AN12 - How to build software for Charge Control Yocto platforms

chargebyte GmbH

Sept 22, 2022

Contents

1 Revisions

Revision	Release Date	Changes
1	February 21, 2022	Initial release
2	September 22, 2022	Changed Company Logo

2 Introduction

2.1 Motivation

The Charge Control platforms are equipped with all the software required to operate the charging process and the external control of it, as described in the corresponding User Guides. Yet under certain circumstances, customers may wish to deploy or run their own software in addition to the operating system and charging stack components. Another case is when the customer has generated a BSP Image for one of the Charge Control platforms, which has no charging stack components, then they would want to deploy their own charging components related software. Note that we have an open-source solution for generating such a BSP image using Yocto, which can be found in this link: Yocto Environment for Tarragon and EVAcharge SE platforms.

Since the Charge Control platforms run on an embedded system using ARM processor cores, the target binaries need to be compiled specifically for this platform and its operating system. This application note suggests several methods to compile such binaries. It is mostly aimed at and most recently tested with the Yocto platforms as targets, but should also work, to the largest extent, for Debian platforms. It is tested for Ubuntu Linux 18.04 and 20.04 virtual machines (VM) as cross-compile environments, but should work with other Linux environments, native or in VMs.

The Charge Control platforms are based on Linux, and therefore, any software we would like to run on these platforms must be able to compile for Linux. Also note that some minor platform-related specifications may apply, such that other considerations may need to be made in case of compiling for an x86/x86_64 desktop platform.

2.2 Methods

There are three methods which customers can use to build their own software on Charge Control target platforms:

1. Building natively on the target
2. Building in an emulation of the target
3. Building in a cross-compile environment

Every method has its own requirements, advantages and disadvantages. These are discussed below.

2.3 Notes

- You will notice that in order to follow this user guide, a root file system with an extension of `.ext4` is required. If you have a firmware that has an extension of `.image`, mount this file (using Linux's squashfs filesystem support), or unpack it using `unsquashfs` from squashfs-tools, and find the contained `.ext4` image.
- A "developer image" maybe needed during the course of this user guide. To get such an image, you can follow the guide to generate your own BSP developer image in the link mentioned above, or you can contact us for other types of images.
- In the course of this user guide, you may find the words "EVAcharge SE" and "Tarragon" mentioned, where compilation steps differ based on which of them is used. These are the names of the underlying hardware for the Charge Control platforms. EVACharge SE is used for CC M, CC P and CC O, while Tarragon is the hardware name for CC C.

3 Method 1: Building natively on the target

3.1 Requirements

The image installed on the target has to be a developer image, so that it contains tools such as gcc, g++. make, cmake, autotools, development libraries and their headers and so forth.

3.2 Advantages

- Customers can simply copy their source code to the target device and launch their build process there.
- No other special build environment is required

3.3 Disadvantages

- Requires a developer build to be deployed on the target.
- Access to a target hardware is needed.
- Relatively low build speed and limited physical memory due to hardware constraints.

3.4 Notes

Customers should be able to compile their programs easily using this method. However, since it has multiple disadvantages, it will not be further discussed in this guide.

4 Method 2: Building in an emulation of the target

4.1 Requirements

This method uses the QEMU emulator on an x86/x86_64 host to do an emulated native build on a significantly more powerful machine. It also integrates more nicely into a standard development process, as a development environment, such as an IDE, or even just a more powerful editor, can be made use of.

This method assumes access to an x86/x86_64 host machine running Debian or Ubuntu Linux (or other derivatives of these distributions), which will serve as the emulation host for development. In addition to that, a recent developer image build for the target platform is required.

4.2 Advantages

- Does not require the presence of the target hardware.

4.3 Disadvantages

- Relatively complex to setup, depending on the application the customer wants to build.
- Requires a recent dump of a developer root file system of the target.

4.4 Steps

1. Prepare the host system to run QEMU by enabling binary format emulation and checking availability of QEMU.

```
sudo apt install binfmt-support qemu-user-static # For Debian/Ubuntu
sudo dnf install qemu-user-static # For RedHat/CentOS/Fedora
```

2. Create a mount point for the root file system container and mount the container to this mount point.

```
sudo mkdir -p /rootfs
sudo mount /path/to/<target_image>.ext4 /rootfs
```

Where `<target_image>` is name of the actual root file system image. The file system of the Charge Control should now be available under `/rootfs`

3. Copy the root file system to a development area in your standard work area and proceed to unmount the container from the mount point.

```
mkdir ~/Development/ChargeControl
sudo cp -a /rootfs ~/Development/ChargeControl/
sudo umount /rootfs
```

Note: If you are using recent host systems, e.g., Ubuntu 20.04, proceed to step 6.

4. On older systems, such as Ubuntu 18.04 and before, the root file system may need to be enhanced with the actual emulator. Note that the path segment `/usr/bin` needs to be identical in source and target.

```
sudo cp -p /usr/bin/qemu-arm-static
  ~/Development/ChargeControl/rootfs/usr/bin
```

5. For complete operation within the emulation, certain system functions may need to be provided as mounts.

```
sudo mount -o bind /proc ~/Development/ChargeControl/rootfs/proc
sudo mount -o bind /sys  ~/Development/ChargeControl/rootfs/sys
sudo mount -o bind /dev  ~/Development/ChargeControl/rootfs/dev
```

6. Launch the emulation as follows.

```
sudo chroot ~/Development/ChargeControl/rootfs
```

You should now be in a shell (bash) which behaves as if it is running on the target system, with its proper paths.

7. To build an application, start a new shell different from the one running the emulation, and copy your code into or create code in a location within the root file system related to the emulation. In our case, here is an example of where you can create the code: `~/Development/ChargeControl/rootfs/home/root/Development/test_prog.c`

8. Edit the code in place with your development tool of choice, from the non-emulated host environment.

9. Run your build process within the emulated environment, i.e., the emulation shell (bash), as if it was native, e.g., by calling `cmake` or `autotools`, in case the corresponding `Makefile.am` and configure scripts exist, or just using `make` or `gcc`.

```
# From the emulation shell
cd home/root/Development
make testprog
./testprog # To test that the build works
```

10. If you are using `configure` scripts to automatically create `Makefile`s, you can simply call the configure script with your prefered command line options within the emulation shell. A call to the script with options representing a particular filesystem layout can look like the following.

```
/configure --prefix=/usr \
           --exec-prefix=/usr \
           --bindir=/usr/bin \
           --sbindir=/usr/sbin \
           --libexecdir=/usr/lib \
           --sysconfdir=/etc \
           --datadir=/usr/share \
           --localstatedir=/var \
           --mandir=/usr/man \
           --infodir=/usr/info
```

11. To deploy your resulting binary or library, you need to copy the project's results out of the emulated environment, onto your target system.

5 Method 3: Building in a cross-compile environment

5.1 Requirements

The goal here is to build applications on a different architecture than the target, but for the target architecture. A typical example is building on x86_64 Linux for armv7 Linux. This is done in order to make use of the speed of a non-embedded system, and to be able to compile within the environment where development of code feels natural. This host environment can be Windows (mostly x86_64), Linux, macOS, or a Linux virtual machine (VM).

During the following steps, it is assumed that there is access to x86/x86_64 host machine running Linux Ubuntu, either natively or in a virtual machine. In addition to that, a recent image build, preferably a developer build, for the target platform is required.

5.2 Advantages

- Building is relatively fast.
- Potential use of the same environment for development and compiling.

5.3 Disadvantages

- Requires a recent dump of a root file system of the target.

5.4 Steps

1. Repeat steps 2 and 3 in Building in an emulation of the target.

```
sudo mkdir -p /rootfs
sudo mount /path/to/<target_image>.ext4 /rootfs
mkdir ~/Development/ChargeControl
sudo cp -a /rootfs ~/Development/ChargeControl/
sudo umount /rootfs
```

2. To setup the cross-compile environment, cross-compilers and related tools are required on the Linux build hosthost, i.e., a tool chain which runs natively, but creates files (objects, binaries, etc.) for the target platform. On Ubuntu, the following commands should install all necessary packages for both Tarragon & EVAcharge SE.

```
# For EVAcharge SE
sudo apt install build-essential libc6-armel-cross libc6-dev-armel-
   cross binutils-arm-linux-gnueabi gcc-arm-linux-gnueabi g++-arm-
   linux-gnueabi pkg-config-arm-linux-gnueabi

# For Tarragon
sudo apt install build-essential libc6-armhf-cross libc6-dev-armhf-
   cross binutils-arm-linux-gnueabihf gcc-arm-linux-gnueabihf g++-arm-
   linux-gnueabihf pkg-config-arm-linux-gnueabihf
```

The tools for both target platforms can be installed in parallel. Note that other cross-compilers instead of or in addition to `gcc`/`g++` can of course be installed, e.g., for Objective C or Pascal or whatever the GNU Compiler Collection supports.

3. To compile plain source files for the target, just use the appropriate compiler depending on the target platform in the command line.

```
arm-linux-gnueabi-gcc sourcefile.c -o targetbinary # For EVAcharge SE
   & C applications

# OR for Tarragon & C++ applications
arm-linux-gnueabihf-g++ -c file1.cpp
arm-linux-gnueabihf-g++ -c file2.cpp
arm-linux-gnueabihf-g++ -o targetbinary file1.o file2.o
```

You can apply further compiler and linker options as usual. If your source code has dependencies to libraries, e.g., that are found within the root file system of the target platform, you need to adjust the include and linker paths, in order to find those dependencies in the root file system. The simplest way is to point the tools to your root file system.

```
# For Tarragon & C++ applications
arm-linux-gnueabihf-g++ --
   sysroot=${HOME}/Development/ChargeControl/rootfs -c file1.cpp
arm-linux-gnueabihf-g++ --
   sysroot=${HOME}/Development/ChargeControl/rootfs -c file2.cpp
arm-linux-gnueabihf-g++ --
   sysroot=${HOME}/Development/ChargeControl/rootfs -o targetbinary
   file1.o file2.o
```

Note: Next steps are in case of using Makefiles, Automake, Autoconf and cmake.

4. As proper projects usually do not rely on "manual" direct compilation as outlined in the previous step, but rather use at least a Makefile, you can build your source files by simply running "`make`", if nothing needs to be configured and no special targets are selected. A Makefile, that is written properly, can make implicit and explicit use of supported variables such as CC, CXX, CPPFLAGS, CFLAGS, CXXFLAGS, LDLIBS, and so on. Using these variables, we can manipulate the build in order to adjust it to our cross-compile environment, by adding them to a standard `make` call. Again, a prerequisite is to have a Makefile in the directory where the `make` command is executed.

```
ROOTFS="${HOME}/Development/ChargeControl/rootfs"
# This command works for Tarragon. In case you are building for
  EVAcharge SE, replace gnueabihf with gnueabi in C or C++ cross-
  compilers
make \
   CC="arm-linux-gnueabihf-gcc --sysroot=${ROOTFS}" \
   CXX="arm-linux-gnueabihf-g++ --sysroot=${ROOTFS}" \
   LDFLAGS="-L${ROOTFS}/usr/lib"
```

Note that when `--sysroot` is given, CPPFLAGS, CFLAGS and so on usually do not need to be modified with paths pointing to the root file system (using e.g. -I and -L), unless non-standard paths are used in the target's root file system. LDFLAGS are modified in order to use the correct libc of the target, not from the compile environment.

5. In case of using Autotools, which consists of autoconf, automake and related scripts designed to create a project's Makefiles from easier templates, one needs to run the script `./configure` first before being able to run `make`. This script automatically adjusts a Makefile by calling it with some command line options. However, sometimes even the script `./configure` does not exist yet, but needs to be created from `configure.ac` (or formerly `configure.in`). To get a quick understanding of these tools, the GNU Automake documentation may help.

   For using the `configure` script to build projects for Tarragon and EVAcharge SE, the command line option `--host` should be set to initialize a cross-compile setup (in addition to your usual options). Additional environment variables are required to make cross-compiler aware of the root file system for possible dependencies.

```
HOST=arm-linux-gnueabihf # This command works for Tarragon. In case
  you are building for EVAcharge SE, replace gnueabihf with gnueabi
ROOTFS="${HOME}/Development/ChargeControl/rootfs"
export CFLAGS="--sysroot=${ROOTFS}"
export CXXFLAGS="--sysroot=${ROOTFS}"
export LDFLAGS="--sysroot=${ROOTFS} -L${ROOTFS}/usr/lib"
/configure --host="${HOST}"
```

If the project uses `pkg-config` to detect dependencies, it becomes more complex. These two additional environment variables (before the above ./configure call) help to find all dependencies.

```
export PKG_CONFIG_SYSROOT_DIR="${ROOTFS}"
export
  PKG_CONFIG_LIBDIR="${ROOTFS}/usr/lib/pkgconfig:${ROOTFS}/usr/lib/${
  HOST}/pkgconfig:${ROOTFS}/usr/share/pkgconfig"
```

If the project uses `libtool` to control the build process, setting two additional environment variables (before the above `./configure` call) helps to overcome the fact that `libtool` filters the "`--sysroot=`" option automatically.

```
export CC="$HOST-gcc $CFLAGS"
export CXX="$HOST-g++ $CXXFLAGS"
```

The subsequent step of running `make` should require no special tasks, as all the created Makefiles are already equipped with the paths to the cross-compile environment.

6. If you are using CMake as an alternative to Autotools for creating Makefiles automatically, which many modern projects support instead of (or in addition to) Autotools, the sequence to build your project would be a little different. CMake understands a set of command line options, notably for specifying the compilers and related tools, and to control the created Makefiles.

   You need to call `cmake` with the following options (in addition to your usual options):

   ```
   ROOTFS="${HOME}/Development/ChargeControl/rootfs"
   # This command works for Tarragon. In case you are building for
     EVAcharge SE, replace gnueabihf with gnueabi in C or C++ cross-
     compilers
   cmake \
          -DCMAKE_SYSROOT="${ROOTFS}" \
          -DCMAKE_SYSTEM_NAME="Linux" \
          -DCMAKE_SYSTEM_PROCESSOR="arm" \
          -DCMAKE_FIND_ROOT_PATH_MODE_PROGRAM=NEVER \
          -DCMAKE_FIND_ROOT_PATH_MODE_LIBRARY=ONLY \
          -DCMAKE_FIND_ROOT_PATH_MODE_INCLUDE=ONLY \
          -DCMAKE_FIND_ROOT_PATH_MODE_PACKAGE=ONLY \
          -DCMAKE_C_COMPILER=arm-linux-gnueabihf-gcc \
          -DCMAKE_CXX_COMPILER=arm-linux-gnueabihf-g++ \
          -DCMAKE_C_FLAGS="-L${ROOTFS}/usr/lib" \
          -DCMAKE_CXX_FLAGS="-L${ROOTFS}/usr/lib"
   ```

   The subsequent step of running `make` should require no special tasks, as all the created Makefiles are already equipped with the paths to the cross-compile environment.

7. To deploy your resulting binary or library, do as you would on a purely native system, i.e., by installing to a proper directory tree, and then packaging the relevant files, or simply copying the resulting binary out from the developments host machine, and deploying it on the actual target device.