White Rabbit Switch: Developer's Manual

Information about software in the White Rabbit switch, for developers and advanced users
August 2017 (wr-switch-sw-v5.0.1)

Alessandro Rubini, Adam Wujek, Benoit Rat, Federico Vaga, ...

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.
- *White Rabbit Switch: Failures and Diagnostics*: describe various failure scenarios of a switch and ways how to recognize them. Additionally describe SNMP exports of a switch (`WR-SWITCH-MIB`).

The official PDF copy of these 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 four manuals is maintained in `wr-switch-sw/doc`. Within the repository, three of them the *User's Manual*, the *Developer's Manual* and the *Failures and Diagnostics* 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

To build your switch firmware you have to first clone the main git repository, which contains all the sources and building scripts:

```
git clone git://ohwr.org/white-rabbit/wr-switch-sw.git
```

Original build scripts were developed by Tomasz Wlostowski who first made the whole thing work and stick together.

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) 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`" (from v5.0 `make nconfig` and `make xconfig` are supported alternatives), 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.

After release 4.1 we support dynamic reconfiguration. As opposed to build-time configuration, run-time configuration is expected to be frequent, and it's thus documented in the WRS *Users' 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 Other Repositories

Not everything that runs in the switch comes from this package: both the *gateware* image and the *lm32* firmware binary are built from sources external to this package.

The *gateware* is being downloaded as a pre-build tar archive (currently called `wrs-gw-v5.0-20161214.tar.gz`). This is built from the `wr-switch-sw-v5.0` tag 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. All relevant commit identifiers are shown by command `wrs_version -t` or in the SNMP version objects (for more information please refer to switch's MIB file `WR-SWITCH-MIB.txt`).

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 do 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-v5.0` tag to get the exact commit.

2.1.2 Portability

The scripts in their current status are not expected to be very portable. A number of *bashisms* exist, 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 spaces are used in directory names; that is because not all uses of shell variables are properly quoted.

The expected build environment is a native GNU/Linux system; success reports about other environments (e.g. cygwin) are welcome, possibly with associated patches.

2.1.3 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 pre-setting 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 7, 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.

2.1.4 Downloading Files

Every downloaded file is saved to the `downloads` directory (`$WRS_DOWNLOAD_DIR` if set, or the default place `$WRS_OUTPUT_DIR/downloads`). You should arrange not to remove that directory when you recompile over and over during development. Download process is divided into two parts. Firstly, our buildsystem downloads only base packages (at91bootstrap, barebox, linux kernel, switch's gateware and the builroot). The rest of packages are downloaded by the buildroot. In the first step the script downloads mentioned packages, before starting any build, to help telling download errors from other issues.

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:

```
2016-12-14 17:10:46: --- Downloading base packages
2016-12-14 17:10:50: Retrieved at91bootstrap-3-3.0.tar.gz from upstream
2016-12-14 17:10:51: Retrieved barebox-2014.04.0.tar.bz2 from upstream
2016-12-14 17:11:21: Retrieved linux-3.16.38.tar.xz from upstream
2016-12-14 17:11:22: Retrieved wrs-gw-v5.0-20161214.tar.gz from upstream
2016-12-14 17:11:27: Retrieved buildroot-2016.02.tar.bz2 from upstream
```

After buildroot is downloaded, it is unpacked and then configured. Buildroot uses simillar mechanism to the one described above to download packages that it needs. Buildroot prints the progress of downloading of each package. After downloading is over you can work even without a network connection.

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 4GB of storage in excess of the 400MB downloaded.

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 (~18MB), as all the compilation steps are quite verbose.

The following example shows a run on a quad core, dual hyperthreaded system (8*6800 bogoMips in total). If files had already been downloaded, the first few step takes only a few seconds, as shown, to verify the checksums:

```
2016-12-14 17:26:39: --- Downloading base packages
2016-12-14 17:26:39: --- Buildroot: unpack and configure
2016-12-14 17:26:39: Uncompressing buildroot
2016-12-14 17:26:40: Configuring with "[...]/../configs/buildroot/wrs_release_br2_config"
2016-12-14 17:26:40: Patching buildroot
2016-12-14 17:26:40: Reconfiguring buildroot
2016-12-14 17:26:41: --- Buildroot: download packages
2016-12-14 17:26:48: --- Buildroot: compiler and filesystem
2016-12-14 17:26:48: Compiling buildroot
2016-12-14 17:47:54: --- AT91Boot
2016-12-14 17:47:54: Patching AT91Boot
2016-12-14 17:47:54: Building AT91Boot
2016-12-14 17:47:55: --- Barebox
2016-12-14 17:47:55: Patching Barebox
2016-12-14 17:47:55: Building Barebox
2016-12-14 17:48:03: --- Linux kernel for switch
2016-12-14 17:48:52: --- Kernel modules from this package
2016-12-14 17:48:56: --- PTP daemon (ppsi repository as a submodule)
2016-12-14 17:49:05: --- User space tools
2016-12-14 17:49:15: --- Deploying FPGA firmware
2016-12-14 17:49:15: Using pre-built binaries from wrs-gw-v5.0-20161214.tar.gz
2016-12-14 17:49:16: --- Wrapping filesystem
2016-12-14 17:49:21: --- Packing into wr-switch-sw-v5.0-20161214_binaries.tar
2016-12-14 17:49:21: 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 6.

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-v5.0.1-20170825_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 12, 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`. Please note that we now support dynamic reconfiguration at run-time. See the WRS *Users' Manual*.

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 `06-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=06
```

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 one or more of them to build a different switch firmware image (or kernel, or whatever).

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-2016.02` 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.

An example alternate configuration file is `configs/buildroot/wrs_with_musl_config`, that builds a version of the White Rabbit Switch with a different system library (*musl* instead of *uclibc*).

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
0008-fix-refresh-value.patch
0009-one-more-fix-to-RAM-timing-according-to-datasheet.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
0010-pm9g45-init-for-wrs-more-relaxed-nand-timings.patch
0011-lib-sdb-and-sdb.h-from-fpga-config-space-sdbfs-commi.patch
0012-libsdb-integrate-in-build-system.patch
0013-commands-add-sdb-commands-to-list-read-setvar.patch
0014-Read-EDI-bytes-in-JEDEC-to-support-AT45DB641E.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=${WRS_OUTPUT_DIR}/build/buildroot-2016.02
export CROSS_COMPILE=${BR}/output/host/usr/bin/arm-linux-
```

2.4.5 The Linux Kernel

The kernel is currently version 3.16.38, 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/v3.16.38*, and are currently the following ones:

```
0001-initramfs-stop-after-one-cpio-archive.patch
0002-mtd_dataflash-Read-EDI-bytes-in-JEDEC-to-support-AT4.patch
0003-wr-switch-sam9m10g45ek-change-USB-vbus_pin-from-PB19.patch
0004-wr-switch-sam9m10g45ek-enable-FPGA-access-from-EBI1-.patch
0005-wr-switch-sam9m10g45ek-store-device-partitioning.patch
0006-wr-switch-sam9m10g45ek-more-relaxed-nand-timings.patch
0007-wr-switch-sam9m10g45ek-provide-bootcount-using-scrat.patch
0008-wr-switch-at91-udc-force-full-speed.patch
0009-hack-architecture-to-boot-on-our-boot-loader.patch
0010-disable-BBT-for-the-nand-flash.patch
```

The configuration we use to build the kernel is not a patch but a plain `.config` file (`configs/wrs_linux_defconfig`), 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; i.e. 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 you to commit.

Currently, the package includes the following modules:

- *htvic.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.
- *wr_clocksource.ko*: uses WR time as a source for system time. This driver uses the WR counters to make host time flow at the right speed as soon as *ppsi* synchronizes. This ensures no drift will accumulate in system time, keeping the same offset (well lower than a second, after the initial ntp-driven setting event). This is considered acceptable, because system time is only used for logging.
- *wrs_devices.ko*: a dummy module that register our platform devices.

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* submodule in this package. The repository is hosted on `ohwr.org`, like others.

PPSi is *Kconfig* based: you may build it in its own directory by using its *wrs_defconfig* and the proper `CROSS_COMPILE` variable.

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.

libwr 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`.

WRS user space includes also some tools and scripts. They are all described in the User's Manual, even though some of them are only useful to software developers.

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 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 extracts a tar file downloaded from `ohwr.org` as part of the initial download step.

The LM32 program is provided as a pre-compiled binary in `binaries/rt_cpu.bin`.

More details about the binaries are in Section 2.1.1 [Other Repositories], page 2.

2.4.10 The Complete Filesystem

The final step in building the switch software is wrapping together the filesystem for the switch, also making the archives.

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 the build system 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 25MB 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
```

To boot with NFS-root you should use a custom boot script, as described in the section *Using wrboot*, in the WRS *User's Manual*. Please note that the kernel now needs `root=/dev/nfs`, as the old convention `root=nfs` is not supported any more.

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-2016.02/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 by the build scripts.
- The file `userspace/devices.tar.gz`
- The file `$WRS_BASE_DIR/authorized_keys` if it exists.

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.

[1] You can use other programs for accessing serial ports, for example tinyserial (`http://brokestream.com/tinyserial.html`)

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.

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 v4.1, 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 6).

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 pre-allocated 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 Management 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

Note: this section may be slightly outdated, it needs review.

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 separated 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
```

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 5.0.1 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 Flash Memory Use in WRS

This 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: the 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, where a partition is currently reserved but is not being used.

The *dataflash* is partitioned too, in the following way:

```
0x0000.0000 - 0x0000.8400   at91boot
0x0000.8400 - 0x0008.c400   Barebox
0x0008.c400 - 0x0009.4800   Barebox-Environment
0x0009.4800 - 0x0009.5040   hwinfo
0x0009.5040 - 0x0084.0000   Available-dataflash
```

All those partitions are available as *mtd* partitions in */dev*; thus, for example, you can replace `barebox.bin` by just writing it to the right device file. The *hwinfo* partition is only available as a read-only file (*/dev/mtd5ro*) because neither users nor developers are ever expected to change the hardware information data.

Overall, the following is the content of `/proc/mtd` after boot. It is divided in stanzas for a better reading: NAND partitioning, dataflash partitioning and the UBI volumes stored withing "UBIfied-NAND" are thus shown in separate blocks:

```
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 (`zImage`) and initial RAM disk (`wrs-initramfs.gz`). It is mounted by the boot loader for the default boot procedure,

`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-firmware.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. Developers can copy individual files, to upgrade only one of boot loader, kernel, *initramfs* and */usr*.

Additionally there are two files stored on `update` volume. File `last_update` contains date when last update was performed. Second file `saved_date` contains date of last gently shut down of a switch. Please note that both dates are saved with best effort. None of them is guaranteed. For more details please refer to Section 5.4 [Time keeping over restarts], page 23.

For further details on the update procedure and exact file names to be used in partial updates during development, please see `/etc/init.d/wrs-boot-procedure` (in the source archive it is distributed in `userspace/rootfs_override/`.

4.1 Restore default Barebox environment

In some cases it might be necessary to restore default Barebox's environment.

Please note that so far switch uses only "Barebox-environment-backup" partition. It doesn't use "Barebox-Environment".

Erasing can be performed from Barebox's commnad line:

```
erase /dev/env0
```

or from Linux:

```
flash_erase /dev/mtd0 0 0
```

5 WRS Internals

5.1 Inter-Process Communication

This chapter described the network of IPC/RPC communications that are active inside the switch.

Currently there are two mechanisms in place: a simple RPC library ("Remote Procedure Call", so a process can ask another process to perform an action, and a shared memory mechanism, so status of each WRS process can be shared with other processes.

Initially, and up to release 4.1 of `wr-switch-sw` everything was RPC-based, including the passing of status information. Starting in November 2014 we introduced shared memory, to lower the CPU usage and increase the ability to monitor overall Switch status.

5.1.1 mini-rpc

The RPC mechanism in the switch relies on the *mini-rpc* package. The package is a submodule, currently at this commit:

```
4c87062d userspace/mini-rpc (v1.0-5-g28dff05)
```

The library is documented in its own doc/ directory. The basic idea is that an RPC server exports stuff through a Unix socket; the library can return the file descriptor to be used for *select/poll* and all RPC is handled by calling a specific function. The client connects to the server and makes function calls using a library helper.

Each procedure being exported is described by a structure, that lists the number and types of all arguments and the return type of the function. This is, unfortunately, heavier than we initially expected.

If you are studying the code and making sense of the RPC calls, the important things to look for are `struct minipc_pd` (procedure definition), `minipc_server_create`, `minipc_client_create`. To see what procedure are exported, look for the function `minipc_export`.

As a special case, the RPC mechanism can happen over shared memory; this is how the user-space programs can interact with the soft-pll (the so-called *rt subsystem*).

The library is built from a *git* submodule in `userspace/mini-rpc`. Previous versions of this package included more copies of the library: one within *ptp-noposix* and a subset in *rt/ipc*. Since version 4.1 (Oct 2014) of *wr-switch-sw* we stopped supporting *ptp-noposix*, and the *rt* subsystem is built from *wrpc-sw* since version 4.0 (Aug 2014).

5.1.2 RPC Sockets and Communication

This section describes the network of RPC calls that exist in the White Rabbit Switch.

RPC servers are created in the following places:

`userspace/wrsw_rtud/rtud_exports.c`
: The socket is called `rtud` and is used by *wrs_vlans* to request actual actions.

`userspace/ppsi/arch-wrs/wrs-startup.c`
: The `ptpd` channel is created to report PTP status information, but *ppsi* is already instrumented to export using shared memory too. Thus, this RPC server will be removed soon.

`userspace/wrsw_hal/hal_exports.c`
: This channel is called `wrsw_hal` but the code uses is through the macro `WRSW_HAL_SERVER_ADDR`. The RPC server is used only for commands. All clients use shared memory to get status information.

`wrpc-sw::ipc/rt_ipc.c`
: This is a memory-mapped channel, at address 0x10007000 (`RTS_MAILBOX_ADDR`).

Clients are created in the following places:

`userspace/tools/wrs_vlans.c`
: `wrs_vlans` connects to the *rtud* to request configuration actions related to vlan setup. All status information is passed through shared memory.

`userspace/tools/wr_mon.c`
: The tty-based monitoring interface connects to *ptpd* (*ppsi*) to enable or disable tracking. All status information is passed from *HAL* and *ppsi* through shared memory.

`userspace/libwr/hal_client.c`
: The library allows connecting to the HAL socket to ask for various actions (all librarized). The function `halexp_client_try_connect()` is called directly by *wr_phytool*.

`userspace/libwr/rt_client.c`
: The hal connects to the *rt* subsystem to control the soft-pll.

Functions exported by the HAL and PTP are defined in two headers: `hal_exports.h` and `ptpd_exports.h`. The former exists in two places, because *ppsi* is a separate package; the two copies are identical and are expected to remain so. The same applies to `rt_ipc.h`, which appears both here and in *wrpc-sw*.

5.1.3 The Functions being Exported

This section lists all functions that are being exported to RPC by the processes (excluding the RT subsystem).

`ppsi::get_sync_state`
: Called by *wr_mon* to fill a `ptpdexp_sync_state_t` structure. This can be moved to shared memory.

`ppsi::cmd`
: Called by *wr_mon* to ask for two wr-servo operations: enable or disable tracking and adjust phase to a specified value.

`hal::pps_cmd`
: Called by library code (`halexp_pps_cmd`), in turn called by `ptpd_netif_adjust_counters`, called by *wr_phytool*. Called by *ppsi* directly (no *libwr*) in `wrs-time.c`.

`hal::lock_cmd`
: Called by library code (`halexp_lock_cmd`) but not used. Called by *ppsi* directly in `wrs-time.c`.

`rtud::clear_entries`
`rtud::add_entry`
`rtud::remove_entry`
: Called by *rtu_stat* when performing actions.

`rtud::learning_process`
: Called by *rtu_stat* to enable/disable learning process on a set of ports.

`rtud::unrec`
: Called by *rtu_stat* to enable/disable dropping of frames which destination MAC address does not match to present RTU rules.

`rtud::vlan_entry`
: Called by both *rtu_stat* and *wrs_vlans*. It is used to pass a number of parameters to *rtud* and make it perform actions.

`rtud::hp_mask`
: Called by *rtu_stat* when setting which priorities are considered High Priority (this only concerns the traffic which is fast-forwarded).

5.1.4 The RT Subsystem

The in-FPGA processor running the real-time subsystem of the switch is using a shared memory connection for communication. The details are documented in the `mini-rpc` manual. Only the hal process sends commands to the RT subsystem.

5.1.5 WRS Shared Memory

The White Rabbit Switch has a shared memory system, with librarized functions to access it. Each process exports status information in its own file, withing */dev/shm/*; each file has the same structure, but the size allocated in shared memory is process-specific.

The initial part of each process' area is a `stuct wrs_shm_head`, which allows to make some sense of the overall area. The structure is filled by library functions and accessed by shared memory users. You can see how it is used in `tools/wrs_dump_shmem.c`.

The following functions are defined. Please look in the source code for details about how they are used:

`struct wrs_shm_head *wrs_shm_get(enum wrs_shm_name name_id, char *name, unsigned long flags);`
`int wrs_shm_put(struct wrs_shm_head *head);`
: Request access to a shared memory area, and stop using it. Flags are `WRS_SHM_WRITE`, `WRS_SHM_READ` and `WRS_SHM_LOCKED`. If `WRS_SHM_WRITE` is set, head is properly initialized. If `WRS_SHM_LOCKED` is set, a writer will mark the area as locked before writing its own `pid`, and a reader will wait for the `pid` to be valid (i.e. it waits for a writer to be there). A writer using the "locked" flag must release the lock after initialization by calling `wrs_shmem_write(WRS_SHM_WRITE_END)`. On error NULL is returned, and errno is set to `EINVAL`, `ETIMEDOUT` or to the error returned by underlying system calls (for example, `EPERM` if the file cannot be mapped).

`int wrs_shm_get_and_check(enum wrs_shm_name shm_name, struct wrs_shm_head **head);`
: Small helper function for opening shmem and checking if initial data is already populated. Returns different errors when opening shmem failed, when version is not initialized (version equals to 0) or when data in shmem is inconsistent.

`void *wrs_shm_alloc(struct wrs_shm_head *head, size_t size);`
: Allocate data space within the shared memory area. The returned pointer can be used directly. Only writers should allocate but the code is not checking for this. The function is used, for example, in `wrsw_hal/hal_ports.c`.

`void *wrs_shm_follow(struct wrs_shm_head *head, void *ptr);`
: A reader can follow a pointer using this function. The writer can allocate shared memory with `wrs_shm_alloc` and store the pointer in the same shared memory area, thus instantiating structures that point to other structures. But the reader processes map the shared memory at a different address: this function can be used to convert a pointer in the writer's address space to a pointer in the reader's address space, or NULL if on error. Please see `tools/wrs_dump_shmem.c` about how this is used.

`void wrs_shm_write(struct wrs_shm_head *head, int flags);`
`void wrs_shm_write_caller(struct wrs_shm_head *head, int flags, const char *caller);`

`flags` is `WRS_SHM_WRITE_BEGIN` or `WRS_SHM_WRITE_END`. Whenever internal consistency of data structure is needed, the writer should call this function before modifying shared structures, and also when all modifications are done and data is internally consistent. It is recommended to use `wrs_shm_write` version which is implemented as a macro, which calls `wrs_shm_write_caller` with a callee's function name as last parameter. Such approach is can be used for tracking source of write begin and write ends.

A call with `WRS_SHM_WRITE_END` is mandatory for writers that used `WRS_SHM_LOCKED` at `wrs_shm_get` time.

`unsigned wrs_shm_seqbegin(struct wrs_shm_head *head);`
`int wrs_shm_seqretry(struct wrs_shm_head *head, unsigned start);`

A reader can use these functions to ensure it reads internally-consistent data from a shared structure. It relies on proper use of `wrs_shm_write()` by the writer.

`int wrs_shm_age(struct wrs_shm_head *head);`

The function returns the age, in seconds, of the last modification to the memory area. It relies on proper use of `wrs_shm_write()` by the writer.

`void *wrs_shm_data(struct wrs_shm_head *head, unsigned version);`

Returns a pointer to data after the `struct wrs_shm_head`.

`wrs_shm_set_path(char *new_path);`

This function can be used to open shmem files from different location than `/dev/shm`. For example, these functions are used by a `wrs_dump_shmem` tool to allow opening shmem files copied from another switch.

`void wrs_shm_ignore_flag_locked(int ignore_flag);`

This function complements function `wrs_shm_set_path`. It allows to ignore the flag `WRS_SHM_LOCKED`. If such flag is not ignored then function `wrs_shm_get_and_check` is not able to open shmem successfully due to lack of process running with the given pid.

5.2 The HAL Process

In the initial days of White Rabbit Switch, developers tried to have everything managed by a single "Hardware Abstraction Layer" process, the HAL. The process still exists but its role is smaller than it used to, so some of the code is likely unused. After release 4.1 we started a serious audit of the HAL.

The implementation is currently split between *userspace/wrsw_hal* and *userspace/libwr*. Some of the *libwr* functions are used by other processes, but some are just librarized parts of the HAL process (again, we are not very clean and tidy).

If you run `ps` in the switch, you'll see there are two HAL process, one being a child of the other. The child is forked by the *mini-rpc* library, to poll shared memory (the communication channel with the *rt* subsystem). *mini-rpc* has one file descriptor per channel, so users can run *select()* or *poll()*, thus this trick to be able to turn a shared-memory channel into a file descriptor. This is documented in *mini-rpc* documentation. The polling time is currently 25ms.

The HAL process is in charge of replying to IPC requests (*hal_exports.c*), driving the fan speed according to temperature (*libwr/fan.c*) and monitoring ports (this is spread in several files related to *i2c* communication like reading SFPs' eeprom, lighting proper LED in the front panel, etc.).

5.3 Reboot/Reset Diagnostics

For management and diagnostics, the WRS keeps track of the number of boots since power on. It does so relying on four 32-bit CPU registers that are unpredictable at power-up and remain unchanged over reboots. They are called "backup registers" in vendor documentation (`GPBR`): they actually read 0 on power-up but the documentation doesn't say anything so we can't count on it.

Our kernel patches use the 16 bytes in the following way; all values are little-endian.

`bytes 0..2`
: A magic number: if not found, then this is a power-on boot. (Note: we may use the `"reset reason"` register as an alternative)

`byte 3`
: Another magic number: it is '`R`' (0x52) if the last reboot has been requested by an operator. The value is reset to '`U`' (for "unknown") right after boot.

`bytes 4..5`
: The number of boots since power on. This is at least 1. If zero, the code is not working.

`bytes 6..7`
: The number of soft reboots. This is incremented by the *reboot* system call, and not by accidental reboots (i.e. panic or other yet-unforeseen situation). A healthy system should feature one soft-reboot less than total boots.

`bytes 8..11`
: (Not implemented yet). The fault address of the last panic. This is the instruction pointer normally printed by the stack backtrace. The register is zeroed at first boot and only modified within *panic*.

`bytes 12..15`
: (Not implemented yet). The `"link register"` register at last panic, like above. This is usually the caller of the function that failed, but it may be a local register if the failing function saved `lr` to the stack and used it as a scratch register.

The kernel exports the current status in `/proc/wrs-bootcount` in a tagged text format for easy parsing.

This is an example, after power-on, one `reboot` command and one hard reset (by pressing the button on the SCB):

```
# cat /proc/wrs-bootcount
boot_count: 3
reboot_count: 1
last_is_reboot: 0
fault_ip: 0x00000000
fault_lr: 0x00000000
```

We will likely introduce support for the hardware watchdog, but it is not supported at this time.

We may also consider to use one of the kernel's mechanisms for persistent storage of information across reboots, to recover panic messages after a reboot.

Note: Values of these registers can be read remotely via SNMP in objects: `wrsBootCnt`, `wrsRebootCnt`, `wrsRestartReason`, `wrsFaultIP` and `wrsFaultLR`.

5.4 Time keeping over restarts

At normal restart, current time is saved in `/update/saved_date`. Later, at boot switch tries to retrieve correct time from ntp server if configured in dot-config. If no correct date can be retrieved via ntp, switch tries to set a date stored in `/update/saved_date`. Please note that at crash, power down or reset by a reset button no date information is saved. After such restart switch will set date to last gentle reboot or to 1st of January 1970 if there were no gentle restarts before.

Date set from file `/update/saved_date` is never correct, but is based on best effort principle.

5.5 Supervision of running processes

During normal operation `monit` supervises several processes running on a wrs switch. Check is done every 10 seconds. As for now supervised processes are: `wrsw_rtud`, `wrsw_hal`, `ppsi`, `wrs_watchdog`, `lighttpd`, `dropbear`, `snmpd`.

In case any of the supervised processes does not run anymore (because of a crash, exit etc.), `monit` restarts missing process. If 5 restarts of a particular process occurs during 10 cycles (10*10 seconds), the entire switch is restarted. The process' name causing a restart is saved in the file `/update/monit_restart_reason` on the flash partition. After next boot this file is moved to `/tmp/monit_restart_reason`, where can be read by i.e. SNMP. Since it is `/tmp` partition, file with restart reason is lost after next boot.

Since `monit` is started from the inittab, even if `monit` crashes for some reason it will be re-spawned by the `init`.

5.5.1 Disabling monit

In some cases, especially during development it is convenient to disable `monit` to avoid annoying re-spawns of the processes and restarts of the entire switch. `monit` can be disabled with command:

```
/etc/init.d/monit.sh stop
```

which will send STOP signal to `monit`. Additionally `monit` will not start after the boot if there is a config item `CONFIG_MONIT_DISABLE=y` in the dot-config.

To re-enable `monit` first make sure there is no `CONFIG_MONIT_DISABLE=y` in dot-config, then execute command:

```
/etc/init.d/monit.sh start
```

Note: Even when `monit` is disabled there is a process `/usr/bin/monit` in a process list, but its state is "stopped" (T).

5.6 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.org` 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.

5.6.1 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).

`wri1.ethaddr`
: The MAC address for the first fiber port (SFP, 1Gb/s). Other ports are assigned sequential addresses starting from this one.

  Note: In the switches which were produced before v5.0 firmware was released this file contained `wr0.ethaddr`.

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

5.6.2 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`).

5.6.3 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.

5.6.4 From the Linux Shell

The information lives in partition `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.

5.6.5 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 bootloader 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.

5.6.6 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 5.6.2 [Creating the Hwinfo File], page 24, 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 wri1.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 wri1.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`.

Appendix A 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.

A.1 DIP Switch HW version

Since v3.3, the backplane includes 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/libwr/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 B 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 C 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 our project pages. Please visit:

- Frequently Asked Questions: `http://www.ohwr.org/projects/white-rabbit/wiki/FAQswitch`
- Issues for WR Switch SW project: `http://www.ohwr.org/projects/wr-switch-sw/issues`
- Issues for WR Switch HDL project: `http://www.ohwr.org/projects/wr-switch-hdl/issues`

If you have any problem with this firmware and you don't find help in the above links, feel free to reach us on the *white-rabbit-dev* mailing list.