APPLICATION NOTE

pylon-Based Applications on Embedded Targets

Document Number: AW001609
Version: 02 Language: 000 (English)
Release Date: 01 February 2021

BASLER
the power of sight

Contacting Basler Support Worldwide

Europe, Middle East, Africa

Basler AG
An der Strusbek 60–62
22926 Ahrensburg
Germany

Tel. +49 4102 463 515
Fax +49 4102 463 599

support.europe@baslerweb.com

The Americas

Basler, Inc.
855 Springdale Drive, Suite 203
Exton, PA 19341
USA

Tel. +1 610 280 0171
Fax +1 610 280 7608

support.usa@baslerweb.com

Asia-Pacific

Basler Asia Pte. Ltd.
35 Marsiling Industrial Estate Road 3
#05–06
Singapore 739257

Tel. +65 6367 1355
Fax +65 6367 1255

support.asia@baslerweb.com

www.baslerweb.com

Table of Contents

1 Introduction

The pylon Camera Software Suite (“pylon” for short) is Basler’s software package that contains an easy-to-use SDK, drivers, and tools that enable you to integrate Basler cameras into your own applications. pylon provides the following components:

- All drivers required for the different camera interfaces (GigE Vision, USB3 Vision, MIPI CSI-2, etc.).
- pylon Viewer: A powerful GUI tool for configuring the camera and grabbing images or videos. It also offers valuable support during the development process.
- Easy-to-use APIs for different programming languages (C, C++, C#, Python, etc.) that enable you to easily integrate camera functionalities into your own application. pylon exposes a unified API for all Basler cameras that allows you to use (or to re-use) the same code for all the different camera interfaces.

pylon supports different operating systems (Windows, macOS, Linux) with its unified API. In this application note, the focus is on Linux for x86 and ARM-based embedded platforms (both 32 and 64 bit) as well as the specific circumstances that need to be considered when working with embedded platforms.

Generally, it is essential to differentiate between three different Linux approaches on embedded targets:

- Targets that run a Linux distribution that supports the Debian package manager dpkg (e.g., Debian, Ubuntu, Linaro, etc.)
- Targets that run a Linux distribution that doesn’t support the Debian package manager dpkg (e.g., RedHat, Suse, etc.)
- Targets for which a custom Linux needs to be built using the techniques provided by the Yocto project

2 Using pylon on Linux Distributions That Support the Debian Package Manager dpkg

Installing pylon on any embedded x86 or ARM target that runs a Debian-based Linux distribution (or any other Linux that supports the Debian package manager dpkg) is pretty straightforward and doesn’t differ from a computer running Ubuntu, for example.

1. Download the Debian installer package for pylon from the Basler website.

   Make sure that you download the package that matches your operating system / hardware architecture. There are different package versions for x86 or ARM, for 32 or 64 bit.

2. After downloading the package, open a shell, cd into the download location, and type:

   ```
   sudo dpkg -i <package-name>
   ```

   pylon is now installed on your system. The default location is **/opt/pylon5**, **/opt/pylon6**, or **/opt/pylon** (i.e., the pylon root directory). Depending on the performance of your embedded system, it may be necessary to manually perform some performance optimizations for GigE Vision or USB3 Vision cameras. The steps required are described in detail in the pylon README, which can be found in the pylon root directory after installation.

After the installation, you can immediately start the pylon Viewer.

For pylon 5, type the following in a shell:

```
/opt/pylon5/bin/PylonViewerApp
```

For pylon 6, type:

```
/opt/pylon6/bin/pylonviewer
```

or

```
/opt/pylon/bin/pylonviewer
```

depending on the pylon version used.

The C/C++ toolchain installed needs to be installed you your Linux distribution in order to create C/C++ applications.

To install the C++ toolchain, type:

```
sudo apt install build-essential
```

You are now ready to build and run pylon-based programs. To start with, you could build the pylon sample programs, which you can find in the **/opt/pylon5/Samples** or **/opt/pylon6/share/pylon/Samples** directories. Basler recommends copying the **Samples** directory into your home directory in order to have full user access. Example:

```
cp -r /opt/pylon5/Samples ~/Pylon-Samples
```

The samples are provided with make files. To build all C++ samples, for example, it is sufficient to go into the C++ subdirectory and to run make.

Basler dart BCON for MIPI Camera Modules Operated With Targets Based on Qualcomm Snapdragon 820

The Basler dart daA2500-60mc is a MIPI CSI-2 camera module designed to be operated with Qualcomm SD820-based embedded targets. Basler offers such a target as part of a dedicated development kit. Basler provides a complete Linaro Linux (Debian-based) image for this board, which you can download from the Basler website.

3 Using pylon on Linux Distributions That Don’t Support the Debian Package Manager dpkg

For those Linux distributions with different package managers than dpkg or those who have no package management at all, Basler also offers pylon for Linux as an archive file:

1. Download the pylon archive from the Basler website (compressed as gzip).

   Make sure that you download the package that matches your operating system / hardware architecture. There are different package versions for x86 or ARM, for 32 or 64 bit.

2. After downloading the package, open a shell, cd into the download location and type:

   ```
   sudo tar -xzf <archive-name>
   ```

   This extracts the archive and creates a subdirectory with the Basler sample programs (**Samples**) and a subdirectory named **pylon-<*version number*>-<*architecture*>**.
3. cd into the subdirectory **pylon-<*version number*>-<*architecture*>**.
4. Install the udev rules to set up permissions for Basler USB cameras.

   ```
   ./setup-usb.sh
   ```
5. Unplug and replug all Basler USB cameras to get the udev rules applied.
6. Extract the archive file containing the pylon SDK (also available in the new subdirectory).

   It needs to be extracted into the pylon root directory. Basler recommends making **/opt/pylon5** or **/opt/pylon6** the pylon root directory.

   pylon is now installed on your system. Depending on the performance of your embedded system, it may be necessary to manually perform some performance optimizations for GigE Vision or USB3 Vision cameras. The steps required are described in detail in the pylon README, which can be found in the pylon root directory after installation.

After the installation, you can immediately start the pylon Viewer.

For pylon 5, type the following in a shell:

```
<pylon root directory>/bin/PylonViewerApp
```

For pylon 6, type:

```
<pylon root directory>/bin/pylonviewer
```

You are now ready to build and run pylon-based programs. To start with, you could build the pylon sample programs, which you can find in the **Samples** directory mentioned above. Before building the samples, the privileges must be set accordingly:

```
chown -R <user>:<group> Samples
```

The samples are provided with make files. To build all C++ samples, for example, it is sufficient to go into the C++ subdirectory and to run make.

4 Using pylon on Targets That Require a Custom Yocto Linux

In contrast to the two approaches explained above where pylon is installed on an already existing Linux system, the Yocto-based approach aims at creating a complete Linux image that contains all required software packages, libraries etc., including for example the Basler pylon Camera Software Suite.

The open source Yocto Project (https://www.yoctoproject.org/) provides all the tools required for creating such a Linux image, the most important being a tool called BitBake. The main idea of Yocto is to create a target image by processing recipes that tell BitBake how to build the different components of the target system.

Each component is described as a so-called meta-layer in the recipe. Everyone who wants to enable integration of their product (e.g., pylon) into a Yocto-built image provides such meta-

layer(s). The meta-layers give BitBake the instructions for building and integrating this specific product. It's pretty much like cooking a hotpot (target image) using a cookbook (recipe) and a pot/oven (BitBake).

Yocto is the ideal technique for cooking a very specific and tailor-made Linux image for a specific (embedded) hardware target. It allows building a Linux system ("firmware") that contains everything the target needs, but not more than this. Another advantage of this approach is to be able to reproduce/rebuild exactly the same Linux image any time (as long as the recipes remain stable, which usually is the case).

Many embedded hardware vendors (e.g., SoC vendors like NXP with its i.MX8 SoC family) support Yocto by default. They offer meta-layers which allow building a Linux image, e.g., for their reference evaluation kits. By adding other meta-layers for additional software packages to the related Yocto project, the image can easily be extended, e.g., to support Basler cameras.

Basler offers Yocto meta-layers to support embedded platforms based on the NXP i.MX8 SoC family. These are provided as a so-called "Camera Enablement Package" (CEP), which you can download from the Basler website. There are different CEPs available that currently natively support NXPs official Evaluation Kits for the i.MX8MQ, the i.MX8M Mini, and the i.MX8QM Plus.

Basler's CEP contains two meta-layers:

- meta-basler-imx8: This layer contains kernel patches for the Basler camera drivers and the device tree for the NXP Evaluation Kits.
- meta-basler-tools: This layer contains the user space aspects of the pylon software (GenTL producer, the pylon SDK, the pylon Viewer, etc.). This meta-layer is independent of the actual SoC and can also be integrated into other ARMv8-based platforms that are not based on the i.MX8 (e.g., Nvidia Jetson).

In addition, the CEP contains a detailed README that explains step-by-step how to build a complete Linux image for the NXP Evaluation Kits.

The CEP can also be used to add pylon to any other hardware that is based on the supported NXP SoCs. In this case, however, it would be necessary to modify the device tree for the relevant hardware platform. Some SOM vendors (e.g., Variscite, SolidRun) supply their i.NX8-based SOMs with a device tree that already includes Basler camera support.

For building the Linux image with BitBake, Basler recommends using a Linux computer (e.g., Ubuntu) with sufficient performance (e.g., Core i7) and a fast SSD with 500 GB free disk space. The build process will still take several hours. To build the image, you need to install a number of tools (including the Yocto toolchain). This is also explained in detail in Basler's README.

After building the Linux image, pylon is installed on the target image system. Depending on the performance of your embedded system, it may be necessary to manually perform some performance optimizations for GigE Vision or USB3 Vision cameras. The steps required are described in detail in the pylon README, which can be found in the pylon root directory after installation.

Typically, you wouldn't build your own pylon-based application for the embedded target on the target itself. Also, you wouldn't usually have a complete C/C++ toolchain or SDKs on the target image as the image is pared down to what the final product actually needs – and nothing more. The typical approach would be to build your application on a dedicated Linux development computer (e.g., Ubuntu) with a cross toolchain and a cross SDK. The good news here is that you can easily generate cross toolchain and cross SDK out of the Yocto project with BitBake. This is all explained in Basler's README.

4.1 Building Basler pylon Samples With the Yocto Cross Toolchain

1. Create the cross toolchain (SDK) on your development computer.

   The procedure assumes that you have already built the Yocto Linux image for the embedded target (e.g., NXP EVK with i.MX8M Mini SoC) as described in the Basler README.

   a. Go into your Yocto build directory (e.g., **imx-yocto-bsp**). This directory contains the **setup-environment** environment setup script.

   b. Source the script with the target build directory as parameter, e.g.:

      ```
      $ source setup-environment build-xwayland-imx8mmevk-basler/
      ```

   c. Build cross toolchain with BitBake:

      ```
      $ bitbake -c populate_sdk fsl-image-validation-imx
      ```

      This may take some time.

2. Install the cross toolchain SDK on your development computer.

   a. cd into **imx-yocto-bsp/build-xwayland-imx8mqevk-basler/tmp/deploy/sdk**.

      Here, you can find the SDK just built as an installation script (e.g., **fsl-imx-xwayland-glibc-x86_64-fsl-image-validation-imx-aarch64-toolchain-4.14-sumo.sh**).

   b. Execute the SDK installation script. It prompts you to specify the installation directory for the cross toolchain SDK.

      The installation is now complete.

3. Cross-build pylon samples for aarch64.

   a. Open a terminal window on your development computer.

   b. For the cross toolchain, you have to source the environment. To do this, type:

      ```
      $ source <cross toolchain installdir> environment-setup-aarch64-poky-linux
      ```

      The pylon samples for aarch64 can be found, for example, in the meta-tools meta layer of the Basler CEP. **meta-basler-tools/meta-basler-common/recipes-camera/pylon/files/** contains a complete pylon installation packaged as a **tar.gz** file. This in turn contains the samples in the **share/pylon/Samples/** directory.

      It's best to extract the pylon **tar.gz** file in some directory, e.g., **~/pylon6-aarch64**, which then becomes the pylon root directory for aarch64.

   c. In order to build the C++ samples, cd into the C++ directory:

      ```
      $ cd ~/pylon-aarch64/share/pylon/Samples/C++
      ```

   d. You now have to source the pylon environment (e.g., PYLON_ROOT).

      This is done with the **pylon-setup-env.sh** script:

      ```
      $ source <pylon-installdir>/bin/pylon-setup-env.sh <pylon-installdir>
      ```

      Example: If you have extracted pylon to **pylon-aarch64**, you must type the following:

      ```
      $ source ~/pylon-aarch64/bin/pylon-setup-env.sh ~/pylon-aarch64
      ```

4. You can now run make to build all sample programs in one go. Afterwards, you may copy the built programs to the aarch64 target and run them from there.

5 Building, Running, and Debugging a Remote Application with Cross Toolchain and Qt Creator

Qt Creator is a user-friendly IDE that helps you to develop, build, run, and debug applications, e.g., using C++, and that can also be used for cross toolchains. In order to use Qt Creator, you need a CMake project (a **CMakeLists.txt** as related project file). Before using Qt Creator, you should first make sure that you can successfully build your application with CMake.

On your development computer, you need to install the cross toolchain SDK, as explained in step 2 page 6. You also need to install Qt Creator. On any Debian-based system (e.g., Ubuntu), you can simply install Qt create by typing:

```
$ sudo apt install qtcreator
```

You can now open a terminal window and source the environment for the cross toolchain as explained in step 3 on page 6. You can now start Qt Creator from within the terminal window:

```
$ qtcreator &
```

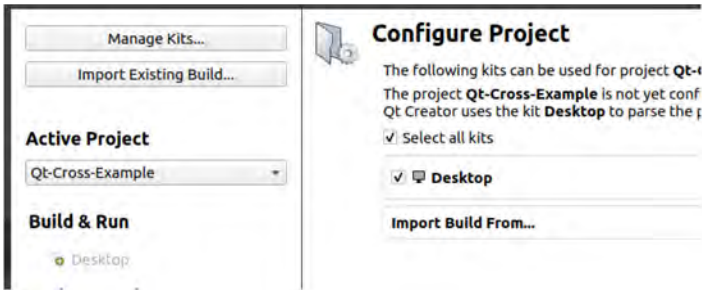
5.1 Creating a Kit for Cross Toolchain

It is important that you verify first whether you are able to build your project with CMake. In some cases, configuring the CMake project with Qt Creator may fail if you haven't created the CMake output before opening the project with Qt Creator.

1. In the Qt Creator menu, go to **File** -> **Open File or Project** and select your CMakeLists.txt as project.

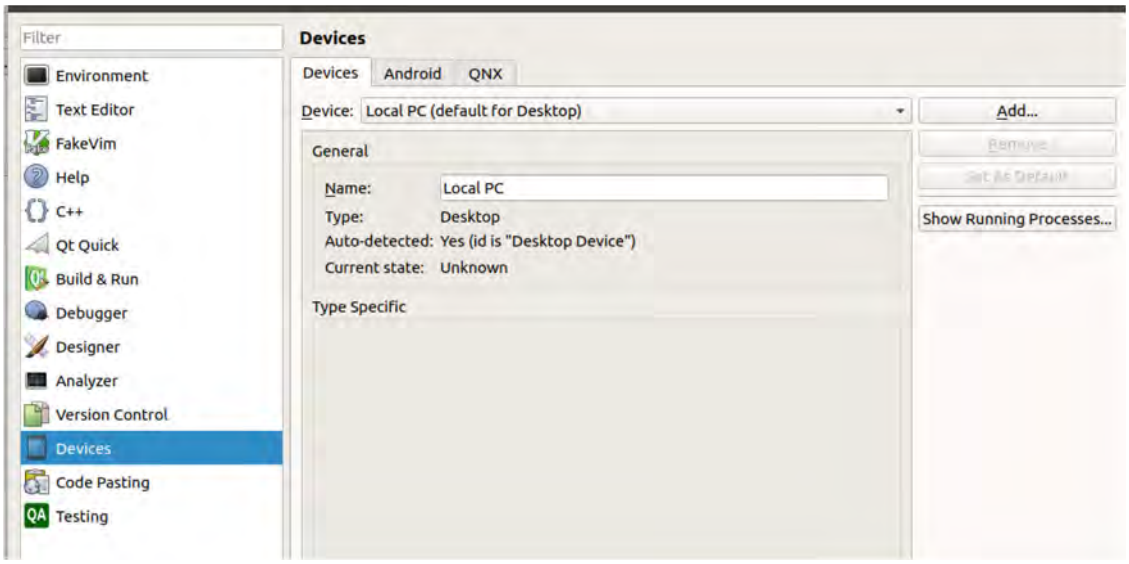
   After opening, you need to create a so-called kit that configures the build and debug environment for Qt Creator.

2. Click the **Manage Kits** button.



   The next step is to create a device that represents the remote target. Before doing so, make sure that your remote target is accessible via ssh.

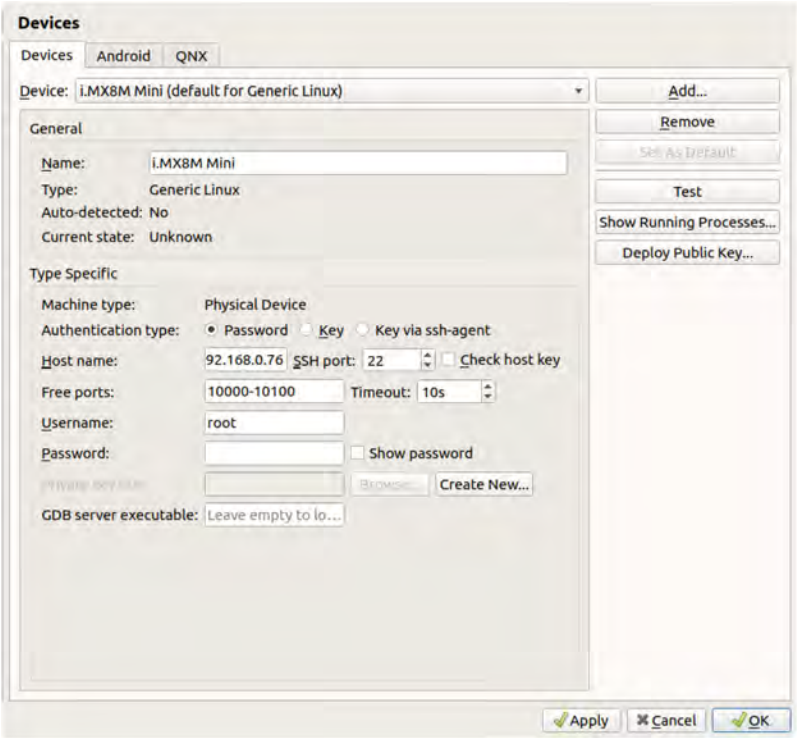
3. In the **Options** dialog, select **Devices** and click **Add**.



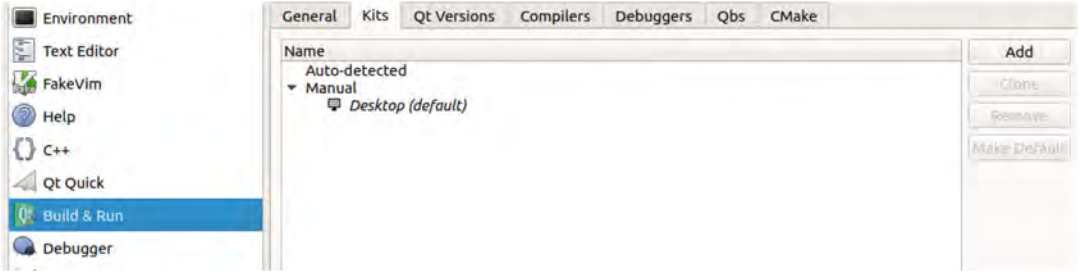
4. In the **Device Configuration Wizard Selection** dialog, select **Generic Linux Device** and click **Start Wizard**.

   This leads to a configuration dialog where you should give the target a meaningful name (e.g., i.MX8M Mini). Enter its IP address, user name, and password (if required).

5. If you click on **Next** the wizard performs a quick connectivity test. If the test succeeds, the **Devices** dialog opens.

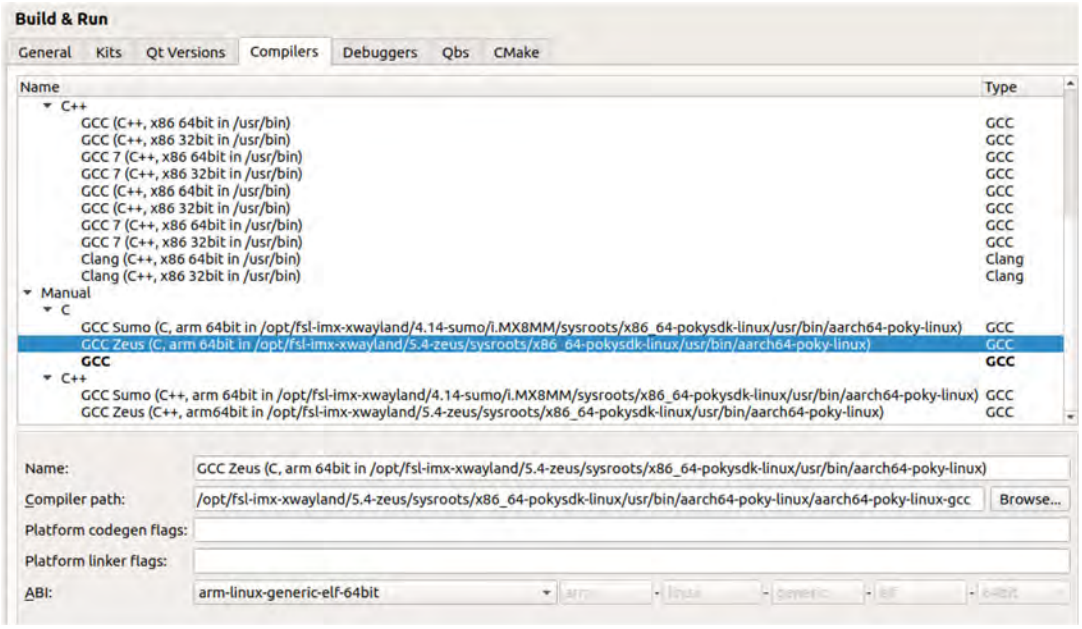


6. Click **OK** to finish the device configuration.
7. In the **Options** dialog, select **Build & Run**.

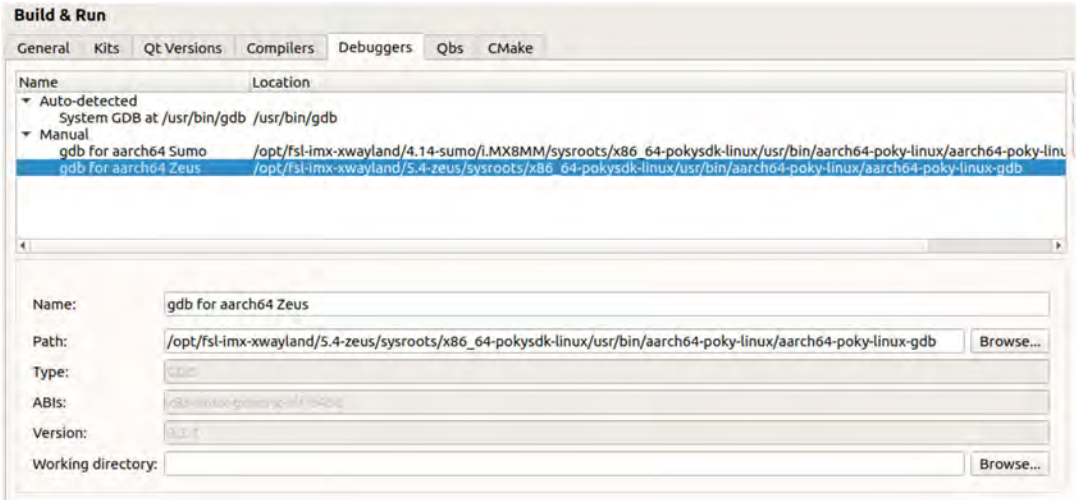


8. In the **Build & Run** dialog, go to the **Compilers** tab and click **Add**.

   You should now add both a C (gcc) and C++ (g++) compiler by browsing to the compilers of the cross SDK you installed previously. You should also give the compilers a meaningful name and you must set the ABI to arm-linux-generic-elf-64bit.

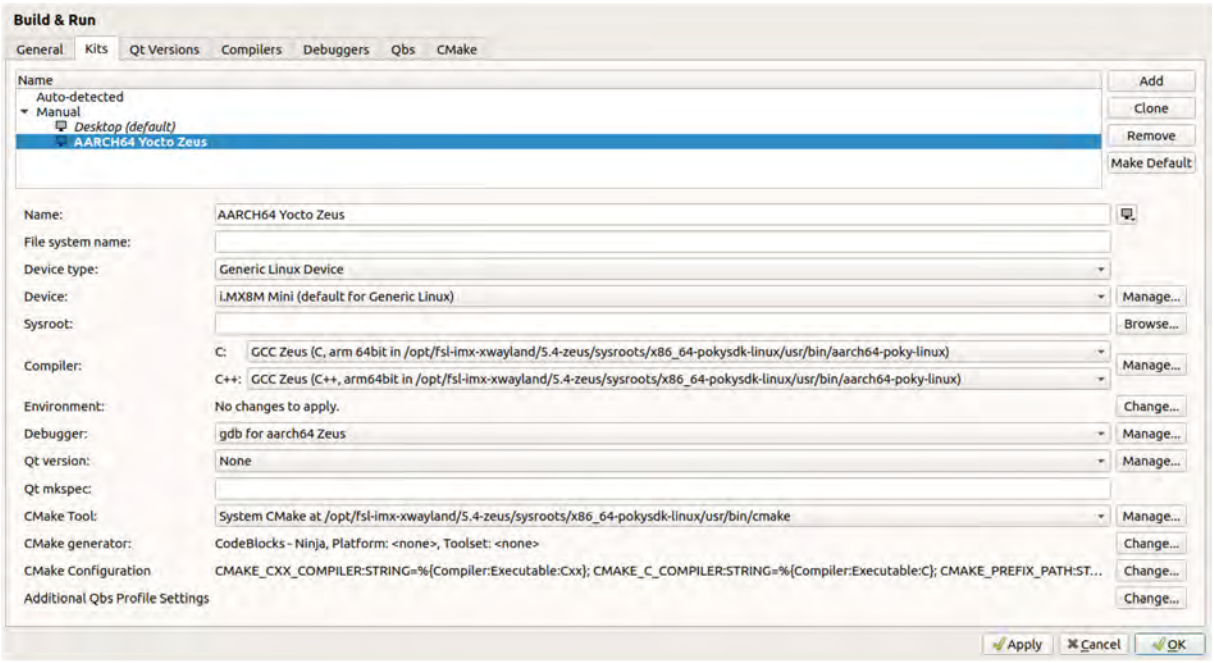


9. Now go to the **Debuggers** tab and add the cross SDK debugger to your kit.



   You should give the debugger a meaningful name (e.g., “gdb for aarch64 Zeus”).

10. Go to the **Kits** tab and click **Add** to finish configuring your kit.



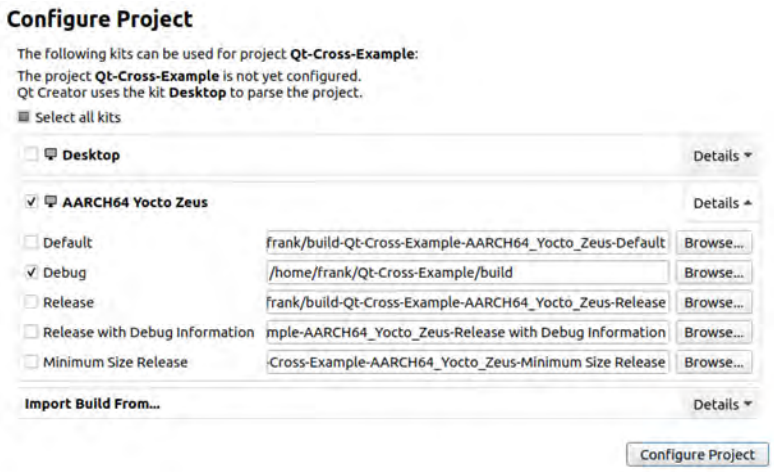
You should give the kit a meaningful name, e.g., AARCH64 Yocto Zeus.

11. Set **Device Type** to **Generic Linux Device**.
12. Set **Device** to what you just created (e.g., “i.MX8M Mini”).
13. Set the C and C++ compilers to the ones you just added and the debugger to the one you just added.
14. Set **CMake Tool** to the one that is part of your cross SDK.
15. Click **OK**.

    The kit creation is now complete.

5.2 Configuring your project

1. Select the kit with which you want to build (e.g., “AARCH64 Yocto Zeus”) and expand the **Details** tab.



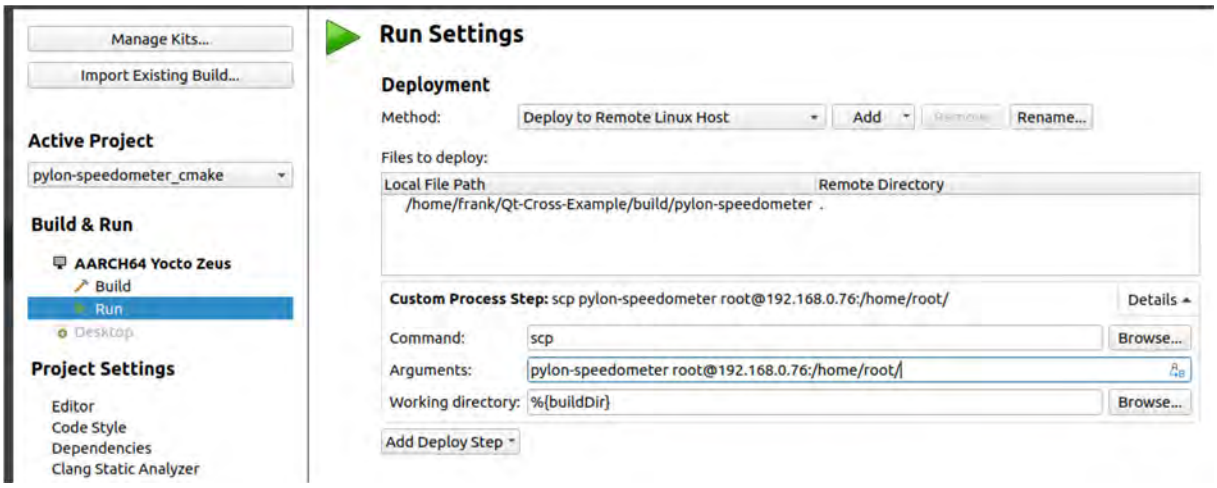
2. Select all the build configurations you want to compile for (and debug).
3. Specify the build folder for the build artifacts.
4. Click **Configure Project**.

   You should now be able to build your application with Qt Creator (Ctrl + B).

5.3 Deploying the Built Program to the Remote Target and Debugging It

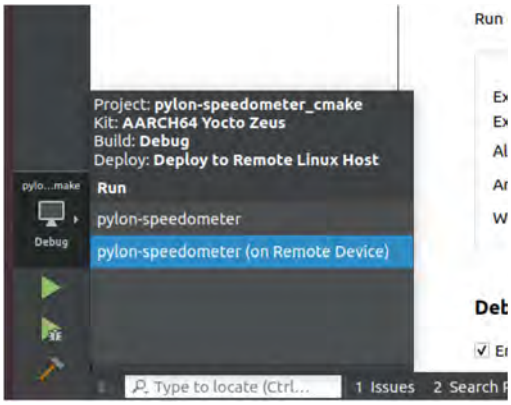
1. Click the **Projects** button in the left window pane and under **Build & Run** select **Run**.

   You are now in the **Run Settings** dialog.
2. Delete any existing deploy step (usually, there is a default deploy step that uses sftp).
3. Click **Add Deploy Step** and select **Custom Process Step**.



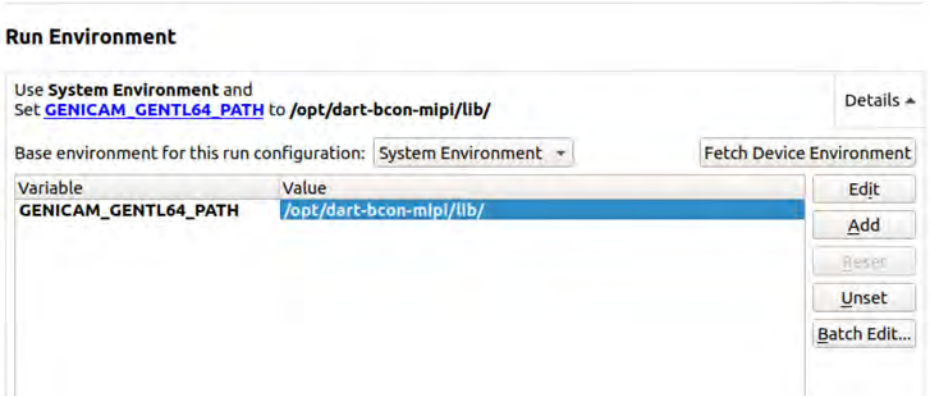
4. As command, enter “scp”.
5. As arguments, enter what you want to copy to which location using this format:

   `<program name> <username>@<ip address>:/<remote location>`
6. Click the **Debug** button in the left pane and tell Qt that you want to debug on the remote device.



As a last step, you still have to set the GENICAM_GENTL64_PATH environment variable to **/opt/dart-bcon-mipi/lib/**.

7. Go to **Run Environment**, open the **Details**, and click **Add** to set the variable accordingly.



You’re now ready to remotely run and debug your application with Qt creator as IDE.

Revision History

Document Number	Date	Changes
AW00160901000	28 April 2020	Initial release version of this document.
AW00160902000	1 Feb 2021	Minor corrections. Added chapter 5.