CONTENTS:

1 About moulin
  1.1 Main purpose ........................................ 1
  1.2 Design .............................................. 1
  1.3 Requirements and Installation ...................... 1

2 Quickstart
  2.1 YAML format ......................................... 3
  2.2 Simple Android build ................................. 3
  2.3 Parameterized Yocto build ............................ 4
  2.4 Parameterized build with multiple images .............. 6

3 Moulin User reference manual .......................... 13
  3.1 Invoking moulin .................................... 13
  3.2 YAML sections ..................................... 14
  3.3 Fetchers ............................................. 16
  3.4 Builders ............................................ 19

4 Rouge User Reference Manual ......................... 25
  4.1 About ............................................... 25
  4.2 Design Principles .................................. 25
  4.3 Requirements ...................................... 25
  4.4 Invoking rouge ..................................... 26
  4.5 Principles of Operation .............................. 26
  4.6 YAML Sections ..................................... 26
  4.7 Block descriptions .................................. 27
  4.8 Examples ............................................ 30

5 Usage notes ............................................. 31
  5.1 Component-specific targets ........................... 31
  5.2 Rouge-specific targets ............................... 31
  5.3 Build inside yocto ................................... 31
Main purpose of moulin is to build complex images for embedded devices. Imagine that you are running hypervisor (like Xen or KVM) on your device and you want to build multiple VM images with one simple command. moulin is made exactly for this task.

But moulin also can be used to build simpler projects like standalone Yocto distribution or AOSP. As moulin project file includes both code location and build instructions, it is much easier to invoke moulin myproject.yaml to get the complete build than to fumble with Yocto layers and configs.

1.1 Main purpose

In modern world even embedded systems can benefit from virtualization technologies. For example, ARMv8 architecture have built-in virtualization extensions, so both Xen hypervisor and KVM can run on variety of ARMv8-based SoCs. In embedded world it is natural to have some way to build the whole bootable image with one command. But there is no easy way to build multiple different virtual machine images. For example, we want to build Linux-based host VM, Android and another Linux distro as a guest VM. moulin was created exactly for this purpose.

Most of the modern BSPs are Yocto-based Linux distributions with vendor-specific changes. Mobile chips likely will have Android Open Source (AOSP) support as well. So right now moulin is focuses on supporting this two build systems. More will be added later.

1.2 Design

moulin is not a replacement for make or bitbake. It merely processes project YAML file and generates Ninja build file. So, actual job runner is the Ninja.

On other hand, moulin project files describe the whole build, with source code locations, build options, dependencies between images and optional parameters.

1.3 Requirements and Installation

moulin requires Python 3.6+ to run. You might need pip or pip3 tool to install it.

moulin generates build.ninja compatible with ninja 1.10+. This is because lower versions of ninja do not support multiple outputs. So if you run ninja and see error message like this

```
multiple outputs aren't (yet?) supported by depslog
```
you need to get newer version from https://github.com/ninja-build/ninja/releases.

Also moulin requires pygit2. If it is not installed on your system, pip will try to install it from repository. It may fail in case if you are missing libgit2 development files or if installed libgit2 is too old. So, it is better to install pygit2 globally. For example, on Ubuntu 18.04 you need to install python3-pygit2 package.

moulin source code is stored at GitHub.
Preferred way to install it right now is to put it your local per-user storage using

```
pip3 install --user git+https://github.com/xen-troops/moulin
```

command. Make sure that your PATH environment variable includes /home/${USER}/.local/bin.

You can encounter problem with pygit2 installation on some Ubuntu or Debian distribution due to this bug. In this case you will need to use the following command

```
PIP_IGNORE_INSTALLED=0 pip3 install --user git+https://github.com/xen-troops/moulin
```

as a workaround.

Alternatively, you can clone the mentioned git repository and invoke moulin.py directly.
This document provides a few examples to give you the feel.

## 2.1 YAML format

YAML (yet-another-markup-language) is a quite simple tree-like file format that can be easily read by humans and machines. If you are not familiar with it, you can take a look at the official specification or read any of multiple guides that are available in the Internet.

For the most of this documentation we assume that you can read and understand YAML files.

## 2.2 Simple Android build

There is example of minimal AOSP build:

```yaml
desc: "Build Android 11 XenVM image for Renesas Rcar H3"
components:
  android:
    sources:
      - type: repo
        url: https://github.com/xen-troops/android_manifest.git
        rev: android-11-master
        manifest: doma.xml
        depth: 1
        groups: all
        dir: ".
    builder:
      type: android
      env:
        "TARGET_BOARD_PLATFORM=r8a7795"
      lunch_target: xenvm-userdebug
      target_images:
        "out/xenvm/userdebug/boot.img"
        "out/xenvm/userdebug/system.img"
```

Let's discuss what is going there. This project file consist of two mandatory sections. **desc** provides description for the build. In this case it tells us that we will build very specific variant of Android 11.

**components** section is the main part of the file. It describes only one component, which is **android**, of course. Every component consists of two main parts: **sources** and **builder**.
sources describe all code sources that should be fetched prior the building. In this particular case we have only one repo - based source.

builder section configures the actual build. In this example we are building Android, so it has Android-specific options like lunch_target.

If the code above is stored in file android-vm.yaml, you can issue moulin android-vm.yaml. This will create build.ninja file in the same directory. After that, just run ninja and it will use repo to fetch given manifest, synchronize sources and start the build.

2.3 Parameterized Yocto build

Now let's consider more complex example:

```
desc: "Xilinx BSP"

variables:
    MACHINE: "this will be overwritten by parameters"

components:
    xilinx-bsp:
        sources:
            - type: git
              url: "git://git.yoctoproject.org/poky"
              rev: gatesgarth
            - type: git
              url: "git://git.yoctoproject.org/meta-xilinx"
              rev: gatesgarth
        builder:
            type: yocto
            conf:
                [MACHINE, "%{MACHINE}"]
        build_target: core-image-minimal
        layers:
            - "../meta-xilinx/meta-xilinx-bsp"
            - "../meta-xilinx/meta-xilinx-standalone"
            - "../meta-xilinx/meta-xilinx-contrib"
        target_images:
            - "tmp/deploy/images/%{MACHINE}/core-image-minimal-%{MACHINE}.tar.gz"
            - "tmp/deploy/images/%{MACHINE}/zImage"
            - "tmp/deploy/images/%{MACHINE}/boot.bin"

parameters:
    MACHINE:
        desc: "Xilinx device for which we will build"
        # Couple of machines are chosen at random just for demonstration purposes
        zc702-zynq7:
            default: true
            overrides:
                variables:
                    MACHINE: zc702-zynq7
        zc706-zynq7:
            overrides:
```

(continues on next page)
This build file allows you to build Xilinx BSP for one of selected devices (zc702-zynq7, zc706-zynq7, qemu-zynq7). Apart from early discussed desc and components sections we can see two new: variables and parameters. But let’s take a look at the component. In this case we are fetching two git repositories. Also we are building Yocto distribution now, we have completely different builder with different option. You can see familiar Yocto settings like list of layers on additional local.conf entries. All those sections are described in detail in reference manual.

variables section describes variables. This is basically strings that can be used in any other part of the file using ${VARIABLE_NAME} syntax. We are using % symbol instead of more familiar $ sign to ensure that it will not clash with bitbake’s or make’s variables. It is really annoying to escape dollar signs in constructions like the following:

```bash
conf:
- [SSTATE_DIR, "${TOPDIR}/../common_data/sstate"]
- [DL_DIR, "${TOPDIR}/../common_data/downloads"]
```

Please note that in this particular example topmost variables section can be omitted, because only one variable will be overwritten by subsequent parameters. But, generally you can define variables there. Also, one variable can refer to another:

```bash
variables:
A: "justA"
B: "%(A)_%(A)" # will be expanded to "justA_justA"
```

% itself can be escaped by doubling it:

```bash
variables:
A: "justA"
B: "%%(A)_%%(A)" # will be expanded to "%(A)_%(A)"
```

parameters section provides means to parameterize your build. If you have such section in your build file, you can get help using moulin itself:

```bash
$ moulin xilinx-bsp.yaml --help-config
usage: moulin.py xilinx-bsp.yaml [--MACHINE {zc702-zynq7,zc706-zynq7,qemu-zynq7}]
```

parameters section consist of one or more parameters, and each parameter can have a number of predefined values, one of which must have default flag set. User can select desired parameter variant with command line:

```bash
$ moulin xilinx-bsp.yaml --MACHINE qemu-zynq7
```

All entries from overrides of the chosen parameter will be applied on top of the build configuration. In this example, it will overwrite MACHINE variable with some meaningful value.

### 2.3. Parameterized Yocto build
2.4 Parameterized build with multiple images

The next example is the most complex one. It shows the main reason why moulin was written in the first place:

```
desc: "Renesas Gen3 build with Xen, Dom0 and DomD"
variables:
    MACHINE: "salvator-x-m3-xt"
common_data:
sources: &COMMON_SOURCES
    - type: git
      url: "/home/lorc/mnt/ssd2/yovrin-test2/src/poky"
      rev: 424296bf9bb4bae27f2b91bce0118df09ce5fa1
    - type: git
      url: "/home/lorc/mnt/ssd2/yovrin-test2/src/meta-virtualization"
      rev: 92cd3467502bd27b98a76862ca6525ce425a8479
    - type: git
      url: "/home/lorc/mnt/ssd2/yovrin-test2/src/meta-arm"
      rev: f7c5e75094f65d105d958b9a59527c5fb0d0f
    - type: git
      url: "/home/lorc/mnt/ssd2/yovrin-test2/src/meta-openembedded"
      rev: f2d02ebaf8ebeb25a1997b30be52486c10e6
conf: &COMMON_CONF
    - [SSTATE_DIR, "/{TOPDIR}/../common_data/sstate"]
    - [DL_DIR, "/{TOPDIR}/../common_data/downloads"]
# This is basically xt_shared_env.inc
# known domains
    - [XT_DIR_REL_DOM0, "dom0"]
    - [XT_DIR_REL_DOMD, "domd"]

# these are the folders within domain's root filesystem where all
# installed artifacts live
    - [XT_DIR_ABS_ROOTFS, "/xt"]
    - [XT_DIR_ABS_ROOTFS_DOM0, "/{XT_DIR_REL_DOM0}/"]
    - [XT_DIR_ABS_ROOTFS_DOMD, "/{XT_DIR_REL_DOMD}/"]
    - [XT_DIR_ABS_ROOTFS_DOMA, "/{XT_DIR_REL_DOMA}/"]
    - [XT_DIR_ABS_ROOTFS_DOMF, "/{XT_DIR_REL_DOMF}/"]
    - [XT_DIR_ABS_ROOTFS_DOMR, "/{XT_DIR_REL_DOMR}/"]
    - [XT_DIR_ABS_ROOTFS_DOMU, "/{XT_DIR_REL_DOMU}/"]

# these are folder names to be used accross domains to install
# various types of artifacts
    - [XT_DIR_ABS_ROOTFS_SCRIPTS, "/{XT_DIR_ABS_ROOTFS}/scripts"]
    - [XT_DIR_ABS_ROOTFS_CFG, "/{XT_DIR_ABS_ROOTFS}/cfg"]
    - [XT_DIR_ABS_ROOTFS_DOM_CFG, "/{XT_DIR_ABS_ROOTFS}/dom.cfg"]

components:
    dom0:
        # build-dir is optional
        build-dir: shared-build2
        sources:
            - *COMMON_SOURCES
```
builder:
  type: yocto
  work_dir: build-dom0
conf:
  - "COMMON_CONF"
  - [MACHINE, "generic-armv8-xt"]
  - ["PREFERRED_PROVIDER_virtual/kernel", "linux-generic-armv8"]

# Remove ptest to reduce the build time
  - [DISTRO_FEATURES_remove, "ptest"]

# For virtualization
  - [DISTRO_FEATURES_append, " virtualization"]
  - [DISTRO_FEATURES_append, " xen"]
# FIXME: normally bitbake fails with error if there are bbappends w/o recipes
  - [SERIAL_CONSOLES, ""]

# Disable shared link for GO packages
  - [XT_GUESTS_INSTALL, "domu"]

  - [MACHINEOVERRIDES_append, ":%{MACHINE}"],
  - [TUNE_FEATURES_append, " cortexa57-cortexa53"]

external_src:
  "domd-install-artifacts": "build-domd/tmp/deploy/images/%{MACHINE}/"

build_target: core-image-thin-initramfs

layers:
  - "../poky/meta"
  - "../poky/meta-poky"
  - "../poky/meta-yocto-bsp"
  - "../meta-arm/meta-arm-toolchain"
  - "../meta-openembedded/meta-oe"
  - "../meta-openembedded/meta-networking"
  - "../meta-openembedded/meta-python"
  - "../meta-openembedded/meta-filesystems"
  - "../meta-virtualization"
# Use inner layers of meta-xt-images and meta-xt-prod-devel
  - "../meta-xt-images/recipes-dom0/dom0-image-thin-initramfs/files/meta-xt-images-extra"
  - "../meta-xt-images/recipes-domx/meta-xt-images-domx/"
  - "../meta-xt-images/machine/meta-xt-images-generic-armv8"
  - "../meta-xt-prod-devel/recipes-dom0/dom0-image-thin-initramfs/files/meta-xt-prod-extra"

target_images:
  - "tmp/deploy/images/salvator-x/core-image-minimal-salvator-x.ext4"
  - "tmp/deploy/images/salvator-x/bl2-salvator-x.bin"
# Dependencies from other layers (like domd kernel image for example)
 additional_deps:
  - "build-domd/tmp/deploy/images/%{MACHINE}/Image"
domd:
  # build-dir is optional

(continues on next page)
build-dir: shared-build2

sources:
- *COMMON_SOURCES
  - type: git
    url: "//home/lorc/mnt/ssd2/yovrin-test2/src/meta-clang"
    rev: e63d6f9abba5348e2183089d6ef5ea384d7ae8d8
  - type: git
    url: "//home/lorc/mnt/ssd2/yovrin-test2/src/meta-python2"
    rev: c96cfe30701ba191903c5f7d560c3ba667d46c9d
  - type: git
    url: "//home/lorc/mnt/ssd2/yovrin-test2/src/meta-renesas"
    rev: c0a59569d52e32c26de083597308e7bc189675dd
  - type: git
    url: "//home/lorc/mnt/ssd2/yovrin-test2/src/meta-selinux"
    rev: 7af62c91d7d00a260cf28e790895539304d100d
  - type: git
    url: "//home/lorc/mnt/ssd2/yovrin-test2/src/meta-xt-prod-devel"
    rev: "REL-v6.0"
  - type: git
    url: "//home/lorc/mnt/ssd2/yovrin-test2/src/meta-xt-images"
    rev: "REL-v6.0"

builder:
  type: yocto

work_dir: build-domd

conf:
- *COMMON_CONF
  - [MACHINE, "{%MACHINE}"
  - [PREFERRED_VERSION_u-boot_rcar, "v2020.01%"
    # override console specified by default by the meta-rcar-gen3
    # to be hypervisor's one
  - [SERIAL_CONSOLES, "115200;hvc0"

  - [XT_GUESTS_INSTALL, "domu"

build_target: core-image-weston

layers:
- "../poky/meta"
- "../poky/meta-poky"
- "../poky/meta-yocto-bsp"
- "../meta-renesas/meta-rcar-gen3"
- "../meta-arm/meta-arm-toolchain"
- "../meta-openembedded/meta-oe"
- "../meta-openembedded/meta-networking"
- "../meta-openembedded/meta-python"
- "../meta-openembedded/meta-filesystems"
- "../meta-selinux"
- "../meta-virtualization"
- "../meta-clang"
- "../meta-python2"

# Use inner layers of meta-xt-images and meta-xt-prod-devel
- "../meta-xt-images/recipes-domd/domd-image-weston/files_rcar/meta-xt-images-
extra"
    - "../meta-xt-images/recipes-domx/meta-xt-images-domx/"
    - "../meta-xt-images/recipes-domx/meta-xt-images-vgpu/"
    - "../meta-xt-images/machine/meta-xt-images-rcar-gen3"
    - "../meta-xt-prod-devel/recipes-domd/domd-image-weston/files/meta-xt-prod-extra"

  target_images:
    - "tmp/deploy/images/%{MACHINE}/Image"

parameters:
  # Prebuilt DDK
  USE_PREBUILT_DDK:
    "no":
      default: true
      overrides:
        components:
          domd:
            sources:
              type: git
              url: "ssh://git@gitpct.epam.com/epmd-aepr/img-proprietary"
              rev: "ef1aa566d74a11c4d2ae9592474030a706b4cf39"
              dir: "proprietary"
            builder:
              conf:
                - [PREFERRED_PROVIDER_gles-user-module, "gles-user-module"]
                - [PREFERRED_VERSION_gles-user-module, "1.11"]
                - [PREFERRED_PROVIDER_kernel-module-gles, "kernel-module-gles"]
                - [PREFERRED_VERSION_kernel-module-gles, "1.11"]
                - [PREFERRED_PROVIDER_gles-module-egl-headers, "gles-module-egl-headers"]
                - [PREFERRED_VERSION_gles-module-egl-headers, "1.11"]
                - [EXTRA_IMAGEDEPENDS_append, " prepare-graphic-package"]

    "yes":
      overrides:
        components:
          domd:
            builder:
              conf:
                - [XT_RCAR_EVAPROPRIETARY_DIR, "./"
                - [PREFERRED_PROVIDER_virtual/libgles2, "rcar-proprietary-graphic"]
                - [PREFERRED_PROVIDER_virtual/egl, "rcar-proprietary-graphic"]
                - [PREFERRED_PROVIDER_kernel-module-pvrsrvkm, "rcar-proprietary-graphic"]
                - [PREFERRED_PROVIDER_kernel-module-dc-linuxfb, "rcar-proprietary-graphic"
                - [PREFERRED_PROVIDER_gles-user-module, "rcar-proprietary-graphic"]
                - [PREFERRED_PROVIDER_gles-user-module-gles-header, "rcar-proprietary-graphic"
                - [BBMASK_append, " meta-xt-images-vgpu/recipes-graphics/gles-module/"
                - [BBMASK_append, " meta-xt-prod-extra/recipes-graphics/gles-module/"
                - [BBMASK_append, " meta-xt prod-vgpu/recipes-graphics/gles-module/"
                - [BBMASK_append, " meta-xt-prod-vgpu/recipes-graphics/wayland/"}
This is an example of real build configuration file. It is still under development, so unlike two previous examples, you would not be able to use this example to make a build. We would not cover it in detail, just give you the list of highlights:

1. It builds two Yocto-based images: `dom0` and `domd`.

2. The same work directory used, so builds can share repositories with layers.
3. There is `common_data` section that provides some options that are shared by both builds: some source code and some `local.conf` entries.

4. `external_src` option is used to provide build artifacts from one component into another. In this way, `Dom0` image can include Linux kernel image generated by `DomD`.

5. There are two parameters: `MACHINE` and `USE_PREBUILT_DDK`. This allows user to chose target machine and build some proprietary drivers if they have access to corresponding repository.

6. Local `git` repos are used.
3.1 Invoking moulin

*moulin* has one mandatory parameter - file name of build description. It should be in YAML format. You may use a regular local file or URL. *moulin* detects URL by the presence of a protocol prefix, like *https://*. If you use the URL to GitHub or another network repository, you can use the URL for the raw file only, not for a web page with that file. For example, this URL points to the correct YAML file: `https://raw.githubusercontent.com/xen-troops/meta-xt-prod-devel-rcar/master/prod-devel-rcar.yaml`. But the following URL can't be used, because it points to GitHub's web page: `https://github.com/xen-troops/meta-xt-prod-devel-rcar/blob/master/prod-devel-rcar.yaml`. Pay attention, that file will be downloaded only if a file with the same name doesn't exist in the current folder. This is done to preserve possible local changes made by a user.

As a result, *moulin* will generate *ninja.build* file. You can then invoke *ninja* to perform the actual build. *moulin* adds *generator* rule in *ninja.build*, so it is not mandatory to invoke *moulin* after you made changes into your YAML file. *Ninja* will detect any changes to this file and invoke *moulin* automatically to re-create *ninja.build*.

If YAML file contains *parameters* key, it is possible to invoke *moulin* with additional command line options. Set of this options depends of contents of YAML file and can be viewed using `--help-config` command line option.

3.1.1 Verbose output

Use `-v` or `--verbose` command line option to increase verbosity of output. With this option enabled *moulin* will give more information about what it is doing.

3.1.2 Dumping intermediate state

`--dump` command line option can be used to force *moulin* to dump intermediate state of processed YAML file. You will see contents of your build config after applying all parameters and expanding all variables. This can come in help during debugging of your build, because you can see what exactly is passed to fetchers and builders.
3.1.3 Internal command line options

There is `--fetcherdep` command line option which is internal, and it is even hidden from `-h` output. It is used by `moulin` to generate dynamic dependency files for Ninja, so Ninja can track changes inside components.

This option is not meant to be used by a user.

3.2 YAML sections

YAML file should consist of number of pre-defined keys/sections which are discussed below. Any unknown keys are ignored. Right now only the following top-level keys are supported:

- desc - mandatory
- min_ver - optional
- components - mandatory
- images - optional. See `rouge` documentation.
- variables - optional
- parameters - optional

3.2.1 Minimal Version

Optional `min_ver` section should hold minimal required version of `moulin`. This is a text string that conforms PEP-440. For example `min_ver: "0.2"`. `moulin` will compare this with own version and will stop if required version is newer.

3.2.2 Mandatory sections: “desc” and “components”

There are only two mandatory sections: `desc` and `components`. `desc` should contain text string that describes the build. It is displayed when `moulin` invoked with `--help-config` command line option.

`components` should contain dictionary where key is a component name and value is another dictionary with component settings:

```yaml
components:
  component1:
    build-dir: "component-build-dir" # Optional
    default: true # Optional
    sources:
      .......
    builder:
      .......
  component2:
    sources:
      .......
    builder:
      .......
```

There are two main parts of each `component` description: `sources` and `builder`.

`sources` is optional and can contain list of source code definition, which will be fetched prior starting a build:
sources:
- type: git
  url: "git://git.yoctoproject.org/poky"
  rev: gatesgarth
- type: repo
  url: https://github.com/xen-troops/android_manifest.git
  rev: android-11-master
  manifest: doma.xml

All supported fetchers are listed in Fetchers section.

builder contains build configuration for a given component. There are multiple builder types supported. They are described in Builders section.

Apart from two mandatory options, component description can contain following optional keys:

- build_dir - build directory name. By default component’s name is used.
- default - if set to true - tells Ninja that this component is a default build target. This can be omitted and Ninja will choose build target by own rules.

### 3.2.3 Variables

variables section is optional. It can contain dictionary of variable’s name-value pairs:

```
variables:
  A: "a"
  B: "1{%{A}}{%{A}}" # will be expanded to "1aa"
  C: "2{%{B}}{%{B}}" # will be expanded to "21aa1aa"
```

Variables can be used anywhere in the YAML file. During internal pre-processing all variable references in form of \%{variable_name} will be replaced with actual variable value.

\% is a special symbol. It can be escaped by doubling it: \%\%.

Variables should be used to decrease amount of hard-coded values. Good candidates that should be moved to variables are path names, branches, hardware identifiers, etc.

### 3.2.4 Parameters

Often it is desired to have some options for a build. For example one can want to support a number of different HW boards, or to enable additional features. It would be not feasible to have separate YAML for every board-feature combination. This is where parameters come to help. All parameters should be stored in parameters section:

```
parameters:
  parameter1:
    desc: "parameter 1 description"
    option1:
      default: true
      overrides:
      ...
  option2:
    overrides:
    ...
```

(continues on next page)
option3:
  overrides:
  ...

parameter2:
  desc: "parameter 2 description"

option1:
  overrides:
  ...

option2:
  overrides:
  ...

option3:
  default: true
  overrides:
  ...

Every parameter should include mandatory desc key. Parameter can have one or more options, one of option should have default flag enabled.

Main part of each option is the overrides section. Contents of this section should correspond to top-level layout of YAML file. All contents of this section will be overlaid on contents of YAML file during pre-processing stage. Rules of this process are:

- Dictionaries are extended with new keys from overrides section.
- If dictionary already have the key:
  - If type of original value differs from type of overrides section value, error is generated.
  - If key’s value is a scalar (number, boolean, string) that it is replaced with value from overrides section.
  - If key’s value is an another dictionary, process start recursively.
  - If key’s value is a list, it is expanded with values from overrides section.
- Order of parameters application is not specified.

Basically, this rules follow the intuitive idea of extending/overwriting original config: primitive values will be overwritten, all other values will be extended.

User can chose parameter’s options using command line arguments, as described in Invoking moulin section.

### 3.3 Fetchers

Fetchers are the moulin plugins responsible for downloading of sources listed in sources section of a component.

moulin will generate phony Ninja target fetch-{component_name} for every component. It can be used to just fetch sources without building anything.
## 3.3.1 git fetcher

*git* fetcher is used to download code from a remote or local git repositories. There is a full list of supported parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td><em>git</em> # Selects <em><code>git</code></em> fetcher</td>
</tr>
<tr>
<td>url</td>
<td>&quot;url://for.repository/project.git&quot;</td>
</tr>
<tr>
<td>rev</td>
<td>revision_name</td>
</tr>
<tr>
<td>dir</td>
<td>&quot;directory/where/store/code&quot;</td>
</tr>
</tbody>
</table>

- **type** - mandatory - should be *git* to enable *git* fetcher.
- **url** - mandatory - repository URL. You can provide any URL that is supported by *git* itself.
- **rev** - optional - revision that should be checked out after cloning. Can be any *git* `tree-ish` like branch name, tag or commit ID. If this option is omitted, *git* will checkout default branch.
- **dir** - optional - directory name which should be used for cloning. If this option is missed, *moulin* will try to guess directory name from *url*. This path is relative to component's build directory.

## 3.3.2 repo fetcher

*repo* fetcher is used to download code using Google's *repo* tool. Full list of supported options:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td><em>repo</em> # Selects <em><code>repo</code></em> fetcher</td>
</tr>
<tr>
<td>url</td>
<td><a href="https://manifest.address/repo.git">https://manifest.address/repo.git</a></td>
</tr>
<tr>
<td>rev</td>
<td>manifest-revision</td>
</tr>
<tr>
<td>manifest</td>
<td>manifest-file.xml</td>
</tr>
<tr>
<td>depth</td>
<td>1</td>
</tr>
<tr>
<td>groups</td>
<td>all</td>
</tr>
<tr>
<td>dir</td>
<td>&quot;.&quot;</td>
</tr>
</tbody>
</table>

- **type** - mandatory - should be *repo* to enable *repo* fetcher.
- **url** - mandatory - manifest repository URL. You can provide any URL that is supported by *repo* itself. This corresponds to *repo*’s `-u` option.
- **rev** - optional - manifest revision. Corresponds to *repo*’s `-b` option.
- **manifest** - optional - manifest file name. Corresponds to *repo*’s `-m` option.
- **depth** - optional - cloning depth of internal repositories. Corresponds to *repo*’s `--depth` option. Setting it to 1 will sufficiently decrease fetching time.
- **groups** - optional - name of manifest groups that should be synced. Corresponds to *repo*’s `-g` option. You can use it to chose which project groups needs to be synced.
- **dir** - optional - directory name which should be used for code storage. If it is missing, *moulin* will use "." to initialize *repo* repository right in component’s build directory, as this is a main *repo* use case.
3.3.3 unpack fetcher

`unpack` fetcher is used to unpack already available archives to a specified directory. Example use-case is when need to use 3rd-party code/resources that are not available in git repository. Full list of supported options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Selects <code>unpack</code> fetcher</td>
</tr>
<tr>
<td>archive_type</td>
<td>type or archive. Now <code>tar</code> and <code>zip</code> are supported.</td>
</tr>
<tr>
<td>file</td>
<td>name of the archive file</td>
</tr>
<tr>
<td>dir</td>
<td>directory name which should be used for code storage. If it is missing, <code>moulin</code> will use &quot;.&quot; to unpack archive right into the component directory.</td>
</tr>
</tbody>
</table>

Right now `unpack` fetcher supports two archive types: `tar` and `zip`.

- `tar` actually supports not only plain `.tar` archives, but also compressed archives like `.tar.gz`, `.tar.bz2` and so on. We rely on `tar` ability to automatically select right de-compressor.
- `zip` - this is classic `zip` format. `unzip` tool is used to decompress this kind of archives, so it should be present on user’s machine.

3.3.4 west fetcher

`west` fetcher is used to download code using Zephyr’s `west` meta-tool. Full list of supported options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Selects <code>west</code> fetcher</td>
</tr>
<tr>
<td>url</td>
<td>manifest repository URL. You can provide any URL that is supported by <code>west</code> itself. This corresponds to <code>west init</code>’s -m option.</td>
</tr>
<tr>
<td>rev</td>
<td>manifest revision. Corresponds to <code>west init</code>’s --mr option.</td>
</tr>
<tr>
<td>file</td>
<td>manifest file name. Corresponds to <code>west init</code>’s --mf option.</td>
</tr>
</tbody>
</table>

For additional details see documentation on `west init`: https://docs.zephyrproject.org/latest/develop/west/built-in.html#west-init

Regarding installation of `west`, please see: https://docs.zephyrproject.org/latest/develop/west/install.html
3.4 Builders

Builders are the moulin plugins responsible for actual image building.

moulin will generate phony Ninja target \{component_name\} for every component. It can be used to build certain component. Please note that this will not build only given component. Any prerequisites will be fetched and build as well.

Builder configuration heavily depends on builder type and is described in next subsections.

3.4.1 yocto builder

Yoctobuilder is used to build OpenEmbedded-based images. It expects that poky repository is cloned in \{build_dir\}/poky and uses it’s poky/oe-init-build-env script to initialize build environment. Then bitbake-layers tool is used to add additional layers and bitbake used to perform the build.

```
builder:
  type: yocto          # Should be 'yocto'
  work_dir: "build"    # Optional
  build_target: core-image-minimal # Mandatory
  conf:
    # Mandatory
    - [MACHINE, "machine-name"]
    - [DISTRO_FEATURES_remove, "feature_to_remove"]
    - [DISTRO_FEATURES_append, "feature_to_add"]
  layers:
    # Optional
    - "./poky/meta-yocto-bsp"
    - "./meta-other-layer/"
  external_src:
    # Optional
    "package-name": "path-to-package-sources"
    "another-package-name": ["path part1", "path part2", "path part3"]
  target_images:
    # Mandatory
    - "tmp/deploy/images/machine-name/Image"
  additional_deps:
    # Optional
    - "path/to/file/generated/by/other/component"
```

Mandatory options:

- **type** - Builder type. Should be yocto for this type of builder.
- **build_target** - *bitbake*’s build target. This will be used to run the build: $ bitbake {build_target}
- **target_images** - list of image files that should be generated by this component as a result of invoking $ bitbake {build_target}. Every component should generate at least one image file.

Optional parameters. Those provide advanced features that may be needed if you are building multiple VMs with cross-dependencies.

- **conf** - list of additional local.conf options. Please note that each entry in conf list is not a key:value pair, but another list of two items. We use this format because it is possible to have multiple local.conf entries with the same key. Those entries will not be written straight into local.conf. Instead new file moulin.conf will be created. This file then will be included from local.conf.
- **layers** - list of additional layers. Those layers will be added to the build using bitbake-layers add-layer {layers} command.
- **work_dir** - *bitbake*’s work directory. Default value is “build”. This is where files like “conf/local.conf” are stored. You can overwrite so you can produce multiple builds from the same (or different) set of Yocto layers.
- **additional_deps** - list of additional dependencies. This is basically target_images produced by other components. You can use those to implement build dependencies between components. For example, if your system needs to have DomU’s kernel image on Dom0 file system, you might want to add path to DomU’s kernel into additional_deps of Dom0’s config. This will ensure that Dom0 will be built after DomU.

- **external_src** - list of external sources for packages. This option will make moulin to generate EXTERNALSRC:pn-{package} in local.conf. This feature is used to provide Yocto build with artifacts that were built outside of the tree. Such artifacts can be provided by another component, for example.

### 3.4.2 bazel builder

Bazel builder is used to build projects based on the Bazel build system provided by Google. It expects that a project with source and bazel configuration files is present in the build directory.

```plaintext
builder:
  type: "bazel"       # Mandatory and must be 'bazel'
  tool: "tools/bazel" # Optional
  startup-options:    # Optional
    - "--max_idle_secs=1"
  command: run       # Optional
  args:              # Optional
    - "--verbose_failures"
    - "--sandbox_debug"
  target:            # Mandatory
  target-patterns:   # Optional
    - "--dist_dir=path_to_dist"
  target_images:     # Mandatory
    - "out/deploy/virtual-device/virtual_device_aarch64/Image"
    - "out/deploy/virtual-device/virtual_device_aarch64/initramfs.img"
```

Mandatory parameters:

- **type** - Builder type. It must be bazel for this type of builder.
- **target** - target name that should be described in the corresponding BUILD.bazel file and must satisfy bazel rules.
- **target_images** - list of artifact files that should be generated by this component as a result of the build. Every component should generate at least one image file.

Optional parameters:

- **tool** - the relative path to the Bazel tool in relation to the ‘build-dir’. If this parameter is not defined, the system-installed Bazel tool will be used.
- **startup-options** - bazel startup options that appear before the command and are parsed by the client.
- **command** - bazel command. By default, build is used.
- **args** - bazel arguments related to the concrete command.
- **target-patterns** - bazel target patterns to be built or parameters to executable target.
3.4.3 android builder

Android builder is used to build Android Open Source Project (AOSP). It expects that AOSP is present in build directory. In most cases AOSP is cloned using repo fetcher.

```
builder:
  type: android # Should be 'android'
  env: # Optional
    - "TARGET_BOARD_PLATFORM=r8a7795"
  lunch_target: xenvm-userdebug
  target_images:
    - "out/xenvm/userdebug/boot.img"
    - "out/xenvm/userdebug/system.img"
  additional_deps: # Optional
    - "path/to/file/generated/by/other/component"
```

Mandatory options:

- **type** - Builder type. Should be android for this type of builder.
- **lunch_target** - lunch's build target. This will be used to run the build: `$ lunch {lunch-target}`
- **target_images** - list of image files that should be generated by this component as a result of invoking `$ m`. Every component should generate at least one image file.

Optional parameters:

- **env** - list of additional environment variables that should be exported before calling lunch.

3.4.4 android_kernel builder

Android Kernel builder is used to build kernel and kernel modules for Android Open Source Project (AOSP). It expects that correct directory layout is present in build directory. In most cases AOSP is cloned using repo fetcher.

```
builder:
  type: android_kernel # Should be 'android_kernel'
  env: # Optional
    - "TARGET_BOARD_PLATFORM=r8a7795"
    - "BUILD_CONFIG=common/build.config.xenvm"
    - "SKIP_MRPROPER=1"
  target_images:
    - "out/android12-5.4/common/arch/arm64/boot/Image"
```

Mandatory options:

- **type** - Builder type. Should be android_kernel for this type of builder.
- **target_images** - list of image files that should be generated by this component as a result of invoking build.sh script. Every component should generate at least one image file.

Optional parameters:

- **env** - list of additional environment variables that should be exported before calling build.sh.
- **additional_deps** - list of additional dependencies. This is basically target_images produced by other components. You can use those to implement build dependencies between components. For example, if your Android build needs Linux kernel built by some other component, you might want to add path to linux kernel image provided by this component into additional_deps. This will ensure that Linux kernel will be built before Android.

3.4. Builders
3.4.5 archive builder

Archive builder is intended to create an archive from other components. It can be used to gather build artifacts, for example. This builder uses `tar` to create archive files. Archives can be optionally compressed as, `tar` is invoked with --auto-compress option.

```
builder:
  type: archive          # Should be 'archive'
  name: "artifacts.tar.bz2"
  base_dir: "yocto/build/tmp/deploy/images/"
  items:
    # items are relative to base_dir
    - "generic-armv8-xt/Image"
    - "generic-armv8-xt/uInitramfs"
```

Mandatory options:

- **type** - Builder type. It should be archive for this type of builder.
- **name** - Name of an archive file. Add a suffix like `tar.bz2` to make `tar` compress archive with desired compressing algorithm.
- **base_dir** - Optional parameter specifying `tar`'s base directory. The default value is “.” if not specified. This is passed to `tar` as -C option. As result, the final archive will contain paths relative to base_dir. Avoid using .. because specified items will be archived by `tar` but all .. will be stripped. As a result, the archive will contain items with unexpected paths.
- **items** - list of files or directories that should be added to the archive. Please ensure that those files or directories are present in other components target_images sections, so Ninja can build correct dependencies. All paths are relative to the base_dir.

3.4.6 zephyr builder

This builder is used to build applications based on Zephyr OS. It uses Zephyr OS meta-tool west. Required code is expected to be fetched by west fetcher.

```
builder:
  type: zephyr
  board: xenvm
  shields:           # Optional
    - "shield1"
    - "shield2"
  target: samples/synchronization
  work_dir: build_dir
  target_images:
    - "zephyr/build/zephyr/zephyr.bin"
  vars:
    - "VAR1=var1_value"
  env:
    - "MY_ENV_VAR=my_value"
  additional_deps:   # Optional
    - "path/to/file/generated/by/other/component"
```

Mandatory options:

- **type** - builder type. Should be zephyr for this type of builder.
• **board** - target board name. For example: `xenvm` or `xenvm_gicv3` for Xen-based builds. Corresponds to `west build`’s `-b` option. See Zephyr’s documentation for the list of allowed values.

• **target** - build target. This will be used to run the build: `$ west build {target}`. For example: `samples/synchronization` or `samples/hello_world`.

• **target_images** - list of image files that should be generated by this builder. For standard build, it is expected to be “zephyr/build/zephyr/zephyr.bin”

Optional parameters:

• **env** - list of additional environment variables that should be exported before calling `west build`.

• **work_dir** - build system’s work directory. Default value is “build”. This is where files produced by build system are stored.

• **additional_deps** - list of additional dependencies. This is basically `target_images` produced by other components. You can use those to implement build dependencies between components. For example, if your system needs to have DomU’s kernel image in your zephyr image, you might want to add path to DomU’s kernel into `additional_deps` of zephyr’s config. This will ensure that zephyr will be built after DomU.

• **shields** - list of shields should be integrated to zephyr board.

• **vars** - list of additional variables that should be passed to cmake via `west build`.

Please note that this builder uses `--pristine=auto` command-line option.

Proper versions of CMake and Zephyr SDK have to be installed on the host.

### 3.4.7 custom_script builder

Custom-script builder is intended to perform custom actions. This builder actually calls script pointed in option `script`. Builder node is stored in the file in `work_dir` directory and that file passed to the script as parameter

```yaml
builder:
  type: custom_script  # Should be ‘custom_script’
  work_dir: "script_workdir"
  script: "path/to/script/custom_script.py"
  config:
    items:
      - "rootfs": "images/spider/rootfs.tar.bz2"
    manifest:
      "id": ""
      "vendorVersion": "0.2.0"
      "fileName": "archive.squashfs"
      "description": "DomD image"
      "bundleType": "full"
    target_images:
      - "custom_script_targets"
    additional_deps:  # Optional
      - "path/to/file_name"
```

Mandatory options:

• **type** - Builder type. It should be `custom_script` for this type of builder.

• **work_dir** - build script work directory. Default value is “script_workdir”. This is where files produced by build system are stored.
• **script** - path to script which performs custom actions. Whole yaml node will be stored to file and name of that file will be passed to **script** as last command line argument

• **target_images** - list of files that should be generated by script.

Optional parameters:

• **args** - additional arguments should be passed to **script**

• **additional_deps** - list of additional dependencies. This is basically **target_images** produced by other components. You can use those to implement build dependencies between components. For example, if your system needs to have DomU’s kernel image for your fota archive, you might want to add path to DomU’s kernel into **additional_deps** of Dom0’s config. This will ensure that fota archive will be built **after** DomU.

• Remaining parameters should be parsed and used by script pointed in **script** option.

For additional details please see [https://docs.zephyrproject.org/latest/develop/west/build-flash-debug.html#building-west-build](https://docs.zephyrproject.org/latest/develop/west/build-flash-debug.html#building-west-build)
4.1 About

*rouge* is a companion tool for *moulin*. Its purpose is to simplify creation of bootable images. It can create partition table, fill partitions with predefined files or raw data. It supports GPT, ext4fs, raw images, Android sparse images. Further formats can be added if needed.

Right now it can be used only as a separate tool, but there are plans to integrate it into *moulin* output.

4.2 Design Principles

*rouge* shares ideas (and code) with *moulin*. Thus, it is very similar in configuring and invoking to *moulin*. It can be used as a stand-alone tool: just provide only images: section in your YAML file. Or you can include this section into the same file, which is used by *moulin* to share common options or variables. In latter case *moulin* will generate additional image-{image_name} rules, so you can build images with Ninja.

4.3 Requirements

To do its job *rouge* invokes number of external utilities. Most of them are available on every system, except Android tools that usually should be installed separately. There is a list:

- *dd* - used to copy raw images
- *mkfs.ext4* - creates ext4 FS
- *mkfs.vfat* - creates vfat FS
- *simg2img* - used to unpack Android sparse image files
- *mcopy* - populates vfat images with files
4.4 Invoking **rouge**

**rouge** uses the same design ideas as **moulin**, and part of the command line options are shared with **moulin**. This includes --help-config, -v and --dump arguments. Please refer to **moulin** documentation for more details. This document describes only argument specific to **rouge**.

```bash
       [-l | -i image_name] build.yaml image_name
```

- `--help-config` - name of one of the images described in images: section of your build configuration file (build.yaml). Basically this is the image you want to create.

- `-l`, `--list-images` - list available images and their descriptions. Please note that actual list of images can depend on build config parameter values. For example, your build config may provide option to enable Android build. If this option is enabled, you may have separate image for Android.

- `--force` - force overwrite existing file. If this option is not give, **rouge** will refuse to create image if output file is already exists.

- `--special` - allow to write to a special file, like a block device. Without this option, **rouge** will refuse to write to, say, /dev/sda. Use this option with care and always double-check device name, as **rouge** will overwrite anything that is stored on that device.

- `-o` - provides output file name. This is optional parameter, by default **rouge** will write to <image_name>.img.

Apart from this options, **rouge** will read and parse all YAML-file related parameters in the same way as **moulin** does. You can check available parameters with --help-config.

4.5 Principles of Operation

**rouge** works in a very simple way. It uses **moulin**’s YAML processor that applies parameters and substitutes variables, then reads images: section, finds requested image specification. For a given image it checks if all mentioned files are present, then calculates sizes of partitions (if any) and total image size. Then it writes data to a given file/block device according to the specifications.

**rouge** tries to use sparse files whenever possible. Sparse file is a file with “holes” in it. It allows you to have a huge file that represents whole disk image with tiny bit of actual information in it. This speeds up image creation process and decreases used disk space. If you are writing resulting image file to your SD card manually, try adding conv=sparse option to your dd command line. This will speed up the writing process. If you want to distribute resulting images, take a look at Intel’s bmap tool. It allows you to share sparse files across devices.

4.6 YAML Sections

4.6.1 Shared sections

**rouge** uses the same YAML processing code as **moulin** so refer to **moulin**’s documentation for the desc, min_ver, variables, parameters sections description. This page describes only parts specific to **rouge**.
4.6.2 Image specifications

Images are specified in the following way:

```yaml
images:
  image_name_a:
    desc: "Description for the first image"
    image_size: 512 MiB
    type: gpt
    ... block description ...
  image_name_b:
    desc: "Description for the second image"
    type: raw_image
    ... block description ...
  image_name_c:
    desc: "Description for the third image"
    type: empty
    ... block description ...
```

images: section contains one or more keys, which serve as image names. Every image can have description, which will be displayed when rouge lists available images. type: key is mandatory as it defines type of block. Supported block types as described in the following sections.

Also you may specify the required size of image using image_size:. Please see section ‘Size Designation’ below for supported notation. If actual size of all partitions will be less than image_size: then image will be blown up to image_size:. If actual size is bigger than specified - error will be printed with explanation like “Actual size (20000) of image is bigger than requested one (10000).”

4.7 Block descriptions

“Block” is a basic rouge entity that describes one partition or partition table. Some block types can be nested. Supported block types are described below.

4.7.1 Size Designation

All block have size parameter. For some block types this parameter is mandatory, for some - optional. Basic unit for size is byte. For example

```yaml
type: empty
size: 4096
```

defines empty block with size of 4096 bytes. rouge supports some SI suffixes:

- **KB** - kilobyte - 1000 bytes
- **MB** - megabyte - 1000 kilobytes or 1 000 000 bytes
- **GB** - gigabyte - 1000 megabytes or 1 000 000 000 bytes
- **KiB** - kibibyte - 1024 bytes
- **MiB** - mebibyte - 1024 kibibytes or 1 048 576 bytes
- **GiB** - gibibyte - 1024 mebibytes or 1 073 741 824 bytes
Suffix must be separated from number by space. For example: size: 4 MiB defines size of 4 mebibytes or 4 194 304 bytes.

### 4.7.2 Empty block

Empty block is a block that does not contain any file or raw image. *rouge* will write nothing into this block if `filled`: zeroes option is not specified.

<table>
<thead>
<tr>
<th>type</th>
<th>empty</th>
<th>defines empty block</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>4096</td>
<td></td>
</tr>
<tr>
<td>filled</td>
<td>zeroes</td>
<td></td>
</tr>
</tbody>
</table>

`size` is mandatory, as *rouge* can’t infer it.

`filled` is optional, with only zeroes value allowed for now. This option may be used if you need the block to be filled with zeroes. For example, this is used for some Android partitions, like ‘rpmbemul’. Use this option only if you really need to. Otherwise you will needlessly increase size and upload time of an image.

### 4.7.3 Raw Image Block

Purpose of this block type is to include any binary data from other file. For example, if your build system creates `.ext4` image with root file system, you can use this block to place that image into GPT partition (which is described below).

<table>
<thead>
<tr>
<th>type</th>
<th>raw_image</th>
<th>defines raw image block</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>100 MiB</td>
<td></td>
</tr>
<tr>
<td>image_path</td>
<td>some/path/rootfs.ext4</td>
<td></td>
</tr>
</tbody>
</table>

`image_path` is mandatory. This is a file to be included into resulting image.

`size` is optional. If it is omitted, *rouge* will use size of file. If provided `size` is smaller than file `size`, *rouge* will stop with an error. Thus, you can create block that is bigger than file, but not smaller.

### 4.7.4 Android Sparse Image Block

It is similar to *Raw Image Block*, but it handles files in Android Sparse image format.

<table>
<thead>
<tr>
<th>type</th>
<th>android_sparse</th>
<th>defines android sparse block</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>3000 MiB</td>
<td></td>
</tr>
<tr>
<td>image_path</td>
<td>android/out/target/product/xenvm/userdata.img</td>
<td></td>
</tr>
</tbody>
</table>

`image_path` is mandatory. This is a file to be included into resulting image. *rouge* will call `simg2img2` tool to unpack it before writing it to a resulting image.

`size` is optional. If it is omitted, *rouge* will use data size, read from the file. If provided `size` is smaller than read `size`, *rouge* will stop with an error. Thus, you can create block that is bigger than unpacked file, but not smaller.
4.7.5 Filesystem Image With Files

This block type allows you to create new filesystem with some files included from your disk. This is ideal for creating boot partitions, where you store kernel, initial ramdisk and so on.

```plaintext
type: ext4 # defines ext4 partition block
size: 30 MiB
items:
  "remote_file1": "path/to/local/file1"
  "remote_file2": "path/to/local/file2"
  "remote_file3": "path/to/local/file3"
  "remote_file4": "path/to/local/file4"
  "remote_dir": "path/to/local/directory/"
```

*type* is required. Defines the filesystem type, currently *ext4* and *vfat* are supported.

*items*: section is optional. It defines remote:local mapping of files that should be presented on newly created filesystem. *remote* part is how the file will be named on new filesystem, while *local* is a path on your disk. You can specify parent folders for *remote* and these folders will be created on the destination filesystem. You may specify not only files but directories also. If the local directory contains subdirectories, they will be created under the *remote* directory. Older versions of *rouge* used *files*: as the name of the section. This name is still possible to use, but it is deprecated. Also only *items*: can contain directories.

*size* is optional. *rouge* will calculate total file size and add some space for the filesystem metadata to determine block size. You can increase size, if wish.

4.7.6 GUID Partition Table (GPT) block

This block type defines GPT along with all partitions. In most cases this is will be your top-level block definition. It can (and should) include other blocks, including other GPT. Inner GPT can come in handy in cases when you are creating image that holds data for multiple virtual machines and wish to provide VM with own GPT.

```plaintext
type: gpt # defines GPT block
partitions:
  boot: # partition label
    gpt_type: 21686148-6449-6E6F-744E-656564454649 # BIOS boot partition (kinda...)
    gpt_guid: 8DA63339-0007-60C0-C436-083AC8230900 # Partition GUID
    type: empty
    size: 30 MiB
  rootfs:
    gpt_type: B921B045-1DF0-41C3-AF44-4C6F280D3FAE # Linux aarch64 root
    type: raw_image
    image_path: "rootfs.ext4"
```

This example defines GPT with two partitions: *boot* and *rootfs*. *boot* is empty block and *rootfs* includes Raw Image block.

*partitions*: section is mandatory. It defines list of partitions, where key is a partition label.

Each partition contains definition of other block type plus optional keys:

*gpt_type*: (which we strongly suggest to provide) key holds GPT Partition Type GUID. List of widely used types can be found on Wikipedia, for example.

*gpt_guid*: key sets the GPT Partition GUID. By default this GUID is generated automatically to ensure that every partition in the world would have unique identifier. But there are some cases when external software depends on exact
value of a partition GUID. In such cases it is possible to hard-code this value. We strongly recommend not to use this key except for the cases when this is necessary because, according to the page 121 of Specification the software that makes copied of GPT-formatted disks and partitions must generate new Unique Partition GUID in each GPT Partition Entry.

sector_size is a custom sector size, 512 by default, but some devices (e.g. fancy flash storage) might have a sector of different size. This key allows tuning for such cases.

rouge will place partitions one after another, aligning partition start to 1 MiB (as per standard recommendation) and partition size to sector size, which defaults to 512 bytes.

### 4.8 Examples

The following example provides multiple different images:

```
min_ver: 0.3
desc: "rouge sample images"

images:
  empty_image:
    desc: "Just empty 32MB file"
    type: empty
    size: 32 MiB

unpacked_userdata:
  desc: "Unpacked android userspace image"
  type: android_sparse
  image_path: "android/out/target/product/xenvm/userdata.img"

simple_bootable_sd:
  type: gpt
  desc: "Full SD-card/eMMC image"
  partitions:
    boot:
      gpt_type: 21686148-6449-6E6F-744E-65656454649 # BIOS boot partition (kinda...)
      type: ext4
      size: 30 MiB
      items:
        "Image": "yocto/build/tmp/deploy/images/generic-armv8-xt/Image"
        "initrd": "yocto/build/tmp/deploy/images/generic-armv8-xt/uInitrd"
    domd_rootfs:
      gpt_type: B921B045-1DF0-41C3-AF44-4C6F280D3FAE # Linux aarch64 root
      gpt_guid: 8DA63339-0007-60C0-C436-083AC8230900 # Partition GUID
      type: raw_image
      image_path: "yocto/build-domd/tmp/deploy/images/machine/core-image-weston.ext4"
```

- `rouge sample_images.yaml -i empty_image` will generate just and empty file. This is the simplest example.
- `rouge sample_images.yaml -i unpacked_userdata` will use `simg2img` to unpack android userdata image.
- `rouge sample_images.yaml -i unpacked_userdata` will generate sort or usable image with two GPT partitions: one with data for bootloader, and other will contain ext4 root image created by Yocto.
5.1 Component-specific targets

As mentioned in chapters “Fetchers” and “Builders” of “User reference manual”, moulin generates some helpful targets for components: to fetch sources, to create a configuration, to build the component. For example, if we have component domu, we will have following self-explanatory targets:

- fetch-domu
- conf-domu
- domu

5.2 Rouge-specific targets

If section images: is present in YAML file, moulin will also generate handy image-{image_name} rules. They can be used to invoke rouge with the same build options, as moulin was invoked.

Moulin also generates {image_name}.img.gz and {image_name}.img.bmap targets.

5.3 Build inside yocto

If your component has yocto builder and you need to work inside the component’s build environment then you need to specify corresponding build folder (see work_dir parameter in builder section of required component) as a parameter to oe-init-build-env. For example to build only kernel-module-gles in domd:

```bash
$ cd yocto
$ . poky/oe-init-build-env build-domd
$ bitbake kernel-module-gles
```