SBXG Documentation

SBXG Team

Jun 29, 2019

SBXG Reference Manual

SBXG is a **build system generator** specialized in building from sources low-level components that are the foundation of Linux-based embedded devices, such as the U-Boot bootloader, the Linux kernel and the Xen hypervisor.

It is designed to offer a **high level of reproductibility and tracability**. Given that the URLs pointing to the different components are always available, SBXG should always generate the same outputs for a given set of inputs. No surprise to be expected.

On top of being able to build just the low-level components, SBXG can generate standalone images, ready to be flashed on an SDcard or used as virtual machine disks. In this mode, SBXG expects the *rootfs* to be available, it does not generate one.

All components (but the cross-compilation toolchain) are built from source, with a configuration file enforced by version. This allows SBXG users to rely on the sources and their own (or pre-packaged) configurations, instead of a black box downloaded from untrusted sources.

SBXG provides default configurations for some boards, toolchains, kernels and u-boot, to demonstrate its capabilities, but one of its goal is to be able to use opaque (private) user configurations that can leave outside of SBXG (e.g. reside in a dedicated source control repository).

Status of the project

SBXG is in **omega** stage (beyond alpha)! The documentation is being redacted, and some key features are being developed.

CHAPTER 1

The Philosophy of SBXG

To summarize the key-points of *the introduction*, SBXG is designed to offer a high level of reproductibility and tracability when it comes to building the low-level components (Kernel, Bootloader) of a system. Configuration should be trivial, easy to maintain and keep track of.

It only concentrates in **building from source** the low-level components, eventually putting them together in a final image. In the later case, it expects the rootfs to be provided: *it does not care about the roots*. Even for kernel modules. How you distribuate these components is up to **you**.

1.1 Positions towards other tools

SBXG is not a innovation. There (hopefully) exist many other tools able to build a kernel. However, we think that SBXG fills a certain void when trying to generate Linux-based embedded systems, for which other tools do not offer a (subjectively judged) acceptable answer.

1.1.1 Why not Buildroot?

Buildroot allows to build root file systems, a Linux kernel and the U-Boot bootloader. If you are interesed in creating a self-contained, minimalistic and finely tailored to your needs, buildroot is the way to go. Don't bother with us! However, if your objective is to build *only the low-level components*, or a whole bunch of them, you should consider using SBXG. Furthermore compiling these components with an explicit configuration (i.e. without fragments) is not handy with buildroot: if you don't modify buildroot itself, you are obliged to add some kind of layer on top of it to generate a meaningful buildroot configuration.

1.1.2 Why not a simple shell script?

If your goal is just to build a couple of low-level components, a hand-crafted shell script is surely enough, given you are familiar with their build systems.

However, if you need to build a lot of them, SBXG may come handy, as it knows how to do. Also, if you want to create a full sdcard image, things may become a bit more challenging. This is even more true when trying to put Xen in the loop. SBXG knows how to do it. It should make things easier for you.

CHAPTER 2

How to install SBXG

SBXG can be divided in two classes of components:

1. the core of SBXG itself, which is a python package; and
2. the run-time dependencies of SBXG, that are third-parties tools the generated build system will rely on.

2.1 Installing the dependencies

Only GNU/Linux distributions are supported. You may want to use the command-line that suits your distribution the best.

2.1.1 Debian-based distributions (including Ubuntu)

```
sudo apt install git python3-pip make curl build-essential autoconf \
  autotools-dev tar swig python-dev libconfuse-dev mtools
```

2.2 Installing the SBXG python package

There are no pip package

Currently, there is no pip package on Pypi. You are obliged to build from sources!

Currently, the only way to install SBXG is by downloading the sources and installing from these sources. So, you must have `git` and `pip3` installed. Then, run the following commands:

```
git clone https://github.com/sbxg/sbxg.git
pip3 install --user -r sbxg/requirements.txt
pip3 install --user sbxg
```

2.3 Understand the use of each dependency

You may have noticed that SBXG requires quite some dependencies. We will explain here in which context they are important.

Python 3.6 SBXG is written in python. This is a no-brainer, without python 3.6 or higher, you will not be able to run SBXG at all.

Make SBXG bootstraps its build system, by generating a Makefile. Therefore, `make` (only GNU make is tested) is a strong requirement for SBXG. No `make`, no possibiltiy to use SBXG's generated files.

mkfs (ext3, vfat) To generate the final image, mkfs (ext3 and vfat) will be required.

curl and tar To download compressed tarballs from the internet, and extract them. This is typically used to retrieve the toolchain and components to be built.

swig It seems that U-Boot requires swig to be built.

autotools SBXG will build from sources a package that uses the autotools. As such, the autotools programs needs to be installed (e.g. autoconf, automake, . . .). This package is genimage.

kernel build essentials SBXG will compile the Linux Kernel and U-Boot. Hence, such a development environment shall be installed.

CHAPTER 3

How to use SBXG

SBXG is distributed as a python package, with a pre-defined entry point. It means you can use SBXG via its python API or as a command-line tool. N

3.1 Command-Line Interface

Stability of the CLI

Currently, SBXG's CLI is not stable. It means it can change at any time, without prior notice.

SBXG's command-line works with different **commands**:

- `show`: to display information on SBXG's files;
- `gen` (or `generate`): to generate a build system able to download and build various un-connected components that share the same toolchain.

Note that SBXG accepts the following arguments **before the commands**. They will be applied to any command that follows:

- `--color`: can be set to `yes`, `no` or `auto`, to respectively enable, disable or auto-detect if SBXG's output should contain colors.
- `-I` (or `--lib-dir`): specify a directory to populate SBXG's components library. If no arguments are provided, SBXG's built-in library will be used. To understand what the library is, please refer to *Understanding the SBXG Library*.

3.1.1 sbxg show

SBXG can display the contents of its library, which has been populated with the `-I` option. Take a look at *Understanding the SBXG Library* if you are unsure of what the library is.

This command will display in a human-readable way on the standard output:

- what are the toolchains that are available;
- what are the linux, u-boot and xen sources available; and
- what are the linux, u-boot and xen configurations available.

Note that this command accepts the `--mi` option (for Machine Interface) that displays the same information but serialized in JSON. This may come handy if used for scripting. Refer to *Machine Interface* for details.

3.1.2 sbxg gen

The `gen` command can be invoked with the following parameters:

- `-L` (or `--linux-source`): the **name** of a file describing how to retrieve the sources of a given version of the Linux kernel.
- `-U` (or `--uboot-source`): the **name** of a file describing how to retrieve the sources of a given version of the U-Boot bootloader.
- `-X` (or `--xen-source`): the **name** of a file describing how to retrieve the sources of a given version of the Xen hypervisor.
- `-l` (or `--linux-config`) the **name** of a Kconfig file describing the various configuration parameters of a given Linux profile.
- `-u` (or `--uboot-config`) the **name** of a Kconfig file describing the various configuration parameters of a given U-Boot profile.
- `-x` (or `--xen-config`) the **name** of a Kconfig file describing the various configuration parameters of a given Xen profile.

The following argument pairs must be used together:

- `-L` and `-l`;
- `-U` and `-u`;
- `-X` and `-x`.

Note that you can use as many arguments as you want. You can build a dozens of Linux kernels in one go for example, as long as they are to be built with the **same toolchain**.

A cross-compilation toolchain can be specified with the `-t` or `--toolchain` option. Note that if this option is not present, SBXG will assume you are building **natively**.

Architecture for cross-compilation

If you want to perform cross-compilation with the default toolchains, your host (the system that builds) MUST be x86. We don't provide toolchains that are not x86 binaries. You can however define your own toolchain.

It takes a **mandatory positional argument** that is the path to the directory in which SBXG will generate its build system.

3.2 Python API

Stability of the Python API

Currently, SBXG's python API is not stable. It means it can change at any time, without prior notice.

CHAPTER 4

Understanding the SBXG Library

SBXG heavily relies on its own **library concept**. It is a finite set of directories that conform to a well-specified file hierarchy. This file hierarchy is explained in the present document.

The following directory hierarchy exposes all the different directories recognized as being part of an *SBXG library*. Here, we assume the library is composed of a single directory, named `sbxg/lib/`:

```
sbxg/lib/
├── toolchains/
├── configs/
├── sources/
├── bootscripts/
├── images/
└── boards/
```

4.1 The `toolchains/` subdirectory

The `toolchains/` directory contains YAML files, each one describing a *toolchain*:

```
sbxg/lib/
└── toolchains
    └── *.yml
```

A toolchain file may contain the following paramters:

Name	Description
url	URL where to download the toolchain from
path	Extracted (`tar -xf`) directory
prefix	Cross-compilation prefix
arch	Linux and U-Boot architecture code name
xen_arch	Xen architecture code name
host	Typically, the prefix without underscore

When **cross-compiling**, all these parameters should be mandatory. Note however that **native compilation** currently relies on the trick that some of these parameters may be not set or set to an empty string.

4.2 The `configs/` subdirectory

The `configs/` directory contains **three** subdirectories. Each of them contains Kconfig files that are used by Linux, U-Boot and Xen to configure their build:

```
sbxg/lib/
└── configs
    ├── linux
    │   └── *
    ├── uboot
    │   └── *
    └── xen
        └── *
```

These files may or may not have an extension. By convention, their name should self-describe their purpose. For example, a Linux 4.14 configuration file that allows to compile minimal Xen guests (domus) may be named: `linux-4.14-xen-domu-minimal`.

4.3 The `sources/` subdirectory

The `sources/` directory contains **four** subdirectories. Each of them contains YAML files that describe how the various components (Linux, U-Boot, Xen and genimage) may be retrieved:

```
sbxg/lib/
└── sources
    ├── genimage
    │   └── *.yml
    ├── linux
    │   └── *.yml
    ├── uboot
    │   └── *.yml
    └── xen
        └── *.yml
```

A source file **must** contain the following paramters:

Name	Description
url	URL where to download the component from
path	Extracted (`tar -xf`) directory

4.4 The `bootscripts/` subdirectory

The `bootscripts/` directory contains Jinja2 template files that must program the U-Boot bootloader at boot-time. These files are associated with a templating context that is described in *Templating Context*. To learn more about bootscripts, please refer to U-Boot's website.

```
sbxg/lib/
└── bootscripts
    └── *
```

These files may or may not have an extension. By convention, their name should self-describe their purpose, and the extension is always `.j2` (but this is not mandatory). For example a boot script allowing U-Boot to boot a sunxi board may be named: `boot-sunxi-default.j2`.

4.5 The `images/` subdirectory

The `images/` directory contains Jinja2 template files that describe how a disk image must be generated by genimage. These files are associated with a templating context that is described in *Templating Context*.

```
sbxg/lib/
└── images
    └── *
```

These files may or may not have an extension. By convention, their name should self-describe their purpose, and the extension is always `.j2` (but this is not mandatory). For example a genimage configuration describing a simple SDcard partitioning may be named: `sdcard-simple.j2`.

4.6 The `boards/` subdirectory

The `boards/` directory contains YAML files, each one describing a *board*:

```
sbxg/lib
└── boards
    └── *.yml
```

A board describes what low-level components should be compiled, and what binaries from these components should be used to generate a final disk image. This file may contain the following top-level entries:

Name	Description
toolchain	Name of the toolchain to be used.
genimage	Name of the genimage source to be used
linux	Name of the Linux kernel source to be used
linux_config	Name of the Linux Kconfig to be used
linux_image	Filename of the Linux executable (e.g. `zImage`)
linux_dtb	Filename of the Linux DTB to be used
uboot	Name of the U-Boot source to be used
uboot_config	Name of the U-Boot Kconfig to be used
uboot_image	Filename of the U-Boot executable
boot_script	Name of the bootscript to be templated
disk_image	Name of the genimage configuration to be templated
root	In the Kernel bootargs, path to the rootfs block device
rootfs	URL to the rootfs (`.ext3`) to be used in the image
linux_bootargs	Additional Linux bootargs to be specified

CHAPTER 5

Templating Context

As explained in *Understanding the SBXG Library*, some files are to be templated by SBXG. Namely: the *boot scripts* and *images configurations*. The underlying templating engine is Jinja2. Templating engines uses what is called a *templating context*: it is the dataset used to output a meaningful result.

Since users are expected to create your own library to tweak SBXG to fit their needs, having a clear documentation of this templating context is mandatory. It can be seen as a stable interface between SBXG and its users.

Currently, no promise is made of a stable interface

As we are still in early development, this may change.

Notion of canonical names

Later in this document, we will mention **canonical names**. These are litteral strings derived from the names of files that are parts of the SBXG library. Canonical names are expected to be used by programming languages, such as GNU make. Which implies that it shall be get rid of unexpected characters (dots, dashes, spaces, …). In the **canonical** form, all the unwanted characters are replaced with *underscores*.

5.1 Top-level entries

The templating context will always contain the following top-level entries. Note that they may be set to `None` if not available, but it is guaranteed that these keys will *exist*.

Name	Type	Description
top_build_dir	`string`	Where components will be built
toolchain	`Toolchain`	Description of the toolchain
downloads	`list<Download>`	List of items to be downloaded
linuxes	`list<Item>`	List of the Linux kernels to be built
uboots	`list<Item>`	List of the bootloaders to be built
xens	`list<Item>`	List of the Xen hypervisors to be built
genimage	`Genimage`	Description of the genimage tool
board	`Board`	Description of the board

The types mentioned in this table will be detailed in the next sections.

5.2 Toolchain data structure

The `Toolchain` type is composed of following entries that were written in the YAML file that describes the toolchain. See *The toolchains/ subdirectory* for details.

5.3 Download data structure

It should be no surprise that the build system generated by SBXG will attempt to *download* the sources of the components to be built. What falls in the scope of a `Download` object is compressed tar archives containing sources. This includes Linux, U-Boot, Xen and genimage. A download receives a **name**, that uniquely identifies it amongs others. It also allows components to be built to *depend* on a download, by referring to its **name**.

Name	Description
name	Canonical name of the download
url	URL where to fetch the component
archive	Filename of the component to be downloaded

5.4 Item data structure

An `Item` describes either a Linux kernel, a U-Boot bootloader or a Xen hypervisor. All entries in an `Item` are of type `string`. It is composed of the entries written in the YAML file that describes these components. See *The sources/ subdirectory* for details. In addition to these fields, the following entries are guaranteed to exist:

Name	Description
config	Full path to the associated Kconfig file
name	Canonical name of the component
download	Name of the associated download information

5.5 Genimage data structure

The `Genimage` type is composed of the entries written in the YAML file that describes genimage. See *The sources/ subdirectory* for details.

5.6 Board data structure

The Board type is a subset of the elements described in *The boards/ subdirectory*, which is defined by the following entries:

- linux_dtb;
- linux_image;
- uboot_image;
- boot_script;
- disk_image;
- root;
- bootargs.

It comes with the following extraneous string entries:

Name	Description
rootfs_url	The URL where the rootfs resides
rootfs_path	Filename of the rootfs after download

CHAPTER 6

Machine Interface

Some commands or python API that SBXG provides expose a **machine interface**, which means formatted data that a script can easily takes as input.

6.1 Contents of the library

As explained in *How to use SBXG*, the `sbxg show` command accepts the `--mi` argument, that returns a JSON-formatted string containing the contents of the library. This paragraph details the format of these data.

The top-level dictionary contains a set of keys that are all lists of two kinds of objects, that we name here `Item` and `TypedItem`. A list of objects of type `T` will be noted `list<T>`.

Top-Level Keys	Type	Description
sources	`list<TypedItem>`	List of the sources
toolchains	`list<Item>`	List of the toolchains
configurations	`list<TypedItem>`	List of the Kconfig files
boards	`list<Item>`	List of the available boards
bootscripts	`list<Item>`	List of the boot scripts
images	`list<Item>`	List of the images descriptions

We now describe the `Item` and `TypedItem` objects:

`Item` Keys	Type	Description
name	`string`	Name of the item (e.g. value to be used)
path	`string`	Absolute path to the associated file

`TypedItem` Keys	Type	Description
name	`string`	Name of the item (e.g. value to be used)
path	`string`	Absolute path to the associated file
type	`string`	Type of the item (e.g. linux, xen, u-boot)

CHAPTER 7

How about the rootfs?

As explained in *the introduction*, SBXG does not care about the **Root Filesystem** (rootfs). The rootfs is completely out of the scope of SBXG, because it is a completely different class of problem that requires specialized tools, and a awful lot lot of community work. Luckily, we already have tremendous work made openly available:

- Buildroot, to generate finely-tailored rootfs, mostly for embedded systems;
- Debootstrap, to retrieve pre-compiled Debian rootfs;
- DFT, a tool to heavily customize Debian rootfs.
- Gentoo stages, to generate or retrieve distribution-levels rootfs.
- And many, many more available choices…

CHAPTER 8

Tutorial: using the built-in library

SBXG provides a built-in library, that can be extended by the users. This allows SBXG to provide users an out-of-the-box experience.

First, to observe the contents of the built-in library, just run:

```
sbxg show
```

and something like this will appear:

```
List of toolchains:
  - local
  - armv7-eabihf

List of sources:
  - linux: linux-4.14.35
  - linux: linux-4.12.0
  - uboot: uboot-2017.07
  - xen: xen-4.8.3
  - genimage: genimage-v11

List of configurations:
  - linux: linux-4.12-sunxi
  - linux: linux-4.14-sunxi-xen-dom0
  - linux: linux-4.14-xen-domu
  - uboot: uboot-2017.07-minimal
  - xen: xen-4.8-sunxi

List of bootscripts:
 - boot-sunxi-default.j2
 - boot-sunxi-xen.j2

List of images:
 - sdcard-simple.j2
 - guest-simple.j2
```

(continues on next page)

(continued from previous page)

```

List of boards:
 - cubietruck-standalone
```

Now, imagine that you want to build a Linux kernel for a *sunxi <https://linux-sunxi.org/Main_Page>* board from your x86 PC. You can see that SBXG provides a configuration for a 4.12 Linux kernel. Granted, this is quite old, but let's say you have an old version of SBXG ;)

So you are interested in the following elements:

- the **toolchain** (`armv7-eabihf`);
- the **linux source** (`linux-4.12.0`); and
- the **linux configuration** (`linux-4.12-sunxi`).

Just tell that to SBXG:

```
sbxg gen -L linux-4.12.0 -l linux-4.12-sunxi -t armv7-eabihf build
```

SBXG will work a bit, and if everything went right, it should end with exit code 0. In the `build/` directory, you now have a generated standalone `Makefile`. You can build everything by running `make` in this directory. You can even pass to `make` the number of jobs to be used for building. It will be used to build the different components:

```
make -C build -j 3
```

Upon successful completion of this command, you will see the following directories:

```
build/
├── armv7-eabihf--glibc--stable-2018.02-2/
├── build_linux_4_12_sunxi/
├── downloads/
├── linux-4.12/
├── Makefile
└── stamps/
```

Let's go through them one by one:

- `armv7-eabihf--glibc--stable-2018.02-2/`: this is where the toolchain was extracted.
- `linux-4.12/`: this is where the sources of the kernel were extracted.
- `downloads/`: you will find here compressed archives that were downloaded.
- `stamps/`: this contains files generated by the Makefile that allows make to only download the archives when needed. Each file contains the URL from which the component was downloaded.
- `build_linux_4_12_sunxi/`: this is where the linux kernel was **built**. Note that SBXG performs builds out-of-tree when possible.

Now that your kernel has been built, and now that you known **where** it was built, you can freely dispose of them. For instance, the `zImage` resides in `build/build_linux_4_12_sunxi/arch/arm/boot/`.

CHAPTER 9

Developping in SBXG

9.1 Setting-up the development environment

SBXG is a python module, that requires python3.6 or higher. To make things simple, we advise you develop in a virtualenv.

You shall begin by installing python3.6 or higher on your platform. Recent distributions should have it already installed. From now on, I assume that the program `python3` is in your `PATH`, and that running:

```
python3 --version
```

yields something like:

```
$ python3 --version
Python 3.6.8
```

I will also assume that the associated program `pip3` (pip for python3) is installed and in your `PATH`. Which means that running:

```
pip3 --version
```

yields something like:

```
$ pip3 --version
pip 9.0.1 from /usr/lib/python3/dist-packages (python 3.6)
```

We now can install the virtualenv package, if it does not already exist:

```
pip3 install --user virtualenv
```

Now, for every new clone of SBXG sources, you **must** create the virtualenv in **the top source directory**:

```
virtualenv --python=python3 .venv
```

This will have for effect to create the directory `.venv/` (which is already described in the `.gitignore` file) and will contain your python environment when developping in SBXG.

Then, **activate your virtualenv**. If you have a POSIX-compatible shell, run:

```
. .venv/bin/activate
```

This will modify your current environment, so you can use the virtualenv. From now on, the `python` and `pip` programs will be the ones of your virtualenv! Not the ones of your system.

Don't forget to activate your virtualenv!

For **every new interactive session** (i.e. when you open a new terminal to develop in SBXG), you **MUST** activate your virtualenv. Otherwise, you will not use it, and weird things may occur!

If you just created your virtualenv, you must install the appropriate python dependencies:

```
pip install -r requirements.txt
```

And you are good to go!

CHAPTER 10

Coding Practises

10.1 Coding Style

Hate it or love it, but SBXG shall conform to PEP8. Docstrings shall conform to PEP257. There is currently no strict enforcement of these rules.

10.2 Linting

We use pylint to lint SBXG. It is not run automatically, because there's nothing worse than seeing that your build fails because of an extraneous whitespace, or a rightful TODO you added. Use the linter to help you code, don't bow to it.

To lint, just run the following from the top source directory:

```
pylint
```

10.3 Tests

We use pytest to run SBXG's tests. It is a bit unusual to test, because it has a lot of interactions with the filesystem, which makes it not obvious to unit test it.

Also, since the goal of SBXG is to generate a build system, one of the possiblities to check that the build system was correctly generated is to run it, and see what happens. That's the simple and obvious method. Problem is that we build sevaral Linux kernels… which take a lot of time to be downloaded and built. So, testing the correctness of the generated file is not that trivial after all. Currently, we run the generated Makefile for a single case that should cover most of the usecases, but it is not systematically executed.

To run the tests, just run the following from the top source directory:

```
pytest
```

10.4 Documentation

Documentation is obviously very important. It's not always clear what to write, and how it should be written, but we try to figure out as we grow. Documentation, which is often looked down upon by developers tries to be a first-class citizen in SBXG. It demands work and dedication, but you'll get used to it. We use Sphinx to manage the documentation.

To build the documentation, run the following from the top source directory:

```
make -C doc html
```

10.5 Contributions

This is currently a mess.

TODO.