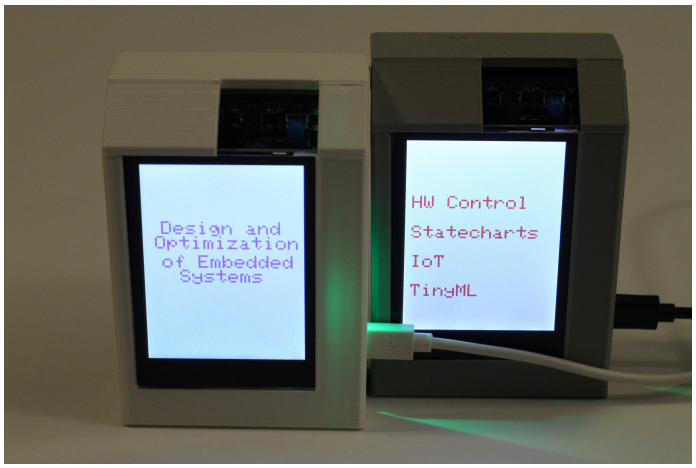
Module 1

Design and Optimization of Embedded and Real-time Systems

1 Welcome to the Course



In this course, you'll learn about modern software architecture and advanced design techniques for embedded systems. You'll gain hands-on experience in developing reactive, responsive and reliable systems with an industrial-strength application framework. You'll also learn practical optimization techniques to integrate various hardware modules, including IMU sensors, WiFi and LCD.

Main topics include:

- Reactive and asynchronous software architecture
- C++ for embedded systems `
- Open-source development environments
- Object-oriented statechart design, patterns and framework
- Hardware interface with UART, I2C, SPI, DMA and interrupts.
- Introduction to IoT and TinyML.

2 Introduction to Software Design

2.1 My First Job

When I first came out to work as an embedded software engineer, a senior told me "Your job of programming is simple. You just need to tell the micro-controller to do what you want it to do."

Obviously it is an over-simplification. It does, however, brings up a couple interesting questions:

1. Do we know ourselves what we want to computer to do? How are we certain that we really know it? Is our knowledge complete and free of contradictions? How can we capture this knowledge?
2. Assuming the answer to the first question above is "yes", how do we transfer this knowledge to the computer in a precise and straight-forward manner?

In the age of AI and machine learning, these might not be the right questions to ask anymore. The questions might be inverted to become "How can a computer tell us what it wants to do?"

Indeed in the fields of safety-critical systems, AI still presents many challenges. For the time being, our questions above are still relevant, and lets take a deeper look at them.

2.2 Two Key Concepts

2.2.1 Knowledge Representation

The first set of questions are about *knowledge representation.* They are very tricky since knowledge is very abstract. Human thoughts are very flexible but often are not precise enough. We do better in reasoning about sequential logic (like if ... then, do ... while, wait until ... etc.) than reasoning about things that can happen in parallel.

The knowledge we are talking about here boils down to system behaviors. Since human minds aren't very good intuitively to reason about it, we need a tool to help us, a tool that helps us define behaviors, capture them and visualize them.

It is such a difficult problem that it took a computer scientist to find a solution. It was David Harel who invented statecharts back in late 1980s to solve this exact problem – how to specify system behaviors for embedded systems (a fighter jet in his case). We will look into the rather interesting history behind it later.

Share with the class: Have you used or heard of finite-state-machines or statecharts?

2.2.2 Knowledge Mapping

Now let's address the second question – how do we transfer our knowledge to the computer in a precise

and straight-forward manner? The short answer is of course by programming. However as anyone with some programming experience will tell, if you simply program your sequential thoughts into code it won't take long for your program to evolve into a big messy ball of spaghetti. This is a *mapping* problem. We need an effective tool to map system behaviors (specification or knowledge) into executable code. Such tools pretty much has existed as long as statecharts. However they are very expensive. It is not until the advent of Quantum Framework/Platform by Miro Samek in 2000 that made it truly affordable and practical to everyone.

Today there are many other frameworks available, including:

1. https://www.itemis.com/en/yakindu/state-machine/
2. https://github.com/google/statechart
3. The Boost Statechart Library (The Boost Statechart Library - Overview - 1.78.0)
4. Qt SCXML (Qt SCXML 5.15.8)
5. xstate (Javascript)
6. Sismic (Python)

QP remains a good candidate for embedded systems due to its small size, performance and few dependencies.

Now we attempt to answer the question of "What is software design?"

Software design is:

1. A complete and precise specification of system behaviors that fulfills the requirements.

   We call it the *behavioral model* which is the representation of our knowledge about what the system is required to do. Statecharts are simple yet powerful tools that help us achieve this.

2. A software architecture that supports the execution of the model.

   Quantum Platform (QP) is one of such tools that help developers map or translate statecharts to C/C++ code effectively.

Share with the class: Have you used or heard of any of the above tools?

2.3 A Couple Myths

2.3.1 First Myth

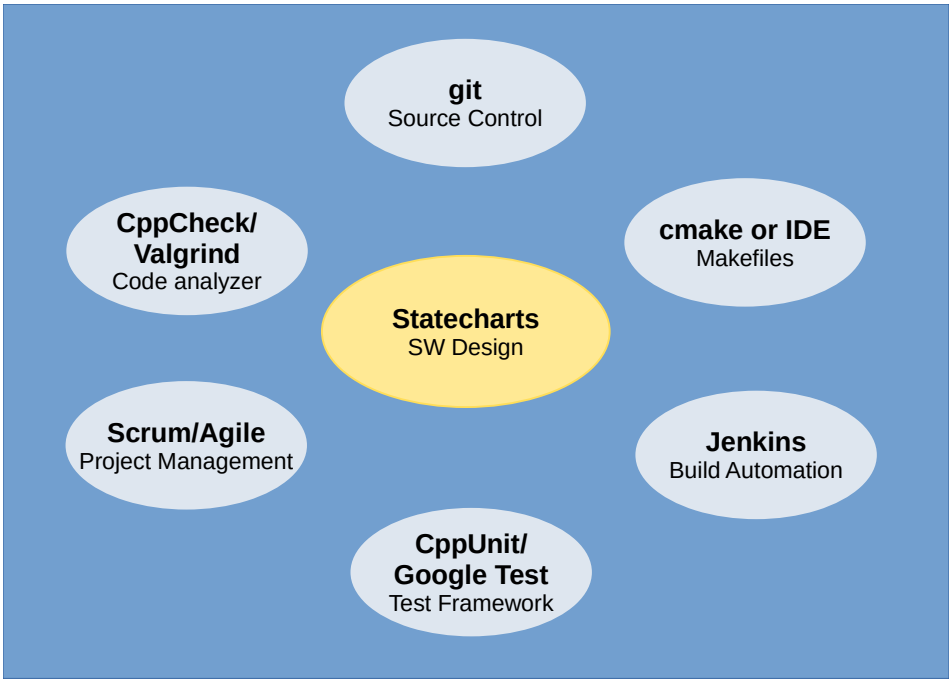
The first myth is that a design would automatically arise through constant integration, refactoring and testing. There is no doubt that *some* design would come up through the above process. It cannot be argued either that all of the above process are critical to a successful software project. However it is

doubtful whether a good, reliable and maintainable design would implicitly come about without employing solid and dedicated design methods and tools.

We are fortunate as software developers nowadays as we have seen a proliferation of many great tools to help us with our development process. To name a few, they include:

1. Git for source control and code review
2. CMake for configuration and build control
3. Jenkins for build automation
4. Unit test frameworks for unit testing
5. Agile and Jira for project management
6. Python for various automation.
7. Static and dynamic code analysis tools.

As said all of the above are very useful tools. They may yield pieces of beautiful code but there is no guarantee that when working together they will produce a reliable and maintainable system as a whole, or a system that meets its specifications or requirements. In other words, they are great tools that *support* our development process, but they do not replace the center piece of the picture which is ***design***. Software development is much more than coding and testing. The design part is often casually done or totally ignored.



Share with the class: What are your favorite software design tools?

2.3.2 Second Myth

The second myth is that embedded systems were equivalent to resource constrained systems, i.e. those with limited computational power and memory. With the advent of low-cost yet powerful platforms such as Raspberry Pi or Beaglebone (GHz CPU with GB of memory), we no longer need to apply techniques specialized for embedded systems.

While it is true that many embedded systems are resource constrained, and many techniques we have learned focus on optimization, resource is *not* the defining property of embedded systems. As we shall see in the upcoming story, embedded systems are all about *control, behaviors, determinism and reliability*.

If the system you are building involves "taking some actions when something occurs while it is in certain conditions", it is qualified as an embedded system regardless to whether it is running on a 64-bit 3 GHz Linux platform processor or an 8-bit 20MHz bare-metal microcontroller.

Share with the class: What are your favorite embedded systems?

3 Stories

3.1 Statecharts

This is the story of statecharts told by the inventor David Harel in his paper:

Statecharts in the making: a personal account (weizmann.ac.il)

Here are some of the interesting quotes from it:

December 1982 to mid 1983: The Avionics Motivation

We cut now to December 1982. At this point I had already been on the faculty of the Weizmann Institute of Science in Israel for two years. One day, the same Jonah Lavi called, asking if we could meet. In the meeting, he described briefly some severe problems that the engineers at IAI seemed to have, particularly mentioning the effort underway at IAI to build a home-made fighter aircraft...

And so, starting in December 1982, for several months, Thursday became my consulting day at the IAI. The first few weeks of this were devoted to sitting down with Jonah, in his office, trying to understand from him what the issues were.

An avionics system is a great example of what Amir Pnueli and I later identified as a reactive system [HP85]. The aspect that dominates such a system is its ***reactivity; its event-driven, control-driven, event-response nature, often including strict time constraints, and often exhibiting a great deal of parallelism****. A typical reactive system is not particularly data intensive or calculation-intensive. So what is/was the problem with such systems? In a nutshell, it is the need to provide a* ***clear yet precise description of what the system does, or should do. Specifying its behavior is the real issue****.*

Here is how the problem showed up in the Lavi. The avionics team had many amazingly talented experts. There were radar experts, flight control experts, electronic warfare experts, hardware experts, communication experts, software experts, etc. When the radar people were asked to talk about radar, they would provide the exact algorithm the radar used in order to measure the distance to the target. The flight control people would talk about the synchronization between the controls in the cockpit and the flaps on the wings. The communications people would talk about the formatting of information traveling through the MuxBus communication line that runs lengthwise along the aircraft. And on and on. Each group had their own idiosyncratic way of thinking about the system, their own way of talking, their own diagrams, and their own emphases.

Then I would ask them what seemed like very simple specific questions, such as: "What happens when this button on the stick is pressed?" In way of responding, they would take out a two-volume document, written in structured natural language, each volume containing something like 900 or 1000 pages. In answer to the question above, they would open volume B on page 389, at clause 19.11.6.10, where it says that if you press this button, such and such a thing occurs... I would say: "Yes, but is that true even if there is an infra-red missile locked on a ground target?" To which they would respond: "Oh no, in volume A, on page 895, clause 6.12.3.7, it says that in such a case this other thing happens." This to-and-fro Q&A session often continued for a while, and by question number 5 or 6 they were often not sure of the answer and would call the customer for a response... By the time we got to question number 8 or 9 even those people often did not have an answer!

In my naïve eyes, this looked like a bizarre situation, because it was obvious that someone, eventually, would make a decision about what happens when you press a certain button under a certain set of circumstances. However, that person might very well turn out to be ***a low-level programmer whose task it was to write some code for some procedure, and who inadvertently was making decisions that influenced crucial behavior on a much higher level****. Coming, as I did, from a clean-slate background in terms of avionics... this was shocking. It seemed extraordinary that this talented and professional team did have answers to questions such as "What algorithm is used by the radar to measure the distance to a target?", but in many cases did not have the answers to questions that seemed more basic, such as "What happens when you press this button on the stick under all possible circumstances?".*

In retrospect, the two only real advantages I had over the avionics people were these: (i) having had no prior expertise or knowledge about this kind of system, which enabled me to approach it with a completely blank state of mind and think of it any which way; and (ii) having come from a slightly more mathematically rigorous background, making it somewhat more ***difficult for them to convince me that a two-volume, 2000 page document, written in structured natural language, was a complete, comprehensive and consistent specification of the system's behavior****.*

...let us take a look at an example taken from the specification of a certain chemical plant.

Section 2.7.6: Security “If the system sends a signal hot then send a message to the operator.”

Section 9.3.4: Temperatures *"If the system sends a signal hot and T >60^0, then send a message to the operator."*

Summary of critical aspects *"When the temperature is maximum, the system should display a message on the screen, unless no operator is on the site except when T <60^0."*

*Despite being educated as a logician, I've never really been able to figure out whether the third of these is equivalent to, or implies, any of the previous two… But that, of course, is not the point. The point is that these excerpts were obviously written by three different people for three different reasons, and that such large documents get handed over to programmers, some more experienced than others, to write the code. I**t is almost certain that the person writing the code for this critical aspect of the chemical plant will produce something that will turn out to be problematic in the best case — catastrophic in the worst**. In addition, keep in mind that these excerpts were found by an extensive search through the entire document to try find where this little piece of behavior was actually mentioned. Imagine our programmer having to do that repeatedly for whatever parts of the system he/she is responsible for, and then to make sense of it all.*

1983: Statecharts Emerging

The goal was to try to find, or to invent for these experts, a means for simply saying what they seemed to have had in their minds anyway. The goal was to find a way to help take the information that was present collectively in their heads and put it on paper, so to speak, in a fashion that was both well organized and accurate.

*I became convinced from the start that the notion of a state and a transition to a new state was fundamental to their thinking about the system... They would repeatedly say things like, "When the aircraft is in air-ground mode and you press this button, it goes into air-air mode, but only if there is no radar locked on a ground target at the time". Of course, for anyone coming from computer science this is very familiar: what we really have here is a finite-state automaton, with its state transition table or state transition diagram. Still, it was pretty easy to see that just having one big state machine describing what is going on would be fruitless, and not only because of the number of states, which, of course, grows exponentially in the size of the system. Even more important seemed to be the pragmatic point of view, whereby **a formalism in which you simply list all possible states and specify all the transitions between them is unstructured and non-intuitive; it has no means for modularity, hiding of information, clustering, and separation of concerns,** and was not going to work for the kind of complex behavior in the avionics system. And if you tried to draw it visually you'd get spaghetti of the worst kind. It became obvious pretty quickly that it could be beneficial to come up with some kind of **structured and hierarchical extension of the conventional state machine formalism**.*

...After a few of these meetings with the avionics experts, it suddenly dawned on me that everyone around the table seemed to understand the back-of-napkin style diagrams a lot better and related to them far more naturally. The pictures were simply doing a much better job of setting down on paper the

system's behavior, as understood by the engineers, and we found ourselves discussing the avionics and arguing about them over the diagrams, not the statocols... I gradually stopped using the text, or used it only to capture supplementary information inside the states or along transitions, and the diagrams became the actual specification we were constructing...

This was how the basics of the language emerged. ***I chose to use the term statecharts for the resulting creatures, which was as of 1983 the only unused combination of "state" or "flow" with "chart" or "diagram".***

8. The Woes of Publication

"I find the concept of statecharts to be quite interesting, but unfortunately only to a small segment of our readership. I find the information presented to be somewhat innovative, but not wholly new. I feel that the use of the digital watch example to be useful, but somewhat simple in light of what our readership would be looking for."

"The basic problem […] is that […] the paper does not make a specific contribution in any area."

"A research contribution must contain 'new, novel, basic results'. A reviewer must certify its 'originality, significance, and accuracy'. It must contain 'all technical information required to convince other researchers in the area that the results are valid, verifiable and reproducible'. I believe that you have not satisfied these requirements."

"I think your contribution is similar to earlier contributions."

"The paper is excellent in technical content; however, it is too long and the topic is good only for a very narrow audience." "I doubt if anyone is going to print something this long."

Regardless, according to a ranking by CiteSeerX in 2015, it stood at 55th position of the most cited articles in computer science. (CiteSeerX — Statistics - Most Cited Articles in Computer Science (psu.edu))

Share with the class: Have you experienced problems in communicating requirements?

3.2 Short Story of QP

While statecharts solve the first problem, that is the need for "a complete and precise specification of system behaviors (model)", we still need to tackle the second problem, that is the need for "a software architecture that supports the execution of the model".

Quantum Platform is one of such tool that works particular well for embedded systems since it's very light-weight and efficient. This is a short story of its evolution.

1. Miro Samek published his article "State-Oriented Programming" in Embedded System Programming in August 2000.

(State-Oriented Programming, 8/2000 (state-machine.com))

2. In 2002 he published the first edition of his book "Practical Statecharts in C/C++". The 2nd edition came out in 2008. (Book: Practical UML Statecharts in C/C++, 2nd Ed. (state-machine.com))
3. He later formed his company Quantum Leaps with a wide customer base.

QP follows the dual-licensing model (GPL and commercial), with GPL exceptions granted to Raspberry Pi/Arduino/mbed-enabled boards. Check its website for latest licensing terms. (Licensing and Pricing (state-machine.com))

Initially QP was called QF (Quantum Framework) and only provides an event processor and event framework. Later he added a real-time kernel called QK and it extension called QXK. The package as a whole is called QP.

QP is a very active project and is constantly updated to fix bugs and add support to new hardware platforms such as the Cortex-M7. It has ports to many RTOSes including uCOS-II and FreeRTOS, if you choose not to use the bundled QK/QXK.

4 STM32CubeIDE

In this part of the course, we will be using the official development tools offered by STMicro. It is called STM32CubeIDE and is based on Eclipse and the GNU/GCC toolchain. Eclipse is a popular alternative to commercial tools like IAR and other free tools such as Visual Studio Code.

STM32CubeIDE is pretty stable and supports multiple platforms (Windows, Mac and Linux). We will be using the Windows version (1.9.0) in this course.

4.1 Installation

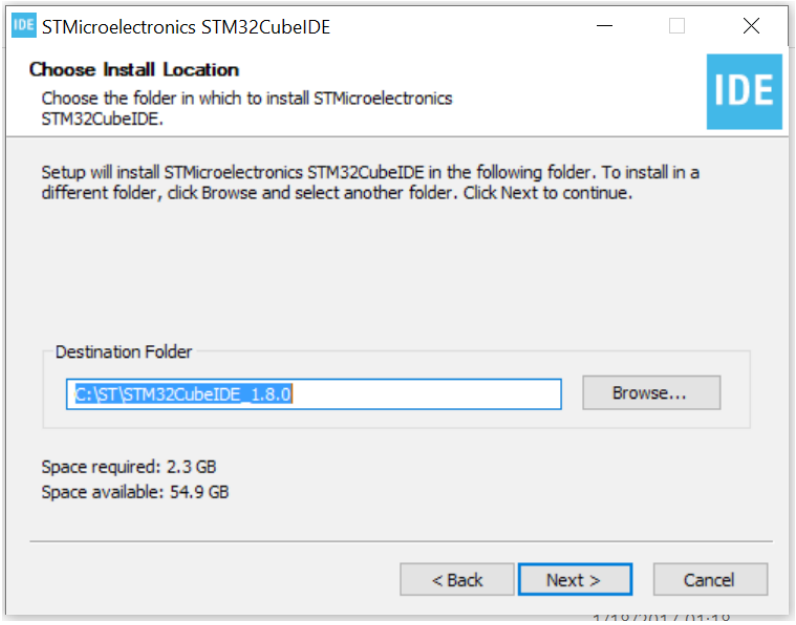
1. Download the installer from STM32 website:

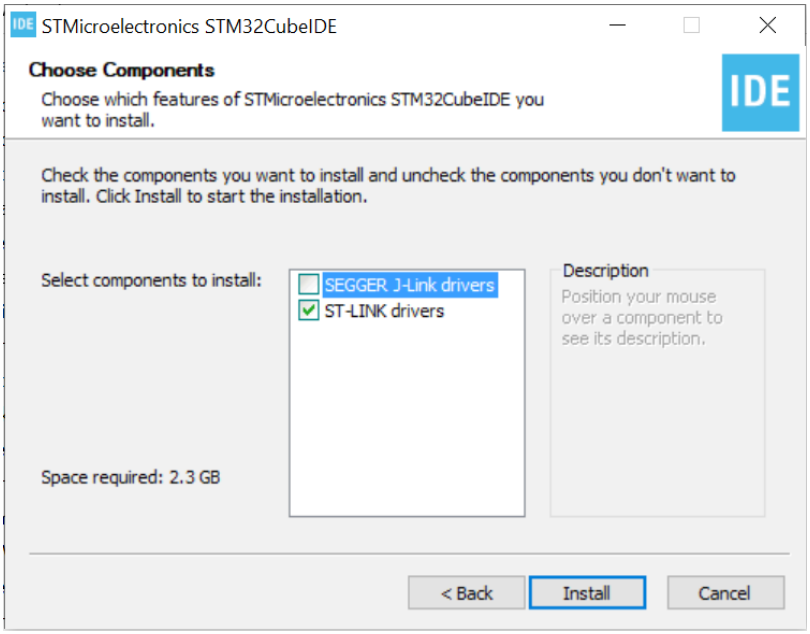
   STM32CubeIDE - Integrated Development Environment for STM32 – STMicroelectronics

   Select version 1.9.0 for Windows.

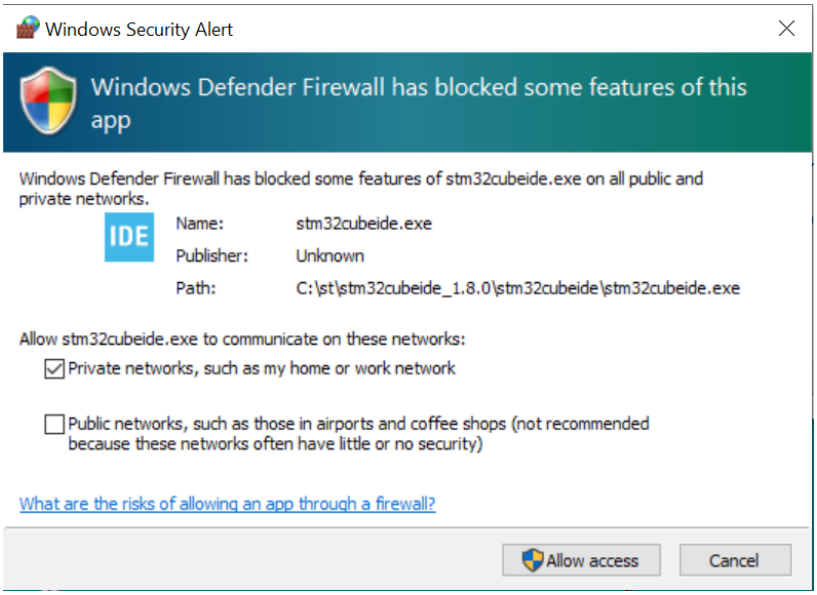
2. Review the license agreement. After accepting it, you will be asked to login to your STM32 account to start the download. Create an account if you have not done so already.
3. Download should automatically start. If not, click the Download button again.
4. Unzip the downloaded file and run the installation *.exe* program. Follow the on-screen instruction to complete the installation.

   It is suggested that you keep the default installation directory (C:\ST\STM32CubeIDE_1.9.0) as we may refer to it later on. Ensure the option for “ST-LINK drivers” is checked.





5. The installed application is named *STM32Cu beIDE 1.9.0*. On first run, your Windows firewall may prompt you to allow network access. The application requires a network port to communicate with the debugger (OpenOCD or ST-LINK GDB Server). The following screenshot shows enabling the option to allow private network access.



4.2 Workspace

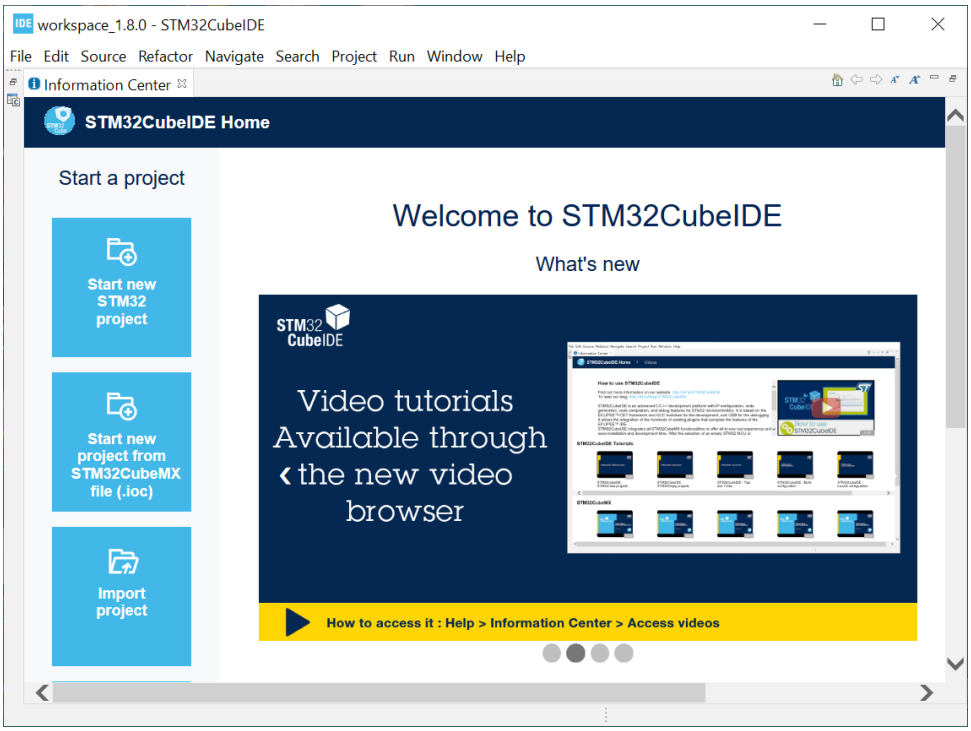
1. In Eclipse, a workspace is your working environment containing one or more projects. This allows you to configure global settings shared among multiple projects. You may also quickly change your environment by switching to a different workspace.

   Every time when Eclipse is launched, you will be asked to select your workspace directory. By default it is set to *C:\Users\<username>\STM32CubeIDE\workspace_1.9.0*. It is recommended that you leave it as default.

   After selecting your workspace directory, click the Launch button to launch the main application window.

2. When STM32CubeIDE is started for the first time, the main window shows the *Information Center*. It allows you to quickly create a new project or import an example project based on your hardware platform.

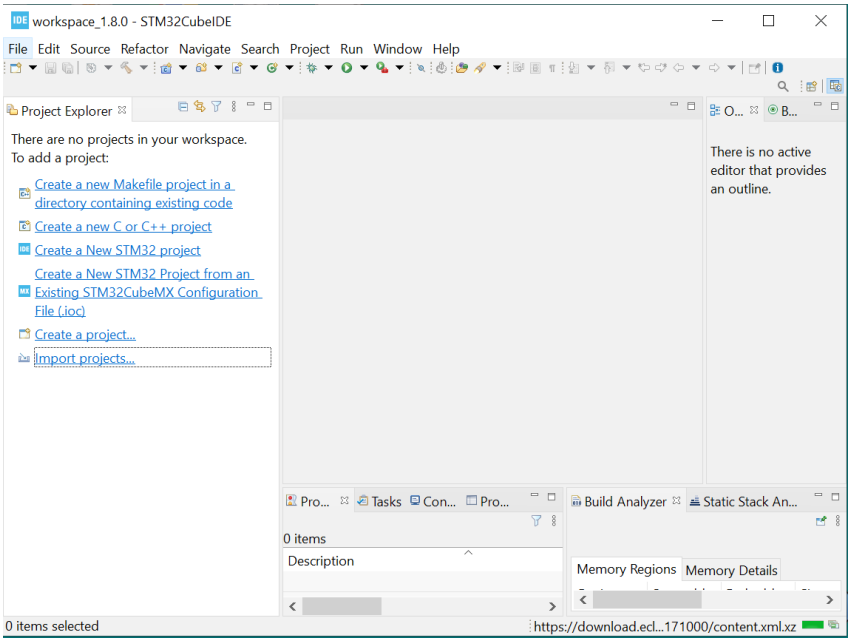
   In this course, you will start with a common baseline project which is already setup. Go ahead to close the *Information Center*.



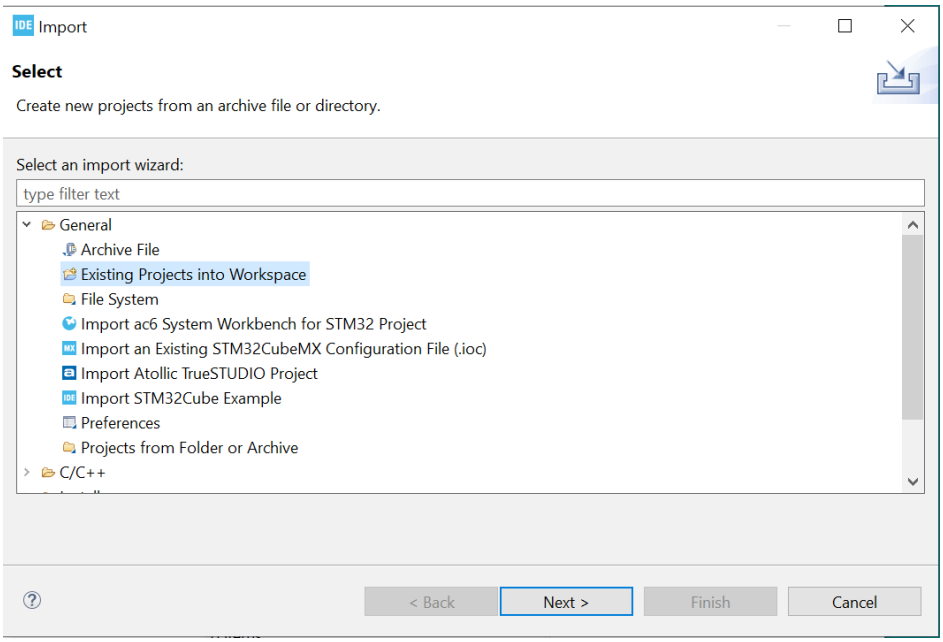
4.3 Import Project

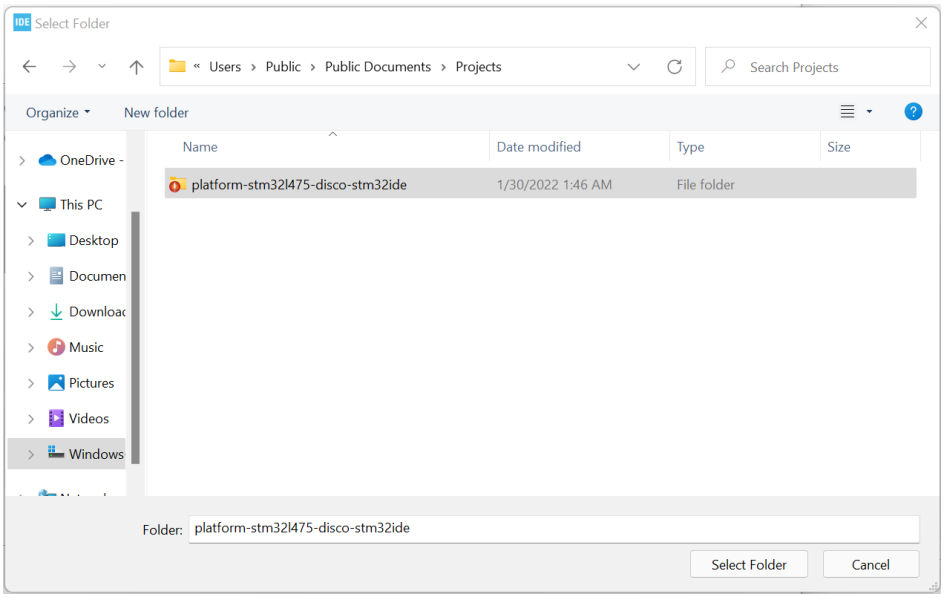
1. Download the baseline project from the course site. Decompress it to a local folder which should by default be named as *platform-stm32l475-disco-stm32ide*.

2. Open *Project Explorer* by selecting from the top menu *Window* > *Show View* > *Project Explorer*. Click *Import projects…*

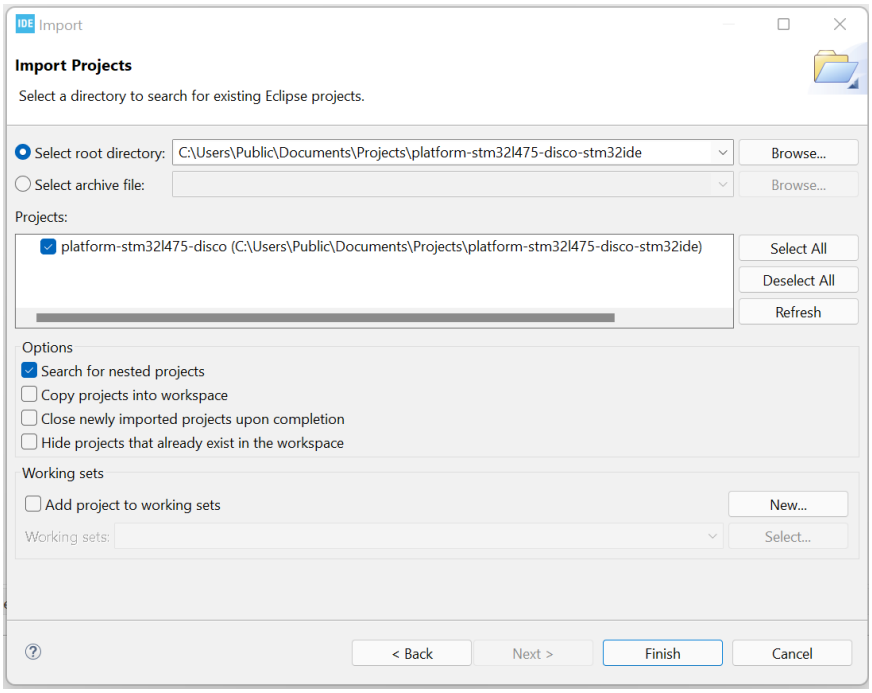


3. Select *General* > *Existing Projects into Workspace* and click the *Next* button. Navigate to and select the baseline project folder (by default named as *platform-stm32l475-disco-stm32ide*). Click the *Select Folder* button.

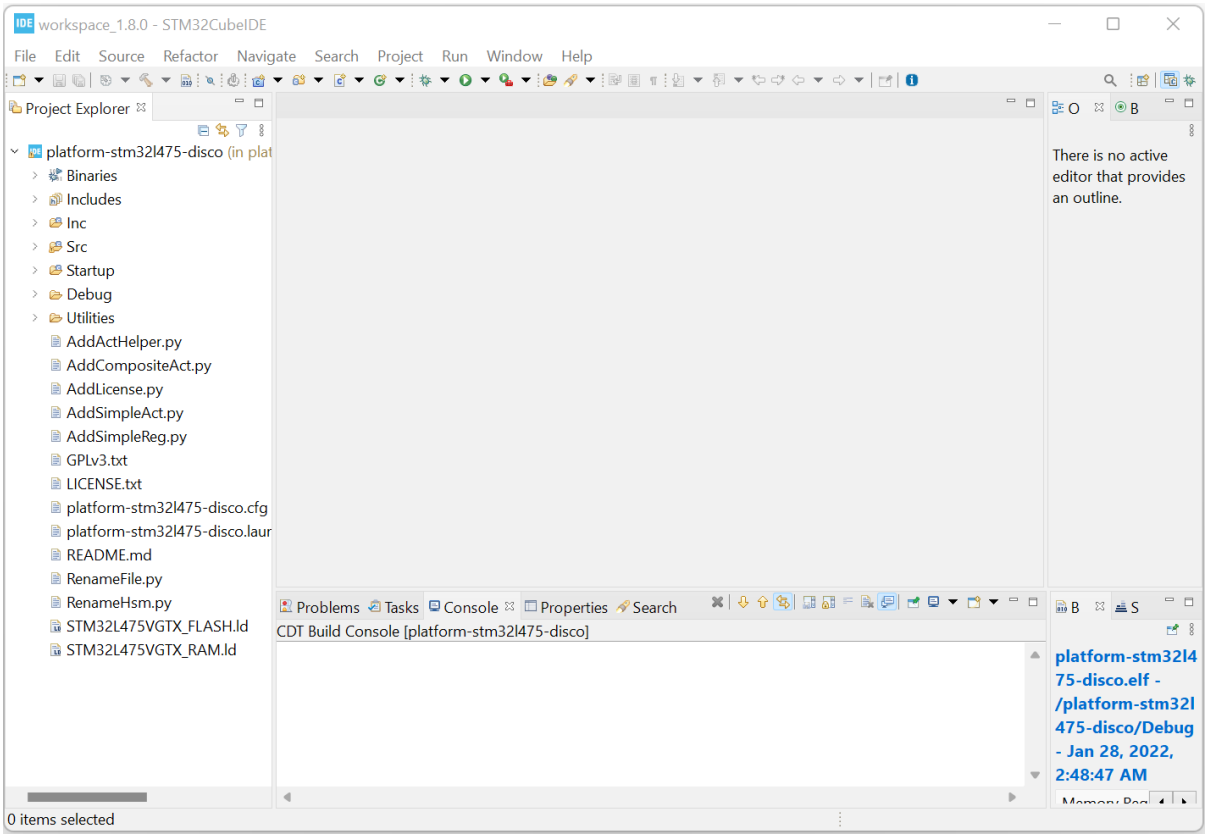




4. Next, you will be presented the *Import Projects* window. You should see the project named *platform-stm32l475-disco* checked in the *Projects* box. Leave all options as default. In particular, DO NOT check the "*Copy projects into workspace*" option to keep the source in the original folder.



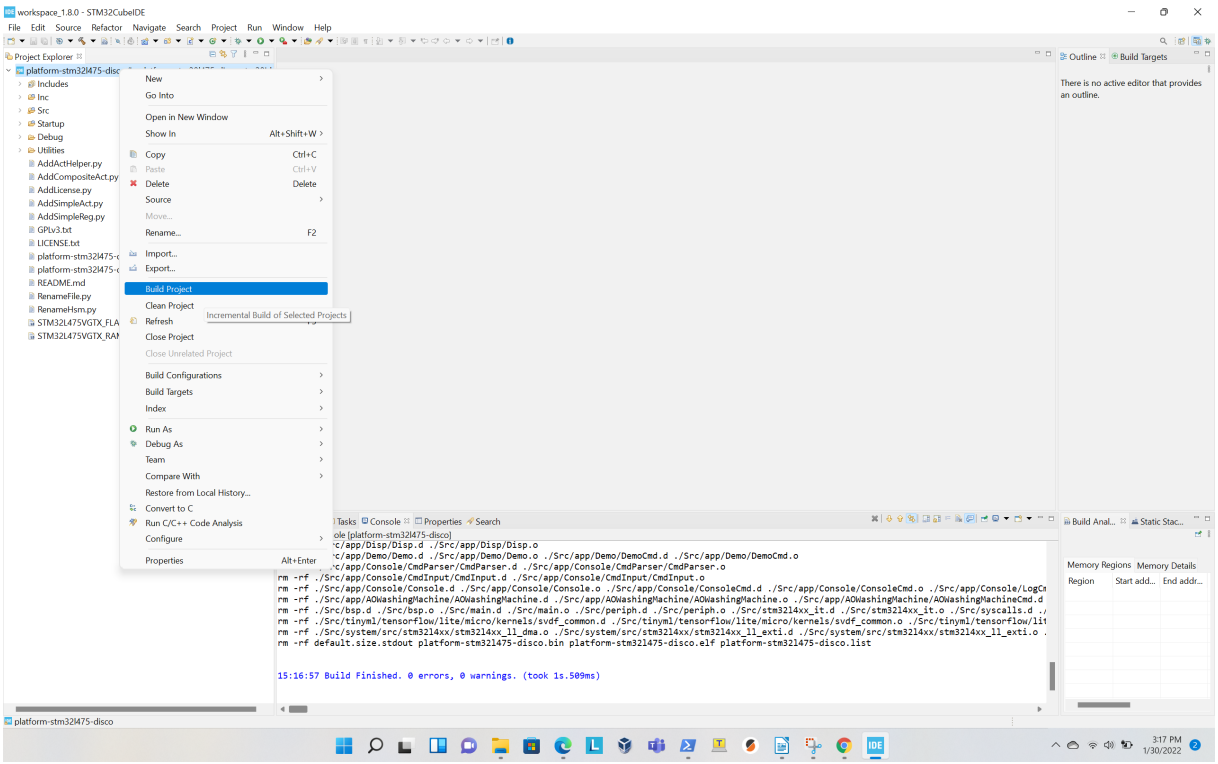
5. Finally, the imported project *platform-stm32l475-disco* appears in the *Project Explorer* panel. Click the > icon next to the project name to expand the project folders.



4.4 Build Project

1. Right-click on the project name in the Project Explorer to bring up the context menu. There are a few important items here:

   - *Build Project* – To compile and link the source code to generate executable *elf* and *bin* files.
   - *Clean Project* – To clean the project. You may run this before archiving your source folder, or to resolve any dependency issues.
   - *Properties* – To configure project settings such as compiler and linker flags, environment variables, etc.
   - *Delete* – To delete the project from the Project Explorer. Note: Make sure the option "*Delete project contents on disk*" is UNCHECKED if you want to keep the source folder on disk so that you can re-import the project later. This is useful when you need to switch between different source folders with the same project names.



2. Click *Build Project* and you should see a message similar to the following when it is built successfully:

```
arm-none-eabi-objcopy  -O binary  platform-stm32l475-disco.elf  "platform-stm32l475-disco.bin"
   text    data     bss     dec     hex filename
 493916     648   81192  575756   8c90c platform-stm32l475-disco.elf
Finished building: default.size.stdout

Finished building: platform-stm32l475-disco.bin
```

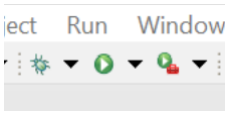
```
Finished building: platform-stm32l475-disco.list

15:25:20 Build Finished. 0 errors, 41 warnings. (took 21s.476ms)
```

It is OK to have warnings but there should be no errors. Most of the warnings are caused by 3rd party libraries.

4.5 Debugger Setup

Under the top menu bar, there are three small icons which are used to start the debugger.



The leftmost icon is to start or configure the *Debugger*. The rightmost icon is to start or configure *External Tools*. There are a couple ways to launch the debugger and they are described in the following sections.

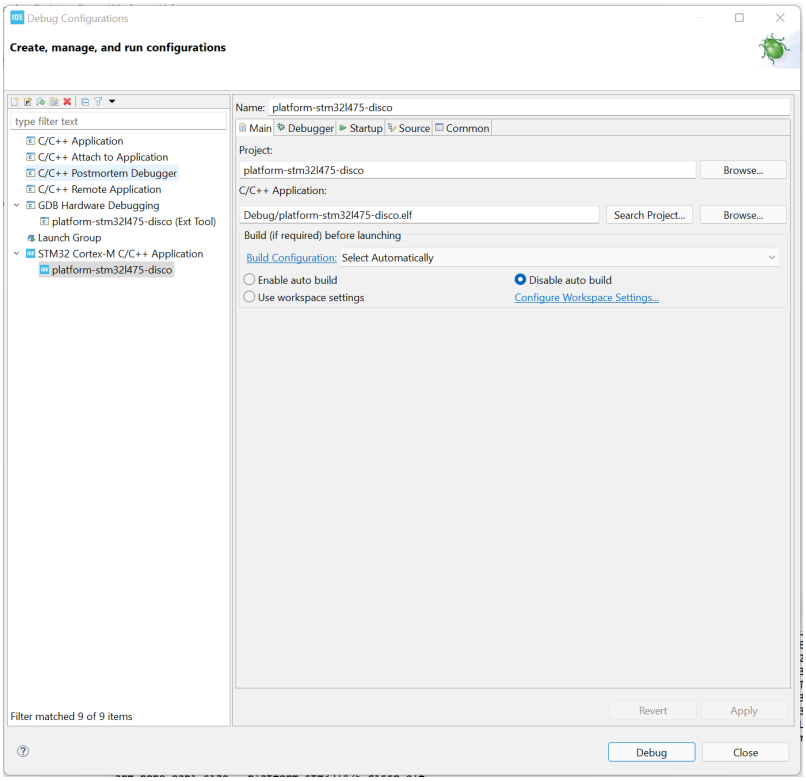
4.5.1 Built-in OpenOCD

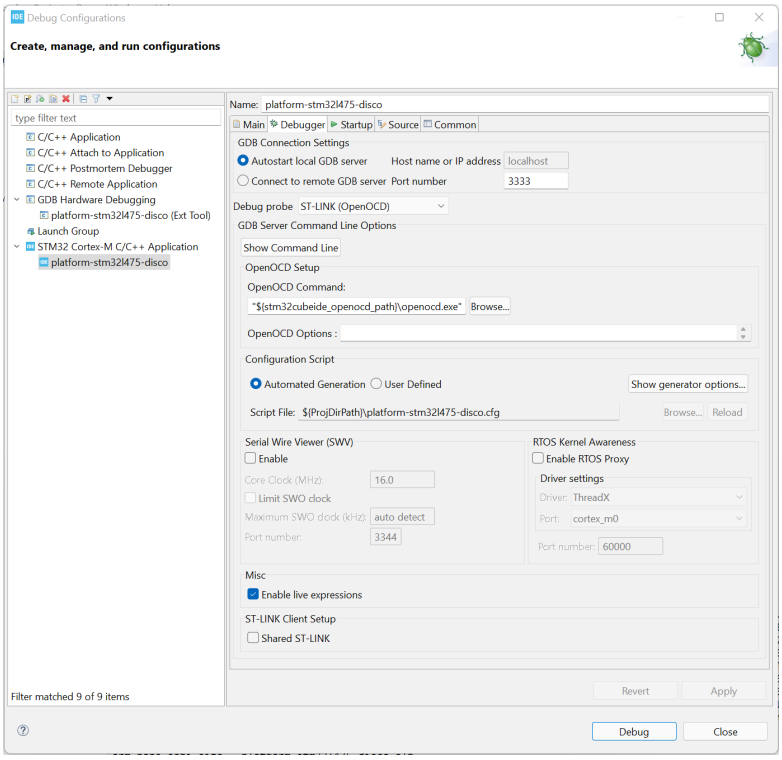
1. Click the *down arrow* next to the *Debugger* icon to pull down a menu. Click the "*Debug Configurations...*" menu item.
2. Click *STM32 Cortex-M C/C++ Application >platform-stm32l475-disco* on the left side panel.
3. The imported project should come with proper default settings. Review those settings with the following screenshots (one for each tab).
4. Click the *Debug* button at the bottom-right corner to start debugging. Alternatively you may launch it via the *favorite* link named *platform-stm32l475-disco* at the top of the pull-down menu.

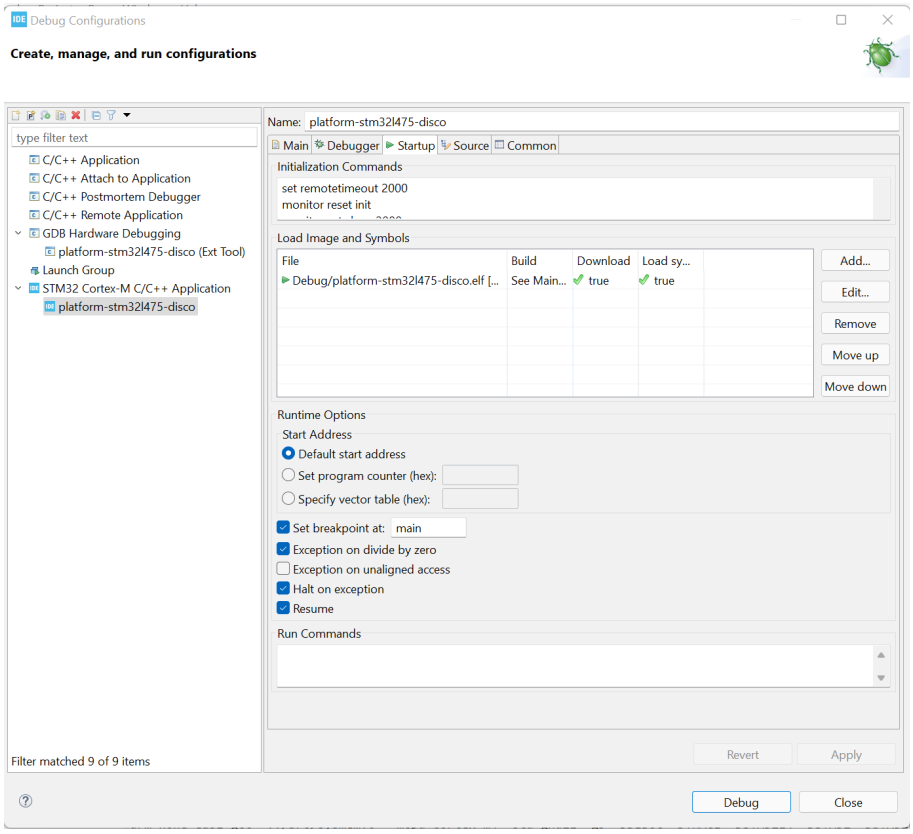
   It will take some time (<30s) to download and flash the program to the target. When it is completed successfully, you should see logs ending like these:

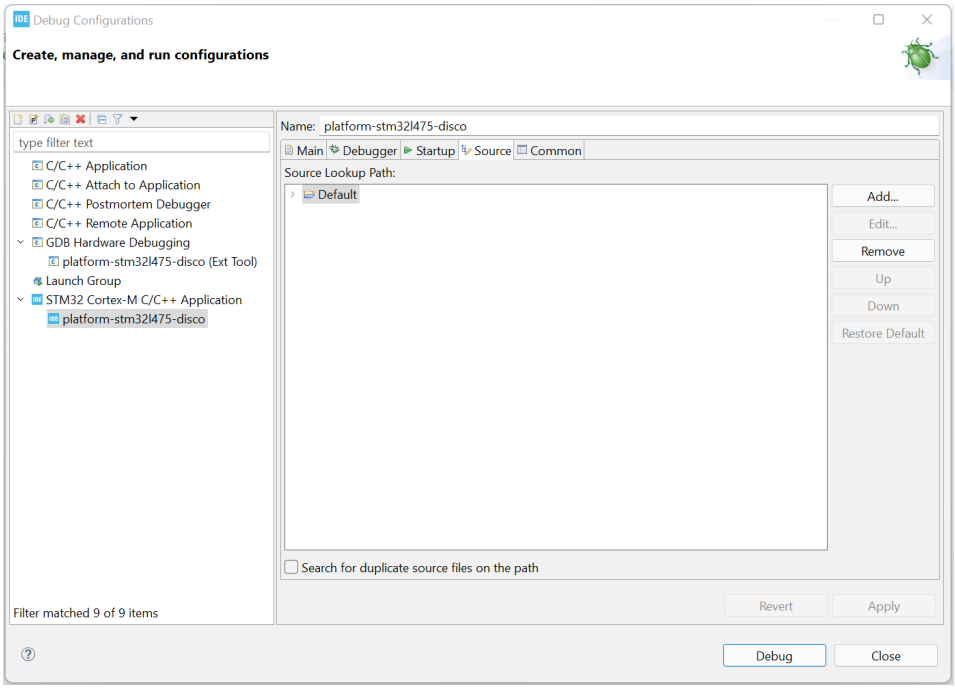
```
Info : accepting 'gdb' connection on tcp/3333
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08066e28 msp: 0x20018000
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08066e28 msp: 0x20018000
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
2000
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08066e28 msp: 0x20018000
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Padding image section 0 at 0x08000194 with 12 bytes
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08066e28 msp: 0x20018000
```

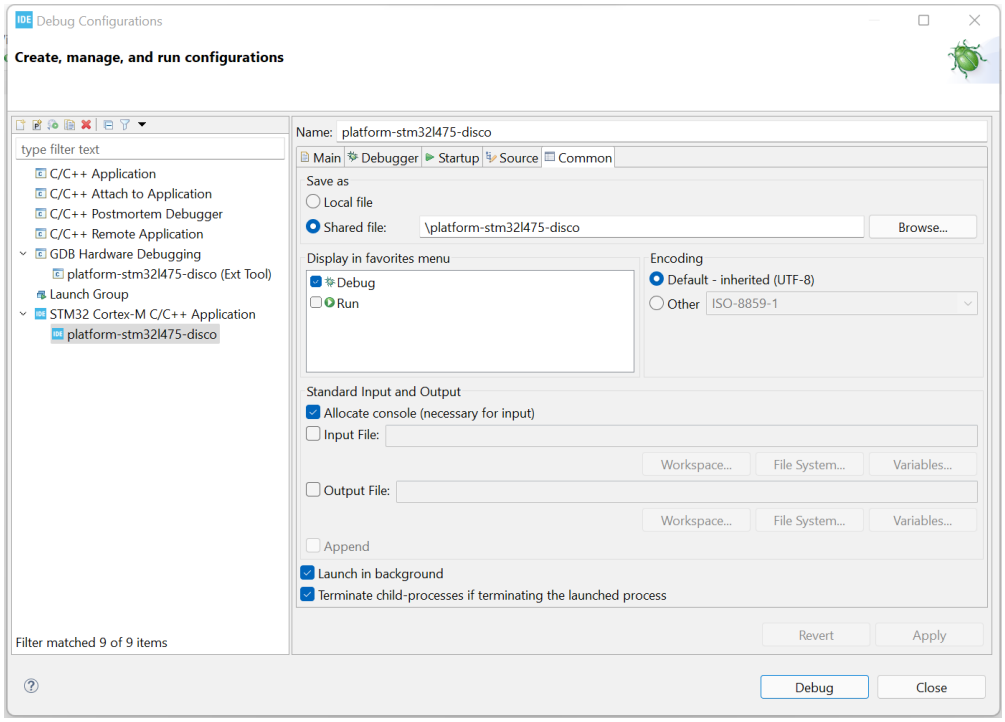
5. This method is convenient since it launches OpenOCD automatically, but it may be slower since it needs to relaunch OpenOCD for each debug session.











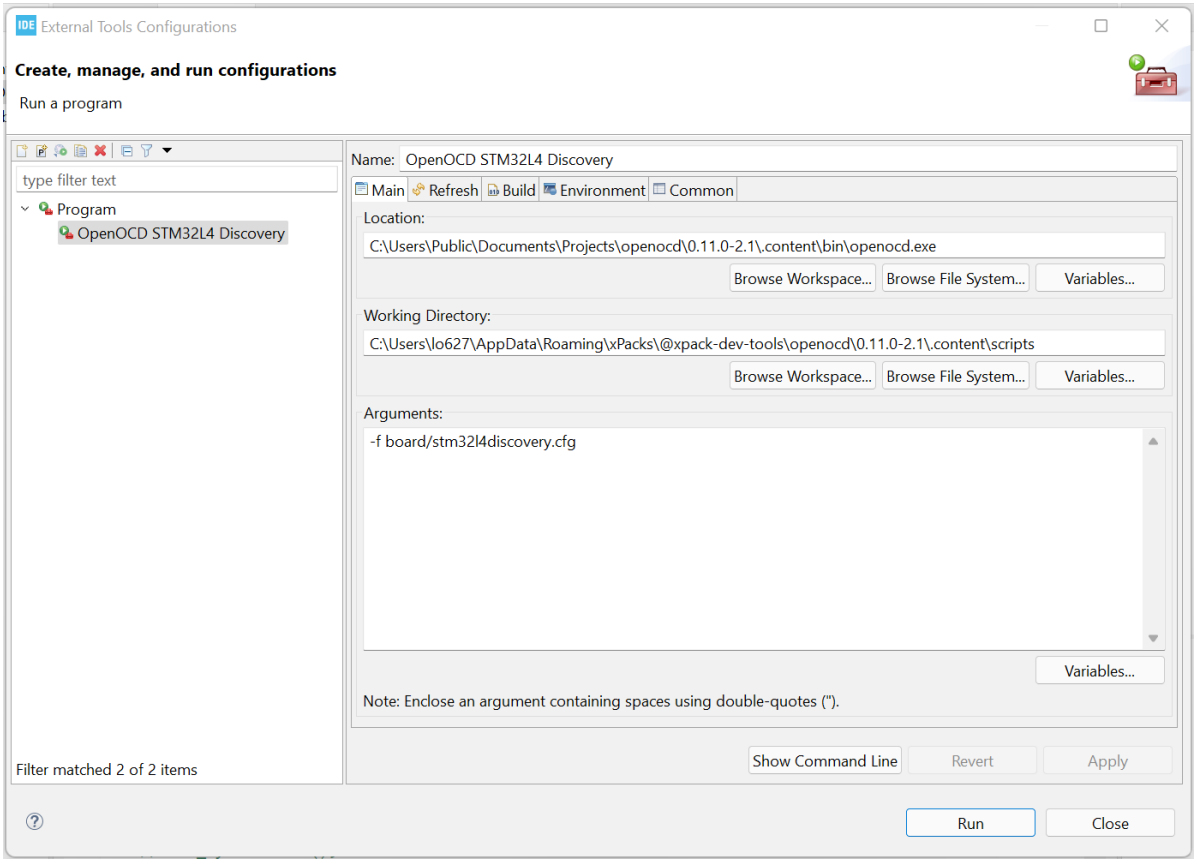
4.5.2 External OpenOCD (Optional)

1. Another way to launch a debug session is to start OpenOCD as an external tool.

   In order to do that, we need to first setup OpenOCD by clicking the down arrow next to the *External Tools* icon to pull down a menu. Click the "*External Tools Configurations…*" menu item.

2. Create a new tool configuration under *Program* on the left side panel. Name this configuration *OpenOCD STM32L4 Discovery.* The key configurations are the path to the *openocd.exe*, path to the *scripts folder* and the specific board *cfg file* for our platform.

   It is assumed that you have OpenOCD installed previously. In the example below, OpenOCD has been installed under *C:\Users\Public\Documents\Projects\openocd*.



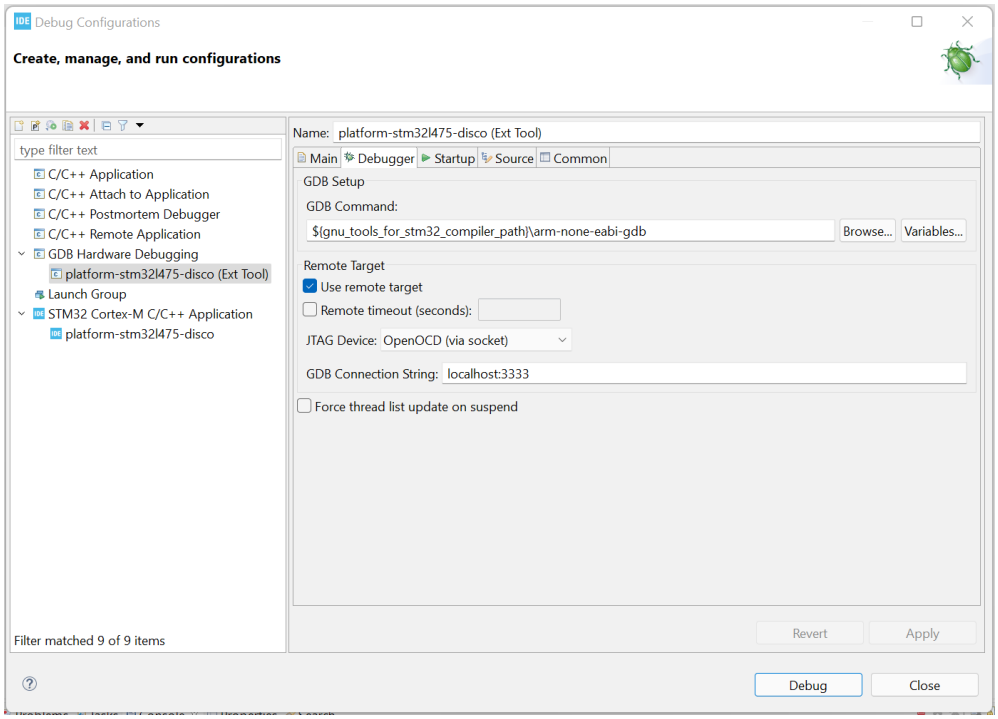
3. Click the *Run* button at the bottom-right corner to start OpenOCD. Alternatively you may run it via the *favorite* link named *OpenOCD STM32L4 Discovery* at the top of the pull-down menu.

   If it is started successfully, the console shows these log messages:

```
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 500 kHz
Info : STLINK V2J39M27 (API v2) VID:PID 0483:374B
Info : Target voltage: 3.246592
Info : stm32l4x.cpu: Cortex-M4 r0p1 processor detected
Info : stm32l4x.cpu: target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for stm32l4x.cpu on 3333
Info : Listening on port 3333 for gdb connections
```

4. Now we go back to set up our debug configuration. Click the *down arrow* next to the *Debugger* icon to pull down a menu. Click the "*Debug Configurations...*" menu item.

5. Click *GDB Hardware Debugging >platform-stm32l475-disco (Ext Tool)* on the left side panel. Review the default settings which are similar to those in the *Built-in OpenOCD* section above, except for the *Debugger tab* shown below.



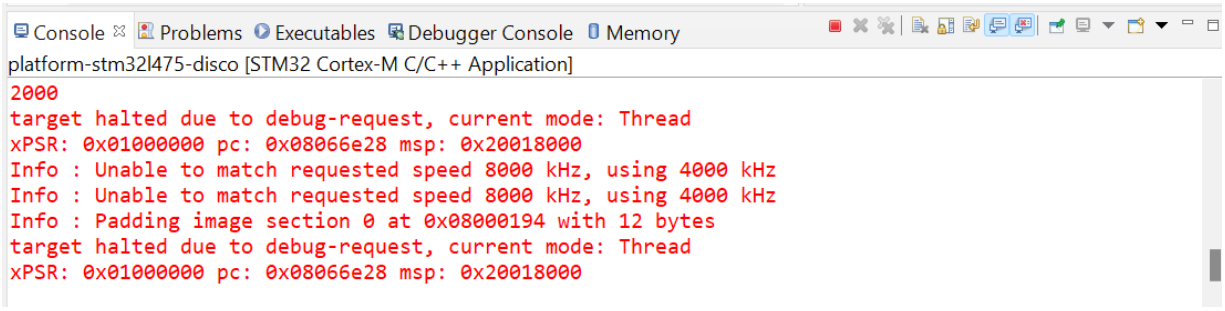
6. Click the *Debug* button at the bottom-right corner to start debugging. Alternatively you may launch it via the *favorite* link named *platform-stm32l475-disco (Ext Tool)* at the top of the pull-down menu. When the program is downloaded and flashed successfully, you should see logs ending like these:

```
xPSR: 0x41000000 pc: 0x08015a9c msp: 0x20017f78
Info : Unable to match requested speed 500 kHz, using 480 kHz
Info : Unable to match requested speed 500 kHz, using 480 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08066e28 msp: 0x20018000
2000
Info : Unable to match requested speed 500 kHz, using 480 kHz
Info : Unable to match requested speed 500 kHz, using 480 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08066e28 msp: 0x20018000
Info : Padding image section 0 at 0x08000194 with 12 bytes
Info : Unable to match requested speed 500 kHz, using 480 kHz
Info : Unable to match requested speed 500 kHz, using 480 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08066e28 msp: 0x20018000
```

7. This method is less convenient as you would need to start OpenOCD manually prior to launching a debug session. However it may be faster since OpenOCD remains running in the background across multiple debug sessions.

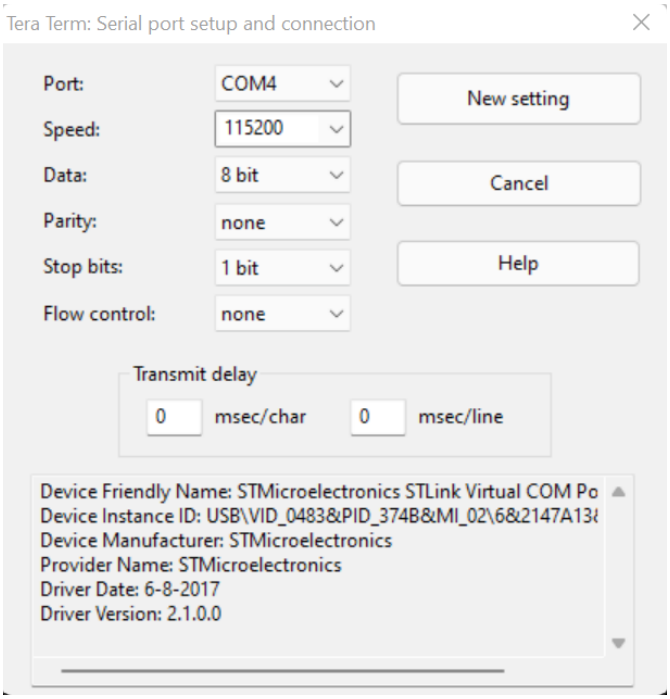
8. In some cases you may need to stop OpenOCD manually. For example, when you unplug the target board while Eclipse is still running, OpenOCD will lose its connection to the target and show an error message repeatedly.

To stop OpenOCD, click the *Red Stop* button and the *Double Cross* button at the top-right corner of the console window.

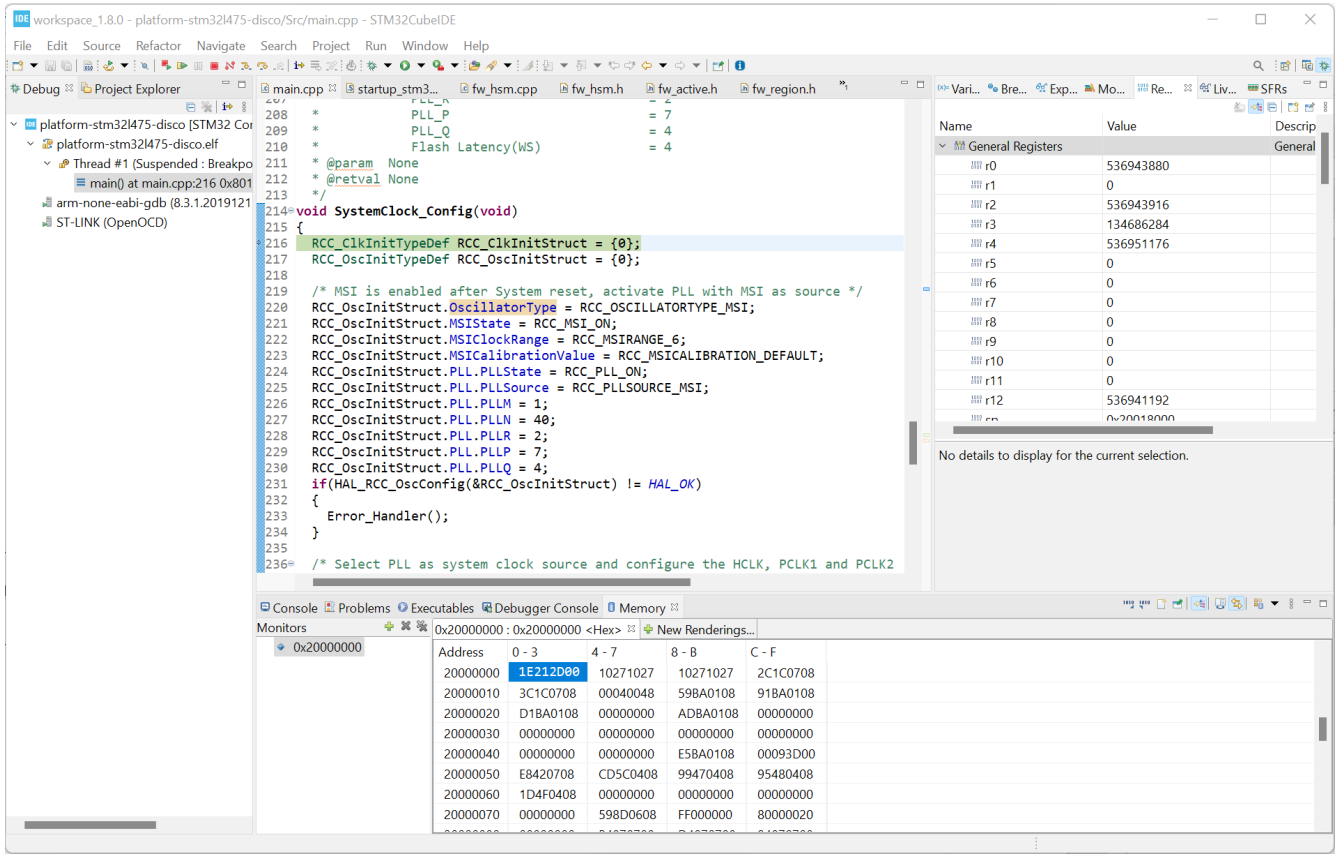


4.6 Run and Debug

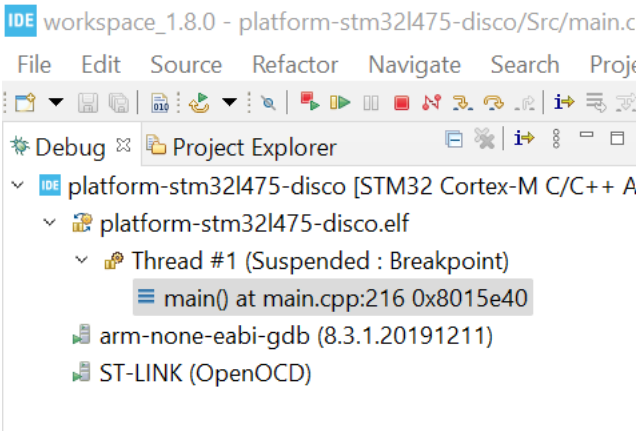
1. Before running our program, you should first start a *terminal* program on Windows, such as *teratherm*. Select the COM port with the name "*STMicroelectronics STLink Virtual COM Port*". Click *Setup* > *Serial port…* and configure the serial port to use 115200bps N81.



2. The Debug window should show the program stopped at the beginning of main(). In the code window, it shows SystemClock_Config() since main() calls this function immediately.:



3. The program control toolbar is at the top left corner. Run the program by clicking the green Play/Resume button. You may suspend/break the program at any time and step into/over/out of a function, etc. Like other debugger IDE, you can set up breakpoints, inspect registers, variables and memory.



4. As the program is running on the target, you should see its debug log in the terminal program, which looks like this:

```
...
*****************************
*   Console STM32L475 Disco  *
*****************************
7 CONSOLE_UART1> 205 SYSTEM(1): Prestarting IDLE_CNT_TIMER
205 SYSTEM(1): maxIdleCnt = 7494000
205 SYSTEM(1): Prestarting EXIT
205 SYSTEM(1): Starting1 ENTRY
205 NODE(15): Stopped NODE_START_REQ from SYSTEM(1) seq=6
205 NODE(15): Stopped EXIT
205 NODE(15): Starting ENTRY
205 WIFI(14): Stopped WIFI_START_REQ from NODE(15) seq=0
206 WIFI(14): Stopped EXIT
206 WIFI(14): Starting ENTRY
206 WIFI(14): Starting DONE from WIFI(14) seq=0
206 WIFI(14): Starting EXIT
...
```

5. You should be able to interact with the console. Try to enter the command “fib” and see what happens. Just hit *enter* again to stop the command.

6. If you want to restart the program from the beginning (without downloading again), you may click the *Restart* button and select “*Reset init*” from the pull-down menu.

7. When you are done with a debug session, click the red Stop/Terminate button to terminate the session. You can switch back to the *C/C++ perspective* by clicking its button at the top-right corner. You can then start a new *code* > *build* > *debug* cycle.

5 Reference

1. Statecharts in the Making: A Personal Account. David Harel. ACM Communications. March 2009. (http://www.wisdom.weizmann.ac.il/~harel/papers/Statecharts.History.pdf)

2. Statecharts: A Visual Formalism For Complex Systems. David Harel. Science of Computing Programming 8. 1987. (http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf)

3. Modeling Rover Communication Using Hierarchical State Machines with Scala. Klaus Havelund. September 2017. (http://rjoshi.org/bio/papers/tips-2017.pdf)

4. Auto-coding UML statecharts for flight software. Ken Clark , Garth Watney. Space Mission Challenges for Information Technology. 2006.

   (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.275&rep=rep1&type=pdf)

5. Automatic code generation for instrument flight software. Kiri L. Wagstaff, Edward Benowitz, Dj Byrne, Ken Peters, Garth Watney. Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2007.

   (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.126.548&rep=rep1&type=pdf)