Exor PLC+HMI Development Kit
User Manual

Exor PLC+HMI Development Kit User Manual

History				
Rev	Date	Description		By
1.3	22/Jul/2015	- Added section 1.3, briefly explaining how to start building from the Docker container. - Section 3.3: solved an inconsistence between paths in build instructions.		N.O.
1.4	4/Apr/2016	- Updated references to new SD card image and JMobile 2.1 - Chapter 4: add note on supported SD-cards. A capacity of at least 4GB is required. - Chapter 6: add note clarifying the purpose of the serial port, serial protocols are not supported		N.O.
1.5	24/May/2016	- Add references to usom01 and usom03 based development kits. - Chapter 1: add VirtualBox virtual machine references. - Chapter 3: update to use Yocto 2.1 recipes.		N.O.
1.6	5/Sep/2016	- Chapter 1: add instructions for installing VirtualBox VM from OVA. Installing Vagrant is no more required. - Chapter 5: add instructions for BSP deploy on eMMC - Chapter 6: add some more informations, section 6.3 has now more detailed instructions. - Chapter 7: add instructions for expansion plugins		N.O.
1.7	30/Jun/2017	- Chapter 4: update instruction for partitioning the SD-card		N.O.
1.8	27/Sep/2017	- Add references to nsom01 based development kit. - Chapter 3: update to use Yocto 2.3 recipes		N.O.
1.9	18/Jun/2018	- Update for release 4.x, Yocto 2.4 - Remove Docker instructions, now unsupported		N.O.
1.10	5/Aug/2018	- Chapter 5: update instructions		N.O.

Reference			
Cross Reference	Filename	Description	
[1]			

Document contains		
		Author(s)
	Functional Specs	
	Usability Specs	
	Techical Specs	
	Test Procedures	
	Technical Documentation	
v	User Documentation	

TABLE OF CONTENTS

1. Title Getting started

To work with the development kits a Linux operating system with a properly configured build environment is required. The simplest way to get started, especially for Windows users, may be using one of our development virtual machines. We provide a VirtualBox VM and a Docker container, both are preconfigured with:

- Yocto workspace for building the BSP
- Preinstalled SDKs to start building your own application for the development kit
- QtCreator IDE with preconfigured target toolchains (Qt 5.9)

If you are already working on a Linux machine or you already have a Linux VM you may consider configuring yourself the build environment instead. In this case skip this chapter and go to chapter 3 if you are interested in building the BSP or chapter 6 if you are interested in building your own applications for the target.

1.1 Running the VirtualBox VM

You can download the Exor's VirtualBox development VM from here:

```
http://download.exorembedded.net:8080/Public/VirtualBoxVMs
```

Instructions found on this document are compatible with versions 4.x of the VM. If you are about to use a greater version please consider looking for an updated version of this manual.

The virtual machine comes in the OVA (Open Virtualization Archive) format. To import it on VirtualBox got to "File" -> "Import Appliance…", select the downloaded .ova file and then click "Import". At this point VirtualBox will give you the opportunity to customize the VM, double-click on entries to edit them.

You will notice there are two network adapters, one is set to work in NAT mode while the second one works in bridged mode, the virtual machine will always use the bridged interface if possible and fall back to the other only if necessary. Adjust both adapters to work with the real network interface you use to have access to internet. Note that if the bridged adapter is not correctly configured you won't be able to resolve the Kit hostname, its IP address has to be used in this case.

Appliance settings

These are the virtual machines contained in the appliance and the suggested settings of the imported VirtualBox machines. You can change many of the properties shown by double-clicking on the items and disable others using the check boxes below.

Description	Configuration
Virtual System 1	
Name	ExorDev-VM
Guest OS Type	Ubuntu (64-bit)
CPU	2
RAM	2048 MB
USB Controller	☑
Network Adapter	☑ Intel PRO/1000 MT Desktop (82540EM)
Network Adapter	☑ Intel PRO/1000 MT Desktop (82540EM)
Storage Controller (SATA)	AHCI
Virtual Disk Image	C:\VirtualBox VMs\ExorDev-VM\ExorDev-VM-…

☐ Reinitialize the MAC address of all network cards

Restore Defaults   Import   Cancel

The default amount of RAM is set to 2GB but if you plan working with Yocto we recommend to increase it to at least 4GB (suggested 6GB), adjusting the number of CPU cores is also a good idea. When you're done click on "Import". Once finished importing you will be able to change VM settings
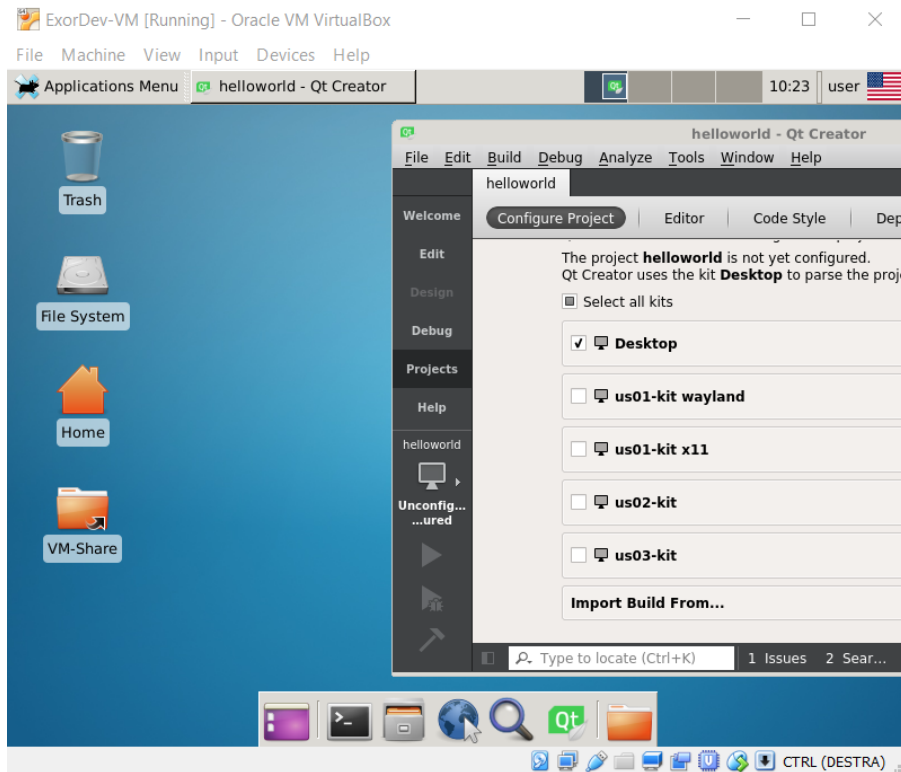
again.

The Linux operating system used is based on Ubuntu 16.04, the default user is:

- username: **user**
- password: **password**

To run a command with root privileges you can use `sudo`, entering the password is not required.

1.1.1 Setup a guest-host shared folder

We recommend configuring a shared folder between host and guest, it's the easiest way to move files from and to the VM. From VirtualBox right-click on Exor's VM and select "Settings…". Now go to "Shared Folders" and click on the add button to the right. Configure as follow:

- **Folder Path**: choose the host folder to share with the virtual machine
- **Folder Name**: must be `share`.
- **Read-only**: leave unchecked.
- **Auto mount**: leave unchecked.
- **Make Permanent**: set checked.

The chosen folder will be available inside the virtual machine from `/home/user/VM-Share`, a link to this location can be also found on the VM's desktop. If the VM was already running a restart will be
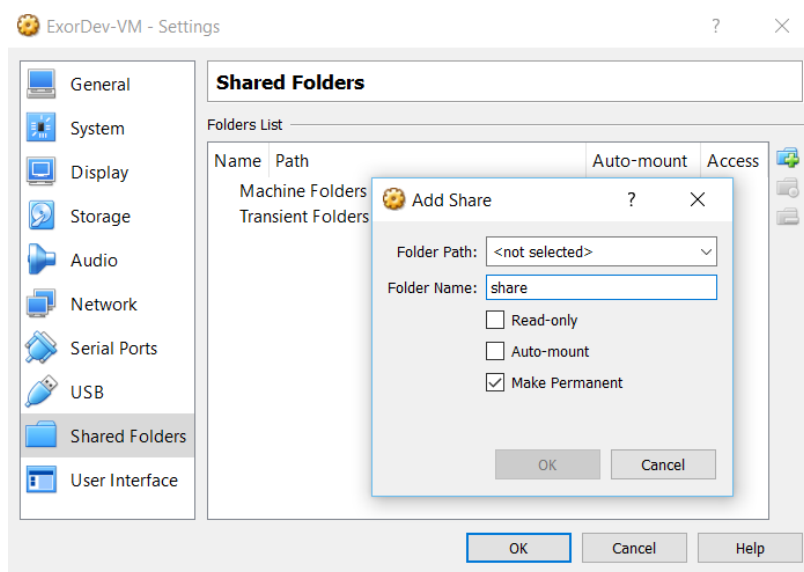
required.

1.1.2 Configuring the SDK

To reduce the initial weight of the VM the SDK is not shipped with it. Scripts named "Install […] SDK.sh" can be found on the desktop, by just executing these with a double-click it's possible to automatically download and install the required SDK files for each device.

During installation QtCreator will be reconfigured, if found running it will be automatically closed during the process.

1.1.3 Using QtCreator

The QtCreator IDE is already installed and configured to deploy and debug applications for each development kit. When creating a new project make sure to select the kit configuration for your device, if not available make sure that the corresponding SDK has been installed using one of the scripts that can be found on the desktop. There's also a "Desktop" kit configuration which can be used to build your application and run it on the virtual machine instead of deploying it, useful for fast testing and heavy profiling.

You will find a helloworld sample project in `/home/user/helloworld`, open it with QtCreator, compile it for your platform and press `Ctrl+R`, a window will pop up in the development kit.

You can find more details about configuring QtCreator in section 6.3, in particular how to change the hostname or IP address of the target device.

1.1.4 Compiling the BSP with Yocto

Inside `/home/user/exor-yocto-4.0` you will find the preconfigured Yocto workspace for building the BSP for your development kit. As we do not update our development virtual machines as often we do with our Yocto recipes you may want to update the meta-exor layer before starting the build:

```
$ cd /home/user/exor-yocto-4.0/git/meta-exor
$ git checkout rocko
$ git pull
```

Go to chapter 3 to go ahead compiling the BSP.

2   The Sato desktop

The development kit will boot with the   default Yocto SATO interface. The machine is configured to run as **root** user and an   empty password

2.1   Network configuration

By default network configuration is done using a DHCP service. To change this and set a static IP click on the Ethernet icon on the right top of the screen and choose "Preferences".
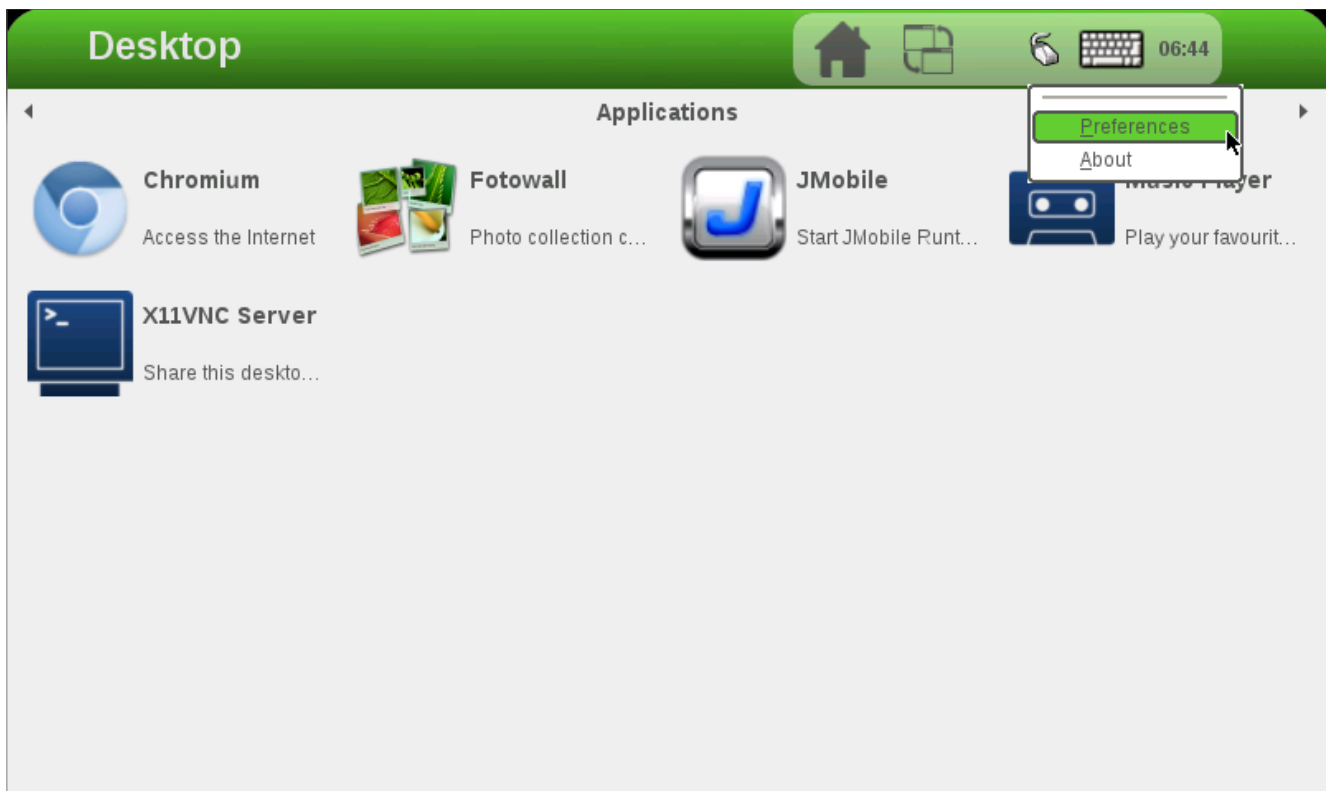
Here, in "Connection Preferences", choose "Ethernet" from thee Services list and select "MANUAL" under "Configuration". Now you can fill in your network configuration. To do this you can toggle the on-screen keyboard or just plug-in a real USB keyboard. Rememer to click on "Apply" when you are done.
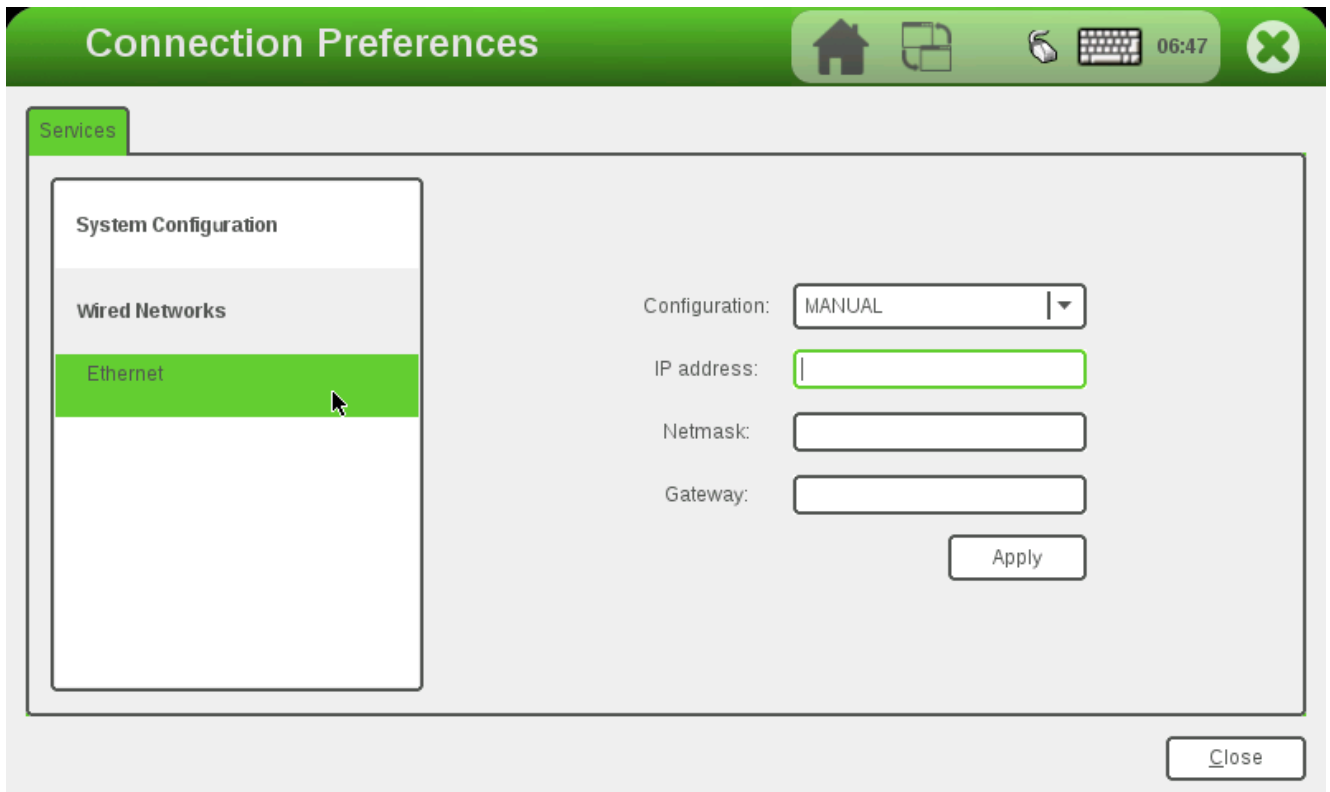
2.2   Start JMobile from Sato

By default, among other applications, a portable version of JMobile Runtime is installed. To launch the HMI just click on the JMobile icon you will find in the "Applications" menu in Sato. A demo project is already loaded for evaluation purpose. As the Runtime is meant to run by his own on the system the Sato user interface will be terminated.

To close JMobile and return to Sato you can both reboot the board or kill the HMI by issuing the following command from an ssh session:

```
$ killall xinit
```

3 Compiling Yocto BSP from scratch.

3.1 Setup the build environment

If you are using Exor's VirtualBox VM you can skip the first two steps: you will find the `exor-yocto-4.0` folder already in the user's home (`/home/user/exor-yocto-4.0`).

1. Create a workspace directory structure:

```
$ mkdir -p exor-yocto-4.0
$ cd exor-yocto-4.0/
```

2. Get the source code from github repositories:

```
$ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > repo
$ chmod a+x repo
$ ./repo init -u https://github.com/ExorEmbedded/exor-bsp-platform -b rocko
$ ./repo sync
```

3. Setup the Yocto environment. From the `exor-yocto-4.0` folder execute:

```
$ source git/yocto-poky/oe-init-build-env build
```

You should now find yourself in a newly created "build" directory located in `exor-yocto-4.0/build`. The source command above

4. Configure Yocto by copying the provided sample configuration files. From the the `build` directory:

```
$ cp ../git/meta-exor/conf/bblayers.conf.sample conf/bblayers.conf
$ cp ../git/meta-exor/conf/local.conf.sample conf/local.conf
```

5. Now edit your `conf/local.conf` and set the `MACHINE` variable to `us01-kit`, `us02-kit` (AlteraKit), `us03-kit` or `ns01-kit`. For example:

```
MACHINE = "us02-kit"
```

You are now ready to build the BSP.

3.2 Optional customizations

Here are some customizations you may be interested in:

- You can force Yocto to build a 32-bit SDK uncommenting the following line in the `build/conf/local.conf` file:

```
#SDKMACHINE ?= "i686"
```

- Uncomment following lines in the `build/conf/local.conf` file to be able to set the number of threads and CPU cores you want to use for the build process:

```
#BB_NUMBER_THREADS ?= "4"
#PARALLEL_MAKE ?= "-j 4"
```

3.3   Compiling Yocto BSP

Make sure to run following commands from your "build" folder:

1. Compile the bootloader:

```
$ bitbake bootloader
```

2. The xloader (for the us01-kit only):

```
$ bitbake xloader
```

3. The Linux kernel:

```
$ bitbake virtual/kernel
```

4. And finally the rootfs:

```
$ bitbake core-image-exor-x11
```

This will build the classic x11 sato image, the one that can be found in the SD-card included with the development kit.

At the end of these operations you will find build output files in `build/tmp/deploy/images/usom0X`:

```
us0X-kit-uboot.tar.gz                      Contains the U-Boot raw image
us0X-kit-xloader.tar.gz                    Contains the xloader raw image (usom01 only)
us0X-kit-kernel.tar.gz                     Contains the kernel zImage and the dtb
core-image-exor-[…]-us0X-kit.tar.gz        Contains the rootfs

us0X-kit-xloader.tar.gz                    Contains the xloader raw image (usom01 only)
us0X-kit-kernel.tar.gz                     Contains the kernel zImage and the dtb
core-image-exor-[…]-us0X-kit.tar.gz        Contains the rootfs
```

3.3.1 Creating the SDK (optional)

Start the SDK build for the x11 image:

```
$ bitbake –c populate_sdk core-image-exor-x11
```

The SDK installer can be found in `build/tmp/deploy/sdk/exor-evm-qt5-sdk.sh`.

4 BSP deploy on SD-card

This section describes how to prepare a bootable SD-card for the evaluation kit, for this remember that only SD-cards with at least 4GB of space are supported.

Also note that following operations can be dangerous, harm your system or cause loss of data. Do not blindly execute these operations if you don't know what they actually do.

For Linux users we will assume below the SD-card device is named */dev/sdb* and its partitions */dev/sdbX*, change these to the actual names.

4.1 Using a ready image

We provide a fully working 4GB image containing the x11-sato environment to let you start using the kit in no time. Note that by using this option, even with a more capable SD, only ~4GB of space will be available to the system.
Download the latest disk image for your evaluation kit:

```
US01kit images:   http://download.exorembedded.net:8080/Public/usom01/sdcard-images/
US02Kit images:   http://download.exorembedded.net:8080/Public/usom02/sdcard-images/
US03Kit images:   http://download.exorembedded.net:8080/Public/usom03/sdcard-images/
US01Kit images:   http://download.exorembedded.net:8080/Public/nsom01/sdcard-images/
```

4.1.1 Under Linux

From a Linux shell:

```
# unzip SDcard-image-4gb.zip
# dd if=SDcard-image-4gb.img of=/dev/sdb bs=64k
# sync
```

Your SD-card is now ready to be used on the development kit.

4.1.2 Under Windows

Download Win32DiskImager from `http://sourceforge.net/projects/win32diskimager/`. From the user interface of Win32DiskImager select the extracted .img image file and the SD-card drive and press "Write".
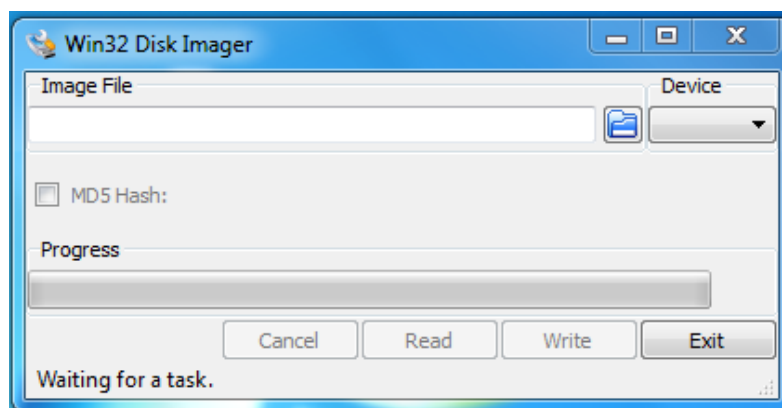
4.2 Using the SD-card installer (Linux users only)

If you want to take advantage of all your SD-card space or you have built by your own some components you want to deploy, it's also possible to use a SD-card installer script:

```
$ wget http://download.exorembedded.net:8080/Public/utils/mkSDTool/mkSDTool-v4.0.sh -O mkSDTool.sh
$ sudo chmod +x mkSDTool.sh
$ sudo ./mkSDTool.sh --machine [us01-x11|us01-wayland|us02|us03|ns01] --device /dev/sdb
```

By default the script will deploy on your SD-card all the necessary files by downloading the needed components from following locations:

```
US01kit:           http://download.exorembedded.net:8080/Public/usom01/
US02kit:           http://download.exorembedded.net:8080/Public/usom02/
US03kit:           http://download.exorembedded.net:8080/Public/usom03/
NS01kit:           http://download.exorembedded.net:8080/Public/nsom01/
```

If you want to provide yourself one or more of these components you can take advantage of following options supported by the mkSDTool script:
- --rootfs <path/to/my/rootfs.tar.gz>
- --kernel <path/to/my/kernel.tar.gz>
- --uboot <path/to/my/u-boot.tar.gz>
- --xloader <path/to/my/xloader.tar.gz> (only for us01)

4.3 Manually

1. Create the SD-card partition layout :

```
# umount /dev/sdb*
# SIZE=`fdisk -l /dev/sdb | grep -m1 Disk | awk '{print $5}'`
# CYLINDERS=$(( $(( $SIZE )) / 255 / 63 /512 ))
# sfdisk --force -D -H 255 -S 63 -C $CYLINDERS /dev/sdb << EOF
  1,5
  6,$(( $CYLINDERS - 10 ))
  $(( $CYLINDERS - 4 )),,a2
  EOF
# mkfs.vfat -n BOOT /dev/sdb1
# mkfs.ext4 -L ROOT /dev/sdb2
```

2. Mount partitions. Execute following operations:

```
// Mount partitions if not already mounted

# mkdir /media/BOOT
# mount /dev/sdb1 /media/BOOT
# mkdir /media/ROOT
# mount /dev/sdb2 /media/ROOT
```

Now the actual deploy phase depends on the specific board. Make sure to follow the appropriate steps:

```
// Deploy files to SD-card for us01-Kit
# mkdir /media/BOOT/boot
# tar xzvf us01-kit-kernel[…].tar.gz --no-same-owner -C /media/BOOT/boot
# tar xzvf us01-kit-bootloader.tar.gz --no-same-owner -C /media/BOOT
# tar xzvf us01-kit-xloader.tar.gz --no-same-owner -C /media/BOOT
# tar xzvf core-image-exor-[…].tar.gz -C /media/ROOT
# sync

// Deploy files to SD-card for us02-kit
# tar xzvf us02-kit-kernel.tar.gz --no-same-owner -C /media/BOOT
# tar xzvf core-image-exor-[…].tar.gz -C /media/ROOT
# tar xzvf us02-kit-uboot.tar.gz
# dd if=u-boot.img of=/dev/sdb3 bs=64k seek=4
# sync
```

```
// Deploy files to SD-card for us03-kit
# mkdir /media/BOOT/boot
# tar xzvf us03-kit-kernel.tar.gz --no-same-owner -C /media/BOOT/boot
# tar xzvf core-image-exor-[…].tar.gz -C /media/ROOT
# tar xzvf us03-kit-uboot.tar.gz
# dd if=u-boot.imx of=/dev/sdb bs=1k seek=1
# sync

// Deploy files to SD-card for ns01-kit
# mkdir /media/BOOT/boot
# tar xzvf ns01-kit-kernel-[…].tar.gz --no-same-owner -C /media/BOOT/boot
# tar xzvf core-image-exor-[…].tar.gz -C /media/ROOT
# tar xzvf ns01-kit-uboot[…].tar.gz
# dd if=u-boot.imx of=/dev/sdb bs=1k seek=1
# sync
```

5 BSP deploy on eMMC

This section describes how to deploy the BSP on eMMC and boot from it. All the kits have the possibility to boot without an SD-Card except for the us03-kit.

On the iMX6Q the location where the bootloader needs to loaded is defined by OTP fuses that on the us03-kit are already set to use the SD-Card. Once the bootloader is loaded into ram the roots used will still be the one on the eMMC and the SD-card could be removed. For more information on OTP fuse programming please refer to NXP processor reference manual (chapter 5 *Fusemap* and chapter 46 *On-Chip OTP Controller*):

https://www.nxp.com/docs/en/reference-manual/IMX6DQRM.pdf

To deploy the BSP to the internal eMMC it is required to define the partition layout and then modify the bootloader environment in order to inform the u-boot on where to look for all the necessary files. Here, for demonstration purposes, we will use the simplest layout, a single ext4 partition. Following instructions needs to be executed on the development kit via ssh, it requires you have a working SD-card and these files available on it:

- The bootloader image, `uboot.img`.
- The rootfs, `core-image-exor.tar.gz`.
- Kernel and dtb or a `kernel.tar.gz` containing both.

Here are the steps to follow:

1) Reformat the eMMC device to have a single partition and create the ext4 filesystem. The eMMC device is defined as `/dev/mmcblk1` on all the development kits except for the us02-kit where it's `/dev/mmcblk0`, for this reason the operation is slightly different for the latter.

```
// Format eMMC and mount rootfs partition for us01-kit, us03-kit and ns01-kit
# umount /dev/mmcblk1p*
# SIZE=`fdisk -l /dev/mmcblk1 | grep -m1 Disk | awk '{print $5}'`
# CYLINDERS=$(( $(( $SIZE )) / 255 / 63 /512 ))
# echo -e "o\nn\np\n1\n2\n\nw" | fdisk -H 255 -S 63 -C $CYLINDERS /dev/mmcblk1
# mkfs.ext4 /dev/mmcblk1p1
# mkdir emmc
# mount /dev/mmcblk1p1 emmc

// Format eMMC and mount rootfs partition for us02-kit
# umount /dev/mmcblk0p*
# SIZE=`fdisk -l /dev/mmcblk0 | grep -m1 Disk | awk '{print $5}'`
# CYLINDERS=$(( $(( $SIZE )) / 255 / 63 /512 ))
# echo -e "o\nn\np\n1\n2\n\nw" | fdisk -H 255 -S 63 -C $CYLINDERS /dev/mmcblk0
# mkfs.ext4 /dev/mmcblk0p1
# mkdir emmc
# mount /dev/mmcblk0p1 emmc
```

2) Deploy rootfs and kernel. Make sure at the end `emmc/boot` contains both a zImage and a dtb.

```
# tar xzvf core-image-exor.tar.gz -C emmc
# tar xzvf kernel.tar.gz -C emmc/boot       // Or just copy zImage and dtb to
emmc/boot # sync
```

3) Deploy the bootloader. Again, this is platform dependent.

```
// Deploy bootloader on eMMC for us01-kit
# echo 0 > /sys/block/mmcblk1boot0/force_ro
# dd if=u-boot.img of=/dev/mmcblk1boot0 bs=512 seek=0
```

```
// Deploy bootloader on eMMC for us02-kit
# echo 0 > /sys/block/mmcblk0boot0/force_ro
# dd if=u-boot.img of=/dev/mmcblk0boot0 bs=512 seek=0

// Deploy bootloader on eMMC for us03-kit and ns01-kit
# dd if=u-boot.imx of=/dev/mmcblk1 bs=512 seek=2
```

Now if you remove the SD-card the bootloader written to the eMMC will be executed (except for the us03-kit, see the note at the beginning of this chapter) but the system won't boot because the u-boot will still look for files inside the SD-card.
To make it work the bootloader environment must be changed. To do this connect to the kit's serial port using a client like putty and while keeping pressed Ctrl+C on the console power off and then on the device. A prompt should appear.

From here execute these commands:

```
// U-boot environment changes for us01-kit, us03-kit and ns01-kit
# setenv mmcboot 'run findfdt; mmc rescan; ext2load mmc 1:1 ${loadaddr} /boot/zImage;
ext2load     mmc 1:1 ${fdtaddr} /boot/${fdtfile}; setenv mmcroot /dev/mmcblk1p1; run
mmcargs; bootz       ${loadaddr} - ${fdtaddr};'
# saveenv

// U-boot environment changes for us02-kit
# setenv mmcroot /dev/mmcblk0p1
# setenv mmcloadcmd ext2load
# setenv bootimage /boot/zImage
# setenv fdtimage /boot/socfpga.dtb
# setenv mmcload "mmc rescan; ${mmcloadcmd} mmc 0:${mmcloadpart} ${loadaddr}
      ${bootimage}; ${mmcloadcmd} mmc 0:${mmcloadpart} ${fdtaddr} ${fdtimage}"
# saveenv
```

To restore the bootloader's environment and boot again from SD-card stop the machine at the u-boot's prompt again and type:

```
# env default -a
# saveenv
```

6 Setup the workspace for building applications

This section describes how to setup a 64bit Linux PC or virtual machine to be able to build applications for the target development kit. Our virtual machine and our Docker image are already preconfigured and ready to use, these steps can be skipped when using one of these solutions.

6.1 Cross development environment setup

Download the latest v4.x SDK from here:

```
US01kit:                http://download.exorembedded.net:8080/Public/usom01/SDK
US02kit:                http://download.exorembedded.net:8080/Public/usom02/SDK
US03kit:                http://download.exorembedded.net:8080/Public/usom03/SDK
NS01kit:                http://download.exorembedded.net:8080/Public/nsom01/SDK
```

Execute the SDK installation file `exor-evm-qt5-sdk.sh` (requires admin privileges):

```
$ cd /opt
$ sudo chmod a+x ./ exor-evm-qt5-sdk.sh
$ sudo ./exor-evm-qt5-sdk.sh
```

You will be asked for the installation directory, press enter to use the default, `/opt/exorintos/2.3.2`. To setup the cross development environment for the current shell run this command (correct the path if you have changed the default installation directory):

```
// Environment setup for us01-kit
$ source /opt/exorintos/2.3.2/environment-setup-cortexa8hf-vfp-neon-poky-linux-gnueabi

// Environment setup for us02-kit and us03-kit
$ source /opt/exorintos/2.3.2/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi

// Environment setup for ns01-kit
$ source /opt/exorintos/2.3.2/environment-setup-cortexa7hf-vfp-neon-poky-linux-gnueabi
```

To build a simple hello world application use the arm cross compiler that should now be reachable from your PATH:

```
$ arm-poky-linux-gnueabi-gcc main.c -o hello_world
```

6.2 Connecting to the device

On each device a console is active over serial port for debugging purposes. An ssh server is also running, useful for having a shell over ethernet or transferring files via sftp. In both cases the username to use is `root`, no password is required.
If your system has an avahi client installed the kit can also be addressed by its hostname:

```
US01kit:            exorUS01kit.local
US02kit:            exorUS02kit.local
US03kit:            exorUS03kit.local
NS01kit:            exorNS01kit.local
```

6.3 QtCreator setup

When developing Qt applications it may be usefull to have the Qt IDE preconfigured to use the toolchain. You can get latest QtCreator package from DIGIA here:

```
http://download.qt.io/official_releases/qtcreator/3.3/3.3.2/qt-creator-opensource-linux-
x86-3.3.2.run
```

Install it in your machine:

```
$ sudo chmod a+x ./qt-creator-opensource-linux-x86-3.3.2.run
$./qt-creator-opensource-linux-x86-3.3.2.run
```

You will find qtcreator installed in `~/qtcreator-3.3.2`. Start it:

```
$ ~/qtcreator-3.3.2/bin/qtcreator
```

We are now going to setup the QtCreator build kit for the target.
From Tools menu select "Options…" -> "Build & Run", then follow these steps:

1) In the "Compilers" tab click on "Add" -> "GCC" -> "C" and select the cross compiler picking it from the SDK installation folder. If the SDK has been installed in the default location the correct path is: `/opt/exorintos/2.4.2/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc`.
Optionally edit "Name" to give a more meaningful name for the entry, select "arm-linux-generic-elf-32bit" as ABI and finally click "Apply".

2) From the same tab now click "Add" -> "GCC" -> "C++" and select `/opt/exorintos/2.4.2/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-g++` instead. Again, select "arm-linux-generic-elf-32bit" as ABI and click "Apply"
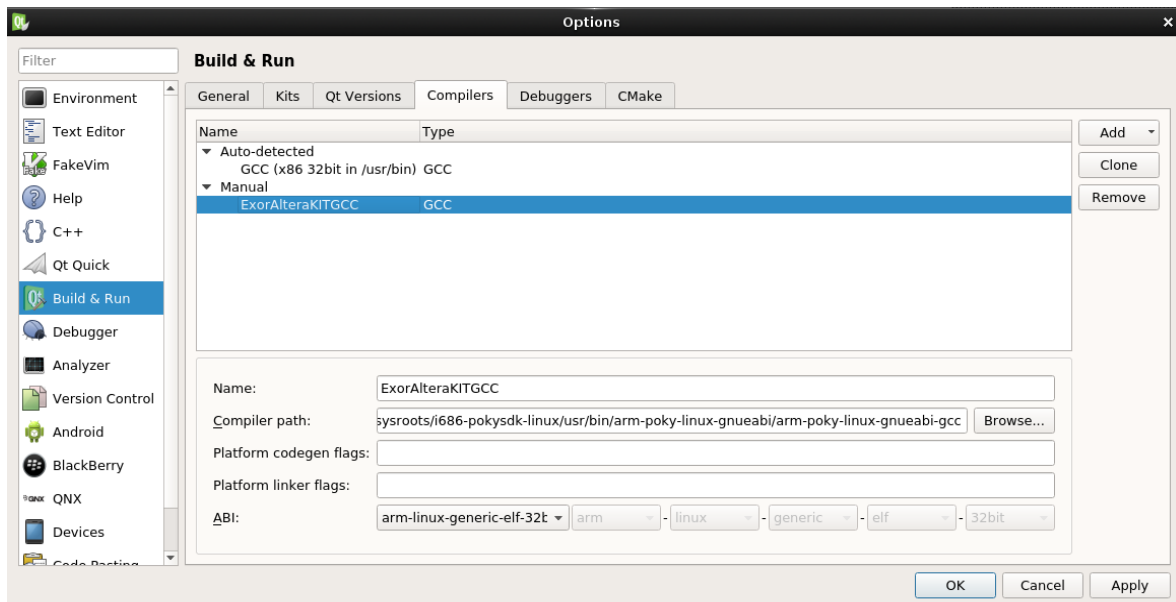
3) From "Debuggers" tab press "Add" and select gdb from the same directory. The default location is: `/opt/exorintos/2.4.2/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gdb` .
Optionally edit "Name", then click "Apply".
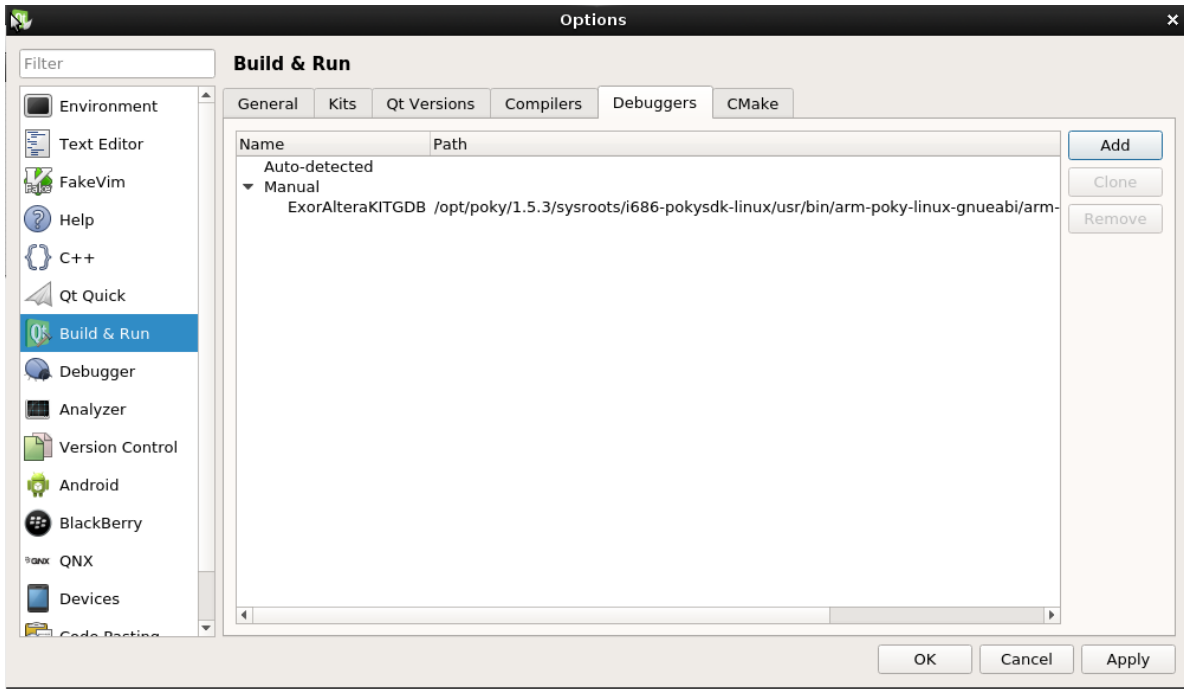
4) From "Qt Versions" tab, press "Add..". The default path to select is:
`/opt/exorintos/2.4.2/sysroots/x86_64-pokysdk-linux/usr/bin/qmake`. QtCreator should automatically recognize the qt version selected. Press "Apply".
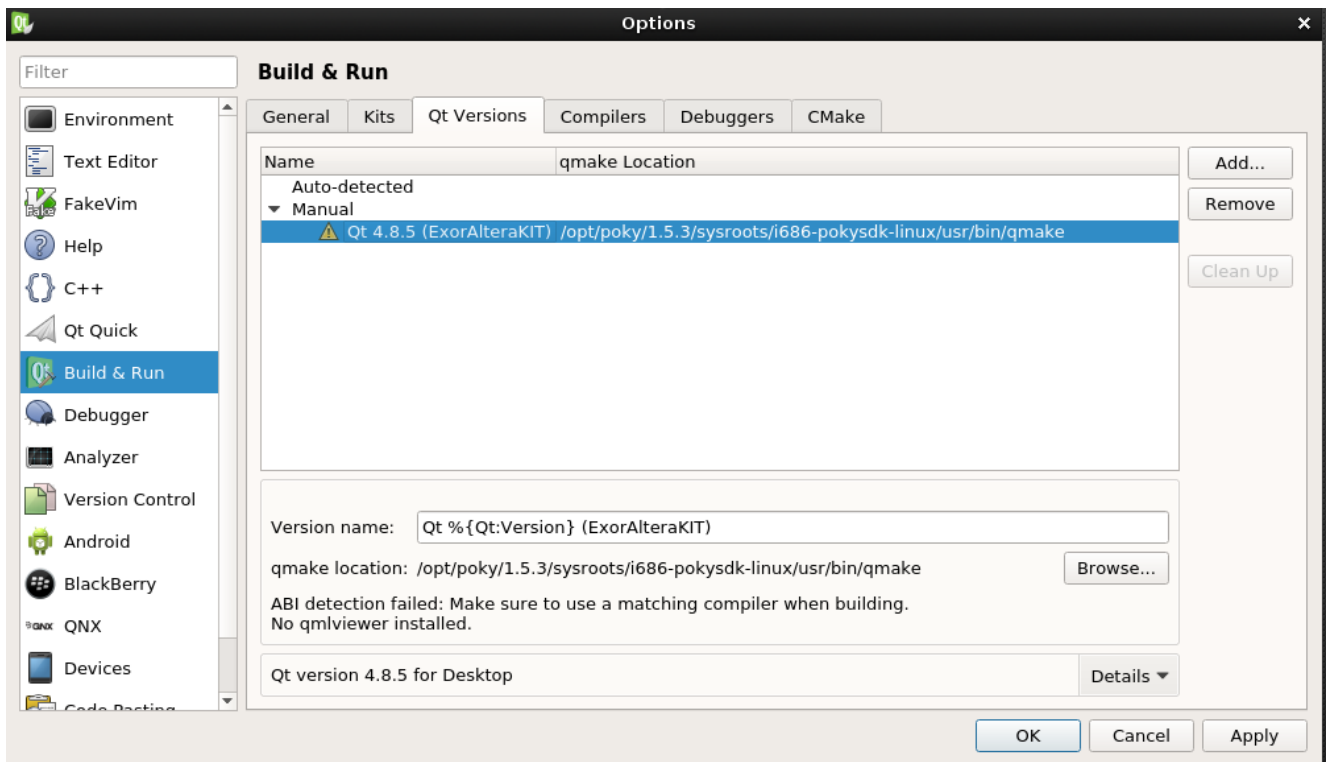
5) This step is required for configuring automatic application deploy on the target.
Move from "Build & Run" section to "Devices". Click "Add..", select "Generic Linux Device" and press "Start Wizard". Fill in these informations:

- **Name**: the device name, for example, `us01-kit`.
- **Host name**:

```
US01-kit:          exorUS01kit.local
US02-kit:          exorUS02kit.local
US03-kit:          exorUS03kit.local
NS01-kit:          exorNS01kit.local
```

- **Username**: `root`.
- **Authentication type**: set to "Password".
- **User's password**: leave empty, no password is needed.

Click "Next" and then "Finish". Qt Creator will attempt a test connection, if the device is already powered on and reachable everything should be ok.
If for any reason you cannot reach the target by its hostname make sure avahi is installed on your system or edit "Host name" to set the actual board IP address instead. Press on "Test" button to check the connection again.
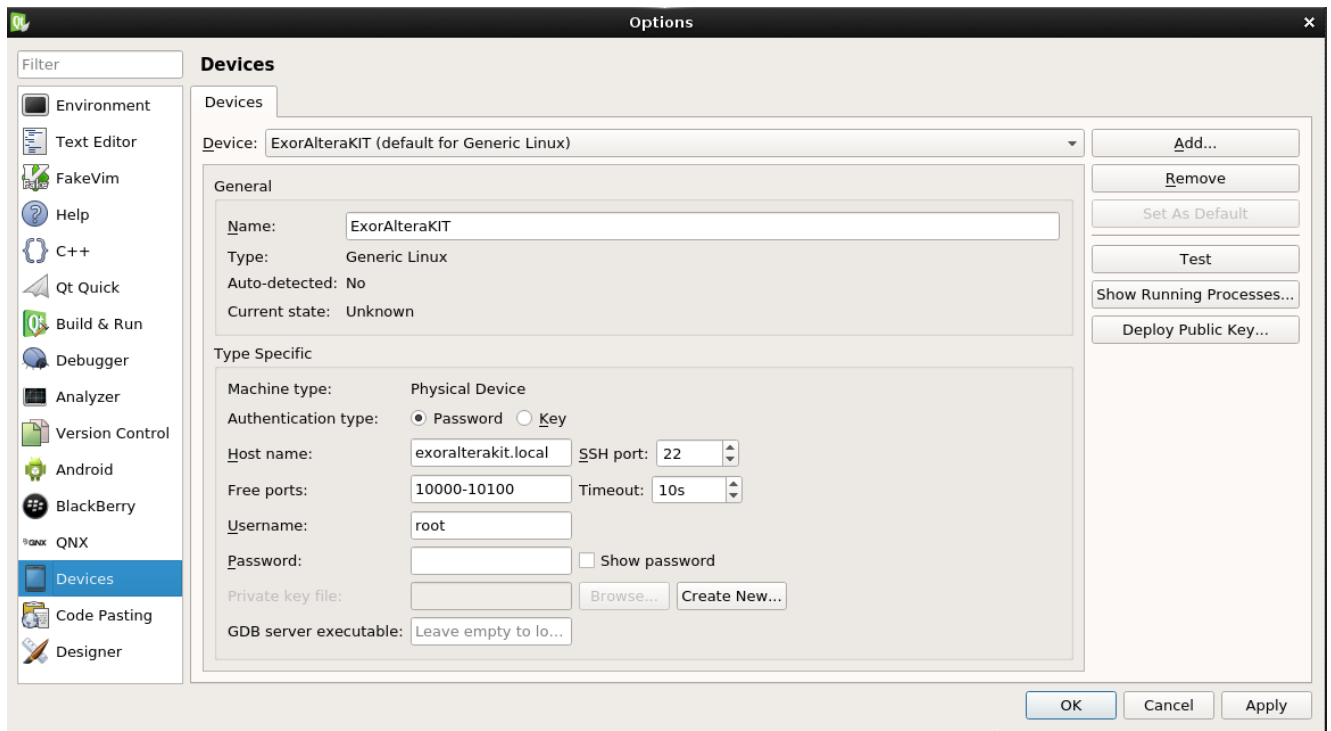
6) Finally move again to "Build & Run" section, "Kits" tab. Combine all pieces together in a new kit. Click "Add" and fill in as follows:
   - **Name**: choose a name for the kit.
   - **Device Type**: select "Generic Linux Device".
   - **Device**: select the device configured in 5).
   - **Sysroot**: if the SDK is installed in the default location, these are the paths to select:

```
US01-kit:    /opt/exorintos/2.4.2/sysroots/cortexa8hf-neon-poky-linux-gnueabi
US02-kit:    /opt/exorintos/2.4.2/sysroots/cortexa9hf-neon-poky-linux-gnueabi
US03-kit:    /opt/exorintos/2.4.2/sysroots/cortexa9hf-neon-poky-linux-gnueabi
NS01-kit:    /opt/exorintos/2.4.2/sysroots/cortexa7hf-neon-poky-linux-gnueabi
```

   - **Compiler**: select C and C++ compilers by name as configured in 1) and 2).
   - **Debugger**: select debugger by name as configured in 3).
   - **Qt version**: select qt version added in 4).
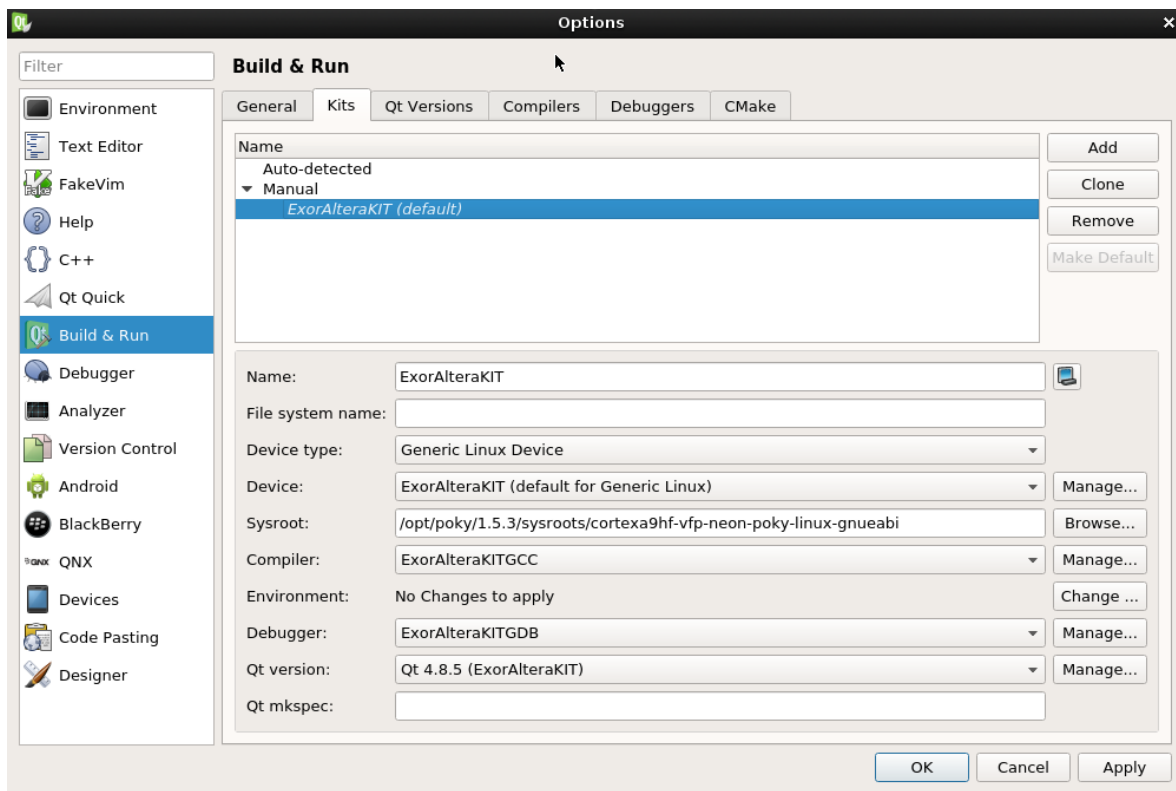
6.3.1 Application deploy

Before starting here, make sure QtCreator has been correctly configured for application deployment and that the development kit is reachable.
1) First, let's create a dummy Qt project. Select "File" -> "New File or Project…" -> "Qt Widgets Application" and click "Choose". Enter a project name, press "Next". Make sure that in the "Kit Selection" wizard dialog the SDK kit for the target is selected.

2) Make sure the target kit the one currently in use by checking in the menu on the left shown below:
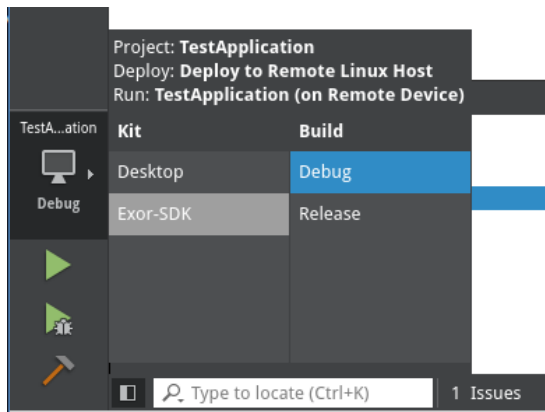
3) Now edit the .pro project file to add these two lines:

```
target.path = /home/root/
INSTALLS += target
```

This will define where the application will be installed on the device (/home/root)
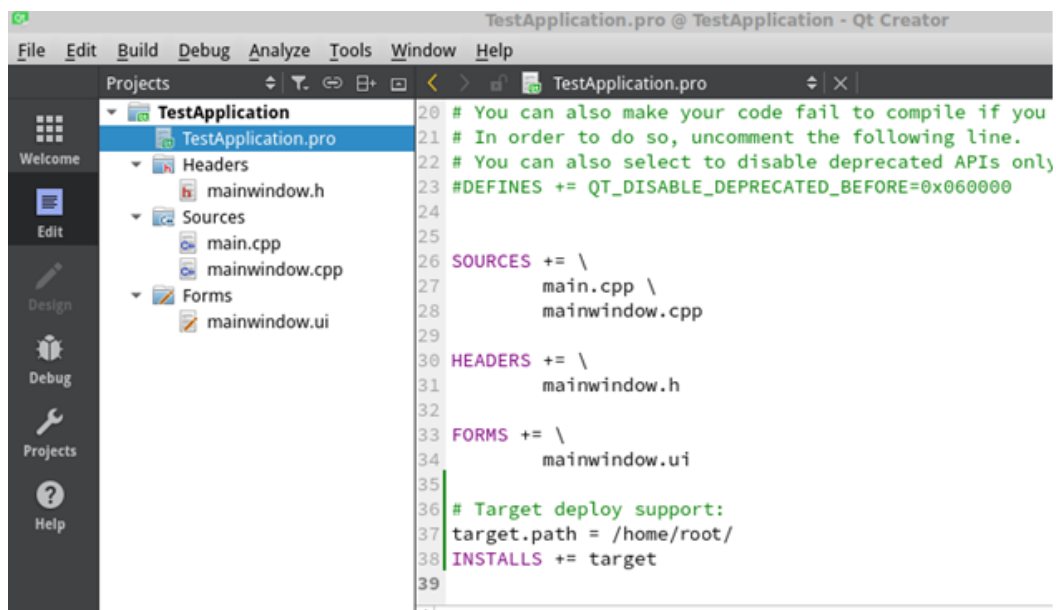
4) Finally press the green "play" button in the menu on the left or use the "Ctrl+R" shortcut. QtCreator should compile the application and an empty Qt window should appear on the device.

7 Using Expansion Plugins

7.1 Use PLCM01 plugin (Canbus)

7.1.1 Plugin connection

The Plcm01 can be plugged in every plugin connector.
If you connect the Plcm01 on the connector "Plugin 1" the system provide the Can0 interface.
If you connect the Plcm01 on the connector "Plugin 2" the system provide the Can1 interface.

7.1.2 System configuration and Plugin use

Once connected the plugin you can power-up the development kit and wait the booting process.
With the following system command you can:

- Set the can interface

```
# ip link set can0 up type can bitrate 250000
```

- Enable the can interface

```
# ifconfig can0 up
```

- Send can packet

```
# cansend can0 -i 0x1A5 0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF
```

- Received can packet

```
# candump can0                                                                25
```

7.1.3 Canbus connector (CN2)
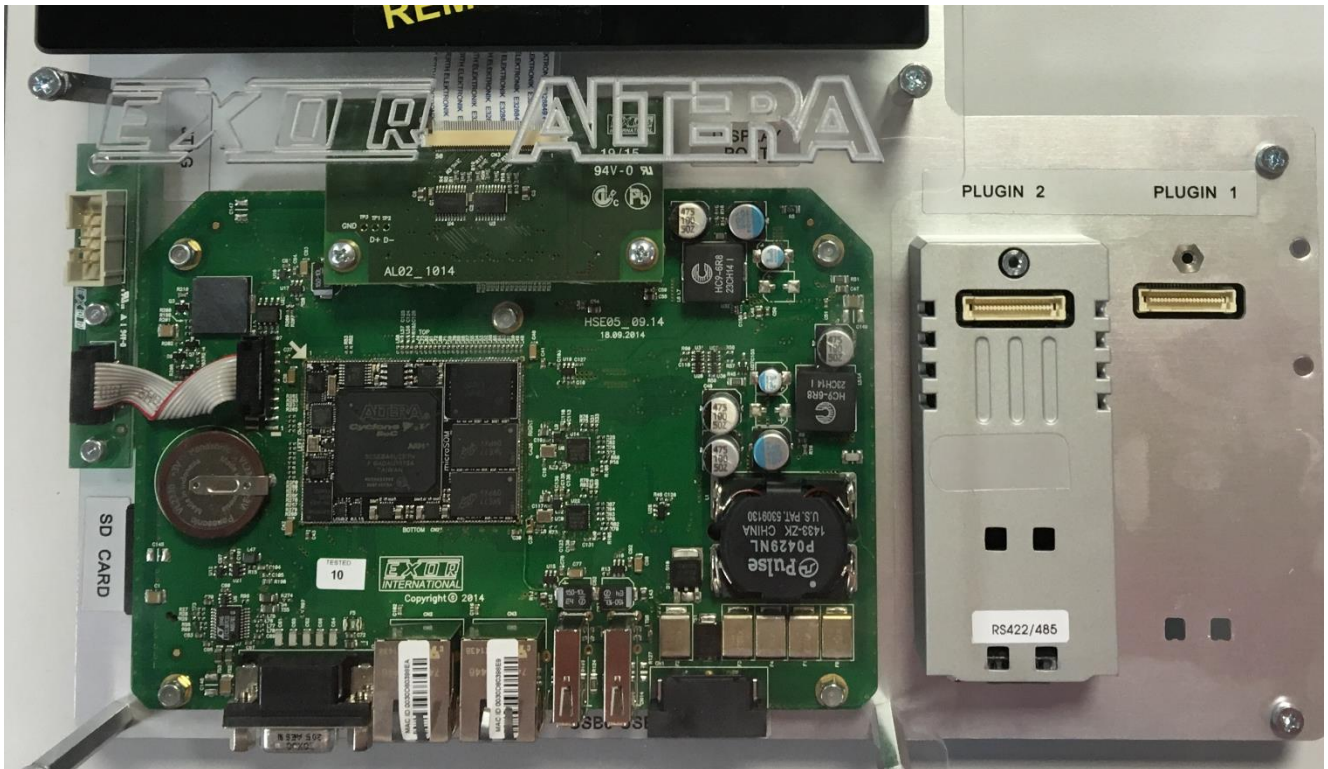
Plcm01 is equipped with a standard male DB9 canbus connector.
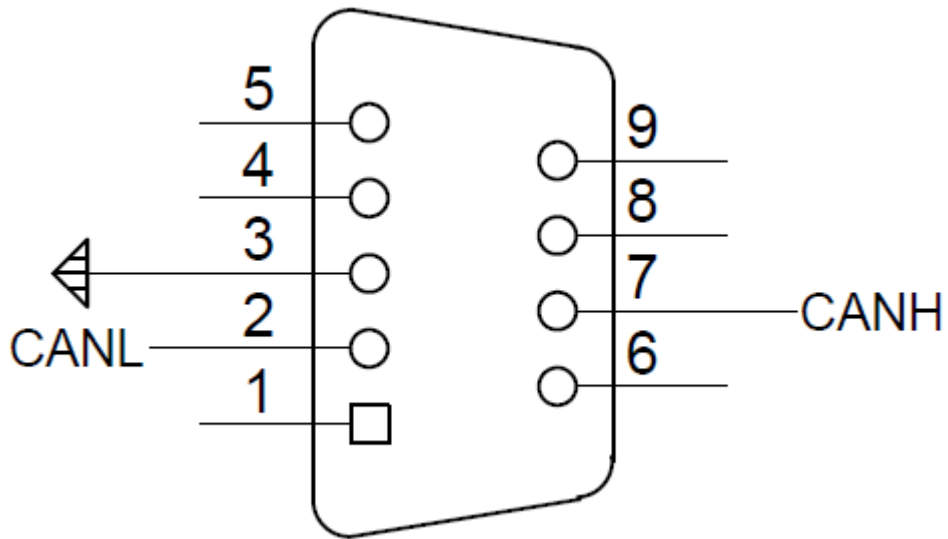
7.2 Use PLCM04 module (RS-422/485)

7.2.1 Plugin connection

The Plcm04 can be plugged in every kit plugin connector.
If you connect the Plcm04 on the connector "Plugin 1" the system provide the ttyS1 interface.
If you connect the Plcm04 on the connector "Plugin 2" the system provide the ttyS2 interface.

7.2.2 System configuration and Plugin use

Once connected the plugin you can power-up the development kit and wait the booting process. The system is just configured to use this module, and you can read/write on the serial port.

7.2.3 Exaple C code

Here a simple example writed in C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/serial.h>
#include <asm-generic/termbits.h>
#include <string.h>
#include <signal.h>

/* Driver-specific ioctls: */
#define TIOCGRS485 0x542E
#define TIOCSRS485 0x542F

#define MSG_LENGTH   255
#define HELLO_WORLD "Hello from Exor uS02 kit\n"

#define SERIAL_PORT_PLUGIN_1 "/dev/ttyS1"
#define SERIAL_PORT_PLUGIN_2 "/dev/ttyS2"

/*
 * SELECT USED PORT
 */
#define SERIAL_PORT SERIAL_PORT_PLUGIN_2

int main(int argc, char const *argv[])
{
    int i, fd, ret=0;
```

```c
    struct serial_rs485 rs485conf;
    unsigned char b[MSG_LENGTH], c[MSG_LENGTH];

    fprintf(stdout, "Start!\n");
    fprintf(stdout, "Open open port %s...!", SERIAL_PORT);
    fd = open(SERIAL_PORT, O_RDWR);
    if (fd < 0) {
        perror ("Open device failure");
        return -1;
    }
    fprintf(stdout, " done!\n");

    fprintf(stdout, "Enable RS485 mode...");
    if (ioctl(fd, TIOCGRS485, &rs485conf) < 0) {
        perror ("ioctl failure");
        return -2;
    }
    rs485conf.flags = SER_RS485_ENABLED | SER_RS485_RTS_ON_SEND;
    if (ioctl(fd, TIOCSRS485, &rs485conf) < 0) {
        perror ("ioctl failure");
        return -3;
    }
    fprintf(stdout, " done!\n");

    //Set custom or std baudrate
    struct termios2 tio;
    ioctl(fd, TCGETS2, &tio);
    tio.c_cflag &= ~CBAUD;
    tio.c_cflag |= BOTHER;
    tio.c_ispeed = 115200;
    tio.c_ospeed = 115200;
    ioctl(fd, TCSETS2, &tio);

    write(fd, HELLO_WORLD, strlen(HELLO_WORLD) );
    while( strncmp(b, "exit", 4) )
    {
        fprintf(stdout, "Waiting data on %s \n", SERIAL_PORT);
        memset( b, 0, sizeof(b) );
        ret = read(fd, b, sizeof(b) );
        printf("Data received: %s\n", b );

        strcpy(c, "uS02 send: \t");
        strcat(c, b);
        write(fd, c, strlen(c) );
        printf("Data sendend: %s\n", c );
    }

    fprintf(stdout, "Close fd...");
    if (close (fd) < 0) {
        perror ("Close device failure");
        return -4;
    }
    fprintf(stdout, "done!\n");
    fprintf(stdout, "Stop!\n");
    return 0;
}
```
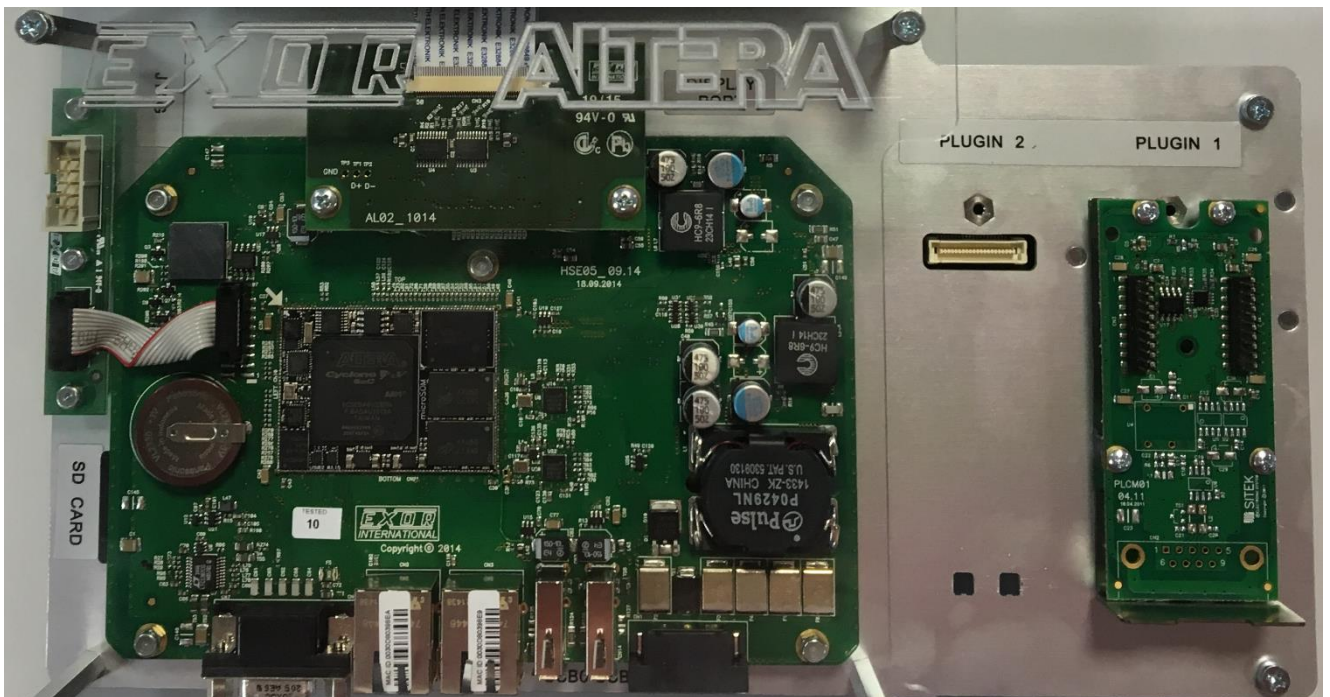
7.2.4 RS485 connector (CN2)

Plcm04 is equipped with a standard male DB9 connector.
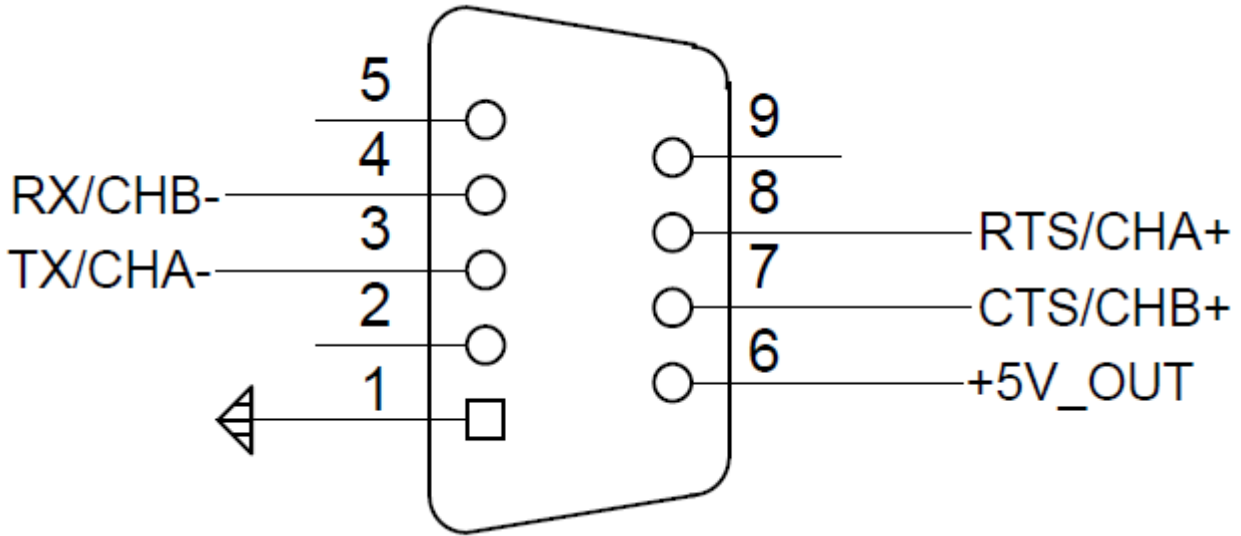
7.3   Use PLCM05 module (Expansion module)

The Plcm05 is a simple plugin build to simplify the connections, and use various interfaces.

7.3.1   SPI Plugin connection

The Plcm05 can be plugged in every development kit plugin connector.
If you connect the Plcm05 on the connector "Plugin 1" the system provide the spidev0.1 interface.
If you connect the Plcm05 on the connector "Plugin 2" the system provide the spidev1.1 interface.

7.3.2 SPI System configuration and plugin use

Once connected the plugin you can power-up the development kit and wait the booting process. The system is just configured to use this module, and you can read/write on the SPI port.

7.3.3 SPI Example C code

Here a simple example writed in C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <errno.h>

#include <fcntl.h>
#include <linux/spi/spidev.h>
#include <stdio.h>
#include <sys/ioctl.h>

#define MSG_LENGTH  2
#define SPI_PORT_PLUGIN_1 "/dev/spidev0.1"
#define SPI_PORT_PLUGIN_2 "/dev/spidev1.1"

/*
 * SELECT USED PORT
 */
#define SPI_PORT SPI_PORT_PLUGIN_1

static void writeSPI(unsigned char buf[2])
{
    int fd;
    unsigned char swap_buf[2];
    char name[20];
    struct spi_ioc_transfer xfer[2];

    sprintf( name, SPI_PORT );
    //fprintf(stdout,"writeSPI on %s \n ", name );
    fd = open(name, O_RDWR);
    if (fd < 0) {
        perror("Open");
        return;
    }
    memset(xfer, 0, sizeof xfer);
    memset(swap_buf, 0, sizeof swap_buf);

    swap_buf[0] = buf[1];
    swap_buf[1] = buf[0];
    xfer[0].tx_buf = (unsigned long)swap_buf;
    xfer[0].len = 2;
    ioctl(fd, SPI_IOC_MESSAGE(2), xfer);
    close(fd);
}

static void readSPI(unsigned char buf[2])
{
    int fd;
    unsigned char swap_buf[2];
    char name[20];
    struct spi_ioc_transfer xfer[2];

    sprintf( name, SPI_PORT );
    //fprintf(stdout,"readSPI on %s \n ", name );
    fd = open(name, O_RDWR);
    if (fd < 0) {
        perror("Open");
        return;
    }
    memset(xfer, 0, sizeof xfer);
    memset(swap_buf, 0, sizeof swap_buf);

    swap_buf[0] = buf[1];
```

```c
    swap_buf[1] = buf[0];
    xfer[0].tx_buf = (unsigned long)swap_buf;
    xfer[0].len = 1;
    xfer[1].rx_buf = (unsigned long) buf;
    xfer[1].len = 1;
    ioctl(fd, SPI_IOC_MESSAGE(2), xfer);
    close(fd);
}

int main(int argc, char const *argv[])
{
    int i;
    unsigned char b[MSG_LENGTH];

    for(i=0; i<0xFFFF; i++)
    {
        memset( b, 0, sizeof(b) );
        b[1] = ((unsigned char) ((i & 0xFF00) >> 8));
        b[0] = ((unsigned char) ((i & 0x00FF) >> 0));

        writeSPI( b );

        usleep(100000);
    }

    exit(0);

}
```

CN4 connector pinout diagram:

Pin	Signal		Pin	Signal
1	+5V		2	+3V3
3	SPI_CLK		4	
5			6	RST_OUT*
7	SDA		8	SCL
9	SPI_SIMO		10	SPI_INT*
11	SPI_CS1*		12	SPI_SOMI
13	(CAN_RX)		14	(CAN_TX)
15	TX		16	
17	RX		18	CTS
19	+3V3		20	
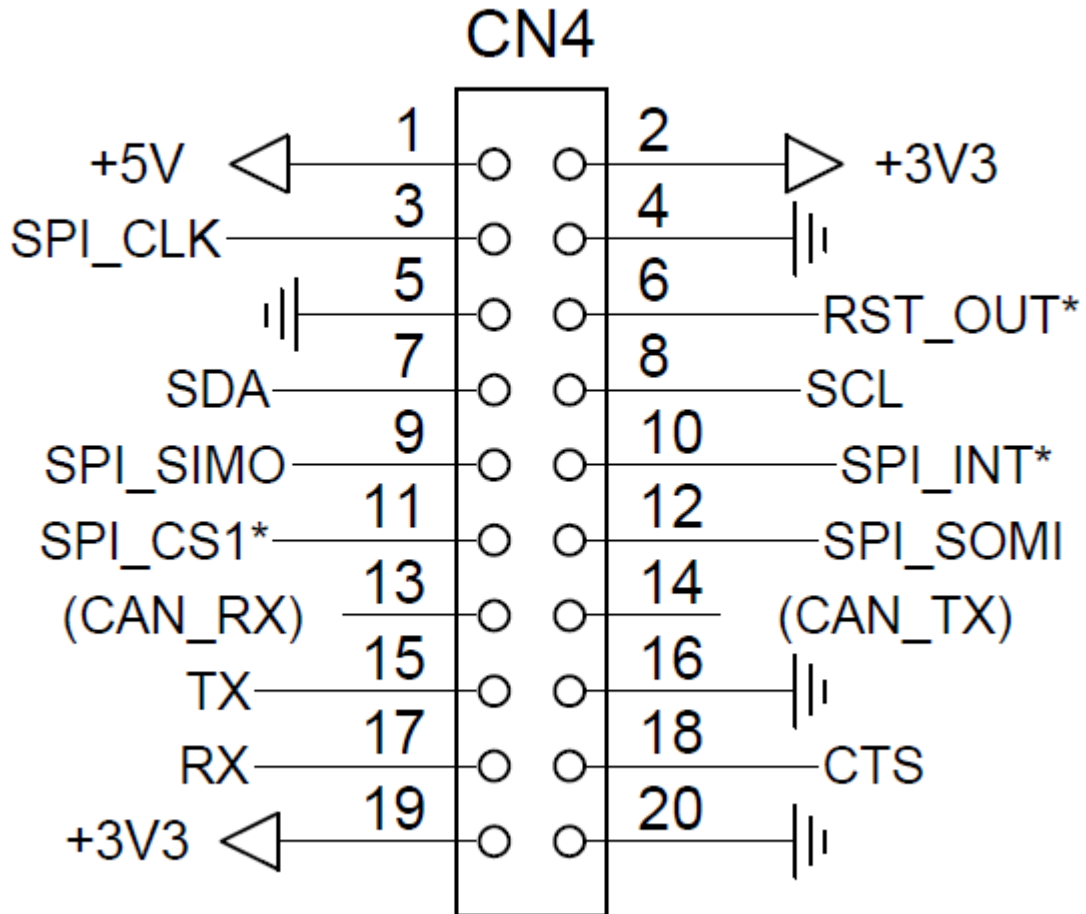
8   Upgrade FPGA firmware (us02-kit only)

Power up the uS02 kit and block the boot process during u-boot countdown by pressing CTRL-C.

```
U-Boot 2015.02.17 (Aug 21 2016 - 12:28:46)

CPU   : Altera SOCFPGA Platform
BOARD : Altera SOCFPGA Cyclone V Board
I2C:   ready
DRAM:  1 GiB
MMC:   initializing dwmmc...
initializing ultisdc...
ALTERA DWMMC: 0, ultisdc: 1
In:    serial
Out:   serial
Err:   serial
Loading configuration from I2C SEEPROM
bootargs environment variable set to "setenv bootargs console=ttyS0,115200 ro
Hit any key to stop autoboot:  0
SOCFPGA_CYCLONE5 #
```

Type the following commands:

```
# mw.l ff210010 7fe
# run bootcmd
```

Now wait the booting process and in Linux terminal type:

```
# dd if=path_to_new_fpga_image.bin of=/dev/mtdblock0 bs=1M
```

After few minutes the command ends, for use new FPGA firmware power-off and power-on the board.

9 JMobile Portable runtime

JMobile is a software suite designed to offer a complete HMI solution with client-server architecture. It is made of several software components, integrated into a unique application. JMobile applies the latest available technology developed for HMI in industrial automation to every situation where a user interface is required. The suite includes commissioning tools, to allow easy maintenance and configuration of multiple remote units, and both desktop and runtime engineering software for application development.

The portable version of JMobile is a standard Linux JMobile runtime provided as a chroot-based container designed to run under Linux 32bit ARM platforms.

> The portable JMobile runtime is provided for rapid prototyping and evaluation purposes and contains a subset (Codesys V3/, Modbus and the internal variables protocol) of the available protocols. In particular serial protocols are not supported, the serial port on the evaluation kits is only meant for debugging purpose.
>
> A closer integration with the final target system and access to the complete set of protocols can be achieved on demand during the product engineering phase.

9.1 JMobile portable runtime installation

By default JMobile in preinstalled on both the standard SD image and the rootfs generated by our standard Yocto recipes. In this case a JMobile icon can be seen on the desktop that can be used to manually start it.

The portable can however also be downloaded separately from here:

```
http://download.exorembedded.net:8080/Public/OpenHMI/
```

Then to install and run it from ssh follow these steps:

1. Copy it into the kit.

```
$ scp jmobile-[…]-portable-devkit.tar.gz root@[hostname]:~
```

2. Connect to the kit:

```
$ ssh root@[hostname]
```

3. Now, from the remote shell, untar the package in a folder with write permissions (e.g. `/opt`)

```
$ tar xzpf jmobile-[…]-portable-devkit.tar.gz /opt
$ rm –rf jmobile-[…]-portable-devkit.tar.gz
```

4. Make sure X11 is not running:

```
# /etc/init.d/xserver-nodm stop
```

5. Start JMobile:

```
# /opt/jmobile_portable/run.sh
```

In both cases it's possible to configure the BSP to automatically start JMobile Runtime at boot:

1) Remove the script xserver-nodm:

```
# update-rc.d –f xserver-nodm remove
```

2) Add a new script to the init sequence:

```
# echo "/opt/jmobile_portable/run.sh &" > /etc/init.d/jmobile
# chmod a+x /etc/init.d/jmobile
# update-rc.d jmobile defaults 99
```

9.2   JMobile OpenHMI Studio quick start guide

To download a free trial of OpenHMI Suite go to our web page dedicated to development kits on exorint.com:

```
https://exorint.com/product-category/embedded/dev-kits/
```

Select the device you are working with then, from the "Download" section, download the latest version of OpenHMI Suite. After installation, start OpenHMI Studio and create a new project from "File" -> "New..":
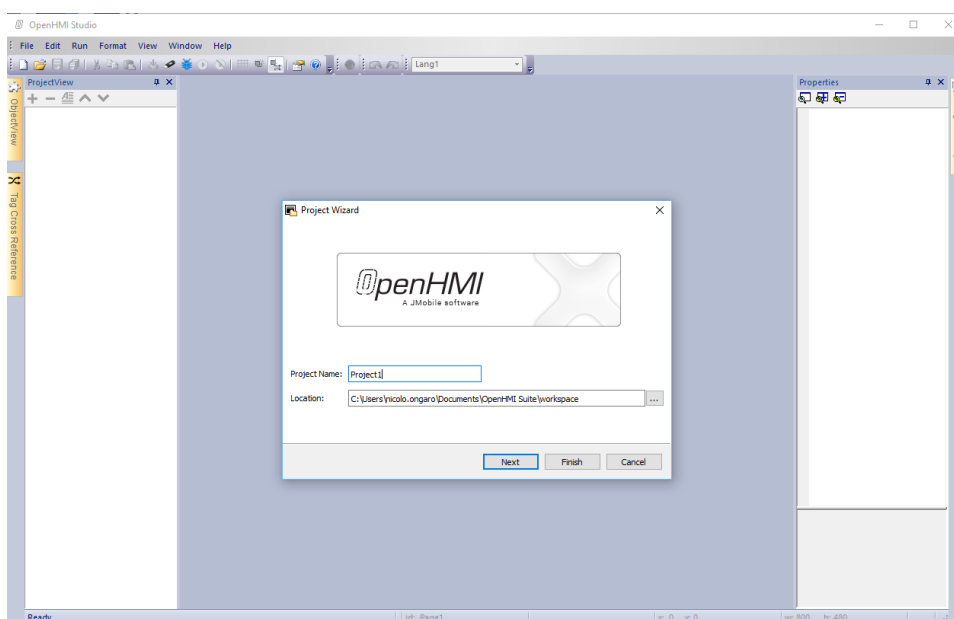
Chose a project name, select a location folder and click on "Next". Select now the correct target corresponding to the board:
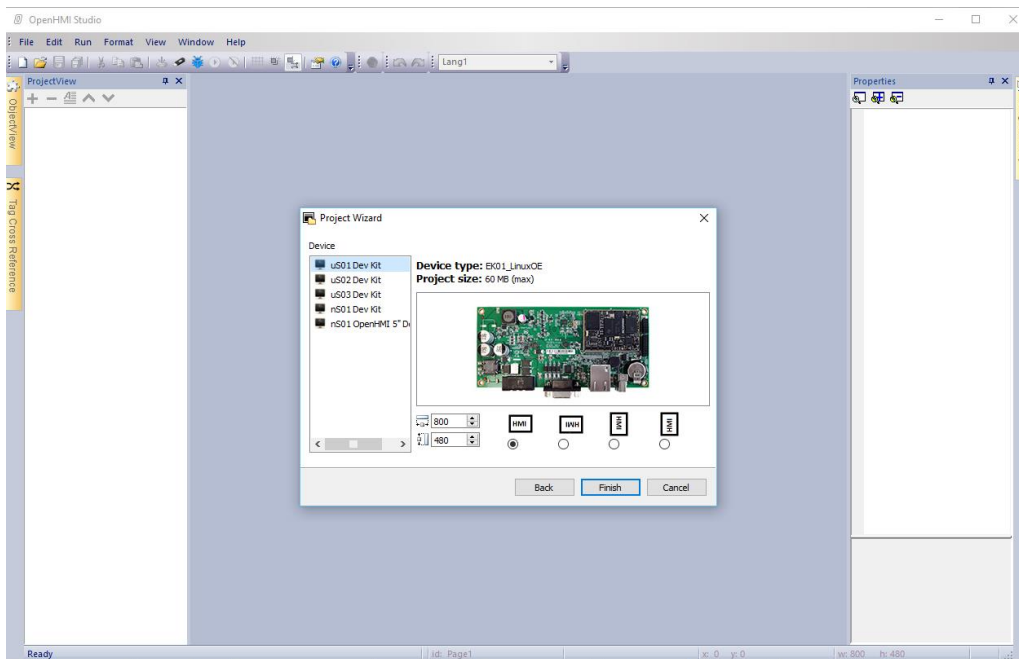
The goal is to create a project simply consisting of an increasing numerical counter. Although at the end it won't do very much, this example project will introduce you to some of the basic mechanics JMobile uses to combine protocols, data and visualization.

To begin with, from the "ProjectView" on the left panel, click on "Protocols". Here click "+" to add a new protocol and select "Variables" as shown in the figure below:
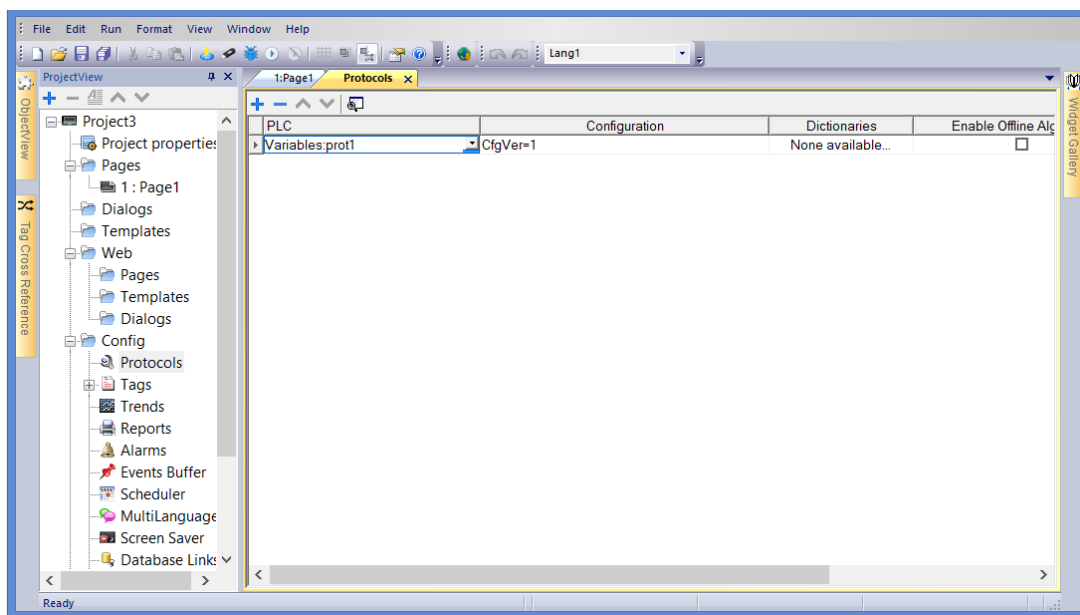
Use the left panel to move to "Tags". Press "+" and add an unsignedShort tag named "Tag1" representing our counter.
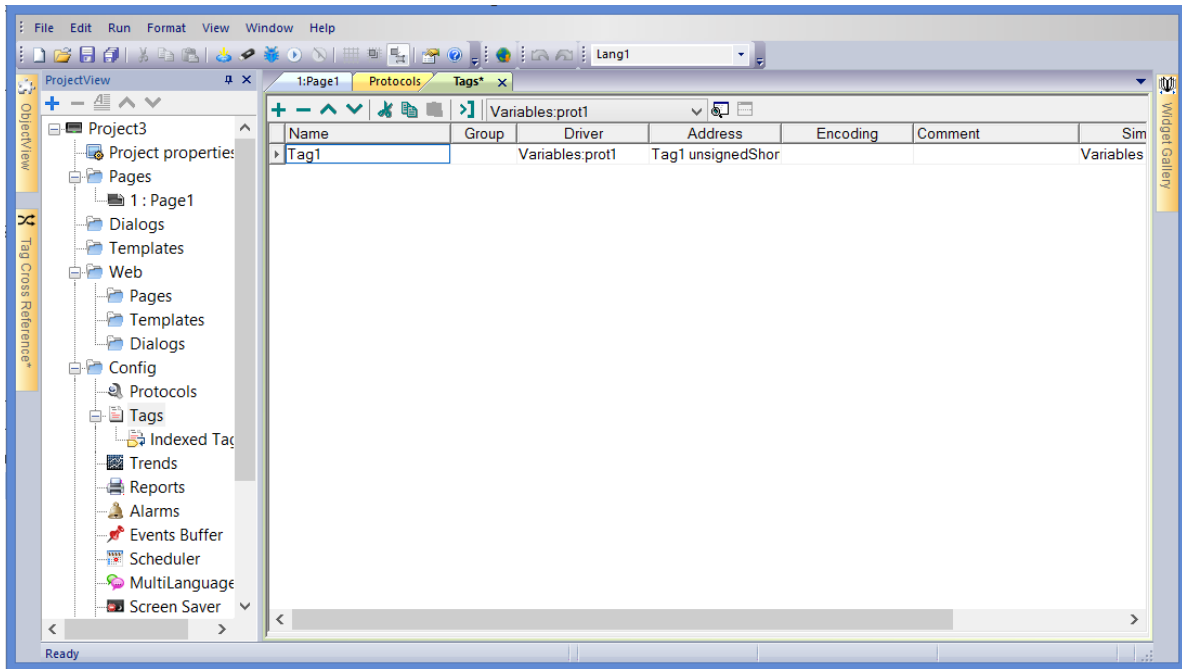
Add a numeric field widget to the project's page by dragging it from the Widget Gallery:
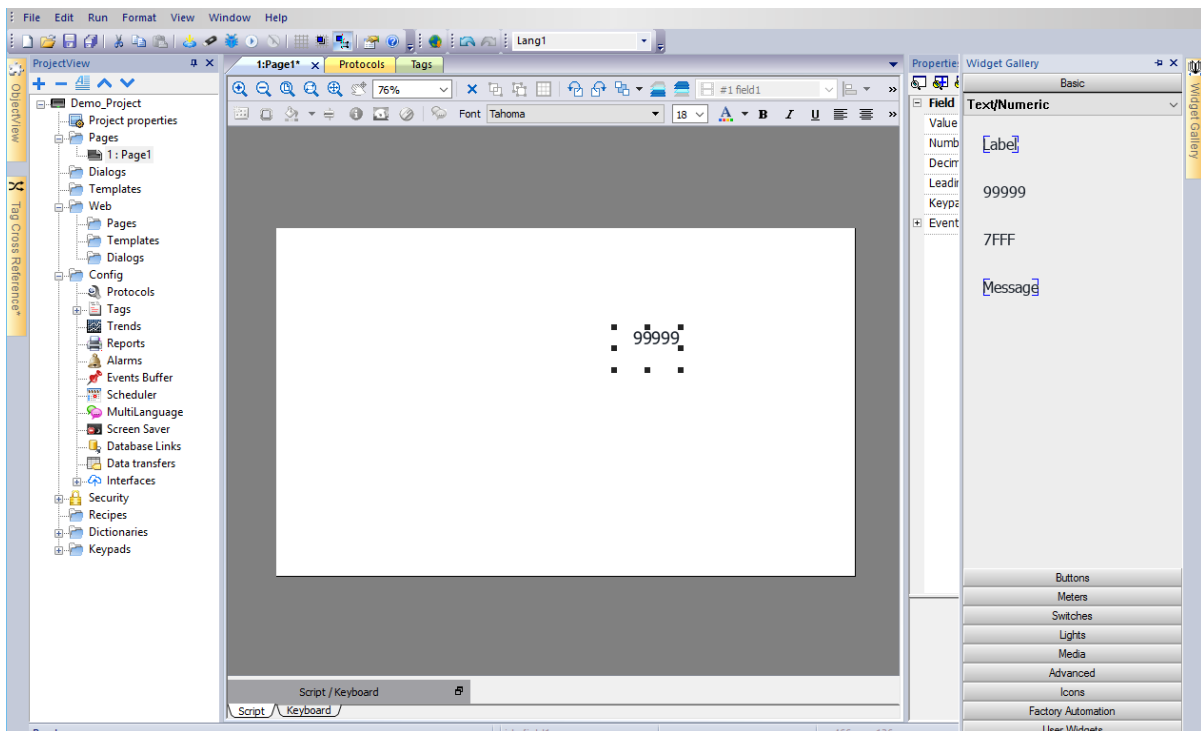
Double click on the numeric filed and select "Tag1" to bind the widget to the tag's value:
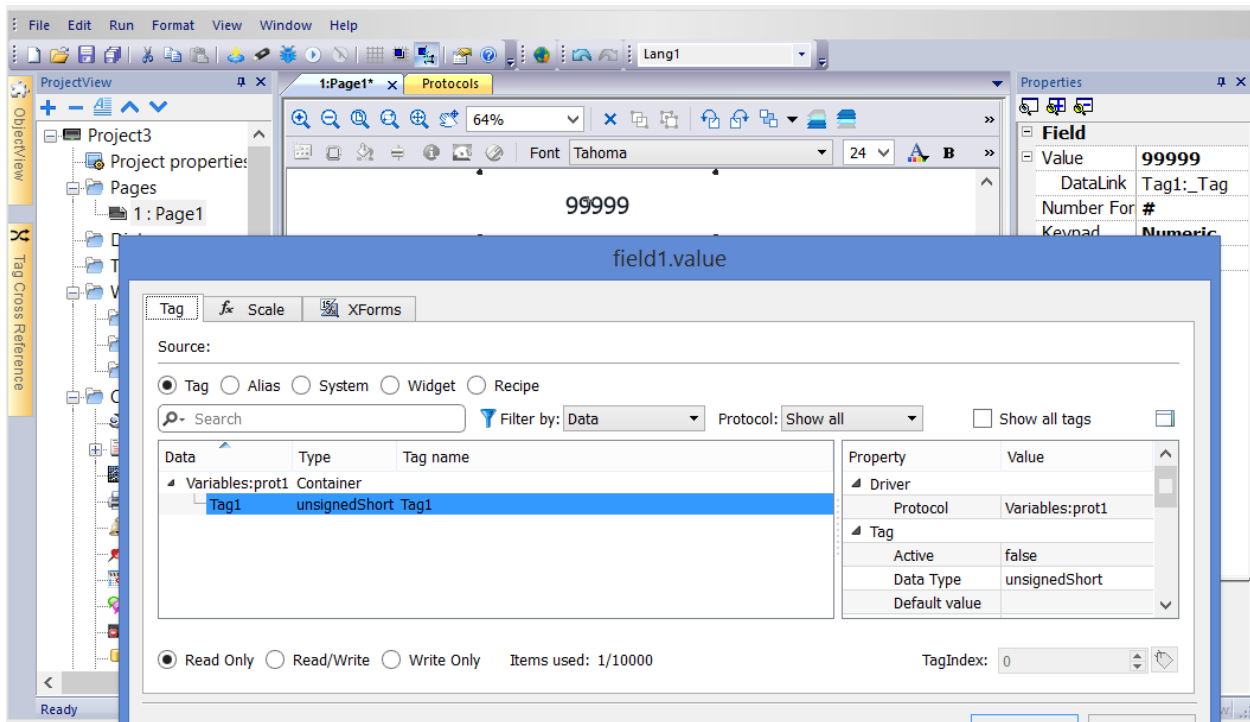
Now move to "Scheduler". Add a HighResolution scheduler with a "StepTag" action to increment the counter. Note in the figure below that our tag has been selected for "TagName" under "Action Properties":
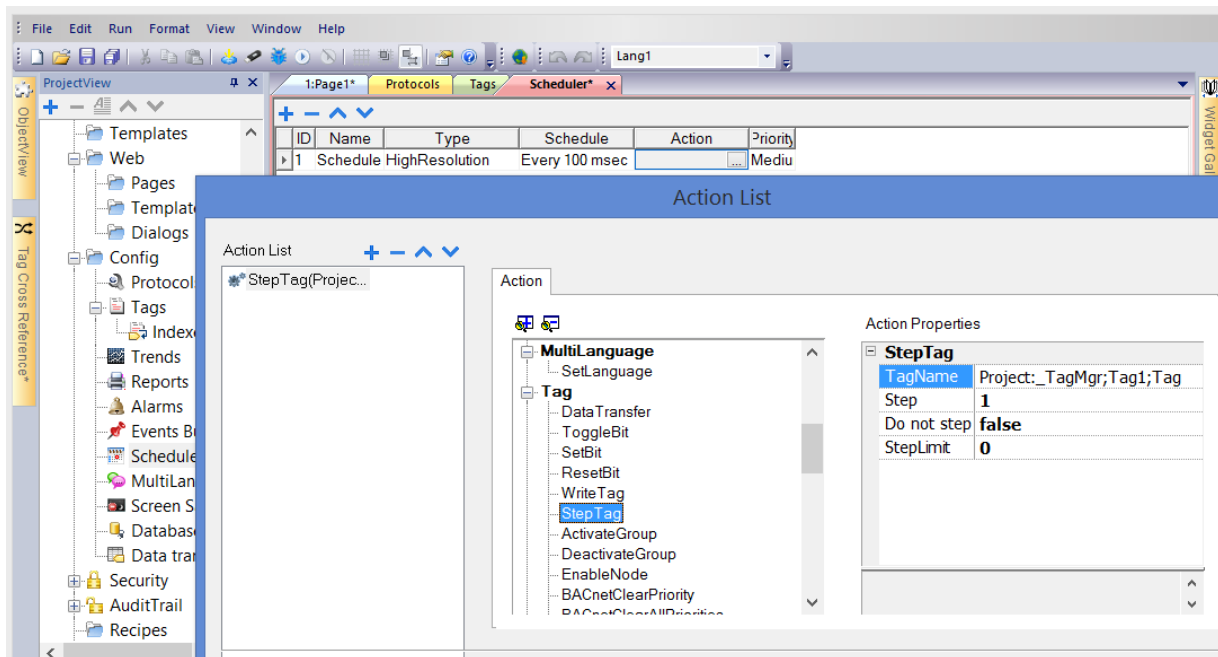
Click on the "Download to Target" icon in the toolbar or simply press Ctrl+D. Make sure the development kit is powered on, that you can reach it over the network and that JMobile Runtime is running. Select the target from the drop-down list and click "Download" to deploy the project:
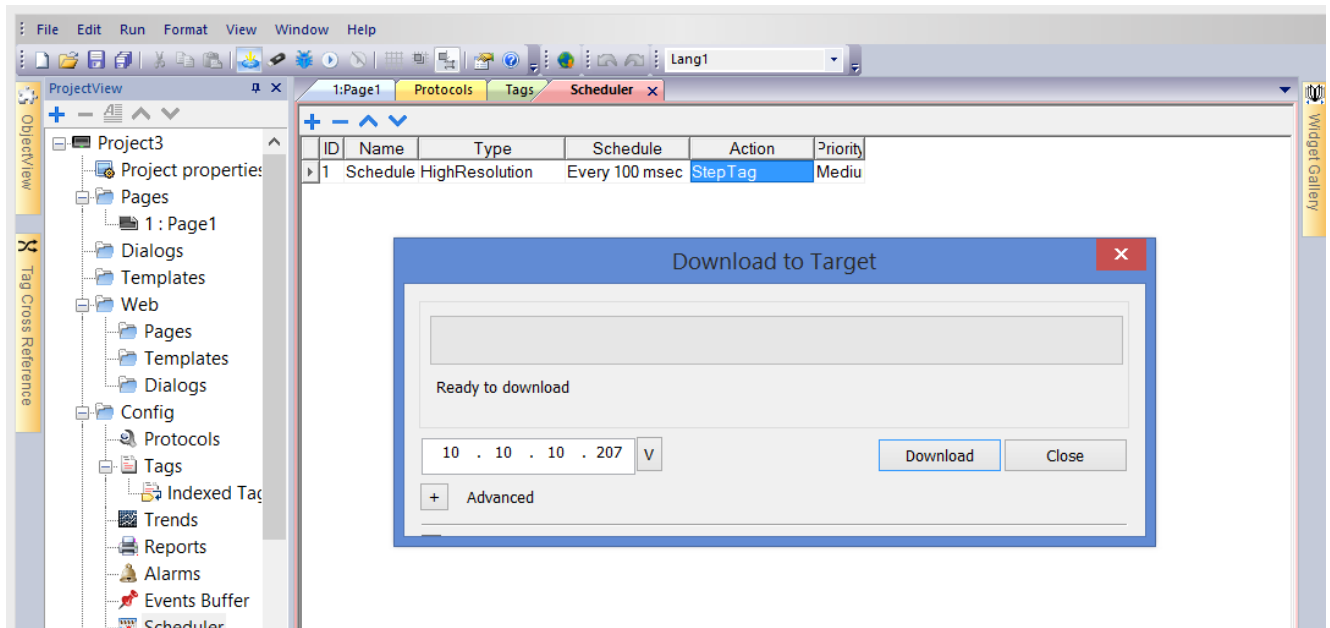
If the JMobile Runtime on the device is found to be an older version the Studio will automatically update it before downloading the project.

The portable version of JMobile uses non standard ports for FTP and HTTP protocols in order to avoid conflicts with other services on the host device:

```
FTP port:  2525
HTTP port: 8585
```

From the OpenHMI Studio nothing needs to be configured as long the device is selected form the target list the correct ports will be used. However when accessing a JM4Web html project page from a browser, the http port needs to be specified in the url bar:

```
http://<target-IP>:8585
```

10 CODESYS V3

The CODESYS V3 programming software can be downloaded for free from the CODESYS web site at www.codesys.com/download.html.
You will need to register before you can download the software. The version used in this chapter is CODESYS v3.5 SP10 Patch 5.

10.1 Enabling CODESYS runtime

The Codesys runtime is included inside the JMobile portable but to make it start it's required to enable it first. To do this all it's required to do is to create a specific file from a ssh shell:

```
# touch /opt/jmobile_portable/mnt/data/hmi/qthmi/codesys_auto
# sync
```

This is assuming the portable is installed in /opt/jmobile_portable. After rebooting the device Codesys will automatically start with JMobile Runtime.

10.2 Installing CODESYS Devices

A device descriptor is required to allow the standard CODESYS V3 to work with the evaluation kits. This is provided inside a Codesys .package file that can be found inside OpenHMI Studio installation directory:

```
C:\Program Files (x86)\Exor\OpenHMI Suite 2.8\CODESYS\V3\CODESYS_JMobile_[…].package
```

This file can be imported from the CODESYS programming software. Select "Tools" -> "Package Manager…" from the toolbar, the below dialog should appear:
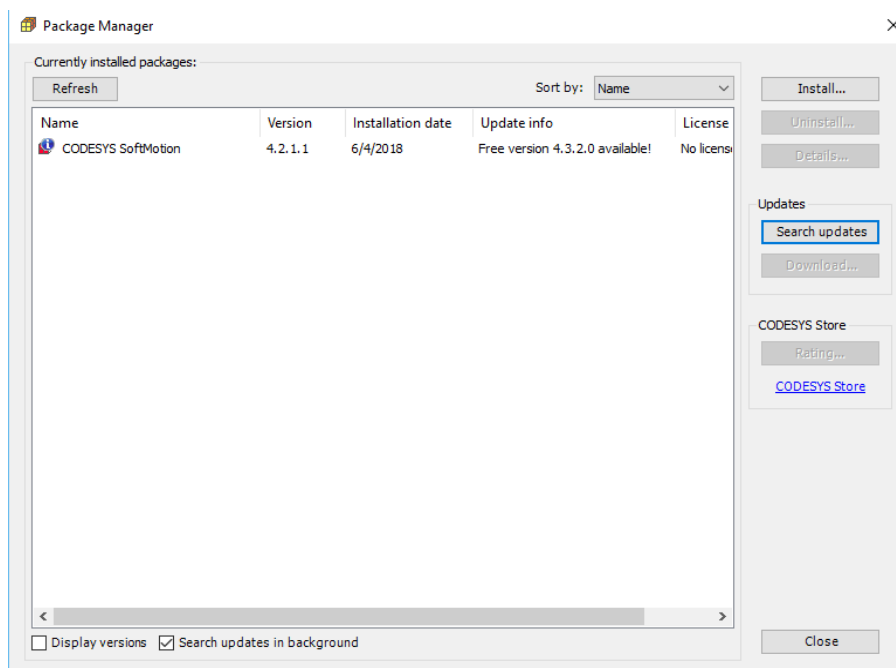
Click on the "Install…" button and browse for the package file. The choice can be confirmed with "Open". Finally select "Typical Setup" and continue trough the guided installation until completed.

10.3 Creation of a new PLC project

To create a new CODESYS V3 project select "File" > "New Project" or click on the corresponding icon from the upper tool bar. The "New Project" dialog will be displayed, here, among the available templates, select the "Standard project" template. Choose a project name and a location then confirm with "OK".
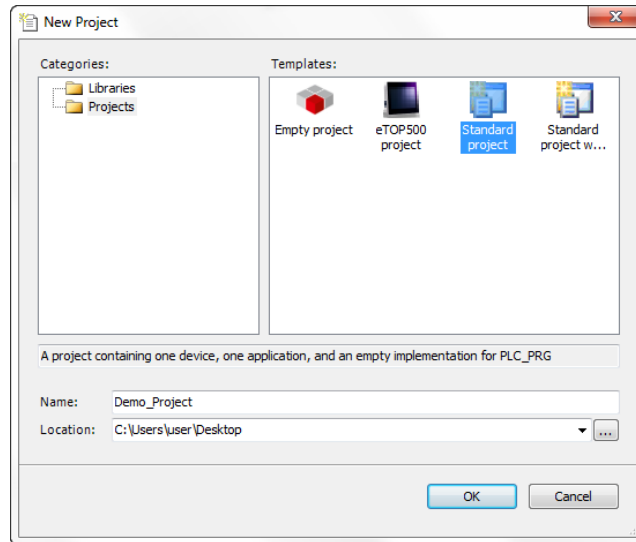
There are two different device descriptors for the development kits. The one to choose depends on whether the hardware does have an FRAM or not:

```
US02-kit, US03-kit, NS01-kit          DevKit(FRAM)
US01-kit, NS01-kit-OpenHMI            DevKit(no FRAM)
```

To complete the project creation, select the one of the two devices above and the preferred programming language of choice.
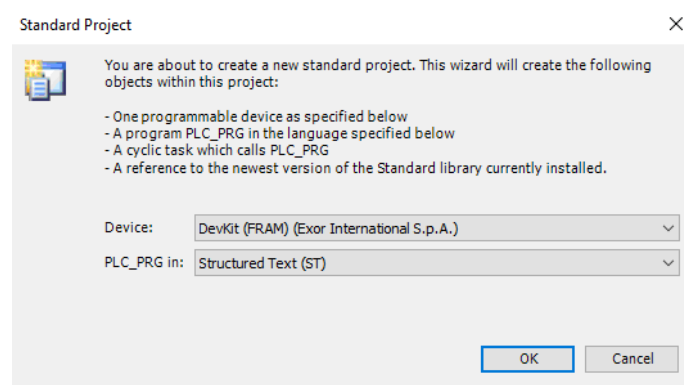
10.4 Communication setup in the CODESYS software

The selection of the target where to download the project must be done from the device communication settings tab before proceeding with the download operation.
Double click on the "Device (DevKit …)" node available in the project tree to display the Device properties in the work area, select the "Communication Settings" tab then click on the "Scan Network…" button.

The "Select Device" dialog will be displayed, this dialog lists all the compatible devices available in the network, select here your device then press on "OK".

In case more panels using CODESYS V3 PLC runtime are present into the network each panel can be recognized by the string between square brackets shown just after the device name. In the figure below, for example, the string is "0000.392B" for the highlight device. The last part of the string "392B" corresponds to the last 2 bytes of the panel IP Address in Hex format so, in this case, the corresponding operator panel is the one with IP address xxx.xxx.57.43 as 39Hex corresponds to 57 Dec and 2B Hex corresponds to 43 Dec.
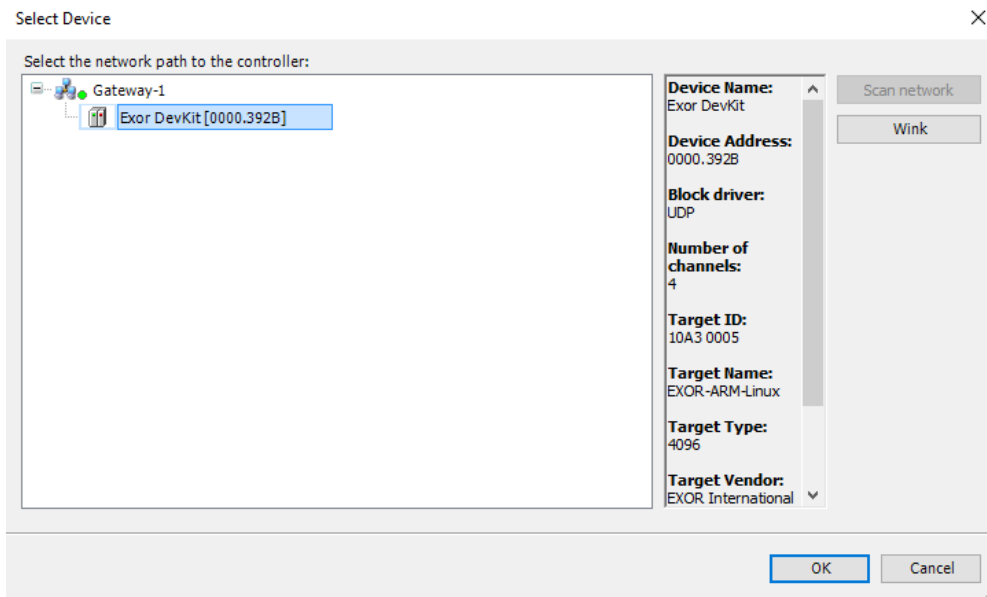
The selected device is then listed in the Communication Settings as shown below. the device properties are listed on screen. A green dot over the device graphical representation informs that the device is correctly recognized and available on the network.
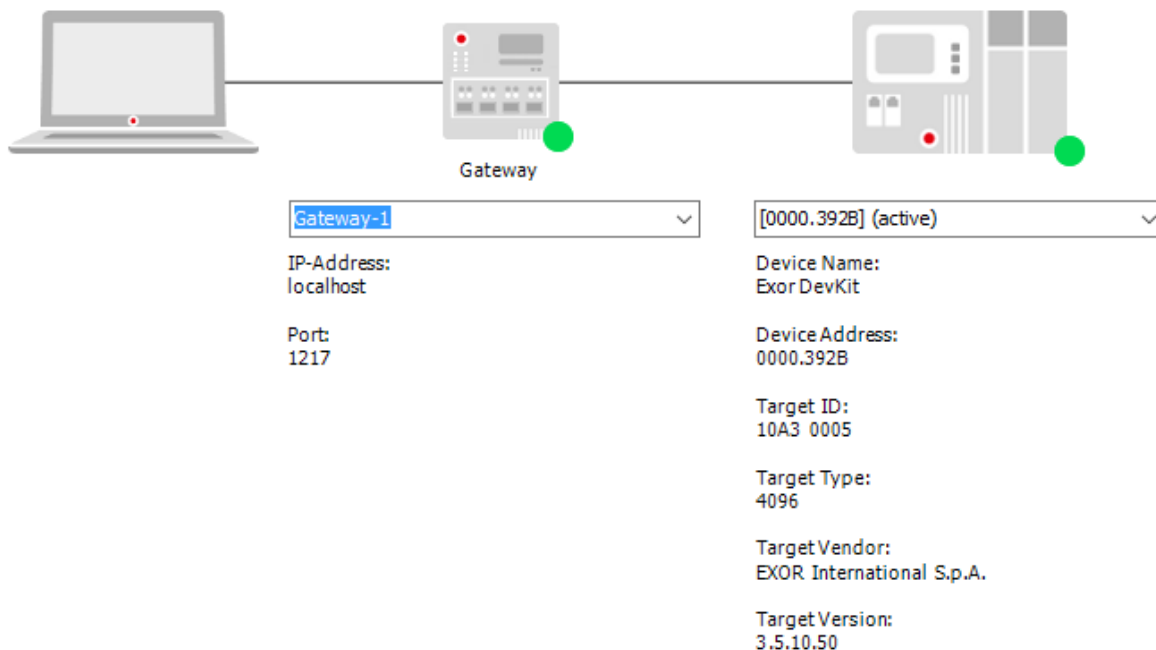
Communication with the available devices is established through a Gateway, a default Gateway is available, and it is generally not needed to change the standard Gateway settings. For more information about the Gateway set-up please refer to CODESYS V3 documentation.

11 Accessing PLC from JMobile

JMobile and CODESYS projects discussed in this section are included in the "JMobile_CDS_demo" folder from the demo projects package "Demo_Projects.zip", downloadable from exorembedded.net (Products > Development Kits).

The JMobile Portable contains also a demo version of CODESYS v3 runtime which is started along with JMobile runtime. Here is presented an example in wich JMobile will be able to access variable values from the PLC.

11.1 Codesys project creation

First of all we need to create a simple Codesys program. Assuming a configured project has already been created on CODESYS v3 and that the Development kit is properly connected we can write these few lines of code inside the PLC_PRG file:
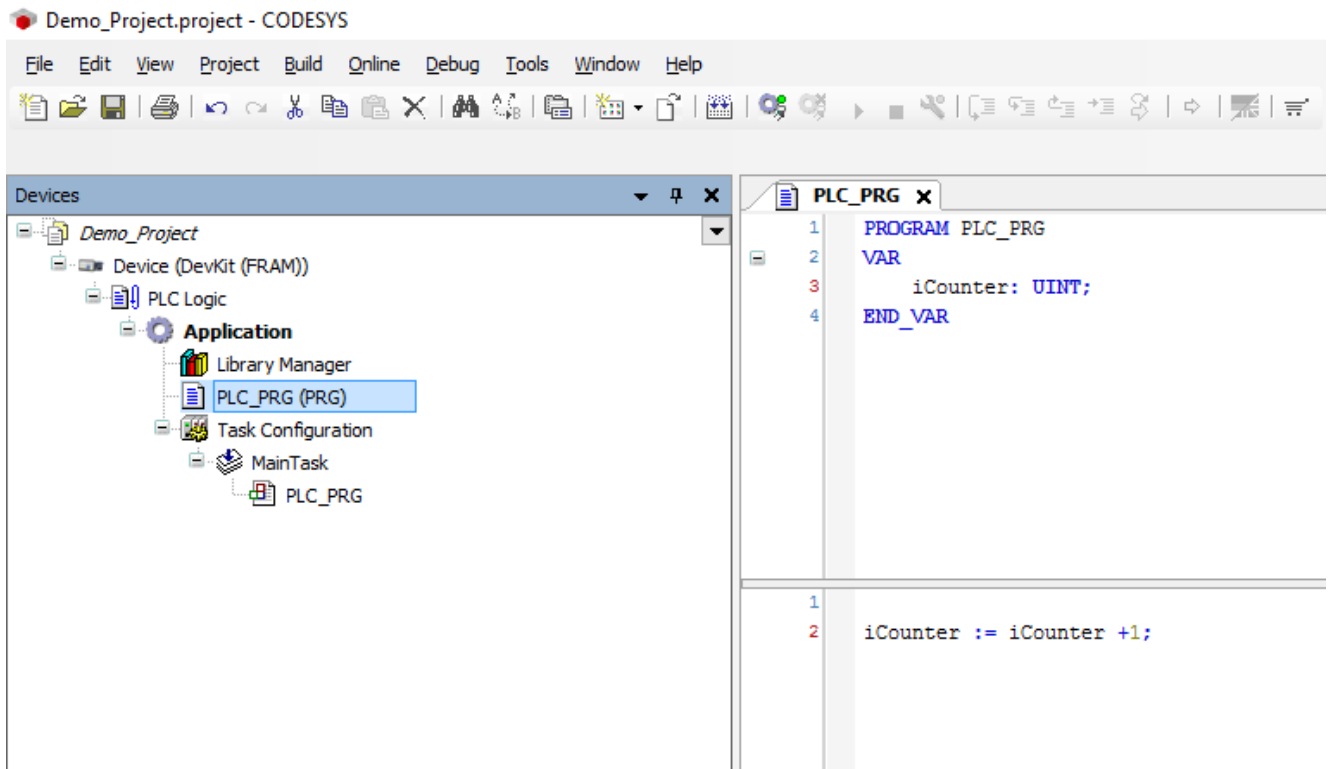
Now right click on "Application" and select "Add Object > System configuration":
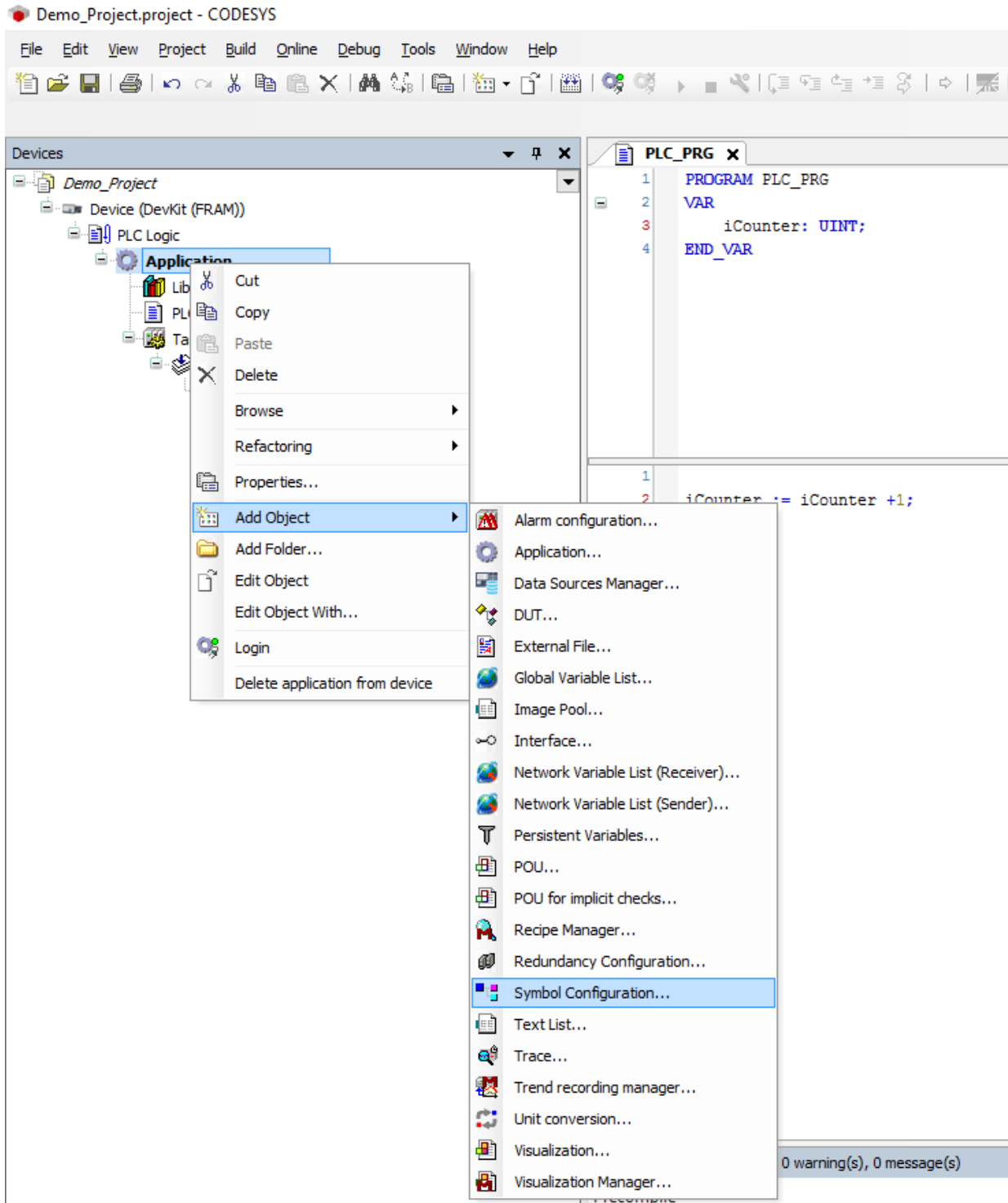
Double click on the newly created "Symbol configuration" object and from the opening tab click "Build". Make sure to check at least the PLC_PRG symbols which contains our iCounter variable:
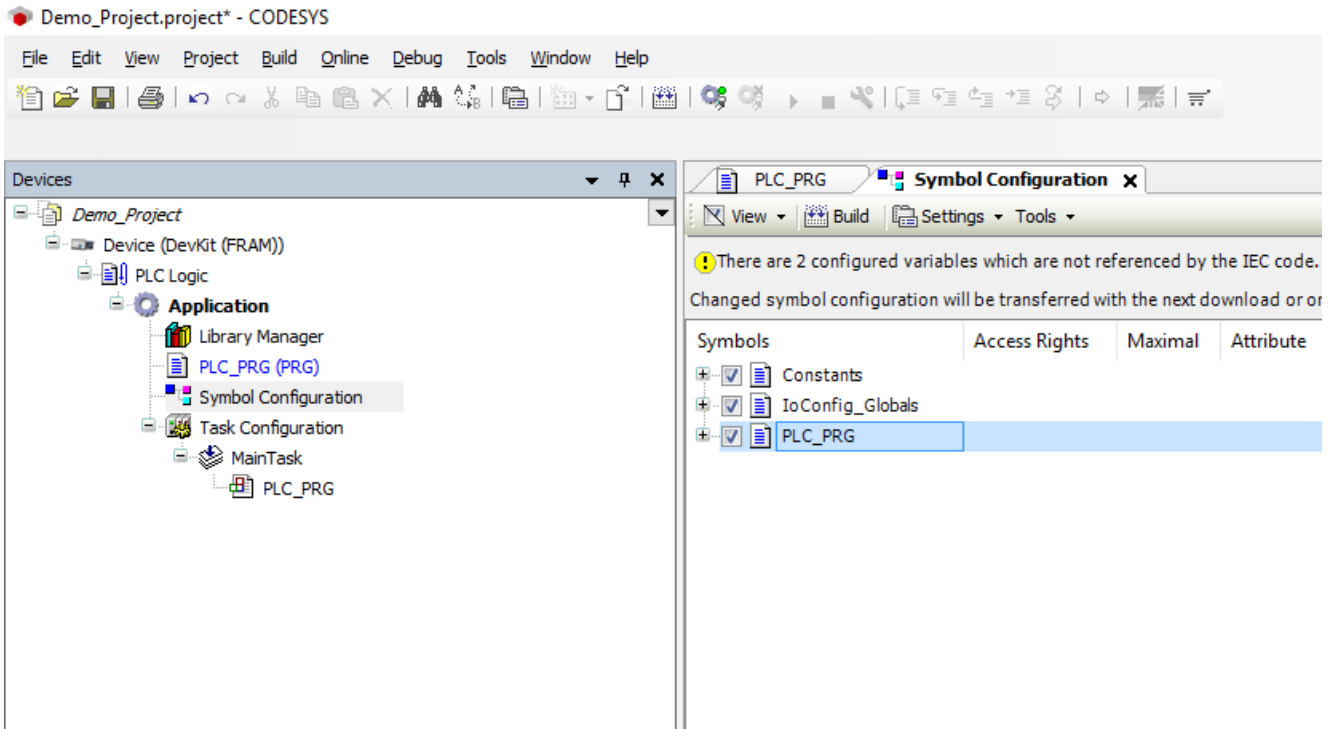
In the main toolbar click on "Build > Generate code" to create, among other files, an xml file that we will later use to import PLC variables on JMobile.

Now from the main toolbar choose "Online" -> "Login" to deploy the program on the device. If you get a warning about an existing program on the PLC click OK to overwrite it with the new one.

Lastly, to run the program on the PLC, right click on Application and choose "Start":
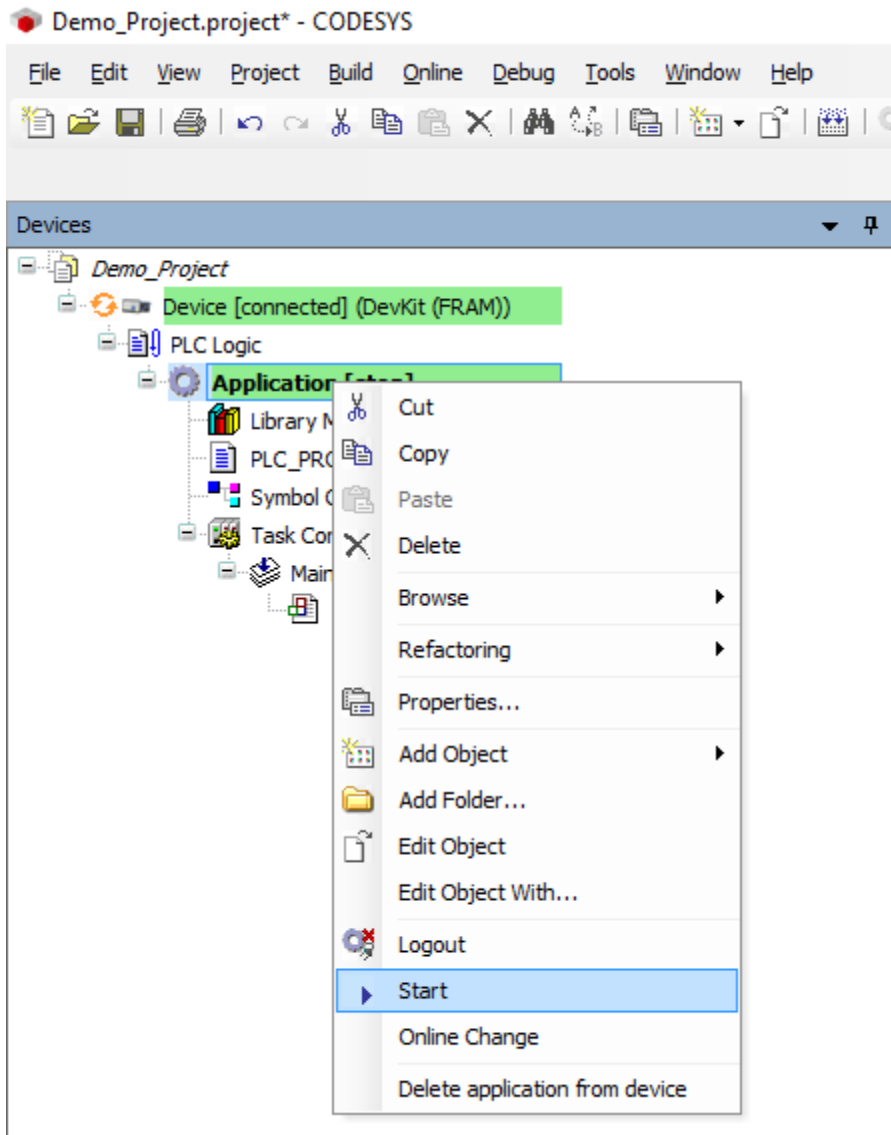
To keep PLC project on CPU also after a reboot, once the device is online, click on "Online > Create boot project".

11.2 CDS3 protocol configuration on JMobile

On the OpenHMI Studio create a project for the target Development Kit. Select "Protocols" from the Project View on the left, click on the "+" button and select "CODESYS V3 ETH".
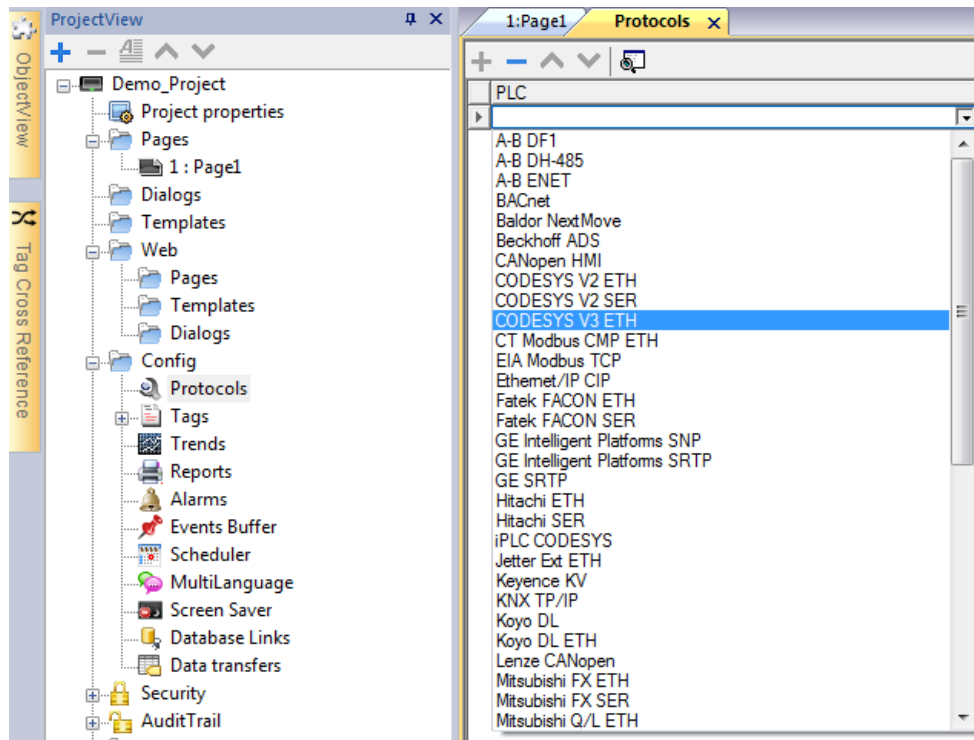
Configure the protocol as shown in the figure below, then click "OK".
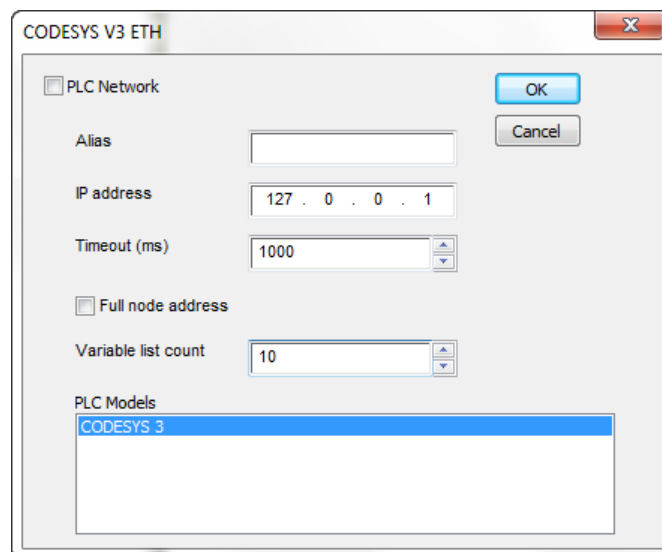

Note that with the above configuration JMobile Runtime will look for Codesys connection on the localhost. This will work only when the project is running on the Development kit. To connect remotely you can enter the Development kit network interface IP instead of 127.0.0.1.

Now, to import tags from our Codesys project, select "Tags" from the Project View. Here you have to:

1. Select "CODESYS V3 ETH:prot1" as protocol.
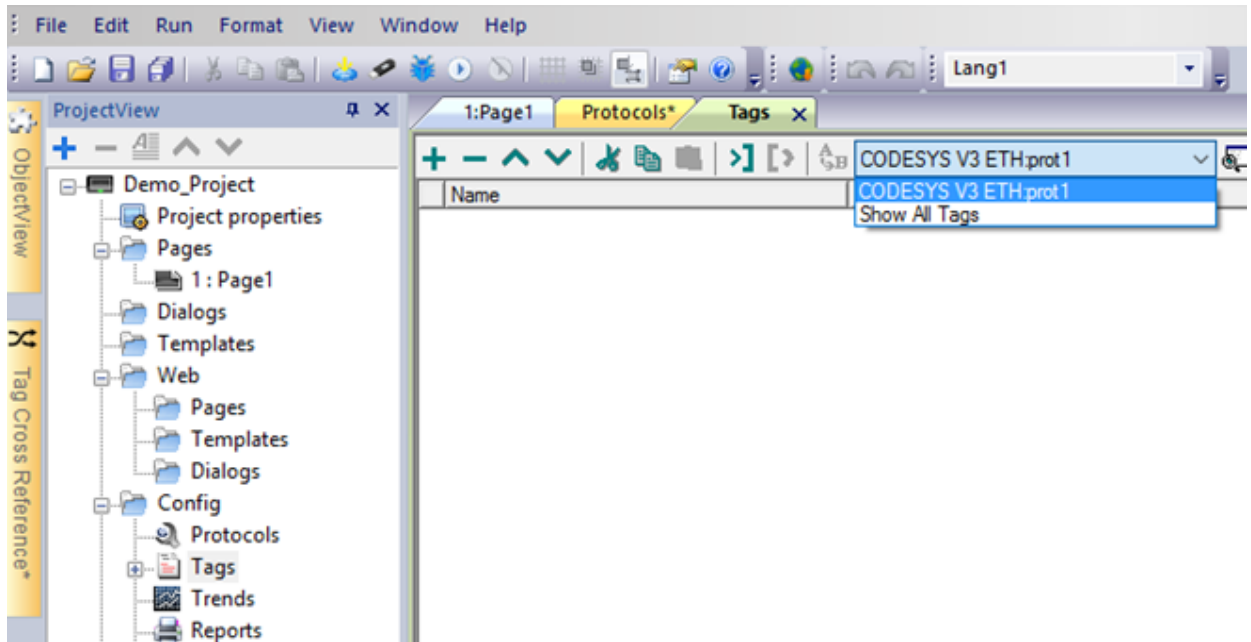
2. Click on the "Import Tags" button . Choose "CodeSys3 v1.0", "Linear" type as tag importer, click "OK" and then browse and select the xml file you will find in the Codesys project folder created before.
The name of this file should be something like "<project_name>.Device.Application.xml".

3. From the lower section of the "Tags" tab select the variable "Application/PLC_PRG/iCounter" and click on the Import Tag(s)" button .
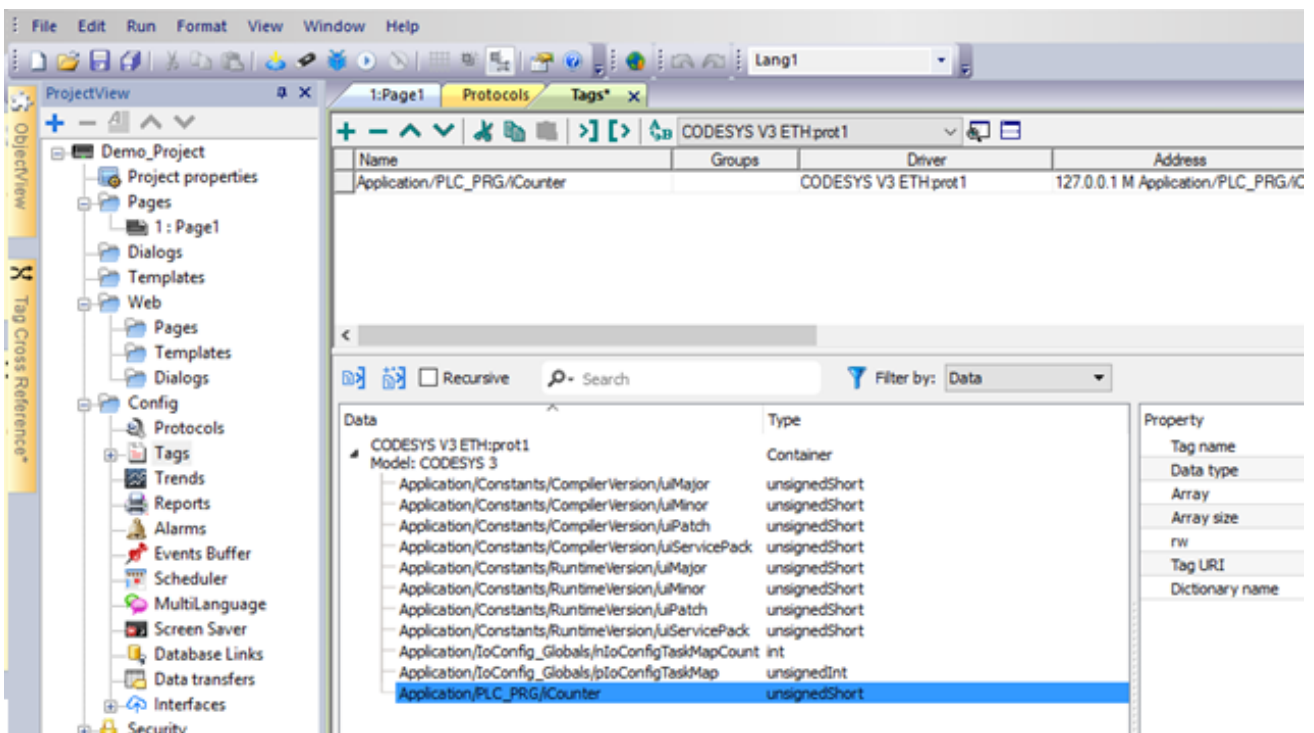
Lastly, on a project page, add a numeric label widget, right click on it and choose "Attach to..". Here select the Codesys tag and click "OK".
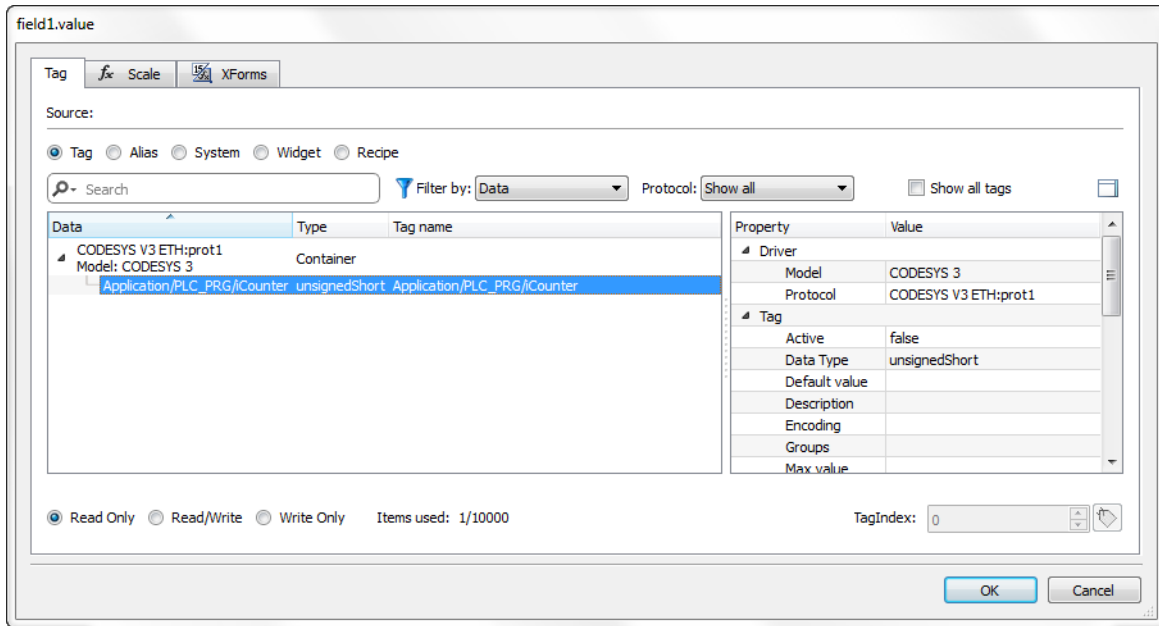
Once you have downloaded the project to the Development kit, if the PLC program is running, you should see the Codesys "iCounter" variable being incremented.