Lab „Platforms for Embedded Systems“

Chapter 01

Prof. Dr. Elmar Cochlovius

- **Goals of Chapter 01:**
  - Learn to develop Software on a Host using an Integrated Development Environment (IDE)
  - Learn to debug Software on the Target: Cross-Debugging
  - Some initial Programming Aspects of Embedded Platforms

- **Overview**
  - Recap: some important Unix-Tools
  - Setup of a Target Image
  - Installing and testing the correct cross-toolchain on the host
  - Using Eclipse as a Cross-IDE

- **Recap: some important Unix-Tools (1)**
  - **`dd:`** command to copy streams of data
    - Example 1: Copying an image to a SD-Card:

```
dd if=linuximage of=/dev/sdb
```

    - Ex. 2: Making a backup of an SD-Card:

```
dd if=/dev/sdb of=linuximage
```

  - Problem (as with **`cp`**): there is no progress indicator
  - Solution: Send signale **`USR1`** to the **`dd`**-process

`ps -auxg | grep dd`	finds the PID
`sudo kill -USR1 <PID>`	sends the signal

- **Recap: some important Unix-Tools (2)**
  - `putty` using UART
    - The USB/serial-Adapter (e.g. using the Prolific-driver) requires some setups:
      - Speed 115200
      - Data bits 8
      - Stop bits 1
      - Parity none
      - Flow control none
      - Connection Type serial
    - Note: save your putty profile (e.g. „target_serial“)

- **Recap: some important Unix-Tools (3)**
  - **`putty`** using UART (2)
    - In case of problems related to access rights („unable to open“):
    - Reason: missing user rights on the host to open a serial connections.
    - Check with:

```
ls -al /dev/tty*
```

    - Indicates missing access rights.
    - Solution: user has to be member of **`dialout`** group

- **Recap: some important Unix-Tools (4)**
  - **`putty`** using UART (3)
    - Adding user **`$USER`** to group **`dialout`**:

```
sudo adduser $USER dialout
```

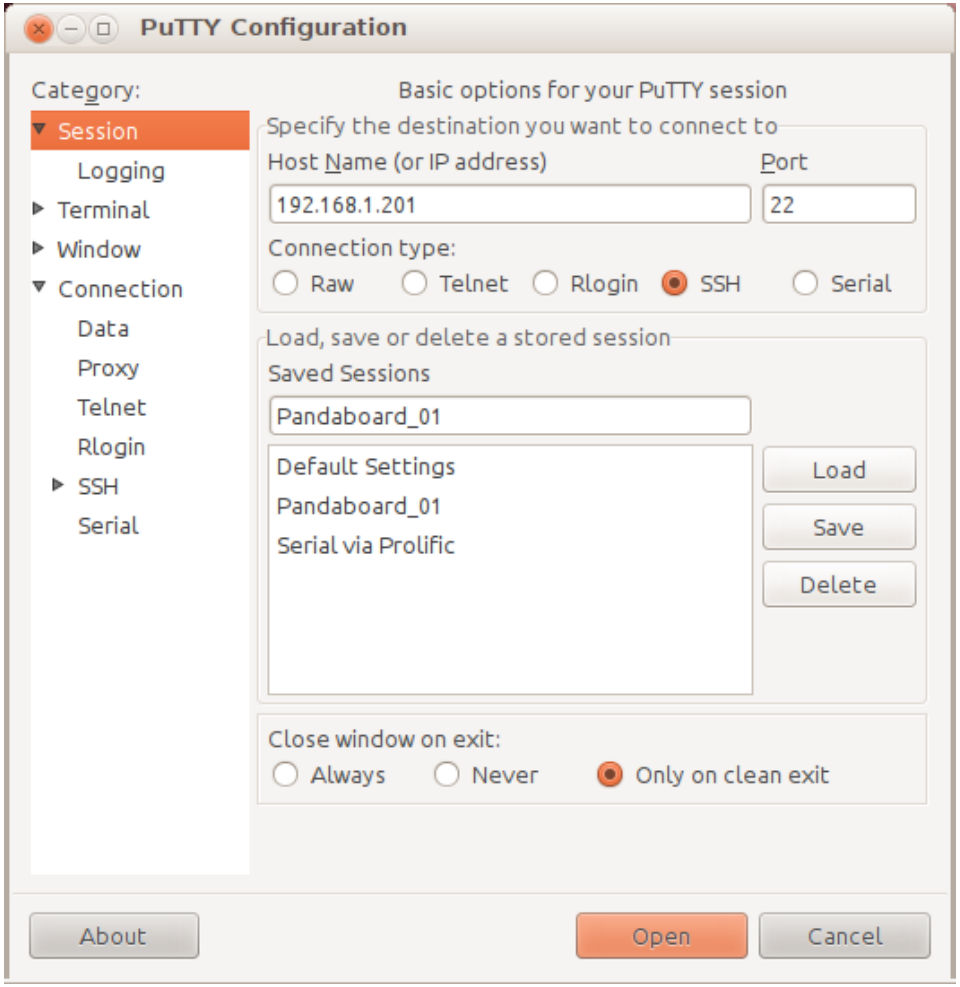
    - Requires logout / login
    - Now, **`$USER`** is member of group **`dialout`**:

```
id -Gn | grep dialout
```

    - Otherwise, this is an ugly hack:

```
gksudo putty
```

- **Recap: some important Unix-Tools (5)**
  - `putty` using IP:
    - Set IP-Adress of the target
    - Save putty-profile



- **Overview**
  - Recap: some important Unix-Tools
  - Setup of a Target Image
  - Installing and testing the correct cross-toolchain on the host
  - Using Eclipse as a Cross-IDE

- **Preparation: install Raspian image on the target (1):**
  - The version used in the Lab is „raspbian-2016-03-18“(!!), and it is available directly from: http://downloads.raspberrypi.org/raspbian/images/
  - Copy image to the SD-Card

```
unzip 2016-03-18-raspbian-jessie.zip
sudo dd bs=4M if=./2016-03-18-raspbian-jessie.img
   of=/dev/sdb ; sudo sync
```

  - Which are the potential problems?
  - How to find the right device of the SD-Card?

  - Note: the same image can be found in the download folder at https://cloud.smarthome.hs-furtwangen.de/index.php/s/q2tBtisvOmxG1UD

- **Preparation: install Raspian image on the target (2):**
  - **Check** that image has been copied correctly:
    - two partitions available on SDCard
  - **Adapt** your SDCard:
    - in `/etc/hostname` and `/etc/hosts` change the name `raspberry` to a unique target name, e.g. `target<number>` to allow network addressing
    - **Note**: starting with Jessie-Pixel (i.e. 2017-04-10), connecting with `ssh` has been disabled by default to increase security.
    - To enable `ssh`, create an empty file, named `ssh`, in the boot partition, i.e. in /boot by

```
touch /boot/ssh
```

    - Also, the default password `raspberry` for user `pi` should be changed!

- **Übersicht**
  - Recap: some important Unix-Tools
  - Setup of a Target Image
  - Installing and testing the correct cross-toolchain on the host
  - Using Eclipse as a Cross-IDE

- **What‘s up next?**
  - **Out goal:** we want to learn and practice

„Plattform-independent Software-Development“

for the Raspberry Pi.

  - Host: PC, i.e. x86 using Ubuntu (64bit),
    - Maybe virtualized, e.g. VM-WarePlayer
  - Target: Raspberry Pi, i.e. ARM-architecture using Raspian (this is a variant of Debian, optimized for BCM-chips)

  - For this, we need:
    - Get a new toolchain
    - Adapt out profile to access the toolchain
    - In case we want to develop OO, we also install g++ on the host
    - Install and setup an IDE for „serious“ SW-Development

Just for your understanding!
Not required when using the vmimage

A new Cross-Toolchain: Actions on Host

- **Background:**
  - Ubuntu already contains a Cross-Toolchain for the ARM architecture. However, this is limited (e.g. missing cross-debugger) and outdated
- Step 1: get current cross-toolchain for Raspberry:

```
gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux.tar
```

- Download e.g. from:

https://releases.linaro.org/14.09/components/toolchain/binaries/ or from:
http://webuser.hs-furtwangen.de/~coe/LabPMS/Res/Downloads/

- Non-standard packets are usually installed under `/opt`. So:

```
sudo mv gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux
/opt/crosstool
```

A new Cross-Toolchain: Actions on Host

- Step 2: Checking tools with

```
ls /opt/crosstool/gcc-linaro-...
```

```
ec@ecubuntu: /opt/crosstool/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux/bin
ec@ecubuntu:/opt/crosstool/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux/bin$ ll
total 22832
drwxr-xr-x 2 ec ec    4096 Sep 11  2014 ./
drwxr-xr-x 7 ec ec    4096 Sep 11  2014 ../
-rwxr-xr-x 1 ec ec  620332 Sep 11  2014 arm-linux-gnueabinf-addr2line*
-rwxr-xr-x 2 ec ec  644400 Sep 11  2014 arm-linux-gnueabihf-ar*
-rwxr-xr-x 2 ec ec 1085592 Sep 11  2014 arm-linux-gnueabihf-as*
lrwxrwxrwx 1 ec ec      23 Sep 11  2014 arm-linux-gnueabihf-c++ -> arm-linux-gnueabihf-g++*
-rwxr-xr-x 1 ec ec  618508 Sep 11  2014 arm-linux-gnueabihf-c++filt*
-rwxr-xr-x 1 ec ec  625128 Sep 11  2014 arm-linux-gnueabihf-cpp*
-rw-r--r-- 1 ec ec    3568 Sep 11  2014 arm-linux-gnueabihf-ct-ng.config
-rwxr-xr-x 1 ec ec 2706116 Sep 11  2014 arm-linux-gnueabihf-dwp*
-rwxr-xr-x 1 ec ec   51508 Sep 11  2014 arm-linux-gnueabihf-elfedit*
-rwxr-xr-x 1 ec ec  626792 Sep 11  2014 arm-linux-gnueabihf-g++*
lrwxrwxrwx 1 ec ec      29 Sep 11  2014 arm-linux-gnueabihf-gcc -> arm-linux-gnueabihf-gcc-4.9.2*
-rwxr-xr-x 1 ec ec  624136 Sep 11  2014 arm-linux-gnueabihf-gcc-4.9.2*
-rwxr-xr-x 1 ec ec   20716 Sep 11  2014 arm-linux-gnueabihf-gcc-ar*
-rwxr-xr-x 1 ec ec   20684 Sep 11  2014 arm-linux-gnueabihf-gcc-nm*
-rwxr-xr-x 1 ec ec   20684 Sep 11  2014 arm-linux-gnueabihf-gcc-ranlib*
-rwxr-xr-x 1 ec ec  326008 Sep 11  2014 arm-linux-gnueabihf-gcov*
-rwxr-xr-x 1 ec ec 3629880 Sep 11  2014 arm-linux-gnueabihf-gdb*
-rwxr-xr-x 1 ec ec  626696 Sep 11  2014 arm-linux-gnueabihf-gfortran*
-rwxr-xr-x 1 ec ec  681964 Sep 11  2014 arm-linux-gnueabihf-gprof*
lrwxrwxrwx 1 ec ec      26 Sep 11  2014 arm-linux-gnueabihf-ld -> arm-linux-gnueabihf-ld.bfd*
-rwxr-xr-x 3 ec ec 1063192 Sep 11  2014 arm-linux-gnueabihf-ld.bfd*
-rwxr-xr-x 1 ec ec   10501 Sep 11  2014 arm-linux-gnueabihf-ldd*
-rwxr-xr-x 2 ec ec 3755040 Sep 11  2014 arm-linux-gnueabihf-ld.gold*
-rwxr-xr-x 2 ec ec  629516 Sep 11  2014 arm-linux-gnueabihf-nm*
-rwxr-xr-x 2 ec ec  769100 Sep 11  2014 arm-linux-gnueabihf-objcopy*
-rwxr-xr-x 2 ec ec  962124 Sep 11  2014 arm-linux-gnueabihf-objdump*
-rwxr-xr-x 1 ec ec     417 Sep 11  2014 arm-linux-gnueabihf-pkg-config*
-rwxr-xr-x 1 ec ec   90348 Sep 11  2014 arm-linux-gnueabihf-pkg-config-real*
-rwxr-xr-x 2 ec ec  644400 Sep 11  2014 arm-linux-gnueabihf-ranlib*
-rwxr-xr-x 1 ec ec  432444 Sep 11  2014 arm-linux-gnueabihf-readelf*
-rwxr-xr-x 1 ec ec  620940 Sep 11  2014 arm-linux-gnueabihf-size*
-rwxr-xr-x 1 ec ec  620524 Sep 11  2014 arm-linux-gnueabihf-strings*
-rwxr-xr-x 2 ec ec  769100 Sep 11  2014 arm-linux-gnueabihf-strip*
ec@ecubuntu:/opt/crosstool/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux/bin$
```

Just for your understanding!
Not required when using the vmimage

- **A new Cross-Toolchain: Actions on Host**
  - Step 3: But how can the shell find these new binaries?
    - We need to adapt the **$PATH**-variable in **~/.profile**

```
#EC: set PATH so it includes linaro g++4.9 crosstools if it \
exists
if [ -d "/opt/crosstool/gcc-linaro-arm-linux-gnueabihf-4.9-\
2014.09_linux/bin" ] ; then
    PATH="/opt/crosstool/gcc-linaro-arm-linux-gnueabihf-4.9-\
2014.09_linux/bin:$PATH"
fi
```

- **Note 1: NO(!)** newlines in the path names („\“ only for printing)
- **Note 2:** carefully observe spaces and semicolons

- **A new Cross-Toolchain: Actions on Host**
  - Step 4: Check **`~/.bashrc`** and export variables required for cross-compilation:

```
# set env Variables for cross-compiling
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-
export CC=${CROSS_COMPILE}gcc
```

- **A new Cross-Toolchain: Actions on Target**
  - For Cross-Debugging, we later need a debugging-server on the target
  - The debugging server connects the ARM-Debugger running on the host (!) with the process to debug running on the target
  - If not yet installed, we install on the target:

```
sudo apt-get install gdbserver
```

  - Also, we install on the target the ssh-server and the nfs-server:

```
sudo apt-get install openssh-server \
      nfs-kernel server
```

- **Target- and Host-Environment**
  - Now it is time for:
    - **Exercise 01 –**

      Install and Setup of the Target- and Host-Environment

- **Overview**
  - Recap: some important Unix-Tools
  - Setup of a Target Image
  - Installing and testing the correct cross-toolchain on the host
  - Using Eclipse as a Cross-IDE

Just for your understanding!
Not required when using the vmimage

- **Background and Overview:**
  - Manual compilation and „pushing over to target“ is a not feasible in industrial SW development projects → lame!!
  - **More efficient Approach:**
    - We benefit from a modern IDE (Integrated Development Environment), which also allows cross-development and debugging.
    - Here we use Eclipse
  - Step 1: Installation of Eclipse Luna, SR2
  - Step 2: Learn some basic concepts and mechanisms for:
    - Developing, compiling and execution of programs
    - Important constraint: single-source approach
  - Step 3: Then, we want to engage in cross-debugging (a.k.a. remote-debugging) of programs

Just for your understanding!
Not required when using the vmimage

Eclipse: Installation

- **Goal:**
  This chapter is focussed on Eclipse, i.e. how can we use this powerful IDE for target- or cross-development and –debugging.
- Usually, the Version, delivered with Ubuntu is limited.
- If already installed, then we remove Eclipse with:

```
sudo apt-get remove eclipse
```

- Now we install Version Luna SR2 (CDT) **manually**.
- Additional hints (also applicable for Luna) are available here:
  http://akovid.blogspot.de/2012/08/installing-eclipse-juno-42-in-ubuntu.html
- The binary we need, is called:
  `eclipse-cpp-luna-SR2-linux-gtk-x86_64.tar.gz`
- You find it here: **`http://www.eclipse.org/downloads/`**
  or here: **`http://webuser.hs-furtwangen.de/~coe/LabPMS/Res/Downloads/`**

Just for your understanding!
Not required when using the vmimage

Eclipse: Installation

- Now we unpack the binary and move it into the **/opt** directory. For this, we use the Unix packet tool **tar**:

```
tar xvzf ./eclipse-cpp-luna-SR2-linux-gtk-\
        x86_64.tar.gz
```

- This creates a directory **./eclipse,** which we move using:

```
sudo mv ./eclipse /opt
```

  into **/opt**

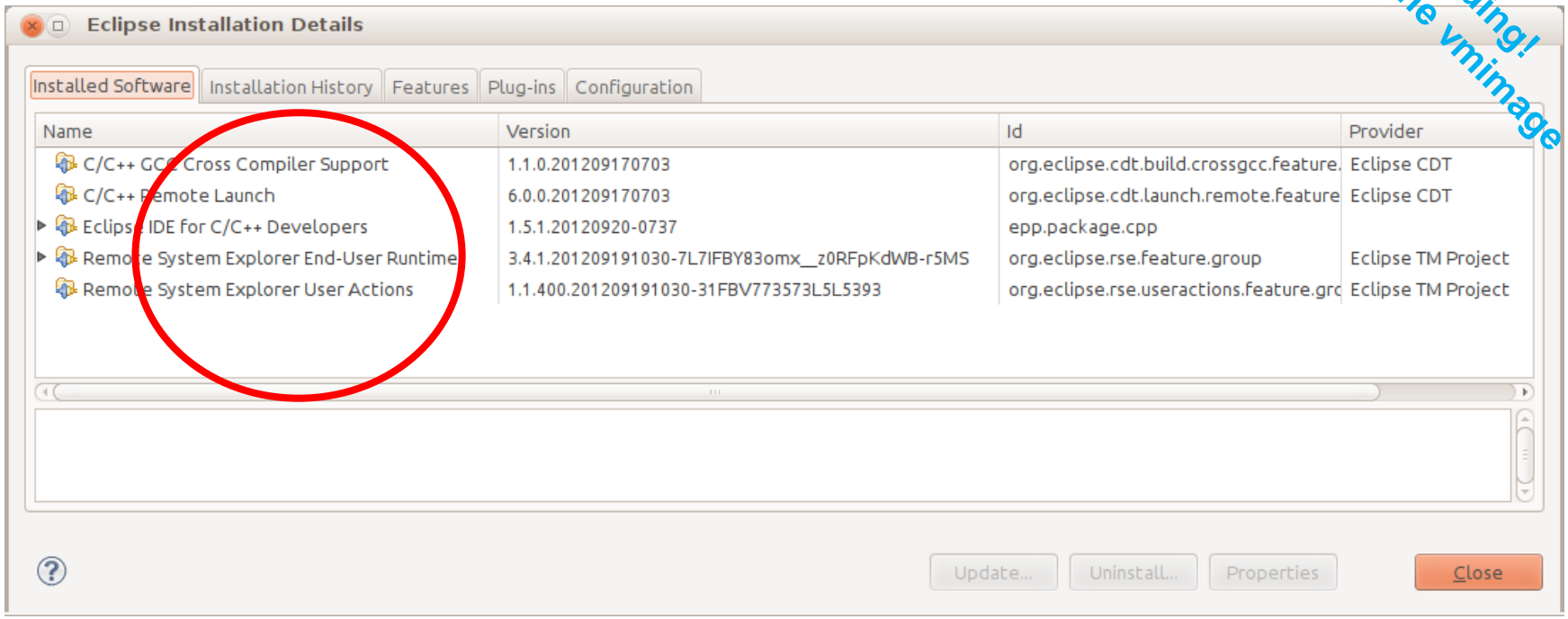
- Eclipse is based on Java. If required, we need to install a current version of the Java environment using:

```
sudo apt-get install default-jre
```

- With **/opt/eclipse/eclipse** we fire up the IDE and enter the „workbench“

Just for your understanding!
Not required when using the vmimage

- **Eclipse as a Cross-IDE**
  - Our installation of Eclipse (Luna 4.4.2) should contain these „features“ (check in → Installation Details):



- **Eclipse as a Cross-IDE**
  - First we test Eclipse using a Hello-World example
  - In general, there a two kinds of projects in Eclipse:
    - Managed-Make projects and
    - Makefile projects
  - **1) Managed-Make Projects:**
    - Eclipse is in full control of creating, managing and adapting the Makefile(s)
    - This is suitable for „standardized“ applications
    - There are issues with „large“ projects having several / many separate executables, i.e. several main()-functions. Eclipse cannot easily manage and correlate these
    - But for „simple“ projects, managed-make projects are very comfortable. We will start with manged-make projects first.

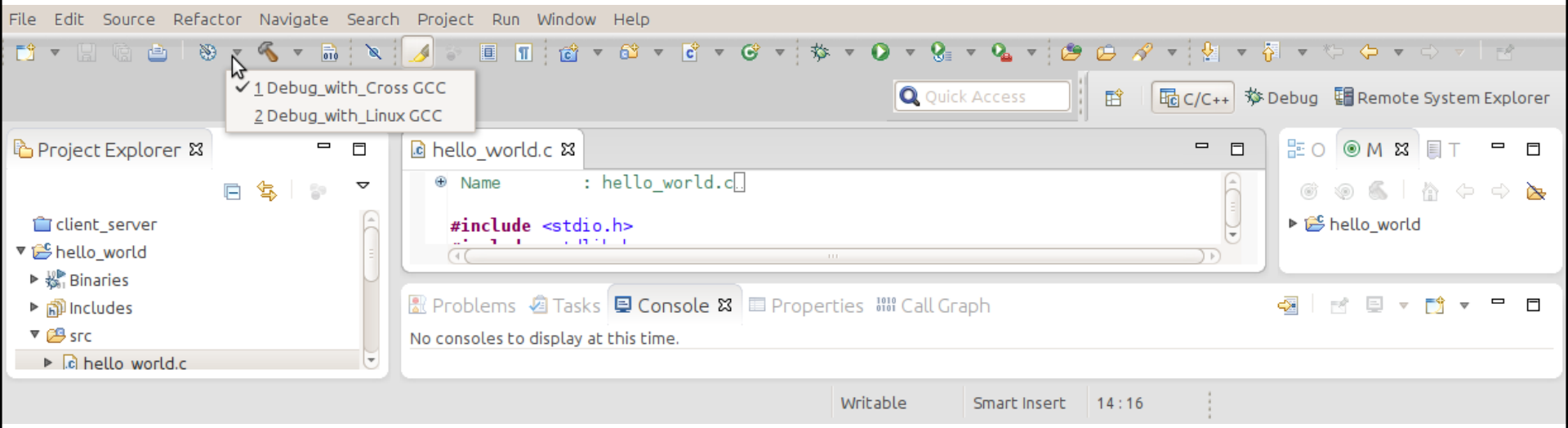
- **Eclipse as a Cross-IDE**
  - **2) Makefile Projects:**
    - The developer is in charge of „his“ Makefile.
    - Eclipse utilizes the makefile to build individual „make-targets“, but does not manage / extend the Makefile
    - This is used to import existing projects into Eclipse
    - Large projects usually bring their own Makefile(s), which should be used
    - Later, we will also work w/ Makefile projects

- **Eclipse as a Cross-IDE**
  - **Fundamental Concepts of the Eclipse-IDE:**
    - **Plattform-Configurations**:
      - what platform are we dealing with?
    - **Build-Configurations:**
      - how do we want to build the project?
    - **Run-Configurations**:
      - how is the project being executed?
    - **Debug-Configurations:**
      - how can we debug the project?
    - **Target-Configurations:**
      - how can we connect to the remote target?

- **Eclipse as a Cross-IDE**
  - **Managed-Make Project„hello_world“ on the Host**
  - Short Overview:
  - Create a Managed-Make projects „hello world“
    - Add. Information can be found here: http://www.lvr.com/eclipse3.htm
  - This is done with:
    - File → New → Project
    - C/C++ → C-Project
    - → Hello World ANSI C Project
    - Toolchains: Select Cross GCC AND Linux GCC
    - Configurations: Only(!) select Debug_with_Cross_GCC AND Debug_with_Linux_GCC
    - Set Cross compiler prefix to (exactly!) **`arm-linux-gnueabihf-`**
  - This will create a ready-to-go Hello-World example in C
  - Build the project (native for x86)
  - Execute the project by creating and starting a run-configuration

Eclipse as a Cross-IDE

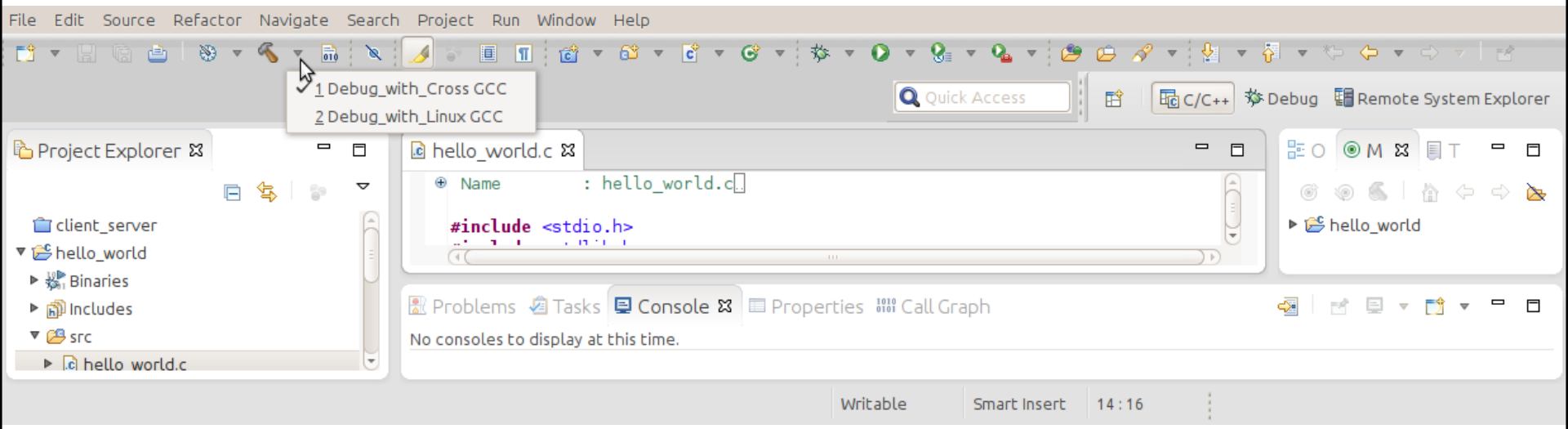
- **Details:** For this, we need the following Eclipse concepts:
  - **1) Plattform-Configurations:**



these are defined during creation of the project.

  - **Here:** we need Linux-GCC (native on x86) and Cross-GCC (used for the target, i.e. ARM-architecture)
    - The currently active plattform configuration determines the toolchain to be used

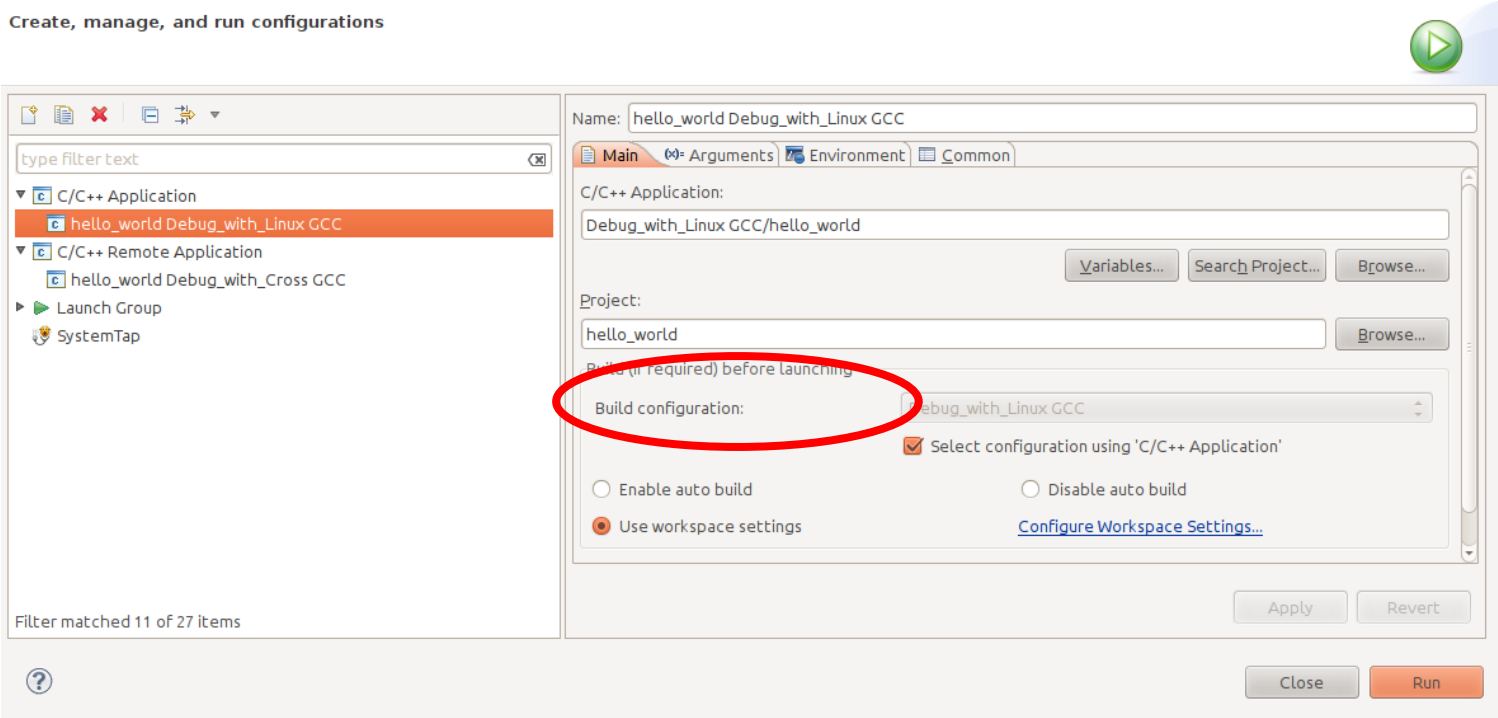
- **Eclipse as a Cross-IDE**
  - **Details:** For this, we need the following Eclipse concepts:
    - **2) Build-Configurations**



- Used to „build“ the project, based on the current platform configuration
- Tries to execute the rules for the makefile-target **`all`** in the makefile managed by Eclipse

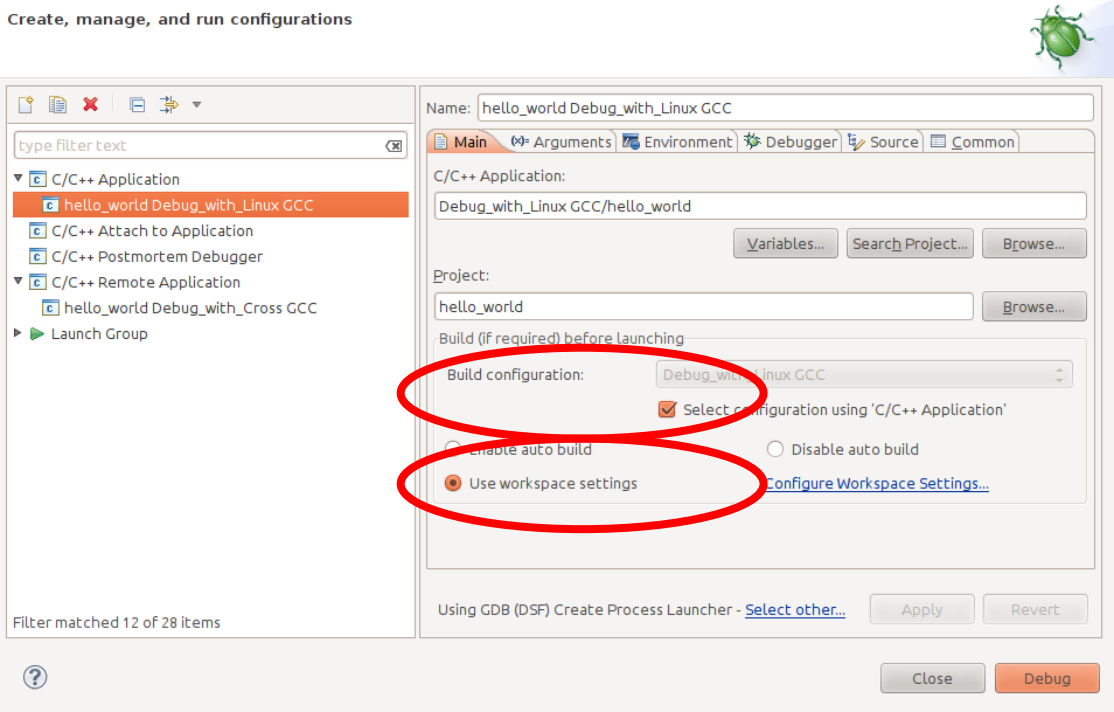
Eclipse as a Cross-IDE

- **Details:** For this, we need the following Eclipse concepts:
  - **3) Run-Configurations**



With Run → RunConfiguration we define, how to execute the binaries, which have been created during the (successful) build process („running“)

- **Eclipse as a Cross-IDE**
  - **Details:** For this, we need the following Eclipse concepts:
    - **4) Debug-Configurations**



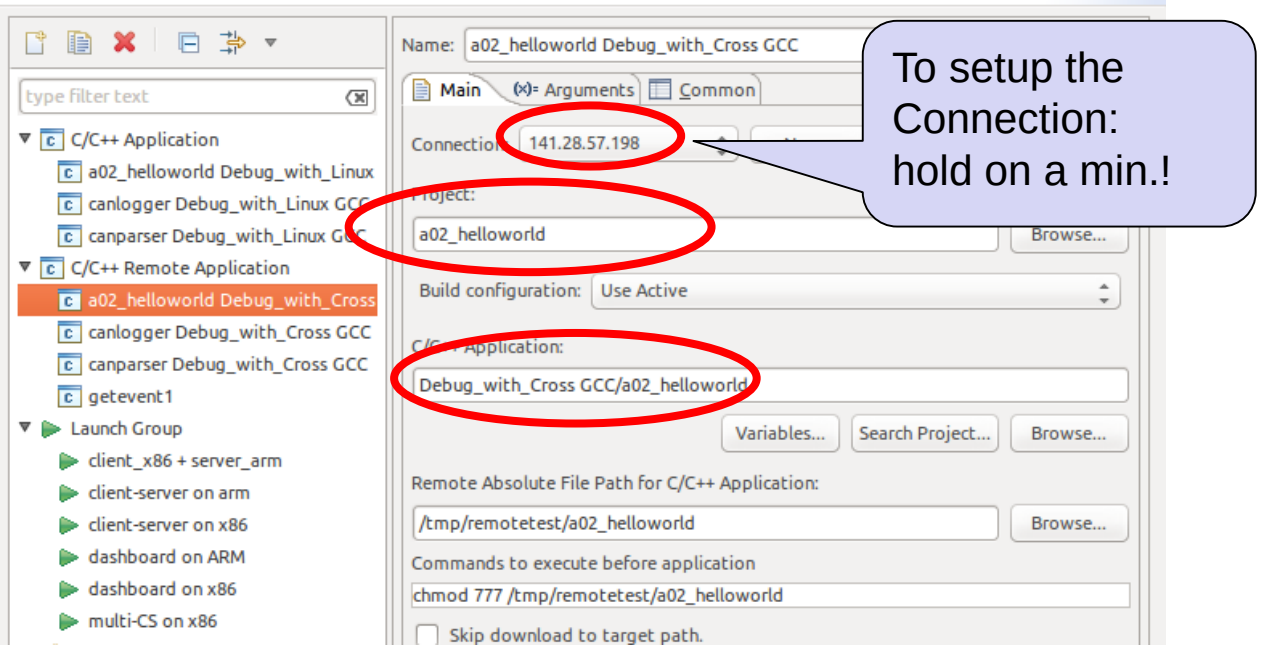
With Debug → DebugConfiguration we define the Debug environment („Debugging“). A Debug Config is an extension of a RunConfig.

- **Eclipse as a Cross-IDE**
  - Now we are ready for:
    - **Exercise 02 –**

      The Eclipse-IDE

Eclipse as a Cross-ID

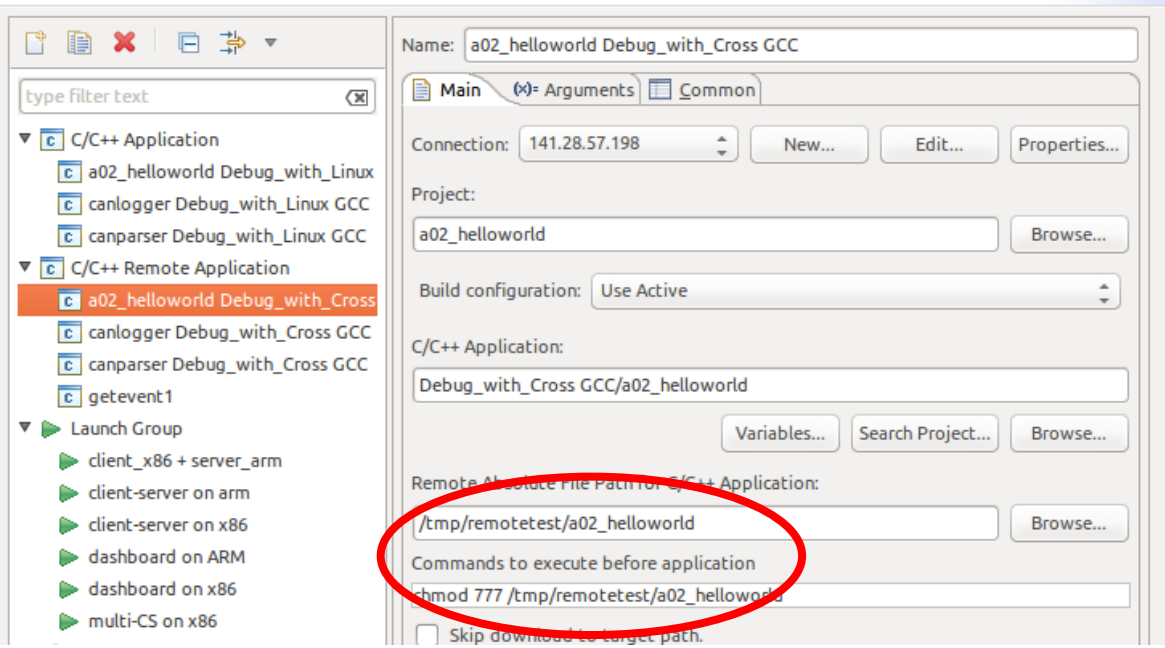
- In exercise 1 we have already done some manual **cross-compilation**. Now, we setup Eclipse to do the cross-compilation for us:
- Select Platform-Config Debug_with_CrossGCC,
- Then build the project
- Create a new Run-Config of Type „C/C++-Remote Application“ and check the default parameters proposed by Eclipse:



- **Eclipse as a Cross-ID**
  - To complete the Remote Run-Config, we need to understand the following details:
  - A) Where on the target should our program be exeuted?
    - Proposal: always use /tmp/remotetest/
    - Benefit: always „clean“ after target reboot
    - But: it has to be created manually after each reboot
  - B) What needs to be done on the target bevor execution can start?
    - Set the permissions of the transferred file to „executable“
    - This is done by:

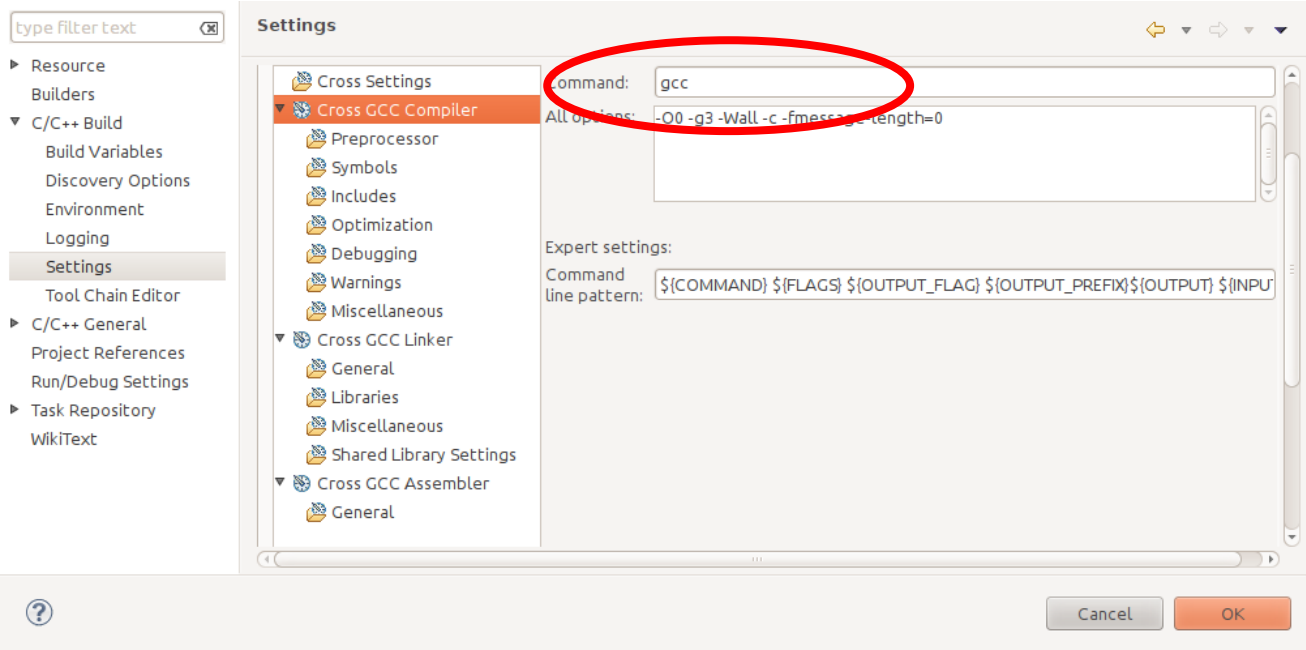
```
chmod 777 <filename inkl. vollst. Pfad>
```

- **Eclipse as a Cross-ID**
  - This information has to be filled into the Cross-Run-Config:
    - Location on the target where to put the executable
    - „Pre-Exec actions“ to be executed on the target BEFORE our executable is started
    - **Note:** here we need absolute path- and filenames



- **Eclipse as a Cross-ID**
  - Next, define a **Connection** to enable Eclipse to talk to our target:
    - Open the „perspective“ Remote System Explorer
    - In File → New → Other → Remote System Explorer → Connection → Next we setup a new Connection
  - The connection should be based on the following parameters:
    1. we want to talk to a target based on Linxux
    2. We use the IP-Address of the target (instead of its hostname), otherwise the cross-debugger will not work correctly
    3. Files should be transferred via ssh
    4. Processes should be standard Linux processes
    5. SSH is also used for shells on the target
    6. And also for terminals

- **Eclipse as a Cross-ID**
  - We make sure, that Eclipse really has selected the correct cross-tools under:
  - Projekt-View → Properties → C/C++Build → Settings shows:



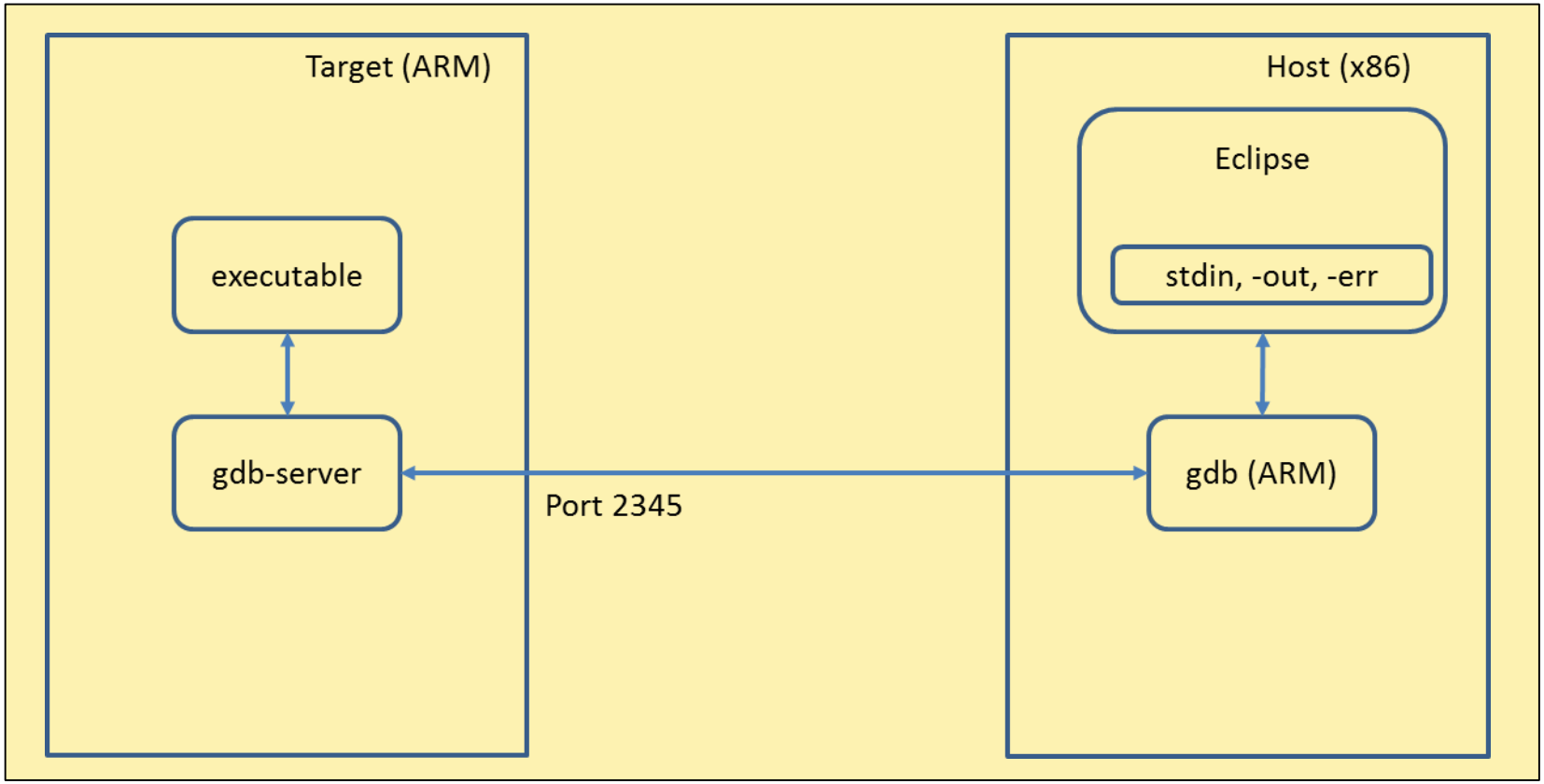
  - **Note:** the cross-prefix **`arm-linux-gnueabihf-`** is automatically added (**however**, this is not avail. for the cross-debugger!)

- **Eclipse as a Cross-ID**
  - Now, we are ready for our first cross-build and cross-execution:
  - When building:
    - We check the console output for compiler / linker errors
    - Doublecheck: the resulting executable is found in **`workspace/hello_world/Debug_with_Cross\ GCC`**
    - Using command:

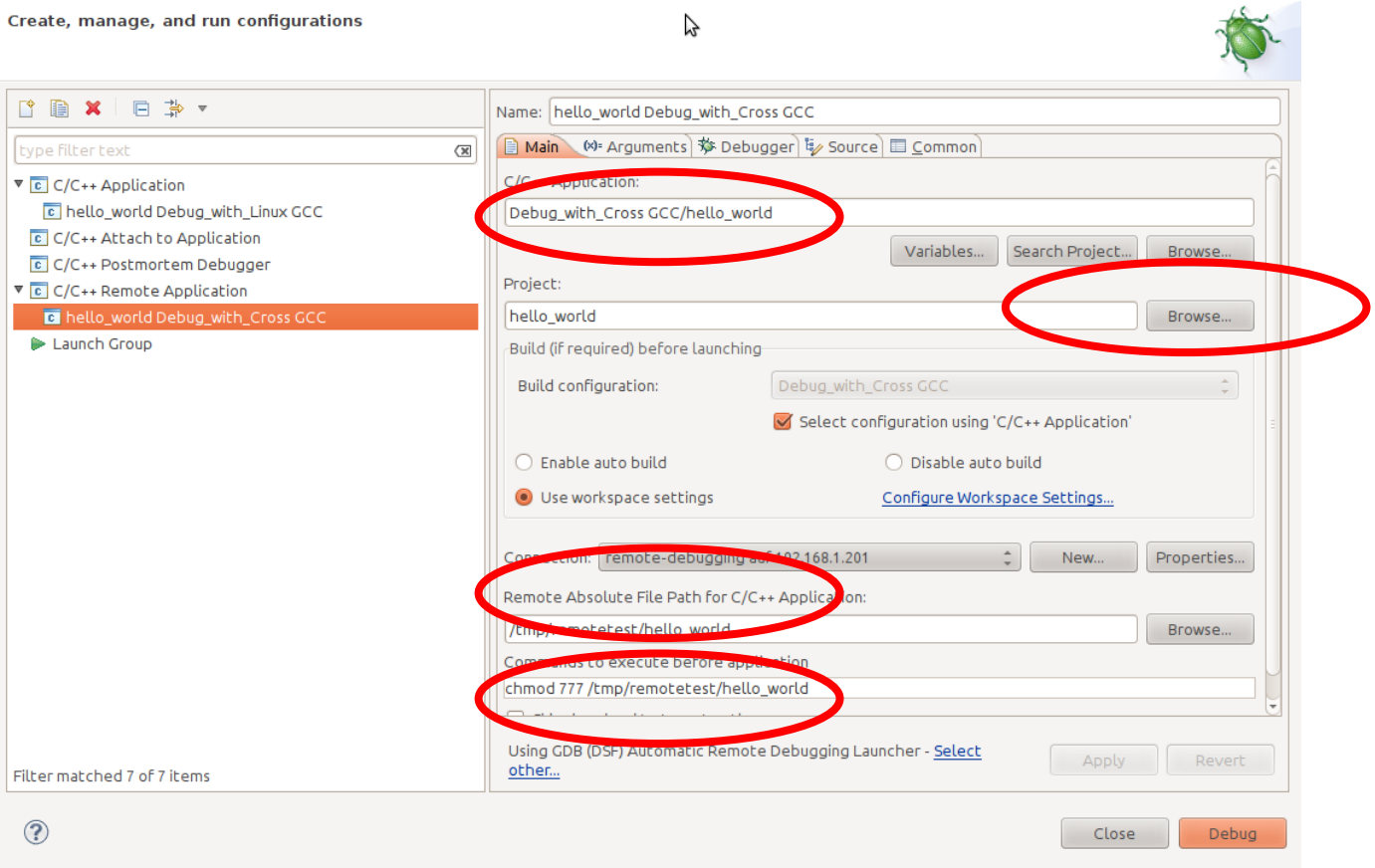
```
file
```

    - We check, that it is indeed an ARM executable
  - We can copy the executable **manually** to the target (**`scp hello_world pi@myraspi:/home/pi`**) and run it
  - Now we let Eclipse do it for us:
    - In Run → Run-Configs we select our Run-Config for Remote Application and start it („Run“)
    - The Eclipse-Console indicates how the connection to the target is established and that the executable is started on the target

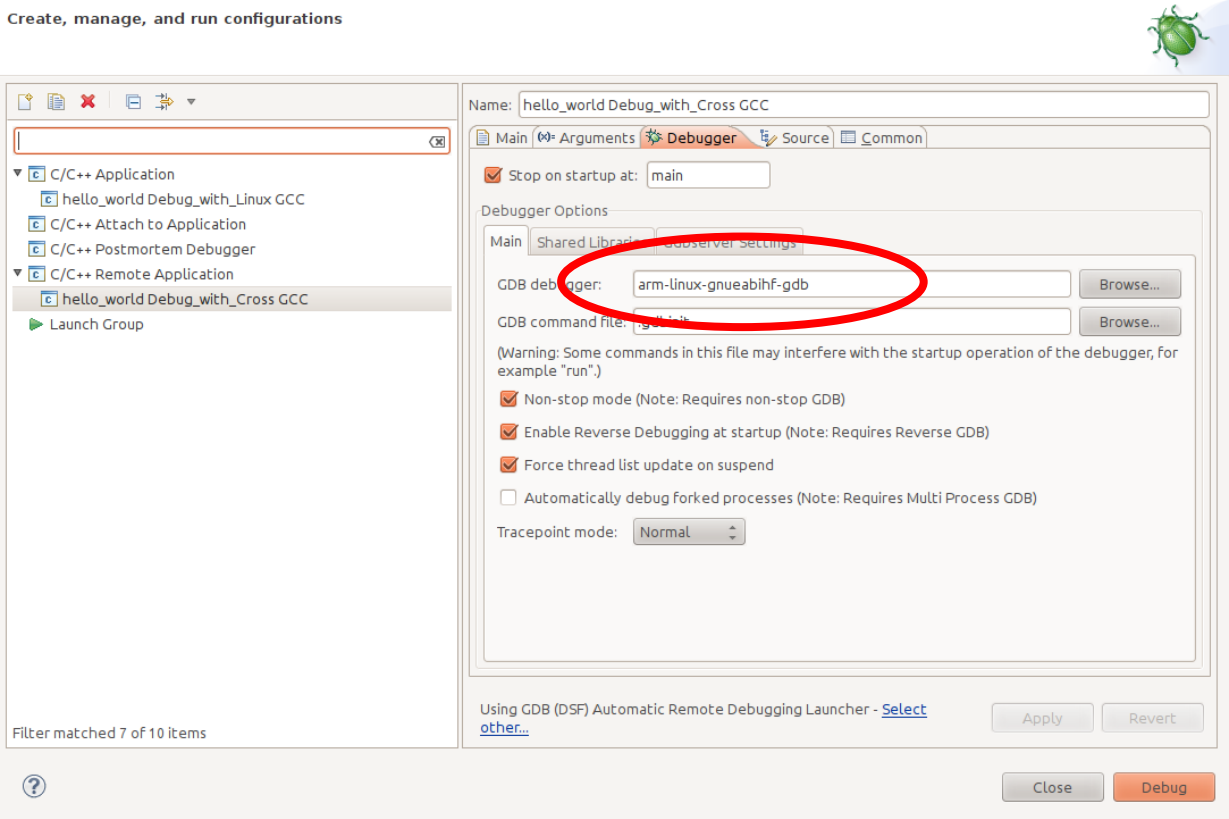
- **Eclipse as a Cross-ID**
  - Next challenge: after Remote-Execution we now want to do Remote-Debugging!
  - Here we need (the IDE) to accomplish the following setup:



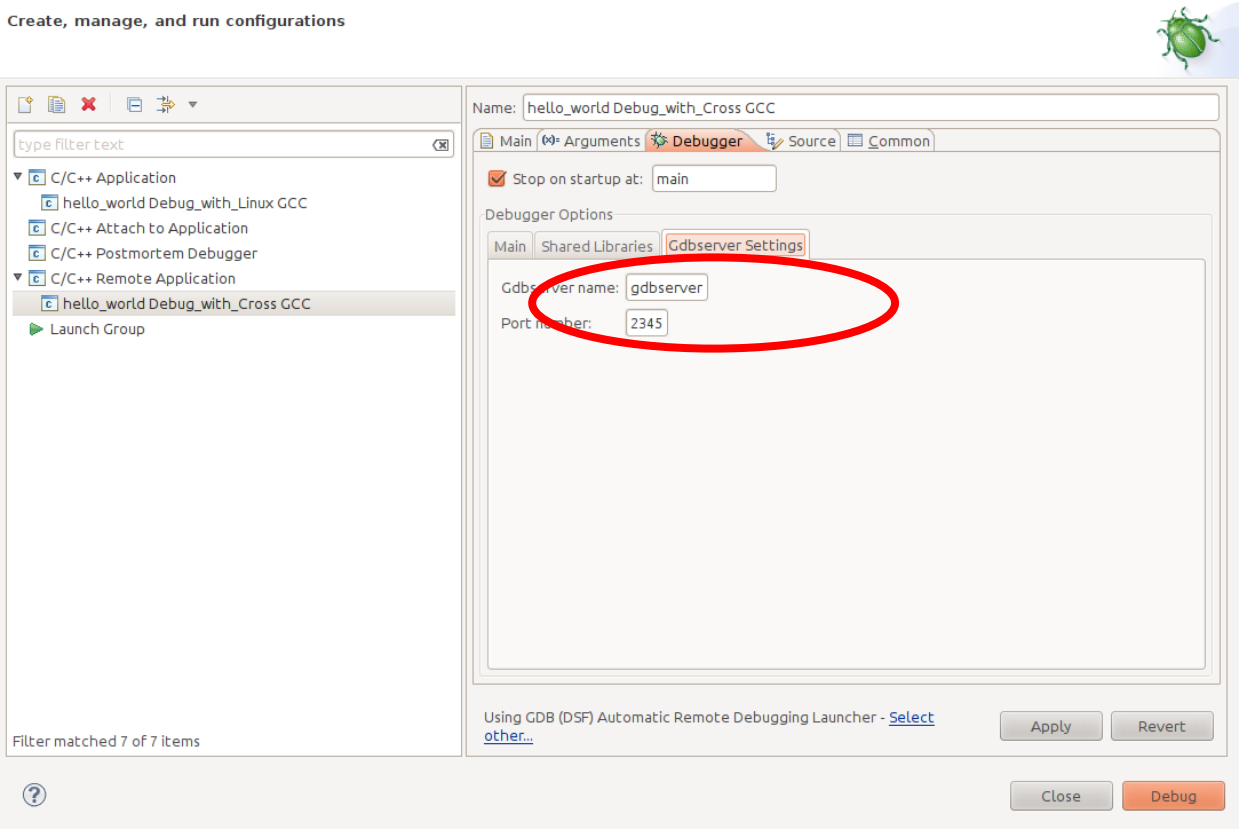
- **Eclipse as a Cross-ID**
  - Setup a Debug-Configuration for Remote-Debugging
  - Note: this extends our Remote-Run-Config



- **Eclipse as a Cross-ID**
  - One more thing to note:
    - Specify the **full name** of the cross-debugger
    - Why?



- **Eclipse as a Cross-ID**
  - Check whether the gdbserver is called correctly (usuall, default entries are o.k.)
  - Default port number is 2345



- **Eclipse as a Cross-ID**
  - Now we are ready for:
    - **Exercise 03 –**

      Remote-Debugging in Eclipse

- **Short Summary: Learning outcomes**
  - Up to now we know how:
    - To setup the target with a Debian/Linux environment and how to configure it
    - To setup the host using a cross-toolchain including remote debugger
    - To install and configure the Eclipse-IDE for cross-development
    - To create Managed-Make project, to develop code both for the host AND for the target
    - To debug the native code for the host in Eclipse
    - To remotely execute the code on the target in Eclipse and how to debug it remotely
  - **Next:**
    - Some small Eclipse-projects to understand selected aspects of embedded programming for suitable platforms