
# QX-IDE 用户手册

v0.6.9

合肥乾芯科技有限公司

表 1：版本历史

| 版本号 | 日期 | 备注 |
|---|---|---|
| 0.0.1 | 2023/06/25 | 初稿 |
| 0.0.3 | 2023/08/15 | 更新 IDE 安装，新建工程，编译、链接，调试<br>添加实时刷新，Flash 烧写 |
| 0.1.0 | 2024/02/01 | 添加工具链内容 |
| 0.1.3 | 2024/03/25 | 添加 qxtools 路径说明<br>更新已验证支持的操作系统列表 |
| 0.2.0 | 2024/06/19 | 添加 Linux 系统环境相关内容<br>删除 Windows 系统对 Python 的依赖<br>添加 4.5.3. 实时刷新绘图、4.9. 示例工程迁移 |
| 0.2.1 | 2024/06/20 | 更新 4.5.3. 实时刷新绘图<br>添加附录 C. 已知问题 |
| 0.3.0 | 2024/08/23 | 更新 4.4. 调试<br>添加 4.8. 串口工具<br>更新部分图片及描述 |
| 0.3.5 | 2024/09/19 | 更新 2. QX-IDE 安装<br>添加 3. QX-IDE 升级 |
| 0.3.7 | 2024/09/25 | 更新 4.4.3. 双核调试，启动 core1 前须保证 core0 暂停执行 |
| 0.3.8 | 2024/09/27 | 添加 **2.4. FAQ** |
| 0.4.0 | 2024/11/25 | 更新 **2.4. FAQ**<br>更新 3.1. 在线升级（OTA）<br>添加 4.10. 升级应用程序工程 |
| 0.4.1 | 2024/12/03 | 添加 4.12. 编辑器自动保存 |
| 0.4.2 | 2024/12/05 | 更新 3.1. 在线升级（OTA） |
| 0.5.0 | 2025/04/22 | 添加 4.3.2. 编译优化、4.7.1. Flash 镜像 |
| 0.5.2 | 2025/05/13 | 添加 4.10.1. 升级失败的解决方法，4.10.2. 手动迁移应用程序，4.13. 迁移应用程序工程到 VSCode<br>添加表 6：模板名称对应关系 |
| 0.5.3 | 2025/05/16 | 添加 QX-IDE 安装路径限制 |
| 0.6.0 | 2025/06/17 | 添加 5. 支持更多的 JTAG 调试器 |
| 0.6.1 | 2025/06/30 | 更新 4.5. 实时刷新 |
| 0.6.2 | 2025/07/07 | 添加 4.14. 引脚功能规划辅助 |
| 0.6.4 | 2025/09/10 | 添加 4.7.3. 烧写工具命令行接口<br>变更工具链名称及该章节格式 |
| 0.6.5 | 2025/09/11 | 更新 4.10. 升级应用程序工程 |
| 0.6.6 | 2025/09/29 | 添加 4.7.2.1. 找不到 Flash 镜像文件的解决方法 |
| 0.6.7 | 2025/09/30 | 添加 4.11. 工程重命名 |
| 0.6.8 | 2025/12/16 | **2.4. FAQ** 添加内容<br>更新 4.3.3. 链接脚本，添加适合更多型号芯片的内容 |
| 0.6.9 | 2025/12/23 | 添加 6.2.6. 内联汇编 |

# 目　录

# 图目录

## 表目录

# 1. 概述

本文档介绍 QXC2000DSP 集成开发环境（QX-IDE）的安装、升级和使用方法。

# 2. QX-IDE 安装

当前已经验证能够安装并正常使用 QX-IDE 的操作系统如表 2 所示。如遇兼容性问题，请联系技术支持。

表 2：已经验证的可使用 QX-IDE 的操作系统

| 索引 | 操作系统 |
|---|---|
| 1 | Windows 10 22H2 64 位 |
| 2 | Windows 11 23H2 |
| 3 | Ubuntu 22.04 LTS |
| 4 | Ubuntu 24.04 LTS |

## 2.1. Windows 操作系统

**方式 1：.exe 安装包**

双击 QX-IDE_installer_v<version>.exe 执行安装程序，选择安装位置如图 1 所示。

**注意：**安装路径不能包含空格、中文及特殊字符。

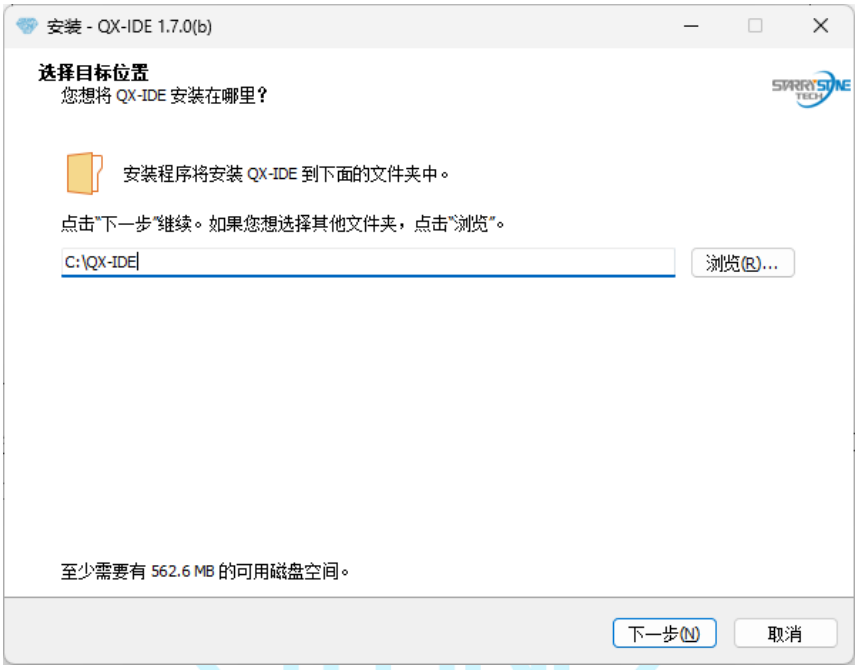



图 1：QX-IDE 安装路径选择

点击下一步，进行其它配置直到出现图 2 所示界面，点击安装。

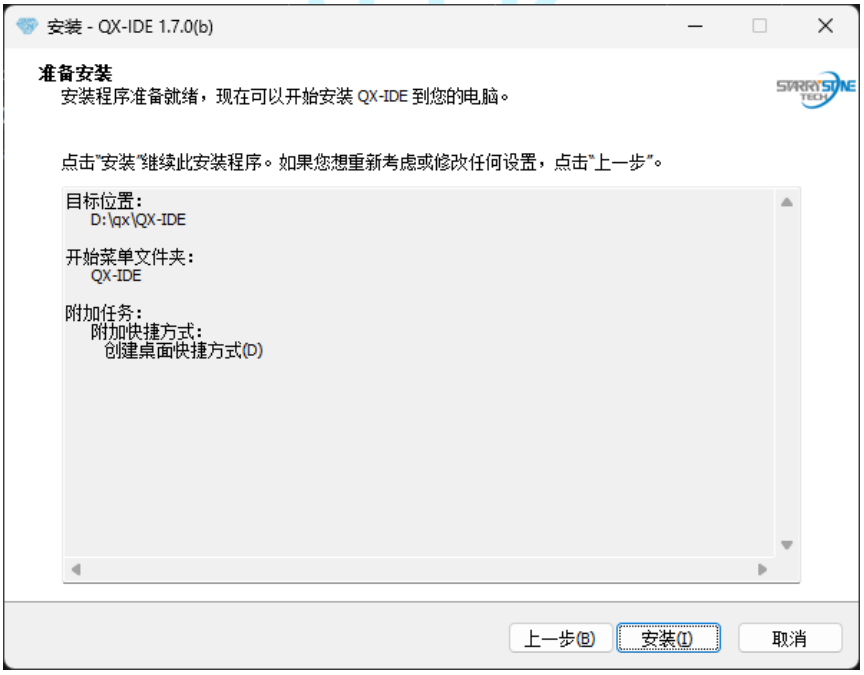



图 2：QX-IDE 安装准备界面

安装完成后，可直接运行 QX-IDE，如图 3 所示。也可以通过桌面图标（如果创建）或者双击安装位置下的 QX-IDE.exe 运行 QX-IDE。


图 3：QX-IDE 安装完成界面

**方式 2：.zip 压缩包**

在希望保存 QX-IDE 的路径下，解压 QX-IDE_ v<version>_windows_x86_64.zip。双击解压位置下的 QX-IDE.exe 运行 QX-IDE。

首次启动 QX-IDE 将进入欢迎界面，如图 4 所示。


图 4：QX-IDE 欢迎界面

## 2.2. Linux 操作系统

解包 QX-IDE_v<version>_linux_x86_64.zip 到任意目录，sudo 权限执行 QX-IDE 启动脚本 start-QX-IDE.sh。参考命令如下：

```
$ unzip xzvf QX-IDE_v<version>_linux_x86_64.zip
$ cd QX-IDE_v<version>_linux_x86_64/
$ sudo ./start-QX-IDE.sh
```

**说明**：使用 sudo 权限启动 QX-IDE 是为了让 IDE 能够卸载 Linux 内核自带的 JTAG 调试器 VCP 驱动程序，QX-IDE 在每次芯片调试前执行以下命令

```
$ sudo rmmod ftdi_sio usbserial
```

（参考 AN_220_FTDI_Drivers_Installation_Guide_for_Linux-1.pdf (ftdichip.com)，1.1 Overview）

## 2.3. 目录结构

QX-IDE 的根目录如图 5 所示，除 IDE 本身还包含以下主要部分

1. doc/：QX-IDE 用户手册（本文档），QXC2000DSP 指令集手册
2. plugins/：包含为 QX DSP 定制的 IDE 组件

   其中，plugins/QXTOOLS_<version>/qxtools/目录包含 DSP 开发工具链。包括编译器、链接器、调试器、Flash 下载工具等。这些工具统一由 QX-IDE 调用。

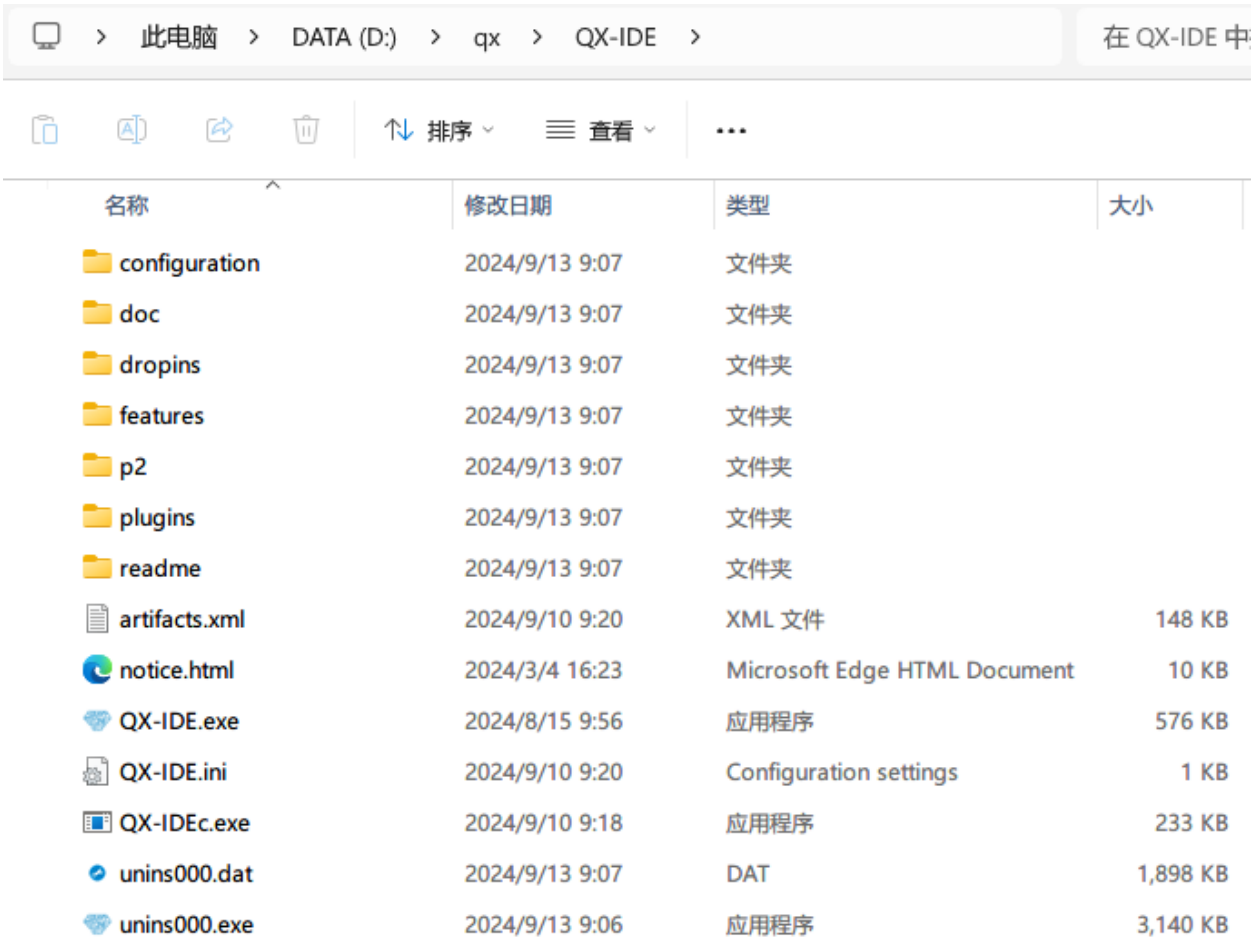



图 5：QX-IDE 根目录

## 2.4. FAQ

**说明：**无论遇到何种问题，建议首先参照 3.1. 在线升级（OTA）将 IDE 升级到当前最新版本。

**Q1：**使用 QX-IDE 调试芯片或者 Flash 烧写时，遇到以下错误：

```
Loading of library file "FTCJTAG.dll" failed!
```

**A1：**QX JTAG 调试器未被操作系统识别，从以下地址下载设备驱动程序并安装。

https://ftdichip.com/wp-content/uploads/2021/08/CDM212364_Setup.zip

正常情况下，Windows 设备管理器应该具有以下两个设备，如图 6 所示。


图 6：QX JTAG 调试器设备列表

**Q2：**IDE 升级后，原来的应用工程无法正常编译、调试。

**A2：**IDE 升级的新增特性和原版本不兼容，可以尝试以下方法

1. 在全新的 workspace 导入原工程
2. 在全新的 workspace 新建工程，并将原应用程序合入

**Q3：**debug 时，在全速运行的情况下，Live Expression View 中的“刷新”和“持续刷新”按钮变灰，如图 7 所示。

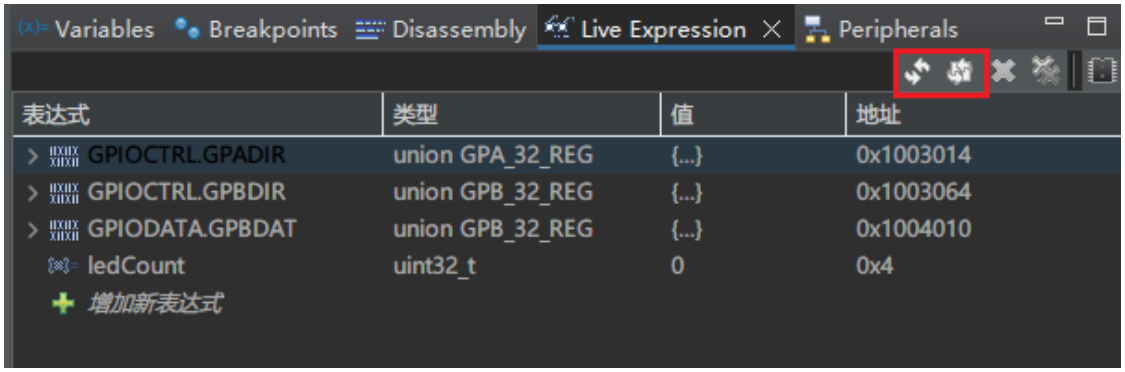



图 7：Live Expression View 刷新按钮不可操作

**A3：**未使用实时调试。

1. 参考 4.5. 实时刷新开启实时调试
2. 右键单击需要调试的工程，依次选择 Debug As ... -> QX Chip Live Debug

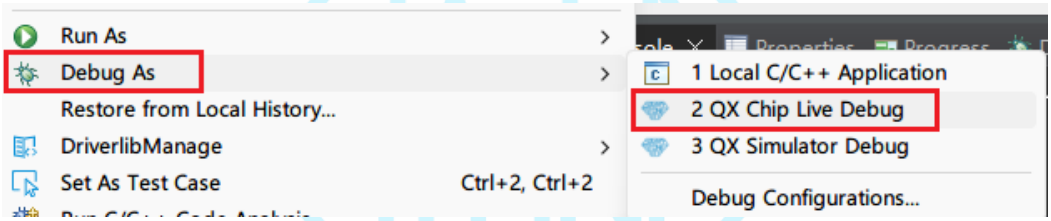



图 8：开始实时调试

**Q4：**在不支持硬件双精度浮点类型的芯片上使用 double 类型计算

**A4：**（说明：不建议进行该操作，这样会增加代码大小 并 影响代码执行效率）

1. 配置编译参数，新增“-mfpmath=fp64”，如图 9 所示。

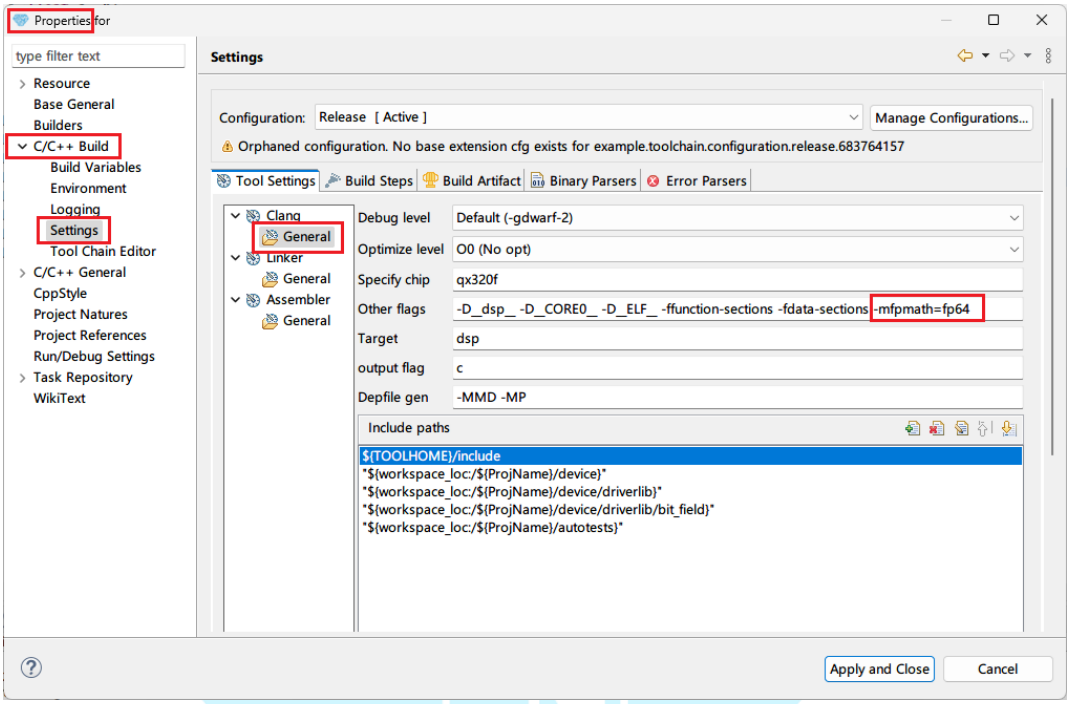



图 9：double 计算编译参数

2. 配置链接参数，新增链接路径“${TOOLHOME}/lib/fp64”，如图 10 所示。

**注意：**“${TOOLHOME}/lib/fp64”须在“${TOOLHOME}/lib“之上，保证优先链接 fp64 的库

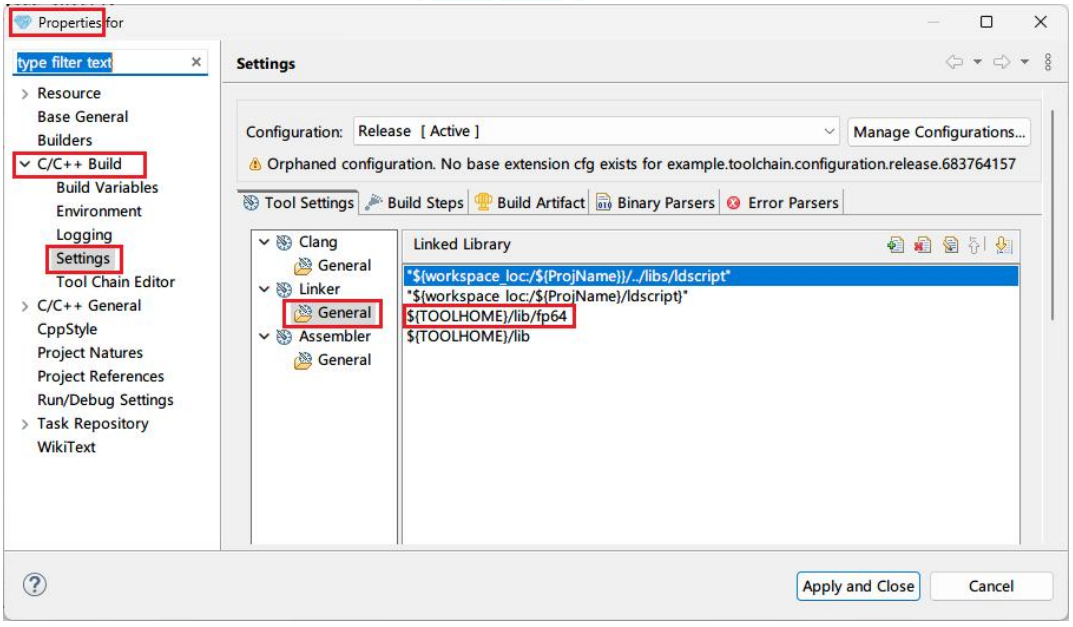



图 10：double 计算链接参数

# 3. QX-IDE 升级

建议用户始终使用最新版本的 QXC2000DSP 集成开发环境，确保其中的以下组件最新，以发挥 QXC2000DSP 的最大能力同时获得更好的用户体验。

1. 设备驱动程序
2. 编译器、汇编器、链接器
3. proxy/FTDI GDB Server：为 JTAG 调试器定制的 GDB Server
4. Flash 下载工具

## 3.1. 在线升级（OTA）

**说明**：请安装 QX-IDE v1.7.2 及以上版本以使用在线升级功能。

点击 Help -> Check for Updates，如图 11 所示。

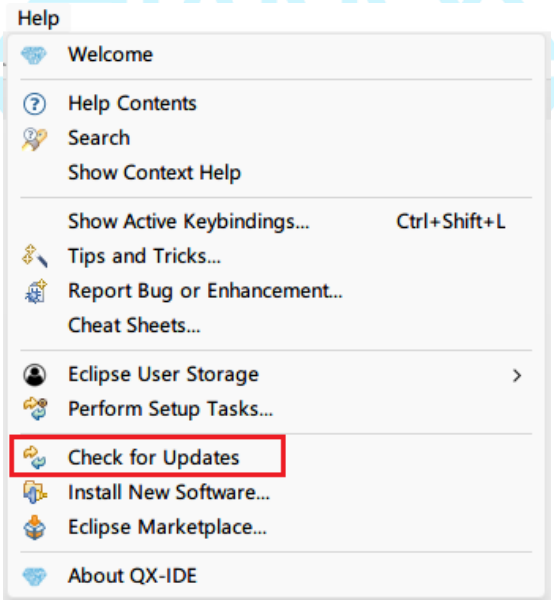



图 11：检查更新

如果有更新，则弹出如图 12 所示窗口，点击 Next >。

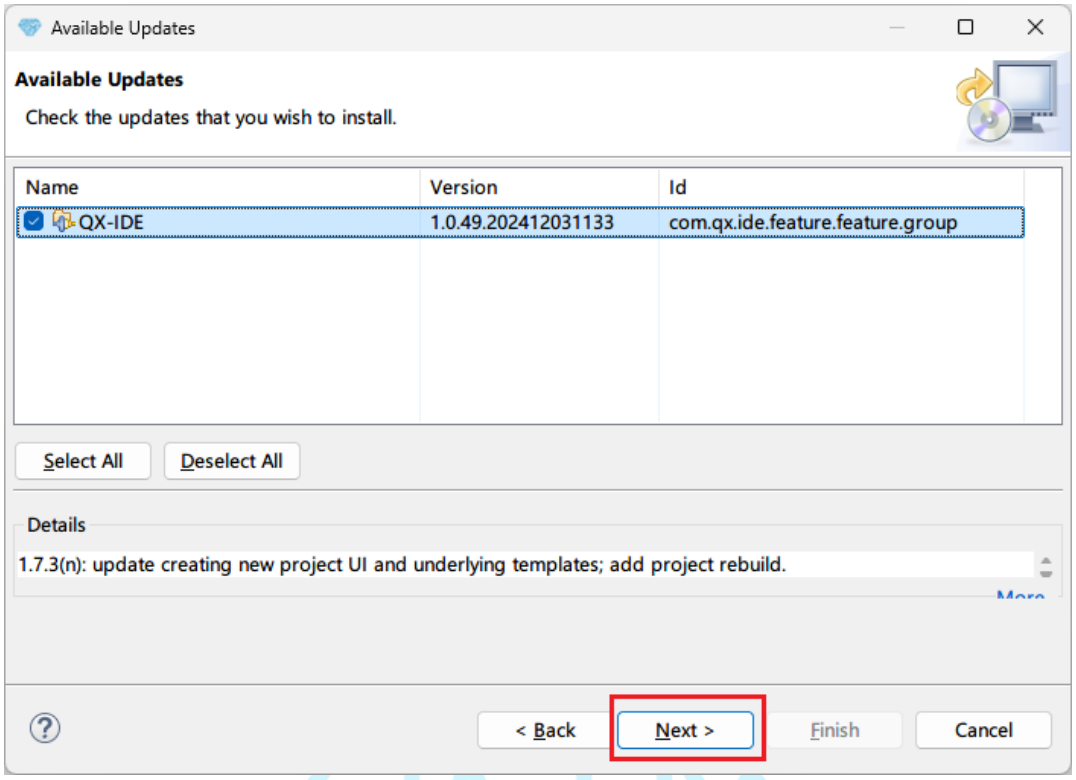



图 12：检查到有更新

在之后的一系列窗口直接点击 Next＞、选择“I accept the terms of the license agreement”、点击 Finish，并在图 13 所示窗口依次点击 Select All 和 Trust Selected，开始下载更新并等待下载完成。

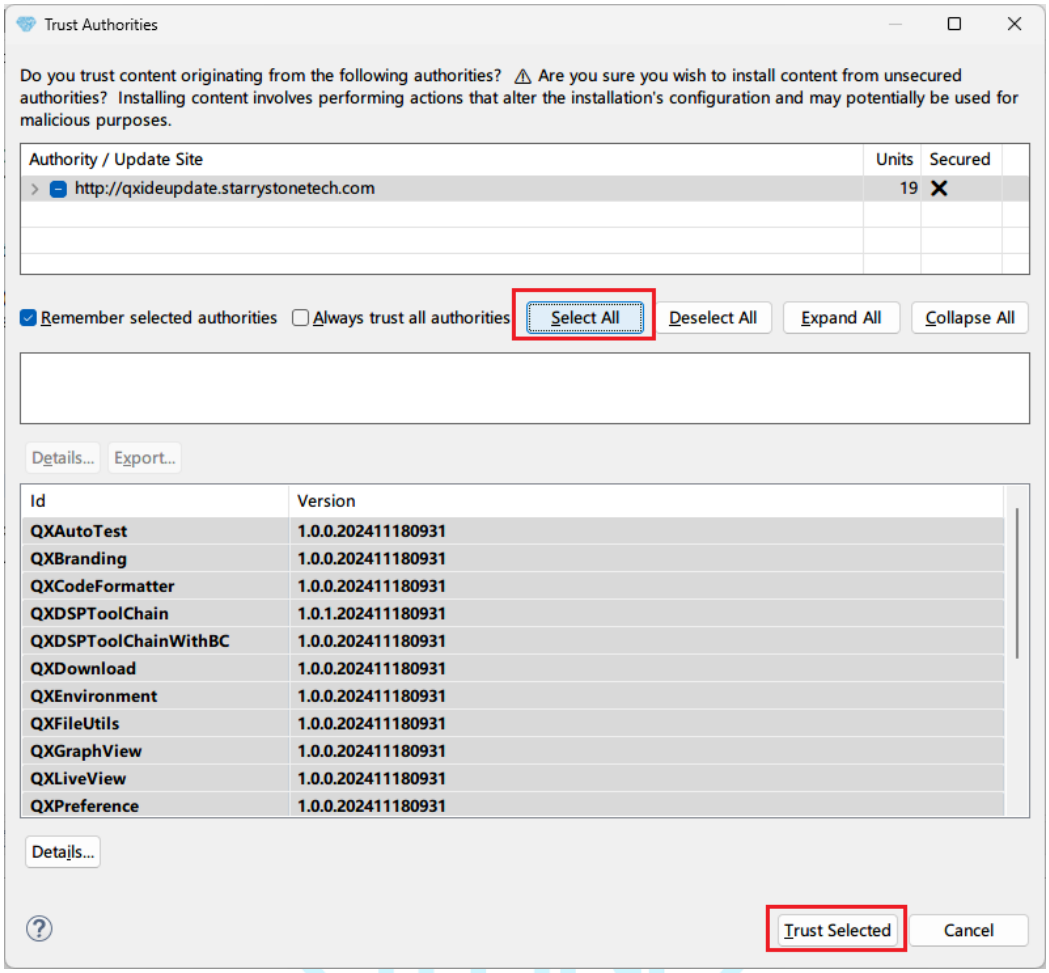

Trust Authorities
Do you trust content originating from the following authorities? ⚠ Are you sure you wish to install content from unsecured authorities? Installing content involves performing actions that alter the installation's configuration and may potentially be used for malicious purposes.
Authority / Update Site
Units
Secured
http://qxideupdate.starrystonetech.com
19
Remember selected authorities
Always trust all authorities
Select All
Deselect All
Expand All
Collapse All
Details...
Export...
Id
Version
QXAutoTest 1.0.0.202411180931
QXBranding 1.0.0.202411180931
QXCodeFormatter 1.0.0.202411180931
QXDSPToolChain 1.0.1.202411180931
QXDSPToolChainWithBC 1.0.0.202411180931
QXDownload 1.0.0.202411180931
QXEnvironment 1.0.0.202411180931
QXFileUtils 1.0.0.202411180931
QXGraphView 1.0.0.202411180931
QXLiveView 1.0.0.202411180931
QXPreference 1.0.0.202411180931
Details...
Trust Selected
Cancel

图 13：开始下载更新

在图 14 所示窗口依次点击 Select All 和 Trust Selected，开始安装更新并等待安装完成。

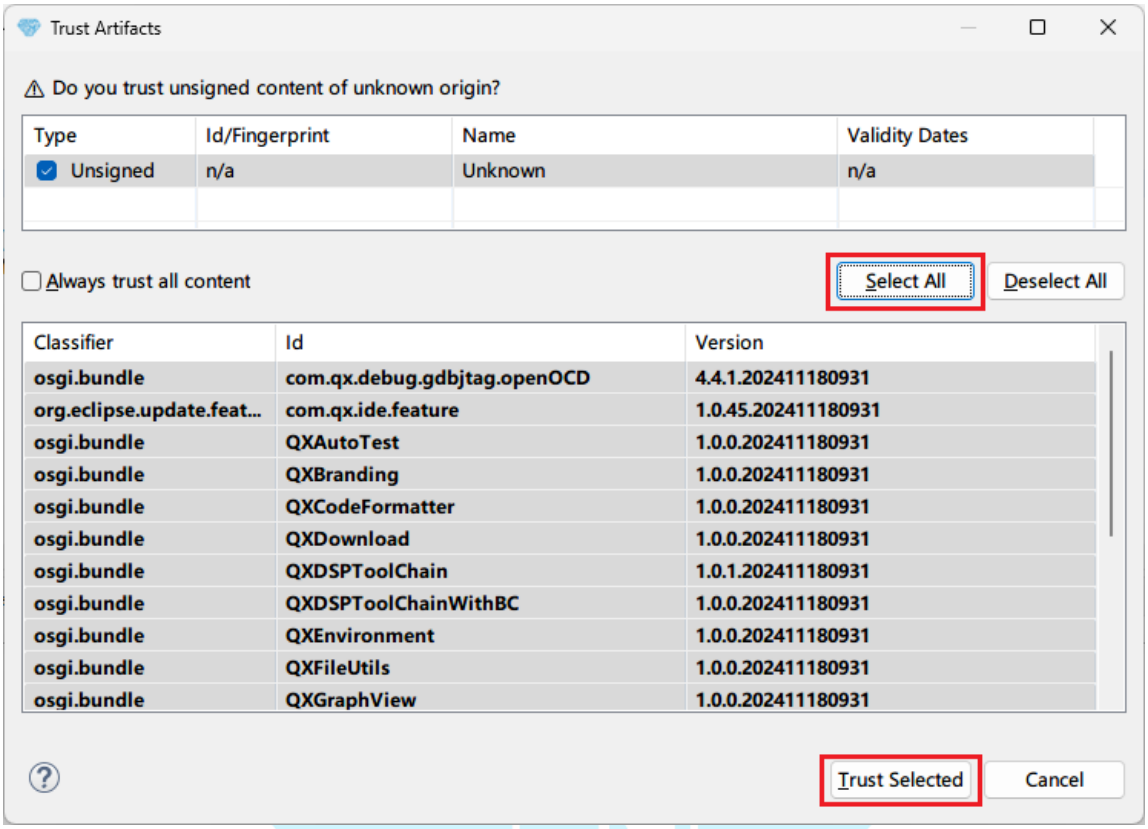



图 14：开始安装更新

重启 IDE 以完成更新。

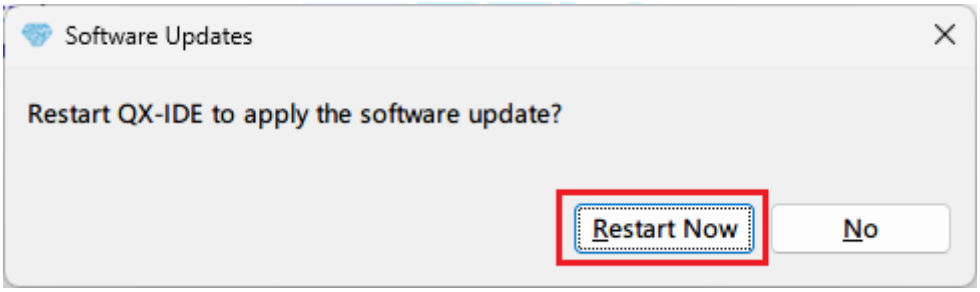



图 15：重启 IDE

## 3.2. 离线升级

当使用环境无法连接网络时，联系技术支持获取当前最新版本的 QX-IDE 安装包或压缩包。

# 4. QX-IDE 使用

## 4.1. 新建工程

点击 File -> New -> QX Project，选择 New QX C2000 Project，点击 Next >。

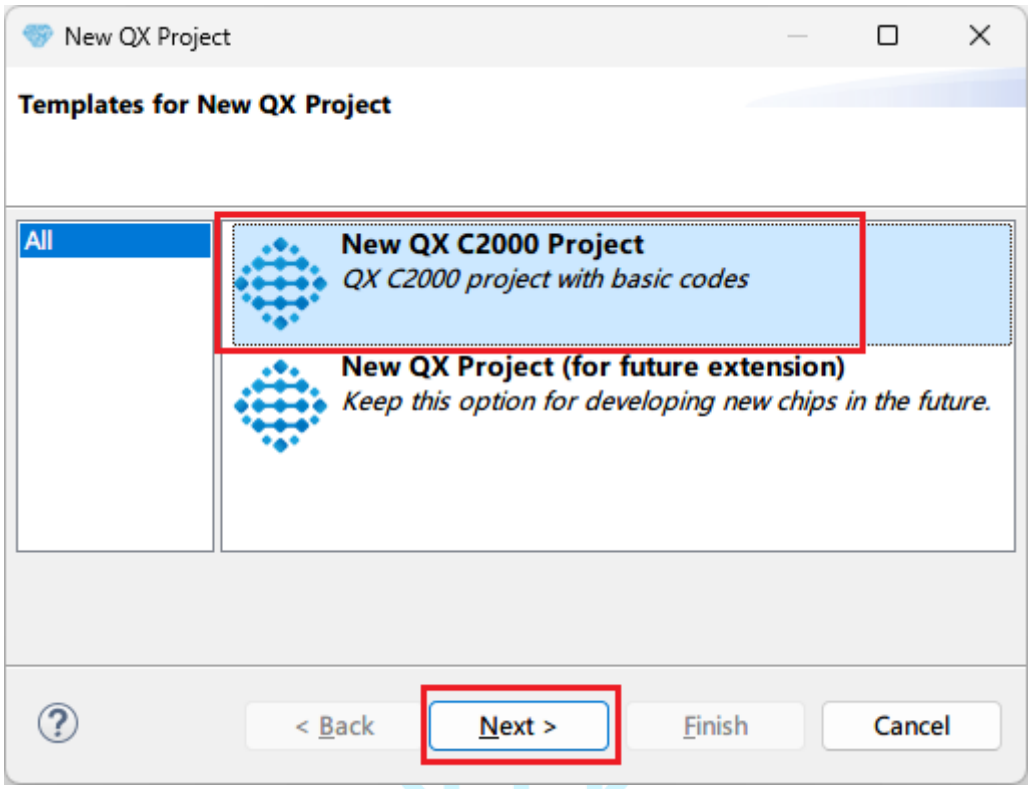



图 16：新建工程

填入 Project name，点击 Next >。如果新建工程目录下存在同名工程目录，会提示错误 Project already exist 并禁止点击 Next >。

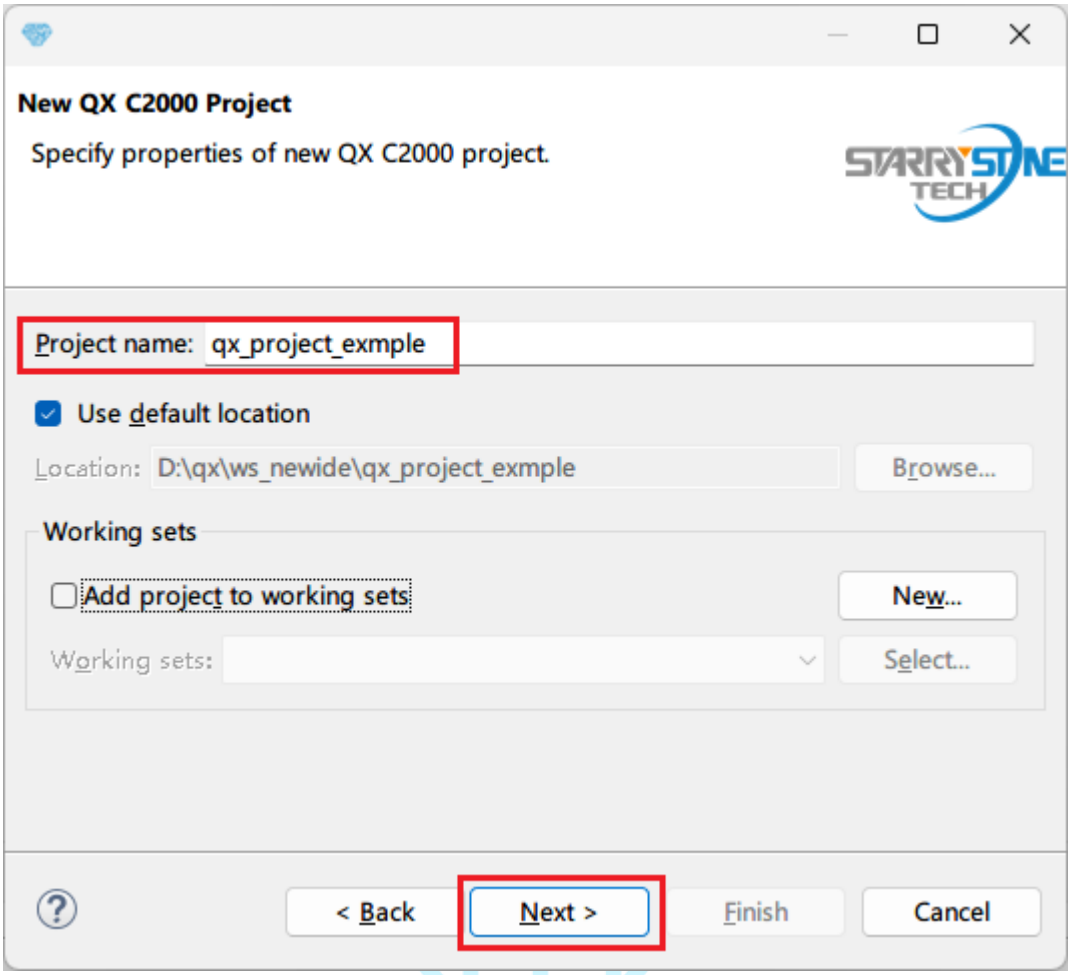



图 17：新建工程 2

在图 18 所示窗口，配置芯片型号，创建裸机工程或 RTOS 工程，创建单核或双核工程；工具链保持“QX C2000 ToolChain”，点击 Finish 完成新建工程。

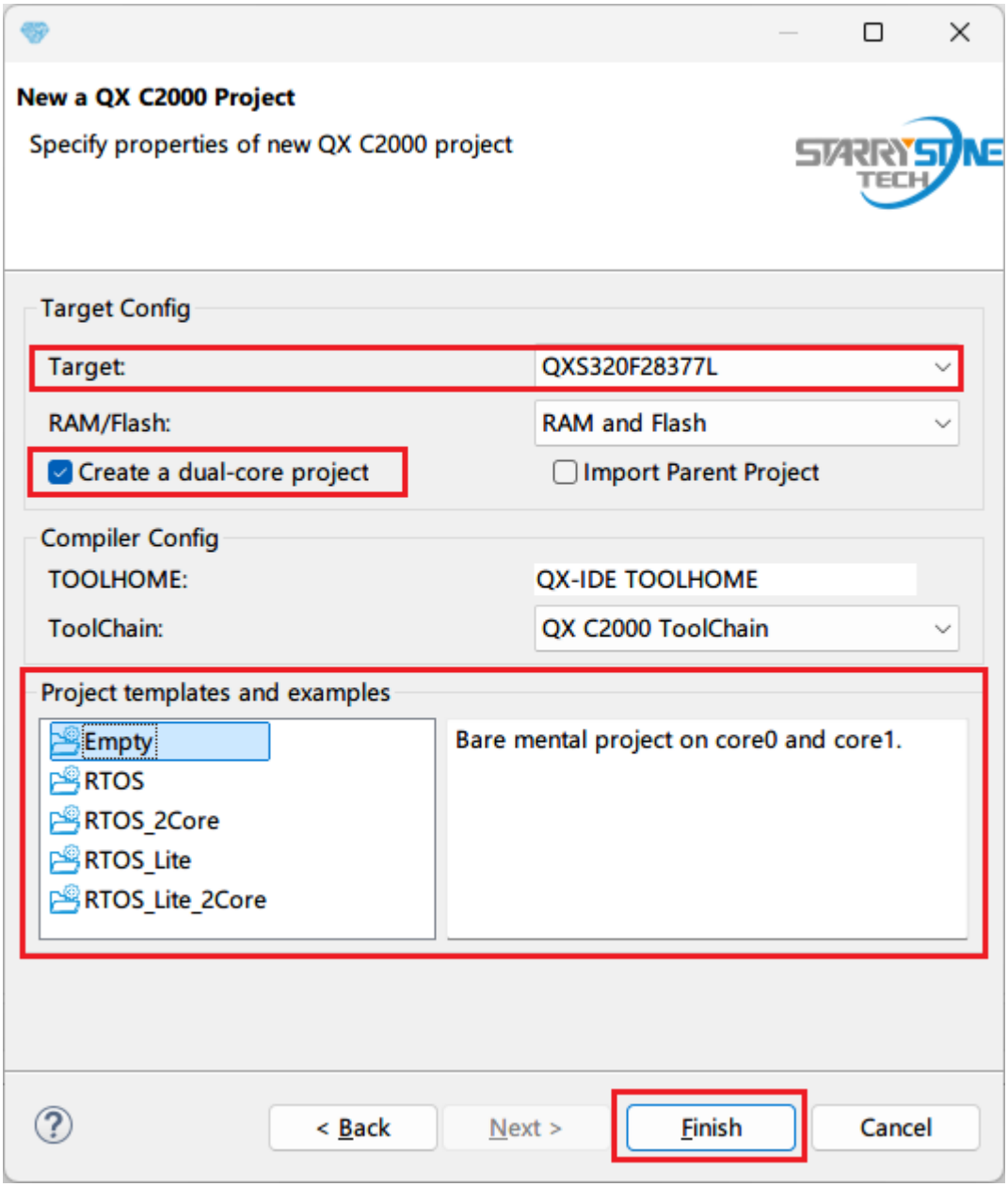



图 18：新建工程 3

#### 4.1.1. 工程目录结构

以裸机工程为例，新建双核工程后目录结构如图 19 所示。相关说明如表 3 所示。

对于双核 DSP，在新建工程时用户可根据具体应用需求选择仅使用单核（core0）还是使用双核（core0 和 core1）。

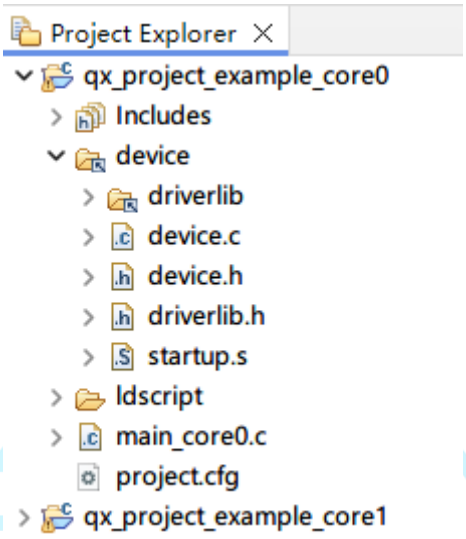



图 19：280049 工程目录结构

表 3：280049 工程目录说明

| 索引 | 目录/文件 | 说明 |
|---|---|---|
| 1 | device/driverlib | 驱动程序，支持比特位域访问 |
| 2 | device/device.h | 设备相关配置头文件<br>设置时钟频率，提供设备初始化函数接口 |
| 3 | device/startup.s | 中断向量表 和 DSP 内核启动代码 |
| 4 | ldscript/ | 链接脚本文件 |

## 4.2. 导入工程

点击 File -> Open Projects from File System ...。

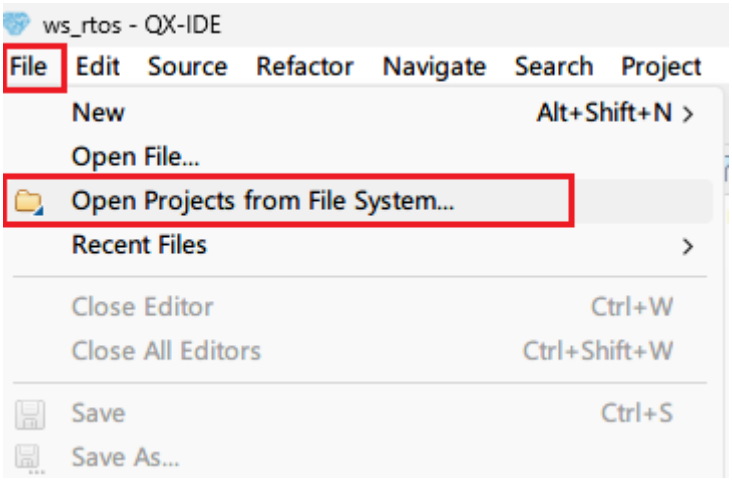



图 20：导入工程

点击 Directory...选择工程根目录，在 Folder 栏选择期望导入的工程目录，点击 Finish。

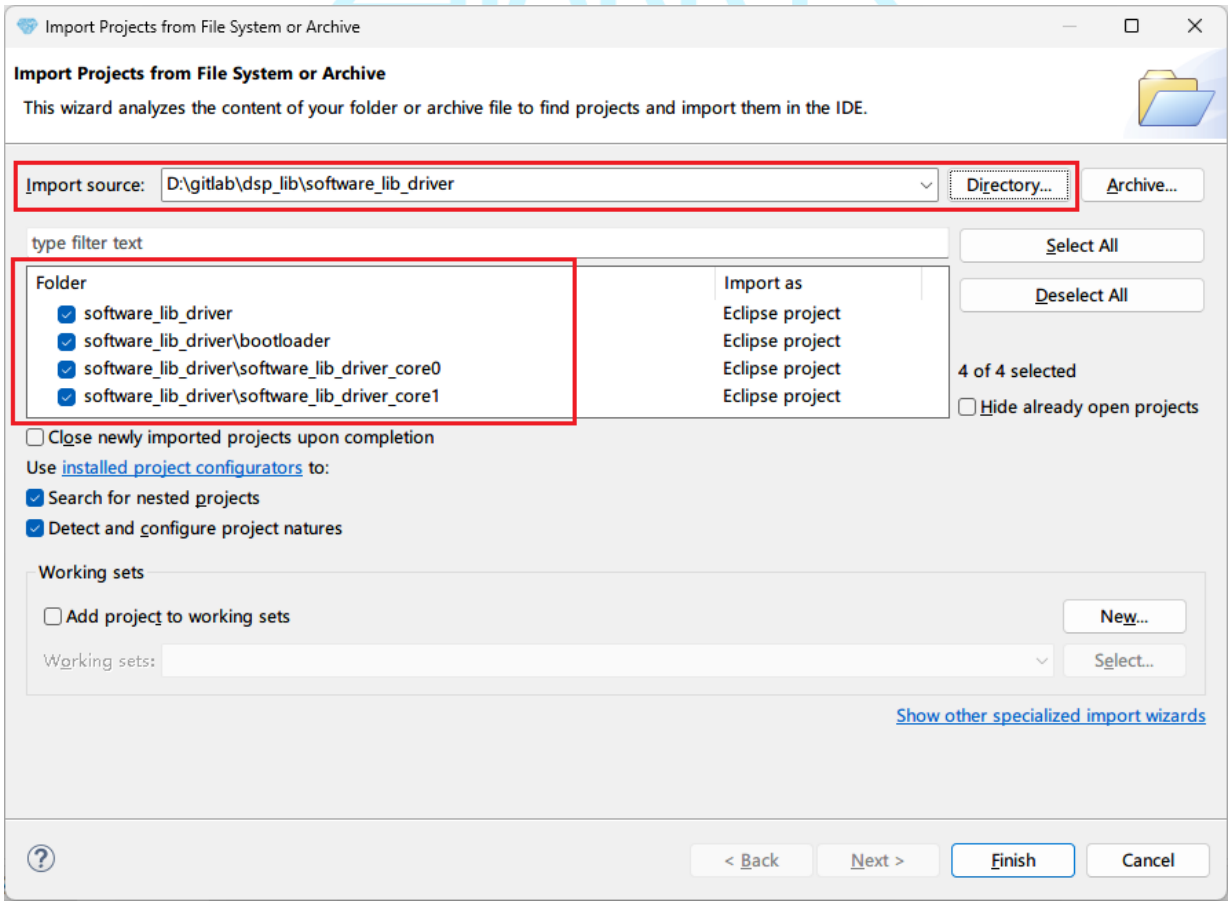



图 21：导入工程 2

## 4.3. 编译、链接

### 4.3.1. 工具链参数设置

Project Explorer 中的工程上右键单击，点击 Properties。选择 C/C++ Build -> Settings 即可进入编译器、链接器、汇编器参数设置界面，如图 22 所示。

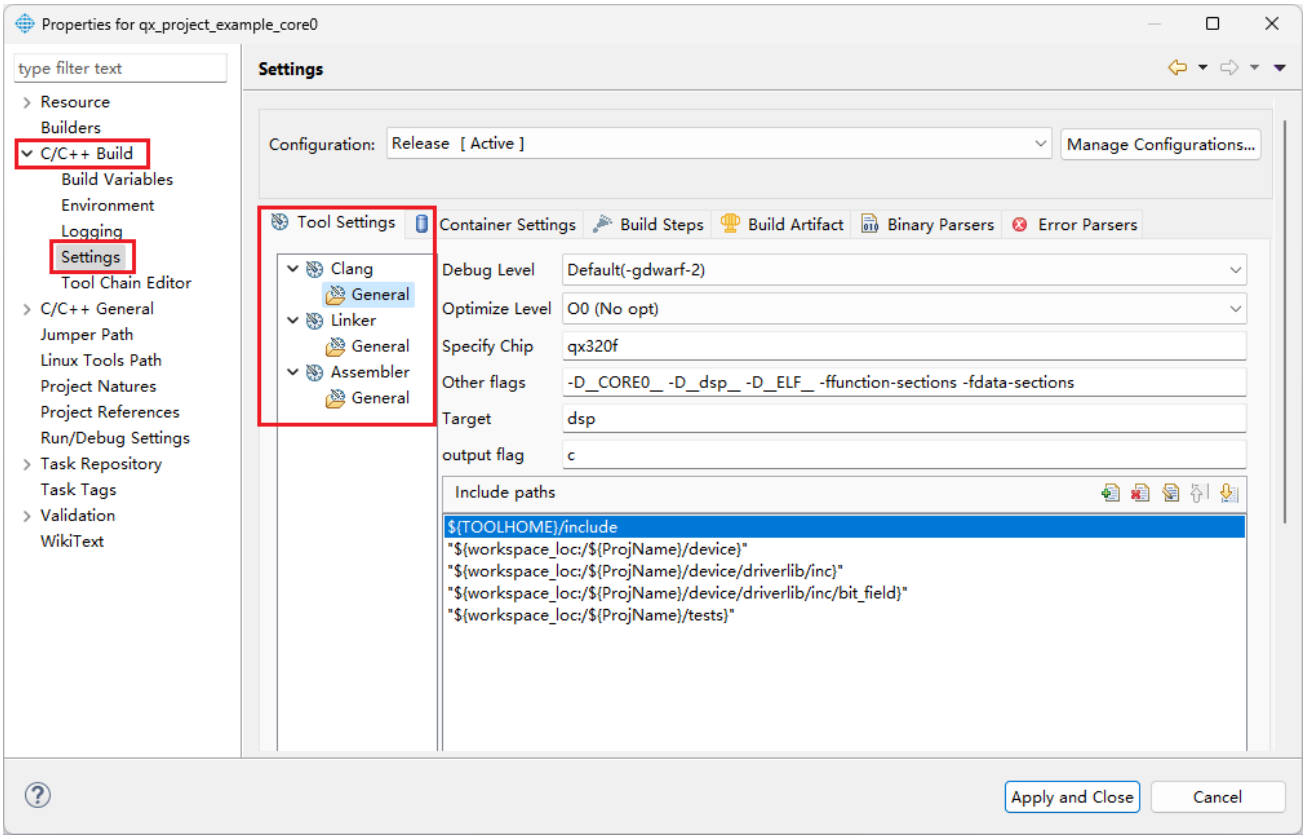



图 22：编译参数设置

### 4.3.2. 编译优化

本节介绍 4.3.1. 工具链参数设置中的编译优化等级（Optimize Level）。

#### 4.3.2.1. -O0

在 O0 等级下编译器只做了将 always_inline 属性的函数内联和消除不可达基本块的优化。O0 几乎没有做任何优化，不会影响到调试。

#### 4.3.2.2. -O1

与 O0 相比 O1 进行了全方面的优化，包括内存优化，冗余消除，保守的循环优化和轻量级的过程间优化等。O1 编译速度较快，debug 信息损失较少。

##### 4.3.2.2.1. 内存优化

O1 的内存方面的优化 PASS 主要有 sroa , mem2reg , memcpyopt , pgo-memop-opt。

- sroa (Scalar Replacement Of Aggregates)
  将结构体/数组分解为标量，消除结构体对齐填充导致的内存浪费，提升标量寄存器利用率。
- mem2reg
  将栈变量提升为寄存器，减少内存操作。
- memcpyopt (MemCpy Optimization)
  将多个在连续内存上的操作合并为 memcpy / memset 。
- pgo-memop-opt
  基于性能分析指导的优化（Profile-Guided Optimization, PGO） 数据，优化内存操作。

##### 4.3.2.2.2. 冗余消除

O1 的消除冗余的优化 PASS 主要有 early-cse , instcombine , adce , deadargelim , libcalls-shrinkwrap , bdce。

- early-cse (Early Common Subexpression Elimination)
  进行局部公共表达式消除,仅处理显式相同指令。
- instcombine (Instruction Combining)
  进行代数化简，常量折叠以简化表达式。

- adce (Aggressive Dead Code Elimination)

  基于控制流图分析删除无副作用且无用的指令。

- deadargelim (Dead Argument Elimination)

  删除未使用的函数参数。

- libcalls-shrinkwrap

  根据上下文将库函数换成更加高效的实现，比如内联实现或者特别的优化版本。

- bdce (Bit-Tracking Dead Code Elimination)

  通过位级别的分析消除与“无用位”相关的冗余代码。

#### 4.3.2.2.3. 循环优化

O1 优化循环的 PASS 主要有 licm , loop-unswitch , loop-unrool , loop-rotate , indvars , loop-deletion , loop-vectorize , loop-load-elim , loop-sink , loop-distribute。

- licm (Loop Invariant Code Motion)

  将循环不变量外提到循环外。

- loop-unswitch

  将循环内不变量条件提升至外层，生成多个循环版本。可能显著增加代码体积，O1 采用保守阈值。

- loop-unroll

  进行较为保守的循环展开。

- loop-rotate

  转换 while 循环为 do-while 结构，将条件判断移至尾部，减少一次分支跳转。

- indvars (Induction Variable Simplification)

  将循环索引替换为更简单形式，消除符号扩展操作。

- loop-deletion (Delete dead loops)

  删除无用的循环。

- loop-vectorize (Loop Vectorization)

  将循环迭代转换为向量操作。

- loop-load-elim

  消除循环内冗余的加载操作，利用缓存或寄存器暂存。

- loop-sink (Loop Sink)

将循环内部的计算指令“下沉”到循环的退出块，以减少循环迭代中冗余指令的执行次数。

- loop-distribute

  将单个循环拆分为多个独立循环，提升数据局部性。

#### 4.3.2.2.4. 过程间优化

O1 负责过程间优化的 PASS 主要有 ipsccp , globalopt , called-value-propagation , globaldce 。

- ipsccp (Interprocedural Sparse Conditional Constant Propagation)

  跨函数传播常量参数，支持稀疏条件传播，保留动态可能路径。

- globalopt (Global Variable Optimization)

  优化全局变量（如常量传播、删除未使用的全局变量）。

- called-value-propagation (Called Value Propagation)

  分析函数指针的静态目标。

- globaldce

  消除全局的死代码。

#### 4.3.2.2.5. 其他优化

O1 新增加的其他优化有 reassociate。

- reassociate

  重新排列算术操作以利用指令级并行。

### 4.3.2.3. -O2

相比较与 O1，O2 的优化更加全面，会进行内联，循环展开更加激进，优化执行次数更多，编译时间更长，性能有明显提升。O2 新增加的优化主要有下面几方面：更激进的循环优化，更激进的过程间优化，更多的内存访问优化，向量化，高级指令优化。

#### 4.3.2.3.1. 内存优化

O2 新增加的内存优化有 gvn , dse 。

- gvn (Global Value Numbering)

  通过值编号消除冗余内存读取，支持部分冗余消除（PRE），移动计算到支配节点。

- dse (Dead Store Elimination)

  消除无后续读取的存储操作。

#### 4.3.2.3.2. 循环优化

O2 新增加的循环优化有 mldst-motion 。

- mldst-motion

  跨迭代合并加载/存储指令。

- O2 的 loop-unroll 更加激进，循环展开次数更多。

#### 4.3.2.3.3. 过程间优化

O2 新增加的循环优化有过程间优化有 inline 。

- inline

  将在内联阈值内的函数内联。

#### 4.3.2.3.4. 向量化

O2 新增加的向量化优化有 slp-vectorizer 。

- slp-vectorizer (SLP Vectorization)

  将平行标量操作合并为向量指令，要求操作数内存连续且无对齐冲突。

#### 4.3.2.3.5. 高级指令优化

O2 新增加的高级指令优化有 jump-threading , tailcallelim，correlated-propagation。

- jump-threading

  合并条件相同的连续分支，消除内部条件判断。

- tailcallelim

  将尾递归调用转换为循环（消除栈帧增长）。

- correlated-propagation

  分析变量之间的关联关系，优化代码中的冗余操作和条件分支。

#### 4.3.2.4. -O3

O3 相比于 O2 新增加的优化并不多，但是相比于 O2 优化的设定更加激进，一些优化重复执行的次数也会更多。由于 O3 较为激进，编译时间会显著增加，可能会显著增加代码体积。

##### 4.3.2.4.1. 循环优化

- O3 的 loop-unroll 比 O2 更加激进，循环展开次数更多。

##### 4.3.2.4.2. 其他优化

O3 除了循环优化外新增加的优化有 aggressive-instcombine , callsite-splitting 。

- aggressive-instcombine

  相较于 instcombine，其匹配更复杂的指令模式进行简化。

- callsite-splitting

  根据调用上下文拆分函数，便于后续内联优化。

- O3 的内联阈值相比 O2 更大，会内联函数体更大的函数。

#### 4.3.2.5. -Os/-Oz

Os 和 Oz 是基于 O2 的优化等级。

Os 和 Oz 相比于 O2 只删除了 libcalls-shrinkwrap 和 pgo-memop-opt 这两个优化，具体的优化过程相同，但是优化的参数有调整。

和 O2 相反，Os 和 Oz 注重代码体积的优化，Os 会产生执行效率尚可但是代码体积比 O2 小的代码，Oz 将为极端，其代码体积比 Os 更小，但是执行效率会较低。

### 4.3.3. 链接脚本

#### 4.3.3.1. QXS320F280049RevA/QXS320F28377LRevA

用户可以通过编辑 ldscript/ldsciprt_Memory.ld 文件（如图 23 所示），调整分配给 DSP core0 和 core1 的 RAM 资源。当不使用 core1 时，允许将所有的 RAM 资源分配给 core0。

IRAM 的分配有以下约束（详见 ldsciprt_Memory.ld 文件注释）

1. IRAM 按照 64KB 颗粒度分配
2. DSP core1 的 IRAM 起始地址须固定在 IRAM 的起始地址（0x0020_0000）

```
MEMORY
{
  /*
   * Part 1: memory Layout
   */

  /* Data RAM */
  GSD0_RAM          : ORIGIN = 0x00000000, LENGTH = 0x00010000
  GSD1_RAM          : ORIGIN = 0x00010000, LENGTH = 0x00010000
  GSD2_RAM          : ORIGIN = 0x00020000, LENGTH = 0x00010000
  GSD3_RAM          : ORIGIN = 0x00030000, LENGTH = 0x00010000

  /* Instruction RAM */
  GSI0_RAM          : ORIGIN = 0x00200000, LENGTH = 0x00010000
  GSI1_RAM          : ORIGIN = 0x00210000, LENGTH = 0x00010000
  GSI2_RAM          : ORIGIN = 0x00220000, LENGTH = 0x00010000
  GSI3_RAM          : ORIGIN = 0x00230000, LENGTH = 0x00010000

  /* RAM for bootloader */
  BOOT_RAM          : ORIGIN = 0x00280000, LENGTH = 0x00002000

  /* Flash Memory: store instruction and data, and copy to RAM by bootloader */
  FLASH_DATA        : ORIGIN = 0x30000000, LENGTH = 0x00100000


  /*
   * Part 2:  application symbols and notes
   */

  /* 1. GSD_RAM is allocated by 64KB blocks (0x10000)
   * 2. Allocate all GSD_RAM to CORE0 is allowed, if not using CORE1
   * 3. Reserve last 4KB of GSD_RAM of each CORE for interrupt vector
   * 4. Reserve 4 bytes at address 0x0 to support comparison with NULL pointer
   */
  GSD_RAM_RESERVED    : ORIGIN = 0x00000000, LENGTH = 0x00000004
  GSD_RAM_CORE0_DATA : ORIGIN = 0x00000004, LENGTH = 0x0001FBFC
  GSD_RAM_CORE0_IVEC : ORIGIN = 0x0001FC00, LENGTH = 0x00000400
  GSD_RAM_CORE1_DATA : ORIGIN = 0x00020000, LENGTH = 0x0001FC00
  GSD_RAM_CORE1_IVEC : ORIGIN = 0x0003FC00, LENGTH = 0x00000400

  /* 1. CORE1 instruction space MUST start from GSI0_RAM (address 0x00200000)
   * 2. GSI_RAM is allocated by 64KB blocks (0x10000)
   * 3. Allocate all GSI_RAM to CORE0 is allowed, if not using CORE1
   */
  GSI_RAM_CORE0     : ORIGIN = 0x00220000, LENGTH = 0x00020000
  GSI_RAM_CORE1     : ORIGIN = 0x00200000, LENGTH = 0x00020000

  /* Fake memory regions for specifying stack size */
  STACK_CORE0       : ORIGIN = 0x00000000, LENGTH = 0x00000400
  STACK_CORE1       : ORIGIN = 0x00000000, LENGTH = 0x00000400
}
```

图 23：ldsciprt_Memory.ld 文件示例

#### 4.3.3.2. QXS320F280049RevB/QXS320F28377LRevB/QXS320F2800137 及其它型号

仍然通过编辑 ldscript/ldsciprt_Memory.ld 文件调整 SRAM 的分配，需要注意的分配约束如下。

1. 远端 RAM 只能分配给代码
2. 每个 DSP 内核至少分配一个近端 RAM BANK，用来存放中断向量表

对于双核 DSP 以及带 CLA 的 DSP，除了编辑 ldscript/ldsciprt_Memory.ld 文件外，还需要写配置寄存器告知 DSP 各 SRAM BANK 的归属。

写配置寄存器的位置有两处

1. bootloader 中的 Core0Boot_ConfigSRAM()函数

```
57 static void Core0Boot_ConfigSRAM()
58 {
59     DevCfgRegs.GSRAM_SEL.all  = 0x0000E000; // CPU1: GS00_RAM ~ GS12_RAM
60                                             // CPU2: GS13_RAM ~ GS15_RAM
61     DevCfgRegs.FARRAM_SEL.all = 0x000000FF; // CPU2: FAR0_RAM ~ FAR7_RAM
62 }
```

2. startup.s 中的调试 bootloader

```
1705 # config SRAM
1706 movigh gr2 0x100||
1707 movigl gr2 0x1000||
1708 movigh gr3 0x0||
1709 movigl gr3 0xe000||
1710 ||store32 gr3 gr2 0xc8
1711 movigh gr3 0x0||
1712 movigl gr3 0x00ff||
1713 ||store32 gr3 gr2 0xc9
```

#### 4.3.4. 编译及编译结果

Project Explorer 中的工程上右键单击，点击 Build Project 编译工程，如图 24 所示。

此时，除编译选中的工程外，还执行以下操作

1. 编译 bootloader（位于工程根目录下的 bootloader 目录；用户可以通过“导入工程”导入 bootloader 到 IDE）
2. 生成 Flash 烧写镜像文件（位于工程根目录下的 flash_image.hex 文件）

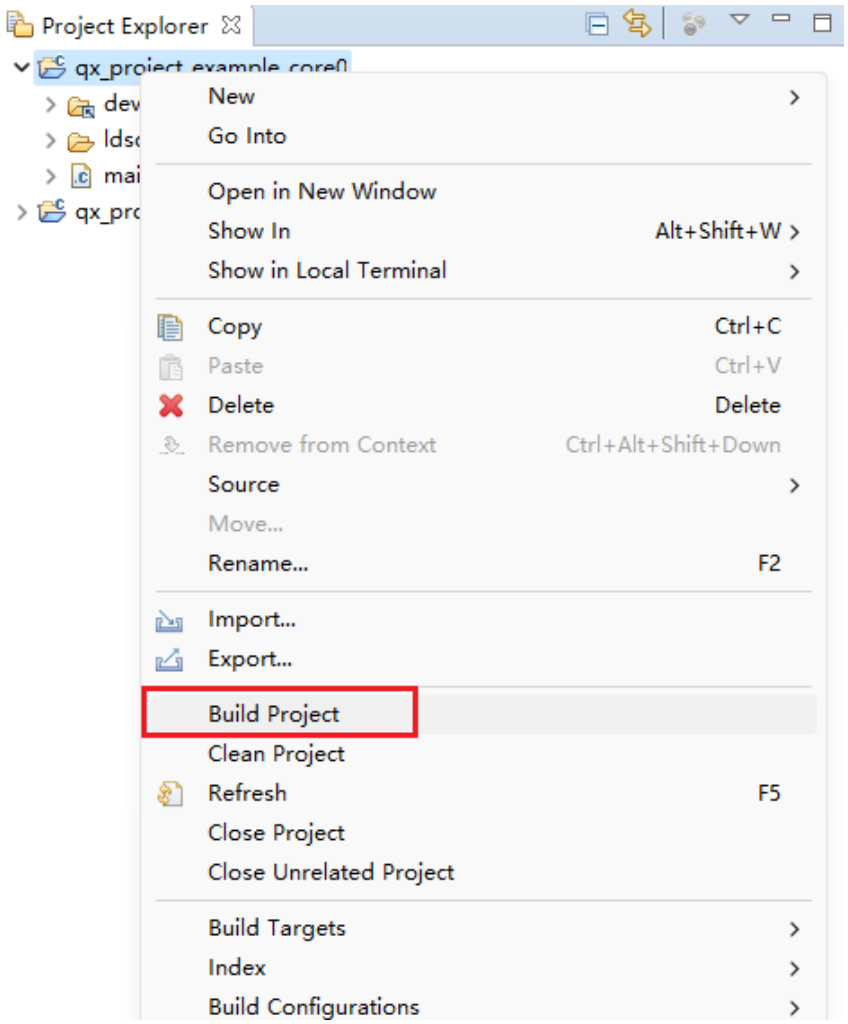



图 24：编译工程

编译成功后，编译结果位于所编译工程的 Release 目录，如图 25 所示。相关说明如表 4 所示。

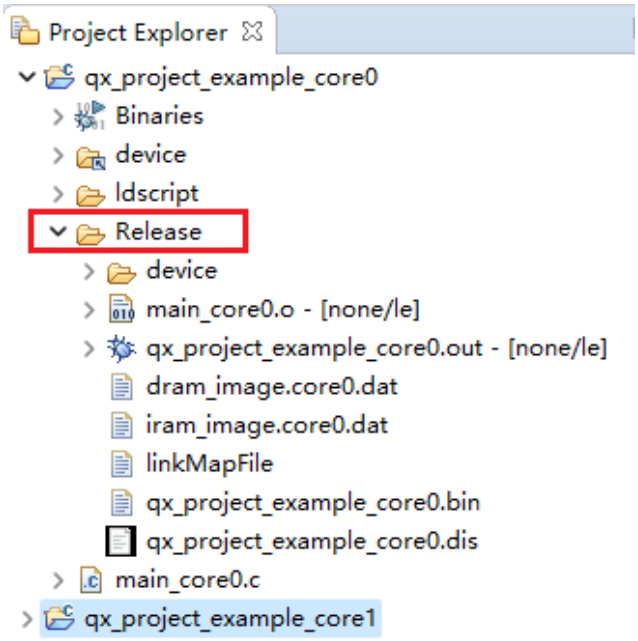



图 25：编译结果目录

表 4：编译结果说明

| 索引 | 目录/文件 | 说明 |
|---|---|---|
| 1 | Release/device | device/目录下源码的.o 文件 |
| 2 | Release/dram_image.core0.dat | DSP 应用程序的数据，位于 DRAM 地址空间 |
| 3 | Release/iram_image.core0.dat | DSP 应用程序的指令，位于 IRAM 地址空间 |
| 4 | Release/<project_name>_core0.map | DSP 应用程序链接映射文件 |
| 5 | Release/<project_name>_core0.bin | Bin 格式的 DSP 应用程序 |
| 6 | Release/<project_name>_core0.dis | DSP 应用程序反汇编文件 |

## 4.4. 调试

### 4.4.1. 双核启动机制

双核 DSP 的双核启动流程如下，

1. 启动 DSP core0
2. DSP core0 开启 core1 时钟以启动 DSP core1

### 4.4.2. 单核调试

右键单击需要调试的工程，对于双核 DSP 右键单击 core0 工程，如图 26 所示。

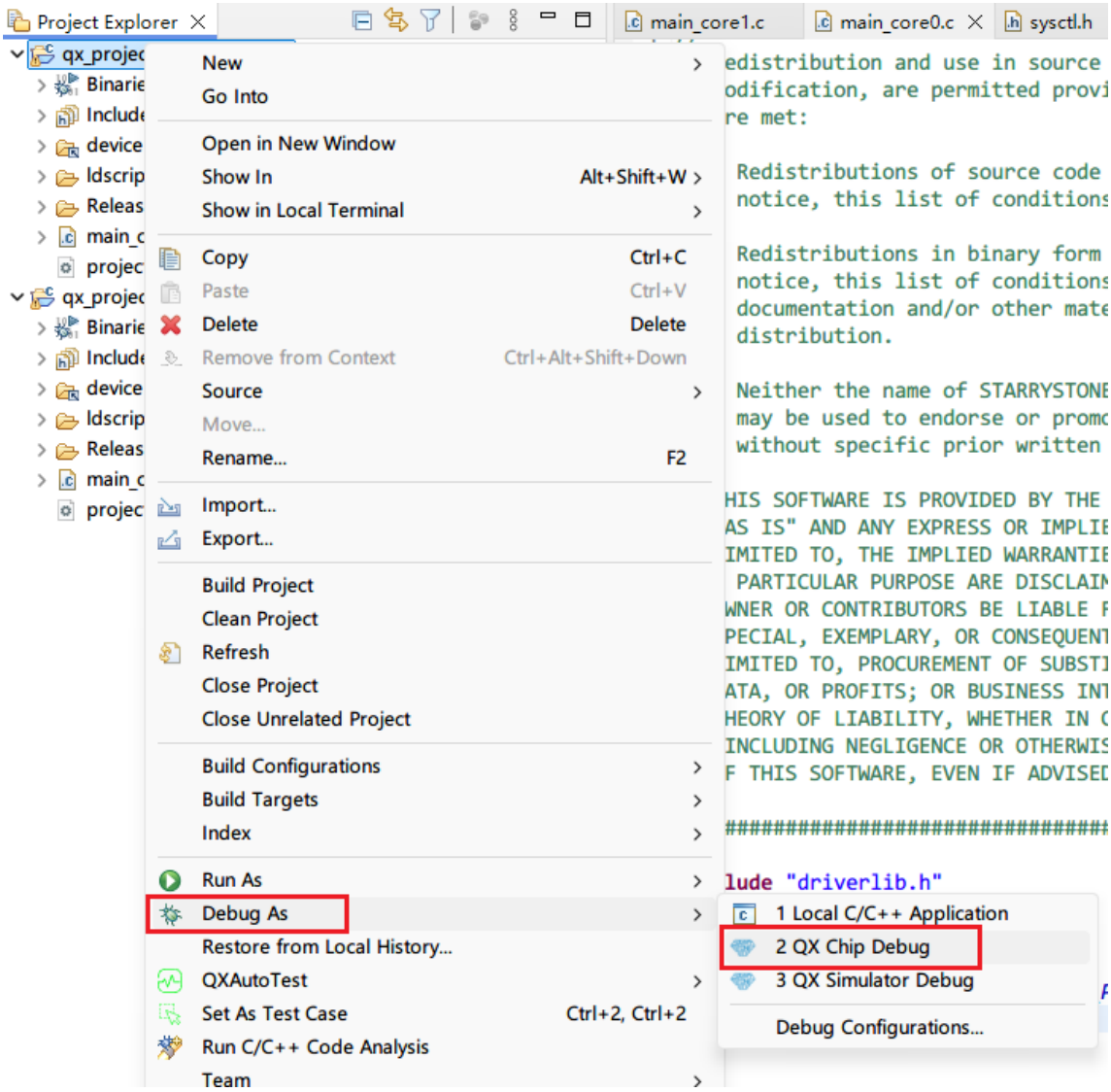



图 26：调试选择

支持的调试方式如下：

1. QX Chip Debug：使用物理芯片调试
2. QX Simulator Debug：使用指令集模拟器调试

   说明：该调试方式不支持外设，对外设地址空间的访问无效

选择期望的调试方式，进入调试界面开始调试，如图 27 所示。默认情况下，程序会停在main()函数的第一条语句。

图 27 中红色方框中为调试所需的基本功能按钮，从左到右依次为：继续执行程序、暂停程序、停止程序、step into、step over、step out。

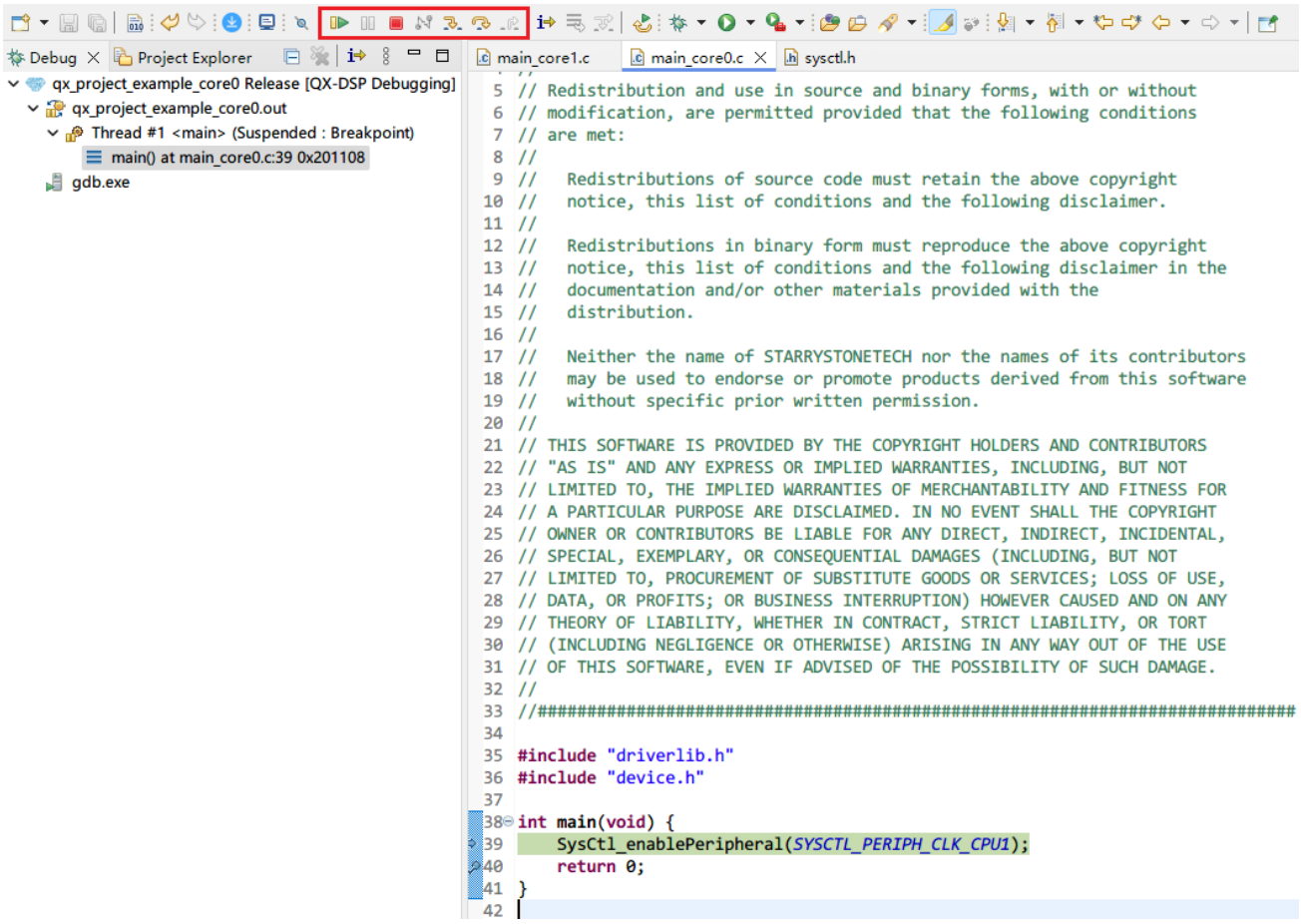



图 27：调试界面

调试过程中，还可以通过点击工具栏上的复位按钮（如图 28 所示），复位程序。

图 28：调试程序复位按钮

### 4.4.3. 双核调试

按照“4.4.2. 单核调试”启动 core0 之后，执行 core0 程序直到开启 core1 时钟，如图 29 所示。

```
35 #include "driverlib.h"
36 #include "device.h"
37
38 int main(void) {
39     SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_CPU1);
40     return 0;
41 }
42
```

图 29：开启 core1 时钟

确保 core0 暂停执行，然后按照“4.4.2. 单核调试”的步骤启动 core1 调试，如图 30 所示。

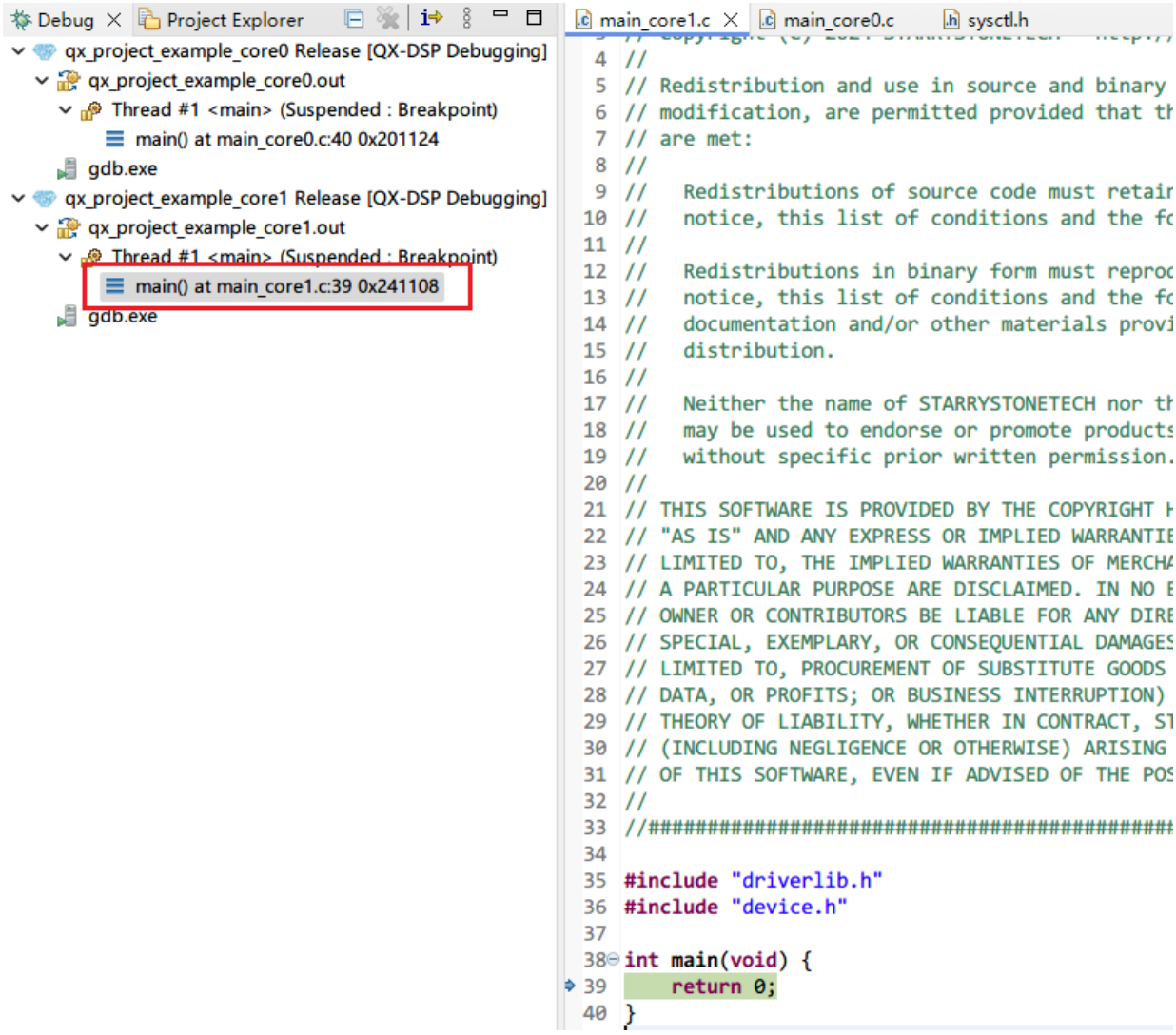



图 30：进入 core1 调试

#### 4.4.4. 调试端口配置

调试器 GDB 和每个 DSP core 之间的交互使用 TCP 端口，对于双核 DSP，使用的端口号连续。通常使用 QX-IDE 的默认端口（TCP Port 3333 和 3334）即可，如果默认端口被占用，可进入调试配置界面配置端口。

Project Explorer 中，在需要配置的 core0 工程上右键单击，选择 Debug As -> Debug Configurations...

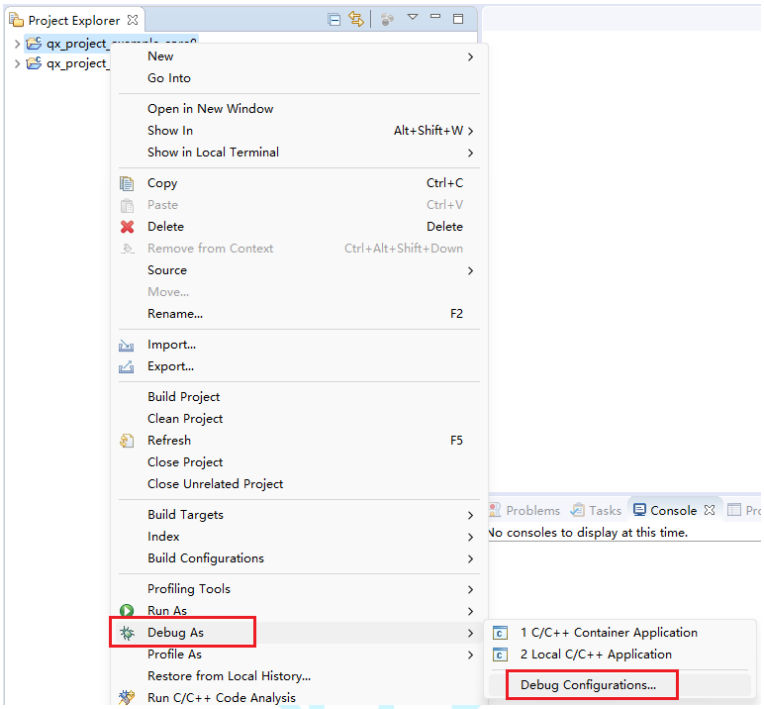
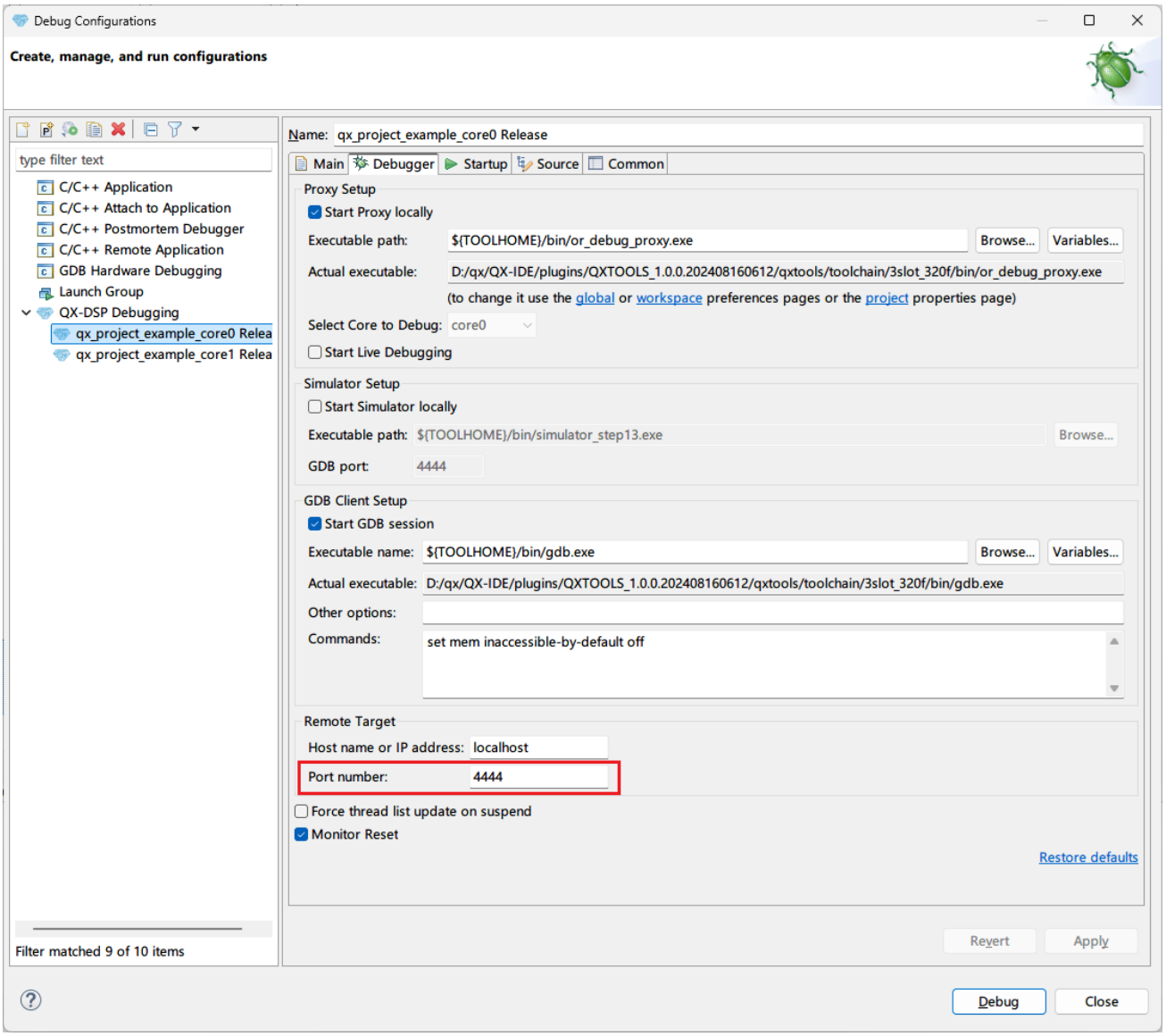



图 31：调试配置

选择 Debugger 页面，在所示红色方框位置配置调试 core0 所用端口，core1 端口自动为该值+1。



## 4.5. 实时刷新

### 4.5.1. 实时刷新配置

DSP 通过 JTAG 调试器连接宿主机调试时，支持全局变量和外设寄存器值的实时刷新。

右键单击要调试的工程，依次选择“Debug As”->“QX Chip Live Debug”进入实时调试，如图 32 所示。


图 32：进入芯片实时调试

或者在调试配置里，选择 Proxy Setup -> Start Proxy locally，选中 Start Live Debugging，启动实时调试，如图 33 所示。

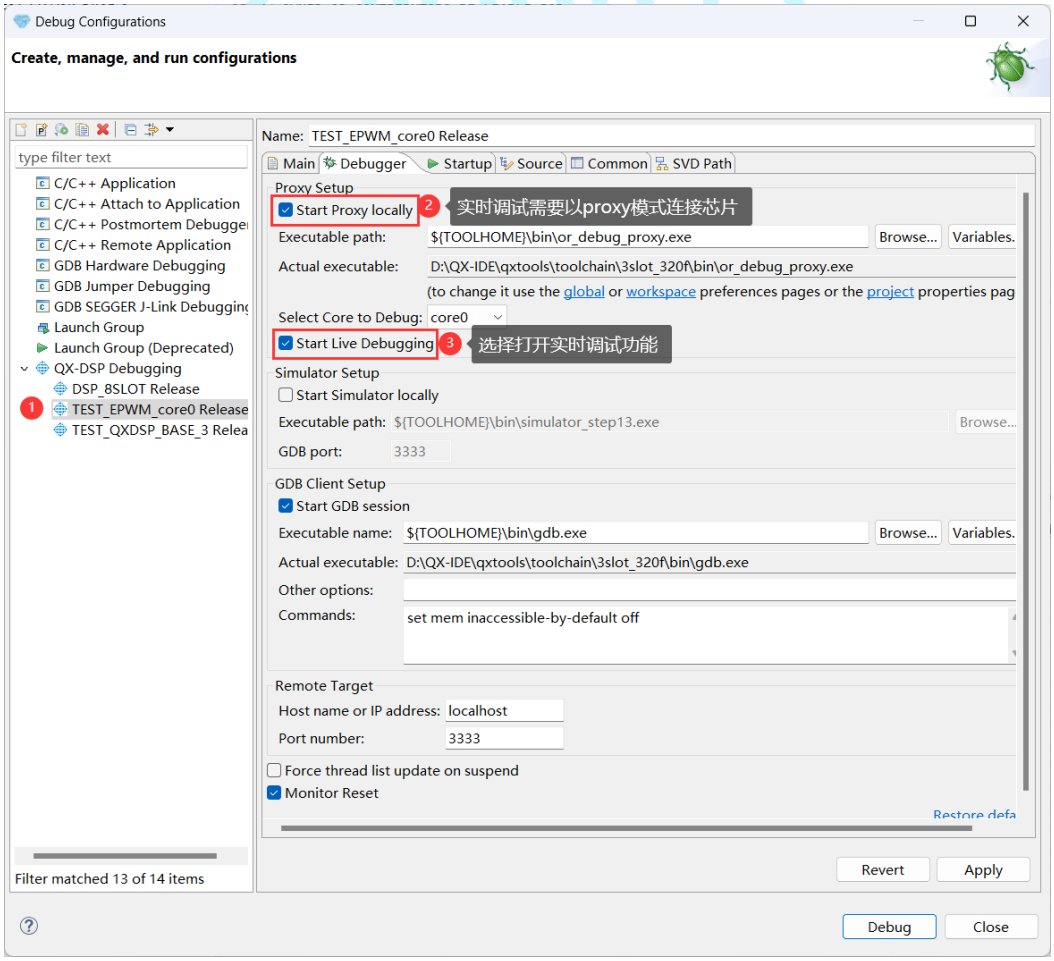



图 33：实时刷新配置

打开 Live Expression 视图（如果 Live Expression 视图未开启，可以按照以下步骤打开）

1、单击 IDE 右上角的按键切换至 Debug 视图

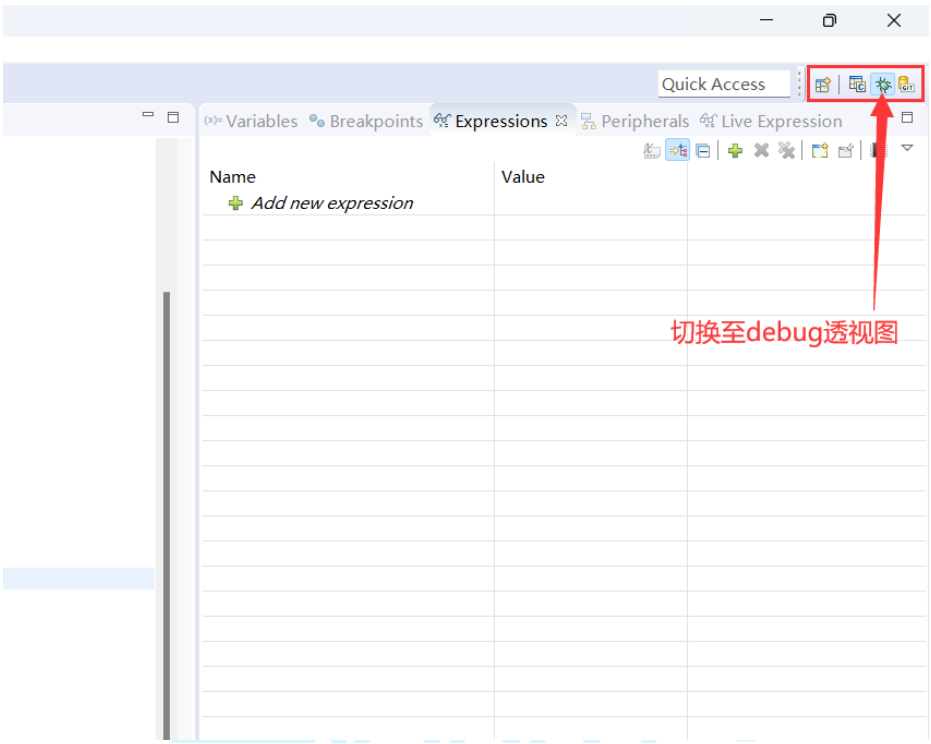



图 34：打开 Debug 透视图

2、在 IDE 上点击 Windows → Show View → Live Expression

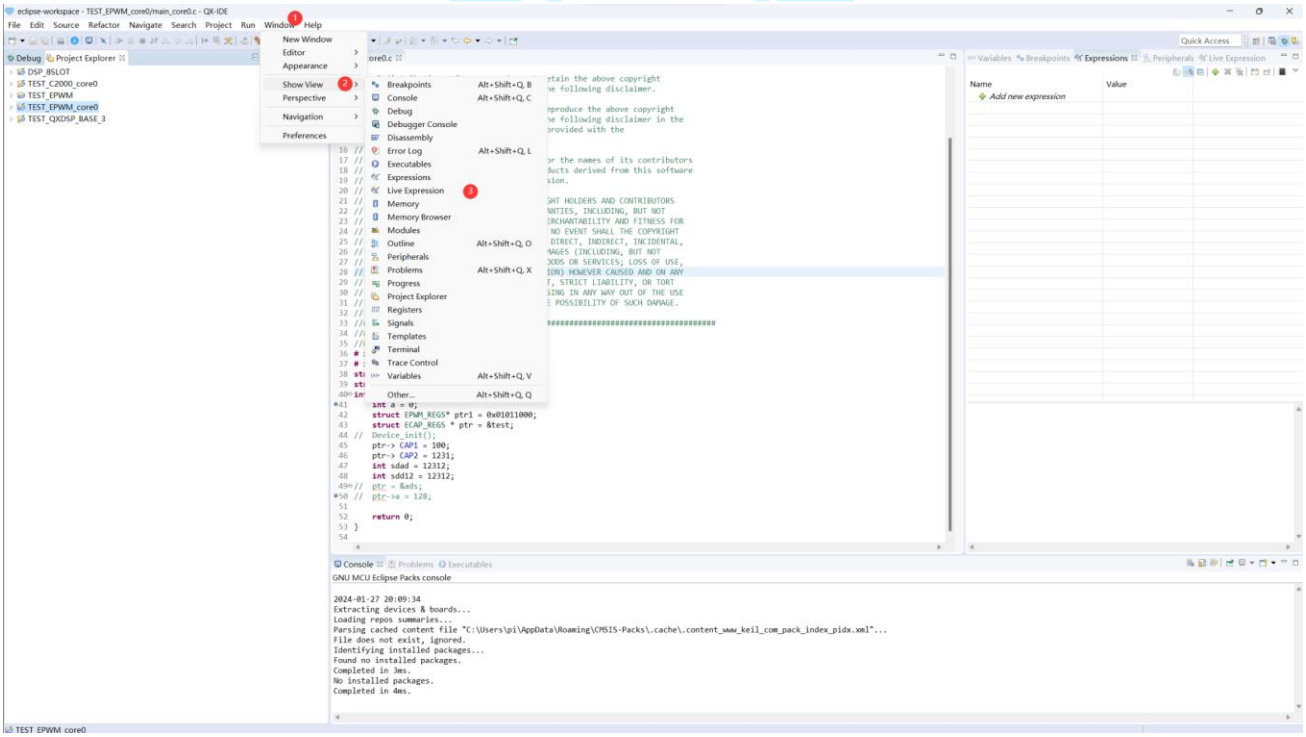
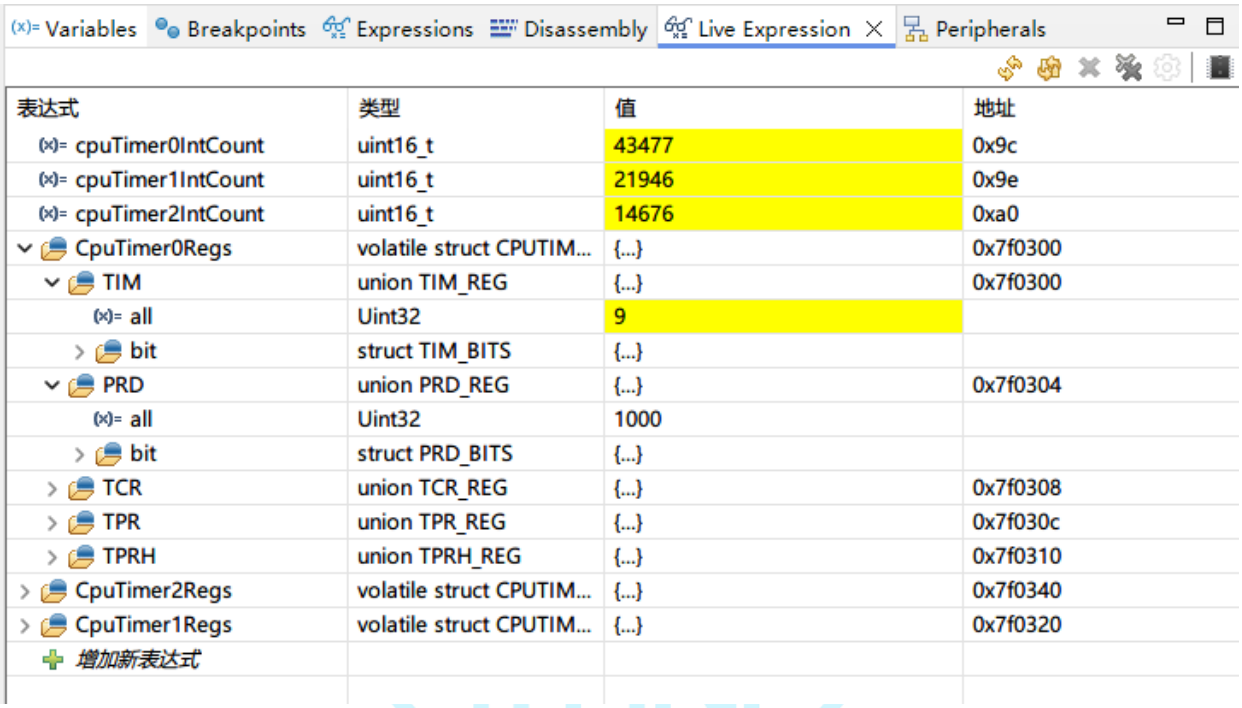


图 35：打开 Live Expression 视图

### 4.5.2. 使用 Live Expression

单击 或 添加需要实时刷新的表达式，当前支持全局变量 和 外设寄存器。

单击 开启实时刷新。未开启实时刷新时，单击 执行单次刷新。




| 表达式 | 类型 | 值 | 地址 |
|---|---|---|---|
| cpuTimer0IntCount | uint16_t | 43477 | 0x9c |
| cpuTimer1IntCount | uint16_t | 21946 | 0x9e |
| cpuTimer2IntCount | uint16_t | 14676 | 0xa0 |
| CpuTimer0Regs | volatile struct CPUTIM... | {...} | 0x7f0300 |
| TIM | union TIM_REG | {...} | 0x7f0300 |
| all | Uint32 | 9 | |
| bit | struct TIM_BITS | {...} | |
| PRD | union PRD_REG | {...} | 0x7f0304 |
| all | Uint32 | 1000 | |
| bit | struct PRD_BITS | {...} | |
| TCR | union TCR_REG | {...} | 0x7f0308 |
| TPR | union TPR_REG | {...} | 0x7f030c |
| TPRH | union TPRH_REG | {...} | 0x7f0310 |
| CpuTimer2Regs | volatile struct CPUTIM... | {...} | 0x7f0340 |
| CpuTimer1Regs | volatile struct CPUTIM... | {...} | 0x7f0320 |
| 增加新表达式 | | | |

图 36：Live Expression View

单击 配置实时刷新间隔，如图 37 所示。

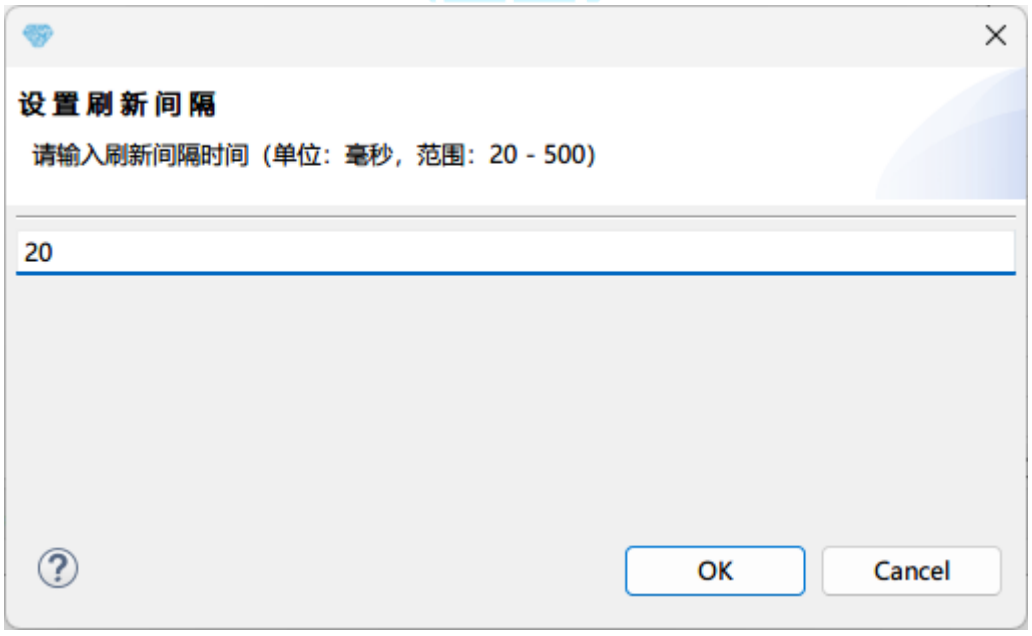



图 37：设置实时刷新间隔

可以在表达式上使用 右键表达式 → 数字格式 → 为每个变量定义数字格式

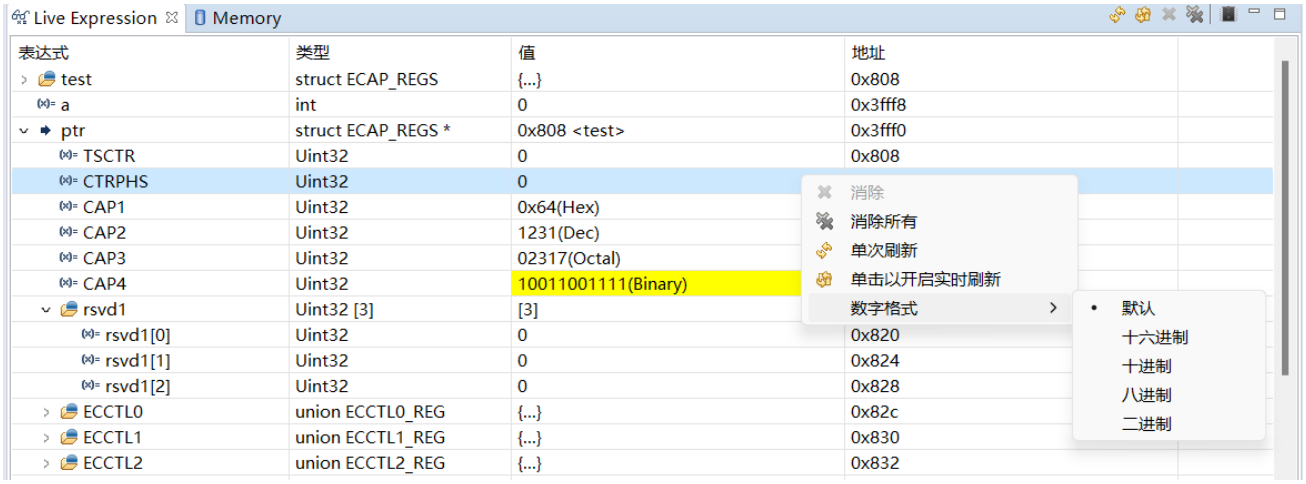



图 38：Live Expression View 设置变量显示格式

如果没有在实时刷新配置中打开实时刷新选项，Live Expression 视图功能和 Expression 视图类似，但无法使用实时刷新功能。

**注意：**实时调试时，点击 Resume 后不能再停下或命中后续断点。

### 4.5.3. 实时刷新绘图

按以下步骤使用该功能。

#### 4.5.3.1. 打开 Live Expression View

参考 4.5.1. 实时刷新配置。

#### 4.5.3.2. 打开 Graph View

Debug 视图下：点击菜单栏 Windows -> Show View -> Graph View

其它视图下：点击菜单栏 Windows -> Show View -> Other...，输入 Graph View 搜索并打开

**注意：**

- 启动 Graph View 之前须先打开 Live Expression View。如果在打开 Live Expression View 之前启动了 Graph View，则须关闭 Graph View，重启 IDE，然后依次打开 Live Expression View 和 Graph View

#### 4.5.3.3. 启动实时调试

参考 4.5.1. 实时刷新配置。

#### 4.5.3.4. 添加变量到 Graph View

Live Expression View 里，在需要绘制波形的变量上：单机鼠标右键->添加到绘图窗口，如图 39 所示。

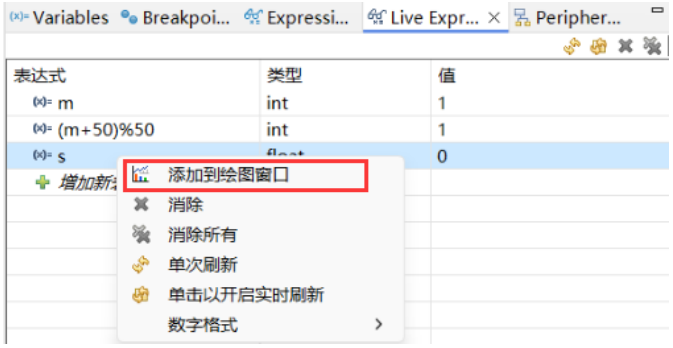



图 39：将变量添加到 Graph View

**说明：**

- 支持绘制外设寄存器。先将外设寄存器添加到 Live Expression View，然后添加到 Graph View
- 支持变量表达式绘制。例如绘制“(m+50)%50”，在 Live Expression 中输入该表达式，然后添加到 Graph View 即可
- 支持绘制多个变量/表达式，它们将会被绘制在同一张图上

**注意：**

- 不支持绘制复合变量（struct、union、数组）和指针

#### 4.5.3.5. 启动绘图

Graph View 按钮及含义如图 40 所示。

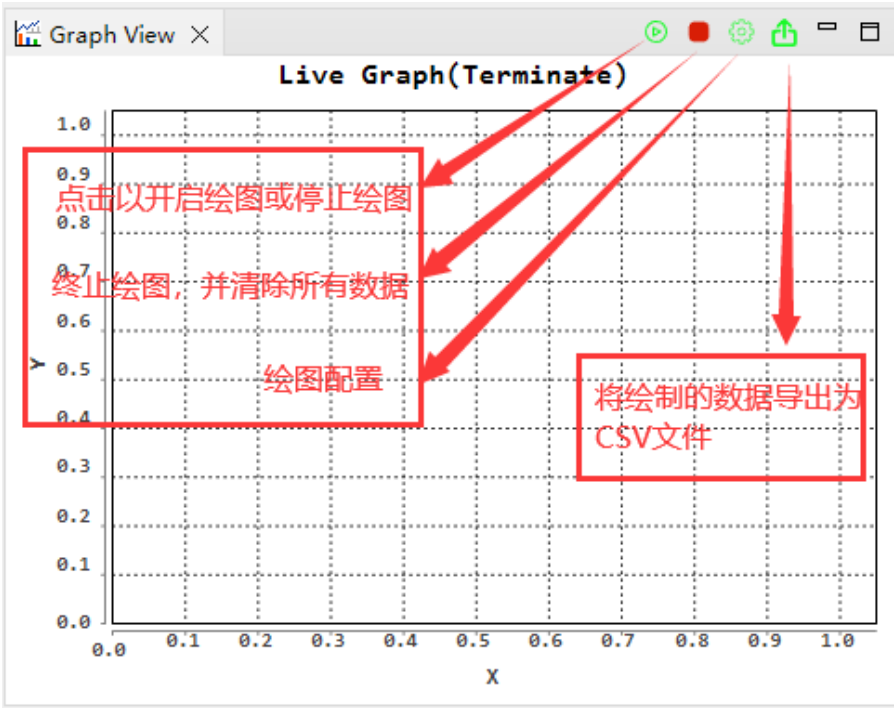



图 40：Graph View 按钮及含义

点击 按钮启动绘图；绘图启动后，点击 按钮暂停绘图。

图 40 标题栏的 Terminate 表示 Graph View 的绘图状态，包含以下几种

- Terminate：不存在需要绘制的变量，也是初始状态；点击 将进入此状态
- Running：正在绘制图形，非 Running 时点击 进入此状态
- Pause：暂停绘制图形，Running 时点击 进入此状态

#### 4.5.3.6. 绘图配置及其它特性

1. 点击  打开绘图配置界面，如图 41 所示。

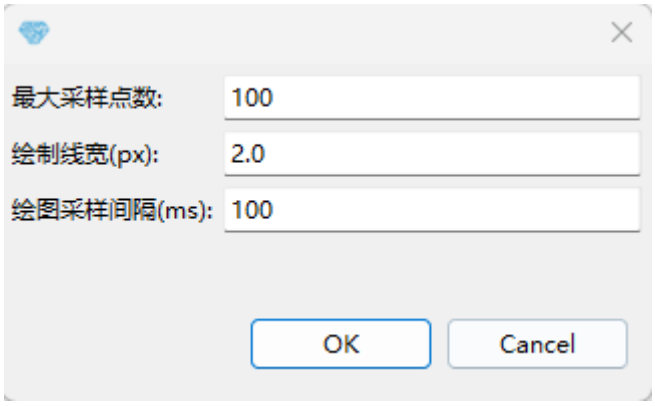



图 41：绘图配置界面

2. 保存绘图为图像

   在图形上右键，在弹出的菜单点击【另存为】或者【打印】，将的图形导出。如图 42 所示。

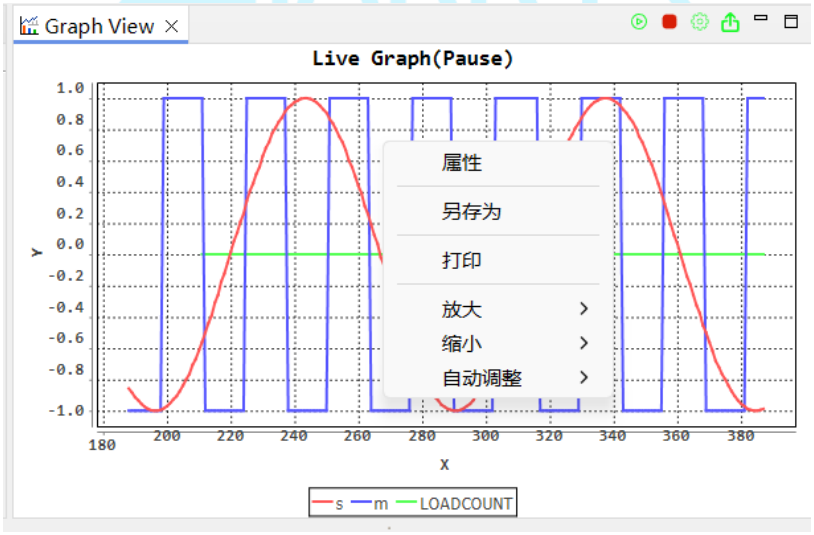



图 42：绘图导出

3. 图形缩放

   在图形上右键，在弹出的菜单选择【放大】、【缩小】、【自动调整】，如图 42 所示。

## 4.6. 外设寄存器查看

QX-IDE 内置了对于外设寄存器的查看功能，您可以在 Expression View、Live Expression View 两个视图中添加外设寄存器监视。点击 按键可以为视图添加外设寄存器。

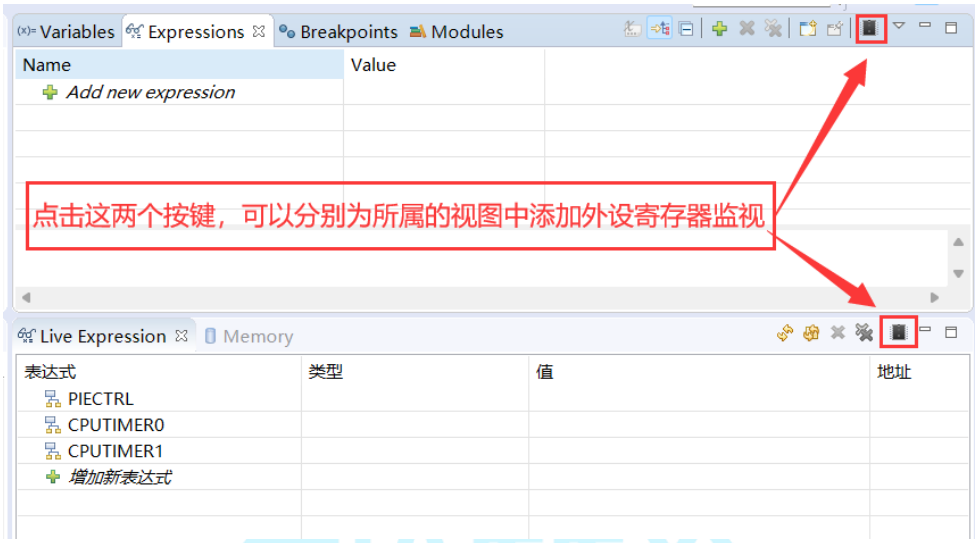



图 43：添加外设寄存器监视

点击 ，您可以选择需要查看的表达式，如下图所示：

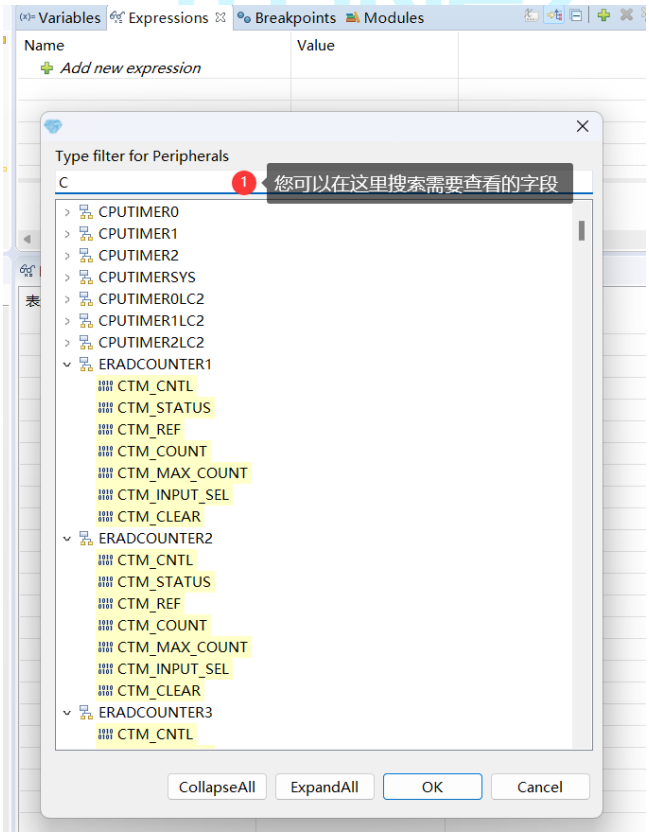



图 44：添加外设寄存器监视

您可以一次性添加多个外设寄存器到视图上，按住键盘上的 CTRL 键，同时点击选中多个寄存器字段，然后点击 OK，如下图所示：

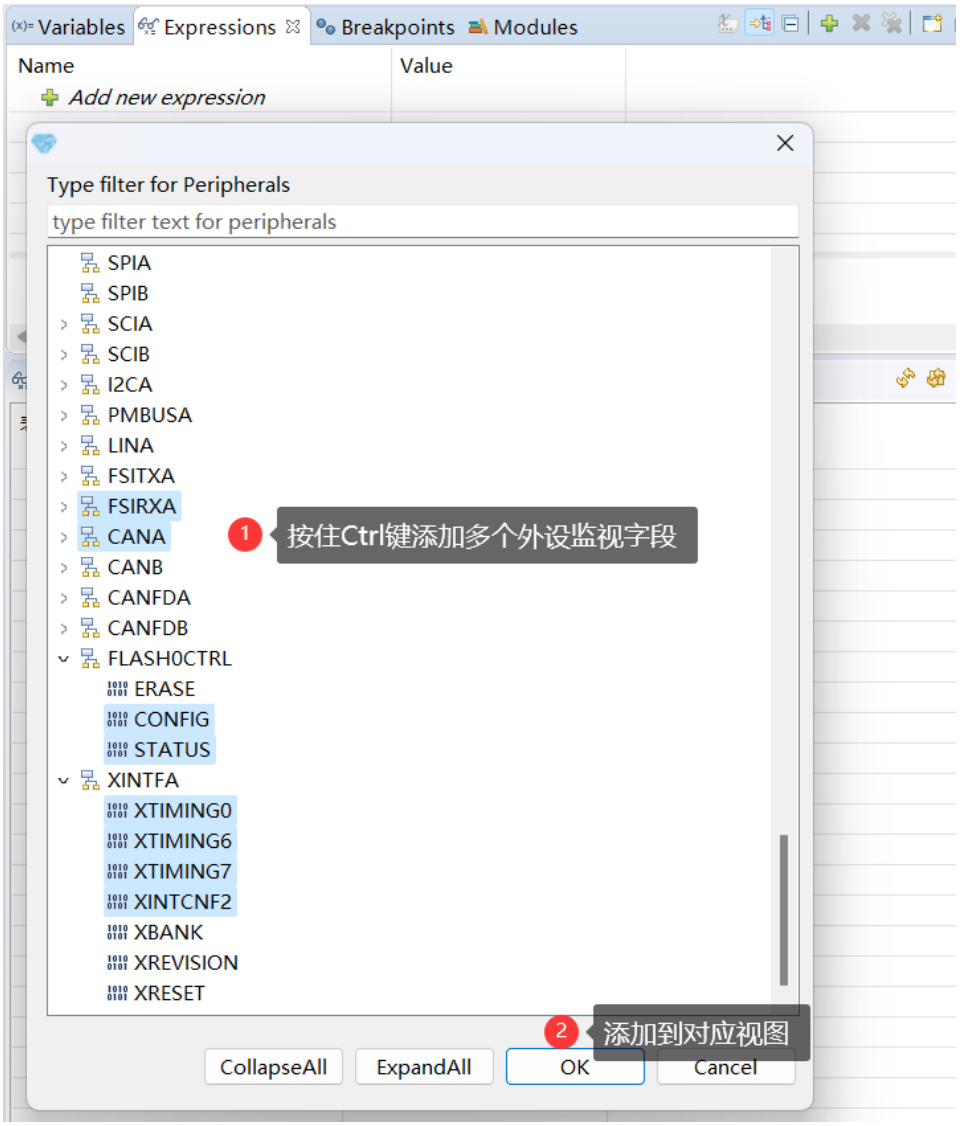



图 45：添加多个外设寄存器监视

您可以在调试中查看外设寄存器的值，如下图所示

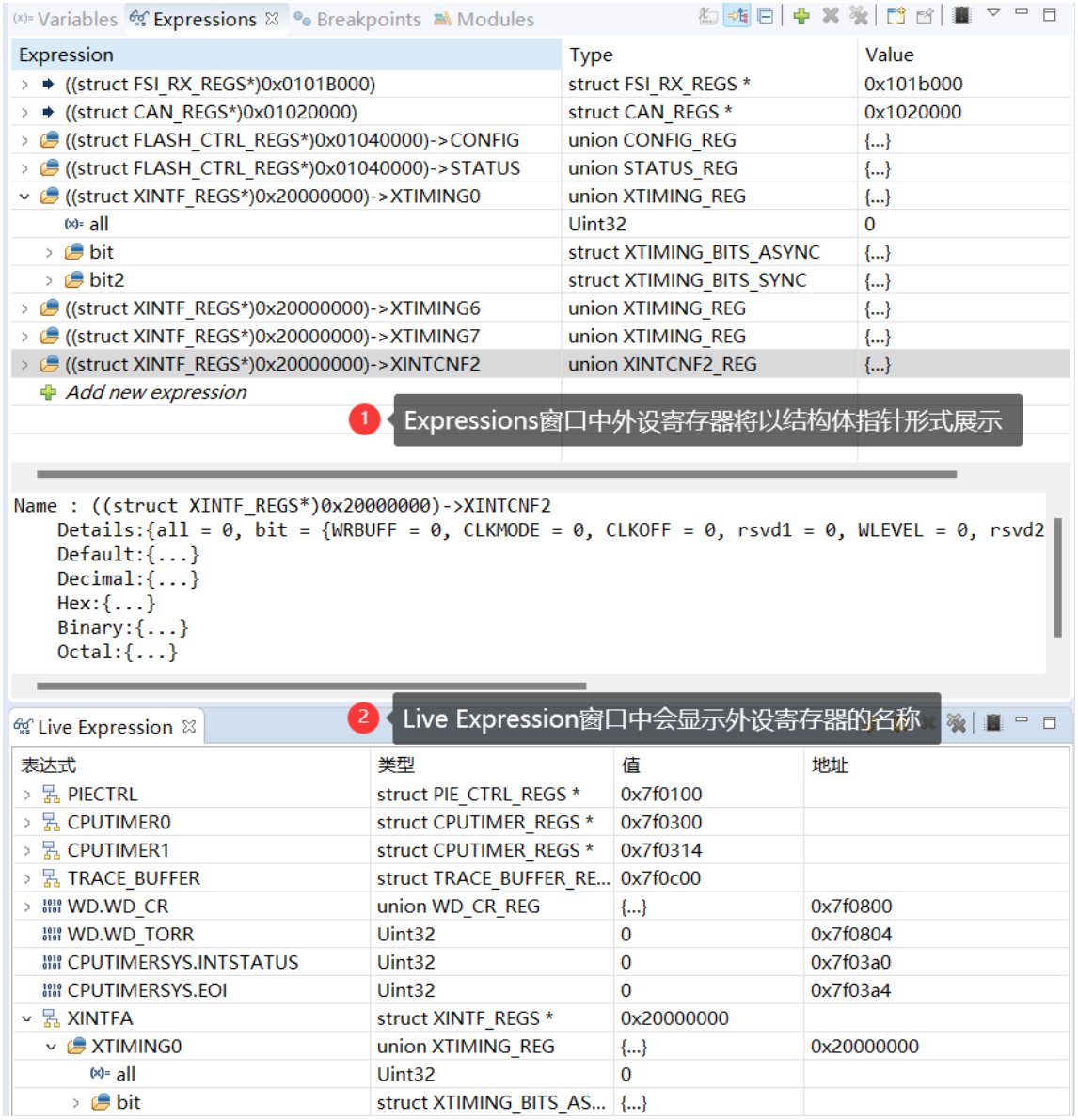



图 46：外设寄存器显示

注意：以上两个视图都可以在调试过程中查看外设寄存器的值

当您在 0 打开了实时调试配置的情况下，您可以在 Live Expression 窗口中通过实时刷新的方式查看外设寄存器的值。

在不启动实时调试的情况下在 Live Expression 窗口也可以查看外设寄存器的值。

请尽量避免在开启实时刷新时向 Live Expression 中添加过大的寄存器字段，尽量添加第二层的字段，以免影响实时刷新的速率。

## 4.7. Flash 烧写

### 4.7.1. Flash 镜像

编译成功后，QX-IDE 的后处理步骤将调用 postbuild.cmd/postbuild.sh 生成 Flash 镜像文件。Flash 镜像文件描述应用程序在 Flash 存储中的内容，并包含以下部分

1. bootloader 指令镜像：iram_image.bootloader.dat。（当前不支持 bootloader 数据）
2. DSP core0 数据镜像：dram_image.core0.dat
3. DSP core0 指令镜像：iram_image.core0.dat
4. （可选）DSP core1 数据镜像：dram_image.core1.dat
5. （可选）DSP core1 指令镜像：iram_image.core1.dat

生成的 Flash 镜像文件位于工程根目录，并有 2 种形式。

- .hex 文本格式：烧写工具目前使用的格式
- .bin 二进制格式：Flash 镜像内容按 32-bit 小端排列

图 47 展示相同内容的 Flash 镜像文件，在两种格式下的保存形式。

```
:format=compact
@0x30000000
94300207 a8308342 94230005 94201001
94300001 94240005 94201001 ec308342
84410304 8441030b a8408082 94300015
80000000 80000000 83ffff65 94010002
94300203 a8308342 94230005 94201001
800000a1 95efc005 95e00011 ec308342
80000000 80010003 80000000 80000000
8000228d edf781e2 99e7be00 80000000
94210005 94201001 80000000 80000000
8430c404 94400f85 a8308700 14400001
946000f5 94600001 94500005 94500221
84514704 94700075 94700001 8451460b
8430c404 a8308780 ec308700 8430c505
84410704 8441060b 94400005 94400201
94312005 94301001 ec308780 8430c405
ec40c002 94400003 a840c002 80000000
```

```
                Offset Bytes
                       00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
                       ------ -----------------------------------------------
0000000000000000 07 02 30 94 42 83 30 A8 05 00 23 94 01 10 20 94
0000000000000010 01 00 30 94 05 00 24 94 01 10 20 94 42 83 30 EC
0000000000000020 04 03 41 84 0B 03 41 84 82 80 40 A8 15 00 30 94
0000000000000030 00 00 00 80 00 00 00 80 65 FF FF 83 02 00 01 94
0000000000000040 03 02 30 94 42 83 30 A8 05 00 23 94 01 10 20 94
0000000000000050 A1 00 00 80 05 C0 EF 95 11 00 E0 95 42 83 30 EC
0000000000000060 00 00 00 80 03 00 01 80 00 00 00 80 00 00 00 80
0000000000000070 8D 22 00 80 E2 81 F7 ED 00 BE E7 99 00 00 00 80
0000000000000080 05 00 21 94 01 10 20 94 00 00 00 80 00 00 00 80
0000000000000090 04 C4 30 84 85 0F 40 94 00 87 30 A8 01 00 40 14
00000000000000A0 F5 00 60 94 01 00 60 94 05 00 50 94 21 02 50 94
00000000000000B0 04 47 51 84 75 00 70 94 01 00 70 94 0B 46 51 84
00000000000000C0 04 C4 30 84 80 87 30 A8 00 87 30 EC 05 C5 30 84
00000000000000D0 04 07 41 84 0B 06 41 84 05 00 40 94 01 02 40 94
00000000000000E0 05 20 31 94 01 10 30 94 80 87 30 EC 05 C4 30 84
00000000000000F0 02 C0 40 EC 03 00 40 94 02 C0 40 A8 00 00 00 80
0000000000000100 01 10 20 94 02 80 30 EC 03 04 30 94 02 80 30 A8
0000000000000110 82 80 30 EC 35 00 30 94 01 00 30 94 05 00 29 94
```

图 47：Flash 镜像示例

左：.hex 文本格式，右：.bin 二进制格式

### 4.7.2. 执行烧写

确保 DSP 和宿主机通过调试器连接并开机，点击 IDE 工具栏中的 Download 按钮启动 Flash 下载，如图 48 所示。

图 48：Flash 烧写按钮

Flash 下载窗口如图 49 所示，说明如下。

1. Select a project：选择要烧写的工程；对于双核 DSP 选择 core0 或 core1 工程效果一样
2. Flash image：显示 flash image 的完整路径；如果 flash image 不存在则内容为空

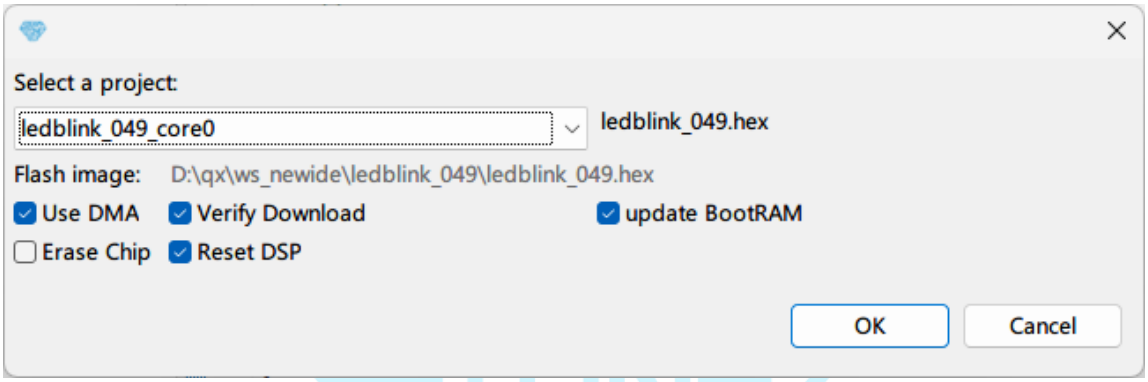



图 49：Flash 下载窗口

确认 flash image 存在并且正确，点击 OK 开始 Flash 烧写。烧写过程中，可以在 IDE 右下角查看下载进度，如图 50 所示。烧写完成后，可以在 Console View 查看烧写日志。

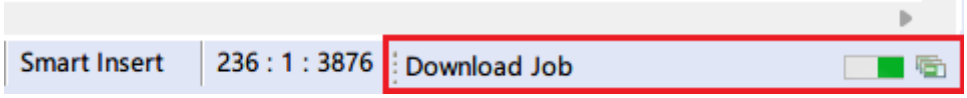



图 50：Flash 烧写进度

#### 4.7.2.1. 找不到 Flash 镜像文件的解决方法

当.hex 文件已经生成，但是 Flash 下载窗口检测不到.hex 文件，多半是由于 Project Explorer 工程层级过深，.hex 文件没有被 IDE 识别，如图 51 所示。

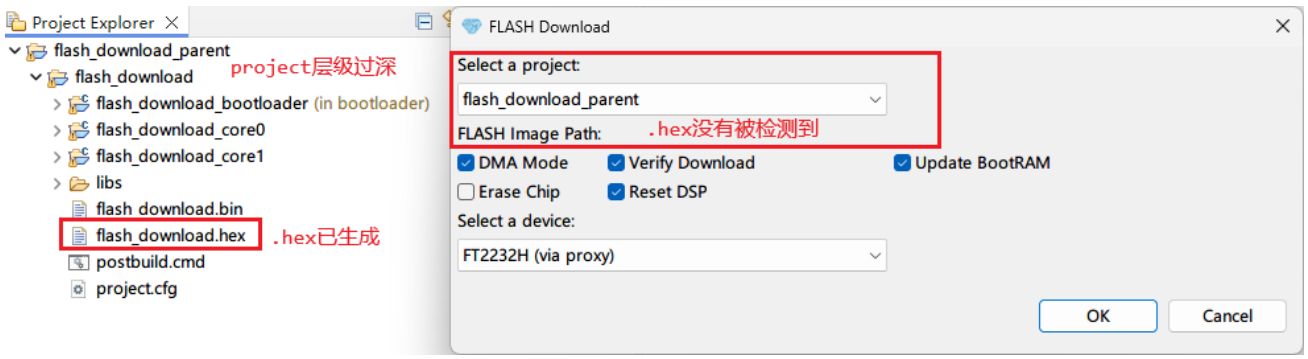



图 51：.hex 文件未被识别

此时，删除所有顶级工程即可，如图 52 所示。右键单击顶层工程，选择“Delete”，在弹出的窗口直接点击“OK”。当 Project Explorer 显示如图 53 所示，即可识别到.hex 文件。

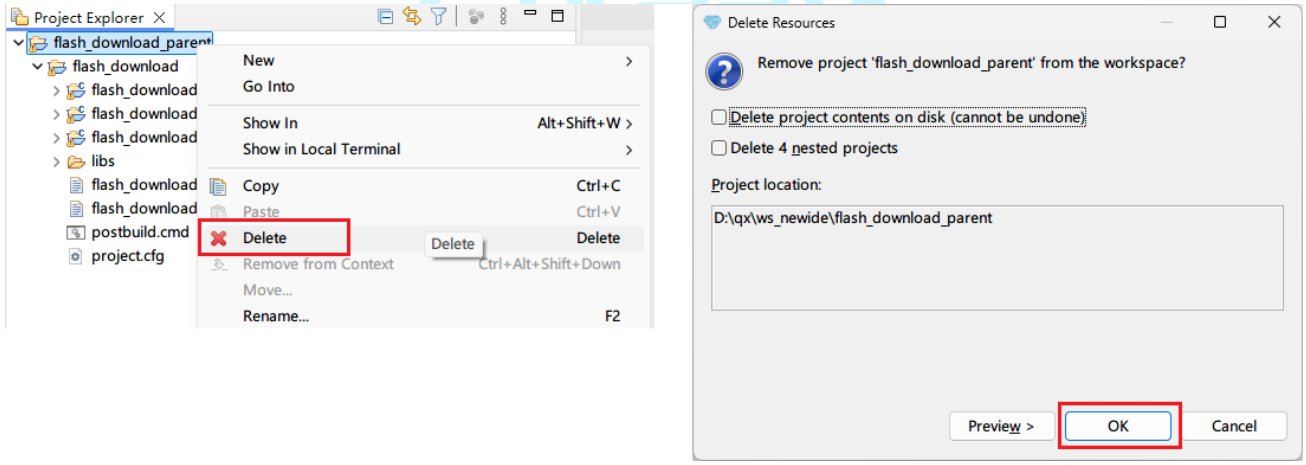



图 52：删除顶层工程

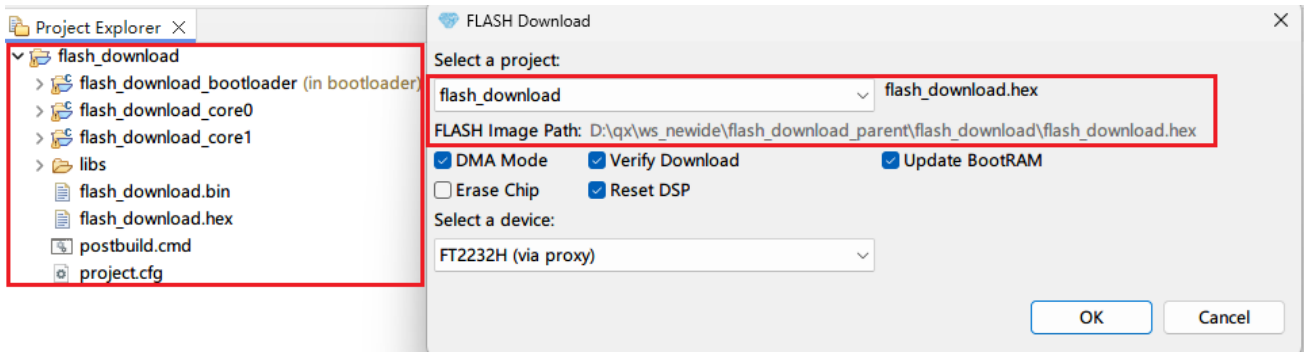



图 53：检测到.hex 文件

#### 4.7.3. 烧写工具命令行接口

QX-IDE 通过调用 Flash 烧写工具提供的命令行接口完成 Flash 烧写。烧写工具位于 QX-IDE 安装目录下的以下位置：

```
plugins\QXTOOLS_*\qxtools\toolchain\3slot_320f\bin\program_flash.exe
```

可通过执行`.\program_flash.exe --help`查看命令行接口，如图 54 所示。

例如，向 QXS320F280049RevB 的 Flash 烧写一般执行如下命令。

```
program_flash.exe download F280049RevB_EVB_examples.hex 0x30000000 -m dma -v -u --chip-type qxs320f28004xrevb --chiperase --reset
```

该命令执行以下动作，

1. 擦除整片 FLASH 数据区
2. 用 DMA 方式将.hex 文件内容写入 FLASH 数据区，从地址`0x30000000`开始写入
3. 写入完成校验数据、更新`BootRAM`、软复位芯片

各参数含义解释如下

```
download                        \\ 执行 Flash 下载
F280049RevB_EVB_examples.hex    \\ 要烧写到 FLASH 数据区的内容，.hex 格式
0x30000000                      \\ 烧写 FLASH 数据区的起始地址
-m dma                          \\ DMA 方式写入 FLASH，写入速度更快
-v                              \\ 写入完成校验
-u                              \\ 校验完成更新 BootRAM
--chip-type qxs320f28004xrevb   \\ 指定芯片型号
--chiperase                     \\ 烧写前 FLASH 数据区全部擦除
--reset                         \\ 烧写、校验、更新 BootRAM 后软复位芯片
```

```
PS D:\qx\QX-IDE\plugins\QXTOOLS_1.0.0.202509050818\qxtools\toolchain\3slot_320f\bin> .\program_flash.exe --help

program_flash: version 1.8.0.20250901

Usage: program_flash <command> <ArgumentsForCommand>

A tool originally designed for programming flash memory in C2000 system. It also does
read/write testing for onboard DRAM, IRAM and flash memory.

<command>        action to take

Supported commands:
  download      : download program file to flash memory
  dcsm          : write DCSM
  flash2jtag    : read data from flash memory and store in file
  reset         : reset all DSP cores
  chiperase     : erase all flash memory
  bootram       : update bootram from boot code in flash
  testiram      : full IRAM  read/write test
  testdram      : full DRAM  read/write test
  testflash     : full Flash read/write test
  testperipheral: peripheral base register read test
  writechipsn   : write chip SN to last 64bit of flash OTP 1 (the 1st OTP bank)
  eraseflashotp : erase FLASH OTP banks

Common optional arguments:
  -w, --write-logfile                           write log file
  --chip-type <chipType>                        specify chip type
                                                  qxs320f28004x (default)
                                                  qxs320f28377L
                                                  qxs320f2833x
                                                  qxs320f2803x
                                                  qxs320f280013x
                                                  qxs320f28004xrevb
                                                  qxs320f28377Lrevb
                                                  qxs320f28p65x
                                                  qxs320f2837xd
  --jtagchannel                                 specify JTAG channel of debugger, A | B (default)

Arguments for command: download

  Positional arguments:
    <ProgramFile>                               file to download to flash, typical filename is `flash_image.hex`
    <FlashStartAddress>                         flash start address to write program, hex 0x----
                                                currently, should be fixed to flash base address 0x3000_0000

  Optional arguments:
    -m <ProgramMode>, --mode <ProgramMode>      direct | dma, default: dma
    -v, --verify                                verify flash content after write
    -u, --update-bootram                        update boot ram after write and verify
    --dcsm                                      underlying chip has DCSM module
    --chiperase                                 erase the whole chip before download
    --reset                                     reset DSP after download
    -b <nBytes>, --ram-access-block-size <nBytes>   number of RAM access bytes for one time, default: 2048
    --dma-ram-buffer-size <bufferSize>          RAM buffer size for DMA download, 128KB | 512KB, default: 512KB

Arguments for command: writechipsn

  Positional arguments:
    <ChipUID>                                   Chip unique ID, 24-bit, hex 0x----

Arguments for command: eraseflashotp
    <OTPBank>                                   FLASH OTP bank to erase, 1 | 2

Arguments for command: dcsm

  Positional arguments:
    <address>                                   address of chip, hex 0x----
    <data>                                      32-bit data to write to <address>, hex 0x----

Arguments for command: flash2jtag
  -s <MemoryStartAddress>                       flash memory start address
  -l <DataSizeInBytes>                          number of bytes to read from flash

Arguments for command: reset
  none

Arguments for command: chiperase
  none

Arguments for command: bootram
  none

Arguments for command: testiram
  -b <nBytes>, --ram-access-block-size <nBytes>     number of RAM access bytes for one time, default: 2048

Arguments for command: testdram
  -b <nBytes>, --ram-access-block-size <nBytes>     number of RAM access bytes for one time, default: 2048

Arguments for command: testflash
  -b <nBytes>, --ram-access-block-size <nBytes>     number of RAM access bytes for one time, default: 2048

Arguments for command: testperipheral
  none
```

图 54：Flash 烧写工具 help 信息

## 4.8. 串口工具

QX-IDE 集成串口工具以便观察 DSP 的串口输出，使用方式如下。

1. 打开 Terminal View

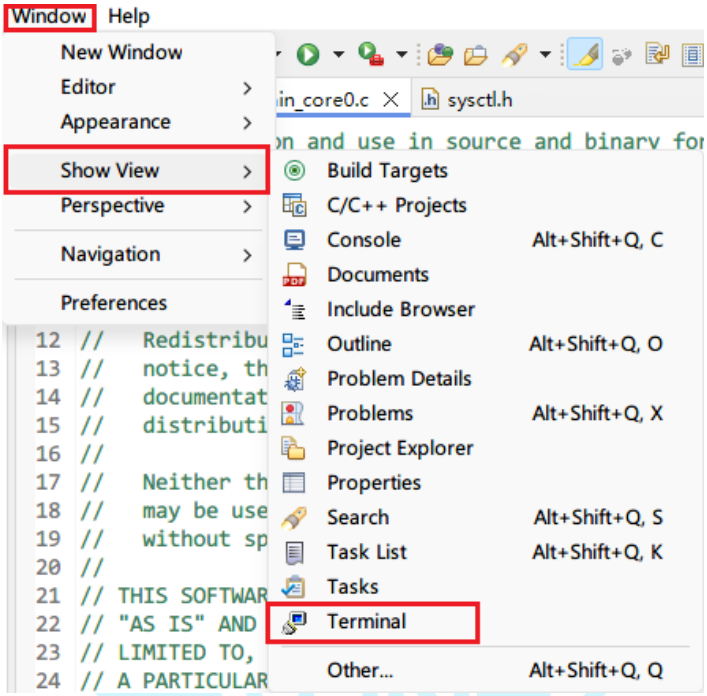



图 55：打开 Terminal View

2. 打开 Terminal，单击图 56 红框图标

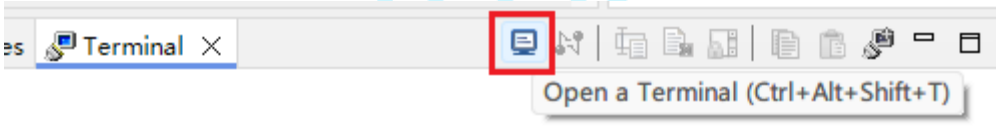



图 56：打开 Terminal

3. 配置 Serial Terminal

Choose terminal 选择 Serial Terminal，Settings 按实际情况配置，点击 OK 即可打开 QX-IDE 中集成的串口工具。

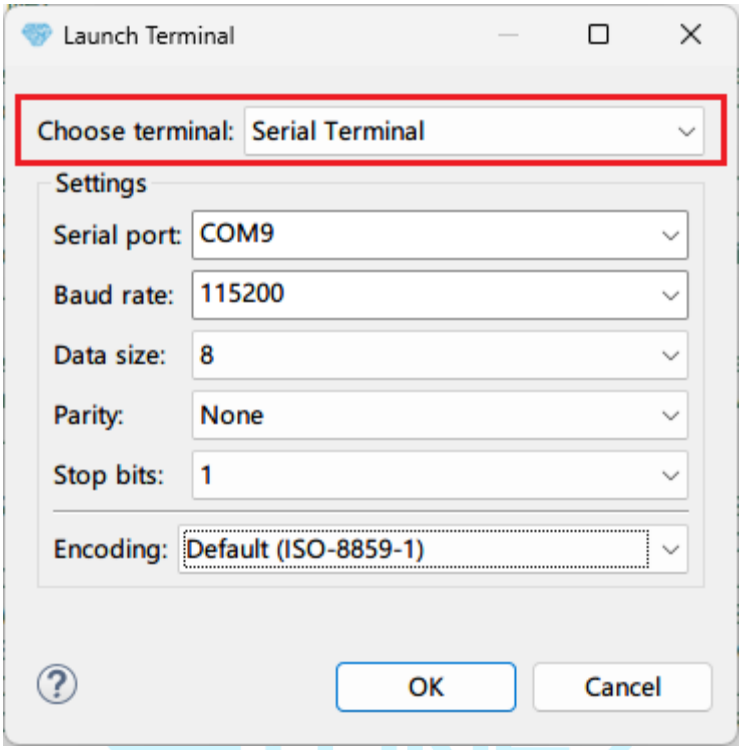



图 57：配置 Terminal

## 4.9. 示例工程迁移

当用户有以下需求时，可按照本节描述进行示例工程迁移。

1. 希望示例工程使用最新的设备驱动程序
2. 未提供 Linux 版本的示例程序需要在 Linux 系统环境中使用

绝大多数情况下，示例工程迁移分两步。

1. 按照“4.1. 新建工程”所述新建工程
2. 将样例*_core0 目录（对于双核示例也包含*_core1 目录）下的程序文件（.h 和.c 文件）拷贝到新建工程对应的*_core0 和*_core1 目录并覆盖已有文件

如图 58 所示将示例工程 eqep_ex4_freq_cal_interrupt 迁移到新建的工程 qx_project_migration。

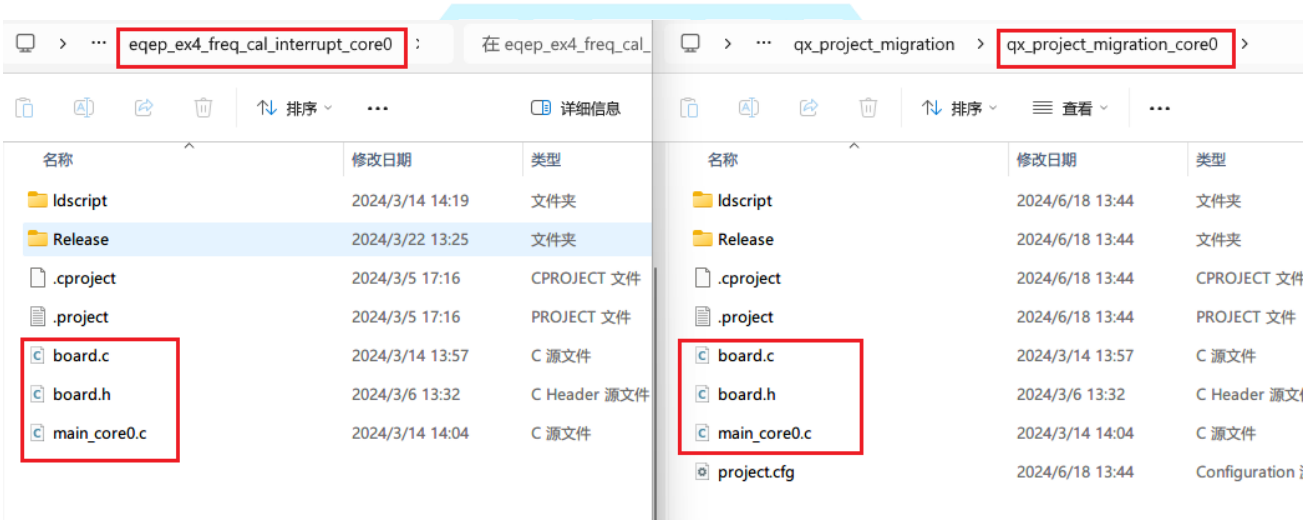



图 58：示例工程迁移举例

## 4.10. 升级应用程序工程

QX-IDE v1.7.3(c)及以上版本支持升级已有应用程序工程的 bootloader、设备驱动库及操作系统源码。右键点击需要升级的应用程序工程（双核工程任选 core0/core1），选择 DriverlibManage -> Update driverlib。

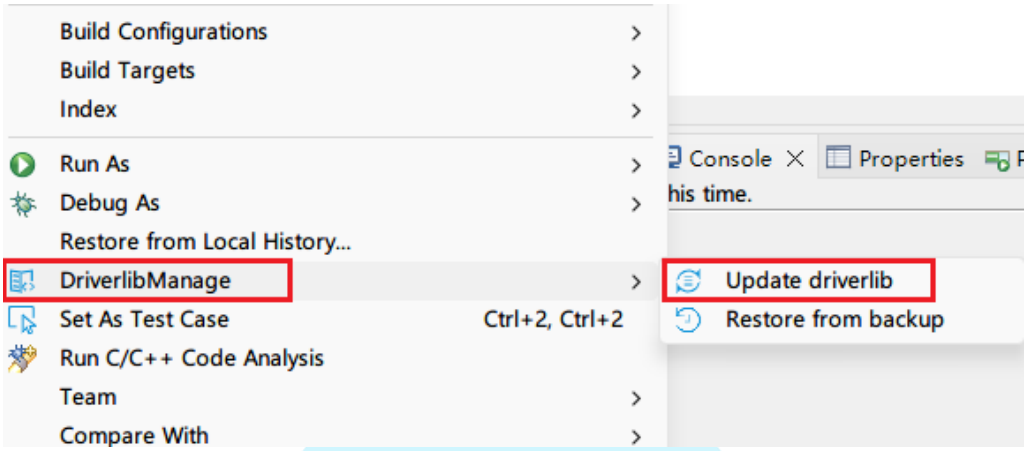



图 59：升级应用程序工程

该操作的升级如下内容

1. bootloader 源码
2. 设备驱动库源码
3. 操作系统源码（仅针对操作系统应用工程）

实际被替换的应用工程文件、文件夹参见 QX-IDE 安装目录下的文件

“plugins\QXTOOLS_<版本号>\qxtools\template\updateList.json”

（其中，<版本号>选择日期最新的目录）

**注意：**升级会覆盖以下用户自定义配置。如有修改，升级后需重新配置。

1. DSP 主频。关联文件如下，

   libs\device\device.h

2. SRAM 划分。关联文件如下，

   libs\ldscript\ldscript_Memory.ld

   libs\ldscript\ldscript_Peripheral.ld

   libs\device\startup.s

   bootloader\bootloadermain.c

升级后，如果发现编译失败（通常是因为“驱动库接口不兼容”或“工程模板结构不兼容”），可以选择 DriverlibManage -> Restore from backup 撤销升级。

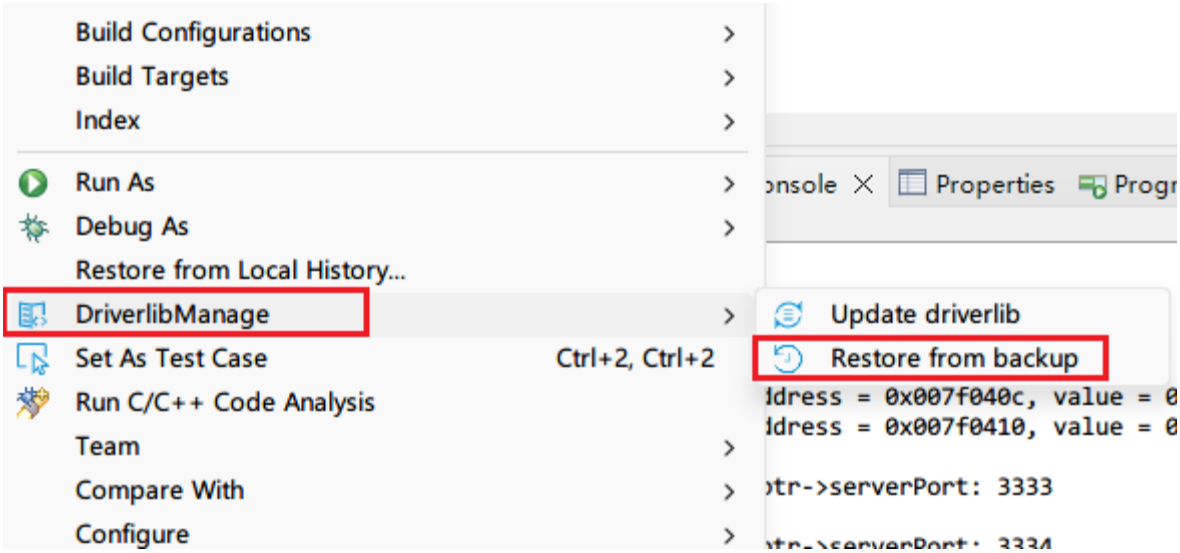



图 60：撤销升级

#### 4.10.1. 升级失败的解决方法

应用工程升级要求“应用工程的芯片名称”和“QX-IDE 中工程模板的芯片名称”一致。从 QX-IDE v1.7.5 开始，芯片名称做了如表 5 所示变动。

表 5：2024 年量产芯片名称变迁

| 索引 | v1.7.5 之前 | v1.7.5 到 v1.7.7 之间 | v1.8.0 及以上 |
|---|---|---|---|
| 1 | QXS320F280049 | QXS320F280049EDB | QXS320F280049RevA |
| 2 | QXS320F28377L | QXS320F28377LEDB | QXS320F28377LRevA |

如果“当前的 QX-IDE”和“新建工程时的 QX-IDE”芯片名称不同，升级将报错如图 61 所示。（后继版本会给出有意义的提示信息）

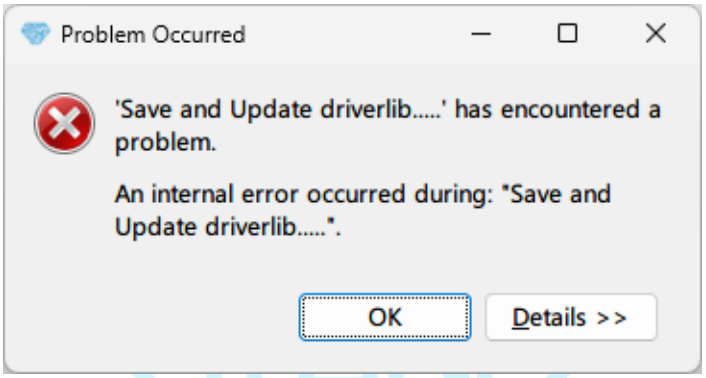



图 61：升级找不到和应用工程芯片名称对应的工程模板

此时，在升级应用工程前，需修改应用工程的“模板名称”及“芯片名称”和当前 QX-IDE 的相同。步骤如下：

1. 打开应用工程根目录，如图 62 所示。

   右键单击应用程序工程，选择“Show In”-> “System Explorer”

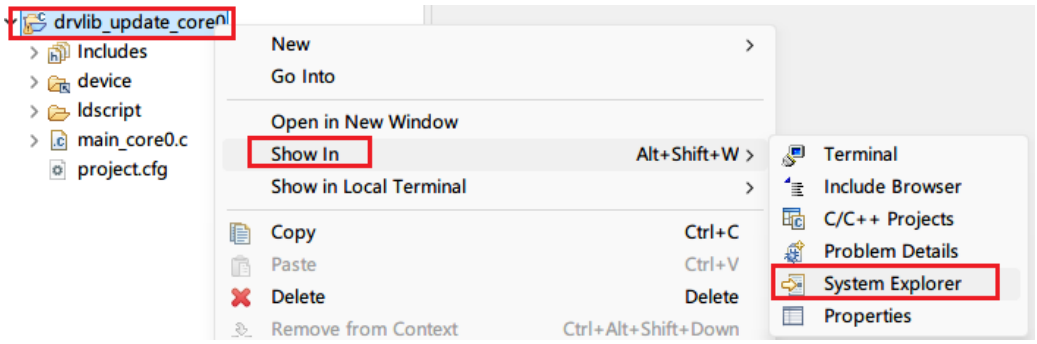



图 62：打开工程所在根目录

2. 编辑根目录下的“project.cfg”文件

   **注意：**不是_core0/_core1 目录下的“project.cfg”文件

   a) 确保“template_name”为对应的模板名称，参考表 6

表 6：模板名称对应关系

| 索引 | 模板名称 | 说明 |
|---|---|---|
| 1 | Empty | 裸机驱动工程 |
| 2 | RT-Thread_Nano_OS | RT-Thread Nano：Core0 OS，Core1 裸机 |
| 3 | RT-Thread_Nano_OS2Core | RT-Thread Nano：Core0 OS，Core1 OS |
| 4 | RT-Thread_OS | RT-Thread：Core0 OS，Core1 裸机 |
| 5 | RT-Thread_OS2Core | RT-Thread：Core0 OS，Core1 OS |
| 6 | uCOS-II_OS | uCOS-II：Core0 OS，Core1 裸机 |
| 7 | uCOS-II_OS2Core | uCOS-II：Core0 OS，Core1 OS |
| 8 | uCOS-III_OS | uCOS-III：Core0 OS，Core1 裸机 |
| 9 | uCOS-III_OS2Core | uCOS-III：Core0 OS，Core1 OS |
| 10 | FreeRTOS_OS | FreeRTOS：Core0 OS，Core1 裸机 |
| 11 | FreeRTOS_OS2Core | FreeRTOS：Core0 OS，Core1 OS |
| 12 | QPC_QK | QPC_QK：Core0 OS，Core1 裸机 |
| 13 | QPC_QV | QPC_QV：Core0 OS，Core1 裸机 |

   b) 确保“specify_chip”和当前 QX-IDE 版本的芯片名称一致

如果“project.cfg”文件不存在，在应用工程根目录下新建该文本文件，内容如下：

(**注意：**将<chip_name>替换为当前 QX-IDE 版本的芯片名称)

```
template_name=<template_name>
chip_family=C2000
project_level=parent_prject
specify_chip=<chip_name>
```

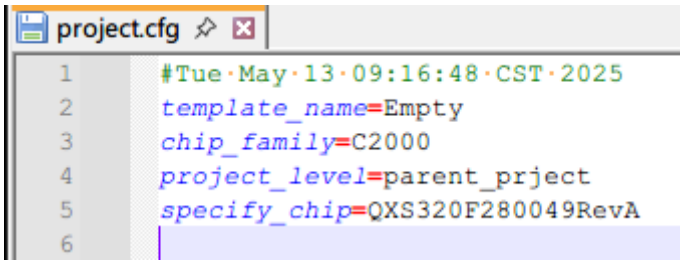



图 63：适用于 v1.8.0 的 project.cfg 文件示例

3. 保存“project.cfg”文件，然后按照“4.10. 升级应用程序工程”再次尝试升级

若仍然升级失败，则需手动迁移应用程序 或者 手动更新应用工程软件库。

### 4.10.2. 手动迁移应用程序

新建工程，然后迁移应用程序到新工程。步骤如下：

1. 按照“4.1. 新建工程”新建工程
2. 复制应用程序源码到新建的工程

   仅复制应用程序源码到新建工程的对应目录即可。如，复制原_core0 目录下的应用程序源码到新建工程的_core0 目录下
3. 如有需要，按照“4.3.1. 工具链参数设置”配置工具链参数，主要是头文件搜索路径 Include paths
4. 编译工程

### 4.10.3. 手动升级应用工程软件库

使用原应用程序工程，从工程模板更新其软件库。参考《QXC2000DSP 开发快速指引》v1.1.3 及以上版本，章节 4。

## 4.11. 工程重命名

在 Project Explorer 窗口右键单击要重命名的工程，选择 Properties；在弹出的窗口选择 Base General，如图 64 所示。

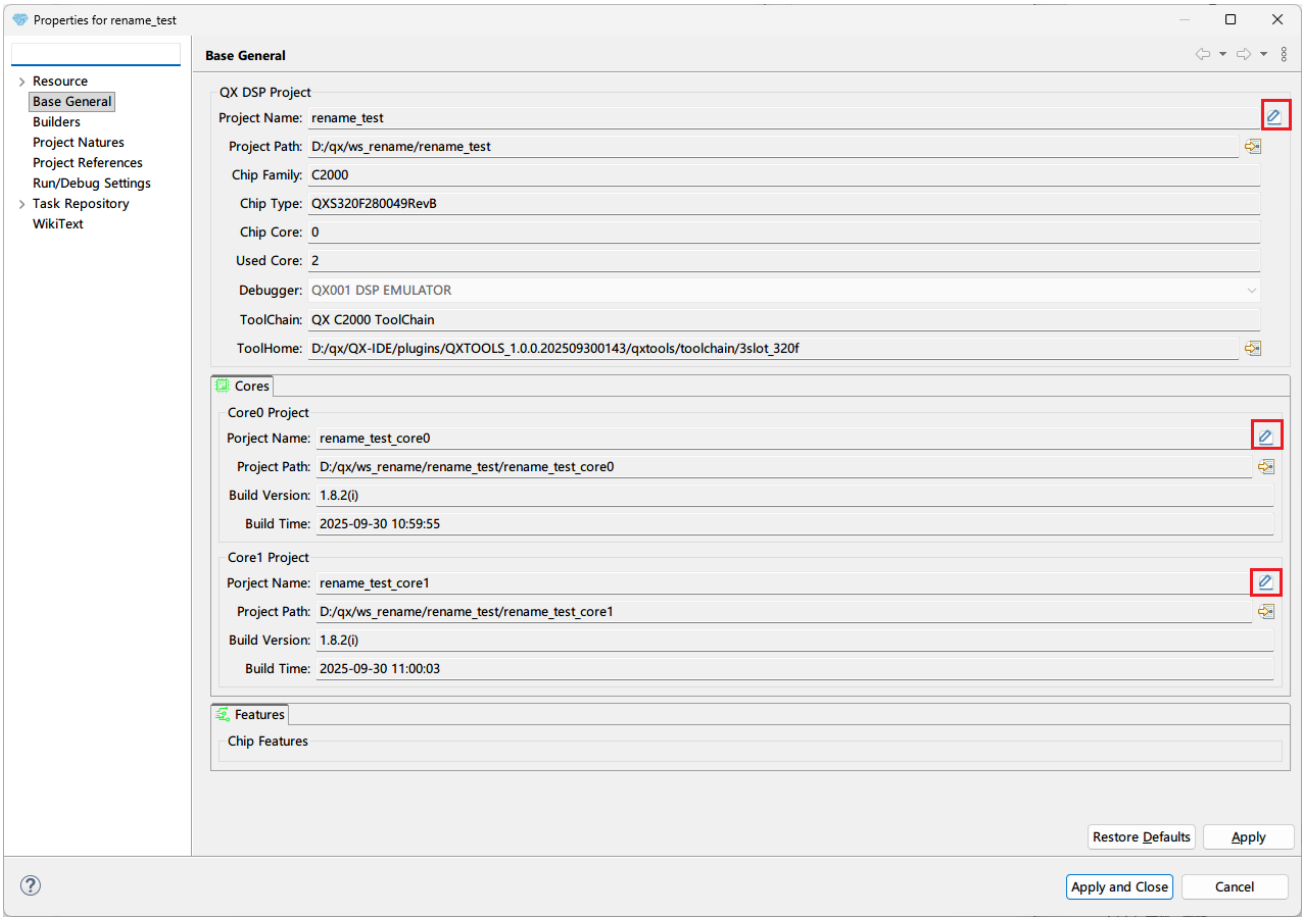



图 64：工程 Properties，Base General 页面

点击图 64 红框中任意一个图标，在弹出的窗口中输入新工程名称，点击 OK。

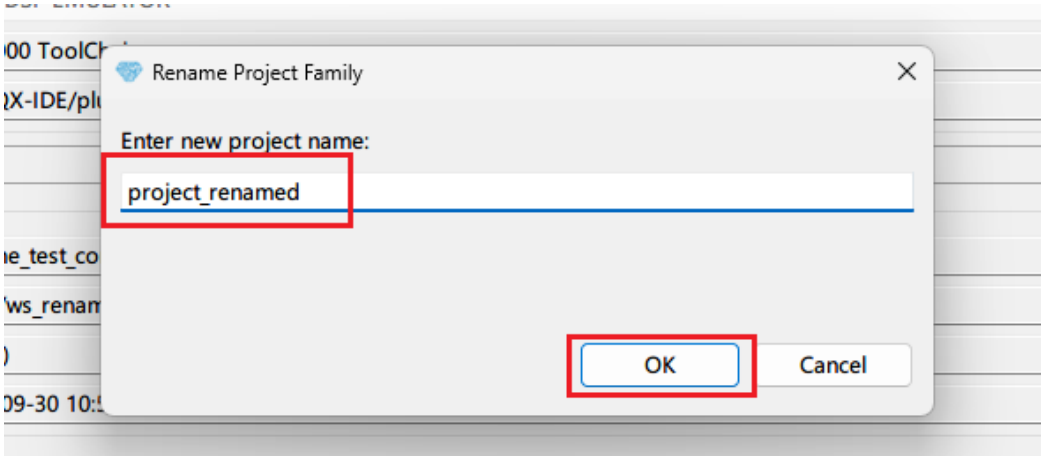



图 65：输入新工程名并确认

## 4.12. 编辑器自动保存

菜单栏选择 Windows -> Preferences。

选择 General -> Editors -> Autosave。选中“Enable autosave for dirty editors”，自动保存时间间隔最小可设置为 1s。如图 66 所示。

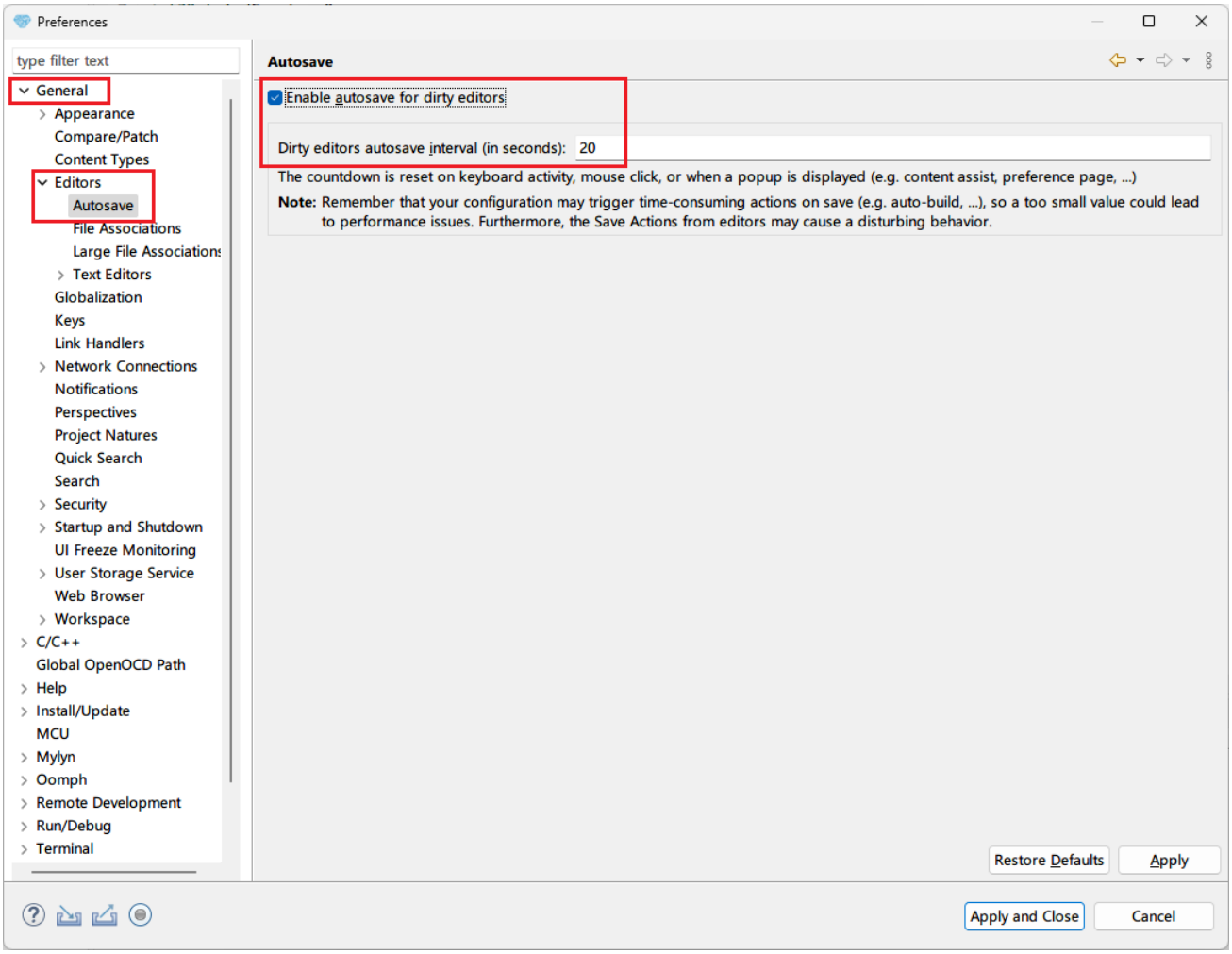



图 66：设置编辑器自动保存

## 4.13. 迁移应用程序工程到 VSCode

我司还开发了 VSCode 插件，让用户通过 VSCode 使用 QXDSP。

由于“应用工程的识别”以及“编译框架”不兼容（VSCode 通过 cmake，Eclipse 通过.project 和.cproject 文），须手动将 QX-IDE 应用程序迁移到 VSCode，参考“4.10.2. 手动迁移应用程序”。

## 4.14. 引脚功能规划辅助

我司正在开发基于图形界面的芯片模块功能配置工具，目前已经开放引脚功能规划辅助功能，本节介绍使用方法。当前支持的芯片型号如表 7 所示。

表 7：支持的芯片型号及封装

| 索引 | 芯片型号 | 封装 |
|---|---|---|
| 1 | QXS320F280049RevA | 100-Pin PZ |
| 2 | | 56-Pin RSH |
| 3 | | 64-Pin PM |

### 4.14.1. 下载、更新

菜单栏单击“Help”，选择“Update QXSystemCfg Tools”，等待下载完成。

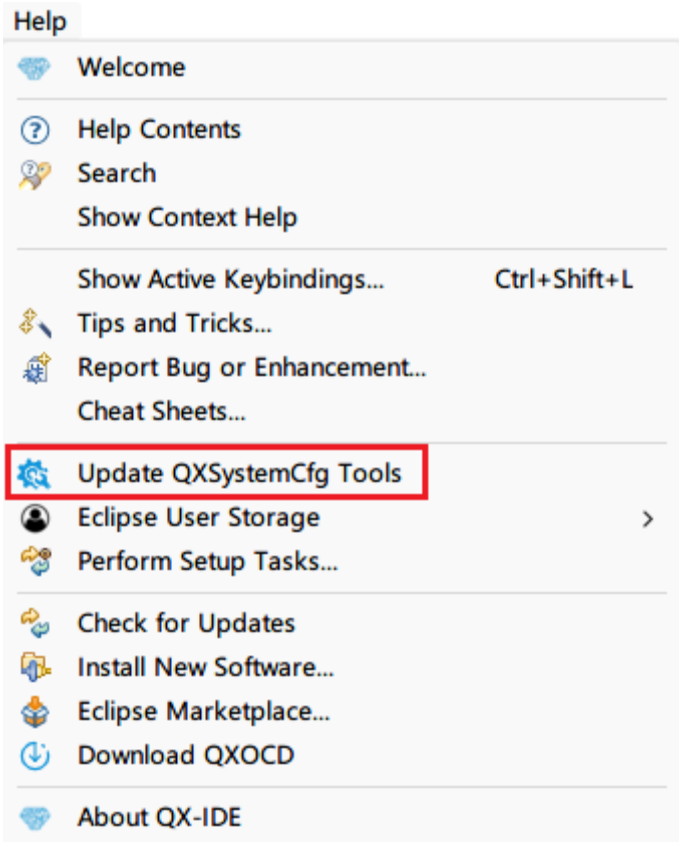



图 67：工具下载、更新

### 4.14.2. 在已有工程新建.syscfg 文件

在已有工程右键单击，依次选择“New”->“File”，在弹出的界面中输入.syscfg 文件名，点击“Finish”。

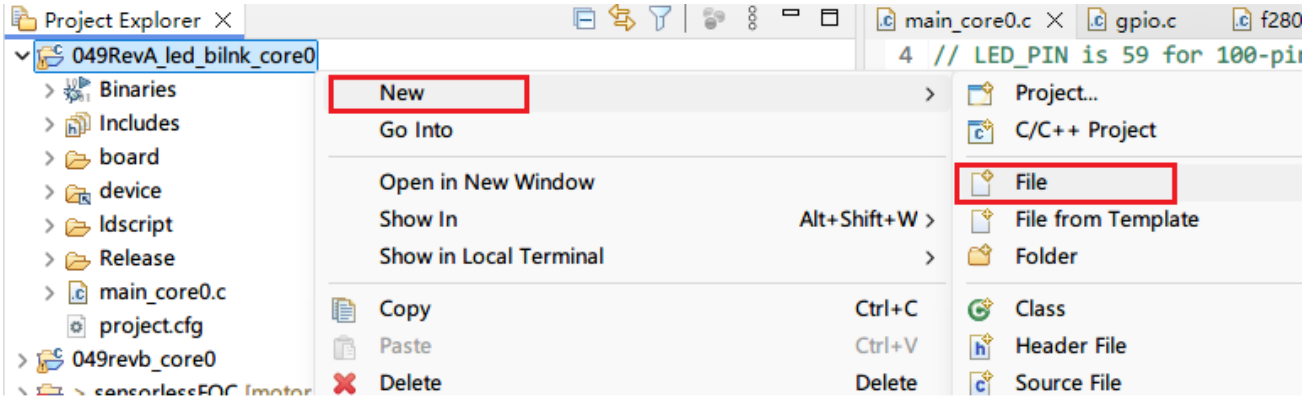



图 68：新建文件

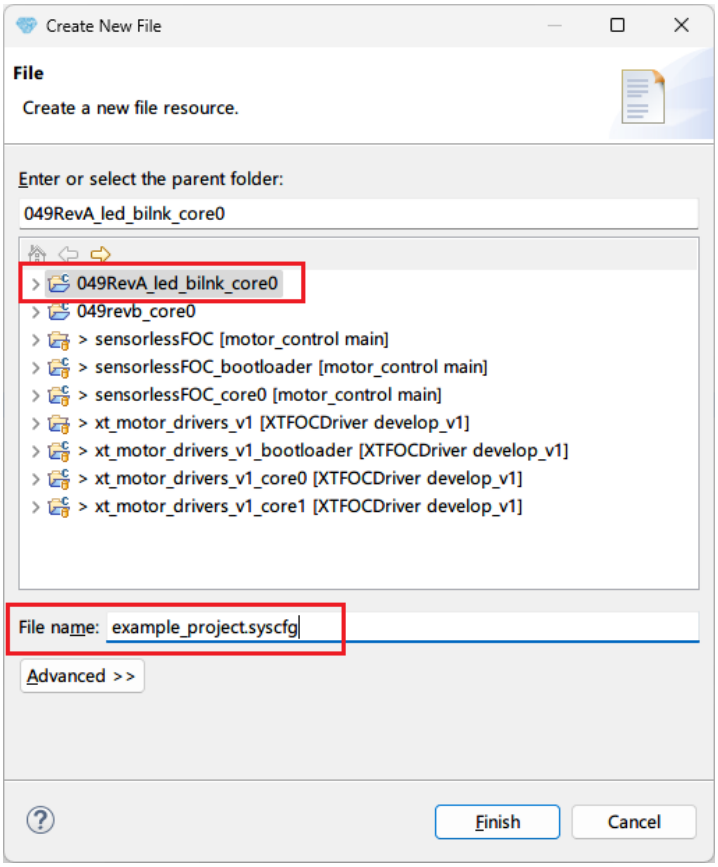



图 69：文件明后缀为.syscfg

### 4.14.3. 使用工具

双击.syscfg 文件，选择需要的“芯片型号”及“封装”，点击“开始”。

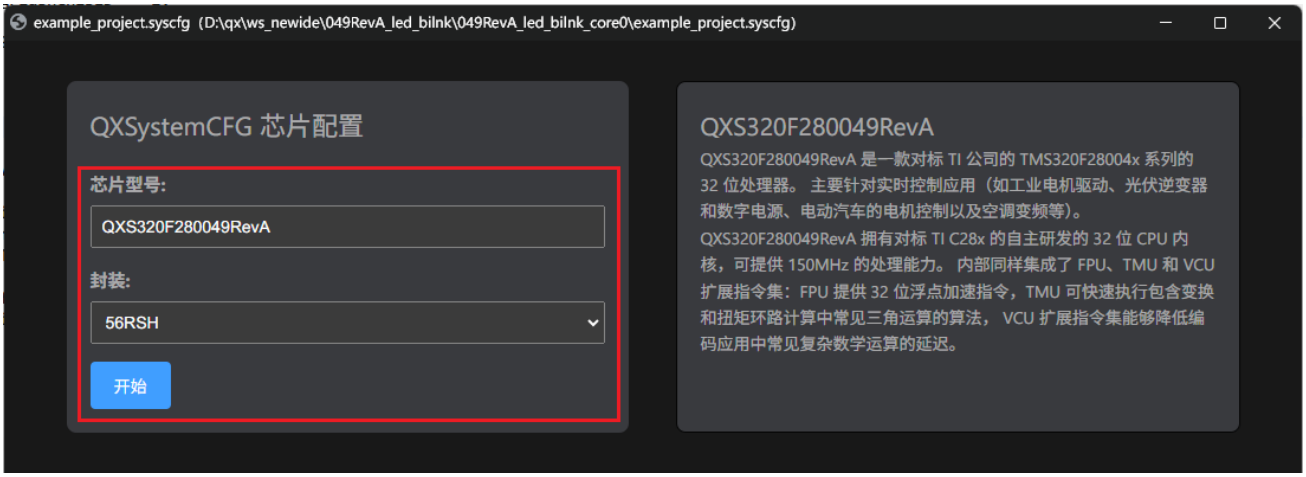



图 70：配置选择

在弹出窗口的最右侧可以进行引脚规划。

可通过“CTRL+S”或者点击左上角的“导出配置文件”保存配置，以便下次双击.syscfg 文件继续上次的工作。

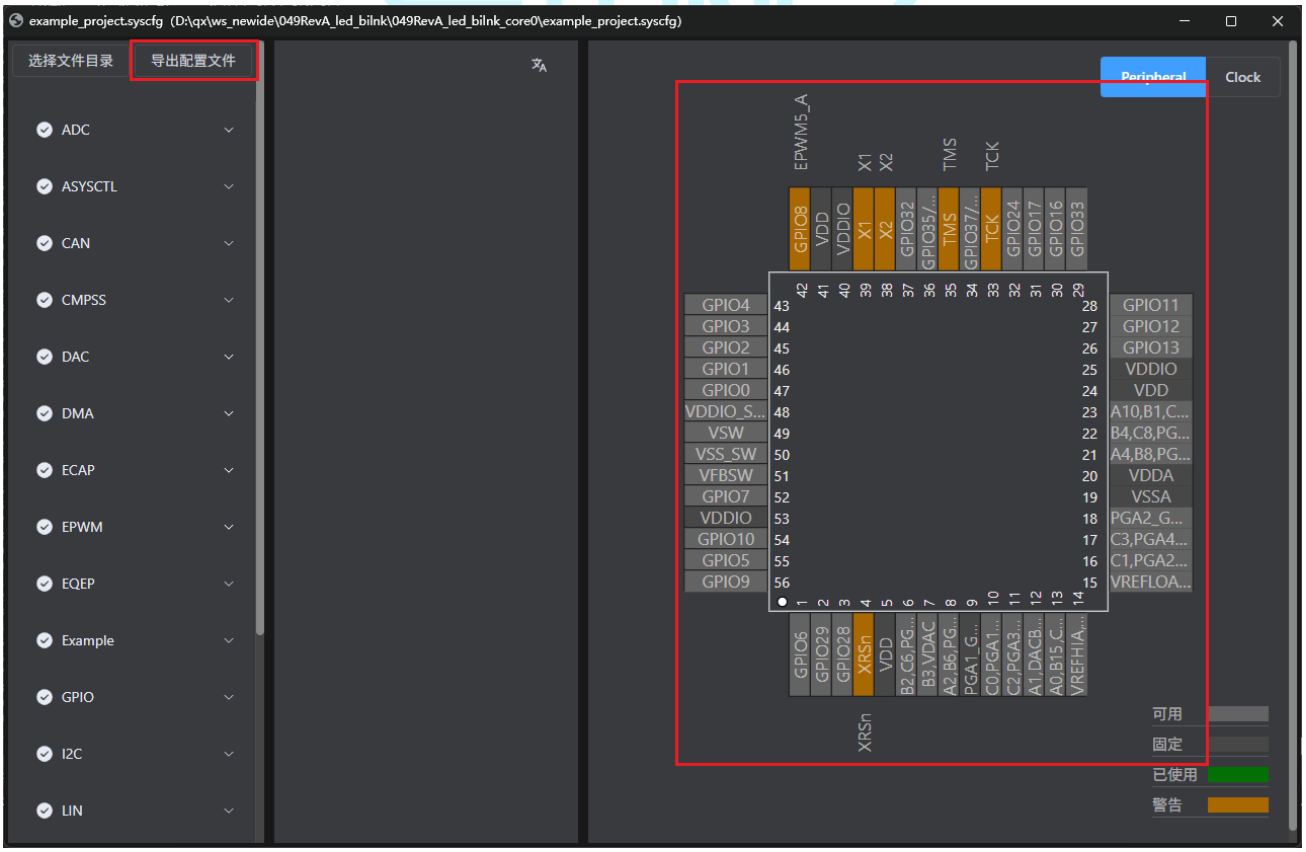



图 71：配置界面

# 5. 支持更多的 JTAG 调试器

工具链最初仅支持“基于 FT2232HL 的 JTAG 调试器”调试 QXDSP 及烧写 Flash。我司还适配了 OpenOCD，支持使用其它 JTAG 调试器（如 DAPLink）开发 QXDSP 应用程序。

本节介绍 OpenOCD 的配置及使用方法。

## 5.1. 下载 OpenOCD

更新 QX-IDE 到 1.8.1 及以上版本。标题栏单击“Help”->“Download QXOCD”，等待弹出的下载窗口自动关闭，代表下载成功。

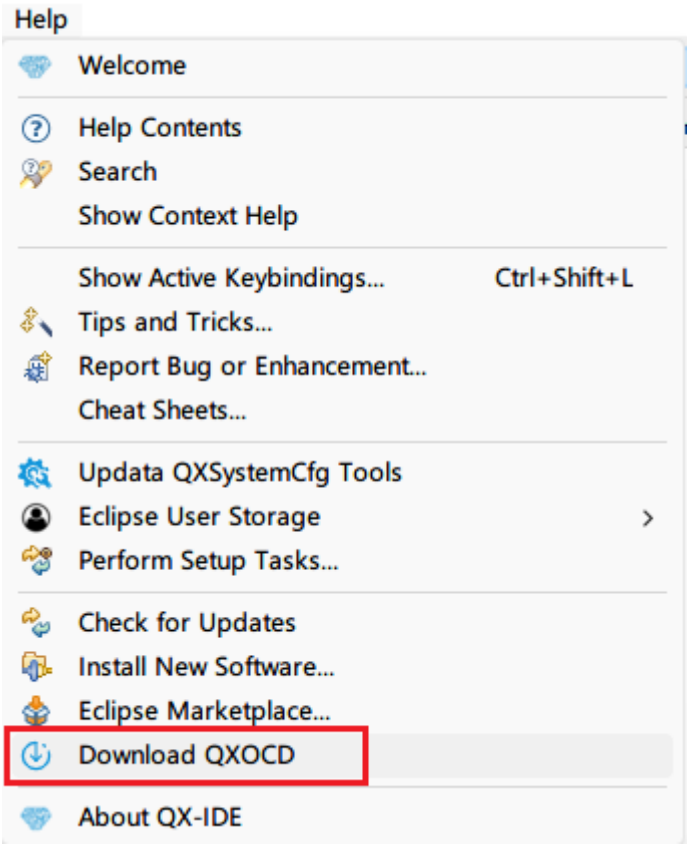



图 72：下载 QXOCD

## 5.2. 使用 OpenOCD 调试

右键单击要调试的工程，在弹出的菜单依次选择“Debug As”->“Debug Configurations...”。

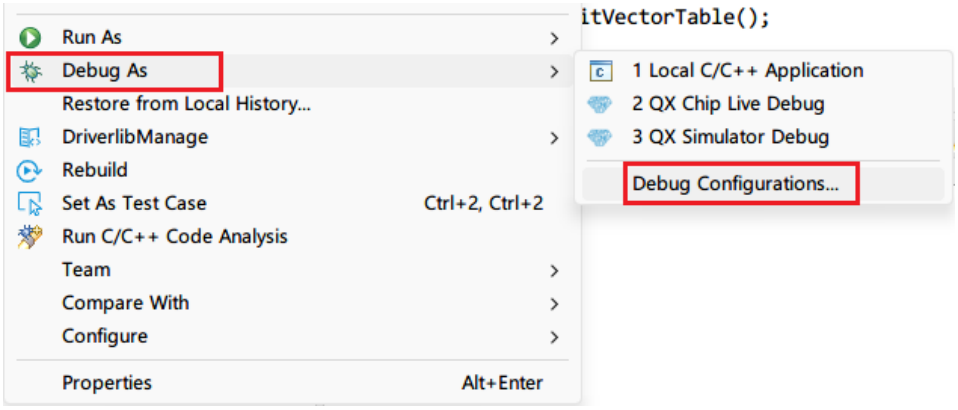



图 73：进入调试配置

在弹出的窗口双击 QXOPENOCD，然后点击 Debug 开始调试。其中，可将 Name 修改成更具标识的名称，方便后继调试。

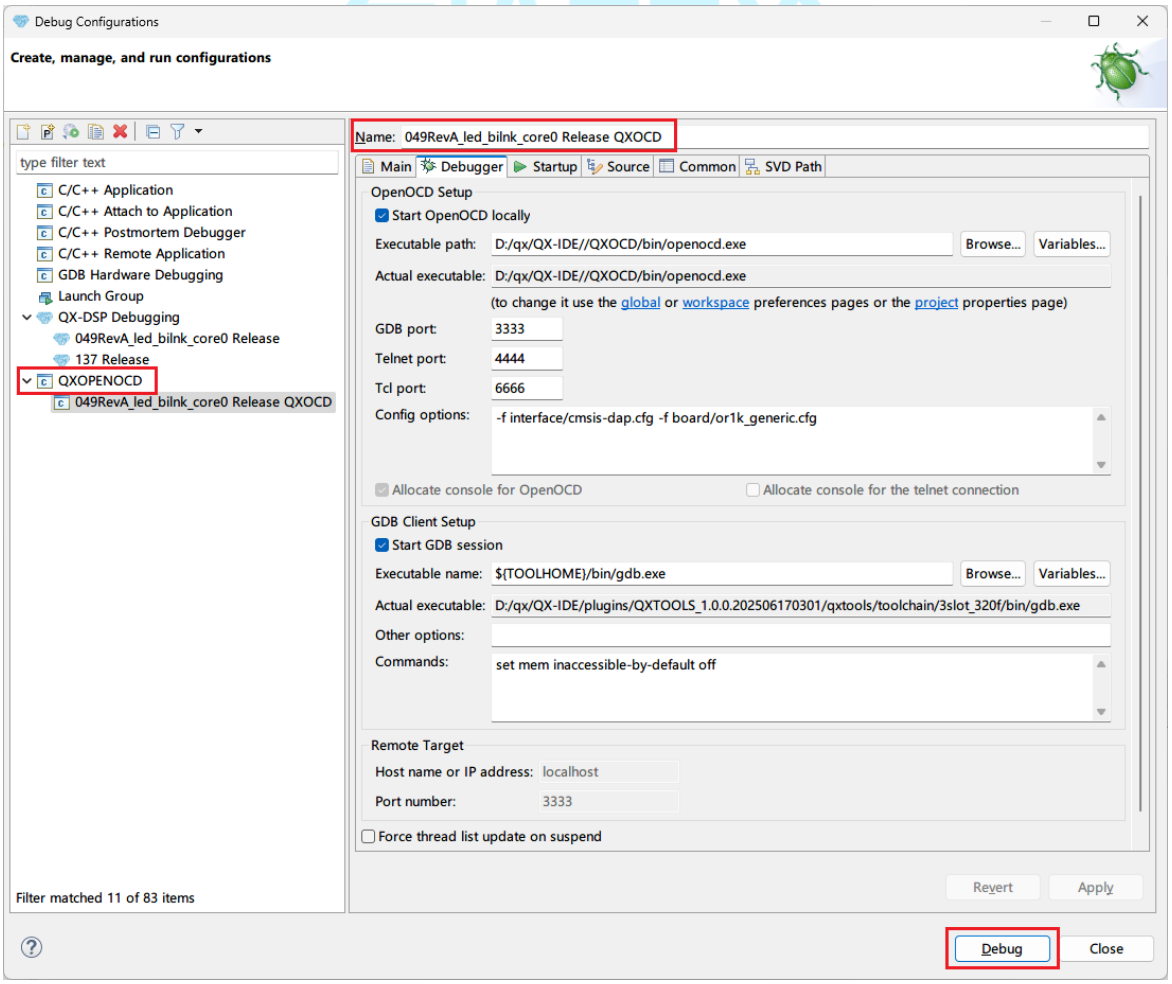



图 74：调试配置

## 5.3. Flash 烧写

参考 4.7.2. 执行烧写。

# 6. QXC2000DSP 工具链

QXC2000DSP 工具链是和 QXC2000DSP 芯片配套的一系列软件开发工具，提供了编译、链接、打包、调试等一系列开发支持。QXC2000DSP 提供了编译器，链接器，调试器等一系列软件开发工具。本章介绍了 QXC2000DSP 的软件构建流程，目录结构等基本信息。

## 6.1. 工具概览

QXC2000DSP 工具链目前提供了以下工具：

| 工具路径 | 工具描述 |
| --- | --- |
| bin/clang.exe | 基于 LLVM 的 C 编译器 |
| bin/llc.exe | LLVM IR 转汇编/目标文件工具 |
| bin/ld.exe | 基于 GNU binutils 的链接器 |
| bin/lld.exe | 基于 LLVM 的新链接器，提供了更快的链接速度和代码优化。 |
| bin/as.exe | 基于 GNU binutils 的汇编器 |
| bin/objdump.exe | 基于 GNU binutils 的 objdump 工具，提供查看目标文件信息，反汇编等功能。 |
| bin/gdb.exe | 基于 GNU binutils 的 GDB 调试器，提供多种调试功能。 |
| bin/simulator_step13.exe | QXC2000DSP 结构仿真器 |
| bin/simulator_iss.exe | QXC2000DSP 功能仿真器 |
| bin/or_debug_proxy.exe | QXC2000DSP 芯片调试 Proxy |

QXC2000DSP 工具链也提供了这些工具的 Linux 版本。

QXC2000DSP 工具链实现了对 C 语言的完整支持，提供了 C 标准库(libc.a)，数学库(libm.a)的实现。为方便调试，QXC2000DSP 工具链支持 open，write 等常见系统调用，允许用户通过仿真器和 GDB 直接对调试机器的文件系统进行访问，从而允许进行文件读取，日志输出等操

作。

本文档接下来分章节介绍使用 QXC2000DSP 工具链进行软件开发所涉及的各个工具。

第二章节介绍 QXC2000DSP 工具链 C 编译器的使用。

第三章节介绍 QXC2000DSP 工具链汇编器的使用。

第四章节介绍 QXC2000DSP 工具链接器的使用。

第五章节介绍 QXC2000DSP 工具链功能仿真器，结构仿真器的使用。

第六章节介绍 QXC2000DSP 工具链调试 Proxy 的使用。

第七章节介绍 QXC2000DSP 工具链提供的其他工具，包括 GDB 调试器等。

第八章节介绍 QXC2000DSP 提供的预编译库，包括运行时库，C 标准库和数学库。

## 6.2. 编译器

QXC2000DSP 工具链提供了编译器将 C 源码编译为 QXC2000DSP 系列芯片支持的目标文件。这些目标文件后续可以被链接器使用从而生成最终的可执行目标文件。QXC2000DSP 工具链编译器目前提供了完整的 C 语言支持。

### 6.2.1. 编译器简介

QXC2000DSP 工具链基于 LLVM 开源编译器基础设施项目进行开发。LLVM 开源编译器提供了一系列用于编译程序的工具，包括 C/C++编译器 clang。LLVM 项目的优点是模块化，可移植性好，优化能力强。QXC2000DSP 工具链基于 LLVM 项目对 QXC2000DSP 系列芯片进行了适配，从而实现对您的代码进行大量优化，针对 QXC2000DSP 系列芯片生成高效代码。

QXC2000DSP 工具链尚未对 C++程序提供完整支持和测试，您可以尝试使用 C++编译器(clang++)对 C++代码进行编译，但是，包括并不限于使用了以下特性的 C++代码可能不能正常编译或生成正确代码：

C++17 及以上的 C++语言特性。

使用 C++标准库定义的类、变量、函数等。

C++异常。

### 6.2.2. 使用 C 编译器

QXC2000DSP 工具链 C 编译器的基本使用方法为：

```
clang -target dsp -mcpu=qx320f [选项] [C 文件路径] -o [输出文件路径]
```

在选项中，需要指定生成文件的格式。选项-c 指定 C 编译器生成目标文件。选项-S 指定 C 编译器生成汇编文件。

实例：以 windows 下生成为例，假设需要生成的源文件为 source.c，工具链位置为<toolhome>。则生成目标文件的编译命令为：

```
<toolhome>/bin/clang.exe -target dsp -mcpu=qx320f source.c -c -o source.o
```

如果想要生成汇编文件，则编译命令中需要其中的-c 选项修改为-S。即：

```
<toolhome>/bin/clang.exe -target dsp -mcpu=qx320f source.c -S -o source.S
```

也可以先使用 C 编译器生成 LLVM IR 中间代码文件，再使用 llc 生成目标文件。llc 工具的基本使用方法为：

```
llc -target dsp -mcpu=qx320f [选项] [中间文件路径] -filetype=[输出类型] -o [输出文件路径]
```

输出类型可以是 asm（输出汇编代码），obj（输出目标文件）。llc 提供了更多细粒度的代码额外生成控制。如果不加任何额外选项，这样生成的目标代码和直接生成目标代码是一致的。

实例：以 windows 下生成为例，假设需要生成的源文件为 source.c，工具链位置为<toolhome>。则生成目标文件的编译命令为：

```
<toolhome>/bin/clang.exe -target dsp -mcpu=qx320f source.c -S -emit-llvm -o source.ll
<toolhome>/bin/llc.exe -target dsp -mcpu=qx320f source.ll -filetype=obj -o source.o
```

如果想要生成汇编文件，则需要把 llc 中的-filetype=obj 换为-filetype=asm。即：

```
<toolhome>/bin/llc.exe -target dsp -mcpu=qx320f source.ll -filetype=asm -o source.S
```

### 6.2.3. 使用 C 编译器优化代码

QXC2000DSP 工具链可以使用-O 选项开启编译器的优化功能，对 C 代码生成的指令进行优化。QXC2000DSP 工具链 C 编译器支持以下优化等级：

-O0：这个优化等级下不会进行任何优化。通常这个优化等级只适用于进行调试。

-O1：这个优化等级会进行一定的优化。

-O2：这个优化等级会启用大多数的优化算法。

-O3：这个优化等级会启用激进的代码效率优化，包括进行一些可能会耗费大量数据内存

空间的操作。

-Ofast：这个优化等级会启用-O3 的所有优化以及其他可能违反严格遵守语言标准的激进优化。

-Os：这个优化等级在 O2 优化的基础上会进行额外的代码尺寸(Code Size)优化。

-Oz：这个优化等级在 Os 的基础上进一步优化代码尺寸。

更多的详细信息请参考 LLVM 项目对优化等级的说明[1]。

### 6.2.4. 命令行选项

Clang/LLVM 提供了丰富的命令行选项对编译器进行控制，并有详细的命令行选项文档[2]。QXC2000DSP 工具链在其基础上提供了若干用于控制生成 QXC2000DSP 系列芯片代码的命令行选项，如下所示。

| 命令行选项 | 描述 | 可选值 |
|---|---|---|
| -mcpu=<cpu> | 指定芯片型号 | qx320f<br>qx320f049v2 |
| -mfpmath=<mode> | 指定 double 精度。默认情况下 double 为 32 位。 | fp32,fp64 |
| -mllvm --fix-debug | 修复函数后的断点在函数未执行前命中的问题，会增加代码尺寸。 | N/A |
| -mllvm --nopack | 强制在每条 VLIW 中只放入一条指令。 | N/A |
| -mllvm --disable-branch-opt | 关闭分支延迟槽优化，会增加代码尺寸。 | N/A |

### 6.2.5. 内建函数

QXC2000DSP 工具链编译器提供了一系列内建函数，这些内建函数直接对应 QXC2000DSP 系列芯片单条汇编指令，从而允许您手动控制编译器生成的汇编指令。例如，您可以使用内建函数在 C 代码中调用 TMU 指令，优化对三角函数的计算。所有的内建函数定义存放在 qx320f.h 头文件中。内建函数的描述记录在附录 A 中。

[1] https://clang.llvm.org/docs/CommandGuide/clang.html#code-generation-options
[2] https://clang.llvm.org/docs/CommandGuide/clang.html

### 6.2.6. 内联汇编

在 C 源码中使用以下语法插入一条汇编指令

```
__asm (“<VLIW>”);
```

要插入多条汇编指令，使用一条或多条__asm 语句

- 多条__asm 语句

```
__asm (“<VLIW_0>”);
__asm (“<VLIW_1>”);
```

- 一条__asm 语句

```
__asm (“<VLIW_0>\n”
       “<VLIW_1>\n”);
```

其中，<VLIW>为 QXDSP 的一条 VLIW。QXDSP 包含 3 个指令槽，<VLIW>语法为

```
<VLIW> ::= slot0_指令 | slot1_指令 | slot2_指令
```

说明：

1. slot0_指令，slot1_指令，slot2_指令 不可同时为空
2. 各 slot 支持的指令见《数字信号处理器 QXS320C28x 指令手册》

## 6.3. 汇编器

QXC2000DSP 工具链汇编器将汇编代码转换为 QXC2000DSP 系列芯片支持的目标文件。

### 6.3.1. 使用汇编器

QXC2000DSP 工具链汇编器的基本使用方法为：

```
as [汇编文件路径] -o [目标文件路径]
```

**实例：**以 windows 下生成为例，假设需要生成的源文件为 source.S，工具链位置为<toolhome>。则生成目标文件的汇编命令为：

```
<toolhome>/bin/as.exe source.S -o source.o
```

### 6.3.2. 汇编器语法

QXC2000DSP 系列芯片为超长指令字(Very Long Intruction Word, VLIW)架构，在一条超长指令字中，具有多个指令槽位用于存放不同类型的指令。在 QXC2000DSP 汇编器的语法中，一行指令被视为一条超长指令字，用“|”作为分隔符。以下为例子：

```
 add gr0,gr1 | store32 gr2, gr30, 1 | load32 gr3, gr30, 2
|store gr0, gr30,3| # 不可省略分隔符“|”,即使对应槽位未存放指令
```

QXC2000DSP 汇编器支持的指令见 QXC2000DSP 指令手册。

## 6.4. 链接器

QXC2000DSP 工具链链接器用于将多个目标文件链接为单个可执行目标文件，并解决其中的符号依赖。链接器通过以下步骤完成链接过程：

1. 文件解析：解析输入的目标文件、共享库和可执行文件，提取它们的符号表、重定位表和其他必要信息。

2. 符号解析：分析每个输入文件中的符号，解析它们的引用和定义。这有助于建立符号之间的关系。

3. 地址分配：分配地址空间，将各个输入文件中的节（sections）映射到适当的地址。这包括重定位地址的计算。

4. 符号解析和重定位：将符号引用解析为符号定义的地址，并进行必要的重定位操作，以

确保符号引用正确。

5. 生成输出文件：将所有输入文件链接成最终的可执行程序或共享库，并生成输出文件。

QXC2000DSP 提供了 ld（旧连接器）和 lld（新链接器）两个链接器。新链接器基于 LLVM lld 项目移植，提供了更快的链接速度，链接时对代码尺寸的优化。

### 6.4.1. 使用链接器

QXC2000DSP 工具链链接器的基本使用方法为：

```
ld [目标文件路径]… -o [可执行目标文件路径] -L [库文件夹] -T[链接脚本路径] -l[链接静态库]
```

实例：以 windows 下生成可执行目标文件为例，假设工具链位置为<toolhome>，需要将 source1.o 和 source2.o 两个目标文件进行链接生成对应的可执行目标文件 source.out，同时该 source1.o 和 source2.o 依赖于 C 标准库和 C 数学库。则生成目标文件的命令为：

```
<toolhome>/bin/ld.exe source1.o source2.o -o source.out -L "<toolhome>/lib" -T link_8slots.x -lm -lc -
lrt -ldspsim
```

使用 lld 链接器的命令和 ld 类似，只需要将链接脚本替换为 lld 的版本：

```
<toolhome>/bin/lld.exe source1.o source2.o -o source.out -L <toolhome>/lib -T link_8slots.lds -lm -lc -
lrt -ldspsim
```

注意到链接过程中需要链接四个相关静态库文件：libm.a，libc.a，librt.a，libdspsim.a。libm 提供了 C 标准库中定义的数学函数实现。libc 提供了 C 标准库的实现。librt 提供了编译器运行时库的实现。libdspsim 提供了系统调用桩函数的实现。librt 中包含了编译器在编译 C 程序时可能隐式调用的相关函数，当编译器需要编译某些 QXC2000DSP 系列芯片无法原生支持的操作时（例如 64 位浮点转 32 位浮点），则编译器会将这些操作转换为对库函数的调用，通过调用库函数完成这些操作。

### 6.4.2. 常用命令行参数

| 命令行参数 | 描述 |
| --- | --- |
| -L | 添加库搜索路径 |
| -l | 指定要链接的库 |
| -Map | 生成链接过程中的映射文件 |
| -T | 指定链接时使用的链接脚本文件 |
| -o | 指定链接后生成的可执行文件名称 |

## 6.5. 仿真器

QXC2000DSP 工具链提供了对 QXC2000DSP 系列芯片的仿真环境。您可以使用仿真器在主机上运行 QXC2000DSP 代码，并借用主机侧的 IO 功能，一般用于仿真环境/调试环境。QXC2000DSP 工具链提供了功能仿真器（simulator_ISS.exe）和结构仿真器（simulator_step13.exe）两种仿真器。其中功能仿真器提供了对 QXC2000DSP 指令的模拟，而结构仿真器对 QXC2000DSP 系列芯片的流水线结构进行仿真，更贴近真机环境。

### 6.5.1. 使用仿真器

仿真器的使用主要可以分为以下两种方式：

连接 gdb 使用。在这种情况下需要先启动仿真器程序，再使用 gdb remote 功能连接至仿真器使用。

对单独文件进行仿真。在这种情况下仿真器从主机端读入 QXC2000DSP 可执行文件，对程序进行仿真。在该模式下首先需要将编译器编译出的.out 可执行文件通过 trobjdat_8slot.py 脚本转换成仿真器可接受的.dat 格式文件，之后再使用仿真器进行模拟。trobjdat_8slot.py 的使用方法为：

```
python .\trobjdat_8slot.py [.out 文件路径]
```

该指令会在工作目录下生成同名.dat 文件。接下来可以使用功能仿真器或结构仿真器运行该.dat 文件。功能仿真器和结构仿真器的使用方法略有不同，功能仿真器的使用方法为：

```
.\simulator_ISS.exe -s [.dat 文件路径]
```

在使用结构仿真器时还需要加入-t 参数，即：

```
.\simulator_step13.exe -s [.dat 文件路径] -t
```

实例：以 windows 下使用功能仿真器对 source.out 进行独立仿真为例，具体的操作如下：

```
python .\trobjdat_8slot.py source.out
.\simulator_ISS.exe -s .\source.dat -log
```

其中，-log 参数会使仿真器输出 log 至 simulator.log 文件。

### 6.5.2. 命令行选项

在仿真器的运行中我们可以通过输入命令行参数或者修改配置文件的方式来更改仿真器的一些配置，下面列出的是有关仿真器的一些命令行参数：

-d：以 debug 模式运行仿真器，通常用于与 gdb 连接，默认端口号为 3333；

-s：必选参数，后面需要输入希望模拟的 dat 文件；

-log：loglevel 决定仿真器输出的 log 等级，其中 loglevel 为 1 代表 error，2 代表 info，3 代表 debug，默认输出 log 名为 simulator.log;

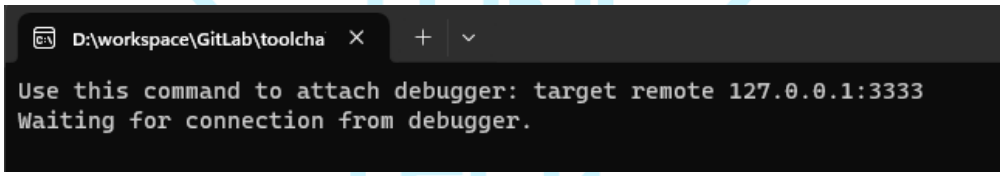
--help：输出仿真器的帮助信息，显示命令行参数的形式与意义。

## 6.6. 调试 Proxy

proxy 是 GDB 和 JTAG 模块的交互中介，负责在 JTAG cable 和 GDB 之间转发消息。它将 GDB 发送的命令与数据使用 JTAG 驱动将其转换成 JTAG cable 需要的 USB 信号，并最终传输到芯片上。同时它将通过 JTAG cable 返回的信号重新转换成 GDB 能够识别的 RSP 协议格式返回给 GDB。

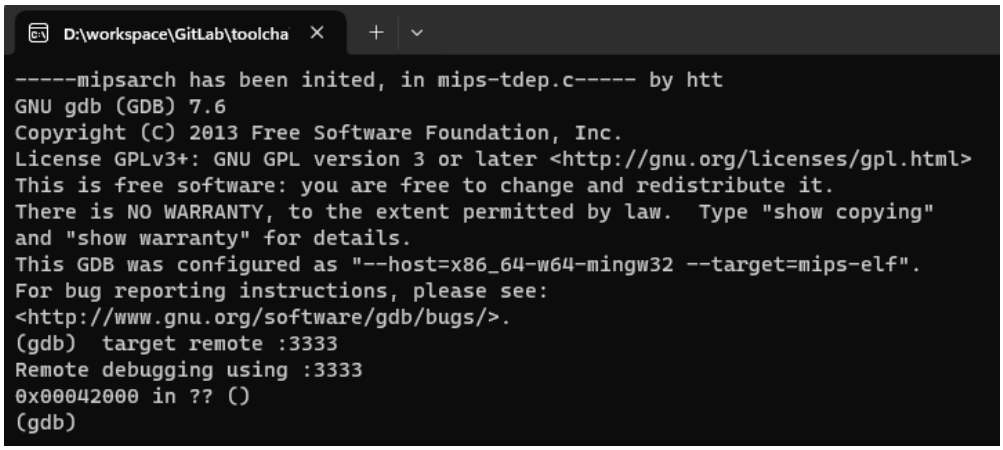
### 6.6.1. 使用调试 Proxy

打开 proxy，确认本机上的对应端口号正确打开。例如仿真器系列软件的默认端口号为 3333。

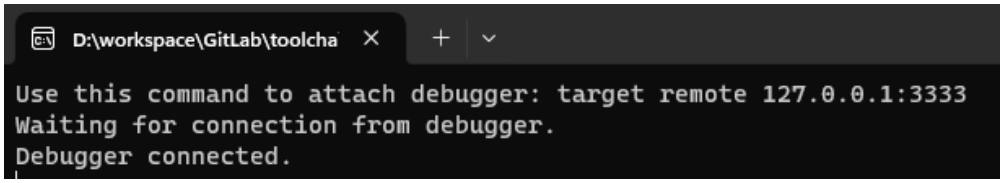



指示 gdb 连接已经打开的端口号，打开 gdb，输入命令

```
target remote :3333
```



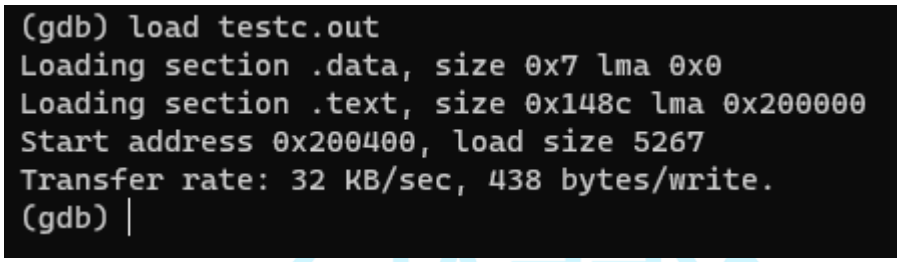
proxy 显示“Debugger connected.”，说明连接成功：




实例：以 Windows 下对 testc.out 进行调试为例。

1. 首先根据上述操作成功建立 Proxy 和 GDB 之间的连接。载入对应已经编译好的 out 可执行文件。XXXX 为任意文件名，输入命令

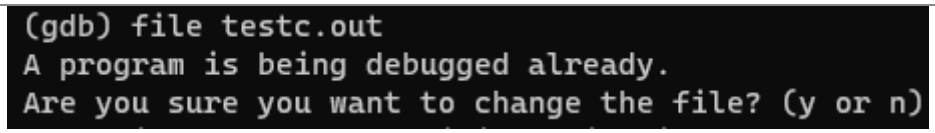
```
load testc.out
```




该文件为 elf 格式，并且带有 dwarf 相关的调试信息。要生成带有 dwarf 相关调试信息的文件，需要在本工具链下的 clang 相关 flags 内加入-g 标志。只有带有 dwarf 调试信息的可执行文件才能在 gdb 调试环境下拥有全部功能。

2. 载入相关编译好 out 可执行文件的符号表。该命令对应的文件与上一步 load 命令的可执行文件名一致，输入命令

```
file testc.out
```




输入 y 完成载入。

3. 使用 gdb，该功能与其他开发环境下的 gdb 指令完全一致，（启动直接使用 c 命令）

注：支持查看变量(p)，源代码(list)，反汇编(disassem)，寄存器(info registers)，汇编级别下一步(si)等功能，满足绝大部分场景开发需要。

## 6.7. 其他工具

GDB 是开源的支持多语言、多处理器架构的源码级别调试器。它具有强大功能的同时具有很好的移植性，可能是目前兼容性最广泛的调试器之一。QXC2000DSP 工具链提供了 QXC2000DSP 系列芯片的 GDB 调试器。您可以使用 GDB 调试器的一系列调试功能在 QXC2000DSP 系列芯片上进行调试。

### 6.7.1. 使用 GDB 调试器

QXC2000DSP 工具链 GDB 调试器支持在开发板或仿真器运行程序并调试。使用 GDB 调试器需要先通过 GDB remote 功能连接到调试 Proxy 或仿真器进程，向调试 Proxy/仿真器进程发送调试指令进行调试。具体用法为运行 GDB 后输入以下指令：

```
target remote ipaddredd:port-number
```

当 ipaddress 为本机地址时可简写为：

```
target remote :<port-number>
```

连接上 Proxy 或仿真器后，便可以使用 GDB 进行程序调试了，具体用法和一般的 GDB 相同，您可以查阅 GDB 文档了解 GDB 调试器的详细功能和用法3。

实例：以 Windows 下对 addc.out 进行调试为例。

连接仿真器/调试 Proxy。首先将 GDB 连接到 Proxy 或仿真器上

```
target remote :3333
```

```
(gdb) target remote :3333
Remote debugging using :3333
0x00002000 in ?? ()
(gdb)
```

加载目标文件：

```
load xxx.out
```

```
(gdb) load addc.out
Loading section .rodata, size 0x4 lma 0x0
Loading section .data, size 0x7 lma 0x4
Loading section .text, size 0x7dc lma 0x200000
Start address 0x200400, load size 2023
Transfer rate: 17 KB/sec, 144 bytes/write.
(gdb)
```

[3] https://sourceware.org/gdb/current/onlinedocs/gdb.html/

根据架构读取目标文件到 GDB：

```
file addc.out
```

```
(gdb) file addc.out
A program is being debugged already.
Are you sure you want to change the file? (y or n)
```

执行指令后会提示是否改变当前文件，输入 y 以继续执行：

```
Are you sure you want to change the file? (y or n) y
-----mipsarch has been inited, in mips-tdep.c----- by htt
Reading symbols from C:\Toolchain\toolchains\toolchain_3slot_320f\bin\addc.out...done.
(gdb)
```

至此文件加载成功。接下来可以使用 GDB 的各项功能进行调试。下面简单演示常用的 GDB 指令。注意：对于程序流的控制，GDB 无法使用 run 命令，而是使用 continue 命令来替代 run。

1）使用 list 或 l 指令显示源码。

```
(gdb) l
5       #else
6       #define TEST_OUT(a)
7       #endif
8
9       #ifdef DSP_VALIDATION
10      #include <swift_debug.h>
11      #endif
12
13      int main(){
14              long long a = 22200000000;
(gdb) list
15              long long b = -22200000000;
16              long long c = a + b;
17              TEST_OUT(c);
18      #ifdef DSP_VALIDATION
19              dbg_output(&c, 1, 4);
20      #endif
21              return 0;
22      }
(gdb) list
Line number 23 out of range; ..\testsuites\scalar\addc.c has 22 lines.
```

2）使用 b 指令添加断点。

```
(gdb) b main
Breakpoint 1 at 0x200750: file ..\testsuites\scalar\addc.c, line 14.
(gdb) b 16
Breakpoint 2 at 0x200764: file ..\testsuites\scalar\addc.c, line 16.
(gdb) b 21
Breakpoint 3 at 0x2007b4: file ..\testsuites\scalar\addc.c, line 21.
(gdb)
```

3）使用 info breakpoints 命令显示断点信息。

```
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00200750 in main at ..\testsuites\scalar\addc.c:14
2       breakpoint     keep y   0x00200764 in main at ..\testsuites\scalar\addc.c:16
3       breakpoint     keep y   0x002007b4 in main at ..\testsuites\scalar\addc.c:21
```

4）使用 delete 命令删除指定断点。

```
(gdb) delete 3
(gdb) i b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00200750 in main at ..\testsuites\scalar\addc.c:14
2       breakpoint     keep y   0x00200764 in main at ..\testsuites\scalar\addc.c:16
(gdb)
```

5）使用 disable 命令禁用断点。

```
(gdb) disable 2
(gdb) i b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00200750 in main at ..\testsuites\scalar\addc.c:14
2       breakpoint     keep n   0x00200764 in main at ..\testsuites\scalar\addc.c:16
```

6）使用 enable 命令启用断点。

```
(gdb) enable 2
(gdb) i b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00200750 in main at ..\testsuites\scalar\addc.c:14
2       breakpoint     keep y   0x00200764 in main at ..\testsuites\scalar\addc.c:16
```

7）使用 continue 命令使程序继续执行。

```
(gdb) c
Continuing.

Breakpoint 1, main () at ..\testsuites\scalar\addc.c:14
14              long long a = 22200000000;
```

8）使用 disassemble 命令显示反汇编。

```
(gdb) disas
Dump of assembler code for function main:
   0x00200720 <+0>:     addi gr30 gr30 0xfffffffc0
   0x00200724 <+4>:     store32 gr31 gr30 0xf
   0x00200728 <+8>:     movigh gr2 0x0 <dbg_addr>
   0x0020072c <+12>:    movigl gr2 0x0 <dbg_addr>
   0x00200730 <+16>:    movigh gr3 0x0 <dbg_addr>
   0x00200734 <+20>:    movigl gr3 0x5 <_stump+1>
   0x00200738 <+24>:    movigh gr4 0x2b39
   0x0020073c <+28>:    movigl gr4 0x1e00
   0x00200740 <+32>:    store32 gr2 gr30 0xe
   0x00200744 <+36>:    movigh gr5 0xffffd4c6
   0x00200748 <+40>:    movigl gr5 0xffffe200
   0x0020074c <+44>:    movigh gr6 0xffffffff
=> 0x00200750 <+48>:    movigl gr6 0xfffffffa
   0x00200754 <+52>:    store32 gr4 gr30 0xc
```

9）使用 info register 命令打印寄存器值。

```
(gdb) info r
          r0       r1       r2       r3       r4       r5       r6       r7
R0    00000000 00000000 00000000 00000005 2b391e00 d4c6e200 ffff0000 00000000
          r8       r9      r10      r11      r12      r13      r14      r15
R8    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
         r16      r17      r18      r19      r20      r21      r22      r23
R16   00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
         r24      r25      r26      r27      r28      r29      r30      r31
R24   00000000 00000000 00000000 00000000 00000000 00000000 001ffdc0 0020041c
          o0       o1       o2       o3       b0       b1       b2       b3
R32   00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
          m0       m1       m2       m3       pc       sp       ra       ps
R40   00000000 00000000 00000000 00000000 00200750 00200750 00200750 00200750
```

10）使用 x 命令打印内存信息。

```
(gdb) x/16xw 0x00200750
0x200750 <main+48>:     0x146fffa5      0xec478182      0xec3781a2      0xec678162
0x200760 <main+64>:     0xec578142      0x18478100      0xa8378162      0x94600001
0x200770 <main+80>:     0x14600045      0xa8578142      0x14800001      0xa87781a2
0x200780 <main+96>:     0x14800015      0xa8978182      0x80000000      0x80000000
```

11）使用 quit 命令退出程序。

```
(gdb) quit
A debugging session is active.

        Inferior 1 [Remote target] will be killed.

Quit anyway? (y or n) y
```

## 6.7.2. GDB 基本命令

### 6.7.2.1. 查看基本信息命令

| 命令 | 缩写 | 说明 |
|---|---|---|
| help | h | 显示帮助信息 |
| help command | h command | 显示命令描述信息 |
| apropos key-word | N/A | 显示搜索的关键词的相关信息 |
| info | i | 显示断点、线程等信息 |
| info breakpoints | i b | 显示断点信息 |
| info breakpoint number | i b number | 显示指定断点的信息 |
| info watchpoints | i watch | 显示观察点信息 |
| info registers | i r | 显示寄存器信息 |
| info threads | i threads | 列出当前的线程 |

### 6.7.2.2. 文件目录操作命令

| 命令 | 缩写 | 说明 |
|---|---|---|
| info files | i files | 显示当前的文件 |
| info share | i share | 显示当前的共享库 |
| file file | N/A | 把 file 当作调试的程序 |
| core file | N/A | 把 file 当作 core 文件 |
| exec file | N/A | 把 file 当作执行程序 |
| symbol file | N/A | 从 file 中读取符号表 |
| load file | N/A | 动态链入 file 文件 |
| path directory | N/A | 把目录 directory 加入到搜索可执行文件和符号文件的路径中 |
| directory directory-name | N/A | 在源代码路径前添加指定的目录 |
| show directories | N/A | 显示源代码目录 |

#### 6.7.2.3. 查看源码信息命令

| 命令 | 缩写 | 说明 |
| --- | --- | --- |
| list | l | 显示源码 |
| list line-number | l line-number | 显示指定行数的源码 |
| list function-name | l function-name | 显示指定名称函数的源码 |
| list start, end | l start, end | 显示指定范围区间的源码 |
| list file-name:function-name | l file-name:function-name | 显示指定文件中指定函数的源码 |
| set list-size number | N/A | 设置 list 命令打印源代码时的行数 |
| show list-size number | N/A | 显示 list 命令打印源代码时的行数 |

#### 6.7.2.4. 程序流命令

| 命令 | 缩写 | 说明 |
| --- | --- | --- |
| continue | c | 继续执行直到下一个断点或观察点 |
| continue number | c number | 继续执行并忽略当前的断点 number 次 |
| kill | N/A | 停止程序执行 |
| quit | q | 退出 GDB |

#### 6.7.2.5. 断点操作命令

| 命令 | 缩写 | 说明 |
| --- | --- | --- |
| break | b | 添加断点 |
| break function-name | b function-name | 在指定的函数处设置断点 |
| break line-number | b line-number | 在指定的行数设置断点 |
| break +offset<br>break -offset | b +offset<br>b -offset | 在当前位置指定偏移行数位置设置断点 |
| break file-name:function-name | b file-name:function-name | 在指定文件的指定函数处设置断点 |
| break file-name:line-number | b file-name:line-number | 在指定文件的指定行数处设置断点 |
| break *address | b *address | 在指定的地址处设置断点 |
| break line-number if condition | b line-number if condition | 如果条件满足在指定位置设置断点 |
| break line thread thread-number | b line thread thread-number | 在指定的线程中中断 |
| tbreak | tb | 设置临时断点 |
| watch condition | N/A | 在条件满足时设置观察点 |
| clear | N/A | 清除所有断点 |
| clear function-name | N/A | 清除指定函数处的断点 |
| clear line-number | N/A | 清除指定行数的断点 |
| delete | del | 删除所有的断点或观察点 |
| delete breakpoint-number | del breapoint-number | 删除指定断点号的断点、观察点 |
| delete breakpoint-range | del breakpoint-range | 删除指定范围断点号的断点（eg: delete 1-4） |
| disable breakpoint-number | N/A | 禁用指定断点号的断点 |

| | | |
|---|---|---|
| disable breakpoint-range | N/A | 禁用指定范围断点号的断点 |
| enable once breakpoint-number | N/A | 设置指定断点有效，当到达断点时设置为无效 |
| enable del breakpoint-number | N/A | 设置指定断点有效，当到达断点时删除它 |
| finish | N/A | 继续执行到函数结束 |

#### 6.7.2.6. 调试命令

| 命令 | 缩写 | 说明 |
|---|---|---|
| step | s | 进入下一行代码的执行，会进入函数内部 |
| next | n | 执行下一行代码，但不会进入函数内部 |
| until line-number | - | 继续运行到指定行号 |
| until function-name | - | 继续运行到指定函数 |
| return | - | 弹出选中的栈帧 |
| return expression | - | 返回表达式的值 |
| stepi | si | 执行下一条汇编指令，进入函数 |
| nexti | ni | 执行下一条汇编指令，不进入函数 |
| where | - | 显示当前的行号和所处的函数 |

#### 6.7.2.7. 栈帧信息相关命令

| 命令 | 缩写 | 说明 |
|---|---|---|
| backtrace | bt | 显示当前的堆栈信息 |
| backtrace full | bt full | 显示所有局部变量的值 |
| frame | f | 切换到当前调用线程的指定堆栈 |
| frame frame-number | f frame-number | 选择指定的栈帧 |
| up number | - | 向上移动指定个数个栈帧 |
| down number | - | 向下移动指定个数的栈帧 |
| info frame frame-number | i frame frame-number | 描述选中的栈帧 |

#### 6.7.2.8. 打印变量命令

| 命令 | 缩写 | 说明 |
| --- | --- | --- |
| print var-name | p var-name | 打印指定变量的值。 |
| print file:var-name | p file:var-name | 打印指定文件中指定变量的值。 |
| print *array-name@length | p *array-name@length | 打印数组 array-name 中的前 length 项。 |
| print/type var-name | p/type var-name | 以 type 的格式打印变量的值。 |

type 的取值如下：

| Type | 说明 |
| --- | --- |
| x | 16 进制 |
| d | 有符号整数 |
| u | 无符号整数 |
| o | 8 进制 |
| c | 字符 |
| f | 浮点数 |
| t | 2 进制 |

#### 6.7.2.9. 其他打印命令

| 命令 | 缩写 | 说明 |
| --- | --- | --- |
| disassemble | disas | 对当前位置进行反汇编 |
| disassemble addr | disas addr | 对指定位置进行反汇编 |
| x/nfu address | - | 查看指定内存地址的数据，n 表示显示的内存单元个数，f 表示显示方式，u 表示一个地址单元的长度 |

显示方式：

| Type | 说明 |
| --- | --- |
| x | 16 进制 |
| d | 10 进制 |
| u | 10 进制无符号 |
| o | 8 进制 |
| t | 2 进制 |
| a | 16 进制 |
| i | 指令地址格式 |
| c | 字符格式 |
| f | 浮点格式 |

单元长度：

| Type | 说明 |
| --- | --- |
| u | 一个地址单元的长度 |
| b | 单字节 |
| h | 双字节 |
| w | 四字节 |
| g | 八字节 |

## 6.8. 库

### 6.8.1. 运行时库

QXC2000DSP 工具链提供了预编译的运行时库（librt.a），用于为硬件不支持的低级功能提供特定于目标的支持。例如，QXC2000DSP 系列芯片缺少支持 64 位除法的指令。librt.a 通过提供特定于 QXC2000DSP 系列芯片并经过优化的 64 位除法函数来解决这个问题，该函数使用 32 位指令实现了 64 位除法。

使用-lrt 可以在链接器中链接运行时库。

### 6.8.2. C 标准库

QXC2000DSP 工具链提供了预编译的 C 标准库，提供了一系列 C 标准规定的通用接口的实现。目前 QXC2000DSP 支持了 ISO C99 标准所定义的绝大部分接口。使用-lc 可以在链接器中链接 C 标准库。

### 6.8.3. 数学库

QXC2000DSP 工具链提供了预编译的数学库，提供了 C 标准中 math.h 头文件规定的数学函数的实现。使用-lm 可以在链接器中链接数学库。

# 附录

## A. 内建函数描述表

要使用内建函数，须包含头文件 qx320f.h （#include <qx320f.h>）

### A.1. 浮点运算指令(FPU)

| 函数名称 | 参数列表 | 返回值类型 | 描述 |
|---|---|---|---|
| __macf | float a, float b | float | 对应 FSMAC，a 对应 Rs，b 对应 Rt。 |
| __sqrtf | float a | float | 对应 FSSQRT，a 对应 Rs。 |
| __absf | float a | float | 对应 FSABS，a 对应 Rs。 |
| __maxf | float a, float b | float | 对应 FSMAX，a 对应 Rs，b 对应 Rt。 |
| __minf | float a, float b | float | 对应 FSMIN，a 对应 Rs，b 对应 Rt。 |

### A.2. 三角函数指令(TMU)

| 函数名称 | 参数列表 | 返回值类型 | 描述 |
|---|---|---|---|
| __sinpuf | float a | float | 对应 SINPUF32，a 对应 Rs。 |
| __cospuf | float a | float | 对应 COSPUF32，a 对应 Rs。 |
| __quadf | float a, float b | float | 对应 QUADF，a 对应 Rs，b 对应 Rt。 |
| __mpy2pif | float a | float | 对应 MPY2PIF32，a 对应 Rs |
| __div2pif | float a | float | 对应 DIV2PIF32，a 对应 Rs |
| __expf | float a | float | 对应 EXPPUF32，a 对应 Rs |
| __logf | float a | float | 对应 LOGPUF32，a 对应 Rs |

# B. GDB 基本命令

## B.1. 查看基本信息命令

| 命令 | 缩写 | 说明 |
|---|---|---|
| **help** | h | 显示帮助信息 |
| **help command** | h command | 显示命令描述信息 |
| **apropos key-word** | N/A | 显示搜索的关键词的相关信息 |
| **info** | i | 显示断点、线程等信息 |
| **info breakpoints** | i b | 显示断点信息 |
| **info breakpoint number** | i b number | 显示指定断点的信息 |
| **info watchpoints** | i watch | 显示观察点信息 |
| **info registers** | i r | 显示寄存器信息 |
| **info threads** | i threads | 列出当前的线程 |

## B.2. 文件目录操作命令

| 命令 | 缩写 | 说明 |
|---|---|---|
| **info files** | i files | 显示当前的文件 |
| **info share** | i share | 显示当前的共享库 |
| **file file** | N/A | 把 file 当作调试的程序 |
| **core file** | N/A | 把 file 当作 core 文件 |
| **exec file** | N/A | 把 file 当作执行程序 |
| **symbol file** | N/A | 从 file 中读取符号表 |
| **load file** | N/A | 动态链入 file 文件 |
| **path directory** | N/A | 把目录 directory 加入到搜索可执行文件和符号文件的路径中 |
| **directory directory-name** | N/A | 在源代码路径前添加指定的目录 |
| **show directories** | N/A | 显示源代码目录 |

### B.3. 查看源码信息命令

| 命令 | 缩写 | 说明 |
| --- | --- | --- |
| list | l | 显示源码 |
| list line-number | l line-number | 显示指定行数的源码 |
| list function-name | l function-name | 显示指定名称函数的源码 |
| list start, end | l start, end | 显示指定范围区间的源码 |
| list file-name:function-name | l file-name:function-name | 显示指定文件中指定函数的源码 |
| set list-size number | N/A | 设置 list 命令打印源代码时的行数 |
| show list-size number | N/A | 显示 list 命令打印源代码时的行数 |

### B.4. 程序流命令

| 命令 | 缩写 | 说明 |
| --- | --- | --- |
| continue | c | 继续执行直到下一个断点或观察点 |
| continue number | c number | 继续执行并忽略当前的断点 number 次 |
| kill | N/A | 停止程序执行 |
| quit | q | 退出 GDB |

### B.5. 断点操作命令

| 命令 | 缩写 | 说明 |
| --- | --- | --- |
| break | b | 添加断点 |
| break function-name | b function-name | 在指定的函数处设置断点 |
| break line-number | b line-number | 在指定的行数设置断点 |
| break +offset<br>break -offset | b +offset<br>b -offset | 在当前位置指定偏移行数位置设置断点 |
| break file-name:function-name | b file-name:function-name | 在指定文件的指定函数处设置断点 |
| break file-name:line-number | b file-name:line-number | 在指定文件的指定行数处设置断点 |
| break *address | b *address | 在指定的地址处设置断点 |
| break line-number if condition | b line-number if condition | 如果条件满足在指定位置设置断点 |
| break line thread thread-number | b line thread thread-number | 在指定的线程中中断 |
| tbreak | tb | 设置临时断点 |
| watch condition | N/A | 在条件满足时设置观察点 |
| clear | N/A | 清除所有断点 |
| clear function-name | N/A | 清除指定函数处的断点 |
| clear line-number | N/A | 清除指定行数的断点 |
| delete | del | 删除所有的断点或观察点 |
| delete breakpoint-number | del breapoint-number | 删除指定断点号的断点、观察点 |

| | | |
|---|---|---|
| delete breakpoint-range | del breakpoint-range | 删除指定范围断点号的断点（eg: delete 1-4） |
| disable breakpoint-number | N/A | 禁用指定断点号的断点 |
| disable breakpoint-range | N/A | 禁用指定范围断点号的断点 |
| enable once breakpoint-number | N/A | 设置指定断点有效，当到达断点时设置为无效 |
| enable del breakpoint-number | N/A | 设置指定断点有效，当到达断点时删除它 |
| finish | N/A | 继续执行到函数结束 |

### B. 6. 调试命令

| 命令 | 缩写 | 说明 |
|---|---|---|
| step | s | 进入下一行代码的执行，会进入函数内部 |
| next | n | 执行下一行代码，但不会进入函数内部 |
| until line-number | - | 继续运行到指定行号 |
| until function-name | - | 继续运行到指定函数 |
| return | - | 弹出选中的栈帧 |
| return expression | - | 返回表达式的值 |
| stepi | si | 执行下一条汇编指令，进入函数 |
| nexti | ni | 执行下一条汇编指令，不进入函数 |
| where | - | 显示当前的行号和所处的函数 |

### B. 7. 栈帧信息相关命令

| 命令 | 缩写 | 说明 |
|---|---|---|
| backtrace | bt | 显示当前的堆栈信息 |
| backtrace full | bt full | 显示所有局部变量的值 |
| frame | f | 切换到当前调用线程的指定堆栈 |
| frame frame-number | f frame-number | 选择指定的栈帧 |
| up number | - | 向上移动指定个数个栈帧 |
| down number | - | 向下移动指定个数的栈帧 |
| info frame frame-number | i frame frame-number | 描述选中的栈帧 |

## B. 8. 打印变量命令

| 命令 | 缩写 | 说明 |
| --- | --- | --- |
| print var-name | p var-name | 打印指定变量的值。 |
| print file:var-name | p file:var-name | 打印指定文件中指定变量的值。 |
| print *array-name@length | p *array-name@length | 打印数组 array-name 中的前 length 项。 |
| print/type var-name | p/type var-name | 以 type 的格式打印变量的值。 |

type 的取值如下：

| Type | 说明 |
| --- | --- |
| x | 16 进制 |
| d | 有符号整数 |
| u | 无符号整数 |
| o | 8 进制 |
| c | 字符 |
| f | 浮点数 |
| t | 2 进制 |

## B.9. 其他打印命令

| 命令 | 缩写 | 说明 |
| --- | --- | --- |
| disassemble | disas | 对当前位置进行反汇编 |
| disassemble addr | disas addr | 对指定位置进行反汇编 |
| x/nfu address | - | 查看指定内存地址的数据，n 表示显示的内存单元个数，f 表示显示方式，u 表示一个地址单元的长度 |

显示方式：

| Type | 说明 |
| --- | --- |
| x | 16 进制 |
| d | 10 进制 |
| u | 10 进制无符号 |
| o | 8 进制 |
| t | 2 进制 |
| a | 16 进制 |
| i | 指令地址格式 |
| c | 字符格式 |
| f | 浮点格式 |

单元长度：

| Type | 说明 |
| --- | --- |
| u | 一个地址单元的长度 |
| b | 单字节 |
| h | 双字节 |
| w | 四字节 |
| g | 八字节 |

## C. 已知问题

1. 当 DRAM 地址 0x0 被分配到.bss 数据段时，0x0 地址上数据的“调试信息/数据类型”无法正确获取，该数据始终被认为是 32bit 数据类型。如图 75 所示 g_bss_sec 的 Type 和 Value。

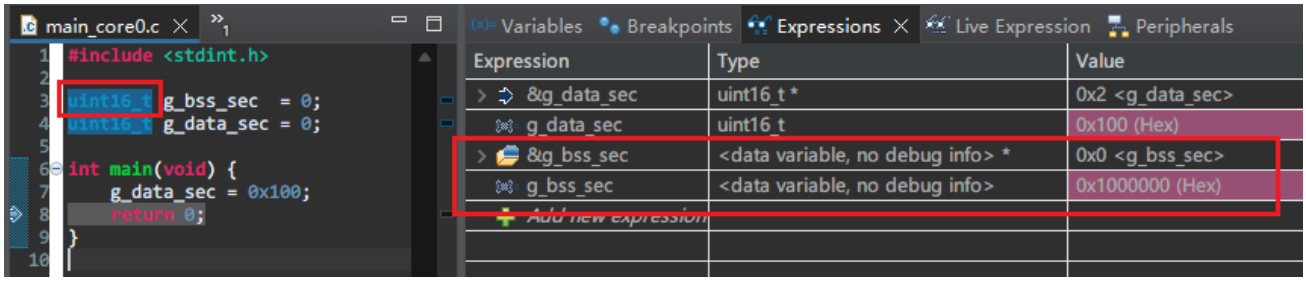



图 75：.bss 数据段地址为 0x0 的数据类型

**影响**：如果位于地址 0x0 的.bss 数据不是 32bit，调试时 IDE 无法正确显示该数据数值。

**解决办法**：默认情况下，链接脚本保留 0x0 处的 32-bit 区域

**历史的临时解决办法**：

1) 始终保持 0x0 的.bss 数据为 32bit。例如定义图 75 中的 g_bss_sec 为 32bit 类型，如图 76 所示。

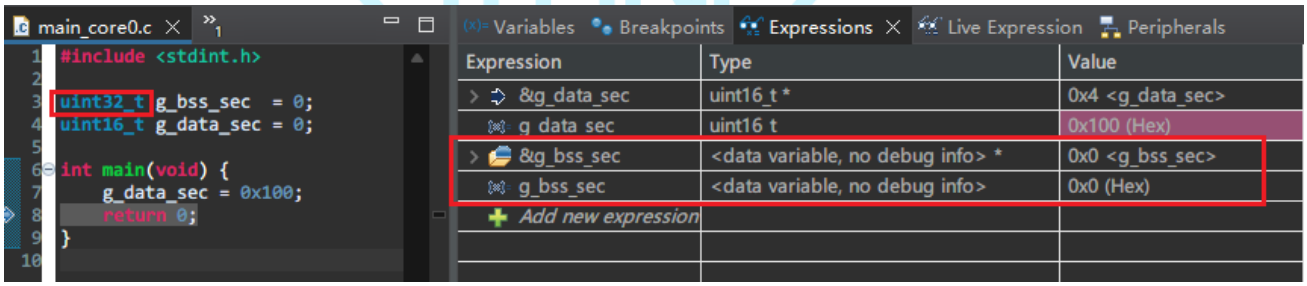



图 76：.bss 数据段地址为 0x0 的数据定义为 32bit 类型

2) 将地址 0x0 分配到.data 数据段，如图 77 所示。

（链接脚本优先将低地址分配给.data 数据段）

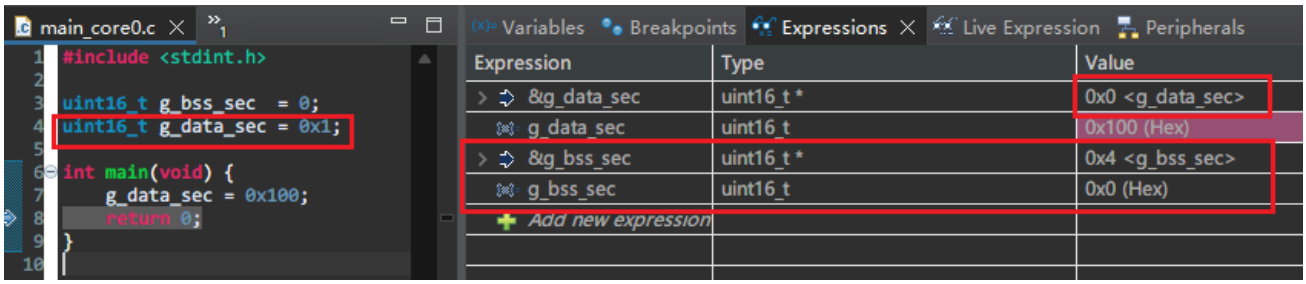



图 77：地址 0x0 分配给.data 数据段

2. （部分解决）Linux 系统环境下，无法正确读 Flash 存储器的数据。读 Flash 存储任意 32bit 对齐的地址返回的数值始终是 0x8ba1992d。

**影响：**

1) （已解决）Flash 烧写时，如果选中“Verify Download”，在校验 Flash 数据阶段会提示“数据不匹配下载失败”。但是实际上，数据已经正确写入 Flash 存储。
（QX 在 Linux 系统环境已做过 Flash 全地址写测试）

2) 通过 QX-IDE 的 Memory View 查看 Flash 地址范围的数据时，显示的数据始终是 0x8ba1992d。

其中 1）已经通过升级 Flash 烧写工具解决，Flash 烧写工具通过 DMA 方式读 Flash。