

KF32 系列

ChipON VSCode Extension 用户使用手册 V1.7

在使用本手册前，请您认真阅读以下使用许可协议。只有在同意以下使用许可协议的情况下，方能使用本手册中介绍的产品。

版权公告

未经上海芯旺微电子技术有限公司书面允许，任何公司、个人不得以任何形式复制本使用手册的全部或部分内容。

重要声明

上海芯旺微电子技术有限公司努力使本手册中提供的信息准确和适用，然而，产品及手册可能包括技术或印刷上的错误。上海芯旺微电子有限公司保留在不事先通知的情况下改变本使用手册全部或部分内容的权力。

目 录

CHIPON VSCode EXTENSION 用户使用手册 V1.7	1
目 录	2
1. 引言	4
1.1 文档说明	4
1.2 软件概述	4
1.3 适用范围	4
1.4 术语与定义	4
2. 环境准备	5
2.1 支持的操作系统	5
2.2 依赖软件及插件	5
3. 安装与卸载	5
3.1 CHIPON VSCode EXTENSION 安装	5
3.2 安装与配置工具链	8
3.3 CHIPON VSCode EXTENSION 卸载	9
4. 界面功能说明	10
5. 项目开发与管理	12
5.1 项目结构	12
5.2 多项目管理	13
5.3 新建项目	15
5.3.1 创建新项目	15
5.3.2 导入项目	16
5.4 项目构建	19
5.4.1 构建配置	19
5.4.2 执行构建	23
5.4.3 清理构建	26
5.5 项目程序下载	27
5.5.1 设备管理	27
5.5.2 下载配置	30
5.5.3 程序下载	33
5.6 在线调试	34
5.6.1 调试配置	34
5.6.2 启动调试	35
5.6.3 调试操作	36

5.6.4 调试断点	40
5.6.5 调试视图	43
5.7 运行时监控变量	53
6. 常见问题与解决方法	56
6.1 安装常见问题与解决方法	56
6.2 项目编译常见问题与解决方法	56
6.3 程序下载常见问题与解决方法	61
6.4 程序调试常见问题与解决方法	61
7. 附录	62
7.1 版本更新记录	62

1. 引言

1.1 文档说明

本文档旨在详细介绍 Visual Studio Code（以下简称 VSCode）集成 ChipON VSCode Extension 后的完整使用流程，包括插件安装配置、界面布局、ChipON KF32 芯片开发相关操作、核心功能、项目开发全流程及常见问题排查。为从事 ChipON KF32 芯片开发的用户提供全面、易懂的使用指导，帮助新手快速上手 ChipON VSCode Extension 集成环境，提升嵌入式项目开发、调试及项目管理的效率；同时为有一定使用基础的用户提供功能参考，充分发挥 VSCode 轻量便捷与 ChipON 专业开发工具的协同优势。

1.2 软件概述

ChipON VSCode Extension 集成环境是基于 VSCode 编辑器，通过安装 ChipON 官方提供的“ChipON Extension Pack”插件包，集成 ChipON 芯片开发所需的编译工具链、调试工具、芯片配置工具等核心组件，形成的嵌入式开发环境。其核心优势在于兼顾 VSCode 的轻量、开源、可扩展特性，以及 ChipON 针对自身芯片的专业开发支持，支持 ChipON KF32 全系列主流嵌入式芯片的代码编辑、编译、烧录、调试等全流程开发，大幅简化开发流程，适配嵌入式开发工程师的使用习惯。

1.3 适用范围

本文档适用于所有使用 ChipON VSCode Extension 集成环境进行嵌入式开发的用户，包括嵌入式开发新手、专业嵌入式工程师等不同群体，涵盖从环境搭建、项目开发到项目交付的全流程操作，无论用户是否有 VSCode 使用经验或 ChipON 芯片开发经验，均可通过本文档快速掌握该开发环境的使用方法。

1.4 术语与定义

- 扩展插件（Extension）：用于增强 VSCode 功能的组件，可实现编程语言支持、主题美化、工具集成等额外功能，由微软官方或第三方开发者提供。
- 工作区（Workspace）：VSCode 中用于管理项目文件的容器，可将多个相关文件、文件夹整合在一起，方便统一编辑、配置和管理，关闭软件后会自动保存工作区状态。
- KF32：ChipON KungFu 系列 32 位 MCU。
- ChipON Extension Pack：ChipON 官方推出的 VSCode 插件集合，是搭建 VSCode ChipON 集成环境的核心组件。
- 工具链（Toolchain）：嵌入式开发中用于代码编译、链接、汇编的工具集合，VSCode

ChipON Extension 集成环境默认集成 LLVM Embedded Toolchain，适配 ChipON KF32 芯片。包含编译、下载、调试、构建等一系列在芯片开发过程中所需要的工具。

- 调试 (Debug)：通过调试工具（如 KF32-Link-A）连接芯片，设置断点、查看变量值、逐步执行代码，排查程序中的错误 (Bug)，是嵌入式开发的核心环节。

2. 环境准备

2.1 支持的操作系统

- Windows 10/11 x64;
- Linux x64

Tips:

目前已经在 Windows10 x64、Linux Ubuntu22.04 x64 版本测试验证

2.2 依赖软件及插件

VSCode ≥ 1.109

VSCode CMake Tools Extension $\geq 1.22.28$

VSCode Intel HEX Format Extension $\geq 1.0.0$

VSCode Motorola S-Record Extension $\geq 0.1.0$

VSCode Memory Inspector Extension $\geq 1.2.0$

VSCode elfPreview Extension $\geq 0.1.3$

3. 安装与卸载

3.1 ChipON VSCode Extension 安装

ChipON VSCode Extension 支持在线安装和离线安装两种方式：

在线安装

打开 VSCode 软件，点击左侧边栏中的插件按钮。在输入框中输入 ChipON。然后会出现 ChipON VSCode Extension 的组件，点击 Install 即可。

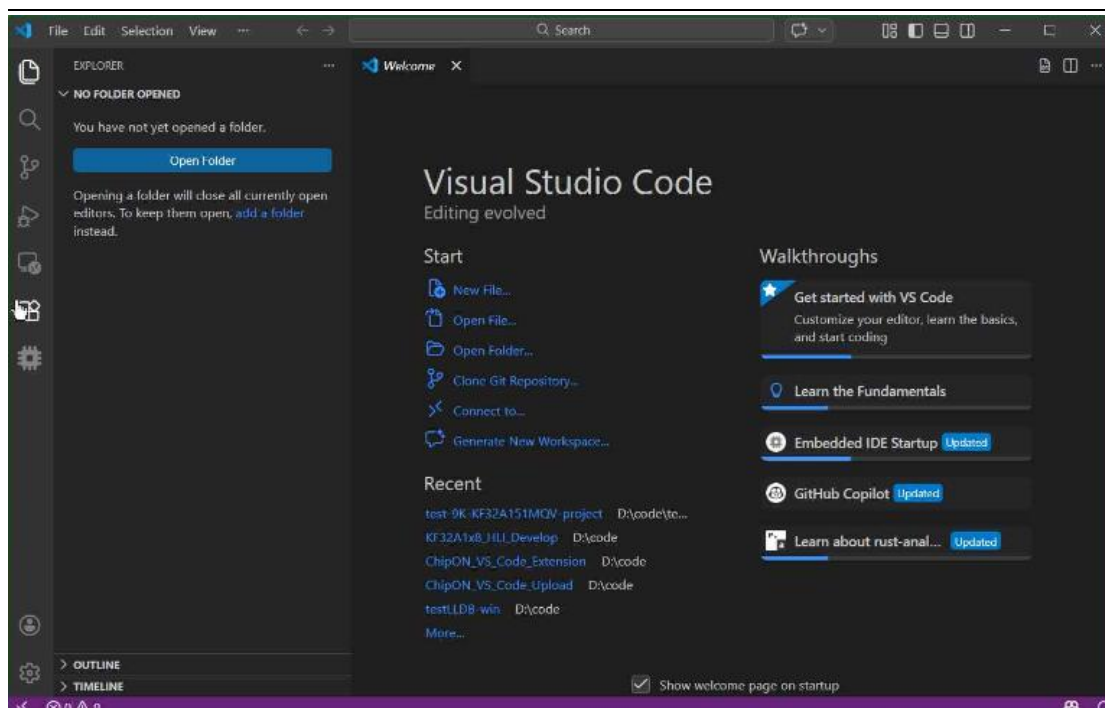


图 3-1 VSCode 初始化界面

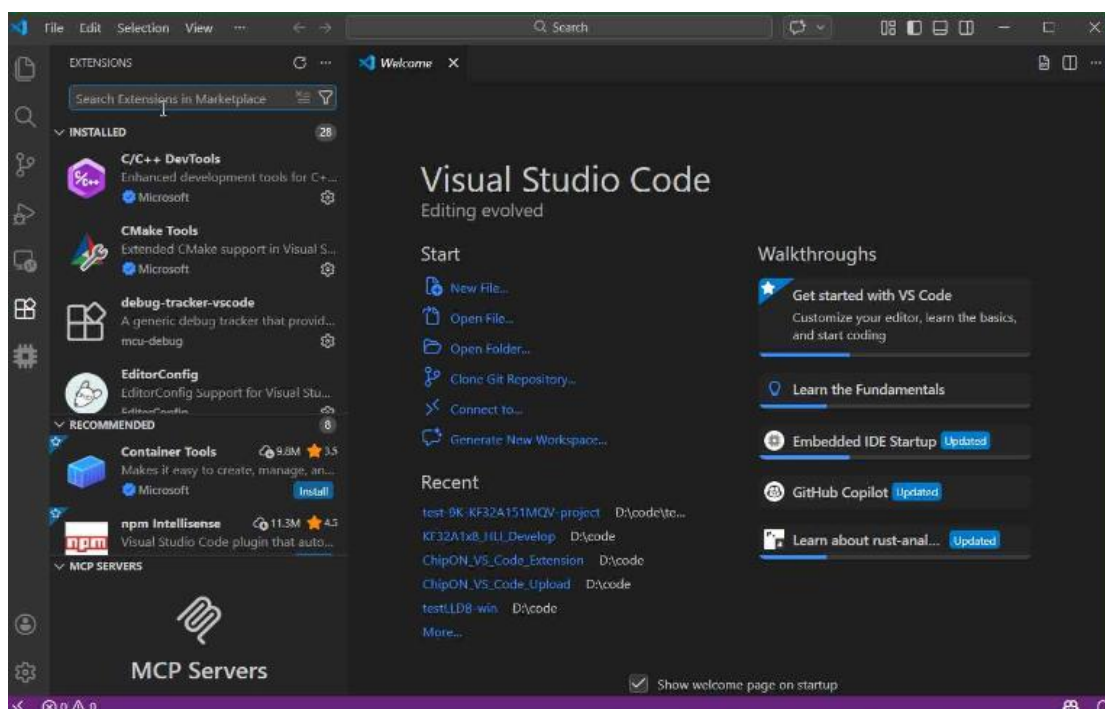


图 3-2 VSCode Extension 界面

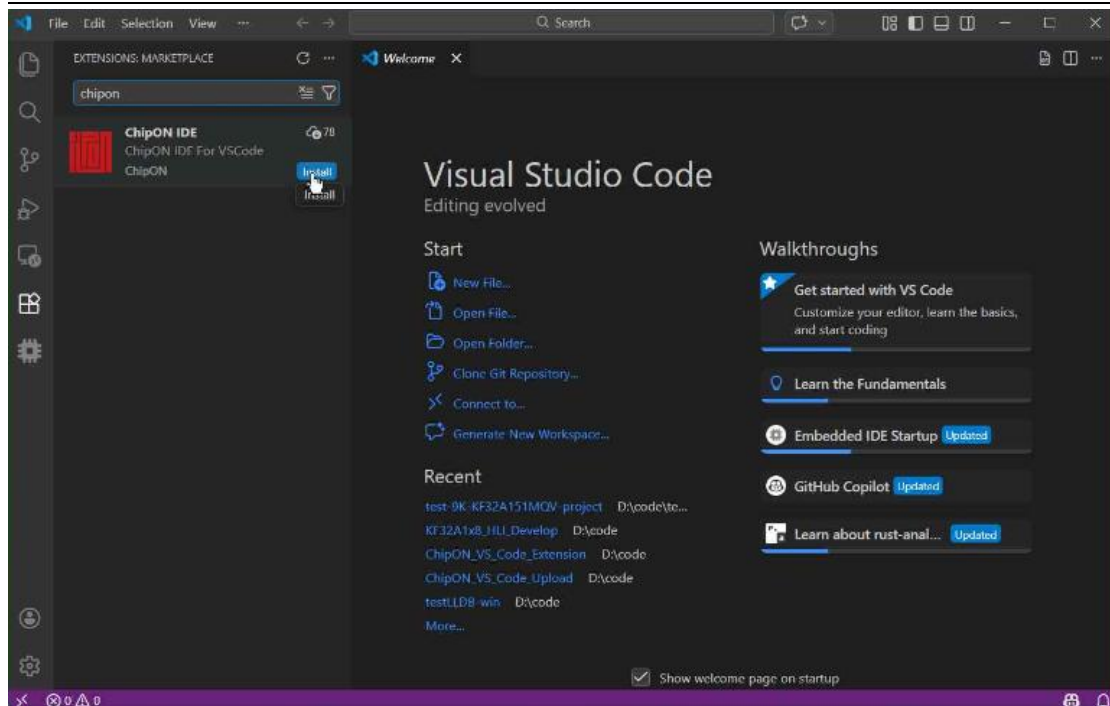


图 3-3 搜索结果界面

离线安装

将对应的插件安装包（如：chipon-ide-0.1.5.vsix）下载到本地后，在 VSCode 中安装插件。点击左侧边栏中的插件 -> 点击右上角的三个点按钮或 Ctrl+Shift+P，输入 Extension: Install From VSIX -> 选择最后一个按钮 Install From VSIX。在打开的文件窗口中选择下载下来的 VSIX 即可。

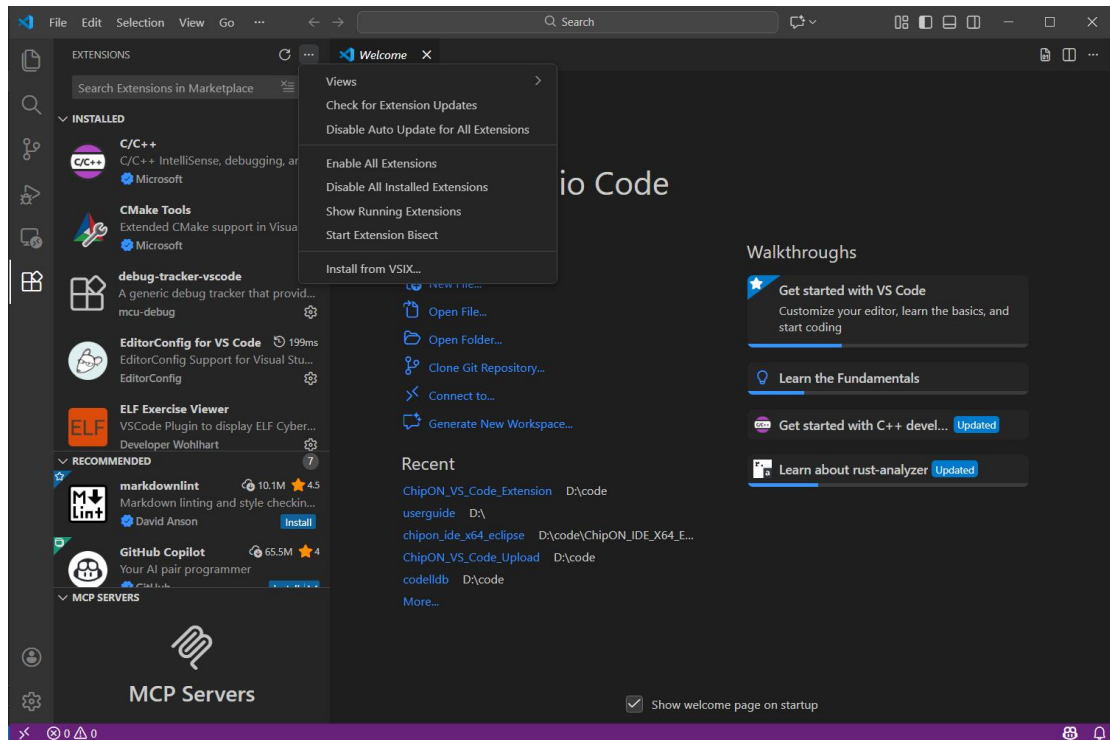


图 3-4 离线安装界面

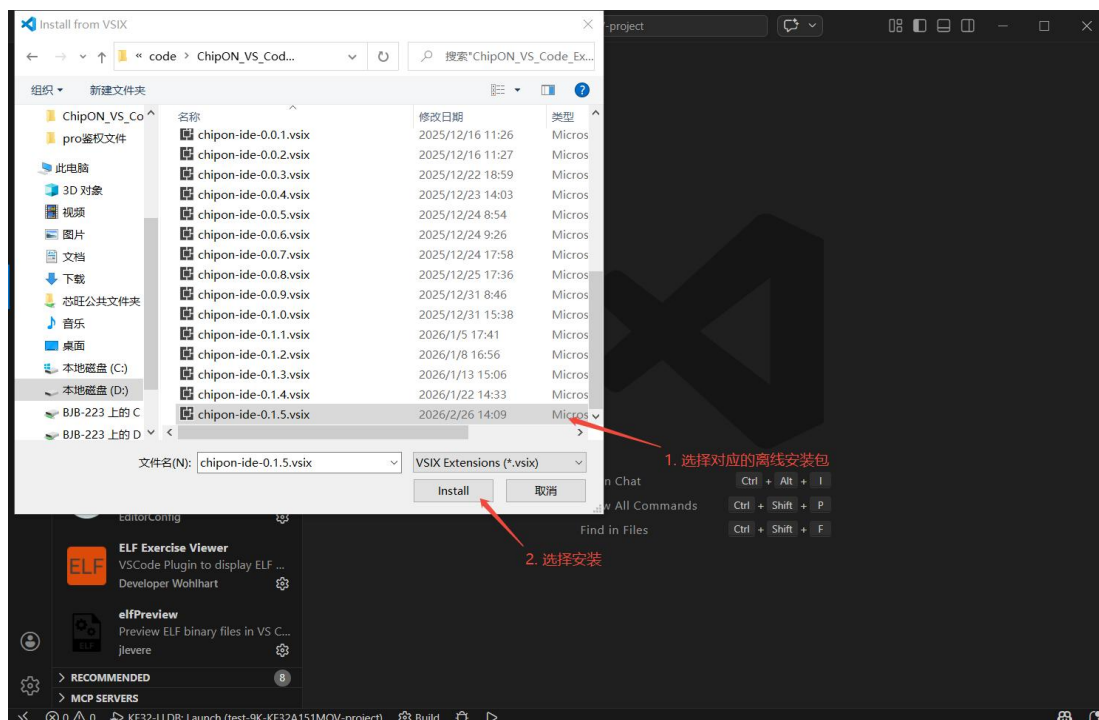


图 3-5 安装操作

3.2 安装与配置工具链

将 ToolChain 安装包下载到本地之后，使用解压软件解压到电脑上的某一个地址下。在解压完成之后，需要在 ChipON VSCode Extension 中配置对应的参数才可以在 Extension 中使用。在 3.1 节安装完成之后，会出现提示“未配置: chipon.toolchain. path ...”，点击提示框右下角的“去设置”按钮，或通过点击 QUICK ACCESS 栏中的“Toolchain Path Configuration”按钮，进入到设置界面进行设置。将 Toolchain 的根目录地址配置到 chipon.toolchain.path 参数中即可。

Tips:

配置到 chipon.toolchain.path 的目录地址，需要是工具链安装包的根目录，即目录地址的子目录就是编译器、调试器等工具的目录。请确保目录地址不包含中文或空格。

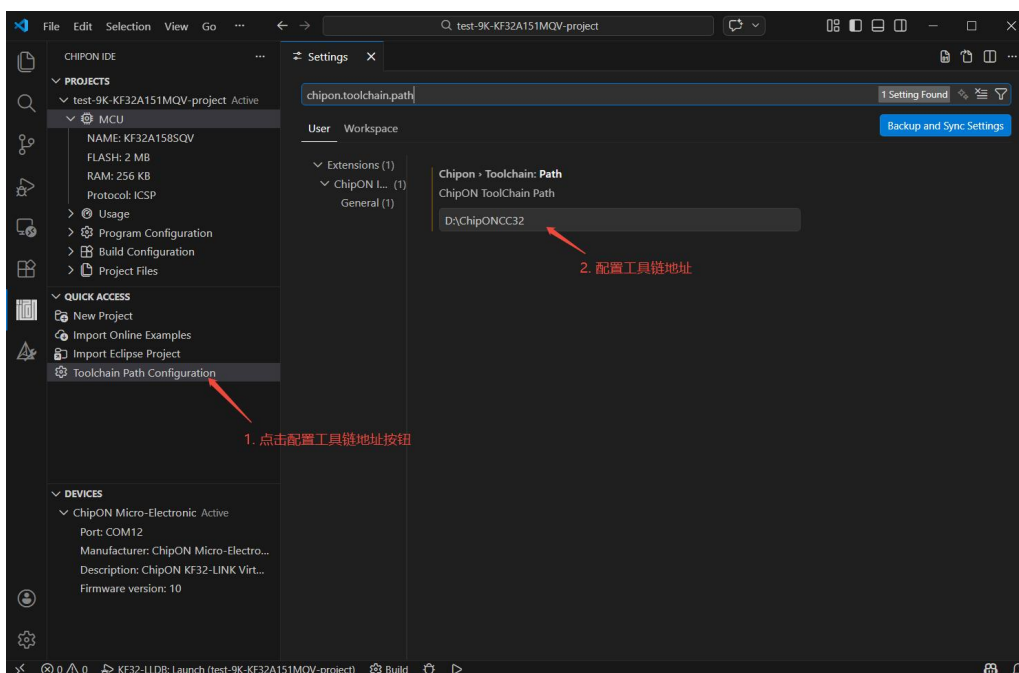


图 3-6 ToolChain 配置地址

3.3 ChipON VSCode Extension 卸载

点击 VSCode 软件左侧边栏，进入到插件界面中，定位到 ChipON VSCode Extension 插件右击，选择 Uninstall，即可将 ChipON VSCode Extension 以及其对应的组件全部卸载。ToolChain 相关的工具包，只需要直接将 ToolChain 所在的根目录删除即可将工具包及其环境全部删除。

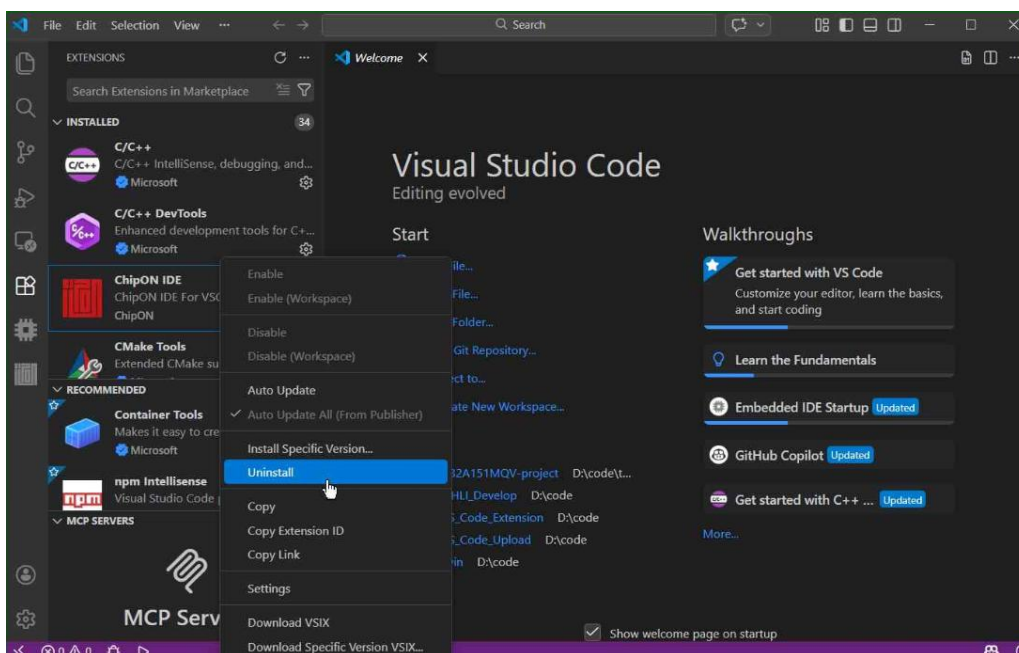


图 3-7 卸载插件

4. 界面功能说明

ChipON VSCode Extension 在运行时的界面如下图所示。

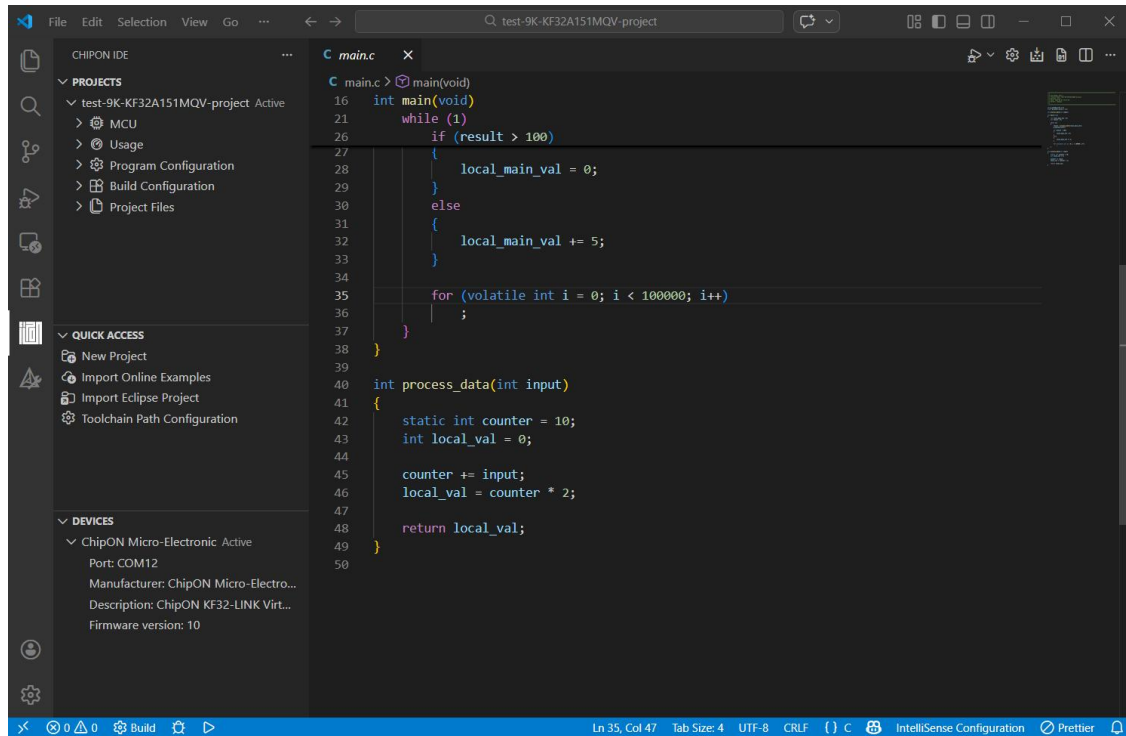


图 4-1 ChipON VSCode Extension 运行时界面

在 Activity Bar 中有 ChipON 图标的按钮, 点击之后, 会进入到 ChipON VSCode Extension 界面。该界面包含以下三个主要区域:

- **PROJECTS:** 展示当前工作区中已打开的项目及其结构, 包括项目名称、MCU 信息、使用统计、程序配置、构建配置以及项目文件列表。
- **QUICK ACCESS:** 提供常用功能的快捷入口, 如新建项目、导入在线示例、导入 Eclipse 项目及配置工具链路径。
- **DEVICES:** 显示当前连接的调试/编程设备信息, 包括端口号、制造商、设备描述和固件版本。

PROJECTS 栏用于显示当前工作区中所有已打开的 ChipON 项目。每个项目在列表中独立展示, 并包含以下二级信息项: MCU、Usage、Program Configuration、Build Configuration、Project Files。当某个项目条目获得焦点 (如被鼠标选中) 时, 其右侧会依次出现**编译**、**编程**、**调试**三个操作按钮, 方便用户快速执行常用任务。

各信息项的具体说明如下:

信息项	说明
MCU	显示芯片型号、FLASH 大小、RAM 大小及通信协议 (Protocol)。
Usage	显示编译后程序对芯片资源的占用率。
Program Configuration	显示简要的编程配置信息。
Build Configuration	显示简要的编译配置信息。

Project Files	以树形结构展示当前项目的文件组成。
---------------	-------------------

在 PROJECTS 栏下的 MCU 信息栏中，显示当前芯片的基本信息，包括芯片型号、FLASH 大小、RAM 大小及协议类型（Protocol）。如需更改芯片型号，请按照以下步骤操作：

1. 在 MCU 信息栏中找到芯片型号右侧的修改按钮，单击该按钮。
2. Extension 将弹出芯片型号选择对话框，从列表中选择所需的芯片型号。
3. 确认选择后，芯片型号自动更新，对应的 FLASH 大小、RAM 大小等信息也会同步调整。

Tips:

建议始终通过上述界面修改芯片型号，以确保相关配置文件（如 .vscode/chipon.json）中的型号信息自动同步。手动编辑配置文件可能导致配置不一致，从而引发编译或调试异常。

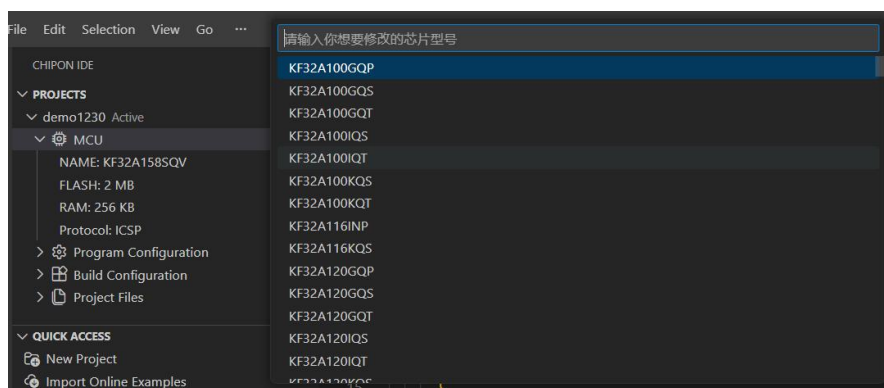


图 4-2 修改芯片型号

每个项目目录下均包含 Usage 信息栏，用于显示编译后程序对芯片内存资源的占用情况，包括：FLASH 占用大小、RAM 占用大小。

Usage 信息在以下两种情况下会更新：

1. 点击项目栏右侧的**编译**按钮，成功编译后自动刷新。
2. 点击 Usage 栏右侧的**刷新**按钮，手动触发更新。

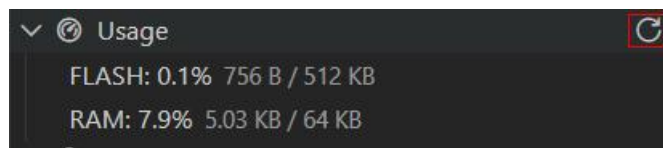


图 4-3 Usage 信息栏界面

DEVICES 栏用于显示当前计算机上已连接的可用调试/编程器（如 KF32-LINK-A）及其基本信息。每个设备在列表中独立展示，并包含以下信息：

信息项	说明
Port	设备占用的通信端口号（如 COM12）。
Manufacturer	设备制造商名称。
Description	设备的描述信息。
Firmware version	设备的固件版本号。

当设备正常连接并被识别时，其名称右侧会显示连接设备的“Active”状态标识。



图 4-4 Devices 栏界面

5. 项目开发与管理

5.1 项目结构

ChipON VSCode Extension 创建的项目典型结构包含以下主要目录与文件：

- `_config` 文件夹：存放项目预编写的 startup 与中断向量源代码文件；
- `.vscode` 文件夹：存放 vscode 相关的设置、KF32 项目相关的信息；
 - `chipon.json`：KF32 项目配置文件；
 - `launch.json`：VSCode 调试配置文件；
 - `settings.json`：VSCode 项目配置文件；
 - `tasks.json`：VSCode 任务配置文件；
- `cmake` 文件夹：存放 KF32 项目编译相关配置信息；
 - `internal` 文件夹：cmake 配置中有关 toolchain、KF32 项目的设置；
 - `CMakeLists.txt`：cmake 配置信息；
 - `CMakePresets.json`：cmake 扩展配置信息；
 - `configPresets.json`：自定义配置信息文件；
 - `Flags.cmake`：预定义配置信息文件；
- `__Kungfu32_chipmodel_define.h`：芯片型号头文件；
- `kf_it.c`：中断函数 c 文件；
- `main.c`：入口主文件；
- `xxxxx.ld`：对应芯片型号的链接脚本。

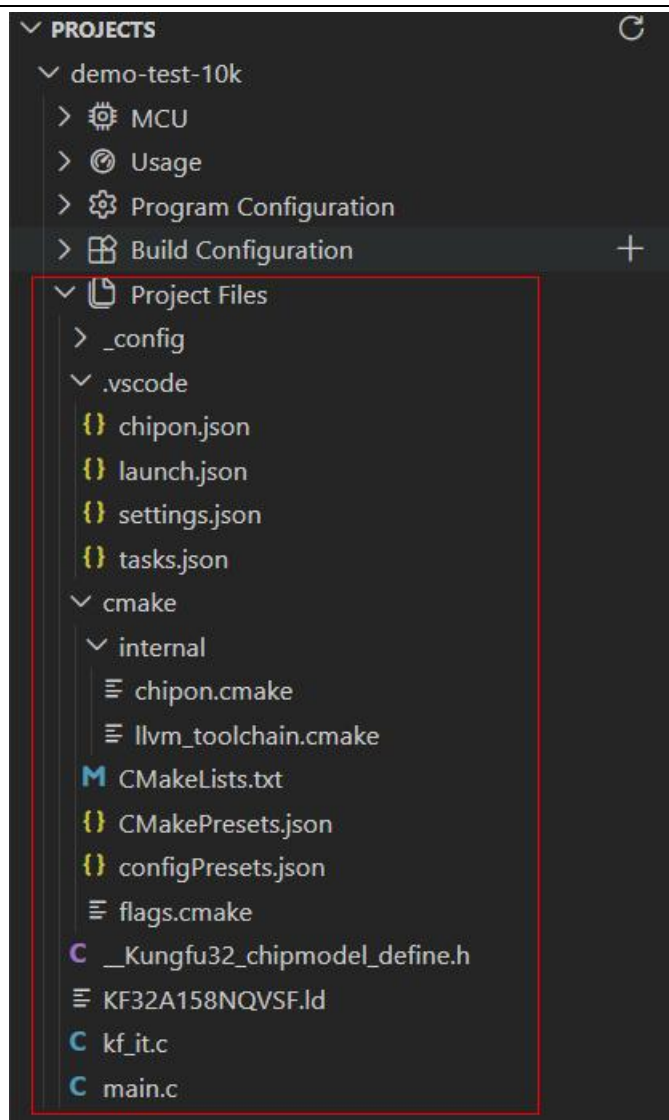


图 5-1 项目典型结构

5.2 多项目管理

ChipON VSCode Extension 支持在同一界面中同时显示、操作和管理多个位于不同路径的项目。以下步骤支持将其他位置的项目添加到当前工作区：

1. 在资源管理器 (Explorer) 视图中，单击鼠标右键，选择 “Add Folder to Workspace” (添加文件夹到工作区)。
2. 在弹出的对话框中，浏览并选中需要导入的项目文件夹，然后单击 “添加” 确认。

完成添加后，PROJECTS 栏中将列出所有已导入的 ChipON 项目。每个项目均可独立进行配置和操作（如编译、编程、调试等），项目之间的设置互不影响。

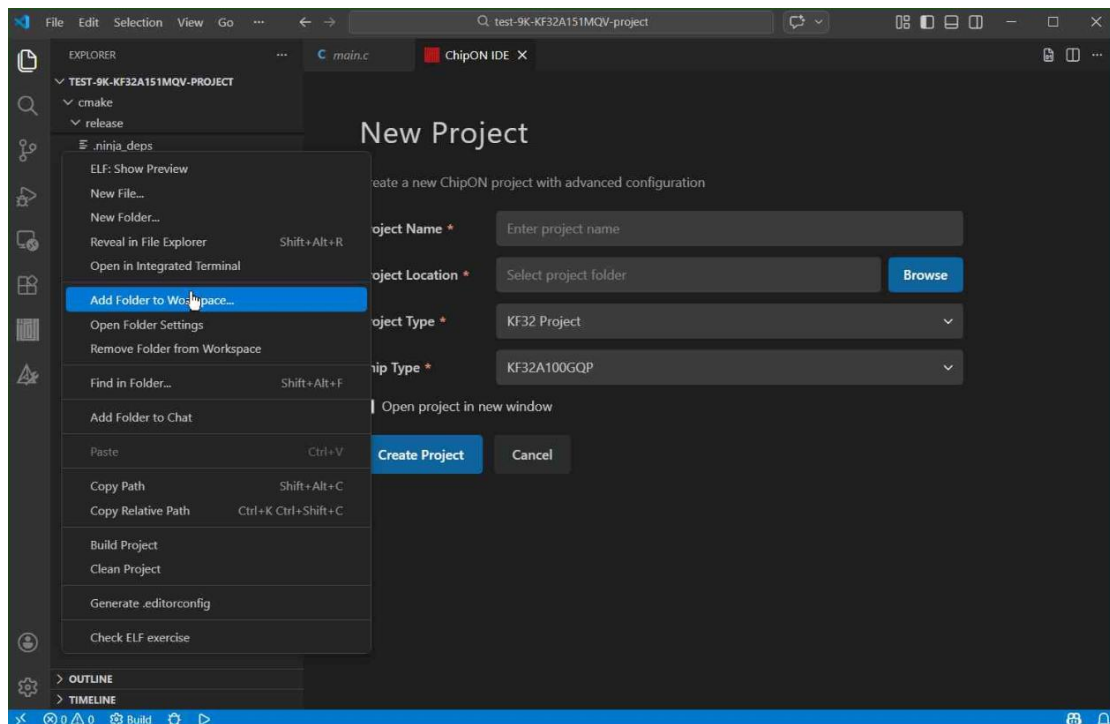


图 5-2 Add Folder to Workspace 操作

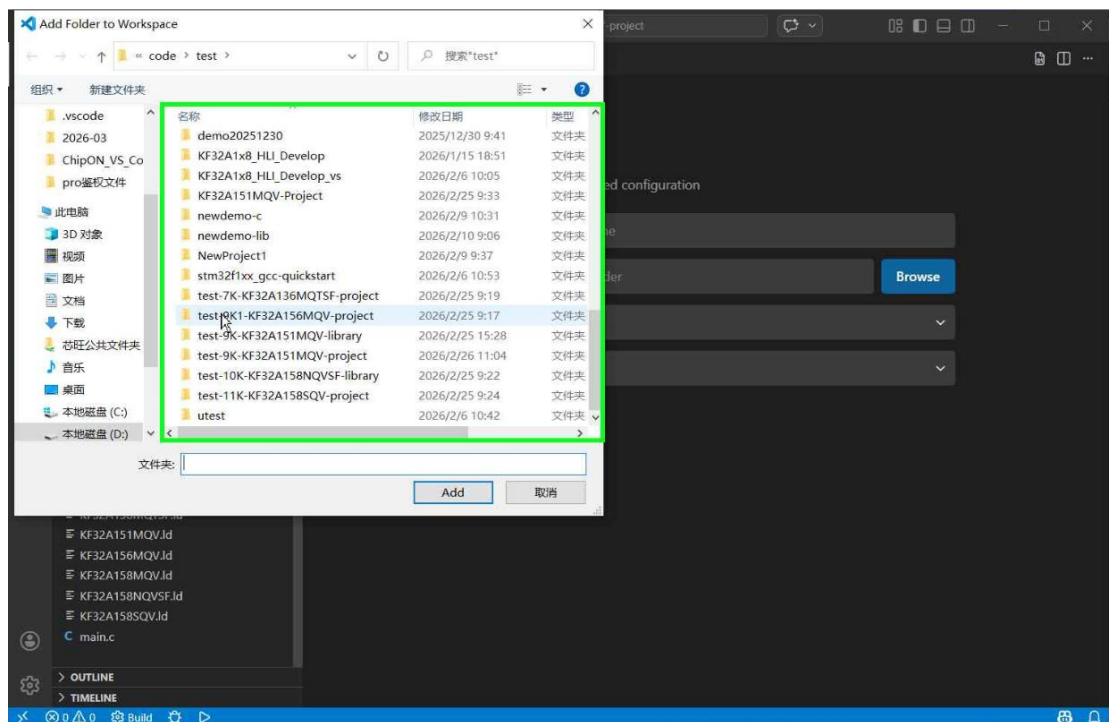


图 5-3 选择其他的项目

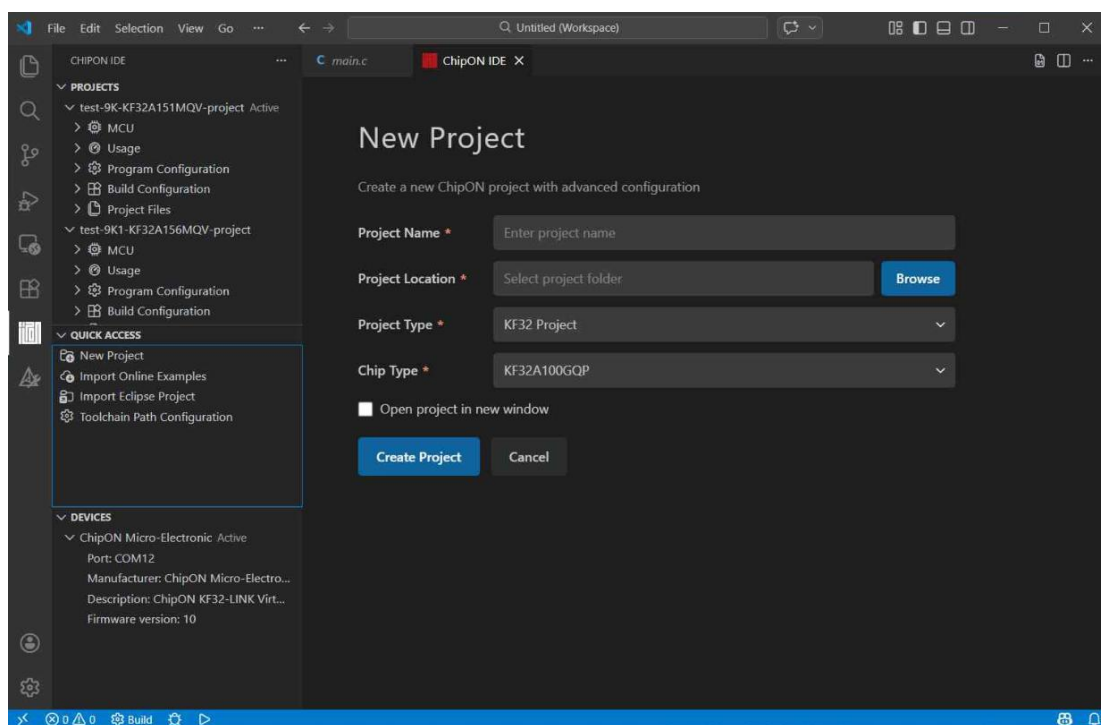


图 5-4 多项目管理功能界面

5.3 新建项目

5.3.1 创建新项目

用户可以通过 QUICK ACCESS 栏快速创建一个新的 ChipON 项目，具体步骤如下：

1. 在左侧 QUICK ACCESS 栏中，单击 New Project 按钮。
2. Editor 区域将打开“New Project”页面。按照页面提示依次填写以下信息：
 - Project Name: 项目名称。
 - Project Location: 项目保存路径。支持单击文件夹图标以可视化方式选择目标文件夹。
 - Project Type: 项目类型。支持选择 KF32 Project(标准应用程序项目)或 KF32 Library Project (库项目)。rc 版本开始支持 C++项目，在选择项目类型之后，可以通过勾选项目类型的 C++选项来新建 C++项目。默认生成 C 项目。
 - Chip Type: 芯片型号。支持输入部分关键字进行模糊查询，从下拉列表中选择。
3. 确认信息无误后，单击 Create Project 按钮。

项目创建成功后，系统会自动在指定位置生成项目文件，并在 PROJECTS 栏中显示新项目及其目录结构，即可开始进行配置与开发。

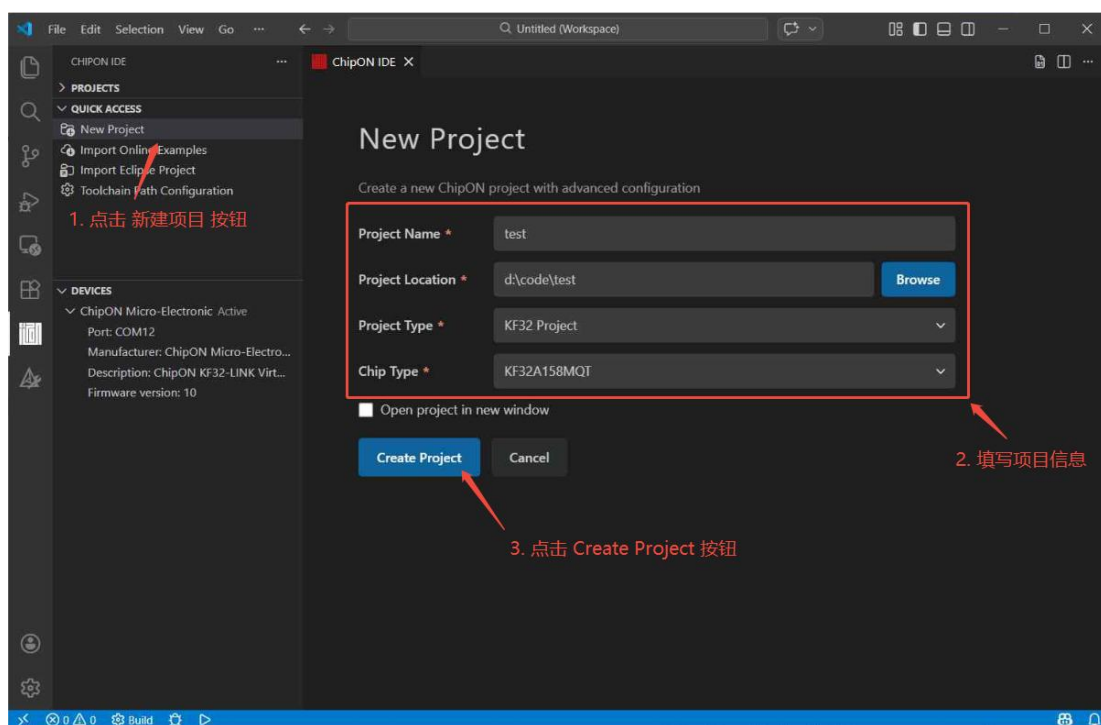


图 5-5 新建项目的过程

5.3.2 导入项目

5.3.2.1 导入 eclipse 项目

ChipON VSCode Extension 支持导入由 ChipON IDE KF32（基于 Eclipse）创建的项目，便于在统一环境中进行管理。导入步骤如下：

1. 在 QUICK ACCESS 栏中，单击 Import Eclipse Project 按钮。系统将在编辑器区域上方弹出输入框，提示输入待导入的 Eclipse 项目路径。
2. 输入源项目所在的完整文件夹路径，然后按回车键确认。系统会再次弹出输入框，要求指定项目导入后的目标存放路径。
3. 输入目标路径（建议使用空文件夹或新建文件夹），按回车键。系统最后一次弹出输入框，用于设置导入后项目的名称。
4. 输入新项目名称，按回车键完成导入。

导入成功后，该项目将自动出现在 PROJECTS 栏中，并保持原有的文件结构和配置，可与其他 ChipON 项目一样进行编译、编程等操作。

Tips:

由于该项目涉及从 GCC 工具链迁移至 LLVM 工具链，用户在迁移后需对编译配置、代码兼容性、及依赖库进行完整检查，确保项目可正常编译与运行。

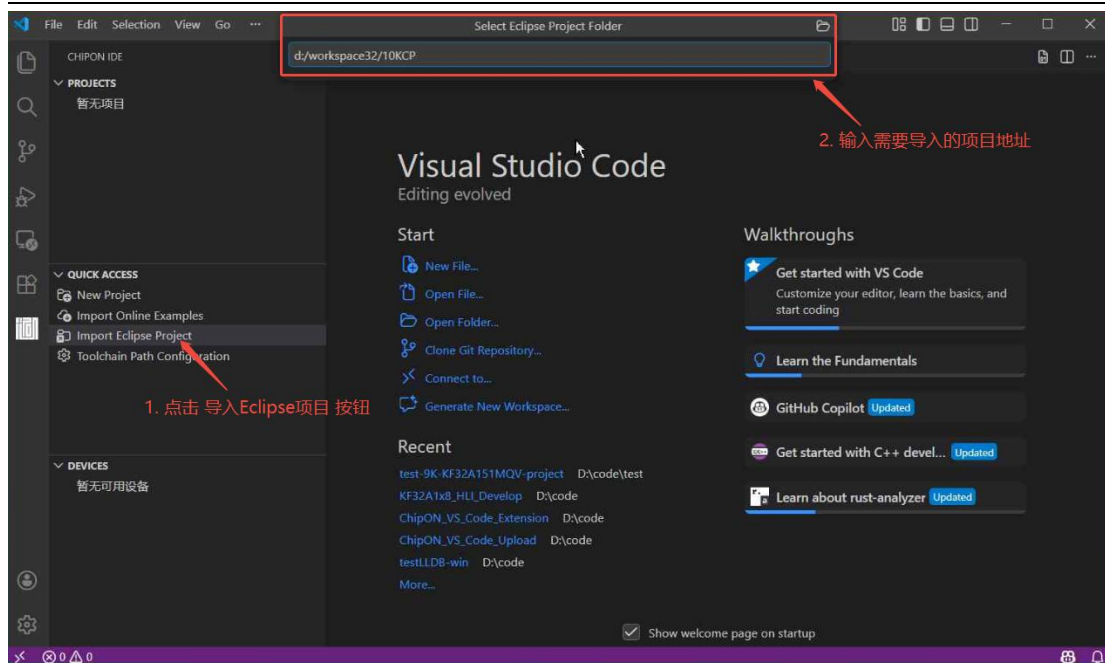


图 5-6 导入 Eclipse Project 过程 1

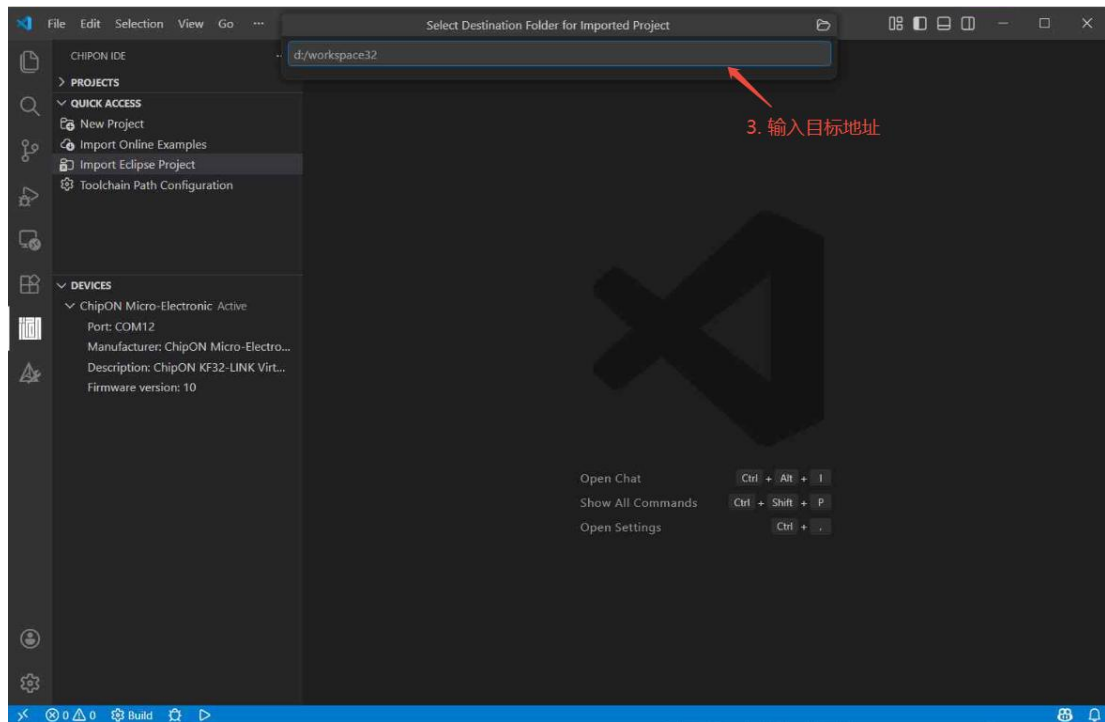


图 5-7 导入 Eclipse Project 过程 2

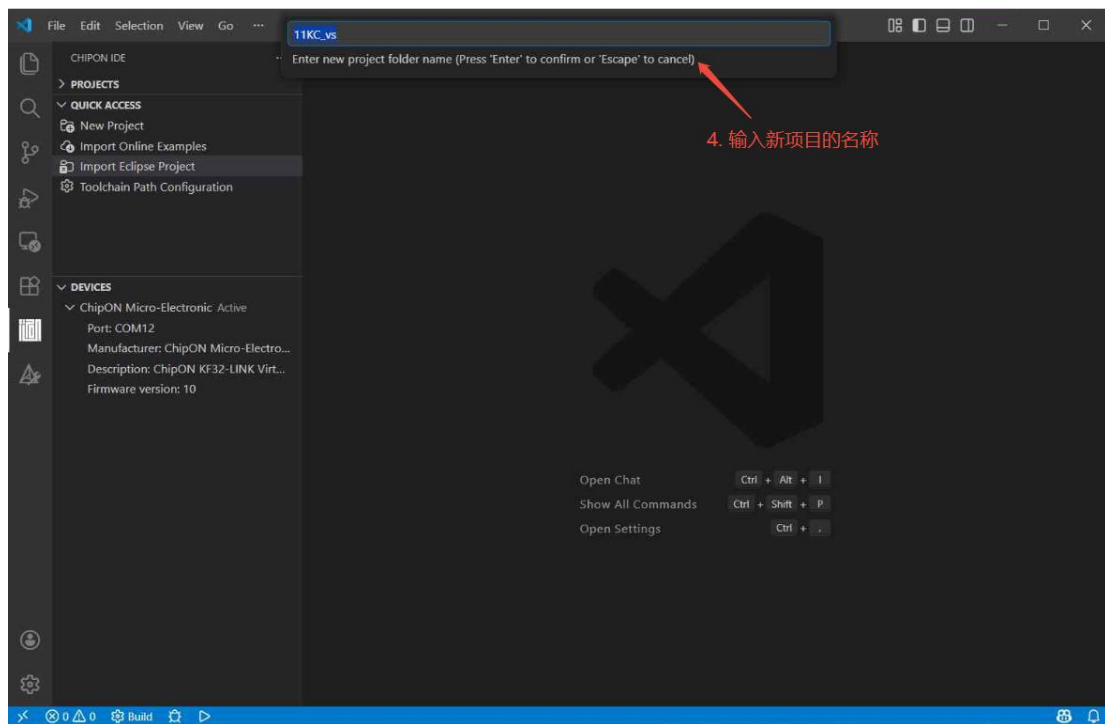


图 5-8 导入 Eclipse Project 过程 3

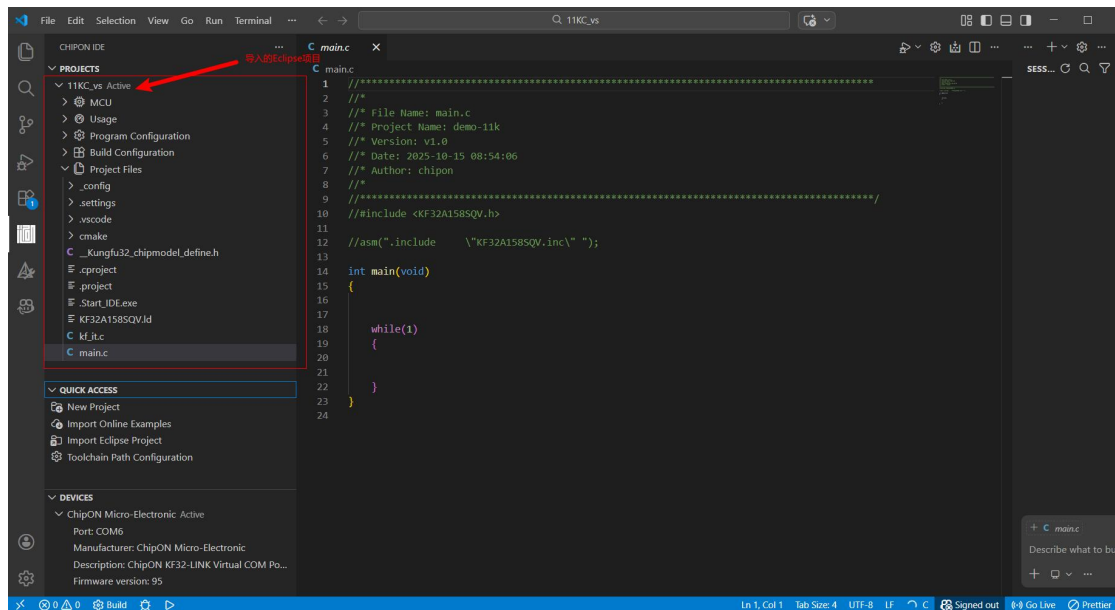


图 5-9 导入 Eclipse Project 结果

5.3.2.2 导入在线样例项目

在联网状态下，您可以直接从 ChipON 示例仓库导入适用于特定芯片型号的示例项目。具体操作步骤如下：

1. 在左侧 QUICK ACCESS 栏中，单击 Import Online Examples 按钮。软件将在编辑区域打开“Import Online Examples”界面。

2. 在界面中依次配置以下参数：
 - 芯片型号：从下拉列表中选择目标芯片型号。
 - 示例项目：选择需要导入的示例项目。
 - 项目名称：输入自定义的项目名称。
 - 项目路径：选择或输入项目存放的目标地址。
 - 确认配置无误后，单击 Import 按钮。
3. 导入完成后，示例项目将自动出现在 PROJECTS 栏中，并保留完整的示例代码与配置，可供直接使用或参考。

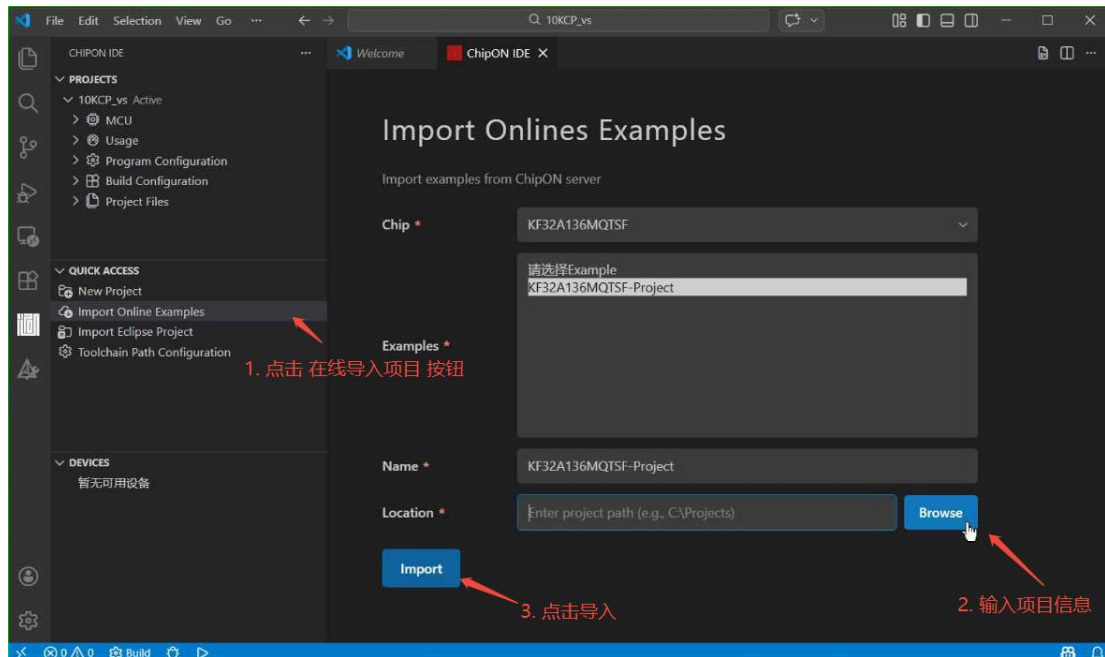


图 5-10 导入在线样例项目的过程

5.4 项目构建

5.4.1 构建配置

5.4.1.1 构建相关配置文件与目录

项目创建后，cmake 目录下会生成一套标准的 CMake 构建脚本：

- cmake/CMakeLists.txt
 - 作用：CMake 工程入口文件。
 - 内容：定义 project()、收集源码文件、添加包含目录、定义编译目标与产物。
 - 修改建议：需要增删源码、头文件目录、链接库或修改链接脚本路径时，请修改此文件。
- cmake/CMakePresets.json
 - 作用：构建配置的存储文件（Build Configuration 的来源）。

- 内容：定义了所有的 `configurePresets` 和 `buildPresets`。
- 修改建议：
 - ◆ 通常建议通过界面提供的 Add/Duplicate/Edit/Delete 菜单进行管理。
 - ◆ 高级用户可手动修改此文件以调整特定配置的 `USER_*_FLAGS`（如 `USER_C_FLAGS`）。
- `cmake/configPresets.json`
 - 作用：定义项目级通用配置（通常被 `CMakePresets.json` 引用）。
 - 内容：包含芯片型号定义（`CONFIG_CHIP_NAME`）和工具链路径（`TOOLCHAIN_HOME`）。
 - 注意：`TOOLCHAIN_HOME` 会根据设置中的 `chipon.toolchain.path` 自动更新，无需手动修改。
- `cmake/flags.cmake`
 - 作用：编译与链接参数的逻辑处理脚本。
 - 原理：该脚本会读取 Preset 中定义的 `USER_C_FLAGS`、`USER_CXX_FLAGS`、`USER_ASM_FLAGS` 和 `USER_LINKER_FLAGS`，并将它们追加到系统默认的 `CMAKE_*_FLAGS` 中。
 - 修改建议：如果需要调整对所有配置都生效的默认编译参数，可修改此文件。
- `.vscode/tasks.json`
 - 插件会自动在该文件中生成标准的 CMake 任务：`CMake: Configure`、`CMake: Build` 和 `CMake: Clean`。
 - 当切换 **Active** 状态配置时，这些任务的 `preset` 字段会自动更新为当前配置名。

5.4.1.2 构建配置管理

用户可以在 Projects 视图的 Build Configuration 节点上进行以下操作：

1. 新增配置 (Add)

点击 Build Configuration 右侧的“+”号 (Add Build Configuration)。输入唯一配置名称。选择构建类型 (Debug / Release)。插件会自动在 `CMakePresets.json` 中生成对应的 `configure` 和 `build preset`。

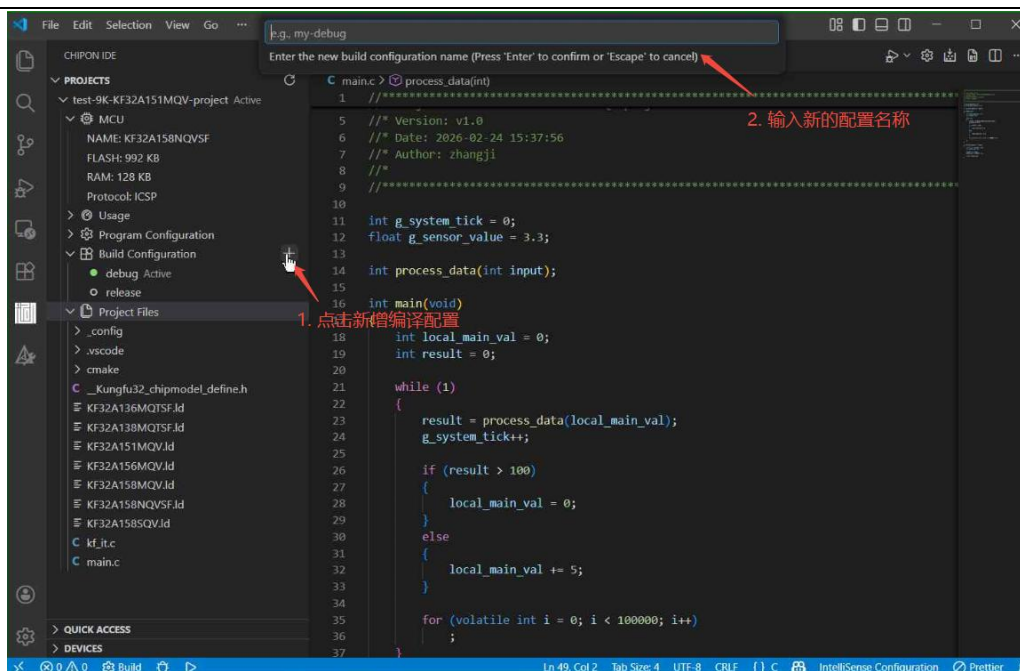


图 5-11 新增编译配置

2. 复制配置 (Duplicate)

在现有配置上点击 Duplicate 图标。插件会自动复制该配置的所有参数，并生成一个带后缀的新名称（如 “debug_1”）。

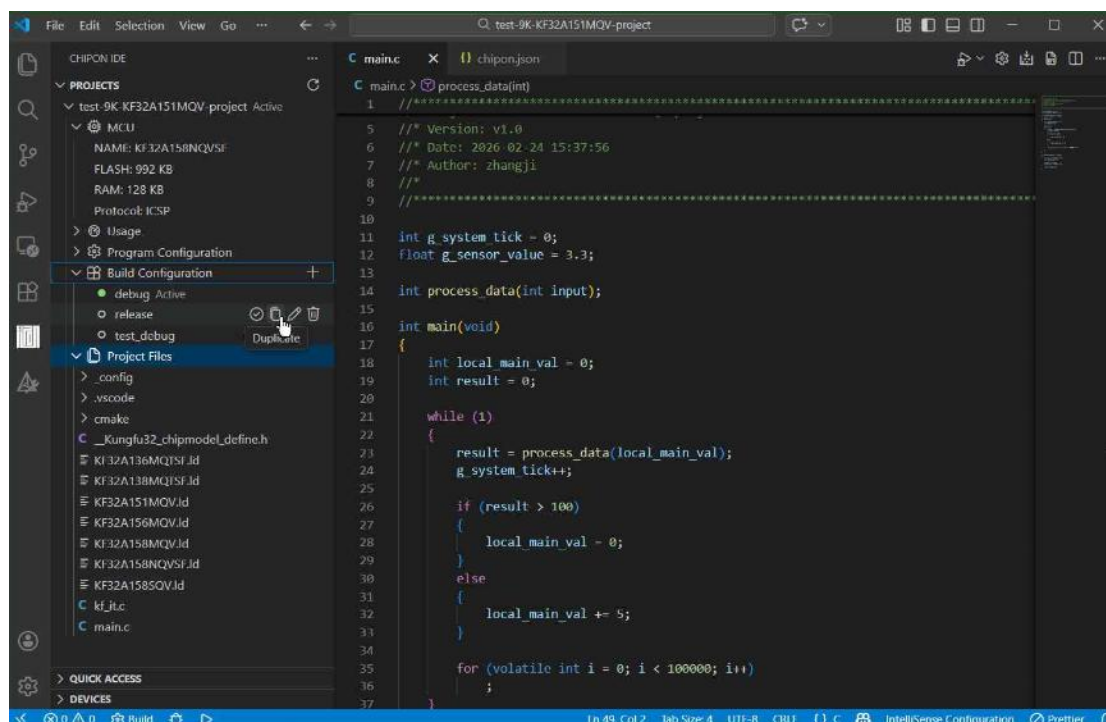


图 5-12 复制编译配置

3. 编辑配置 (Edit)

在配置上点击 Edit 图标。打开可视化编辑页面，允许你修改独立的编译配置：C Flags（C 编译选项）、C++ Flags（C++ 编译选项）、Assembly Flags（汇编选项）、Linker Flags（连接器选项）。

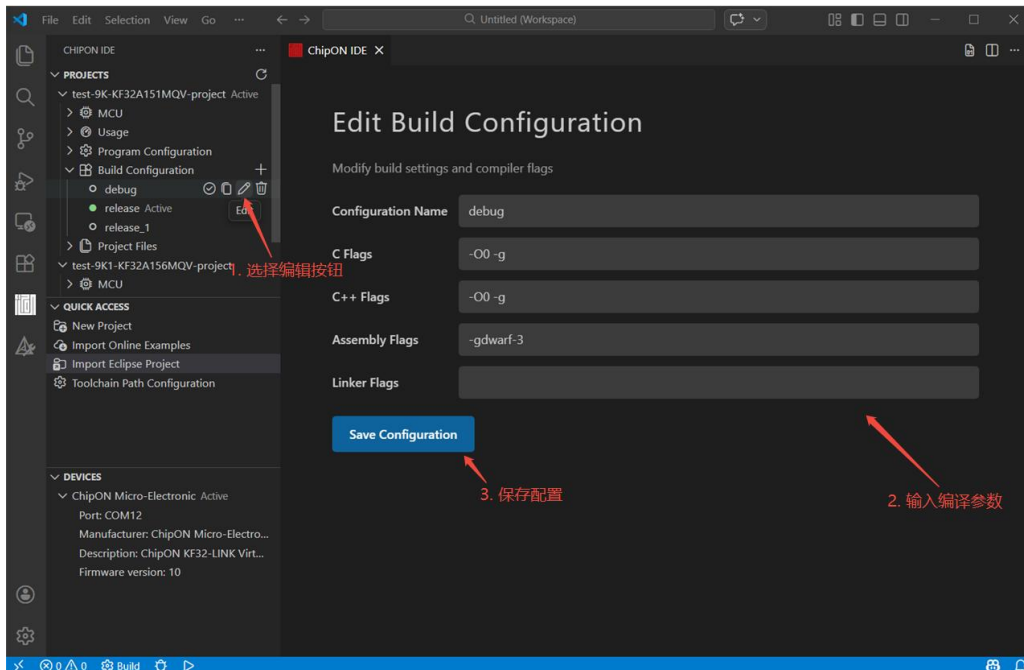


图 5-13 编辑编译配置

4. 删除配置 (Delete)

点击 Delete 图标可删除指定配置。

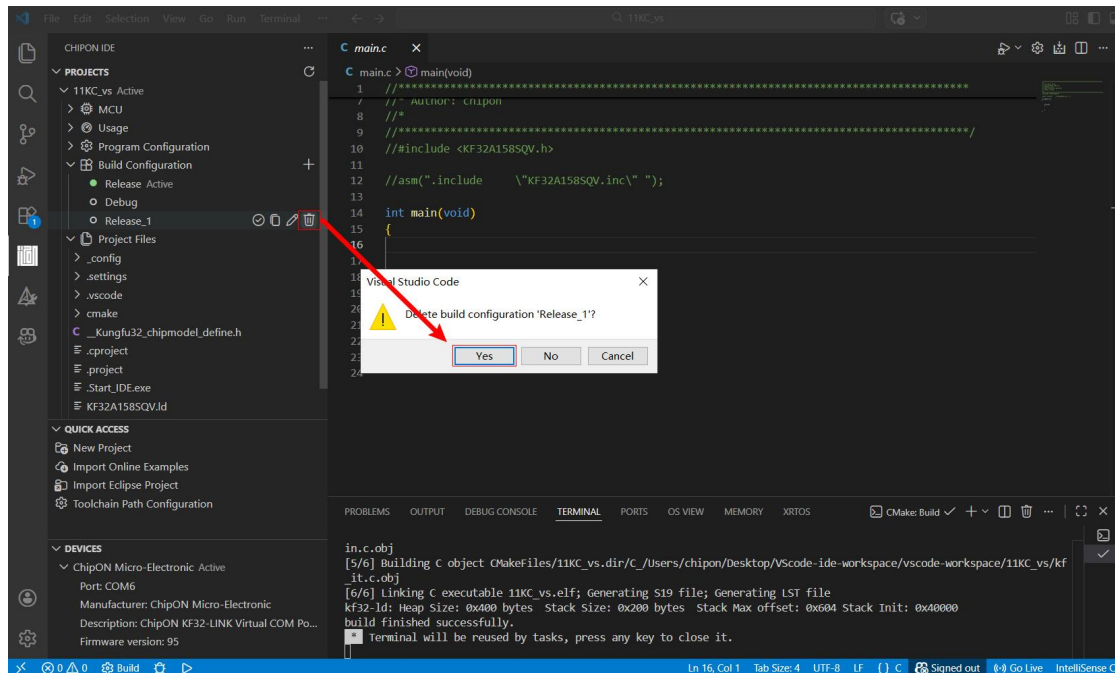


图 5-14 删除编译配置

5.4.2 执行构建

首先设置默认 Build Preset 配置。在 ChipON Projects 视图中，展开项目节点->Build Configuration。选中目标配置右侧图标，点击 Set as Default。该配置名称旁会出现 Active 标记，表示已激活。

然后执行 Build。可通过以下任一入口触发构建：

- Projects 视图：在项目根节点右侧点击 Build Project 图标（或右键菜单选择）。
- Explorer 视图：在项目文件夹上右键点击 Build Project。
- 命令面板：使用快捷键：Ctrl+Shift+P，然后输入 ChipON: Build Project。

Tips:

以下场景会触发全量编译：

1. 首次构建
2. 执行 Clean 后再 Build
3. 切换 Build Preset（如 Debug ↔ Release）
4. 修改了全局编译选项
5. 删除了 Build 输出目录

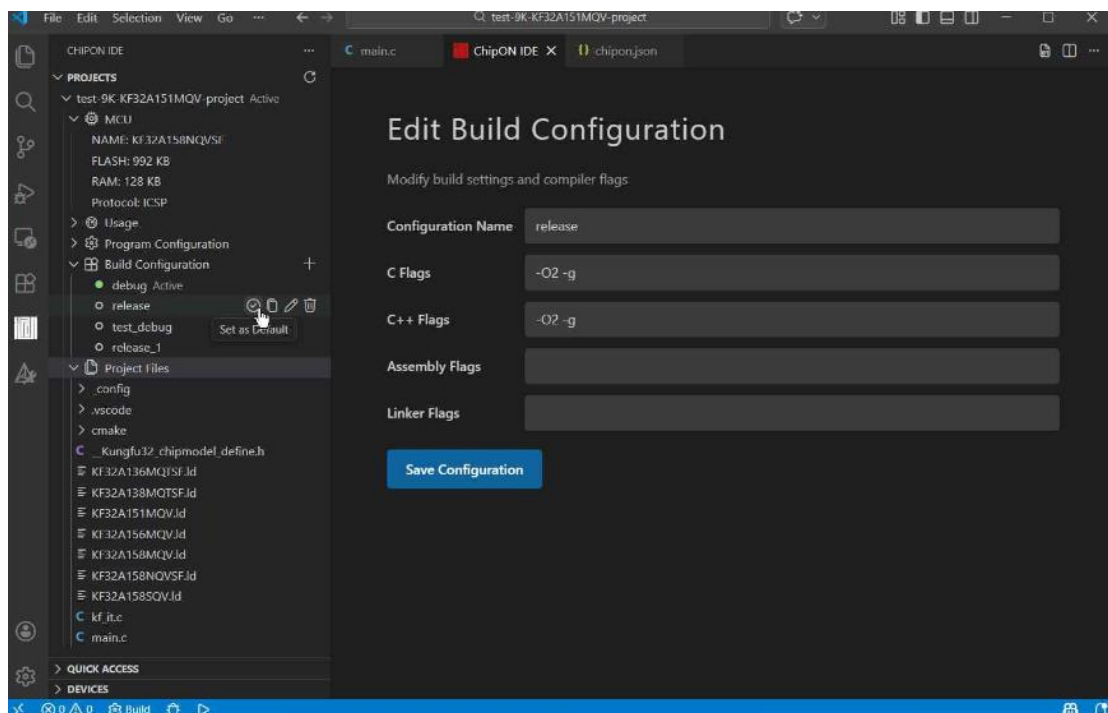


图 5-15 激活编译配置

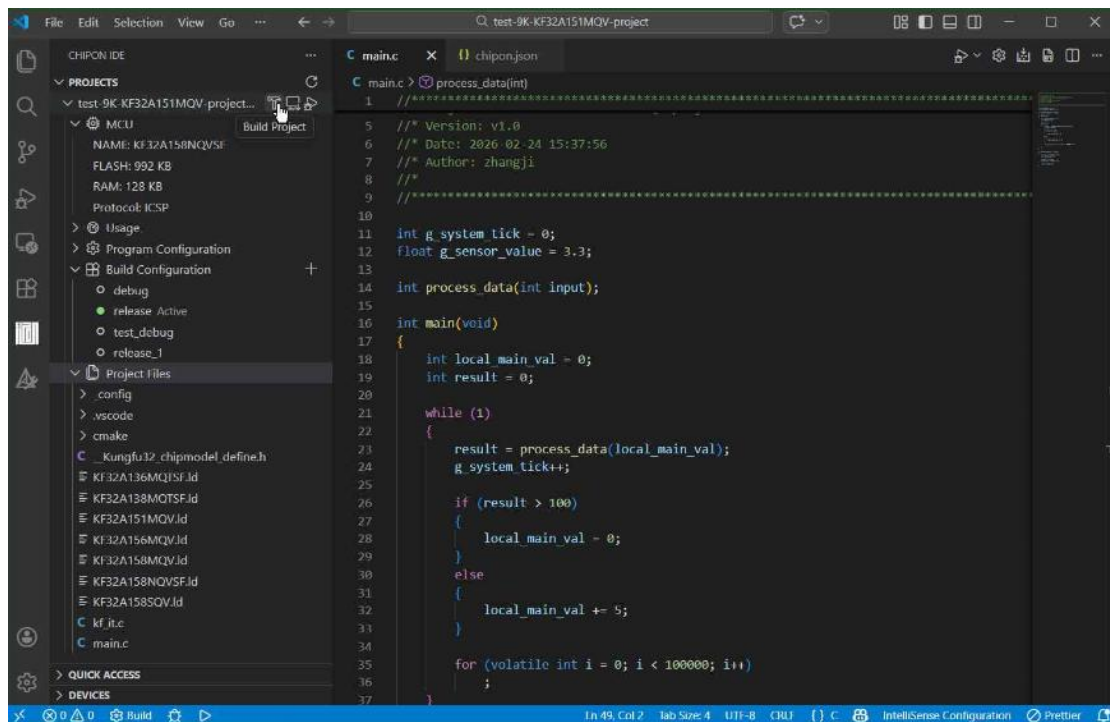


图 5-16 执行编译

Tips:

构建时注意事项:

1. 插件自动检测是否需要执行 CMake Configure (例如首次构建或配置发生变化时)。
2. 如果执行 CMake Configure, 则执行日志输出显示在 Output 面板的 "CMake/Build" 窗口中。
3. 如果修改参数后未自动触发 CMake Configure (例如手动调整了 vscode 设置), 可以手动在 Projects 视图手动执行 Configure。
3. Build 执行时, 编译日志显示在 Terminal 面板。

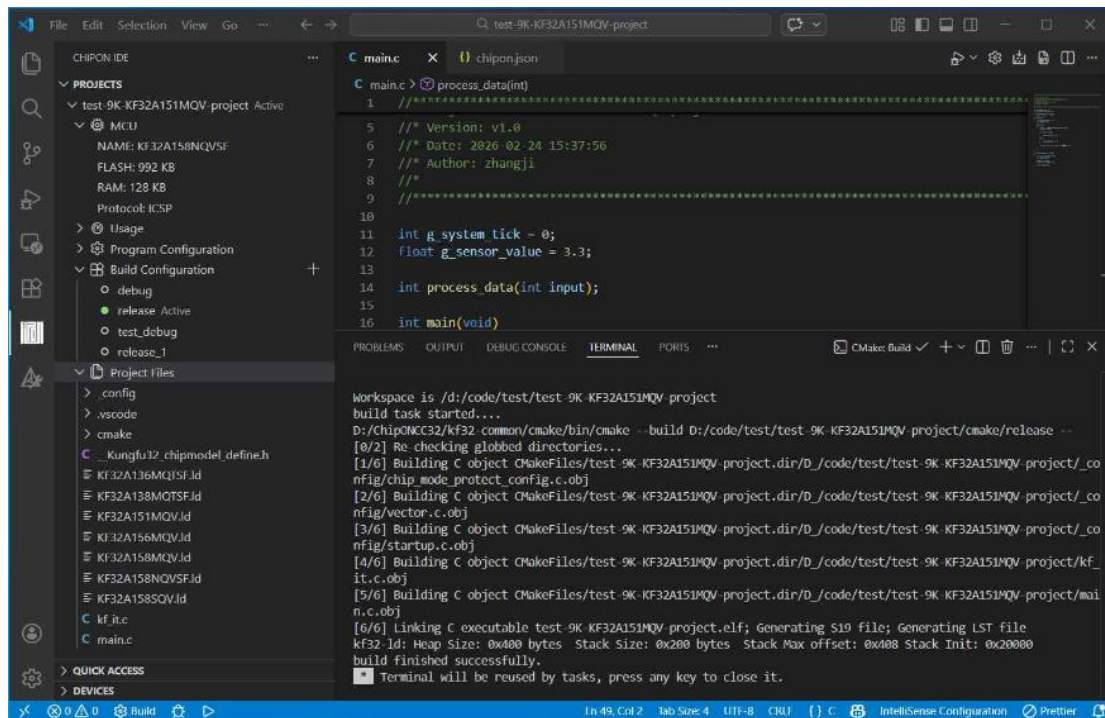


图 5-17 编译成功效果

构建成功后，会在 cmake/【配置名】文件夹下生成如下文件：

- project-name.elf: ELF 格式可执行文件（用于调试）
- project-name.hex: Intel HEX 格式文件（用于下载）
- project-name.bin: 原始二进制文件
- project-name.map: 链接映射文件（查看内存布局）

5.4.2.1 命令行构建

通过界面上触发 Build 操作后会先执行 configure 再执行 Build，输出日志在 Output 窗口，可以找到完整命令行，例如

- Configure 命令输出

```
[proc] Executing command: D:/toolchains/ChipONCC32/kf32-common/cmake/bin/cmake
-DCMAKE_BUILD_TYPE=Debug "-DUSER_C_FLAGS=-O0 -g" "-DUSER_CXX_FLAGS=-O0 -g"
-DUSER_ASM_FLAGS=-gdwarf-3 -DUSER_LINKER_FLAGS= -DCURRENT_PRESET=debug
-DCONFIG_CHIP_NAME=KF32A100GQP -DCONFIG_ARCH=kf32r -DENABLE_CPP=true
-DCMAKE_TOOLCHAIN_FILE=D:/toolchains/ChipONCC32/kf32-toolchain-v0.1.6/cmake/llv
m_toolchain.cmake -S D:/code/vs/chipon/test/project/cmake -B
D:/code/vs/chipon/test/project/cmake/debug -G Ninja
```

- 以及 Build 命令输出

```
[proc] Executing command: D:/toolchains/ChipONCC32/kf32-common/cmake/bin/cmake
--build D:/code/vs/chipon/test/aewwq/cmake/debug --
```

Tips:

请确保当前 shell 窗口包含工具链目录环境变量，例如：
CHIPON_TOOLCHAIN="D:\\toolchains\\ChipONCC32\\kf32-toolchain-v0.1.6"

5.4.3 清理构建

ChipON VSCode Extension 支持执行清理构建产物（build 目录下的对应文件）的功能。右击 Projects 视图项目 -> Clean Project，或 Explorer 视图 -> Clean Project。该操作等同于执行 CMake 的 clean target。

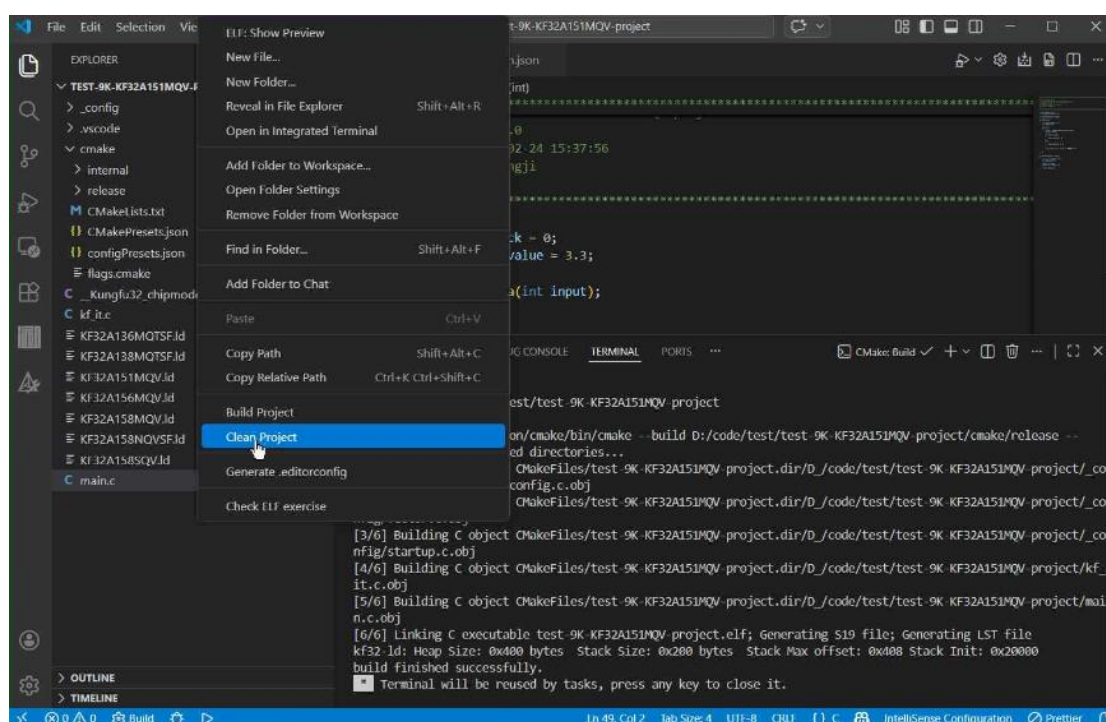


图 5-18 执行清理项目

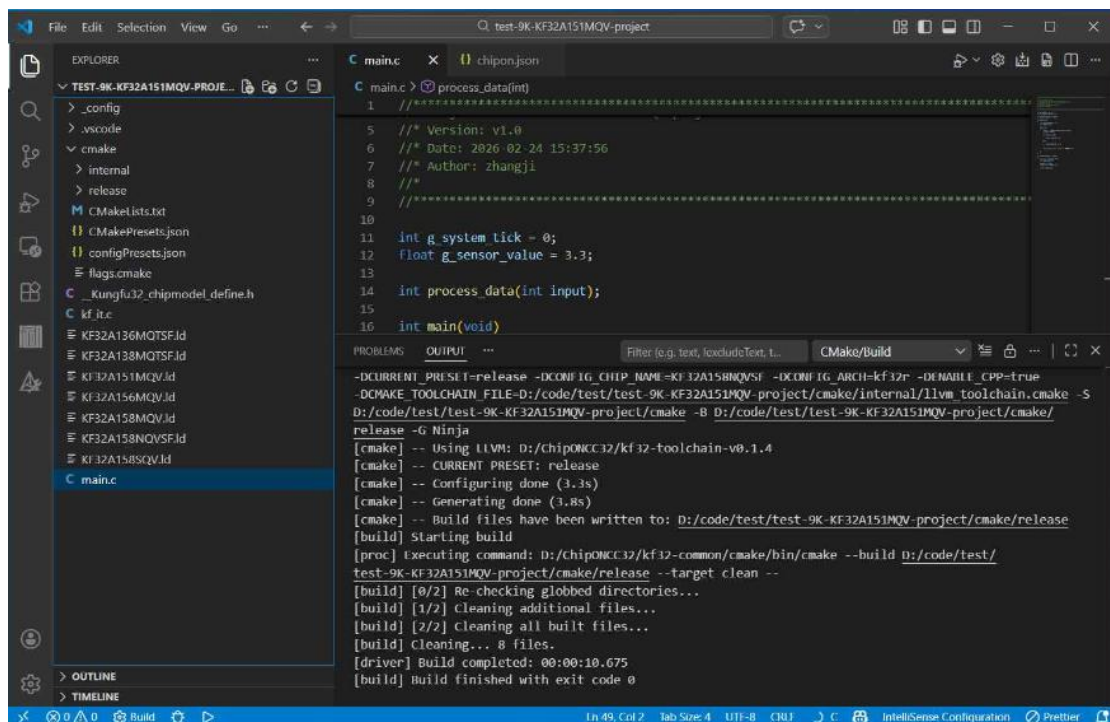


图 5-19 完成执行清理

5.5 项目程序下载

5.5.1 设备管理

ChipON VSCode Extension 启动时会自动扫描当前可以使用的设备，也可以通过点击 DEVICES 栏右侧的刷新按钮，刷新 DEVICES 栏。DEVICES 栏支持多设备管理。标记 **Active** 的设备即是当前激活的设备，程序下载、程序调试都会操作当前处于 Active 状态的设备。



图 5-20 DEVICES 栏界面

未激活的设备右侧的 Connect Device 按钮即可激活该设备。全局可以激活的设备仅有一个。

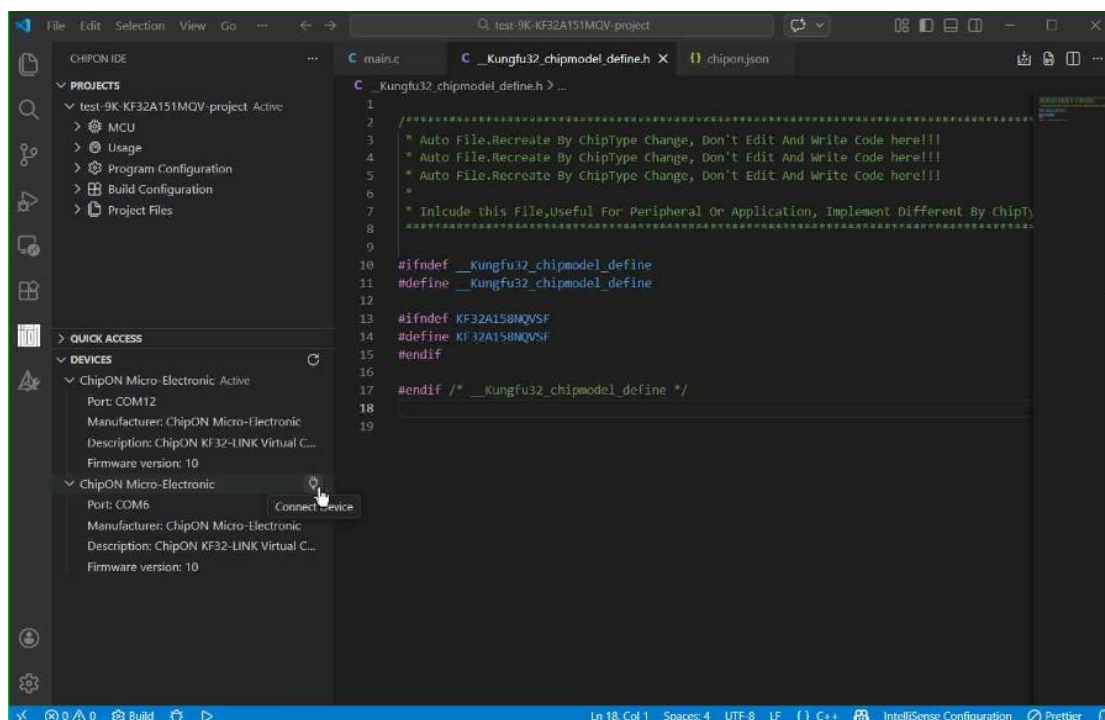


图 5-21 激活设备功能

ChipON VSCode Extension 还提供固件升级和安装驱动的功能。支持对 KF32 编程器（KF32-LINK-A）设备进行固件升级与连接驱动的安装。

固件升级：

在激活的设备右侧，点击 Open Firmware Updater 的按钮打开固件升级的程序。

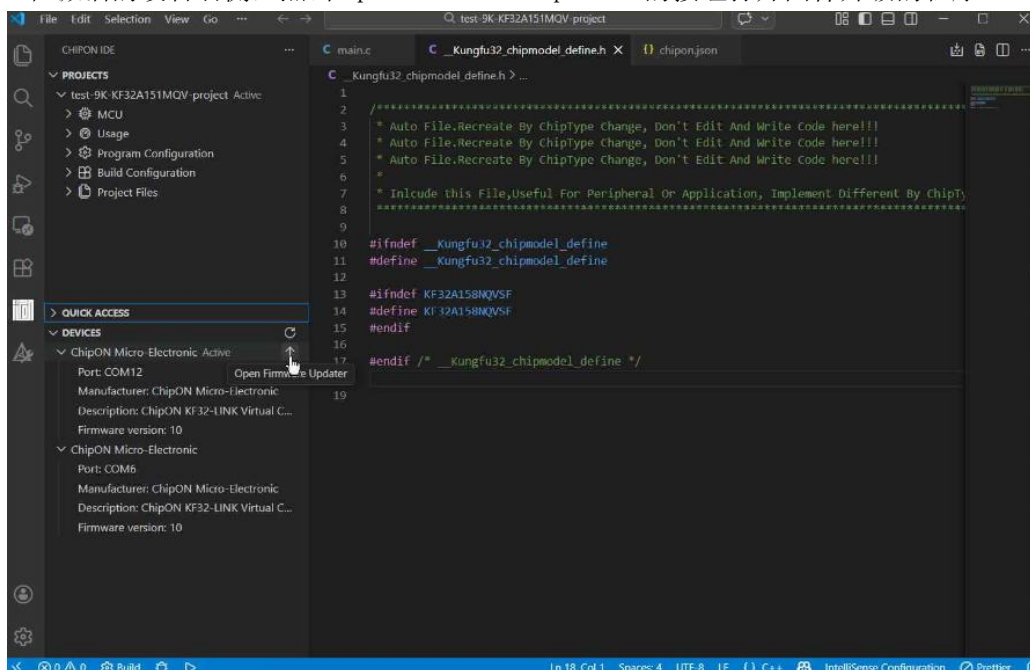


图 5-22 固件升级功能



图 5-23 固件升级软件界面

驱动安装:

通过 `ctrl+shift+P` 唤起命令行,在输入框中输入 `ChipON: Install KFLINK USB Driver`, 点击命令即可以唤起安装驱动的程序。按照设备驱动程序安装向导完成驱动安装。

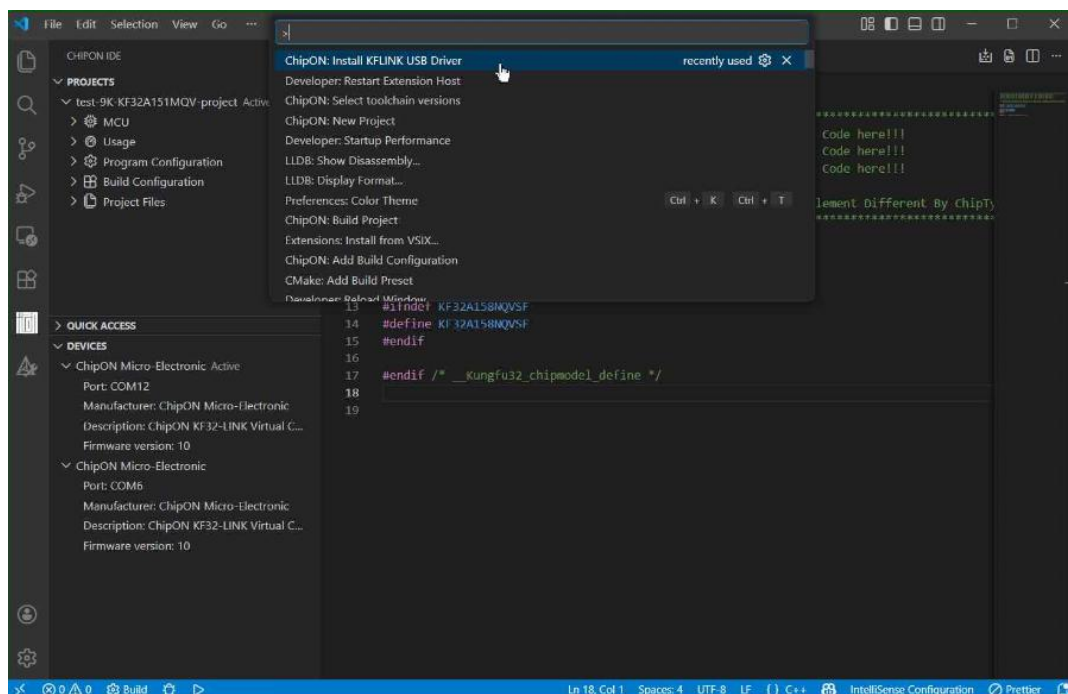


图 5-24 安装驱动功能



图 5-25 安装驱动界面

Tips:

固件升级和安装驱动的功能仅在 windows 下支持。

5.5.2 下载配置

项目下所有的下载配置都存放在 .vscode/chipon.json 的 programSettings 参数中。一个完整的下载配置如下所示：

```
{
  "name": "default",
  "syncToProject": true,
  "mode": "NO_ISP",
  "encryption": "Protect_NO",
  "authToken": "xxxx",
  "programPath": "xxxx",
  "areasSetting": {
    "enabled": false,
    "areas": [["0x00000000", 1]],
  },
}
```

下载配置参数解析：

- name: 配置名称。
- syncToProject: 是否同步到项目。如果选择 true，则 mode 参数和 encryption 参数设置的值会在下载按钮触发的时候，同步到项目代码中。否则，不同步。
- mode: 模式。
- encryption: 加密保护模式。
- authToken: 调试鉴权密文。该参数为可选参数，不会在默认项目中出现。如果芯片通

过 PRO 进行了调试鉴权加密，该参数需要配置从 PRO 中下载的密文。

- **programPath**: 指定下载文件地址。该参数为可选参数。如果不配置该参数，那么 Extension 会根据 build 的配置参数进行自动查找下载文件。配置了该参数，Extension 会直接获取指定的文件进行下载。
- **areasSetting**: 指定 Flash 地址范围下载的配置。
 - **areasSetting.enabled**: 指定 Flash 地址范围下载是否启用。该参数如果配置为 true，则表示会按照 areas 中的参数进行指定 Flash 地址范围下载。否则，则是全范围下载。
 - **areasSetting.areas**: 表示的是指定 Flash 地址范围下载地址范围的数组。该参数是二元的数组，每个元素的第一个参数是起始地址，第二个参数是需要下载的长度。第一个参数需是 1K 的倍数，第二个参数的单位默认是 1K。

用户可以在 Projects 视图的 Program Configuration 节点上进行对下载配置的复制、编辑、删除操作：

1. 复制配置 (Duplicate)

在现有配置上点击 Duplicate 图标。插件会自动复制该配置的所有参数，并生成一个带后缀的新名称（如 “default_1”）。

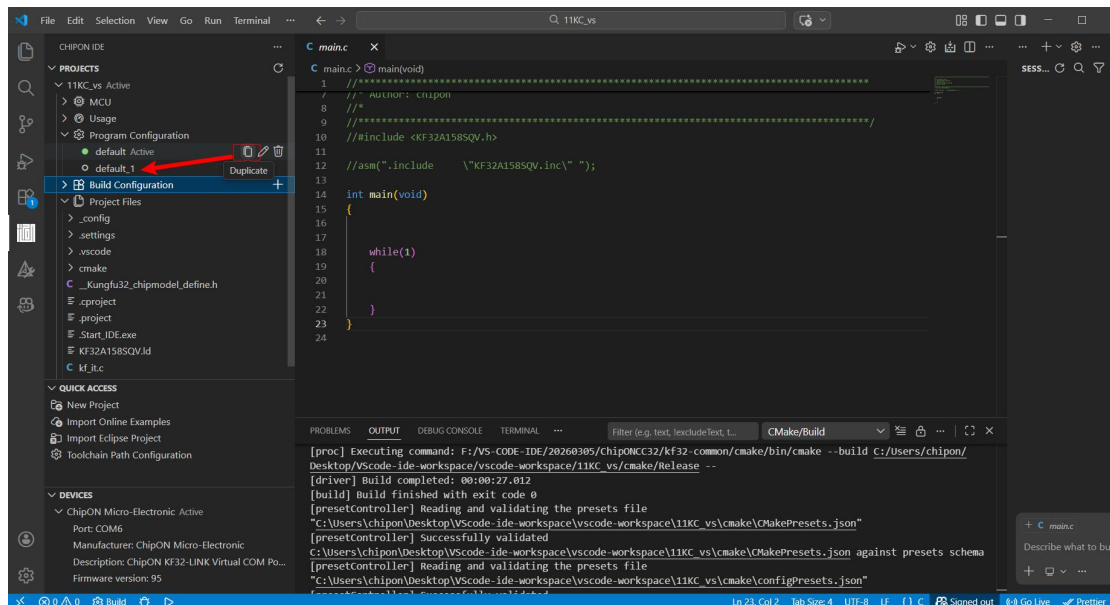


图 5-26 复制编程配置

2. 编辑配置 (Edit)

在配置上点击 Edit 图标。然后会跳转到 chipon.json 文件中配置上述的下载配置。

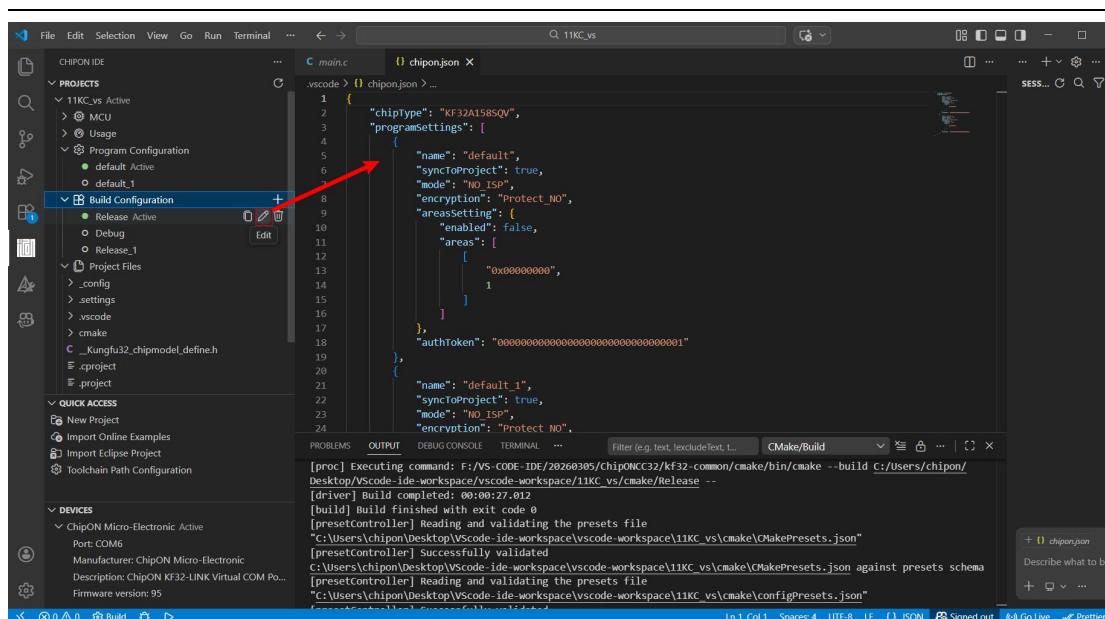


图 5-27 编辑编程配置

3. 删除配置 (Delete)

点击 Delete 图标可删除指定配置。

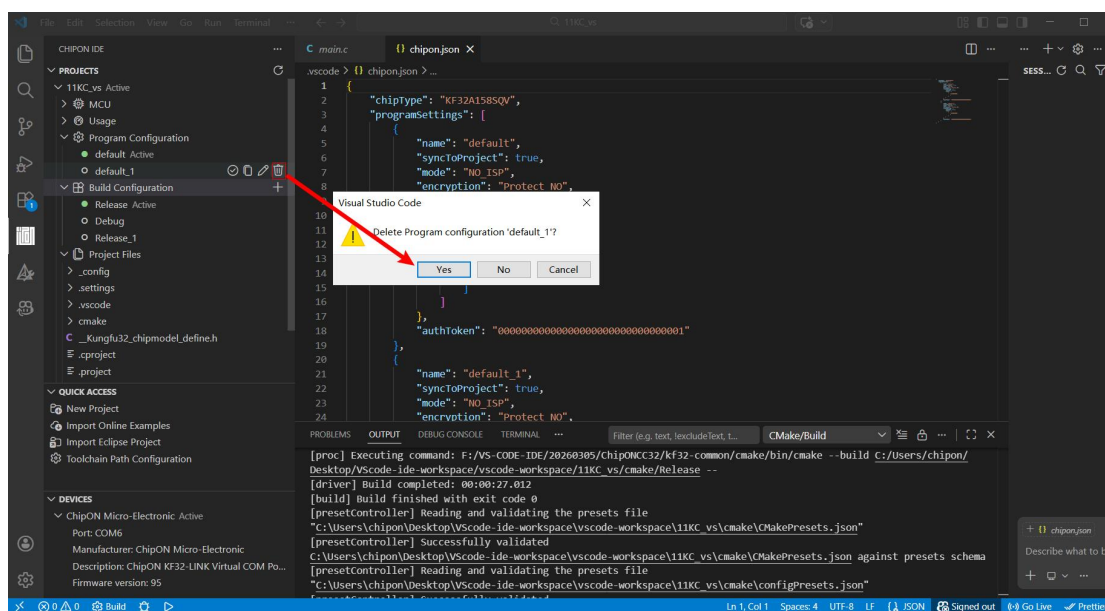


图 5-28 删除编程配置

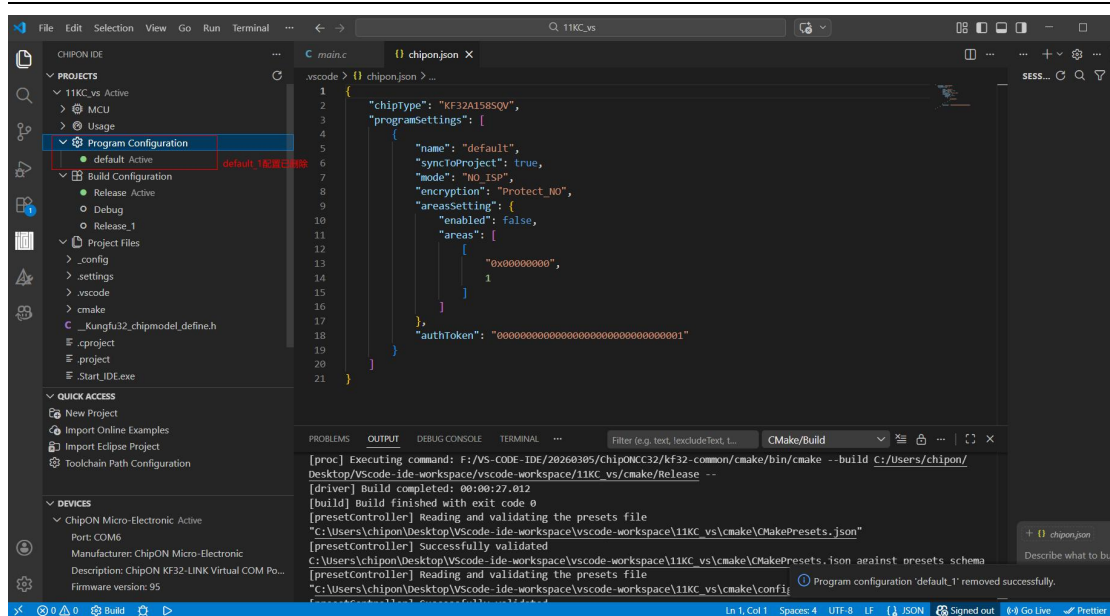


图 5-29 删除编程配置结果

5.5.3 程序下载

ChipON VSCode Extension 支持程序下载功能。点击项目右侧的下载按钮（第 2 个）即可按照激活的配置信息对激活的设备进行下载工作。下载信息会在 Terminal 中显示。

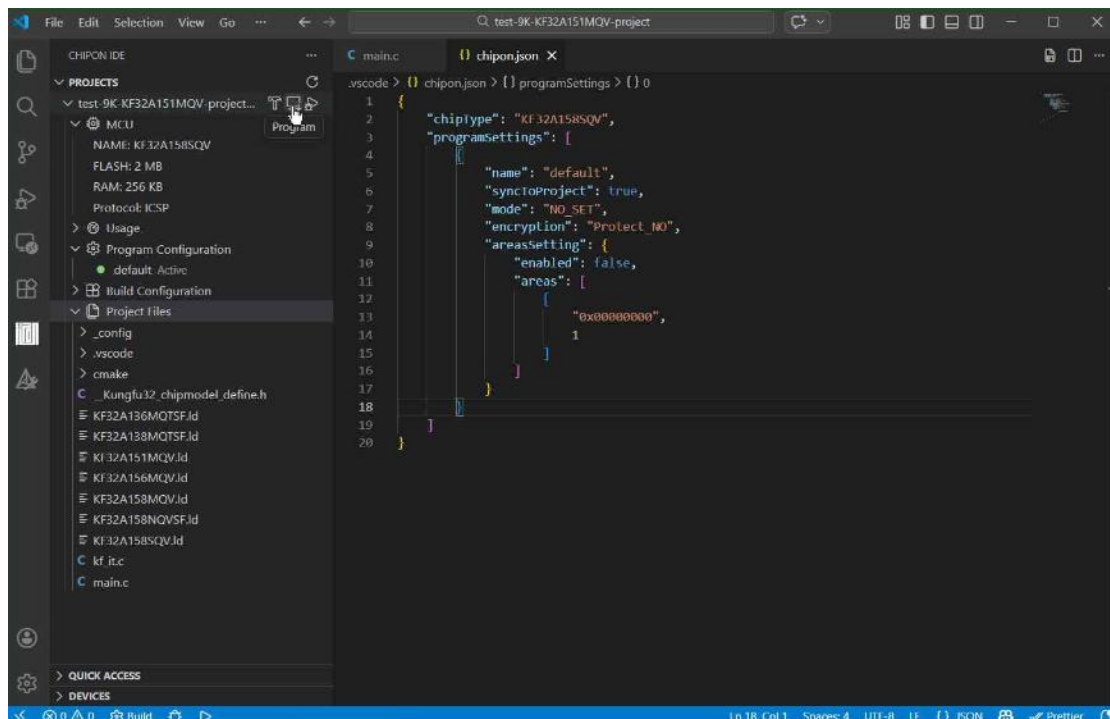


图 5-30 执行下载

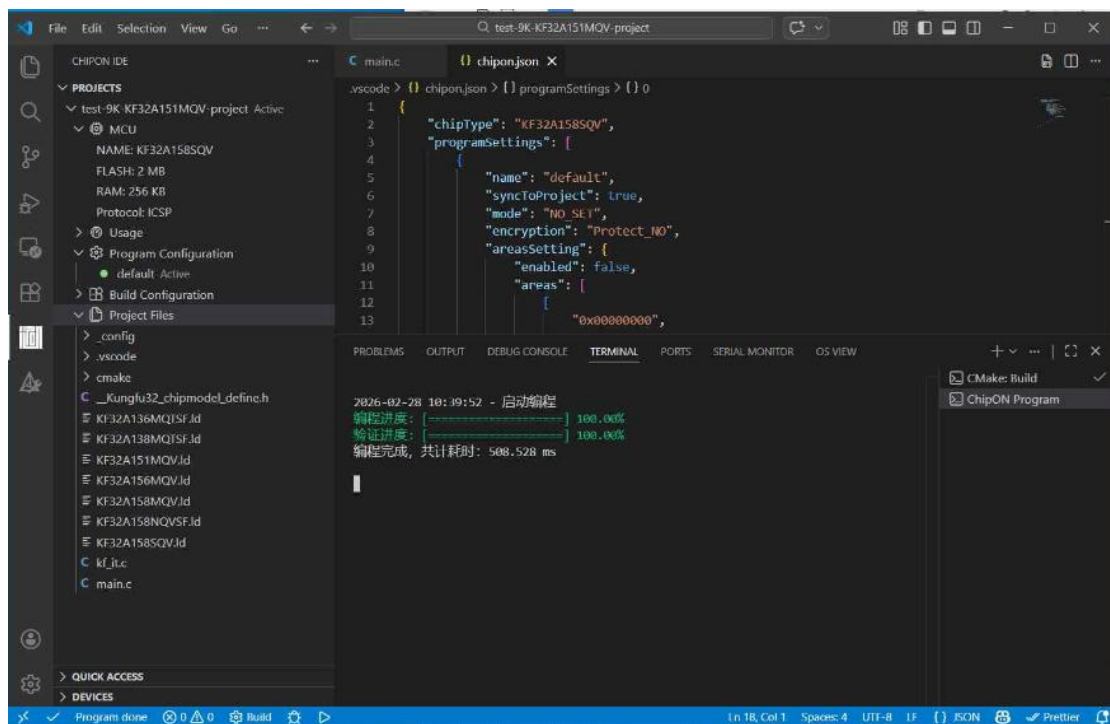


图 5-31 下载成功效果

5.6 在线调试

5.6.1 调试配置

项目下的调试配置存放在 `.vscode/launch.json` 文件中。默认的调试配置如下所示：

```
{
  "name": "KF32-LLDB: Launch",
  "type": "kf32-lldb",
  "request": "launch",
  "startSettings": {
    "startmode": "main",
    "port": 3333, // 默认 3333
    "openocdReadyTimeout": 5000
  },
  "programBeforeDebug": false
},
{
  "name": "KF32-LLDB: Attach",
  "type": "kf32-lldb",
```

```
"request": "attach",
"startSettings": {
  "startmode": "halt",
  "port": 3333, // 默认 3333
  "openocdReadyTimeout": 5000
},
}
```

配置参数解析：

- name: 该调试任务的名称。
- type: 固定为 kf32-1ldb, 表示使用支持 kf32 指令集的 1ldb 进行调试。
- request: 任务启动形式, 支持 launch 和 attach 模式。
- elfPath: 指定的 elf 文件地址。可选参数。如果填写该参数, 将以该参数指向的 elf 文件进行调试; 否则, 根据构建的配置进行查找, 然后启动调试。
- startSettings: 启动参数。
 - startSettings.startmode: 启动模式。目前仅支持 launch 模式下 main, attach 模式下 halt。
 - startSettings.port: 设置端口。
 - startSettings.openocdReadyTimeout: openocd 启动超时时间, 单位为 ms。默认时间为 5s。
- programBeforeDebug: 在调试前编程, 默认 false。

修改调试启动的配置参数需要打开 .vscode/launch.json 文件根据上述的参数解析, 修改配置。

5.6.2 启动调试

启动 Debug(Launch / Attach)的方式: 1. 打开调试视图, 在 ChipON 项目树里点击 Show Debug 或点击 VSCode 左侧边栏 Run and Debug 图标打开 VS Code 的运行与调试视图。2. 在调试视图中, 点击启动按钮, 即可启动调试。

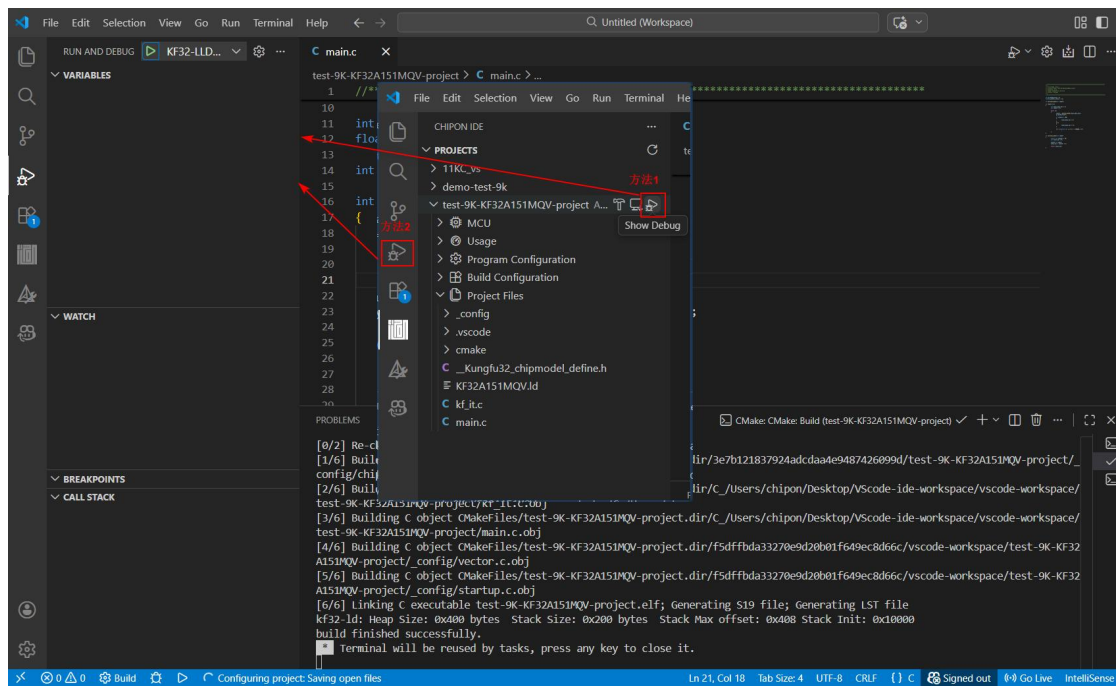


图 5-32 打开调试视图

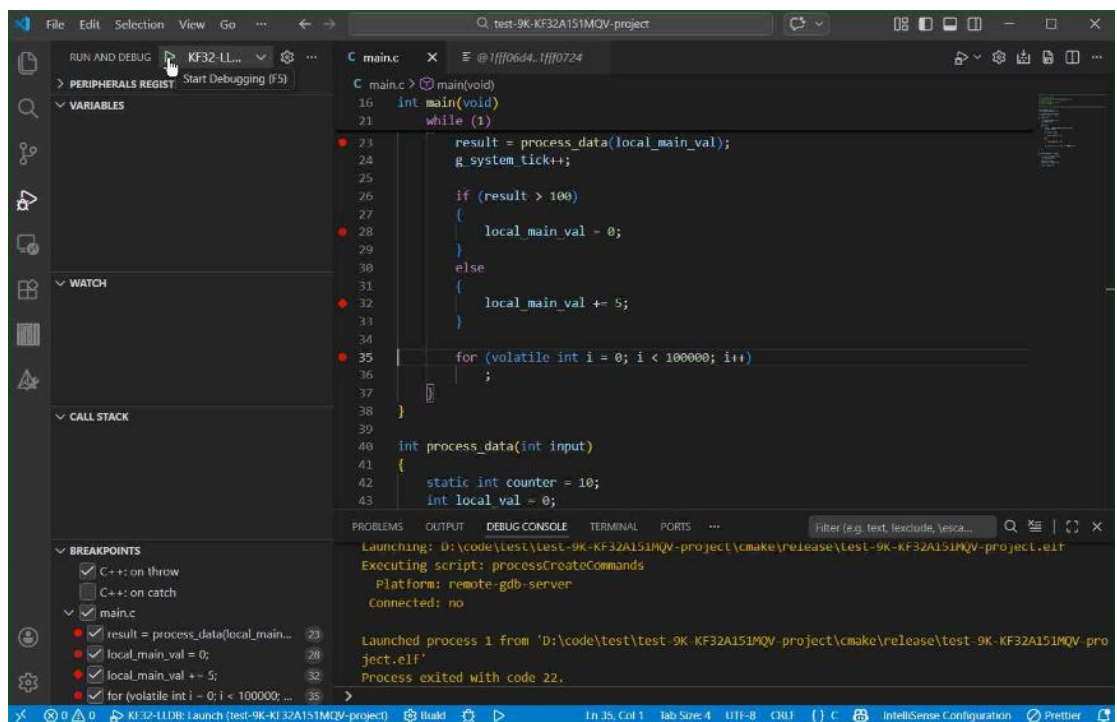


图 5-33 启动调试

5.6.3 调试操作

ChipON VSCode Extension 在调试时支持 Continue、Step Over、Step In、Stop、Restart。在启动调试后，界面会出现调试的浮窗，调试时所需要使用到的按键都在浮窗上。

- Continue 操作：触发 Continue，有断点时程序会运行到下一个断点处暂停。无断点时，正常情况下程序持续运行，若出现程序运行错误会出现程序运行暂停或程序退出的情况。

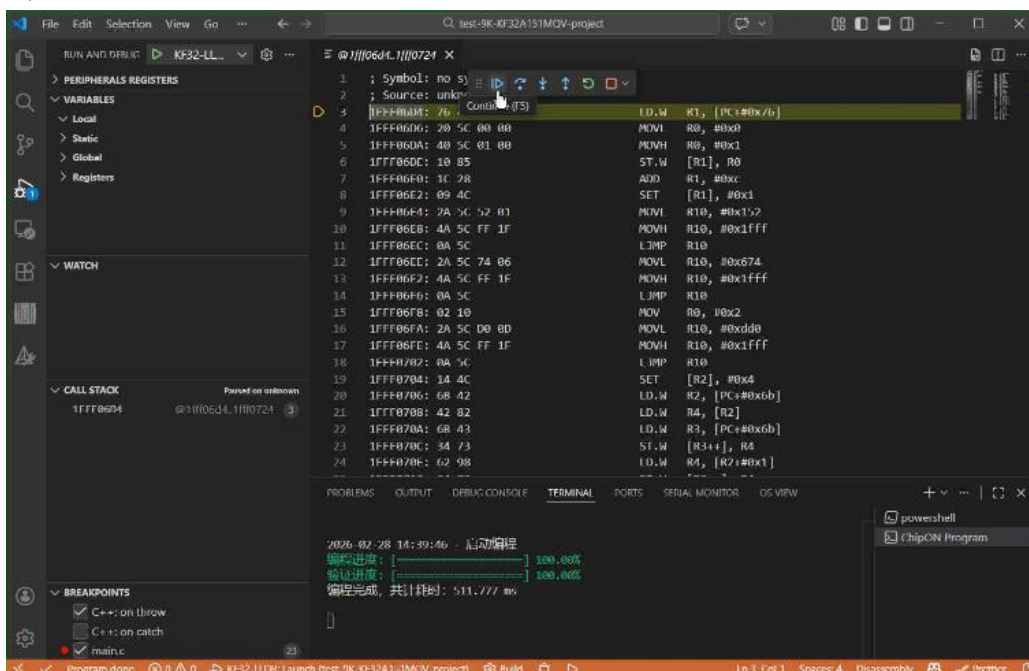


图 5-34 Continue 操作

- Step Over 操作：单步跳过 / 逐过程操作。执行当前行代码。如果当前行不包含函数调用：直接执行该行，停在下一行。如果当前行包含函数调用：将该函数作为一个整体执行完毕，不进入函数内部，直接停在调用该函数之后的下一行代码。

Tips:

如果当前激活的界面为反汇编视图界面，那么 Step Over 操作会变为指令 Step Over 操作。

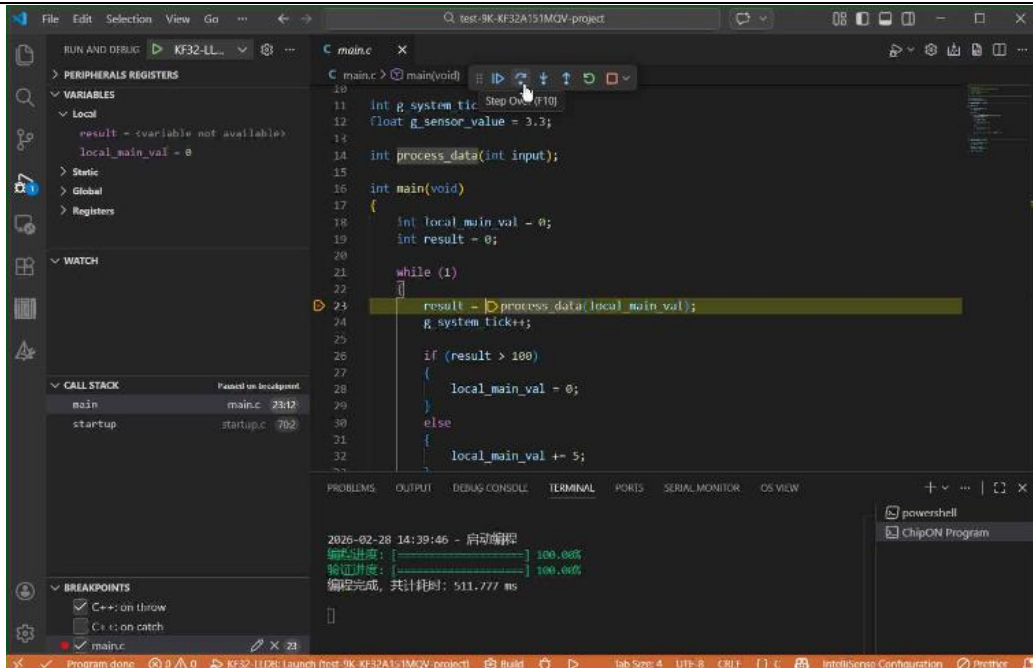


图 5-35 Step Over 操作

- Step In 操作：执行当前行代码。如果当前行不包含函数调用：效果同 Step Over，停在下一行。如果当前行包含函数调用：调试器会进入该函数内部，并停在函数体的第一行代码。

Tips:

如果当前激活的界面为反汇编视图界面，那么 Step In 操作会变为指令 Step In 操作。

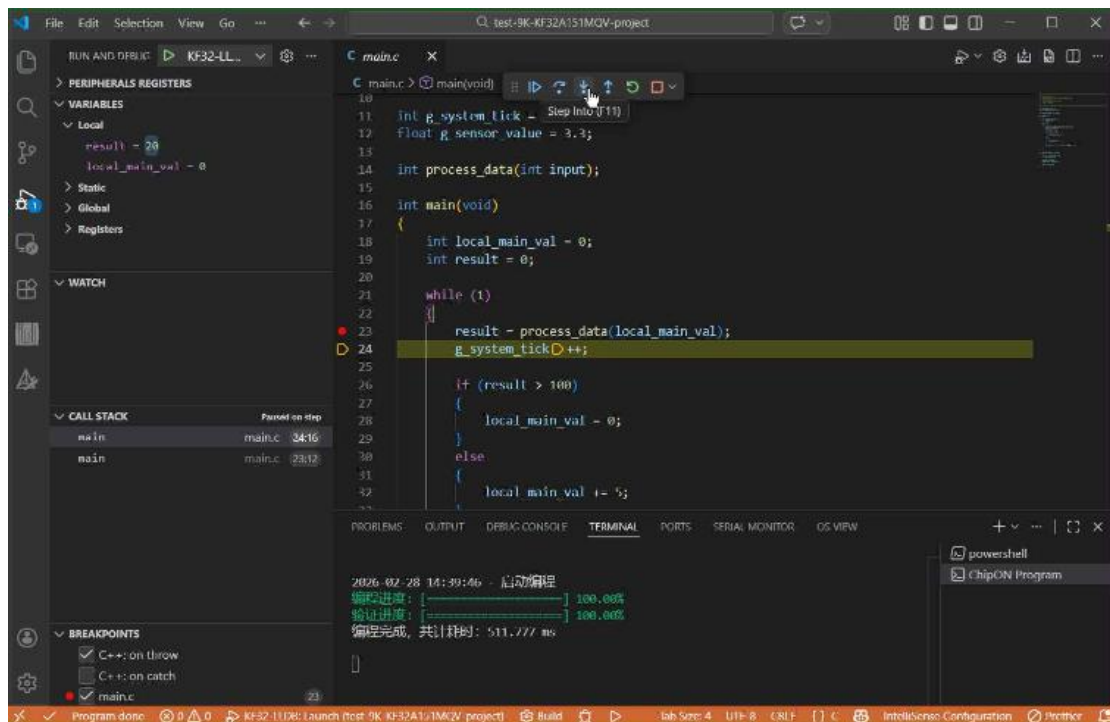


图 5-36 Step In 操作

- Stop 操作：点击后，会终止此次调试会话。
- Reset 操作：点击调试过程中的绿色按钮，即可 reset 芯片，会暂停到 startup 方法处。在调试运行过程中或暂停状态均可以点击按钮进行 reset 操作。

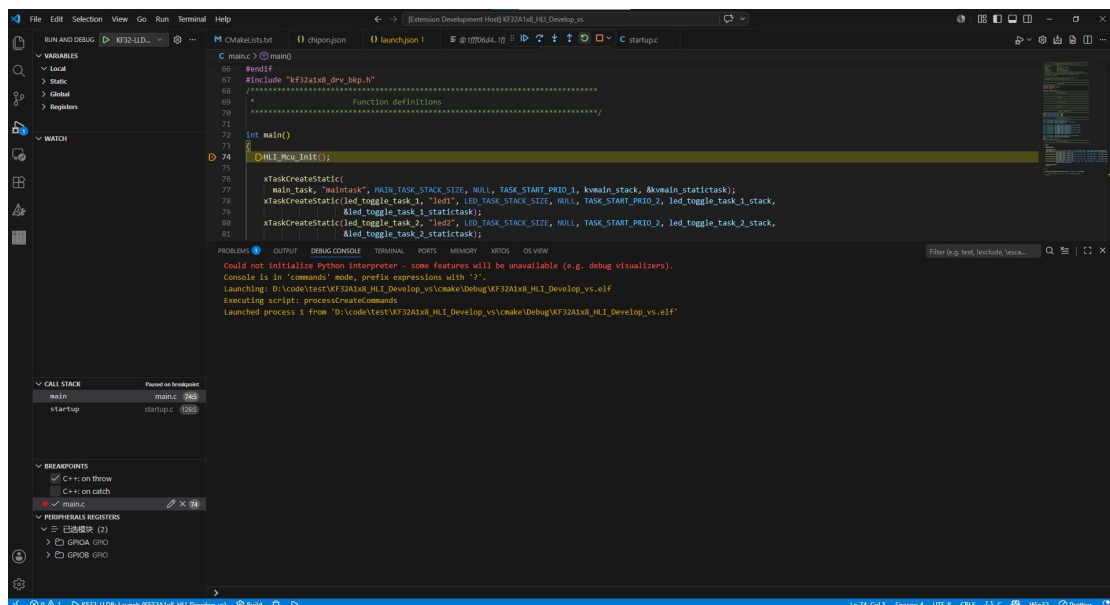


图 5-37 reset 操作

- Restart 操作：暂不支持功能。可以使用 Stop 操作加启动调试来实现 Restart 的功能。

5.6.4 调试断点

ChipON VSCode Extension 支持普通断点、条件断点、日志断点、触发断点、Watchpoint。目前断点（包含普通断点、条件断点、日志断点、触发断点）支持个数为 4 个，WatchPoint 支持个数为 2 个。

- 普通断点（Line Breakpoint）

- 设置：在代码行号左侧空白处单击（红点）。
- 启用/禁用：断点上右键可禁用/启用；也可在 Run and Debug 的 BREAKPOINTS 面板统一管理。
- 命中效果：程序暂停，进入调用栈、变量、寄存器/反汇编（如开启）等调试流程。
- 在 ChipON VSCode Extension 中，支持在反汇编视图中设置普通断点。

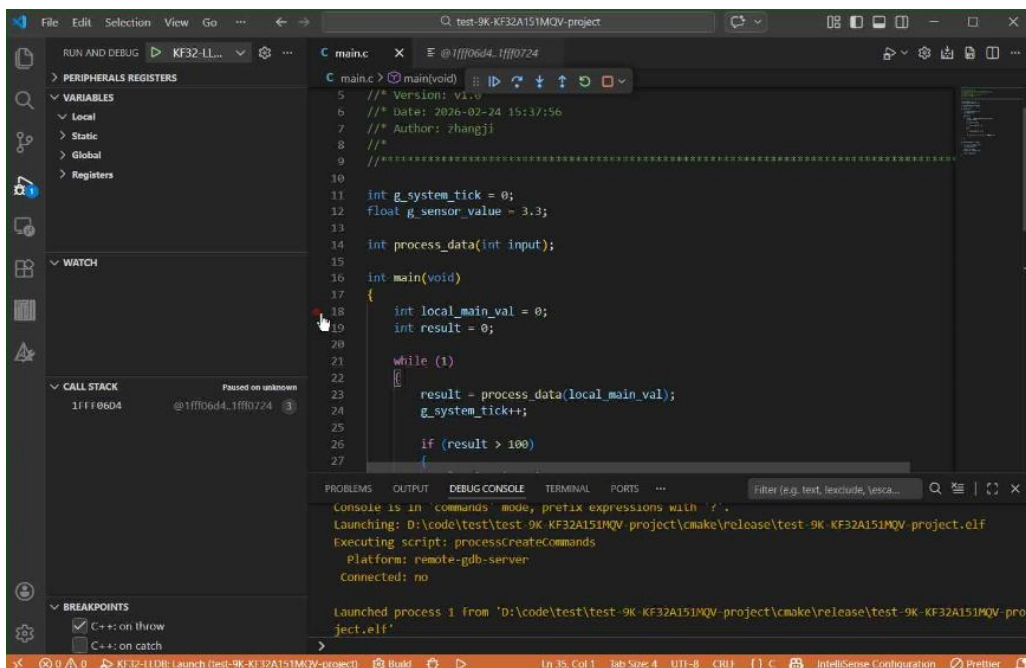


图 5-38 普通断点

- 条件断点（Conditional Breakpoint）

- 设置方式：在行号左侧右键断点位置 → 选择 条件断点（Conditional Breakpoint）。
- 条件表达式：填写表达式（由 LLDB 负责求值），例如：i == 10、ptr != 0 && *ptr == 0x5A
- 使用建议：条件复杂时会增加暂停判定开销；尽量让条件简单、可快速计算。

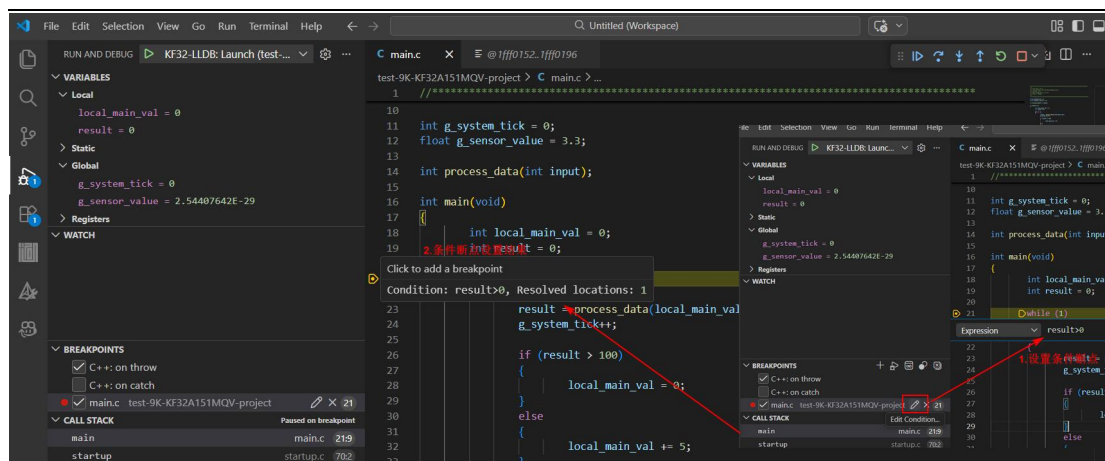


图 5-39 条件断点

● 触发断点 (Triggered Breakpoint)

- 作用：在其他断点触发后才激活该断点。
- 设置方式：行号左侧右键 → 添加触发断点，选择需要触发的断点。

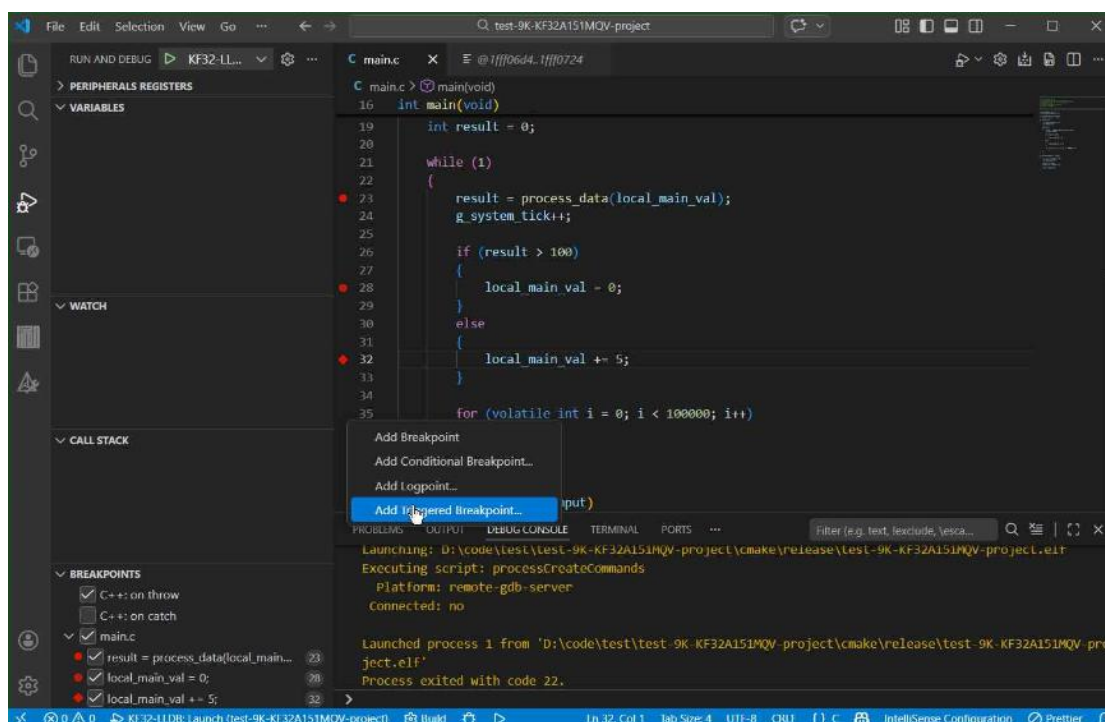


图 5-40 触发断点

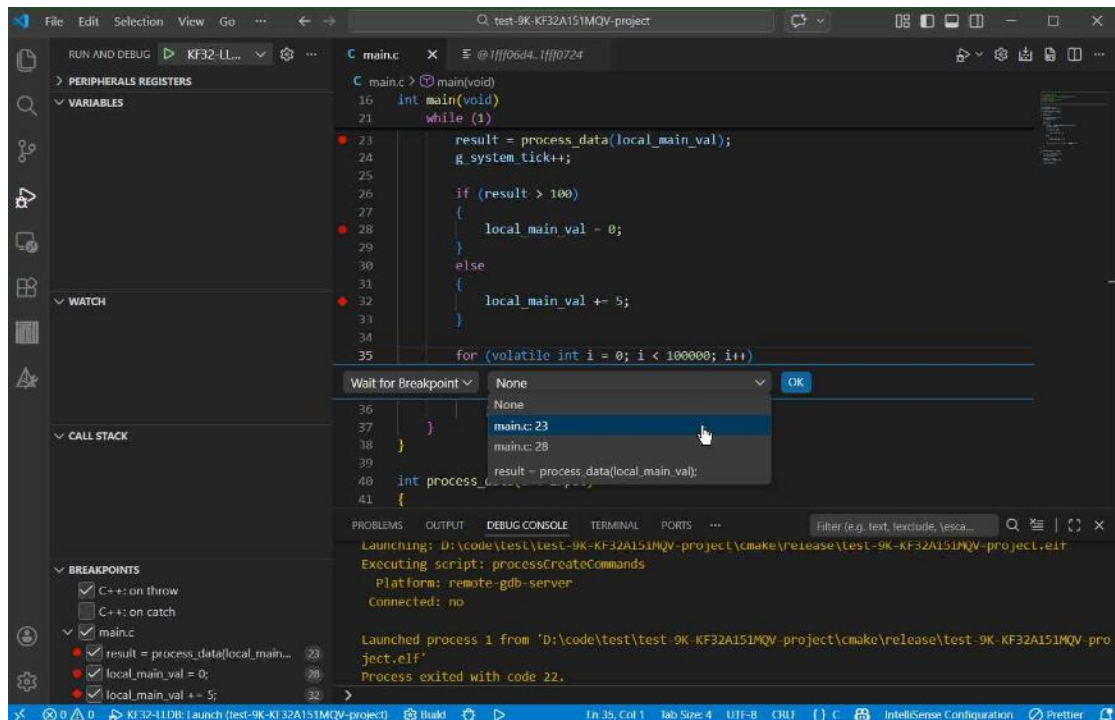


图 5-41 选择触发断点条件

● 日志断点 (Logpoint)

- 作用：命中时不暂停，只把日志输出到 Debug Console（适合替代临时 printf）。
- 设置方式：行号左侧右键 → 添加日志点(Logpoint)，填写要输出的消息。
- 输出位置：通常在 VS Code 的 Debug Console。

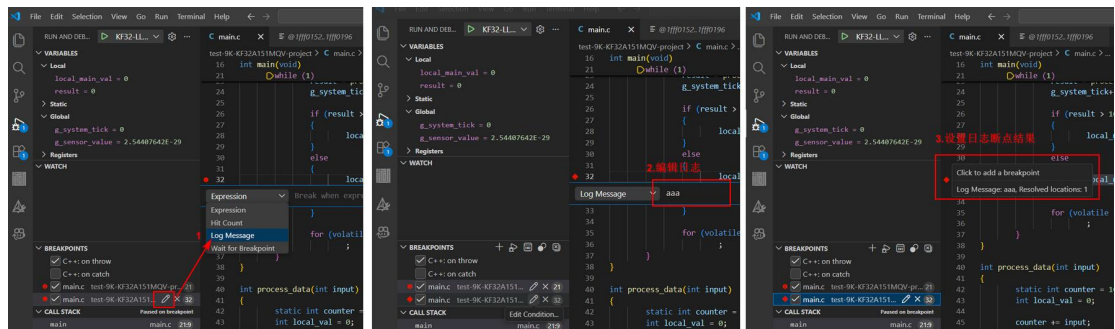


图 5-42 日志断点

● 监控点 (Watchpoint)

- 作用：对特定地址进行读/写/读写操作的监控。即对某个特定地址进行指定操作时，该断点被触发暂停。
- 设置方式：在变量视图中，对变量元素进行右击，选择需要的类型断点或在 BreakPoint 中点击 Add Data Breakpoint At Address 按钮，然后输入起始地址和结束地址/长度，即可设置。

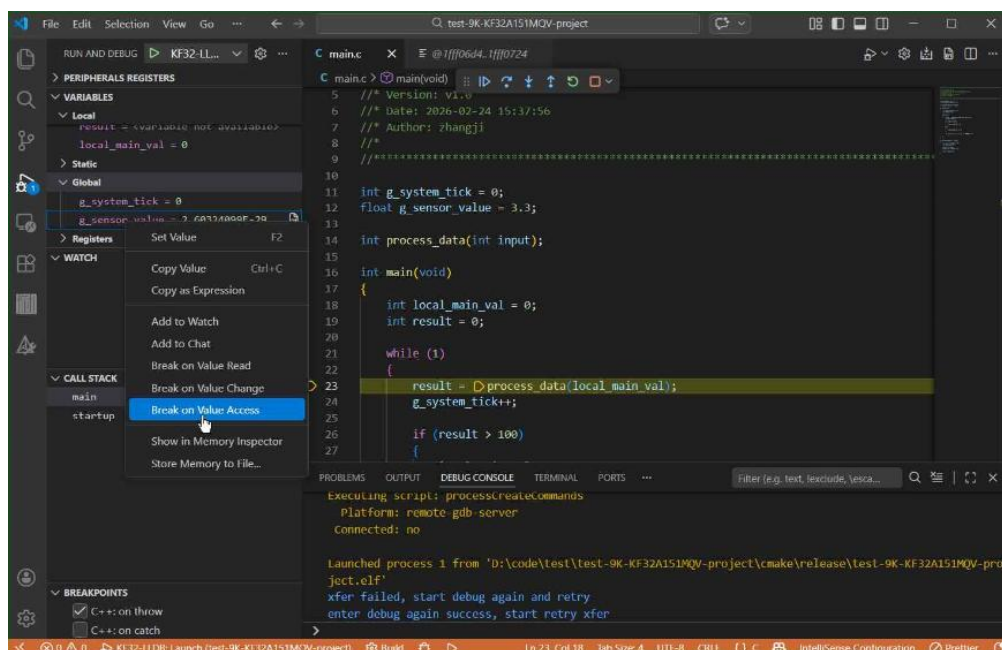


图 5-43 设置 Watchpoint

5.6.5 调试视图

5.6.5.1 Variables (变量视图)

- 功能：调试过程中，显示当前作用域内所有可访问的变量及其值。
- 特点：
 - ◆ 动态更新：单步执行代码时，变量视图中的变量值会实时刷新。
 - ◆ 可编辑：可以双击变量的值进行修改。
 - ◆ 展开对象：对于对象、数组、结构体，可以点击箭头展开查看内部成员。
- Variables 视图位于 VSCode 调试界面的上部。包含 Local（局部变量）、Static（静态变量）、Global（全局变量）、Registers（寄存器）四个部分。

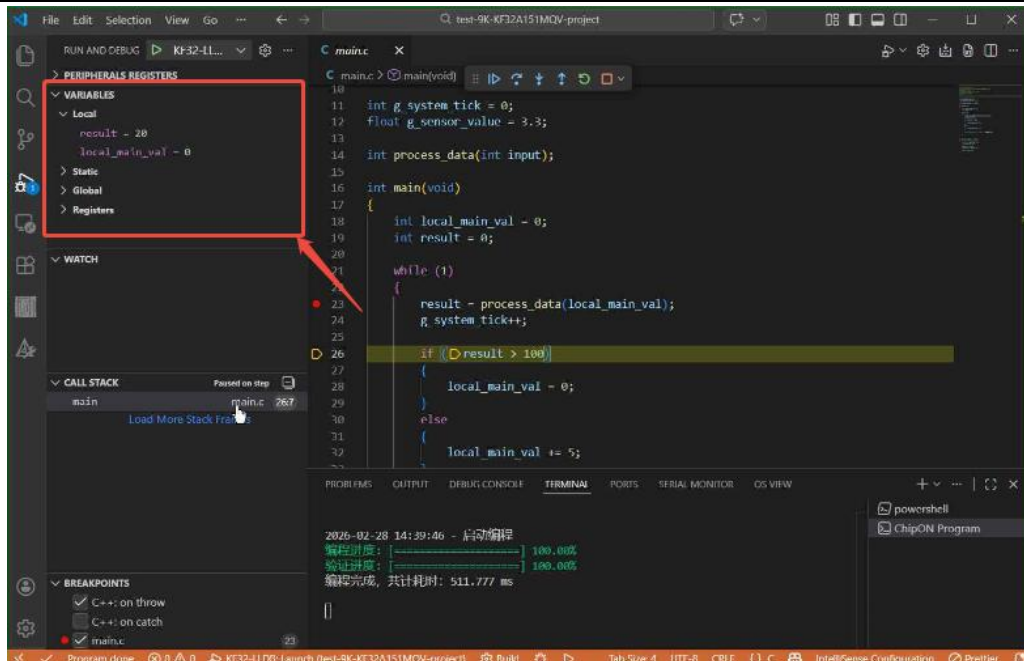


图 5-44 变量视图

5.6.5.2 Watch (监视视图)

- 功能：允许用户手动添加特定的变量名或表达式进行持续监控。
- 特点：
 - ◆ 自定义表达式：不仅可以监视变量 `x`，还可以监视表达式 `x * 2 + y.length` 或 `myObject.data[0].id`。
 - ◆ 持久性：即使变量离开了当前作用域（例如函数执行结束），只要表达式合法，它仍会尝试计算（若不可用则显示错误）。
 - ◆ 条件评估：常用于验证复杂的逻辑判断是否按预期执行。
- Watch 视图位于 VSCode 调试界面的中间部分。点击 Watch 视图右侧的“+”号，可以添加自定义表达式。

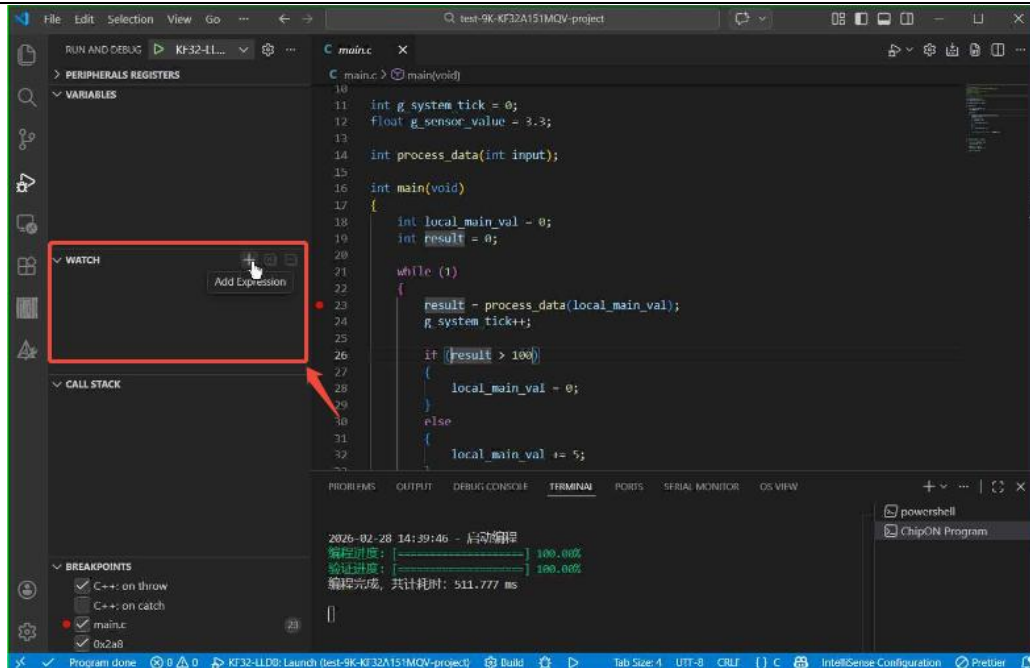


图 5-45 监控视图

5.6.5.3 Call Stack (函数调用堆栈视图)

- 功能：显示导致当前程序暂停的函数调用链路。
- 特点：
 - ◆ 时间倒序：列表顶部是当前正在执行的函数（帧），往下依次是调用它的函数，直到最底部的入口函数（如 main 或 app.start）。
 - ◆ 导航跳转：点击列表中的任意一行，编辑器主窗口会跳转到该函数被调用的位置，且“变量视图”会切换到该函数的作用域。
- 点击 Load More Stack Frame 可以展开 Call Stack。需要查看某一个 Frame 的反汇编视图，右击该 Frame，选择 Open Disassembly View。

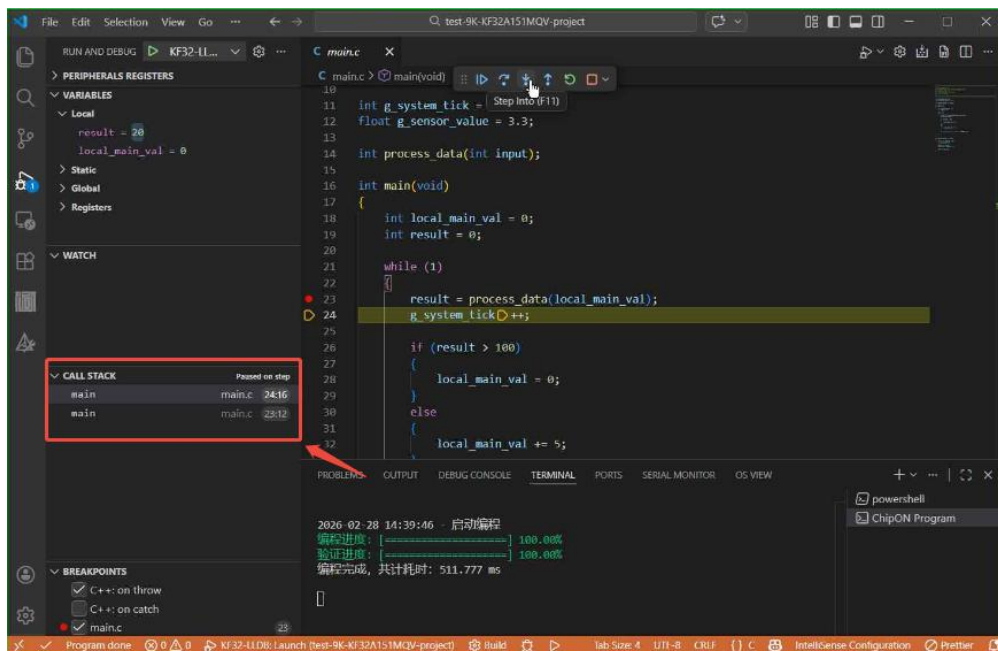


图 5-46 Call Stack 视图

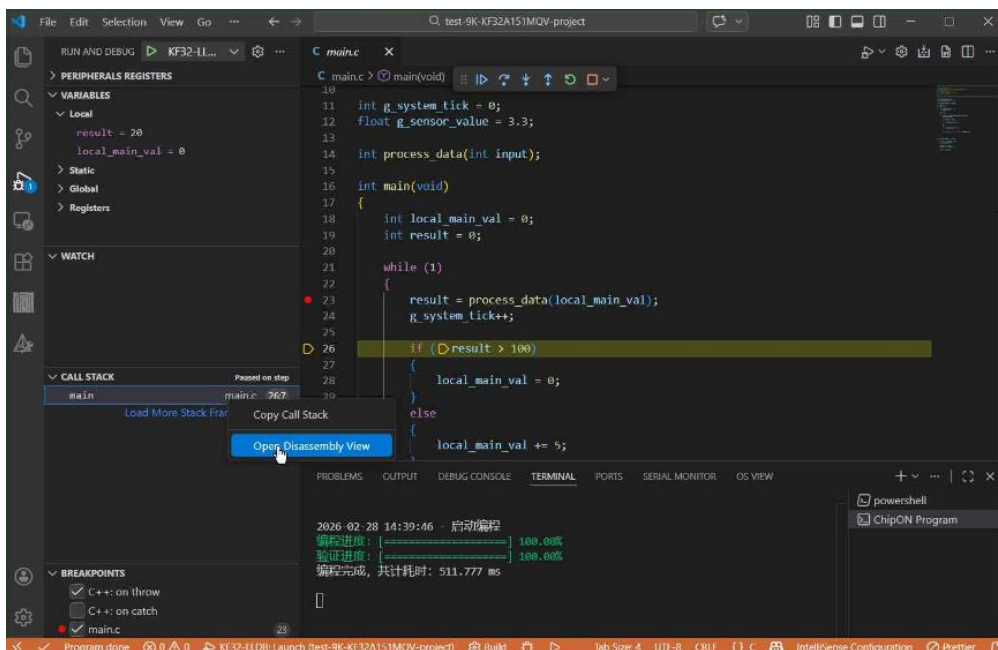


图 5-47 打开反汇编视图操作

5.6.5.4 Breakpoints (断点视图)

- 功能：列出当前工作区中设置的所有断点。
- 特点：
 - ◆ 统一管理：可以看到哪些文件有断点，断点是否生效（实心红点表示生效，空心或灰色表示未命中/禁用）。

- ◆ 批量操作：可以一键禁用所有断点、删除所有断点，或勾选/取消勾选特定断点。
- ◆ 高级设置：在此视图中可以右键编辑断点属性，设置条件断点（Condition，如 $i > 10$ ）、命中次数（Hit Count）或日志消息（Log Message，不暂停只打印）。

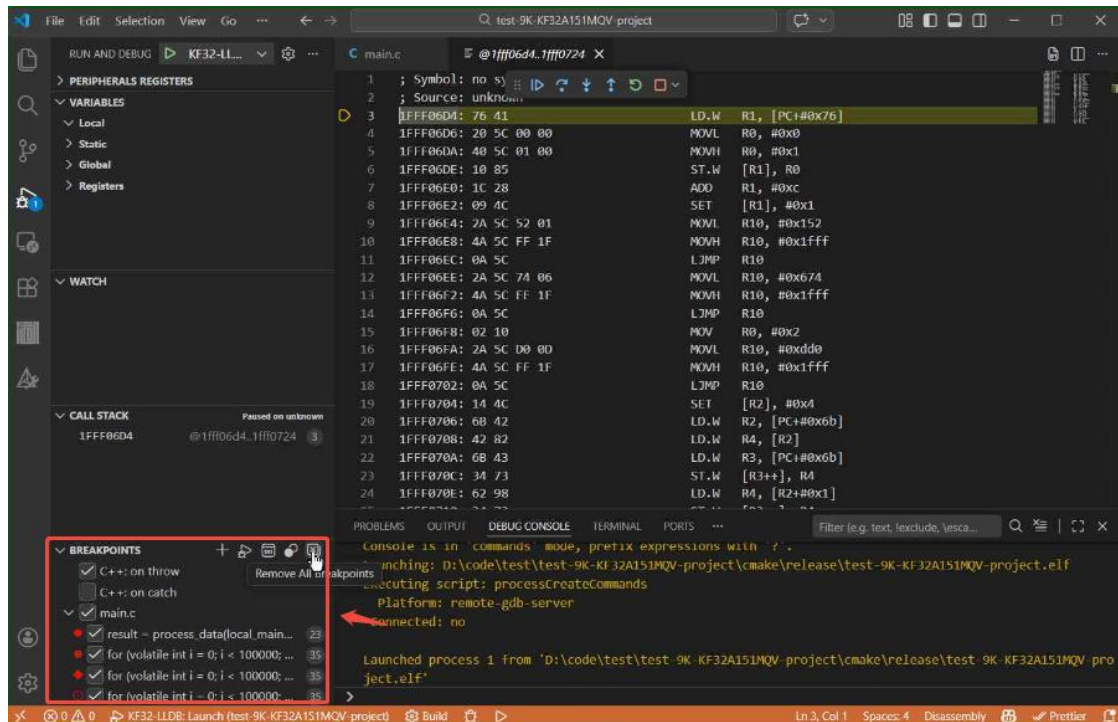


图 5-48 断点视图

5.6.5.5 Memory（内存视图）

- 功能：以十六进制（Hex）和 ASCII 码形式直接展示计算机物理内存或虚拟内存的内容。
- 特点：
 - ◆ 地址定位：输入一个内存地址（如 0x7ffee4b 或变量取址 &myVar），查看该地址及其后续字节的数据。
 - ◆ 原始数据：绕过变量类型系统，直接看二进制位。这对于排查内存泄漏、缓冲区溢出、指针错误至关重要。
 - ◆ 编辑内存：允许直接修改内存中的十六进制值。
- 内存视图可以通过两种方式唤出：1. 在 Variables 视图中，右击变量，选择 Show in Memory Inspector；2. 通过 `ctrl+shift+P` 唤出命令中心窗口，然后输入 Show Memory Inspector 命令。在内存视图中，通过设置起始地址、偏移、长度来查看不同地址的数据。同时，在视图中双击对应地址，可修改对应的值。
- 内存视图右上角的设置按钮中，可以设置视图中数据的进制、显示内容等等配置。

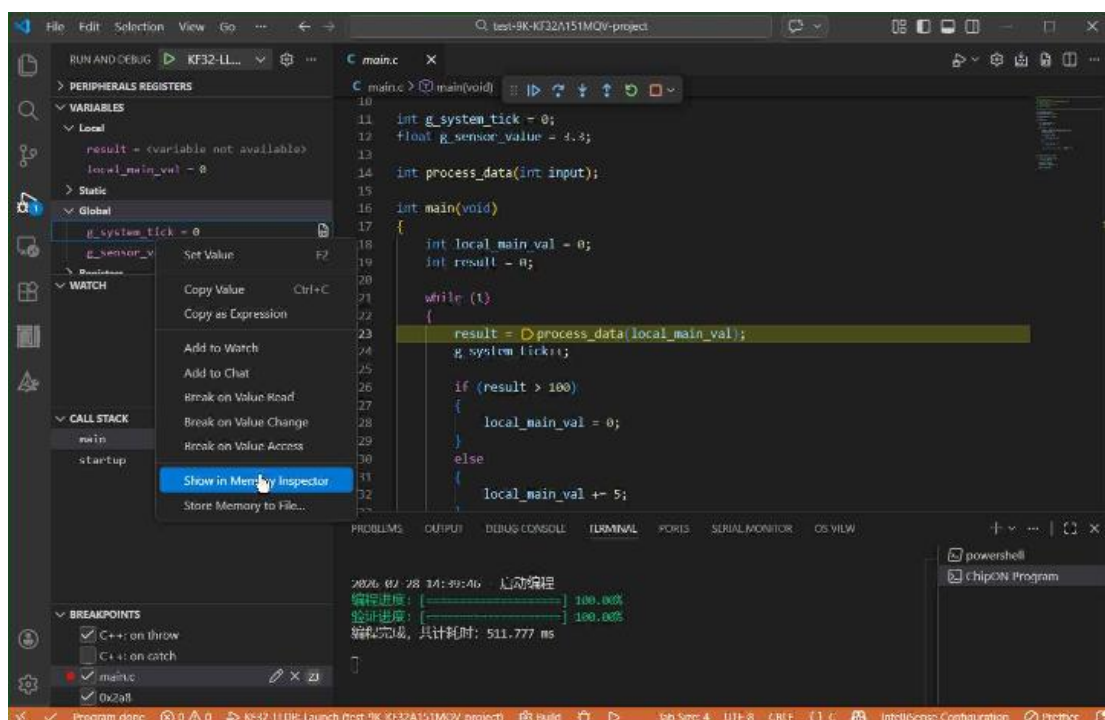


图 5-49 打开内存视图操作

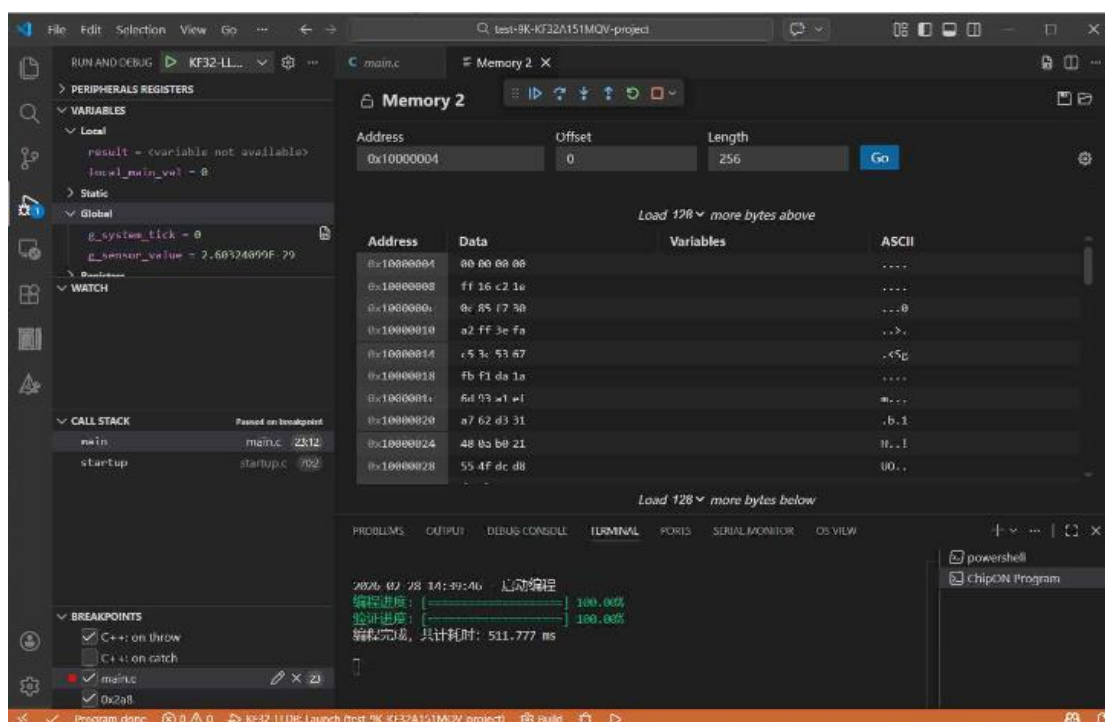


图 5-50 内存视图

5.6.5.6 Peripheral Registers (外设寄存器视图)

- 功能：显示微控制器（MCU）内部特殊功能寄存器（SFR）的值。
- 特点：
 - ◆ 硬件映射：将内存地址映射为具体的硬件功能名称（如 GPIOA_MODER, RCC_PLLCR）。
 - ◆ 位域解析 (Bitfields)：不仅显示寄存器的整数值，还会将每个比特位 (Bit) 拆解显示其含义（例如：Bit 0 = Enable, Bit 1 = Interrupt Mask）。
 - ◆ 实时读写：允许开发者直接修改寄存器位来控制硬件（如点亮 LED、重置外设），而无需重新编译代码。
- 外设寄存器视图在 Debug 界面中，展开 Peripherals Registers 栏即是。外设寄存器视图默认会展示所有外设寄存器组的信息。可以选择界面右上角的过滤按钮，选择需要的外设寄存器组进行展示。

Tips:

外设寄存器视图仅在调试过程中且处于暂停时才可显示数据。

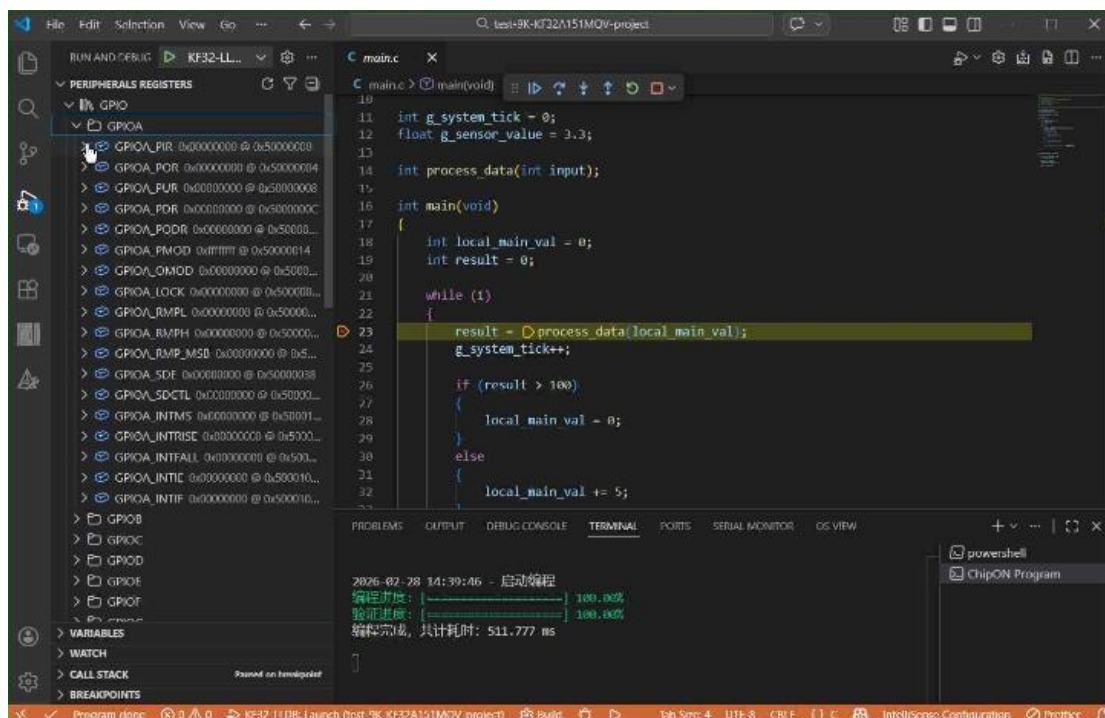


图 5-51 外设寄存器视图

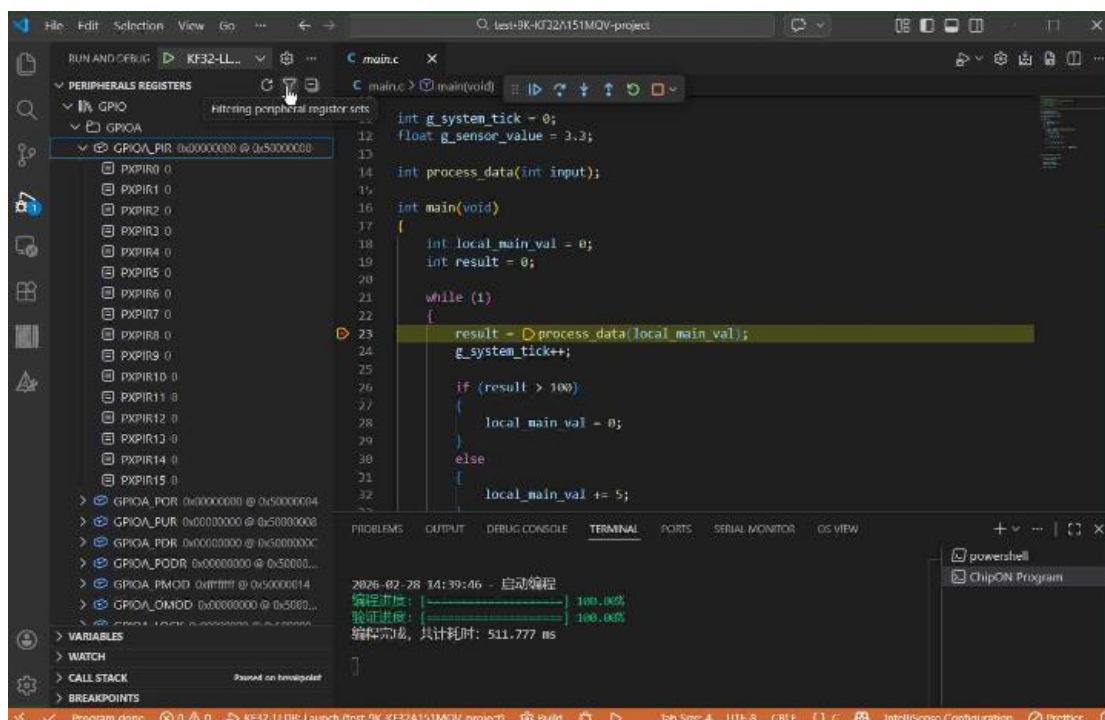


图 5-52 过滤外设寄存器

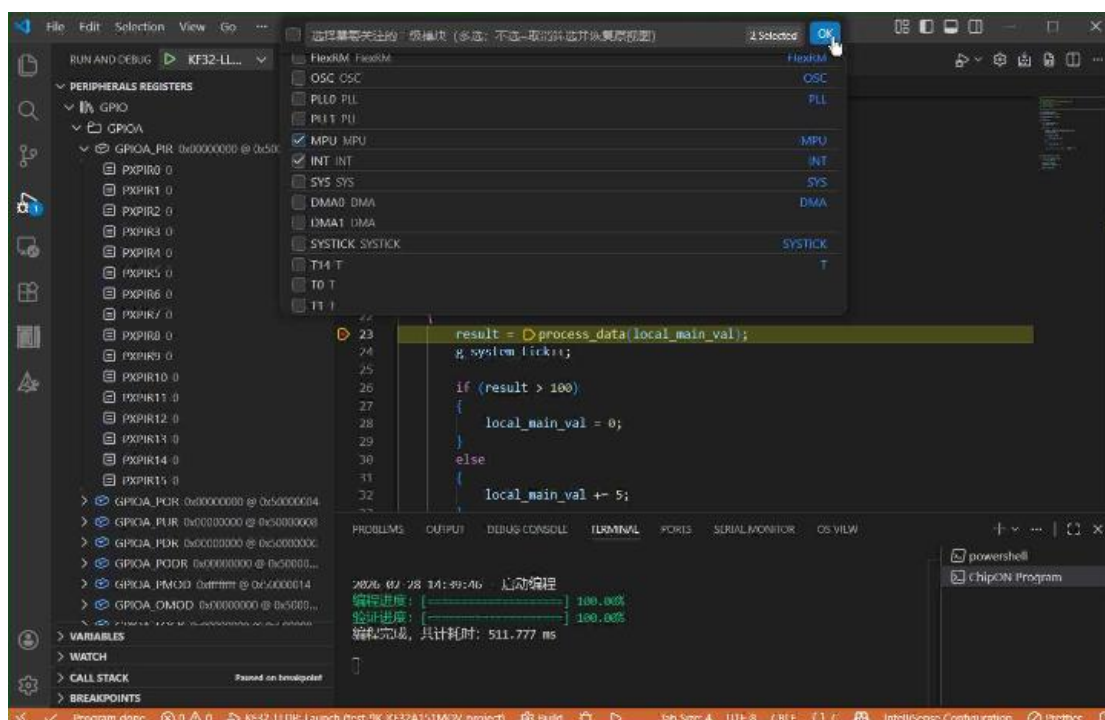


图 5-53 选择需要显示的外设寄存器

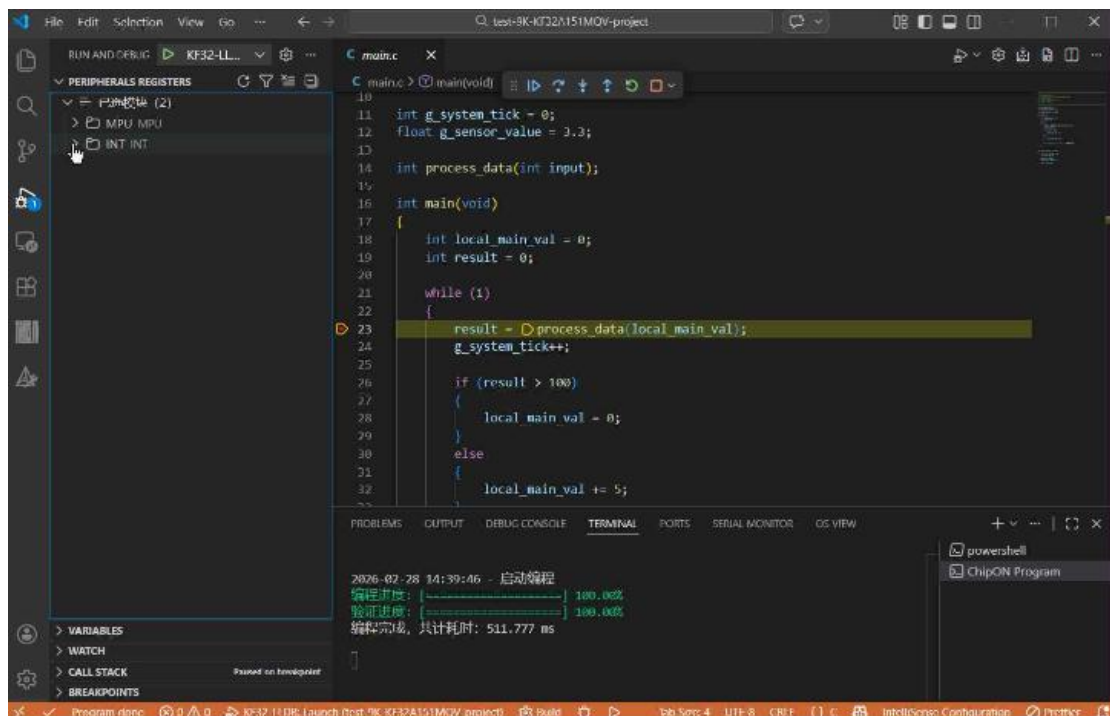


图 5-54 过滤外设寄存器后显示效果

5.6.5.7 Disassembly (反汇编视图)

- 功能：将当前执行位置的机器码还原为可读的汇编指令，并支持与源代码进行对照显示。它展示了 CPU 真正执行的底层操作，而非编译器抽象后的代码逻辑。
- 特点：
 - ◆ 指令级单步调试：在此视图中，Step Over (F10) 和 Step In (F11) 的操作粒度从“源代码行”变为“单条汇编指令”。这对于分析编译器优化（如循环展开、函数内联）后的真实执行流至关重要。
 - ◆ 地址与偏移：每一行都标有内存地址（如 0x400520）和相对于函数入口的偏移量（如 main+42），方便与崩溃日志（Core Dump）或内存视图中的地址进行精确匹配。
 - ◆ 机器码字节：显示指令对应的原始十六进制字节码（如 48 89 e5），用于逆向分析。
- 反汇编视图可通过两种方式唤出：1. 在 Call Stack 视图中，右击对应 Frame 的 Open Disassemble View 按钮，2. 在调试过程中，由 ChipON VSCode Extension 主动唤起。

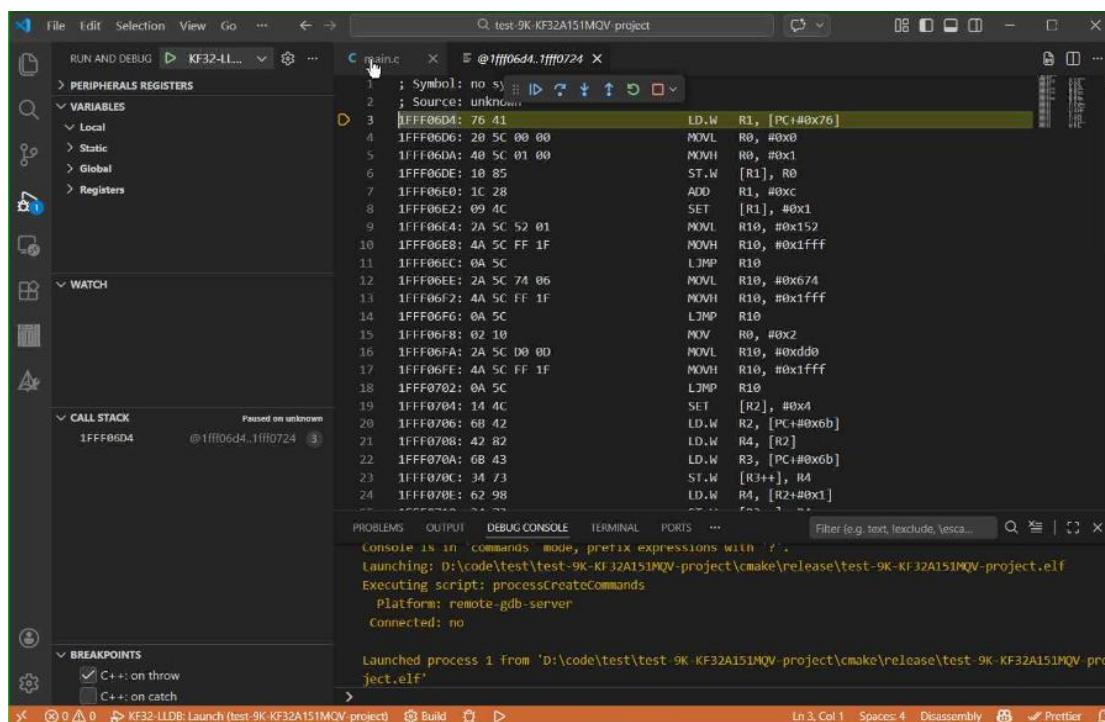


图 5-55 反汇编视图

5.6.5.8 OS View (OS 视图)

- 功能：专门用于实时监控和分析基于 FreeRTOS 操作系统的嵌入式应用程序状态。通过读取目标芯片内存中的 FreeRTOS 内核数据结构（如 TCB 链表、队列控制块等），将其可视化展示。这个视图包含了 Task、Timer、Queue 和 Dump 四个子视图。
- OS View 视图位于底部栏。OS View 视图数据仅在调试状态且处于暂停的情况下才会出现数据。

Tips:

OS 视图仅在调试过程中且处于暂停时才可显示数据。

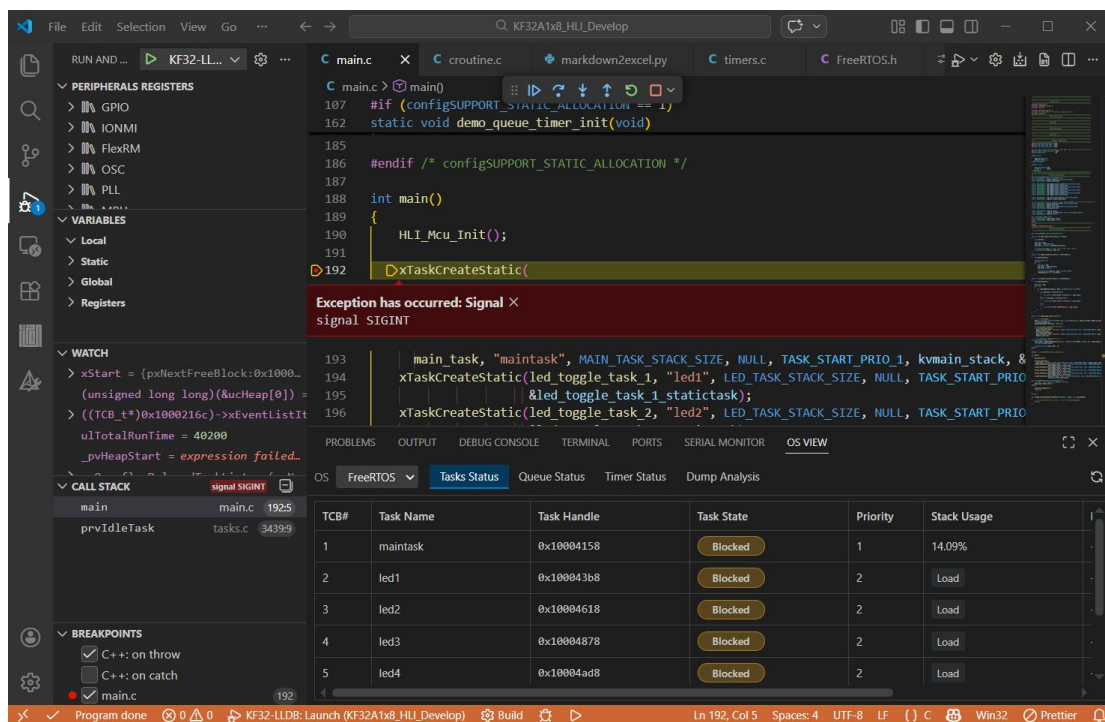


图 5-56 OS 视图

5.7 运行时监控变量

首先，打开变量监控视图：

1. 在底部面板区域，点击“Variable Monitor”标签页；
2. 视图分为三个区域：左侧边栏 — 变量列表（添加/删除/显隐管理）、中间图表区 — 时间序列折线图、底部状态栏 — 监控状态、上次更新时间。

Tips:

如果看不到底部面板，可通过菜单 View → Appearance → Panel Position 调整，或使用快捷键 Ctrl+J 打开。

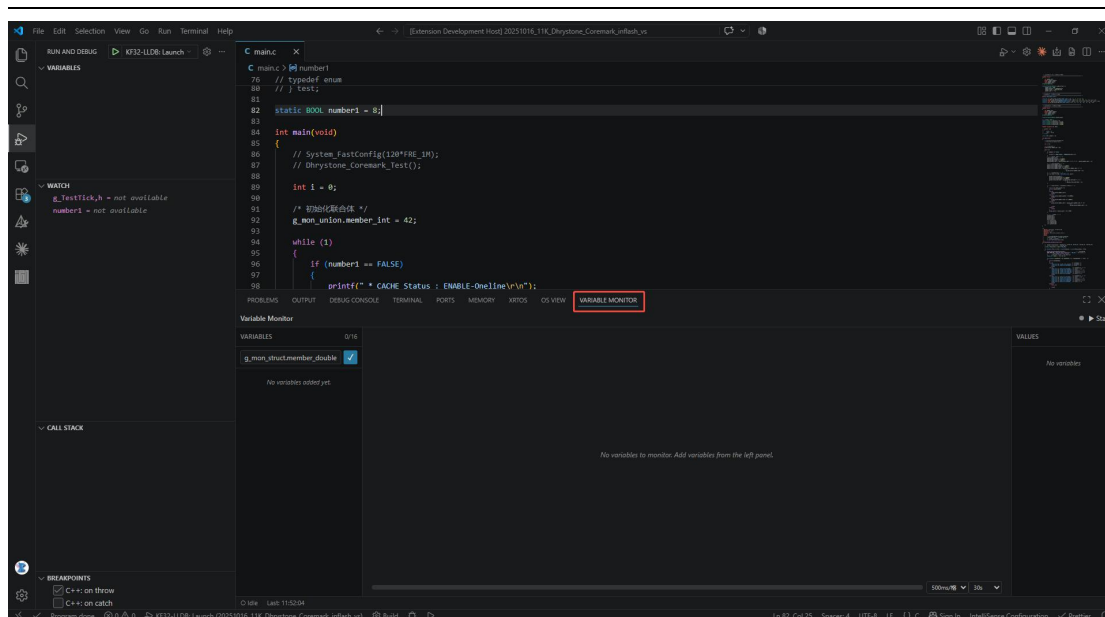


图 5-57 运行时监控变量视图

然后，启动调试会话。变量监控依赖活跃的调试会话，使用前必须先启动调试：

1. 确保项目已完成编译（生成 .elf 文件，需带 -g 调试符号）
2. 按 F5 或点击侧边栏调试按钮，选择 kf32-lldb 配置启动调试
3. 调试会话启动后，Variable Monitor 面板的 Start 按钮变为可用

Tips:

若未启动调试直接点击 Start，会弹出提示：“No active debug session. Please start a debug session first.”

状态栏将显示 “Idle”，表示当前无监控活动。

调试会话结束后监控自动停止，已采集的历史数据可在面板中继续浏览。

添加监控变量。支持两种方式添加变量：

方式 A：手动输入

1. 在左侧边栏顶部的输入框中输入变量名，例如 g_counter
2. 支持直接输入表达式，如 g_struct.field、g_array[3]
3. 按 Enter 或点击右侧 ✓ 按钮确认添加
4. 每个变量自动分配一种颜色，最多同时监控 16 个变量

方式 B：从下拉列表/树形结构中选择（推荐）

1. 点击输入框，系统自动加载可监控的变量列表
2. 若 DWARF 调试信息可用，将显示树形层级结构
3. 点击叶子节点，变量名自动填入输入框，确认后加入监控列表
4. 支持输入关键字过滤（按变量名或类型匹配高亮显示）

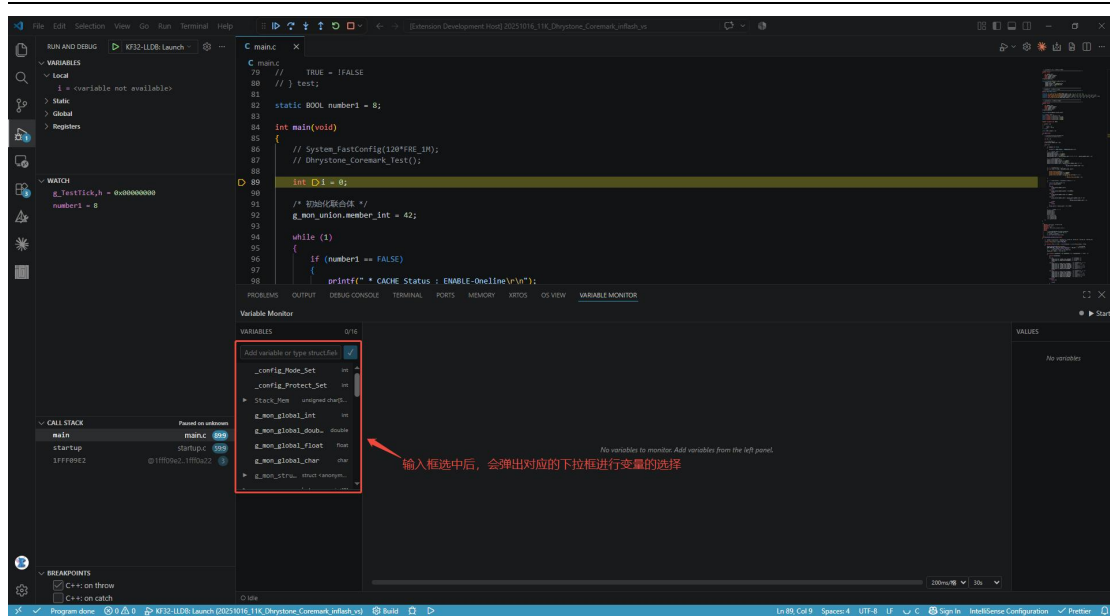


图 5-58 添加运行时监控变量

Tips:

只有带有 DWARF 类型信息的变量才会出现在树形列表中；链接器生成的符号（如 `__heap_end`、`__bss_start`）会被自动过滤。

若下拉列表为空并提示错误，请检查：① 项目是否以 `-g` 编译；② 调试会话是否已启动。

已添加的变量不会重复出现在下拉列表中。

启动/停止监控

1. 添加至少一个变量后，点击工具栏右侧的“► Start”按钮
2. 监控启动后：
 - 按钮变为“■ Stop”
 - 状态栏显示“● Monitoring”及上次数据更新时间
 - 图表区域开始实时绘制变量值曲线
3. 点击“■ Stop”停止监控，图表保留已采集的历史数据，可继续浏览分析

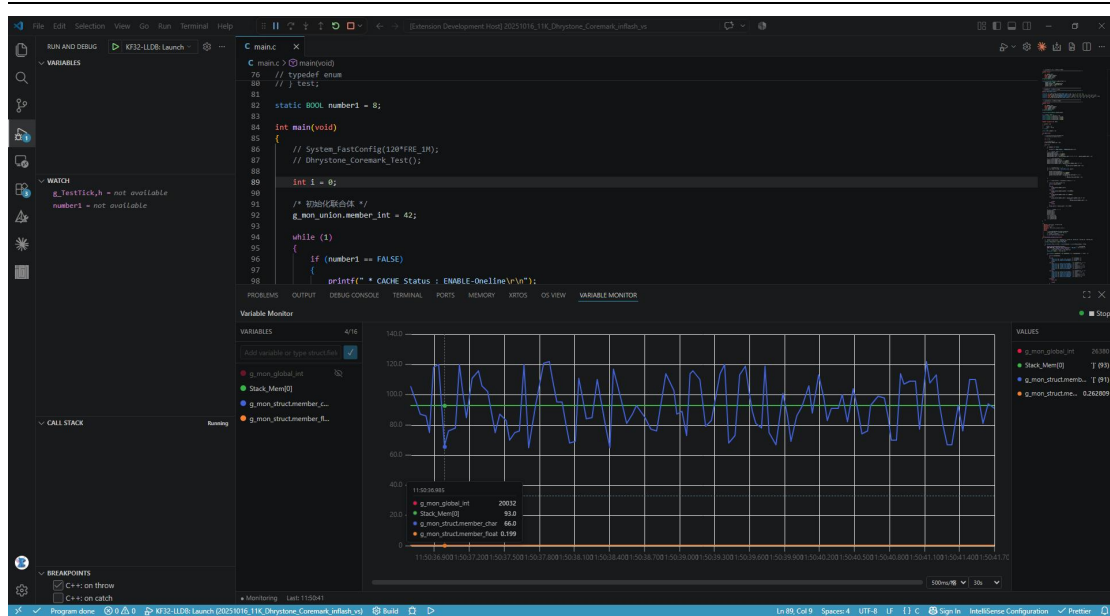


图 5-59 启动运行时变量监控

Tips:

如果调试目标正在运行（未停在断点），点击 Start 后状态栏会短暂显示 “⌛ Waiting for target to stop...”，当程序下一次停在断点时自动开始采集数据。

点击 Stop 后，数据仍缓存在面板中，可拖动滚动条查看历史。切换到其他视图再切回时数据不丢失。

如果选择的变量在运行时采集不到数据，会在左侧的数据栏中提示数据检测不到。

6. 常见问题与解决方法

6.1 安装常见问题与解决方法

Q1: 能否在类似 Cursor、Trae 的 AI IDE 中安装？

可以。目前 ChipON VSCode Extension 仅在 VSCode 官方市场上架。如果需要在类似 Cursor、Trae 的 AI IDE 中安装，参照离线安装的方式安装。

6.2 项目编译常见问题与解决方法

Q1: 点击 Build Project 没有任何反应或报错？

1. 检查是否已安装并启用 CMake Tools 扩展。
2. 检查 Output 视图、控制台视图是否有报错信息。

Q2: 修改了 CMakeLists.txt 或 Presets 后, 构建似乎没有生效?

1. 构建流程通常会检测变化并触发 Configure。
2. 如果遇到异常, 可尝试手动执行 Clean Project 后再 Build。
3. 或者在命令面板执行 CMake: Configure 强制重新配置。

Q3: 源文件是如何被包含到工程中的?

Cmake 项目不同于 Eclipse 项目, 需要手动指定编译所需要的全部源文件, 当前在 `cmake/CMakeLists.txt` 文件中通过 `COMMON_SOURCES` 字段全部列出

Q4: 如何支持 C++?

C++ 默认处于关闭状态。需要在 `configPresets.json` 中将 `ENABLE_CPP` 设置为 `true`:

```
{
  "cacheVariables": {
    "ENABLE_CPP": true
  }
}
```

启用后, CMake 将启用 C++ 相关工具链配置。

Q5: 如何排除某个文件夹/文件, 使其中的源文件不参与编译?

由于当前需要手动指定所有源码文件, 不再需要排除功能。

Q6: 修改配置后编译结果不对, 如何重新 Configure?

当你修改了 `CMakeLists.txt`、`CMakePresets.json`、`configPresets.json`、`flags.cmake` 等配置文件后, 如果编译结果与预期不符, 建议按以下步骤操作:

方法 1: 重新 Configure (推荐首先尝试)

在 VS Code 中:

1. 按 `Ctrl+Shift+P` 打开命令面板
 2. 执行 "CMake: Configure", 或者在 CMake Tool Window 中点击 Configure 按钮。
- 这会重新解析所有 CMake 配置文件并生成构建缓存。

方法 2: 删除 CMake Cache (解决顽固缓存问题)

如果以上方法无效, 可以完全删除 `cmake` 输出目录后重新进行 build 操作。

Q7: 如何自定义编译选项 (如添加宏定义或警告开关)?

在 `CMakePresets.json` 对应 Preset 的 `cacheVariables` 中修改 `USER_C_FLAGS` / `USER_CXX_FLAGS`:

```
{
  "name": "Debug",
  "cacheVariables": {
    "USER_C_FLAGS": "-O0 -g -DMY_DEBUG_MACRO=1 -Wall",
    "USER_CXX_FLAGS": "-O0 -g -DMY_DEBUG_MACRO=1 -Wall",
    "USER_ASM_FLAGS": "-gdwarf-3"
  }
}
```

}

或者在扩展底部的功能视图找到 Build Configuration 面板，通过 Edit 按钮进行可视化编辑。推荐通过 USER_*_FLAGS 添加自定义选项，而不是直接修改 flags.cmake。这样可以保持系统默认 flags 和用户自定义 flags 的分离。

Q8: 链接失败 显示.text 和.data 重叠

新版工具链中“xx = 地址”写法属于绝对地址语义（不同于 GCC 的相对地址语义），必须保证落在 flash 范围内，否则链接失败。段内 0x0200/0x0400/0x0800/0x0C00 几处同理。

GCC ld 文件与 LLVM ld 文件对比如下：

```
SECTIONS
{
/*#####*/
.text :
{
    . = 0x0000;
    KEEP (*vector.o(.text*)) /* chip interrupt vector ,writed in file named
    KEEP (*vector.obj(.text*)) /*compatible*/
    KEEP (*vector.c.obj(.text*)) /*compatible*/
    KEEP (*vector.asm.obj(.text*)) /*compatible*/
    . = ALIGN(4);
    __vec_end__ = .;

    *(.text*) /* default function code space */
    /*-----*/
    *(.ctors*)
    *(.dtors*)
    /*-----*/
    . = ALIGN(4);
    PROVIDE_HIDDEN ( __preinit_array_start = .);
    KEEP (*(.preinit_array*)) /* class init function list space */
    PROVIDE_HIDDEN ( __preinit_array_end = .);
    . = ALIGN(4);
    /*-----*/
}
```

图 6-1 GCC ld 文件样例

```
SECTIONS
{
/*#####*/
.text :
{
    // . = 0x0000; /* 默认从 Flash 起始地址开始放置，即 ORIGIN(flash)，因此该行可省略 */
    // 注意：在新版工具链中，此类赋值表示绝对地址，而非 GCC 中常见的相对地址语义
    // 因此需确保该地址在 Flash 范围内：即 ORIGIN(flash)至 ORIGIN(flash) + LENGTH(flash)
    // 否则会导致链接失败
    KEEP (*vector.o(.text*)) /* chip interrupt vector ,writed in file named vector.s or
    KEEP (*vector.obj(.text*)) /*compatible*/
    KEEP (*vector.c.obj(.text*)) /*compatible*/
    KEEP (*vector.asm.obj(.text*)) /*compatible*/
    . = ALIGN(4);
    __vec_end__ = .;
    /*-----*/
    *(.text*) /* default function code space */
    /*-----*/
    *(.ctors*)
    *(.dtors*)
    /*-----*/
    . = ALIGN(4);
}
```

图 6-2 LLVM ld 文件样例

Q9: hex 文件尾部填充 0，实际 hex 大小与计算显示大小不符

现象：编译生成的 HEX 文件中会包含大量连续的“0”填充，导致体积偏大。

原因：

在 GCC 的链接脚本中（.ld 文件），通常将 .data、.bss、堆栈（全部整合在同一个 .data 输出段中。这种结构会导致链接器为 .bss、堆和栈这些本不应占用 Flash 存储空间的段，也在加载地址（LMA）中分配了空间。最终结果是，生成的 HEX 文件中会包含大量连续的“0”填充，以补足这些虚拟地址，从而显著增大 HEX 文件的体积，既浪费存储资源，也影响固件的烧录效率。

解决方案：

为解决此问题，建议对链接脚本进行如下重构：

1. 从 .data 段中拆分出 .bss (NOLOAD) 独立段（纯 RAM，不再占 flash）
 2. 从 .data 段中拆分出 .heap_stack (NOLOAD) 独立段（含 .userdata、堆、栈，纯 RAM）
- GCC ld 文件与 LLVM ld 文件对比如下：

```

/*#####*/
.data : AT(__text_end__)
{
    __data_start__ = .; /* address must aligned by 4 bytes under zero offset */
    /*-----*/
    KEEP (*(.ramvector*)) /* default server for vector written in ram */
    /*-----*/
    __ldata_start__ = ALIGN(4);
    *(__ldata*) /* Reset but Keep Space */
    /* . = 0x8000 ; */ /* Max length limit is 32K byte ,if need */
    __ldata_end__ = ALIGN(4);
    /*-----*/
    | . = ALIGN(4);
    *(__ldata*) /* server space for ram function */
    *(__inrodata*)
    *(__inrodata*)
    /*-----*/
    | . = ALIGN(4);
    *(__data*) /* global variable with initialization */
    | . = ALIGN(4);
    __data_end__ = .;
    /*-----*/
    | . = ALIGN(4);
    __bss_start__ = .; /* global variable with no initialization */
    *(__COMMON*)
    *(__bss*)
    | . = ALIGN(4);
    __bss_end__ = .;
    /*-----*/
    ... .. |
    /*-----*/
    __initial_sp = ORIGIN(ram) + LENGTH(ram);
    PROVIDE(__Heap_length__ = 0x400 );/* defaultl size */
    /*-----*/
} > ram
    
```

图 6-3 GCC ld 文件样例

```

/*#####*/
.data : [redacted]
{
    _data_start__ = .; /* address must aligned by 4 bytes under zero offset */
    /*-----*/
    KEEP (*(.ramvector*)) /* default server for vector written in ram */
    /*-----*/
    _lpdata_start__ = ALIGN(4);
    /*(.lpdata*) /* Reset but Keep Space */
    /* . = 0x8000 ; */ /* Max length limit is 32K byte ,if need 32K a
    _lpdata_end__ = ALIGN(4);
    /*-----*/
    | . = ALIGN(4);
    /*(.indata*) /* server space for ram function */
    /*(.inrdata*)
    /*(.inrodata*)
    /*-----*/
    | . = ALIGN(4);
    /*(.data*) /* global variable with initialization */
    | . = ALIGN(4);
    _data_end__ = .;
    /*-----*/
} > ram AT > flash
// 使用 AT > flash 写法, 比AT(__text_end__)更好

.bss (NOLOAD): /* bss , 纯 RAM, 不再占 flash */
{
    | . = ALIGN(4);
    _bss_start__ = .; /* global variable with no initialization */
    /*(COMMON*)
    /*(.bss*)
    | . = ALIGN(4);
    _bss_end__ = .;
    /*-----*/
} > ram

.heap_stack (NOLOAD) : /* 堆 + 栈, 纯 RAM, 不再占 flash */
{
    ...
    __initial_sp = ORIGIN(ram) + LENGTH(ram);
    PROVIDE(__Heap_length__ = 0x400 );/* defaultl size */
    /*-----*/
} > ram

```

图 6-4 LLVM ld 文件样例

Q10: 栈指针异常导致的运行时错误

现象: 程序运行时, 栈指针 (SP) 被错误地指向了 Flash 地址空间, 而非预期的 RAM 区域。这导致函数调用前后硬件自动执行的压栈/出栈操作 (保存和恢复寄存器现场) 实际在 Flash 上执行写入, 从而导致程序运行时错误, 引发程序跑飞、复位等异常行为。

原因及解决方案:

修改前: `__initial_sp = LENGTH(ram);`

修改后: `__initial_sp = ORIGIN(ram) + LENGTH(ram);`

LENGTH(ram) 只是 RAM 区域的长度, 不是实际的结束地址。正确的栈顶应该是 RAM 的起始地址加上长度, 即 `ORIGIN(ram) + LENGTH(ram)`。这样 SP 在启动时就能正确指向 RAM 的末尾 (栈向下增长), 确保所有栈操作均在可读写的 SRAM 中执行, 恢复正常的函数调用和中断响应机制。

Q11: 链接时 memmove 行为 crash

现象: 链接过程中发生 crash, 且堆栈中显示是 memmove 系统调用导致的。

原因及解决方案:

```
...  
__LMA_STACK_LIMITS__ = DEFINED(Stack_Mem)? (__initial_sp -  
( __Logic_Stack_Start__ - __Logic_Stack_End__ )): (. );  
. = __LMA_STACK_LIMITS__ ;  
  
__initial_sp = LENGTH(ram);  
...
```

原来的 gnu 脚本导致 __initial_sp 值出错, __LMA_STACK_LIMITS__ 值出错, 最终导致位置计数器出错。以下给位置计数器的赋值也会得到一样的结果。

```
...  
.data:  
{  
...  
s = 0x0;  
. = s;  
}  
...
```

发生这种情况时需要检查位置计数器的赋值。

6.3 程序下载常见问题与解决方法

Q1: 为什么在 Devices 栏中显示设备, 但下载时仍报找不到设备?

Devices 栏中的设备不会实时更新。如果遇到此类问题, 建议先检查设备链接情况, 然后点击 Devices 栏中的刷新按钮, 刷新设备情况。

Q2: 下载时遇到串口错误, 该怎么办?

建议首先检查设备链接情况(包括上位机和编程器的链接情况、编程器和芯片的链接情况), 然后通过 PRO 工具确认芯片是否存在加密或鉴权密钥。如果存在上述情况, 先使用 PRO 工具解除对应加密或鉴权密钥或在编程配置中添加鉴权密钥, 然后再进行下载。如无法解决问题, 请联系技术支持人员。

6.4 程序调试常见问题与解决方法

Q1: 为什么有时候会出现调试进入失败?

可以尝试在 launch.json 中对应的任务中设置 openocdReadyTimeout(详见第 5.6.1 节)以延长 openocd 的超时时长, 然后再次尝试启动调试。如果再次启动调试失败, 请联系技术支持人员。

Q2: 如果在进入调试前, 设置了大于 4 个断点, 启动调试后, 会出现什么样效果?

如果在调试前, 设置了大于 4 个断点, 那么在启动调试后, 会按照断点视图中的断点顺

序依次设置，通常为前 4 个断点生效。如果在调试中设置断点，断点个数计数按设置顺序设置断点。

7. 附录

7.1 版本更新记录

版本	日期	变动说明	其他
V1.0	2026-03-05	首个发布版本	
V1.1	2026-03-11	补充 5.4.2.1 命令行构建	
V1.2	2026-04-01	1. 调整 cmake 配置源码的方式说明。 2. 更新 debug 配置说明 5.6.1 。	
V1.3	2026-06-03	1. 更新 C++项目相关表述，见 5.3.1 2. 更新 reset 功能，见 5.6.3 3. 增加从 beta 版本升级到 rc 版本时编译需要修改的相关配置答复，见 6.2 Q8	
V1.4	2026-07-16	1. 补充 6.2 Q8 的代码改动规则 2. 高亮部分 TIPS	
V1.5	2026-08-04	1. 补充说明新增的 configure 按钮	
V1.6	2026-08-10	1. 调整本文介绍中调试下各视图说明为独立章节 2. 补充 6.2 Q8 的代码改动规则以及链接脚本规则 3. 增加运行时监控变量功能使用说明，见 5.7	
V1.7	2026-09-08	1. 更新运行时监控变量视图的功能描述 2. 修改 6.2 Q8 问题和解决方案描述，补充 Q9-Q11 问题关于编译时链接脚本描述	