Porting Linux on a MPC8313ERDB

Many ways to build and deploy a Busybox/Linux OS using Buildroot, TFTP and U-Boot

Ivan Simonini <ivan.simonini@roundhousecode.com>

Abstract

This wants to be a guide to the sometimes frightening task of porting a working OS to an obscure hardware platform. Though it took several weeks to get it done with the first time, the entire operation is not particularly tough or slow.

There are no more than seven steps, from ashes to a complete working installation; you just need a machine with a Ethernet port, a Serial port, an Internet connection, a few packages to be installed and a decent amount of patience.

Given a basic knowledge of the correct tools, it eventually became pretty easy.

Table of Contents

1. The easy way 1
    1.1. Configuring BuildRoot 1
    1.2. Package selection 2
    1.3. Kernel configuration 2
2. Deploy on the board 2
    2.1. Das U-Boot 3
    2.2. TFTP 3
    2.3. BOOTP 3
    2.4. Boot command 4
    2.5. Image conversion 4
    2.6. Boot script 4
    2.7. Install the images 5
3. Troubleshooting 5
    3.1. Conflicting versions 5
    3.2. Old kernel conflicts 5
    3.3. Missing toolchain support 5
    3.4. Missing dtb file 5
4. The hard way 5
    4.1. Toolchain 6
    4.2. Kernel 6
    4.3. BusyBox 6
    4.4. Virtual disk 7
    4.5. Install 7
5. Software sources and version numbers 8

1. The easy way

So, you find yourself with the need to put a Linux installation on a board. So did I. *Don't panic*. There is a easy way, there is a single reasonably-easy-to-use tool capable of doing all the work you need: its name is `BuildRoot`.

Once configured, a single **make** call will compile and build everything you need.

1.1. Configuring BuildRoot

```
$ cd buildroot/
$ make defconfig
$ make menuconfig
```

You'll see the GUI of `KConfig` (originally born to be the Linux kernel interactive configuration menu). There are a number of option you can customize, first of all the *target architecture* (`powerpc` for this example); if your target architecture has a variant, you can also customize that (or `generic` if there is none, like in this example).

Then, you shall set your *build option*. It is important, to save time, to set the `number of jobs`

to the highest value possible: this should be one plus the number of CPUs available on your system. You can also enable `compile cache` (to speed up the compilation) if you want, add the `documentation` on your target (there won't be any onboard if you don't), and choose to prefer `static libraries`.

Then you need to customize your *toolchain*. First, you need to choose the *kernel headers version* (`3.3.x`), the *uClibc* version (`0.9.33.x`), the *binutils* version (`2.21.1`), the *GCC compiler* version (`4.5.x`). You can now choose to add support for a number of features, like `large files`, `IPv6`, `WCHAR`, for threading, for locale, for languages (other than C).

Then, it's time to give a name to our new system in the *system configuration* menu, where you can set the `hostname` and the initial message (which is shown before login). Here you can select the default port on which run **getty**, which will be the main login prompt for your system. The default value is `ttyS0`, the first serial port; if you change this value, be sure to select a valid connection, otherwise your system will become unusable.

You must also choose the output format for the filesystem image: there are many; and for many of them you may specify the preferred compression method. For this example, I choose to create an `ext2` image compressed with **gzip**. Note that your system and your kernel shall support the chosen compression algorithm, or else your image won't be loaded.

1.2. Package selection

Now, for the *package selection*, you shall select every user-space software component needed by your system. `BusyBox` can provide many of them, others can be added to your configuration (as BuildRoot will download and compile them for you).

Keep in mind that this is the default config by Buildroot, and it's pretty lightweight; this means that many features you may want are probably missing; for this example, I had to add **pgrep** and **netcat**.

It is also true that you may specify an arbitrary configfile, which will be passed to BusyBox before starting its compilation. If this does not please you, you can also run

```
$ make busybox-menuconfig
```

and select whatever combination you like.

1.3. Kernel configuration

The final (and most delicate) step is to adjust the kernel settings. There are just a few: first, you must specify a version; you can select one (usually, the latest stable release) or maintain the `same as toolchain kernel headers`, which is the wisest choice. If you're experimenting or if you are looking for trouble, you may choose a specific version, a custom tarball or even a custom `git` tree, or choose some patches to apply.

The second thing to do is give the name of a kernel configfile: you can choose a custom one, or use a default one; in this case, you must specify its name; for this example it was `83xx/mpc8313_rdb`. Finding the correct name to insert here is tricky: you shall search the kernel source tree for this files. You'll find them in the `arch/${ARCH}/configs/` folder, where `${ARCH}` is your target architecture.

```
$ cd output/
$ cd toolchain/
$ cd linux-${VERSION}
$ ls arch/${ARCH}/configs/
```

Keep also in mind that Buildroot will add the suffix "_defconfig" on its own, so be sure to remove it when you insert the name, or you'll experience a config-time error.

Finally, you can choose the output format for your kernel (I choose `uImage`, optimal for uBoot) and if you want the kernel installed on your filesystem image or not.

In case you need to perform some fine tuning on your kernel configuration, you can always access the interactive menu via

```
$ make linux-menuconfig
```

If you feel satisfied, it's time to compile with

```
$ make
```

Brace yourself, now, because BuildRoot needs to download the source code for everything you choose to build, and then there is compilation; depending on your connection speed and on your CPUs, this whole operation can take one or more hours.

After this, you'll find that Buildroot created some files for you, in the `output/images/` folder. You shall see at least a filesystem image, maybe a compressed one (if you selected a compression) and the kernel image; if you see them, you're ready to deploy them, jump to Section 2, "Deploy on the board". If you don't, Section 3, "Troubleshooting" gives you advice to quickly fix some issues; if you still don't, then you can try Section 4, "The hard way" or eventually give up.

2. Deploy on the board

Having a kernel image and a filesystem is not enough, of course; you need to deploy all this data to the board to make it work.

What you need to do is upload three different image files to the board – optionally burn them to FLASH memory – and then boot the system from them.

First, you must connect a host machine to U-Boot, then set the network interface, upload the data via TFTP and finally boot.

2.1. Das U-Boot

The first application you'll see running on the MPC8313ERDB when you switch it on is U-Boot, a universal bootloader that gives all kinds of power to control your hardware before booting an operating system.

In order to interface your laptop with the U-Boot you need a serial connection. On the board, use the UART0 port (the upper one); on your laptop, you can use whatever serial port you like (if you happen to have one) or you can use a serial-USB adapter (one comes with the board). Once established the physical connection, you'll need a terminal emulation program on your host (I used minicom) with some simple configuration: connection port, baudrate, bit-per-character and parity and stop bits.

I used ttyUSB0 (because I used the USB adapter) and the highest available speed (as the board allowed) 115200 BPS, 8N1.

U-Boot is a shell-like program with a number of embedded standard commands, and its behaviour can be controlled modifying one of its many *environment variables* via the command **setenv**. You can use **help** or **?** to get help on a certain command's syntax.

2.2. TFTP

U-Boot can recieve data (your filesystem and kernel images, for example) from the network and store it in memory. To recieve something, just type

```
=> tftp <destination> <filename>
```

where <destination> is the memory location where you want to store the file.

In order to make these files available to U-Boot, you must set up the network (see Section 2.3, “BOOTP”) and have an TFTP server running. I choose to use tftpd, a simple application which runs under xinetd. Just save this file as /etc/xinetd.d/tftp:

```
service tftp {
  disable     = NO
  type        = UNLISTED
  id          = tftp-dgram
  socket_type = dgram
  protocol    = udp
  user        = root
  wait        = yes
  port        = 69
  server      = /usr/sbin/in.tftpd
  server_args = -s <path/to/dest>
}
```

where <path/to/dest> is whatever location you like to store your images in. Note that you may need to restart your network server in order to this service actually running.

2.3. BOOTP

There are two ways, on U-Boot, to set up the network. You can manually set one of the two network interfaces with the U-Boot command **setenv**, modifying the current values of the environment variables ipaddr, serverip, subnetmask…

The other way, much more feasible for network-based boot, is to set a bootp server on your host and let U-Boot get itself an IP address via dhcp. I choose to use bootpd, another simple application which runs under xinetd. You can configure it by saving this file as /etc/xinetd.d/bootp:

```
service bootps {
  disable     = no
  type        = UNLISTED
  id          = bootp-dgram
  socket_type = dgram
  protocol    = udp
  user        = root
  wait        = yes
  port        = 67
  per_source  = 1
  server      = /usb/sbin/bootp
}
```

This is not enough, tough: it just configure the service. Information about the addresses and the default content to send are stored in /etc/bootptab, another simple text file:

```
.common:sm=255.255.255.0: \
  :sa=192.168.2.1: \
  :gw=192.168.2.1:
testboard:tc=.common: \
  :ht=ether: \
  :ha=00e00c007e21: \
  :bf=<startup>: \
  :ip=192.168.2.7
```

where you can specify a number of parameters: the subnet mask sm, the server address sa, the gateway address gw, the bootfile name bf, the host hardware type ht, the physical hardware address ha, and of course the IP address ip.

The first configuration with a valid hostname (testboard) is used, the ones with an invalid

hostname (`.common`) are dummy but can be included in other hosts' configuration using the `tc` tag. Consult the **$ man bootptab** page for more information and tags.

Special attention shall be paid to the `bf` tag: its content is the name of the boot file, which is send to the reciever when the IP connection has been established. This file can be a filesystem image, a kernel image, or even a pre-boot script.

2.4. Boot command

If you do not interact with U-Boot before the timeout runs out, or you explicitly call the **boot** command, what happens is actually

```
=> run bootcmd
```

where `$bootcmd` is one of the many variables U-Boot keeps in its environment. Its default value looks something like **bootm fe10000 fe300000 fe700000**.

The **bootm** command boots the system from memory; three different images' location must be specified: the kernel image, the filesystem image and the device tree blob. You can specify arguments from the main memory (**0x0000000 ~ 0x8000000**), or from the on-board FLASH memory (**0xFE00000 ~ 0xFE7FFFF**).

It is wise to put those images in the chosen location before you load them: when you are still testing, you shall prefer to store the data in the main memory, by uploading them via **tftp**; when you're done testing and need a more persistent installation, you shall burn the images on FLASH, via **cp.b**.

2.5. Image conversion

Before you upload the images to the board, know that U-Boot can work only with a specific file format. The kernel `uImage` you compiled is already well-formatted, as you can see by calling **file** on it; the `.dtd` file is just binary data for the kernel, and there is no need to change them. The filesystem image, tough, is different: you have to convert it to an image before uploading it to the board.

To do that, you'll need the **mkimage**, which a tiny application you should find already packaged for your distribution.

mkimage can convert one or more source files to a single U-Boot image; to convert your filesystem image type

```
$ mkimage -A ppc -O linux \
      -T ramdisk -C gzip \
      -d <rootfs.gz> <rootfs.img>
```

in this example I gave only the needed options (target architecture, operating system, content type and compression, plus the source file) but you can also specifiy the load address (`-a N`, default 0), the entry point (`-e N`, default 0), a name for the image (`-n <name>`, default empty) and if the image should be executed in place (`-x`).

Note that I set the filetype to `ramdisk`, as my whole system is volatile; when you're willing to burn the images on FLASH, you may want to specify `-T filesystem` instead (see **$ mkimage -T help** for the complete list of supported filetypes).

2.6. Boot script

When you boot, there may be specific parameters you want to pass to the kernel; there may be some software you want to run; there may be some environment changes you want to apply. Eventually, there could be an established configuration you want to launch automatically.

The easiest way to do that is to upload and run a script. U-Boot allows you to run command (as variables) via the **run** command, but you may also execute a script (stored in the main memory) by calling the **autoscr** command. If you specify something like

```
=> setenv bootcmd=" \
     setenv loadaddr 100000; \
     dhcp; \
     autoscr 100000"
```

you'll obtain a custom boot sequence: U-Boot will execute the content of the script it obtained as part of the dhcp procedure (remember Section 2.3, “BOOTP”?) The purpose of setting the `$loadaddr` variable before calling the **dhcp** is that the `bootfile` store location is specified by that variabile: you may do something slightly different, like

```
=> setenv bootcmd="dhcp; \
     autoscr $loadaddr"
```

if you don't want to modify `$loadaddr`. Anyhow, the most important part is, obviously, the content of the script. Two properties: it must contain valid U-Boot commands, and it must be properly formatted as an image file.

First, its content: your basic script shall upload all the necessary image files, set the kernel arguments and eventually boot. Here's an example:

```
tftp 1000000 uimage
tftp 2000000 rootfs
tftp 3000000 mpc8313erdb.dtb
run ram_args addip addtty
run run_vscld1
bootm 1000000 2000000 3000000
```

As you can see, this script stores three images into three differnet location, then it runs four different

Second, the format. As seen in Section 2.5, “Image conversion”, you'll need to **mkimage** the script; this time, far less options are necessary: just call

```
$ mkimage -T script \
    -d <filename> <startup>
```

Your script is now ready to be uploaded and executed by U-Boot through the **autoscr** command.

If you configure your `/etc/bootptab` so this script is the default boot file, U-Boot will recover that when it calls **dhcp**.

2.7. Install the images

To store permanently an image file on the board, you have to copy it on FLASH, for the main memory. Once the file is stored in memory, you have to deactivate the FLASH memory protection, copy the data on it and finally restore the protection.

```
=> protect off <first> <last>
=> erase <first> <last>
=> cp.b <source> <first> $filesize
=> protect on all
```

You can use the **cp** variant **cp.w** or **cp.l** if you want to specify the number of chunck (the third parameter) in words or in long words instead of bytes; but the `$filesize` variable is automatically set to the (byte) count of the last **tftp** operation you performed, so it's ready-to-use with **cp.b**.

I advice you to use extreme caution when you choose the install location for a file: you have to avoid collision with other files.

3. Troubleshooting

Murphy's law is always standing. Not everything works the first time, nor with default configuration. Here's a list of some quick fixes I had to apply myself.

3.1. Conflicting versions

Not every version of every tool can work together. The golden rule is to use the latest available version of every tools you need, of course, but when you're stuck with a certain (old?) version of something, my advice is to search for the release date of that specific version, then use that as a reference to identify compatible version of your other tools.

For example, I started using `buildroot-2012.02` (released in february 2012) and `linux-2.6.35`, which is pretty old (released in october 2010). The `uClibc-0.9.33.x` test suit wouldn't even start.

3.2. Old kernel conflicts

As you may know, the Linux Kernel possesses a very long list of customizable features. Some configuration, though they compiled perfectly, could not run on my board; the kernel seemed to crash right after the boot, or after its decompression.

The problem was due to a conflict between two kernel options: `tickless system` and `high resolution timer support`. I found that the kernel could run without them, or with one of them; just one, though, and not both.

For my purpouses, I choose to mantain support for a better timer, sacrificing the power-saving tickless system.

Note that this issue disappeared using later versions.

3.3. Missing toolchain support

There is a number of building/compiling issue related to missing support by the toolchain, missing libraries, unsatisfied dependeciens etcetera.

You need to consider the needs of the packets you want to compile *BEFORE* you create or get your toolchain. The most common are support for `large files`, `IPv6`, `WCHAR`, runtime support for `Objective-C` or `C++`, support for shared libraries or dynamic modules.

3.4. Missing dtb file

If there's something BuiltRoot can't do is creating the device tree. Although you may be lucky enough to find it somewhere (in the `images/` folder on the CDROM you found in the box, for example) your like will be simple; but when this is not the case, know that the Linux kernel sources can.

First, you have to locate the kernel source folder: on my BuildRoot, it was `output/toolchain/linux-<version>/`. Here, you just need to

```
$ make mpc8313erdb.dtb
```

and be prepared for trouble. Should **make** being unable to compile the device tree, that coulb be due to the missing **DTC**. To build this application, you shall

```
$ make scripts/
```

and get prepared for more trouble. If this operation doesn't work, you may need to reconfigure the kernel.

Also remember that specifying the target architecture is important, as environment variable (**$ export ARCH=powerpc** in your current shell) or as **make** variable (**$ make ARCH=powerpc <target>**). See also Section 4.2, “Kernel”.

4. The hard way

Maybe you're working on an obscure unsupported piece of hardware; maybe `BuildRoot` is giving you trouble; maybe you're insane.

To get yourself a working Linux installation, you can always do it by yourself, one piece at a time. I did. Here's a step-by-step guide.

But before that, let me tell you this: there is *absolutely no need* to follow this path unless easier solutions are already working.

4.1. Toolchain

Crosscompiling for any architecture is a pain; it is so because, in order to do that, you need a fairly large gathering of applications and libraries: putting that together is the real problem.

Fortunately, you are not alone; many people around dealt with this very problem, and some of them created tools to help others; one is Dan Kegel (see both http://kegel.com/crosstool/ and http://crosstool-ng.org/), others are Erik Andersen and the BuildRoot team (see http://buildroot.uclibc.org).

With a little effort, you should be able to use one of this two application to build a toolchain for your target. They both work the same way: you download the source, decompress them somewhere, than you configure them by calling

```
$ make menuconfig
```

and select what you need. Remember that you must specify your target architecture first, than tune the settings for the needs of the packages you want to install, shall those need a language runtime, the float-operations support, or specific versions of specific libraries.

Whichever tool you choose, it shall compile (after a reasonably long time – remember the build options) a toolchain. The toolchain itself isn't an application; it is a collection of many application (compiler, linker, archiver and many more) plus kernel headers and a number of libraries.

To effectively use your toolchain to compile any sourcecode, you shall link it properly. How you do this depends slightly on the code you're compiling. Many well-managed packages (BusyBox and the Linux kernel, for example) are crosstool-ready.

You shall configure some parameters before starting the compilation. Depending on the application (consult the `README` file) you may have to specify

```
$ export ARCH=powerpc
$ export CROSS_COMPILE=\
    powerpc-linux-
$ export CC=\
    /path/to/toolchain/bin/gcc
```

or something similar. Anyhow, you have to include the toolchain `bin/` folder in your path

```
$ export PATH=\
    /path/to/toolchain/bin:$PATH
```

4.2. Kernel

Compiling the Linux kernel is a fairly simple task; the tricky part is compiling a kernel that works.

Once you downloaded the kernel source and properly linked your toolchain, the tough part is the configuration. Remember that, unless your target is a pretty obscure hardware, the Linux kernel should already have a default configuration for your architecture; you just need to find it.

```
$ export ARCH=powerpc
$ make defconfig
$ make 83xx/mpc8313_rdb_defconfig
```

If you have trouble identifying the correct name for your configfile, you can take a look in the `arch/$ARCH/configs/` folder. After that, tune the kernel for your needs with

```
$ make menuconfig
```

and be prepared to try a few alternatives before getting a working kernel. I had, for example, to remove the `tickless system` in order to have the `high resolution timer support` working. The problem is that you won't be sure if your kernel is ready (even when it compiles correctly) until you run it; so keep backups of the configuration you built and get ready to remake the compilation.

4.3. BusyBox

Even when you have a kernel ready to run, your system won't go anywhere without a working filesystem with something on it.

If you are reading this document, I guess you don't need a fancy installation, you rather need something that works – preferibly fast – and does not eat space up. BusyBox is what you need.

To set, configure and build BusyBox, follow the same procedure as you did for the kernel: download, decompress and then

```
$ make menuconfig
```

Now, configuring BusyBox can take long: there are a few starting configuration, and only two of them are somehow useful: `defconfing` and `allnoconfig`.

Both of them are a starting point, which one you shall prefer to the other is matter of space: `defconfig` gives you all the reasonable selected content (the resulting BusyBox can be pretty big, over `2MB`, which may be too much in some cases); the other one, instead, gives you an empty configuration.

You shall add everything you need and remove anything in excess.

For example, I wanted a light installation, so I started from the empty one and then added the basic `binutils` (commands like **ls**, **mv**, **cd** – remember, if you don't include them, you won't have them), some system utils like **pgrep** and **netcat**.

When you do this, remember that you have to select *each command* you will use; if you don't, if you forget to add something, you will have to recompile BusyBox and repeat the install process.

There are also some other options you may activate: BusyBox can be compiled as a single stand-alone binary (to save space) or as a normal (read dynamically-linked) binary (this require that you also install all the needed libraries, but it should minimize the system RAM footprint (when you run many similar processes). The choice is yours.

4.4. Virtual disk

Once you have a pretty BusyBox configuration, you have to install it on a suitable filesystem. This is the filesystem you'll have to compress and then reformat as a U-Boot image; therefore, it can't be just an ordinary folder.

What you need to do is create a virtual disk. First, you shall prepare a regular file to emulate the disk, then this file shall be formatted as a filesystem, which you shall mount; after the install, you can compress that file and make an image out of it.

```
$ dd if=/dev/zero \
    bs=1K count=16000 \
    of=<disk>
$ /sbin/mkfs.ext2 <disk> \
    -F -m 0 32000
```

With this two commands, you create a zero-filled file, `16MB` big; then, you format it as a `extend 2` filesystem with `32000` blocks. I choose to specify this many blocks to force the filesystem to have many *inodes*: my BusyBox installation contains many many small files; shall this disk run out of inodes, it couldn't store any more content, even if the storage isn't full.

I advice you to do this when you work in restriction: having enough memory, you can simply create a big disk.

4.5. Install

Now that you have a virtual disk, you can install BusyBox and then build a fully operational OS.

First, mount your disk; then, install BusyBox

```
$ make \
    CONFIG_PREFIX=<path/to/disk> \
    install
```

where `<path/to/disk>` is the mountpoint of you disk; you may also set that value as part of your configuration; but know that if you don't, the default value is the `_install/` folder inside BusyBox, which is not very useful.

Do not attempt to perform your install there, and then copy it to the disk: there are many links (soft or hard, depending on your configuration) that won't work on the final image once you move them.

Now that your disk as BusyBox install, don't cheer yet: that is not enough, and even if the kernel will boot, that system won't respond as you hope.

First, there a bunch of folder you need to create (empty): they are `/dev`, `/etc` (maybe `/etc/init.d`), `/proc`, `/sys` and maybe `/usr`.

Then, if there are any needed libraries, you shall copy them all in the `/lib` folder. There are many more files that should be copied on the disk at this point: configuration files, man pages, documentation, maybe even user data. If you did your work well, you can find something called `sysroot` as part of your toolchain: the content of that folder is, probably, something you want to add to the disk.

This is almost it: check the content of the `/etc` folder on the disk. No matter what, it should contain *at least* two files: `/etc/inittab` and `/etc/init.d/rcS`. Those are required for Linux to work correctly, the kernel will *PANIC* if they're missing.

Now, about their content: you're not bound to specific rules, but there are some lines you shall add. For `/etc/inittab`:

```
::sysinit:/etc/init.d/rcS
::askfirst:-/bin/sh
```

which is pretty simple. Check whichever Linux distribution you like the most to find a valid alternative; but, remember, that this file will be parsed by the BusyBox **init**, which is slightly less powerful than the original. Now, for the `/etc/init.d/rcS`:

```
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
/sbin/mdev -s
```

which is, one more time, pretty simple. As this is Bourne Shell script, you can do what you want; check any Linux distribution's `/etc/init.d/` (or `/etc/rc.d`, also) to gain inspiration.

Finally, **sync** your filesystem, **umount** it, compress it and convert it to a U-Boot image as seen in Section 2.5, “Image conversion”.

5. Software sources and version numbers

I have experienced many problems, including odd compilation problems, which are due to incompatibility between different versions of different tools.

The golden rule is to use the latest stable version available. If you are bound to use a particular (old) version of something, I suggest you track the time when your tool was released and try to use other components that are about the same age.

Here's a list of what I used for my tests.

Table 1. Tested version numbers

Component	Source URL	Tested with versions
Das U-Boot	http://sourceforge.net/projects/u-boot	1.1.6
Buildroot	http://buildroot.uclibc.org/	2012.02, 2012.05
Linux Kernel	http://kernel.org/	2.6.36-4, 3.2.7, 3.3.7
BusyBox	http://busybox.net/	1.1.0, 1.18.5, 1.19.1, 1.20.1
uClibc	http://uclibc.org	0.9.31, 0.9.32, 0.9.33
binutils	http://ftp.gnu.org/gnu/binutils	2.21.1, 2.22
gcc	http://gcc.gnu.org/	3.4.3, 4.5.3, 4.7