A Short Reference Manual of STM32CubeIDE

Jianhua Liu

v0.3, 3/22/2024

Contents

1 Introduction

STM32CubeIDE, called CubeIDE in the sequel for simplicity, is an Eclipse-based toolchain for developing embedded application projects targeted to running on STM32 MCUs. Being Eclipse-based, it is a cross-platform toolchain, which can be used on Windows, macOS, and Linux.

CubeIDE has the following functionalities:

- Creating a project quickly based on supported STM32 MPBs, including Nucleo and Discovery boards.
- Creating a project quickly based on any STM32 MCUs.
- Configuring the hardware peripherals using the integrated [CubeMX].
- Configuring multiple builds (will be called **build targets** or **targets** in the sequel to avoid ambiguity) and building them.
- Debugging the project based on two debug probes: ST's ST-Link and SEGGER's J-Link.
- Viewing variables of a running program in a **Live Expression** window.

Here, we focus on using CubeIDE for cross-compilation, which means that **we compile on a PC for MCU targets**. Yet, CubeIDE can also be used to develop native code, the code running on PC, as illustrated in the Developing native C/C++ projects with CubeIDE section at the end of this manual.

For some basic concepts of CubeIDE, see Sections 1.2 and 1.4 of *STM32CubeIDE user guide, UM2609*. This is readily available from https://www.st.com/en/development-tools/stm32cubeide.html#documentation.

Click here (https://www.youtube.com/watch?v=4wT9NhlcWP4&t=530s) for a series videos advanced features of CubeIDE.

Note that this is a short reference manual of CubeIDE, which is under development. If you see any issues or can extend some of the topics, please let the instructor know. You will be awarded with extra credits.

2 Installation of CubeIDE

The installation is straightforward. We just need to go to https://www.st.com/en/development-tools/stm32cubeide.html to download the appropriate version of the installer to install. Like for CubeMX, you may have to enter your name and email address in order to download the software.

Currently, we will use v1.12.1 for convenience. v1.13.1 will work, but you will need to do forced file conversions.

As indicated in Reasons of using a standalone CubeMX, we will use the standalone version of CubeMX for hardware configuration and code generation. As such, there is no need to manage embedded software packages here.

3 Specifying your workspace for CubeIDE

As you have read or will read from the structure of project folders, you need to use a specific folder for all the projects for this class.

- If you use Windows, you need to use the `C:\proj_mp` folder.
- If you use Linux or mcOS, you need to use the `~/proj_mp` folder.

When you open up CubeIDE for the first time, you need to choose the above folder as your workspace. Check the **Use this as the default and do not ask again** box.

4 Basic components of CubeIDE

4.1 Project-related components

CubeIDE is based on Eclipse, and hence, we need to use a number of terminologies from Eclipse:

- A **workspace** is a collection of many projects or library in a single folder. When we use CubeIDE, we should should a workspace to work on. For us, all the course related projects are collected in a single workspace called `proj_mp` under `C:` on Windows and the home directory of macOS or Linux. This can be set up at the first running of CubeIDE. Set it as default.
- A **project** is a collection of many folders and files that are related to a
- When we have a number projects, each for a single different hardware board, we call them **related projects**. This is not the concept of CubeIDE; it's just a illustration that we can use different hardware boards to perform similar functionalities.

4.2 GUI-related components

The CubeIDE window has the following GUI-related components used for code editing, navigation, building, and debugging.

4.2.1 Perspectives

CubeIDE mainly has two different top-level layouts: the C/C++ perspective and the debug perspective.

Each perspective can be customized according to the user's need. If the views of a perspective are out of our control, we can reset the perspective to its default by clicking **[Window » Perspective » Reset Perspective. . .]**. This can also be done by right-click the **[Perspective]** button and choose **[Reset]**.

4.2.2 Areas

Each perspective has a number of areas. The C/C++ perspective has the following areas:

- The project area, which is on the left side of the CubeIDE window.
- The editor area, at the right side of the project area. Note that many files can be edited in the editor area in different tabs.
- The **outline** area, at the right side of the editor area.
- The **console** area, which is below to the right of the project area and below the editor area.
- The **analyzer** area, which is to the right of the console area.

Note that

- The project spans the entire height of the window. Its width can be changed.
- The editor and the outline areas have the same height, and their widths can be adjusted.
- The console and analyzer areas have the same height, and their widths can be adjusted.
- The entire width of the editor and outline areas is the same as that of the console and analyzer areas.
- If needed, we can add more areas in one of the above basic areas.

4.2.3 Views

CubeIDE use many views to display various information. This views are located in the above areas as tabs.

We can add or remove views as needed easily.

Note that we can have detached views that are not attached to one of the above areas. We can change an attached view to a detached one or vice verse easily.

4.2.4 Other GUI elements

There are a number of bars visible all the time.

- The **project info bar**. This is the bar at the top of the CubeIDE window, which displays the information of the workspace and the active project.
- The **menu bar**. This is located below the project info bar, used to show the menus.
- The **toolbar**. This is located below the menu bar, used to show the tool icons.
- The **status bar**. This is the bar located at the bottom of the CubeIDE, used to show various status info.

There are some other GUI elements that are only visible when used.

- The **context menus**. There a large number of context menus. They are only visible when we right-click on some items, such as the project.
- The **dialogs**. These are interactive windows popped up when we choose an item in the context menu or in the menu bar. With dialogs, we can configure CubeIDE or the project.

The dialog can have the following GUI elements:

- **Panes**. Many times, the items of the dialog are organized in the left pane. When we click one item in the left pane, the detailed contents will be displayed in the right pane.
- **Tabs**. A pane can have multiple tabs to organize the information.
- **Nodes**. If there are too many items in a (tab of a) pane, they may be organized in notes. We need to click open an node to see the subsequent items.
- **Pages**. There may be multiple pages to show the contents in a certain sequence.
- **Sections**. The contents of the right pane are usually organized in sections.
- **Fields**. Here is where the text can be input to do some searches or setups.

There are also the following common GUI elements:

- Radio box.
- Checkbox.
- List box.
- Dropdown.

4.3 Other project components and terminologies

- Board is called a MCU target.
- A build has two perspectives:
  - The build configuration, the configuration or setup used to build the project.
  - The build output, the compiled `elf` file and related files.
- Debug configuration, the configuration used to debug the build image.

5 Importing a project generated by CubeMX

To make the discussion easier, we use the `cc3d_uart_redirect` project as an example. This project can run on four different MPBs. Below, we use F412dsc as an example. The project files generated by CubeMX are saved in the following folder.

```
cc3d_uart_redirect/cc3d_uart_redirect_f412dsc
```

After the project is generated by CubeMX for CubeIDE, we need to import it to CubeIDE. This can be done using two approaches.

- The **easy and quick approach**:
  - In a File Explorer, go to the `cc3d_uart_redirect_f412dsc/STM32CubeIDE` folder.

  - Click the `.project` file to let CubeIDE automatically start and import the project. (Clicking the `.cproject` file should work as well.)

- The **standard import approach**:
  - In CubeIDE, click **[File » Import. . .]**. The (1st) **Import** dialog will pop up.
  - In the dialog, choose **[General » Projects from Folder or Archive]** and then click **[Next]**. The (2nd) **Import Projects from File System or Archive** dialog will pop up.
  - In this new dialog, click **[Directory. . .]** button on the right of the **Import source** field. The (3rd) **Browse for Folder** dialog will open.
  - In this new dialog, go to the `cc3d_uart_redirect_f412dsc/STM32CubeIDE` directory generated by CubeMX. (We should be able to see the `Application` and `Drivers` directories now.) Click **[Select Folder]**. This 3rd dialog will close, and we are back to the 2nd dialog.
  - In this dialog, we should be able to see that under the **Folder** column, the checkbox before **STM32CubeIDE** is checked. Click **[Finish]**. The project is then imported to CubeIDE.

Now, we assume the project is imported to the workspace of CubeIDE, and we are ready to manage the project to add user code, as illustrated below.

6 Managing a project

6.1 Creating folders of the project

Assume we continue the above `cc3d_uart_redirect` project where we have the `cc3d_uart_redirect_f412dsc` folder already.

Note that we want to add the `cc3d_uart_redirect_g431n32` folder as well, which is similar to the `cc3d_uart_redirect_f412n32` folder for the files for F412n32. This is to illustrate the folder structure for accommodating multiple MPBs in a project.

We need to create four folders in **File Explorer** to host library and user code as well as test code and Renode scripts. They are listed below:

```
cc3d_uart_redirect   (root folder)
    cc3d_uart_redirect_f412dsc    (folder for F412dsc)
    cc3d_uart_redirect_g431n32    (folder for G431n32)
    library      (folder for library source files)
    renode       (folder for Renode script files)
    src_cc3d_uart_redirect  (folder for user source files for this project)
    test_cc3d_uart_redirect (folder for the test code for source files)
```

Note that some of the above folders can be created by copying from the similar projects. If needed, we need to change the file names to reflect the nature of the files.

For the above `cc3d_uart_redirect` project, we have the following files in each of the above 4 folders:

```
library     (folder for library source files)
    supported_mcu.h
    uart_redirect.c
    uart_redirect.h
renode
```

```
    cc3d_uart_redirect_f412dsc.resc
src_cc3d_uart_redirect
    _cc3d_uart_redirect_main.c
test_cc3d_uart_redirect (empty folder as we do not have test code)
```

Note that the above folder structure is the same as what we have discussed in the User source files in a project section of a loose note.

Now, we are ready to discuss how to add and include library and user files in the project so that they can be managed by Eclipse.

6.2 Adding the hook for user code

We will use a hook in the generated code by [ref-CubeMX] to use our user code. The simplest approach we can do is to add two lines of code in the *generated* `main.c` file as the hook. This is done in the **User code** block just before the `while (1)` loop, as shown below:

```
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  /* USER CODE BEGIN 2 */
  extern void mp_main(void);  // <-- Prototype of the HOOK
  mp_main();                  // <-- The HOOK function
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
```

By using the hook, we will not have to add code in the generated files. This way, we will be able to have clean reusable code in user source folders that can be shared by multiple MPBs.

6.3 Adding two source folders to the project

To organize the project files in a hierarchical manner, we can add folder for source files to the project in CubeIDE. One is `lib` and the other is `src`. This can be done by the following steps:

- Right-click the `cc3d_uart_redirect_f412n32` project and choose **[New » Folder]** in the context menu.
- Provide the folder name in the **Folder name** edit field and click **[Finish]**.

Note that these folders are different than the one we created in File Explorer. The folders just created here are managed by CubeIDE and the ones we created in File Explorer are not.

Note also that there are another ways of doing the same thing, for example:

- Clicking the **[Arrow]** to the right side of the **[New]** button.
- Using the context menu: **[New » Source Folder]**.

Now, we can move on to add files to the above folders so that they can be managed by CubeIDE. There are two approaches to add these files, depending on the purpose of the files.

- If the file is unique for each MPB, we need to add a physical copy of this file. For example, when we use FreeRTOS, we need to add a copy of the `FreeRTOSConfig.h` file to each MPB.
- If the file is common for all the MPBs, then we need to add a link of the file so that the change of the file to one MPB will be automatically reflected to the other MPBs. This is the case of most source files.

Below, we will discuss how to add files using these two different approaches.

6.4 Adding physical files to the project

There are three different approaches to do this, as discussed at https://dzone.com/articles/how-add-existing-files-eclipse.

- Copy and paste.
- Drag and drop.
  - Select the source files to copy by using **Ctrl** click.
  - Drag the selected source files and drop them to the intended source file folder of the current project.
  - Choose **[Copy files]** and click **[OK]** in the dialog popped up when we drop the files.
- Import the files.
  - Click on the folder of the project to where we want to add the files.
  - Click **[File » Import]**.
  - In the **Import** window, select **[General » File system]**. Click **[Next]**.
  - Browse and choose the files to import.
  - To copy the files to the project folder, just need to click **[Finish]** after choosing the files.

The first two approaches are preferred.

6.5 Adding linked files to the project

We can use two approaches to add linked files to the project.

- Drag and drop.
  - Select the source files to link by using **Ctrl** click.
  - Drag the selected source files and drop them to the intended source file folder of the current project.
  - Choose **[Link files]** and click **[OK]** in the dialog popped up when we drop the files.
- Add the files step-by-step.
  - Right-click the folder of the project where we want to add linked files.
  - In the context menu, choose **[New » File]**.
  - In the popup window, click **[Advanced »]** and check **Link to file in the file system**.
  - Click **[Browse. . .]** to find the files to link. Then click **[Finish]**.

6.6 Linking a folder

Sometimes, we need to link a folder of library files, such as the folder for FreeRTOS. This is similar to linking files and can be done in the following steps.

- Right-click the project folder that we want to add the linked folder to.
- In the context menu, choose **[New » Folder]**.
- In the popup window, click **[Advanced »]** and check **Link to alternate location (Linked Folder)**.
- Click **[Browse. . .]** to find the folder to link. Then click **[Finish]**.

6.7 Including header files to the project

Including header files can be done by setting the include path, as detailed below for the `library` folder under the root folder of the project. Note that this is the folder in

- Right-click the project and choose **[Properties]** in the **Context Menu**. A **[Properties for . . .]** dialog will pop up.
- In the left pane of the dialog, click **[C/C++ General » Paths and Symbols]**. The right pane will be the **Paths and Symbols** pane.
- In this pane, click the **Includes** tab.
- Under **Languages**, choose **[GNU C]**.
- In the right pane, we should be able to see all the *include directories*.
- Click **[Add ..]** to open a **Add directory path** dialog.
- Type `../../lib` in the edit field of this dialog.
- Check all the following checkboxes:
  - **Add to all configurations**.
  - **Add to all languages**.
- Repeat the above from clicking **[Add ..]** to add another by typing `../../src` instead of `../../lib`.
- Click **[Ok]**.

Note that for the above two checkboxes, we may not need to check them for all the include folders as some of them may not be needed.

Another way to include the path to the header files in the previous step is to browse to the folder holding the include file with the following options

- Click **[File system. . .]**.
- In the new dialog, browse to select the folder and click **[Select Folder]**.

Note that we need to **include** all the header file folder.

6.7.1 Another approach to include header files

We can include the header files in an approach different than what we did in the ref man of CubeIDE. It is done in the following steps:

- Go to the **Properties** dialog of the project.

- Go to **C/C++ Build » Settings**. In the right pane, click the **MCU GCC compiler** node and click **Include paths**.
- Click the **Add** path icon and add the paths.

Note that we need to include the header files for the assembly code as well. We just need to click the **MCU GCC Assembler** node this time.

6.8 Adding symbols

Symbols can be used for conditional building, and they are indispensable when we need to build multiple targets with different configurations.

Just for practice purposes, we need to add the `CubeIDE` symbol to all the projects for all configurations (builds). This can be done together with the above step for Including header files to the project. Assume we are in the **Paths and Symbols** pane. Do the following:

- In this pane, click the **Symbols** tab.
- Under **Languages**, choose **[GNU C]**.
- In the right pane, we should be able to see all the existing *Symbols*. Click **[Add ..]** to open a **Add symbol** dialog.
- Type `CubeIDE` in the edit field of this dialog.
- Check all the following checkboxes:
  - **Add to all configurations**.
  - **Add to all languages**.
- Click **[Ok]**.

We should be able to see `CubeIDE` is added in the list of *Symbols*.

We can add more symbols for different build so that we can perform conditional build.

7 Code navigation in CubeIDE

Code navigation can be done by **ctrl-clicking** (pressing the Ctrl key and clicking) the function name. With this, CubeIDE will bring us to the function or macro definition.

Sometimes, if the source file is too big, CubeIDE will have the **Scalability** Mode turned on, which turns off the code navigation features. For this case, we will be reminded about the *scalability* issue. We just need to change the size to 10000 in the line of **Enable scalability mode when the number of lines of the file is more than**.

If we want to change it with being reminded, it can be done below:

- Click **[Windows » Preferences]** in CubeIDE to bring out the **Preferences** dialog.
- Click the **[Editor » Scalability]** to show the **Scalability** pane.
- Change the value of **Enable scalability mode when the number of lines of the file is more than**: to 10000. Modern PCs are very fast and this setting will not create performance issues.

7.1 Skipping the `__weak` function

Many times, we see the `__weak` functions defined as placeholders for code compilation. The normal version of these functions should be defined by third-party developers or the user. For the code navigation of these function, the above approach will only lead to the weak functions, at least this is true for CubeIDE v1.13.1.

To skip the weak function to navigate to the normal function, we need to use the following approach:

- Use the above *ctrl-clicking* approach to navigate to the weak function.
- Ctrl-click the name of the function again to go to the header file where the function is declared.
- Keep ctrl-clicking the function name. Now, a **Open Declaration** dialog will open up for us to choose the file to open with.
- Choose the file where the normal function is defined to read the code.

8 Building executables with CubeIDE

Now we assume we have performed all the following tasks:

- MCU and middleware configuration and code generation.
- Project importing to CubeIDE.
- Embedding a user function hook in the generated `main` function in the `main.c` file.
- Creating all our user file folders.
- Copying or linking user files to the project in CubeIDE.
- Including folders for header files.
- Adding needed symbols.

Below, we will give the definitions of executables and build configurations first and then move on to discuss how to build them for a project using CubeIDE.

8.1 Definition executables

An executable is the output of a build system that can run on a simulator or MPB. It is mostly a .elf file. For the Keil toolchain, it is a .axf (Arm eXecutable File) file, which is similar to a .elf file.

Note that the executable .elf file is also called an **image** as it can be downloaded into the flash memory of the MCUs.

Note that there can be multiple executables for a project depending on the build configurations.

8.2 Definition of build configurations

The building of each executable is controlled by a **build configuration**, which is a file describing the various aspects of the build, including the optimization level of the build, symbols defined in the toolchain that can be used by the C preprocessor directives for conditional building.

8.3 Building executables of a project with CubeIDE

Building a project with CubeIDE is straightforward. Usually, for the learning purpose, we mainly work with the **Debug** target, which is the default.

Now, make sure we are in the *C/C++* perspective of CubeIDE.

If we want to change to another build target, we need to click the **[Arrow]** to the right of the **[Build]** button and select the configuration (for the build target) we want to build.

To build a selected configuration, we just need to click the **[Build]** button. The project will be built for us—many object files and an `elf` file will be generated.

Note that CubeIDE, like all the modern build system, uses incremental building, which means that the unchanged source file will not get build. To force a rebuild of everything, which is called clean rebuild, we need to use the context menu, as shown below:

- Right click on the title of the current project to bring up the context menu. Choose **[Clean Project]** to remove all the built file.
- Click the **[Build]** button rebuild it again.

It seems that when the included header file is changed, the change cannot be detected by the build system. As such, we need to fake a change in the related source file and then start the build. Or, we just do a clean rebuild.

9 Debugging with CubeIDE

9.1 Debugging a project with CubeIDE

Basic debugging in CubeIDE is very easy. Make sure we have SWD enabled, as shown in the Setting up SWD and SWV section of [ref-man-CubeMX], before generating the code for the project and building the project.

If this is the first time you plug in the USB cable for the ST-Link into your PC, you may see that a **ST-LINK firmware verification** dialog pops up, asking you to *Proceed with update* or not. Click the **[Yes]** button to update the firmware. It then goes to the next dialogs:

- In the **STLinkUpdate x.xx.x** dialog, click **[Open in update mode]**.
- Keep the defaults and click the **[Upgrade]** button to proceed.
- **Upgrade successful** will be displayed in the status bar of the dialog, and the new version number will be displayed in the dialog. You can close the dialog now.

Before the first debugging of a new project, we need to do a little bit setup, as shown below:

- Click on the title of the project in the **Project Explorer** view (in a left side pane in the CubeIDE window) to make sure CubeIDE associates the operations to this project.
- Click the **[Debug]** button in the toolbar. (If we hover the mouse on it, we will be able to see the target name to be used for debugging.) This will pop up an **Edit Configuration** dialog with the **Main** tab open.
- The default is good for us. We just need to make sure the **Projects** and **C/C++ Application** fields are what we wanted to see. (If we did not click the wrong project, these two fields will be correctly filled. If we see something wrong, just click **[Cancel]** and redo from the beginning.)
- Keep other default choices.
- No need to change anything in the other tabs.

- Click **[Ok]** to start the debugging. It will take a moment to load the `elf` file to the MCU on the MPB. After this is done, we will be shown a **Confirm Perspective Switch** dialog.
- We can check the **Remember my decision** checkbox or just ignore it. Click **[Switch]** to switch to the **Debug Perspective**.

Here is an alternative approach:

- Right-click on the project
- select “Debug As”
- click on “STM32 Cortex-M C/C++ Application”

By default, when we start the debugger, the target image is downloaded to the MCU.

The subsequent debugging sessions of the same project do not need the above setup. When we click the **[Debug]** button, it goes to the **Debug Perspective** directly.

One of the most powerful functionalities of a debugger is to halt the program at preset breakpoints. When the program is halted, we can inspect the values of variables, values at different memory locations, and various values of registers, including general-purpose registers, special core registers, and peripheral registers. Note that we can inspect the values of variable live using SWV based on tracing. But this is a later topic.

9.2 Toolbar buttons for debugging

Now, assume we are in the **Debug** perspective. We are shown a number of new buttons on the toolbar. The following are used often, from left to right:

- **[Reset the device and restart the debug session]**. During the debugging of a program, we can reset the device and restart the debugging again. The reset can be configured, as shown in Sec 3.5.7.2 of the user guide.
- **[Skip all breakpoints]** ([Ctrl + Alt + B]). When we run the program, the program will stop at the breakpoints so that we can inspect the interested values. With this button, we can disable these breakpoints without deleting them. We can enable them by clicking this button again.
- **[Terminate and Relaunch]**. We can terminate the running of a program and then relaunch it again by using this button.
- **[Resume]** ([F8]). We can resume (or start if this is the first running) of the program.
- **[Suspend]**. We can suspend the running of the program by clicking this button.
- **[Terminate]** ([Ctrl + F2]). We can terminate the debugging of the program and go back to the *C/C++* perspective.
- **[Disconnect]**. We can terminate the debugging of the program and go back to the *C/C++* perspective by using this button as well. Need to figure out the difference between this button and the **Terminate** button.
- **[Step into]** ([F5]). When the program is halted at a function, we can step into the function by clicking this button. When the program is halted at a common statement, we can step to the next statement by clicking this button.
- **[Step over]** ([F6]). When the program is halted at a function, we can step over the function by clicking this button. When the program is halted at a common statement, we can step to the next statement by clicking this button as well.

- **[Step return]** ([F7]). When we have stepped in a function, we can return back by clicking this button.
- **[Instruction stepping mode]**. We can go to the assembly instruction stepping mode by clicking this button.

We can use a combination of the above buttons to step through a program to see the intended values that will be discussed below.

However, stepping from the beginning of the program can be very time consuming, and we usually use **breakpoints**, discussed below, to halt the program at the intended lines of code.

9.3 Setting up breakpoints

There are two types of breakpoints, the standard breakpoint at which the program will halt unconditionally and the conditional breakpoint at which the program will halt only when a certain condition satisfies.

We can set up a (standard) breakpoint on a specific line, called target line, as detailed below:

- Click on the target line of code where we intend to create a breakpoint. Now, a blue vertical bar will appear in from of each breakable line of the function where the target line resides.
- Double-click on the blue strip in front of the target line. Now, a **breakpoint icon**, a small dot, will appear, indicating the breakpoint is created.

To remove a breakpoint, we just need to double-click on the small dot. When it disappears, the breakpoint is removed.

To **change a standard breakpoint to a conditional one**, we need to

- Right-click on the breakpoint icon and select **[Breakpoint Properties. . .]** to bring up the **Properties for C/C++ Line Breakpoint** dialog.
- In the dialog, click **[Common]** to show the dialogs in the **Common** page. Here we need to type in the needed condition in the **Condition** field. Click **[Apply and Close]**.

Note that the conditions are written in C-style, and we can use all the global and local variables when creating the conditions.

We can temperately disable all the breakpoints by clicking on the **[Skip all breakpoints]** button.

9.4 Viewing values

When we run the program by hitting the **[Resume]** button, the program will run and halt at (or more precisely, before the execution of) a breakpoint. Now, we can view various values.

Here we discuss how to see different types of values using the following views at different tab on the right pane of the **Debug** perspective:

- The **Variables** view. This view is used to show the **local** variables. We can see the values of these variables when they are in focus, which means that we have halted the program in the functions (or code block) where they are defined. All the variables in focus will be listed at the top part of the view. We can see the summary information of a specific variable at the bottom of this view by selecting it in the top part. We can also see the detailed info of an array type variable by expending it.

- The **Expression** view. This view is used to show the **global** variables. To see the values in this view, we need to manually add expressions here. Note that *expression* here means we don't have to use the name of the variables defined in the program. We can use any legitimate expressions based on variables here. For example, if we have defined `i` in the program, we can display `2*i` here.
- The **Registers** view. This view is used to show the contents of all general-purpose registers. One of the issues here is the format of display. The default is Decimal, and many times we need to use Hexadecimal. To display in Hexadecimal format, we need to hold the [Ctrl] key and click on the registers we want to change display format to select them. Then, we use right click to bring out the context menu and choose **[Number Format » Hex]**.
- The **SFRs** view. Here, we can see the contents of all special-function registers. These are the registers for Cortex-M core and peripherals.

Note that we don't have to go to **Variables** or **Expressions** views to see the variables. We can see them by hovering the mouse on the variable when the program halts.

9.5 Using live expression window

We can use this window do observe the expressions based on **global variables** in real time, which is especially useful when we want to see something in read time without halting the program.

9.6 Inspecting memory

When the programs halts at a breakpoint or paused with step through, we can inspect the contents in the memory as well.

This is done in the **Memory** view at the bottom pane of the **Debug** perspective. Now, click **[Memory]** to display this view.

To add a new *Memory Monitor*, click the **[Add Memory Monitor]** button and input the address. If we don't know the exact address, we start with `0x20000000`, the start address of the memory. Now, we will have a new **Memory** Monitor created. There can be multiple renderings for the same monitor.

- The `<Traditional>` rendering has two sections. The first is used to display 32-bit numbers in hexadecimal, and the second is used to display each byte in ASCII mode. If we have strings, this is the best choice.
- The `<Hex>` rendering is used to display the numbers in **bytes** although four bytes are grouped together. This is a preferred way to display 8-bit numbers in hexadecimal.
- The `<Hex Integer>` rendering is used to display numbers in **words** in decimal.
- The `<Signed Integer>` and `<Unsigned Integer>` renderings are used to display the numbers in `int32_t` and `uint32_t`, respectively.

10 Working multiple targets in CubeIDE

A target is an executable `.elf` that is built with special configurations for a special purpose. For example, we usually have at least two targets for a single project. One is the Debug, where we do not want to have code optimizations so that we can see the one to one mapping of the C source code to the asm code. The other is Release, where we want to have optimizations for size of code or the speed of execution.

Sometimes, we need to create even more targets for a project. For example, we can have one target for each set of unit tests of the functions used in the project.

10.1 Creating an additional target configuration

According to the CubeIDE manual, a target is called **build configuration**. We use the term **target** as it fit to the MP application better.

Do the following to create and configure a new target in an existing CubeIDE project, which has two existing targets already:

- Right-click the project in the **Project Explore** to bring up the context menu.
- In the context menu, click **[Build Configurations » Manage. . .]** to bring up a **Manage Configuration** dialog.
- In the dialog, click **[New. . .]** to go to the **Create New Configuration** dialog.
- In this new dialog, fill in the **Name** filed, say *Unity*. Optionally, we can create a **Description** for the build. Also, we need to choose an item in the **Copy settings from** group. We usually choose **Existing configuration** and select **Debug**. Click **[Ok]**.

10.2 Suggested target names

We use the following suggested target names in our projects:

- `Debug` for the normal App. We don't have to worry about using `Release` for class projects.
- `DebugSoln` for the normal App with the solution version of some functions. This is usually used for the TAs.
- `Unity` for the normal unit test.
- `UnitySoln` for the normal unit test with the solution version of some functions. This is usually used for the TAs as well.

10.3 Conditional compilation in multiple targets

In the source file folder of every project, we have a `_xxx_main.c` file where we start the execution of the user code. We can add conditional compilation directives there to invoke different code to build a specific targets for a project. This will be something like

```c
#if defined(UNIT_TEST)
    extern void run_unity(void);
    run_unity();
#else
    extern void run_app(void);
    run_app();
#endif
```

Here, `UNIT_TEST` is the symbol we have defined only in the `Unity` target. This can be done as detailed in the Adding symbols section above.

Note that the `run_unity()` and `run_app()` functions should be defined in their corresponding source files, which should be included in the project. Within these two functions, we can run all our test or application code.

10.4 Excluding a file from a certain build target

With multiple build, we sometimes need to exclude some files from a certain build target. This can be done using the steps below:

- Right-click the file to exclude from some build targets in the **Project Explore** to bring up the context menu.
- In the context menu, click **[Resource Configurations » Exclude from Builds. . .]**. A **Exclude from build** dialog will pop up.
- In the dialog, check the build targets that we want to exclude the file. Click **[Ok]**.

10.5 Setting active build target

Note that when building a specific target repeatedly, we can set that target as active, which can be built by default. The setup can be done in the steps below:

- Right-click the project in the **Project Explore** to bring up the context menu.
- In the context menu, click **[Build Configurations » Set Active » Target]**. Here **Target** is the target we want to set to be active.

10.6 Creating a new debugging configuration

By default, CubeIDE has the debugging configuration tied to the Debug target. If we need to debug a new target, we need to specifically create and set up its debug configuration. This can be done via the following steps, illustrated using an example with a target named `Uart`.

- Make sure the target has been set up as **active**.
- Click the triangle beside the Debug button or right-click the project title and select **[Debug As]** in the context menu. Then choose **[Debug Configurations]**. The **Debug Configurations** dialog will pop up. (This may be doable by clicking **[Debug As » xxx]**. Need to try out.)
- In this dialog, we need to do THREE actions:
  - Click the **[Duplicate the currently selected configuration]** button at the top of the left pane.
  - Providing an appropriate name for the new debug configuration in the **[Name]** field. Here, we can just use the executable name plus `(UART)`.
  - Choosing the correct executable in the **C/C++ Application** field. This can be done by clicking the **[Search Projects. . .]** button. We will be shown all the executable `.elf` files in a new dialog. Choose the file in the `Uart` folder and press **[Ok]**, and we will be back to the **Debug Configurations** dialog.

  Now, hit **[Debug]** to start the debugging of the `Uart` target.

Now, a debug configuration is created for this `Uart` target. Next time we need to debug this target, we just need to click the debug configuration title.

11 Adding more libraries or functionalities

11.1 Including compiled DSP library

ARM provides a lot of DSP functions and corresponding header files. When we use the data types of `q31_t` and `q15_t`, we need to include `arm_math.h` in the project. We also need to include the *compiled* library files (to avoid recompilation).

Unfortunately, these files cannot be configured in CubeMX directly. We need to configure them in CubeIDE. Below are the steps we need to do this.

First, we need to define a symbol `ARM_MATH_CM4` so that the needed library can be used.

Then, we need to prepare the files to include in the project. These files are saved in the directory of the Cube repositories. For example, we can find them in the following folders:

- Header files: `C\Users\xxx\STM32Cube\Repository\STM32Cube_FW_F4_V1.27.1\Drivers\CMSIS\DSP\Include`.
- Compiled library: `C\Users\xxx\STM32Cube\Repository\STM32Cube_FW_F4_V1.27.1\Drivers\CMSIS\Lib\GCC`.

We need to copy the header files to the `lib/dsp/Include` folder and the compiled files to the `lib/dsp/lib_gcc` folder. Now, these files are ready to be included in the project. The header files are included in the same way as other header files.

Note that we need to link the `dsp` folder to the `lib_src` folder to use the library files.

To use the compiled DSP library, we need to

- Right-click the project and choose **[Properties]** in the **Context Menu**. A **[Properties for . . .]** dialog will pop up.
- In the left pane of the dialog, click **[C/C++ Build » Settings]**. The right pane will be the **Settings** pane.
- In this pane, click the **Tool Settings** tab.
- Under **MCU GCC Linker**, choose **[Libraries]**.
- In the right pane, configure the following two sections:
  - Add *[arm_cortexM4lf_math]* in the **Libraries (-l)** section.
  - Add `${workspace_loc:/${ProjName}/lib_src/dsp/lib_gcc}` in the **Library search path (-L)** section. (This can be done by clicking the **[Workspace. . .]** button and clicking the string of folders.)
- Click **[Apply and Close]**.

The about steps are adapted from https://community.st.com/s/article/configuring-dsp-libraries-on-stm32cubeide.

11.2 Using FPU and floating-point functions

When we use floating-point operations in the project, we need to choose how to perform the floating-point operations—using the FPU or software. We also need to choose the floating-point application binary interface (ABI). To print out or scan to floating-point numbers, we need to include the needed library as well.

The above can be done by using the following steps:

- Right-click the project and choose **[Properties]** in the **Context Menu**. A **[Properties for . . .]** dialog will pop up.
- In the left pane of the dialog, click **[C/C++ Build » Settings]**. The right pane will be the **Settings** pane.
- In this pane, click the **Tool Settings** tab.
- Click **[MCU Settings]**.
- Choose the following two:
  - **Floating-point unit**: Choose *None* if we do not want to use the FPU. Otherwise, choose *FPv4-SP-D16*. (Version 4, single precision calculation, 16 double precision registers—for data transmission only.)
  - **Floating-point ABI**: Choose *Hardware implementation* if we have chosen to use the FPU.
- Check the following two:
  - **Use float with printf from newlib-nana**.
  - **Use float with scanf from newlib-nano**.
- Click **[Apply and Close]**.

12 Including source files in assembly source files

Sometimes, we need to include source files in assembly source files. For example, we can have a number of source files using some macros defined in a common file. In this case, we need to use the `.include` directive to include the definition file. Assume the macro file is called `mp_macros.txt`. Then, we need to use `.include "mp_macros.txt"` in the source file before the macros will be used.

For the compiler to be able to find the macro file, we need to put it in the include path for the Assembly. This is done in the same way for C.

13 Managing a FreeRTOS-based project

As we mentioned in the ref manual of [ref-man-CubeMX], we will use the native FreeRTOS API, and we need to manage the project manually. This can be done by the following steps.

13.1 Creating folders to hold library files

We will have the following subfolders in the `library` folder.

13.1.1 Creating a folder to hold the source files of FreeRTOS core

We create the following folder to hold the source files of the FreeRTOS core:

```
freertos_core (folder)
    corutine.c
    event_groups.c
    list.c
    manifest.yml
    queue.c
    stream_buffer.c
    tasks.c
    timers.c
```

Note that we only need to create this folder once which can then be copied to each project.

13.1.2 Creating a folder to hold the include files of FreeRTOS core

We create the `freertos_include` folder to hold the include files located in the `FreeRTOS/Source/include` folder.

Again, we only need to create this folder once which can then be copied to each project.

13.1.3 Creating a folder to hold the portable files of FreeRTOS

We create the `freertos_portable` folder to hold the `heap_4.c` file and the toolchain and architecture specific files. For example, we use the following file structure:

```
freertos_portable (folder)
    GCC_ARM_CM4F (folder)
        port.c
        portmacro.h
    heap_4.c
```

Note that `GCC_ARM_CM4F` stands for the GCC toolchain and ARM Cortex-M4 architecture. If we use other toolchain or other architecture, we need to use the corresponding files, which can be found in the `Source/portable` folder of FreeRTOS.

Note also that `heap_4.c` is the mostly used heap file. If we use another heap file, we need to put the corresponding file here instead of `heap_4.c`. The heap files can be found in the `MemMang` folder of the `Source/portable` folder.

For most projects, we only create this folder once.

13.1.4 Creating a folder to hold sample `FreeRTOSConfig.h` files

Each project will have a unique `FreeRTOSConfig.h` file. We can create a `freertos_config` folder to hold the collection of these configure files.

13.2 Managing a FreeRTOS-based project in CubeIDE

Once we have the above library source files, we just need to add them in the project. Here, we list the folders and files we need to manage without giving the details steps of how to do them, as they are the same as what we have discussed in the Managing a project section.

To include the header files, we can use the same trick we have used before.

The only exception is the `FreeRTOSConfig.h` file. We need to have a folder to hold it. The best place may the the `rtos_config_mpb` folder under the root of the project. We just need to include this folder in the include path.

Below is an illustration of the contents of the `FreeRTOSConfig.h` file

13.3 The contents of the configuration file

```
extern uint32_t SystemCoreClock;

#define configUSE_PREEMPTION              1
#define configUSE_IDLE_HOOK               1
#define configUSE_TICK_HOOK               0 // 1
#define configCPU_CLOCK_HZ                ( SystemCoreClock )
#define configTICK_RATE_HZ                ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES              ( 5 )
#define configMINIMAL_STACK_SIZE          ( ( unsigned short ) 130 )
#define configTOTAL_HEAP_SIZE             ( ( size_t ) (20 * 1024) ) //( 75 * 102
    4 ) )
#define configMAX_TASK_NAME_LEN           ( 10 )
#define configUSE_TRACE_FACILITY          1
#define configUSE_16_BIT_TICKS            0
#define configIDLE_SHOULD_YIELD           1
#define configUSE_MUTEXES                 1
#define configQUEUE_REGISTRY_SIZE         8
#define configCHECK_FOR_STACK_OVERFLOW    0 // 2
#define configUSE_RECURSIVE_MUTEXES       1
#define configUSE_MALLOC_FAILED_HOOK      1
#define configUSE_APPLICATION_TASK_TAG    0
#define configUSE_COUNTING_SEMAPHORES     1
#define configGENERATE_RUN_TIME_STATS     0
```

14 Other useful operations in CubeIDE

14.1 Importing multiple projects into a folder in CubeIDE

We can follow the steps below to import multiple projects in a folder in CubeIDE for better project organization:

- Create a folder in the workspace.
- Cope the projects into the folder.
- In CubeIDE, click “File -> Open project from file system”.
- Change to the folder and follow the hints.

14.2 Opening multiple workspaces

CubeIDE is based on Eclipse. As such, we can run a multiple copies of CubeMX, each is for a specific workspace. This way, we can better organize multiple projects without overcrowding a single workspace.

14.3 Saving a project to a new one

See the post of Rajat at https://community.st.com/s/question/0D53W000002hFWqSAM/what-is-the-best-way-to-duplicate-a-project-in-a-workspace-in-stm32cubeide

Note that we can write a Python code to do this automatically.

15 Issues to avoid

15.1 Resource exists with a different case

Sometimes, we see the following error: "A resource exists with a different case." This is caused by having the save name (with different cases) of two folders. For example, if we have a folder name as `soln`, and we have a build named `Soln`, we will see this trouble. The solution is to use different names. We can easily change the build to `Solns` to avoid the trouble.

15.2 Old builds

When we change to another build target, we need to **clean all** the builds. Otherwise, there may be some object files of old build used, which leads to troubles that evades the debugging.

15.3 Warning for `cannot find entry symbol`

Sometimes, when we change the configuration of the project in CubeMX and regenerate code, we will see an warning when rebuilding the target:

```
cannot find entry symbol Reset_Handler; defaulting to 0000000008000000
```

This is usually cause by the issue of CubeMX/CubeIDE which by accident excludes some of the folders generated by CubeMX. This issue can be handled by checking each generated folder to make sure the checkbox before **Exclude resource from build** is not checked.

For example, let's take a look how to handle the `Application/Startup` folder.

- Right-click the `Startup` folder. In the context menu, click **[Properties]**. The **Properties for Startup** dialog will pop up.
- We will see that **C/C++ Build » Setting** is automatically selected. In the right pane, we should be able to see the **Exclude resource from build** checkbox. Make sure it is not checked.

We can check each generated folder to make sure all these boxes are not checked. The project should be able built without the warning.

16 How to patch a project

When a team of developers work on a project together, the newest changes are pushed to the repository in GitHub and everyone in the team can do a pull to synchronize with the new changes. Yet, this is not always the case. Sometimes, we need to use a patch file to update multiple sources. This is especially true when the users do not use a version control system. For example, when we use SEGGER SystemView, we need to patch the FreeRTOS files so that the FreeRTOS code can be changed to embed SystemView calls to trigger the recording of the events, and this is conveniently done using a patch file, which contains the diffs of multiple files.

Below is a sample section of the patch file:

```
@@ -5311,12 +5325,14 @@ static void prvAddCurrentTaskToDelayedLi
                  {
                      /* Wake time has overflowed.  Place this item in the
                         overflow
```

```
                          * list. */
+                        traceMOVED_TASK_TO_OVERFLOW_DELAYED_LIST();
                         vListInsert( pxOverflowDelayedTaskList, &( pxCurrentTCB->xStateListItem ) );
                     }
                     else
                     {
                         /* The wake time has not overflowed, so the current block list
                          * is used. */
+                        traceMOVED_TASK_TO_DELAYED_LIST();
                         vListInsert( pxDelayedTaskList, &( pxCurrentTCB->xStateListItem ) );

                         /* If the task entering the blocked state was placed at the
@@ -5345,11 +5361,13 @@ static void prvAddCurrentTaskToDelayedLi

                 if( xTimeToWake < xConstTickCount )
                 {
+                    traceMOVED_TASK_TO_OVERFLOW_DELAYED_LIST();
                     /* Wake time has overflowed.  Place this item in the overflow list. */
                     vListInsert( pxOverflowDelayedTaskList, &( pxCurrentTCB->xStateListItem ) );
                 }
                 else
                 {
+                    traceMOVED_TASK_TO_DELAYED_LIST();
                     /* The wake time has not overflowed, so the current block list is used. */
                     vListInsert( pxDelayedTaskList, &( pxCurrentTCB->xStateListItem ) );
```

Here is a brief explanation of the above code:

- The line with `@@ -n1,n2+n3,n4@@ abc` starts a new diff. `-n1` is the line number in the original file, and `+n3` the line number of the new file. `n2` is the number of lines starting from `n1` that are listed below, including blank lines; and `n4` is the number of lines starting from `n3` that are listed below. `abc` is the function where the change happens.
- Those lines starts with a single space are context lines, which are used to provide the context information if the line numbers are not accurate.
- Those lines start with `-` will be removed in the new file, and those with `+` will be added.

For more details, see https://www.cocoanetics.com/2011/12/how-to-make-and-apply-patches/.

To patch in CubeIDE, we need to perform the following steps:

- Read the patch file and determine the folder structure of the library files to be patched. Make sure the structure of the library files are the same as specified in the patch file. Include the library files in the project.
- Right click the project. In the context menu, click **[Team » Apply Patch. . .]**.
- In the **Apply Patch** dialog, click **[Browse. . .]** to find the patch file. Select the patch file and click **[Next]**.

- Select the folder of the library files to apply the patch. Click **[Next]**. We will be able to see related files are opened. We may need to play with the level of **Ignore leading path name segments** to find the files to patch.
- We can click each file to check the patch contents.
- When everything is fine, we can click **[Finish]** to finish the patching.

Note that if the original files have changed somehow, CubeIDE may not be able to find the code segments to patch. If this is the case, we need to do a manual patch—editing.

17 ITM redirection

ITM is an optional application-driven trace source that supports printf style debugging to trace operation system and application events.

Here is the code for ITM redirection:

```
//////////////////////////////////////////////////////////////////////////////
// printf like feature using ARM Cortex M3/M4/M7 ITM functionality
// This function will not work for ARM Cortex M0/M0+
//////////////////////////////////////////////////////////////////////////////


//Debug Exception and Monitor Control Register base address
#define DEMCR                        *((volatile uint32_t*) 0xE000EDFCU )

/* ITM register addresses */
#define ITM_STIMULUS_PORT0           *((volatile uint32_t*) 0xE0000000 )
#define ITM_TRACE_EN                  *((volatile uint32_t*) 0xE0000E00 )

void ITM_SendChar(uint8_t ch)
{

    //Enable TRCENA
    DEMCR |= ( 1 << 24);

    //enable stimulus port 0
    ITM_TRACE_EN |= ( 1 << 0);

    // read FIFO status in bit [0]:
    while(!(ITM_STIMULUS_PORT0 & 1));

    //Write to ITM stimulus port0
    ITM_STIMULUS_PORT0 = ch;
}
```

Note that this is provided in `SEGGER_RTT_Syscalls_GCC.c` file in the `Syscalls` folder.

18 Tricks for the editor and code navigation

(https://www.youtube.com/watch?v=4wT9NhlcWP4&t=530s)

18.1 Code navigation

18.1.1 Symbol hyperlink

In the C/C++ perspective, we can hold Ctrl and then click on a function or macro to see the definition of the interested. To go back, we click the **Back** icon on the icon bar.

18.1.2 Brace navigation

We can easily see where the code starts and ends using brace navigation. To do this, we just need to click anywhere to the right of the brace. The matching brace will be highlighted.

18.1.3 Outline view

We can view the outline of the file in the editor in the **Outline** view to the right of the editor.

- To open it up the view, we need to click **Window » Show View » Outline**.
- Here are the tricks of using this view.
  - When we click on a line in this view, the detailed code will be shown in the editor view.
  - When Trick 2 is to use Ctrl + O to display the outline of the file so that we can see the functions defined in a source file. A related trick is to use the Outline view of the editor.

18.1.4 Include Browser

Include Browser is a view of the editor.

To open up the view, we need to click **Window » Show View » Include Browser**.

Here is a two-step approach:

- Click the file in the editor view and it will show in the **Outline** view as well.
- In the **Outline** view, right-click a line and then choose **Open Browse View**.

This is especially useful to see the include files of the `main.c` file.

We can open a header file by double-clicking on the file name in this view.

18.1.5 Macro Expansion Browser Popup

When we hover the cursor over the macro, the macro value will be shown in the popup.

18.1.6 Type Hierarchy Popup

When we hover the cursor over the *type*, the popup will show. If we need to see the details of the type, we can move the cursor down or press F2.

18.2 Code editing

18.2.1 Using spaces instead of tabs

- Click **[Window » Preferences]**.
- In the **Preferences** dialog, click **[C/C++ » Code Style » Formatter » Edit. . .]**.

- In the new page of the dialog, type a new **Profile name**, say `erau_mp`. (This is based on the **K&R [built-in]** profile.)
- In the **Indentation** tab of the same page, change the **Tab policy** to *Spaces only* in the **General Settings** section.
- Click **OK** to save the changes.

18.2.2 Formatting source code

This can be done in one of the following two ways:

- Press **Ctrl+Shift+F** in the active editor tab.
- Right-click in the editor. In the context menu, click **[Source » Format Source]**.

19 Debug and trace

19.1 Debug / trace interfaces

For Cortex M3/M4/M7/M33, we have the following debug interfaces:

- JTAG / SWD. SWD just use less wires than the traditional JTAG.
- Track port with serial wire viewer

19.2 Serial wire viewer (SWV)

SWV adds:

- Real-time trace using the SWD port and the SWO pin.
- Advanced debugging without halting the MCU.

The data can be:

- Plotted in graphs
- Logged in files.

19.3 What we can do with SWV?

- Data trace and live watch or timeline graph. Only for global variables.
- `printf()` redirection
- Timing measurement
- Exception log and timeline graph
- Statistical profiling
- Event notification

19.4 Code analyzing tools

19.4.1 Build analyzer

We can check the code size using this tool.

19.4.2 Static stack

We can check each function size using this tool.

20 Using ITM

20.1 Basic setups

For debug, we need to choose **Trace asynchronous sw** to use SWO.

In the code, we need to define

```
#define ITM_Port32(n) (*((volatitle uint32_t *)(oxE0000000+4*n)))
```

We can use the above with

```
TIM_Port32(31) = 1;
printf("GPIO Init Done\n");
TIM_Port32(31) = 2;
```

To redirect printf, we can use the following code:

```
int _write(int file, char *prt, int len) {
    int DataIdx;
    for (DataIdx = 0; Datadx < len; DataIdx++) {
        ITM_SendChar(*ptr++);
    }
    return len;
}
```

We also need to enable the SWV in the debug configuration dialog. Make sure the clock is the same as what we have in clock tree.

In the debug mode, we can see the change of the variable using the **Live Expression** view.

20.2 Viewing SWV data trace

We need to open this view in the **Debug** perspective by

- Clicking **Windows » SWV » SWV Data Trace** to show the view.
- Making the view a big larger to see more info.
- Clicking the **Configure trace** icon on the view-name bar to the right of the view tab.
- Setting the trace variables or address. By default, we use port 0.
- Starting the trace by clicking the **Start Trace** icon to the right of the **Configure Trace** icon and then starting the program.

20.3 Viewing SWV data trace timeline graph

We need to open this view in the **Debug** perspective by

- Clicking **Windows » SWV » SWV Data Trace Timeline Graph** to show the view.

If we have collected the values using other views, the graph will shown. Otherwise, we need to start the trace first.

20.4 Redirecting `printf` to SWV ITM data console

We need to open this view in the **Debug** perspective by

- Clicking **Windows » SWV » SWV ITM Data Console** to show the view.

By default, we use Port 0.

Again, we need to start trace before running the program.

The results of `printf` will be printed in this console.

20.5 Viewing SWV trace log

We need to open this view in the **Debug** perspective by

- Clicking **Windows » SWV » SWV Trace log** to show the view.

We need to go to the **Configure trace** dialog to configure the port used for the trace log. Earlier, we have used port 31. Now, we need to enable this port. We need also disable Port 0 so that we will not be distracted.

To make the trace log clean, we will remove all the **Trace Events** at the top of this dialog.

Now, we can see the trace log with timestamp so that we can see the time between two trace events. This way, we can record the time spend in each segment of code, including the functions.