bbfs 1.3

Building a filesystem based on busybox alone
January 2009

Alessandro Rubini (rubini@gnudd.com)

bbfs

This package describes my own choice of embedded distribution. It describes how to set-up a busybox-only filesystem and how to customize it for the specific target machine.

Please note I am not claiming this package is optimal. I may have overlooked some important details or I might be duplicating some other work I ignore (the world is too big to know everything), but I publish it in the hope to be useful to someone. Actually, while documenting it I found a few places far less than professional, but it might be useful anyways. Any feedback is welcome.

1 The embedded distribution

A GNU/Linux distribution is a big collection of packages. Several packages are inter-dependent, so their version numbers must be matched in strange ways, so the dependency database is a serious work for the distribution to maintain, and huge data on each and every installed host.

The embedded distribution, on the other hand, is usually just the minimal stuff that is needed to bring up the system and fire the relevant application. At least, this is what happens in the industrial world. While *uclinux-dist*, *openwrt* and other embedded distributions are the right choice for some class of devices, they are sometimes overkill as well.

Therefore, my personal choice for embedded systems that are meant to be used in industrial environments is building a busybox-only filesystem, offering support for adding the relevant application and data. When the firmware is build from *busybox* alone, you minimize the effort in upgrading your embedded systems 5 years from now, since the only dependency you get is with *busybox*, a package which is very unlikely to go unmaintained or to introduce unexpected incompatibilities.

The following table describes the components of a full-featured embedded distributions and how they are supplied in this package.

The compiler or cross-compiler
: Some distributions include the cross-compiler to be used to compile them. I personally prefer to either build my compiler with *crosstool* or use the tool-chain in *ELDK* (or the one in my distribution for native compilation).

The boot-loader
The Linux kernel
: Both must be customized for the target system, so I'd better port and compile them separately. They don't need to be included in a small embedded distribution.

The C library and the dynamic loader
: Unless everything is statically compiled, the C library (*libc* and all annexes) must be present in the target filesystem. Since the compiler already includes one, I copy that very library, with the dynamic loader. You can use a compiler based on *glibc* or *uclibc*, the right one will be copied to the target.

/sbin/init
/bin/sh
Unix tools
Network services
: All of these are included in *busybox*, so I use them in my target filesystem.

Other tools
: Some simple but useful tools are missing from *busybox*; most notably *devmem2* and *flash_eraseall*. I compile them separately from a directory with a trivial *Makefile*, and the procedure is integrated in the build script. While the latest *busybox* includes *devmem* (with the same syntax as the *devmem2* I offer, the tool is preserved in this package because I still use it with older releases of *busybox* as well.

Users and passwords
: Since the system is minimal, there are no users nor passwords. Telnet is enabled and the *root* user has an empty password. You should change `/etc/passwd` (for example using the *passwd* command and copying the resulting file) for any serious of half-serious use.

Other packages and the application
: Sometimes other packages are needed, like *ssh*, *bash* or *strace*. I compile them separately according to specific needs (just like the main embedded application). If the number of such packages is big, then you should switch to a real embedded distribution and abandon *bbfs*.

2 Contents of a root filesystem

Once the kernel mounts the root filesystem, it looks for `/dev/console` and `/sbin/init`. Everything is then handled by the *init* process (whose *stdin*, *stdout* and *stderr* are connected to the console). Unless your *init* is a self-contained application doing its I/O without accessing device files, you'll need quite some additional files in the root filesystem.

This is the layout of the root filesystem built by *bbfs*. I tend to mount this filesystem as an *initramfs* or *initrd* on those system that still run 2.4.

`/dev`
: The device directory I use is static. Tools like *udev* are often overkill in embedded systems. The contents of `/dev` are extracted from a user-supplied archive or copied from a user-supplied directory. This package includes an example directory with the device files I use.

`/etc`
: In `/etc` you need a few configuration files for *busybox* and possibly other tools you include in the filesystem. In *bbfs* `/etc` is either extracted from user-supplied archive or copied from a user-supplied directory. This package includes an example directory with working `/etc` files. `/etc/rc` is specially important, as it's what drives the system up (according to the `/etc/inittab` I use). `/etc/rc` defines an empty password for the administrator; you are expected to change it.

`/bin`
`/sbin`
: These directories include the Unix commands, and are filled with symbolic links to *busybox*. While I prefer hard links, I had some problems with them, so *bbfs* uses symbolic links. In addition to *busybox*, there are small tools, like *devmem2*.

`/lib`
: The directory includes shared library, copied from the compiler. I chose to only copy the libraries I use, you can edit the script to fix it.

`/tmp`
`/var`
`/mnt`

These directories are present but they are not filled.

`/usr`
`/var`

These directories are used as mount points by `/etc/rc`.

`/proc`
`/sys`

These directories are used as mount points for *procfs* and *sysfs*.

If done well, the *busybox-fs* can be used unmodified on several different computers, provided their `/usr` tree is customized for the specific application. I personally use the same images on several different ARM systems.

3 Mounting /usr and /opt

Usually, in my embedded systems, I have a flash partition or two where additional programs can be installed. Thus, *bash* can be `/usr/bin/bash` and *Xfbdev* can be `/opt/xorg/bin/Xfbdev`. So the root filesystem lives in *initramfs* but it mounts the other partitions at boot time.

The file `/etc/rc`, as distributed, mounts both `/usr` and `/opt`. The former from `/dev/mtdblock5` and the latter from `/dev/mtdblock6`. This matches the following flash layout; if your setup is different (or if you don't mount `/opt`), you'll have to customize `/etc/rc`:

- mtd0: boot-loader
- mtd1: boot-loader configuration
- mtd2: application configuration and status
- mtd3: kernel
- mtd4: bbfs.cpio.gz
- mtd5: /usr
- mtd6: /opt

In order to mount the two partitions the `/etc/rc` script includes the device name and mount options in its own text, but it will use the `USR` and `OPT` environment variables if defined.

You can set environment variables in the shell running `/etc/rc` by passing them in the kernel command line. When the kernel boots, it passes all command-line arguments it doesn't know to the *init* process: arguments that include the `=` character are set in the environment and the rest build the command line.

Therefore with this `/etc/rc` you can change what is mounted on `/usr` and `/var` each time you boot, without modifying the root filesystem. To avoid quoting hell, `/etc/rc` turns underscores into spaces. For example, with the following argument in the kernel command line you can mount `/usr` over the network during development:

```
USR=-t_nfs_-o_nolock_192.168.16.1:/opt/root/usr-target
```

This will result in the following message being printed on the console:

```
Mounting -t nfs -o nolock 192.168.16.1:/opt/root/usr-target into /usr
```

and in the following command being invoked:

```
mount -t nfs -o nolock 192.168.16.1:/opt/root/usr-target /usr
```

4 build-busybox-fs

The `build-busybox-fs` script in this package is what builds the root filesystem in the way described above. It allows some external configuration, but might need to edit it directly to better fit your needs (for example, to change the *busybox* configuration.

This chapter describe how to use the script and how to customize it. Finally, it describes the script in detail by quoting its code.

4.1 Building the base filesystem

The core of the filesystem is made up of *busybox* and the system libraries. The script is therefore run from within the *busybox* source tree, already uncompressed.

You need to set two environment variables first: `CROSS_COMPILE` (the default with version 1.2 of this package is native compilation, while I mainly compile fomr ARM systems) and `DOCDIR`. The former is the cross-compilation prefix and the latter is the directory where additional files (and, usually, documentation) are found. This package includes everything that is needed to build a filesystem, therefore you can use it as your own `DOCDIR`:

```
export DOCDIR=/path/to/bbfs-1.3
export CROSS_COMPILE=arm-linux-
cd /path/to/busybox-1.13.1
$DOCDIR/build-busybox-fs
```

After those commands have completed, you'll find the `_install` subdirectory hosting the whole tree, the file `ramdisk.cpio.gz` (already suitable for *initramfs*) and the compilation log in `bbb.log`.

In future revisions of the script I will probably place the output files outside of the *busybox* source tree.

Please note that you don't need to be `root` to compile *bbfs*, but the script uses `sudo` to gain superuser privileges when needed (to create devices and change file ownerships). So you will be prompted for your own password. If you want to run under `fakeroot` instead, you need to edit the script and avoid calling *sudo* (this ought to be fixed, actually).

If you need to download *busybox* as well, you can run the `README` file of this package, which is actually a shell script that runs the command above and a little more, with explanation.

4.2 Customizing your tree

The `build-busybox-fs` script performs a few choices. It currently behaves according to my personal preferences but there's always room for customization, either by editing the script or by setting environment variables. The following table describes my choices and how to change them:

busybox configuration
: The script applies the default configuration and then changes some of the choices. For example, I remove `fdisk` and enable NFS-mount. By only changing a few options, I can use the same script with different versions of *busybox*, and I fear little incompatibility with future versions. To change this, you need to edit the script (look for "activate" in the code).

System libraries
: I copy system libraries from the compiler itself. The list of libraries being copied over reflects my own needs, but the script is designed to work with bot *glibc* and *uclibc*. To change the list of libraries you'll need to edit the script (look for `LIBS=` in the code).

/dev

The device special files are copied from the `dev` subdirectory of `DOCDIR`, if it exist. You can override it by setting `DEVDIR` in your environment. The devices are also extracted from `$DOCDIR/dev.tar.gz`, if it exists. Whatever the source, any file names starting with *at* (`@`) will be replaced by devices according to the *genromfs* convention. See the `/dev` directory in this package for an example. To customize, set `DEVDIR` or `DOCDIR` to your own place.

/etc

The configuration files are copied from the `etc` subdirectory of `DOCDIR`, if it exist. You can override it by setting `ETCDIR` in your environment. The files are also extracted from `$DOCDIR/etc.tar.gz`, if it exists. To customize, set `ETCDIR` or `DOCDIR` as needed.

Additional tools

If the environment variable `BINDIR` points to a directory or if `bin` exists under `DOCDIR`, a compilation is fired in there and any executable file is copied to `bin` in the target filesystem. See `bin/` in this package for an example. To customize, set your own `BINDIR` or `DOCDIR`.

Adding custom files to the tree

The script supports overlaying a tree over the target filesystem. This means you can add your own application and data file in one shot, provided it already exists in the build host. If the variable `OVLDIR` points to a directory, or `rootfs-overlay` exists within `DOCDIR`, then those file are overlaid to the target filesystem. There are no overlay files in this package. To customize, set `OVLDIR` or `DOCDIR`.

Initramfs or initrd

The script creates `ramdisk.cpio.gz` for use with *initramfs*. If you need to create *initrd* too, you can set `BBFS_INITRD` to a non-empty string in your environment. Building *initrd* requires a few more calls to *sudo* and creates `ramdisk.gz`. I use it rarely, so filesystem size is hardwired to 4MB in the distributed script.

4.3 Detailed look at the script

This section includes parts of the script itself to explain it. You are expected to edit it (I do it for specific projects), as this is not a fool-proof all-automatic distribution, so it may fail and may need to be fixed. If you are fluent in *sh* and Unix, though, you may avoid reading this section and read the script directly.

Firs of all, I ensure both `DOCDIR` and `CROSS_COMPILE` are set, to catch common errors. Native compilation is a special case for me, and `CROSS_COMPILE` can be an empty string in that case. Thus, "`test -z "$n"`" won't work and *grep* is used instead.

```
for n in DOCDIR CROSS_COMPILE; do
    env | grep "^$n=" && continue
    echo "Environment variable $n is not set" >&2
    exit 1
done
```

Then the script sets the directories to copy stuff from. You can set them individually, otherwise they live under `DOCDIR`:

```
if test -z "$ETCDIR"; then ETCDIR=$DOCDIR/etc; fi
if test -z "$DEVDIR"; then DEVDIR=$DOCDIR/dev; fi
if test -z "$BINDIR"; then BINDIR=$DOCDIR/bin; fi
if test -z "$OVLDIR"; then OVLDIR=$DOCDIR/rootfs-overlay; fi
```

To compile *busybox*, the default configuration is chosen, then the `.config` file is edited in-line with a function called `activate` (not shown here), and the resulting configuration is re-fed to

the system. Several configuration options are changed, only a few are shown here. Please note that I disable *fdisk*, *fsck* and other stuff that may be important on an x86 disk. That's because I use this mainly on ARM, PowerPC and MIPS systems.

```
make defconfig
activate true CONFIG_INSTALL_NO_USR
activate false CONFIG_FEATURE_SHADOWPASSWDS
activate false CONFIG_LOCALE_SUPPORT
make silentoldconfig
make && make install
```

At this point, *busybox* is installed in `_install` with no `/usr` subdirectory. So missing pieces are created. We save the installation place in `TARGET`.

```
cd _install
export TARGET=$(/bin/pwd)
mkdir tmp mnt etc var proc sys lib dev usr opt
```

`/etc` is extracted from an archive in `DOCDIR` (not shown here) or copied from `ETCDIR`:

```
if [ -d $ETCDIR ]; then
    echo "Populating etc from $ETCDIR"
    cp -a $ETCDIR/* ./etc
fi
```

`/dev` is filled in the same way: either an archive or a directory to copy from. In this case *sudo* is used, so device files are actually created.

```
if [ -d $DEVDIR ]; then
    echo "Populating dev from $DEVDIR"
    sudo cp -a $DEVDIR/* ./dev
fi
```

To allow using a non-privileged `DOCDIR`, the *genromfs* conventions is used. Thus, any file called `@name,type,major,minor` will be turned in a device node at this point:

```
find dev -name @* -print | while read f; do
    d=$(dirname $f)
    b=$(basename $f)
    sudo mknod $d/$(echo $b | tr -d @ | tr ',' ' ')
    sudo rm $f
done
```

To fill `/lib`, we ask the compiler where the libraries live (the real script is a little longer than this). So we copy stuff from there. A glob expression is used so both *glibc* and *uclibc* can be copied over (according to the compiler being used). Similar code is used for the dynamic loader, but is not included here.

```
LIBDIR=$(dirname $(${CROSS_COMPILE}gcc -print-file-name=libc.a))
echo "Populating lib from $LIBDIR"
LIBS="libc.so.[0-6] libm.so.[0-6] libcrypt.so.[0-1] librt.so.[0-2]"
cd $LIBDIR
for n in $LIBS; do
       [ -L $n ] && cp -a $n $(readlink $n) $TARGET/lib
done
```

Finally, custom stuff is added. A `BINDIR` directory is compiled (see this package for a working example); moreover, an overlay directory is copied over the target filesystem, so you can have anything you want, provided you compiled it separately.

```
if [ -d $BINDIR ]; then
    echo "Compiling in $BINDIR"
    make -C $BINDIR || exit 1
    cp $(find $BINDIR -type f -perm -1) bin
fi
if test -d "$OVLDIR"; then
    echo "Copying $OVLDIR to target tree"
    cp -va $OVLDIR/* .
fi
```

The filesystem is ready, we only need to *chown* it to root and make the bootable archive. If `BBFS_INITRD` is set, an ext2 filesystem is build (I might use *genext2fs* but I use the old way with *sudo*); the *cpio* archive is always built, using *mkinitramfs* from `DOCDIR`.

```
sudo chown -R 0.0 .
if [ ! -z "$BBFS_INITRD" ]; then
   [.... boring, not shown ....]
fi
$DOCDIR/mkinitramfs . ../ramdisk.cpio.gz
```

Table of Contents