SONiC Buildimage Repository

Complete Guide

Table of Content

Introduction

This repository contains code for SONiC to build an (ONIE) compatible network operating system (NOS) installer image for network switches, and also how to build docker images running inside the NOS. SONiC buildimage is a GNU make based environment for build process automation. It consists of two main parts:

Backend

collection of makefiles and other helpers that define generic target groups, used by recipes. **Makefile**, **slave.mk** and **sonic-slave/Dockerfile** are the backend of buildimage.

- *slave.mk* is the actual makefile. It defines a set of rules for *target groups* (more on that later). You can find a make rule for every target that is defined in recipe there. *Makefile* is a wrapper over sonic-slave docker image.
- Every part of build is executed in a docker container called sonic-slave, specifically crafted for this environment. If build is started for the first time on a particular host, a new sonic-slave image will be built form *sonic-slave/Dockerfile* on the machine. It might take some time, so be patient. After that all subsequent make commands will be executed inside this container. *Makefile* takes every target that is passed to make command and delegates it as an entry point to a container, making process of running container transparent.

Frontend

collection of recipes, that define metadata for each build target

- **rules/** has a collection of recipes for platform independent targets. Every recipe is a file that describes a metadata of a specific target, that is needed for its build.
- You might find **rules/config** very useful, as it is a configuration file for a build system, which enables/disables some tweaks.
- **dockers/** directory is a place where you can find Dockerfiles for generic docker images.
- **src/** is a place where a source code for generic packages goes. It has both submodules (simple case, just run dpkg-buildpackage to build), and directories with more complicated components, that provide their own Makefiles. **platform/** contains all vendor-specific recipes, submodules etc.
- Every **platform/[VENDOR]/** directory is a derived part of buildimage frontend, that defines rules and targets for a concrete vendor.

Along with building image details, it contains source code for reference modules.

In order to build sonic locally, we have offical guide We have internally our own build-system guides as well as below:

1. Building a SONiC-vs Image.docx
2. Building a SONiC-PINS Image.docx
3. Custom SONiC-VS Image Build Guide.docx

Repository Structure

The SONiC Build image repository comprises of a total of eight folders:

- **device**
- **dockers**
- **files**
- **installer**
- **Platform**
- **Sonic-slave**
- **src**

sonic-buildimage/device

This folder contains all the sonic devices name and config detailed of each device makes.

The **device** folder specifically designed to map device level configuration setup at each asic.

It contains devices details of accton, alphanetworks, arista and many more.

sonic- buildimage/docker

This folder contains all existing Docker files for generic docker images along with it contains docker of each subsystem. It have all external third party docker information as well.

sonic-buildimage/files

This folder contains sonic build system sub dependency relevant scripts and tools used for parsing and database configs. It also contains image-config for each of supported features which have its internal daemon process and handler service that needs to be mapped and linked with the sub-system. All linux built-in packages are also available in this folder

sonic-buildimage/installer

This folder contains installer procedures for SONiC , designed to run in different environments (SONiC, ONIE, or BUILD). Here's a breakdown of its key functions:

1. **License and Copyright**: It begins with copyright notices and an SPDX license identifier, indicating the script's licensing terms.
2. **Trap Management**: The script includes a function to append commands to a signal trap, allowing it to clean up properly upon receiving signals like EXIT or INT.
3. **Configuration Reading**: The read_conf_file function reads configuration settings from files, processing variable assignments and removing comments.
4. **Environment Detection**: It checks the environment to determine if it's running in SONiC, ONIE, or BUILD by examining the directory structure and specific files.
5. **Machine Configuration**: It attempts to read the machine configuration from several possible locations to understand the system it's running on.
6. **Installation Logic**: Depending on the installation environment:
    a. For **ONIE**, it prepares to create partitions and mount file systems.
    b. For **SONiC**, it checks the currently running version to prevent overwriting an existing installation unless forced.
    c. For **BUILD**, it prepares to create a raw disk image.
7. **Directory Management**: It ensures the target installation directory exists and is clean before proceeding.
8. **Payload Handling**: It unzips the installation payload (which contains the necessary files for SONiC) to the target directory, handling special cases for file systems and Docker-related files.
9. **Bootloader Update**: The script updates the bootloader menu with the installed image, ensuring that the system can boot into the new installation.
10. **Mode Setting**: If available, it sets the system to NOS mode, allowing the installed operating system to be used.

sonic-buildimage/platform

The folder defines various configuration settings for the build process, including compiler options, flags, and variables specific to the platform or architecture being targeted (in this case, Barefoot).

1. **Dependencies Management**: It often specifies dependencies between different modules or components, ensuring that when one part of the software is modified, the

necessary parts are rebuilt. This is crucial for large projects to maintain consistency and avoid errors.

2. **Target Definitions**: It contains various targets for the make command, which can include building libraries, executables, or performing clean-up tasks. Each target specifies what files need to be compiled or linked and what commands should be executed.
3. **Conditional Logic**: It can contain conditional logic to customize the build process based on the environment or specific parameters. For example, different configurations might be applied based on whether the build is for development or production.

sonic-buildimage/sonic-slave-debian release

There are several SONiC slave branches, such as sonic-slave-bookworm and sonic-slave-bullseye, are specifically designed to adapt the SONiC platform to different versions of the Debian operating system.

1. **Purpose**: These branches are often used to manage platform-specific configurations or to maintain compatibility with different versions of the underlying operating system or software stack.

- **Naming Convention**: The name typically includes "sonic-slave" followed by the name of a specific Debian release (like bookworm, bullseye, buster, etc.). This indicates that the branch is designed to work with that particular version of Debian.

Each branch corresponds to a particular Debian release, which allows developers to leverage the unique features and package availability of that release. By maintaining separate branches, the Debian release can ensure compatibility and stability across various environments, accommodating different hardware setups and operational requirements.

sonic-buildimage/src

The src directory contains the core source files and submodules necessary for building the SONiC. This directory includes various components that are essential for networking functionality, such as DHCP, IP management, and system utilities.

The structure allows for modular updates, where each submodule can be independently maintained and updated, ensuring that the overall system remains stable while benefiting from the latest enhancements and fixes. Recent commits indicate ongoing development efforts to keep various submodules, like sonic-gnmi and sonic-p4rt, up-to-date with the latest codebase, reflecting a continuous integration approach.

Directory	Subdirectory	Description

sonic-buildimage/src/bash/	**files/**	Contains various files used by the Bash scripts, possibly including utility scripts or configuration files essential for the proper operation of the Bash environment within SONiC.
	patches/	Holds patches that address specific issues within the Bash environment, such as fixing bugs or adding new features. For example, there may be patches for authorization issues or performance improvements.
	.gitgnore	A file that specifies intentionally untracked files to ignore in the Git repository. This helps maintain a clean repository by excluding unnecessary files that don't need to be versioned.
	makefile	A file used to manage the build process for the Bash-related components, specifying how to compile and link files, as well as any dependencies needed for successful builds.
sonic-buildimage/src/sonic-buildhooks/	**debian/**	Contains Debian packaging files and configurations.
	hooks/	Contains custom hooks used in the build process.
	scripts/	Scripts related to the build process.
sonic-buildimage/src/ sonic-config-engine/	**Data/** **tests/**	Contains data files for L1, L2, L3 configuration generation. Contains tests related to configuration functionalities.
	config_samples.py	Contains sample configurations for various scenarios.
	minigraph.py	Handles the generation of the minigraph configuration.

	openconfig_acl.py	Implements OpenConfig ACL functionalities.
	smartswitch_config.py	Configuration generator for smart switch topologies.
	sonic-cfggen	Main configuration generator for SONiC.
	sonic_yang_cfg_generator.py	Handles YANG configuration generation functionalities.
sonic-buildimage/src/ sonic-containercfgd	**containercfgd/**	Contains configuration and logic for managing container settings.
	tests/	Contains tests for validating the functionalities of the containercfgd.
	pytest.ini	Configuration file for pytest to define test parameters and settings.
	setup.cfg	Setup configuration for packaging and distribution.
	setup.py	Setup script for package installation.
sonic-buildimage/src/ sonic-yang-models/	**yang-models/** **doc/**	Directory containing the YANG models for various SONiC components. Documentation for YANG models used in SONiC.

	yang-templates/	Templates for generating YANG models.
	tests/	Tests for validating YANG model functionalities.
	setup.cfg	Setup configuration for building and packaging YANG models.
	setup.py	Setup script for package installation.

Other than buildimage repo directory also holds submodules details used in sonic sub system. Each submodule is linked to its own repo with specific commit.

e.g sonic-py-swsssdk, sonic-sairedis, sonic-swss, sonic-utilities.

Submodule Update in sonic-buildimage

To update specific submodules such as sonic-swss and sonic-utilities within the SONiC Buildimage repository, you can follow these steps. These submodules are typically part of the SONiC ecosystem and updating them ensures that you have the latest code from their respective repositories.

Note: In general, this submodule update is not required as conde in different branches once merged automatically become part of build repo. BUt still to know the required steps, consider following:

1. **Edit the .gitmodules File to Add Specific Repositories**

The .gitmodules file contains configuration information about submodules, including their paths and repository URLs. If you want to replace the current submodule repositories (e.g., sonic-swss and sonic-utilities) with your own, you'll need to update the .gitmodules file.

```
cd sonic-buildimage
nano .gitmodules
```

You need to modify the url fields for both submodules to point to your custom repositories. For example:

```
[submodule "sonic-swss"]
   path = sonic-swss
   url = https://github.com/your-username/sonic-swss.git

[submodule "sonic-utilities"]
   path = sonic-utilities
   url = https://github.com/your-username/sonic-utilities.git
```

After editing the .gitmodules file, save the changes:

- If you're using nano, press Ctrl + X, then Y, and then Enter to save

2. **Update Submodules**

Now that you've updated the .gitmodules file to point to your specific repositories, you need to sync and update the submodules to reflect these changes.

a. Initialize and Update the Submodules

Run the following commands to initialize and update the submodules with the new repository URLs:

```
git submodule sync              # Syncs the submodule configuration
git submodule update --init --recursive
```

3. **Commit the Changes to .gitmodules and Submodule References**

You now need to commit the changes you made to the .gitmodules file as well as the updated submodule references.

a. **Stage the Changes**

First, stage the .gitmodules file and the updated submodule references:

```
git add .gitmodules sonic-swss sonic-utilities
```

b. **Commit the Changes**

Commit the changes with a descriptive message:

```
git commit -m "Updated .gitmodules to use custom repositories for sonic-swss and sonic-utilities"
```

c. **Push the Changes (If You're Working on a Branch)**

If you are working on a specific branch and need to push the changes to the remote repository, use:

```
git push origin <your-branch-name>
```

4. **Update the Submodules to Latest Commit (Optional)**

If you want to make sure that the submodules are updated to the latest commit from the newly added repositories, you can go inside each submodule and pull the latest changes.

a. **Update sonic-swss Submodule**

```
cd sonic-swss
git checkout master      # Ensure you're on the desired branch (typically 'master' or 'main')
git pull origin master   # Pull the latest changes
cd ..
```

b. **Update sonic-utilities Submodule**

```
cd sonic-utilities
git checkout master      # Ensure you're on the desired branch (typically 'master' or 'main')
git pull origin master     # Pull the latest changes
cd ..
```

5. **Verify the Updates**

To verify that everything is working properly, check the status of the submodules again:

```
git submodule status
```

This should now show the latest commit hashes from the custom repositories you've configured.

Contributing to Build image repo

Contributing to the **SONiC Buildimage** repository depends on feature use case and its scope. Below is a detailed guide for each of these contribution types based on experience till now, please consider this may vary for each feature.

Contributing to SONiC Buildimage: Overview

1. **Adding a new Docker image**: Modify Dockerfiles to create new Docker containers.
2. **Adding a new systemd process**: Modify systemd service definitions and add related configuration.
3. **Adding a new feature**: Add new feature components, which may include YANG models, configuration files, and more.

1. Adding a New Docker Image

To add a Docker container in the **SONiC Buildimage** repository, let's walk through the steps using **STP (Spanning Tree Protocol)** as an example. This example will guide you through adding a Docker container for an STP.

1. : **Create the Dockerfile for the STP**

Creating a Dockerfile.j2 template for the STP container. Below is a detailed walkthrough of how to integrate the Dockerfile.j2 and make the appropriate changes to the repository:

```
cd <path-to-sonic-buildimage>/dockers
```

2. **Create the docker-stp/Dockerfile.j2 file**:

If it doesn't already exist, create the directory and the Dockerfile template:

```
mkdir -p docker-stp
touch docker-stp/Dockerfile.j2
```

3. **Edit the Dockerfile.j2** to include the necessary packages and setup for the STP service. You can copy and paste the content from your provided example and make any adjustments if needed.

```
{% from "dockers/dockerfile-macros.j2" import install_debian_packages,
install_python_wheels, copy_files %}
FROM docker-config-engine-bookworm-
{{DOCKER_USERNAME}}:{{DOCKER_USERTAG}}

ARG docker_container_name
RUN [ -f /etc/rsyslog.conf ] && sed -ri
"s/%syslogtag%/$docker_container_name#%syslogtag%/;"
/etc/rsyslog.conf

## Make apt-get non-interactive
ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update     && \
   apt-get install -f -y  \
      libdbus-1-3      \
      libdaemon0       \
      libjansson4      \
      libpython2.7      \
      libjemalloc2      \
      ebtables

{% if docker_stp_debs.strip() -%}
# Copy locally-built Debian package dependencies
{{ copy_files("debs/", docker_stp_debs.split(' '), "/debs/") }}

# Install locally-built Debian packages and implicitly install their
dependencies
{{ install_debian_packages(docker_stp_debs.split(' ')) }}
{%- endif %}

RUN apt-get clean -y     && \
   apt-get autoclean -y  && \
   apt-get autoremove -y && \
   rm -rf /debs
```

```
COPY ["start.sh", "/usr/bin/"]
COPY ["supervisord.conf", "/etc/supervisor/conf.d/"]
COPY ["critical_processes", "/etc/supervisor"]

ENTRYPOINT ["/usr/local/bin/supervisord"]
```

We need to add rules for new container and other config as well.
As a sample case please see below PR

https://github.com/sonic-net/sonic-buildimage/pull/20417

2. Adding a New Docker Image

To add a new systemd service, you'll need to modify several files in files/image-config/ to include the service definition, handlers, and the necessary configuration files.

1. **Define the New Service in features/service**:

Create a new systemd service definition file in files/image-config/features/service.

```
[Unit]
Description=My Custom Service

[Service]
ExecStart=/usr/local/bin/my-service
Restart=always

[Install]
WantedBy=multi-user.target
```

2. **Create a Feature Handler in features/handlers**:
   If needed, create a handler script for the service in features/handlers.

Example handler (my-service-handler.py):

```
import os
from sonic_platform import platform

def setup_service():
   # Code to setup the new service, e.g., configure systemd
   os.system('systemctl enable my-service')
```

```
    os.system('systemctl start my-service')

def cleanup_service():
    # Code to stop and remove the service if needed
    os.system('systemctl stop my-service')
    os.system('systemctl disable my-service')
```

A sample systemd process code PR is below:

https://github.com/sonic-net/sonic-buildimage/pull/20355

3. Adding a New YANG model

If you're adding a completely new feature to SONiC, you may need to add new YANG models to src/sonic-yang-models and implement any necessary configuration or service components.

1. **Add YANG Models**:
   YANG models define the structure of configuration and operational data for the new feature. These models go in the src/sonic-yang-models directory.

   ```
   cd src/sonic-yang-models
   touch my-feature.yang
   ```

   Example YANG model (my-feature.yang):

   ```
   module my-feature {
     namespace "urn:my-feature";
     prefix mf;

     container config {
       leaf param1 {
         type string;
       }
     }
   }
   ```

2. **Add YANG test cases**:

   Yang test cases are added in tests/test-config

   Reference code PR for adding Yang as below:

https://github.com/sonic-net/sonic-buildimage/pull/20354