Compiling Individual Packages

You may know me from former titles such as “Batocera Linux Buildroot Modifications” and “How to work on Batocera (and not recompile everything)”.

If you'd instead like to only test certain config files (like es_features.cfg), you can edit them directly in the live system and save them via overlays.

Compiling the entirety of Batocera is optional, but recommended as it automatically sets up your enviroment for compiling individual packages. batocera.linux is based on Buildroot, manual available here.

Package compilation

For each package (ie. **kodi**, **emulationstation**), Buildroot does the following:

1. do prerequisites
2. download
3. extract
4. patch
5. configure
6. build
7. install staging
8. install target

More exactly:

1. do the prerequisite packages recursively if not yet done (ie **sdl2** if you want to build **emulationstation**)
2. download the package in the dl directory according to information from `package/package-name/package-name.mk` or `package/batocera/package-name/package-name.mk`
3. extract the package in `output/build/package-name-version`
4. patch the package from `*.patch` files (files next to the `.mk` file or `board/batocera/patches/` or `board/batocera/rpi/patches/`)
5. configure the package according to the `.mk` file
6. build the package according to the `.mk` file
7. install in the `output/rpi3/host/` staging directory, in case of libraries
8. install in the `output/rpi3/target/` directory

For each step, Buildroot creates an empty flag file. For example: `.stamp_downloaded` in `output/rpi3/build/package-name-version`.

In case a step failed, just run `make` or `make package-name` again to restart/continue the build where you left.

To completely rebuild a package:

```
rm -rf output/rpi3/build/package-name-version
make package-name
```

For example, to build EmulationStation again from the latest `master` git version:

```
rm -rf output/rpi3/build/batocera-emulationstation-master
make batocera-emulationstation
```

Individual packages can also be compiled using the container using:

```
make x86_64-pkg PKG=batocera-splash
```

replacing the components as necessary.

Main directories/files

When you clone the batocera.linux git repository, you mainly get the following directories:

- `.config`: the list of all packages, stating if you are going to build it or not.
- `dl`: the directory where all files are downloaded, for example, `batocera-emulationstation-master.tar.gz`
- `output`: the directory where everything is built.

In other words, in order to reinitialize your build, you can type:

```
rm -rf .config dl output
```

I personally use the following command to confirm I have no other remaining files:

```
git status --ignored
```

My `dl` directory is a link on `../DL` in order to use the same download directory for all my builds, to avoid to download the source archives each time again and again. Note that you may have to remove some packages like `batocera-emulationstation-master.tar.gz` or `batocera-configgen-master.tar.gz` while the contains is the master branch of their respective git repositories. In other words, whenever the content is dynamic (`master`) and not a precise GitHub version. If you commit something on `configgen`, while the file is in the `dl` directory, buildroot will not download it again as it is seen as the same filename.

Working on packages

Let's say you want to work on the package “retroarch” to make it evolve via a patch.

First, compile a version of Batocera for the architecture you will test.

Once this is done, you don't need to build it for your tests again. Just run the following commands:

```
mv output/rpi3/target output/rpi3/target_
cd output/rpi3/build/retroarch-version
git init
git add <the file(s) you want to modify>
# edit the file(s)
git diff > ../../package/batocera/retroarch/001-mymodification.patch
cd ../../..
rm -rf ouput/rpi3/build/batocera/retroarch-version
make retroarch-patch # to confirm the patch is applied
make retroarch # to finish other steps
```

At this step, output/target contains only the retroarch package. You can install it on your target machine via the command:

```
rsync -av output/target/ root@batocera.local:/
```

Remember, Batocera system modification are done on the Linux overlay, i.e. in RAM. Rebooting Batocera without saving your overlay acts as if you hadn't changed it – in other words, it's a nice way to make tests while your system is not modified, but those changes are not persistent. If you want to permanently save the modification and keep them upon reboot, you need to use batocera-save-overlay from the machine itself.

Working on EmulationStation

EmulationStation is a separate git repository from Batocera system. To work on EmulationStation, the simplest way is to directly get it on your computer and compile it without buildroot.

How to add an individual package

This is for generic packages that aren't necessarily emulators.

The following is a (very) brief summarization of chapter 19 of the Buildroot manual. It is recommended to read that in full in case of any questions.

Todo:

- Explain the differences between each different kind of package container (generic, cmake, autotools, etc.).
- Provide rough syntax for each one.
- Cover all stages.
- Define when certain steps aren't absolutely necessary.

1. Add the package's folder to the appropriate location. For generic packages like this, it will most likely be `package/batocera/utils/package-name/`.

It's worth checking if the package you want is already included in buildroot, just check in the `buildroot/package/Config.in` file. If so, skip steps 2 and 3.

2. In that folder, create the `Config.in` file. This should contain the package's "metadata" for compilation purposes. Add into this file its:
    1. Buildroot package header: `config BR2_PACKAGE_<package>`
    2. Its bool shortname used for compilation and other packages it may depend on: `bool "<package shortname>"`
    3. Specify other packages depends on: `depends on BR2_PACKAGE_<other package>`
    4. A help section describing the package in more detail.

For more information on Buildroot's `Config.in` syntax, refer to chapter 16.1 in Buildroot's manual.

3. Create the makefile. Its filename should be the same as the package's bool shortname, followed by the `.mk` extension: `<package-shortname>.mk`
    1. If pulling from a Github source, use the following syntax:

```
<package>_VERSION = <commit hash>
<package>_SITE = $(call
github,<author>,<repository>,$(<package>_VERSION))
```

    2. Each step of package compilation needs to be put into its appropriate function.
        1. If needing to set particular config options while building, place them in `<package>_CONF += <options>`. For example:

```
GZDOOM_CONF_OPTS += -DCMAKE_BUILD_TYPE=Release -DDYN_GTK=OFF -
DDYN_OPENAL=OFF
```

        2. If needing to set particular environmental variables before building, place them in `<package>_CONF_ENV += <environment variables>`. For example:

```
GZDOOM_CONF_ENV += ZMUSIC_LIBRARIES="/x86_64/target/usr/lib/"
ZMUSIC_INCLUDE_DIR="/x86_64/target/usr/lib/cmake/ZMusic/"
```

        3. If compiling from the package's downloaded build folder (such as when downloading a Github repo), put this in `define <package>_BUILD_CMDS`. Inside of that function, be sure to use the appropriate flag to set the correct working directory: `$(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(@D)` The "make" compiler uses its own instance, which may not be sharing the same current working directory. A generic compilation script would be as follows:

```
define <package>_BUILD_CMDS
    $(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(@D)
```

```
endef
```

4. When compilation is finished, if not compiled as a part of a large over-arching package, copy over the resulting files/binaries to the appropriate locations in the install step (if the compilation process itself doesn't already do so). For example, when compiling an ordinary binary:

```
define <package>_INSTALL_TARGET_CMDS
    mkdir -p $(TARGET_DIR)/usr/bin
    cp $(@D)/output_executable $(TARGET_DIR)/usr/bin
endef
```

3. When done defining actions to perform for each step, end the makefile with one file line defining which package container to use. For “generic” packages: `$(eval $(generic-package))`.

For more information on Buildroot's makefile syntax, refer to chapter 16.2 in Buildroot's manual.

4. Add the package to the list of compiled packages at `package/batocera/core/batocera-system/Config.in`. This is where you would refer to its Buildroot package header, not its shortname.
5. Add the package to the list of sources in the appropriate section in `Config.in`. That's right, the file in the root directory of the repo.

An example of a recent pull request that adds a simple package:
https://github.com/batocera-linux/batocera.linux/pull/5812/files

How to add an emulator

1. Add the initial scripts for successful compilation. Ensure that there are no valid build errors with it. Refer to buildroot's documentation for further info.
    1. Create its initial configuration script: https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/emulators/<new emulator>/Config.in
    2. Call it from https://github.com/batocera-linux/batocera.linux/blob/master/Config.in
    3. Create its makefile for package compilation: https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/emulators/<new emulator>/<new emulator>.mk
    4. If the emulator requires certain files in certain locations in userdata, copy them to `/usr/share/batocera/data/`.
    5. If the emulator cannot bind functions to the gamepad's buttons and needs to use a virtual keyboard, add the appropriate evmapy `.keys` file to `<shortname>.<emulator>.keys`.
    6. If building from source, add the git repository check to the update script: https://github.com/batocera-linux/batocera.linux/blob/master/scripts/linux/checkPackagesUpdates.sh
2. Add your system/emulator to the EmulationStation system configuration at https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/emulationstatio

n/batocera-es-system/es_systems.yml (there needs to be at least one default core for the system, described later).
    - You can check a list of system shortnames that can be automatically scraped for here: https://github.com/batocera-linux/batocera-emulationstation/blob/master/es-app/src/PlatformId.cpp
3. Add your system/emulator to the features list (https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/emulationstation/batocera-es-system/es_features.yml) and any advanced system settings if necessary.
4. Create and test its config generator at https://github.com/batocera-linux/batocera.linux/tree/master/package/batocera/core/batocera-configgen/configgen/configgen/generators.
    1. Define the generator (syntax: `from generators.<shortname>.<shortname>Generator import <CamelCased shortname>Generator`, shortname is the system name) with its system shortname here: https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/core/batocera-configgen/configgen/configgen/emulatorlauncher.py (you can test this locally with `/usr/lib/python3.9/site-packages/configgen/emulatorlauncher.py`). The `commandArray` is ultimately the line that is used to run the emulator from bash.
    2. Include it in the packages list (syntax: `configgen.generators.<shortname>`) here: https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/core/batocera-configgen/configgen/setup.py.
    3. Create the “main” configuration generator Python script here: https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/core/batocera-configgen/configgen/configgen/generators/ followed by <shortname>/<shortname>Generator.py.
    4. If appropriate, split the file into multiple Python scripts called by <shortname>Generator.py in the `generators/<shortname>/` folder.
    5. Call the files if required here: https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/core/batocera-configgen/configgen/configgen/batoceraFiles.py.
5. Configure the default emulator (if you've added a whole new system) with https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/core/batocera-configgen/configs/configgen-defaults.yml (if you're merely adding an emulator to an already existing system and you don't want to make it the default, no need to touch this file).
6. Configure the default settings for particular architectures (such as if your emulator requires certain settings to function on a particular architecture) at https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/core/batocera-configgen/configs/.
7. Add your BR2 package to the compilation process (and limit your emulator to particular archs (typically, complex standalone emulators are x86/x86_64 only)) in https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/core/batocera-system/Config.in.
8. If BIOS files are required, set them in https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/core/batocera-scripts/scripts/batocera-systems
9. (Optional) If you need to add an explanation for the emulator's exclusion from a particular platform for the compatibility chart, add it to https://github.com/batocera-linux/batocera.linux/blob/master/package/batocera/emulationstation/batocera-es-system/systems-explanations.yml

If you'd like an example of a recent pull request that adds a whole new emulator:
https://github.com/batocera-linux/batocera.linux/pull/4742/files