Introduction

Install LPCXpresso IDE

Our development board is an LPCXpresso 1549 from NXP and we use MCUXpresso development tools for development and debugging. MCUXpresso features an Eclipse based IDE with GCC toolchain and support for integrated debug probe found on the development board.

Go to http://nxp.com/mcuxpresso/ide and download and install MCUXpresso. During installation allow all suggested NXP drivers to be installed.

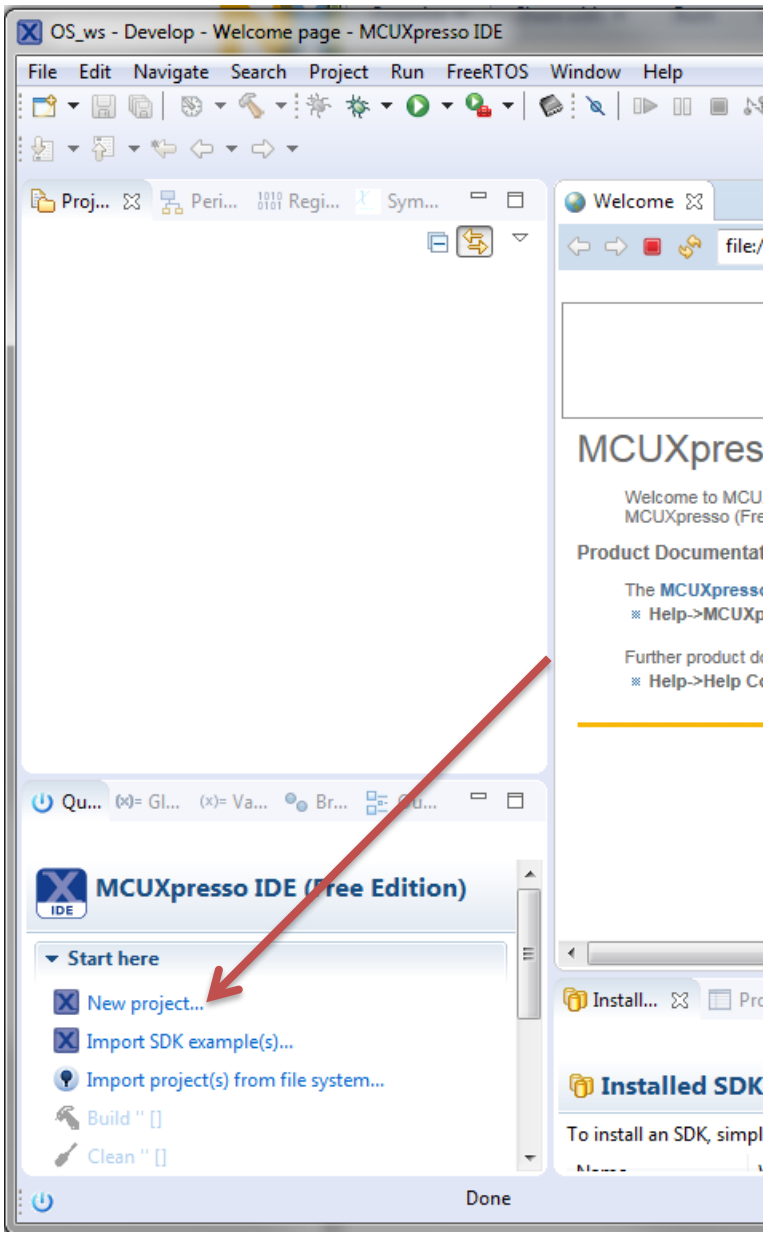
MCUXpresso comes with a vast set of libraries and example code for different platforms. The examples and libraries are installed with the IDE. The files go into
<your installation directory>\MCUXpressoIDE_10.2.1_795\ide\Examples

Create a new workspace and import driver libraries

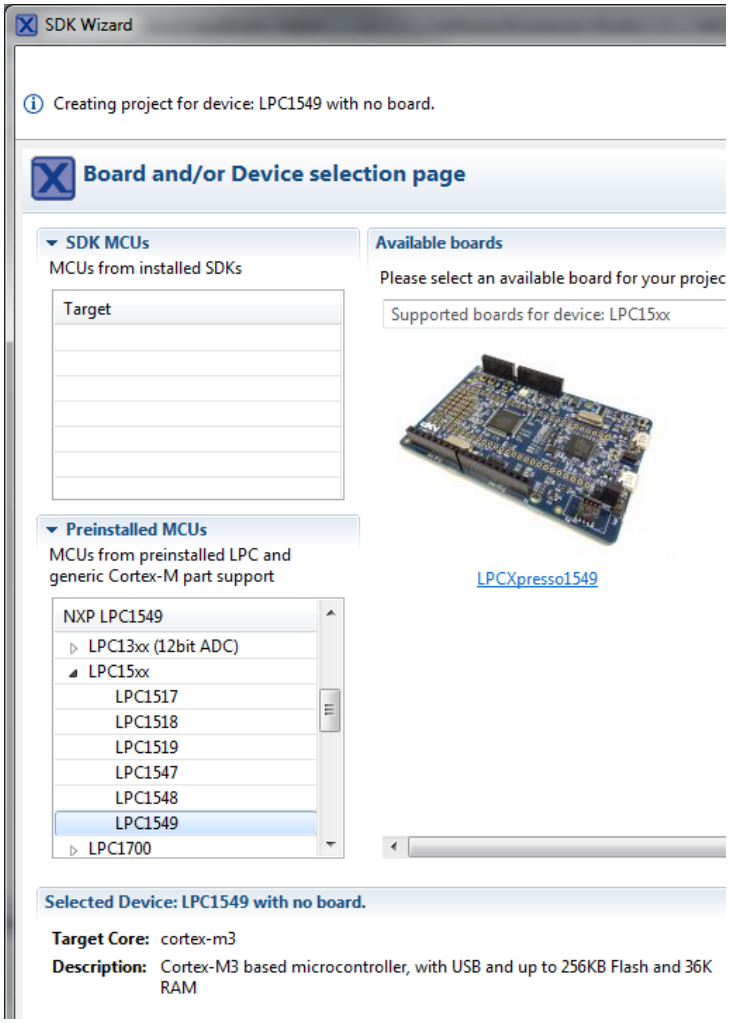
The IDE installation includes LPCOpen software platform that contains peripheral drivers and sample code. Each processor family requires different drivers and it is up to the developer to select and import driver packages to be used.

When you start the IDE you will be prompted to select a workspace. If you select a folder that doesn't have a workspace an empty workspace is created in the directory. A workspace typically contains multiple projects: peripheral driver library projects and your own project(s) that make a reference to (i.e. use) the driver projects.

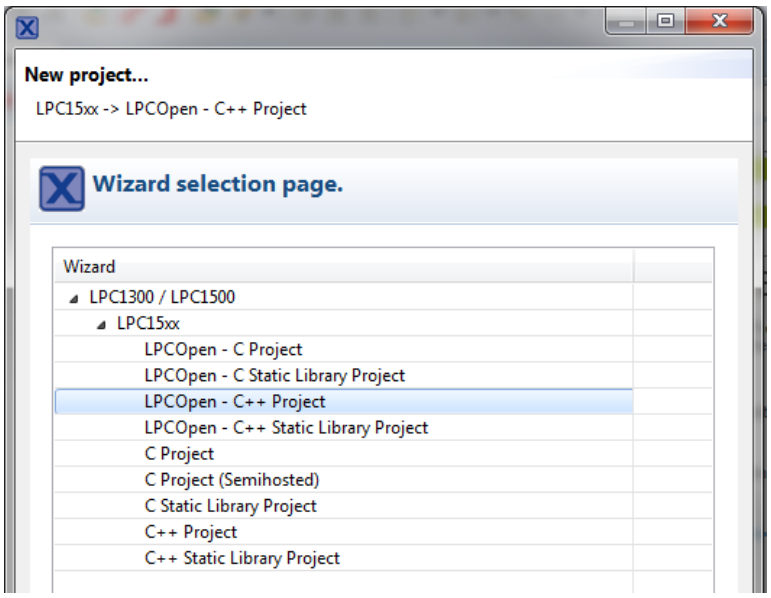
The easiest way to import drivers and to create a new project is to use the project wizard. The project wizard is launched from the quick start window. See screen shot below. The wizard will ask you to import driver libraries when you create the project. However first you need to select processor and project type.



Our development board is based on LPC1549 processor so we select LPC1500-series from **Preinstalled MCUs**. Expand **LPC15xx** categories and select **1549**. Then click the picture of your board on the wizard and press next.

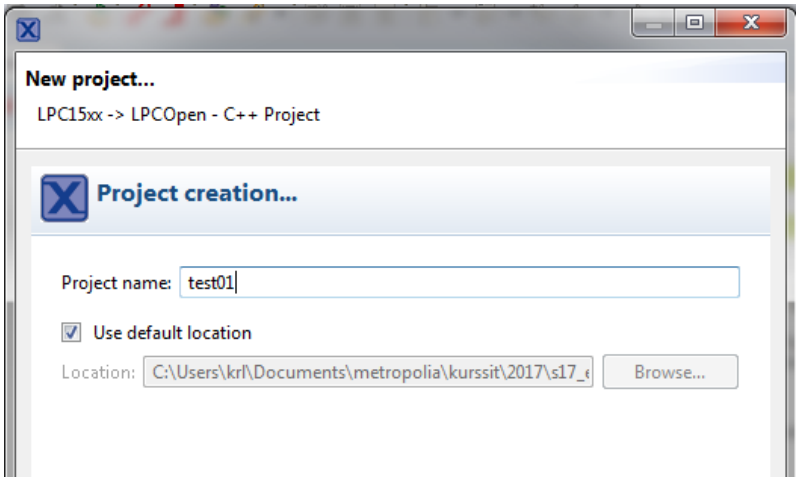


Then you need to select project type. Select **LPCOpen – C++ Project** and press next.



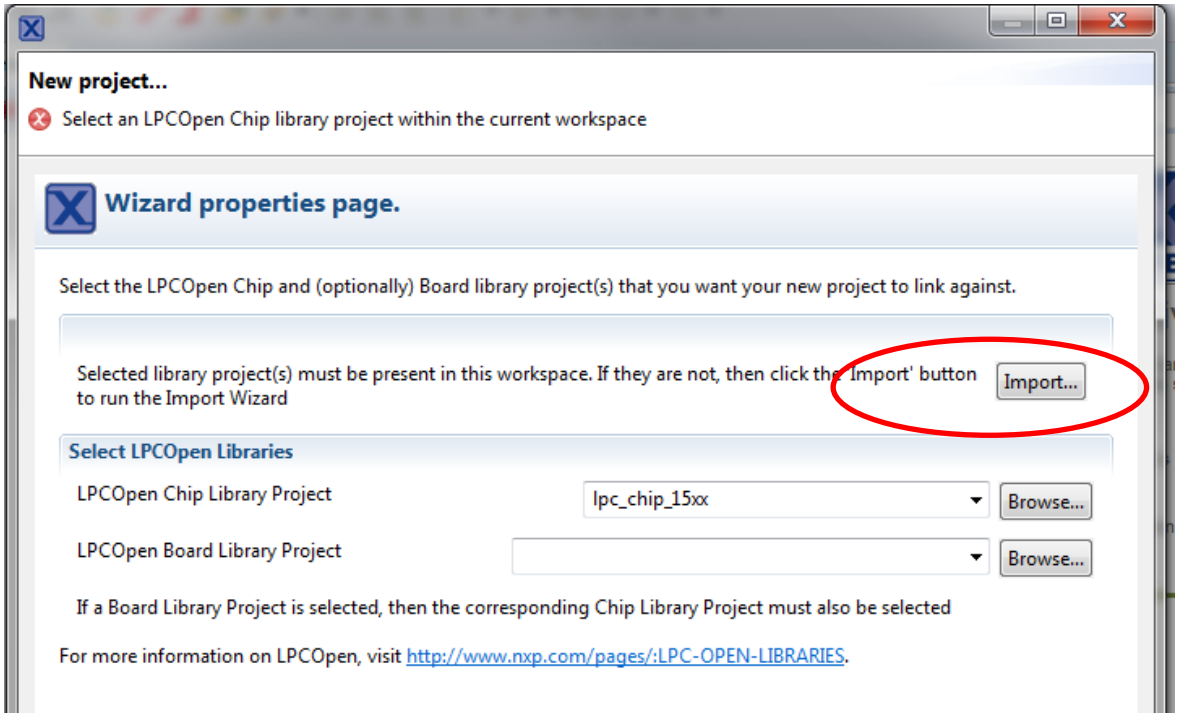
Then name your project. **Don’t use spaces in the project names and make sure that your path does not contain any spaces.** Some of the tools may produce unexpected errors with path name contains spaces.

Just press next after entering the project name.

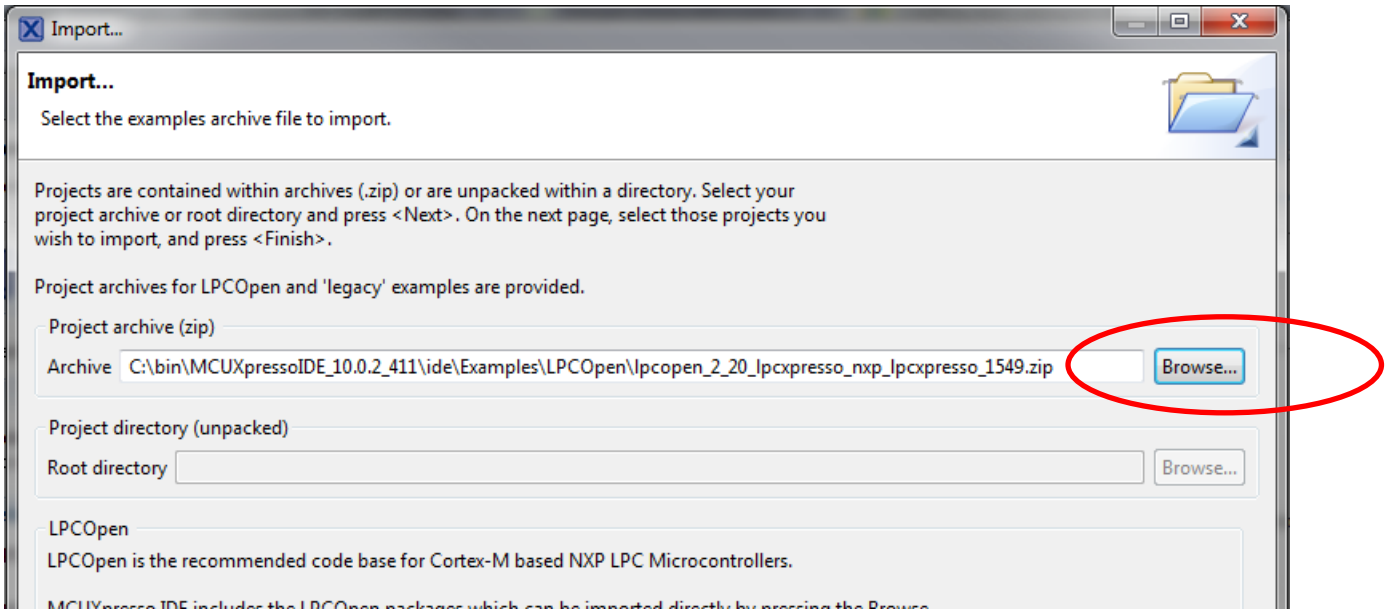


Then we add chip and board libraries to the project. This step is very important since **there is no easy way of adding chip and board libraries to a project that was created without them.**

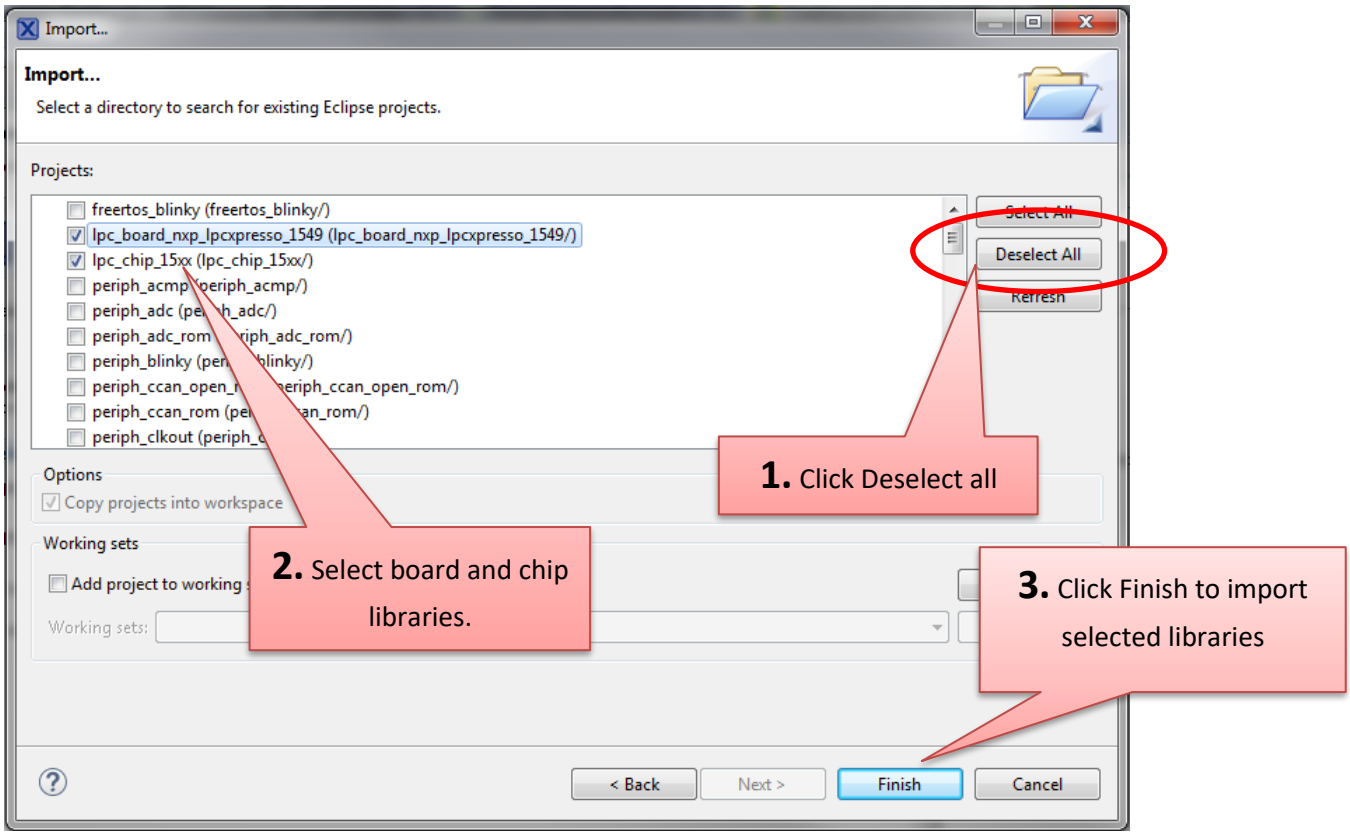
If you are adding a project to an empty workspace you need to import chip and board libraries in to the workspace. Click import button to start the import wizard.



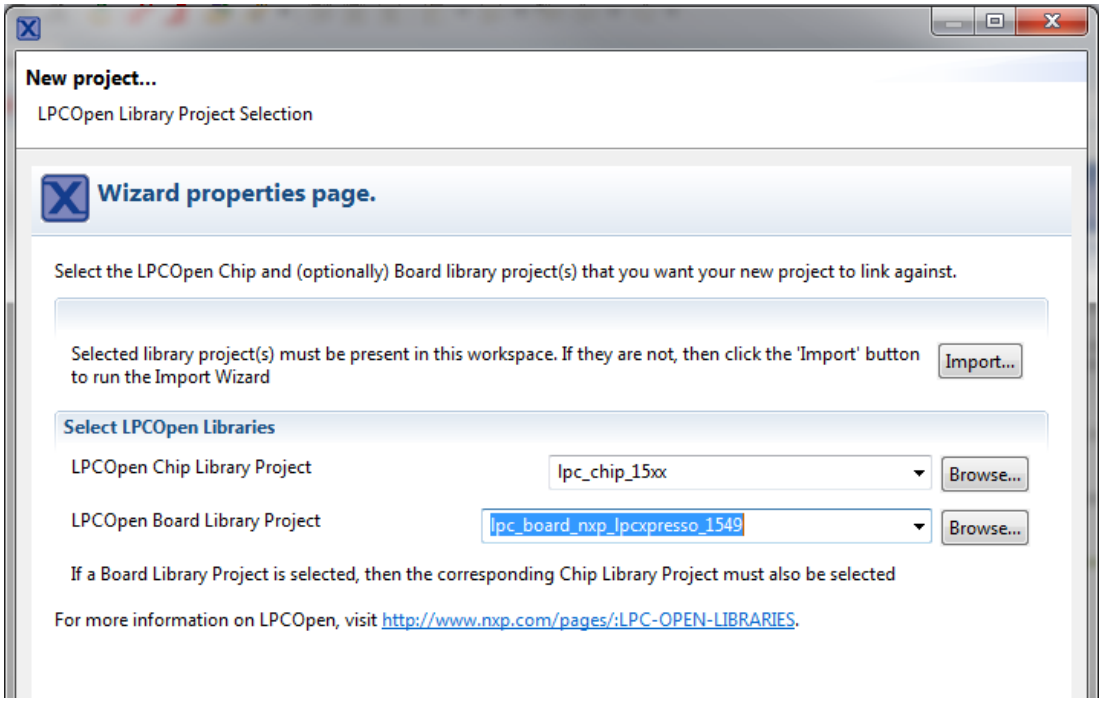
During installation the library archives are copied in to the installation directory. Click Browse and navigate into your installation directory. Then locate the library archives in:
<your installation directory>\MCUXpressoIDE_10.0.2_411\ide\Examples\LPCOpen and select lpcopen_2_20_nxp_lpcxpresso1549.zip



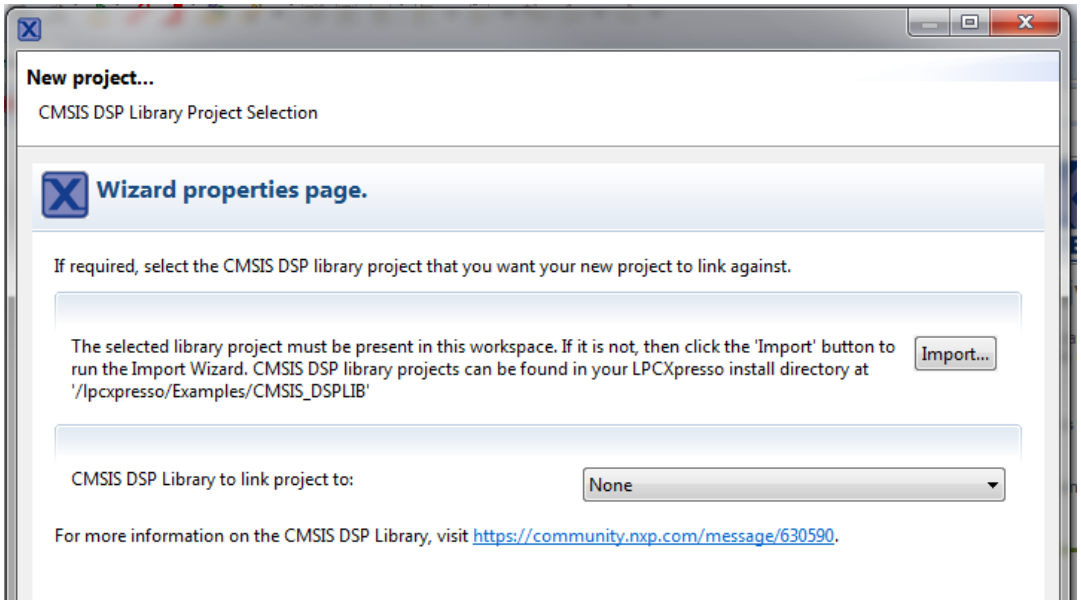
Then press next and a menu with a list of projects in the archive is shown. Click **deselect all** and manually check board and chip libraries. See the screenshot below.



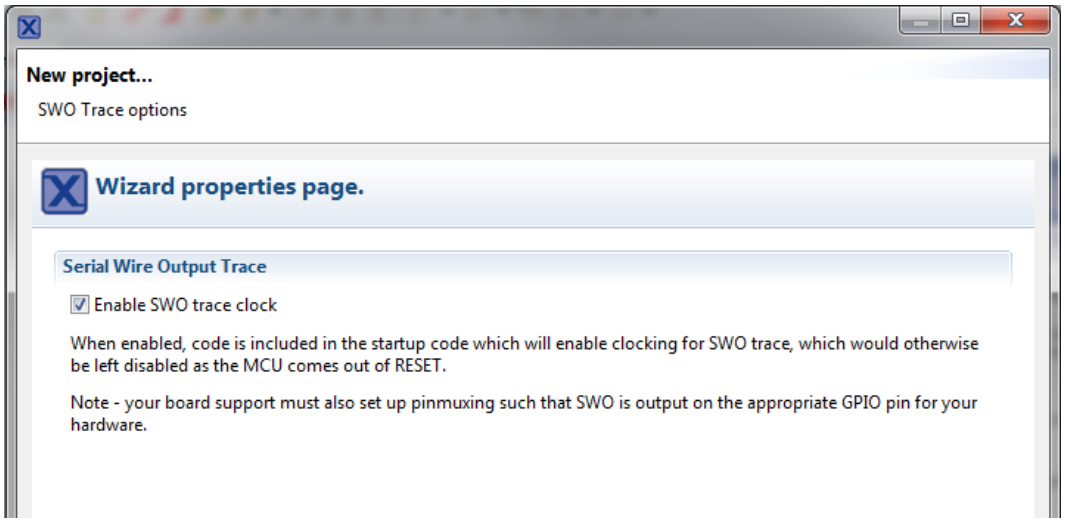
After the import you are returned to the New project wizard. **Make sure that you have selected both chip and board libraries** and press next.



Don’t add a DSP library.

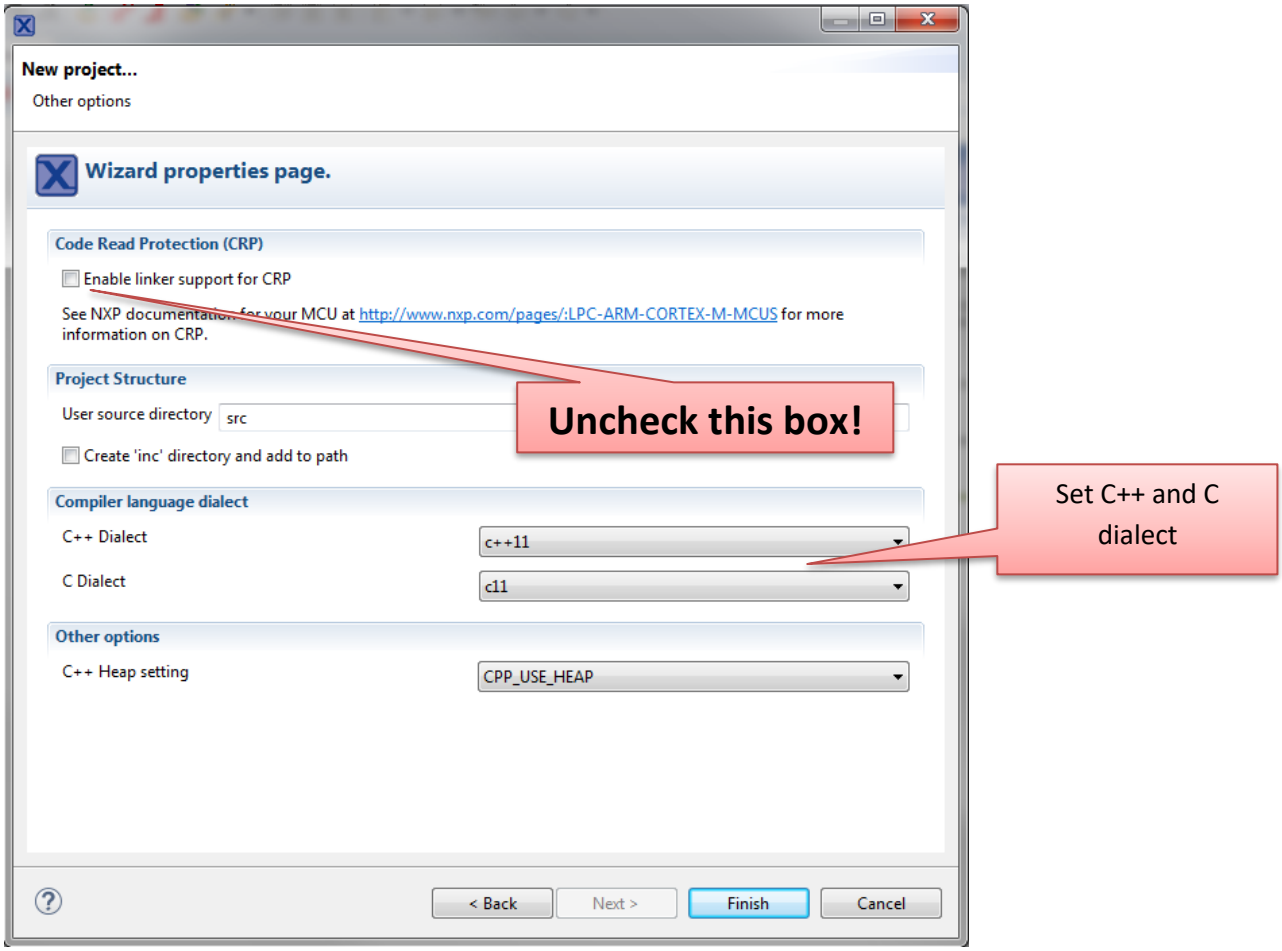


Make sure that Enable SWO trace clock is selected. Debugging will not work without SWO trace clock.



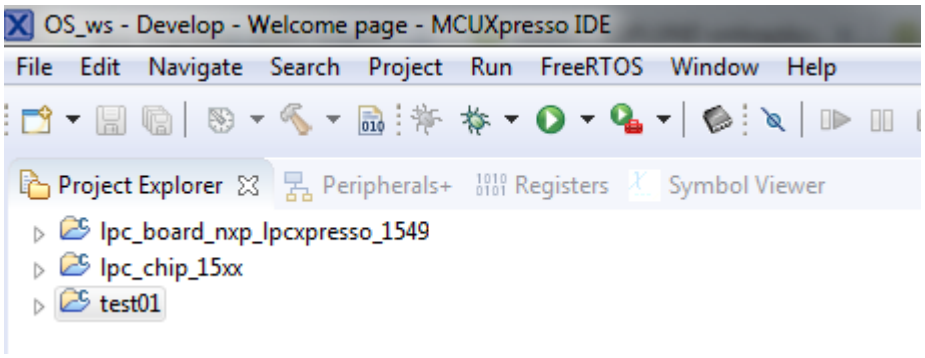
Then select dialect for C and C++. Use c++11 and c11.

Note that there is a bug in the project wizard which does not set C++ dialect properly. At some point in the future this will (hopefully) be fixed so it is a good habit pick the dialect in the wizard.



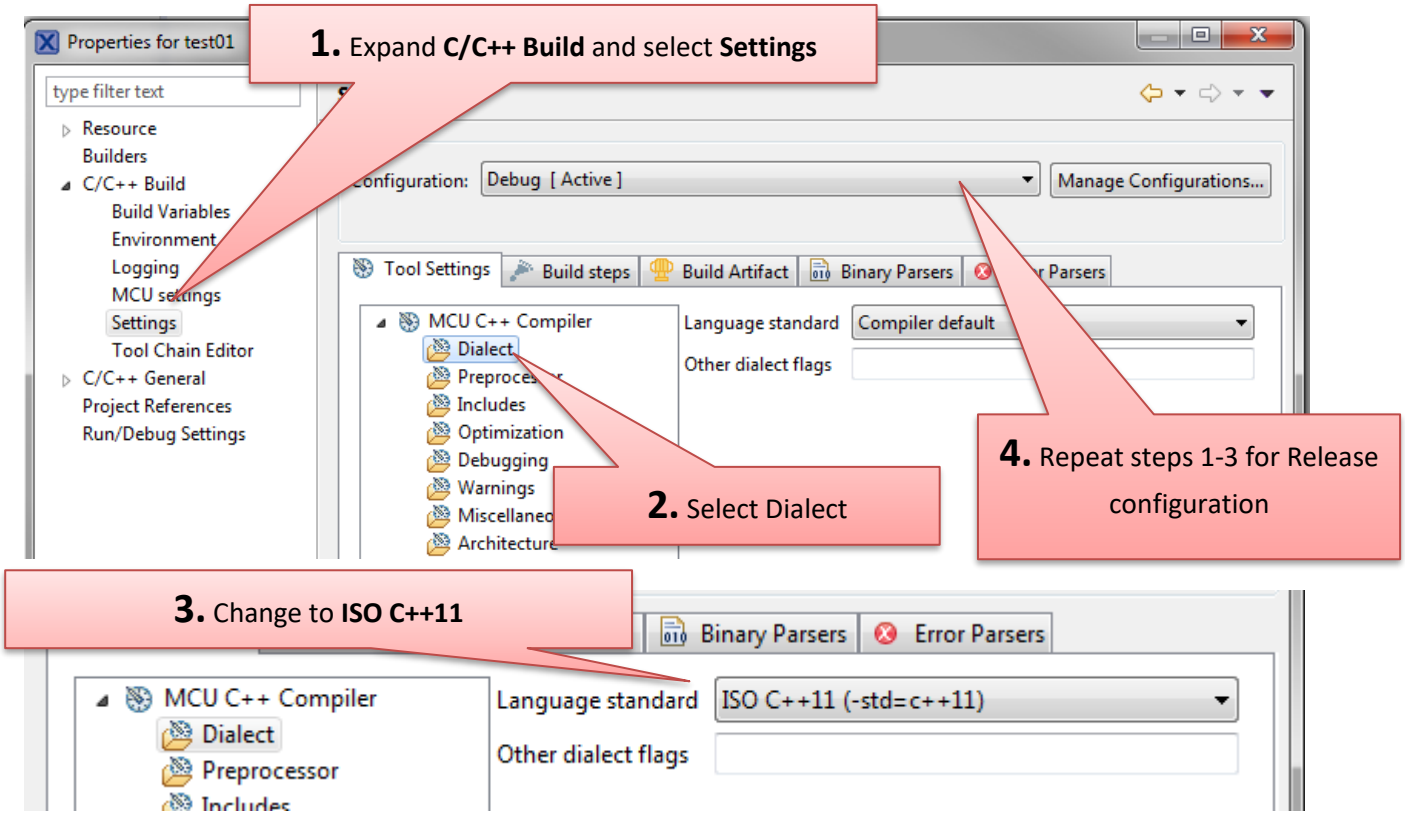
Press Finish and the project will be created. There are still a few steps that need to be taken before you are done.

Now you should have three projects in your workspace.



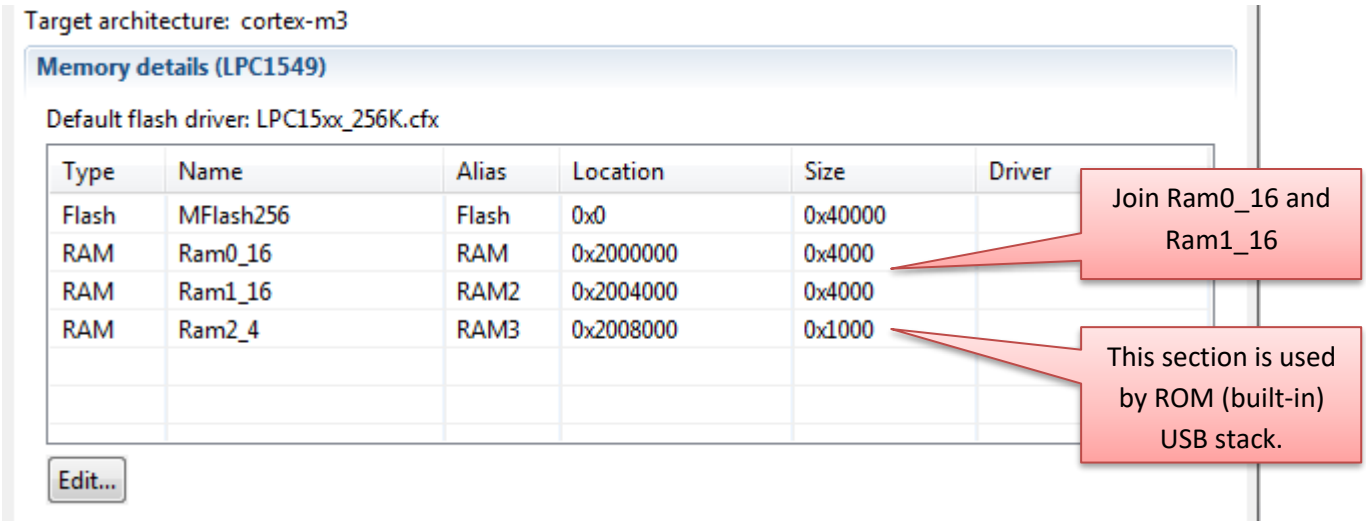
The next thing to do is to set C++ dialect of your project.

Right click the project name and select **properties** at the bottom of the popup menu.

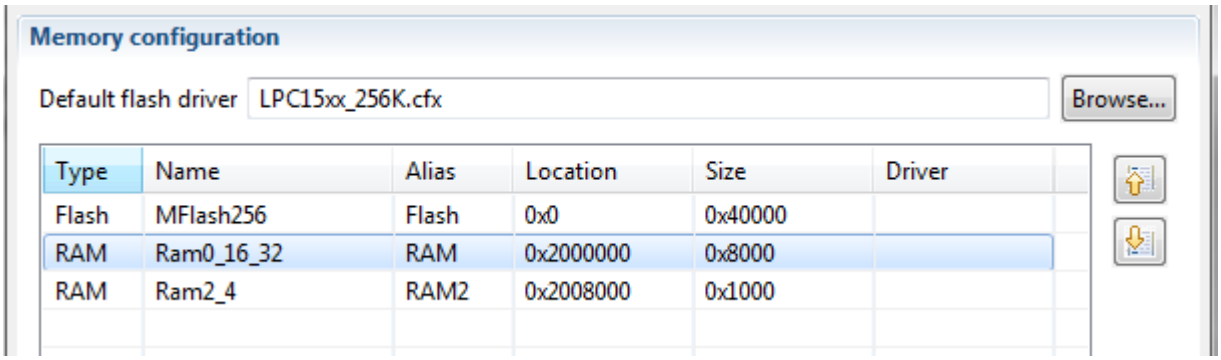


Then we need to increase the size of RAM available for linker. The onchip RAM is divided into three sections that can be enabled or disabled individually. All three sections are enabled at boot so there is no need to enable them. The default linker script divides RAM into three separate sections and by default linker uses only the section at the lowest address. Since the sections are adjacent in the RAM we can join them into a bigger section thus allowing linker to use more RAM.

Go to project properties →C/C++ Build → MCU settings. The default memory MAP looks like this:

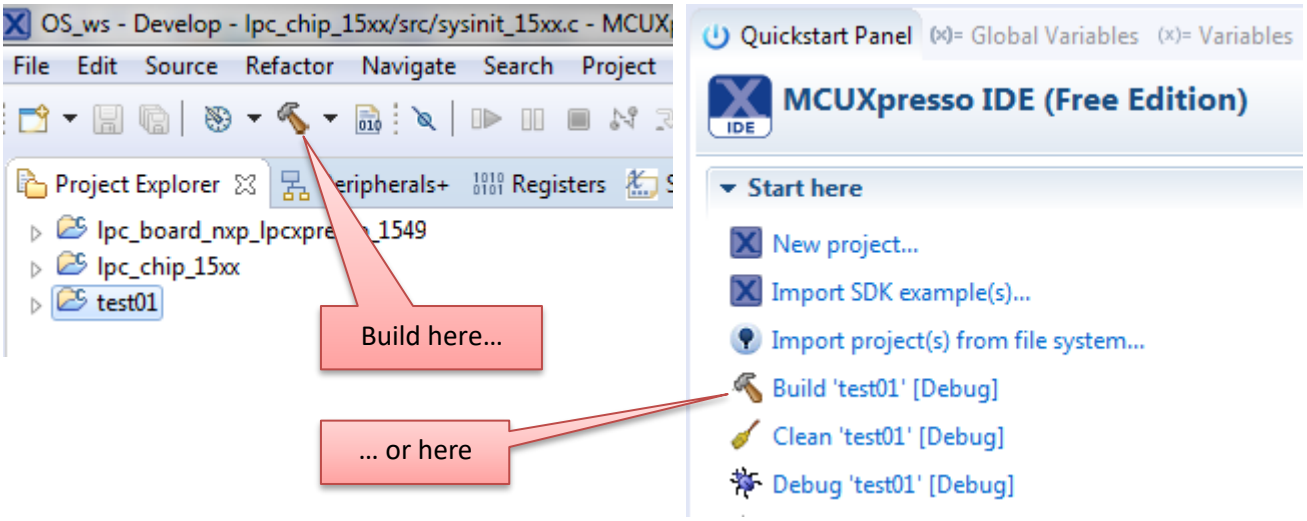


We are going to join the first two RAM sections into a bigger section. Click edit and click on Ram0_16 and then click join. You should get the following memory configuration.



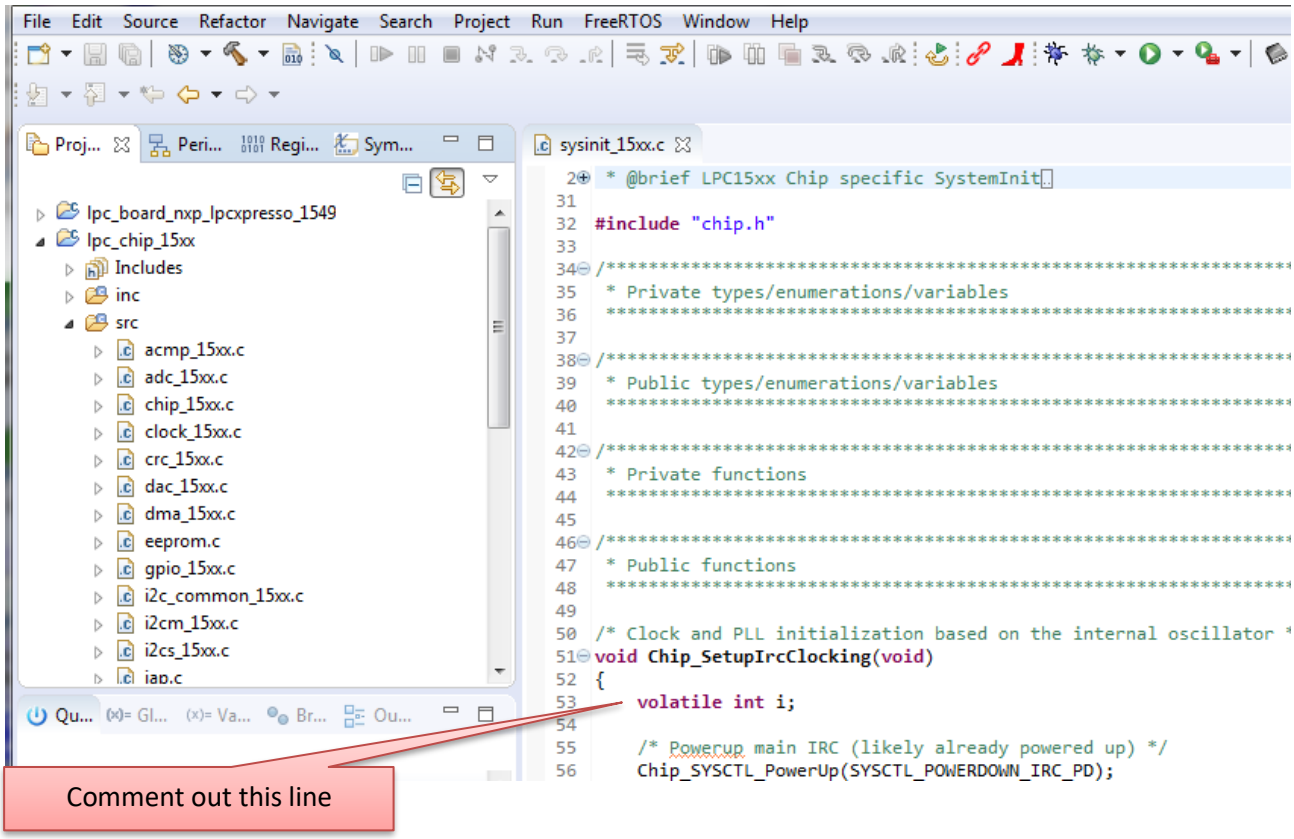
Press OK to close the editor and OK again to apply the changes you made.

The next thing on the list is to test that your project compiles. Click your project in the project explorer and click build.



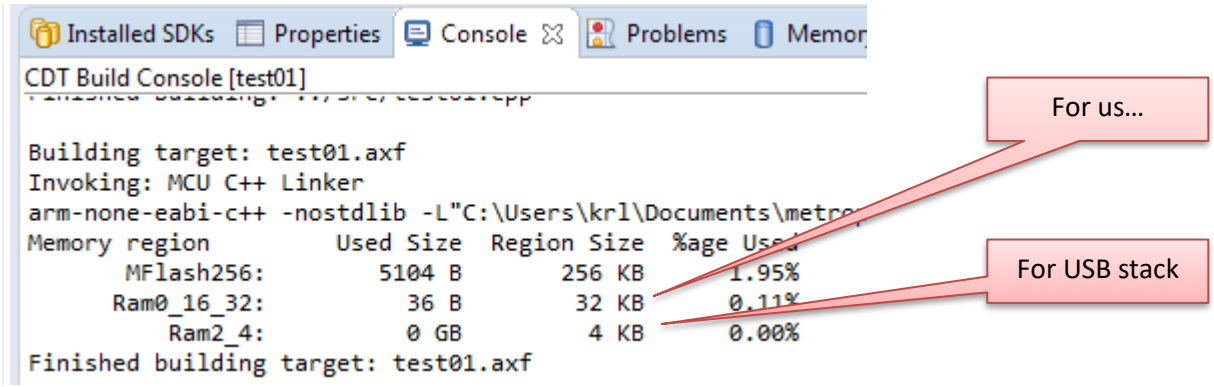
There will be warning about unused variable in the chip library.

Then we need to get rid of the warning in the chip library. Open sysinit_15xx.c in the editor



Then build your project again. If the build finishes with no errors or warnings you can continue to adding FreeRTOS in your project.

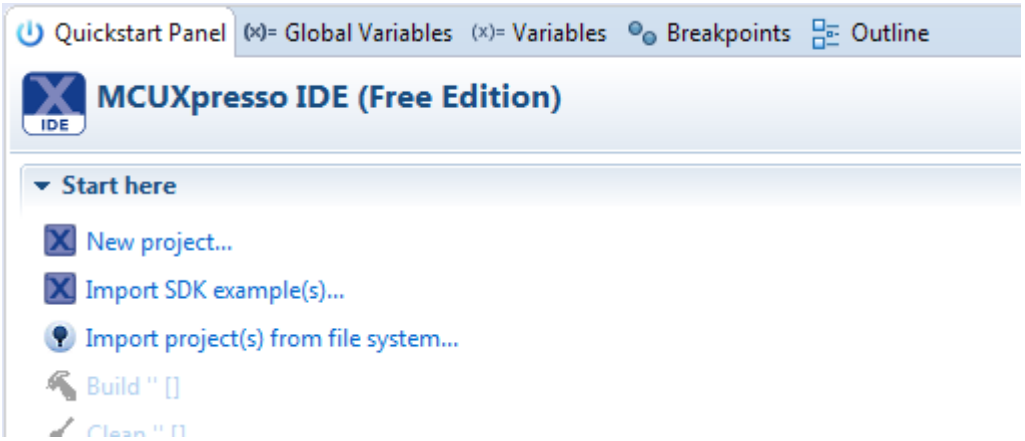
Remember to check the flash and RAM usage from the console window to be sure that your memory configuration is OK.



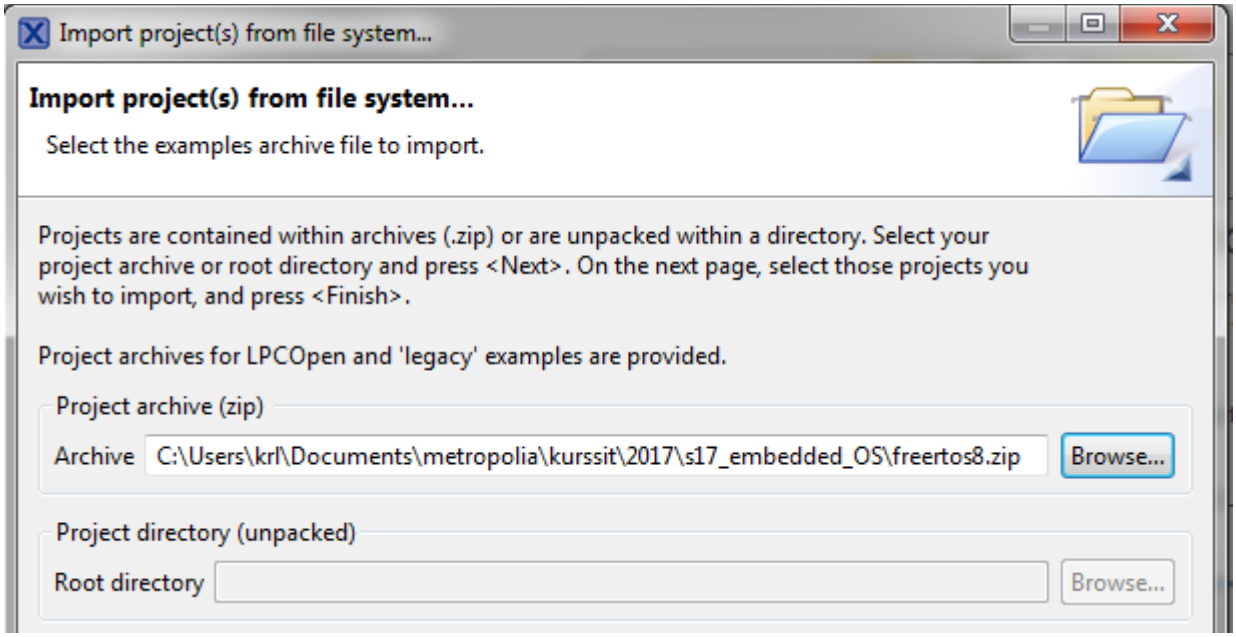
Import FreeRTOS

MCUXpresso comes with a FreeRTOS example project. Unfortunately, the version NXP provides is outdated and does not support some of the advanced debugging features so we import the lastest version of FreeRTOS from course workspace. Even the latest version requires minor tweaking to enable smooth debug flow with LPCXpresso so a preconfigured library is provided.

Download freertos10.zip from the course workspace. Then goto **Quickstart Panel** and click **Import project(s) from file system…**

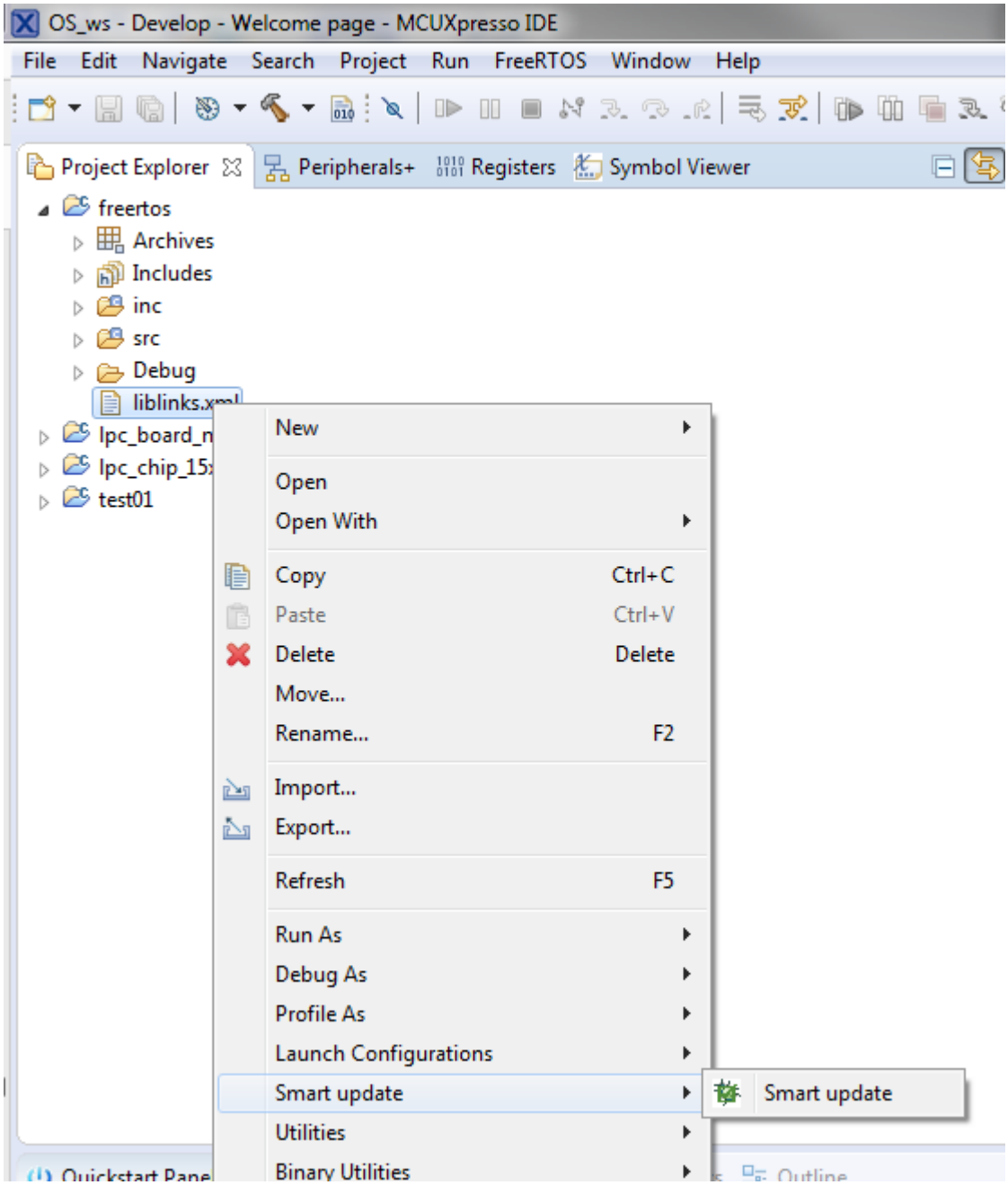


In the wizard click the topmost browse button and select the zip file.

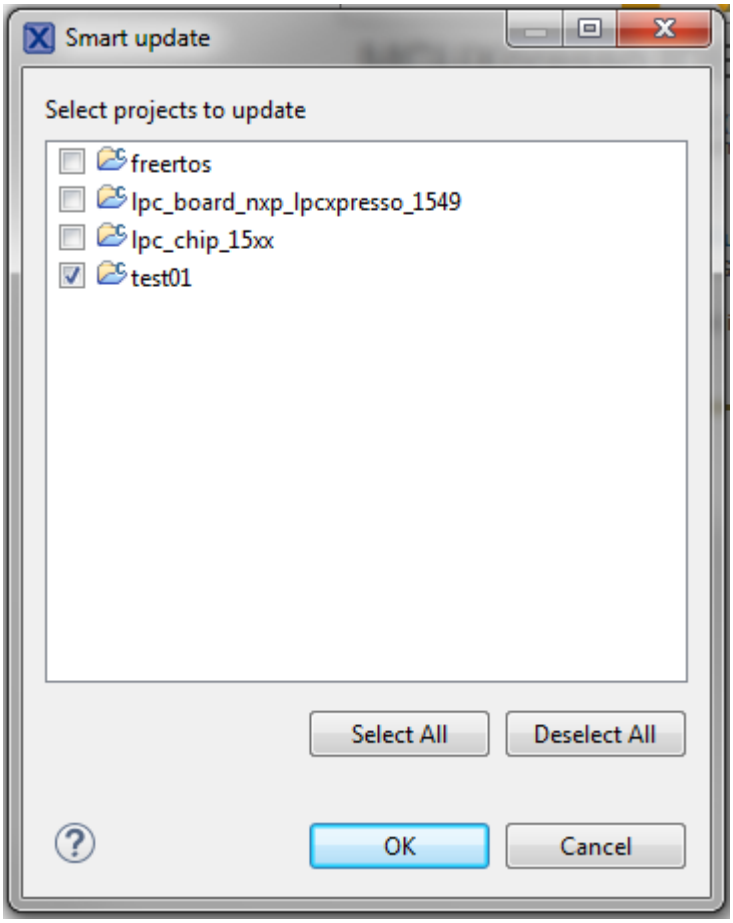


Then press **next** and you'll see a window where you can select which projects to import. There is only one project in the zip file and it is automatically selected so just press **Finish**.

Now that freertos library has been imported you need to tell IDE to link the library to your project. Expand the library and right click **liblinks.xml**. Select **Smart update** from the popup menu.



Select the projects that will use freertos library (in this example the project is called test01). Note that you can use smart import also later if you add a new project that uses freertos to same workspace.



Press OK and you are done.

You now have a C++-project that uses FreeRTOS – the only thing missing is a piece of software that puts FreeRTOS into action.

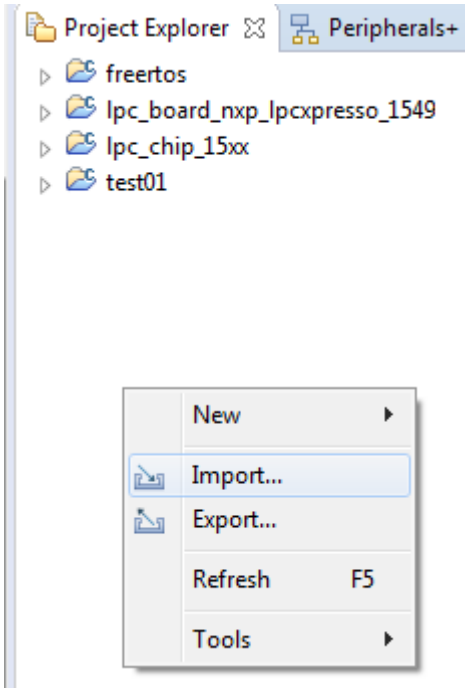
I have included an example file (blinky.cpp) that is essentially the same file that can be found in the example projects of LPCOpen. The only differences are that the file has .cpp. You can find the example in blinky.zip in the course workspace.

You can open/extract the zip-file using your favorite tool and copy the code manually.

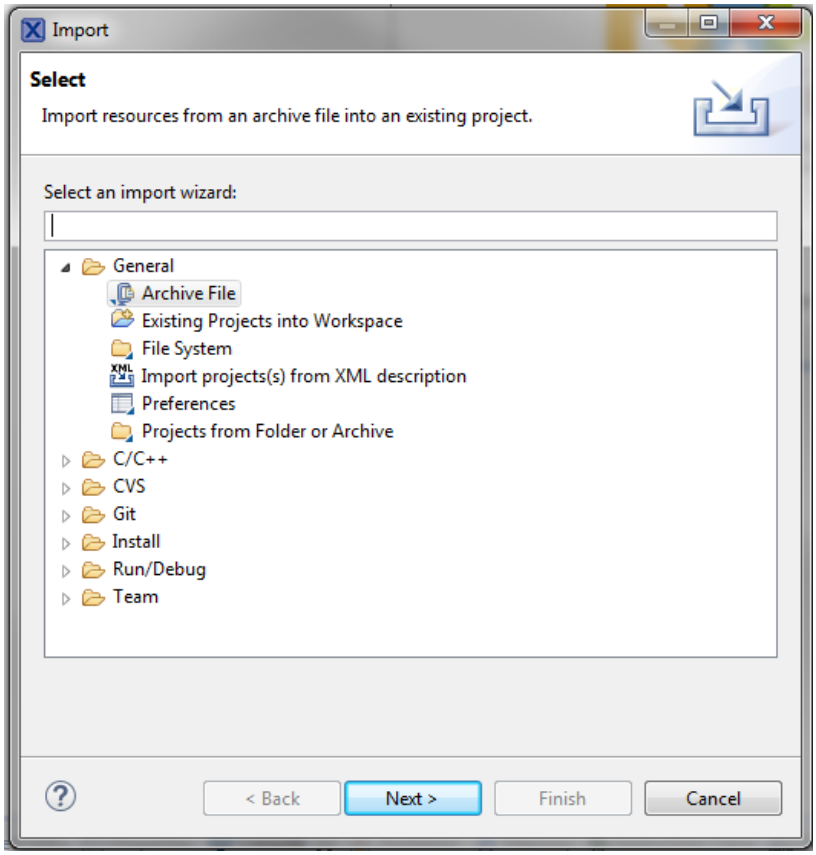
… or

You can also import sources directly from zip-files using the import wizard. The following shows the import wizard work flow for those who wish to do everything in the IDE. If you are going to copy files manually then you can skip directly to running your project (after you have copied the example code into your project).

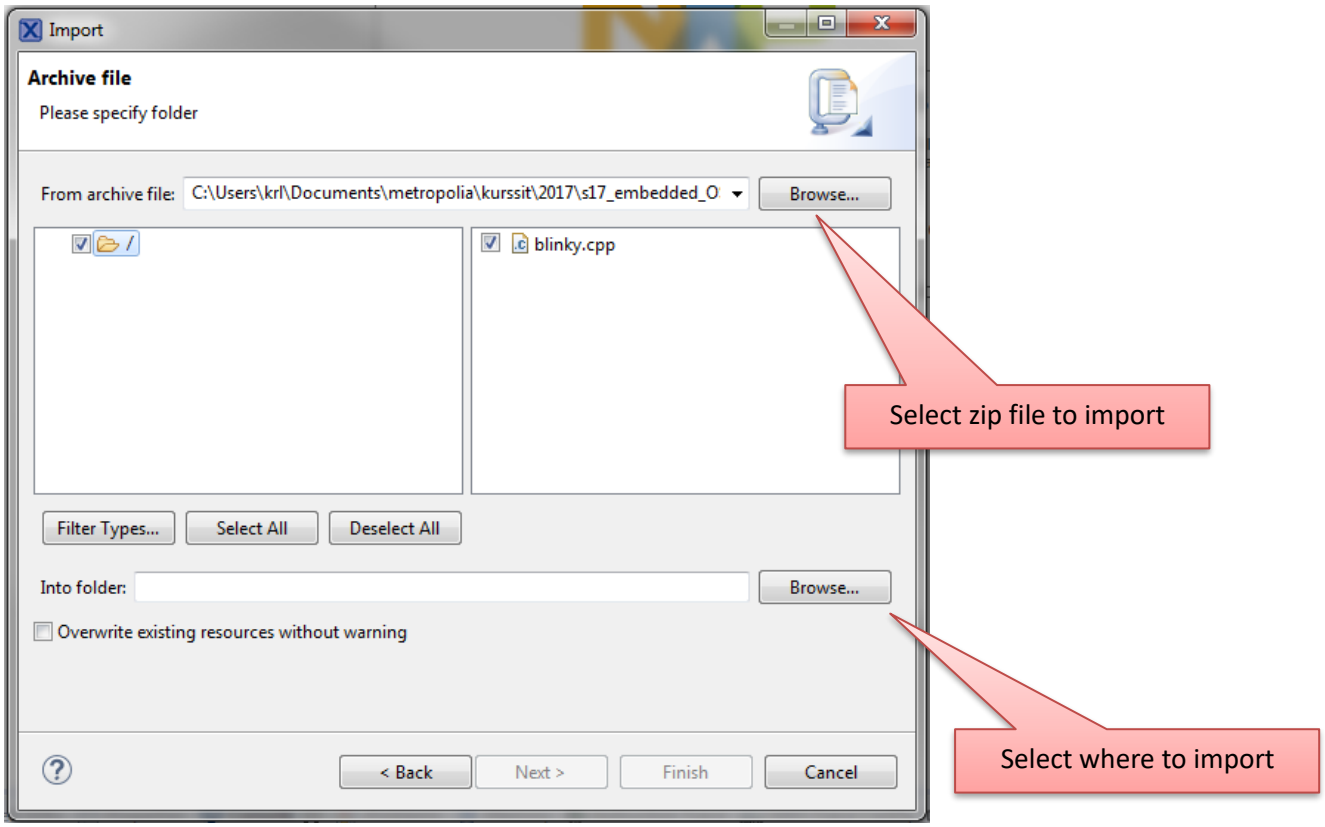
Right click the background of project explorer and select import.



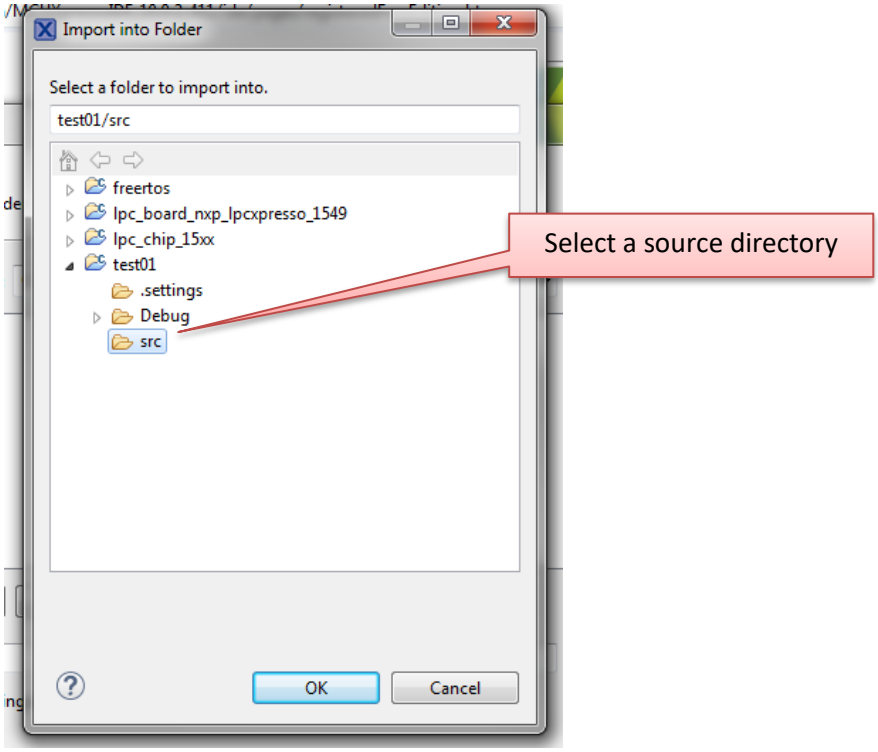
Select General → Archive File in the popup menu.



Browse to the zip-file you downloaded in From archive file.



Then select folder where to import the file.



Press OK here and Finish in the wizard.

Now you have a file that makes the onboard leds blink and sends some debug output. You may need to do additional tweaking to get your project to compile. Blinky.cpp contains main()-function which may conflict with your existing sources. Options are:

1. Copy code from blinky.cpp to your project main file and then delete blinky.cpp
2. Delete your existing project main file (and rename blinky.cpp if you wish).

Note that in a C++-project the file containing main() must always be a C++ file. MCUXpresso allows you to mix C and C++ but I recommend that you always write C++ files even if you are not using objects.

Running and debugging a FreeRTOS application

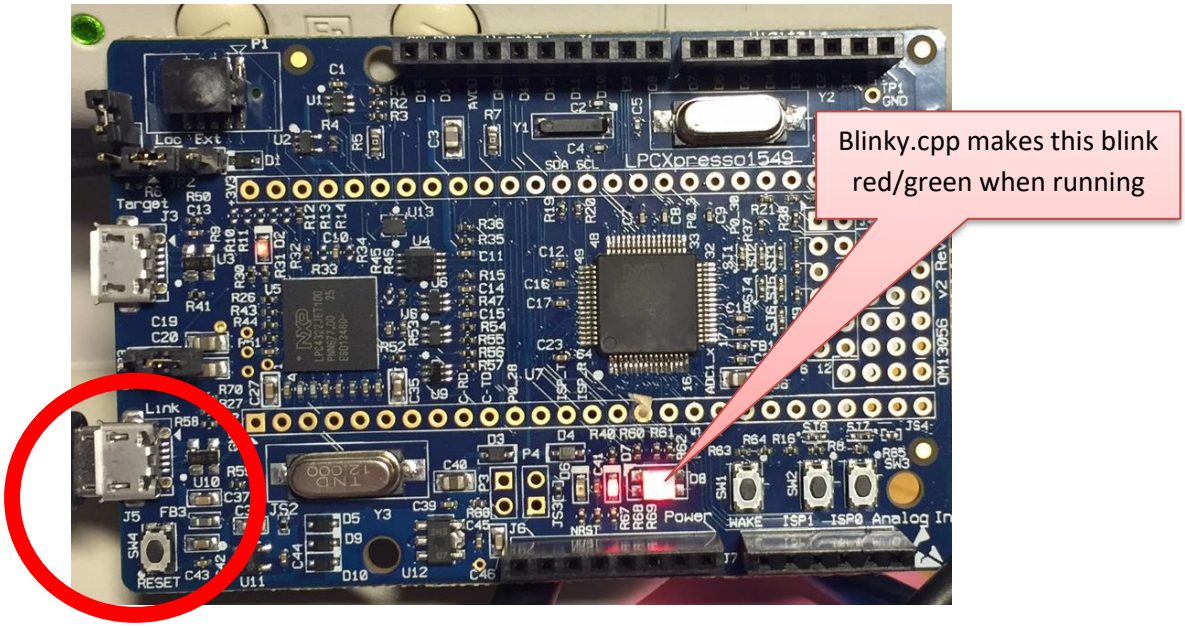
When your project builds without errors you can start debugging.

Compile your code. You can ignore the warnings from board and chip libraries but you must make sure that your own code produces no warnings.

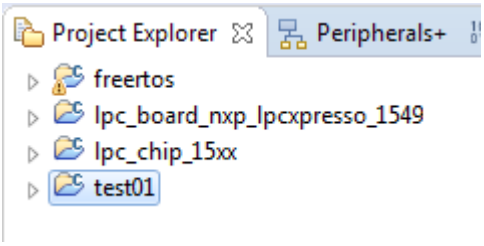
Before building your project you must select library type for your project and the libraries. Select each library and in the quick start panel set the library type to either **newlib (nohost)** or **newlibnano (nohost)**. Note that your library types must match with your main project. If you go for newlib (nohost) in your project then you must do the same for chip and board libraries.

When your project builds without errors you can start debugging.

Connect your development board to your computer. Note that development board has two micro-USB connectors. Make connection using the **Link** connector (next to the reset button).



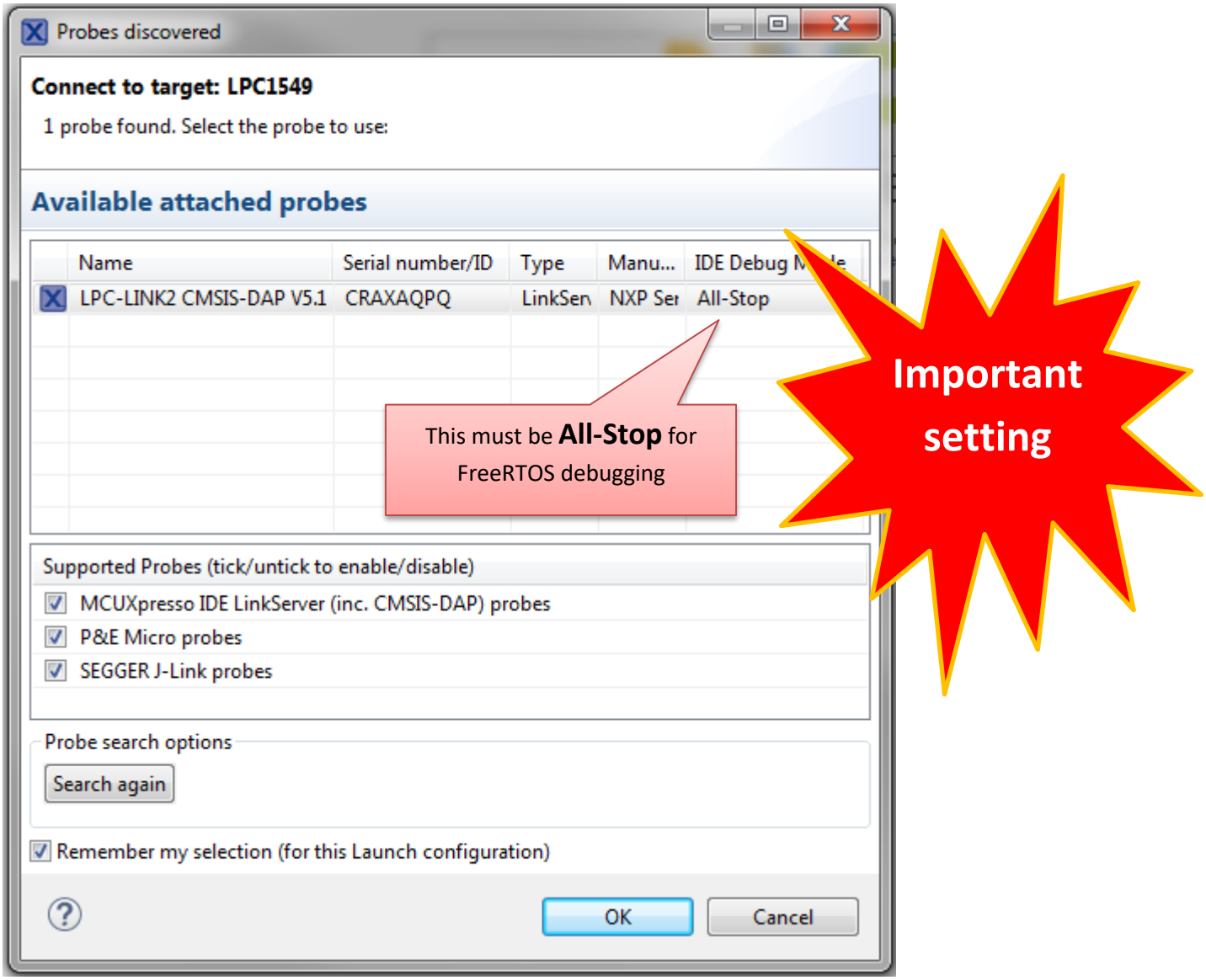
Then click on your C++ project in the Project explorer.



Then click Debug in the Quick start Panel.

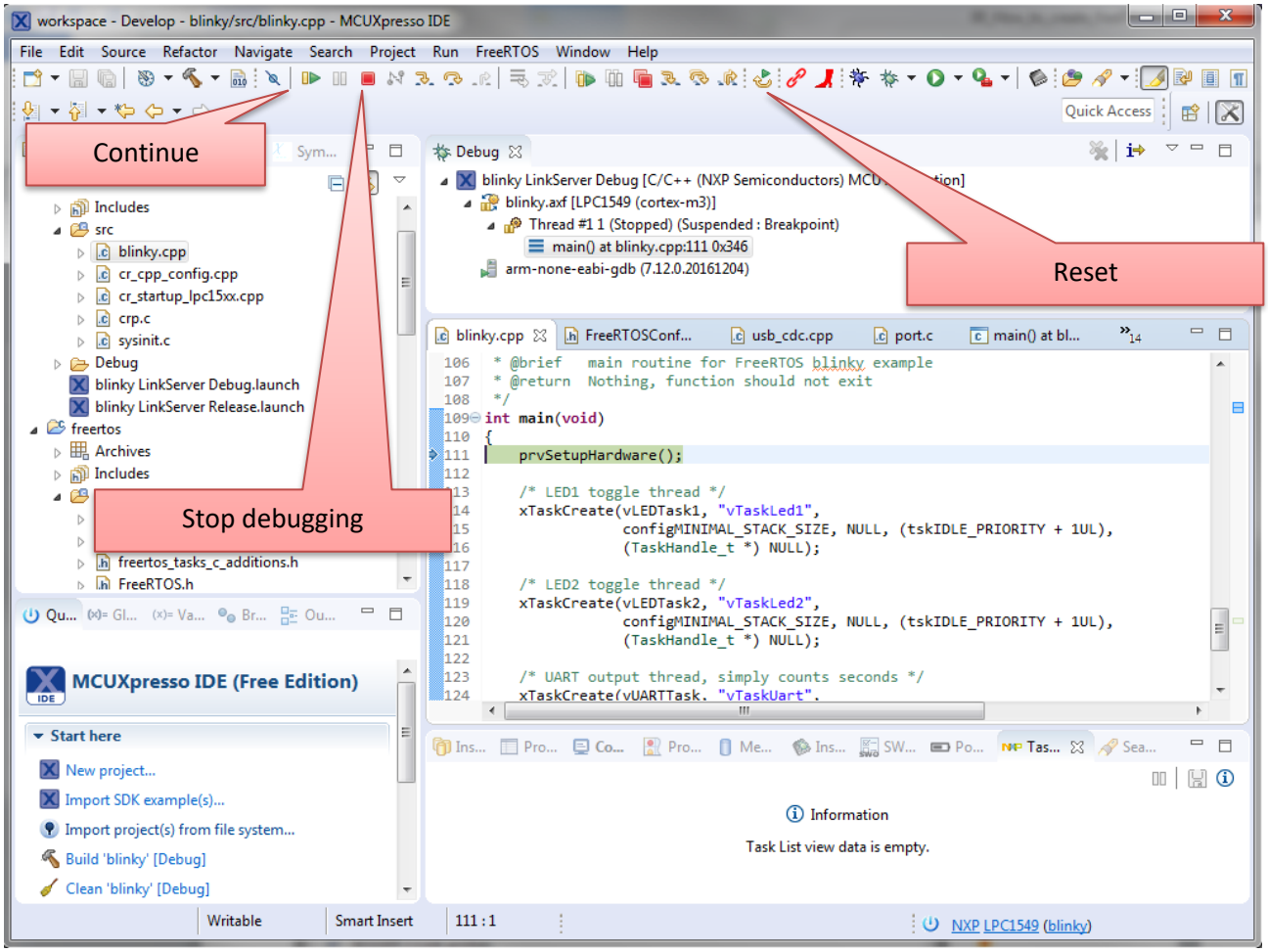
MCUXpresso has built in support for FreeRTOS kernel aware debugging. Kernel aware debugging allows debugger to read kernel data structures and display information about the state of each task. A more detailed data collection can be enabled in FreeRTOSConfig.h.

To debug a project click **debug** in the Quickstart panel. Note that Project explorer must be on the project you want to debug to make quickstart buttons work. The first time that you start debugging a project an important dialog is shown. The dialog is shown only once so it is extremely important that you set correct IDE debug mode. By default debug mode is Non-Stop and it must be changed to **All-Stop** to enable kernel awareness in full.



When your debugging session starts the program execution stops at the first executable line of main function. You can step into or over source lines, set break points and run your program. When you let the program continue you should see the on-board led blinking vigorously.

Debug view before program is run. There is an automatic break point at beginning of main(). Note that quite a bit initializations have taken place before we land here.



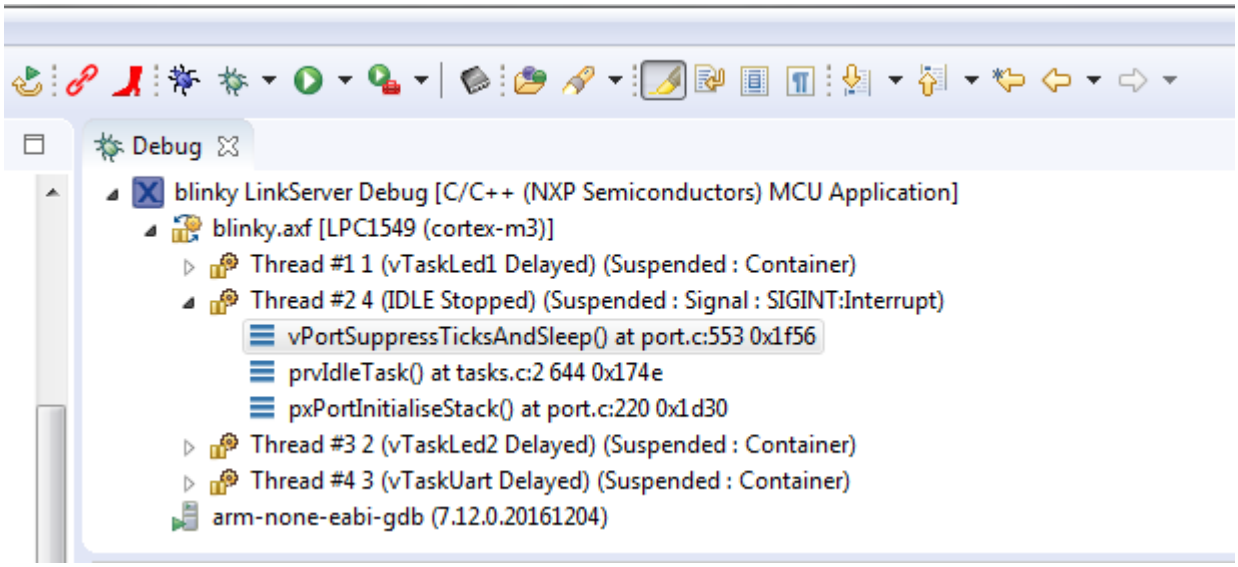
The debugger has a built in USB VCOM port that is routed to pins that the board library uses for debug output and standard output.

Use control panel to find out which comport the debugger creates and open the com port for example with PuTTY using 115200,8,N,1 setting. You should see **Tick:** and a number printed once a second.

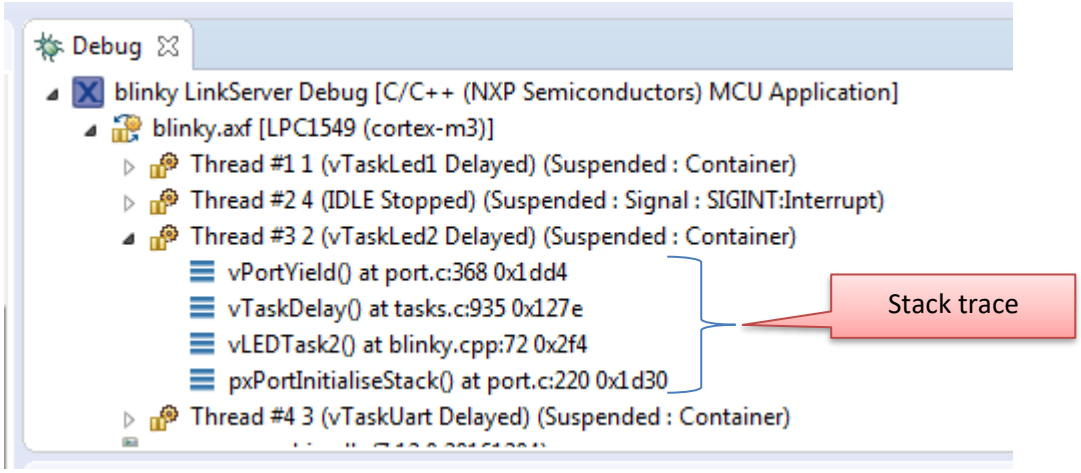
The debug COM-port is created when you start the first debug session and exists only during debugging. This port is part of the debugging system and it will not work without debugger. The CPU has also a true USB port that can be used without debugger (the other USB connector). Using that port is beyond the scope of this introduction.

If you want to restart your program there is no need to stop debugging. Just pause your program and then press restart. Pause button is activated after your program starts running.

When program is paused task aware debugger allows you to inspect running tasks and their states individually.

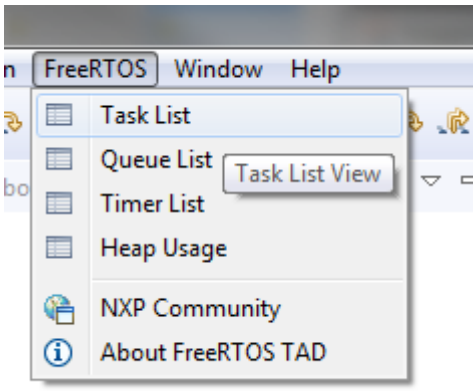


You can expand/collapse tasks and view stack trace of each task. Stack trace shows you function call trace. Top function is the most recently called function.

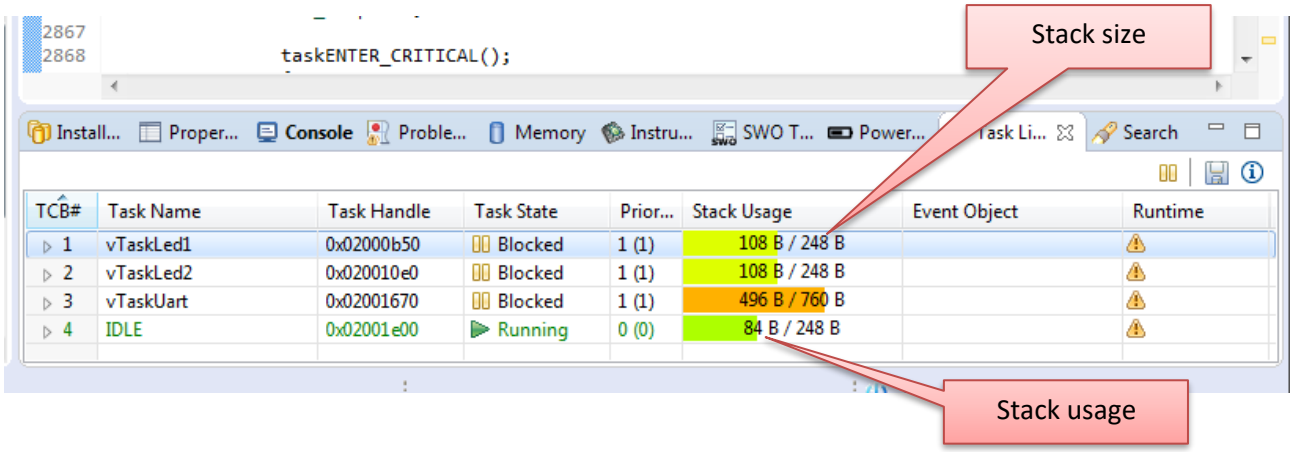


In the screenshot above vPortYield is the currently executing function. It was called by vTaskYield() which in turn was called by vLEDTask() etc.

You can view task summary by selecting FreeRTOS →Task List in the menu row.



Task list is displayed in the lower part of the window.



Stack usage statistics shows how much stack was allocated for each task and the maximum stack usage so far.