2009 Eighth International Conference on Networks

Designing and Implementing an Embedded Linux for Limited Resource Devices

Hannu Lyytinen, Keijo Haataja, Pekka Toivanen
Department of Computer Science
University of Kuopio
P.O.Box 1627, FIN-70211 Kuopio, Finland
E-mail: hannu.lyytinen@cs.uku.fi, {keijo.haataja, pekka.toivanen}@uku.fi

***Abstract*—In this paper, the implementation details of an embedded Linux for limited resource devices are provided. Due to lack of an actual hardware to test the implemented embedded Linux, a simulator for development work had to be built. Thus, the implementation details of the simulator as well as an analysis of simulation results are described. Moreover, some new ideas that will be used in our future research work are proposed.**

***Index Terms*—Embedded Linux, Embedded System, Limited Resource Devices, Simulator, uClinux.**

I. Introduction

At first electronics found its way to mechanical devices. Many mechanically performed functions were easier to implement using their electronic counterparts. On the other hand, electronic circuits inherited also the disadvantages of mechanical devices: it was not straightforward to rewire the device in case a design flaw manifested itself. Moreover, it was difficult to add new features for devices.

The history of embedded systems spawned in 1960s from an idea to control the logics using a program stored in a memory. For example, telephone switchboard operators were replaced with the hardware that was controlled by a stored program [1]. Later on, when the history of microprocessors began, programs were the vital part of devices.

Embedded systems are widely used all over the world. For instance, they help pilots to keep airplanes in the air [2]. Moreover, the functions of entertainment electronics are controlled by microprocessors [3]. These examples are easy to perceive as embedded systems. On the other hand, Personal Digital Assistants (PDAs) obscure the thin line between embedded systems and traditional computers. Therefore, it is very difficult to give an exact definition of an embedded system.

According to Williams [1], embedded systems can be differentiated from traditional computers based on their purpose of use. On the other hand, embedded systems perform certain narrow tasks, while typical PCs are general purpose devices. Embedded systems can be seen as a way to build devices more easily. Moreover, embedded systems work in terms of their environment, while PCs are guiding the actions of their users. [2]

There does not exist any comprehensive or clear literature, i.e. books, web pages or research articles, on how to build your own embedded Linux operating system for limited resource devices. Therefore, the purpose of this paper is to introduce and clarify the designing process and implementation details of such a system.

The results of this paper: In this paper, the implementation details of both an embedded Linux for limited resource devices and a simulator for testing the operating system are provided. A simulation of the implemented operating system is also described as well as an analysis of the simulation results. Moreover, some new ideas that will be used in our future research work are proposed.

The rest of the paper is organized as follows. Section II explains the basics of embedded Linux operating systems and their software development processes. The implementation details of both the embedded Linux and the simulator are provided in Sect. III. A simulation on the implemented operating system as well as the analysis of the simulation results are provided in Sect. IV. Finally, Sect. V proposes some new ideas that will be used in our future research work and concludes the paper.

II. An Overview of Embedded Systems

Embedded systems can be thought of as special purpose computers. Typically they offer limited functionality compared to their counterpart, the general purpose desktop computer. Additionally, their environment imposes several requirements for the hardware. Some of these requirements are cost effectiveness and robustness against extreme temperatures, humidity, vibration and sometimes even radiation. On the other hand, the device itself shall not interfere with its environment, so electromagnetic compatibility must be addressed as well. Moreover, the embedded device might be required to run on battery power only. [2]

The ordinary desktop computers utilize operating systems to abstract the underlying hardware under standard interfaces. On embedded systems, the application software can comprise of a simple assembly language program. However, as the size and features of the embedded software grows, an operating system is often used to reduce the programmer's workload.

Many special purpose embedded operating systems exist. They try to address the difficulties presented by the embedded devices, namely limited processor and memory resources as well as the need of robustness and real-time applications. Sometimes the hardware lacks vital resources in which the lack of Memory Management Unit (MMU) is perhaps the most

978-0-7695-3552-4/09 $25.00 © 2009 IEEE
DOI 10.1109/ICN.2009.27

important. The common desktop operating systems utilize MMU in order to implement virtual memory.

The application development on these special purpose embedded operating systems is hindered by unfamiliar development conventions. These include application programming interfaces and development toolchain. On the other hand, general purpose operating systems lend themselves to easy application development due to well established development framework. Therefore, it is desirable to use as general purpose operating system as possible. This requires addressing of the problems introduced by the aforementioned special requirements for the embedded usage.

III. Designing and Implementing an Embedded Linux and a Simulator

After a thorough investigation, we decided that our embedded Linux will use Hitachi (Renesas) H8S/2239 microcontroller [4]. It contains 384 kB of Flash ROM, 32 kB of RAM and four Serial Communication Interface (SCI) ports. Due to lack of an actual hardware to test the implemented embedded Linux, a GNU Debugger (GDB) [5] based simulator for development work had to be built. Since the memory capacity of H8S/2239 microprocessor is clearly too low for an embedded Linux, we enlarged the memory capacity of our simulated hardware: ROM was extended up to 2 MB and RAM up to 8 MB.

Since the H8S/2239 microcontroller has no MMU, uClinux [6] was the only possible choice for our embedded Linux. Yoshinori Sato [7] has ported Linux into H8/300 architecture [8] and community members have contributed support for individual microcontroller models and platforms. However, there is no support for H8S/2239 microcontroller. Technical details of Sato's work can be found in [7].

In what follows, we describe the implementation details of our embedded Linux for limited resource devices.

We used *GNU Compiler Collection* (GCC) version 4.3.0 [9]. Moreover, we used *GNU binutils* [10] and *uClibc library* [11] which both were dated as 20.6.2004. GNU binutils was downloaded using Concurrent Versions System (CVS) [12], while uClibc was downloaded using Subversion system [13].

The H8S port of uClinux does not currently support Executable and Linking Format (ELF) binaries. An alternative to ELF is the FLAT file format, which is a lightweight object file format. GCC and binutils do not support this format directly, since they produce object files in ELF format by default. Therefore, we had to add *elf2flt* program [14] into toolchain by replacing the linker with a wrapper, which processes the resulting ELF executable with elf2flt. We used elf2flt program that was dated as 20.6.2004. It can be downloaded from the CVS server of uClinux project [14].

BusyBox version 1.10.2 [15] was used to provide de facto UNIX tools. *ROMfs* version 0.52 [16] was used as a file system that also requires *genromfs* program [16]. We built our simulator on the top of the GDB debugger version 6.1.1 [17]. Linux kernel version 2.6.16 [18] with the corresponding uClinux project patch file [19] was used in our embedded Linux. Moreover, few other patch files and a control file for the linker was required. We wrote a Bourne shell script that can be used to download all required aforementioned files. The script among the additional patch files can be downloaded at [20].

Minor patches to the crosscompiling toolchain and the target software are needed. The patches are publicly downloadable at the project's website [20]. In the following breakdown of the patches we refer to the files at the website, unless otherwise stated. The H8 port of GCC worked quite well, but certain structures of the source code triggered an internal compiler error. Earlier versions of GCC also have the same bug, but instead of crashing they will produce erroneous object code. The patch file *gcc-4.3.0.patch* contains a workaround for the bug.

The Linux kernel contains general framework for the H8/300 architecture, including the newer H8S. Several microcontrollers in this architecture are supported quite extensively. Unfortunately, our H8S/2239 is not among them. The closest fit is H8S/2678, which strictly speaking is of the H8SX architecture. The biggest difference between the architectures are few additional instructions, but the Linux kernel does not make use of these extended instructions. Nor does the GCC compiler unless explicitly instructed to do so.

The majority of no-MMU code has been merged into the mainstream Linux kernel branch already, but we still need the uClinux patch. The vanilla Linux kernel has a bug in the MTD device handling code, making the ROM chip mapping impossible. This is serious, since our root filesystem will reside on a ROM chip. The bug is fixed in the uClinux patch file *linux-2.6.16-uc0.patch.gz* [19].

The only modification needed outside the H8-specific tree is implementing the SCI port handling code in *drivers/serial/sh-sci.c* and *drivers/serial/sh-sci.h*. Based on the H8S/2678-specific code, the new code handles the H8S/2239, in which the memorymapped I/O registers differ in width and location. Additionally, the associated IRQ numbering is different. The rest of this minor port consist of defining the memory mapped I/O registers in a processor-specific header file, recognizing and fixing different I/O register widths and fixing the hard-coded timer IRQ number. These changes are incorporated in the patch file *linux-2.6.16-h8s2239.patch*.

As mentioned before, the root file system will reside on a ROM chip. Upon creating the filesystem image with *genromfs* [16], the raw binary object is converted into a linkable ELF object file using the toolchain linker. A special linker script for this purpose is contained in the file *ldscript*.

The target userland code does not compile and link cleanly on this target. The patch file *uclibc.patch* fixes two compile time errors in *uClibc*, whereas *BusyBox* linking issues are fixed in the patch *busybox-1.10.2.patch*. The BusyBox configuration files are supplied in the file *busybox-etc.tar*. The whole system is booted on a simulated hardware. The CPU simulation in the GDB debugger is quite slow due to slow instruction decoding algorithm. The patch in the file *gdb-6.1.1-speedup.patch* contains a faster algorithm, which enhances simulation performance dramatically. The I/O simulation, namely integrated timers and SCI ports, is based on Yoshinori Sato's early work on the H8 hardware simulation. Along with the possibility of

multiple SCI ports, a few fixes were required in order to get simulation working. The I/O patch is included in the file *gdb-6.1.1-io.patch*.

Compiling the toolchain is quite straightforward. Most software packages utilize *GNU Autoconf* [21] suite to extract system dependent compile time requirements. Therefore, the target machine can be specified to be different than the host machine simply by specifying *−−target=h8300-elf* option to *configure*. Another option worth mentioning is the installation prefix for the toolchain, which in this case is *−−prefix=/h8s*. However, the first step is to declare a few environment variables, as illustrated in Fig. 1.

```
export PATH=$PATH:/h8s/bin
export CROSS_COMPILER=h8300-elf-
export CROSS=h8300-
export ARCH=h8300
export LDEMULATION=h8300self
```

Fig. 1: Defined environment variables.

When compiling the *binutils* suite, it is essential to explicitly enable the Binary File Descriptor (BFD) library build. It is required later when compiling *elf2flt*. The steps required to compile and install *binutils* are illustrated in Fig. 2.

```
configure --prefix=/h8s --target=h8300-elf \
--enable-install-libbfd
make
make install
```

Fig. 2: Compiling the GNU binutils software package.

A full featured C compiler installation contains target system libraries and headers. These are not built yet because we will need a working C compiler in order to do so. To circumvent this dilemma, a temporary version of the compiler is built as illustrated in Fig. 3.

The option *−−with-build-sysroot* in Fig. 3 instructs the GCC compile process that the headers and the libraries are supposed to reside there, but they are not used in any way. This is in contrast with the semantics of the *−−with-sysroot* option used later, which enables the build process to utilize those headers and libraries when building a full featured compiler.

```
( cd gcc-4.3.0; patch -p1 < gcc-4.3.0.patch; )
configure --prefix=/h8s --target=h8300-elf \
patch -p1 < gcc-4.3.0.patch
configure --prefix=/h8s --target=h8300-elf \
--disable-libmudflap --disable-libssp \
--with-build-sysroot=/h8s/sysroot
make
make install
```

Fig. 3: Compiling a temporary version of GCC.

The rest of the options in Fig. 3 merely disable those features, which are not supported on the H8 architecture.

The C library needs header files that are created during the kernel configuration. The kernel is usually configured interactively using the *make menuconfig* command. An alternative way is to use the existing configuration file *kernelconfig* as Fig. 4 illustrates.

```
cp kernelconfig .config
patch -p1 < linux-2.6.16-h8s2239.patch
make prepare-all
```

Fig. 4: An example of kernel configuration.

Figure 4 also illustrates the applying of patches. The command *make prepare-all* creates the required header files for the C library.

Like the kernel, the uClibc library could be configured interactively, but the existing configuration file *uclibcconfig* can be used analogously when compiling the C library as Fig. 5 illustrates. After that, the C library is patched, compiled and installed.

```
cp uclibcconfig .config
patch -p1 < uclibc.patch
make
make install
```

Fig. 5: An example of the C library compilation.

After the C library has been compiled, the full featured version of GCC must be compiled as Fig. 6 illustrates. Moreover, a symbolic link is required in order to fix the lack of *limits.h* header file.

```
configure --prefix=/h8s --target=h8300-elf \
--disable-libmudflap --disable-libssp \
--with-sysroot=/h8s/sysroot
make
make install
cd /h8s/lib/gcc/h8300-elf/4.3.0/include
ln -s ../include-fixed/limits.h .
```

Fig. 6: The installation of the final version of GCC.

When the final version of GCC has been successfully installed, it is time to install *elf2flt* program (see Fig. 7).

Now everything is ready to compile and install the first application program, BusyBox, into our Embedded Linux as Fig. 8 illustrates.

BusyBox installation is now located at *_install* directory that includes the contents of embedded device's root file system. Initialization files of the system, such as *inittab*, can be added into file system by extracting *busybox-etc.tar* file. Moreover, we had to create two device nodes, *tty* and *console*, as Fig. 9 illustrates.

```
configure --prefix=/h8s --target=h8300-elf \
--with-libbfd=/h8s/i686-pc-linux-gnu/h8300-elf/lib/\
libbfd.a --with-libiberty=/h8s/lib/libiberty.a \
--with-bfd-includedir=/h8s/i686-pc-linux-gnu/h8300-elf/\
include --with-binutils-include-dir=/h8s/sources/src/\
include
make
make install
```

Fig. 7: The installation of elf2flt program.

```
cp busyboxconfig .config
patch -p1 < busybox-1.10.2.patch
make
make install
```

Fig. 8: The installation of BusyBox program.

At this point, the software of our embedded Linux is ready and it is time to compile the kernel. The root file system will be included into kernel as an object file. First, we installed *genromfs* program as Fig. 10 illustrates. Second, we created an image file of BusyBox *_install* directory by using *genromfs* program. Third, we converted the image file into ELF format by using the linker. It is worth noting that the controlling file of the linker, *ldscript*, is also required. Finally, we compiled the kernel.

When the kernel has been successfully compiled, *vmlinux* file will be located at the root of the directory tree. Normally, when developing on the actual hardware, *vmlinux* file is converted to a raw binary format more suited to burning to ROM. This can be done by using the command *make vmlinux.bin*. However, due to the use of simulator, we skipped this part.

The final phase before the actual simulation was the installation of the simulator (see Fig. 11). When the simulator has been successfully installed, our embedded Linux for limited resource devices is ready.

```
cd _install
tar xf busybox-etc.tar
mkdir dev
cd dev
mknod tty c 5 0
mknod console c 5 1
```

Fig. 9: Commands for creating device nodes tty and console.

```
make
cp genromfs /h8s/bin
genromfs -f rootfs.img -d /h8s/sources/busybox-1.10.2/\
_install
h8300-elf-ld -o romfs.o --oformat=elf32-h8300 \
--format=binary rootfs.img -T ldscript -mh8300self
make vmlinux
```

Fig. 10: Commands for compiling the kernel.

```
patch -p1 < gdb-6.1.1-iosim.patch
patch -p1 < gdb-6.1.1-speedup.patch
configure --prefix=/h8s --target=h8300-elf
make
make install
```

Fig. 11: Commands for installing the simulator.

IV. Simulation

The simulated SCI ports are represented by pseudo terminal (PTY) slave nodes on the host machine. A terminal program, such as *minicom*, is connected to the terminal node. First, we started the simulator and defined the hardware to be simulated as Fig. 12 illustrates. Second, the debugger printed out corresponding device nodes of the simulated serial ports. Finally, we started the terminal program by using the command *minicom -o -p /dev/pts/4*.

```
h8300-elf-gdb vmlinux
target sim --h8300s
SCI0 = /dev/pts/4
SCI1 = /dev/pts/5
SCI2 = /dev/pts/6
SCI3 = /dev/pts/7
```

Fig. 12: Commands for starting the simulator and defining the hardware to be simulated.

At this point, programs can be loaded into the memory of the simulator by using the command *load*. After loading a program, the simulator prints out sizes of the loaded segments and their corresponding positions in memory as Fig. 13 illustrates.

As Fig. 13 depicts, Linux kernel requires 859 kB of ROM, kernel-related data 41 kB of ROM, and the file system 435 kB of ROM. Therefore, the total amount of ROM required is 1335 kB. When the program has been loaded into memory, it can be executed by using the command *run* as Fig. 14 illustrates.

```
Loading section .vectors, size 0x1fc vma 0x0
Loading section .text, size 0xd6a40 vma 0x200
Loading section .rodata, size 0xa5c6 vma 0xd7000
Loading section __param, size 0xc8 vma 0xe2000
Loading section .data, size 0x14158 vma 0x400000
Loading section .romfs, size 0x6cc00 vma 0x420804
Start address 0x400
Transfer rate: 11600144 bits in <1 sec.
```

Fig. 13: An example output of the simulator.

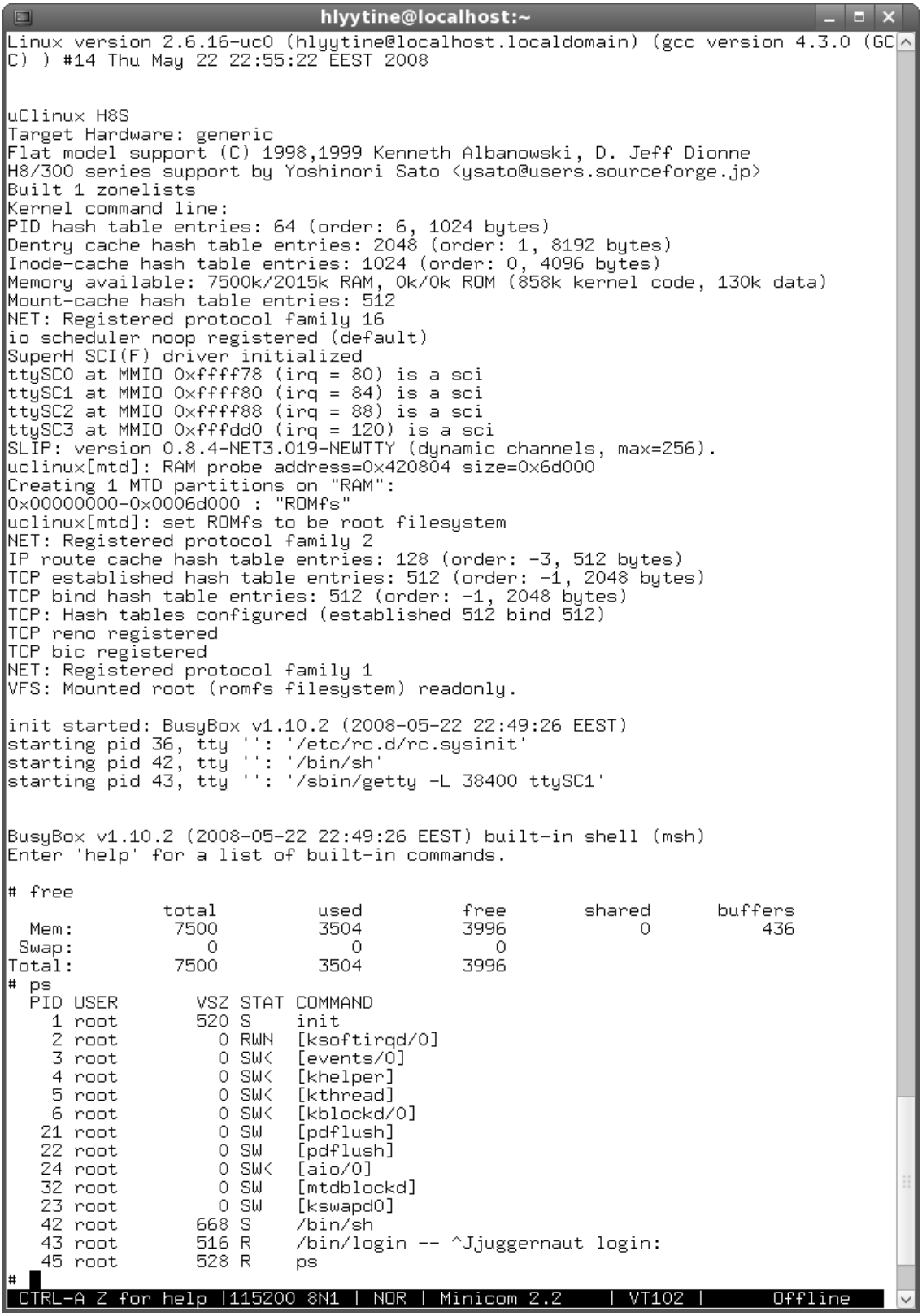

Fig. 14: Our embedded Linux in action.

As Fig. 14 depicts, the command *free* shows that approximately 4 MB of RAM has been used and another 4 MB of RAM is still available. Moreover, as the command *ps* shows, there are four user processes running on our embedded Linux consuming over 2 MB of RAM. Therefore, the embedded Linux kernel requires only 2 MB of RAM to work. In practice, *as little as 4 MB of RAM is enough for our embedded Linux with BusyBox* in case of a very limited resource device that still needs Linux and its versatile features.

V. Conclusions and Future Work

The designing process and implementation details of our embedded Linux for limited resource devices were provided. Due to lack of an actual hardware to test the implemented embedded Linux, a simulator for development work had to be implemented. Thus, a designing process and implementation details of the simulator as well as the brief analysis of the simulation results were described.

The simulation proved that our embedded Linux kernel requires only 2 MB of RAM to work. Thus, as little as 4 MB of RAM is enough for our embedded Linux with BusyBox in case of a limited resource device that still needs Linux and its versatile features.

In our future research work, we will investigate the current state of various architectures in order to find out the supported features of both the kernel and toolchains. In addition, we will investigate what are the biggest obstacles in embedded Linux usage in order to determine the worldwide commercial potential for embedded Linux products. Moreover, we will further investigate what is the minimum size of an embedded Linux that still is versatile and functional as presented in this paper. Shared libraries and directly ROM-executable code (XIP, Execute In Place) are definitely worth investigation.

We feel that embedded Linux operating systems are suitable for such devices, where simple microcontroller-based solutions are inadequate and PC-based solutions would be too heavy or cumbersome. Moreover, an embedded Linux is a very neat solution in prototypes, where new ideas are to be tested rapidly.

References

[1] J. Williams, "Embedding Linux in a Commercial Product: A Look at Embedded Systems and What It Takes to Build One", *Linux Journal*, vol. 1999, no. 66, Oct. 1999.

[2] A.S. Berger, *Embedded Systems Design – An Introduction to Processes, Tools and Techniques*, CMP Books, 2001.

[3] R.U. Rehman, C. Paul, *The Linux Development Platform: Configuring, Using, and Maintaining a Complete Programming Environment*, Prentice Hall, 2003.

[4] Renesas – H8S/2258, H8S/2239, H8S/2238, H8S/2237, H8S/2227 Groups Hardware Manual. Available: http://documentation.renesas.com/eng/products/mpumcu/rej09b0054_h8s2239.pdf. [Accessed Nov. 25, 2008].

[5] Free Software Foundation, GDB: The GNU Project Debugger. Available: http://www.gnu.org/software/gdb. [Accessed Nov. 25, 2008].

[6] uClinux – Embedded Linux/Microcontroller Project. Available: http://www.uclinux.org. [Accessed Nov. 25, 2008].

[7] Y. Sato, Porting to H8/300 Architecture. Linux H8/300 porting project. [Online]. Available: ftp://ftp.realtimelinuxfoundation.org/pub/events/rtlws-2003/proc/sato.pdf. [Accessed Nov. 25, 2008].

[8] Renesas Technology – H8 Family. Available: http://www.renesas.com/fmwk.jsp?cnt=h8_family_landing.jsp&fp=/products/mpumcu/h8_family. [Accessed Nov. 25, 2008].

[9] The GNU Operating System – GNU Compiler Collection (GCC) 4.3.0. Available: ftp://ftp.gnu.org/pub/gnu/gcc/gcc-4.3.0. [Accessed Nov. 25, 2008].

[10] The GNU Operating System – GNU binutils. Available: http://ftp.gnu.org/gnu/binutils. [Accessed Nov. 25, 2008].

[11] Erik Andersen, uClibc library. Available: http://uclibc.org. [Accessed Nov. 25, 2008].

[12] Free Software Foundation, CVS – Concurrent Versions System. Available: http://www.nongnu.org/cvs. [Accessed Nov. 25, 2008].

[13] CollabNet, Subversion – Open Source Version Control System. Available: http://subversion.tigris.org. [Accessed Nov. 25, 2008].

[14] The uClinux CVS Repository – elf2flt program. Available: http://cvs.uclinux.org. [Accessed Nov. 25, 2008].

[15] Erik Andersen, BusyBox 1.10.2. Available: http://busybox.net/downloads. [Accessed Nov. 25, 2008].

[16] SourceForge, Linux ROM filesystem – ROMfs 0.52 and genromfs program. Available: http://downloads.sourceforge.net/romfs/genromfs-0.5.2.tar.gz. [Accessed Nov. 25, 2008].

[17] The GNU Operating System – GDB debugger 6.1.1. Available: ftp://ftp.gnu.org/pub/gnu/gdb/gdb-6.1.1.tar.gz. [Accessed Nov. 25, 2008].

[18] CSC, Linux kernel 2.6.16. Available: ftp://ftp.funet.fi/pub/linux/kernel/v2.6/linux-2.6.16.tar.gz. [Accessed Nov. 25, 2008].

[19] uClinux – Embedded Linux/Microcontroller Project, Linux kernel 2.6.16 patch. Available: http://www.uclinux.org/pub/uClinux/uClinux-2.6.x/linux-2.6.16-uc0.patch.gz. [Accessed Nov. 25, 2008].

[20] H. Lyytinen, Linux on H8S/2239. Public Software Packages, University of Kuopio, Department of Computer Science. [Online]. Available: http://www.cs.uku.fi/tutkimus/GPL/gpl.shtml. [Accessed Nov. 25, 2008].

[21] The GNU Operating System – GNU Autoconf. Available: http://www.gnu.org/software/autoconf. [Accessed Nov. 25, 2008].