AN5394

Application note

Getting started with projects based on the STM32L5 Series in STM32CubeIDE

Introduction

This application note describes how to get started with projects based on STM32L5 Series microcontrollers in STMicroelectronics STM32CubeIDE integrated development environment.

1 General information

STM32CubeIDE supports STM32 32-bit products based on the Arm® Cortex® processor.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

1.1 Prerequisites

The following tools are prerequisites for understanding the tutorial in this document and developing an application based on the STM32L5 Series with Arm® TrustZone® enabled:

- STM32CubeIDE 1.2.0
- STM32CubeProgrammer (STM32CubeProg) 2.3.0: configuration of Option Bytes
- STM32Cube_FW_L5_V1.0.0: STM32CubeL5 firmware with example project, and HAL and CMSIS drivers

Users are advised to keep updated with the documentation evolution of the STM32L5 Series at www.st.com/en/microcontrollers-microprocessors/stm32l5-series.

1.2 The use cases in this document

In the STM32CubeIDE context, users have a number of different ways to explore and get started with the development of projects based on the STM32L5 Series:

- Import an STM32CubeIDE project from the STM32CubeL5 MCU Package to learn by using a working example
- Create an STM32CubeMX project using the STM32CubeMX tool integrated inside STM32CubeIDE, or the stand-alone STM32CubeMX tool
- Create an empty project in STM32CubeIDE and write their own code
- Create an empty project in STM32CubeIDE and copy resources from the example project template available in the STM32CubeL5 MCU Package

The following approach is recommended to get familiar and successfully started with a project development based on the STM32L5 Series:

1. Import a TrustZone® example project, which is part of the STM32CubeL5 MCU Package. This is the quickest way to understand the CMSIS and HAL drivers provided for bootstrapping the STM32L5 device.
2. Create an empty project as the production project and copy the code from the STM32CubeL5 MCU Package. In empty projects, users are in full control of the source code and configuration files, which are not touched by STM32CubeMX. This gives users higher flexibility, but require a slightly steeper learning curve.
3. Create an STM32CubeMX project to use the graphical interface to configure the hardware and generate the corresponding HAL drivers. This can be used as the production project or playground to explore and learn more.

Some template projects are supplied in STM32CubeIDE project format; these are template projects with and without TrustZone® enabled. For example:

- Using `TZEN = 1`:
  `STM32Cube_FW_L5_V1.0.0\STM32Cube_FW_L5_V1.0.0\Projects\STM32L552E-EV\Templates\TrustZoneEnabled\`
- Using `TZEN = 0`:
  `STM32Cube_FW_L5_V1.0.0\STM32Cube_FW_L5_V1.0.0\Projects\STM32L552E-EV\Templates\TrustZoneDisabled\`

This application note refers to the `TrustZoneEnabled` project template mentioned above, where TrustZone® is enabled by Option Byte `TZEN` configured to 1.

The `readme` file for this project template describes how to configure the Option Bytes to match the code; It provides a good template to learn some important configuration use cases.

After their first learning experience, users can choose between creating an empty project, start with an STM32CubeMX-managed project for their own application development, or try both.

Firmware STM32Cube_FW_L5 contains many other example projects for different peripherals with STM32CubeIDE project files. These can be imported into STM32CubeIDE and studied for learning how to use STM32L5 peripherals.

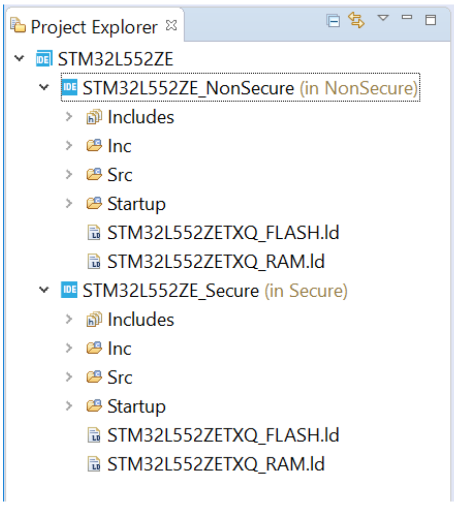
1.3 Option Bytes

To learn more about the Option Bytes, refer to the the reference manual for microcontrollers in the STM32L5 Series (RM0438). For the specific example project template that are the basis of this application note, the correct Option Bytes values are listed in the `readme.txt` file in the example project. STM32CubeProgrammer (STM32CubeProg) must be used to program the Option Bytes.

1.4 Specific hierarchical project structure for secure and multi-core MCUs

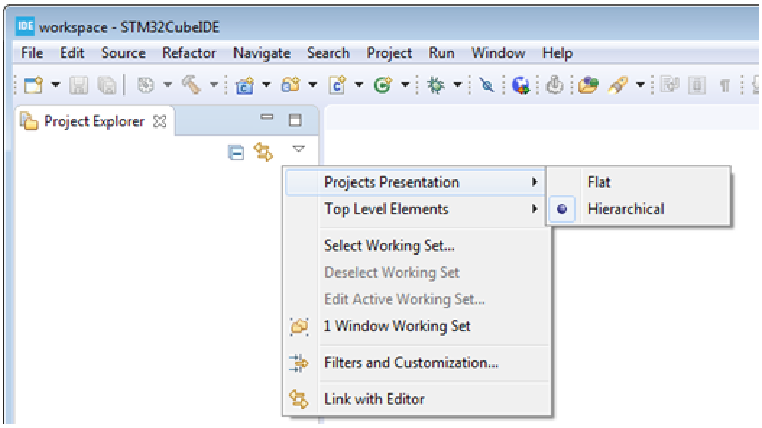
Before importing or creating projects, it is important to consider some project structural concepts. After creating an STM32L5 project, the project structure is automatically made hierarchical. The project structure for single-core projects is flat. In a multi-core project, or a project with a TrustZone®-enabled MCU like in the STM32L5 Series, the hierarchical project structure is used. When the user creates or imports a project, it consists of one root project together with sub-projects referred to as MCU projects. The MCU projects are real CDT™ projects; They can contain build and debug configurations while the root project cannot. The root project is a simple container that allows sharing common code between the secure and non-secure MCU projects (in the case of the STM32L5 Series), as illustrated in Figure 1.

Figure 1. STM32L5 project with hierarchical project structure



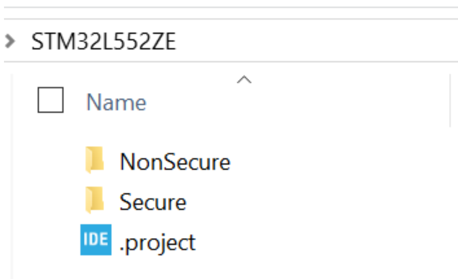
If the setting has been changed or the project is otherwise not in a hierarchical structure, it can be changed as shown in Figure 2.

Figure 2. Changing the visual representation between flat and hierarchical project structure



In the file system, the two MCU projects are located inside the root project, which only contains one `.project` file.

Figure 3. Root project with `.project` file



2 Creating and importing projects

This chapter describes how to import or create projects based on the STM32L5 Series. It starts by explaining how to import the example project template available in the STM32CubeL5 MCU Package. After importing, building, debugging and adding some function calls to non-secure callables, it shows how to create an own empty project, copying the very same resources from the example project as a base template for the continued application development.

Note: It is not recommended to continue the application development in the example project itself mainly because all resources in this project are linked into the project:

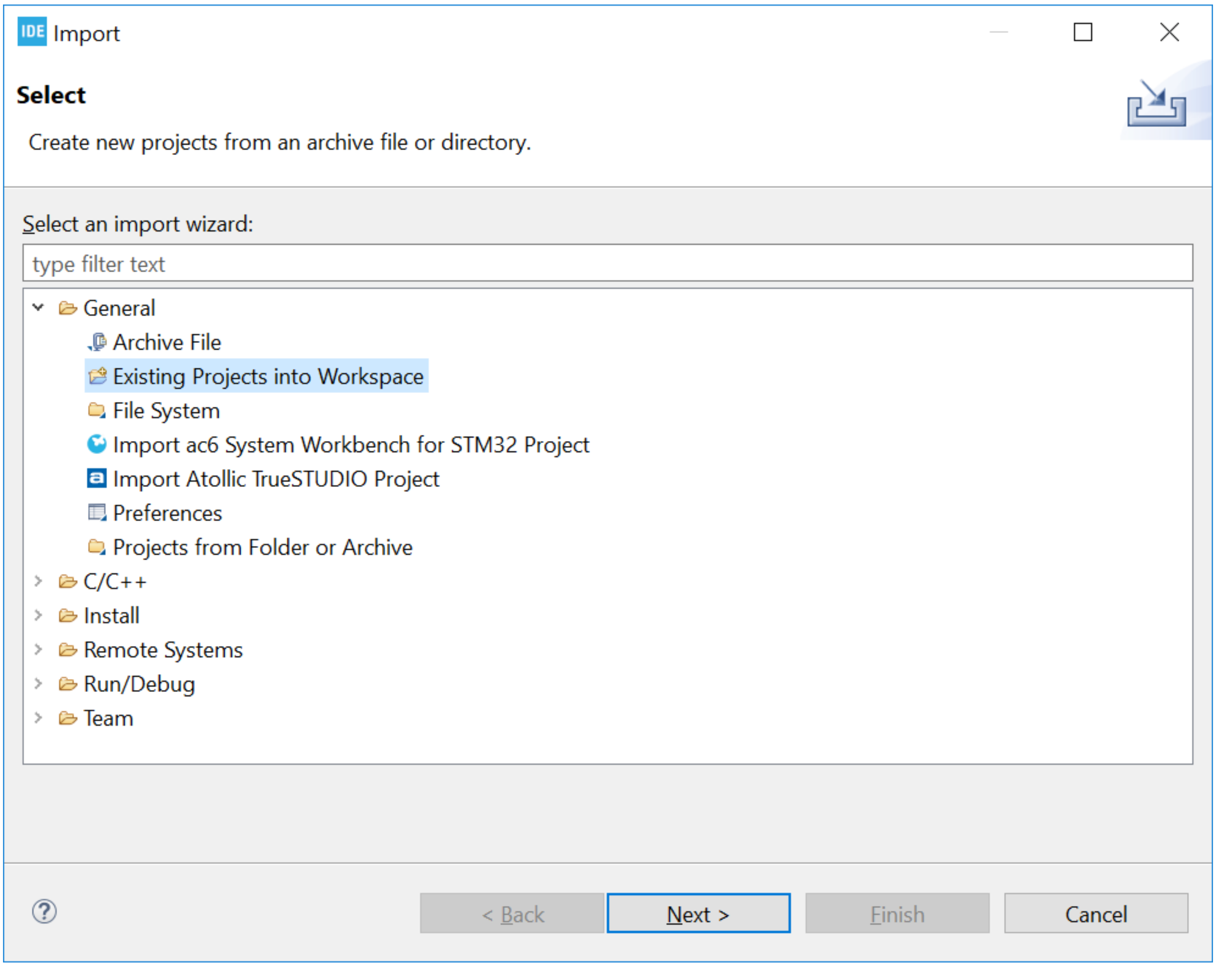
- *This means that the project is not self-contained, making version control more difficult*
- *Driver resources are shared with all other projects*
- *The Eclipse CDT™ indexer cannot always resolve linked resources and properly enable all code navigation and visualization features*

Creating a new empty project using the example project as a template is a better approach for continued application development.

2.1 Importing the TrustZone® project template for STM32CubeIDE

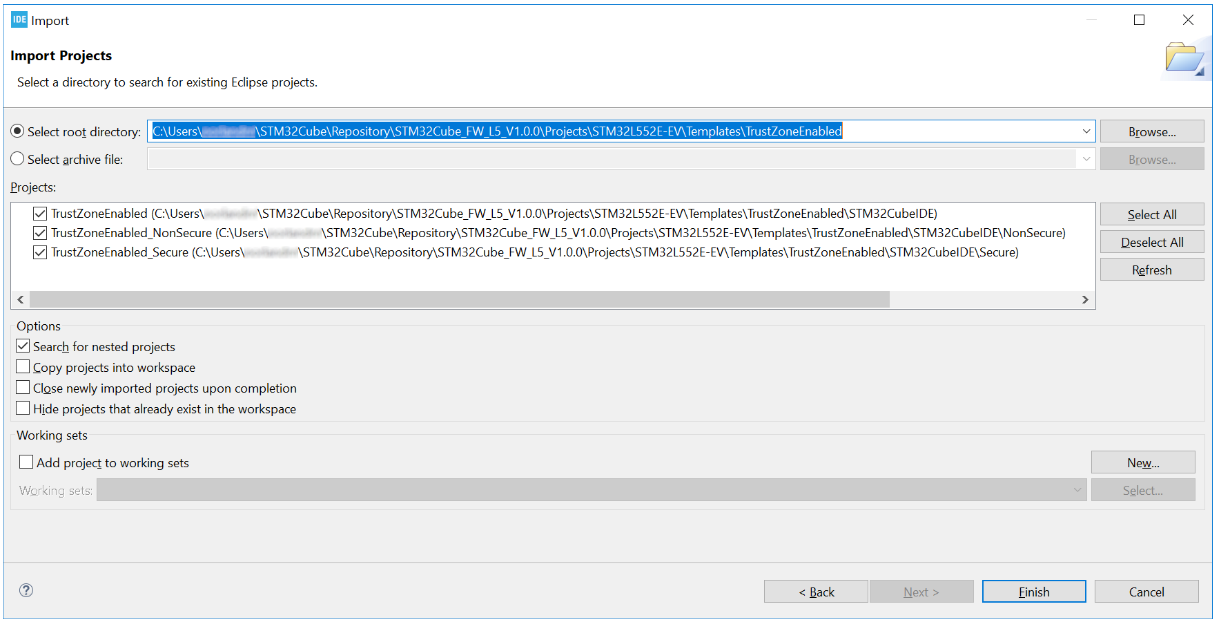
To import the STM32CubeL5 template project into STM32CubeIDE, first go to [**File**]>[**Import**] and select *Existing Projects into Workspace* as shown in Figure 4.

Figure 4. **Import project template**



Then select the appropriate project, which in Windows® by default is located in the `User` folder, such as `$HOME\STM32Cube\Repository\STM32Cube_FW_L5_VX.X.X\Projects\STM32L552E-EV\Templates\TrustZoneEnabled\` (refer to Figure 5).

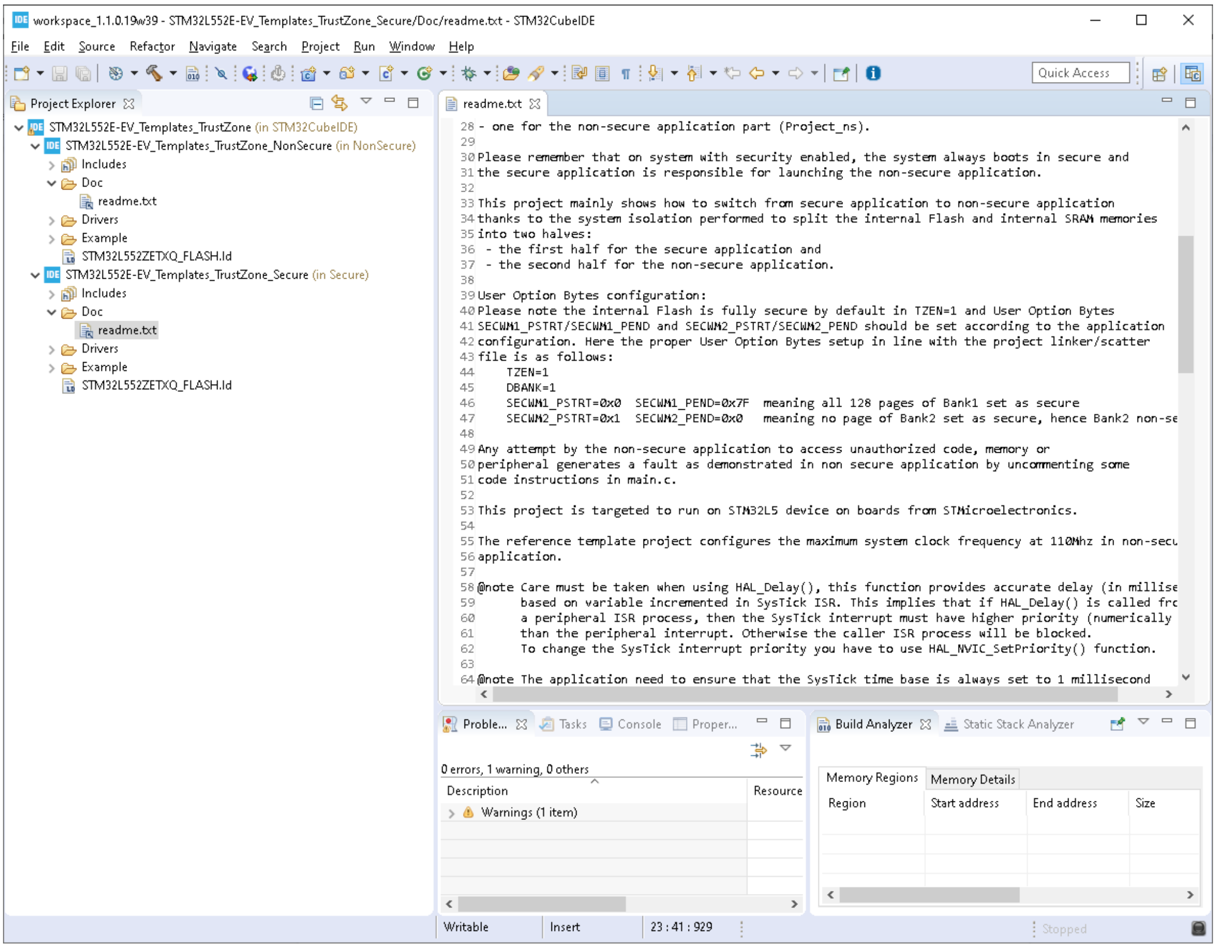
Figure 5. Selection of the STM32L552E-EV template project



Attention: *When importing projects from STM32Cube MCU Packages, do not use the "Copy projects into workspace" setting since it breaks the links to shared code such as HAL and CMSIS drivers in MCU Packages.*

After selecting all three projects, click on [**Finish**] to import the template project.

Figure 6. Building the imported template project



2.2 Exploring the example project

To get familiar with the example project, start by reading the project `readme.txt` file, which is linked to the `Doc` folder.

2.2.1 Option Bytes

Set the Option Bytes according to the `readme.txt` file using STM32CubeProgrammer. For TrustZone®-enabled projects, these settings are typically:

- `TZEN = 1`
- `DBANK = 1`
- `SECWM1_PSTRT = 0x0` and `SECWM1_PEND = 0x7F`, meaning that all 128 pages of Bank1 are set as secure
- `SECWM2_PSTRT = 0x1` and `SECWM2_PEND = 0x0`, meaning that no page of Bank2 is set as secure, hence making Bank2 non-secure

Important remarks:

- The `SECWMx` Option Bytes are not visible until the `TZEN` is enabled and applied to target
- Always double-check the `readme.txt` file for correct Option Byte values

For more information, refer to the STM32CubeProgrammer user manual (UM2237).

2.2.2 Explore the linker script, memory partitioning, and SAU initialization

Each secure and non-secure project is built with its own linker script. This section presents some of the differences between secure and non-secure linker scripts in the case of an STM32L552ZE microcontroller.

The Flash memory size of the STM32L552ZE is 512 Kbytes. With the Option Byte `DBANK = 1`, the Flash memory is split up into two banks of 256 Kbytes each: one secure and one non-secure bank. Both banks are then further split into regions for different types of toolchain outputs.

Secure Flash linker script:

```
MEMORY
{
  RAM      (xrw)   : ORIGIN = 0x30000000,  LENGTH = 96K
  ROM      (rx)    : ORIGIN = 0x0C000000,  LENGTH = 248K
  ROM_NSC  (rx)    : ORIGIN = 0x0C03E000,  LENGTH = 8K   /* non-secure callable region */
}
```

Non-secure Flash linker script:

```
MEMORY
{
  RAM      (xrw)   : ORIGIN = 0x20018000,  LENGTH = 96K
  ROM      (rx)    : ORIGIN = 0x8040000,   LENGTH = 256K
}
```

In this example, the secure application starts at `0x0C00 0000` while the non-secure application starts at `0x0804 0000`. In the secure linker script, an area of 8 Kbytes is set aside to contain the non-secure callables, which is the glue that allows a non-secure application to call a defined set of functions in the secure area.

The linker script must be aligned with the memory partitioning header file. In this example, the memory partitioning is defined in file `partition_stm32l552xx.h`. This header file contains the settings to configure the SAU. A summary of one memory region is presented below:

```
/* Initialize and enable the SAU */
#define SAU_INIT_CTRL          1
#define SAU_INIT_CTRL_ENABLE   1
...
/* <e>Initialize SAU Region 1 with memory attributes */
#define SAU_INIT_REGION1       1
#define SAU_INIT_START1        0x08040000      /* start address of SAU region 1 */
#define SAU_INIT_END1          0x0807FFFF      /* end address of SAU region 1 */
/* Region can be set as: 0 = non-secure, 1= secure, non-secure callable */
#define SAU_INIT_NSC1          0
```

In total, eight memory regions can be configured in non-secure or secure/non-secure callable. The SAU is configured as part of the boot sequence:
`Reset_Handler()` → calls `SystemInit()` → calls the inlined `TZ_SAU_Setup()` function.

`TZ_SAU_Setup()` sets up the SAU.

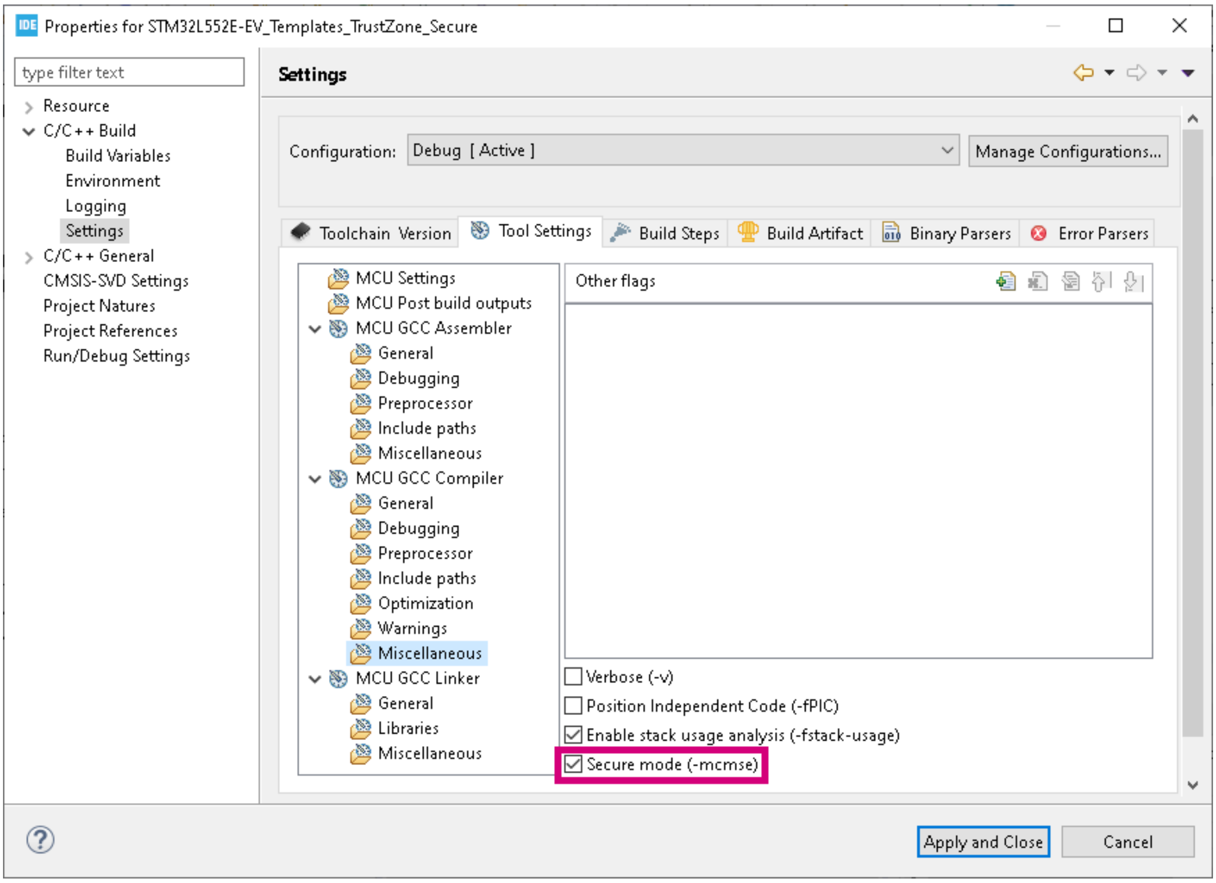
Note:
1. *The linker script and partition header file must be kept in coherence. If the linker script for the non-secure project defines a certain Flash area to be used for non-secure use, then this area must also be described properly in the partitioning header file. In the examples above, both the linker script and partitioning header file point out `0x0804 0000` to be used as non-secure Flash area.*
2. *The CPU cannot access the non-secure memory before the SAU is initialized. Any attempt by the CPU or the debugger to read the non-secure Flash results in RAZ (Read As Zero): Only zeros are returned. The `TZ_SAU_Setup()` function must be executed to give access to the non-secure Flash.*
3. *The debugger is able to program the non-secure binary by configuring the SAU with a dummy configuration. The debugger issues the reset command after Flash loading is complete to allow debug using the SAU initialized with the user's SAU settings.*

2.2.3 TrustZone®-related build settings

In a TrustZone®-enabled STM32L5 project, some additional build settings are configured. The user is advised not to change these settings in the typical use case.

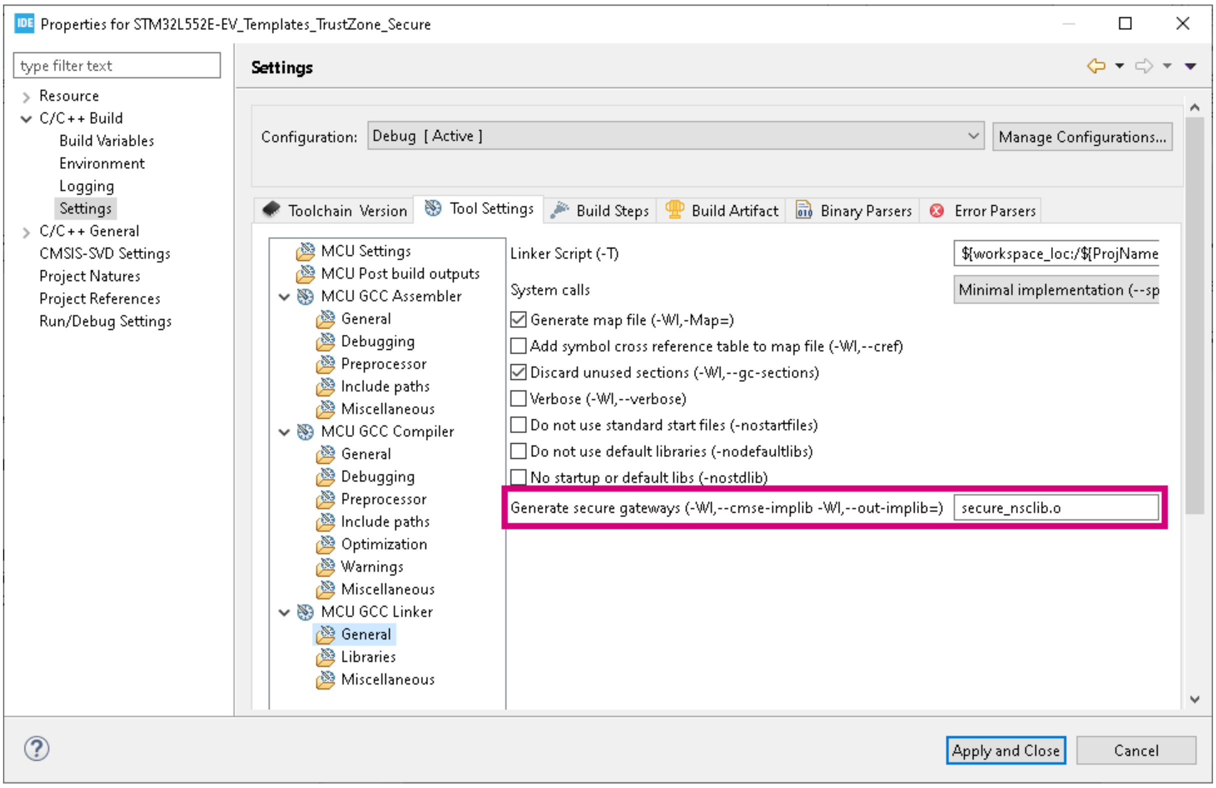
In the secure project, the compiler is called with the `-mcmse` attribute.

Figure 7. STM32L5 secure project: compiler call



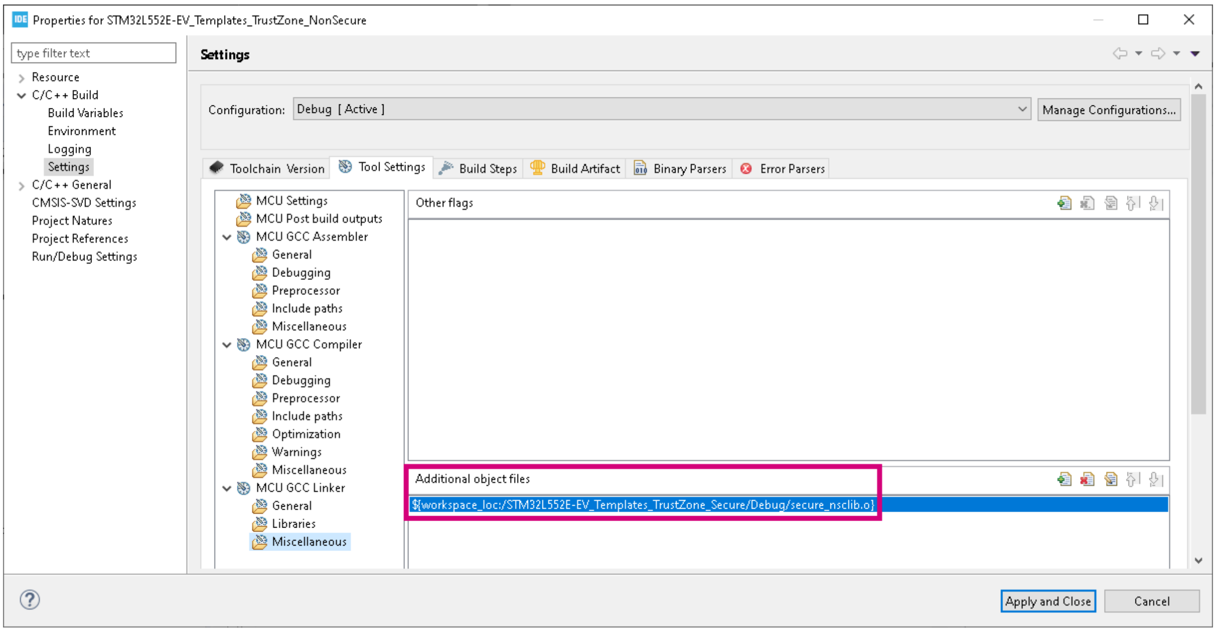
In the secure project, the linker is configured to generate secure gateways for the non-secure callables.

Figure 8. STM32L5 secure project: linker configuration



In the non-secure project, the linker is configured to link objects from the non-secure callable library that is built by the secure application.

Figure 9. STM32L5 non-secure project: linker configuration



No change is needed in this example project. The non-secure project includes the secure library from the secure project. Therefore, the secure project is scanned for changes and built before the non-secure project, if necessary.

User checkpoint

At this point, it is recommended that users evaluate their proper understanding of the build mechanism before proceeding further. Try building the non-secure project and thereby auto-trigger the build of the secure project.

2.2.4 RDP-level 0: loading and debugging both secure and non-secure projects

Loading the applications into the STM32L5 target can be done with any of the following tools:

- ST-LINK GDB server, by invoking a bundled STM32CubeProgrammer CLI version
- OpenOCD
- STM32CubeProgrammer stand-alone

This application note focuses on the use of the ST-LINK GDB server and OpenOCD. Unless noted otherwise, all screenshots apply to both.

Note: It is not possible to create debug configurations for the root project but only for the two application projects: secure and non-secure.

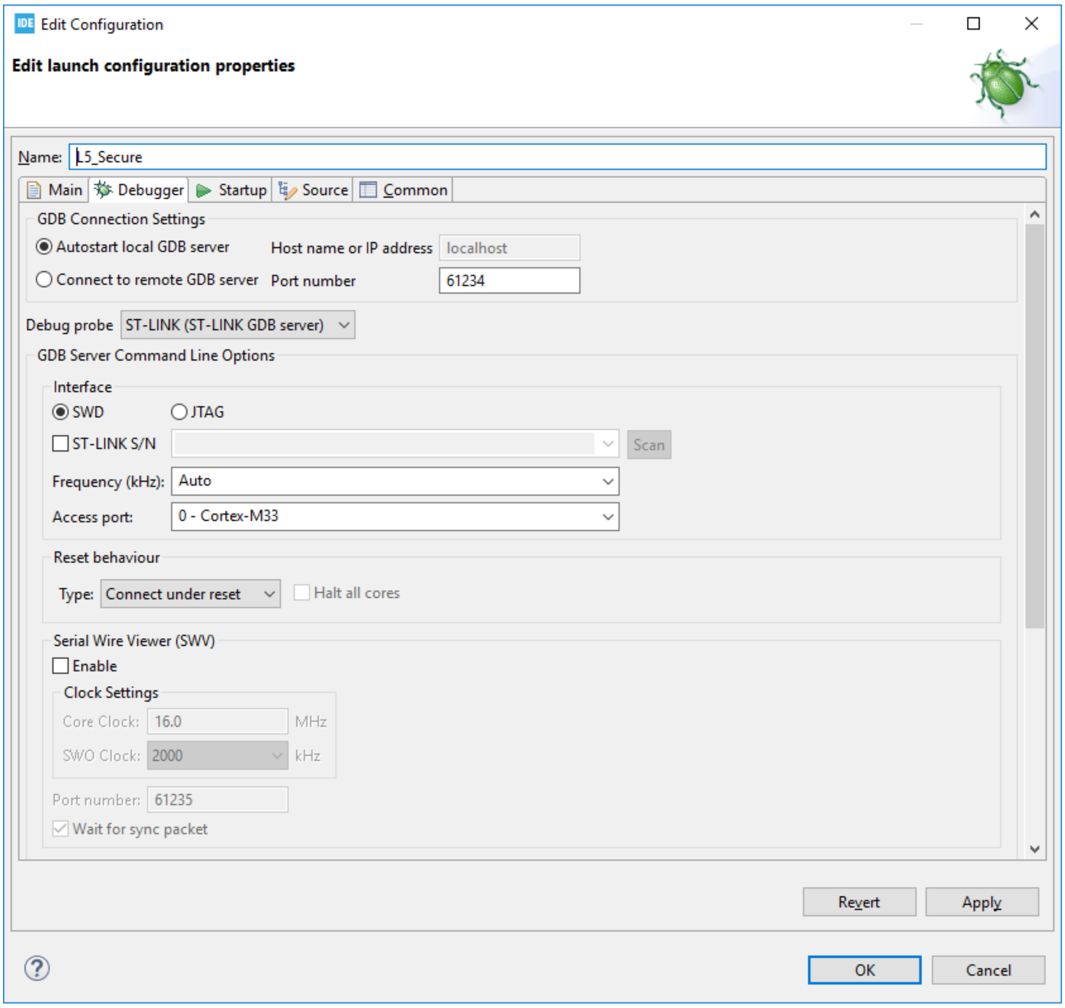
In the use case considered, it is assumed that the user has full access to all code and wants to debug the complete application. The Option Byte RDP-level must remain set to 0 (`0xAA`).

To create a debug configuration, perform the following steps:

1. Select the secure project in [**Project Explorer**].
2. Right-click [**Debug As…**] and select [**STM32 Cortex-M C/C++ Application**].

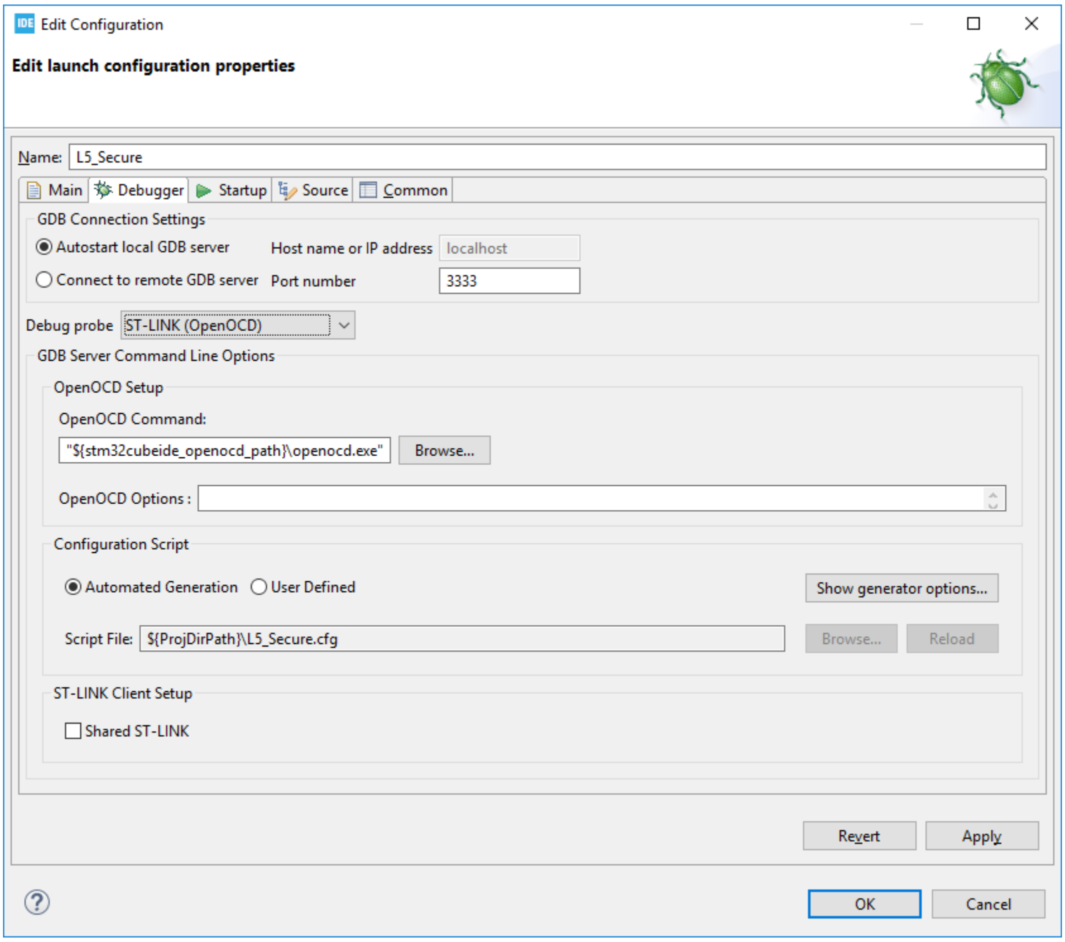
3. Move to the *Debugger* tab.
   – To use the ST-LINK GDB server, keep all fields with their default values as shown in Figure 10.

Figure 10. *Debugger* tab with ST-LINK GDB server selection



– To use OpenOCD, select the [**Debug probe**] as *ST-Link (OpenOCD)* and keep the default values as shown in Figure 11.

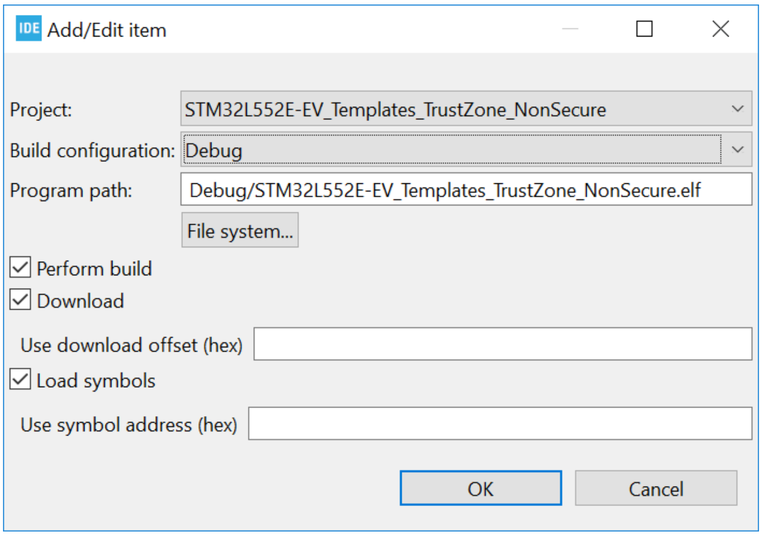
Figure 11. *Debugger* tab with OpenOCD selection



4. Move to the *Startup* tab. In the *Load Image and Symbol* table, only the secure binary is currently added. In order to load and debug both the secure and non-secure binaries, the user must manually add the non-secure binary to the load list.

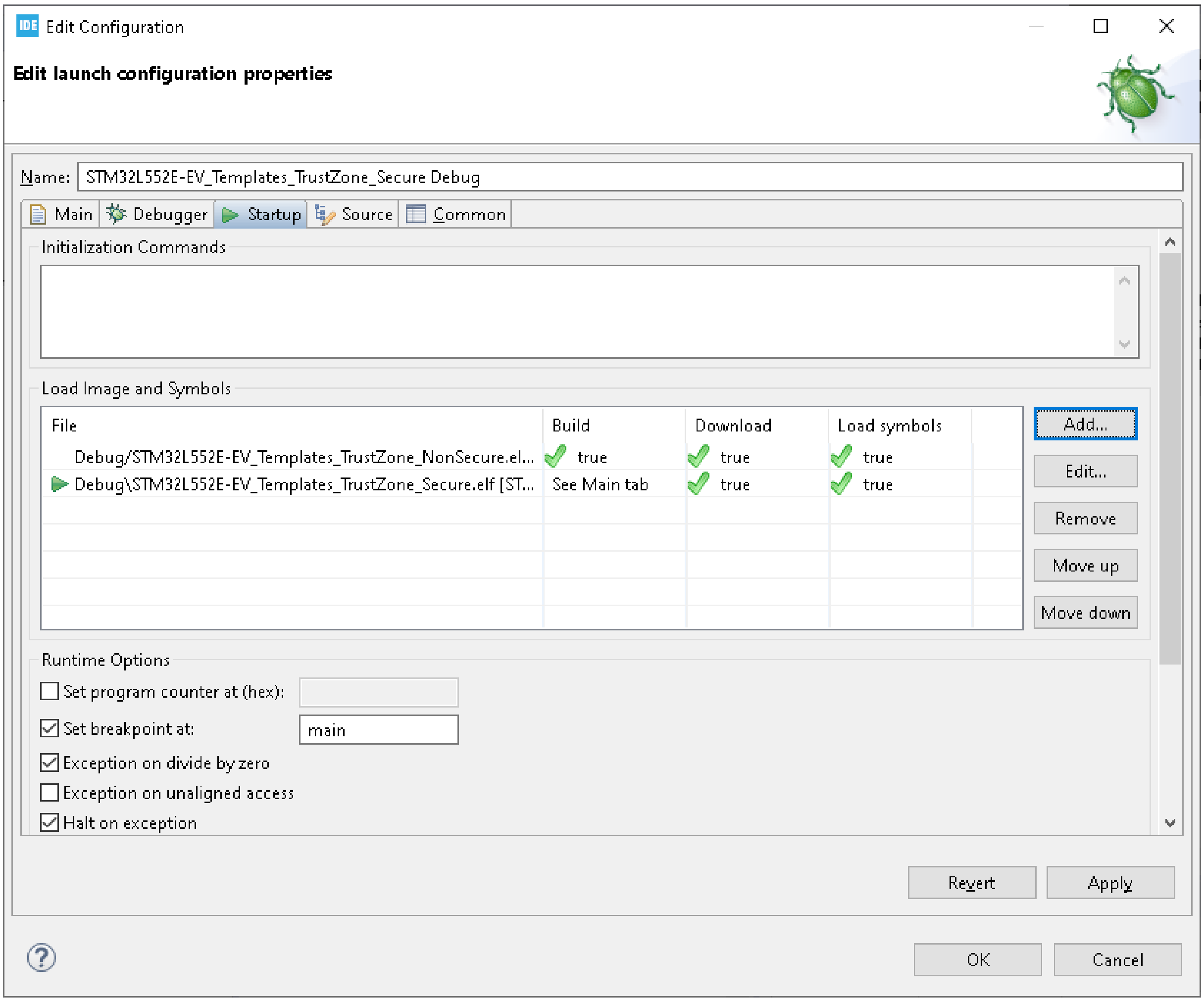
5. Click [**Add…**]
   a. *Project*: Select the non-secure project
   b. *Build Configuration*: Debug
   c. Make sure that the *Download* and *Load symbols* checkboxes are checked

Figure 12. Add binary and symbol loads to non-secure project



After adding the non-secure `elf` file to be part of the list of `elf` files to be loaded to the embedded target, the load list looks as shown in Figure 13.

Figure 13. Startup configuration load list



Attention: *STM32L5 Series devices always boot in secure state when TrustZone® is enabled. The debugger sets the Program Counter using information from the last image in the Load image and Symbols table. Make sure the secure image is at the bottom of the load list.*

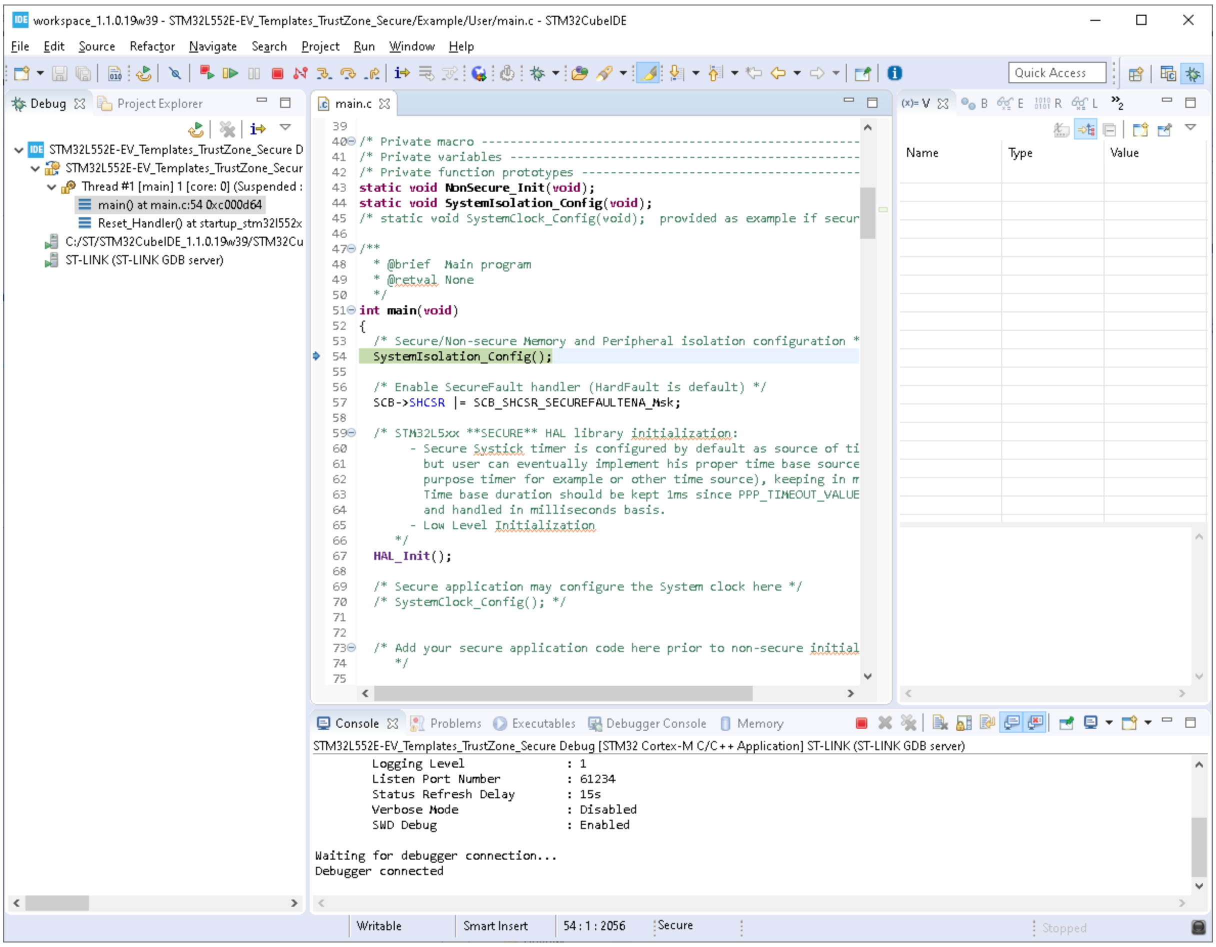
Note: Before launching a debug session, STM32CubeIDE checks if some code changes require a new build. The time required for such a check grows with large code bases. Some users may want to disable this check to get a faster debug launch procedure. In this case, they must also keep track of build changes and make sure to build manually.

The debug configuration setup presented above guarantees that any change in the code since the last build triggers a new build. Both images are downloaded and GDB loads symbols from both binaries in order to be able to map instructions to C code.

Assuming that the STM32L5 device is connected and that Option Bytes are properly configured, click [**OK**] to launch the first debug session.

The application halts at the first line in `main()` of the secure application.

Figure 14. Debugger halted on secure main function



In `main.c:54 - SystemIsolation_Config()`: In this function, the secure side books memory blocks and peripherals to be accessed by the non-secure side.

Use stepping to step-through the first lines in the secure application and learn how the device is configured to make the jump from secure to non-secure context.

In `main.c`: There is a function call to `NonSecure_Init()`. Step into this function as shown in Figure 15.

Figure 15. Initialization of the secure to non-secure jump

```
  */
static void NonSecure_Init(void)
{
  funcptr_NS NonSecure_ResetHandler;

  SCB_NS->VTOR = VTOR_TABLE_NS_START_ADDR;

  /* Set non-secure main stack (MSP_NS) */
  __TZ_set_MSP_NS((*(uint32_t *)VTOR_TABLE_NS_START_ADDR));

  /* Get non-secure reset handler */
  NonSecure_ResetHandler = (funcptr_NS)(*((uint32_t *)((VTOR_TABLE_NS_START_ADDR) + 4U)));

  /* Start non-secure state software application */
  NonSecure_ResetHandler();
}
```

- `main.c:137`: The non-secure vector table offset register is initialized with the address to the interrupt vector for the non-secure application.
- `main.c:140`: The non-secure Main Stack Pointer is initialized.
- `main.c:143`: The address for the non-secure reset handler is fetched from the non-secure interrupt vector table. This is the second entry in the table.
- `main.c:146`: Executes the function pointer to jump to the non-secure reset handler.

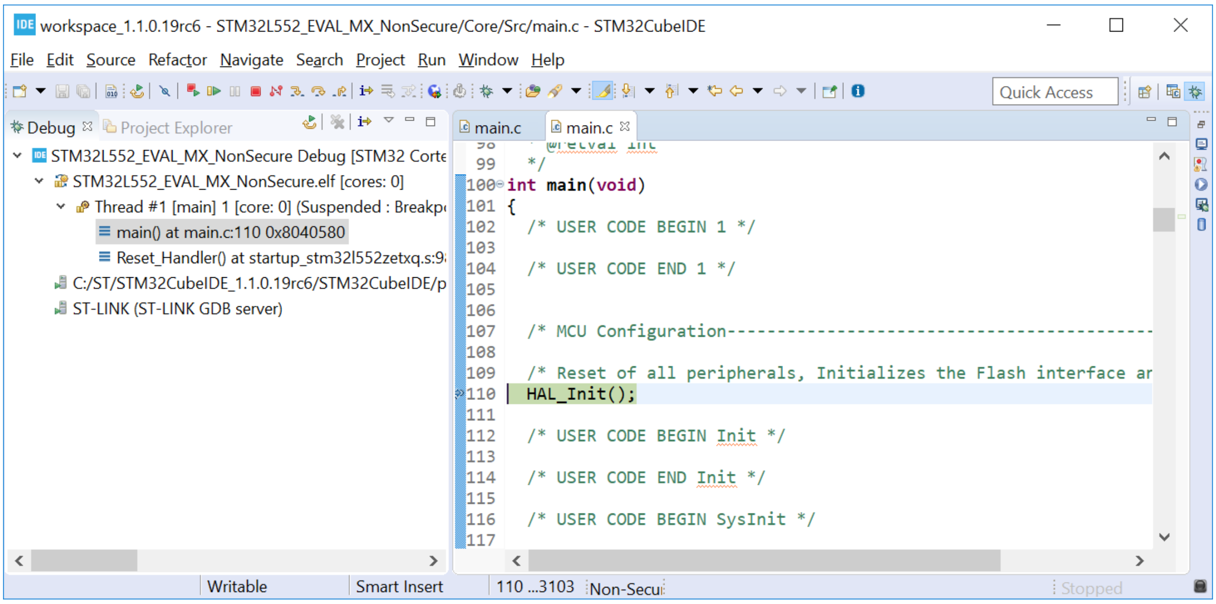
The function pointer mechanism is the only way to jump from secure to non-secure context. To learn more look inside file `main.h` in the secure project. It contains the following lines:

```
# define CMSE_NS_CALL  __attribute((cmse_nonsecure_call)) /* Function pointer declaration in
non-secure */
typedef void CMSE_NS_CALL (*funcptr)(void);
typedef funcptr funcptr_NS; /* typedef for non-secure callback functions */
```

Note: *The breakpoint in main is only set for the secure application. Therefore, in order to halt on first line in non-secure main, the user must set a breakpoint manually.*

After performing a *Step into* operation on `main.c:146`, or if the user manually sets up a breakpoint in non-secure `main()` and presses *Continue operation*, the execution halts on the first line in non-secure `main()` as shown in Figure 16.

Figure 16. Jump made from secure to non-secure



2.2.5 RDP-level 0.5: loading and debugging the non-secure project

In RDP-level 0.5 the debugger is not able to read or write any information related to the secure side. Consequently, the STM32L5 Series device must already contain a secure image, which initializes the SAU properly, to be able to debug the non-secure project. Furthermore the non-secure linker script and the SAU setup must be in sync. The example project used in this application note can be used as reference.

The presentation is the current section assumes that the secure side is already programmed and the RDP-level Option Byte is set to 0.5.

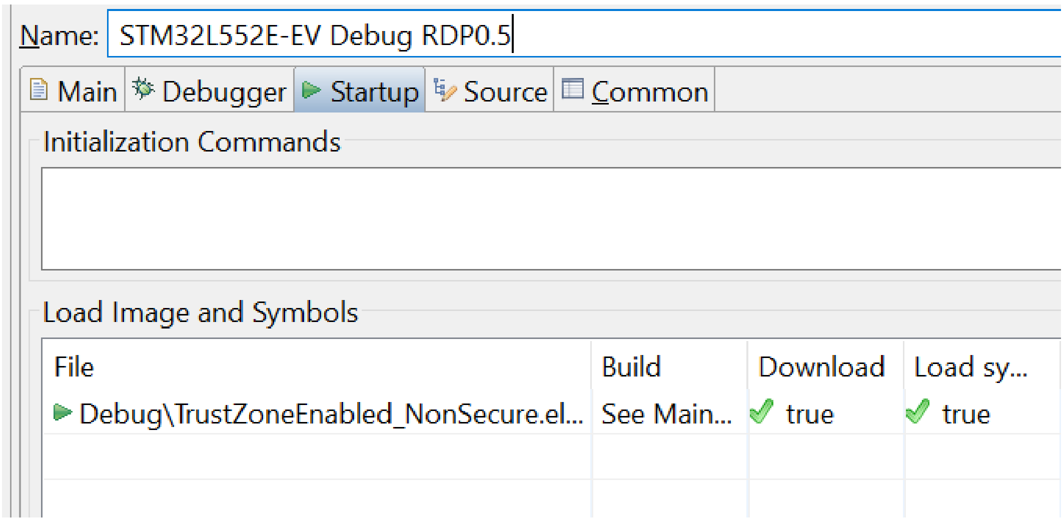
Debug configuration

To create the debug configuration to load the non-secure image, perform the following steps:

1. Select the non-secure project in [**Project Explorer**].
2. Right-click [**Debug As…**] and select [**STM32 Cortex-M C/C++ Application**].
3. Give the configuration a useful name so that it can be easily identified. In the case illustrated in Section 2.2.5 , the suffix *RDP0.5* is appended.

4. Move to the *Startup* tab. Verify that *Download* and *Load symbols* are set true for the non-secure project `elf` file.

Figure 17. Startup configuration load list for RDP 0.5



Click [**OK**] to launch the debug configuration. The device performs a reset and the debugger waits for the CPU to make the jump from the secure to non-secure context before it can halt execution.

Note: *In RDP-level 0.5, if the user tries to halt the CPU while it is executing secure code, the halt event is kept pending until the CPU returns to the non-secure side. If the CPU spends more than two seconds in the secure side, the halt operation is in timeout.*

In the "Debug Configurations" dialog, the "Startup" tab contains a "Max halt timeout(s)" selection, which can be configured for the debug probe of the ST-LINK GDB server to wait for longer timeout. For both debug probes to wait for longer timeout, ST-LINK GDB server and OpenOCD, a `.gdbinit` file needs to be created. This file must be available in `PROJECT_ROOT/.gdbinit` and contain the following commands to `gdb.`, where the values are in seconds and can be changed according to application needs:

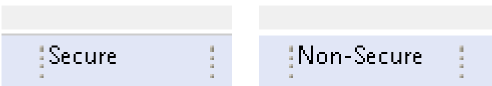
- `set remotetimeout 50`
- `set tcp connect-timeout 50`

2.2.6 TrustZone® specific extension in *Debug* perspective and views

Specific features are available in STM32CubeIDE to support STM32 devices with enabled TrustZone®.

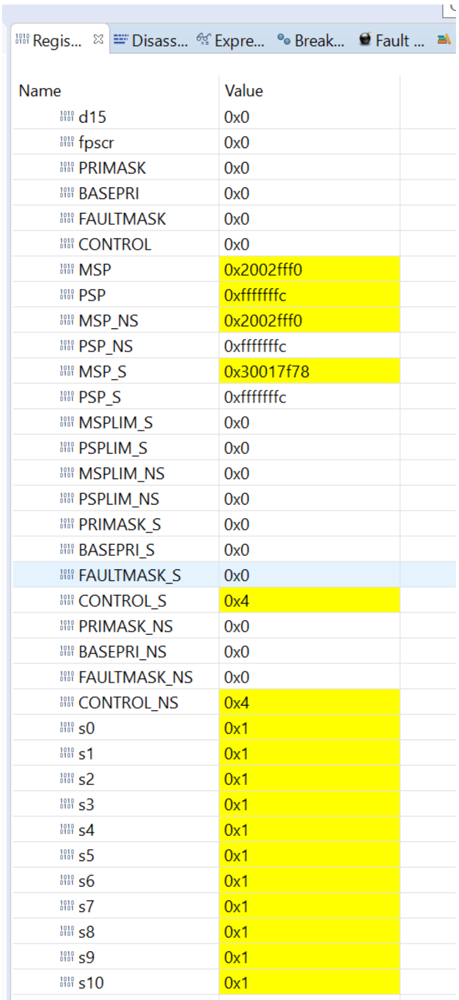
At the bottom of the *Debug* perspective, there is a *Security* indicator. This indicator shows either the *Secure* or *Non-Secure* status depending on the context in which the application execution is halted, secure or non-secure. Both states of this indicator are shown in Figure 18.

Figure 18. *Security* indicator



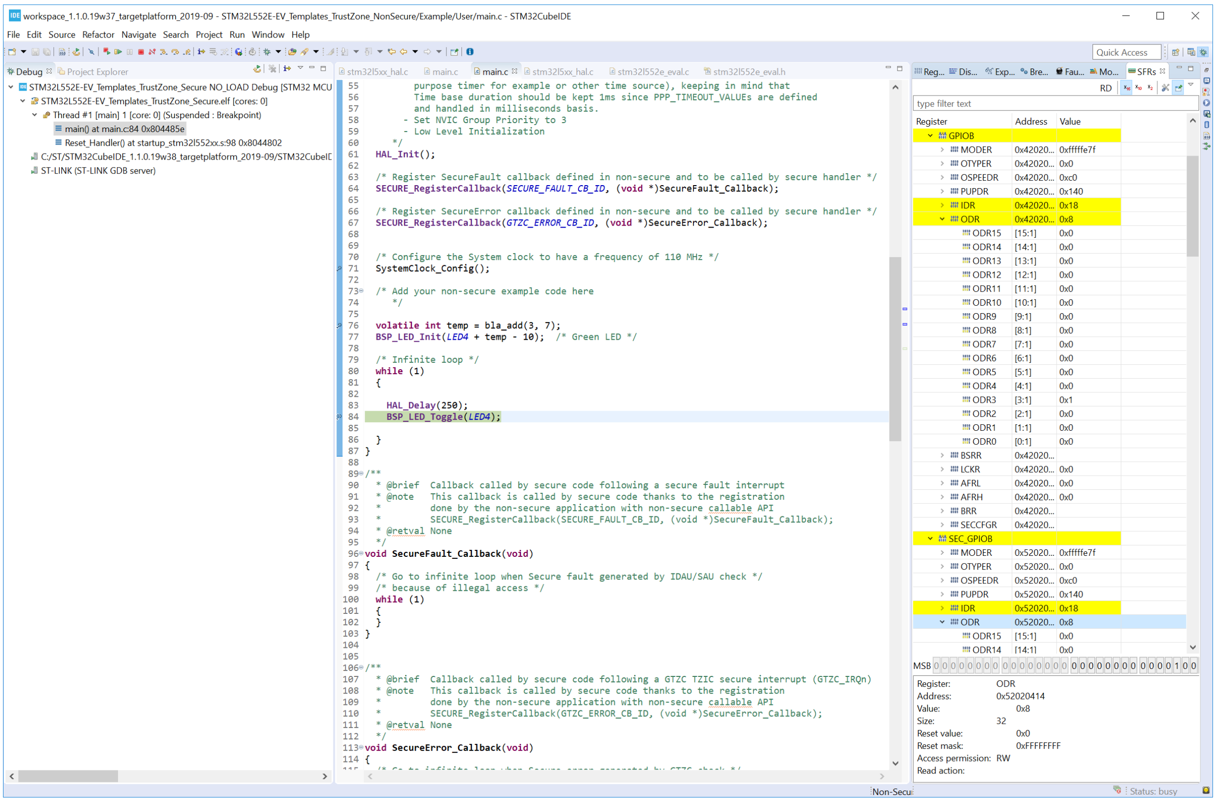
The *Registers* view is extended to display the banked secure and non-secure registers. These registers are suffixed by *_S* for secure and *_NS* for non-secure registers.

Figure 19. Banked secure and non-secure registers



The *SFRs* view shows the the content of file `CMSIS-SVD`. For each peripheral that can be shared between secure and non-secure context, this file contains both the non-secure and secure memory address.

Figure 20. SFRs view showing peripherals with both secure and non-secure address spaces



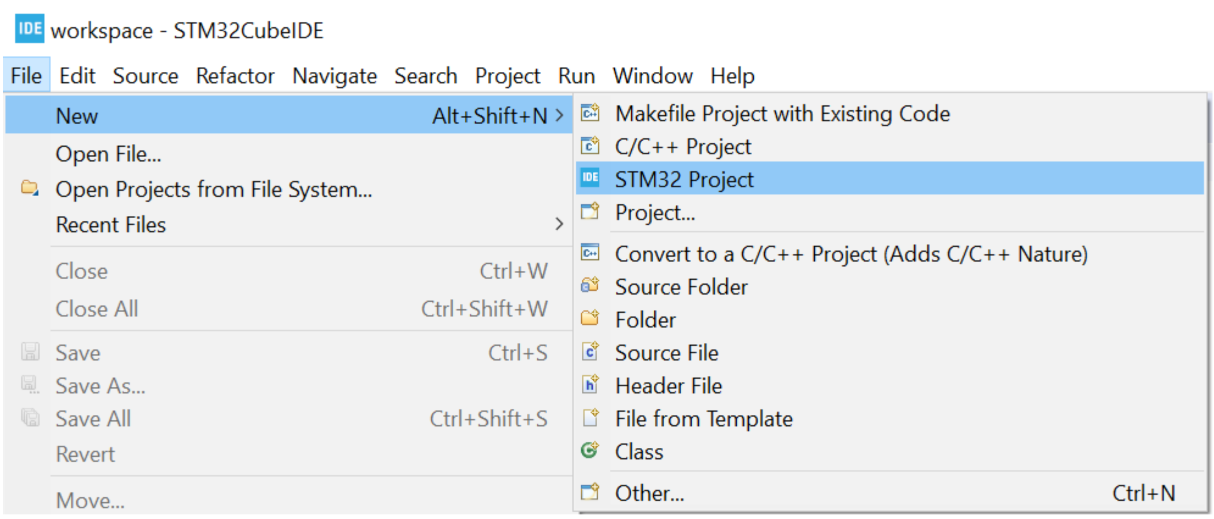
The *Fault Analyzer* view is also updated. It shows the new exception types applicable to the STM32L5 devices and calculates the exception stack frame based on fault conditions and FPU usage.

2.3 Create an empty project with TrustZone® enabled

This chapter assumes that the reader is familiar with Section 2.1 Importing the TrustZone project template for STM32CubeIDE and knows about Option Bytes, memory partitioning, building, and debugging.

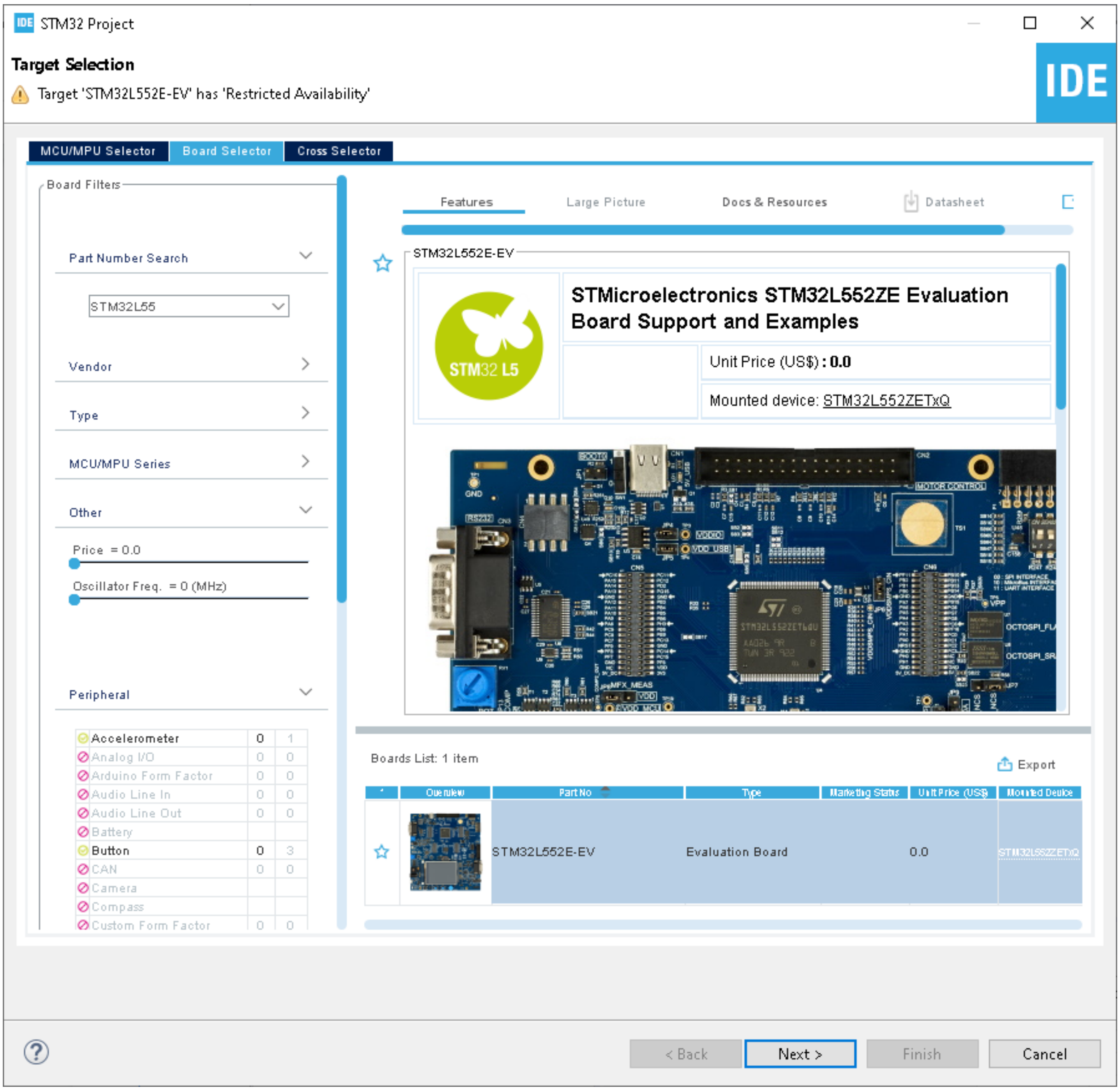
To start a new project, go to [**File**]>[**New**]>[**STM32 project**].

Figure 21. New empty project creation



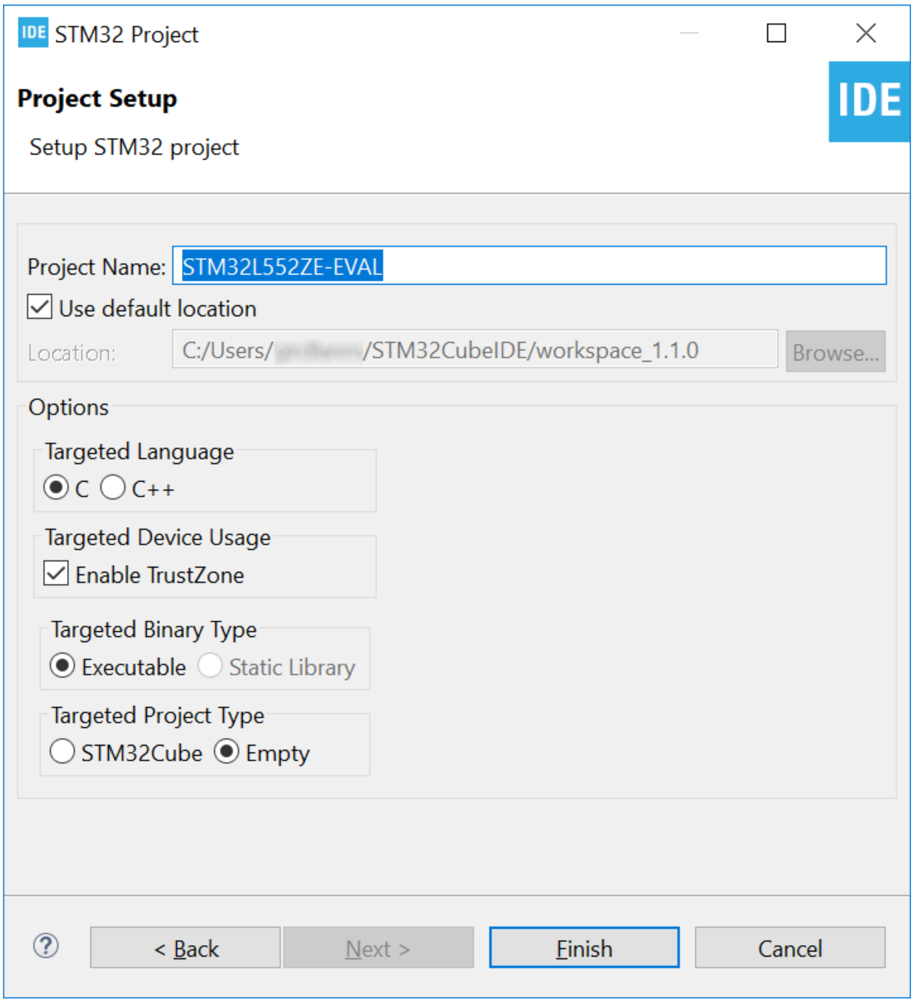
Select the MCU or board. In this example, an STM32L552E-EV Evaluation board is selected. Click [**Next**].

Figure 22. Target selection



After selecting an MCU or board, the next step is the *Project Setup* step.

Figure 23. **Project setup - Select Empty**



- Name the project.
- Make sure that *Enable TrustZone* under *Targeted Device Usage* is checked.

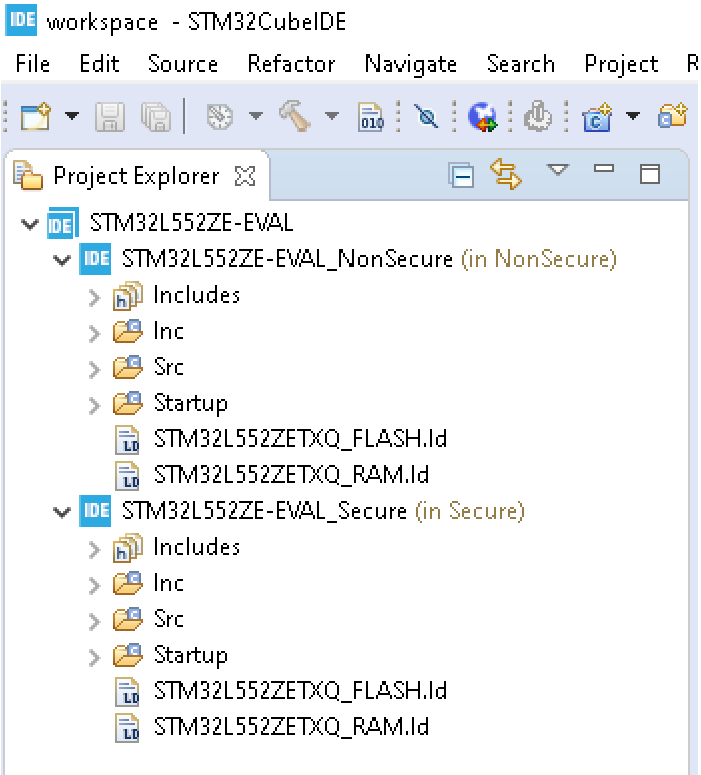
  This guarantees that the project is generated as three projects in a hierarchical structure instead of a single flat project. The root project has no CDT nature and therefore is not aware of concepts such as build configurations or debug configurations. The root project is just a container for the two target projects that permits sharing some common code between the two MCU projects.
  - The first MCU project is related to the non-secure part of the application. In that sense it resembles a legacy STM32 project.
  - The second sub-project is related to the secure part of the application. This project configures the compiler to build with the `-mcmse` flag and the linker to produce an object file for the non-secure callable functions.

  If *Enable Trust Zone* is not checked, the STM32L5 works like any other single-core STM32 microprocessor from a security standpoint. This is not the main use case, and hence not documented in this application note.
- *Targeted Project Type*: Select *Empty*. This results in an empty project skeleton, which can be manually populated with files.

Note:
1. *The "Enable TrustZone" selection is irreversible. It leads to either one flat project structure with TrustZone® disabled, or one hierarchical project structure with TrustZone® enabled. There is no way to switch between the two project structures once the projects are created.*
2. *Enabling TrustZone® in the project wizard does not enable Option Byte `TZEN`, which must also be set using STM32CubeProgrammer.*

After, naming the project, checking *Enable Trust Zone*, and setting *Targeted Project Type* to *Empty*, click on [**Finish**]. The empty project structure is created as illustrated in Figure 24.

Figure 24. Empty project structure



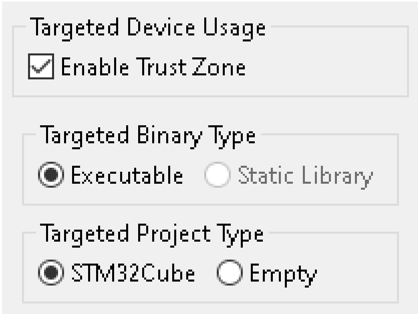
One root project contains two sub-projects. Only the two MCU sub-projects can be built and debugged. The root project is only a container. The secure `main()` function must be updated with code to call the non-secure application. How this is done can be studied in the secure template project, in other secure firmware example projects, or by creating an STM32CubeMX project with TrustZone® enabled.

How to debug these projects is described in Section 2.2.4 RDP-level 0: loading and debugging both secure and non-secure projects or Section 2.2.5 RDP-level 0.5: loading and debugging the non-secure project.

2.4 Create an STM32CubeMX project with TrustZone® enabled

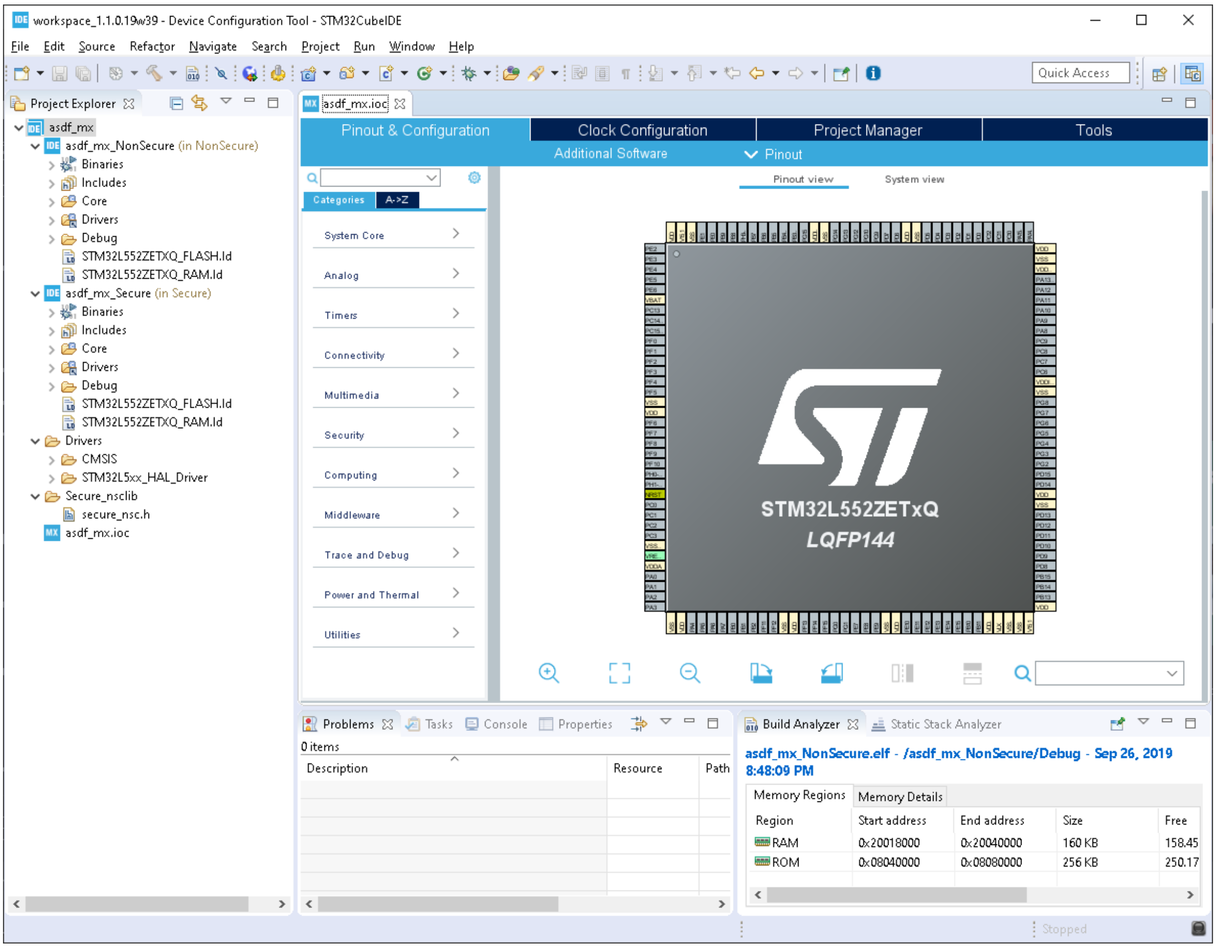
Perform the same steps as for the empty project in Section 2.3 Create an empty project with TrustZone enabled in order to create a project with an `.ioc` file where resources are controlled by STM32CubeMX. Remember to check the *Enable Trust Zone* checkbox and set the *Targeted Project Type* to *STM32Cube*.

Figure 25. **Project setup - Select STM32Cube**



[**File**]>[**New STM32 Project**]>[**Project Setup**] page which is the second step in the project wizards. Figure 26 shows a screenshot of the STM32L5 project generated by STM32CubeMX integrated in STM32CubeIDE.

Figure 26. **Generated project from STM32CubeMX**



The `Drivers` folder containing CMSIS and HAL code is stored in the root project and links are created to its code in order to build the code in the two MCU projects.

How to debug these projects is described in Section 2.2.4 RDP-level 0: loading and debugging both secure and non-secure projects or Section 2.2.5 RDP-level 0.5: loading and debugging the non-secure project.

3 Making calls from the non-secure to the secure domain

This section illustrates how to make a function call from the non-secure application to the secure application. This is done using GCC pragmas and a glue layer called the non-secure callable, abbreviated as NSC.

In order to make a call from a non-secure function to a secure function, the secure function must be defined with `__attribute((cmse_nonsecure_entry))` as shown in the example below:

```
uint32_t __attribute((cmse_nonsecure_entry)) getSecureKey(void)
{
    return 0xdeadbeef;
}
```

*Important: The function prototype in the corresponding header file must **not** use the `uint32_t __attribute((cmse_nonsecure_entry))`.*

To quickly try this example:

- Copy-paste the code snippet above into the secure application `main.c`. Place the snippet inside the `USER CODE BEGIN PV Begin/End` section or similar.
- In the root project, open the `Secure_nsclib/secure_nsc.h` header file. Add the following line into `secure_nsc.h`:

  ```
  uint32_t getSecureKey(void);
  ```

  This header file is a convenient example since it is included by both the secure and non-secure projects.
- Add a call in non-Secure `main.c`:

  ```
  int temp = getSecureKey();
  ```

- Build the non-secure project. This triggers the build of the secure project.
  The build of the secure project results in `secure_nsclib.o`, which is linked by the non-secure application to allow non-secure-to-secure transaction.
- Place a breakpoint on the line containing `int temp = getSecureKey();`
- When reaching the breakpoint, enable the instruction stepping mode. Use *step into*.
  - A veneer function, `_getSecureKey_veneer`, is generated in the non-secure code to handle the long jump between non-secure and secure memory addresses. This is not security related but the result of a long jump.
  - The next step consists in executing the `sg` (secure gateway) instruction in the secure memory, which authorizes non-secure-to-secure transactions.
  - After the authorization, the execution branches to the `getSecureKey()` function in the secure memory.
  - It is possible that this function calls another secure function that is not a non-secure callable.
    - In RDP-level 0, instruction stepping can be continued also in this function.
    - In RDP-level 0.5, stepping in secure application is not allowed.
  - In the prologue of `getSecureKey()`, all relevant registers are cleaned from the secure-side leaking information into the non-secure side.
  - When the non-secure call is finished, execution returns to the non-secure side.

4 FAQs

4.1 The debugger crashes after loading the non-secure and secure images in RDP0

Answer #1

Double-check that the secure image is at the bottom of the load list table in the debug configuration. The last image in this list is used to setup the Program Counter boot address, which must be in the secure memory. Refer to Section 2.2.4 RDP-level 0: loading and debugging both secure and non-secure projects.

Answer #2

In RDP-level 0.5, the debugger times out if the application spends more than two seconds in the secure context. Try extending the timeout as described below.

In the *"Debug Configurations"* dialog, the *"Startup"* tab contains a *"Max halt timeout(s)"* selection, which can be configured for the debug probe of the ST-LINK GDB server to wait for longer timeout. For both debug probes to wait for longer timeout, ST-LINK GDB server and OpenOCD, a `.gdbinit` file needs to be created. The GDB client also must be instructed to use longer timeout values, which is done by creating a file in the project root folder named `.gdbinit`. This file must contain two lines:

- `set remotetimeout 50`
- `set tcp connect-timeout 50`

Refer to Section 2.2.5 RDP-level 0.5: loading and debugging the non-secure project.

4.2 I get secure GTZC interrupt at various times during debug

The application jumps to the GTZC interrupt routine if any of the GTZC/TZIC illegal access flags are raised and corresponding interrupt is enabled. The GTZC/TZIC illegal access flags can be raised because the debugger (ST-LINK GDB server or OpenOCD) tries to access non-secure memory addresses before the SAU is properly initialized.

For example, the following features of STM32CubeIDE and other IDEs trigger memory reads:

- Setting a breakpoint on a memory address triggers a read on this address.
- Having a *Memory Browser*, *Expressions* or any other view that reads data from memory, triggers reads on halt events or other non-transparent IDE events.

Possible workarounds are:

- Do not enable GTZC interrupt on debug builds.
- Clear the necessary GTZC/TZIC illegal access flags after SAU initialization before enabling the GTZC interrupt.

Revision history

Table 1. Document revision history

Date	Version	Changes
9-Jan-2020	1	Initial release.
12-Feb-2020	2	Updated *Section 4.2 I get secure GTZC interrupt at various times during debug*.
23-Jul-2020	3	Extended the description to the debug using OpenOCD in RDP-level 0: loading and debugging both secure and non-secure projects, RDP-level 0.5: loading and debugging the non-secure project and The debugger crashes after loading the non-secure and secure images in RDP0.

Contents

List of tables

List of figures