# Hacking Embedded Linux Based Home Appliances

Alexander Sirotkin
Metalink Broadband
demiurg@mtlk.com

August 30, 2007

**Abstract**

Embedded Linux is an interesting phenomenon. With millions of devices shipped to-day, from residential gateways to cellular phones, it may be Linux's biggest success story ever. However, it gets disproportionately small amount of interest from the open source community. Part of the blame is on the companies developing embedded Linux based products, which feel uncomfortable with the idea of opensource. However, even bigger problem can be the fact that embedded systems are perceived as black boxes which are very hard to hack, contrary to an "open" PC.

This article tries to break this stigma by showing Linux developers how to hack various Linux based devices that they might already have at home. If we understand that it's easy and fun to hack embedded systems, I believe embedded Linux issues will get more attention from opensource community, which will eventually result in more embedded-friendly Linux kernel, more embedded-oriented opensource projects and better Linux-based products.

## 1 Disclaimer

The views expressed in this paper are those of the author only, and should not be attributed to Metalink Broadband. No proprietary information that is not publicly available was used in this paper. Following procedures described below may void your warranty.

## 2 Introduction

Embedded appliances that surround you are generally perceived as "black boxes" which you can only use the way manufacturer designed them, contrary to an "open" PC which you can extend, modify and hack the way you want it, especially if that PC runs Linux. However, this perception is no longer true as many embedded systems manufacturers switched to much more standard hardware and Linux in order to cut costs. As a result, today's embedded systems are significantly more "open" and easy to hack than you might think.

Why would anyone want to hack or write opensource project for embedded system, when you already have a very powerful and relatively cheap PC ? There are many possible answers to this question:

- Embedded systems require significantly lower power than a PC which in my opinion is a huge advantage. It will allow you, among other things, to keep you system always on without spending too much energy.
- As a direct result of being low power, embedded systems rarely require active cooling and therefore are much more quiet.
- Embedded boards boot much, much faster than a PC - I would prefer working on embedded just for that reason alone.
- As you already have many embedded appliance in your home, it's sometimes much more convenient to run certain applications on these systems and not on a PC.
- It is fun!

Here are a few interesting things that you can do with Linux where embedded system is preferable to a PC. Let's start with all kinds of networking devices - residential gateways (RG), access points, DOCSIS modems, etc... RG is a low power device which can be always on and is usually connected to the network. As such, it is a very good candidate to run various networking applications, such as your personal web server, bittorent and many more. It will probably require an external storage, which can be either Network Attached Storage (NAS) or external USB hard drive. I think that it makes absolutely no sense to keep your PC always-on in order to run some file-sharing application, especially if your PC is a laptop.

The next popular Linux based appliance is your set-top-box, digital media adapter (DMA), personal video recorder (PVR) or even TV set. If it comes with a hard drive, it may be an even better choice for running bittorent or other file sharing applications. And if you are really adventurous, you can try to modify your PVR application to add more functionality to it or maybe just port MythTV or Freevo.

Another interesting idea is to run an Asterisk opensource PBX on one of these appliances, although you really need a TDM interface to fully exploit it's power.

I think that I've covered only a small part of applications that can be done on embedded Linux platform and I hope that once you understand that embedded Linux appliance is actually no less "open" platform than a PC, a multitude of new applications and embedded opensource projects will arise.

## 3 Know your enemy

Before you rush to hack anything, it usually pays off to spend a few minutes reading about it's system architecture. Having a general view of your system will help you later and is interesting enough by itself. Let's start with a hardware and take a brief look at three typical systems - set-top-box reference design by ST Microelectronics, personal video recorder (PVR) reference design by Sigma Designs and wireless router reference design by Broadcom. The above vendors have significant market share in their respective areas and as such provide a good example of a typical system.

ST Microelectronics STB7109 is a set-top-box (STB) decoder chip designed for low-cost high-definition (HD) applications. It is based on 300MHz SH4-202 (ST40-202) CPU core with integrated MPEG video decoder, audio decoder, Ethernet, SATA and USB 2.0 host controller. STB7109E-REF (mb448) reference board also includes 2x64MB DDR1 RAM (one DDR bank is used by the decoder and another by the OS), 8MB flash and various video output interfaces, including HDMI. ST Microelectronics chips are used in many set-top-boxes, for instance these produced by Advanced Digital Broadcast (ADB) and Scientific Atlanta.

Broadcom BCM94712 is based on BCM4712 System on a Chip (SoC) which includes a 200MHz MIPS 4K core, Wi-Fi MAC/baseband and Ethernet and USB interfaces. It usually has 2MB flash and 8MB SDRAM. Two additional chips found on the board are AC101L Ethernet PHY and BCM2050 2.4GHz radio. Broadcom chips are used in many wireless routers, for instance these produced by Linksys.

Sigma designs EM8624L is based on MMU-less ARM7 CPU core typically running at 200MHz with integrated video and audio decoding engines, Ethernet MAC (many Sigma Designs chips do not have integrated Ethernet, in which case it is connected via PCI), PCI, IDE and USB 2.0 host controllers. The reference board comes with 16MB flash and 128MB DDR1 RAM. Sigma Designs chips can be found in many PVR systems, for instance these produced by Kiss Technologies.

Software stack typically consists of either "standard" embedded Linux bootloader (such as Das U-Boot) or vendor proprietary one, Linux kernel (uClinux in case of MMU-less processors), uClibc library (as glibc memory requirements are not suited for embedded systems), busybox (which is a package containing many standard Unix utilities optimized for embedded systems) and a set of very vendor and application specific drivers and applications. uClinux embedded Linux distribution is a good example of such software stack, so if you'd like to start working on embedded project I suggest considering uClinux, even if your CPU has MMU. This paper is by no means a uClinux primer, but there is link to a few good ones in the references section.

## 4 Getting the hardware

Basically, you have two options - hack one of the off-the-shelf appliances that run embedded Linux or purchase a Single Board Computer (SBC). While the latter is more "clean" and easy approach, the former can be significantly cheaper - as you might imagine, you cannot compete on hardware prices with companies like Linksys. Let's first take a look at a few affordable SBCs designed to run Linux.

iPac-9302 by EMAC (http://www.emacinc.com) is an ARM9 based board with 16MB flash and 16MB SDRAM, USB, Ethernet, RS-232, EEPROM and some additional peripherals and extensions options. In the most basic configuration it costs $150 and comes with Linux 2.6 Board Support Package (BSP) and development tools.

FOXLX816 by ACME systems (http://www.acmesystems.it) is a MIPS based board with very similar specifications - 8MB flash, 16MB SDRAM, USB, Ethernet, RS-232 and various extension options. It's power consumption is as little as 1 Watt. Linux 2.4 and 2.6 BSP and development tools are provided and it costs 130EUR.

RouterBOARD 153 from RouterBOARD (http://www.routerboard.com/) is the cheapest SBC I'm aware of - for $119 you get a board with 177MHz MIPS 4Kc CPU, 32MB SDRAM, 64MB flash, 5 Ethernet ports and three miniPCI slots. It runs proprietary RouterOS, but Linux is supported. RouterBOARD also has a cheaper version of this board that is priced just $59 - it does not officially support Linux, but has been reported to work. If you considering to buy one of these I'd probably recommend this one, because of the low price and the fact that it has PCI, which is not always present in embedded systems and will allow you to extend by connecting off-the-shelf various peripherals.

Many more SBC can be found through http://linuxdevices.com web site, which is an extremely good source for embedded Linux information.

As the (direct) prices for these boards are relatively high, it is worth looking for affordable SBC on eBay. Although non-x86 based embedded systems are significantly cheaper in large quantities, they tend to be quite expensive when purchased for personal use.

You might also want to take a look at x86 SBCs. Intel sells rather chip Celeron based D201GLY Mini-ITX board, which comes with 1.33GHz CPU, Ethernet, PCI, USB and IDE controllers and a video card. It supports DDR2, but usually sells with no memory for $77 on eBay. However, it's not exactly and embedded board with 27 Watt power consumption (which is not bad by PC standards, but some embedded systems consume less than 5 Watts) and it requires active cooling, i.e. CPU fan. Nevertheless, it's a nice option and similar Mini-ITX boards are available from many manufacturers.

Using one of the above or similar SBC has a huge advantage (or disadvantage, depending on how you look at it) that once you purchase the board, it comes ready for development - with working RS-232 console, Linux sources and compilation environment. However, if you'd like to save a few bucks or just enjoy making stuff work not the way it was originally designed you might prefer using one of your existing appliances or purchasing, for instance, WRT54G Linksys router for $50 and extending it. Beware, that if you do this you will also have to find a way to connect to the appliance, get the sources and build a compilation environment before you actually start coding. If this is what you'd like to do - read on.

## 5 Getting the shell prompt

Assuming you decided to hack one of you existing appliances, as this is clearly much more interesting than using SBC, the first thing you'd like to do is to get a shell prompt.

Disclaimer - this guide assumes that your hardware manufacturer did not actively tried to prevent you form hacking your unit. Examples of such hard-to-hack devices are Series 2 TiVo which has a firmware/hardware to verify that the software image was not modified (the so called "tivoization"), TViX M-4100SH which has a proprietary firmware format with (probably) encrypted filesystem image (who also accidentally decided not to release the source for their GPL based product) and some others. Hacking these kinds of appliances is a subject for completely different talk and can cause some legal complications as SW reverse engineering is illegal in some countries. Fortunately, most embedded Linux appliances follow some de-facto standards and conventions which are the topic of this talk. As for TiVo-like hard-to-hack appliances - if you enjoy reverse engineering for the sake of it, then they can be fun, but if instead you prefer to create something useful - there are plenty of alternatives which are much more hack-friendly.

### 5.1 RS-232

Although there are some other options (discussed below) the most generic and reliable way to connect to the appliance is serial console. It will allow you to see all kinds of error messages, including kernel panic and most important, it will allow you to interact with the bootloader, which in some cases may be the only way to program a new firmware image to flash. Most, if not all, embedded systems have UART controllers used for debug and development, which are not connected in the production system. In order to use it, you will have to open your appliance (which will probably void the warranty) and do some soldering (which will not help with the warranty either).

A universal asynchronous receiver/transmitter (UART) controller is a hardware that translates data between parallel (visible to the computer, i.e. software) and serial interfaces. It is usually part of the embedded CPU which is the reason it is present even in production systems. UART pins, however, cannot be directly connected to a RS-232 interface - they have to go through a voltage level converter chip, which will translate logic UART levels to the external RS-232 levels. Maxim (http://www.maxim-ic.com) MAX232 is the most frequently used (but not the only one) level converter - look for this chip on your embedded board, but usually it will not be there as it is a good candidate for cost reduction.

The easiest way to use the RS-232 level converter is to purchase a connector kit, which basically includes a MAX232 chip, 9-pin female RS-232 connector and a header. The header is very convenient, as you would probably prefer not to solder the converter permanently to the board - soldering just the header will allow you to easily connect and disconnect the converter and probably reuse it with multiple systems. There are many MAX232/MAX233 kits on the market, for instance AD233AK, which is sold for $19 by CompSys (http://www.compsys1.com). As shown in figure 1 typical RS-232 adapter has a DB9 RS-232 connector for the PC and 4 pin (the extra 5th pin of this particular one is not connected) header to connect Tx, Rx, Ground and 5v on a micro controller board.

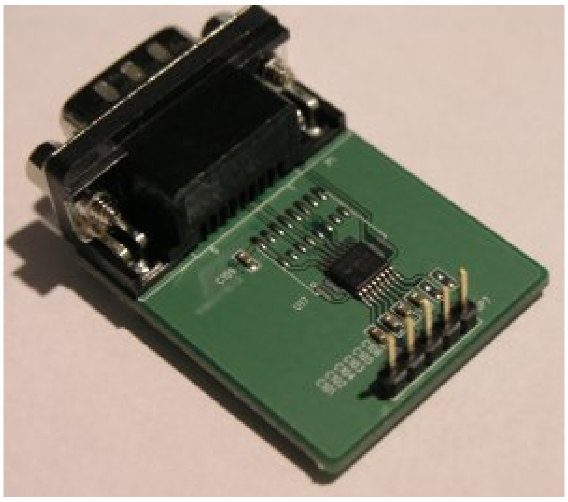

Figure 1: Typical RS-232 level converter kit

If you are really into soldering it's quite easy to build the above converter yourself as the schematics are easily available (for instance from CompSys site itself). As illustrated by figure 2 all you have to do is to connect MAX233 $T1_{in}$ and $R1_{out}$ to UART TX and RX pins on your board respectively, and $V_{cc}$ and $G_{nd}$ to 3.3 to 5 volts power supply and ground. MAX233 $R1_{in}$ and $T1_{out}$ go to pins 2 and 3 on the DB9 connector according to RS-232 standard and the following pins have to be connected to each other according to Maxim specification: 17 to 12 ($V_{-}$), 16 to 10 ($C2_{-}$) and 15 to 11 ($C2_{+}$).

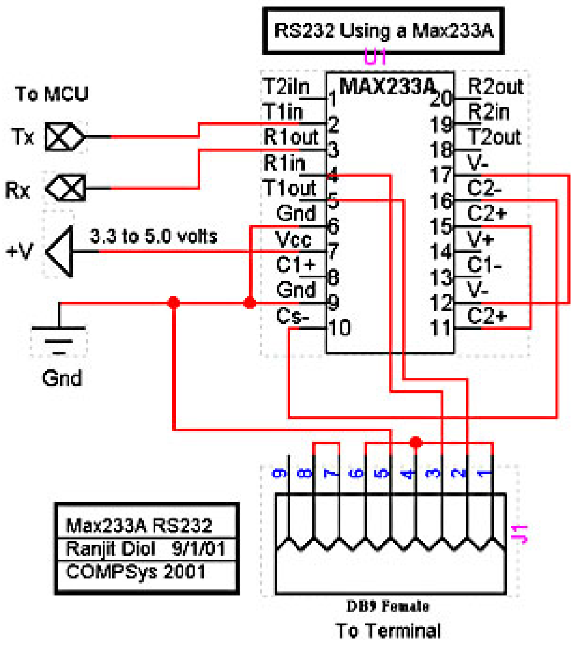

Figure 2: AD233AK schematic

Note that MX233 supports two UARTs, so half of the pins are not connected. Also note that there are no RTS/CTS pins, as MAX233 does not support flow control, so be sure to disable hardware flow control in your terminal emulator.

Now, to the really tricky part - locating UART pins on your board. Remember that we are looking for transmit (TX), receive (RX), voltage ($V_{cc}$) and ground ($G_{nd}$) pins. It's tricky, because in the general case you won't have the board schematic showing you the pins locations. If you manage to get a chip pinout, you can probably trace UART pins on the Printed Circuit Board (PCB), but chances of getting either datasheet are not very good as chip vendors usually provide these only to their customers. If you are the first one to hack that particular hardware you will probably have to use the oscilloscope to find the UART pins.

Oscilloscope (sometimes called scope) is a piece of electronic test equipment that allows signal voltages to be viewed as a two-dimensional graph of one or more electrical potential differences (vertical axis) plotted as a function of time. This device can be quite expensive and it's definitely not worth to buy one just to get UART working, but since you will only need it for a few minutes you will probably be able to borrow it from a university lab or some other place. First look at your board and try to find possible candidates for UART pins, there are no specific rules, but there must be at least 4 UART pins, probably located near each other. Once you found a group of pins that looks like UART, you have to verify that it is indeed UART and identify the UART pins. Finding $G_{nd}$ is easy as it will most probably be connected to the PCB ground plane, as shown in figure 3 (note the "cross" connecting two pins in the lower-right corner). Finding $V_{cc}$ is easy too - just look for 3.3V pin using the scope, but TX and RX are not.

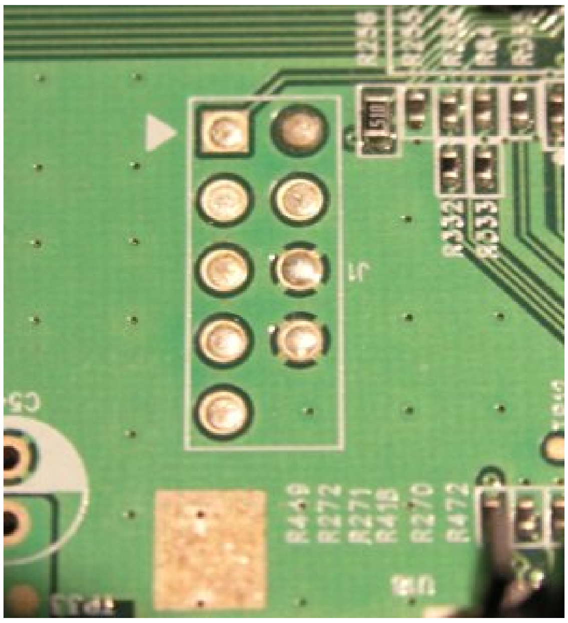

Figure 3: Ground - the "cross" connecting two pins in the lower-right corner

Once you found the ground, connect the scope ground wire to it and try to find UART TX. In order to find it we exploit the fact that whatever software is running on this board, it probably prints something via UART during boot process. We repeatedly connect the scope probe to each pin, reboot the board and if we see any activity on the scope monitor, there is a chance that it is TX. In order to verify this, we look for the smallest pulse (hoping that it represents one bit), measure it's length (which represents pulse duration) and calculate bit (baud) rate by dividing 1 second by the pulse duration. We then compare the resulting bit rate to one of the standard UART rates (the following rates are most common - 9600, 19200, 38400, 57600 and 115200). If it matches - viola, it's UART TX!

Let's illustrate the process described above using an example. Figure 4 shows the scope monitor capture for the shortest pulse while the scope probe was connected to the upper-left pin on the PCB shown in figure 3 (conveniently marked by an arrow). As you might see, the pulse duration is 8.8$\mu$s,

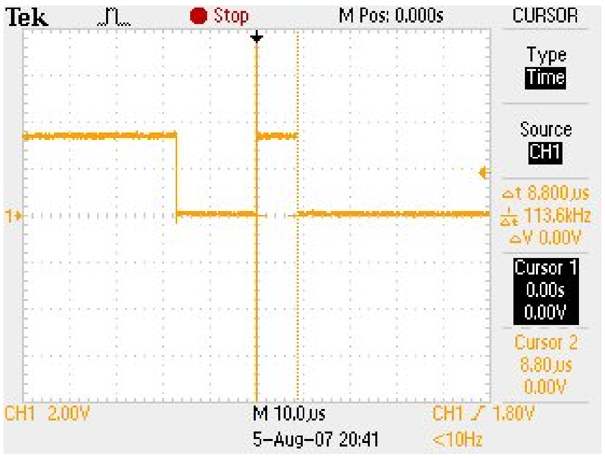

Figure 4: Scope screen capture for the shortest pulse - UART TX candidate

which gives us $1^9/8.8 = 113636.36$ bit rate, which is quite close to 115200. Meaning that this pin is with very good probability a UART TX. I'm afraid that the only way to find the last missing pin - RX, is elimination.

After locating all the pins you can solder the header to these pins, connect RS-232 level converter kit and run your favorite terminal emulation software, hoping to see the familiar message:

```
Linux version 2...
```

If you'd prefer to avoid opening your appliance you may try your luck with the firmware upgrade option discussed below. However, be prepared to fallback to hacking UART as it is highly probably that your newly created firmware image will not boot for the first time - and then your only chance of getting it to work again is to program a working firmware image using bootloader via UART.

## 5.2 Modifying the root filesystem

Assuming you either managed to connect via RS-232 or don't want to make any hardware changes to your appliance, the next step is to modify the root filesystem in order, for instance, to put a telnet or ssh daemon on it, as I would image everybody will want a network access to their embedded box.

The first and easiest thing to do is to ask your appliance manufacturer for the sources. If the sources are not available, you can (and probably should) inform the manufacturer that he is actually required by GPL to provide you the sources for your Linux based, and thus covered by GPL, product. Remember that he is required to provide complete sources and the means to compile them, providing partial sources that cannot be used to recreate the image is not enough. You can always point out to Linksys as a relatively good example of GPL compliance. If you manage to get the sources, then the process of modifying the root filesystem is straightforward - just modify the root filesystem directory and recreate the image using makefiles or scripts provided by the vendor. If the vendor does not cooperate - report to the GPL violation to FSF and proceed to the next step.

If you don't have the sources, you can try to modify the binary filesystem image, which will require some basic knowledge of embedded Linux filesystems internals. Most probably the root filesystem of your device will be either cramfs or squashfs, which are two compressed read-only filesystems that are usually used on embedded Linux. These filesystems can be mounted on a PC (squashfs may require a kernel patch as it is not part of the kernel.org tree). You can get the filesystem image either from running system (provided you managed to get an RS-232 working) or from a firmware upgrade image (provided your appliance supports firmware upgrade). The former case is straightforward - just use dd(1) to read the filesystem image from flash partition /dev/mtdblock/[x] mounted as /, transfer it via tftp to the PC and mount to read its content.

If you did not manage to get shell access via RS-232 you will probably have to use firmware upgrade image which is less trivial as this image has proprietary format and will differ from vendor to vendor. However, even without the exact knowledge of firmware image format it is still possible to identify the filesystem image type and location, i.e. offset, inside the firmware image as filesystem image will be usually stored unmodified (because there is no technical reason to modify it - it is already compressed). In order to find the filesystem offset inside the firmware upgrade image we will look for the filesystem superblock, which has a magic number identifying this filesytem and some other important information.

Cramfs super block which is located at the beginning of the filesystem image has the following structure (defined in include/linux/cramfs_fs.h):

```
struct cramfs_super {
        __u32 magic;                    /* 0x28cd3d45 - random number */
        __u32 size;                     /* length in bytes */
        __u32 flags;                    /* feature flags */
        __u32 future;                   /* reserved for future use */
        __u8 signature[16];             /* "Compressed ROMFS" */
        struct cramfs_info fsid;        /* unique filesystem info */
        __u8 name[16];                  /* user-defined name */
        struct cramfs_inode root;       /* root inode data */
};
```

In order to find the Cramfs offset inside the firmware image you will have to search for 0x28cd3d45 magic number. Note that the data is in host-endian byte order, which may not necessarily match your PC little endian byte order. In that case the magic byte order will be reversed and you wont be able to mount the image on a PC. You can still use uncramfs (or cramfsck) to extract the image content, however you won't be able to use mkcramfs to recreate the image in the byte order different from that of your host. One of the ways to overcome this is to use QEMU to run a Linux compiled for big endian MIPS (or any other) system and make all cramfs manipulation in emulated big endian system.

Squashfs is a little less common, but still used frequently. There are two versions of squashfs - the original one uses GZIP compression algorithm and Squashfs LZMA. This filesystem is not part of the Linux kernel, so kernel patch is required if you'd like to mount it on your PC. Squashfs super block structure (defined in /include/linux/squashfs_fs.h) is too large to print here, but the important part for our purposes is the magic number 0x73717368 located at the beginning. As with cramfs, this magic number can be in reverse order if your embedded system endianess does not match that of your PC. Squashfs LZMA magic number is 0x71736873. Squashfs has a very convenient utilities that you can use to open (unsquashfs) and create (mksquashfs) the filesystem image, which do not require that the host, i.e. PC, byte order match target, i.e. embedded, byte order.

Let's try to modify Linksys WRT150N firmware as an exercise. Using

```
hexdump WRT150NV11_1.01.9_US_code.bin
```

we find Squashfs LZMA magic number at offset 0x82E2C:

```
...
0082e20 3d81 4988 f5ff 0cca 008d 0000 7368 7371
...
```

Using dd(1) we extract the filesystem in the following way

```
dd if=WRT150NV11_1.01.9_US_code.bin of=fs.bin skip=536108 bs=1
```

then open it using unsquashfs-lzma, modify, compress using mksquashfs and put it back at the correct offset in the firmware image. It is best to keep the original filesystem image size at this stage (since you don't know the firmware image format used by the firmware upgrade software), but once you have telnet running and have a normal Linux shell access you can always write a completely new filesystem image without any restrictions directly to the appropriate /dev/mtdblock/[x] device.

Of course, in case of Linksys this exercise is not necessary as the company provides source code for the product, but the approach can be used to modify any firmware image, unless the manufacturer made a special effort to make it difficult, for instance by encrypting the firmware upgrade image.

Apart from cramfs and squashfs there are a few other less frequently used embedded Linux filesystems: romfs and JFFS2.

Romfs is a simple read-only filesystem. It can be easily identified by looking for the "-rom1fs-" ASCII string, which is located at the beginning of romfs filesystem. It's superblock is defined (in include/linux/romfs_fs.h) as follows:

```
struct romfs_super_block {
        __be32 word0;
        __be32 word1;
        __be32 size;
        __be32 checksum;
        char name[0];           /* volume name */
};
#define ROMSB_WORD0 __mk4('-','r','o','m')
#define ROMSB_WORD1 __mk4('1','f','s','-')
```

Romfs image can be created using genromfs utility and it can be mounted on any Linux system as it always uses big-endian byte order.

Another flash filesystem that can be found in embedded devices is JFFS2. It is rarely used because of 5 spare flash sectors overhead (to implement garbage collection), but systems that require read/write filesystem on flash have little choice. Contrary to the filesystems discussed above JFFS2 is a log-structured filesystem as it was specifically designed for flash devices. What this means for us is that there is no well defined superblock at the beginning of filesystem image that will help us to identify the filesystem type and offset, but rather JFFS2 image consists of a list of nodes, all of which have similar header of the form (defined in include/linux/jffs2.h):

```
struct jffs2_raw_[dirent, inode, summary] {
    jint16_t magic;      /* A constant magic number.  */
    jint16_t nodetype;
    jint32_t totlen;     /* Total length of this node (inc data, etc.) */
    jint32_t hdr_crc;
...
}
```

With magic equals 0x1985 (as usual, it is in host byte order format). Unfortunately, this magic number is too short to identify the filesystem offset with sufficient probability. However, we can look for a eraseblock header node with has a nodetype of 0x2005 and would always be located at the beginning of the flash sector.

Linux kernel, which is usually part of the same firmware image is much harder to find, because it is usually compressed using different algorithms. Don't worry - you don't really want to modify it anyway, unless you have full BSP sources.

## 6 Crash-course in embedded Linux development

Assuming you either managed to get shell access to your appliance or found a way to modify the firmware upgrade image, the next thing you would probably like to do is to put some new application to it. Let's use as an example a Sigma Designs Promedia 8510 Networked DVD Player reference board, on which many PVR designs are based (for instance popular Kiss Technologies DP-450 DVD player). For the purpose of our exercise EM8510 hardware is not very different from EM8624L described above.

In order to write an application for your EM8510 you will need to set up an embedded Linux "development environment", including

- Linux kernel sources - optional, unless you want to modify the kernel itself or one of the drivers. Note that you need the kernel with all the modifications made by hardware vendor to support your device, i.e. BSP.
- uClibc - as glibc memory requirements are too high for embedded systems chances are that your system uses uClibc. Note that as a first phase you can use glibc and link your application statically, which will be much easier. Eventually, however, you will have to link your application with uClibc.
- Cross compiler toolchain for your processor, little endian ARM7 core in case of EM8510

Getting all of the above for Sigma Designs 8510 is easy, as the kernel sources were released by Sigma Designs in order to be [partially] compliant with GPL. These sources can be obtained from uClinux web site (http://www.uclinux.org/pub/uClinux/ports/arm/EM8500/), along with ARM cross compiler toolchain, uClibc and additional utilities.

### 6.1 uClinux peculiarities

This seems to be a good place to list some uClinux aspects which makes it different from vanilla Linux. Note that although many embedded systems are based on uClinux, there are some which are not and there are many who chose to use uClinux on hardware with MMU just because it is a very convenient embedded Linux distribution and therefore features below do not necessarily apply to your particular system. Or in other words, try not to get confused as there are uClinux kernel and uClinux distribution, and the latter can be used without the former. Back to the uClinux kernel.

First and foremost, there is no MMU, so the whole memory management is very different. There is no memory protection for userspace applications, but this is not so important for embedded systems. What's more important is the fact that uClinux applications have fixed stack size and since many Linux application programmers forgot long time ago what stack is, many applications can have an unreasonably large stack - and if it grows beyond stack size limit, you will discover some very interesting effects. All standard binary executable formats are not supported, instead a new flat (BINFLT) format is used. Since it is not supported by standard GNU binary utilities, uClinux executables has to be converted using “elf2flt” utility. fork() system call does not exist, but vfork() does. So if you application uses fork in a way that cannot be easily converted to vfork(), you are in trouble.

uClinux also supports the so called execute-in-place (XIP) functionality, which allows running applications directly from flash without loading them into RAM, provided they reside on a suitable filesystem, i.e. romfs. Although this looks cool I can’t think of many applications where you would need such functionality, because RAM prices are significantly lower than flash, so trading RAM for flash is a strange trade-off, especially taking into account performance hit and the fact that XIP does not work with compressed filesystems such as cramfs.

Shared libraries work different in uClinux and are rarely used.

## 6.2 Telnet for DP-450

Although the best way to put telnet daemon on your DP-450 would be to recompile the busybox it already comes with we will use a standalone telnet daemon as an example. I suggest you do the same the first time you are trying to modify the root filesystem, because messing up busybox can leave your embedded Linux without a shell (all busybox applications are part of the same single binary). Since you will probably want to re-compile busybox anyway we will use it as an example, i.e. put two versions of busybox - the old one that is know to work and the new one with telnet daemon.

Assuming you downloaded and installed ARM cross compiler toolchain (arm-linux-tools-xxx.tar.gz) the following steps will produce a statically linked ARM executable in BINFLT format:

```
make menuconfig
make LDFLAGS="-Wl,-elf2flt=-s32768" CROSS=arm-elf- CFLAGS_EXTRA="-D__uClinux__"
```

At the “menuconfig” stage remember to set “USING_CROSS_COMPILER” and disable “ash” shell, as it won’t run on uClinux. “CROSS” parameter tells make to use arm cross compiler toolchain and ”-Wl,-elf2flt=-s32768” linker option instructs it to produce not “elf”, but rather “binflt” binary for uClinux. Also note that the stack size for this application is fixed and specified as an argument for elf2flt utility. In the above example I used 2.95.3 gcc compiler, which is pretty old, but still not uncommon in embedded and busybox version 1.0, to match the compiler version. As the above compilation process is a bit non-standard, there are a good chances that the compilation will fail with various errors, depending on the busybox and compiler versions. As there are too many things that can go wrong in the compilation process for embedded Linux, I believe it is beyond the scope of the paper.

Now that you have the telnet daemon all you have to do is to re-generate the romfs DP-450 root filesystem image (using genromfs utility) with it. Well, almost, since DP-450 has proprietary /bin/init application. Since you don’t have the init sources you will probably have to write your own small init, which will run telnet daemon in the background and then continue standard DP-450 initialization by running the original /bin/init.

## 6.3 Using uClibc

In the above example we ignored the usage of shared libraries completely, which is what I suggest you to do if you are running uClinux, i.e. you have no MMU. In fact this is not the most generic example, as most embedded processors today come with MMU and creating complex applications without shared libraries would be hard. glibc, which is Linux standard library, is rarely used on embedded because of it's high memory requirements. uClibc is usually used instead, which was specifically developed for embedded systems and optimized for low memory footprint. When working with uClibc, remember that:

- Some glibc features are missing in uClibc (low memory footprint comes at a price)
- Since gcc is pretty much written to work with glibc, uClibc provides compiler and linker wrappers (or a special toolchain) that should be used to simplify the build process

The following functionality is missing in uClibc: libdb (database library), libnss (network authentication and DNS resolution), libnsl (Network Information Service (NIS)), librt (Asynchronous I/O, POSIX message queues, shared memory and timers) and some other. As you may see, none of the above is essential in embedded.

Apart from the uClibc limitations listed above, there is one significant difference when working with uClibic compared to standard glibc - you need a different uClibc toolchain. In the past (prior to uClibc version 0.9.22) it was possible to use uClibc with your standard toolchain using uClibc gcc/g++ compiler wrappers (which relied on gcc ”-nostdlib” and ”-nostdinc” options). However, because of the increased complexity of uClibc this no longer works and you need a complete compiler toolchain built for uClibc. You can either get a pre-compiled version from the uClibc site or use their very convenient “buildroot” script to create your own uClibc toolchain for your architecture.

## 6.4 Kernel

Hacking embedded Linux kernel is both simple and extremely non-trivial. It is simple because if you have some experience with kernel programming, you will find that embedded kernel programming is not very different. It can be very difficult, though, if you don't get the kernel sources with BSP for your particular HW, writing the BSP by yourself is not simple, especially when detailed HW documentation is not available.

Linux BSP guide is beyond the scope of this article, but I'd like to list here a few differences in kernel mode programming between vanilla and embedded Linux, which you may find useful if you are doing a small patches to already working embedded Linux kernel:

- Memory allocation is different, although this fact is hidden from you by an API. It is still important to understand the mechanism behind this API in order to use it efficiently. You can still use kmalloc, but this kmalloc may not necessarily use standard “slab” and “page_alloc” code. The main reason for this is that the memory/performance trade-off of the power-of-two algorithm which is true for vanilla Linux is not true for embedded systems with low memory. There are two solutions to this, the original uClinux "page_alloc2" algorithm and Linux 2.6 slob allocator. In order to understand the difference between the two approaches you have to remember that SLOB (or SLAB) is the algorithm used by kmalloc, i.e. it is responsible for small memory allocations (it allocates the power-of-two bytes) while "page_alloc2" (or "page_alloc") does large allocations (it allocates the power-of-two pages). BTW, "vmalloc" on uClinux just calls to "kmalloc", so beware if you allocate large buffers using this function.

- It is not uncommon to find embedded systems without PCI bus. In SoC devices can be connected directly to a CPU local bus and for external devices many vendors use there proprietary buses (in order to save cost, power or for historical reasons). Examples of such buses are ST MPX and TI VLYNQ. Although bus drivers are usually very different, Linux 2.6 has a nice framework of platform "pseudo" bus drivers, which I would use if I had to write a bus driver and I hope that your BSP vendor thinks the same. From the device driver writer perspective the only difference is that you should use "platform_get_irq()" and "platform_get_resource()" functions instead of "pci_find_capability()". Writing bus driver is beyond the scope of this paper (and I'm afraid there no guide on this subject, but platform bus driver is pretty self-explaining).

Beware, that apart from this your particular kernel may contain many non-standard modifications added by your HW vendor for a multitude of reasons - shortcuts to get performance improvements, workarounds for HW bugs, etc. One of the typical examples is a "fast path" between two network devices which does not go through Linux network stack, which is a sort of "optimizations" that many vendors are doing in order to squeeze more performance from slow embedded processors.

If you are new to embedded, my advice is to start from applications and dive into kernel space later, preferably after you get the BSP sources and HW documentation.

## 7 Resources

A few useful web sites for embedded Linux developer.

### 7.1 Related opensource projects

If you are into hacking embedded Linux appliances, it's good to know you are not alone and take a look at some fun projects other people did for their embedded stuff:

- OpenWRT is an opensource firmware for residential gateways, originally created for Linksys WRT54G wireless routers. It has grown into an embedded Linux distribution and now supports a wide range of devices - http://www.openwrt.org.

- SlugOS is the collective name for a group of firmware distributions that run on Linksys NSLU2 (Network Storage Link for USB 2.0 Disk Drives) - http://www.nslu2-linux.org.

## 7.2 Embedded Linux tools

Although support for some embedded processors has been merged into the main Linux tree, you really should check the websites below for latest kernels and cross compiler toolchains for your embedded CPU:

- MIPS - http://www.linux-mips.org
- ARM - http://www.arm.linux.org.uk, http://www.arm.com/linux/index.html
- SuperH - http://www.linux-sh.org, http://sourceforge.net/projects/linuxsh, http://www.stlinux.com
- PowerPC - http://penguinppc.org
- MMU-less processors - http://www.uclinux.org

## 7.3 More information

If you'd like to read more about embedded Linux, take a look at the following:

- "uClinux for Linux Programmers" LinuxJournal article - http://www.linuxjournal.com/article/7221
- Guide to getting started with uClinux - http://www.uclinux.org/get_started
- Glibcs and uClibc differences - http://www.uclibc.org/downloads/Glibc_vs_uClibc_Differences.txt
- All about Linux-powered devices - http://www.linuxdevices.com