User Manual

Document No.: AN1126

G32R5xx IDE and Tool Chain User Guide

Version: V1.2

1 Introduction

In modern embedded system development, choosing appropriate development tools and environments is crucial for successfully achieving the project goals. Whether starting a new project or porting an existing one, developers need to adapt to different development environments and chip architectures. This document provides a guide to help developers configure and use common development environments when working with G32R5xx series MCUs (including G32R501 and G32R502).

To make full use of the information in this guide, users shall be familiar with the characteristics of the devices in use. Refer to the following technical documents:

- G32R5xx / G32R501 / G32R502 User Manual, Datasheet, Programming Manual, and Migration Manual (AN1138)
- Core-related documents, including the Armv8.1-M Architecture Reference Manual, etc.
- SDK Quick Start Guide (AN1125)

This document describes project configuration in Keil MDK, IAR Embedded Workbench for Arm, Eclipse, and pyOCD, covering project creation, build and link, download, and debug. Example projects in the SDK are organized by device under paths such as “driverlib/g32r501/examples/” and “driverlib/g32r502/examples/”.

This document guides readers through basic development skills for G32R5xx in the environments above.

Contents

2 Full Name and Abbreviation Description of Terms

The keywords and descriptions used in this document are as shown in Table 1.

Table 1 Abbreviation Description

Full name in English	English abbreviation
Configurable Static Memory Subsystem	CFGSMS
Software	SW
Hardware	HW
Interface	IF
AHB slave interface	ahbs_if

3 R5xx Series MCU Introduction

The G32R5xx (R5xx) series MCU is based on the Arm Cortex-M52 core. Among them:

- G32R501 is a dual-/single-core Cortex-M52 device;
- G32R502 is a single-core Cortex-M52 device.

On-chip memory and peripheral scale vary by orderable part number; ITCM / DTCM / SRAM sizes can be configured through CFGSMS. Later chapters describe project and debug setup for Keil MDK, IAR Embedded Workbench for Arm, Eclipse, and pyOCD. Example paths are split by device (for example driverlib/g32r501 and driverlib/g32r502).

4 MDK-ARM development tool chain

During the migration from Txx320F28004x series MCU to G32R5xx series MCU, the porting of development tools is an important link. Code Composer StudioTM (CCStudio) integrated development environment (IDE) and MDK-ARM are both commonly used tools in embedded development.

This chapter will introduce how to migrate the existing CCStudio projects to the MDK-ARM environment, and discuss in detail the compatibility of C language and assembly code, and the configuration of linker script files.

4.1 Debugger Support

Debugger support for G32R5xx (including G32R501 / G32R502):

- Geehy-Link (WinUSB), DAP Link (the firmware version is CMSIS-DAP V2 and above)
- J-Link V12 (J-Link V7.94g and above)
- Ulink Pro

4.2 IDE Version

Please make sure to use Keil MDK V5.40 or higher.

Note: When the Device is Cortex-M52, F12 (Go to Definition) may not work on MDK versions below 5.43; use MDK 5.43 or later when source navigation is required. On MDK 5.43.1, some 16-bit register fields may display incorrectly; use MDK 5.40 temporarily or verify values in the Memory window.

4.3 Project Operation

This section describes Pack installation, project open/create, build/download, and debug related operations in Keil MDK.

Note:

（1） Keil MDK **5.43** or later is recommended (Cortex-M52 / F12 and related capabilities).

（2） Screenshots and step-by-step examples below use the v5.40 UI; menu paths are equivalent on 5.43.

（3） On MDK 5.43.1, some 16-bit register fields may display incorrectly; use 5.40 temporarily or verify values in the Memory window.

4.4 Install Pack support

You can choose any of the following methods for installation:

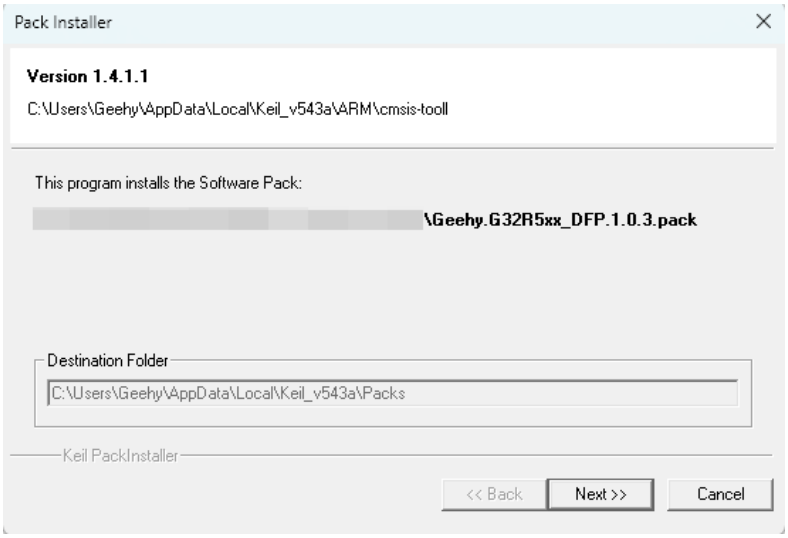
Direct installation

1. Look for the file “Geehy.G32R5xx_DFP.x.x.x.pack” under the package directory of SDK.
2. Double-click the file and install in the pop-up installation interface (as shown in Figure 1) according to the instructions.

Note: The following screenshots use a G32R501 example (target name often g32r501). When developing for G32R502, select the matching G32R502 device/target, replace g32r501 with g32r502 in paths, and use

r502_dbg.ini as the MDK initialization file.

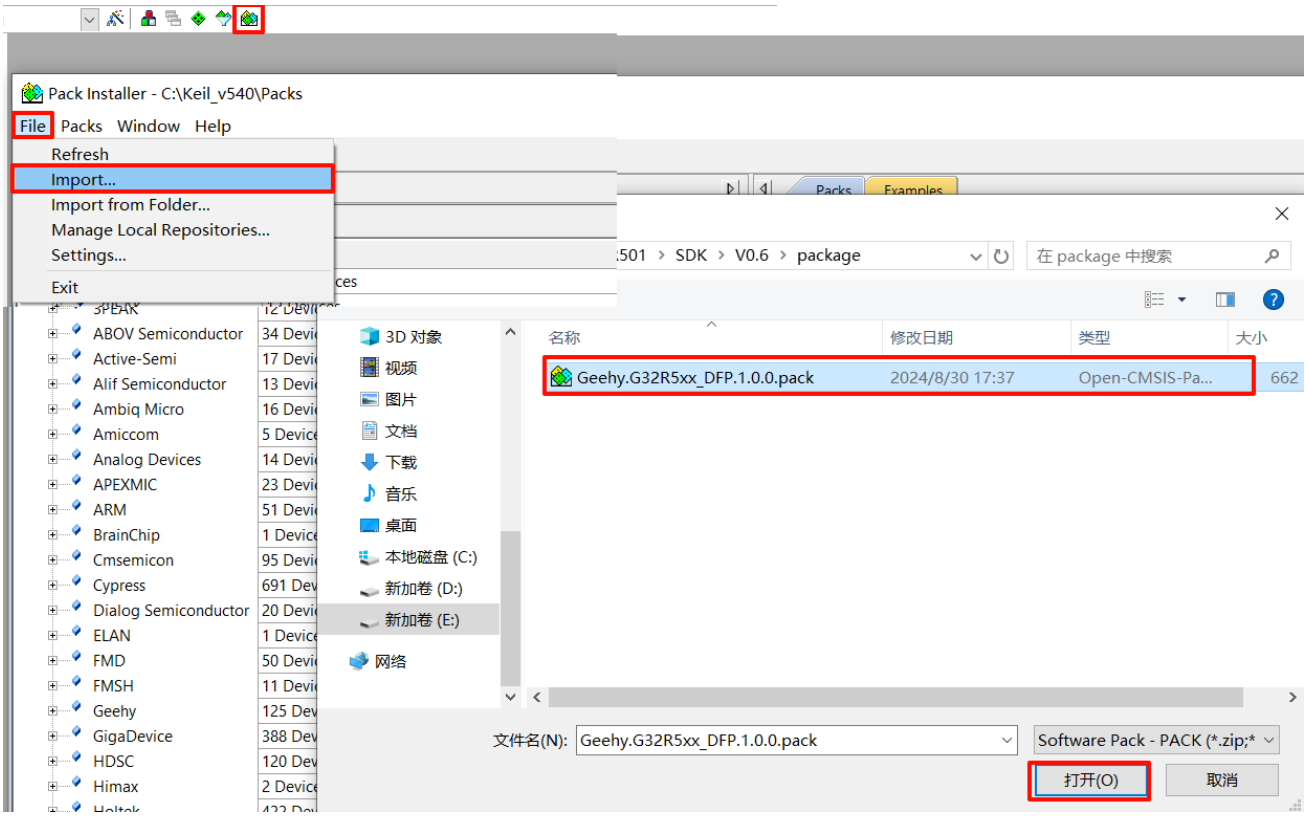
Figure 1 Click to install Geehy.G32R5xxDFP.x.x.x.pack



Use Pack Installer (as shown in Figure 2**)**

1. Open MDK-ARM v5.40.
2. In the menu bar, click “Project” → “Manage” → “Pack Installer”.
3. In the new window, select "File" → "Import...".
4. In the file browser, find “Geehy.G32R5xx_DFP.x.x.x.pack” under the package directory of SDK, and select it.
5. Click "Open" to install.

Figure 2 Import Geehy.G32R5xxDFP.x.x.x.pack in Pack Installer



4.5 Open the example project

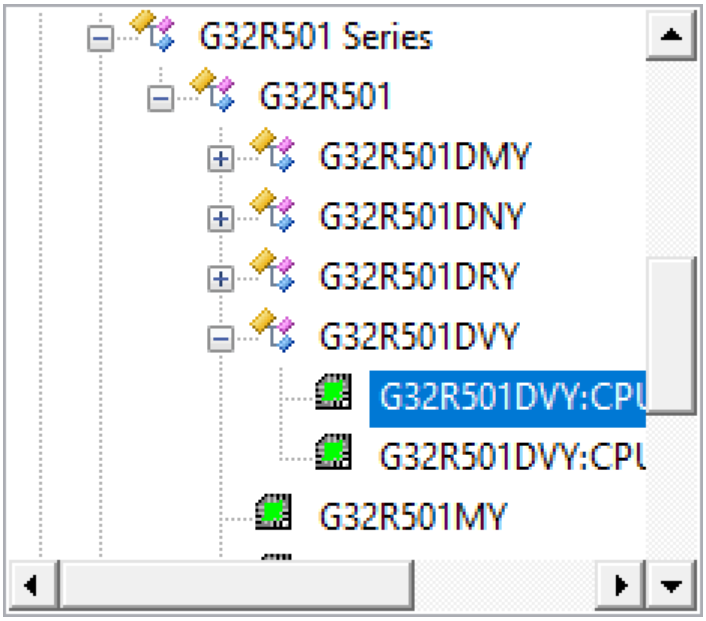
1. Start MDK-ARM v5.40.
2. Click "Project" → "Open Project" in the menu bar.
3. Browse the path of the SDK project file you provide, select the corresponding “.uvprojx” file, and then click "Open".
4. Or after installing MDK-ARM v5.40, directly click the “.uvprojx” project file to open it.

Note: Please complete the above steps after Pack support installation; otherwise MDK-ARM will indicate that the chip cannot be found.

4.6 Project establishment

1. Start MDK-ARM v5.40.
2. Click "Project" → "New uVision Project" in the menu bar.
3. Select the path to save the project, enter the project name, and click "Save".
4. In the pop-up dialog box, select the appropriate G32R5xx series MCU model, and click "OK".

Figure 3 Select the microcontroller



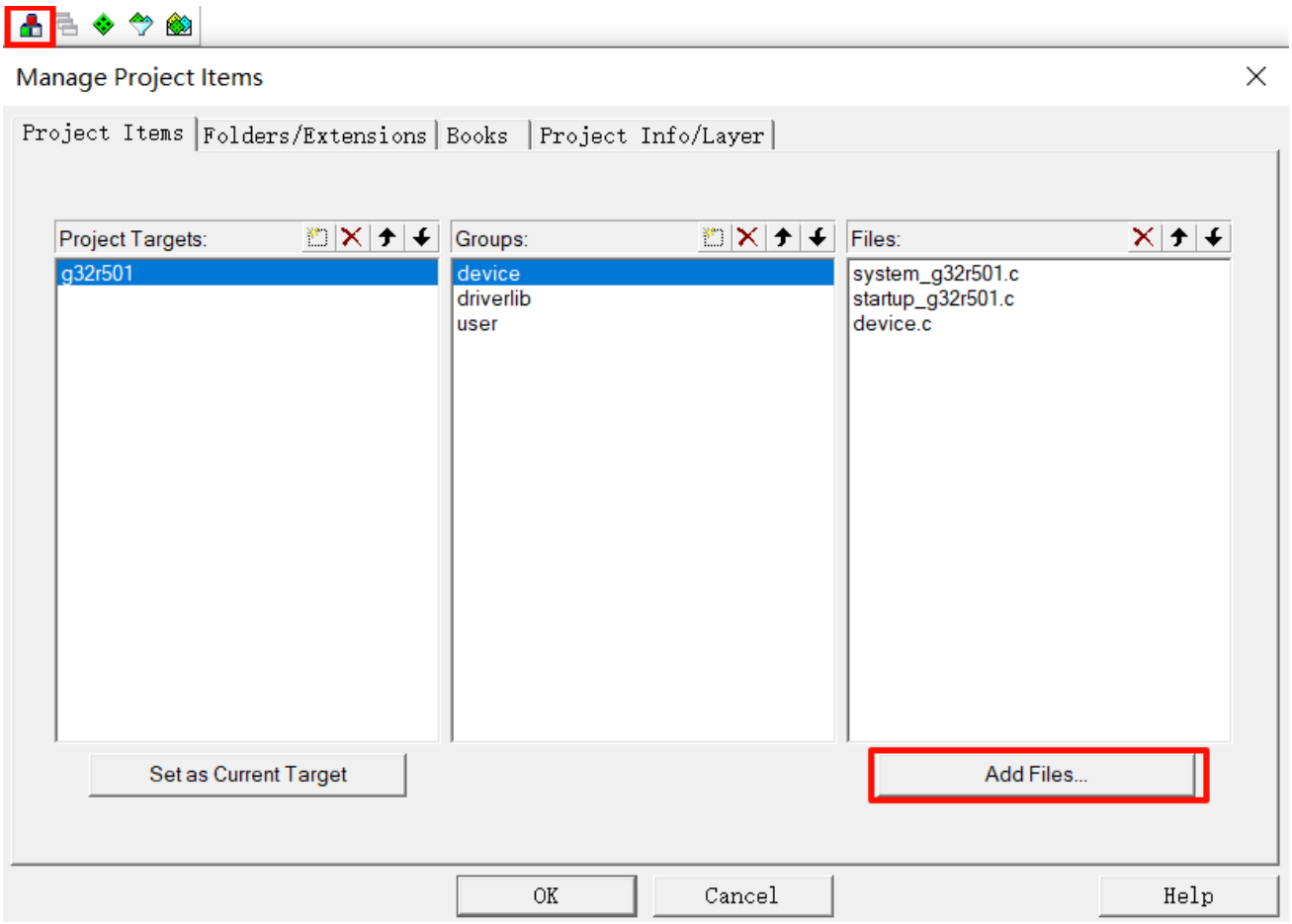
4.7 File import

1. Make sure your new project is already open in the Project window.
2. Click "File Extensions, Books and Environment…".

Figure 4 File Extensions

3. Click "Source Group 1" or the folder where you want to add files, and select "Add Files...".
4. In the pop-up file browser, find and select the source files to be imported (e.g. .c and .h files), and then click "Add".

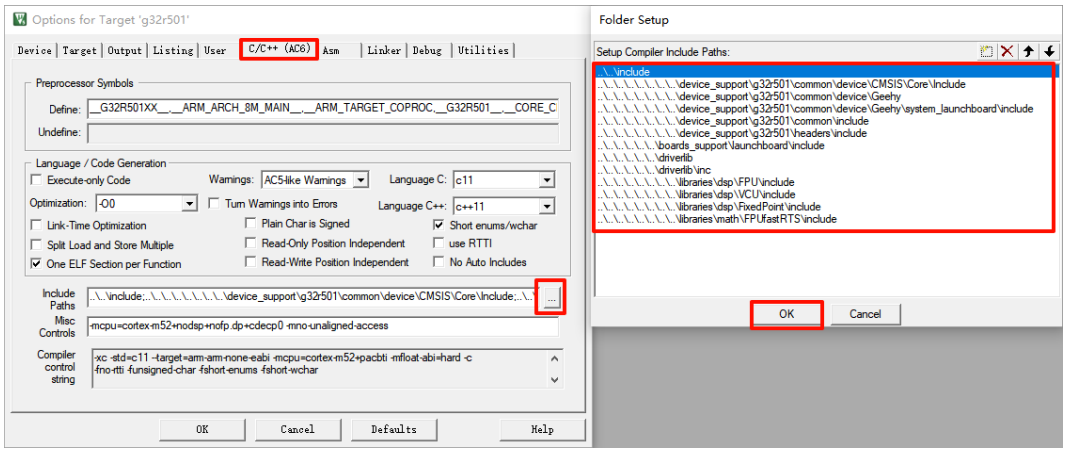
Figure 5 Add source files



5. To create a new source file, right-click "Source Group 1" and select "Add New Item"; enter the file name, select the file type (e.g. C file or assembly file), and click "Add".

6. Select "Project" → "Options" in the menu, check "Include Paths" under the "C/C++" tab, and ensure that all necessary library file paths have been added (as shown in Figure 6).

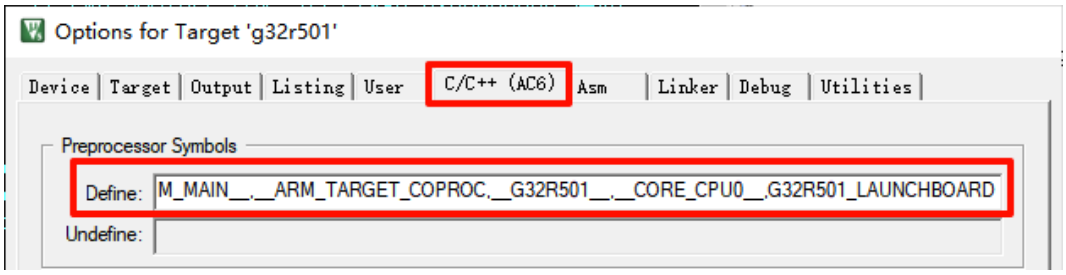
Figure 6 Add include paths



4.8 Configure macro definition

In "Project"->"Options", select the "C/C++" tab, add the required macro definition, and configure the corresponding compilation control.

Figure 7 Configure macro definitions



4.9 Compilation command control

In MDK-ARM, compilation commands can be controlled through the "Misc Controls" under the "C/C++(AC6)" tab in the "Options for Target" window of the project attributes. Under the "C/C++(AC6)" tab, users can set the preprocessor directives, compiler flags, and so on, refer to the compiler command support in Table 2; for more related information, please consult the help documentation in MDK.

Table 2 Compiler Command

Characteristic/Option	Scalar FP half-precision	Scalar FP single-precision	Scalar FP double-precision	MVE integer	MVE FP half-precision	Custom Datapath Extension (CDE) cp0
cortex-m52	Included	Included	Not included	Not included	Included	Not included
cortex-m52+ nomve	Included	Not included	Included	Not included	Included	Not included
cortex-m52+ nomve.fp+ nofp.dp	Included	Not included	Included	Not included	Included	Not included
cortex-m52+ nomve+ nofp.dp	Not included	Not included	Included	Not included	Included	Not included
cortex-m52+ nomve.fp+ nofp	Included	Not included	Not included	Not included	Included	Not included
cortex-m52+ nopacbti	Included	Included	Not included	Not included	Not included	Not included

Characteristic/Option	Scalar FP half-precision	Scalar FP single-precision	Scalar FP double-precision	MVE integer	MVE FP half-precision	Custom Datapath Extension (CDE) cp0
cortex-m52+ nomve+ nopacbti	Included	Not included	Not included	Not included	Not included	Not included
cortex-m52+ nomve.fp+ nofp.dp+ nopacbti	Included	Not included	Not included	Not included	Not included	Not included
cortex-m52+ nomve+ nofp.dp+ nopacbti	Not included	Not included	Not included	Not included	Not included	Not included
cortex-m52+ nomve.fp+ nofp+ nopacbti	Not included	Not included	Included	Not included	Not included	Not included
cortex-m52+ cdecp0	Not included	Not included	Not included	Not included	Not included	Included

Figure 8 MDK compiler documentation

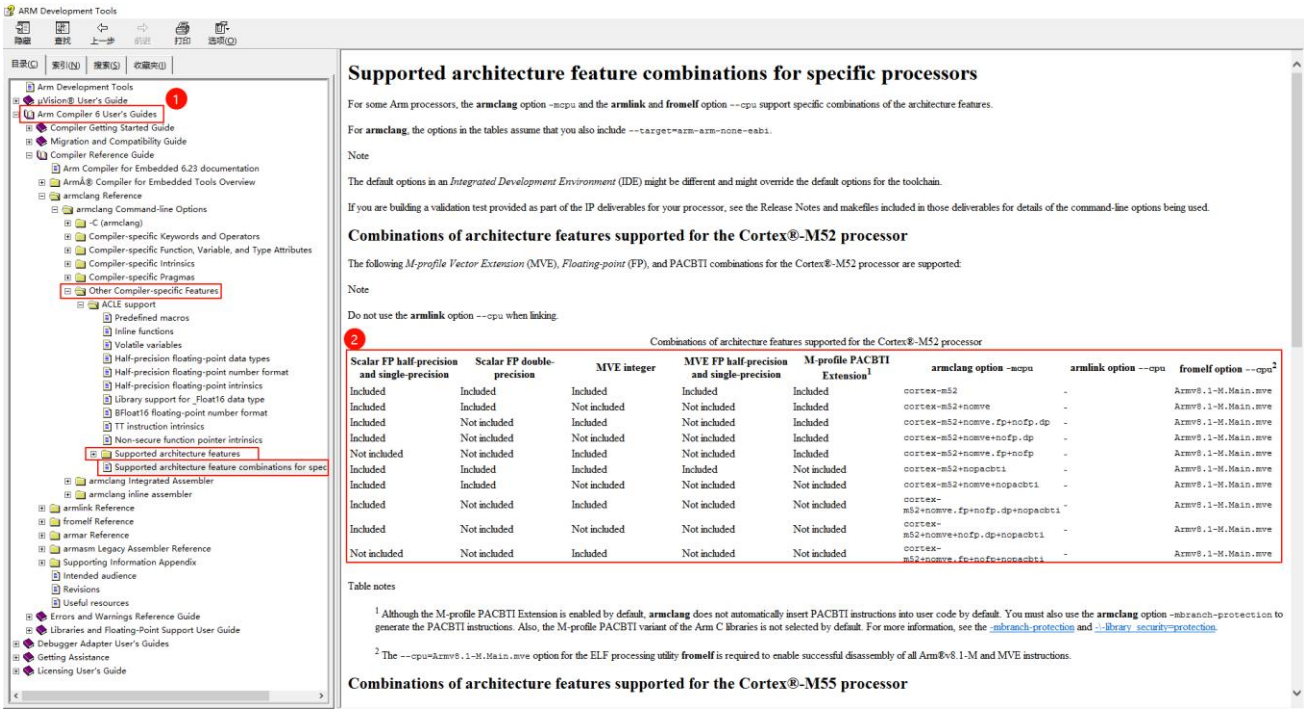
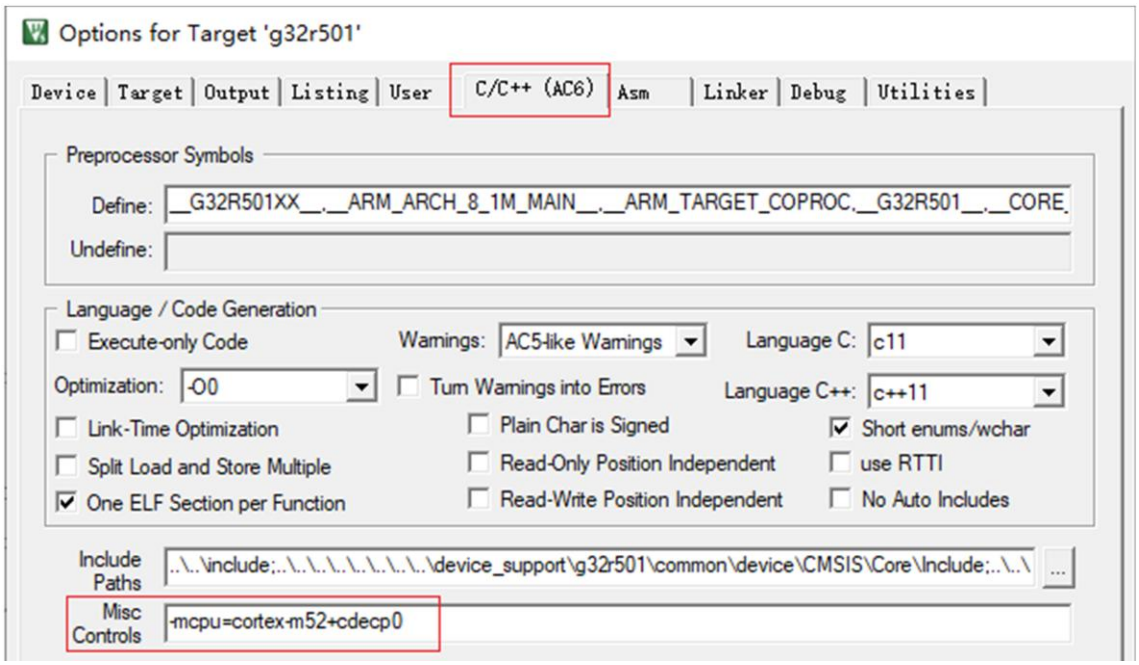


Figure 9 Compilation command control

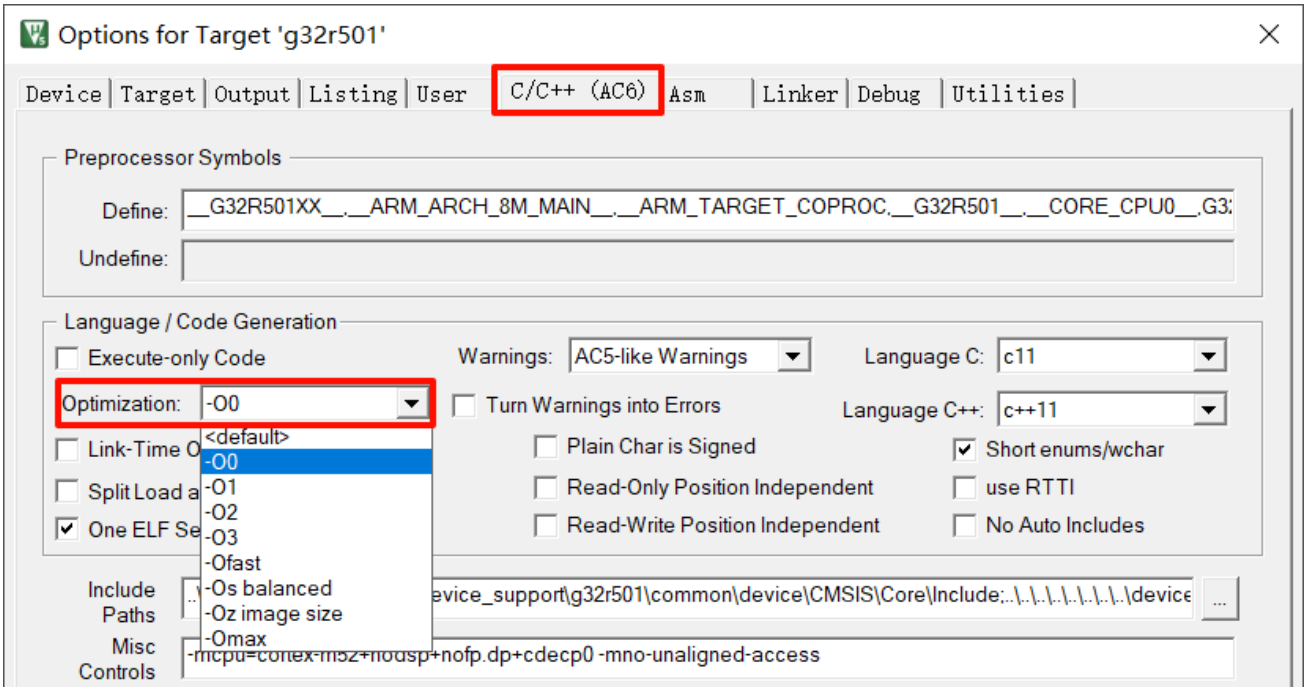


4.10 Compilation optimization level settings

MDK-ARM provides multiple optimization level settings, which can be adjusted at the global, single-file, and single-function levels:

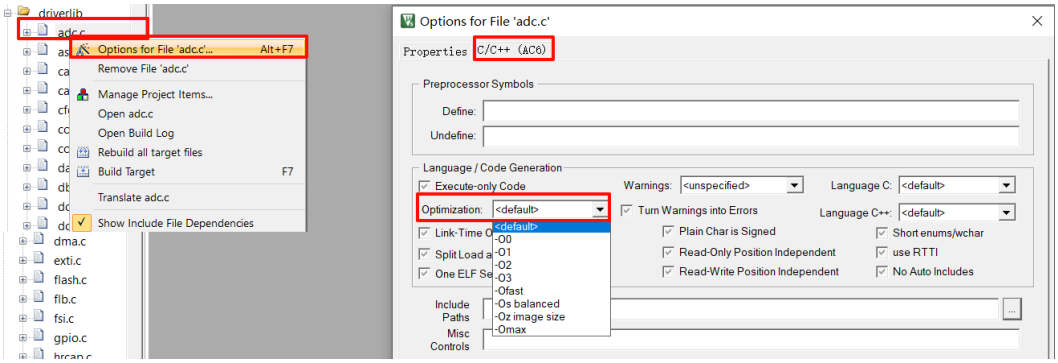
1. Global optimization level settings: In the "Options for Target" window, select the "C/C++" tab, and then select an appropriate optimization level from the "Optimization" drop-down menu.

Figure 10 Global optimization level



2. Single-file optimization level settings: Right-click a specific source file, select "Options for File...", and then set the optimization level in the "C/C++" tab.

Figure 11 Per-file optimization level

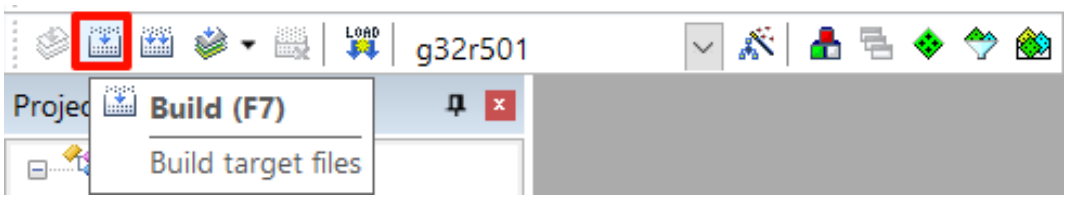


3. Single-function optimization level settings: Specific compiler instructions can be used for optimization setting before the function, e.g. using __attribute__((optnone)) to declare no optimization or use of __attribute__((optimize("O2"))) for optimization.

4.11 Program compilation

1. In the menu bar, click "Project" → "Build Target" (or directly click the "Build" button on the toolbar, or press "F7").

Figure 12 Build the project

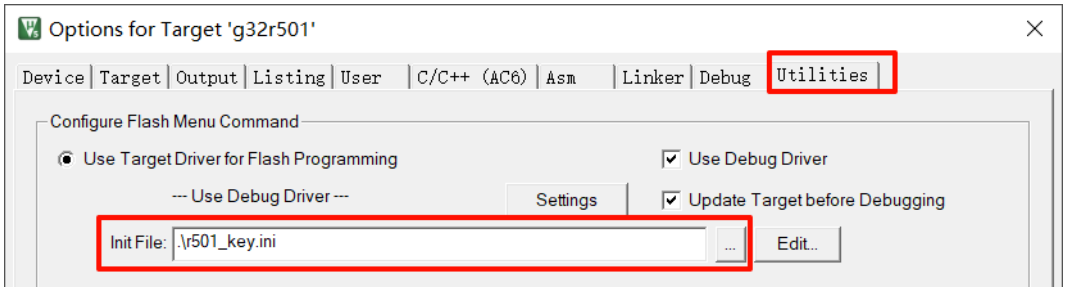


2. After the compilation process is completed, check for any errors or warning messages in the output window. If there are errors, make corresponding modifications according to the prompt.

4.12 Program download

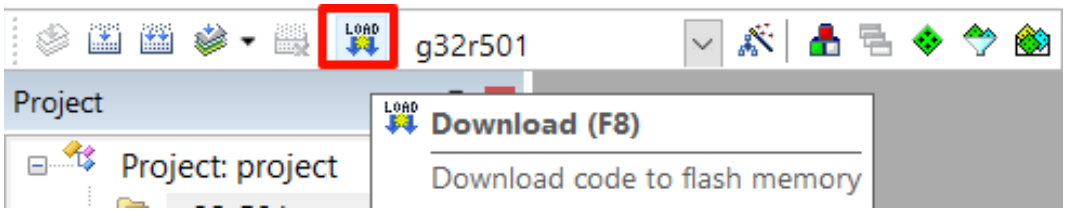
1. Click "Project" → "Options" in the menu bar.
2. In the pop-up dialog box, select the "Debug" tab.
3. In the "Use" drop-down menu, select the appropriate debugging probe (for example Geehy-Link or J-Link; see the J-Link download and Devices registration subsections later in this chapter).
4. Open **Options for Target**; in the "Utilities" tab, select the appropriate downloading tool.
5. Click "Settings" to load the specific download script file (when the probe is Geehy-Link).

Figure 13 Select the download algorithm / flash programming file



6. On the toolbar, click the "Download" button (small arrow icon) or select "Flash" → "Download" from the menu (or press "F8" directly).

Figure 14 Download the program



7. Ensure that the target device is connected, wait for the download to be completed, and

view the output window to confirm the download status.

4.13 Program Debug

4.13.1 J-Link Debug

4.13.1.1 Install J-Link device support (required first)

SEGGER J-Link does not recognize G32R501 / G32R502 by default. Before debugging with J-Link, install the device database, connect script, and Flash algorithm from the SDK into the local JLinkDevices directory. After installation, J-Link tools can select the matching device name; you usually **do not** need to copy a ".jlinkscript" into each project.

Software and hardware

- SEGGER J-Link software must support Cortex-M52 (reference: J-Link V7.94g or later; newer recommended).
- Prefer J-Link probe V12 or later.

Files to prepare

- G32R501: from "device_support/g32r501/common/Jlink/" take "G32R501.xml" and "Geehy_G32R501_ConnectCore.jlinkscript"; from SDK "package/FLM/" take "G32R5xx_Program_Algorithm.FLM" (for OTP debug also take "G32R5xx_OTP.FLM").
- G32R502: from "device_support/g32r502/common/jlink/" take "G32R502.xml" and "Geehy_G32R502_ConnectCore.jlinkscript"; from SDK "package/FLM/" take "G32R502_Program_Algorithm.FLM" (for OTP debug also take "G32R502_OTP.FLM").

The FLM files in SDK "package/FLM/" take precedence; if the DFP pack contains same-named algorithm files, their content should match the SDK copies. Copy the three files (xml, jlinkscript, main Flash FLM) into the same J-Link Devices install directory.

Windows installation

1. In File Explorer address bar, enter "%APPDATA%\SEGGER\JLinkDevices\" and press Enter (create if missing).
2. Create "Geehy\G32R501\" and/or "Geehy\G32R502\".
3. Copy the three files for that device into the folder. Examples:

- G32R501: "C:\Users\<your_user>\AppData\Roaming\SEGGER\JLinkDevices\Geehy\G32R501\"
- G32R502: "C:\Users\<your_user>\AppData\Roaming\SEGGER\JLinkDevices\Geehy\G32R502\"

4. Close and reopen all J-Link related programs (J-Flash, Ozone, IDE debug sessions, JLink.exe, etc.) so the device database reloads.

Linux / macOS

Copy the same three files to:

- G32R501: “$HOME/.config/SEGGER/JLinkDevices/Geehy/G32R501/”
- G32R502: “$HOME/.config/SEGGER/JLinkDevices/Geehy/G32R502/”

Verify after installation

1. Open J-Link Commander (“JLink.exe”) or J-Flash.
2. Enter “device ?” and review the device list.
3. Confirm the vendor Geehy lists the expected part names, for example:

- G32R501 single-core examples: “G32R501VY” / “MY” / “RY” / “NY” / “RC”
- G32R501 dual-core examples: “G32R501DVY” / “DMY” / “DRY” / “DNY”
- G32R502: “G32R502MC” / “KC” / “CC” / “RC”

4. Connect the target board and run an erase or program once to confirm DCS unlock and Flash programming.

Note: If Zone1 / Zone2 CSM passwords in OTP were changed from factory defaults, edit “Z1_CSM_Buff” / “Z2_CSM_Buff” at the top of the corresponding “Geehy_G32R501_ConnectCore.jlinkscript” or “Geehy_G32R502_ConnectCore.jlinkscript” **before** copying into JLinkDevices.

4.13.1.2 Configure J-Link in Keil MDK

After the previous section and restarting tools:

1. Open **Options for Target**.
2. In **Device**, select the G32R501 / G32R502 part that matches the board (must match installed J-Link Devices names, for example G32R501DVY or G32R502MC).
3. On the **Debug** tab, select **J-Link / Cortex-M**.
4. Ensure the MDK project folder has the matching initialization file and set **Debug → Initialization File**:

- G32R501: “r501_dbg.ini”
- G32R502: “r502_dbg.ini”

The ini handles Keil-side MSP/PC/DCS init and is independent of the JLinkDevices connect script; both must be correct.

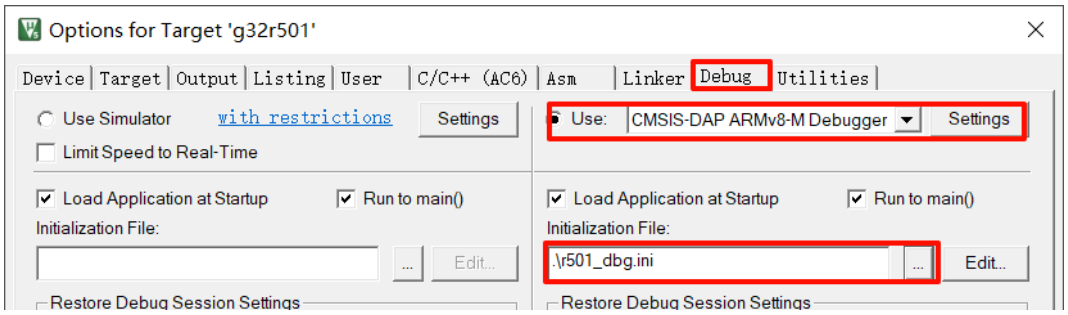
1. After the local device pack is installed, projects usually **do not** need a separate ".jlinkscript"; "G32R501.xml" / "G32R502.xml" bind the ConnectCore script automatically.
2. When using J-Link Debug, clear any leftover download Init File under **Utilities** if old settings remain.
3. Build, download, then click **Debug** to start a session (breakpoints, variables, single-step, etc.).

Note: Copying "JLinkSettings.JLinkScript" into each project is obsolete; connect scripts are supplied by the local JLinkDevices pack.

4.13.2 GEEHY LINK Debug

1. Open the project options: Select "Options for Target".
2. In the "Debug" tab, choose the appropriate Debugger.
3. In the "Settings" menu, click the "Load" button to load a specific Debug initialization script file.

Figure 15 Select CMSIS-DAP and the Debug initialization script



4. After downloading, click the "Debug" button to start debugging the program; in debugging mode, you can set breakpoints, view variables, and single-step execution.

4.14 C Language Compatibility

When migrating the tool chains, the compatibility of C language code needs special attention. Different compilers may have differences in file format support, built-in functions, memory operation, and data type. To ensure that the code can be compiled and run correctly in a new compilation environment, necessary adjustments and optimization shall be made on the existing code.

4.14.1 File format support

Support .c/.h files.

4.14.2 Floating-point constants

Per ISO/IEC C, an unsuffixed floating constant has type “double”; suffix “f” or “F” gives “float”; suffix “l” or “L” gives “long double”. Append “f” to constants that must be single precision.

4.14.3 Use of sizeof

Different compilers and architectures may have differences in the size of data type, so special attention shall be paid to the results of the sizeof operator on different platforms. A size comparison of common data types on G32R501 and Txx320F28004x is provided in Table 3.

Table 3 Size Comparison of Different Data Types on G32R501 and Txx320F28004x

Data type	G32R501	Txx320F28004x
char	1	1
short	2	1
int	4	1
long	4	2
long long	8	4
float	4	2
double	8	4
long double	8	4
void	4	2
int8_t	1	1
uint8_t	1	1
int16_t	2	1
uint16_t	2	1
int32_t	4	2
uint32_t	4	2
int64_t	8	4
uint64_t	8	4

4.15 Assembly Compatibility

There may be significant differences in instruction set and assembly syntax among different target platforms, so the existing assembly code needs to be rewritten and adapted to ensure correct operation on the new platforms.

4.15.1 File format support

AC6 is based on LLVM and Clang technology, mainly using GNU-style assembly syntax. Assembly file formats supported by AC6:

.s file: GNU-style assembly file format.

Meanwhile, AC6 supports the use of inline assembly in C functions.

4.15.2 Assembly code format requirements

Single assembly file: Here is a simple assembly code example, which defines an assembly function add, used to add two integers. The content of the file “add.s” is as follows:

```
.syntax unified
.global add
.type add, %function
add:
@ Function entry
@ Parameters: r0 and r1
@ Return value: r0
adds r0, r0, r1   @ Add r0 and r1, store result in r0
bx lr             @ Return to calling function
.end
```

Use inline assembly in C function: The following is an example of using inline assembly in C function, and an inline assembly function add_inline is defined to add two integers:

```
// Inline assembly function
static inline int add_inline(int a, int b) {
    int result;
    __asm volatile (
        "adds %0, %1, %2\n"
        : "=r" (result)        // Output operand
        : "r" (a), "r" (b)     // Input operands
        : "cc"                 // Clobbered registers
    );
    return result;
}
```

4.16 Linker Script Files

The linker script files are used to define the memory layout and section allocation of the

program. In the migration process, it is necessary to use linker script files in the corresponding format according to different target platforms and development environments. The G32R5 series MCU uses linker script files in the ".sct" format in the MDK development environment, which comply with Arm Company's specifications.

Differences in linker script files

File format:

- The G32R5 series MCU uses linker script files in the ".sct" format, which comply with Arm Company's specifications.
- Txx320F28004x uses the linker script files in ".CMD" format, which comply with the company's specifications.

Memory layout and section allocation:

- The ".sct" file of the G32R5 series MCU arranges memory allocation by defining the load region and execution region.
- The ".CMD" file of Txx320F28004x arranges memory allocation by defining the Memory section (MEMORY) and section allocation (SECTIONS).

4.17 RAM Operation

The Txx320F28004x compiler supports the Progma syntax, which tells the compiler that if a specific function, target file, or the attributes of a section of code are modified, such as CODE_SECTION, a Section is assigned to a certain function, and its usage is as follows:

```
# pragma CODE_SECTION(funcA, "codeA")
```

- G32R5 can implement the same function using the following statement:

```
__attribute__((section("xxx")))
```

Where, xxx represents the SECTION name which a certain function or global variable is assigned. During use, users can refer to the following format:

```
__attribute__((section("itcm.ramfunc"))) void SysCtl_delay(uint32_t count){}
```

You only need to specify the attributes when defining functions and variables.

Note: When using "__attribute__((section("itcm.ramfunc")))", a declaration of the "itcm.ramfunc" field in the .sct (linker script) is required: ".ANY (itcm.ramfunc)"

5 IAR Embedded Workbench for Arm development tool chain

5.1 Debugger Support

Please refer to Chapter 4.1.

5.2 IDE Version

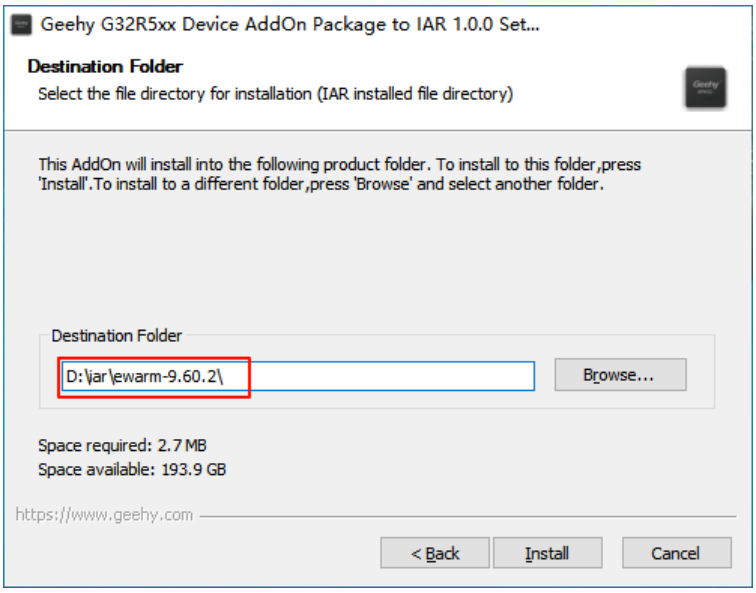
- G32R501: use IAR Embedded Workbench for Arm **9.60.2 or later**.
- **G32R502 series: IAR Embedded Workbench for Arm 9.70.4 or later is required**; versions below that do not meet development requirements for this series.

5.3 Install the Chip Support

Before officially using IAR Embedded Workbench for Arm to develop G32R5 series MCU, please install the chip support package first. The path for chip support is: ..\utilities\G32R5xx_AddOn\G32R5xx_AddOn_vx.x.x.exe

Open G32R5xx_AddOn_vx.x.x.exe with the administrator rights and go to the interface for selecting the path to install the chip support. This path is the installation path for IAR Embedded Workbench for Arm, e.g. the example: D:\iar\ewarm-9.60.2\.

Figure 16 Install chip support with G32R5xxAddOnvx.x.x.exe

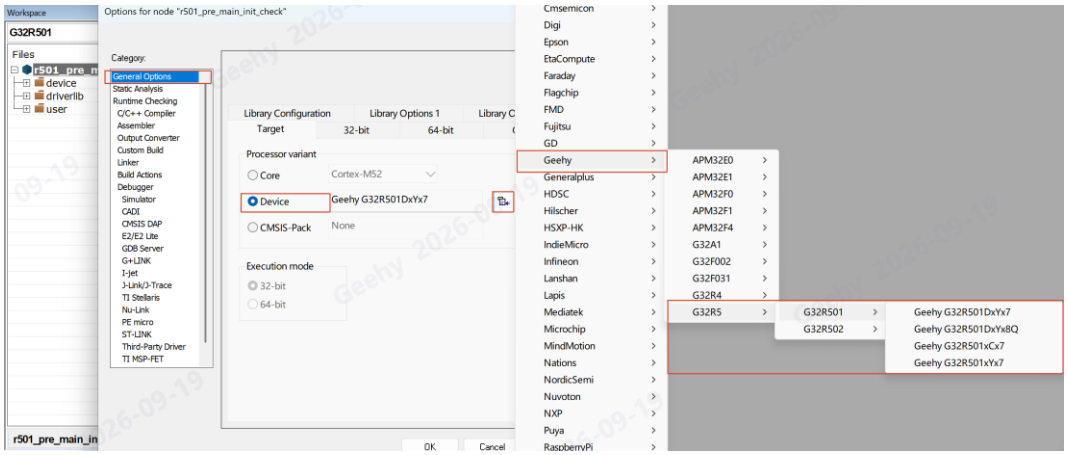


If the software cannot detect the IAR Embedded Workbench for Arm installation path on the computer, manually select the **installation root directory** (not a subfolder such as “config” under the install tree).

After selecting the correct path and adding the chip support, open IAR Embedded Workbench for Arm, select "New project", and in the chip selection tab, the G32R5 series MCU list can be

seen.

Figure 17 G32R5 series MCU list



5.4 Project Operation

5.5 Open the example project

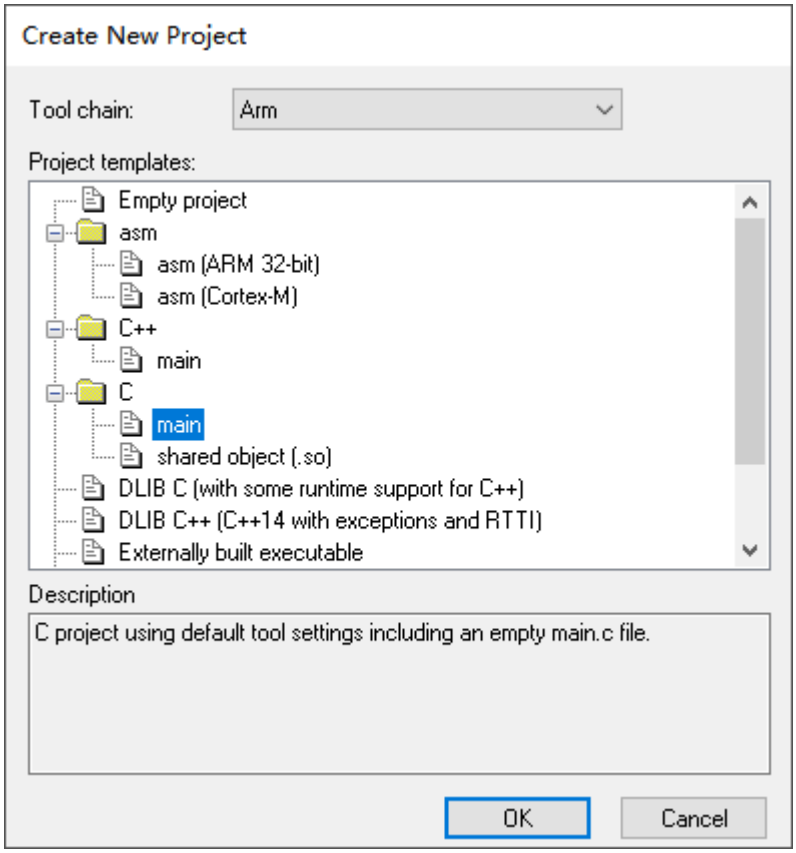
1. Start IAR Embedded Workbench for Arm 9.60.2.
2. Click "File" → "Open Workspace..." in the menu bar.
3. Browse the path of the SDK project file you provide, select the corresponding .eww file, and then click "Open".
4. Or, after installing IAR Embedded Workbench for Arm 9.60.2, directly click the project file ".eww" to open it.

Note: Please complete the above steps in Section 5.3 Install Chip Support; otherwise, the IAR Embedded Workbench for Arm will prompt that the chip cannot be found.

5.6 Project establishment

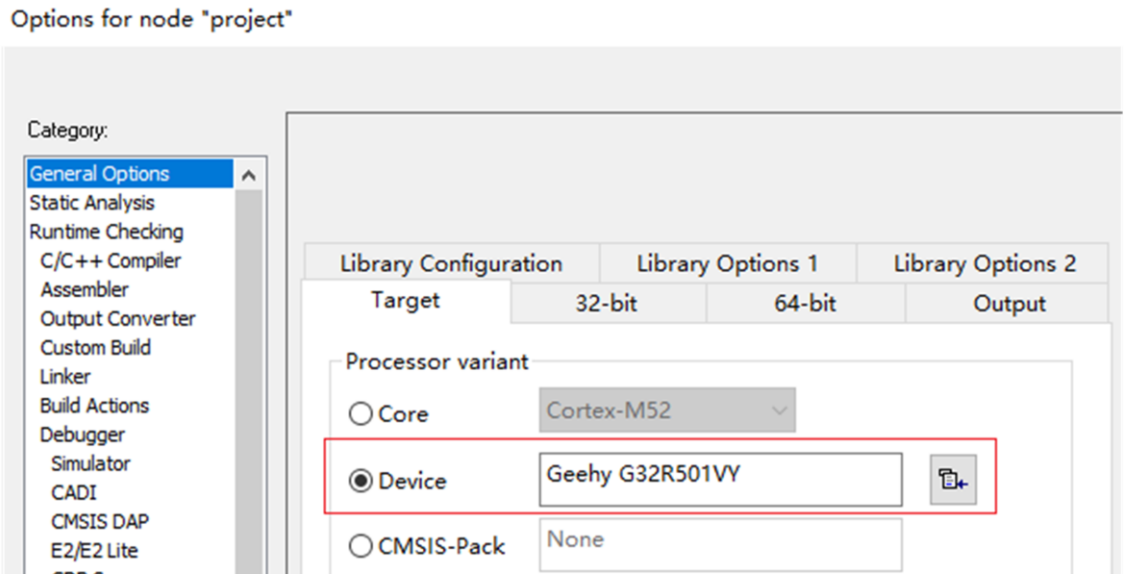
1. Start IAR Embedded Workbench for Arm 9.60.2.
2. Click "Project" → "Create New Project" in the menu bar.
3. Select "Tool chain" as "Arm".
4. Select the "main" under "C" and then click "OK".
5. Select the path to save the project, enter the project name, and click "Save".

Figure 18 Create New Project



6. Right-click the project name and select "Options...".
7. Select "Device" under "Target" in "General Options".
8. Select the appropriate G32R5xx series MCU model, and click "OK".

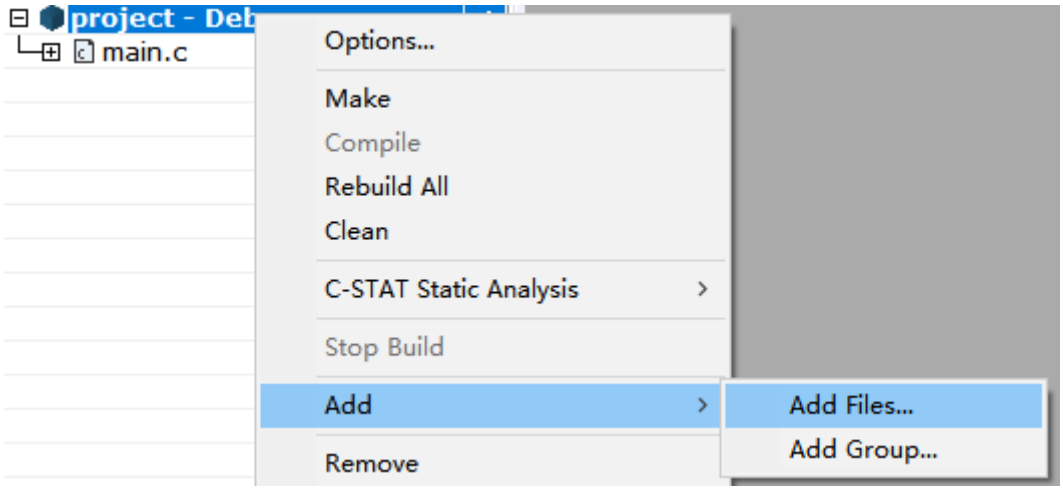
Figure 19 Select the microcontroller



5.7 File import

1. Make sure your new project is already open in the Project window.
2. Right-click the project name, select "Add", and then select "Add Files".

Figure 20 Add source files

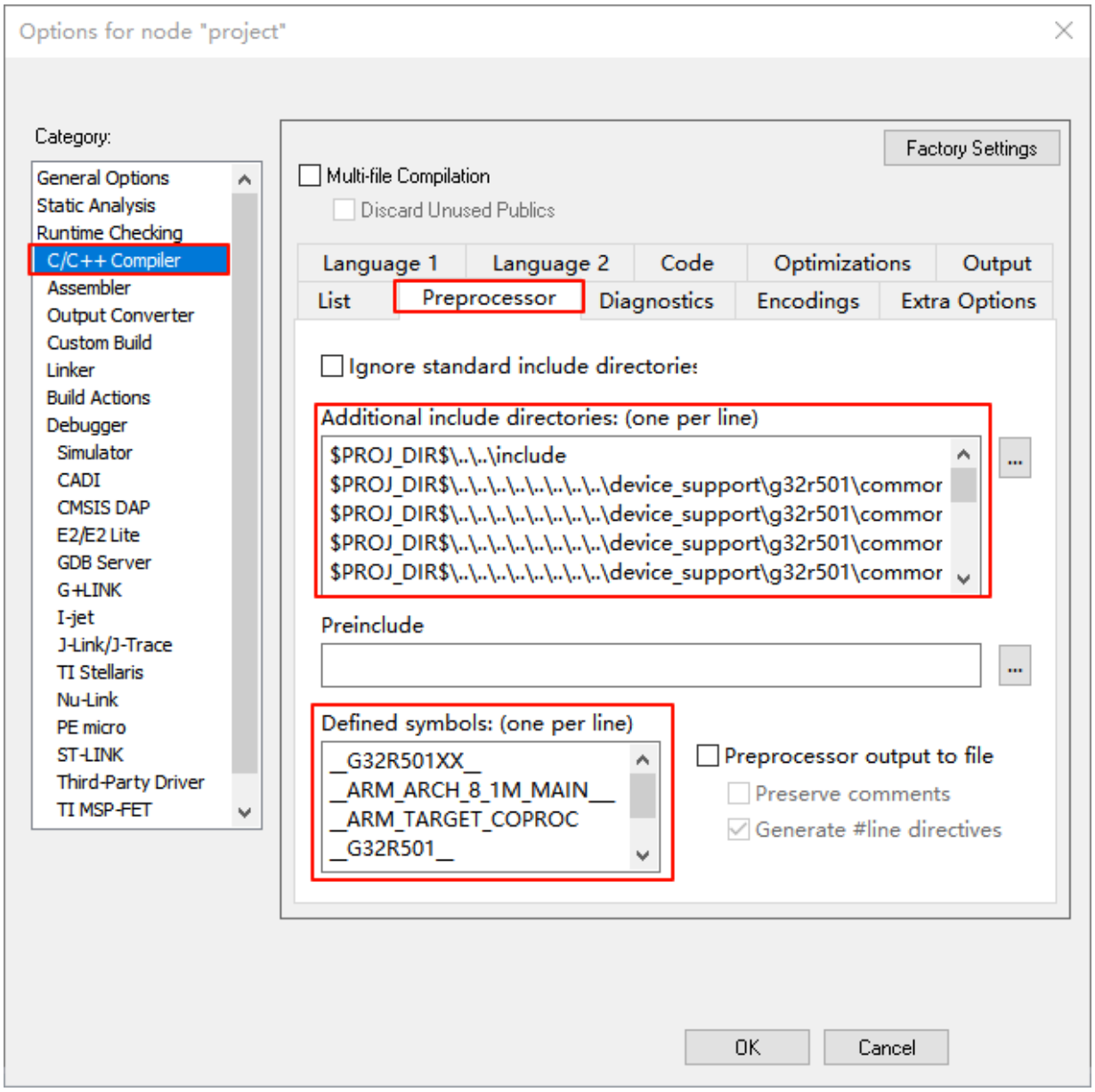


3. In the pop-up file browser, find and select the source files to be imported (e.g. .c and .h files), and then click "Add".
4. If a new source file is needed, click "File" on the toolbar and select "New File"; press "Ctrl+S" to save, enter the file name and type, then add as above.

5.8 Configure header file path and macro definition

1. Right-click the project name and select "Options...".
2. Select “Preprocessor” under the “C/C++ Compiler” tab.

Figure 21 Configure include paths and macros



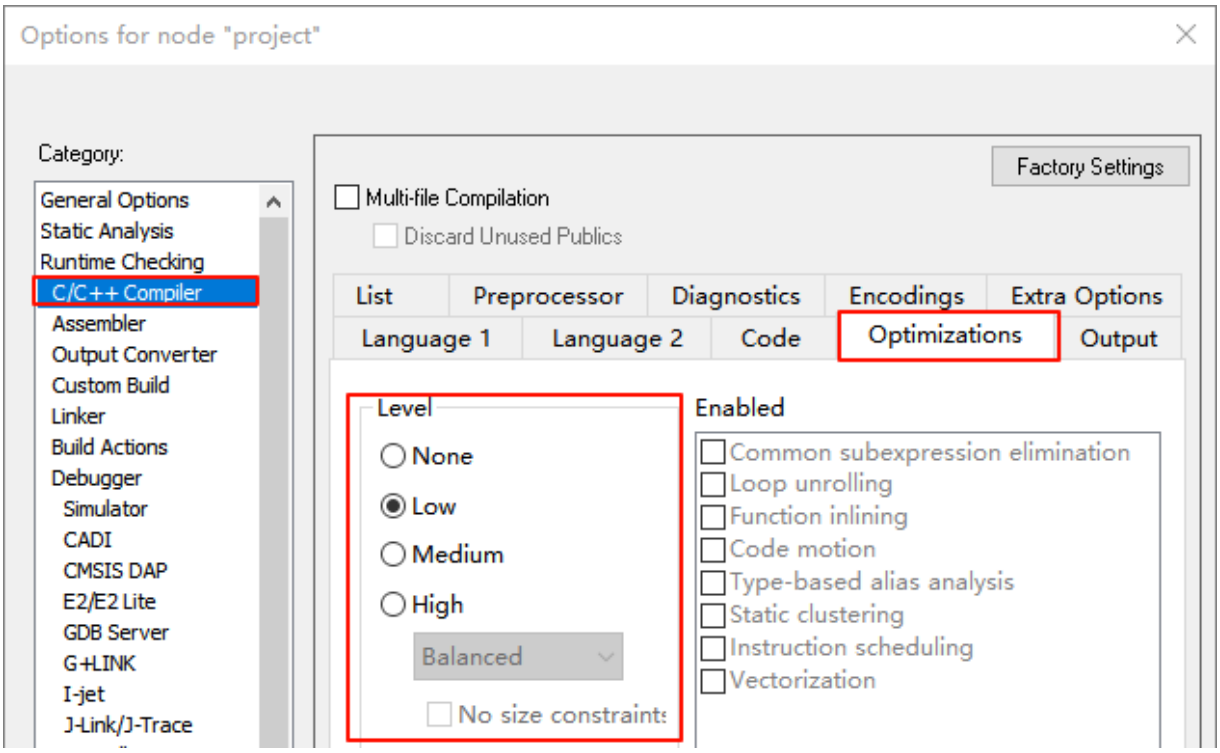
3. Add all necessary library file paths in “Additional include directories: (one per line)”.
4. In Defined symbols: (one per line), add the required macro definitions.

5.9 Compilation optimization level settings

IAR Embedded Workbench for Arm provides multiple optimization level settings, which can be adjusted at the global, single-file, and single-function levels:

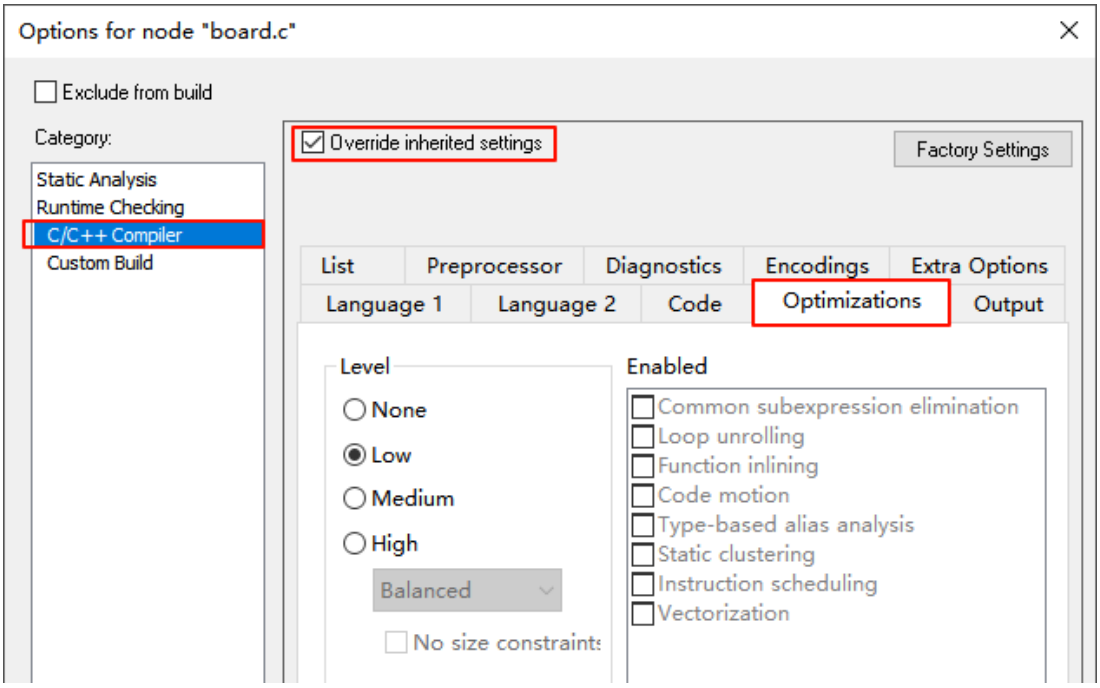
1. Global optimization level settings: Open the "C/C++ Compiler" tab and then select the appropriate optimization level in "Optimizations".

Figure 22 Global optimization level



2. Single-file optimization level settings: Right-click a specific source file, select "Options...", check "Override inherited settings", and then select the appropriate optimization level in "Optimizations".

Figure 23 Per-file optimization level

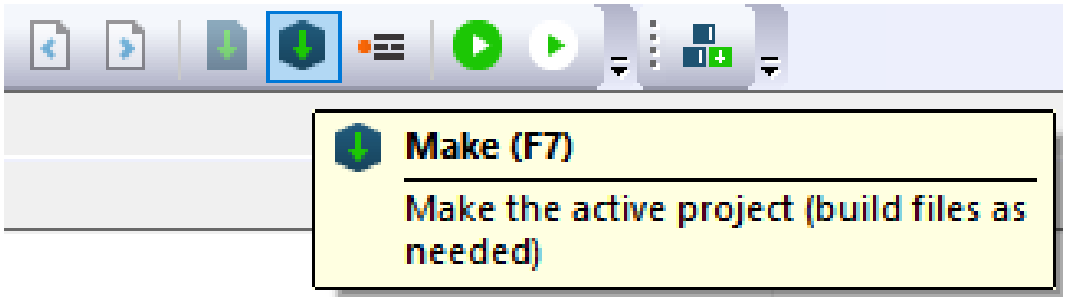


3. Single-function optimization level settings: Specific compiler instructions can be used before the function to set the optimization level, e.g. declaring no optimization using “_Pragma("optimize=none")”.

5.10 Program compilation

1. In the menu bar, click "Project" → "Make" (or directly click the "Make" button on the toolbar, or press "F7").

Figure 24 Build the project



2. After the compilation process is completed, check for any errors or warning messages in the output window. If there are errors, make corresponding modifications according to the prompt.

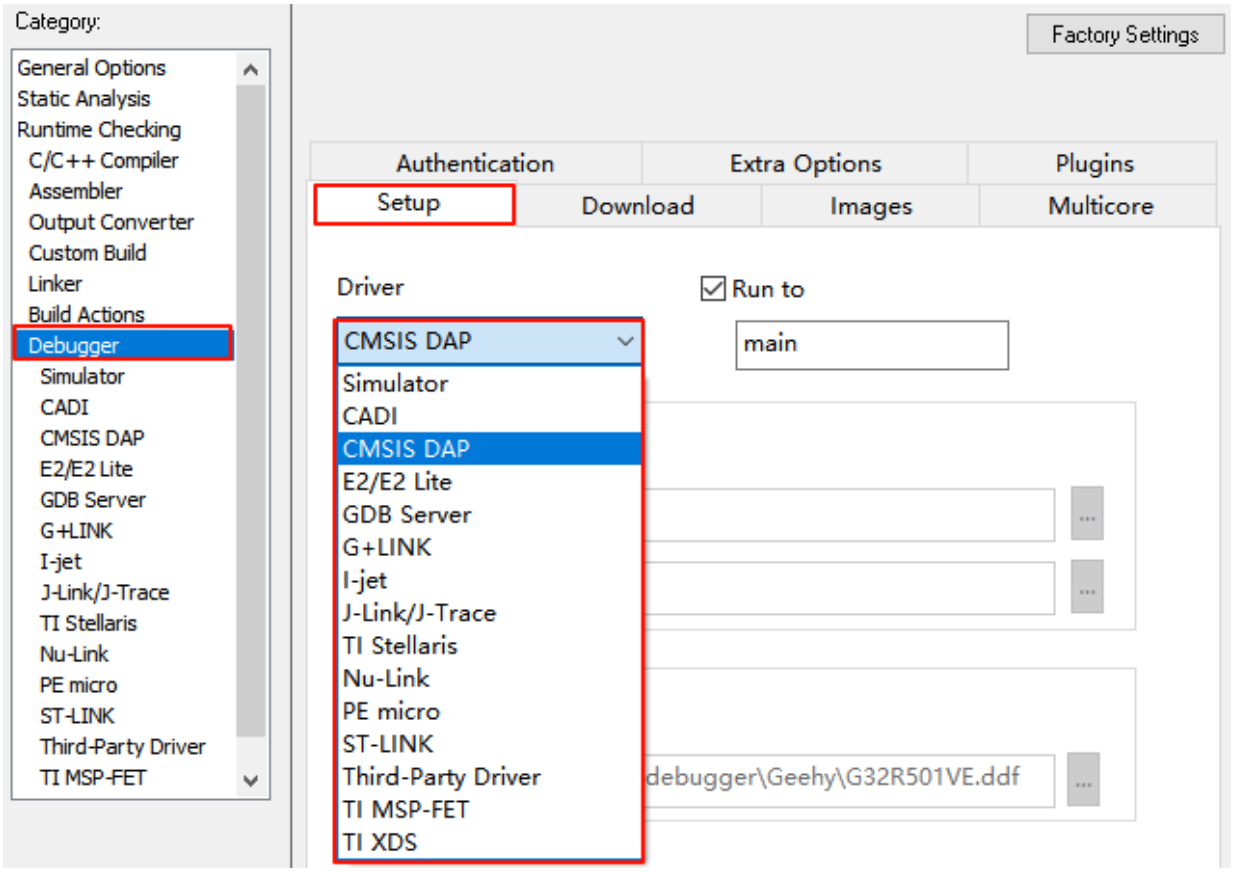
5.11 Program Debug (download)

1. Right-click the project name and select "Options...".

2. In the pop-up dialog box, select the "Debugger" tab.

3. In the "Driver" drop-down menu under the "Setup" tab, select the appropriate Debugger (e.g. Geehy-Link (CMSIS DAP), J-Link, etc.).

Figure 25 Select the Debugger



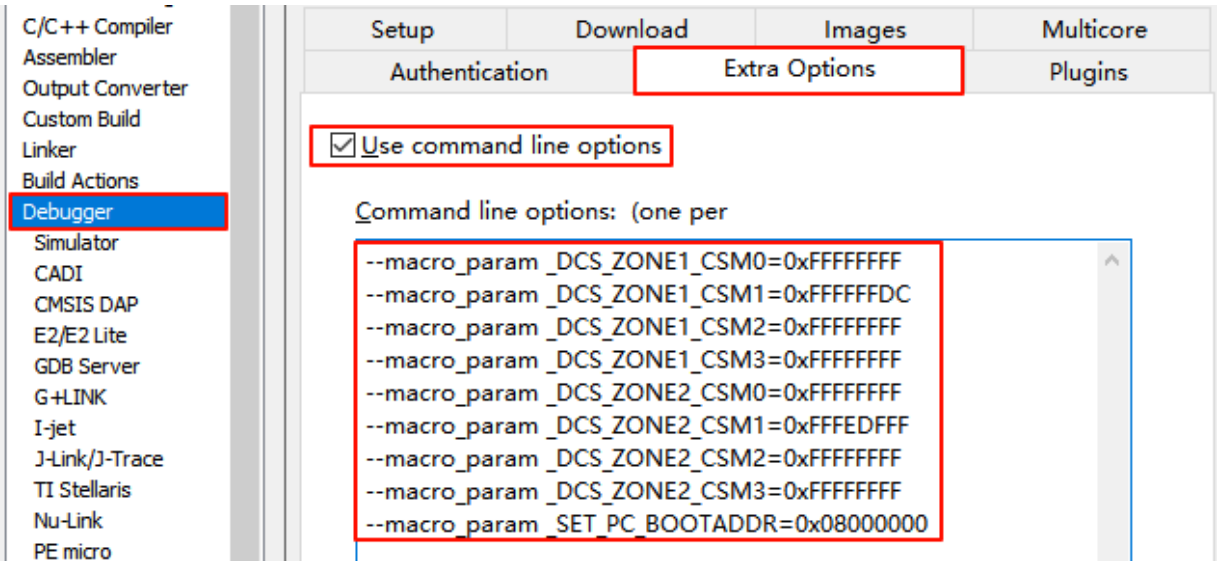
4. Right-click the project name again, select "Options...", and open the "Debugger" tab.

5. Check "Use command line options" under the "Extra Options" tab, and add the DCS key and boot-address options to "Command line options", for example:

```
--cdecp=0
--macro_param _DCS_ZONE1_CSM0=0xFFFFFFFF
--macro_param _DCS_ZONE1_CSM1=0xFFFFFFDC
--macro_param _DCS_ZONE1_CSM2=0xFFFFFFFF
--macro_param _DCS_ZONE1_CSM3=0xFFFFFFFF
--macro_param _DCS_ZONE2_CSM0=0xFFFFFFFF
--macro_param _DCS_ZONE2_CSM1=0xFFFEDFFF
--macro_param _DCS_ZONE2_CSM2=0xFFFFFFFF
--macro_param _DCS_ZONE2_CSM3=0xFFFFFFFF
--macro_param _SET_PC_BOOTADDR=0x08000000
```

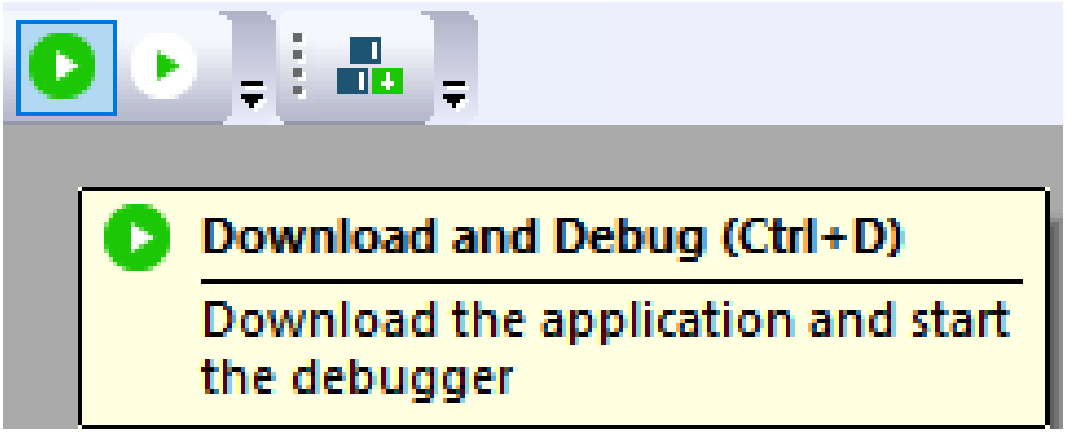
- “--cdecp=0”: enables CDE support
- “--macro_param _DCS_ZONEx_CSMx”: DCS key support; not required when the default key is used
- “--macro_param _SET_PC_BOOTADDR”: sets the boot address; not required when the default is 0x08000000

Figure 26 Configure Debug command-line options



6. On the toolbar, click "Project" and select "Download"; use "Erase memory", "Download active application", or "Download file..." as needed.
7. Click the "Download and Debug" button on the toolbar or press "Ctrl+D" to start debugging.

Figure 27 IAR Download and Debug



8. In debugging mode, you can set breakpoints, view variables, and single-step execution.

Problems in Debugging

During debugging, the Flash content in the Memory window does not update.

Reason: The AccType of the Flash corresponding Memory region in the ddf (device description file) is set to "R", which means the Debugger has read-only access to the Flash and cannot modify its content. Therefore, the Debugger will not update the values of the corresponding Memory region during the debugging process.

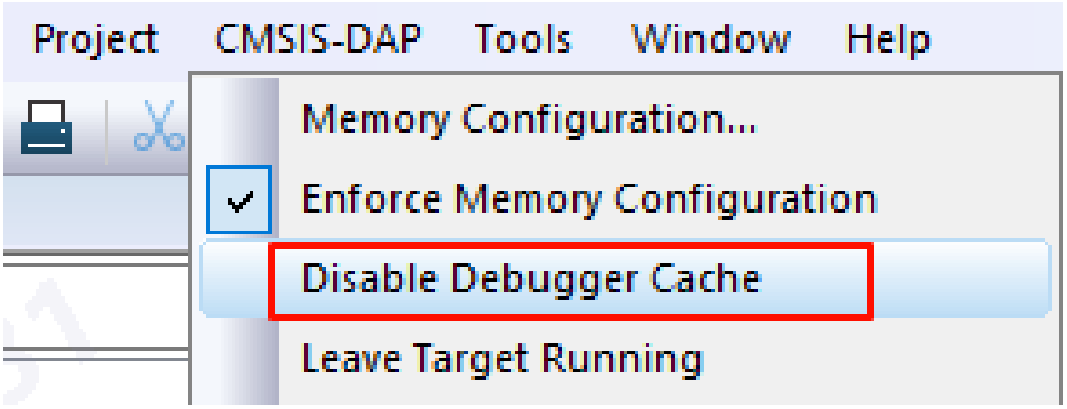
Figure 28 .ddf file (excerpt)

```
[Memory]
;;          Name                 AdrSpace    StartAdr     EndAdr       AccType   Width
Memory =   Boot_ROM             Memory      0x10000000   0x1001FFFF   R
Memory =   Secure_ROM           Memory      0x10020000   0x1002FFFF   R
Memory =   ITCM_Flash           Memory      0x00100000   0x0019FFFF   R
Memory =   BusMatrix_Flash      Memory      0x08000000   0x0809FFFF   R
Memory =   CPU0_ITCM            Memory      0x00000000   0x0000BFFF   RW
Memory =   CPU1_ITCM            Memory      0x00000000   0x00001FFF   RW
Memory =   CPU0_DTCM            Memory      0x20000000   0x20003FFF   RW
```

Solution:

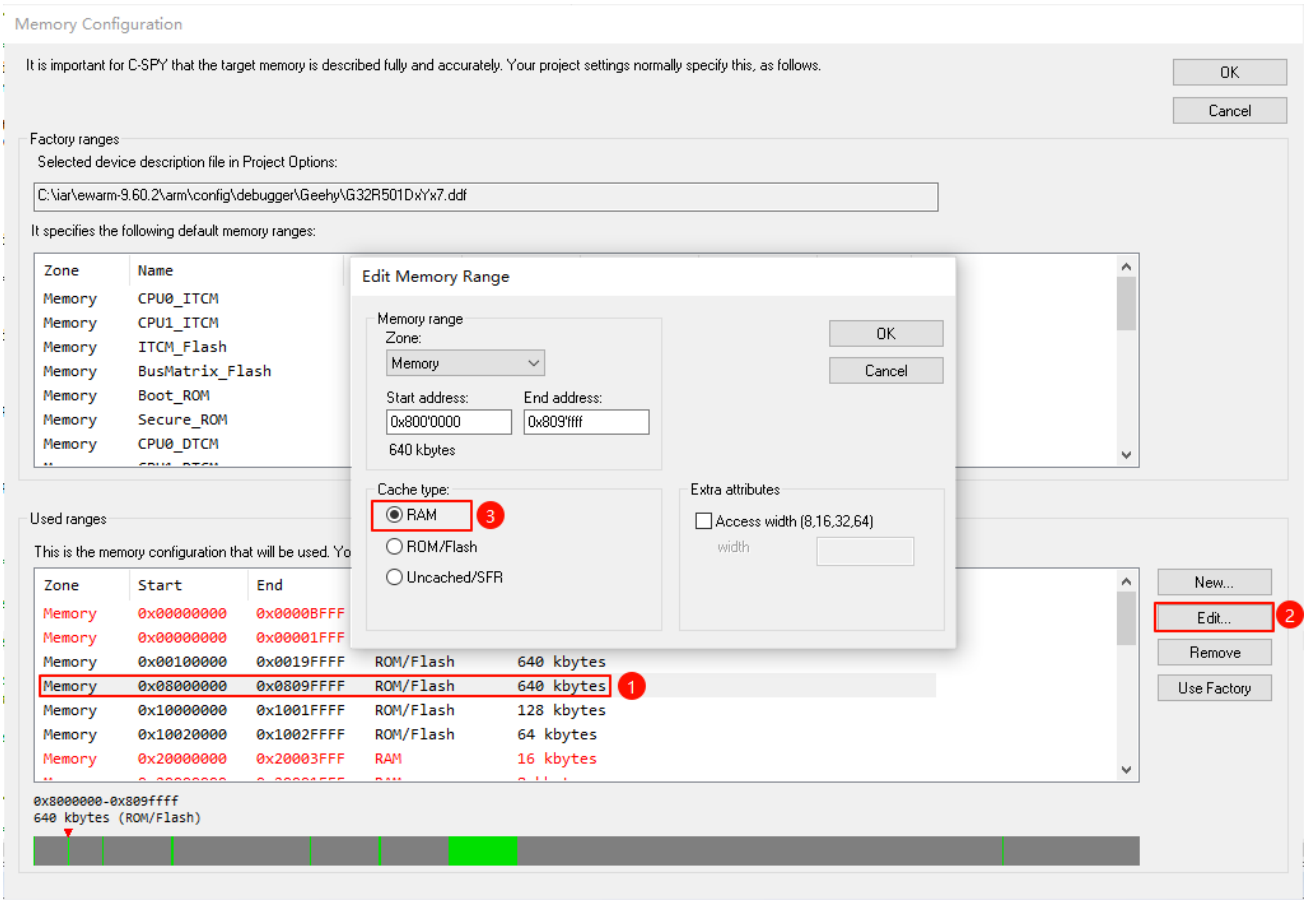
- Enable “Disable Debugger Cache”

Figure 29 Disable Debugger Cache



- Change the Cache Type of the Flash corresponding Memory region to RAM. Click on Memory Configuration to enter the Memory Configuration window and modify the Cache Type.

Figure 30 Modify Cache Type



- Modify the AccType of the Flash corresponding Memory region in the ddf file to RW.

Figure 31 Modify .ddf AccType

```
[Memory]
;;          Name                AdrSpace    StartAdr     EndAdr       AccType   Width
Memory =   Boot_ROM            Memory      0x10000000   0x1001FFFF   R
Memory =   Secure_ROM          Memory      0x10020000   0x1002FFFF   R
Memory =   ITCM_Flash          Memory      0x00100000   0x0019FFFF   R
Memory =   BusMatrix_Flash     Memory      0x08000000   0x0809FFFF   RW
Memory =   CPU0_ITCM           Memory      0x00000000   0x0000BFFF   RW
Memory =   CPU1_ITCM           Memory      0x00000000   0x00001FFF   RW
Memory =   CPU0_DTCM           Memory      0x20000000   0x20003FFF   RW
Memory =   CPU1_DTCM           Memory      0x20000000   0x20001FFF   RW
```

5.11.1 J-Link Debug

5.11.1.1 Install J-Link device support (required first)

SEGGER J-Link does not recognize G32R501 / G32R502 by default. Install the device pack before use; otherwise IAR cannot select the chip correctly.

Files to prepare

- G32R501: from “device_support/g32r501/common/Jlink/” take “G32R501.xml” and “Geehy_G32R501_ConnectCore.jlinkscript”; from SDK “package/FLM/” take “G32R5xx_Program_Algorithm.FLM” (for OTP debug also take “G32R5xx_OTP.FLM”).
- G32R502: from “device_support/g32r502/common/jlink/” take “G32R502.xml” and “Geehy_G32R502_ConnectCore.jlinkscript”; from SDK “package/FLM/” take “G32R502_Program_Algorithm.FLM” (for OTP debug also take “G32R502_OTP.FLM”).

The FLM files in SDK “package/FLM/” take precedence; if the DFP pack contains same-named algorithm files, their content should match the SDK copies.

Windows

1. In File Explorer, enter “%APPDATA%\SEGGER\JLinkDevices\” and press Enter (create the folder if missing).
2. Create “Geehy\G32R501\” or “Geehy\G32R502\” and copy the three files into it.
3. Example: “C:\Users\<your_user>\AppData\Roaming\SEGGER\JLinkDevices\Geehy\G32R501\”
4. Close and reopen IAR and other J-Link tools.

Linux / macOS

Copy to “$HOME/.config/SEGGER/JLinkDevices/Geehy/G32R501/” or “.../G32R502/”.

Verify

Open J-Link Commander, enter “device ?”, and confirm the Geehy vendor lists the expected part (for example “G32R501DVY”, “G32R502MC”).

If OTP keys were changed, edit “Z1_CSM_Buff” / “Z2_CSM_Buff” at the top of the ConnectCore script before copying to the directory above.

5.11.1.2 Configure J-Link in IAR

1. Right-click the project name and choose **Options…**.
2. Open the **Debugger** tab; under **Setup** → **Driver**, select **J-Link**.
3. Choose the G32R501 / G32R502 part that matches the board (must match installed J-Link Devices names).
4. When using the J-Link Flash algorithm above, **disable** IAR **Use flash loader(s)** to avoid conflict with the J-Link-side algorithm.
5. Confirm download and reset options, connect the target, download, and enter debug.

Note: DCS unlock on connect/reset is handled by the installed ConnectCore script; you usually do not copy a

standalone “.jlinkscript” into the project.

5.12 Assembly Compatibility

There may be significant differences in instruction set and assembly syntax among different target platforms, so the existing assembly code needs to be rewritten and adapted to ensure correct operation on the new platforms.

5.12.1 File format support

IAR Embedded Workbench uses the specific IAR assembly syntax. Assembly file formats supported:

.s file: IAR-style assembly file format.

Meanwhile, it supports the use of inline assembly in C functions.

5.12.2 Assembly code format requirements

Single assembly file: Here is a simple assembly code example, which defines an assembly function add, used to add two integers. The content of the file “add.s” is as follows:

```
SECTION .text:CODE   ; Define code section
PUBLIC add           ; Declare global symbol
add:
; Function entry
; Parameters: r0 and r1
; Return value: r0
ADD r0, r0, r1       ; Add r0 and r1, store result in r0
BX lr                ; Return to calling function
END
```

Use inline assembly in C function: The following is an example of using inline assembly in C function, and an inline assembly function add_inline is defined to add two integers:

```
// Inline assembly function
static inline int add_inline(int a, int b) {
int result;
__asm volatile (
"adds %0, %1, %2\n"
: "=r" (result)        // Output operand
: "r" (a), "r" (b)     // Input operands
: "cc"                 // Clobbered registers
);
return result;
```

```
}
```

5.13 Linker Script Files

The linker script files are used to define the memory layout and section allocation of the program. In the migration process, it is necessary to use linker script files in the corresponding format according to different target platforms and development environments. The G32R5 series MCU uses the linker script files in the ".icf" format in the IAR Embedded Workbench for Arm development environment, which comply with IAR Company's specifications.

Differences in linker script files

File format:

- The G32R5 series MCU uses linker script files in the ".icf" format, which comply with IAR Company's specifications.
- Txx320F28004x uses the linker script files in ".CMD" format, which comply with the company's specifications.

Memory layout and section allocation:

- The ".icf" files of the G32R5 series MCU allocate memory attributes by defining different "regions" and their attributes.
- The “.CMD” file of Txx320F28004x arranges memory allocation by defining the Memory section (MEMORY) and section allocation (SECTIONS).

5.14 RAM Operation

Please refer to Chapter 4 RAM Operation.

6 Eclipse

6.1 Debugger Support

- Geehy-Link (WinUSB), DAP Link (firmware version CMSIS-DAP V2 or above)
- J-Link V12 (J-Link V7.94g or above)

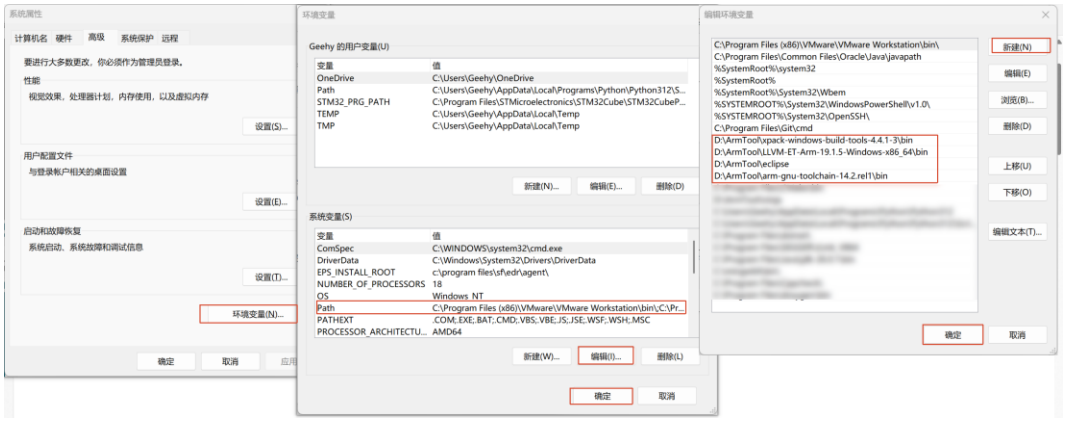
6.2 IDE Version

Ensure the use of Eclipse 4.35 or a newer version of the IDE.

6.3 LLVM_For_ARM_Toolchain

1. Download the LLVM_For_ARM_Toolchain compiler. From the LLVM_For_ARM_Toolchain repository, download the LLVM-ET-Arm-19.1.1-Windows-x86_64 compiler.
2. Extract the downloaded compressed package (example extraction path: D:\desktop\clang\utilities\LLVM-ET-Arm-19.1.1-Windows-x86_64).
3. Add the "bin" folder from the extracted directory to the system environment variable.

Figure 32 Add system environment variables



6.4 GDB Service

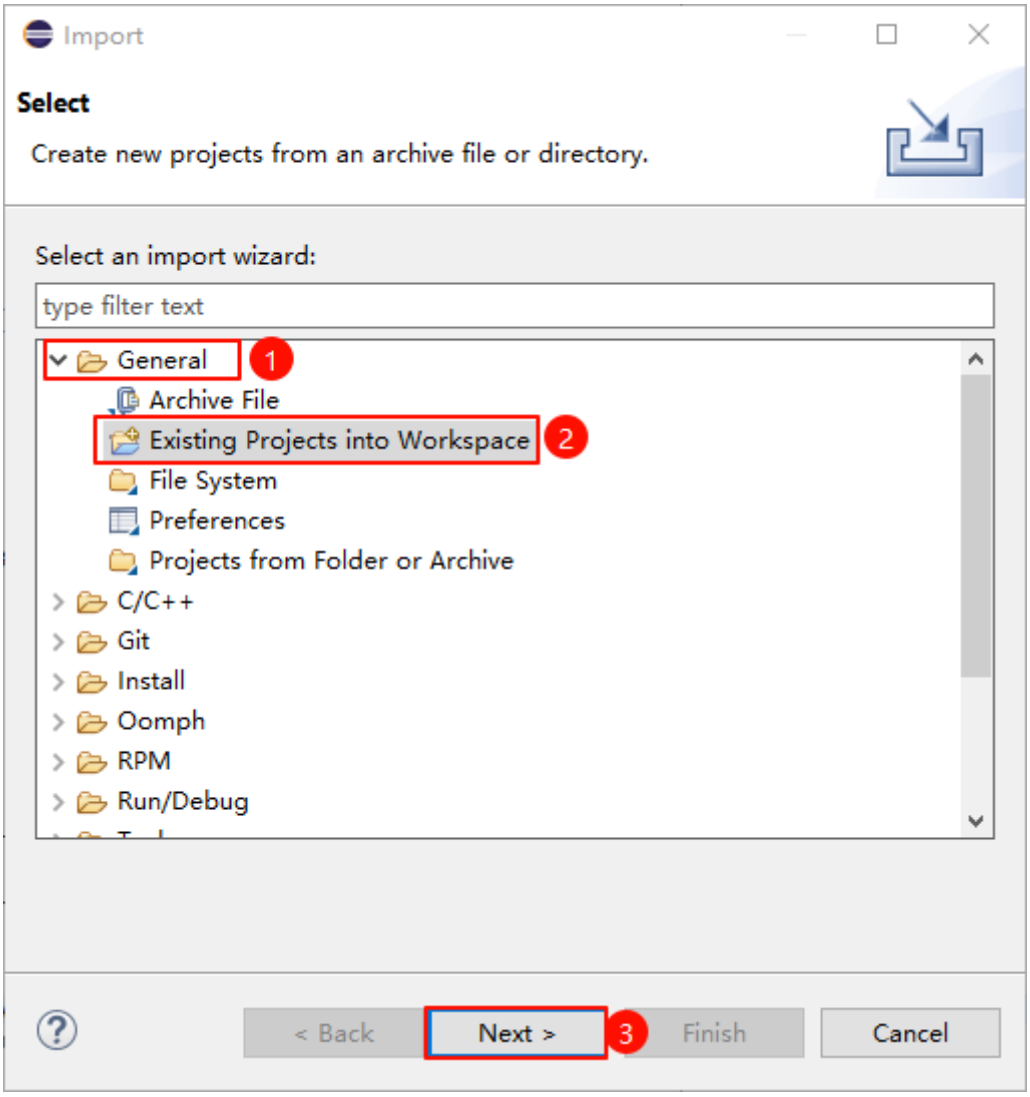
It is recommended to use the arm-none-eabi-gdb.exe provided by Arm. The version of arm-none-eabi-gdb.exe used in the example is 14.2.

6.5 Project Operations

6.6 Opening the Example Project

1. Run Eclipse 4.35.
2. Click "File" → "Import..." → "General" → "Existing Project into Workspace" in the menu bar.

Figure 33 Import project



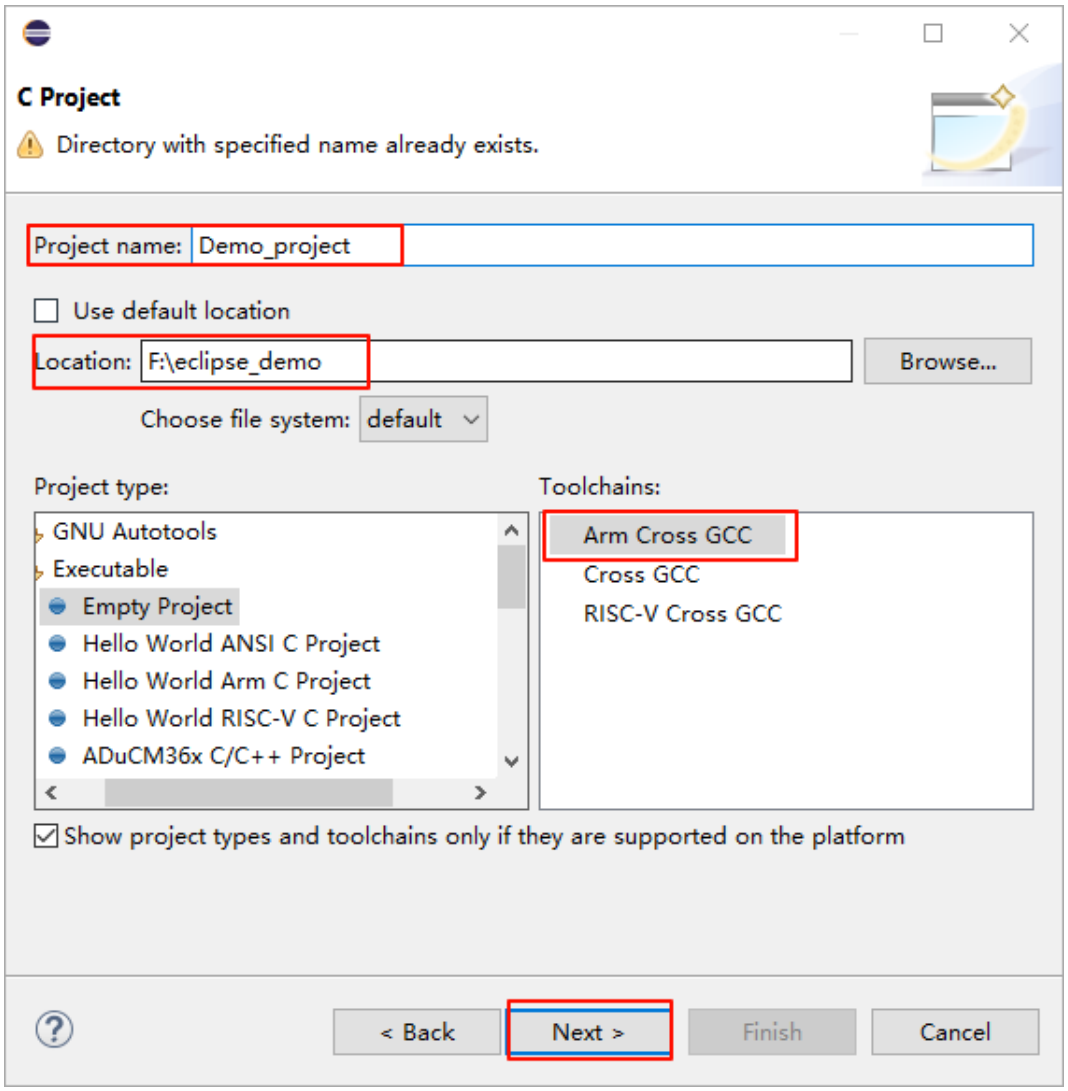
3. Click "Select root directory", navigate to the path of the SDK project file you provided, select the corresponding project folder, and then click "Finish".

Note: Please complete the above steps after finishing the LLVM compilation configuration in section 6.3.

6.7 Project Creation

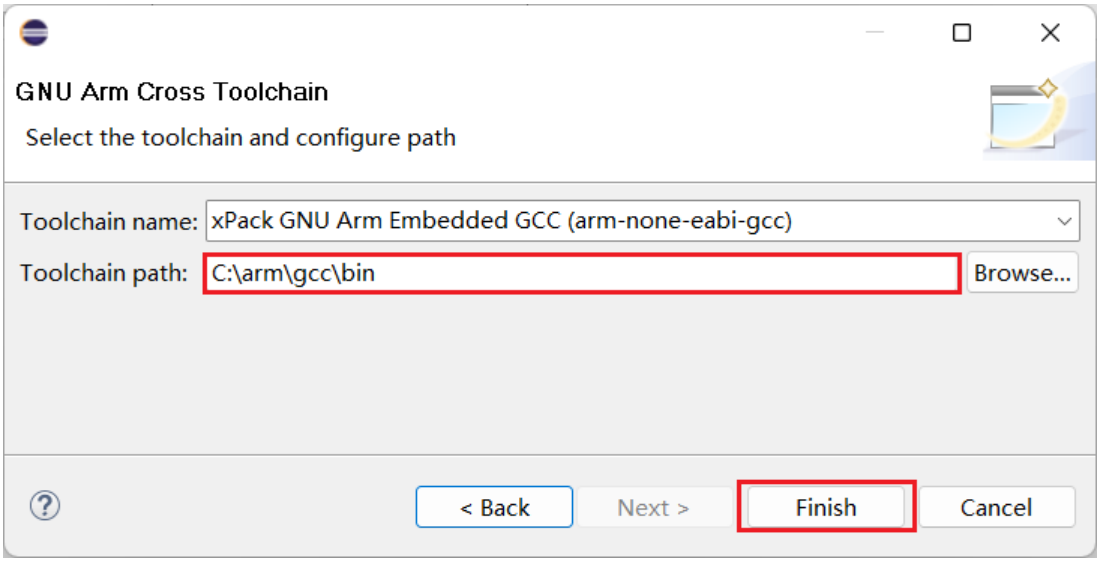
1. Open Eclipse 4.35.
2. Under "File" → "New", select to create a new C/C++ Project, and choose C Managed Build.
3. Enter the project name, configure the project type (recommended under the Project directory), and select Arm Cross GCC as the toolchain.

Figure 34 Configure the project



4. If Arm Toolchains are correctly configured, the path is selected automatically; otherwise click Browse for the absolute path.

Figure 35 Set Arm Toolchains Path

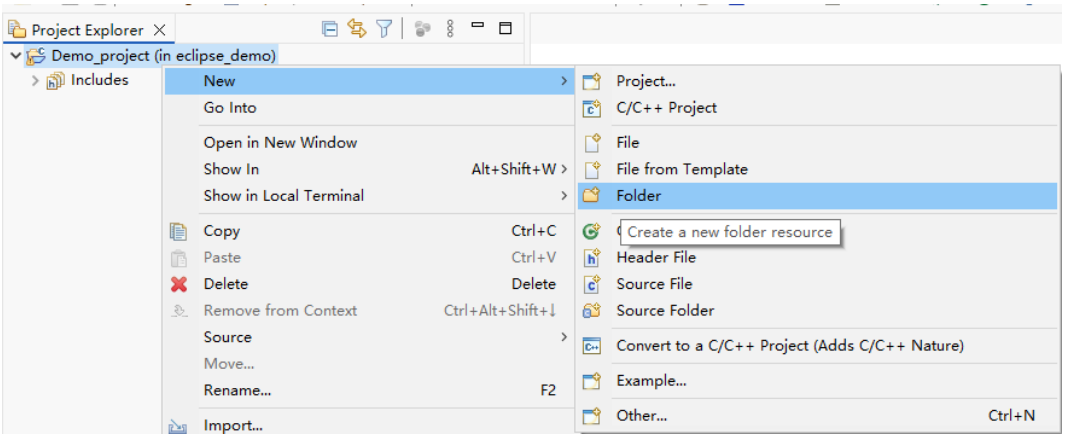


5. Click “Finish” to complete the project setup.

6.8 File Import

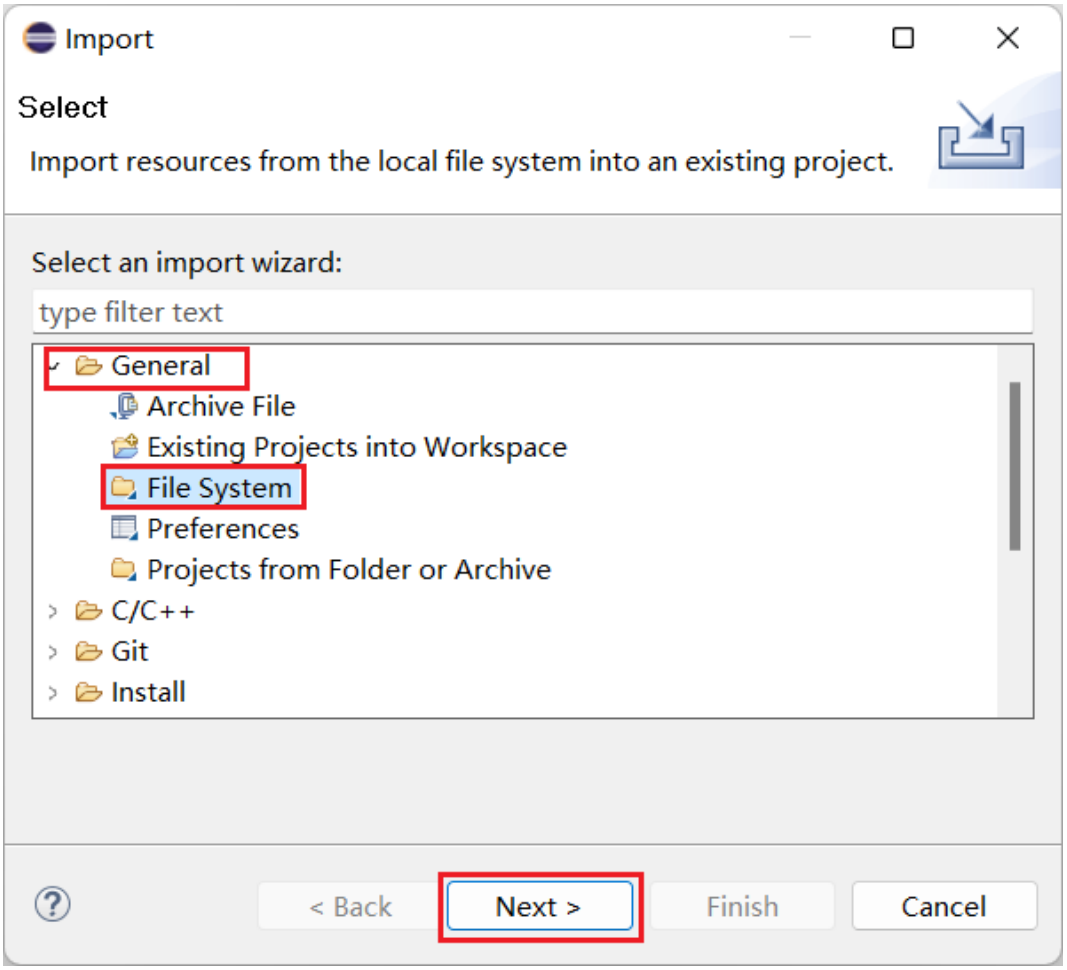
1. Right-click on the project folder to create a virtual folder.

Figure 36 Create a folder



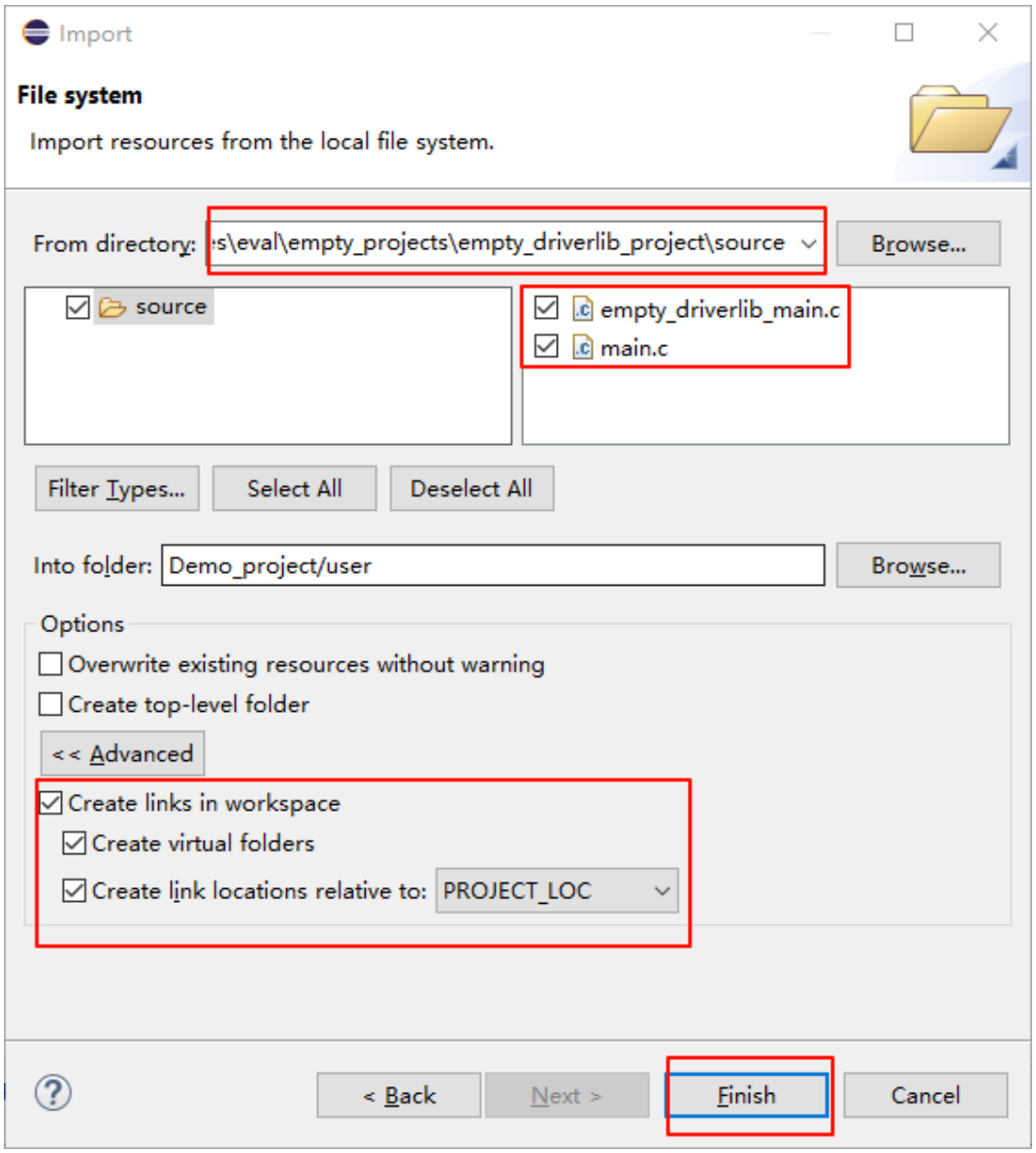
2. Select the folder, right-click and choose Import to directly import files.

Figure 37 Import file



3. When importing files, select "File System" as the import method; in the pop-up dialog, choose the path and check the files to import.

Figure 38 Select file



4. To create a new source file, click "File" → "New", select the source file type, and choose the folder and filename.

6.9 Configure Eclipse global tool paths (Clang / Make / GDB)

Before project-level **compilation configuration**, write host toolchain paths into Eclipse **global** preferences so every project does not need repeated Browse, and so Eclipse is less likely to miss tools that exist only in the system PATH.

Eclipse + Clang projects in this SDK typically need:

- LLVM-ET-Arm (“clang”, “llvm-objcopy”, …), for example “D:\desktop\clang\utilities\LLVM-ET-Arm-19.1.1-Windows-x86_64\bin”

- xpack-windows-build-tools (“make”), for example “C:\Tools\xpack-windows-build-tools-4.4.1-3\bin”
- Arm GNU Toolchain (“arm-none-eabi-gdb” for Debug), for example “C:\Program Files (x86)\Arm GNU Toolchain arm-none-eabi\14.2.rel1\bin”

Note: Download and unpack of LLVM-ET-Arm are described under **LLVM-ET-Arm toolchain** above. This section only covers writing paths into Eclipse.

Recommended steps:

1. **Window** → **Preferences**.
2. Expand **MCU** (or **C/C++** → **Build** → **Environment**, depending on the GNU MCU Eclipse plug-in version).
3. Set global tool paths (labels vary by plug-in version):

- **Global LLVM / Clang path** (or Toolchain path): LLVM-ET-Arm “bin”
- **Build tools / Make path**: xpack-windows-build-tools “bin” (so “make.exe” resolves)
- **GDB Client** (if offered on a global Debug page): “arm-none-eabi-gdb.exe”

4. If the plug-in has no dedicated global-tool page, edit **C/C++** → **Build** → **Environment**, prepend the three “bin” directories to “PATH”, and apply to the workspace.
5. Click **Apply and Close**.

After importing a G32R5xx Eclipse project, confirm **Properties** → **C/C++ Build** → **Settings** → **Toolchains** uses the built-in llvm clang and xpack make (consistent with **Configure ToolChain** below). If tools are still not found, recheck path spelling and that each entry points to a “bin” directory.

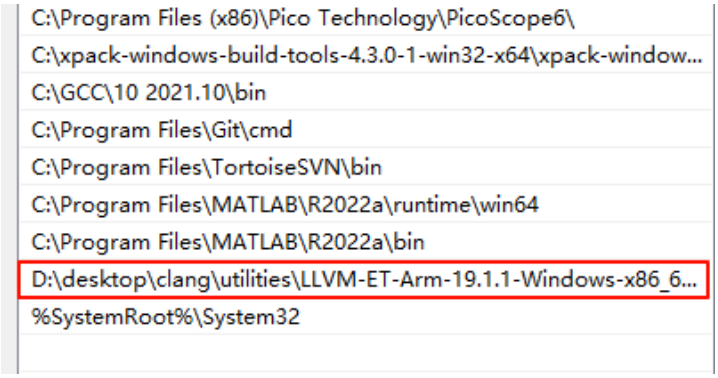
Note:

Avoid unquoted spaces in paths where possible; quote paths in Environment or command lines when required.

Global pyOCD path is configured separately under **Integration with Eclipse** (**Global pyOCD Path**); do not mix it with Clang/Make paths.

Complete this section before **Compilation Configuration** below.

Figure 39 Toolchain paths in the system Path environment variable (example)



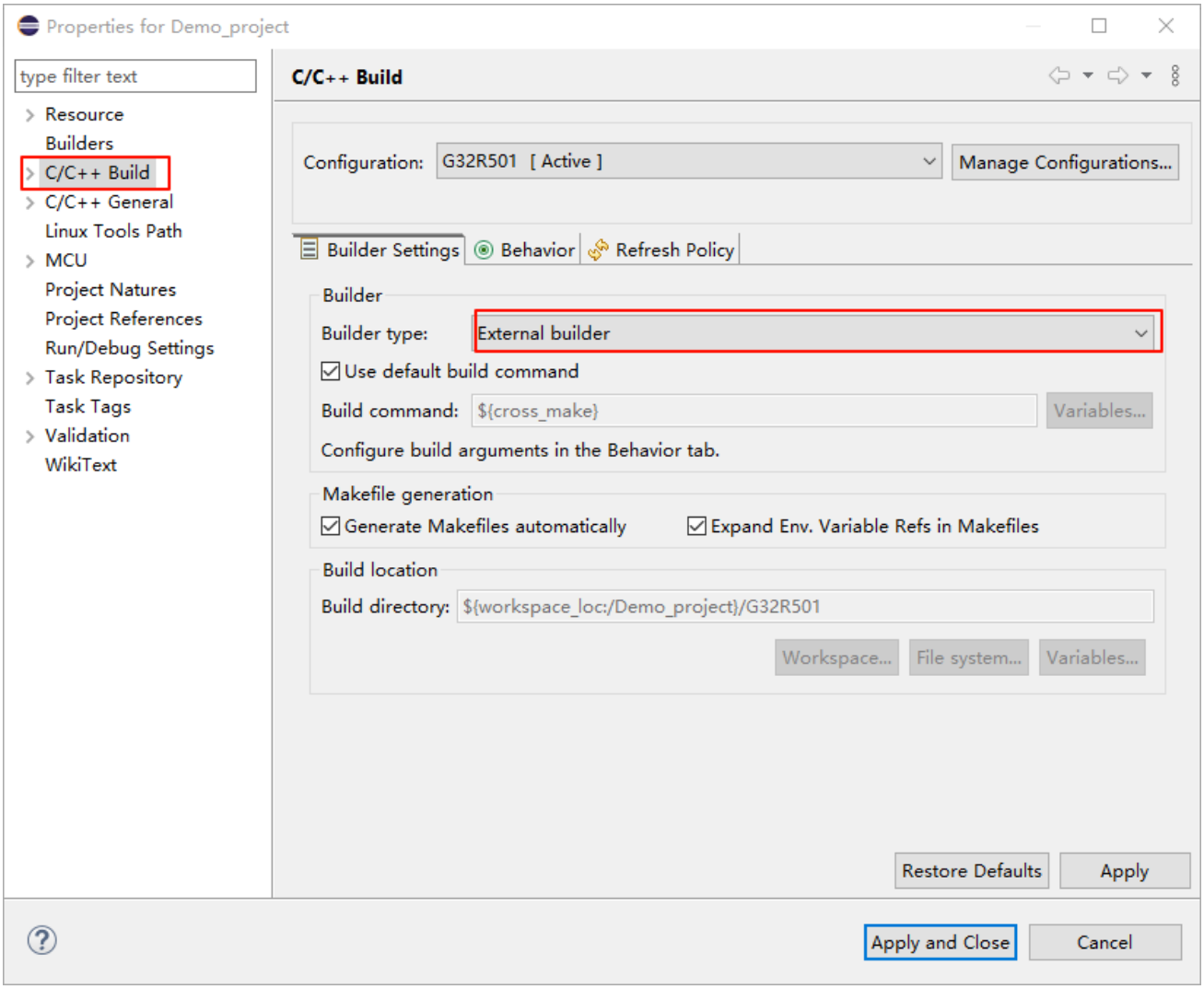
Note: If the Preferences page for separate Clang/Make/GDB global paths differs from this Path example, follow the actual Eclipse MCU plug-in menus on the host. The Path must resolve to the toolchain executables in use.

6.10 Compilation Configuration

Right-click the project name and select **Properties**.

Choose the **External builder** configuration:

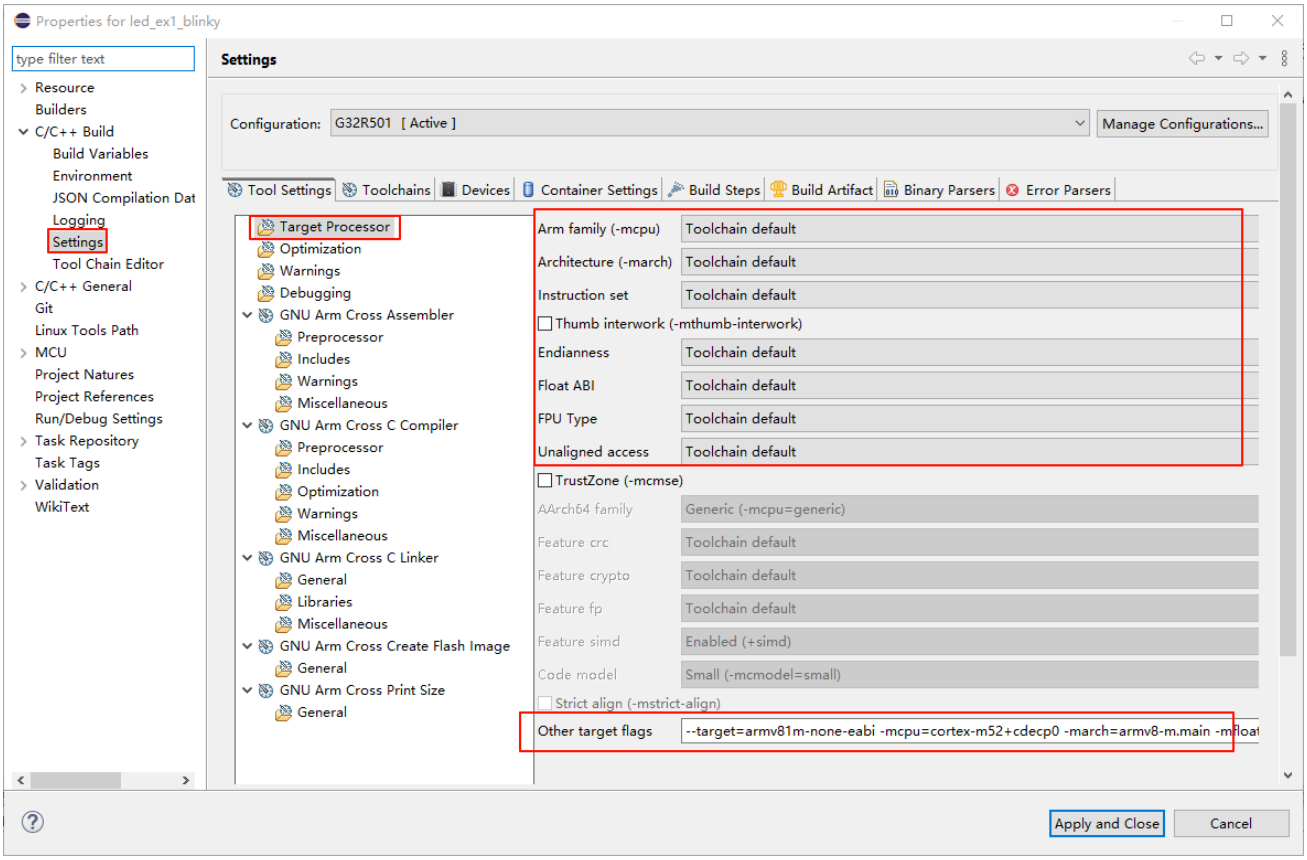
Figure 40 External builder



Under the "C/C++ Build" tab, select "Settings" -> "Tool Settings" -> "Target Processor"

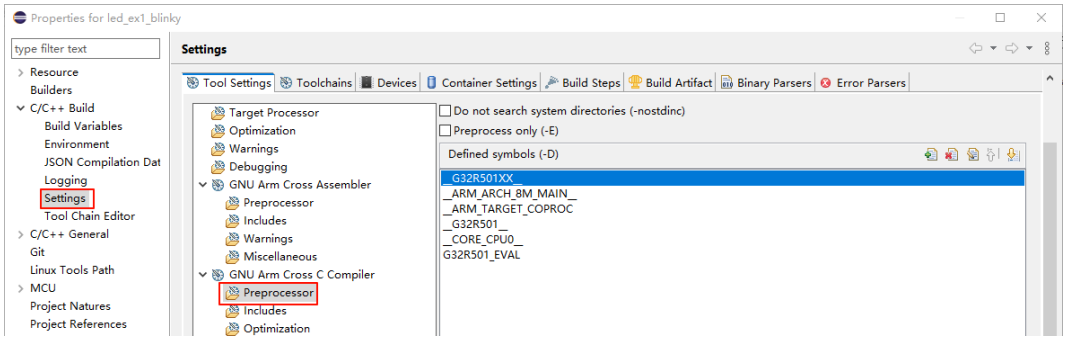
Set all dropdown configuration items to default, then manually add the command line: “--target=armv81m-none-eabi -mcpu=cortex-m52 -mfpu=none -fno-exceptions -fno-rtti -lcrt0-semihost –lsemihost” (These parameters can be modified according to the actual chip specifications.)

Figure 41 Target Processor



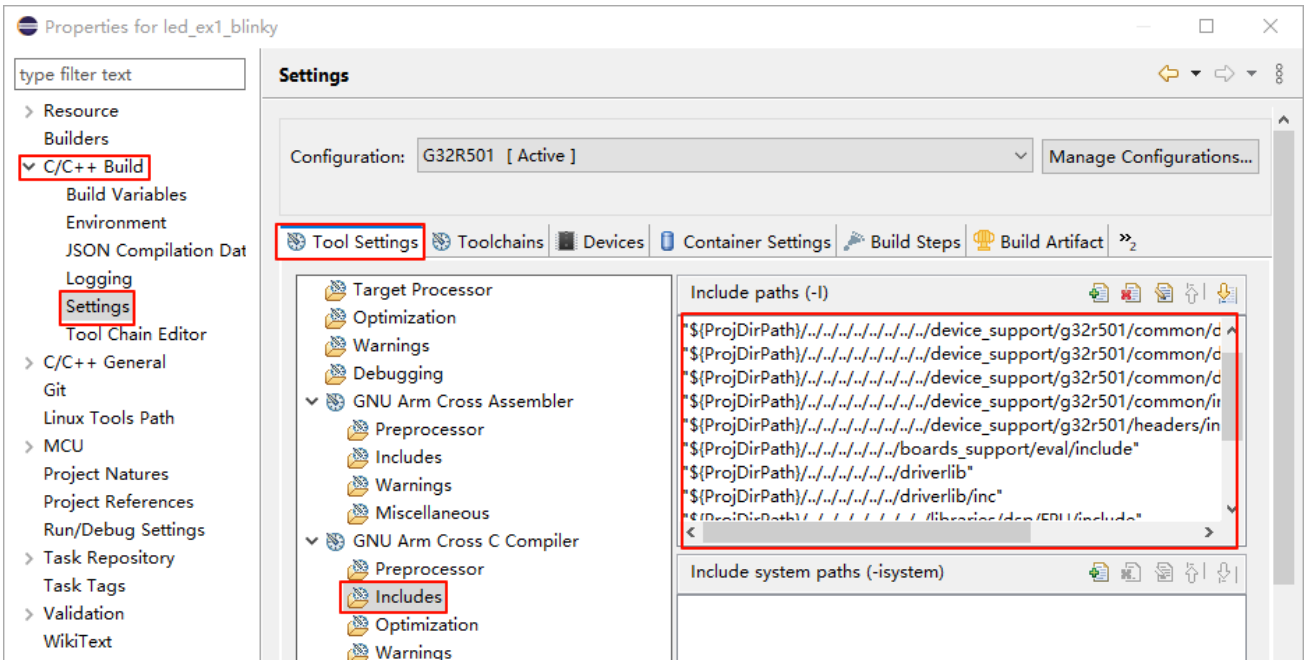
Under the "C/C++ Build" tab, select "Settings" -> "Tool Settings" -> "GNU Arm Cross C Compiler" -> "Preprocessor"

Figure 42 Add macros



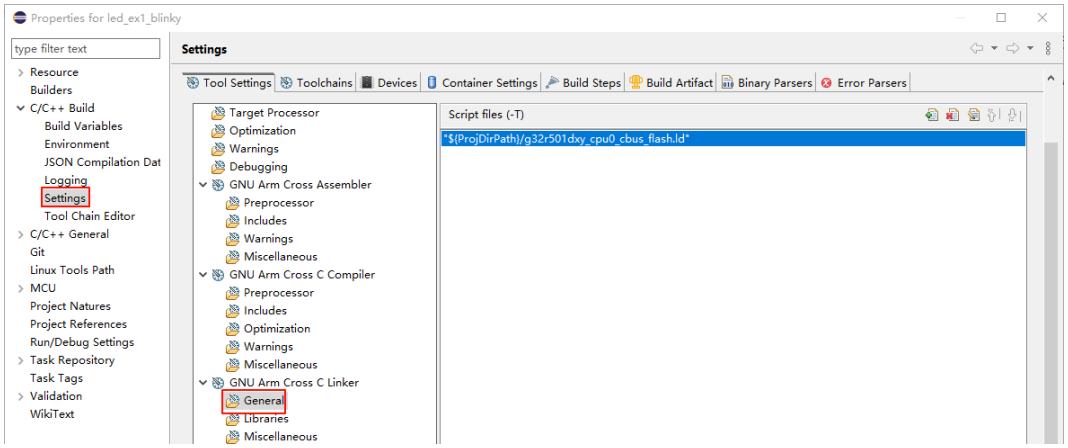
Under the "C/C++ Build" tab, select "Settings" -> "Tool Settings" -> "GNU Arm Cross C Compiler" -> "Includes".

Figure 43 Configure include paths



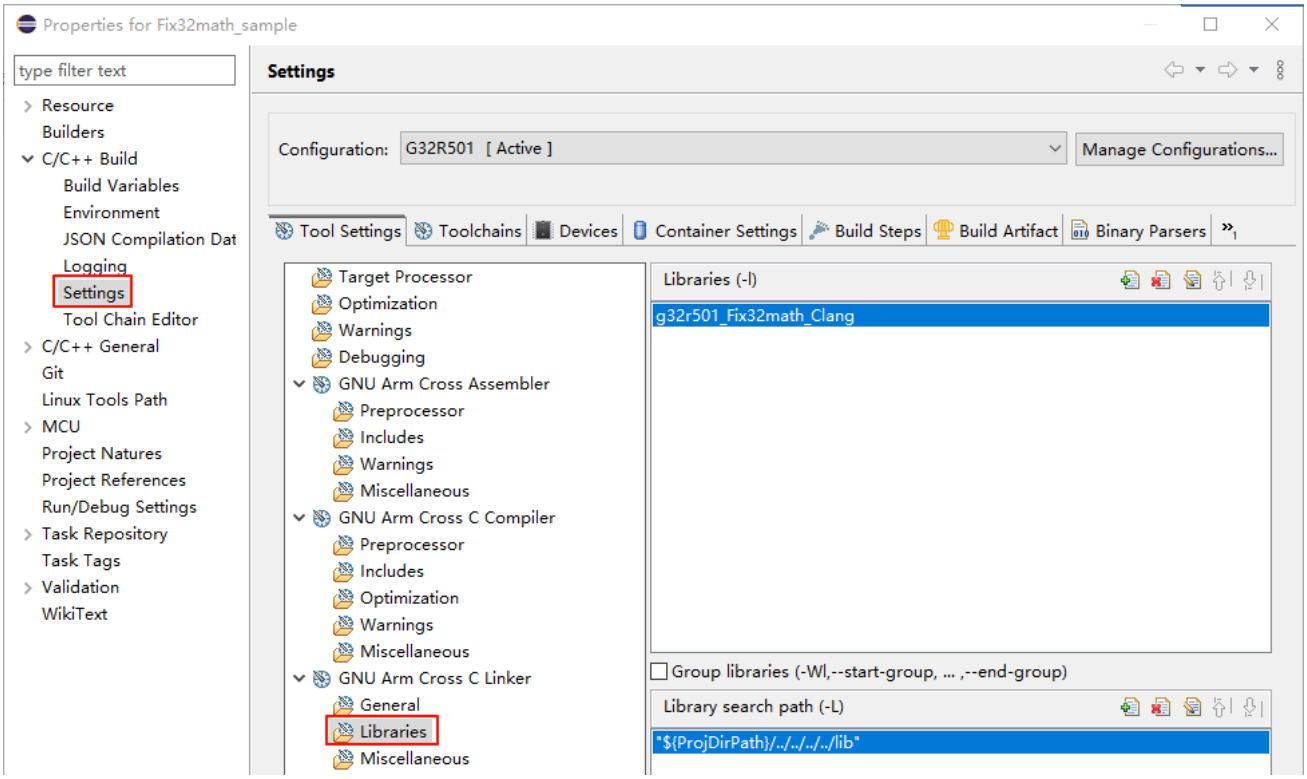
Under the "C/C++ Build" tab, select "Settings" -> "Tool Settings" -> "GNU Arm Cross C Linker" -> "General".

Figure 44 Add linker script



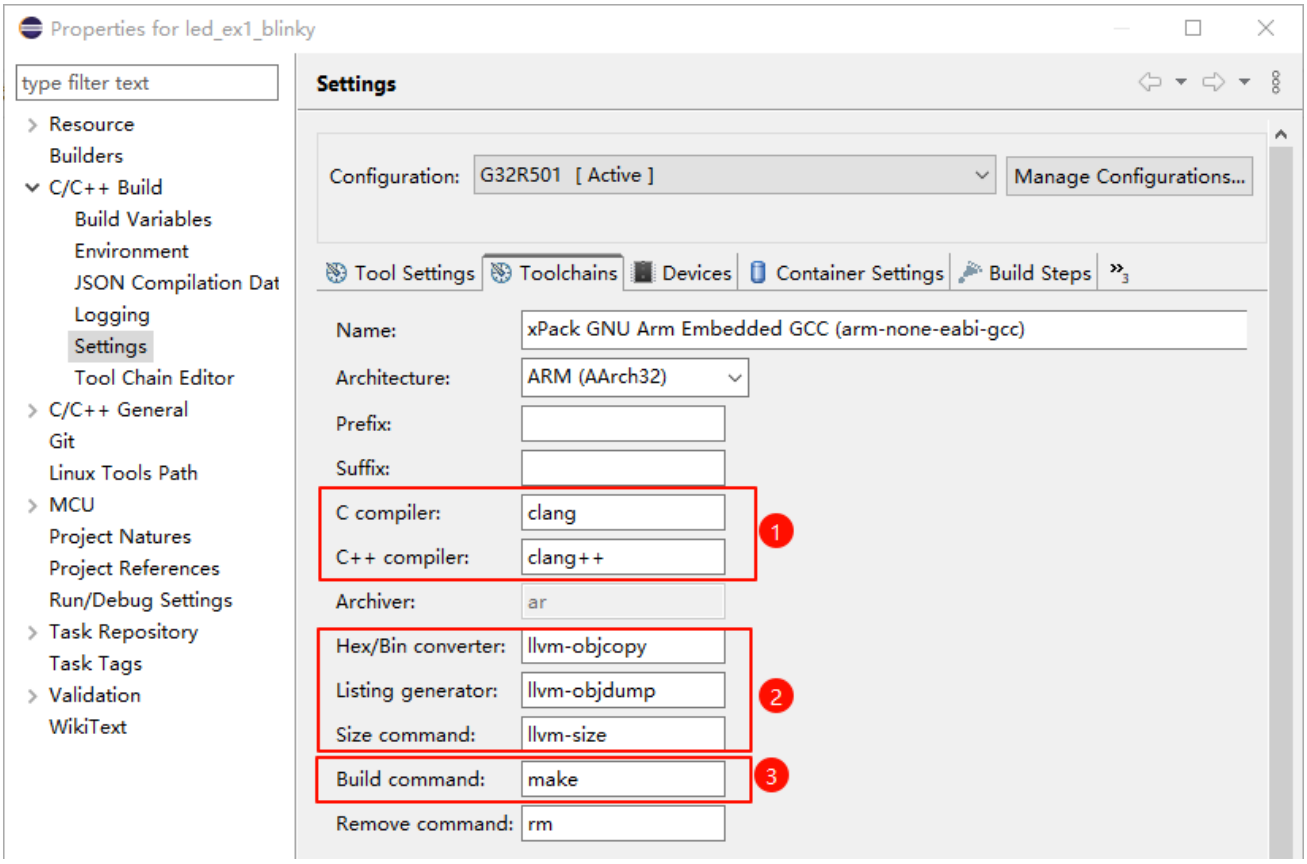
Under the "C/C++ Build" tab, select "Settings" -> "Tool Settings" -> "GNU Arm Cross C Linker" -> "Libraries".

Figure 45 Add libraries



Configure “ToolChain”.

Figure 46 Configure ToolChain



Use the built-in clang compiler from llvm.

Built-in tools from the llvm toolchain.

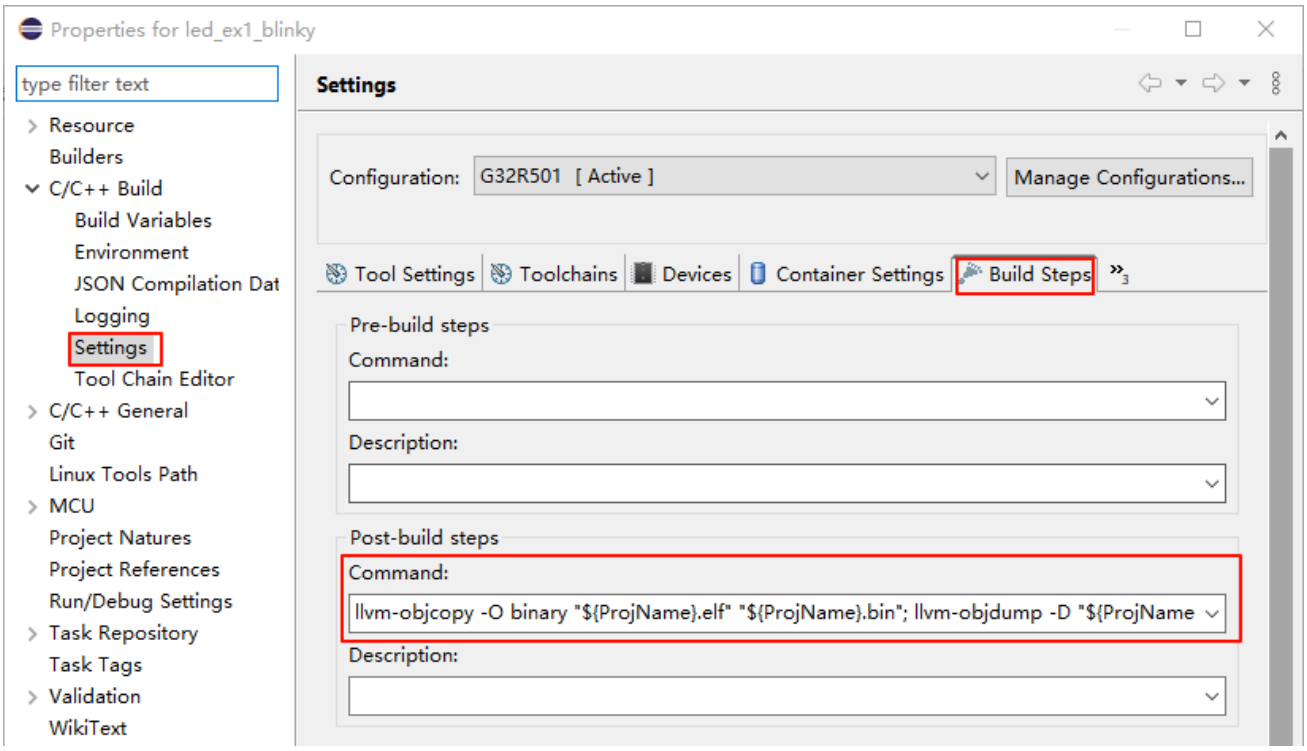
Use the built-in make compiler from xpack.

Configure "Build Step":

Under the "C/C++ Build" tab, select "Settings" -> "Build Step" -> "Post-build steps" -> "Command".

Add the command: “llvm-objcopy -O binary "${ProjName}.elf" "${ProjName}.bin"; llvm-objdump -D "${ProjName}.elf" > "${ProjName}.dump"”.

Figure 47 Build Step

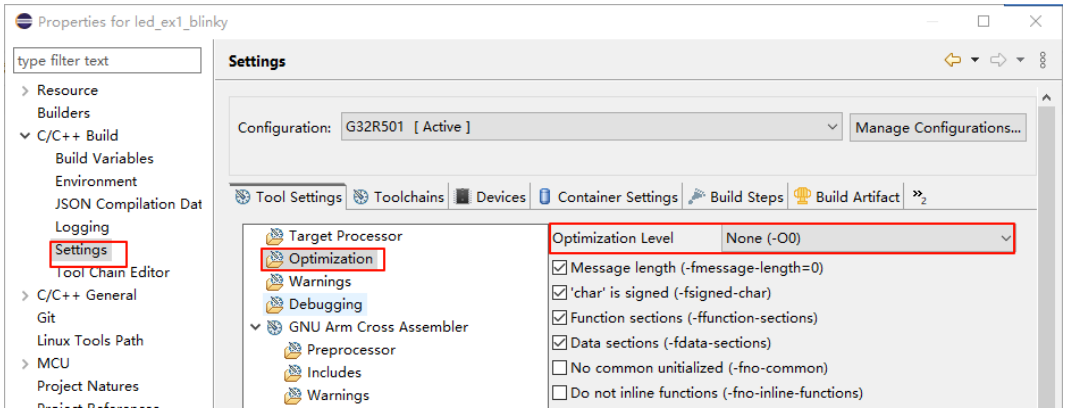


6.11 Compilation Optimization Level Settings

Eclipse provides multiple optimization level settings that can be adjusted at the global, single-file, and single-function levels:

1. Global Optimization Level Setting: Under the "C/C++ Build" tab, select "Settings" -> "Tool Settings" -> "Optimization" -> "Optimization Level".

Figure 48 Global optimization level



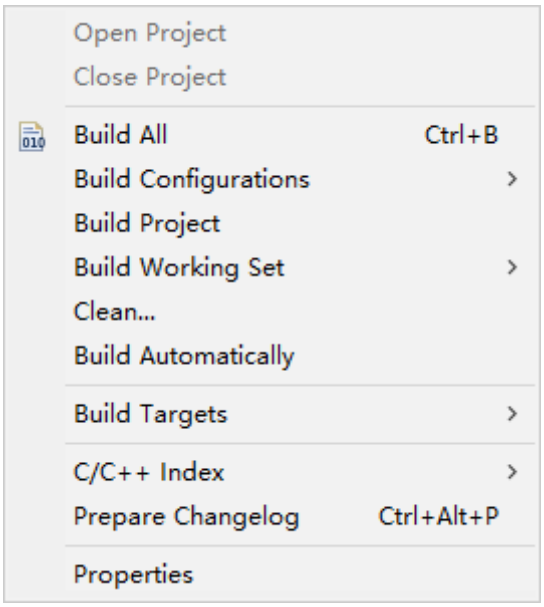
2. Single-Function Optimization Level Setting: Specific compiler directives can be used before functions to set optimization levels, such as using “_Pragma("optimize=none")” to declare no optimization.

6.12 Program Compilation

1. Click **Project**, then select **Build Project** to compile the current project.

Note: Build Project compiles only the current project; **Build All** compiles every project in the workspace.

Figure 49 Build the project



Note:

（1） Save the project before building; otherwise the last saved snapshot may be built.

（2） After changes, run **Clean**, then build; ".elf", ".hex", and ".bin" files are produced when complete.

（3） Wait for compilation to finish and check the output window for errors or warnings; fix any issues indicated.

6.13 Program Debug (download)

6.13.1 J-Link Debug

6.13.1.1 Install J-Link device support (required first)

SEGGER J-Link does not recognize G32R501 / G32R502 by default. Install the device pack before using Eclipse J-Link GDB Server.

Files to prepare

- G32R501: from "device_support/g32r501/common/Jlink/" take "G32R501.xml" and "Geehy_G32R501_ConnectCore.jlinkscript"; from SDK "package/FLM/" take "G32R5xx_Program_Algorithm.FLM" (for OTP debug also take "G32R5xx_OTP.FLM").
- G32R502: from "device_support/g32r502/common/jlink/" take "G32R502.xml" and "Geehy_G32R502_ConnectCore.jlinkscript"; from SDK "package/FLM/" take "G32R502_Program_Algorithm.FLM" (for OTP debug also take "G32R502_OTP.FLM").

The FLM files in SDK "package/FLM/" take precedence; if the DFP pack contains same-named

algorithm files, their content should match the SDK copies.

Windows

1. In File Explorer, enter “%APPDATA%\SEGGER\JLinkDevices\” and press Enter (create the folder if missing).
2. Create “Geehy\G32R501\” or “Geehy\G32R502\” and copy the three files into it.
3. Example: “C:\Users\<your_user>\AppData\Roaming\SEGGER\JLinkDevices\Geehy\G32R502\”
4. Close and reopen Eclipse and other J-Link tools.

Linux / macOS

Copy to “$HOME/.config/SEGGER/JLinkDevices/Geehy/G32R501/” or “.../G32R502/”.

Verify

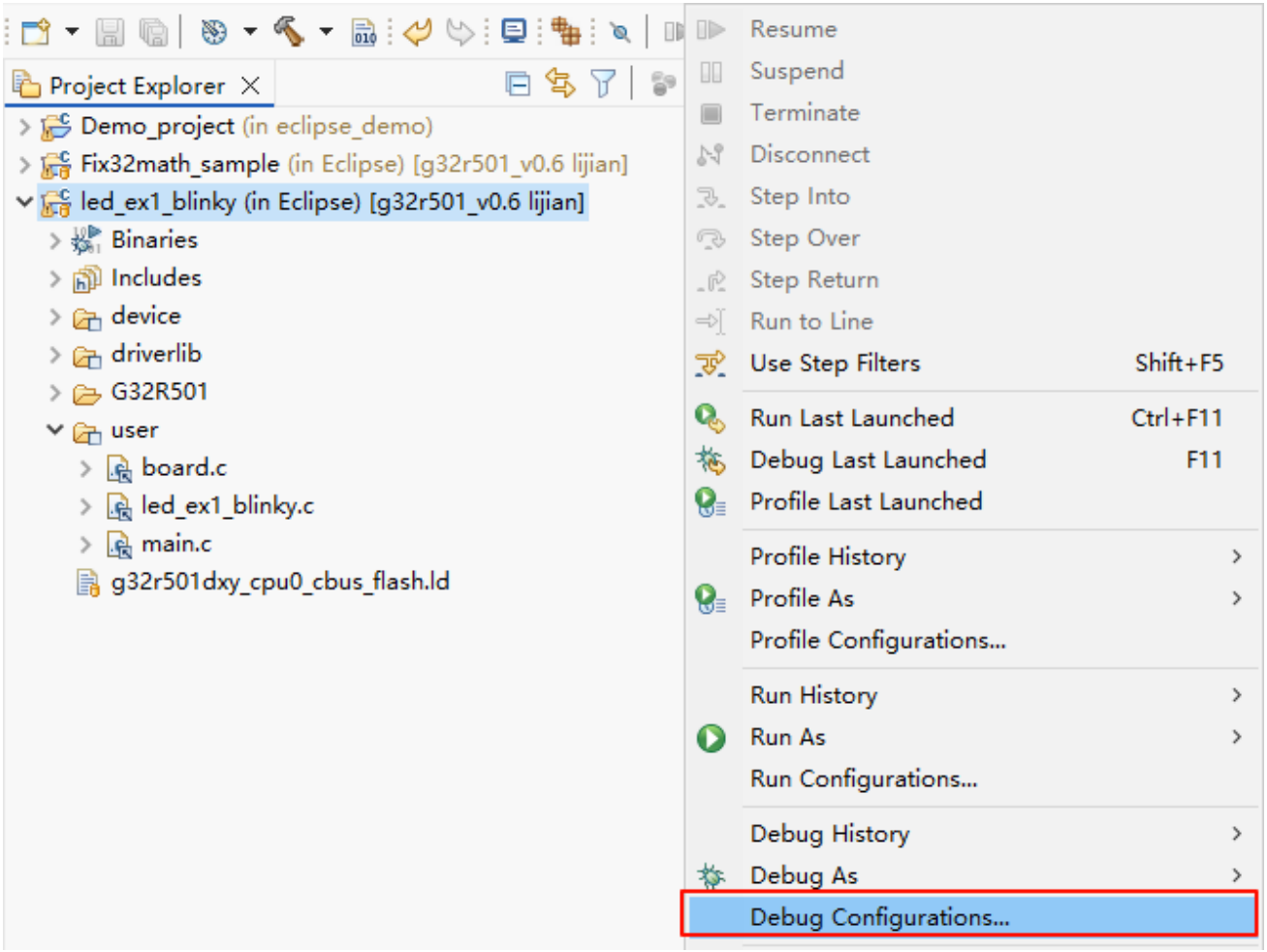
Open J-Link Commander, enter “device ?”, and confirm Geehy lists the expected part. If OTP keys were changed, edit “Z1_CSM_Buff” / “Z2_CSM_Buff” in the ConnectCore script before installation.

6.13.1.2 Configure J-Link in Eclipse

Open the Debug configuration interface:

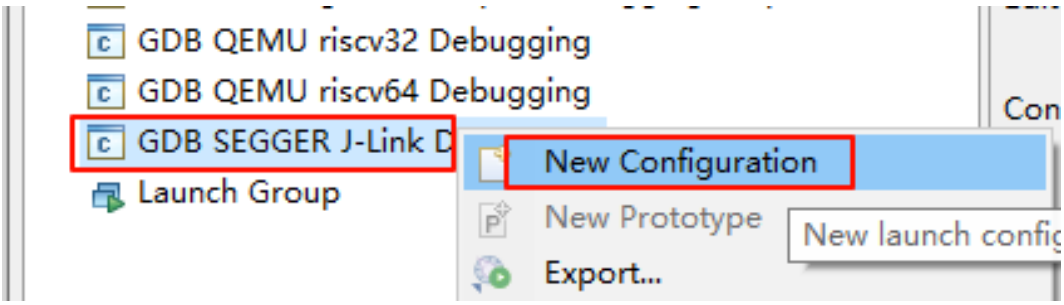
In the project, click **Run** in the menu bar, then select **Debug Configurations**.

Figure 50 Debug configuration view



In the new window, select **GDB SEGGER J-Link Debugging**, right-click, and choose **New Configuration**.

Figure 51 Create a J-Link Debug configuration



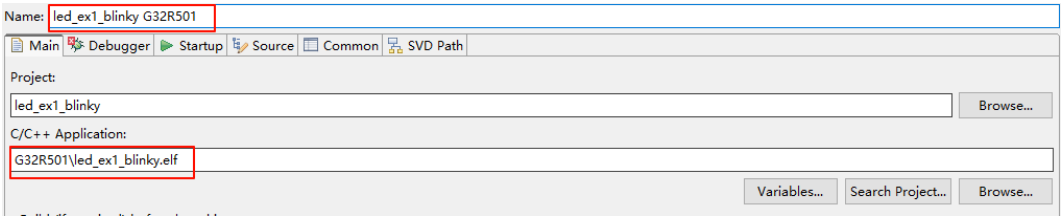
Configure the Main tab:

Name the debug configuration at the top.

Click **Browse...** and select the project for this configuration.

Select the debug ELF file, for example “G32R501\led_ex1_blinky.elf”. Relative paths (to the project file) or absolute paths are supported.

Figure 52 Configure Main

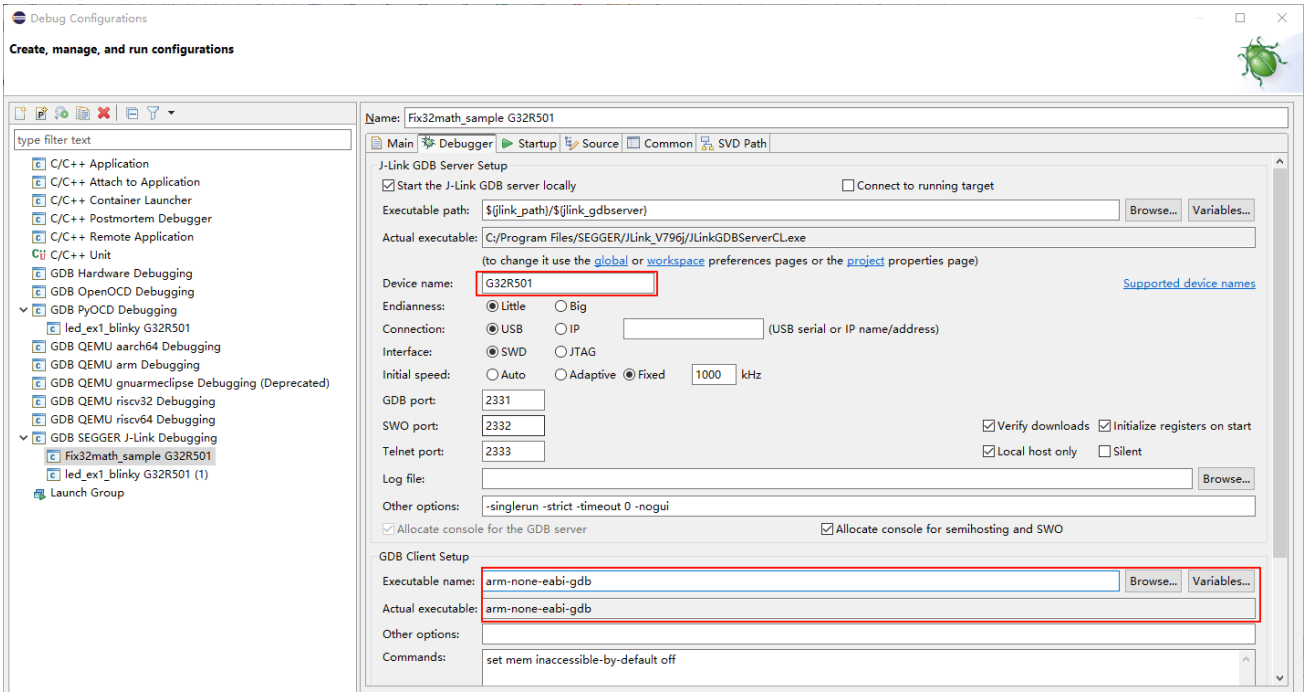


Configure the Debugger tab:

Select the target device (for example G32R501).

Choose the GDB service; Arm “arm-none-eabi-gdb.exe” is recommended.

Figure 53 Debugger tab



Configure the Startup tab:

When the chip uses **factory-default DCS keys** and the program boots from the **default Flash address** (CBUS Flash is typically “0x08000000”), you **do not** need to paste DCS KEY and vector-table scripts into **Initialization Commands** and **Run/Restart Commands**.

Recommended approach (avoids long memory-write command lists in Startup):

1. In the SDK, copy these two files into the Eclipse project directory (same level as the “.elf”, or the debug working directory):

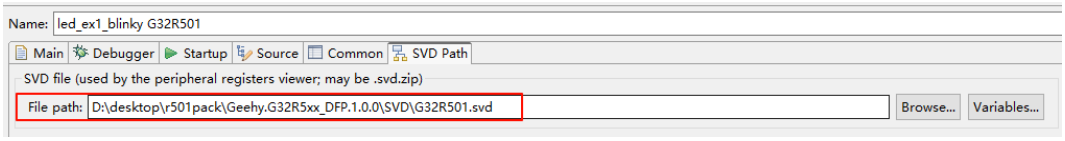
- G32R501: “device_support/g32r501/common/pyOCD/pyocd.yaml” and “pyocd_user.py”
- G32R502: “device_support/g32r502/common/pyocd/pyocd.yaml” and “pyocd_user.py”

2. Open **Debug Configurations** and select **GDB PyOCD Debugging** (or an existing pyOCD configuration in the project).
3. Set **Working directory** to the folder containing “pyocd.yaml” / “pyocd_user.py”.
4. In pyOCD / config options, specify “pyocd.yaml” (or the CLI equivalent “--config pyocd.yaml”).
5. Click **Apply**, then start debug. On connect, “pyocd_user.py” writes default DCS keys and sets the vector-table base.

Edit “pyocd_user.py” only when OTP keys were changed or the boot address is not default Flash (for example ITCM “0x00000000”). With SDK default templates, users usually need neither file edits nor Startup command blocks.

Configure the SVD tab:

Figure 54 Configure SVD path



Finally, click **Apply** at the bottom right to save all settings.

6.13.2 GEEHY LINK Debug

When downloading and debugging with Geehy-Link (WinUSB), use pyOCD. Open Chapter 7 of this document and complete, in order: install Python and pyOCD 0.44 or later → copy “target_G32R501xx.py” / “target_G32R502xx.py” into local “pyocd/target/builtin/” and register in “__init__.py” → copy the device “pyocd.yaml” and “pyocd_user.py” into the project working directory → start debug. Chapter 7 lists full commands and paths; no other document is required.

6.14 C Language Compatibility

Please refer to Chapter 4 "C Language Compatibility".

6.15 Assembly Compatibility

The instruction sets and assembly syntax of different target platforms may vary significantly. Existing assembly code may need to be rewritten and adapted to ensure correct operation on new platforms.

6.15.1 File Format Support

Eclipse uses a specific GCC assembly syntax. The assembly syntax of GCC (GNU Assembler, abbreviated as GAS) is consistent with MDK (Keil's Arm assembler, i.e., armasm) in terms of core instruction sets, but there are significant differences in pseudo-instructions, syntax formats, and symbol definitions. Below is a detailed comparison.

.S files: GCC-style assembly file format.

Additionally, it supports inline assembly within C functions.

6.15.2 Assembly Coding Format Requirements

Single assembly file: Here is a simple example of assembly code that defines an assembly function “add” to add two integers. The content of the file "add.s" is as follows:

```
.section .text   //Define code section
.global add    // Declare global symbol
add:
//Function entry
//Parameters: r0 and r1
//Return value: r0
add r0, r0, r1    // Add r0 and r1, store result in r0
bx  lr          // Return to calling function
```

Using inline assembly in C functions: The following is an example of using inline assembly in a C function, defining an inline assembly function add_inline to add two integers:

```
// Inline assembly function
static inline int add_inline(int a, int b) {
int result;
asm volatile (
"ADD %0, %1, %2"
: "=r" (result)
: "r" (a), "r" (b)
:
);
return result;
}
```

6.16 Linker Script Files

Linker script files are used to define the memory layout and section allocation of a program.

During the migration process, it is necessary to use linker script files in the appropriate format according to the target platform and development environment. The G32R5 series MCU uses ".ld" format linker script files under the Eclipse development environment, adhering to Eclipse's specifications.

Differences in Linker Script Files

File Format:

- The G32R5 series MCU uses ".ld" format linker script files.
- The Txx320F28004x uses ".CMD" format linker script files, following its company's specifications.

Memory Layout and Section Allocation:

- The ".ld" file of the G32R5 series MCU allocates memory attributes by defining different "MEMORY" and its "SECTIONS."
- The ".CMD" file of the Txx320F28004x arranges memory allocation by defining memory segments (MEMORY) and section allocations (SECTIONS).

6.17 RAM Operating

Please refer to Chapter 4 RAM Operation.

7 pyOCD adaptation for G32R5xx

7.1 Background

To help users download and debug G32R5xx MCUs (G32R501 / G32R502) with open-source tools, the SDK provides pyOCD target support files and session templates.

7.2 Version requirement

Use **pyOCD 0.44 or later**. That baseline already supports the Cortex-M52 core; **do not** patch the pyOCD source tree only to add M52 core support.

7.3 Register device targets on the local pyOCD install

The SDK provides one target module per device. Copy it into the installed pyOCD "target/builtin/" directory and register it in the same directory's "__init__.py" so IDE and CLI sessions can select the target.

Source files in the SDK:

- G32R501: "device_support/g32r501/common/pyOCD/target_G32R501xx.py"
- G32R502: "device_support/g32r502/common/pyocd/target_G32R502xx.py"

Target names (for selection):

- G32R501 single-core: "g32r501xx"
- G32R501 dual-core: "g32r501dxx" (the dual-core "pyocd.yaml" template defaults to this)
- G32R502: "g32r502xx"

Where to copy on the host (locate first, then copy)

The destination relative to the pyOCD install root is always:

"pyocd\target\builtin\"

That folder should already contain "target_.py" and "__init__.py". Typical full paths on Windows (replace "<user>" and "Python312" with your account and Python version):

pip install into official Python (most common)

"C:\Users\<user>\AppData\Local\Programs\Python\Python312\Lib\site-packages\pyocd\target\builtin\"

Some bundled layouts ("Scripts_internal")

"C:\Users\<user>\AppData\Local\Programs\Python\Python312\Scripts_internal\pyocd\target\builtin\"

Editable source install ("pip install -e .")

"<your_pyocd_source_root>\pyocd\target\builtin\"

Recommended check:

Open CMD and run:

```
pip show pyocd
```

Read the **Location** line (pyOCD package directory), then open "pyocd\target\builtin\" beneath it (if Location already ends in "...\pyocd", open "target\builtin\").

Or print the path from Python:

```
python -c "import pyocd, pathlib; print(pathlib.Path(pyocd.__file__).resolve().parent / 'target' / 'builtin')"
```

The printed path is where you copy "target_G32R501xx.py" / "target_G32R502xx.py" and edit "__init__.py".

Note: If multiple Python installs exist, ensure "python", "pip", and "pyocd" on PATH belong to the same environment; registering in one install while running another will not show new targets.

Registration steps

1. Confirm **pyOCD 0.44 or later** is installed (see **pyOCD installation** below) and open the local "...\pyocd\target\builtin\" directory.
2. Copy "target_G32R501xx.py" and/or "target_G32R502xx.py" from the SDK into that "builtin" directory (keep file names).
3. Edit "__init__.py" in the same directory. In the existing "from . import ..." block, add:

```
from . import target_G32R501xx
from . import target_G32R502xx
```

In "BUILTIN_TARGETS", add entries such as:

```
'g32r501xx': target_G32R501xx.G32R501xx,
'g32r501dxx': target_G32R501xx.G32R501Dxx,
'g32r502xx': target_G32R502xx.G32R502xx,
```

4. Save the file. With "pip install -e .", changes usually take effect immediately; with a normal install, restart the shell before verifying.

Verify on Windows:

```
pyocd list -t | findstr g32r50
```

```
```

Linux / macOS:

```
pyocd list -t | grep g32r50
```

The list should include “g32r501xx”, “g32r501dxx”, and/or “g32r502xx” as registered. You may also run:

```
pyocd list -t -n g32r50
```

Note: pyOCD 0.44+ already includes Cortex-M52 core support. **Do not** hand-edit “component_ids” / “core_ids” to add the core; only copy and register the target modules above.

7.4 Recommended session files (including default keys)

Besides target registration, place session files in the **current project working directory** (same directory as the “.elf” / Debug configuration working directory).

Copy from the SDK (back up same-named files first):

- G32R501: “device_support/g32r501/common/pyOCD/pyocd.yaml”, “pyocd_user.py”
- G32R502: “device_support/g32r502/common/pyocd/pyocd.yaml”, “pyocd_user.py”

“pyocd.yaml” typically points to “user_script: pyocd_user.py” and sets “target_override”. Example (G32R502; use the matching G32R501 template as needed):

```
* target_override: g32r502xx
user_script: pyocd_user.py
flash.timeout.init: 30.0
frequency: 8000000
persist: true
```

“target_override” values:

- G32R501 dual-core default: “g32r501dxx”
- G32R501 single-core only: “g32r501xx” (adjust “session” per template comments)
- G32R502: “g32r502xx”

“pyocd_user.py” contains the default DCS key table and vector-table base:

- With **factory-default keys** and boot from **Flash “0x08000000”**, **do not** edit this file
- Edit key pairs or base constants only when OTP keys changed or the boot address moved (for example ITCM)

From the working directory:

```
pyocd gdbserver --config pyocd.yaml
```

Or:

```
pyocd cmd --config pyocd.yaml
```

In Eclipse: open **Debug Configurations**, select a pyOCD configuration, set the working directory to the folder containing the two files, specify “pyocd.yaml”, then start debug.

Command-line self-test (board connected)

```
pyocd list -p
pyocd erase --config pyocd.yaml -c 0x08000000
pyocd load --config pyocd.yaml <your_program.elf>
pyocd gdbserver --config pyocd.yaml
```

7.5 pyOCD installation

7.5.1 Windows

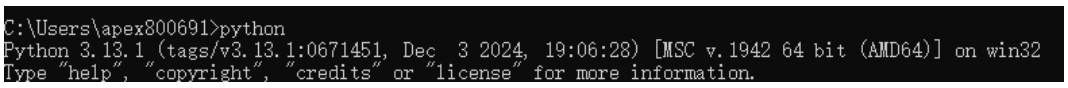
7.5.1.1 Install Python

pyOCD requires Python. Install from the Python official site.

Note: After installation, add Python to the system PATH so it is available from CMD.

Verify: press Win+R, enter CMD, run “python”, and confirm the version and REPL start. Exit with “exit()” or “quit()”.

Figure 55 Verify Python installation



7.5.1.2 Install pyOCD

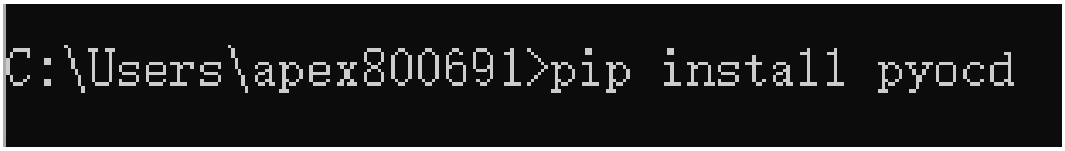
pyOCD is a Python package; online install is recommended so dependencies resolve automatically.

Example:

```
pip install "pyocd>=0.44"
```

After install, confirm the version is 0.44 or later.

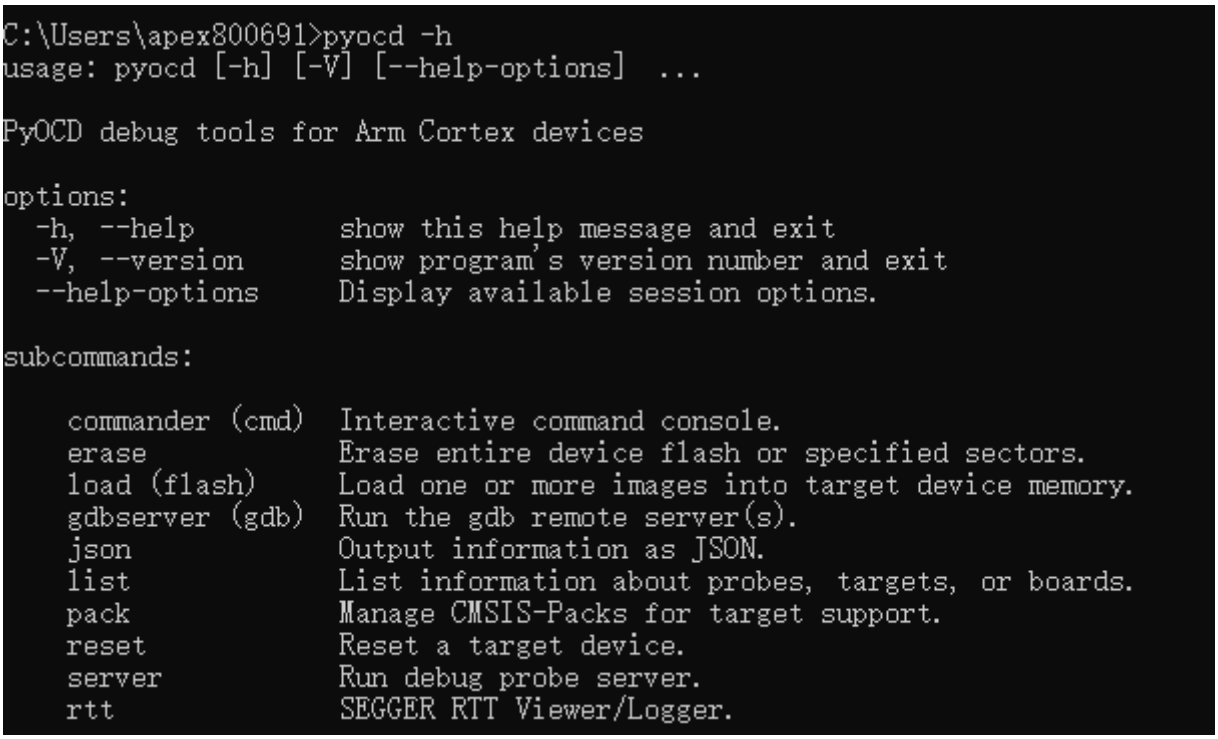
Figure 56 Install pyOCD



Note: Add the pyOCD executable directory to PATH. Use **pyOCD 0.44 or later**.

Verify: in CMD run “pyocd -h” and confirm the help text appears.

Figure 57 Verify pyOCD installation



7.5.2 Ubuntu

7.5.2.1 Install Python

Ubuntu often includes Python. Check versions:

```
python --version
```

If Python or pip is missing:

```
sudo apt update
sudo apt install python3 python3-pip
```

7.5.2.2 python3-venv

Some Ubuntu Python environments are externally managed; use a venv instead of installing into the global environment.

Install venv support:

```
sudo apt install python3-venv
```

Create a virtual environment (example name “venv”):

```
python3 -m venv venv
```

Activate it:

```
source venv/bin/activate
```

Deactivate later with:

```
deactivate
```

7.5.2.3 Install pyOCD

In the activated virtual environment:

```
pip install pyocd
```

Verify:

```
pyocd --version
```

Locate the install (for later source edits):

```
pip show pyocd
```

7.5.2.4 pyOCD USB permissions

If pyOCD cannot access the probe as a normal user, add a udev rule (Geehy-Link USB vendor ID is 314B):

1. Create a new rules file:

```
sudo nano /etc/udev/rules.d/99-pyocd.rules
```

2. Add:

```
SUBSYSTEM=="usb", ATTR{idVendor}=="314b", MODE="0666"
```

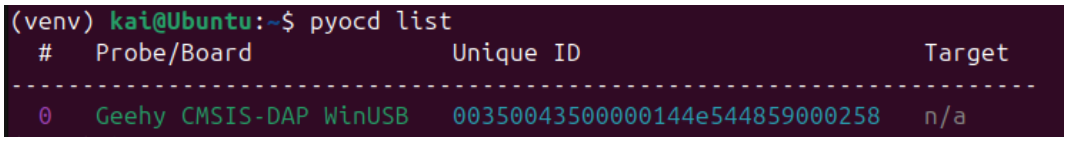
3. Save (Ctrl+O, Enter) and exit (Ctrl+X), then reload rules:

```
sudo udevadm control --reload-rules
sudo udevadm trigger
```

4. Verify probe detection:

```
pyocd list
```

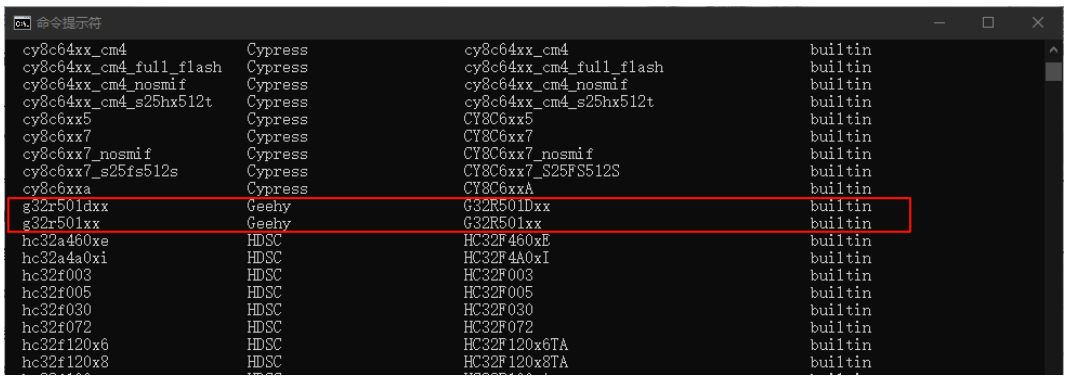
Figure 58 Probe serial number on Ubuntu



7.5.2.5 Confirm target registration

After completing **Register device targets on the local pyOCD install**, run “pyocd list -t” (filter with “grep g32r50”) and confirm “g32r501xx” / “g32r501dxx” / “g32r502xx” appear. If not, recheck files in “pyocd/target/builtin/”, “__init__.py”, and that CMD uses the same Python / pyOCD install.

Figure 59 Supported target list



7.6 Command Line Usage

pyOCD supports CMD/terminal operation (ensure pyOCD is on PATH).

1. Connect the board or probe to the PC.
2. Prepare session files in the working directory (same as **Recommended session files**):

- Copy “pyocd.yaml” and “pyocd_user.py” from “device_support/g32r501/common/pyOCD/” or “device_support/g32r502/common/pyocd/”.
- Confirm “target_override” in “pyocd.yaml” matches registered target names.
- With default keys and default Flash boot address, do not edit “pyocd_user.py”.

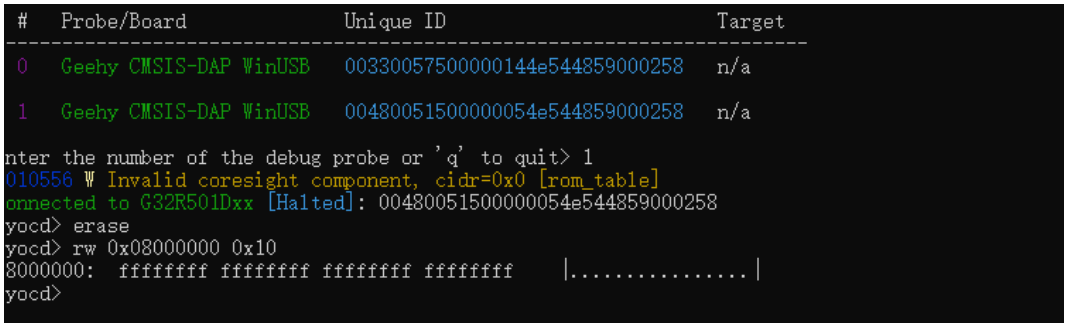
3. From that directory run:

```
pyocd commander --config pyocd.yaml
```

Or:

```
pyocd cmd --config pyocd.yaml
```

Figure 60 pyocd commander



7.7 Integration with Eclipse

Eclipse + pyOCD works best with Eclipse 202503.

Configure pyOCD globally in Eclipse (some hosts do not propagate PATH correctly):

1. On the menu bar, open **Window** → **Preferences**.
2. Expand **MCU**.
3. Select **Global pyOCD Path**.
4. On the right, choose the directory containing “pyocd.exe”.
5. Click **Apply and Close**.

Figure 61 Global pyOCD Path steps 1–2

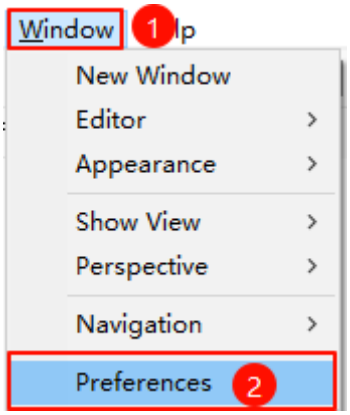
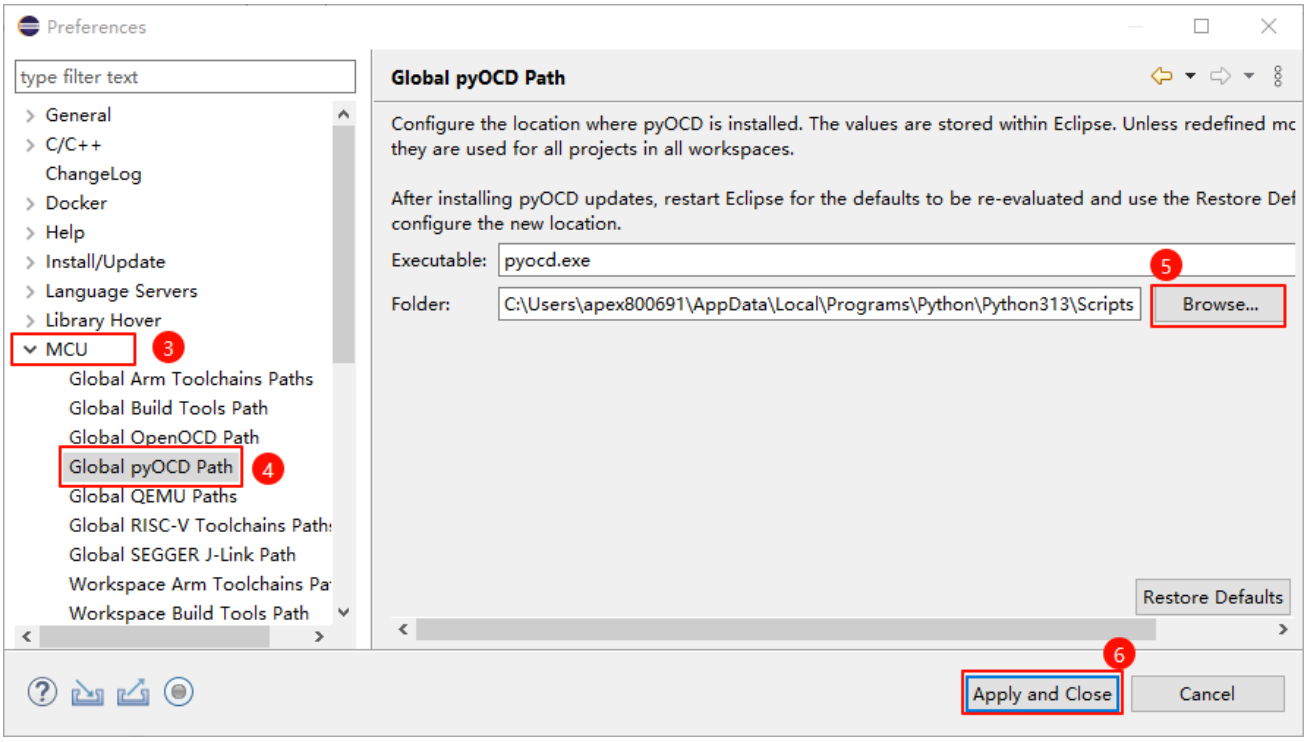


Figure 62 Global pyOCD Path steps 3–6



7.7.1 Single-core Debug configuration

After importing the project into Eclipse and confirming it builds, configure debug as follows.

Create a new debug configuration

1. Left-click the Debug icon to show debug options.
2. Select **Debug Configurations...**.
3. In the new window, select **GDB PyOCD Debugging** and right-click.
4. Choose **New Configuration**.

Figure 63 New Configuration steps 1–2

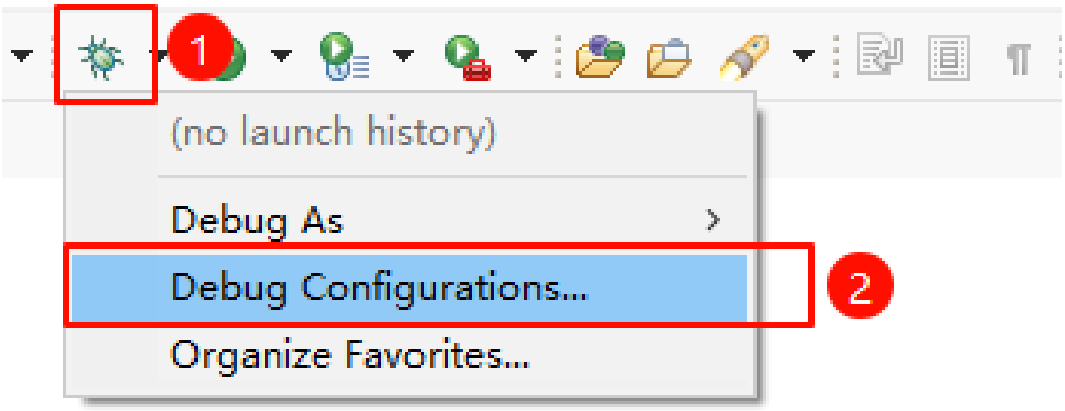
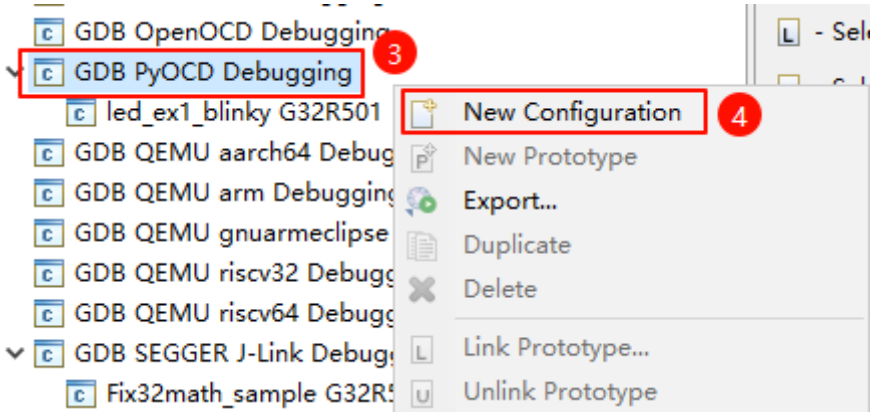


Figure 64 New Configuration steps 3–4



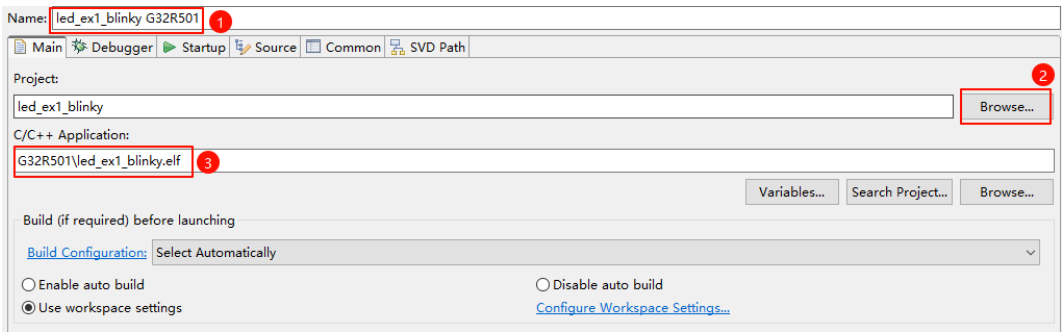
Configure the Main tab:

Name the debug configuration at the top.

Select **Browse...** and choose the project.

Select the debug ELF file, for example “G32R501\led_ex1_blinky.elf”. Relative or absolute paths are supported.

Figure 65 Configure Main



Configure the Debugger tab:

Select the Geehy-Link Debugger to be used. The characters in parentheses represent the Debugger serial number.

Choose the corresponding debug device, e.g., “Geehy > G32R501xx (g32r501xx)”.

For the reset mode, select "Software (SYSRESETREQ)."

For the configuration file, use both “pyocd.yaml” and “pyocd_user.py” (copy from “device_support/g32r501/common/pyOCD/” or “device_support/g32r502/common/pyocd/” into the project working directory). In the Eclipse Debugger tab, prefer the **absolute path** to “pyocd.yaml”; “pyocd_user.py” alone is also acceptable. Avoid Chinese characters or spaces in paths.

Example “pyocd.yaml” (G32R502; for G32R501 use the matching template; dual-core default may be “g32r501dxx”):

```
* target_override: g32r502xx
user_script: pyocd_user.py
flash.timeout.init: 30.0
frequency: 8000000
persist: true
```

The vector-table base is configured in “pyocd_user.py” (for example “VECTOR_TABLE_ADDRESS” / “CORE0_SET_ADDR”), not in the yaml.

Command-line examples:

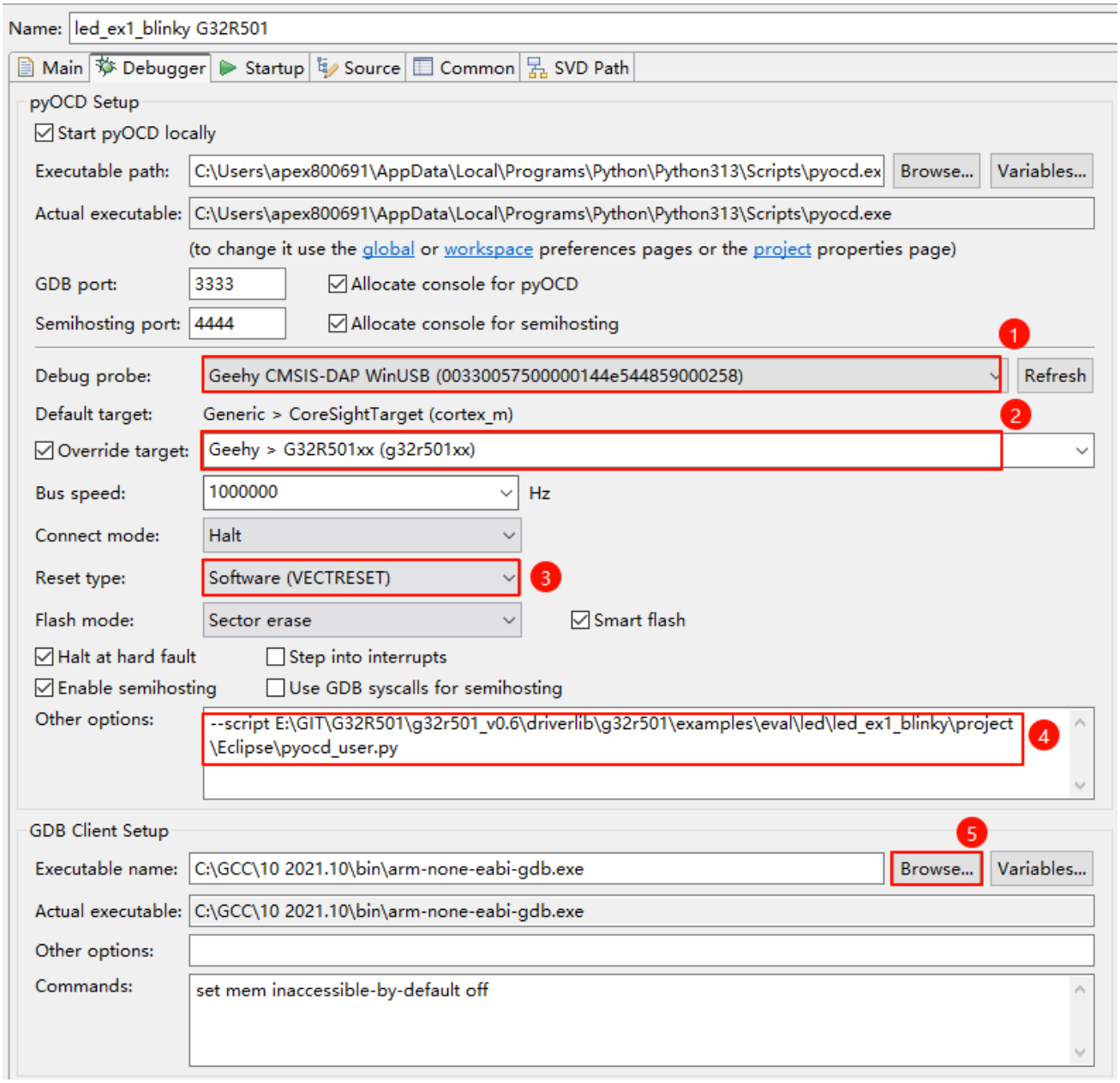
```
pyocd gdbserver --config pyocd.yaml -j <dir containing yaml and pyocd_user.py>
```

Or:

```
pyocd gdbserver --config <absolute-path>/pyocd.yaml --target g32r502xx
```

For the GDB service, use Arm “arm-gnu-toolchain-14.2.rel1\bin\arm-none-eabi-gdb.exe”.

Figure 66 Configure Debugger



Configure the Startup tab:

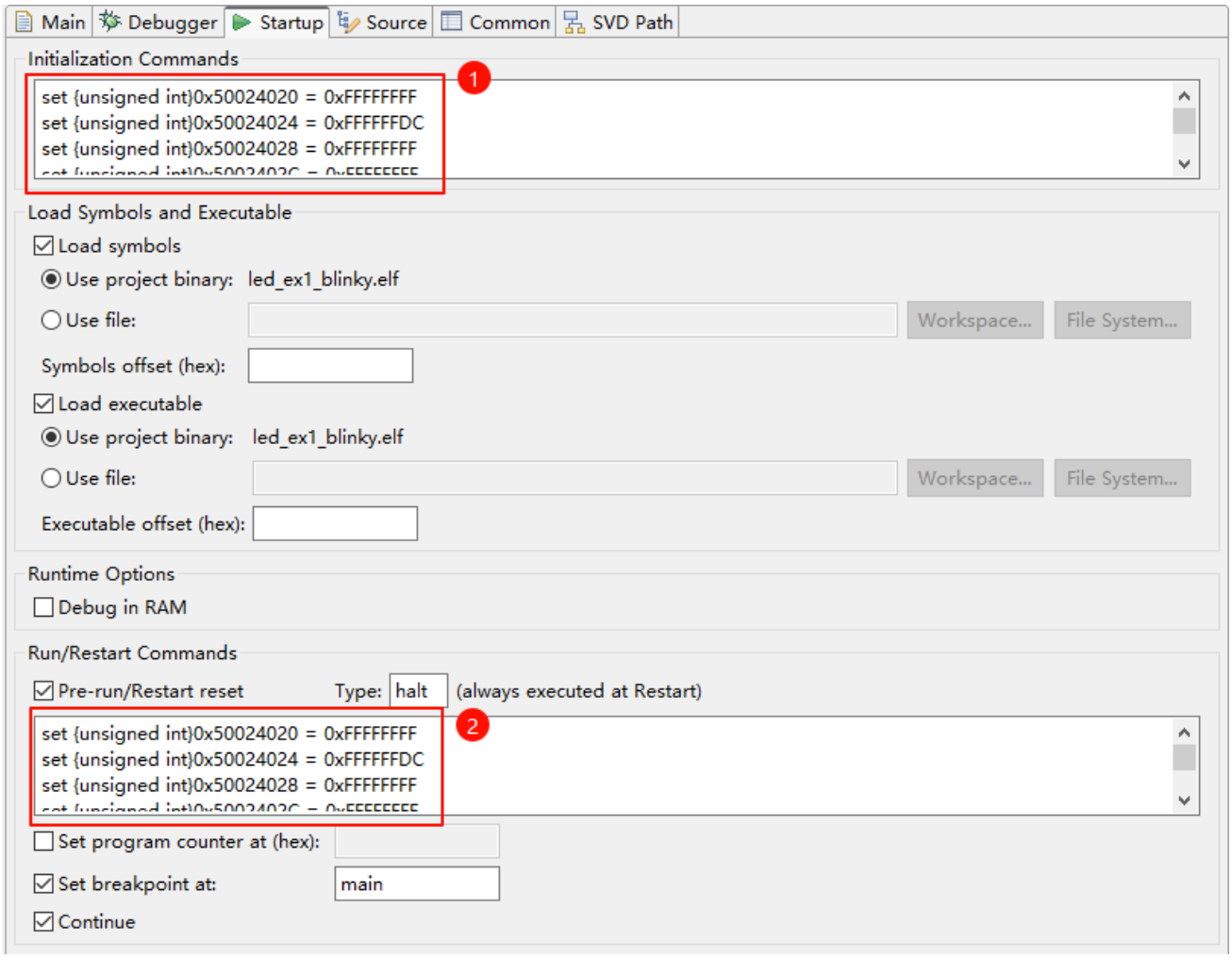
With **factory-default keys** and a boot address / vector table matching the default configuration, you **may leave Initialization Commands** and **Run/Restart Commands** empty.

To change keys or the boot address, prefer maintaining “pyocd.yaml” + “pyocd_user.py” as in the previous section. Only when a user script cannot be used, add DCS KEY commands in both Startup fields (must match “pyocd_user.py” and the chip):

```
set {unsigned int}0x50024020 = 0xFFFFFFFF
set {unsigned int}0x50024024 = 0xFFFFFFDC
set {unsigned int}0x50024028 = 0xFFFFFFFF
```

```
set {unsigned int}0x5002402C = 0xFFFFFFFF
set {unsigned int}0x500240A0 = 0xFFFFFFFF
set {unsigned int}0x500240A4 = 0xFFFEDFFF
set {unsigned int}0x500240A8 = 0xFFFFFFFF
set {unsigned int}0x500240AC = 0xFFFFFFFF
set $sp = *(unsigned int*)0x08000000
set $pc = *(unsigned int*)0x08000004
set $xpsr = $xpsr | (1<<24)
```

Figure 67 Configure Startup



Finally, click **Apply** at the bottom right.

7.7.2 Dual-core Debug configuration

For dual-core debug, see AN1128 *G32R501 Dual-Core Debug Guide*.

7.7.3 Program fails to run after exiting debug on Ubuntu

7.7.3.1 Cause

On Ubuntu, Eclipse may terminate the current thread before “arm-none-eabi-gdb” sends “resuming core 0 [cortex_m]”, so the chip does not reset correctly.

7.7.3.2 Solution

Use the terminal pyOCD gdbserver approach (similar to dual-core debug configuration).

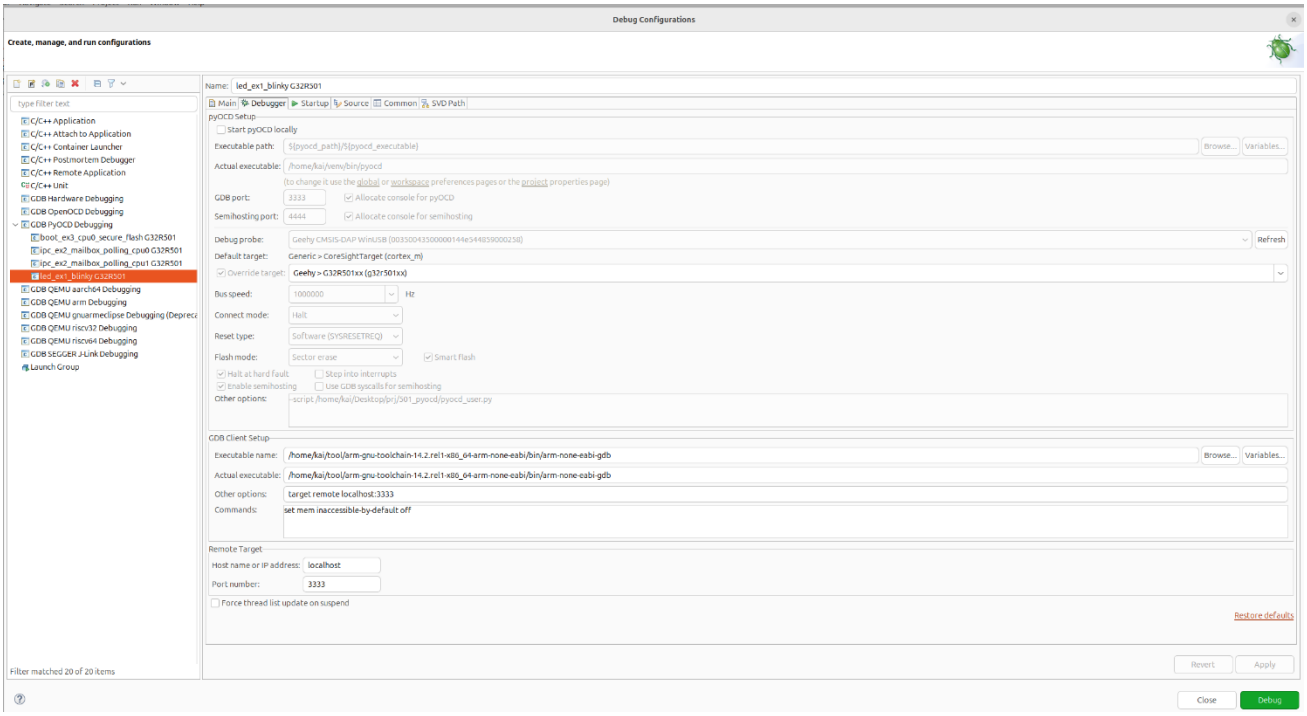
Start pyOCD gdbserver in a terminal:

Figure 68 Start pyOCD gdbserver

```
(venv) kai@Ubuntu:~/Desktop/prj/501_pyocd$ pyocd gdbserver
0001289 I Patched target_G32R501xx DECRYPT_KEYS via pyocd_user.py! [pyocd_user]
0001327 I Target type is g32r501xx [board]
0001366 I DP IDR = 0x6ba02477 (v2 rev6) [dap]
0001383 I AHB-AP#0 IDR = 0x84770001 (AHB-AP var0 rev8) [discovery]
0001389 I AHB-AP#1 IDR = 0x84770001 (AHB-AP var0 rev8) [discovery]
0001395 I AHB-AP#2 IDR = 0x84770001 (AHB-AP var0 rev8) [discovery]
0001401 I AHB-AP#0 Class 0x1 ROM table #0 @ 0xe00ff000 (designer=a75:Arm China part=4d2) [rom_table]
0001406 I [0]<e000e000:SCS Star-MC2 class=9 designer=a75:Arm China part=d24 devtype=00 archid=2a04 devid=0:0:0> [rom_table]
0001409 I [1]<e0001000:DWT Star-MC2 class=9 designer=a75:Arm China part=d24 devtype=00 archid=1a02 devid=0:0:0> [rom_table]
0001411 I [2]<e0002000:BPU Star-MC2 class=9 designer=a75:Arm China part=d24 devtype=00 archid=1a03 devid=0:0:0> [rom_table]
0001414 I [3]<e0000000:ITM Star-MC2 class=9 designer=a75:Arm China part=d24 devtype=43 archid=1a01 devid=0:0:0> [rom_table]
0001417 I [5]<e0041000:ETM Star-MC2 class=9 designer=a75:Arm China part=d24 devtype=13 archid=4a13 devid=0:0:0> [rom_table]
0001422 I [6]<e0003000:??? class=9 designer=a75:Arm China part=d24 devtype=16 archid=0a06 devid=0:0:0> [rom_table]
0001425 I [7]<e0042000:CTI Star-MC2 class=9 designer=a75:Arm China part=d24 devtype=14 archid=1a14 devid=40800:0:0> [rom_table]
0001428 I [8]<e0046000:PMC-100 class=9 designer=43b:Arm part=9ba devtype=55 archid=0a55 devid=145509d6:c105c04:0> [rom_table]
0001436 I AHB-AP#1 Class 0x1 ROM table #0 @ 0xe00ff000 (designer=a75:Arm China part=4d2) [rom_table]
0001442 I [0]<e000e000:SCS Star-MC2 class=9 designer=a75:Arm China part=d24 devtype=00 archid=2a04 devid=0:0:0> [rom_table]
0001447 I [1]<e0001000:DWT Star-MC2 class=9 designer=a75:Arm China part=d24 devtype=00 archid=1a02 devid=0:0:0> [rom_table]
0001450 I [2]<e0002000:BPU Star-MC2 class=9 designer=a75:Arm China part=d24 devtype=00 archid=1a03 devid=0:0:0> [rom_table]
0001455 I [3]<e0000000:ITM Star-MC2 class=9 designer=a75:Arm China part=d24 devtype=43 archid=1a01 devid=0:0:0> [rom_table]
0001459 I [5]<e0041000:ETM Star-MC2 class=9 designer=a75:Arm China part=d24 devtype=13 archid=4a13 devid=0:0:0> [rom_table]
0001464 I [6]<e0003000:??? class=9 designer=a75:Arm China part=d24 devtype=16 archid=0a06 devid=0:0:0> [rom_table]
0001467 I [7]<e0042000:CTI Star-MC2 class=9 designer=a75:Arm China part=d24 devtype=14 archid=1a14 devid=40800:0:0> [rom_table]
0001470 I [8]<e0046000:PMC-100 class=9 designer=43b:Arm part=9ba devtype=55 archid=0a55 devid=145509d6:c105c04:0> [rom_table]
0001478 I CPU core #0: Star-MC2 r0p1, v7.0-M architecture [cortex_m]
0001478 I   Extensions: [DSP, FPU, FPU_DP, FPU_V5, MPU] [cortex_m]
0001478 I   FPU present: FPv5-D16-M [cortex_m]
0001479 I core is created and initialized. [target_G32R501xx]
0001484 I 4 hardware watchpoints [dwt]
0001486 I 8 hardware breakpoints, 1 literal comparators [fpb]
0001493 I 4 hardware watchpoints [dwt]
0001495 I 8 hardware breakpoints, 1 literal comparators [fpb]
r501 connect in did_connect...
0001550 I Semihost server started on port 4444 (core 0) [server]
0001569 I GDB server started on port 3333 (core 0) [gdbserver]
```

Configure the Eclipse Debugger tab:

Figure 69 Eclipse Debugger



Note: The directory used to start pyOCD gdbserver in the terminal must contain the single-core “pyocd.yaml” file.

8 Revision

Table 4 Document revision

Date	Version	Description
2025.1	1.0	● Initial release
2025.4	1.1	● New chapters added
2026.8	1.2	● Applicable products listed as G32R501 and G32R502 ● Keil project creation adds microcontroller selection screenshot ● Program Debug reorganized into J-Link / GEEHY LINK subsections (IAR and Eclipse likewise) ● Added J-Link Devices and local pyOCD registration steps ● G32R502 requires IAR 9.70.4 or later ● FLM taken from SDK package/FLM (including Keil/J-Link file lists) ● Eclipse pyOCD adds yaml and command-line examples ● Factory-default keys can omit long Startup commands ● Added Eclipse global tool paths (Clang / Make / GDB) ● Refreshed IDE/debug screenshots to match current tool chains ● Corrected Startup SP/PC dereference from vector table ● Added Eclipse system environment variable screenshot

Statement

This document is formulated and published by Geehy Semiconductor Co., Ltd. (hereinafter referred to as “Geehy”). The contents in this document are protected by laws and regulations of trademark, copyright and software copyright. Geehy reserves the right to make corrections and modifications to this document at any time. Read this document carefully before using Geehy products. Once you use the Geehy product, it means that you (hereinafter referred to as the “users”) have known and accepted all the contents of this document. Users shall use the Geehy product in accordance with relevant laws and regulations and the requirements of this document.

1. Ownership

This document can only be used in connection with the corresponding chip products or software products provided by Geehy. Without the prior permission of Geehy, no unit or individual may copy, transcribe, modify, edit or disseminate all or part of the contents of this document for any reason or in any form.

The “极海” or “Geehy” words or graphics with “®” or “TM” in this document are trademarks of Geehy. Other product or service names displayed on Geehy products are the property of their respective owners.

2. No Intellectual Property License

Geehy owns all rights, ownership and intellectual property rights involved in this document.

Geehy shall not be deemed to grant the license or right of any intellectual property to users explicitly or implicitly due to the sale or distribution of Geehy products or this document.

If any third party’s products, services or intellectual property are involved in this document, it shall not be deemed that Geehy authorizes users to use the aforesaid third party’s products, services or intellectual property. Any information regarding the application of the product, Geehy

hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party, unless otherwise agreed in sales order or sales contract.

3. Version Update

Users can obtain the latest document of the corresponding models when ordering Geehy products.

If the contents in this document are inconsistent with Geehy products, the agreement in the sales order or the sales contract shall prevail.

4. Information Reliability

The relevant data in this document are obtained from batch test by Geehy Laboratory or cooperative third-party testing organization. However, clerical errors in correction or errors caused by differences in testing environment may occur inevitably. Therefore, users should understand that Geehy does not bear any responsibility for such errors that may occur in this document. The relevant data in this document are only used to guide users as performance parameter reference and do not constitute Geehy's guarantee for any product performance.

Users shall select appropriate Geehy products according to their own needs, and effectively verify and test the applicability of Geehy products to confirm that Geehy products meet their own needs, corresponding standards, safety or other reliability requirements. If losses are caused to users due to user's failure to fully verify and test Geehy products, Geehy will not bear any responsibility.

5. Legality

USERS SHALL ABIDE BY ALL APPLICABLE LOCAL LAWS AND REGULATIONS WHEN USING THIS DOCUMENT AND THE MATCHING GEEHY PRODUCTS. USERS SHALL UNDERSTAND THAT THE PRODUCTS MAY BE RESTRICTED BY THE EXPORT, RE-EXPORT OR OTHER LAWS OF THE COUNTRIES OF THE PRODUCTS SUPPLIERS, GEEHY, GEEHY

DISTRIBUTORS AND USERS. USERS (ON BEHALF OR ITSELF, SUBSIDIARIES AND AFFILIATED ENTERPRISES) SHALL AGREE AND PROMISE TO ABIDE BY ALL APPLICABLE LAWS AND REGULATIONS ON THE EXPORT AND RE-EXPORT OF GEEHY PRODUCTS AND/OR TECHNOLOGIES AND DIRECT PRODUCTS.

6. Disclaimer of Warranty

THIS DOCUMENT IS PROVIDED BY GEEHY "AS IS" AND THERE IS NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, TO THE EXTENT PERMITTED BY APPLICABLE LAW.

GEEHY'S PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED FOR USE AS CRITICAL COMPONENTS IN MILITARY, LIFE-SUPPORT, POLLUTION CONTROL, OR HAZARDOUS SUBSTANCES MANAGEMENT SYSTEMS, NOR WHERE FAILURE COULD RESULT IN INJURY, DEATH, PROPERTY OR ENVIRONMENTAL DAMAGE.

IF THE PRODUCT IS NOT LABELED AS "AUTOMOTIVE GRADE," IT SHOULD NOT BE CONSIDERED SUITABLE FOR AUTOMOTIVE APPLICATIONS. GEEHY ASSUMES NO LIABILITY FOR THE USE BEYOND ITS SPECIFICATIONS OR GUIDELINES.

THE USER SHOULD ENSURE THAT THE APPLICATION OF THE PRODUCTS COMPLIES WITH ALL RELEVANT STANDARDS, INCLUDING BUT NOT LIMITED TO SAFETY, INFORMATION SECURITY, AND ENVIRONMENTAL REQUIREMENTS. THE USER ASSUMES FULL RESPONSIBILITY FOR THE SELECTION AND USE OF GEEHY PRODUCTS. GEEHY WILL BEAR NO RESPONSIBILITY FOR ANY DISPUTES ARISING FROM THE SUBSEQUENT DESIGN OR USE BY USERS.

7. Limitation of Liability

IN NO EVENT, UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL GEEHY OR ANY OTHER PARTY WHO PROVIDES THE DOCUMENT AND PRODUCTS

"AS IS", BE LIABLE FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, DIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE DOCUMENT AND PRODUCTS (INCLUDING BUT NOT LIMITED TO LOSSES OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY USERS OR THIRD PARTIES). THIS COVERS POTENTIAL DAMAGES TO PERSONAL SAFETY, PROPERTY, OR THE ENVIRONMENT, FOR WHICH GEEHY WILL NOT BE RESPONSIBLE.

8. Scope of Application

The information in this document replaces the information provided in all previous versions of the document.