embedded Linux

by Klaas van Gend

In this paper, the author presents the components required to build an embedded Linux system. In order to understand the requirements for the embedded Linux system, first its usage within a larger framework will be described. Then, the necessary packages are introduced, followed by some special considerations for embedded (Linux) systems.

1. Sioux Machine2World

Sioux is an innovative company that specialises in the software development for technical applications. Sioux was founded in 1996 and has grown to over 100 specialists, operating from Eindhoven, Rotterdam, Sittard and Antwerp. Sioux supports a broad spectrum of companies and research institutes in the areas of embedded software and software for technical/scientific systems. It is exactly this focus on technical software development that enables Sioux to bring high added value towards its customers.

One of our focus points is the Machine2World platform, a generic component-based platform for remote access, service and diagnostics of embedded systems. Machine2World consists of an Application server, currently based on the Microsoft .Net technology and several embedded 'Machines', which talk to the Application Server. The 'Machines' and the Application Server communicate using web services over a wide variety of protocols, like TCP, GPRS and infrared. Machines can be implemented on a variety of platforms, among which Linux.

One of our customers plans to use Machine2World to remotely monitor Vending Machines. For this, the new Machine board talks to the embedded control board of *existing* Vending Machines. It communicates observed data using GPRS (the packet data version of GSM) to the Application Server. A service engineer using the Application Server then can decide whether he needs to pay the machine a visit.

2. Machine Hardware: DIMM-PC

Because our customer expects to sell their remote-operation board as an option to vendors, the cost price of the hardware is important.

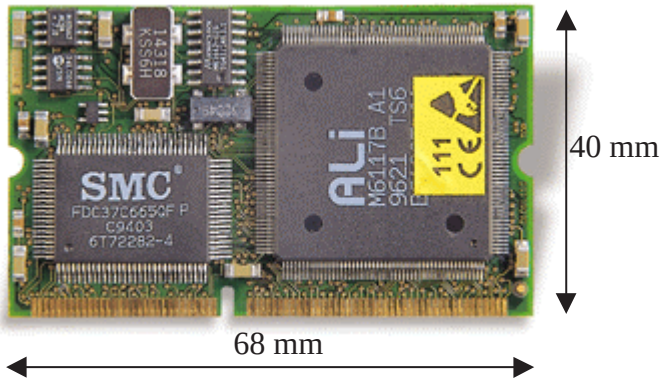

Figure 1 DIMM-PC

It is decided that the first release should be created using Commercial Off The Shelves (COTS) hardware. The Jumptec DIMM-PC platform was chosen. A DIMM-PC is a nearly complete x86-based system, the size of a DIMM module, approx. a credit card. Figure 1 shows a sample of the DIMM-PC. Through the DIMM connector, serial and parallel ports can be attached, or e.g. IDE disks. Also, the complete ISA bus is available on the DIMM connector for expansion, e.g. to add an Ethernet or VGA adapter.

The DIMM-PC of choice, the DIMM PC/386, features an i386 processor at 12-40 MHz, 4 MB of RAM and 8 MB of Flash IDE. This is a regular

flash chip, combined with a controller that maps the flash memory to an IDE interface. This way, the flash memory can be accessed like a hard disk. This is very convenient for developers, as no difficult boot strategies and/or flash file systems have to be written, just use the regular booting process. Jumptec provides DIMM-PCs up to Pentium-class at 133 MHz.

3. Machine Software: Linux

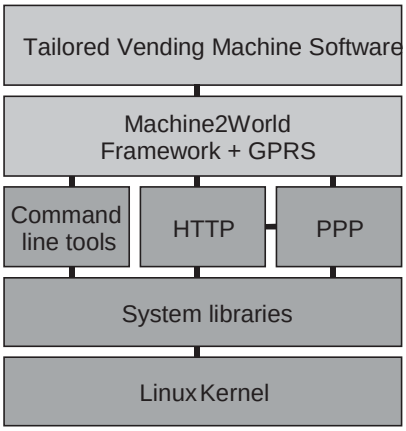

Figure 2 layering of the embedded system

Linux was chosen as operating system, because it is powerful, scaleable, well known and free. There are no real-time requirements, thus a regular Linux kernel was used. Because of the hardware limitations, however, several of the usual components of a GNU/Linux system had to be replaced by smaller variants, focussed on embedded systems.

The layering of the software is depicted in Figure 2. The dark shaded boxes are open source projects, the lightly shaded boxes are Sioux-specific development. These boxes are described below.

3.1 Linux Kernel

For this product, we chose to use the Linux 2.2 kernel series. Only the bare minimum was installed; among others console tty (the development system did contain a vga adapter), serial ports, IDE disks, the kernel-side of the PPP daemon, support for ram disks and math emulation.

Support for loadable kernel modules was also removed, as it won't be necessary to change the configuration of a running system. This resulted in a bootable zipped kernel image (called ***bzImage***) of approx. 410 kB size. The running kernel is 750 kB code and 400 kB of buffers and memory I/O space.

To boot the kernel, LILO is used because the kernel resides on a IDE disk.
Unfortunately, optimisation of the allocated kernel buffer structures (e.g. by limiting the maximum number of threads or limiting the number of ram disks) did not result in reduced memory consumption.

3.2 System Libraries: uClibc

Nearly all UNIX applications require a C library, which contains the implementation of functions like ***printf()***, ***malloc()*** or ***getpwent()***. Regular Linux servers and desktop systems use the GNU C library, which is a complete implementation for a very broad range of computer systems. Because none of the libc developers ever paid attention to less memory-privileged systems, the GNU libc is huge.

As a spin-off of the ì CLinux project, Erik Andersen started writing on a libc replacement. This evolved to the current uClibc, a libc implementation for embedded systems. Reducing the footprint of the libc was done by rewriting functions in an efficient way, removing backward compatibility to e.g. ancient versions of UNIX or AmigaOS. Another win was the removal of UNIX functions like ***wordexp()***, which is huge in code size, but isn't used today in any Linux app. uClibc isn't finished, but very useable. It comes with special compiler wrappers and it can be placed in parallel to your regular GNU compilers and libraries on a development system.

Several other large code pieces are configurable: if you don't need regular expressions in your C library, you win 27 kB.
Next to the C library, uClibc also provides a runtime linker to combine the application and several necessary shared libraries, the math library *libm*, the POSIX threading library *libpthread* and the password/authentication library *libcrypt*.

The uClibc libraries are more than fifteen times smaller then the current GNU libc family, in general total size is 200 kB or less.
There are some important pitfalls, though: using the gcc compiler, to reduce application size, default constructors are never called. This can be overridden by using the -uclibc-ctors option.

More information on uClibc can be obtained by following the references in the appendix.

3.3 Command line tools: BusyBox & TinyLogin

Another important part of GNU/Linux contains shells like ***bash*** and ***tcsh*** and commands like ***ls***, ***cut***, ***vi*** and ***sed***. Each command accepts loads of command line options. Implementing these all by themselves as GNU did, results in 200+ applications of 4 kB or more each. This is very inefficient for disk usage.

The BusyBox project aims to replace all regular system commands by a single executable. In the ***/bin*** directory, there is only a single executable called ***busybox***, and loads of soft links pointing from e.g. ***ls*** to the BusyBox executable. If a user types ***ls –al***, the BusyBox application receives through the C variable ***argv[0]*** which application to mimic. It then also knows how to interpret its arguments. BusyBox implements all commonly used options. Because all code is included in a single executable, lots of option-code was reused, resulting in an even smaller executable.

Next to implementing the commands, BusyBox also provides a shell, ***ash,*** the ***init*** process, a simplified ***vi*** editor and the ***syslogd*** daemon. There is no support for ***bash***, but ***ash*** is a good substitute for most of the scripts. Of course, it is fully configurable which commands should be kept out of the executable, thus reducing footprint even further.

TinyLogin works similar to BusyBox, but is focused on users, groups and passwords. It contains up to 31 commands like ***passwd***, ***addgroup***, ***login*** and ***getty***. TinyLogin is smaller than BusyBox, usually below 40 kB. More discussion on users, groups and passwords is presented in paragraph 4.2.

3.4 HTTP: Embedded web servers

The Machine is accessed via a Web Service. In essence, this is nothing more than a CGI interface, which accepts data, processes it and returns an answer. The SOAP protocol is used to define the data. The interface was defined in a WSDL file, which enables the ‘so called’ .Net development environment to generate proxy-classes for accessing the web service on the machine.

For connecting to the machine, the HTTP protocol was chosen, the machines are thus reachable over regular internet connections. This also means that a web server is required to route the HTTP request to the web service.

For this HTTP service, two embedded web servers were evaluated: THTTPD and Boa. THTTPD, the Tiny/Throttling HTTP Daemon, was developed as a small replacement for larger servers like Apache. It has all usual features like authentication, CGI and optionally SSL. The most outstanding feature is throttling, it is possible to limit data transfer per connection or per user. Boa is an even smaller web server, specially targeted for embedded systems. Unlike most other web servers, Boa doesn’t use multiple threads or processes to serve multiple clients at a time, but uses a smart usage of the ***select()*** function. This reduces kernel scheduling activities. In case of CGI, Boa does start additional threads, however.

Because only the CGI functionality was needed, in a later stadium of development we decided to implement the HTTP protocol by ourselves. This reduced the memory footprint by 120 kB, as compared to using Boa.

3.5 GPRS, PPP and SSL

For communication with the outside world, a connection has to be created. Because Ethernet connections are not available in e.g. station halls and stores, the wireless GPRS protocol was chosen. GPRS is a recent enhancement to the GSM standards. It allows for packet-oriented connections, which are not paid on a connection time basis, but per (mega) byte. A GPRS modem is connected over a serial line. In addition to the usual modem AT commands, the PIN-code for

the SIM card must be set to use this modem. We run a PPP connection over GPRS, using the PPP daemon by Paul Mackerras et al.

It was a requirement to have a kind of SSL to protect the data streams against potential crackers. Unfortunately, SSL implementations are heavyweights, because the SSL protocol is complex. It features several ciphers and multiple key exchange algorithms.

Tools like *stunnel* are based on *OpenSSL*, and are thus too large: stunnel is more than a megabyte (stripped!) on disk, and up to four megabytes running. ***Zebedee*** contains most functionality of stunnel, but supports only the Diffie-Hellman and Blowfish protocols. Additionally, Zebedee uses (b)zlib for datacompression. Zebedee takes less than 300 kB disk space. Implementations of Zebedee exist for Linux and Windows platforms, which at least suited our needs.

4. Special Considerations

The above mentioned packages are all optimised and limited to function in an embedded environment. Some of these limitations and optimisations deserve special attention.

4.1 Binary size reduction by compiler options and strip

In order to ease the run-time linking process and to improve the readability of the executables, the GNU compilers usually add loads of information to an executable. In general, all function names, variable names and such are all found in the binary, even if no debugging support is compiled in. This leads to an increase of the binary size by 30% for a regular C program, up to 400% for a serious C++ binary using templates and STL. Using the ***strip*** command, it is possible to remove all unnecessary information from the binary.

The GNU compilers have several other options that are of use to generate small executables. One of the more common-known compiler options are the optimise flags –O1 up to –O3. Less known is the –Os flag, which means 'optimise for size'.[1]

On processors with a limited number of registers like the Intel families, the –fomit-frame-pointers flag also reduces code size. Using this option, no register is used to pass frame pointers, which is used for some complex debugging tricks. Note that enabling this option doesn't always result in a smaller footprint. Experimentation is suggested. There are several more options that can lead to footprint reduction. It is advised to read the information pages and experiment.

4.2 Users, Groups and Authentication

Because embedded systems do not need all security features of large, multi-user UNIX systems, several shortcuts can be taken to reduce memory footprint and complexity.
In general, UNIX systems tend to store user accounts in ***/etc/passwd***, which is world-readable, but the actual encrypted passwords are either stored in a database or in a file like ***/etc/shadow***, accessible only by root. Several modern UNIX systems also feature PAM, Pluggable Authentication Modules, which allow for authentication via a Windows domain server or by smart cards. PAM is flexible, but large and time-consuming. Therefore, it is not implemented in uClibc and TinyLogin. Using a password database is not implemented. It is even possible to remove the need for the shadow file. This makes sense as there are no users on the system trying to decrypt the root password. There is no support for a Name Service Caching Daemon (***nscd***), because is expected that there are not many calls to authentication, nor that there are many users on the embedded system.

In fact, some embedded systems only define a root user – all applications run as root.
If the embedded system provides connections to

1 *For modern gigahertz processors, using the –Os flag may yield higher speeds. Using small executables, more of the executable can reside in the processor cache. If the processor runs out of the cache, retrieving non-cached instructions results in thousands of wasted processor cycles. Of course, this is not relevant for our 40MHz 386SX processor.*

the internet however, it still is a good idea to run all internet daemons either via ***chroot*** or at least, have their privileges dropped to 'nobody' level.

4.3 Disk management

Because the Jumptec board provides an IDE disk interface for the flash memory, a root file system is needed. Linux supports many file systems, but how to choose one? An embedded system has to prevent disk corruption due to unforeseen power-downs. Thus, a file system should either implement journaling, or should be mounted read-only. Paid by Ericsson, Red Hat developed the Journaling Flash File System for use in embedded systems, but JFFS uses regular flash memory to create a disk emulation. Because we already have a Flash IDE disk, JFFS is not an option.

If we want to use a file system which fully supports UNIX (i.e. atime, user/groups and soft and hard links), the choice is limited to file systems like MINIX and EXT2. The Linux kernel implementation of MINIX is 20 kb smaller than the EXT2 implementation. Another advantage of MINIX is that the fs tools ***fsck.minix*** and ***mkfs.minix*** are available within BusyBox. This turned out not to be a real advantage, as the size increase of Busybox is near the size of the ***fsck.ext2*** and ***mkfs.ext2*** executables, anyway.

4.4 System Init

If you have a single monolithic program, there is no need for an init procedure, just have the kernel start your program. For all more complicated cases, BusyBox implements the init process in a very straightforward manner. Init is always the first process to be started, PID 1. As in larger systems, init reads the */etc/inittab* file. There are some important differences, though.

By default, there are no run levels. Your embedded system either runs or not.
There is an option to start an initialisation script, which in turn may call other scripts. Such scripts are used to initialize the ram disk, fsck the log partition or setup networking options.

Another implemented feature is to start binaries in 'respawn' mode. If a respawned application exits or crashes, it will be restarted by the init process. This is very convenient for daemons. If one wants to, it is possible to mimic general UNIX initialisation (either BSD or Sys V) to a large degree, at the cost of more CPU cycles and more disk/RAM usage. The BusyBox implementation thus is flexible enough, and suited our needs perfectly.

5. Conclusions

Using freely available packages, it is possible to set up an embedded Linux system for general use. For this case, there were no timing constraints that required the use of real-time extensions to the Linux kernel.
Setting up an embedded system is not an easy task. There are lots of configuration choices per package, and many interactions between configuration options of different packages. It is a time-consuming, but rewarding task.

Help provided by FAQs, IRC and mailinglists proved very useful. Most packages are developed by seriously involved communities. If asked politely, all necessary support is available. In this case, Linux has proven to be scalable to fit in embedded systems.

6. Appendix

6.1 Packages and References

The following packages are discussed in this article:

Package	Licence	WWW
Linux kernel	GPL	http://www.kernel.org
uClibC	LGPL	http://www.uclibc.org
BusyBox	GPL	http://www.busybox.net
TinyLogin	GPL	http://www.busybox.net/tinylogin
PPPD	BSD or GPL (per file)	http://www.samba.org/ppp
Thttpd	BSD	http://www.acme.com/software/thttpd/
Boa	GPL	http://www.boa.org/
Zebedee	GPL	http://www.winton.org.uk/zebedee/

For more info on embedded Linux in general, the following book is recommended:
Craigh Hollabaugh: *Embedded Linux, Hardware, Software, and Interfacing*
Pearson Education. 2002, Indianapolis, ISBN 0-672-32226-9

For more information on the Machine2World platform or Sioux Technische Software Ontwikkeling B.V., please visit http://www.sioux.nl