Gaisler Buildroot

Gaisler Buildroot User's Manual

USER MANUAL

RELEASED OCTOBER 2024

Table of Contents

1. Introduction

1.1. Overview

This Buildroot release for LEON and NOEL is suitable for using Buildroot as a frontend for building both the Linux kernel and a root file system with user space software. It contains default configurations for LEON and NOEL systems that can be used as a baseline. Nothing however is preventing it to be used to build only a userspace environment, that can be used together with an externally built kernel.

The main documentation for how to work with Buildroot is found in the regular Buildroot manual. It is provided in several different forms in the `docs/prebuilt` directory, including as text, PDF and HTML. In the top level directory there is a general Buildroot `README` file as well as a changelog, in the `CHANGES` file, for the upstream Buildroot release the Gaisler Buildroot release is based on. This manual and a changelog for the Gaisler Buildroot release can be found in the `docs/gaisler` directory.

This document describes the particulars of the Buildroot release for LEON and NOEL. It describes what it adds in on top of the official Buildroot release it is based upon, as well as giving some pointers and tips. Our latest Linux kernel release is included as kernel patches that are automatically used when using our default configuration. Our LEON toolchain, the MKLINUXIMG second stage bootloader as well as optionally the MKPROM2 boot loader for LEON, and OpenSBI for NOEL, can be downloaded automatically and configured within the Buildroot configuration interface.

This Buildroot release is not aimed to be used under the Linuxbuild LEON Linux kernel and userspace build environment, that is being phased out. It is rather aimed to be used instead of Linuxbuild.

1.2. Host System Requirements

The Buildroot frontend is only supported under Linux. See the section “System requirements” in Chapter 2 of the regular Buildroot manual for details on what is required to be installed on the host system.

In particular for configuring Buildroot, the Linux kernel and some other components it is advisable to install development versions of the Qt5 library.

1.3. Installation

Download gaisler-buildroot-2024.02-1.2.tar.bz2 available from https://gaisler.com. Unpack it anywhere with

```
tar xf gaisler-buildroot-2024.02-1.2.tar.bz2
```

This will unpack to a directory `gaisler-buildroot-2024.02-1.2`. Enter it with:

```
cd gaisler-buildroot-2024.02-1.2
```

This is the top level Buildroot directory from which everything is done unless otherwise specified. Relative paths in configurations are in general relative to this Buildroot top level directory. This goes for relative paths mentioned in this manual as well.

Tools specific for LEON such as LEON Linux toolchains, MKLINUXIMG, and MKPROM2, and likewise NOEL OpenSBI for NOEL, are downloaded automatically, potentially after selecting between versions in the Buildroot configuration. For both LEON and NOEL it is possible to build a toolchain using Buildroot, but a pre-built toolchain is downloaded by default.

2. Quick start guide

Install Buildroot

```
tar xf gaisler-buildroot-2024.02-1.2.tar.bz2
```

Enter the Buildroot top level directory.

```
cd gaisler-buildroot-2024.02-1.2
```

Set up the default configuration, e.g.

```
make gaisler_leon_defconfig
```

for a LEON system with memory at 0x40000000, or

```
make gaisler_noel64_defconfig
```

for 64-bit NOEL-V. Other possible default configurations are listed in Section 3.1.

Optionally, do additional configurations in the Kconfig interface:

```
make xconfig
```

See Section 1.2 for system requirements. See the configuration section printed by `make help` for additional means of configuration, e.g. `make menuconfig`.

Start the process of downloading and building everything:

```
make
```

2.1. Running the default LEON RAM image

The resulting RAM image can be found at `output/images/image.ram`. It is ready to be loaded and run in e.g. GRMON or TSIM on a LEON system with memory at address 0x40000000. Symbols for the kernel for debugging purposes can be found in `output/images/vmlinux`. They can be loaded from within GRMON or TSIM with the **symbols** command, or in GDB with the **symbol-file** command.

Loading and running the image using GRMON3 can be done, replacing "-debuglink" with an appropriate debug link, with e.g.

```
grmon -debuglink -nosram -nb -u -e "load output/images/image.ram; symbols output/images/vmlinux; run"
```

Loading and running the image in the TSIM3 simulator can be done with e.g.

```
tsim-leon3 -nosram output/images/image.ram -sym output/images/vmlinux -e "run"
```

2.2. Running the default NOEL RAM image

The resulting RAM image can be found at `output/images/fw_payload.elf`. It is ready to be loaded and run in e.g. GRMON on a suitable NOEL system. Symbols for the kernel for debugging purposes can be found in the kernel build directory, generally `output/build/linux-`*`VER`*`/vmlinux`, where *`VER`* is the base kernel version, or `custom` if working with an external Linux kernel source directory. They can be loaded from within GRMON with the **symbols** command, or in GDB with the **symbol-file** command.

Loading and running the image using GRMON3 can be done, replacing "-debuglink" with an appropriate debug link, and "somedtb" with an appropriate DTB file, with e.g.

```
grmon -debuglink -u -e "dtb somedtb; load output/images/fw_image.ram; run"
```

3. Configuring and Building

General information on how to configure Buildroot and different packages can be found in the regular Buildroot manual.

3.1. Default Configurations

This chapter lists the different default configurations for LEON and NOEL systems that are provided in the release. These can be built upon to suit a particular LEON and NOEL system. These default configurations also chooses a default Linux kernel configuration.

Table 3.1. Default Configurations

Configuration	Target	Notes
gaisler_leon_defconfig	General LEON3/4/5 systems	Assumes memory at 0x40000000
gaisler_noel64_defconfig	General 64-bit NOEL-V systems	Assumes memory at 0x0
gaisler_noel32_defconfig	General 32-bit NOEL-V systems	Assumes memory at 0x0
gaisler_gr740_defconfig	GR740	General GR740 config to base board specifics on
frontgrade_ut700_defconfig	UT700	General UT700 config, with errata fixes enabled, to base board specifics on

As an example, to use e.g. “gaisler_leon_defconfig” configuration, just do

```
make gaisler_leon_defconfig
```

3.2. Buildroot Configuration

Buildroot, what packets to build and some configuration of built packets can be configured using e.g.

```
make xconfig
```

or

```
make menuconfig
```

See the regular Buildroot manual for details and alternatives.

The Buildroot configuration file is by default placed in `.config` in the top level directory. The configuration file itself is unaffected by `make clean`.

See also the LEON Linux User's manual [https://www.gaisler.com/doc/leon-linux.pdf] for some pointers on useful Buildroot packages for certain drivers and kernel subsystems for Linux on LEON.

3.3. Linux Kernel Configuration

The kernel can be configured using e.g.

```
make linux-xconfig
```

See Chapter 4 for Linux kernel matters.

3.4. MKLINUXIMG Configuration for LEON

The MKLINUXIMG second stage bootloader is enabled and configured as part of the Buildroot configuration under the bootloader section for LEON. Just as regular Buildroot packages, it is downloaded and installed automatically.

See also the LEON Linux User's manual [https://www.gaisler.com/doc/leon-linux.pdf] for input on specific configuration needs for different drivers and subsystems for Linux on LEON.

3.5. OpenSBI Configuration for NOEL

The OpenSBI is enabled and configured as part of the Buildroot configuration under the bootloader section for NOEL. It is downloaded and installed automatically.

3.6. MKPROM2 Configuration for LEON

The optional MKPROM2 bootloader is enabled and configured as part of the Buildroot configuration under the bootloader section for LEON. Just as regular Buildroot packages, it is downloaded and installed automatically.

3.7. Building

When all configuration has been done, build everything with

```
make
```

The resulting images will by default be placed in the `output/images`. For example:

Table 3.2. Produced in `output/images`

File	Description
`image.ram`	Executable RAM file produced by MKLINUXIMG for LEON.
`fw_payload.elf`	Executable RAM file produced by OpenSBI for NOEL.
`vmlinux`	Kernel file. Use as symbol file for kernel debug. Found in kernel build directory for NOEL.
`image.prom`	Executable ROM file from MKPROM2 for LEON.
`rootfs.cpio`	CPIO Root file system, suitable for initramfs.

3.7.1. Rebuilding

When adding new packages via the Buildroot configuration, or reconfiguring the kernel via the Buildroot make system, it is often enough to just do

```
make
```

but sometimes that is not enough. See the section “Understanding how to rebuild packages” in the regular Buildroot manual for more details. See also the section “Understanding when a full rebuild is necessary” in the regular Buildroot manual for details on rebuilding and when that is necessary.

Beware of lost Linux kernel configurations (and other configurations done via separate configuration targets) on a full rebuild. See Section 4.2 on how to prevent that.

3.8. Project-specific customization

It is recommended to read the chapter about “Project-specific customization” in the Buildroot manual as it describes best practices when customizing Buildroot for a project. In Buildroot there are several config options available for customizing the generated target filesystem. Here follows a breif introduction to some of these options.

The layout of the final rootfs can be customized using filesystem overlays. A filesystem overlay is a tree of files that is copied directly over to the target filesystem after it has been built. The rootfs overlay is enabled by providing a list of paths (space-separated) to the config option BR2_ROOTFS_OVERLAY.

If the system need to set specific permission or ownership on files or device nodes this can be achived by providing a list (space-separated) of paths to permission tables to the config option BR2_ROOTFS_DEVICE_TABLE. Custom user accounts can be added by providing a list (space-separated) of paths to user tables to the config option BR2_ROOTFS_USER_TABLES.

Post-build scripts are shell scripts called after Buildroot builds all the selected software, but before the rootfs images are assembled. The post-build scripts can be used to modify or delete files on the target filesystem or by running extra commands before generating the file system image. Enable this by providing a list (space-separated) of post-build script paths to the config option BR2_ROOTFS_POST_BUILD_SCRIPT.

There is also a similiar option for post-image scripts, which are run after all images have been created and it is enabled by providing a list (space-separated) of post-image script paths to the config option BR2_ROOTFS_POST_IMAGE_SCRIPT.

For certain packages it is possible to customize features and options using fragment files. The options specified in the fragment files are merged with the main configuration file during the build. For example the Linux kernel can be configured by providing a list of space-separated paths to fragment files to the config option BR2_LINUX_KERNEL_CONFIG_FRAGMENT_FILES. Another example is BusyBox which can be configured by providing a list of space-separated paths to fragment files to the config option BR2_PACKAGE_BUSYBOX_CONFIG_FRAGMENT_FILES.

3.8.1. Project customization example

In Buildroot there is a mechanism called “br2-external”. This mechanism allows to keep package recipes, board support, rootfs overlays, configuration files etc outside of the Buildroot tree in an external location but still having it integrated in the build logic. The location of this kind of structure is refered to as “br2-external tree”. Buildroot can then be instructed to include one (or more) “br2-external tree(s)” by setting the BR2_EXTERNAL (to the path(s) of the trees) variable when invoking make. Buildroot include custom package recipies specified in the files “Config.in” and “external.mk” of the “br2-external tree”. These packages can then be configured under “External options” menu using Buildroot's top-level configuration menu.

An example of a br2-external tree is available in `docs/gaisler/br2-external-example.tar.bz2` and this section shows an example of how to integrate it in Buildroot.

The example shows how to use the config options mentioned in Section 3.8 such as customizing the rootfs by using a filesystem overlay, adding a custom user and setting ownership. It also include a post-build script which modifies a file in the target file system and fragment files for Linux and BusyBox. The example also includes a custom package recipe for a helloworld application. The system will get the following customization:

- A custom user, bob, is added
- The filesystem overlay adds /home/bob and /etc/buildtime
- Ownership of /home/bob is set to bob
- The post-build script file will modify /etc/buildtime
- BusyBox will include the applet **timeout**
- The Linux kernel will have timestamps enabled for printk
- A custom package, helloworld, is enabled and installed.

The example include a defconfig called “my-example_defconfig”and it contains all the config options to achieve the customization mentioned above:

```
BR2_ROOTFS_OVERLAY="$(BR2_EXTERNAL_PROJECT_EXAMPLE_PATH)/board/my-board/overlay/"
BR2_ROOTFS_POST_BUILD_SCRIPT="$(BR2_EXTERNAL_PROJECT_EXAMPLE_PATH)/board/my-board/post-build.sh"
BR2_ROOTFS_DEVICE_TABLE="system/device_table.txt $(BR2_EXTERNAL_PROJECT_EXAMPLE_PATH)/board/my-board/device_table.txt"
BR2_ROOTFS_USERS_TABLES="$(BR2_EXTERNAL_PROJECT_EXAMPLE_PATH)/board/my-board/user_table.txt"
BR2_LINUX_KERNEL_CONFIG_FRAGMENT_FILES="$(BR2_EXTERNAL_PROJECT_EXAMPLE_PATH)/board/my-board/kernel.config"
BR2_PACKAGE_BUSYBOX_CONFIG_FRAGMENT_FILES="$(BR2_EXTERNAL_PROJECT_EXAMPLE_PATH)/board/my-board/busybox.config"
BR2_PACKAGE_HELLOWORLD=y
```

The option BR2_EXTERNAL_PROJECT_EXAMPLE_PATH (as can be seen in the paths above) is a special option created by Buildroot when using a “br2-external tree”. Buildroot gets the name from a file called `external.desc` which should be located at the path given to the BR2_EXTERNAL variable when invoking **make**. This is the content of `external.desc` from the example:

```
name: PROJECT_EXAMPLE
desc: Project example of a br2-external tree
```

Note that when changing the name in `external.desc` all references must be updated where it is used.

Here follows instructions on how to integrate the example:

Enter the Buildroot top directory, extract the archive and place it in an external directory.

```
tar xf docs/gaisler/br2-external-example.tar.bz2 -C <dir_path>
```

In this example the project will be based on the default configuration “gaisler_leon_defconfig” and the board config “my-board_defconfig” from the br2-external example tree. The mentioned board configs needs to be merged before configuring Buildroot. There is a support tool available for merging config files in `support/kconfig/merge_config.sh`

```
CONFIG_=BR2_ support/kconfig/merge_config.sh -m configs/gaisler_leon_defconfig \
             <dir_path>/br2-external-example/configs/my-board_defconfig
```

Buildroot needs to be configured and aware about the location of the br2-external example tree. This is done once by setting BR2_EXTERNAL when invoking a **make** target that configures Buildroot. Open up the configuration editor, save the changes and exit, for example, if using xconfig:

```
make BR2_EXTERNAL=<dir_path>/br2-external-example xconfig
```

Now the configuration is done and it's possible to build a system image by invoking **make**. Buildroot will remember BR2_EXTERNAL so there is no need to pass it at every **make** invokation.

3.9. NFS Examples

With NFS it is possible to access and share files with others over a computer network. The first example cover how to mount a NFS share. This is very convenient in some situations, for example, when the system needs access to shared data or if the data is too large to be built-in to the image. The second example covers how the kernel can boot and use a root file system from an NFS share instead of using a RAM file system.

The examples are based on the Buildroot configurations mentioned in Section 3.1. For a custom configuration, make sure that the Linux kernel has been configured with:

- Aeroflex Gaisler GRETH Ethernet MAC support (GRETH)
- NFS client support (NFS_FS)

It is fundamental that the client can reach the server. Make sure that it is possible to configure the network interface of the client so it is possible to **ping** the intended NFS server.

```
# Set up the mac-address of the device to avoid collisions
ifconfig eth0 hw ether <mac address>
# Assign an IP and enable the eth0 interface
ifconfig eth0 <NFS client IP address> up
# Ping the NFS server
ping <NFS server IP address>
```

3.9.1. How to mount an NFS filesystem

3.9.1.1. Server setup

The first step is to setup and configure an NFS server:

- Install the NFS server software
- Start the NFS server
- Configure access to the NFS server

These steps might differ depending on the running system of the server. An example of how this can be done on a Ubuntu based system will be covered, but in general, when using a major Linux distribution, the necessary NFS server software is installed using the package manager of the running distribution. After the NFS server has been installed and started the final step is to configure it so clients are allowed to access the shared directories. The NFS server is configured using a file called “exports” which usually is installed in `/etc` and it controls which directories that can be shared over NFS. The general syntax of an entry in `/etc/exports` is:

```
<export> <host>(<options>)
```

- **export** is the directory being exported.
- **host** is the host or network to which the export is being shared.
- **options** contains the options set for the host or network.

A change to `/etc/export` is applied by invoking:

```
sudo exportfs -ra
```

In this example the following entry is used:

```
/home/export *(rw,sync,no_subtree_check,no_root_squash)
```

Table 3.3. Description of the options used in the example

Option	Description
/home/export	The directory that will be exported.
*	Allow all incoming IP addresses. This can be restricted to a single IP address or to a range.
rw	Mounts of the exported file system is read+writeable. The file system can be made readonly by setting it to “ro”.
sync	Replies to requests will wait until after changes to be committed to stable storage has been made.
no_subtree_check	Disables subtree checking as it might cause problems with accessing files that are renamed while a client has them open.
no_root_squash	Allows the root of the client to access the share as the local root (of the NFS server). This setting could be a potential security risk so it is recommended that it is only used in trusted environments and that the exported directories are not important to the server (i.e limit the data share to what is needed by the client).

The complete syntax of the exports file can be read in the man page:

```
man exports
```

Here follows an example on how to setup an NFS server on a Ubuntu based system.

Start by installing the NFS server software:

```
sudo apt install nfs-kernel-server
```

The next step is to start the NFS server:

```
sudo systemctl start nfs-kernel-server.service
```

Then the final step is to configure the NFS server by adding this entry to the `/etc/exports` file:

```
/home/export *(rw,sync,no_subtree_check,no_root_squash)
```

The change needs to be applied:

```
sudo exportfs -ra
```

Now the directory is ready to be accessed by a client through the NFS server.

3.9.1.2. Client setup

On the client side the only thing that is required is to setup the network interface:

```
# Set up the mac-address of the device to avoid collisions
ifconfig eth0 hw ether <mac address>
# Assign an IP and enable the eth0 interface
ifconfig eth0 <IP address> up
```

The client can now to mount the network share:

```
mount <NFS Server IP>:/home/export /mnt -onolock>
```

Please note that the **mount** request might fail if Buildroot has been configured with the package BR2_PACKAGE_UTIL_LINUX_MOUNT enabled. If that is the case then the package BR2_PACKAGE_NFS_UTILS needs to be enabled. By default the **mount** command comes from the Busybox application suite and it works without adding any additional packages.

3.9.2. How to boot Linux from NFS

This example shows how a system can be configured to use a root filesystem from a network share instead of the built-in RAM file system (initramfs) which is typically used in the configs mentioned in Section 3.1. This allows for a flexible system where different root file systems can be used with the same kernel build.

3.9.2.1. Buildroot configuration

The first step is to configure Buildroot. Open up the configuration editor, for example, if using xconfig:

```
make xconfig
```

Under “Filesystem images”select:

- “tar the root filesystem ”(BR2_TARGET_ROOTFS_TAR)

Under “Filesystem images” **unselect**:

- “initial RAM filesystem linked into linux kernel” (BR2_TARGET_ROOTFS_INITRAMFS)

The final step is to add a couple of NFS root related parameters to the kernel command line so the kernel can set up the network interface and mount the NFS share on boot. Here is an overview of the parameters:

```
root=/dev/nfs
nfsroot=[<server-ip>:]<root-dir>[,<nfs-options>]
ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<autoconf>:<dns0-ip>:<dns1-ip>:<ntp0-ip>
```

See `Documentation/admin-guide/nfs/nfsroot.rst` in the Linux kernel source for more information about the parameters.

Here is an example of a configuration:

- server-ip: 192.168.0.69
- client-ip: 192.168.0.244
- gw-ip: 192.168.0.1
- netmask: 255.255.255.0
- hostname: soc
- device: eth0
- root-dir: /home/export/nfsroot
- nfs-options: nolock,tcp,vers=3

The expanded parameters would then look like this:

```
root=/dev/nfs
nfsroot=192.168.0.69:/home/export/nfsroot,nolock,tcp,vers=3
ip=192.168.0.244:192.168.0.69::255.255.255.0:soc:eth0:none:192.168.0.1 rw
```

Under “Bootloaders”under “mklinuximg” append the NFS parameters (space separated) to:

- “Commandline to pass to the kernel” (BR2_TARGET_MKLINUXIMG_CMDLINE)

Exit the Buildroot configuration tool and save the changes.

3.9.2.2. Linux kernel configuration

The second step is to configure the kernel. Open up the Linux configuration editor, for example, if using xconfig:

```
make linux-xconfig
```

Under “File systems” under “Network File systems”select:

- “Root file system on NFS” (ROOT_NFS)

Under “Device Drivers” under “Generic Driver Options” select:

- “Maintain a devtmpfs filesystem to mount at /dev” (DEVTMPFS)
- “Automount devtmpfs at /dev, after the kernel mounted the rootfs ”(DEVTMPFS_MOUNT)

Under “General setup” **clear**:

- “Initramfs source file(s)” (INITRAMFS_SOURCE)

Under “General setup” **unselect**:

- “Initial RAM filesystem and RAM disk (initramfs/initrd)” (BLK_DEV_INITRD)

Everything is now configured for NFS boot. Exit the kernel configuration tool and save the changes.

3.9.2.3. Build & deploy

The next step is to build and unpack the root filesystem to the nfs root directory. In the example `/home/export/nfsroot` is used as the root directory. It is expected that `/home/export` has been exported as described in the NFS mount example (see Section 3.9.1).

Start a build by invoking:

```
make
```

Now extract the generated `rootfs.tar` from `output/images` to the nfs-root directory (note that **sudo** is used):

```
sudo tar -xavf output/images/rootfs.tar -C /home/export/nfsroot
```

Now everything is ready for the NFS boot. On a successful boot the following output is printed:

```
IP-Config: Complete:
     device=eth0, hwaddr=00:00:7c:cc:01:45, ipaddr=192.168.0.244, mask=255.255.255.0, gw=255.255.255.255
     host=soc, domain=, nis-domain=(none)
     bootserver=192.168.0.69, rootserver=192.168.0.69, rootpath=
     nameserver0=192.168.0.1
VFS: Mounted root (nfs filesystem) on device 0:12.
```

If the following is printed when the network service is started:

```
Starting network: ip: RTNETLINK answers: File exists
FAIL
```

The output indicates that the NETLINK interface already has been setup and it will not cause any issues on the running system. This happens because the localhost interface is setup by the kernel during boot. It is possible to avoid the warning by using the ifupdown package in Buildroot (BR2_PACKAGE_IFUPDOWN). Before it can be selected the package BR2_PACKAGE_BUSYBOX_SHOW_OTHERS must be selected.

4. Building the Linux Kernel in Buildroot

This chapter touches on matters related to setting up, configuring and building the Linux kernel specifically in the Buildroot environment. For our general documentation on LEON Linux, see the LEON Linux User's manual [https://www.gaisler.com/doc/leon-linux.pdf]. As usual, the regular Buildroot manual is the main source of information on these matters. This chapter aims to give some helpful pointers and tips.

4.1. Linux Kernel Configuration

The kernel can be configured using e.g.

```
make linux-xconfig
```

or

```
make linux-menuconfig
```

See the regular Buildroot manual for details and alternatives to the above configuration methods. See for general LEON Linux kernel documentation including specific kernel configuration options relatedfor our drivers and kernel subsystems.

4.2. Caveats on Loosing the Kernel Configuration

The Linux kernel configuration file is by default placed in `output/build/linux-`*`version`*`/.config`, where *`version`* corresponds to the kernel version being used. It is important to note that when doing

```
make clean
```

the Linux kernel configuration file will be removed together with everything else that is cleaned up, due to the fact that it resides in the `output/build` directory.

To save the Linux kernel configuration, configure *Buildroot* in the Kernel section, like so:

- Select the “Using a custom (def)config file” option.
  - BR2_LINUX_KERNEL_CUSTOM_CONFIG
- Set the “Configuration file path” to a filename (outside of the `output` directory) where you want to save the kernel configuration. A relative path will put it relative to the Buildroot top directory.
  - BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE
- Clear the “Additional configuration fragment files” if is set to anything.
  - BR2_LINUX_KERNEL_CONFIG_FRAGMENT_FILES

and then save the configuration to the configured destination with

```
make linux-update-config
```

to save the full configuration file, or

```
make linux-update-defconfig
```

to save a slimmed down configuration file with only changes from default values.

This procedure is not automatic, and one of the two last steps needs to be repeated after changing the Linux kernel configuration for those changes to survive a `make clean`.

Other packages with their own *`pkgname`*`-xconfig` configuration target (or similar) can be affected in the same way. See the section on “Storing the configuration of other components” in the regular Buildroot manual for details.

4.3. Setting up external Linux kernel source directory

When Buildroot is configured to build the kernel, as is the case in our default configurations, the kernel source code is downloaded and patched. This is fine when not having any needs to change any Linux kernel source code. When actively doing kernel development it is better to set the kernel source in an external (to Buildroot) directory and configure Buildroot to use that.

The LEON Linux kernel releases can be downloaded as separate packages and set up a git work directory with the LEON Linux kernel as a git branch. This can be more suitable as a kernel source setup for kernel development.

For NOEL, the patches under `board/gaisler/noel-common/patches/linux/`*`VER`* where *`VER`* is the base kernel version, is the canonical source for our Linux kernel patches for NOEL. the kernel can be cloned from official sources and those paches can be applied on top of the corresponding upstream kernel version.

When the kernel sources has been set up externally, create a `local.mk`, in the same directory as the Buildroot `.config` file, containing

```
LINUX_OVERRIDE_SRCDIR = /path/to/linux-src
```

filling in the path to the checked Linux kernel source tree. This will make Buildroot use these source files instead, by syncing the sources to its build directory and. To rebuild after a kernel source code change, do

```
make linux-rebuild all
```

to make sure that both the sources are synced properly and that the entire build chain is triggered.

Note that the caveats in Section 4.2 still applies when using external sources. See the “Using Buildroot during development” section in the regular Buildroot manual for more details on setup up external sources in general.

5. Toolchains

The regular Buildroot manual is the main source of information on the various options for toolchains under Buildroot. This chapter gives additional information on specific support from our side.

5.1. External Toolchains

For both LEON and NOEL, in our Buildroot releases we support using the latest external toolchain released by us and have it automatically downloaded and used. Our default Buildroot configuration (defconfig) files are set up to use this by default. Although we make it possible to use and build other toolchains, this toolchain choice is our officially supported one.

A different version than the one that was the latest one at the time of the Buildroot release can be used via the normal external Buildroot configuration methods.

5.2. Buildroot-built toolchains

Buildroot can be set up to build a toolchain from scratch. This is what the regular Buildroot manual calls internal toolchain. This is the Buildroot default when not using any of our default configurations.

For both LEON and NOEL, our Buildroot release makes is possible to build an internal toolchain with the same GCC and Glibc versions as our external toolchain, including our extra patches on top of the upstream versions they are based upon.

5.3. Architecture choices for LEON

For LEON we support choosing between “leon3” and “leon5” as SPARC architecture variants in Buildroot. The former is suitable for LEON3 and LEON4, but also works fine with LEON5. The latter has instruction timing tuned to LEON5 specifically.

5.3.1. Errata workarounds for UT700

When choosing “leon3” as architecture we also support enabling errata fixes for UT700. This can be done both using our external toolchain, or building a Buildroot-built toolchain when our specific GCC and Glibc versions are selected.

6. Upgrading Buildroot

6.1. Upgrading to a new Gaisler Buildroot release

These are the basic steps for upgrading from an one Gaisler Buildroot release to another.

- Unpack the new Gaisler Buildroot release as per Section 1.3.
- Preserve your Linux kernel config as per Section 4.2.
- Preserve other external configurations and additions. See the “Quick guide to storing your project-specific customizations” section in the regular Buildroot manual for details. Note that these guidelines suggests storing them in-tree, but it is also possible to store them out of tree with absolute paths.
- Copy the Buildroot .config configuration file to the new Buildroot base directory (or external output directory if using that).
- Copy over any user added in-tree files or directories referred to by the Buildroot .config to the new directory.
- Rebuild from the new installation.

6.2. Follow upstream stable branch

This section gives some pointers on how to follow an upstream stable Buildroot branch instead of waiting for a new Gaisler Buildroot release from us. See Section 6.1 on how to preserve and move configurations from one Buildroot directory to the new one created by the procedure below.

The example below shows the procedure from the point of view of Gaisler Buildroot 2024.02-1.2 that is based upon upstream Buildroot release 2024.02.6. That the upstream release in this case is 2024.02.6 can be seen in the `CHANGES` changelog file in `docs/prebuilt`.

Check out the upstream Buildroot release that the Gaisler Buildroot release that you are using is based on, e.g.

```
git clone git://git.buildroot.net/buildroot
cd buildroot
git checkout -b branchname 2024.02.6
```

Create a git commit out of the Gaisler Buildroot release. Adjust path to archive as needed.

```
git checkout -b custom-gaisler-buildroot 2024.02.6
tar xf /tmp/gaisler-buildroot-2024.02-1.2.tar.bz2 --strip-components=1
sed -i '/export BR2_VERSION :=/ s/-.*$//' Makefile
git add *
git commit -m "Gaisler Buildroot 2024.02-1.2 additions"
```

Rebase upon the later upstream release, in this example 2024.02.7.

```
git rebase 2024.02.7
```

In case of rebase conflict, standard git procedures apply. Make sure to include the **sed** command above to avoid an otherwise highly probable rebase conflict.

7. Support

For support contact the support team at support@gaisler.com.

When contacting support, please identify yourself in full, including company affiliation and site name and address. Please identify exactly what product that is used, specifying if it is an IP core (with full name of the library distribution archive file), component, software version, compiler version, operating system version, debug tool version, simulator tool version, board version, etc.

The support service is only for paying customers with a support contract.

Frontgrade Gaisler AB
Kungsgatan 12
411 19 Göteborg
Sweden
frontgrade.com/gaisler
sales@gaisler.com
T: +46 31 7758650
F: +46 31 421407