White Rabbit Switch: software build scripts

August 2012
How to rebuild the whole software package from sources

Alessandro Rubini, Benoit Rat, Federico Vaga et al.
**Table of Contents**

**Introduction** ................................................................. 1

1 **Overview** ................................................................. 1
   1.1 Portability ............................................................ 1
   1.2 Environment Variables ............................................. 2
   1.3 Downloading Files .................................................. 3

2 **Building** ................................................................. 3
   2.1 Building Procedure .................................................. 3
   2.2 Build Script Description .......................................... 4
      2.2.1 Release Package ............................................... 4
      2.2.2 Rebuilding Parts .............................................. 4
      2.2.3 Rebuilding From Scratch .................................... 5

3 **Flashing of WRS-3** .................................................... 5
   3.1 USB connections .................................................... 5
   3.2 Flashing Procedure ................................................. 6
      3.2.1 Flash Script Description ..................................... 7

4 **Booting with Barebox** ................................................ 8
   4.1 Description of the menus ......................................... 8
   4.2 Using wrboot ........................................................ 9

5 **The Individual Build Steps** ......................................... 9
   5.1 The Compiler ........................................................ 9
   5.2 Buildroot ............................................................ 10
   5.3 The IPL .............................................................. 10
   5.4 The Boot Loader ................................................... 10
   5.5 The Linux Kernel ................................................ 11
   5.6 Kernel Modules .................................................... 12
   5.7 PTPd ................................................................. 12
   5.8 User Space Applications ......................................... 12
   5.9 VHDL and LM32 Binaries .......................................... 13
   5.10 The Complete Filesystem ....................................... 13

6 **Code layout in a production switch** ............................. 14

**Appendix A**  Schematics are Available ............................. 15

**Appendix B**  Older Hardware Releases ............................. 15
   B.1 Flashing v3.1/v3.0 ................................................. 15
   B.2 Serial Ports in 3.0/3.1 .......................................... 16
Appendix C  Installing from Jtag ................. 16
  C.1 Using PEEDI .................................................. 17
  C.2 Using SAM-ICE .................................................. 17
  C.3 Installing the Boot Loader from JTAG ....................... 17

Appendix D  Bugs and Troubleshooting .............. 18
Introduction

This document describes the software build procedures for the White Rabbit Switch. The procedure described is for version 3 of the hardware project, which has been successfully booted for the first time on Sep 13th 2011. The different steps focus on the v3.2, however you can find some (though limited) documentation for v3.0 & v3.1 in Appendix B [Older Hardware Releases], page 15.

Note: the switch, as shipped, works perfectly with the provided binaries (http://www.ohwr.org/projects/wr-switch-sw/files), and most users will only need to run the flasher – see Chapter 3 [Flashing of WRS-3], page 5. This document as a whole is mainly aimed at developers who want to customize their own switch, which is actually a GNU/Linux host.

If you have an older Version-2 switch, please refer to the v2 branch of the repository at git://ohwr.org/white-rabbit/wr-switch-sw.git (or git@ohwr.org:white-rabbit/wr-switch-sw.git if you are granted write access).

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.

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.

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 should work even if your default shell is ash or dash.

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.

### 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 rebuilding everything, the defaults are applied for each unset variable; but when rebuilding a sub-package you will need to set the variables beforehand. Each sub-package complains if it needs variables that are not set in your 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.

**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.

**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 5.1 [The Compiler], page 9 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.
Chapter 2: Building

If you compiled for version 2 you will notice we do not have \texttt{WRS_WR_REPOPOSITORY} any more. We used to rely on upstream archives to be part of the \textit{White Rabbit} subversion tree, but I did not add the new archives in there because of their size.

1.3 Downloading Files

Every downloaded file is saved to the download directory (if set \texttt{WRS.DOWNLOAD\_DIR} else the default \texttt{\$WRS\_OUTPUT\_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, because I think this eases development in a way: you first wait a while but can tell download errors from other issues, and then you can build it all even without a network connection.

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

- If the file exists in the download directory, the \texttt{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 \texttt{wrs\_download}, in the file \texttt{scripts/wrs\_functions}, based on \texttt{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 Building

2.1 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):

/\texttt{path/to/wr-switch-sw/build/wrs\_build-all}

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

/\texttt{path/to/wr-switch-sw/build/wrs\_build-all} > \texttt{logfile}

Please note that there are also a number of warning messages being printed to \texttt{stderr}. It is a few hundred lines over the many minutes it takes to build \texttt{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 pair of seconds to verify the checksums:

2012-07-06 17:23:57: --- Downloading all files
2012-07-06 17:24:02: --- Buildroot compiler and filesystem
2012-07-06 17:24:02: Uncompressing buildroot
2012-07-06 17:24:02: Patching buildroot
Chapter 2: Building

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 your 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 one step, you can remove the marker in build/_done (the markers are empty files with name like 01-buildroot and 04-kernel).

### 2.2 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
```

#### 2.2.1 Release Package

Once all the building steps have succeed, you can easily create a package in `WRS_OUTPUT_DIR` using the following command:

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

This will generate a `tar.gz` package. The name of the package is composed by the prefix and the description of the current status of the git repository. This is the same naming policy of the kernel and other packages.

The prefix is `wrs-firmware-`; the information about the current status are retrieved by using the git command `git describe --always --dirty`. For example, if you are building the tag `wr-switch-sw-v3.0`, your package name will be `wrs-firmware-wr-switch-sw-v3.0.tar.gz`. If you are building on some personal commit the name includes some extra information to describe your commit: number of commits form the last tag (if any), SHA1 code of the commit. If you have uncommitted changed, the suffix `-dirty` is there too.

#### 2.2.2 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 kernel you should execute:

\[ /path/to/wr-switch-sw/build/wrs_build-all --step-04 \]

You can list all compiled module by calling

\[ /path/to/wr-switch-sw/build/wrs_build-all --list \]

As an alternative, you can run the individual script from within scripts/, after setting the proper environment variables.

### 2.2.3 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:

\[ /path/to/wr-switch-sw/build/wrs_build-all --clean \]

### 3 Flashing of WRS-3

This chapter describes the different steps to install the WRS-3 with the correct firmware. This procedure describes the installation of the switch with a SCB v3.2 and a Mini-Backplane v3.2. If you have an older version you might look at the Appendix B [Older Hardware Releases], page 15 and the footnotes.

#### 3.1 USB connections

In order to perform the flashing operation easily, you should connect two mini-USB cables to the two front panel sockets. One is used to communicate with the internal ROM of the CPU, the other is a serial port used to interact with the various applications and FPGA. One of the GPIO pins of the CPU is connected to a switch, so that the same serial port can be used for either device (CPU and FPGA) in different times.

The diagnostic serial port is the right mini-USB \emph{(Test)} port of the switch. This is the debug UART of the ARM (default at boot time) and can later be switched to the FPGA. Once connected \texttt{/dev/ttyUSB0} should appears on your machine, and you can connect using minicom\footnote{You can use other programs for accessing serial ports, for example tinyserial} like this:

\[ /dev/ttyUSB0 \]
Unfortunately, the preferred way to communicate with the CPU internal ROM is through the other USB port, using the **left** mini-USB port of the switch. When the ROM is unable to boot user code from its SPI flash memory, it accepts to be enumerated by the host. You can see the enumerated device in your own host:

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

The kernel should automatically load the proper device driver, and you are expected to see `/dev/ttyACM0` or equivalent in your system.

If the device is not enumerated, this mean that there is already some code in the *dataflash*, which the switch tries to boot. In order to disable the dataflash you need to open the switch box and fit a 1mm jumper² on the *DFEN* pin as shown in picture below. Most of the switches are shipped with a jumper already plugged between two GND pins. If your switch has a jumper plugged, you can use it to disable the *dataflash*, a jumper in the switch,

With the jumper in place, you should reset the machine pressing the button near the 20-pin JTAG connector. When you see that the USB device has been enumerated, you should remove the jumper so the programming procedure can access the *dataflash* device.

### 3.2 Flashing Procedure

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. To the package is the `.tar.gz` file created by “wrs_build-all --pack” (see Section 2.2.1 [Release Package], page 4).

The command to flash is this:

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

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

---

² On v3.0 & v3.1 this jumper does not exist. Refer to Section B.1 [Flashing v3.1/v3.0], page 15
Chapter 3: Flashing of WRS-3

The process calls the flasher program twice (so you’ll see the initialization strings two times) and takes around 3 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
[...]
Programming DataFlash...
  0x70000000 : /tmp/at91bootstrap.bin ; size 0xf7c (3Kb)
DataFlash: Writing 197452 bytes at offset 0x9400 buffer 70000000....
Programming DataFlash Done!!!
[...]
Programming NandFlash...
  0x70100000 : /tmp/zImage ; size 0x1666c4 (1433Kb)
[...]
Programming NandFlash Done!!!
Closing...
```

If you look at the serial port, during programming, you will see messages like these:

```
-I- Startup: PMC_MCKR 1202 MCK = 100000000 command = 0
-I- --- EXTRAM ISP Applet 2.9 ---
-I- -- AT91SAM9G45-EK
[...]
-I- End of applet (command : 2 --- status : 0)
```

### 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] MAC [<firmware>.tar.gz] [DEV]

MAC: MAC address in hexadecimal separated by ':' (i.e, 00:01:02:03:04:05)
<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 (dataflash), nf (nandflash), test.
  -e           Completely erase the memory (Can erase your configuration)
  -b|--build   Use file that you have build in the WRS_OUTPUT_DIR
  --test       Use file for testing the switch (not available)
  --silent     Don’t ask MAC or S/N and use default 02:0B:AD:C0:FF:EE
```

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 the `--silent` flag is used the MAC will be by default 02:0B:AD:C0:FF:EE; otherwise MAC is mandatory. If MAC is not specified the script exit. In this version, the MAC address is unused but this field is mandatory for the script; so, we suggest to use the `--silent` option. Future version of the switch will use the MAC address configuration.

You can also flash the image you have build using Chapter 2 [Building], page 3 by adding the tag `-b|--build`.

If you want to flash your own `at91boot.bin`, `barebox.bin`, `kernel` or `file-system` you can just place them in the `$WRS_OUTPUT_DIR/images` directory and call the script with `-b|--build` option. You can select a mode using the `-m|--mode` flag to choose to write in dataflash (df), nandflash (nf), both (default) or ddr memories (ddr)^3. The memory mode is used to select the kind of memory to write. The flash script is used to install different binaries on these memories:

^3 The ddr memory mode is only for developers
• dataflash: at91boot and barebox binaries
• nandflash: kernel zImage and the file-system

You can also erase the selected memory before write your binaries; to do this add the option
-e. Note that if you use this option with the default memory mode you will erase both the
nandflash and dataflash. If you want to erase only one memory please select it with -m|--mode
option.

The script performs the following steps:
• It compiles the loader (usb-loader/ subdir).
• Check if SAMBA bootloader is present.
• It picks the correct binaries according to the options.
• Optionally, it changes the default MAC address in barebox default environment, so you can
make all switches different.
• It erases the memory selected by mode.
• And finally, it writes the corresponding binaries to dataflash, nandflash or ddr.

4 Booting with Barebox

After the initial installation, the boot loader will offer you an interactive menu, where the first
entry is selected by default. The menu is a simple ASCII interface on the serial port, and looks
like the following:

Welcome on WRSv3 Boot Sequence
  1: boot from nand (default)
  2: boot from TFTP script
  3: edit config
  4: exit to shell
  5: reboot

If flashing of the whole system was successful, the first entry will simply work, booting the switch
without any network access. Although a DCHP client runs by default after boot, everything
will work even if you leave the Ethernet port unconnected or you have no DHCP server when
the switch is operational.

If booting from NAND memory fails (for example because you erased the kernel partition) the
menu is re-entered automatically.

The other entries are provided to help developers.

4.1 Description of the menus

The individual menu items perform the following actions:

1: boot from nand (default)
  This entry is selected by default after 10 seconds of inactivity on the serial port. It
  boots the system from its own NAND memory. This “just works”.

2: boot from TFTP script
  This entry tries to download a barebox script from your TFTP server; if successful it
  then executes it. Developers are expected to customize the script to support any kind
  of environment, from customized kernel command-line to NFS-Root environments.
  See Section 4.2 [Using wrboot], page 9 for details.

3: edit config
  This fires the editor on the configuration file, and saves it to flash when the user is
  done. This is useful to change the MAC address of the ARM network port. Please
note that saving save the whole ‘/env’ file tree, so you can also change the init scripts interactively and have them stored persistently on the flash.

4: exit to shell
By choosing this entry, the user can access the shell-like interface of barebox. The entry is aimed at developers who know what they are going to type.

5: reboot
This entry is useful to see and log the exact boot messages. Since the serial-USB converter is switch-powered and not USB-powered, you won’t be able to hook at the serial port soon enough after power-on. Actually, the menu timeout is left to 10 seconds and not less for the very same reason.

4.2 Using wrboot
If you use the wrboot script option, you can for example run an NFS-Root system or do whatever customization and testing you want.

The provided procedure tries to load the script from TFTP under three different names, from most specific to most generic, and the first match will be used. The first name is MAC-address-based, the second is IP-address-based and the third is just ‘wrboot’.

This is for example what I see in my logs when only providing ‘wrboot’:

dhcpd: DHCPOFFER on 192.168.16.224 to 02:0b:ad:c0:ff:ee via eth0
atftpd[5623]: Serving wrboot-02:08:AD:CO:FF:EE to 192.168.16.224:1029
atftpd[5623]: Serving 192.168.16.224/wrboot to 192.168.16.224:1030
atftpd[5623]: Serving wrboot to 192.168.16.224:1031
mountd[21014]: NFS mount of /tftpboot/192.168.16.9 attempted from 192.168.16.9

We chose to place the IP-address-based name in a subdirectory because this is the default place where the NFS-Root filesystem is mounted from, as shown in the log excerpt above. So you’ll have your ‘wrboot’ in the same place.

The ‘binaries’ subdirectory of this package includes two known-working wrboot scripts as examples; one if for use with static IP addresses and the other relies on DHCP. If you want to override the default NFS-Root directory mounted from the server (which is /tftpboot/<ip-address>) you can add something like the following line to your ‘wrboot’ script:

`bootargs="$bootargs nfsroot=/opt/root/wrs-3"
`

If you use static IP addresses, please note that you should fix ‘/etc/init.d/S40network’ in the filesystem for your switch, so it doesn’t run the DHCP client.

5 The Individual Build Steps

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

5.1 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.
5.2 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 patches/buildroot/buildroot-config-wrswitch. The configuration is then changed only if you pre-set your own CROSS_COMPILE variable.

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. After making your choices, copy back the file .config to patches/buildroot/buildroot-config-wrswitch in this package.

You can also set WRS_BUILDROOT_CONFIG to the full pathname of your configuration file of choice. The file must be a copy of the .config after the make menuconfig step described above. 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.

5.3 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. For simplicity, a known-working binary is part of the binaries directory of this package as at91bootstrap.bin, the same name used later in the installation instructions.

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.

5.4 The Boot Loader

The switch uses barebox as a boot loader. We are running version 2012-05, 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 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-add-DHCP-retries-by-tom.patch
- 0005-gpio-add-function-to-check-them.patch
Chapter 5: The Individual Build Steps

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/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 Chapter 4 [Booting with Barebox], page 8.

Building barebox relies on a Kconfig setup, and the configuration file we use is ‘patches/barebox/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):

```bash
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):

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

A pre-built binary is available as binaries/barebox.bin. The ELF version is copied to images as well, as images/barebox; this file includes the symbol table and may (or may not) be useful.

5.5 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:

```bash
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-fix-nand-parition-layout-and-usb-vbus.patch
0006-fiq-support.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
5.6 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.
- \( at91\_softpwm \): a tool that generates a PWM signal for the fan.
- \( wr\_rtu.ko \): the routing-table interface between the switching core and the associated user-space daemon.

5.7 PTPd

The Precision Time Protocol Daemon being used is hosted in a different repository, but it is registered as a \textit{git} submodule in this package. The repository itself is on \textit{ohwr}, like others. We are working on a better code base for a portable PTP, but it is not yet ready as of this release.

A plain \textit{make} in the \texttt{ptp-noposix} directory might fail, because of missing environment variables. because after building \texttt{ptpd} other steps are tried, but they are only needed for the freestanding environment (i.e., LM32 with supporting code) and will fail for this \texttt{arm-linux} hosted build.

Additionally, the script installs headers for the HAL and \textit{libptpnetif}.

5.8 User Space Applications

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

The subdirectories in ‘\texttt{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:

\begin{itemize}
  \item \texttt{mini-rpc} \quad 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.
  \item \texttt{libswitchhw} \quad A series of utility functions to access the switch itself.
  \item \texttt{wrsw\_hal} \quad The main application program for the White Rabbit Switch operation. The script installs the executable in \texttt{images/wr/bin}.
  \item \texttt{wrsw\_rtud} \quad The daemon for the routing table unit, used for routing around data frames. It is installed in \texttt{images/wr/bin}.
\end{itemize}

The most important tools in ‘\texttt{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 former reads from a file using mmap (usually you run it on /dev/mem) and writes to stdout. The latter reads from stdin and writes using mmap. They are classic tools distributed in the Linux Device Drivers examples since 1998.

The program is a simple program for talking with serial ports.

A tool to read and write PHY registers in the switch

A simple monitor of White Rabbit status. It prints to stdout using the standard escape sequences.

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=).

5.9 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 extract a tar file downloaded from ohwr.org as part of the initial download step. The tar is currently called wrs3-gw-v3.0-20120801.tar.gz and has been built from the wr-switch-sw-v3.0 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 precompiled binary in binaries/rt_cpu.bin. The respective source code is in the ‘rt’ directory of this package, but the build scripts don’t deal with it to avoid requiring an LM32 development environment.

5.10 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 three output files, that can be used in different situations, like initramfs or unpacking for NFS-Root. All of them are around 15MB of data, and the flashing script only uses the first listed here:

`wrs-image.jffs2.img`

This is a raw image in JFFS2 format, which is ready to be written to NAND flash. This is used bt the USB flasher.

`wrs-image.cpio.gz`

If you prefer to run an initramfs system, this is the file you should use. We used it a lot during development, to avoid wearing the flash device and wasting time waiting for the flasher to run.

`wrs-image.tar.gz`

We added this out of laziness, because it’s easier to create your NFS-Root directory from tar than from cpio.
You can uncompress either `cpio` or `tar`, for example to run NFS-root, by running one of the following commands in a newly-created empty directory:

```bash
zcat $WRS_OUTPUT_DIR/images/wrs-image.cpio.gz | sudo cpio --extract
```

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

Each of the three 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 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.

## 6 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 `barebox`. 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 (or at least the `dataflash` area).

NAND memory is used for user-data: the boot loader configuration, the kernel and the filesystem. This is how the memory is used. While it definitely looks suboptimal, it is the result of our own history of development and there are no problems with this “strange” split up of space:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x0004</td>
<td>Empty (space for a barebox)</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x0008</td>
<td>Barebox environment</td>
</tr>
<tr>
<td>0x0008</td>
<td>0x0010</td>
<td>Empty</td>
</tr>
<tr>
<td>0x0010</td>
<td>0x0090</td>
<td>Kernel (plenty of space)</td>
</tr>
<tr>
<td>0x0090</td>
<td>0x0400</td>
<td>Empty</td>
</tr>
<tr>
<td>0x0400</td>
<td>0x0c00</td>
<td>Filesystem space, jffs2</td>
</tr>
<tr>
<td>0x0c00</td>
<td>0x2000</td>
<td>Available</td>
</tr>
</tbody>
</table>

When you boot Linux, the three empty areas are not visible, while the last area is accessible. This is the content of `/proc/mtd` after boot:

```
dev:  size  erasesize  name
mtd0: 00800000 00020000 "Kernel"
mtd1: 08000000 00020000 "Filesystem"
mtd2: 00040000 00020000 "Barebox Environment"
mtd3: 14000000 00020000 "Available"
```
If you are customizing the switch, you may use the `flash_eraseall` and `cat` to replace individual parts of the system, like the kernel, or erase the barebox configuration to restore the factory defaults. Note however that you shouldn’t modify the dataflash device (/dev/mtd5) unless you really know what you are doing.

Appendix A  Schematics are Available

The switch schematics for all PCB versions (3.0, 3.1 and 3.2 of the SCB as well as both 3.1 and 3.2 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 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.

Appendix B  Older Hardware Releases

So, if you have an older PCB in your hands, your hardware is unsupported, but here are some notes that may be useful. The differences are minor, mainly in GPIO routing, but there is no complete list of changes readily available, unfortunately.

B.1 Flashing v3.1/v3.0

To flash any switch using the USB flasher, you need to force the ROM to run the boot protocol called SAMBA Monitor which mean that you must prevent the ARM from finding valid code in the data-flash.

In version 3.2 (see Section 3.1 [USB connections], page 5) there is a jumper to disable dataflash, however for version v3.0 & v3.1 it does not exist. Thus, you need to short pins 1 and 4 of the dataflash chip, so the CS* pin never goes low and the internal ROM won’t be able to access your at91boot code.

I used two wires to be shorted together as needed. The next figure shows both the shorted pins (the dataflash is on the left and there are two arrowheads pointing to them) and the USB cable (on the right).
After shorting the pins you can press reset (the button near the USB connector). If things go well, the device is enumerated as shown; you must un-short the wires at this point or you will not be able to write the new information to `dataflash`.

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

The device should also appear as `/dev/ttyACM0` or equivalent, and the further flashing procedure is the same as in version 3.2.

**B.2 Serial Ports in 3.0/3.1**

If you have a 6-port mini-backplane, you can connect the 6 pins of P2 to the similar connector on the backplane labeled as `Connect to MCH/P2`. The backplane hosts a CP2102 USB adapter and a mini-USB socket. Note that you may connect all 6 pins, but only TX and RX signals are needed. If you need more details for this setup, please write to the mailing list.

The port uses 2.5V signals, but a level converter for 3.3V usually works without issues or damages; the figure below shows the connection (black is GND, orange is RX and white is TX).

---

**Appendix C 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 your, JTAG is the only way to communicate with the switch.

Each debugger has its own command language, so you will need to adapt to yours. What is shown here refers to the *peedi* & *sam-ice* tools. These notes are only aimed at low-level developers and leave a lot unsaid.
Appendix C: Installing from Jtag

C.1 Using PEEDI

As a first step, you will need to ensure the JTAG clock is slow enough. The clock can be no faster than 1/6th of the CPU clock, so you need 3kHz at most (the G45 starts up with an internal oscillator, which has an unpredictable value between 20kHz and 40kHz). Then, I would verify that the internal SRAM is working; I do that with cool food and bad coffee instead of the usual smelly dead beef.

```plaintext
    clock init
    mem write 0x300000 0xc001f00d
    mem write 0x300004 0xbadc0ffe
    mem read 0x300000 2
    ==> 0x300000: 0xC001F00D 0xBADC0FFE
```

Now, you can load your `at91bootstrap` to the internal SRAM, retrieving it from the network or your host filesystem. Since no boot loader is there, you should place a breakpoint after `at91bootstrap` initialized SDRAM and the PLL. Finally you can load `barebox` and jump to it. Such step is better performed with the full JTAG clock, or it would take several dozen minutes.

```plaintext
    mem load at91bootstrap.bin 0x300000
    break add hard 0x300088
    go
    ## wait for the breakpoint to happen
    break del all
    clock normal
    mem load barebox.bin 0x73f00000
    go
```

C.2 Using SAM-ICE

This section follows the same steps as in Section C.1 [Using PEEDI], page 17, but using the syntax of the SAM-ICE tool recommended by Atmel.

```.start
Checking SRAM
    speed 2
    w4 0x300000 0xc001f00d
    w4 0x300004 0xcbadc0ffe
    mem32 0x300000 2
Uploading at91bootstrap & barebox to DDR
    speed 2
    r
    wreg "R15 (PC)" 300000
    loadbin /tftpboot/at91bootstrap.bin 0x300000
    SetBP 0x300088 H #Check 0x300088
    g
    speed a
    loadbin /tftpboot/barebox.bin 0x73f00000
    ClrBP 1
    g
```

C.3 Installing the Boot Loader from JTAG

After you managed to load and run `barebox.bin`, you will see the following messages on the serial port, at 115200,8N1. The first 4 lines are printed by `at91bootstrap`, the rest by `barebox`.

```
Start AT91Bootstrap...
    DDR2 Config: 0x39 (NC=0xa, NR=0xd, CAS=0x3, ba_offset = 0x18)
    DDR2 Config: 0x39 (NC=0xa, NR=0xd, CAS=0x3, ba_offset = 0x18)
    Compiled by federico (Jul 31 2012 16:14:34)
    git rev:
        Begin to load image...
```
Appendix D: Bugs and Troubleshooting

When the boot loader is running, you can boot a kernel and use its own /dev/mtd devices to write to the DataFlash and NAND memories.

According to the partition table you have in your kernel sources, you will see a different set of mtd files, but you can identify them by looking at /proc/mtd:

```
# cat /proc/mtd
dev  size  erasesize  name
mtd0:  00800000  00020000  "Kernel"
mtd1:  08000000  00020000  "Filesystem"
mtd2:  00040000  00020000  "Barebox Environment"
mtd3:  14000000  00020000  "Available"
mtd4:  00840000  00000420  "spi0.0-AT45DB642x"
```

Here above, the DataFlash is /dev/mtd4, whereas the former partitions refer to NAND memory.

You should then write `at91boot` to offset 0 of the DataFlash and `barebox` to offset 0x8400 (33792):

```
cat at91bootstrap.bin > /dev/mtd4
dd bs=33792 seek=1 if=barebox.bin of=/dev/mtd4
```

Now you can detach the debugger, press reset and see `barebox` starting without the need for the JTAG any more. The following steps are the same as already described.

Appendix D Bugs and Troubleshooting

Even if we released the package as version 3.0 (being the first release for version 3 of the hardware), 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 http://www.ohwr.org/projects/wr-switch-sw/wiki/Bugs and http://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.