White Rabbit Switch: Developer's Manual

Information about software in the White Rabbit switch, for developers and advanced users
October 2014 (wr-switch-sw-v4.1)

Alessandro Rubini, Benoit Rat, Federico Vaga et al.

Table of Contents

Introduction

The White Rabbit switch (or WRS) is a major component of the White Rabbit (WR) network. Like any modern managed switch, the WRS includes a CPU with its own operating system.

This manual is for people rebuilding or modifying WRS software. It explains how to rebuild the whole software stack from source and how is the switch itself designed.

1 WRS Documentation

Up to and including release 4.0 of WRS software this manual was the only official documentation item; now that the device is mature and the deployed base increases, we reorganized documentation.

1.1 The Official Manuals

This is the current set of manuals that accompany the WRS:

- *White Rabbit Switch: Startup Guide*: hardware installation instructions. This manual is provided by the manufacturer: it describes handling measures, the external connectors, hardware features and the initial bring-up of the device.
- *White Rabbit Switch: User's Manual*: documentation about configuring the WRS, at software level. This guide is maintained by software developers. The manual describes configuration in a deployed network, either as a standalone device or as network-booted equipment. The guide also describes how to upgrade the switch, because we'll release new official firmware images over time, as new features are implemented.
- *White Rabbit Switch: Developer's Manual*: it describes the build procedure and how internals work; use of scripts and WRS-specific executables and so on. The manual is by developers and for developers. This is the document to check if you need to customize your *wrs* rebuild software from new repository commits that are not an official release point, or just install your *wrs* with custom configuration values.

The official PDF copy of the three manuals at each release is published in The *files* tab of the software project in `ohwr.org`: (`http://www.ohwr.org/projects/wr-switch-sw/files`). This doesn't apply to release 4.0 and earlier.

The source form of all three manuals is maintained in `wr-switch-sw/doc`. Within the repository, both the *User's Manual* and the *Developer's Manual* are always tracking the software commits, while the *Startup Guide* may not be authoritative because it is bound to device shipping rather than software development.

1.2 Supported Hardware Versions

This document applies to versions 3.3 and 3.4 of the WRS device.

Very few specimens of `wrs` 3.0 though 3.2 were manufactured; if you are the owner of one of them, please refer to version 3.3 of the *wrs-build* document, that includes appendixes about using older versions. As usual, it is in the *files* tab of `ohwr.org`.

V1 and V2 were development items, never shipped.

2 Building WRS Software

2.1 Overview

The scripts build over previous work by Tomasz Wlostowski, who first made the whole thing work and stick together – a serious result from serious efforts, I am really amazed by his achievements.

The purpose of the build-script rewrite is achieving the following targets:

- One-command build. The non-technical user should be able to rebuild the whole software package with a single command. This includes the IPL and boot-loader even though they are expected to be pre-installed in the switch with no real need for upgrading.
- Sub-package separation. Users and developers should be able to rebuild each sub-package by itself. Sub-packages are the kernel, *buildroot*, libraries and so on. If you have a problem (or a customization you need on one sub-package), you should be able to work on the specific part ignoring the whole as much as possible.
- Documentation. The steps are documented as much as possible, because mishaps do happen, and you should easily understand where the problem is.
- Avoid redundant downloads. People with non-mainstream network connections would rather avoid downloading the same package over and over. Thus, a centralized download directory is defined where all external packages are retrieved. Even if you "make distclean" in the build scripts you will not need re-get everything from the network. In a similar mood, people who already have a local copy of the big packages (kernel, barebox, white-rabbit svn) will not need to re-download not even the first time they build the WRS software.

After release 3.3, we decided to add *Kconfig* support. This means that the first build step is expected to be "`make menuconfig`", like it happens for the kernel. The default configuration is selected by default when one of the build scripts is run, so the procedure for the final user is the same as for v3.3 and earlier. A build with a non-default configuration, however, is not considered as "supported", and *Kconfig* is there mainly to help developers try new packages and setups without changing the repository or introducing problems for other users. For some more information about *Kconfig* in this package, see the WRS *User's Manual*.

The build system is set up as a mix of scripts and makefiles. Every sub-package is built by its own script and/or Makefile, and configuration is passed over through environment variables. The top-level build script sets all environment variables, while keeping defaults from your preexisting environment – so you can override anything even when rebuilding it all from scratch.

2.1.1 Portability

The scripts in their current status are not expected to be very portable. I am sure a number of *bashisms* exist, and I did no effort to even identify them. To relieve the user from possible pain, internally the name *bash* is used instead of *sh*, so things work in systems where the default shell is *dash*, provided *bash* is installed.

Similarly, the scripts are likely to fail if you use spaces in directory names; that is because not all uses of shell variables are properly quoted. I urge you to use directory names with no spaces in them, or to submit a patch to fix the scripts.

It should go without saying that the build environment is expected to be a native GNU/Linux system; success reports about other environments (e.g. cygwin) are welcome, possibly with associated patches.

2.1.2 Environment Variables

The scripts use a number of environment variables; you can pre-set them as you wish. If they are not pre-set, defaults apply as described below.

When building running the *build/wrs_build-all* script (whether you build everything or rebuild individual steps) the defaults are applied for each unset variable. Developers working under the hood will need to set the variables. Each sub-package complains if it needs variables that are not set in their environment.

The following variables are used in one or more parts of the scripts; let me restate, though, that sensible default values apply by default, so this list is mainly for your curiosity unless you are a developer.

They are listed in an order that seems logical to me, but may sound random to a different person, please forgive this. Most of the variables are prefixed with `WRS_` to make them easily identified in the overall mess of variables and command names (all scripts used here have a similar prefix for the same reason).

`WRS_BASE_DIR`
: The absolute pathname of the build directory (i.e., the `build/` subdirectory of `wr-switch-sw`). The variable is internally set to the directory name of the main script. Note that the script cannot be run from the same directory or from the wr-switch-sw project directory (i.e.: `./wrs_build-all ./build/wrs_build-all` are not allowed), you must call it from your output directory using a pathname to invoke it. This variable cannot be overridden in the main script, but must be pre-set if you run a sub-script to rebuild only part of the software suite.

`WRS_OUTPUT_DIR`
: The absolute pathname of the directory where output is placed. It defaults to the current directory whence you invoke the script (i.e., you can invoke `/path/to/wrs_build-all` to have all output in the current directory). Compilation happens in a *build* subdirectory of `WRS_OUTPUT_DIR`, done-markers are placed in a *_done* subdirectory and final images are placed in a *images* subdirectory.

`WRS_DOWNLOAD_DIR`
: The absolute pathname of the directory where downloaded files are placed. If unset it defaults to `$WRS_OUTPUT_DIR/downloads`, which is created if needed. By presetting this variable you can simply recursively delete the output directory to force a full rebuild, without the overhead of re-downloading everything. I personally preset this so it always points to the same place, even when I remove the whole output directory.

`WRS_HW_DIR`
: The absolute pathname of the directory where you build HDL, if any. If this variable is set, FPGA binaries will be copied from there instead of being extracted by the official archive on `owhr.org`. This is only used by HDL developers.

`CROSS_COMPILE`
: The variable is the usual cross-compilation prefix. For example, `arm-linux-` if you have *arm-linux-gcc* in your path, or a full pathname without the trailing `gcc`. If unset, it defaults to the compiler that *buildroot* self-builds. See Section 2.4.1 [The Compiler], page 6 for some more details.

Other variables are used internally in the script; since they are only useful to people working on the script itself, they are documented in place.

2.1.3 Downloading Files

Every downloaded file is saved to the `downloads` directory (`$WRS_DOWNLOAD_DIR` if set, or the default place `$WRS_OUPUT_DIR/downloads`). You should arrange not to remove that directory when you recompile over and over during development. I chose to make the first script download

everything, before starting any build, to help telling download errors from other issues. Also, after downloading is over you can work even without a network connection.

For each upstream archive needed, the following steps are performed:

- If the file exists in the download directory, the *md5sum* is checked; on success, nothing else is done.
- If the previous step fails, the file is retrieved from upstream.
- If the previous step fails, the file is downloaded from the buildroot web site.

The policy just described is implemented in *wrs_download*, in the file `scripts/wrs_functions`, based on `download-info` in the main build directory.

The messages of a download run are like the following ones:

```
2012-01-12 18:30:46: --- Downloading all files
2012-01-12 18:30:53: Retrieved at91bootstrap-3-3.0.tar.gz from upstream
2012-01-12 18:31:00: Retrieved buildroot-2011.11.tar.bz2 from upstream
[...]
2012-01-12 18:37:53: Retrieved uClibc-0.9.32.tar.bz2 from upstream
2012-01-12 18:37:56: Retrieved zlib-1.2.5.tar.bz2 from upstream
```

2.2 Building Procedure

If you just want to build stuff, with no concern about network downloads and without even knowing what is happening, just create a directory where you want the output to be generated and start compilation. Note that it takes around 3GB of storage.

Then run this (but please read more for a better command):

```
/path/to/wr-switch-sw/build/wrs_build-all
```

Note that progress messages are sent to *stderr*, so you may want to save *stdout* to a file, like this (again, it is recommended you read further for a better command):

```
/path/to/wr-switch-sw/build/wrs_build-all > logfile
```

Please note that there are also a number of warning messages being printed to *stderr*. It is a few hundred lines over the many minutes it takes to build *buildroot*, but you can safely ignore them, trusting the build process will complete successfully.

The progress messages look like what is shown here below. The log file will be rather big (6 or 7MB or so), as all the compilation steps are quite verbose.

The following example shows a run on a quad core system (18k bogoMips in total). If files had already been downloaded, the first step takes only a few seconds, as shown, to verify the checksums:

```
2014-06-21 17:01:57: --- Downloading all files
2014-06-21 17:02:02: --- Buildroot compiler and filesystem
2014-06-21 17:02:02: Uncompressing buildroot
2014-06-21 17:02:02: Patching buildroot
2014-06-21 17:02:02: Reconfiguring buildroot
2014-06-21 17:02:04: Compiling buildroot
2014-06-21 17:26:40: --- AT91Boot
2014-06-21 17:26:40: Patching AT91Boot
2014-06-21 17:26:40: Building AT91Boot
2014-06-21 17:26:41: --- Barebox
2014-06-21 17:26:44: Patching Barebox
2014-06-21 17:26:44: Building Barebox
2014-06-21 17:26:58: --- Linux kernel for switch
2014-06-21 17:29:15: --- Kernel modules from this package
2014-06-21 17:29:19: --- PTP daemon (ppsi repository as a submodule)
2014-06-21 17:29:26: --- User space tools
2014-06-21 17:29:33: --- Deploying FPGA firmware
2014-06-21 17:29:33: Using pre-built binaries from wrs-gw-v4.0-dev-20140328.tar.gz
2014-06-21 17:29:33: --- Wrapping filesystem
```

```
2014-06-21 17:29:46: --- Packing into wr-switch-sw-v4.0-20140621_binaries.tar
2014-06-21 17:29:46: Complete build succeeded, apparently
```

You may prefer to save *stderr* with *stdout* to the log file but still see the time-stamped messages from the scripts. In this case you can issue the following command – which is what I used to generate the terse output shown above:

```
/path/to/wr-switch-sw/build/wrs_build-all 2>&1 | tee logfile \
     | grep "^20..-..-.. ..:"
```

If you are lucky, everything completes by itself. The time taken depends on you CPU, disk and network speed. At the end you will find your final files in the `images` subdirectory,

If you are not too lucky, the build stops because you have found a bug in the build scripts; most likely because your setup differs from the ones we have been testing on.

In order to re-run the build from the beginning, please remove (or rename) the output directory and reissue the command. To only redo some steps, please see Section 2.3.3 [Rebuilding Parts], page 5.

2.3 Build Script Description

The `wrs_build-all` can be used to quickly build the White Rabbit Software as seen above. However it admits other functionalities detailed in this chapter. You might also want to check its embedded documentation using:

```
/path/to/wr-switch-sw/build/wrs_build-all --help
```

2.3.1 Release Package

By default, a complete compilation creates a "release" package, i.e. a *tar* archive of all files needed to flash a brand new WR Switch. The example above shows that the name is something like:

```
wr-switch-sw-v4.0-20140621_binaries.tar
```

In other words, we include both the tag name (from `git describe`) and a date. If the repository is not checked-out at a release (a "tag"), this will be apparent in the filename used for the output.

The "official" release package is available from `ohwr.org`, in the *Files* section of the `wr-switch-sw` project.

In any case, the file must be renamed to `wrs-firmware.tar` to be used at installation time. See Section 3.2 [Flashing Procedure], page 13 for details.

2.3.2 Build Time Configuration

Some details of the complete firmware archive depend on the values of active `Kconfig` variables. If no manual configuration is performed, what applies is `configs/wrs_release_defconfig`.

If you want to customize your configuration to install several switches pre-configured for your network, we suggest you rebuild the *firmware* archive after running `make menuconfig` to select your own values

2.3.3 Rebuilding Parts

When the main script succeeds in building one part (sub-package), it creates a file in the `build/_done` directory.

When you rebuild everything, steps for which the marker file exists are not rebuilt. To force rebuilding of one specific part, just remove the marker. Markers are numbered, reflecting the order of compilation steps, but they also have a name: names like `04-kernel` should be self-explicative.

To ease the rebuilding of a specific module a shortcut has been created in the `wrs_build-all` script. For example if you want to recompile the kernel alone you should execute.

```
/path/to/wr-switch-sw/build/wrs_build-all --step=04
```

You can list all compiled modules by calling

```
/path/to/wr-switch-sw/build/wrs_build-all --list
```

If you want to rebuild various modules at the same time, you should run something similar as:

```
/path/to/wr-switch-sw/build/wrs_build-all --step="5 7"
```

Please note that the final step (“*wrap root filesystem*”) is always performed, to ensure any changes are applied in the generated firmware file.

An alternative way to build parts, though a more difficult one, is running the individual script from within *build/scripts/*, after setting the proper environment variables.

2.3.4 Rebuilding From Scratch

If you have updated the repository with new modifications, you might want to check that you can rebuild from scratch. To clean your output directory by deleting all compiled modules (except downloaded files), just call:

```
/path/to/wr-switch-sw/build/wrs_build-all --clean
```

2.4 The Individual Build Steps

This chapter details the individual build steps, for the users that want to customize their switch in any way.

2.4.1 The Compiler

The predefined compiler used here is the one built by *buildroot*. The default configuration selects this choice. If you pre-set a different `CROSS_COMPILE` prefix in your environment, your own choice will be used by modifying the *buildroot* configuration file. Note, however, that not all cross-compilers will work (*buildroot* wants one that has been configured with `--sysroot` and it is quite unlikely yours has been).

In practice, you may want to set `CROSS_COMPILE` when you compile the boot loader and kernel by themselves, and avoid it when compiling the complete package.

2.4.2 Buildroot

The distribution being used here is *buildroot*. It is the first step being built, because it creates the cross-compiler it will use. This compiler is later used to compile all other software for the White Rabbit Switch.

The configuration for *buildroot* comes from `configs/buildroot/wrs_release_br2_config`. The configuration is then changed only if you pre-set your own `CROSS_COMPILE` variable. A different configuration can be chosen in the Kconfig interface, by running “`make menuconfig`” or equivalent, in the top-level source directory.

If you want to change the configuration, you can do so after the first build iteration: change directory to `build/buildroot-2011.11` and run `make menuconfig` (this the Buildroot configuration, not the one of wr-switch-sw). After making your choices, copy back the file `.config` to `configs/buildroot` in this package, so you can select it by running `make menuconfig` in wr-switch-sw.

Then, please run `configs/buildroot/RUNME` (without arguments) in order to remove your local pathnames in the configuration file; the file without local pathnames can be committed and published for other people to use.

You can also set `WRS_BUILDROOT_CONFIG` to the full pathname of your configuration file of choice (this used to be the only way to pass a custom configuration file). The file must be a copy of the `.config` after the `make menuconfig` step described above, within buildroot. Note that if

the variable is not pointing to a regular file it is ignored with a simple warning – rather than stopping the build procedure.

2.4.3 The IPL

The version of *at91bootstrap* being used in the switch as *Initial Program Loader* is version 3.3, download from `timesys.com/` (the full URL is in *build/download-info*. The patches we applied are in the directory *patches/at91boot/v3.3*, and we are piggy-backing on the Atmel evaluation board without even changing the board name):

```
0001-printf-added-files-from-pptp-unchanged.patch
0002-printf-fixes-and-addition-to-makefile.patch
0003-build-Add-gitversion-to-binary-and-a-script-to-compi.patch
0004-board-9g45ek-fix-ddr-config-for-WRS-V3.patch
0005-boot-disable-watchdog-asap-added-flip_leds-count-run.patch
0006-boot-Correct-crash-due-to-an-Atmel-bug-during-boot-w.patch
0007-gpios-Correct-FPGA-LED-problems-and-add-CPU-LEDs-FAN.patch
```

The script *wrs_build_at91boot* uncompresses, patches and builds, leaving `images/at91bootstrap.bin` after it is over. This file is the one to be loaded in the hardware.

If you build using a local *git* repository, we suggest to use `git am --whitespace=nowarn` because we have a number of white space errors, and we apologize for that.

Warning: with most distributions, this compilation step will print a scary message about memory corruption. The message is reporting a bug in the configuration program which has no actual effects and can be ignored. Maybe we will switch to another version in the future that doesn't show the bug, or to the newer *barebox* that obsoletes *at91boot*.

2.4.4 The Boot Loader

The switch uses *barebox* as a boot loader. We are running version 2014-04, with a few local patches and the chosen configuration file. Note that we are piggy-backing on the Ronetix PM9G45 board, out of laziness.

The patches are part of this package in *patches/barebox/v2014.04/* and the set is made up of the following ones:

```
0001-sam945-include-mtd-nand.h-in-device-file.patch
0002-arm-change-prompt-for-pm9263-wrs-piggy-backs-on-that.patch
0003-nand-wrs-our-nand-is-16-bit-connected-fix-accordingl.patch
0004-gpio-add-function-to-check-them.patch
0005-wrs-on-pm9g45-change-nand-setup.patch
0006-wrs-on-pm9g45-add-dataflash-initialization.patch
0007-barebox-add-MAC-addresses-to-environment.patch
0008-wrs-on-pm9g45-force-memory-to-64MB.patch
0009-pm9g45-init-for-wrs-move-environment-for-the-UBI-mov.patch
```

If you build using a local *git* repository, we suggest to use `git am --whitespace=nowarn` because we have a number of white space errors, and we apologize for that.

The *barebox* boot loader is organized as a small Unix-like environment, and its own configuration and scripts live in a small filesystem. To ease modification of such configuration and boot steps the build script copies over the configuration instead of patching it in the sources. You can thus edit the files you find in '`patches/barebox/v2014.04/env`' and rebuild your customized bootloader. The script that is executed at boot time is '`env/bin/init`' and as you see it calls the other ones. The menus included in the shipped configuration are described in the the WRS *User's Manual*.

Building *barebox* relies on a *Kconfig* setup, and the configuration file we use is '`patches/barebox/v2014.04/wrs3_defconfig`'. Again, this is copied over and not patched in (see the simple '`build/scripts/wrs_build_barebox`' for details).

After patching and copying over the files, the following commands build the boot loader using the cross-compiler built by *buildroot*. If you run these by hand you can use a different compiler (as shown):

```
export CROSS_COMPILE=/opt/arm-2010q1/bin/arm-none-eabi-
export ARCH=arm
make wrs3_defconfig
make
cp barebox.bin images/
```

To use the same compiler the scripts use, you need this setting (which is split in two lines with a local variable to fit the page with in documentation):

```
BR=${WRS_OUTPUT_DIR}/build/buildroot-2011.11
export CROSS_COMPILE=${BR}/output/host/usr/bin/arm-linux-
```

2.4.5 The Linux Kernel

The kernel is currently version 2.6.39, compiled from an uncompressed tar file (so not within a *git* repository). The upstream vanilla kernel is downloaded, then local patches are applied (they come from a *git* repository, but they are currently applied with a simple *patch* command).

The relevant patches are available in *patches/kernel/v2.6.39*, and are currently the following ones:

```
0001-wrs3-changes-to-g45ek.patch
0002-initramfs-stop-after-one-cpio-archive.patch
0003-at91-NR_IRQS-increase-by-64-to-fit-custom-muxes.patch
0004-irq-export-symbols-for-external-irq-controller.patch
0005-Change-Vbus-pin.patch
0006-arm-fiq-allow-modules-to-exploit-the-fiq-mechanism.patch
0007-mtd-nand-sam9g45-can-hwecc-like-9263.patch
0008-wrs3-use-correct-nand-partitioning.patch
0009-at91-udc-force-full-speed.patch
0010-sam9m10g45ek-for-wrs-new-partitioning.patch
0011-sam9m10g45ek-for-wrs-final-partitions-for-V4.1.patch
0012-sam9m10g45ek-for-wrs-more-relaxed-nand-timings.patch
```

The configuration we use to build the kernel is not a patch but a plain `.config` file, in the same directory as the patches, so you can change it easily, if needed. As an alternative, you can also set `WRS_KERNEL_CONFIG` to the full pathname of your configuration file of choice. The file must be a copy of the `.config` found in the main kernel directory, (for example the one left after the `make menuconfig` step). Note that if the `WRS_KERNEL_CONFIG` variable is not pointing to a regular file it is ignored with a simple warning, without stopping the build procedure.

The build scripts copy both *zImage* and all compiled kernel modules to the *images/* directory of the build place. This currently includes modules

2.4.6 Kernel Modules

In the next step the scripts compile modules that are part of this package. The step depends on the kernel being available in the build directory. The modules are then copied into the '`images/wr/lib/modules/`' subdirectory of the main build directory.

Please note that modules (and later user-space) are compiled in-place; ie. not in the output directory. The disadvantage is that your repository becomes dirty with output and intermediate files. The advantage is that any modification you make to the code is already in the repository for your to commit.

Currently, the package includes the following modules:

- *wr_vic.ko*: the interrupt controller for in-FPGA devices.

- *wr-nic.ko*: the network "card" driver for WR ports.
- *wr_rtu.ko*: the routing-table interface between the switching core and the associated user-space daemon.
- *wr_pstats.ko*: exports per-port statistics to /proc/sys.
- *at91_softpwm.ko*: a driver that generates a PWM signal for the fan.

2.4.7 PTPd

Configuration used to support two different PTP engines, but now we only support PPSi.

The code is hosted in its own repository; it is a *git* submodules in this package. The repository is hosted on `ohwr`, like others.

A plain *make* in `userspace/ppsi` will likely fail, because of missing environment variables.

Additionally, the script installs headers for the HAL and *libptpnetif*.

2.4.8 User Space Applications

The filesystem of the switch includes some user-space applications and tools. Some of the *tools* are actually used by the init scripts and some are just utilities for the developer.

The subdirectories in '`userspace`' include the various applications needed for the operation of the switch itself, as well as support libraries used by the applications themselves.

The main components are:

mini-rpc A remote procedure call library used by most other programs to exchange information among themselves or query the LM32 that is running on the FPGA.

libswitchhw
A series of utility functions to access the switch itself.

wrsw_hal The main application program for the White Rabbit Switch operation. The script installs the executable in `images/wr/bin`.

wrsw_rtud The daemon for the routing table unit, used for routing around data frames. It is installed in `images/wr/bin`.

The most important tools in '`userspace/tools`' are the following:

'`load-virtex`'
'`load-lm32`'
They load into the FPGA the gateware and the LM32 application. They are used by the init scripts of the Linux system. The LM32 loader can also change variables in the loaded binary, and read or write variables without stopping the running CPU. This is limited to 32-bit integer variaables, though. See the commit message for details.

'`mapper`'
'`wmapper`' The former reads from a file using *mmap* (usually you run it on */dev/mem*) and writes to *stdout*. The latter read from *stdin* and writes using *mmap*. They are classic tools distributed in the *Linux Device Drivers* examples since 1998.

'`com`' The program is a simple program for talking with serial ports.

'`wr_phytool`'
A tool to read and write PHY registers in the switch

'`wr_mon`' A simple monitor of White Rabbit status. It prints to *stdout* using the standard escape sequences for color output. The `-b` command line options removes color change (b/w).

'`wr_date`'
: The program can read or set the White Rabbit date. When setting, using "`wr_date set` *value*" assigns an arbitrary date, and "`wr_date set host`" passes the host time to White Rabbit. If the file `/etc/leap-seconds.list` exists, it is used to pass the TAI offset to the kernel, and to consider it in setting White Rabbit time to the current TAI value. The program is meant to prime the White Rabbit counter at boot time, and is run by `/etc/init.d/S70wr_date` – this script uses NTP to set host time as a first step, if `/wr/etc/wr_date.conf` exists and includes a line of the form `ntpserver 192.168.16.1`.

'`wrs_version`'
: Print information about the SW & HW version of the WRS. Please check the help message. See also Section 6.1 [DIP Switch HW version], page 20.

'`shw_ver`'
: A symbolic link to `wrs_version`, to be compatible with older versions that used this tool name. The name is inconsistent with anything else in the switch, so it is being replaced.

'`wrs_vlans`'
: The tool allows to configure and unconfigure the VLAN settings for each port and for the RTU daemon. The `--help` option lists all configuration items of the tool.

'`sdb-read`'
: The tool, copied from the `fpga-config-space` project, is documented in the next section,

Please note that to compile the applications and tools outside of the build scripts you need to specify both the kernel directory (`LINUX=`) and the cross-compiler to use (`CROSS_COMPILE=`).

2.4.9 sdb-read

[Note: this documentation section comes from the `ohwr` project called `fpga-config-space`.]

The *sdb-read* program can be used to access an *sdbfs* image stored in a disk file or an FPGA area in physical memory. It works both as *ls* (to list the files included in the image) and as *cat* (to print to its own *stdout* one of the files that live in the binary image).

The program can be used in three ways:

`sdb-read [options] <image-file>`
: This invocation lists the contents of the image. With `-l` the listing is *long*, including more information than the file name.

`sdb-read [options] <image-file> <filename>`
: When called with two arguments, the program prints to *stdout* the content of the named file, extracted from the image. Please note that if the file has been over-sized at creation time, the whole allocated data area is printed to standard output.

`sdb-read [options] <image-file> <hex-vendor>:<hex-device>`
: If the second argument is built as two hex numbers separated by a colon, then the program uses them as vendor-id and device-id to find the file. If more than one file have the same identifiers, the *first* of them is printed.

The following option flags are supported:

`-l`
: For listing, use *long* format. A *verbose* format will be added later.

`-e <entrypoint>`
: Specify the offset of the magic number in the image file.

`-m <size>@<addr>`
`-m <addr>+<size>`

Either form is used to specify a memory range. This is the preferred way to read from a memory-mapped area, like an FPGA memory space. Please note that in general you should not read a "file" in FPGA space, because this would mean read all device registers. The form "`<image-file> <filename>`" is thus discouraged for in-memory SDB trees (i.e. where `<image-file>` is `/dev/mem`).

`-r`

Register the device with a *read* method instead of the *data* pointer. In this way the tool can be used to test the library with either access method. If *mmap* fails on the file (e.g., it is a non-mappable device), *read* is used automatically, irrespective of `-r`.

2.4.10 VHDL and LM32 Binaries

The gateware binaries that are needed to run the FPGA are added to the target filesystem by the '`wrs_build_gateware`' script. If the variable `WRS_HW_DIR` is set, the script uses it to retrieve the binaries you just compiled (but the script is not compiling gateware).

If the variable is not set, the script extract a tar file downloaded from `ohwr.org` as part of the initial download step. The tar is currently called `wrs-gw-v4.0-20140807.tar.gz` and has been build from the `wr-switch-sw-v4.0` of the `wr-switch-hdl` repository. Please note that the repository uses *git* submodules, so it depends on other `ohwr` repositories too, which in turn have not been tagged because the submodule mechanism ensures you'll get the exact version you need.

The LM32 program is provided as a pre-compiled binary in `binaries/rt_cpu.bin`. The respective source code is the *wrpc-sw* package, because all WR installations run the same PLL software code and we chose to avoid duplication. Moreover, *wr-switch-sw* builds to not require an LM32 development environment.

If you need to rebuild the `rt_cpu.bin` file from source, to make your own modifications, you can run `make wr_switch_defconfig` in *wrpc-sw* and then `make`. Please checkout the `wr-switch-sw-v4.0` tag to get the exact commit.

2.4.11 The Complete Filesystem

The final step in building the switch software is wrapping together the filesystem for the switch, also making the archives and the *jffs2* image file.

The step of setting up the complete filesystem is performed by '`build/scripts/wrs_build_wraprootfs`'. The script does not leave a directory tree on disk because that would require administrator privileges. We think it is best not to call *sudo* from within build scripts, to respect our users' security concerns.

The script creates an archive for the whole filesystem, called `wrs-image.tar.gz`. It is used by the installation procedure and it is ready to be unpacked for NFS-Root. It is currently slightly less than 20MB of data.

To make your NFS-root place, you can run the following command in a newly-created empty directory:

```
tar xzf $WRS_OUTPUT_DIR/images/wrs-image.tar.gz
```

The archives include a number of device special files in *dev*. The pre-created devices come from *userspace/devices.tar.gz*. Note that the buildroot output directory, *build/buildroot-2011.11/output/target* does not include any device (and no white-rabbit specific files), so it cannot be used as a root filesystem by itself.

The content of the final filesystem comes from several sources:

- The *buildroot* output (from its own '`output/target/`').

- The switch-specific overlay (‘`userspace/roofs_override`’).
- The ‘`images/wr`’ and ‘`images/lib`’ trees, filled but the build scripts.
- The file ‘`userspace/devices.tar.gz`’
- The file ‘`$WRS_BASE_DIR/authorized_keys`’ if it exists.
- The `CONFIG_` items, used to pre-set configuration files.

The final step allows a predefined set of users to enter as system administrator without the need to type a password (which, anyways is empty by default). It is useful if you *scp* files in and out of the switch. In the shipped binaries no user is authorized, but the root password is still the empty string.

3 Flashing the Switch

This chapter describes the steps to install the WRS with the current firmware images. As far as hardware is concerned, this procedure describes the installation of the switch with a SCB version 3.3 or 3.4, and a MINI-BACKPLANE version 3.3. Older versions are not documented here any more (please get an older manual, if needed).

Note: Most likely you won’t need to reflash your WRS. Even if you rebuilt software from scratch, the “update” procedure that is available since August 2014 (release 4.0 of `wr-switch-sw`) allows complete replacement of the firmware image without physical access to the device.

3.1 USB connections

In order to perform the flashing operation easily, you should connect two *mini-USB* cables to the switch ports (Actually, one is enough, but the second one is useful to get more diagnostics while flashing).

The two back panel *mini-USB* sockets correspond to the serial port of the FPGA and the ARM. They are labeled **FPGA test** and **ARM test**. You should connect to “ARM test” to get diagnostics.

You can connect to it using minicom[1] like this:

```
minicom -D /dev/ttyUSB0 -b 115200
```

The port, however, will only appear on the PC after the switch is turned on, so you may want to delay this command.

The front panel USB connection, labeled as **management** USB port, communicates with the internal ROM of the CPU. It is the one used to perform the flashing procedure. No special program is needed, as the flashing tool will communicate with this port by itself.

You first need to set up the switch in "*Flashing mode*" to continue with the flashing procedure. To do so, you should turn on the power while pressing the **flash button** on the rear panel.

If the operation succeed you should see the message `bootROM` appears on the ARM UART. (You will likely not see it, because your *minicom* or equivalent can’t run before the switch is turned on).

You can also see the enumerated device in your own host:

```
$ lsusb | grep Atmel
Bus 001 Device 025: ID 03eb:6124 Atmel Corp. at91sam SAMBA bootloader
```

Finally, the kernel should also load the proper device driver, and you are expected to see `/dev/ttyACM0` or equivalent in your system. This is the device used for flashing.

If it is not the case, this means that the button used to disable the dataflash and enter "*Flashing mode*" is not working. You should contact support.

[1] You can use other programs for accessing serial ports, for example tinyserial

3.2 Flashing Procedure

If the update procedure, described in the *User's Manual* is not suitable for you (because, for example, you are the manufacturer), you need physical access to your WRS to flash it.

Unlike what happened with previous releases (up to the end of 2013), the filesystem of the switch can't fit in RAM memory during installation any more: the image is now downloaded through the network. Thus you need to following items to flash a switch:

- The USB cable connected to the front "management" USB port
- A Linux host connected as a master to this cable
- An Ethernet cable connected to the front "management" Ethernet port
- A DHCP server on your network, offering an IP address to the switch
- A TFTP server, exporting the file `wrs-firmware.tar`

The flashing procedure will use the *server address* reported by DHCP as IP address for the TFTP transfer.

Also, since release 4.1, you should not provide MAC addresses while flashing any more. The two MAC addresses are expected to be stored in *dataflash* by the manufacturer and not changed any more. If you upgrade your switch from a previous software version, please refer to the *User's Manual* for details.

The tool used to flash the firmware into the switch is the *USB-loader* we inherited from Atmel. The '`flash-wrs`' script is what you'll use to run the loader with appropriate parameters.

The script can be invoked in the following way to flash a *package* into the switch. The package is the `wrs-firmware.tar` file created by "`wrs_build-all`" (see Section 2.3.1 [Release Package], page 5).

Note: the release file must be renamed to `wrs-firmware.tar`, because the pathname is hard-wired in the installation procedure.

The command to flash is as follows:

```
/path/to/wr-switch-sw/build/flash-wrs -e wrs-firmware-<revision>.tar.gz
```

You can also flash the image you have built using the procedure described in Chapter 2 [Building WRS Software], page 2 by adding the tag `-b|--build`. To use this option you must call the script from the build directory, or define the `WRS_OUTPUT_DIR` environment variable.

```
/path/to/wr-switch-sw/build/flash-wrs -e -b
```

Please note that the "`-e`", which requires erasing the whole data flash, is almost mandatory because otherwise bits of your previous installation may leak into the newly-programmed one. Only on factory-new devices you can avoid this "`-e`" argument.

Note: White Rabbit switches are shipped with their preallocated MAC addresses, reported in a sticker on the back side of the switch; if re-flashing, you may want to use the same values.

Please remember that bits 0 and 1 of the first byte are special: if the first byte is odd, the MAC address is reserved for multicast transmission (the script doesn't check, and the kernel will refuse to enact such address). Bit 1 is set for "locally assigned" numbers: while official MAC addresses have bit 1 clear, if you choose your unofficial addresses you should set the bit.

If you don't configure a MAC address, a warning will be displayed and you can abort the procedure. If you don't abort the flashing procedure, the script will use default MAC addresses. Default MAC addresses are: 02:34:56:78:9A:BC for MAC1 (the Ethernet port of the ARM CPU) and 02:34:56:78:9A:00 for MAC2 (the base address for the 18 SFP ports).

```
tornado% ~/wip/wr-switch-sw/build/flash-wrs -e -b
flash-wrs: Working in /tmp/flash-wrs-1vV9z6
Warning: you did not set the MAC1 value; using "02:34:56:78:9A:BC"
Warning: you did not set the MAC2 value; using "02:34:56:78:9A:00"
```

```
flash-wrs: Waiting for at91sam SAMBA bootloader on usb.
    Please check the Managment USB cable is connected
    and keep pressed the Flash button while
    resetting/powering the switch.
.............. Ok
flash-wrs: I'm talking with the switch;
   please release the flash button and press Enter to start flashing:
```

If the script cannot find the Atmel programming interface on your USB bus, it prints a message and waits for the switch to be turned on in the proper way (with the button pressed or, for older hardware versions, the jumper plugged).

The process calls the flasher program twice (so you'll see the initialization strings two times) and takes slightly less than 5 minutes. the longest step is erasure of *DataFlash*: if run without `-e` the script takes 2 minutes.

This is the summary of the output you are expected to see, trimmed to save pages:

```
Initializing SAM-BA: CPU ID: 0x819b05a2

[...]

Initializing DDR...
loading applet isp-extram-at91sam9g45 at 0x00300000
Initializing DDR > Done

Initializing DataFlash...
loading applet isp-dataflash-at91sam9g45 at 0x00300000
Initializing DataFlash > Done!

Erasing DataFlash [... there is a long delay here ...] > DONE

Programming DataFlash...
  0x70000000 : at91bootstrap.bin ; size 0xf7c (3Kb)
DataFlash: Writing 3964 bytes at offset 0x0 buffer 70000000....ABCDEF OK
  0x70008400 : barebox.Fb09jx ; size 0x2f1bc (188Kb)
DataFlash: Writing 192956 bytes at offset 0x8400 buffer 70000000....ABCDEF OK
Programming DataFlash Done!!!
[...]

Initializing NandFlash...
loading applet isp-nandflash-at91sam9g45 at 0x00300000
Initializing NandFlash > Done!

Erasing NandFlash > DONE

[...]

Initializing DDR...
loading applet isp-extram-at91sam9g45 at 0x00300000
Initializing DDR > Done

Loading DDR...
          0x70000000 : /tmp/flash-wrs-tAqUAs/bb.new ; size 0x637b0 (397Kb)
          0x71000000 : /data/morgana/build-v4/images/zImage ; size 0x1afb44 (1726Kb)
          0x717ffff8 : /tmp/flash-wrs-tAqUAs/magicstr ; size 0x8 (0Kb)
          0x71800000 : /data/morgana/build-v4/images/wrs-initramfs.gz ; size 0x196f84 (1627Kb)
DDR: Writing 3842688 bytes at offset 0x0 buffer 70000000....ABCDEF
Closing...
Formatting UBI device... done
Getting tftp://192.168.16.1/wrs-firmware.tar ... done
Extracting filesystem...  done
```

The longest steps are erasing *dataflash* (it takes 2 minutes) and the last three steps: formatting, tftp and extraction; each of them takes around 1 minute.

Please note that the IP address used in the TFTP transfer depends on the DHCP handshake: the value above is what your developers use. The name `wrs-firmware.tar`, on the other hand, is hardwired: it matches the result of a firmware build, and the file name used within the archive of official binaries we ship at release time.

It is suggested to look at the CPU's serial port during programming, where you will see messages like these:

```
   -I- Statup: PMC_MCKR 1202 MCK = 100000000 command = 0
   -I- -- EXTRAM ISP Applet 2.9 --
   -I- -- AT91SAM9G45-EK
   [...]
   -I-     End of applet (command : 2 --- status : 0)
[...]
barebox 2014.04.0 #1 Tue Jun 24 09:05:43 CEST 2014
Board: White Rabbit Switch
[...]
Booting kernel for NAND flashing procedure
100Mbps full duplex link detected
DHCP client bound to address 192.168.16.246
[...]
Uncompressing Linux... done, booting the kernel.
[...]
/etc/init.d/wrs-boot-procedure: Running

Formatting UBI device... [...] done
UBI: attaching mtd1 to ubi0
UBI: physical eraseblock size:   131072 bytes (128 KiB)
UBI: logical eraseblock size:    129024 bytes
UBI: smallest flash I/O unit:    2048
UBI: sub-page size:              512
[...]
Getting tftp://192.168.16.1/wrs-firmware.tar ... done
UBIFS: mounted UBI device 0, volume 1, name "usr"
Extracting filesystem...  done
Requesting system reboot
Restarting system.
```

Please note, however, that many more messages flow, as formatting/mounting/umounting UBI devices is very verbose in the kernel. The sequence above is a summary of what happens at installation time.

3.2.1 Flash Script Description

The `flash-wrs` script can be used to quickly flash the White Rabbit switch as seen above. However it admits other functionalities detailed in this chapter. You might also want to check its embedded documentations using:

```
$ ./build/flash-wrs --help
Usage: ./build/flash-wrs [options] [<firmware>.tar.gz] [DEV]

MAC:  MAC address in hexadecimal seperated by ':' (i.e, AB:CD:EF:01:23:45)
<firmware>.tar.gz: Use the file in the firmware to flash the device
DEV:  The usb device (by default it is /dev/ttyACM0)
Options:
  -h|--help  Show this help message
  -m|--mode  can be: default (df and nf), df (dataflash),
 nf (nandflash), ddr (ddr memories).
  -g|--gateware  Select the gateware: 18p (18 ports, default), 8p (8 ports)
  -e   Completely erase the memory (Can erase your configuration)
  -b|--build  Use files that you have built in the WRS_OUTPUT_DIR
  -m1|--mac1  Default MAC address for the Ethernet port on board
  -m2|--mac2  Default base MAC address for the switch ports
```

The *DEV* is optional and the default is `/dev/ttyACM0`. If your system maps the Atmel ROM to a different device name, please pass the name on the command line. The script wants a full pathname starting with `/`.

If you want to flash the *at91boot.bin*, *barebox.bin*, *kernel* or *file-system* that you just built, you can just call the script from the build directory and use the `-b` option.

The official binaries for installation of version 4.0 of this package are available in the *files* tab of this project in `ohwr.org`. We don't provide a complete link here, but one is available in the list of downloaded files: `build/download-info`.

You can select a mode using the `-m|--mode` flag to choose to write in dataflash (df), nandflash (nf) or both (default). The memory mode is used to select a partial re-flashing; this is how the switch firmware is split among the two memories:

- dataflash: *at91boot* and *barebox* binaries
- nandflash: *kernel*, *initramfs* and /usr *file-system*

You can select which type of gateware you want to flash on your switch. Currently we only support LX240T (the current circuit doesn't fit in the LX130T any more). 8-port images are sometimes used for testing. And you can select this option like this:

```
$ ./build/flash-wrs --gateware 8p <...>
```

You can also erase the dataflash memory before writing your binaries; to do this add the option `-e`. There is no need to especially erase nand flash, because the installation procedure does the right thing with it in any case.

The script performs the following steps:

- It compiles the loader (*usb-loader/* subdir).
- It checks if the SAMBA bootloader is present.
- It picks the correct binaries according to the options.
- Optionally, it changes the default MAC addresses in *barebox* default environment, so you can use a different MAC for each switch.
- Optionally, it erases the dataflash memory.
- It places a magic string in RAM, to tell barebox we are installing
- It loads the kernel and filesytem to RAM and boots them
- It reads /dev/ttyACM0 to report the messages printed during flashing

The boot loader being booted finds the the magic string in memory, and changes the kernel command line to force installation-mode. The kernel and filesystem being booted in the switch are the same images for installation and run-time. (Releases before 4.0 built a special installation filesystem, but now the procedure is simplified).

3.2.2 Rebuilding Sam-ba Applets

The loader depends on code by the CPU vendor, which is very bad quality as typical in the field. If, by unlucky chance, you need to rebuild the applets, you need a specific version of the cross-compiler, and everything else will spit horrible errors.

A binary copy is uploaded in the *Files* sections of the OHWR project. The direct link is `http://www.ohwr.org/attachments/download/3138/cd-g__lite.tar.gz` (the name was `cd-g++lite.tar.gz`, but OHWR changed the `+` into `_`).

To build, you can run Benoit's script `usb-loader/samba_applets/isp-project/build.sh`.

4 Code layout in a production switch

This final chapter is a summary of how we used the two internal flash memories in the switch, when programmed with the official firmware binaries. It is meant for people who want to better understand the boot procedure and possibly customize stuff using higher-level tools, like erasing and rewriting flash-memory areas from Linux itself.

Unfortunately, the CPU is not able to boot from NAND memory directly, so the first steps of booting are performed from the *dataflash* device. Such an SPI memory is used to host the IPL (*at91boot*) and the executable code of the *barebox* boot loader. The user is not expected to ever erase this memory; if it happens, the system won't boot and you'll be forced to re-flash it entirely, which requires access to the back side of the switch..

NAND memory is used for user-data: tke boot loader configuration, the kernel and the filesystem.

This is how the memory is used:

```
0x0000.0000 - 0x0010.0000    Barebox-environment-backup
0x0010.0000 - 0x2000.0000    UBIfied-NAND
```

The first area is used to save the boot loader's configuration (if ever changed from the default and saved), and the second one is later split in UBI volumes. In the future we plan to move the barebox environment to dataflash memory.

The *dataflash* is partitioned too, and such partitioning is visible. (thus, you can replace `barebox.bin` by just writing it to the right device file). Overall, this is the content of '`/proc/mtd`' after boot:

```
dev:    size   erasesize  name
mtd0: 00100000 00020000 "Barebox-environment-backup"
mtd1: 1ff00000 00020000 "UBIfied-NAND"
mtd2: 00008400 00000420 "at91boot"
mtd3: 00084000 00000420 "Barebox"
mtd4: 00008400 00000420 "Barebox-Environment"
mtd5: 00000840 00000420 "hwinfo"
mtd6: 007aafc0 00000420 "Available-dataflash"
mtd7: 0201d800 0001f800 "boot"
mtd8: 0961e000 0001f800 "usr"
mtd9: 0961e000 0001f800 "update"
```

If you are customizing the switch, you may use the UBI commands to change volumes: the commands are installed in the system, within the *initramfs* image so they can be used before the flash is accessed.

This is the role of the three UBI volumes (you can change the size of the volumes or add new ones, but these three names appear in the boot scripts):

`boot`

: The boot volume hosts the kernel and *initramfs* image. It is mounted by the boot loader for the default boot procedure, and is not mounted by the kernel by default.

`usr`

: This is the main filesystem, mounted under `/usr` during normal operation. Both `/wr` and `/var` point to `/usr/wr` and `/usr/var`. Moreover, the boot procedure copies `/usr/etc` to `/etc` as a first step, so any on-flash configuration is actually used by the running system.

`update`

: The volume is a storage place for firmware upgrade. If you copy `wrs-firwware.tar` in this volume, the next boot will completely replace `/usr` with this new image. If the tar file includes them, the kernel and *initramfs* image are replaced as well.

If you want to mount UBI partitions, the command is, for example:

```
mount -t ubifs ubi0:boot /boot
```

where "`ubi0`" refers to the first (and only) UBI partition, and `boot` refers to the symbolic name of the volume, as listed above.

For further details on the update procedure, please see `/etc/init.d/wrs-boot-procedure` (in the source archive it is distributed in `userspace/rootfs_override/`.

5 SDB and Hardware Information

This chapter describes how hardware information is stored and retrieved in version 4.1 and later.

There are a number of information items that should be known to the software, like the MAC addresses, the serial number of the device, and what FPGA model is mounted on the PCB. This information must be available in some storage device, written at manufacture time and never modified.

The storage device of choice, given the hardware architecture of the White Rabbit Switch, is the *dataflash*. The data format we chose is SDB, that we are using in a number of situations.

A description of SDB is to be found in the `ohwr` project called *fpga-config-space*. The *Self Describing Bus* was born to describe address spaces (i.e. the cores that are part of an FPGA design) but is also a good way to implement a small filesystem in limited storage.

5.1 Hwinfo Placement in Dataflash

The *hwinfo* SDB image in the White Rabbit Switch lives at offset 0x94800 in *dataflash*, and is 2112 bytes long (0x840: two or eight erase regions, according to the device in use).

The area is available as `/dev/mtd5` from the Linux kernel, and can be accessed as a partition from *barebox* (the default boot scripts create it as `/dev/dataflash0.hwinfo`).

The binary image includes 4 files, stored as an SDB filesystem:

`manufacturer`
: The manufacturer name, as an ASCII string.

`scb_version`
: The "Switch Core Board" version, which is in the digit.digit form, like `3.3` or `3.4`.

`eth0.ethaddr`
: The MAC address for the management Ethernet port (RJ45, 100Mb/s).

`wr0.ethaddr`
: The MAC address for the first fiber port (SFP, 1Gb/s). Other ports are assigned sequential addresses starting from this one.

`hw_info`
: A line-oriented ASCII file including other "`tag: value`" information.

5.2 Creating the Hwinfo File

The *hwinfo* file is created using *gensdbfs*. The tool is part of *fpga-config-space* and is not included in `wr-switch-sw`, because the package ships a pre-built base image that is then edited in-place.

To re-build the image, please follow the instructions included as comments in the configuration file, `hwinfo-sdb/--SDB-CONFIG--` and the respective commit message. You most likely won't need to rebuild the image, unless you want to add data files or change the manufacturer name from the default.

A pre-built image is included as `binaries/sdb-for-dataflash`.

The script *build/wrs_hwinfo* can be used to edit the file upgrading the MAC addresses and the tagged text file. The tool creates a copy of the base file and modifies it in `/tmp`. It finally prints the new file name on *stdout*.

The following example with "strange" values by design shows how to use the script, assuming `/tftpboot` is the public directory accessed by the TFTP server.

```
F=$(./build/wrs_hwinfo \
   -m1 00:02:04:06:08:0a \
   -m2 22:33:44:55:66:77 \
   -v 3.3 \
   -x "fpga: LX240T" -n 12345)

chmod a+r $F
cp $F /tftpboot
```

Please check the source code for details about the command-line options. The *version* argument is mandatory, because the software must know what version the SCB is (this is not really needed to identify 3.4 from 3.4, but we don't know if we will be able to auto detect 3.5 or 4.0).

Some information items are not really mandatory (the script will not fail if the are not specified), but should be defined anyways because SNMP code retrieves them to tell network administrators. Currently this only applies to the serial number (`-n`)

5.3 Storing Hwinfo in a White Rabbit Switch

You can store your *hwinfo* using either the WRS shell or the boot loader. The former technique can be performed remotely, the latter requires access to the serial console, on the backplane.

5.4 From the Linux Shell

The information lives in partion `mtd5`, as shown in `/proc/mtd` (Memory Technology Device). To avoid accidental erasure, our filesystem doesn't include the device file `/dev/mtd5`, but only the read-only counterpart, `/dev/mtd5ro`.

If you built your own *hwinfo*, with your MAC addresses, and you copied it to */update* (a good staging place for files copied over the network), the following commands will place the information in place:

```
test -c /dev/mtd5 || mknod /dev/mtd5 c 90 10
flash_erase /dev/mtd5 0 0
cat /update/hwinfo > /dev/mtd5
rm /dev/mtd5
```

The commands create the device file if missing (to prevent an ugly error message if you already created it), erase and overwrite the data area, and finally remove the device file.

You new MAC addresses will be effective at the next boot.

5.5 From the Barebox Shell

If you prefer creating or replacing *hwinfo* from the boot loader, this is the procedure. You can't run it with software version 4.0 or earlier, because our boot loader was unable to access *dataflash*. V4.1 fixed this problem and introduced the `sdb` command in the booloader to access your *hwinfo* structure.

```
erase /dev/dataflash0.hwinfo
dhcp 5
tftp hwinfo /dev/dataflash0.hwinfo
```

The code above assumes that the *hwinfo* file is available from your DHCP/TFTP server. According to your configuration one or more commands may be redundant.

The commands rely on a partition being there, and the default script already created it using the following command:

```
addpart /dev/dataflash0 0x840@0x94800(hwinfo)
```

You can verify successful writing by running *sdb ls*, as shown below.

5.6 Accessing Hwinfo at Run Time

You can access the hardware information from *barebox* using the new *sdbinfo*, *sdbread* and *sdbset* commands. The following example shows an example session, using the file built in Section 5.2 [Creating the Hwinfo File], page 18 and featuring a simplified *barebox* prompt of "`bb>` ". Please note that most of this is already in the boot scripts, as we now extract the mac addresses from sdb.

```
bb> addpart /dev/dataflash0 0x840@0x94800(hwinfo)
bb> sdb ls /dev/dataflash0.hwinfo
46696c6544617461:2e202020  00000000-0000083f .
46696c6544617461:7363625f  00000240-00000243 scb_version
46696c6544617461:7772302e  00000220-00000231 wr0.ethaddr
46696c6544617461:6d616e75  00000260-0000026f manufacturer
46696c6544617461:68775f69  00000420-0000083f hw_info
46696c6544617461:65746830  00000200-00000211 eth0.ethaddr

bb> sdb cat /dev/dataflash0.hwinfo manufacturer
Seven Solutions
bb> sdb cat /dev/dataflash0.hwinfo hw_info
fpga: LX240T
scb_serial: 12345
bb> sdb set /dev/dataflash0.hwinfo wraddr wr0.ethaddr
bb> echo $wraddr
22:33:44:55:66:77

bb> sdb set /dev/dataflash0.hwinfo eth0.ethaddr
00:02:04:06:08:0a
```

Barebox passes the MAC address information to the Linux kernel by setting proper environment variables using *sdbset*.

After boot, *sdb* can be accessed using the *sdb-read* command (that this package copied from `fpga-config-space`. The SNMP code accesses the files directly, by linking the *libsdb* code base – again, from `fpga-config-space`.

6 Schematics are Available

The switch schematics for all PCB versions (3.x of the SCB as well as both 3.1, 3.2 and 3.3 of the backplane) are available on the Open Hardware Repository, at `http://www.ohwr.org/documents/180`, which can also be reached from the *Documents* tab of the *White Rabbit* project.

Please note that only version 3.2 and 3.3 of both the motherboard and the backplane has been shipped commercially; you are interested in previous versions only if you are an early developer and have one of those in your hands.

6.1 DIP Switch HW version

Since v3.3, the backplane include a DIP switch configured by the manufacturer in order to define a specific SCB and backplane version. This setup is then read by the software in order to load the correct FPGA binaries and use the proper I/Os. Please be aware that if you upgrade your SCB from LX130T to LX240T but keep the same backplane you might need to change the DIP

switch configuration. Check the code from `userspace/libswitchhw/i2c_io.c` code to know how to reconfigure the DIP switch for you upgraded device.

For example, the v3.3 backplane with v3.3 LX240T SCB must be configured as bellow:

```
+--------------+---+---+---+---+
| DIP position | 1 | 2 | 3 | 4 |
+==============+===+===+===+===+
| DIP value    | 1 | 1 | 1 | 0 |
+--------------+---+---+---+---+
```

Appendix A Installing from Jtag

As an alternative to the serial flasher, you can take control of the system with a JTAG debugger. Please note that the *USB Flasher* is **really** the preferred technique, but in case it doesn't work for you, JTAG is the only way to communicate with the switch.

Previous versions of this manual included detailed instructions about such recovery procedure, but we have not been using JTAG for a long while, so we didn't update the information to the V4 filesystem layout.

If you need to boot from JTAG, please refer to documentation in version 3.3 or earlier of `wr-switch-sw` for generic ideas, knowing the details are different.

Appendix B Bugs and Troubleshooting

Even if the package is already released and used in production, some details can be suboptimal, while some procedures may be tricky and need more explanation.

We are collecting all those issues in the *wiki* page of the project, to avoid frequent updates to this manual to just collect those details. So please visit www.ohwr.org/projects/wr-switch-sw/wiki/Bugs and www.ohwr.org/projects/wr-switch-sw/wiki/Troubleshooting if you have any problem with this package, but feel free to reach us on the mailing list if you don't find help there.