NL Build System

Henry Hoegelow

March 2, 2023
Version 0.1

Abstract

The build system of our C15 project has become quite rotten in the last years. There were a lot of attempts to integrate new concepts and ideas. Most of them left some traces in the project, some stuff is still used, others is not, documentation and a clear concept is missing. The new build system described here aims at solving all the issues found during the last years with a clear and well documented concept.

1 General

1.1 Conventions

There are some conventions that glue the different parts of the build system together:

- Target names are tied to the folder structure:

  ```
  /os/pi4/factory => make os-pi4-factory
  /update/c15 => make update-c15
  /update/c15/epc => make update-c15-epc
  ```

- Machine names have to be used consistently in /build-environments, /os and /updates folders:
  - epc
  - beaglebone
  - pi4
  - web
  - playcontroller
  - supervisor

2 Scopes

2.1 os

The targets in the /os folder produce images (.img or .iso) or root filesystem tars.

2.1.1 factory

Targets in the /os/{machine}/factory folders shall produce images that can be used to initially hook up a machine in the factory. For the epc, the will be an iso file that can be used to install the base os, for pi4 and beaglebone this may be an image to be written on a sd card.

2.1.2 update

Targets in /os/{machine}/update folders shall produce tar files containing complete root file systems for the given machine, without any product specific software. In contrast to our packages, which should not be installed at this place, all system libraries needed by those packages have to be installed here.

This is not the best solution and may be improved later.

2.2 packages

All our self-written software is divided into packages. The term "package" is used to underline that each of this sub-projects is required to pack itself into a debian package. The CMakeList files in the packages subfolders should not fiddle with podman or docker, nor should they care for installing libraries, tools or whatsoever. Packages, when building, expect to find everything they need in their environment.

Maybe we should modify packages CMakeLists, so they check for all tools needed and disable un-buildable targets if tools are missing?

2.3 build-environments

Build-environments, located in /build-environments/{machine}, have to bundle all tools and libraries needed to build packages for a given machine. They have to play together nicely with update os tars in a sense, that build-environments have to provide tools and development versions of libraries that are expected to exist in the update root filesystem by the runtime linker.
A build-environment has to provide a toolchain.cmake file setting up all cmake variables needed for a proper cross build.

2.4 updates

In the updates folder, all previously explained concepts coalesce in targets producing update files to be put on USB sticks, spread over-the-air or uploaded onto the target devices.
Here, root file systems produced by targets in /os/{machine}/update are extended with packages, cross-built in /build-environments/{machine}/ and bundled by update-specific shell scripts.

3 Podman

The build system heavily relies on podman. It is used to solve various problems, previously sorted out by handwritten solutions:

- Provide environment with tools and libraries
- Provide environment to emulate foreign architecture
- Store packages and tools needed to build our targets
- Cache stuff on the developer computer
- Provide independence from 3rd-party servers by running our own docker registry
- Support versioned collections of packages and tools
- Speed up build

3.1 Registry

The docker registry (currently hosted by quelltextlabor.de) allows read-only access for "normal" developers and "read/write" access for so-called "pod-maintainers".

3.1.1 Developer roles

developer "Normal" developers have only read access to the docker registry. This means, that the build system will try to fetch ready-built containers from the registry during build. You can switch to this role by issuing the follwing command in your build directory:

```
cmake -DDEVELOPER_ROLE=developer .
```

If the system cannot fetch a podman container in the required version, the build will fail. Most probably this is because you changed a Dockerfile or some variables or files, that are baked into the container. If you did this deliberatly, you should change your role to pod-maintainer.

pod-maintainer Pod-Maintainers have read/write access to the registry. They can change contents of Dockerfiles and files or variables that somehow influence the building of a container. As soon as a container could be successfully built on the pod-maintainers machine, it is uploaded to the registry so it is available for other developers. Switch to this role by issuing:

```
cmake -DDEVELOPER_ROLE=pod-maintainer .
```

On next build time, you will be asked for a password. If you don't know the right password, you are probably not a NL developer.

3.1.2 Versioning

All containers used here share the same name: "generic". They differ by the version postfix, which is generated by hashing the Dockerfile contents and all files or variables given when registering the container. This means, that you can unscrupulously switch between branches, relying on properly versioned pods.

Note, that this is much better than before, but still not perfect: If you do only a slight change to the Dockerfile, it will be rebuilt. If rebuilding the image incorporates fetching stuff from the internet (which it usually does) than the result of the build might differ significantly from the previous image. This can even break the build - so be careful with being a pod-maintainer

4 Configuration

4.1 Why?

The whole project is configurable by yaml files, describing global properties like number of voices, available parameters, parameter ranges and so on. This properties have to be used in multiple packages and it is very crucial that they are in sync. Also, package authors are free to decide which tools or languages they use, so this global properties have to be available for C/C++, java, javascript, typescript,...

4.2 How?

We use TypeScript for transpiling the yaml files into target language code files. The TypeScript environment is somewhat special and needs careful setup. Thatswhy I moved it into a pod and integrated it into the build system in a way, that the actual transpilation process is done only once for the whole project. The result is than injected into the

build-environments, so that the environments do not have to care for all the TypeScript foo.

5 CMake

The project uses CMake as build system generator. You can use it with make or ninja, both should work.

5.1 Includes and Functions

Repeating tasks are implemented in include files and functions, residing at /cmake.

5.1.1 registerPod

To make use of the automated pod handling, pods have to be registered using registerPod():

registerPod(
[*DEPENDS_ON_POD ${HASH_OF_OTHER_POD}*]
[*TWEAK_IMAGE skript.sh*]
[*CONFIGURED_DEPENDENCIES filename ...*]
[*CONFIGURED_DEPENDENCIES_COPYONLY filename ...*]
[*DEPENDENCIES target ...*]
[*BUILT_DEPENDENCIES file ...*]
[*USED_VARIABLES ${VAR} ...*])

for example:

```
registerPod(
CONFIGURED_DEPENDENCIES_COPYONLY sda.sfdisk
CONFIGURED_DEPENDENCIES_COPYONLY install-factory-os.sh
CONFIGURED_DEPENDENCIES generatePkgBuild.sh install.sh
USED_VARIABLES ${PACKAGES})
```

This will configure the files sda.sfdisk, install-factory-os.sh, generatePkgBuild.sh and install.sh, the latter two with replacing of cmake variables. It will calculate a hash of

- the configured Dockerfile.in in the current folder
- the configured files given as dependencies
- the used variables

and use the resulting hash as version number of the pod. The resulting pod will be referenced as generic:«SHA1».

5.1.2 configuration

The /cmake/configuration.cmake file should be included by every package that needs access to the generated files. When building natively, it will just include the generated files. On cross-build, it will add the injected generated files from the outer project to the include paths.

5.1.3 package

Each package has to include the /cmake/package.cmake file and end with a line calling the "deb" function:

deb("*dependency-target, ...*", "*description*")

for example in audio-engine/CMakeLists.txt:

```
deb("nltools" "Audio Engine for Nonlinear Labs synthesizers")
```

This will generate a debian package {PROJECT_NAME}.deb that can be used by the tools and scripts to finally build the update.

5.1.4 crossBuild

The function crossBuild() adds a new target producing a tar with one all the enumerated packages built for the given machine:

crossBuild(
NAME xyz
MACHINE abc
[*DEPENDS foo ...*])

for example in /updates/c15/epc:

```
crossBuild(
    NAME update-c15-epc-bins
    MACHINE epc
    DEPENDS nltools audio-engine[nltools])
```

This will pick the build environment for the epc, build the packages for nltools and audio-engine (while providing the nltools library in the statging dir for building the audio-engine) and bundle both resulting packages in a tar named update-c15-epc-bins.tar.

5.1.5 buildImage

To merge a update rootfs with the cross-built packages, the buildImage function can be used:

buildImage(
NAME xyz
BASE abc
ADD bar ...
[*DEPENDS foo ...*]
[*POST_PROCESS_POD pod*]
[*POST_PROCESS_SCRIPT script.sh*])

for example in /updates/c15/beaglebone:

```
buildImage(
    NAME update-c15-beaglebone
    DEPENDS ${POD_TARGET}    DEPENDS update-c15-beaglebone-bins
    DEPENDS os-beaglebone-update-rootfs
    DEPENDS ${CMAKE_BINARY_DIR}/.../os-beaglebone-update-rootfs.tar
    DEPENDS update-c15-beaglebone-bins.tar
    BASE os/.../os-beaglebone-update-rootfs.tar
    ADD updates/c15/beaglebone/update-c15-beaglebone-bins.tar
    POST_PROCESS_POD ${POD_URI}
    POST_PROCESS_SCRIPT bundle.sh)
```

This will depend on the targets update-c15-beaglebone-bins and os-beaglebone-update-rootfs and the files os-beaglebone-update-rootfs.tar and update-c15-beaglebone-bins.tar. If all dependencies are fulfilled, it will run the bundle.sh script in the pod represented by the Dockerfile in the current directory. The bundle.sh should unpack the given tars and install all the packages inside into the rootfs specified by 'base'. The resulting rootfs will be update-c15-beaglebone.tar.