

CC Linux

Programmer's Guide



Contents

1	Introduction	4
1.1	Convention and Definitions.....	4
1.2	References.....	4
2	Interfaces	5
2.1	Standard libraries	5
2.2	CCAux library	5
2.3	CCAux API calling convention.....	7
3	Board Support Package	8
3.1	Downloading and installing the BSP.....	8
3.2	BSP structure	8
3.3	Using the BSP	11
4	Software Development Kit	17
4.1	Downloading and installing the SDK	17
4.2	Using the SDK.....	17
4.3	Debugging remotely	18
5	Special considerations	20
5.1	Ethernet, setting a static IP-address	20
5.2	CAN	20
5.3	Analog video.....	22
5.4	Graphics, Qt (without Weston)	23
5.5	Graphics, Weston	24
5.6	Serial Number Broadcast interface	24
5.7	Polarity of PWM outputs	25
5.8	Suspend	25
5.9	General-purpose input/output (GPIO)	25
6	Build Examples	26
6.1	Building applications with the SDK.....	26
6.2	Qt application development.....	27
6.3	Building any (platform supported) version of Qt with the SDK.....	27
7	Using an IDE in CC Linux application development	30
7.1	vscode integration	30
7.2	qtcreator integration.....	34
8	Working with Systemd	37
8.1	Creating services with dependencies	37
8.2	Viewing dependencies with systemctl list-dependencies	38
8.3	Adding systemd .devices	39
	Technical support	41
	Trademarks and terms of use	42

Revision history

Revision	Date	Comments
1.4.0	2018-11-22	Released. Document compatible with devices running CC Linux 1.4.x.x
1.4.1	2019-08-26	Added chapter on peripherals turned off during suspend. Document compatible with devices running CC Linux 1.4.x.x
1.4.2	2019-12-11	Added CCpilot X900 related information Document compatible with devices running CC Linux 1.4.x.x
1.5.0	2020-08-28	Document compatible with devices running CC Linux 1.5.x.x
2.0.2	2021-05-17	Document compatible with devices running CC Linux 2.0.x.x
2.0.3	2021-12-22	Added CCpilot V1000/V1200 related information with IDE examples
2.0.4	2022-03-03	Added Yukon development board related information.
2.0.5	2023-01-04	Added information for building boot-up time optimized images.
2.0.6	2022-02-24	Added information for configuring CAN timing settings
2.0.7	2022-03-03	CfgIn and PWMOut is now supported for Yukon

1 Introduction

This document contains reference information describing application development and APIs used when developing applications for the CCpilot products supported in the CC Linux platform. Additionally, this document contains information on how to build a custom Linux operating system for your device based on the CC Linux reference system.

Several functionalities are available using the operating system or standard APIs. These may be briefly mentioned but are not described in detail within this documentation.

A good prior understanding of Linux programming is needed to fully benefit from this documentation. It is also recommended to read the CC Linux - *Software Guide* prior to reading this document.

1.1 Convention and Definitions

This document covers all devices included in the CC Linux platform. For the sake of example, the CCpilot VS device is sometimes used throughout this document. Any significant device deviations will be stated. When the <xx> is used, it should be replaced with the proper device name (VS, for instance).



The observe symbol is used to highlight information in this document, such as differences between product models.



The exclamation symbol is used to highlight important information.

Text formats used in this document:

Format	Use
<i>Italics</i>	Paths, filenames, definitions.
Bolded	Command names and important information

1.2 References

For further information on the device software and the available APIs see the following references.

- [1] CC Linux – Software Guide
- [2] CCpilot VI - Technical Manual
- [3] CCpilot VS - Technical Manual
- [4] CCpilot V700 – Technical Manual
- [5] CCpilot V1000/V1200 – Technical Manual
- [6] CCAux API documentation
- [7] Telematics API documentation
- [8] Yocto Project Development Tasks Manual:
www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html
- [9] Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual:
<https://www.yoctoproject.org/docs/3.0.3/sdk-manual/sdk-manual.html>

2 Interfaces

This section covers basic information on how to access the device hardware. Most of the hardware is accessed using the default Linux interfaces but some specific interfaces may require additional functions to be accessed.

Table 1 lists the API used to access each interface. These interfaces can be grouped into two categories; Standard operating system libraries (Std. API) and CCAux Library (CCAux API).

See also the operating system specific sections and additional documentation describing the software API.

Table 1: APIs used for different interfaces in CCPilot devices

Functionality	Std. API	CCAux API	Comment
CAN	√		SocketCAN available through Linux.
Ethernet	√		Standard Linux kernel drivers.
USB	√		Standard Linux kernel drivers.
RS232	√		Standard Linux kernel drivers.
RS485	√		Standard Linux kernel drivers.
Digital video	√		QtMultimedia, gstreamer ¹
Analog video in	√	√	Video4Linux2 API, gstreamer, and QtMultimedia can be used directly without CCAux API.
Digital output		√	
PWM output		√	
Configurable input		√	
Analog input		√	
Status indicator		√	
Backlight		√	
Ambient light sensor		√	
Buzzer		√	
Power management		√	

¹ On CCPilot VI the video stream needs to be rotated counter clockwise due to display orientation. This can be done by setting the *videoflip* method to *counterclockwise* in the gstreamer pipeline.



Your device might not support all interfaces stated here. See the *Product Leaflet* or *Technical Manual* of your device for a complete list of available interfaces.

2.1 Standard libraries

Most interfaces are accessed using standard libraries and access methods. Various access methods are possible to be used depending on the development environment and additional installed frameworks.

The standard libraries used for Linux are described in their respective documentation sources, such as MAN pages.

2.2 CCAux library


The CCAux API gives access to several hardware specific interfaces. The API functions of this library are documented in the *CCAux API documentation*.

Table 2 gives a brief introduction on the API's found therein and their functions. Most API functions can be used from the pre-installed *CCSettingsConsole* application as well.

Table 2: Short description of CCAux API contents and the supported devices

API	Description	Supported on device					
		VS	VI	X900	V700	V1x00	Yukon
About	Hardware information API related to the hardware configurations.	Yes	Yes	Yes	Yes	Yes	Yes
Adc	Read built in ADC voltage information.	Yes	Yes	Yes	Yes	Yes	Yes
AuxVersion	Read firmware version information.	Yes	Yes	Yes	Yes	Yes	Yes
Backlight	Control display backlight settings and configure automatic backlight functionality.	Yes	Yes	Yes	Yes	Yes	Yes
Battery	Control battery related settings, if a battery is connected.	No	No	No	No	No	No
Buzzer	Control the built-in buzzer.	Yes	Yes	Yes	Yes	Yes	Yes
CanSetting	Control CAN settings. ¹	Yes	Yes	Yes	Yes	Yes	Yes
CfgIn	Get/set state of configurable input signals.	Yes	Yes	No	No	No	Yes
Config	Control internal and external power up and power down settings and time configurations, including power button and on/off signal configurations.	Yes	Yes	Yes	Yes	Yes	Yes
Diagnostic	Get run time information about the device.	Yes	Yes	Yes	Yes	Yes	Yes
DigIO	Get/set state of Digital Output signals.	Yes	No	No	No	No	No
FirmwareUpdate	Update/verify System Supervisor (SS) and other firmware. ²	Yes	Yes	Yes	Yes	Yes	Yes
FrontLED	Override the default LED behavior.	Yes ³	Yes ⁴	Yes	Yes	Yes	Yes
LightSensor	Read the light sensor values and get raw and/or calculated values.	Yes	No	Yes	Yes	Yes	Yes
Power	Read power status and control functions for advanced shut down behavior.	Yes	Yes	Yes	Yes	Yes	Yes
PWMOut	Get/set state and settings of PWM Output signals.	No	Yes	No	No	No	Yes
SoftKey	Get press status of buttons. Get/set button backlight settings.	No	Yes	No	No	No	No
Video	Control the analog video streams in terms of channel, size, scaling etc.	Yes	No	No	No	No	No

¹ Most settings for CAN usage are available over the SocketCAN interface.

 ² **Consider careful usage for these functions, erroneous usage can result in a non-functional device.**

³ CCPilot VS does not have a front LED, so this API controls the button backlight LED instead.

⁴ CCPilot VI does not have a front LED, so this API controls the button backlight LEDs instead.

Additionally, the FrontLED class is deprecated on VI as it handles RGB LEDs which VI button backlight does not

have. Calling this class on VI will result in a grey-scaled mapping in order to keep backwards compatibility. For new developments, the SoftKey class should be used instead.

2.3 CCAux API calling convention

The standard way to call CCAux API functions is shown below. Please adhere to this calling convention. Example code snippets for each function are available in the *CCAux API documentation*. Moreover, chapter 6.1 gives an example on how to call the CCAux API in order to set the front LED color.

```
/* Usage in CCAux API 2.x */  
#include "Module.h"  
  
MODULEHANDLE pModule = CrossControl::GetModule();  
eErr err = Module_function_1(pModule, arg, ...);  
Module_release(pModule);
```

3 Board Support Package

To provide the possibility to create custom Linux images for CCpilot devices, a board support package (BSP) has been created. The BSP is a Yocto-project build system that produces complete Linux images for the CCpilot devices. It also includes the necessary application and driver code, as well as example or template code that may serve as a basis for further application and driver development.

The open-source Yocto system has built-in package support with many thousands of maintained packages available, while also providing a set of standard tools and build guidelines. The BSP adds necessary drivers and applications required for the CCpilot board images. For more information regarding the Yocto project, please refer to the *Yocto Project Development Tasks Manual* [7].

3.1 Downloading and installing the BSP

The BSP package comes in the form of a gzipped tar archive which can be acquired from the CrossControl support site. The BSP was created on an x86-64 machine running Ubuntu 18.04.5 LTS and must be unpacked to a Linux host machine/build server. A large amount of disk space (at least several hundred gigabytes) and memory is required for the Yocto system, as the directory structure tends to grow quickly in size. Please refer to the *Yocto Project Development Tasks Manual* [7] for information about required dependencies.

3.2 BSP structure

This chapter will go through CrossControl specific parts of the BSP structure. For other third-party components (OpenEmbedded, Yocto, etc.), please refer to their respective reference documentation.

Initially, the BSP base directory will only contain CrossControl directories. All other required third-party tools are provided as hash-files which can be downloaded from remote repositories upon first build. The following subsections will describe these initial directories.



Internet connection is required to build a BSP.



While most components in the BSP are released under various open-source licenses, others are of proprietary nature and protected under the CrossControl Software License. Be aware of which license applies and please comply with the license terms.



VS is the internal CrossControl code name for the CCpilot VS architecture/platform; therefore, the BSP contains references to the VS platform such as "PLATFORM_VS". The same applies for other CCpilot devices:

Device	Code name
CCpilot VS 12"	vs
CCpilot Vi 2 nd gen	vi2
CCpilot X900	xm9
CCpilot V700	v700
CCpilot V1000/V1200	v1x00
Yukon development board	yukon

Additionally, CrossControl specific components are named “cc”, such as “meta-cc” and “recipes-cc”.

3.2.1 apps

The *apps* directory contains the source code to CrossControl-developed libraries and applications such as *CCAux API library*, *CCAux daemon*, *CCSettingsConsole*, etc.

3.2.2 drivers

The *drivers* directory contains the source code to the following CrossControl-developed drivers:

ss

Handles communications between the System Supervisor (SS) and the main processor (MP, i.e. the Linux system) over the MP-SS SPI or I2C bus. This driver is required for the *CCAux API* to work.

<xx>-io

Is used to set the direction of the internal UART, in order to support either bootloader output to the update board or to the serial port of the SS.

3.2.3 meta-3rd-party

Table 3: Content of the subdirectories in the meta-3rd-party directory in the BSP.

Directory	Content
meta-kontron	This is the base (meta) layer that contains various 3-rd party and Kontron software which is common for all x86 boards produced by Kontron Europe.

3.2.4 meta-cc

The *meta-cc* directory contains the Yocto layer with all CrossControl developed board-specific recipes. See Table 4 for more details of the subdirectories.

Table 4: Content of the subdirectories in the meta-cc directory in the BSP.

Directory	Content
conf	meta-cc layer configuration files for the <xx> machines. The machines are based on more general machines. For instance, the 'vs' machine is based on the 'mx6q' machine, which in turn is based on the 'mx6' machine in the meta-freescale layer.
recipes-bsp	Contains the u-boot specific patches which adapt the default u-boot configuration for the device's chipset.
recipes-cc	Contains recipes for CrossControl developed libraries and applications. These have the source code stored locally in the apps/ directory.
recipes-connectivity	Contains recipes for Ethernet configuration.
recipes-core	<ul style="list-style-type: none"> Image recipes (both for the main and rescue system) for all supported devices; these are used to produce the reference CC Linux image. This is a good starting place to add or remove features from the image. Psplash recipe, used to create a boot splash image during the time in which the Linux system loads Init script recipes

recipes-devtools	A recipe for input utilities, which are used to enable the touch panel driver for devices with a touch panel, such as CCPilot VS.
recipes-kernel	<ul style="list-style-type: none"> Kernel recipes; board-specific kernel patches, device tree, kernel configuration files
recipes-multimedia	<p>Recipes for the CrossControl developed drivers in the drivers directory</p> <ul style="list-style-type: none"> Contains recipes for adding mp4 video format support and Qt integration to gstreamer.

3.2.5 meta-freescale-append

Append and configuration files for meta-freescale Yocto layer. See Table 5 for more details of the subdirectories.

Table 5: Content of the subdirectories in the meta-freescale-append directory in the BSP.

Directory	Content
conf	meta-freescale layer configuration files.
recipes-bsp	Placeholder directory for bsp append files
recipes-fsl	Placeholder directory for fsl append files
recipes-graphics	Placeholder directory for graphics append files

3.2.6 meta-freescale-distro-append

Append and configuration files for meta-freescale-distro Yocto layer. See Table 6 for more details of the subdirectories.

Table 6: Content of the subdirectories in the meta-freescale-distro-append directory in the BSP.

Directory	Content
conf	meta-freescale-distro layer configuration files.

3.2.7 meta-imx-append

Append and configuration files for meta-imx Yocto layer.

3.2.8 meta-openembedded-append

Append and configuration files for meta-openembedded Yocto layer.

3.2.9 meta-intel-append

Append and configuration files for meta-intel Yocto layer.

3.2.10 Poky-append

Append and configuration files for poky Yocto layer.

3.2.11 platform

The *platform* directory contains a directory for each of the CC Linux devices as well as one *common* directory. These directories contain environment-setup scripts, extra Makefile rules to build individual system components with bitbake, and the configuration files needed for configuring the build system. The file *local.conf* can be modified to change the default Yocto download

directory etc. If you want to add a new layer to the Yocto build, it must be added to the *bblayers.conf* file.

When building, the working directory will be located in *platform/<xx>/build*.

3.2.12 tools

The *tools* directory contains scripts to be used for device OS update. See the *CC Linux – Software Guide* for instructions on how to use them.

3.2.13 uuu-package

Contains necessary binaries and template files to use with flashing the processor with the NXP provided uuu-tool.

3.2.14 binaries

All built images, update files, etc. are placed in the *binaries* directory which is created upon the first build.

3.3 Using the BSP

The purpose of this chapter is not to give the reader a full description of the Yocto build system and all its possibilities; but rather give useful tips on how to build Linux images and application binaries from within the BSP.

3.3.1 Building Linux images

Out of the box, the BSP produces a CC Linux image containing a reference implementation. The reference implementation is intended to be used as a basis for creating custom images.



The first time building the image can take several hours, depending on your host machine. Subsequent builds will be quicker as Yocto reuses all unchanged components from previous builds.

Depending on your host machine, you might need to edit the *platform/<xx>/local.conf* file. For instance, the default download directory can be changed; and if you have limited disk space, there is a setting for removing temporary files once builds are completed. However, removing temporary files will slow down your build process.

In the BSP root directory there is a Makefile containing rules to make main and rescue images for all CC Linux devices. For instance, to make the CCpilot V700 releaseimage, type

```
$ make v700-release-image
```

make will automatically source the environment scripts in the platform directory and invoke bitbake to bake the image recipe in the *meta-cc/recipes-core/images* directory.

The resulting image is located in *platform/<xx>*.

To build your own custom Linux image, either edit the image recipe right away, or make a new recipe and add a new rule for it to the Makefile. If choosing the latter, don't forget to add a rule to the *platform/<xx>/Makefile* as well.



See the *CC Linux - Software Guide* for instructions on how to program the images to the device.

3.3.2 Building boot-up time optimized image

The CCLinux distribution features are intended for a wide variety of use cases. This is also reflected in the base distribution boot-up time performance. Supporting a wide variety of use cases affects the boot-up time to some extent. There is, however, an optimization flag which can be set in the platform local.conf. Although the flag is possible to set for all CCLinux distributions, the intended and tested usage is primarily with the V1000/V1200 devices. To enable the flag, uncomment the following line in local.conf:

```
DISTRO_FEATURES:append = " minimalboot"
```

Then you can build the image as usual:

```
make v1x00-release-image
```

Enabling the flag builds an image with the following differences compared to a regular CCLinux image:

- The default init system is changed to minit (minimal init) which brings up a user interface with a set of minimal services (CAN0 and eth0).
- A set of boot-up time optimizations are enabled to system components which may not be fully compatible with regular CCLinux distribution features.

Using Tera Term to measure boot-up time

The system boot-up chain consists of the following components:

Component	Description
SS (System Supervisor)	An auxiliary microcontroller-based CPU which does the initial power-up sequence for the system. The SS firmware is provided by CrossControl.
BootROM (SCU)	The System Controller (SCU) takes care of executing the initial stages of the BootROM (the primary program loader), stored in its Read Only Memory (ROM). It reads the boot mode pins, configures DDR and loads its own image as well as the bootloader image.
ATF	The Arm Trusted Firmware (ATF) provides a reference trusted code base for the Armv8 architecture. The binary is included in the bootloader binary. It starts in the early stages of U-Boot.
U-Boot	U-Boot configures some additional hardware devices and prepares the system to boot-up the kernel by loading device tree (FDT) and the kernel image.
Linux Kernel	The kernel configures hardware devices and manages the system resources. After the initial hardware configuration done, the kernel starts the user space with the system init.
System Init	The system init loads the kernel modules and starts system daemons and applications.

Tera Term is a serial terminal which can be used for capturing boot-up logs with timestamps. Firstly enable the console timestamps by selecting the *File -> Log* menu and select *Elapsed Time (Logging)* in the dropdown menu. The timestamps are relative to the PC clock. To measure time taken for an individual component in the boot, find the relevant log line in the boot-up log. Then find the timestamp for the next component and calculate the differences of the timestamps.

Measuring the kernel boot-up time

For measuring the kernel boot-up time, the following configuration needs to be done:

Build a debug kernel so that `CONFIG_PRINTK_TIME` and `CONFIG_KALLSYMS` are enabled. Edit `linux-imx_5.15.bbappend` append in the meta-imx-append layer and comment out lines beginning with `release-kernel.cfg`. This removes the `release-kernel.cfg` fragment and leaves the needed debug options to the kernel.

Append the option `initcall_debug` to the kernel command line. This can be done in the following U-Boot patch: `meta-bsp/recipes-bsp/u-boot/u-boot-imx/0009-v1x00-quiet-boot.patch` or a new patch can be created which modifies the kernel command line accordingly.

Rebuild the image and boot up the image. You should see the following kind of information in dmesg:

```
[ 0.140513] calling acpi_init+0x0/0x3f8 @ 1
[ 0.140565] ACPI: Interpreter disabled.
[ 0.140570] initcall acpi_init+0x0/0x3f8 returned -19 after 0 usecs
[ 0.140584] calling pnp_init+0x0/0x28 @ 1
[ 0.140623] initcall pnp_init+0x0/0x28 returned 0 after 0 usecs
```

Now, export the dmesg and produce the boot-up graph. The `bootgraph.pl` in the kernel scripts directory can be used for this purpose:

```
dmesg > boot.log
./bootgraph.pl boot.log > boot.svg
```

Customizing the system init

The system init is the first component in Linux, which is executed after the kernel is ready to start the user space. The CCLinux distribution uses the systemd system init. However, for the purpose of a fast-booting system, systemd has some overhead. When the `minimalboot` distribution flag is enabled in `local.conf`, the default init is changed to minit (minimal init). The minimal is by purpose kept very simple to have only a minimal set of features for starting a single application. The default minit init script is contained in the meta-cc layer and is written in bash (`recipes-cc/minit/files/minit.default`). The startup order is the following:

- Load early kernel modules which are needed for the system. This includes for instance the SS kernel module for the communication with SS.
- Start the application.
- Late load kernel modules. This includes ethernet and CAN.
- Late launch services. The first step is to let the system settle to reduce the possible application load. Then start ccauxd, getty and configure ethernet and CAN.
- Wait indefinitely for the system termination.

There are couple of ways to measure the time taken for startup in the system init:

- systemd
 - Use the `systemd-analyze plot` command to create a boot-up diagram with the timing for the services started in the boot.
- minit
 - Minit prints timestamps in the boot which can be observed with a serial console.

Recommendations for a fast-booting application

Fast startup should be taken as a design concern since the application is created and not as an afterthought. It can be time-consuming to optimize an existing application if the initial boot-up time has not been a concern when creating the application. Here is a brief list of things that should be considering for the application start up performance:

- Use KMS (EGLFS in Qt) for OpenGL rendering. Wayland/Weston startup adds an additional delay before an application can be started.
- Use the best optimization flags the compiler provides, for instance `-O3` with `gcc`.
- Consider implementing part of the functionality as dynamically loaded libraries (plugins) so that the main application binary needs to be smaller.
- Do not expect hardware devices to be available right at the application startup.
 - For instance, CAN / network connectivity should be done in a manner that there is a separate callback in the application when CAN / network connectivity is available, and the application should be able to start without CAN. The worst kind of implementation is where the application keeps polling for the hardware interfaces during startup and does not start before the interfaces are available.
- Use lazy loading for resources such as graphics. Do not load all the image assets during the application startup. Instead, load only those assets that are needed for the startup. The rest of the assets can be loaded in a background thread.
- Do not overload the main thread. Instead use multiple threads for tasks. Try to keep the main loop as lean as possible so that events can be processed in a short response time. This will also create a better user experience.
- If using Qt and QML, notice that loading QML can be slow without precompiled QML. Consider using the QML compiler to precompile the QML assets.
- JavaScript logic should be avoided in QML, use C++ instead to implement the logic and call the C++ functions from QML.

3.3.3 Building applications

Provided there exists a recipe for the application, building with the BSP can be done in two ways. The first method is the most straightforward whereas the second can be more convenient during development as it is faster and more flexible.

First method:

1. Include the recipe for the application in the image recipe:

```
IMAGE_INSTALL += "applicationname"
```

2. Make the image as usual:

```
$ make <imagerule>
```

Second method:

1. Make sure you are in the `platform/<xx>` directory and source the environment script:

```
$ source oe-env
```

2. Use `bitbake` to build your application:

```
$ bitbake <applicationname>
```

Third method:

You may also use devtool to edit and modify the program developed. For details, see the Yocto documentation on how to work with devtool.

A brief example is shown here.

1. Run `devtool modify <recipename>`. This will fetch the sources for the recipe and unpack them to a `workspace/sources/<recipename>` directory and initialise it as a git repository if it isn't already one. If you prefer you can specify your own path, or if you already have your own existing source tree you can specify the path along with the `-n` option to use that instead of unpacking a new one.
2. Make the changes you want to make to the source
3. Run a build to test your changes - you can `bitbake <recipename>` or build an entire image incorporating the changes assuming a package produced by the recipe is part of an image. There's no need to force anything - the build system will detect changes to the source and recompile as necessary.
4. If you wish, test your changes on the target. There's a "devtool deploy-target" command which will copy the files installed at `do_install` over to the target machine assuming it has network access, and any dependencies are already present in the image.
5. Repeat from step 2 as needed until you're happy with the results.
6. At this point you will almost certainly want to place your changes in the form of a patch to be applied from the metadata - devtool provides help with this as well. Commit your changes using "git commit" (as many or as few commits as you'd like) and then run either:
 - o `devtool update-recipe <recipename>` to update the original recipe - usually appropriate if it's your own recipe or you're submitting the changes back to the upstream layer
 - o `devtool update-recipe -a <layerpath> <recipename>` to put your changes in the form of a `bbappend` to be applied by a different layer. This is usually the desired method if your changes are customisations rather than bugfixes.
7. If you're finished working on the recipe, run `devtool reset <recipename>`.

All methods will put the resulting application binaries in `platform/<xx>/build/tmp/work`.



In the second method, additional bitbake commands can be invoked. For instance, one can use the `-c` flag to recompile the application without re-fetching the source files.

4 Software Development Kit

This section is dedicated to useful tips and hints about how to use the Software Development Kit (SDK) on a Linux development host machine for application development and debugging purposes.

The SDK is based on the Yocto Project SDK. Refer to the *Yocto Project Application Development and the Extensible Software Development Kit (eSDK)* [8] manual for additional information and tips.



On some platforms, a separate debug-sdk is available on request to debug system libraries. Additionally, this can be built from the BSP. This option is not available for the Yukon and V700 platforms, please consult the Programmer's manual for further instructions on debugging system libraries.

4.1 Downloading and installing the SDK

The toolchain included in the SDK is an ARM GNU/Linux cross compiler based on the standard GNU GCC compiler toolchain. Additionally, the SDK contains header and library files for CCAux API and other peripherals. In order to download the SDK, visit the CrossControl support site. For support on SDK issues, please contact CrossControl directly.

The SDK comes in the form of a self-extracting shell script. It contains precompiled binaries for Linux host systems. There are two versions of the SDK; one for 32-bit i686 hosts and one for 64-bit x86 hosts. To install the SDK, move the script to the host computer and run the following shell command:

```
$ sh scriptname.sh -d <sdk install dir>
```

Omitting the `-d` flag will install the SDK to the default directory, `/opt/cclinux/1.0.0`.

Example output from installing the CCpilot VS SDK to `/opt/sdk/`:

```
$ sh CCLinux-SDK-toolchain-x86_64-CCpilot-VS-v1.4.1.0.sh -d /opt/sdk/  
CCLinux Distribution SDK installer version 1.0.0  
=====
```

You are about to install the SDK to "/home/lisa/vs-sdk/sdkmapp".
Proceed[Y/n]? Y
Extracting SDK.....done
Setting it up...done
SDK has been successfully set up and is ready to be used.

4.2 Using the SDK

4.2.1 Starting a new session

Once installed, the SDK contains the `sysroots` directory with target and host system root file systems, and an environment setup file; `environment-setup-<XXXX>` (*file ending varies depending on your platform*). The environment setup file exports the correct `$PATH` and other important build environment variables, such as `$CC` and `$CFLAGS`. (Read the file content to get an understanding of the default settings).

This file needs to be sourced before using the SDK each time a new shell session is started:

```
$ source environment-setup-<XXXX>
```

Now the cross-compiler is ready to use.

4.2.2 Using correct development headers

The SDK is released containing binary images. This package contains libraries available at the unit and all header files for utilizing them. The header files are located in:

```
<sdk-install-dir>/sysroots/<targetsysroot>/usr/include/
```

and the libraries are found in these two directories:

```
<sdk-install-dir>/sysroots/<targetsysroot>/lib/
```

```
<sdk-install-dir>/sysroots/<targetsysroot>/usr/lib/
```

These directories are automatically added to the search path when sourcing the environment setup script and don't need any special consideration.

If additional development files placed outside of the SDK installation directory are to be used they can be added to the compiler search path by appending the `$CFLAGS/$CXXFLAGS` variable. This can be done either in the project settings or in the *Makefile* using:

```
CFLAGS+= -I<path-to-headerfile-dir> -L<path-to-library-dir>
```

4.2.3 Compiler optimizations

The environment setup script automatically sets the `-mcpu=<cpu>` compiler flag for optimizing the code for the instruction set generated for the specific processor. Additionally, floating point computations are automatically set to use the available hardware acceleration.

To see which optimizations are done by default, study the content of the environment setup script. All these settings can be overridden after the script has been sourced. See the compiler documentation for additional information about available optimizer flags.

4.2.4 Special considerations using CCAux API

In order to build applications using functions from the CCAux API library, two steps need to be given special consideration in either the project file or the *Makefile*:

1. The `$LD` variable should be overridden to use the same content as the `$CXX` variable
2. The `$CFLAGS/$CXXFLAGS` must be appended with the `-DLINUX` flag

If either of these steps are omitted, the build will fail. See chapter 6.1 for examples of how to build CCAux API applications.

4.3 Debugging remotely



To debug system libraries, you must have the `sdk-with-debugging` installed on your host machine and use it to build your application.

To use **GDB** to debug an application running on the device, the application must have been compiled with the `-g` flag. Start **gdbserver** on the device:

```
~$ gdbserver :10000 testApplication
```

Then start the host **GDB** and connect to the server:

```
$ arm-poky-linux-gnueabi-gdb testApplication
$ (gdb) target remote Y.Y.Y.Y:10000
```

Above Y.Y.Y.Y is the IP address of the device. Then issue the **set sysroot** and **set substitute-path** commands. Notice that you'll have to substitute \$SDKTARGETSYSROOT text with environment variable content of the same name. GDB cannot read that variable.:

```
$ (gdb) set sysroot $SDKTARGETSYSROOT  
$ (gdb) set substitute-path $SDKTARGETSYSROOT/usr/src
```

You can now debug the application normally, except that instead of issuing the run command one should use continue since the application is already running on the remote side.

Note that it is possible to fully debug the application but not to make system calls made by the application. Such system calls include calls to the soft float library, like divide, add or multiply on floating point variables. It is therefore recommended to use *next* rather than *step* when such system calls are being made.

5 Special considerations

This section is dedicated to device specific requirements that require extra attention and consideration when programming.

5.1 Ethernet, setting a static IP-address

There are several ways of setting the IP address of a device. The default method is DHCP, but a static IP address can also be used. This can be done through the network interfaces configuration file.

5.1.1 File method for IP address configuration

This method requires knowledge about the interfaces file format, but a sample is given below.

```
$ sudo nano /etc/systemd/network/eth0.network
```

Sample of network file setting dynamic IP address:

```
[Match]
Name=eth0

[Network]
DHCP=ipv4
```

Sample of network file setting static IP address:

```
[Match]
Name=eth0
[Network]
Address=192.168.1.20/24
Gateway=192.168.1.1
DNS=192.168.1.1
```

Once the file has been edited, it is recommended to either reboot the device, or to bring the network interfaces down and up again, for the IP address configuration to take effect:

```
$ sudo systemctl restart systemd-networkd
```

5.2 CAN

In Linux, CAN is interfaced using SocketCAN which is a standard used in the Linux kernel.

Usage of SocketCAN requires knowledge of some system specific settings and details described herein. For additional SocketCAN information see the official SocketCAN documentation.

CAN Bittimings are set by default for settings considered robust for most applications. The default bittimings are set as follows for V700 and V1000/V1200:

```
ip link set can0 type can tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 5
ip link set can1 type can tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 5
(v1000/v1200 only)
ip link set can2 type can tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 5
(v1000/v1200 only)
ip link set can3 type can tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 10
(All interfaces)
ip link set dev canX txqueuelen 1000
```

The default settings can be modified in file `/etc/udev/rules.d/16-cc-can-config.rules`

Refer to <https://www.kernel.org/doc/Documentation/networking/can.txt> for details.

5.2.1 Changing interface names with udev

In some cases, it may be beneficial to change interface names to arbitrary values. An example of this are the interfaces on the V1000 and V1200-devices where one of the can channels is interfaced over SPI to enable powerup on CAN.

In practice, legacy software naming prevents renaming the can channels many times.

To mitigate the order, an example script for udev could be:

```
SUBSYSTEM=="net", KERNELS=="5a8d0000.can", ACTION=="add", NAME="canfd0"
SUBSYSTEM=="net", KERNELS=="spi0.0", ACTION=="add", NAME="canfd1"
SUBSYSTEM=="net", KERNELS=="5a8e0000.can", ACTION=="add", NAME="canfd2"
SUBSYSTEM=="net", KERNELS=="5a8f0000.can", ACTION=="add", NAME="canfd3"
```



Warning: do not try and swap the kernel assigned can (`can0-n`) names for different interfaces, as this will end up in a race condition with indeterministic channel names.

5.2.2 Configuration of the device interface

The device node files for the CAN interfaces are `can0 ... canX` for a device with $(X+1)$ CAN interfaces. The interfaces should be shown when listing all network interfaces with the `ifconfig` command.

The CAN bus itself is not initialized during start-up. Before any communications can be executed, the user must set correct bus speed (as an example 250kbps) by first writing the value into the bitrate parameter:

```
$ sudo ip link set can0 type can bitrate 250000
```

To work properly with external CAN devices, the sample point of baud rate timings might need to be configured:

```
$ sudo ifconfig can0 type can sample-point 0.78
```

and then setting interface up with **ifconfig**:

```
$ sudo ifconfig can0 up
```

After this, **ifconfig** should show `can0` as a network interface:

```
$ ifconfig
can0    Link encap:UNSPEC  HWaddr 00-00-00-00-00-00
        UP RUNNING NOARP  MTU:16  Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:10
        RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
        Interrupt:31
```



In CCpilot VS, the device drivers are implemented as loadable kernel modules. The modules are `can_dev.ko` and `flexcan.ko`. Startup scripts handle the loading of the kernel modules upon start-up. The loaded modules can be checked via terminal access using the **lsmod** command:

```
$ lsmod | grep can
flexcan    10092      0
can_dev    8641        1 flexcan,xilinx
```

Since the drivers are compiled as modules, unnecessary protocols may be removed or new modules inserted, according to user needs.

5.2.3 CAN-FD

Some devices support CAN with flexible data rate (FD), see the *Technical Manual* of your device for more details. The nominal and data baud rates are setup using socketCAN in a similar fashion as standard CAN (example 250 kbps nominal, 1 Mbps data):

```
$ sudo ip link set can0 type can bitrate 250000 dbitrate 1000000 fd on
```

In order to work properly with external CAN-FD devices the sample point of baud rate timings might need to be configured:

```
$ sudo ip link set can0 type can sample-point 0.78 dsample-point 0.8
```

5.2.4 Configuring the CAN socket transmission buffer

By default, the CAN driver is configured with a transmission buffer that can hold up to 10 CAN frames. As each frame is sent over the bus, the buffer is cleared. However, it is possible to write frames to the socket faster than the frames are sent, especially if your messages are low-priority frames on a high-traffic bus. If your application needs to send more than 10 CAN frames in bursts, it might be a good idea to increase the size of the transmission buffer:

```
$ ifconfig can0 txqueuelen 100
```



Note: For spi-based can interfaces (Yukon-platforms) it might be necessary to increase the buffer length even more i.e 1000.

5.2.5 Bus recovery options

It is possible to implement automatic bus recovery after bus off has occurred. State changes are automatically detected, and controller is re-initialized after the specified time period.

Automatic bus recovery from bus off state is by default turned off. It can be turned on using the **ip** command, where the wanted restart period in milliseconds is set. For example, a 100 ms restart period for can0 is set from command line like this:

```
$ ifconfig can0 down  
$ ip link set can0 type can restart-ms 100  
$ ifconfig can0 up
```

Same commands apply for all available CAN interfaces by replacing can0 appropriately. The restart period interval is possible to set as needed by the application. Value zero turns automatic bus recovery off.



Warning: Enabling automatic bus recovery may disturb other nodes on the bus, if CAN interface is incorrectly initialized.

5.3 Analog video

For QML applications with analog video, CrossControl recommends using the *QtMultimedia* framework rather than CCAux API since the video performance is higher and development process quicker. For non-QML applications, CCAux API is recommended.

Qt is not part of the default image, but downloadable from the support site as a separate package. Additionally, you may contact support on how to build Qt with the SDK for the required platform.

5.3.1 Analog Video using QtMultimedia

You may download CC Linux examples such as QML application *CCVideo* for displaying analog video using a *QtCamera* instance, from the support site. This application is for test purposes only, CrossControl recommends implementing the video functionality into your application for correct behavior.

QtMultimedia uses the *gstreamer* backend and the application must export the correct video source:

```
setenv("QT_GSTREAMER_CAMERABIN_VIDEOSRC", "imxv4l2videosrc", 1);
```

5.3.2 Analog Video using CCAux API

There are some design constraints on the usage of analog video and CCAux API that it is important to highlight, to create a better understanding of what can be done and what is necessary to do within the applications. Below is a brief description of the video API for developers to consider when building their application.

The most important CCAux Video API functions are as follows:

Initialize (will open file handles, setup basic settings and request frame buffers), select deviceNr=1 for input channel 1:

```
Video_init(VIDEOHANDLE pObj, unsigned char deviceNr)
```

Select the active channel 1-4, corresponds to the physical port number:

```
Video_setActiveChannel(VIDEOHANDLE pObj, VideoChannel channel)
```



Note that CCPilot VS has one analog video channel only. CCPilot VI and CCPilot v700 do not have any analog video channels.

Set the area of the display where the video will be shown:

```
Video_setVideoArea(VIDEOHANDLE pObj, unsigned short topLeftX, unsigned short  
topLeftY, unsigned short bottomRightX, unsigned short bottomRightY)
```

Enable or disable (horizontal) mirroring of the video image:

```
Video_setMirroring(VIDEOHANDLE pObj, CCStatus mode)
```

Show (or hide) video image:

```
Video_showVideo(VIDEOHANDLE pObj, bool show)
```

Further and more detailed API information can be found in the *CCAux API documentation*.

5.4 Graphics, Qt (without Weston)

For the best graphical performance, it is recommended to use the Qt-framework with KMS support.

With our i.MX8-devices using KMS and the proprietary driver, there is a need for special configuration to set the plugin to the correct color mode:

First define the correct mode in kms.json.

```
{
  "device": "/dev/dri/card0",
  "outputs": [
    { "name": "LVDS1", "mode": "800x480", "size": "800x480", "format":
      "abgr8888" } ]
}
```

Then export the necessary configuration parameters:

```
export QT_QPA_EGLFS_KMS_CONFIG="/path-to-directory-with/kms.json"
export QT_QPA_EGLFS_INTEGRATION=eglfs_kms
export QT_QPA_EGLFS_KMS_ATOMIC=1
```

And finally launch your application with:

```
./Application -platform eglfs
```

There is also a native Vivante platform available, (though this usage is, at present, discouraged as the performance is not on par with EGLFS KMS). To use the EGLFS Vivante platform, export the following configuration:

```
export QT_QPA_EGLFS_INTEGRATION=eglfs_viv
export QT_QPA_EGLFS_FORCEVSYNC=0
export QT_QPA_EGLFS_FORCE888=1
./application -platform eglfs
```

5.5 Graphics, Weston

The graphics framework uses the Wayland protocol reference implementation Weston for graphic operations. Wayland is fast and efficient, and is used by most modern advanced Linux systems, giving it vast standard support in the Linux user space.

For example, Qt has a plugin that enables the Qt libraries to be built for Weston. For Qt applications the impact for that means that it simply needs to be started with a specific flag, *platform wayland-egl*, and built with the correct development libraries. In CC Linux, this flag has been set to the default graphics framework, hence there's no need to pass the flag.

Weston includes a windowing system, enabling several applications to overlap while in operation.



The Wayland protocol does not, by default, allow its clients (Qt applications etc.) to have any information about where the client is positioned on the screen. Therefore, some Qt functions, like **QWidget::pos()**, will always give a zero return. This issue has been resolved in CC Linux by adding an extension to wayland, where the window starting coordinates can be given to the application. Please refer to the Software Guide for more information.

5.6 Serial Number Broadcast interface

The device has a Serial Number Broadcast service (SNB). The SNB does not have a programming interface at the device end, but the broadcasted data output can be handled elsewhere; including in another device if required.

The message sent is a multicast UDP datagram to address 224.0.0.27. The message contains a char array with three values separated by tabs; Serial number, Firmware version and device type. The sender's IP address is available in datagram headers.

Example data contents (without quotes):


```
"PR01<tab>10.0.0<tab>0"
```

An example implementation of the data listener is available in the BSP: `apps/snb/snb_reader.c`.

5.7 Polarity of PWM outputs

The PWM outputs can be either high or low sided, which will affect the duty cycle behavior. For high sided outputs, a duty cycle of 90% will result in the output signal being 90% high and 10% low. For low sided outputs, the opposite is true - the output signal will be 10% high and 90% low.

Refer to the *Technical Manual* of your device in order to know if it has PWM outputs and if they are high or low sided.

5.8 Suspend

Upon suspend of the device, some peripherals are turned off. These will need to be restarted by the user application upon resume from suspend. The following peripherals are affected (available peripherals differ for different devices, see the Technical Manual for your device):

- Buzzer/Speaker
- PWM outputs
- Digital outputs

The CCAux API function `PowerMgr_hasResumed()` can be called from within the user application in order to detect a resume from suspend event, see the CCAux API documentation [5].

5.9 General-purpose input/output (GPIO)

In the kernel version 5.15 and later the sysfs interface for GPIOs have been removed (traditionally found in `/sys/class/gpio`). Instead, the `libgpiod` utilities should be used. For instance, to set GPIO bank 3 GPIO 15 as low, issue the following command:

```
gpioset gpiochip3 15=0
```

For more information, read the libgpiod manual pages.

6 Build Examples

Source code for the example application included in the CC Linux reference image (*CCSettingsConsole*), is provided in the BSP. This can be used as a template or starting points for your applications.

Example code on how to call the functions provided by CCAux API are found in the *CCAux API documentation* and in the *examples* directory of the CCAux API source code. Additionally, this chapter provides examples on the build process.

6.1 Building applications with the SDK

This chapter gives two examples on how to build a small example application which uses CCAux API functions to set the color of the status LED. The first example shows how to build a C++ application using a Makefile. The second example shows how to build a Qt application with **qmake** to auto generate the Makefile.

6.1.1 C++ Makefile example

This example shows how to build a C++ application *example* using a Makefile. The source file *example.cpp* is listed below. The file consists of a simple main function calling the status LED functions.

```
/**
 * example.cpp
 */

#include <stdio.h>
#include <FrontLED.h>
#include <CCAuxErrors.h>

using namespace CrossControl;

int main(void)
{
    printf("Setting FrontLED to blue!\r\n");

    FRONTLEDHANDLE pLed = GetFrontLED();

    eErr err;
    err = FrontLED_setStandardColor(pLed, BLUE);

    if(err != ERR_SUCCESS)
        printf("An error occurred!\r\n");

    FrontLED_release(pLed);
    return 0;
}
```

Following is an example *Makefile* that is used to build *example*. The additional *-Wall* flag enables all warnings and is recommended. This file can easily be expanded to build more complex applications.

```
#
# Makefile for example using FrontLed
#
# NOTE: before running make, the SDK environment must be set up by
# sourcing the environment file
#

TARGET=example
LD = $(CXX)
```

```
CC_OBJS = $(TARGET).o
C_OBJS =

OBJS = $(CC_OBJS) $(C_OBJS)

CXXFLAGS+= -DLINUX -Wall
CFLAGS+= -DLINUX -Wall
LDFLAGS+= -lcc-aux2 -lpthread

CCCMD = $(CC) -c $(CFLAGS)
CXXCMD = $(CXX) -c $(CXXFLAGS)

all:      clean $(TARGET)

$(TARGET): $(OBJS)
           $(LD) -o $@ $(OBJS) $(LDFLAGS)

# pattern rules for object files
%.o: %.c
           $(CCCMD) $< -o $@

%.o: %.cpp
           $(CXXCMD) $< -o $@

clean:
           rm -rf *.o *.elf *.gdb *~ $(TARGET)
```

To build example, make sure the environment setup script has been sourced, and then issue the following command:

```
$ make example
```

make will expand the content of the *Makefile* and the previously sourced environment setup script to the following output (in this instance for CCPilot VS):

```
arm-poky-linux-gnueabi-g++ -march=armv7-a -marm -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a9 --sysroot=/opt/vs-sdk/sysroots/cortexa9hf-neon-poky-linux-gnueabi -c -O2 -pipe -g -feliminate-unused-debug-types -DLINUX -Wall example.cpp -o example.o
arm-poky-linux-gnueabi-g++ -march=armv7-a -marm -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a9 --sysroot=/opt/vs-sdk/sysroots/cortexa9hf-neon-poky-linux-gnueabi -o example example.o -wl,-O1 -wl,--hash-style=gnu -wl,--as-needed -lcc-aux2 -lpthread
```

6.2 Qt application development

CrossControl provides a pre-configured virtual machine with open-source Qt IDE and development Qt runtime for all CC Linux-based display computers.

For additional information and downloads, see <https://crosscontrol.com/software-solutions/>

6.3 Building any (platform supported) version of Qt with the SDK

In addition to the CrossControl provided version of Qt. You may build and integrate a preferred version for the platform, as long as Qt provides support for this.

1. Download (or build) the SDK for your platform, and install it in to a preferred location. For this manual, the target platform is V1200/V1000. This guide will assume that the location of the SDK is:

```
/opt/cclinux-v1x00-2.0.5.0
```

2. Download the preferred version of Qt (<https://download.qt.io/>) and extract the source package to a preferred location. In this example, we will use:
3. https://download.qt.io/official_releases/qt/5.12/5.12.12/single/qt-everywhere-src-5.12.12.tar.xz
4. Source the cross-compilation environment from the sdk:

```
tuomas@YoctoDev:~/dev/qt-everywhere-src-5.12.12$ . /opt/cclinux-v1x00-2.0.5.0/environment-setup-aarch64-poky-linux
```

5. In this example, Qt will be compiled as a static library.



Note, that this will enforce certain licensing requirements for linked applications!

```
./configure -static \  
            -opensource \  
            -confirm-license \  
            -prefix /opt/qt-5.12.7 \  
            -hostprefix /opt/cclinux-v1x00-2.0.5.0/sysroots/x86_64-  
cclinuxsdk-linux/usr \  
            -device linux-imx8-g++ \  
            -device-option CROSS_COMPILE=$CROSS_COMPILE \  
            -sysroot /opt/cclinux-v1x00-2.0.5.0/sysroots/aarch64-poky-linux \  
\  
            -nomake tests -nomake examples
```

The significant options are as follows:

```
-static      # Build qt statically, so that only the linking binary needs to  
             be copied to the device. Remove to build normal libraries. You will need to  
             copy them to the target device. Note, that statically compiled application  
             must be open source.  
  
-prefix      # Prefix of the library/tools installation location under the  
TARGET root.  
  
-device      # Qt compilation device configuration under  
qtbase/mkspecs/devices (see Linux repository)  
  
-hostprefix  # Target directory for HOST tools (qmake etc)  
  
-nomake      # Use to skip building of various parts of qt  
  
-skip        # Use to skip build of specific target libraries.
```

6. After configure has finished, run make with the appropriate number (-j) of threads for your build system, and install the libraries in to the SDK for easy integration.

```
make -j8  
make install
```

7. This will integrate the qt-version in to your current installed SDK and enable building and debugging of qt-applications.

```
git clone ssh://git@euupsjb:7999/qt-example.git  
cd ccmultitouchdemo  
. /opt/cclinux-v1x00-2.0.5.0/environment-setup-aarch64-poky-linux  
qmake
```

```
make
```

8. For qt-creator integration, refer to: [qtcreator integration](#)

7 Using an IDE in CC Linux application development

In many cases it is much easier to use a graphical development environment to develop and debug code. This section will provide two different use cases. First, using vscode and then finally using qt-creator to build and deploy graphical applications on to the target.

The guide assumes that the setup of your system is the same as required for building the BSP. That is, a modern Linux distribution with the necessary tools available.

7.1 vscode integration

1. Refer to the online guide for installing vscode for your preferred distribution.

<https://code.visualstudio.com/docs/setup/linux>

2. Download (or build) the SDK for your platform, and install it in to a preferred location. For this example, the target platform is V1200/V1000. This guide will assume that the location of the SDK is:

```
/opt/cclinux-v1x00-2.0.5.0
```

3. Clone/create your application and launch vscode from the directory. For this example, we will use the example application described in 6.1.1.
4. Optional: Install C/C++ extension for intellisense for proper code parsing.
5. Open C/C++ configurations by pressing shift-ctrl-p and selecting C/C++ Edit configurations (JSON) and add the following:

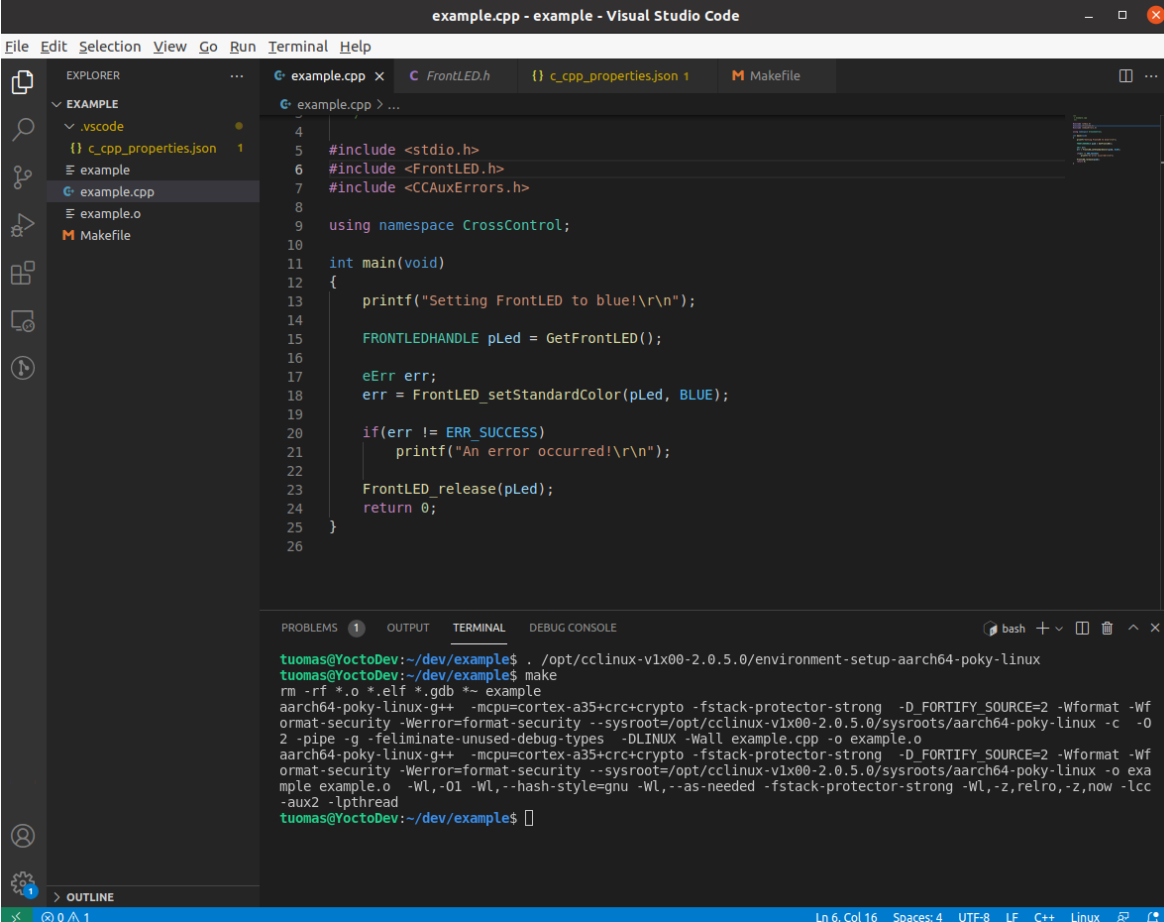
```
{
  "configurations": [
    {
      "name": "Linux",
      "includePath": [
        "${workspaceFolder}/**",
        "/opt/cclinux-v1x00-2.0.5.0/sysroots/aarch64-poky-
linux/usr/include/**",
        "/opt/cclinux-v1x00-2.0.5.0/sysroots/aarch64-poky-
linux/opt/qt5.12.7/include/**"
      ],
      "defines": ["LINUX",
        "PLATFORM_V1x00"],
      "compilerPath": "/opt/cclinux-v1x00-2.0.5.0/sysroots/x86_64-
cclinuxsdk-linux/usr/bin/aarch64-poky-linux/aarch64-poky-linux-g++",
      "cStandard": "c11",
      "cppStandard": "gnu++14",
      "intellisenseMode": "linux-gcc-arm64"
    }
  ],
  "version": 4
}
```

Note, that the includepath and defines will vary based on what is accessible and configured. In the example above the main things to note are:

- a) includepath: Point this to all the include-files you are using. The example above also has the qt-libraries installed.
- b) defines: Use this for intellisense parsing, such as working with the CCApi where you have multiple platforms defined, but only want to see the one you are working on.

- This should now enable you to edit (with autocomplete on headers) and compile the application in VSCode:

NOTE: Remember to source the cross-compilation environment in the terminal before compiling!



```
example.cpp - example - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
EXAMPLE
  .vscode
  {} c_cpp_properties.json 1
  example
  example.cpp
  example.o
  Makefile
example.cpp > ...
4
5 #include <stdio.h>
6 #include <FrontLED.h>
7 #include <CXAuxErrors.h>
8
9 using namespace CrossControl;
10
11 int main(void)
12 {
13     printf("Setting FrontLED to blue!\r\n");
14
15     FRONTLEDHANDLE pLed = GetFrontLED();
16
17     eErr err;
18     err = FrontLED_setStandardColor(pLed, BLUE);
19
20     if(err != ERR_SUCCESS)
21         printf("An error occurred!\r\n");
22
23     FrontLED_release(pLed);
24     return 0;
25 }
26
PROBLEMS 1 OUTPUT TERMINAL DEBUG CONSOLE
bash
tuomas@YoctoDev:~/dev/example$ ./opt/cclinux-v1x00-2.0.5.0/environment-setup-aarch64-poky-linux
tuomas@YoctoDev:~/dev/example$ make
rm -rf *.o *.elf *.gdb *~ example
aarch64-poky-linux-g++ -mcpu=cortex-a35+crccrypto -fstack-protector-strong -D_FORTIFY_SOURCE=2 -Wformat -Wformat-security -Werror=format-security --sysroot=/opt/cclinux-v1x00-2.0.5.0/sysroots/aarch64-poky-linux -c -O2 -pipe -g -feliminate-unused-debug-types -DLINUX -Wall example.cpp -o example.o
aarch64-poky-linux-g++ -mcpu=cortex-a35+crccrypto -fstack-protector-strong -D_FORTIFY_SOURCE=2 -Wformat -Wformat-security -Werror=format-security --sysroot=/opt/cclinux-v1x00-2.0.5.0/sysroots/aarch64-poky-linux -o example example.o -Wl,-O1 -Wl,-hash-style=gnu -Wl,--as-needed -fstack-protector-strong -Wl,-z,relro,-z,now -lcc -aux2 -lpthread
tuomas@YoctoDev:~/dev/example$
```

7.1.1 Setting up gdb

Vscode can also be set up to remotely debug the target application. To do this:

- Add new file under .vscode called launch.json
- Copy the following configuration to launch.json (again noting the paths as appropriate for your specific platform). Also note, the IP-address and port of the target device and replace as necessary.

```
{
    // Use Intellisense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit:
    https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "(gdb) Launch",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}",
            "args": [],
            "stopAtEntry": false,
```

```
    "miDebuggerServerAddress": "10.131.48.53:3000",
    "cwd": "${workspaceFolder}",
    "environment": [],
    "externalConsole": false,
    "MIMode": "gdb",
    "miDebuggerPath": "/opt/cclinux-v1x00-2.0.5.0/sysroots/x86_64-
cclinuxsdk-linux/usr/bin/aarch64-poky-linux/aarch64-poky-linux-gdb",
    "miDebuggerArgs": "",
    "setupCommands": [
      {
        "description": "Enable pretty-printing for gdb",
        "text": "-enable-pretty-printing",
        "ignoreFailures": true
      }
    ]
  }
]
```

3. Deploy the target application to the device and launch it on the target noting the selected port in the debugger configuration:

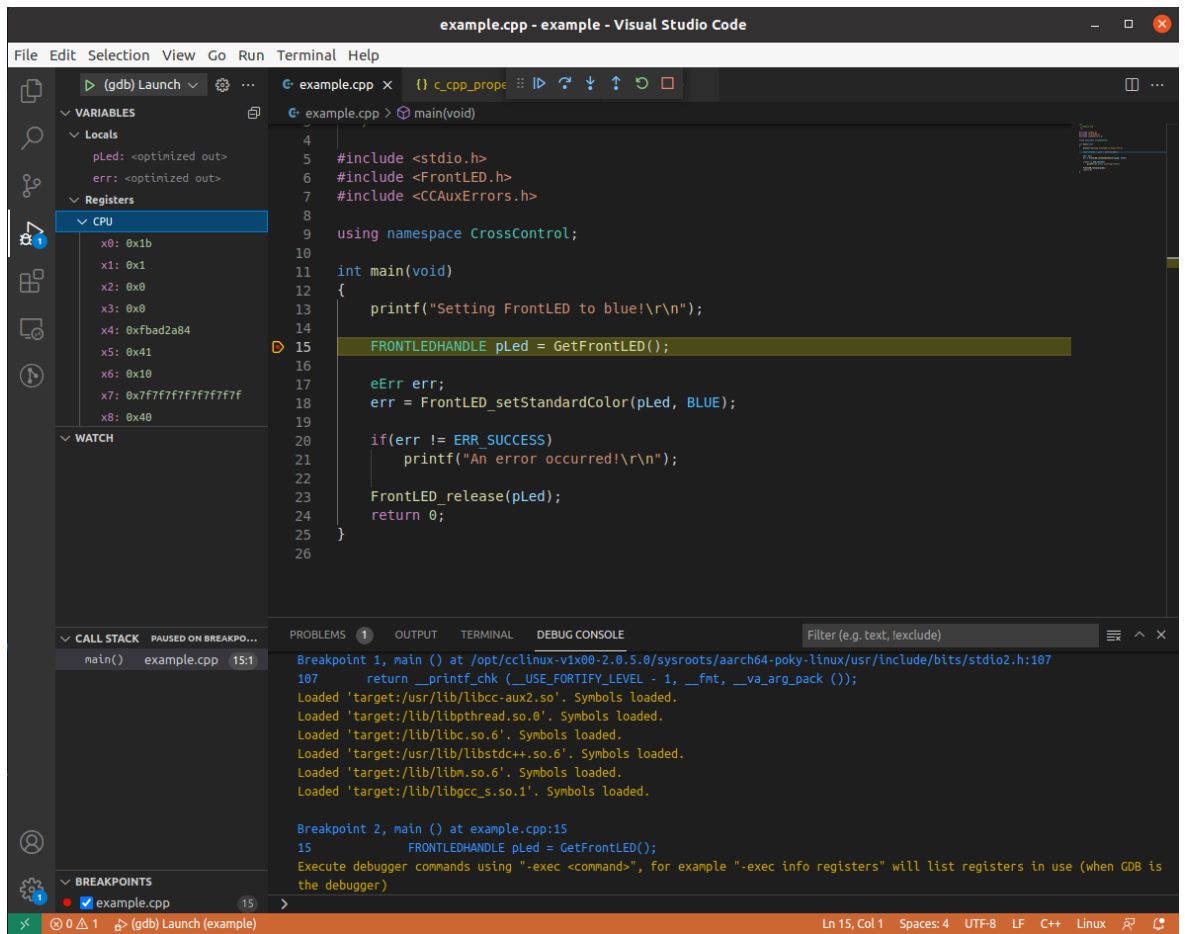
```
tuomas@YoctoDev:~/dev/example$ sftp ccs@10.131.48.53
sftp> put example
Uploading example to /home/ccs/example
example
100% 18KB 2.8MB/s 00:00
sftp> exit

tuomas@YoctoDev:~/dev/example$ ssh ccs@10.131.48.53
ccs@10.131.48.53's password:
Welcome to ccpilot-v1x00-release-2.0.5.0-44-ga9fd1d47-virt (brant)
Built from: ccpilot-v1x00-release-2.0.5.0-44-ga9fd1d47-virt
on: 20211217
Built by: root@yoctobuild.ad.centromotion.com
ccs@v1200:~$ gdbserver localhost:3000 /home/ccs/example
```



This method also works when editing applications withing the Yocto system with devtool. The only difference is that deployment can be automated with devtool-deploy.

4. Finally, set any wanted breakpoints, and launch the debugging in vscode from Run -> Start debugging or simply pressing F5. This will attach gdb to the application and pause on the selected breakpoint.



7.2 qtcreator integration

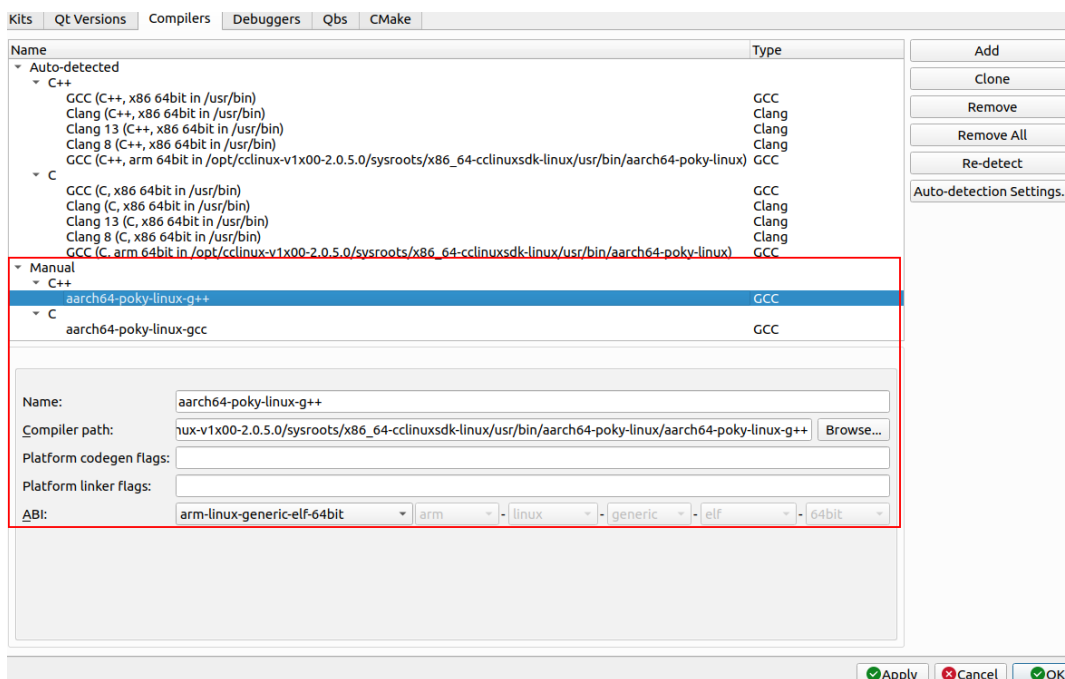
In addition to vscode, qtcreator can be used to create and debug qt-applications on the target.

It is assumed that the SDK has been installed, with a specific version of qt installed in the the SDK as defined in Qt application development

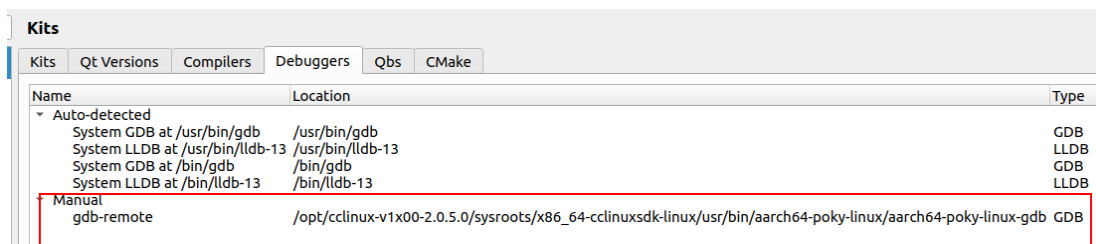
The assumed location for the SDK is:

```
/opt/cclinux-v1x00-2.0.5.0
```

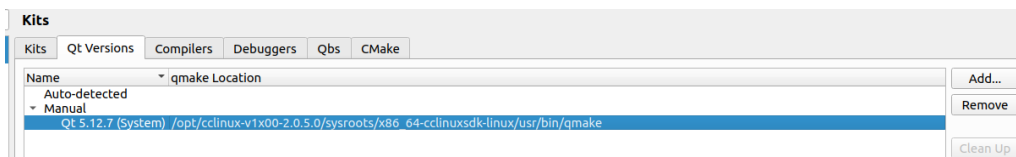
1. Install qt-creator for your preferred distribution.
2. Launch qt-creator and open Options.
3. Qt may, or may not automatically find the cross-compilers, but it is safer to manually add and name both C and C++ compilers:



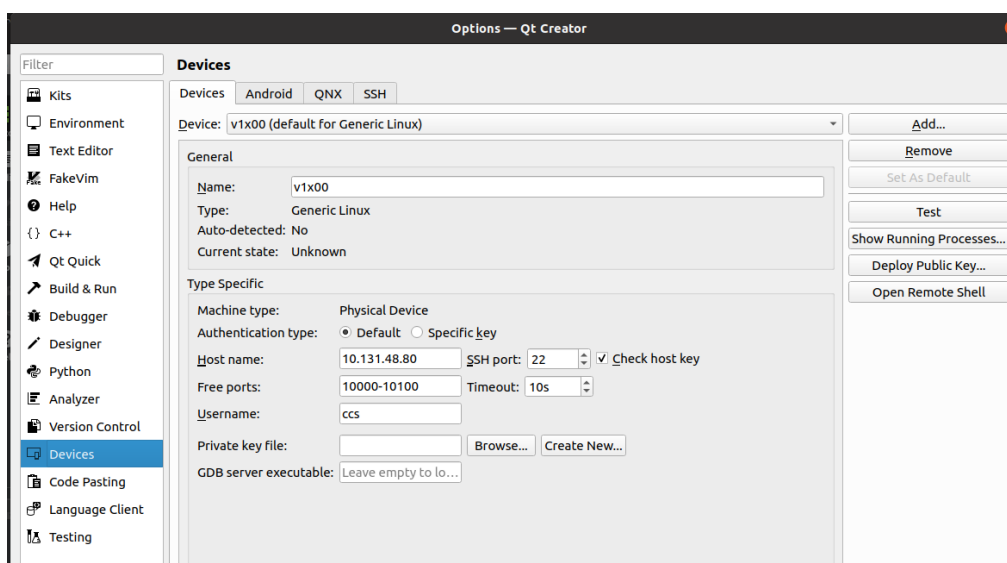
4. Next, add the debugger path and binary:



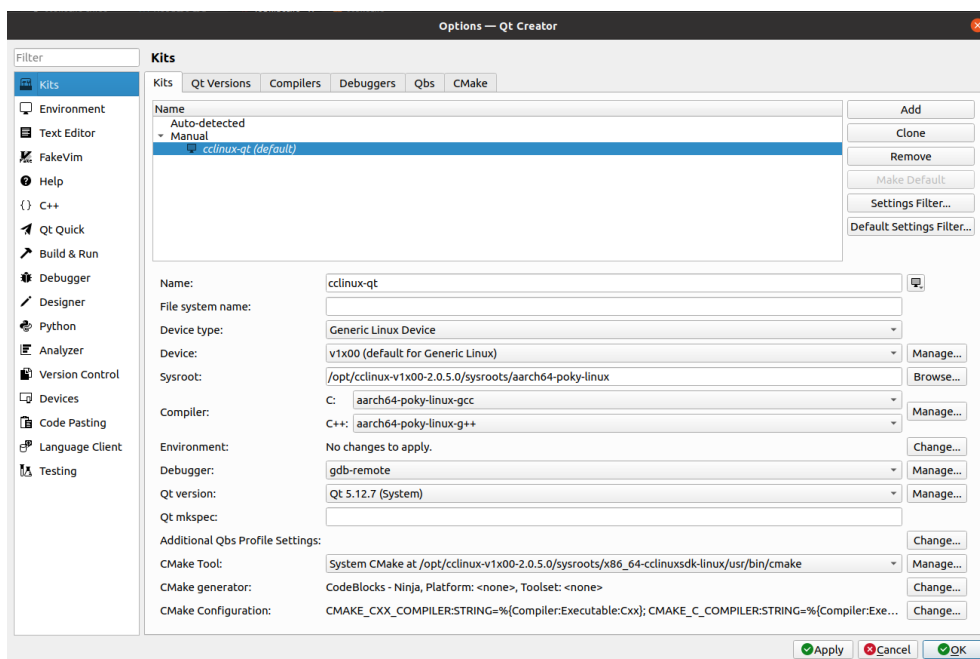
5. Then, configure the Qt version by telling qtcreator the location of qmake:



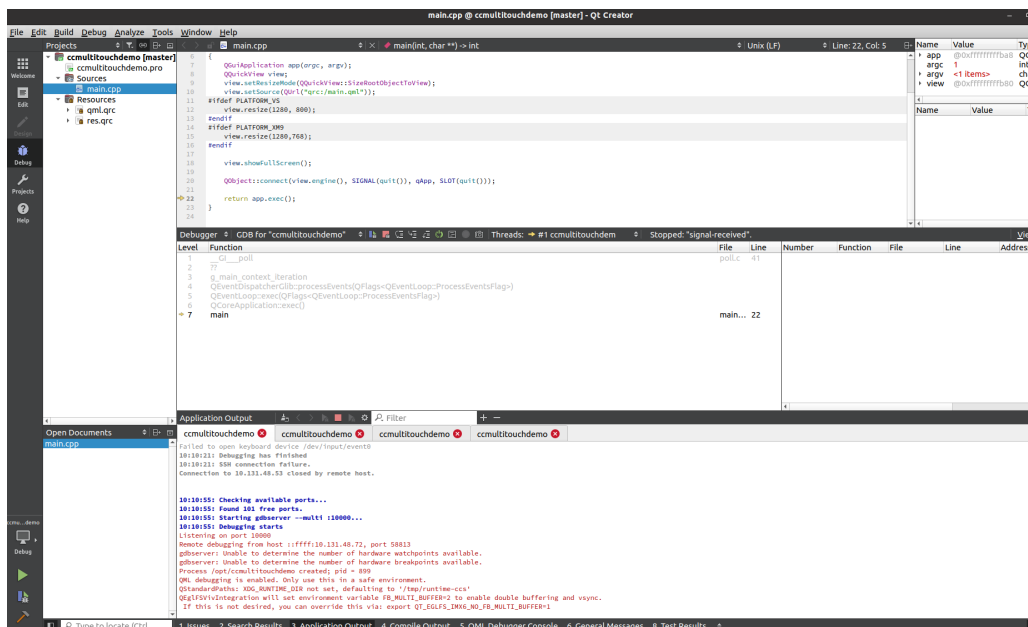
- Before adding the Kits, you can add the target device for easy deployment and integration. Got to devices and select add with the following options noting the hostname and target device name:



- Finally, add the kit with the specific options pointing to the correct cross-compilers and device:



- After this you should be able to automatically build, deploy run and debug the application on the target device directly with qt-creator. Example, debugging a CrossControl demo application on the target device:



8 Working with Systemd

The basics of creating, configuring, and starting systemd services are explained in the software guide. This section provides a more detailed look in to how systemd can be configured in special cases i.e when applications and services need to be tied to specific devices or other services.

8.1 Creating services with dependencies

Sometimes services can include dependencies to other services and a requirement might be that these services need to be executed in a specific order. Additionally, services may require a specific device to be operational before the service can be started.

8.1.1 Example: service that depends on other services

Assume you have a `backend.service` that provides connectivity and a `userapp.service` that provides the user interface. Thus, you want to make sure that the backend is running before the user application is started.

`backend.service`:

```
[Unit]
Description="Our backend that provides data"

[Service]
Type=simple
ExecStart=/usr/bin/launch-backend.sh

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

`user.app.servivce`:

```
[Unit]
Description="User interface to view data"
After=backend.service
Requires=backend.service

[Service]
Type=simple
ExecStart=/usr/bin/launch-gui.sh

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

Finally, enable the service with `systemctl enable user.app.service`. This will automatically load the backend also.

8.1.2 Example: service that depends on a specific device

Assume that your backed provides data from a can-bus, and thus requires the driver to be loaded before starting the service.

First find out the `.device` that you want to bind the service to:

```
root@v700:~# systemctl --type=device --all|grep can0
sys-subsystem-net-devices-can0.device
loaded active plugged /sys/subsystem/net/devices/can0
root@v700:~#
```

Then, add the binding to the service that requires the specific interface.

```
[Unit]
Description="Our backed that provides data from CAN0"
BindsTo=sys-subsystem-net-devices-can0.device
After=sys-subsystem-net-devices-can0.device

[Service]
Type=simple
ExecStart=/usr/bin/launch-backend.sh
```

Now, your service will be bound to wait after the specific device driver has been loaded.

If you cannot find the device, you will need to add a specific udev-rule, creating a .device node with system. See 7.3

8.1.3 Service file locations

For systemd to find service files they must be located in predefined locations. These locations can be checked for each system with the following commands.

For user files:

```
root@v700:~# systemd-analyze --user unit-paths
...
/home/root/.config/systemd/user.control
/home/root/.config/systemd/user
/etc/systemd/user
...
```

For system related files:

```
root@v700:~# systemd-analyze --system unit-paths
...
/etc/systemd/system
/usr/local/lib/systemd/system
/lib/systemd/system
/usr/lib/systemd/system
...
```

8.2 Viewing dependencies with systemctl list-dependencies

Finding out dependencies might become cumbersome at times and systemd provides tools to help out in this task. It is possible to view dependencies for each service, but also in the reverse.

As an example the System Supervisor runs a specific service on the device on each boot. To see the dependencies for this, run:

```
root@v700:~# systemctl list-dependencies --all ss.service
ss.service
• └─system.slice
• ┌─sysinit.target
•   └─dev-hugepages.mount
•   └─dev-mqueue.mount
•   └─kmod-static-nodes.service
•   └─ldconfig.service
•   └─psplash-start.service
•   └─rngd.service
•   └─sys-fs-fuse-connections.mount
•   └─sys-kernel-config.mount
•   └─sys-kernel-debug.mount
•   └─systemd-ask-password-console.path
•   └─systemd-journal-catalog-update.service
•   └─systemd-journal-flush.service
•   └─systemd-journald.service
```

```

•   |--systemd-machine-id-commit.service
•   |--systemd-modules-load.service
•   |--systemd-sysctl.service
•   |--systemd-sysusers.service
•   |--systemd-tmpfiles-setup-dev.service
•   |--systemd-tmpfiles-setup.service
•   |--systemd-udev-trigger.service
•   |--systemd-udevd.service
•   |--systemd-update-done.service
•   |--systemd-update-utmp.service
•   |--local-fs.target
•   | |--systemd-remount-fs.service
•   | |--tmp.mount
•   | |--var-volatile-cache.service
•   | |--var-volatile-lib.service
•   | |--var-volatile-spool.service
•   | |--var-volatile-srv.service
•   |   └--var-volatile.mount
•   └--swap.target

```

This will provide you with a dependency tree of the service.

To find out what services depend on a specific one, use the `-reverse` flag:

```

root@v700:~# systemctl list-dependencies ss.service --all --reverse
ss.service
•   |--ccauxd.service
•   |   └--multi-user.target
•   |       └--graphical.target
•   └--sys-module-ss.device

```

The reverse depends will help you to see, where a specific service is started if it is not listed in any of the targets (but is launched as a dependency from another `.service`).

8.3 Adding systemd .devices

By default, systemd maps most of the devices (all tagged with “systemd”). However if the device is not visible you may create a mapping to use. For this example, let’s use the light sensor that is located on the i2c-bus, but also as a symlink on `/dev/lightsensor`.

First use `udevadm` to check the details of the device:

```

root@v700:~# udevadm info
/sys/devices/platform/bus@5a000000/5a810000.i2c/i2c-16/16-0044
P: /devices/platform/bus@5a000000/5a810000.i2c/i2c-16/16-0044
L: 0
E: DEVPATH=/devices/platform/bus@5a000000/5a810000.i2c/i2c-16/16-0044
E: DRIVER=isl29018
E: OF_NAME=isl29035
E: OF_FULLNAME=/bus@5a000000/i2c@5a810000/isl29035@44
E: OF_COMPATIBLE_0=isil,isl29035
E: OF_COMPATIBLE_N=1
E: MODALIAS=of:Nisl29035T(null)Cisil,isl29035
E: SUBSYSTEM=i2c
E: USEC_INITIALIZED=4914427

```

Then, create a rule in `udev`, to add the tag “systemd” to the device, and also map the `/dev/lightsensor` link to make it more useable.

Under `/etc/udev/rules.d/lightsensor.rules`, add:

```
SUBSYSTEM=="i2c", KERNELS=="16-0044", TAG="systemd",  
ENV{SYSTEMD_ALIAS}="/dev/lightsensor"
```

Save the file, and reload udev:

```
udevadm control --reload-rules && udevadm trigger
```

Device will now be visible as a systemd.device:

```
root@v700:~# systemctl --type=device --all |grep -e 044 -e lightsensor  
dev-lightsensor.device  
loaded active plugged /dev/lightsensor  
sys-devices-platform-bus\x405a000000-5a810000.i2c-i2c\x2d16-  
16\x2d0044.device loaded active plugged  
/sys/devices/platform/bus@5a000000/5a810000.i2c/i2c-16/16-0044
```

Additionally, you will be able to see the added tags with udevadm:

```
root@v700:~# udevadm info  
/sys/devices/platform/bus@5a000000/5a810000.i2c/i2c-16/16-0044  
P: /devices/platform/bus@5a000000/5a810000.i2c/i2c-16/16-0044  
L: 0  
E: DEVPATH=/devices/platform/bus@5a000000/5a810000.i2c/i2c-16/16-0044  
E: DRIVER=isl29018  
E: OF_NAME=isl29035  
E: OF_FULLNAME=/bus@5a000000/i2c@5a810000/isl29035@44  
E: OF_COMPATIBLE_0=isl,isl29035  
E: OF_COMPATIBLE_N=1  
E: MODALIAS=of:Nisl29035T(null)Cisl,isl29035  
E: SUBSYSTEM=i2c  
E: USEC_INITIALIZED=4914427  
E: SYSTEMD_ALIAS=/dev/lightsensor  
E: TAGS=:systemd:
```

8.3.1 Launching services from udev (not recommended)

In some cases it might make sense to launch services directly from udev rules. In this case, just add the necessary service to the rule.

```
ENV{SYSTEMD_WANTS}+="lightsensor.service"
```

This will run the specific service when the device is connected. The pitfall is, that if you have the service enabled elsewhere, it will be hard to know where it is launched from. Thus, the preferred way is to run all services from targets.

Technical support

Additional sources of information are available on the CrossControl support site:

<https://crosscontrol.com/support/>

You will need to register to the site in order to be able to access all information available.

Contact your reseller or supplier for help with possible problems with your device. To get the best help, you should have access to your device and be prepared with the following information before you contact support.

- The part number and serial number of the device, which you can find on the brand label.
- Date of purchase, which can be found on the invoice.
- The conditions and circumstances under which the problem arises.
- Status indicator patterns (i.e., LED blink pattern).
- Prepare a system report on the device, using CCSettingsConsole (if possible).
- Detailed description of all external equipment connected to the unit (when relevant to the problem).

Trademarks and terms of use

© 2022 CrossControl

All trademarks sighted in this document are the property of their respective owners.

CCpilot is a trademark which is the property of CrossControl.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

CC Linux is an official Linux distribution pursuant to the terms of the Linux Sublicense Agreement

Microsoft® and Windows® are registered trademarks which belong to Microsoft Corporation in the USA and/or other countries.

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Qt is a registered trademark of The Qt Company Ltd. and its subsidiaries.

CrossControl is not responsible for editing errors, technical errors or for material which has been omitted in this document. CrossControl is not responsible for unintentional damage or for damage which occurs as a result of supplying, handling or using of this material including the devices and software referred to herein. The information in this handbook is supplied without any guarantees and can change without prior notification.

For CrossControl licensed software, CrossControl grants you a license, to under CrossControl's intellectual property rights, to use, reproduce, distribute, market, and sell the software, only as a part of or integrated within, the devices for which this documentation concerns. Any other usage, such as, but not limited to, reproduction, distribution, marketing, sales and reverse engineering of this documentation, licensed software source code or any other affiliated material may not be performed without the written consent of CrossControl.

CrossControl respects the intellectual property of others, and we ask our users to do the same. Where software based on CrossControl software or products is distributed, the software may only be distributed in accordance with the terms and conditions provided by the reproduced licensors.

For end-user license agreements (EULAs), copyright notices, conditions, and disclaimers, regarding certain third-party components used in the device, refer to the copyright notices documentation.