Application Note

CC27xx & CC23xx 实用软件 Debug 方法以及软件设计常见问题解答

Shuyang Zhong, Island Wei
Simon Huo, Conner Li

摘要

随着 CCS 20.0 的发布和逐渐完善，对 CC2340R5 & CC2745R10-Q1 的一些 debug 方法与之前有所不同，所以本文会以新版本 CCS 20.0 为开发平台来介绍一些实用 Debug 的操作方法。本片文章需要读者具有一定的软件 debug 能力以及基础的 IDE，SDK 以及硬件概念。

内容

1 软硬件设计参考链接

在接触到 TI SimpleLink 家族的时候，搭建环境是第一步，熟悉软硬件设计是第二步，那么我们将一些对用户有用的技术文档链接放在这里，可以在开发过程中参考。

1.1 数据手册和用户手册

芯片产品页面：TI 的数据手册和用户手册都会发布在官网的产品页面下，客户只需要找到产品页面就能下载一切公开的设计资料和应用手册。

- https://www.ti.com/product/CC2340R5
- https://www.ti.com/product/CC2340R5-Q1
- https://www.ti.com/product/CC2745R10-Q1
- https://www.ti.com/product/CC2745P10-Q1

开发板产品页面：开发板界面拥有开发板的 BOM 以及设计图。

- LP-CC2340R5
- LP-CC2745R10

仿真器产品页面

- LP-XDS110

1.2 环境搭建必要软件以及推荐软件

- 必要软件：

  综合开发环境 https://www.ti.com/tool/download/CCSTUDIO

  软件开发组件 https://www.ti.com/tool/download/SIMPLELINK-LOWPOWER-F3-SDK

  编译器组件 https://www.ti.com/tool/download/ARM-CGT-CLANG
- 建议软件：

  独立烧录工具 https://www.ti.com/tool/download/UNIFLASH

  独立图形化配置工具 https://www.ti.com/tool/download/SYSCONFIG

1.3 SDK 示例

- BLE 协议栈 API 介绍文档 API Reference
- 外围设备驱动 API 介绍文档 TI Driver API Reference

1.4 硬件设计参考文档

CC2745 开发板硬件设计文件

- RHA 封装 LAUNCHXL-CC2745R10-Q1 Design Files (Rev. B)

CC2340 开发板硬件设计文件

- RGE 封装 LP-EM-CC2340r5-RGE-4x4-IS24 Design Files (Rev. A)
- RKP 封装 LP-EM-CC2340R5 Design Files

硬件迁移

- 从 CC26xx 系列迁移到 CC23xx 和 CC27xx Hardware Migration to CC2340R5

天线设计

- CC234x and CC27xx Hardware Configuration and PCB Design Considerations

1.5 功耗计算表

https://www.ti.com/tool/BT-POWER-CALC

1.6 产品 QDID 认证链接

CC2340R5 https://www.ti.com/product/CC2340R5

CC2340R5-Q1 https://www.ti.com/product/CC2340R5-Q1

CC2745-Q1 https://qualification.bluetooth.com/ListingDetails/234268

1.7 HSM (CC2745) 认证信息

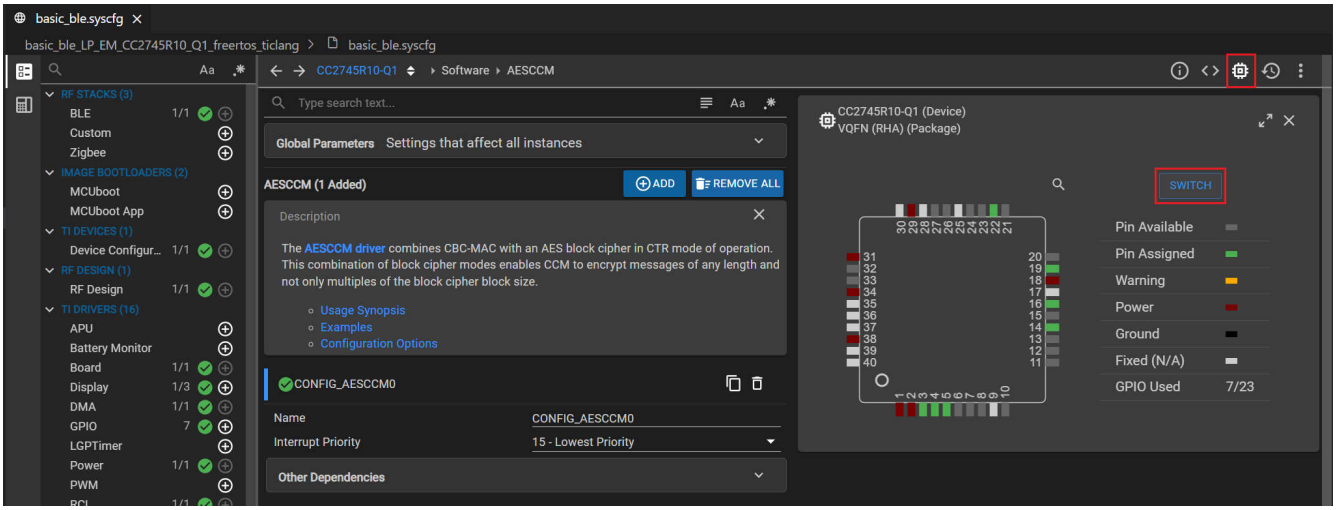
HSM 是 CC2745 内部独立的一个硬件安全模块，主要负责密钥加解密和密钥读写保护的操作。

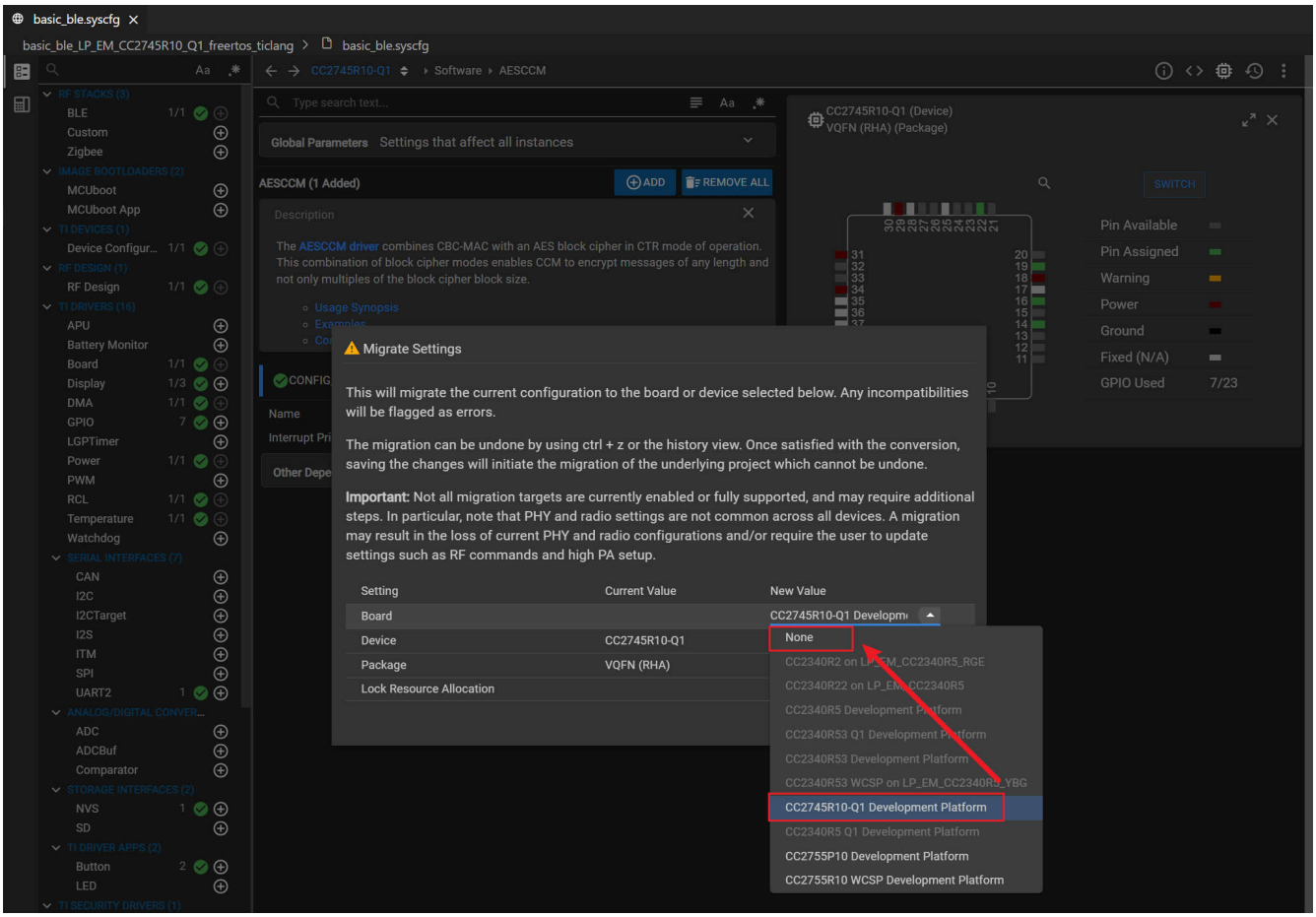
对于用户来说，比较希望指导 HSM 的安全等级，那么在这里举三个例子：SESIP，CRNG 和 NIST CAVP compliance。

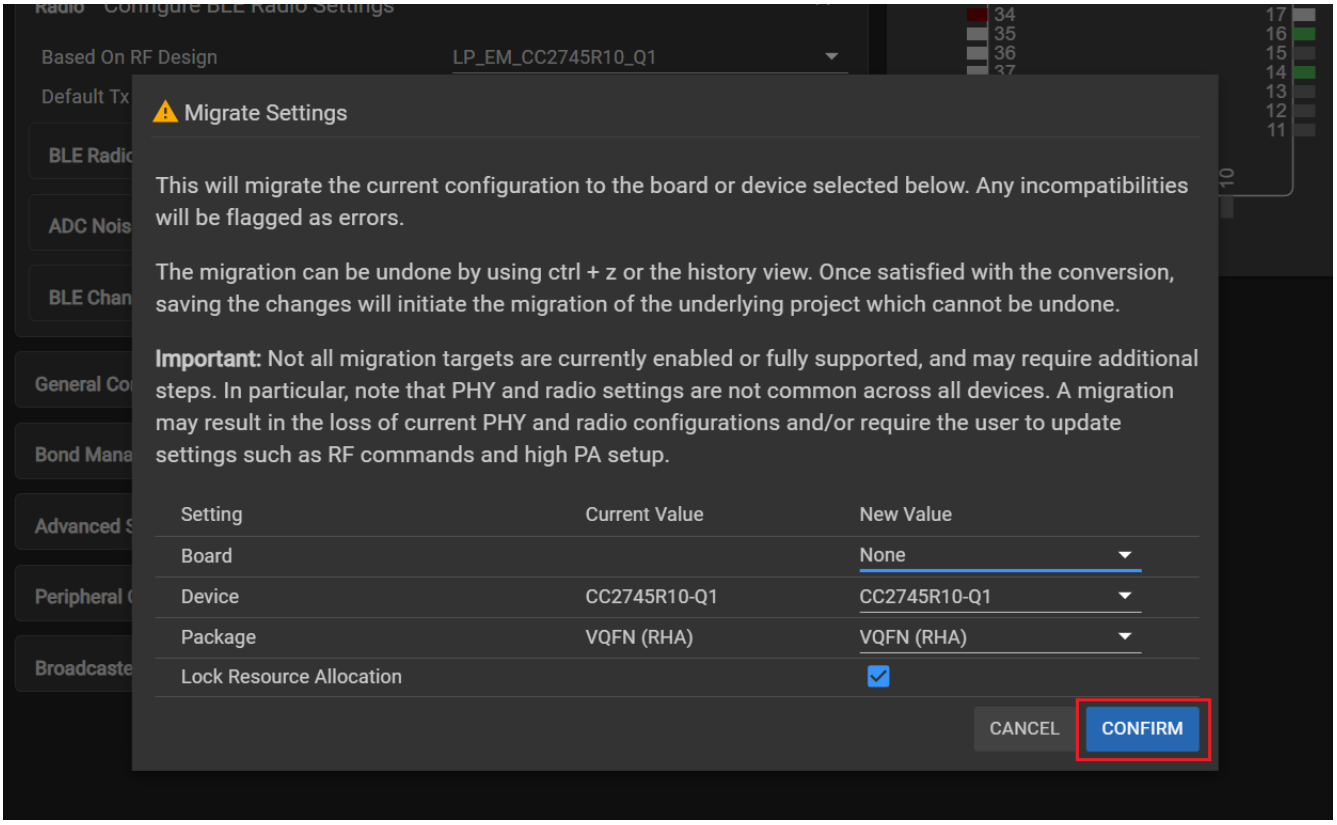
2 基于 CCS 20.0 的 Debug 方法

2.1 如何将工程文件映射为客户自己的板子

在客户的开发过程中，如果没有修改会遇到想要用的 PIN 口并不能被映射成相关功能，甚至会发生 IO 口冲突，这是很正常的现象，因为 TI SDK 提供的 Demo 都是基于开发板的硬件设计来映射相关接口。要解决这个问题，就应该使用 CCS 中内嵌的 Sysconfig 将 Board 配置为 NONE，设置好对应的芯片，然后点击 CONFIRM 即可完成转换。



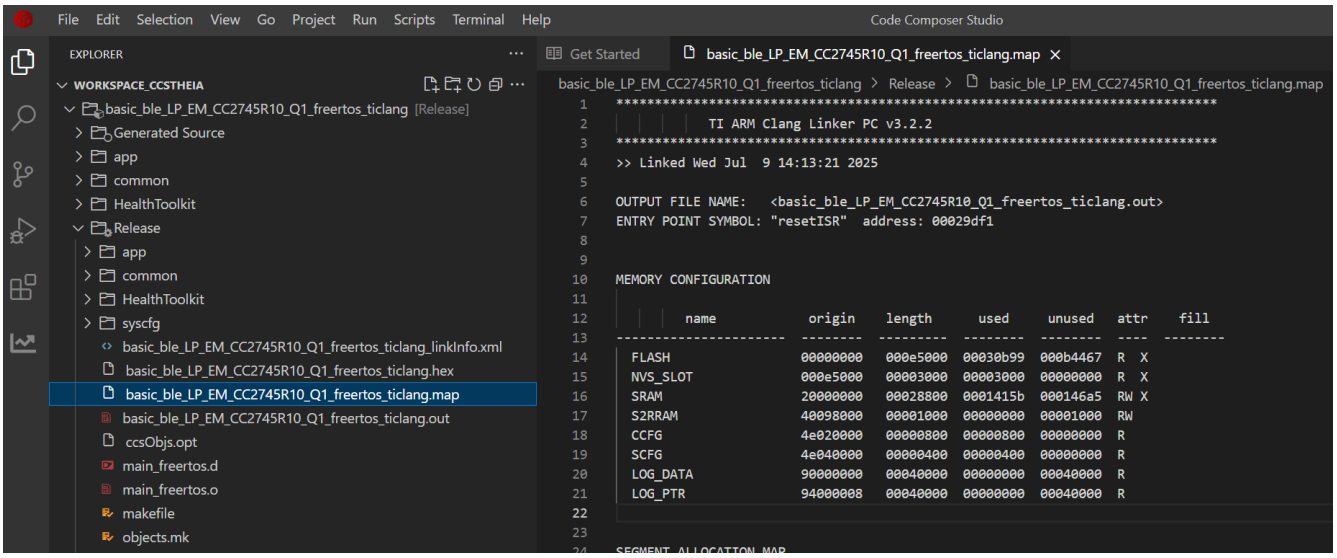




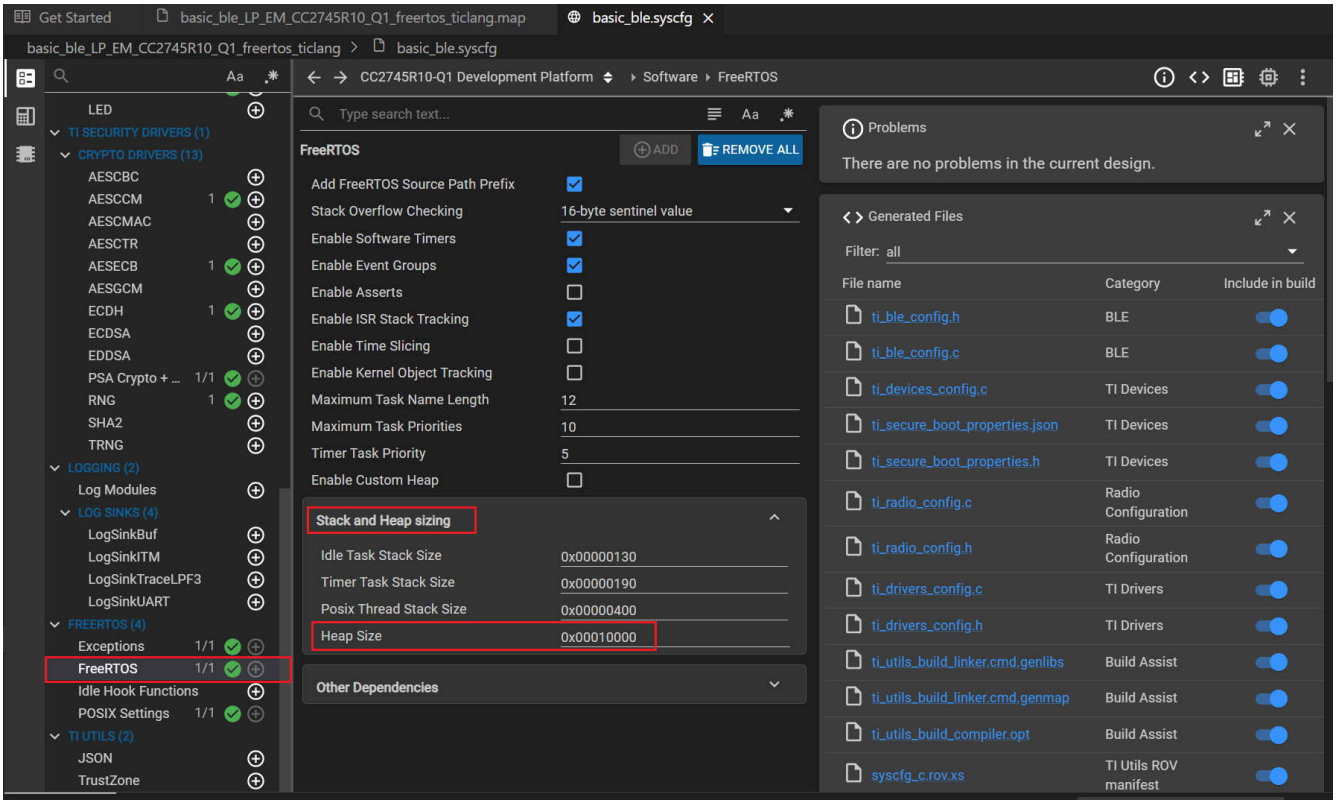
2.2 结合 map 文件和 ROV 评估 RAM 消耗和检查 Stack Overflow

在 BLE 的使用中，经常有如何确定我需要多少 Heap 分配给协议栈的问题，对于这种问题我们可以结合 .map 文件和 ROV 观测 Heap 用量来实现。

导入工程编译之后，你会看到在工程目录下的 Release/Debug 文件夹中找到对应工程名称的 .map 文件，此文件你可以看到芯片中 SRAM 和 Flash 以及 CCFG/SCFG 在芯片中的详细使用情况。



此处你看到的是该工程在当前配置下的全部用量和剩余量，此文件的生成对应工程目录下的 .cmd 文件。在 SimpleLink F3 SDK 中全面使用 Free-RTOS，在 Sysconfig 中有对 Free-RTOS 的 Heap 空间做分配，用户可以根据自身情况增大或者减小 Heap 的分配。如果您在 .map 文件中搜索 .bss 会发现 Free-RTOS 管理下的 Heap 是被分配在了.bss 段。

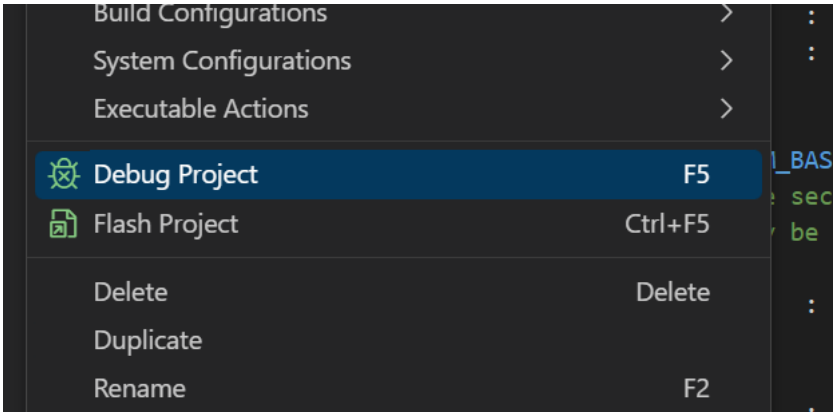


.bss 段在 .cmd 文件中指向了 RAM 空间。

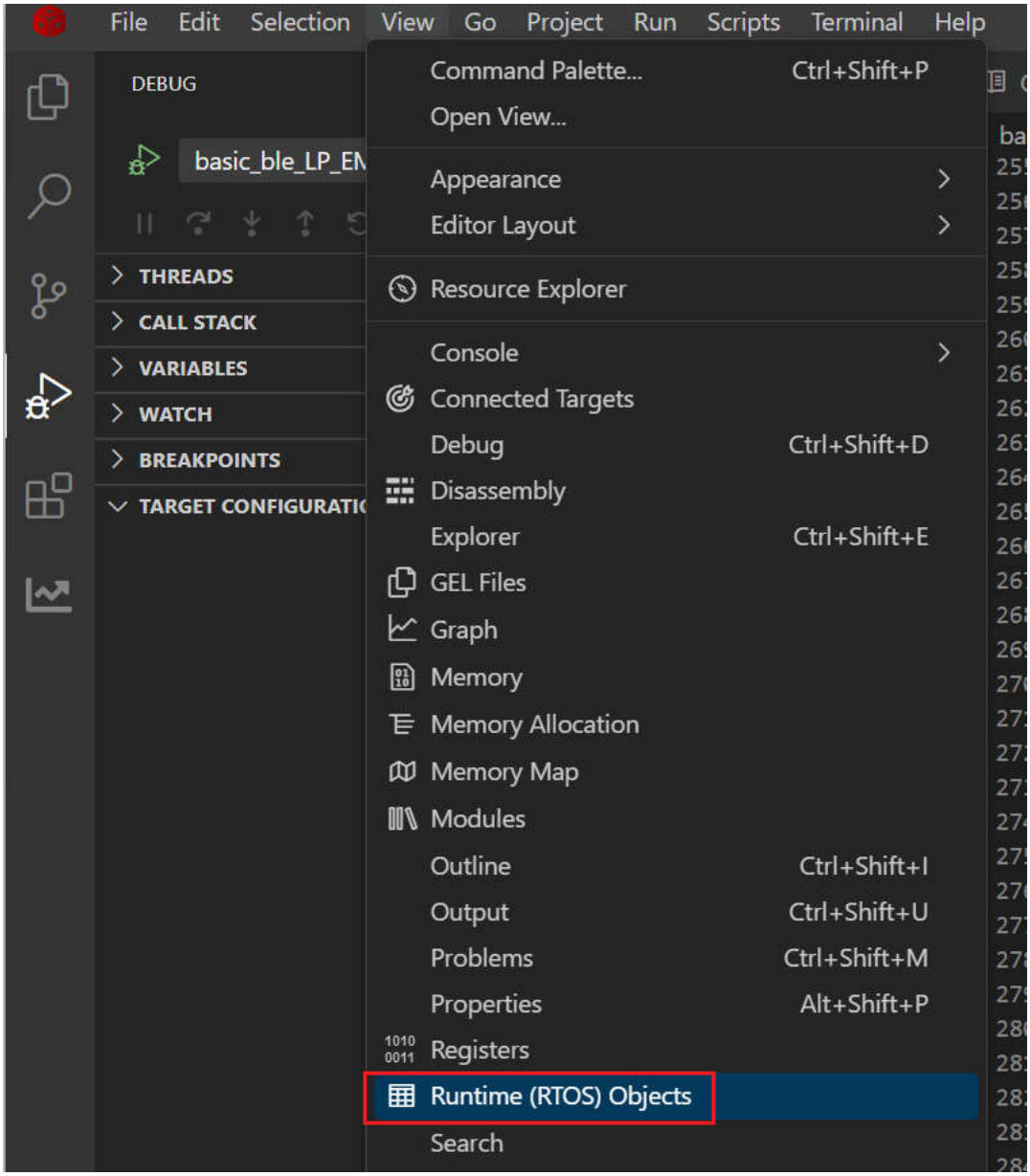
```
273  #if defined(RAM_BASE) && defined(RAM_SIZE)
274      .ramVecs        :   > SRAM, type = NOLOAD, ALIGN(256)
275      .data           :   > SRAM
276      .bss            :   > SRAM
277      .sysmem         :   > SRAM
278      .stack          :   > SRAM (HIGH)
279      .nonretenvar    :   > SRAM
280  #endif
```

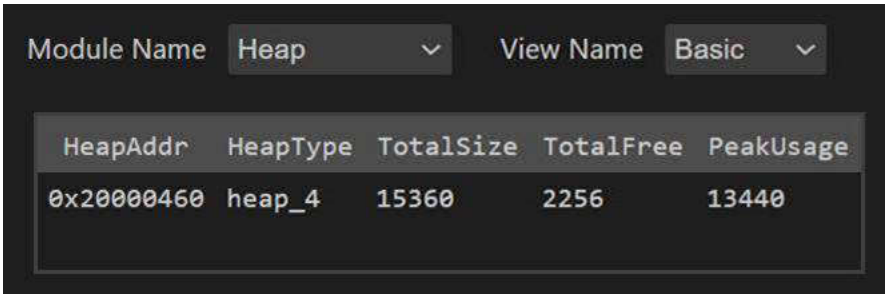
```
2642
2643  .bss        0    20000460    00012bd0     UNINITIALIZED
2644                     20000460    00010000       ti_freertos_config.o (.bss.ucHeap)
2645                     20010460    00000580       hsmddk_cc27xx_its.a : adapter_psa_preprovisioned
2646                     200109e0    00000210       (.common:gl_PSA_Key)
2647                     20010bf0    0000020c       secure_drivers_cc27xx.a : HSMLPF3.c.obj (.bss.op
2648                     20010dfc    00000200       hsmddk_cc27xx_its.a : tfm_internal_trusted_stora
```

我们在看过 .map 文件之后，想要知道在我们的应用要求下给协议栈分配多少 Heap 空间，我们以 BLE 作为 peripheral role 为例使用 ROV 监控可以同时维持 4 路连接的 Heap 的峰值消耗。打开 CCS 20.x，选中您想要测试的工程，右键该工程，并点击 Debug Project。



在正确连接并开始 Debug，点击工具栏中的 View -> Run Time Object，启动 ROV 功能，在 Module Name 中选中 Heap，右边使用 Basic 或者 Detailed 都可以，点击运行让代码正常运行，用 4 个对端设备依次连接仿真设备。在我的代码环境下，可以看到维持 4 路连接下，Heap 的峰值使用量为 13440 Bytes （13kB），那么我们就知道在这种情况下，给 Heap 分配不少于 13440 的大小，程序就可以正常运行。

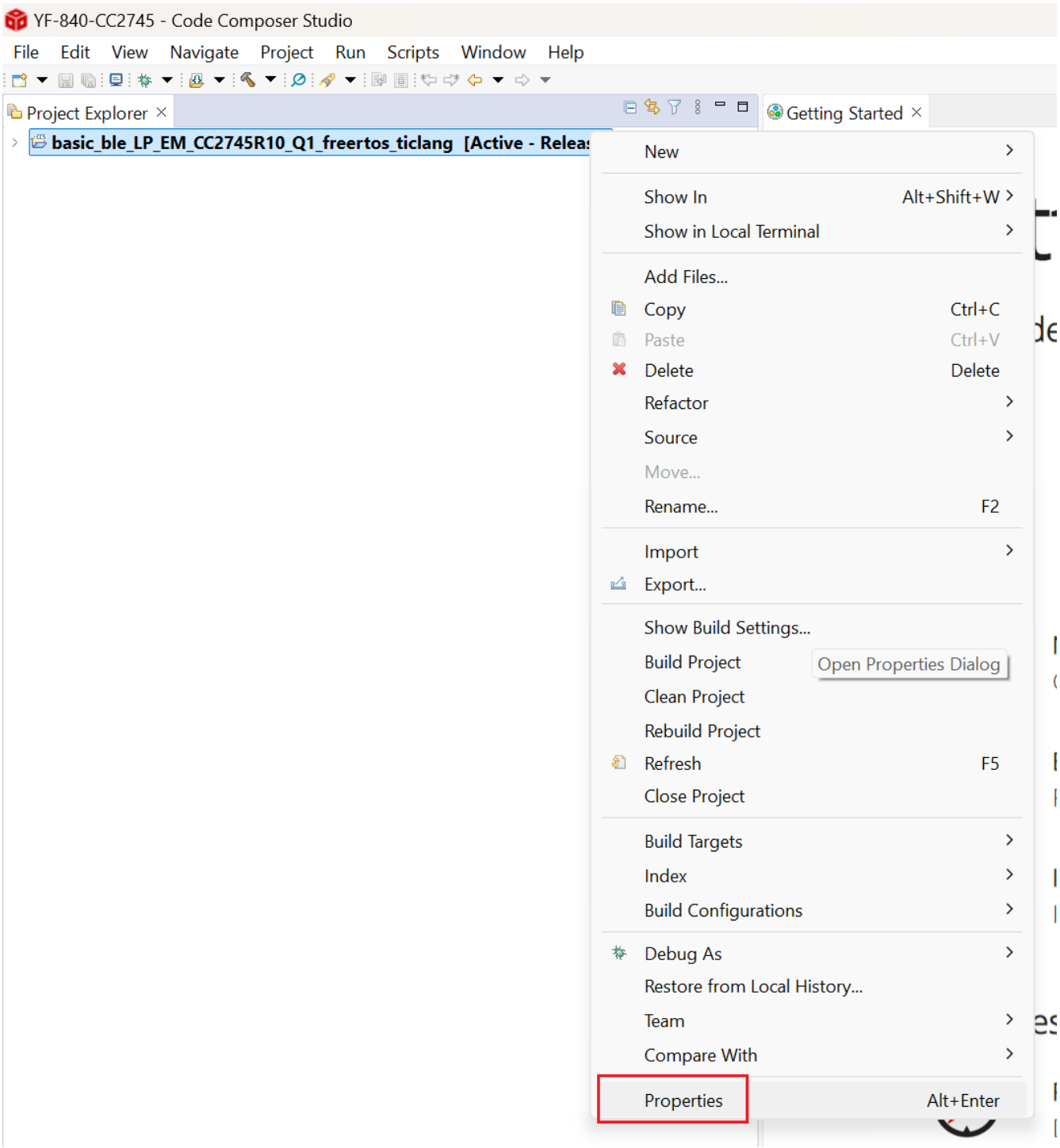


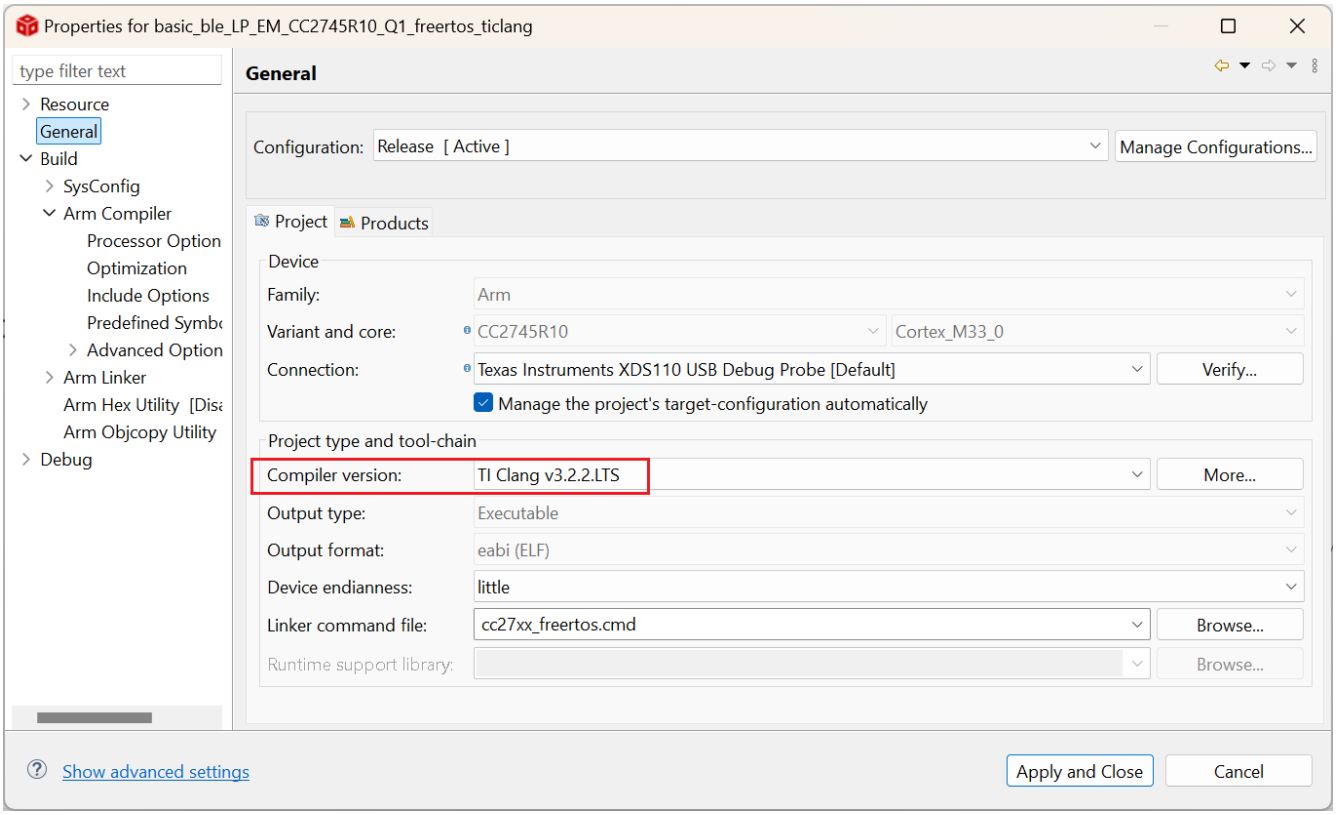


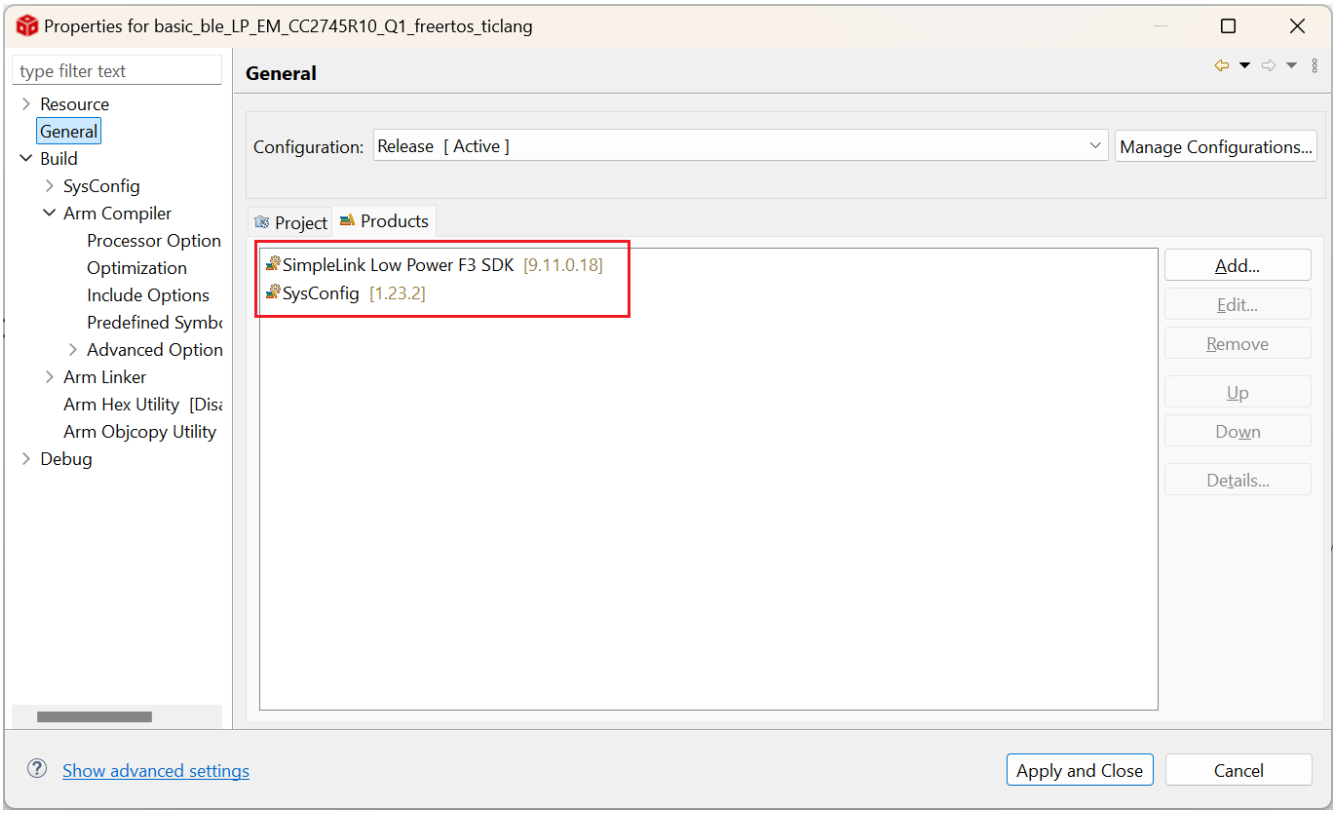
对于代码中可能出现的问题，我也同时建议使用 ROV 先检查 Stack 是否出现溢出现象，先检查这个对解决问题很有帮助。同样的使用方式，只是在 ROV 的 Module Name 需要选择 System Stack, 并同时检查 Tasks 的执行情况。如果 System Stack 显示 Stack Overflow 出现，那么久应该从 Tasks 中来观察了。

2.3 如何将你在 CCS 12.x 的工程移植到 CCS 20.x 中调试

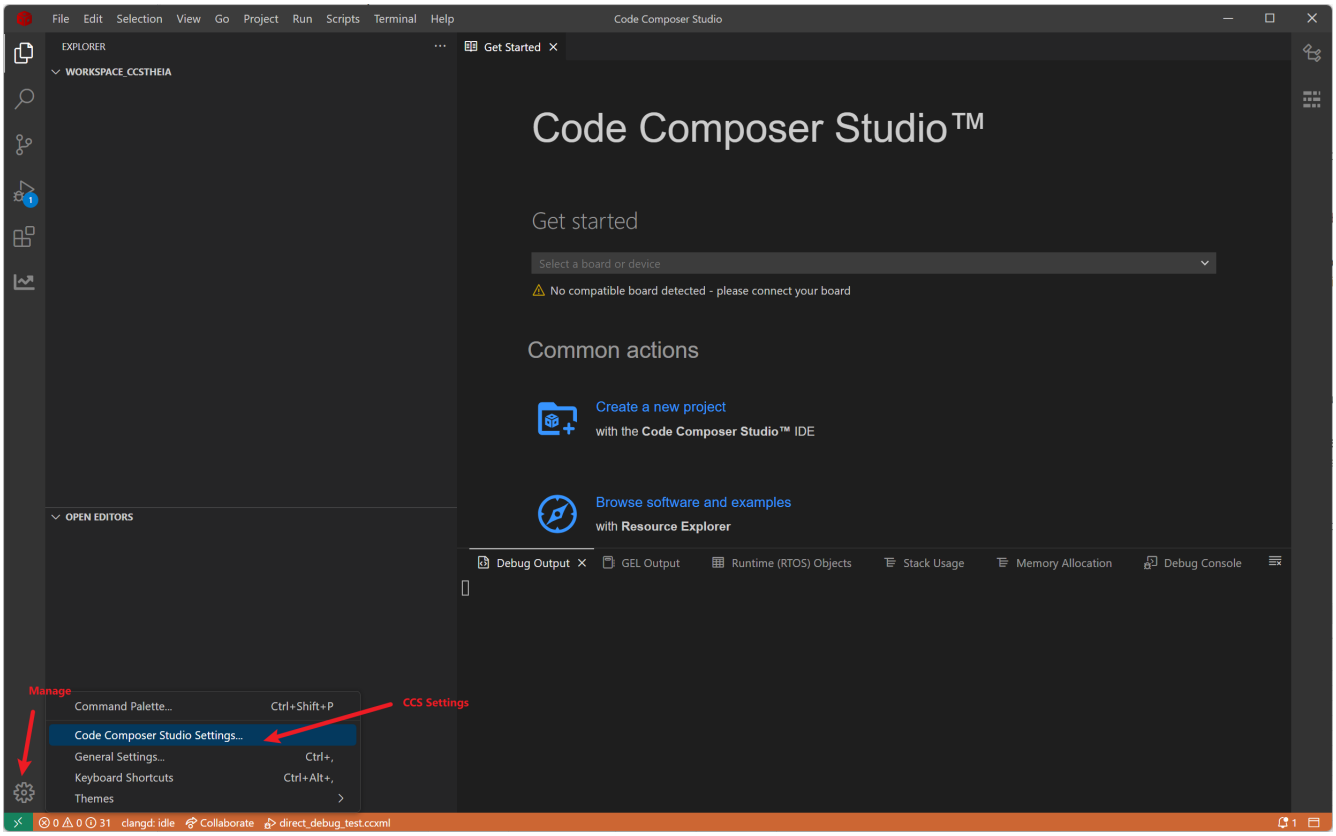
将工程转移需要先查看工程的依赖是否在两个版本的 CCS 中都能找到，那么首先我们要做的就是打开工程。

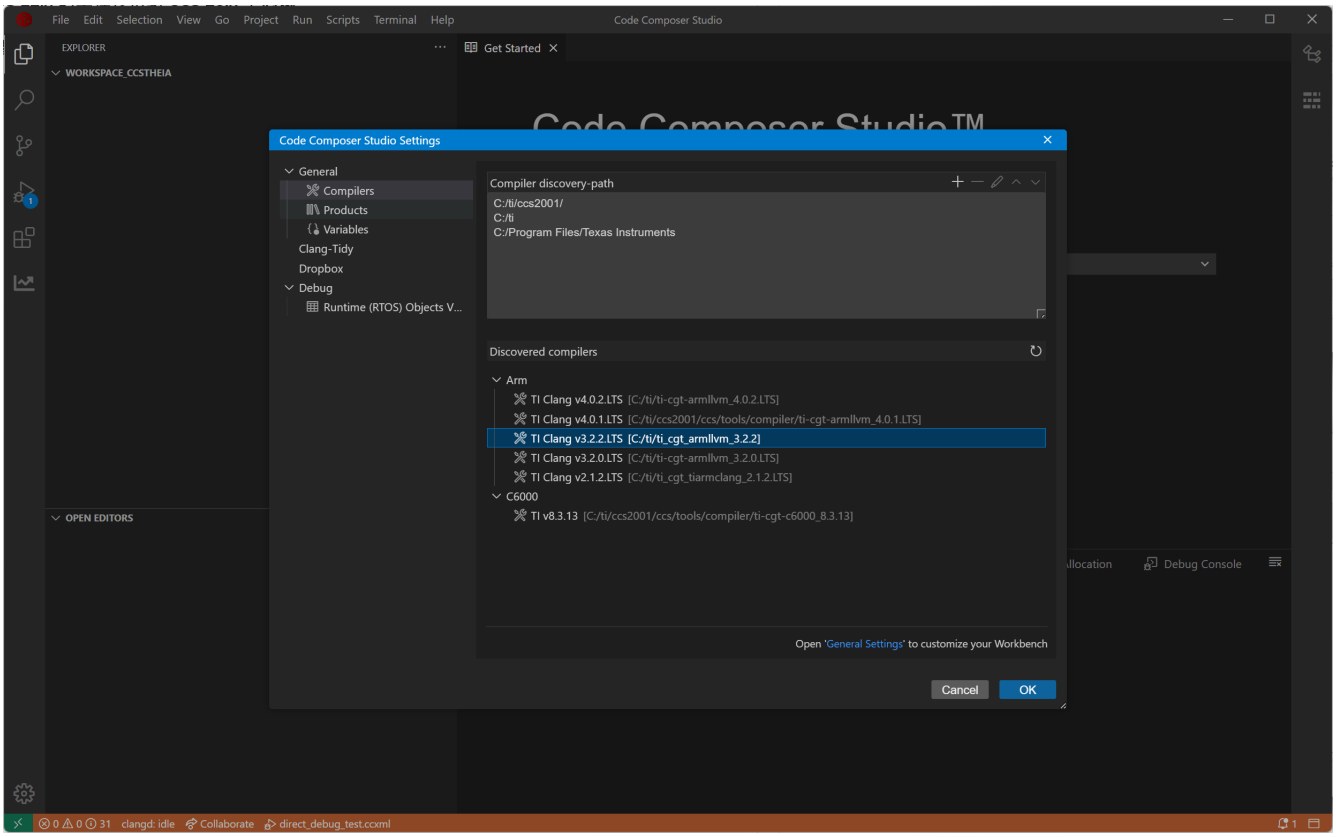


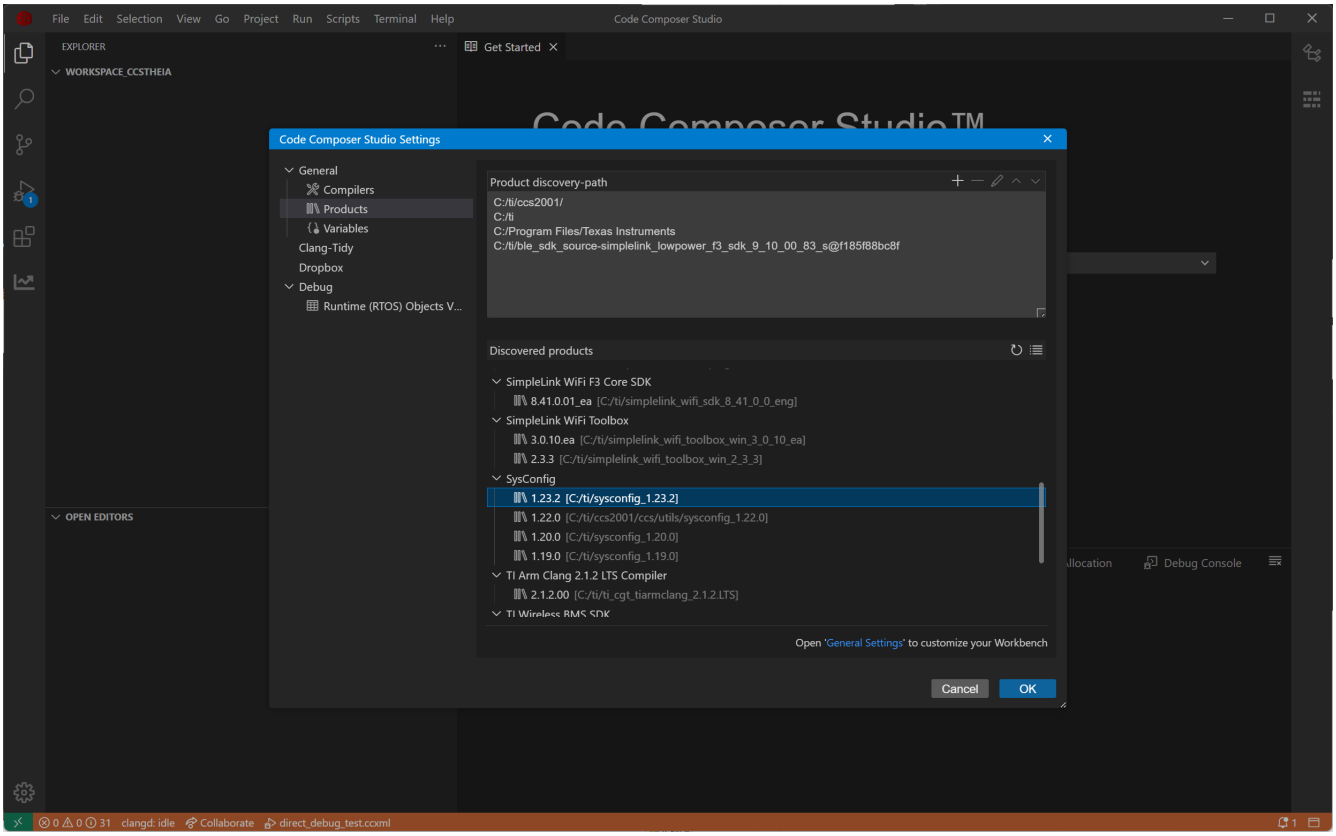




在 CCS 20.0 中在 Manage 和 Code Composer Studio Settings 中查看是否具有相同的依赖项。

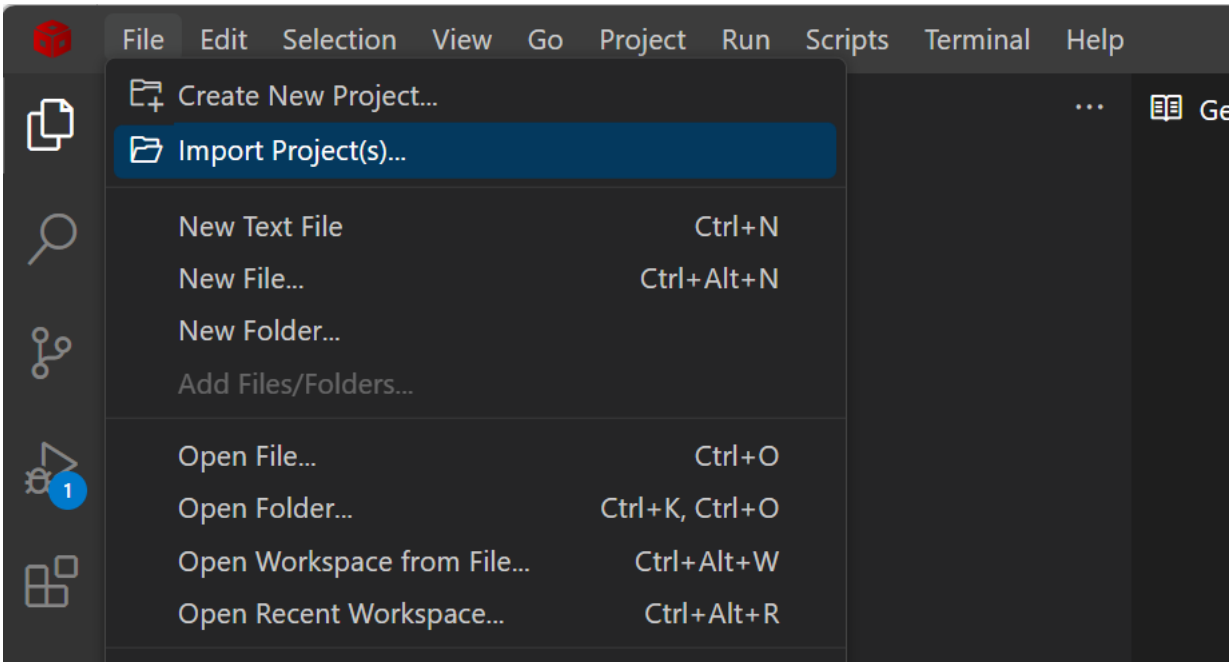




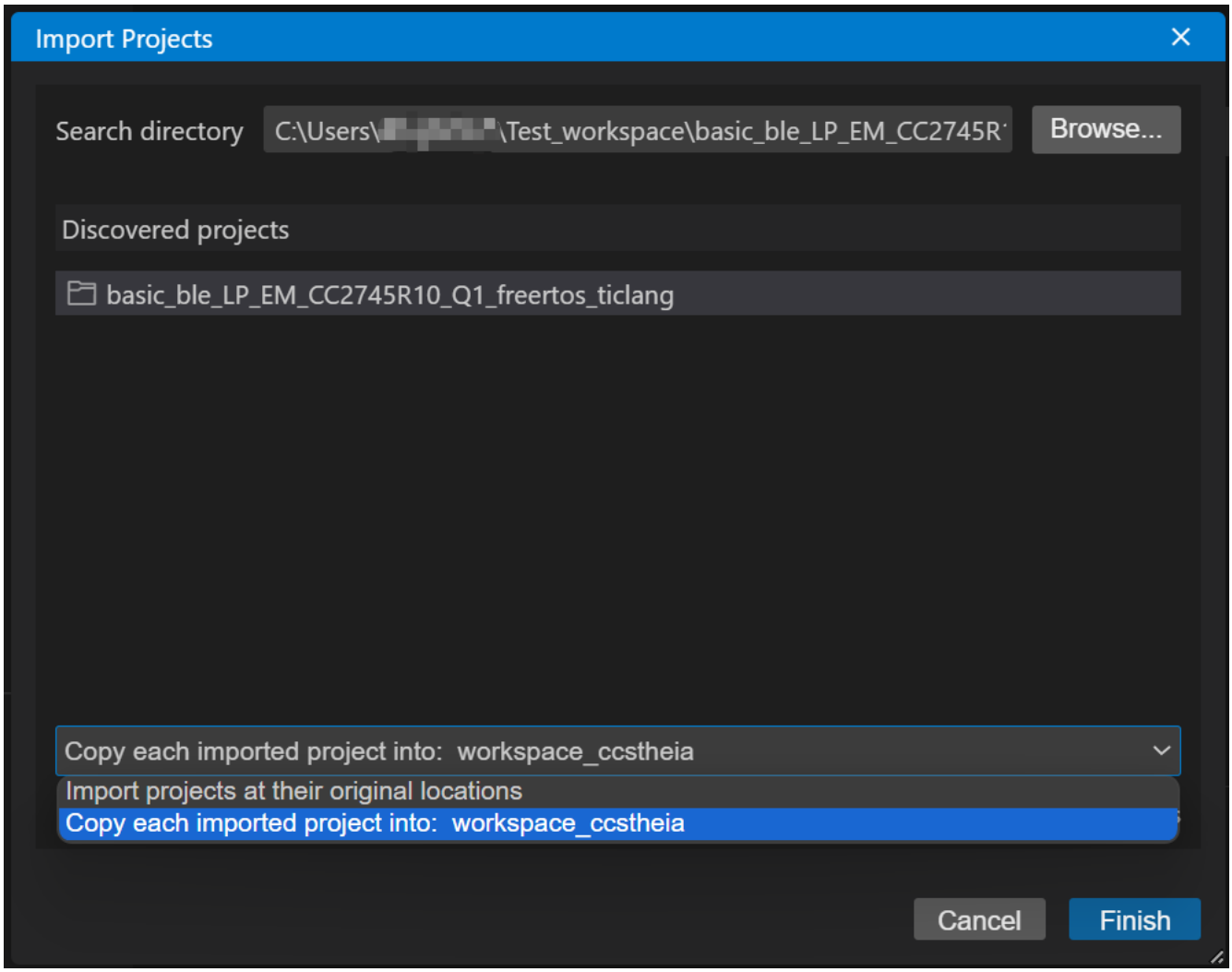


如果没有请进入官网安装，安装成功后点击 Discovered Products 右边的刷新案件。

如果依赖都已经具备，那么就可以手动将工程 copy 到 CCS 20.x 的工作空间。



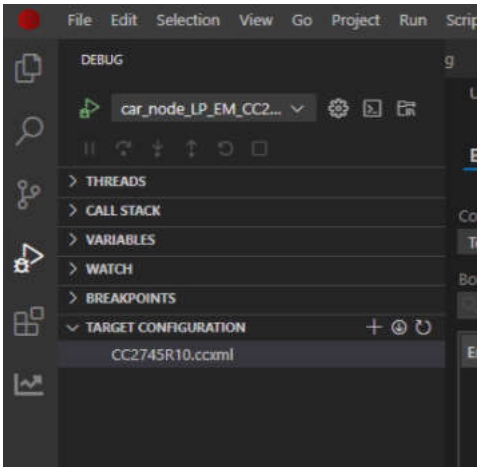
请选择 Copy each imported project int: xxx_workspace.



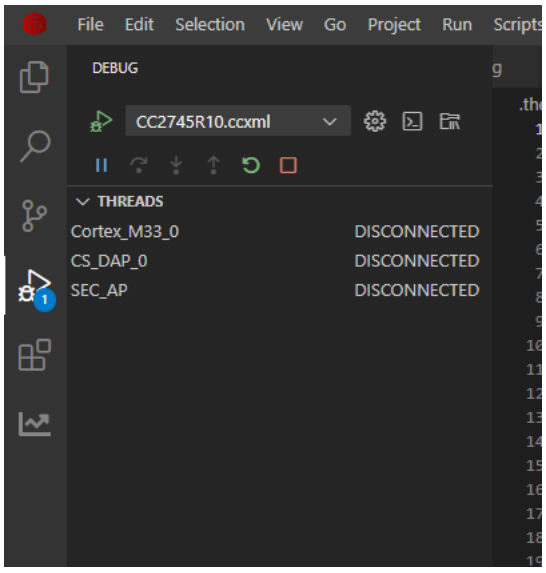
2.4 对 CC2745 进行 chip erase

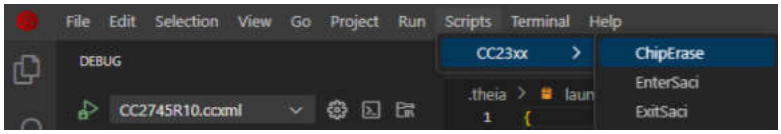
Uniflash 在当前去除了对 CC2745 进行 chip erase 的功能，对于一些需要确认 flash 遗留状态问题或需要删除包括配对记录在内的应用代码时，工程师常常感到困惑。对此我们可以在 CCS 中进行 chip erase。

在 CCS20.x 中打开 Debug→Target Configurations→右边第二个角标 Import Target Configuration→在 workspace 中选择任意 CC2745 工程的 targetConfigs 文件夹→CC2745R10.ccxml 并右键点击后选择 Start Project-less Debug



在 THREADS 窗口中右键单击 Cortex_M33_0 并选择 Show All Cores，之后选中 CS_DAP_0（**不要点击连接**）

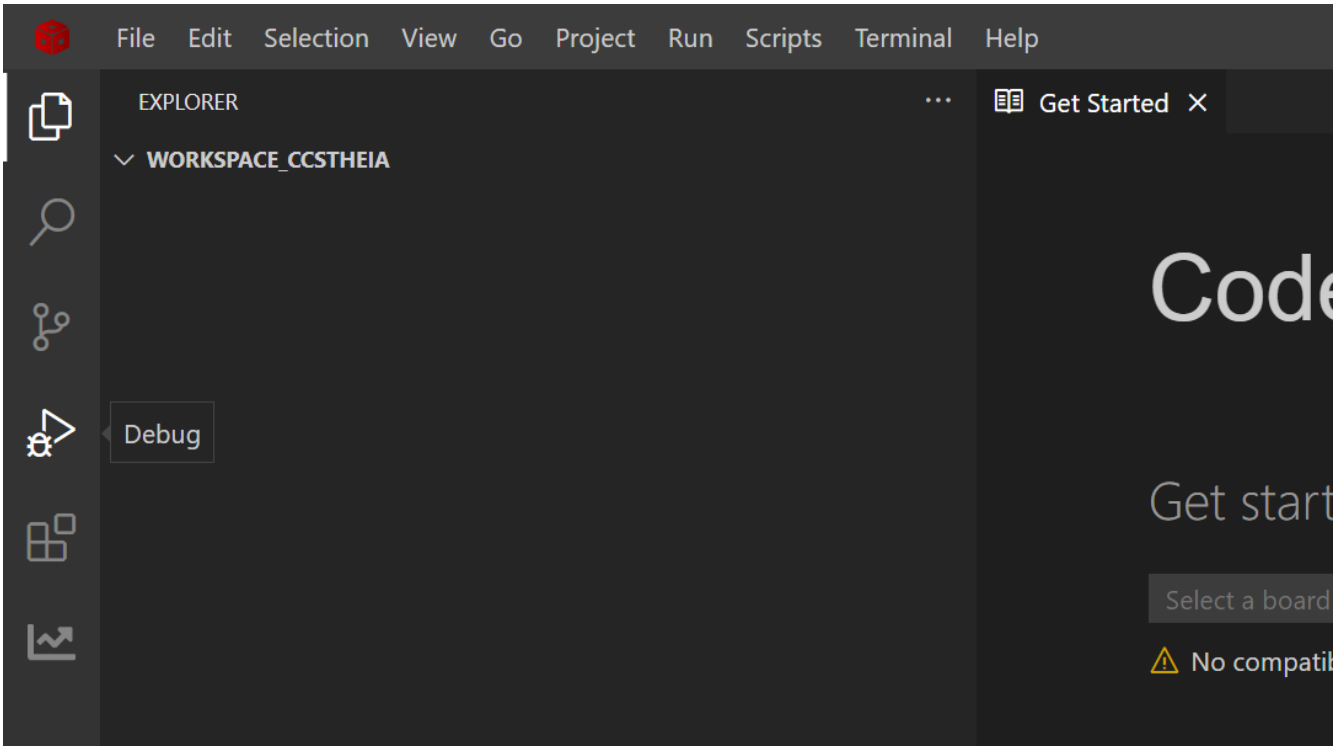




点击 Scripts→CC23xx→ Chip Erase。无需在意它是 CC23xx，它们都属于 TI 车规无线连接的统一系列芯片。点击 Scripts→ CC23xx→ Chip Erase。无需在意它是 CC23xx，它们都属于 TI 车规无线连接的统一系列芯片。

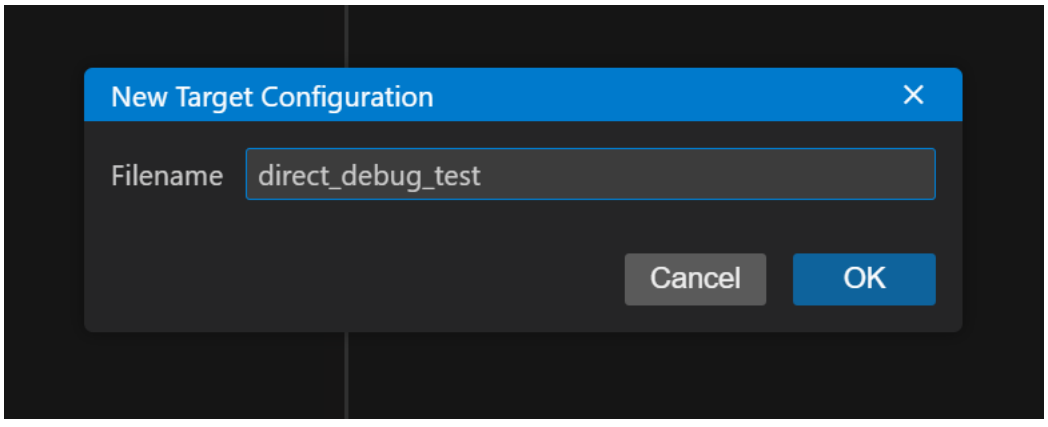
2.5 如何在不导入工程代码的情况下在 CCS 20.x 直接 debug 一个 项目工程（.out）

打开 CCS 20.x，点击左侧的 Debug 页面。

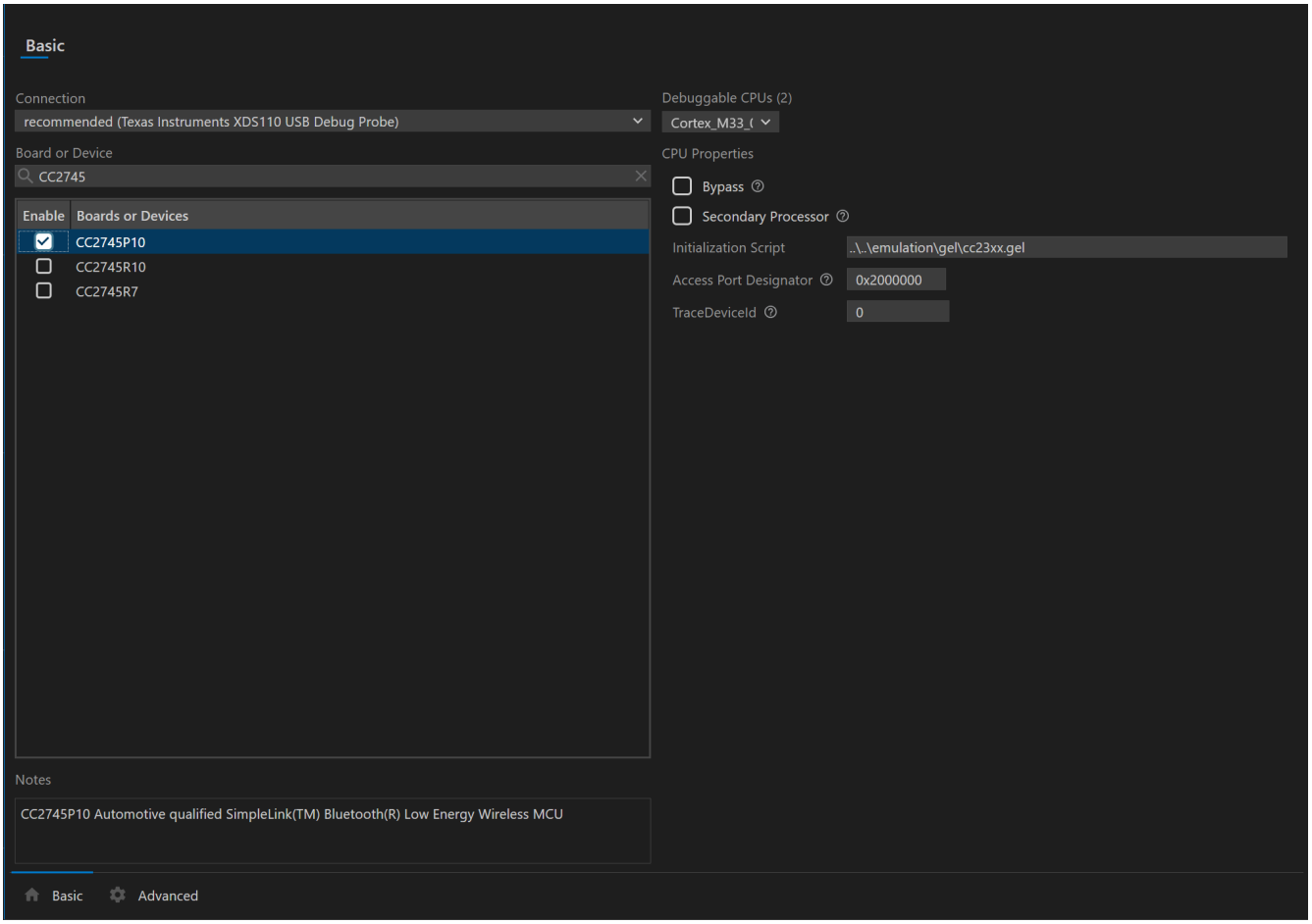


您将会看到 Debug 页面的详细情况，从上到下分别是 Thread / Call Stack / Variables / Watch / Breakpoints / Target Configuration. 点击 Target Configuration 右边的 + (New Target Configuration).

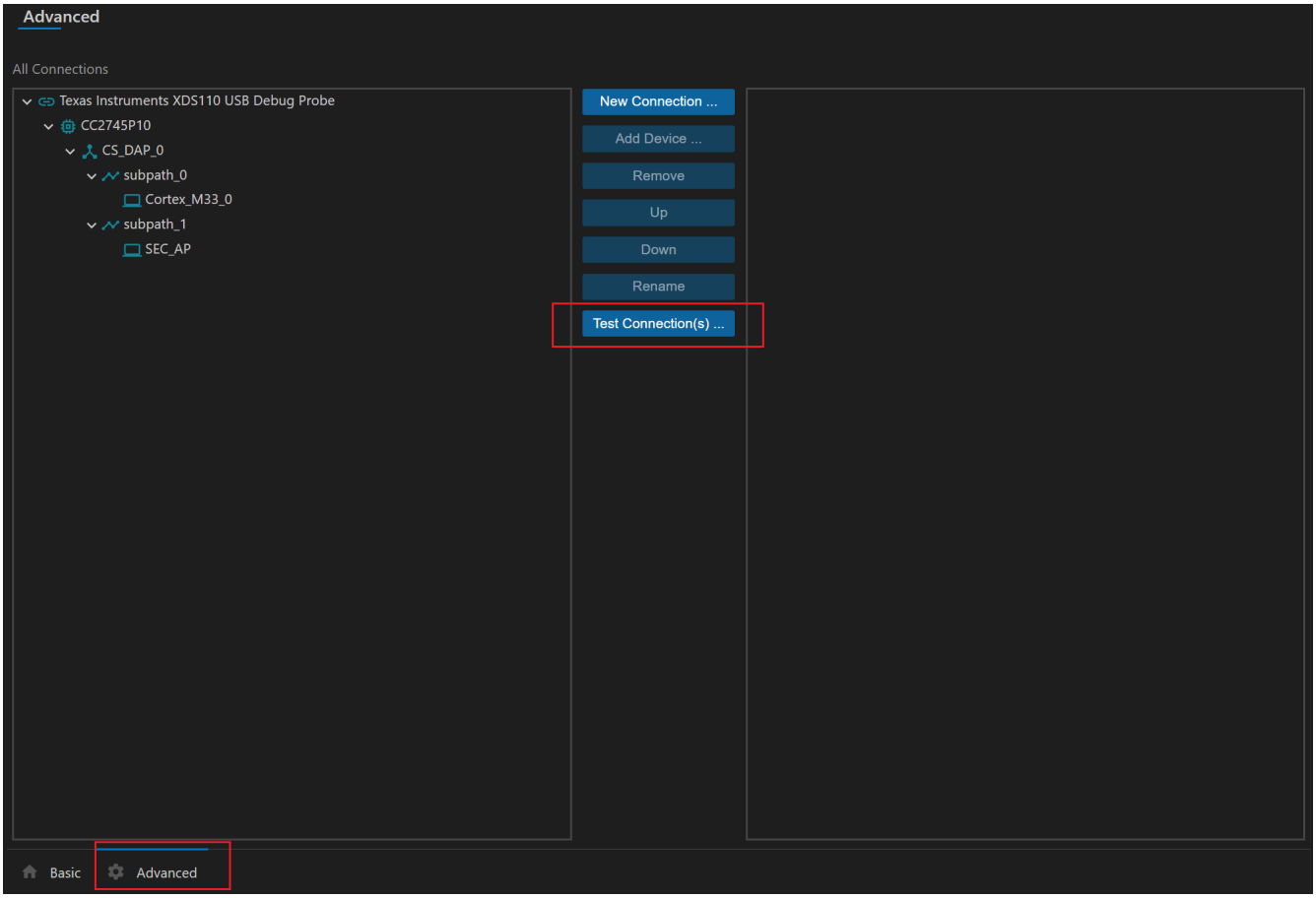
给您的配置文件（.ccxml）起一个名字，点击 OK。



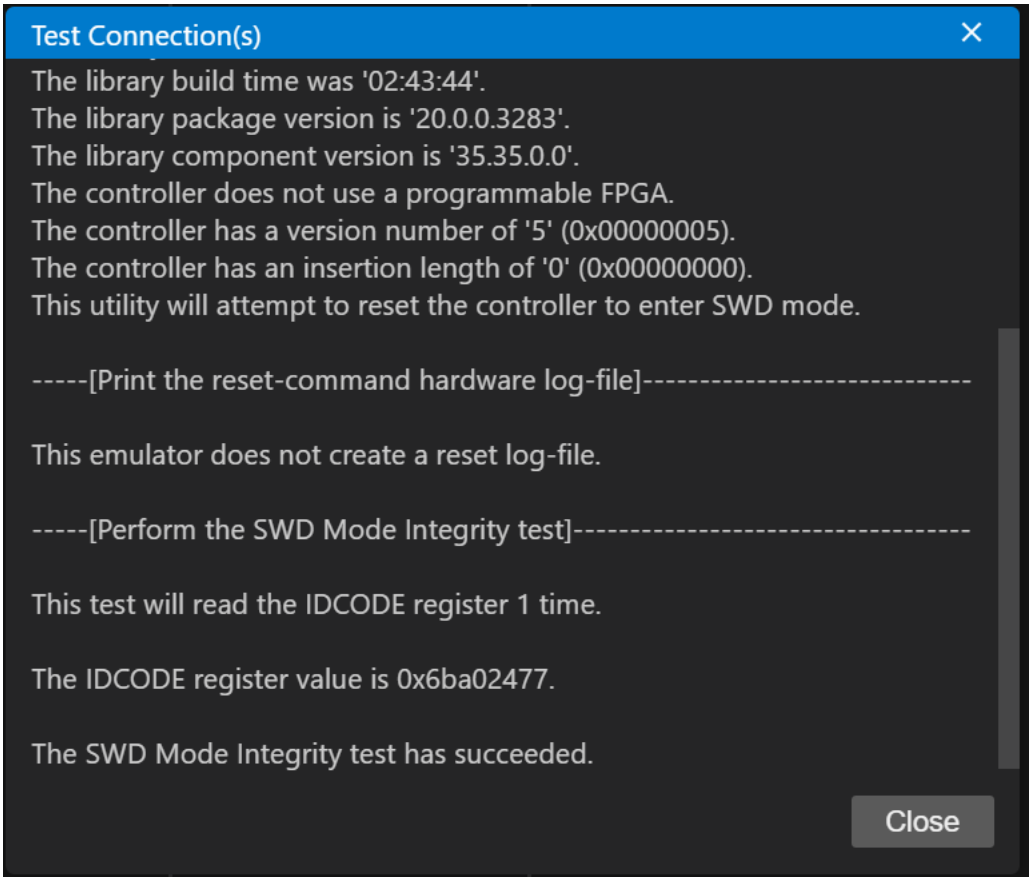
然后你可以看到一个 Basic 文件，我使用的仿真器是 TI 的 XDS110， 所以在 Connection 里可以选择 Texas Instruments XDS110 USB Debug Probe，根据您使用的芯片或板子系列名称，在 Board or Device 中搜索 CC2340 或者 CC2745，此例选择 CC2745R10。



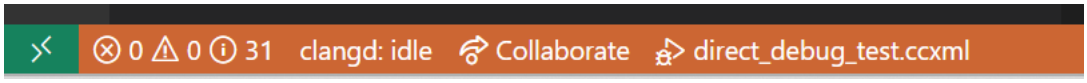
此时可以使用 XDS110 连接您的板子到电脑，为了测试您的配置是否正确，可以点击下方 Advanced，点击 Test Connection 测试是否可以和板卡通信。

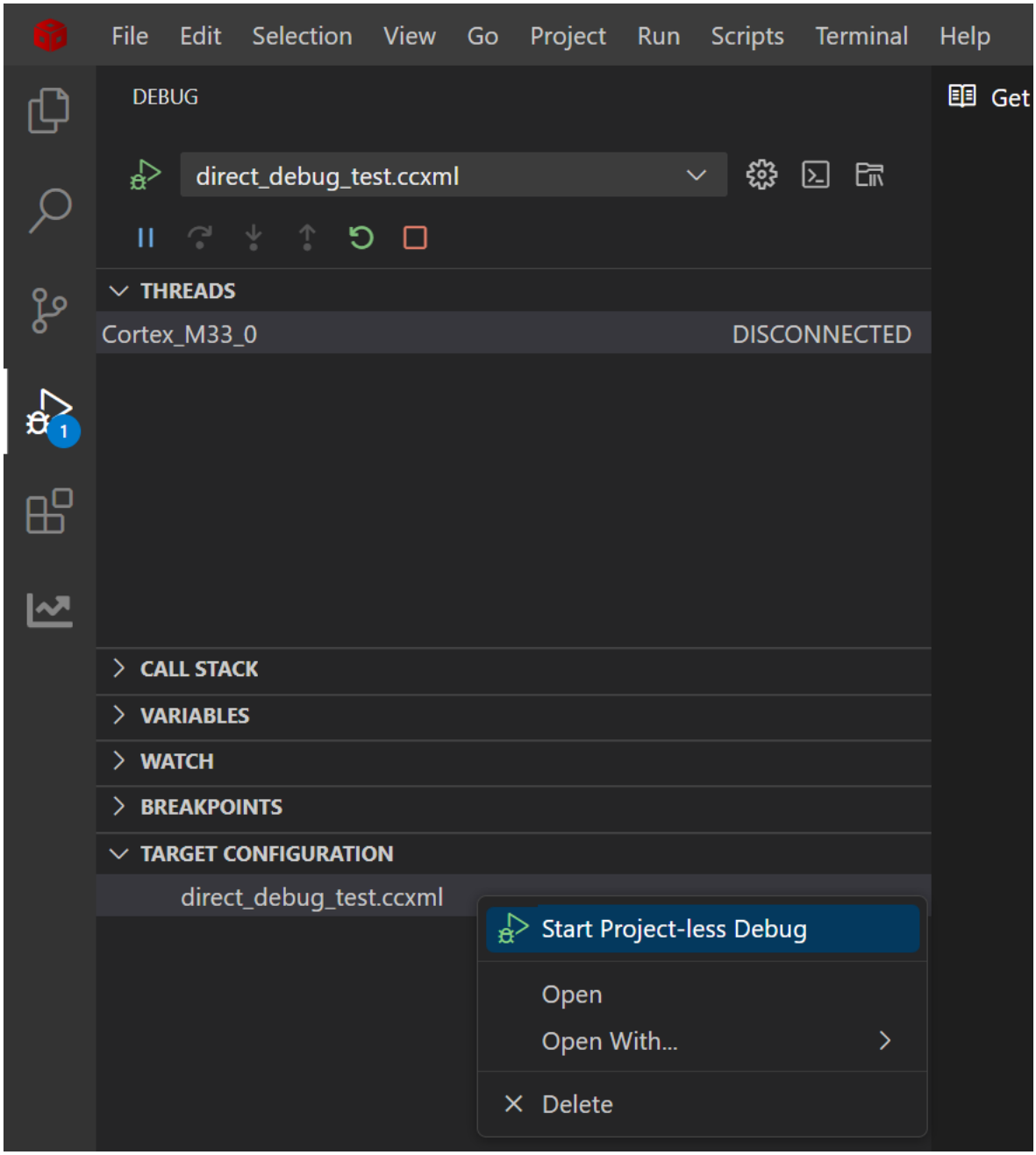


如果有以下日志和界面展示，那么证明可以正常连接到板卡。

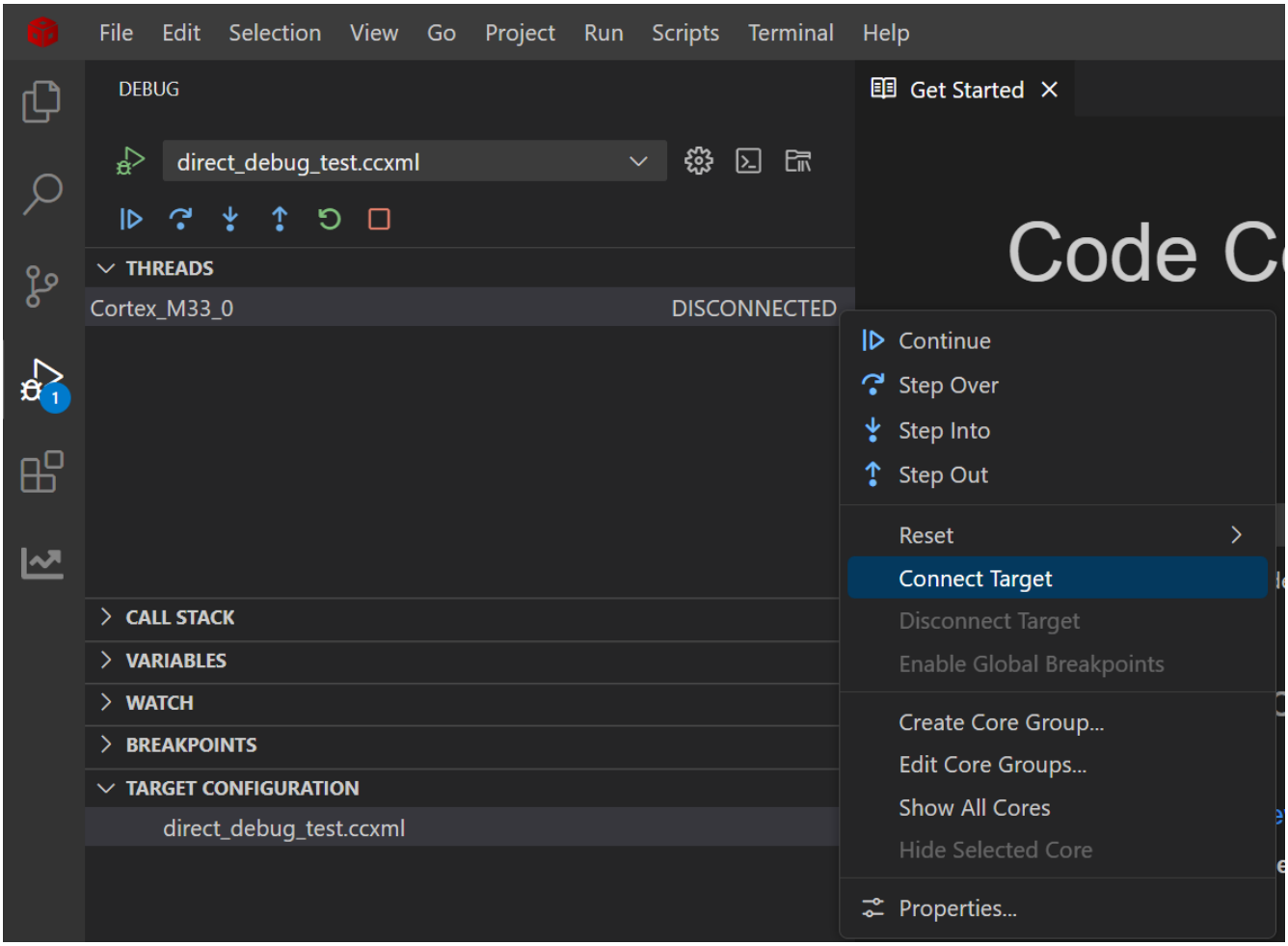


接下来鼠标右键点击刚刚创建好的.ccxml 文件，点击 Start Project-less Debug，便可以看到 CCS 界面下方有已经 launch 了对应配置，并且状态是 idle。

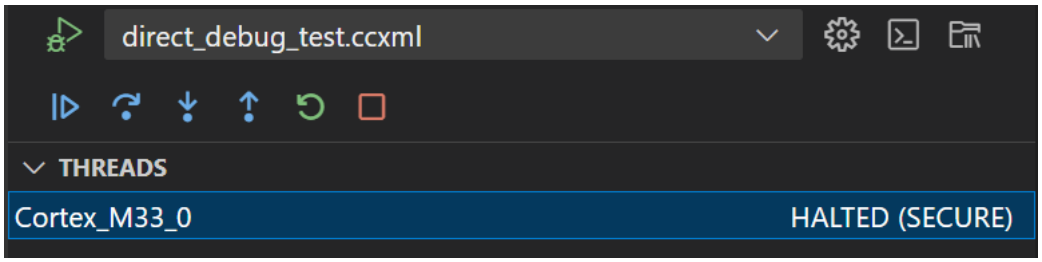




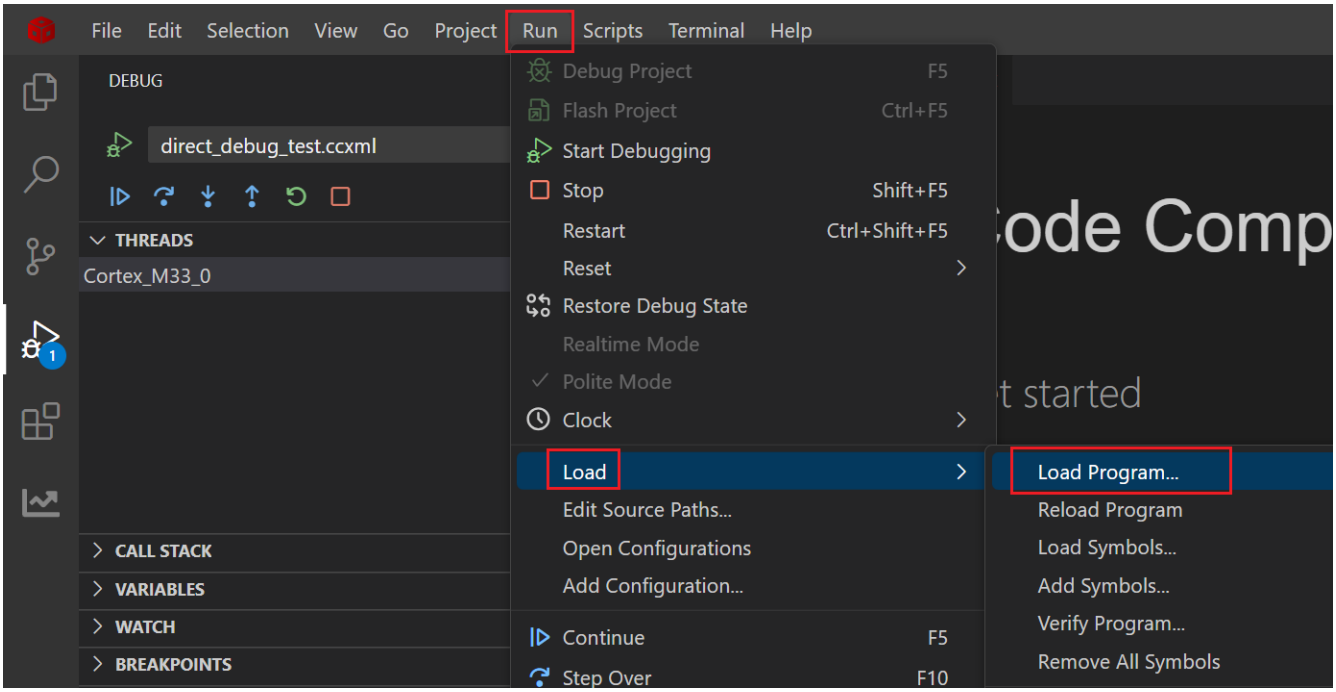
此时你会看到 Cortex_M33_0 是 Disconnected 状态，鼠标右键点击 Cortex_M33_0 并点击 Connect Target。



可以看到连接装填改变为 HATLTED（SECURE）。

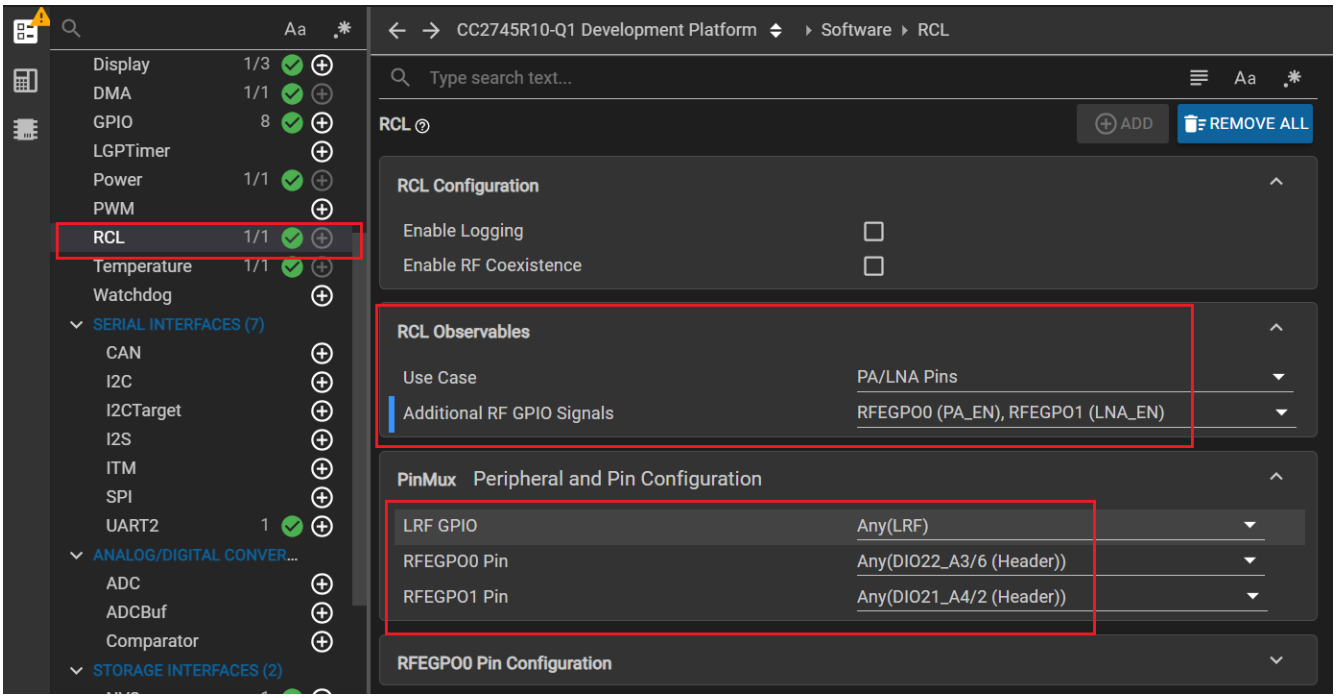


然后点击 Run →Load → Load Program⋯ 选择你想要 debug 的 .out 文件，加载即可。

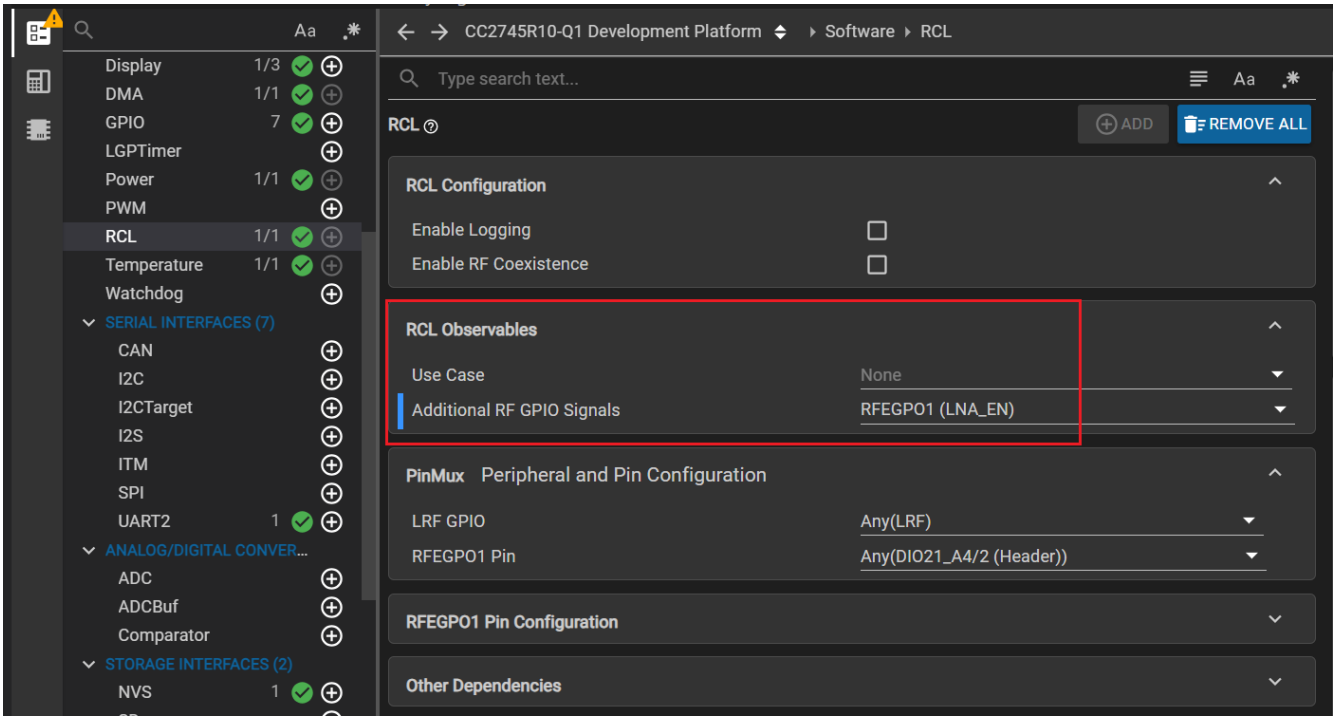


2.6 如何观察芯片的射频是否正常工作

CC23xx 和 CC27xx 系列芯片都支持将芯片内部射频信号映射到 PIN 脚来观察射频的工作状态，有连个信号可供观察，也是最多使用的两个信号：LNA 和 PA，LNA 对应接收机（Rx），PA 对应发射机（Tx）。使能方法如下：



可以看到已经将 PA_EN 和 LNA_EN 信号映射到了 DIO22 和 DIO21。另外，如果在您的板子上没有多余 IO 可供使用，或者只想看到一个信号就足够确定问题，那么可以将 Use Case 中的 PA/LNA Pins 取消勾选，然后直接引出对应的信号并选择映射的 PIN 脚，即可链接逻辑分析仪进行观察。



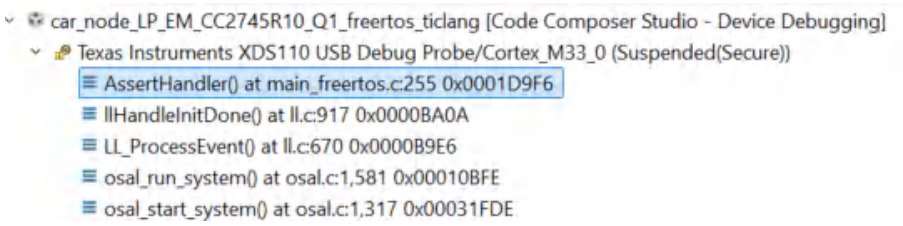
需要注意的是，只有 LRF Digital Output 的 PIN 脚支持被使用，所以在刚开始设计自定义板卡的时候应该考虑到射频信号 debug 的功能，对这些 PIN 脚进行预留或复用。

LRF Digital Output	LRFD0	19	I/O	O	LRF digital output 0
	LRFD1	20			LRF digital output 1
	LRFD2	21			LRF digital output 2
	LRFD3	22			LRF digital output 3
	LRFD4	23			LRF digital output 4
	LRFD5	24			LRF digital output 5
	LRFD6	32			LRF digital output 6
	LRFD7	33			LRF digital output 7

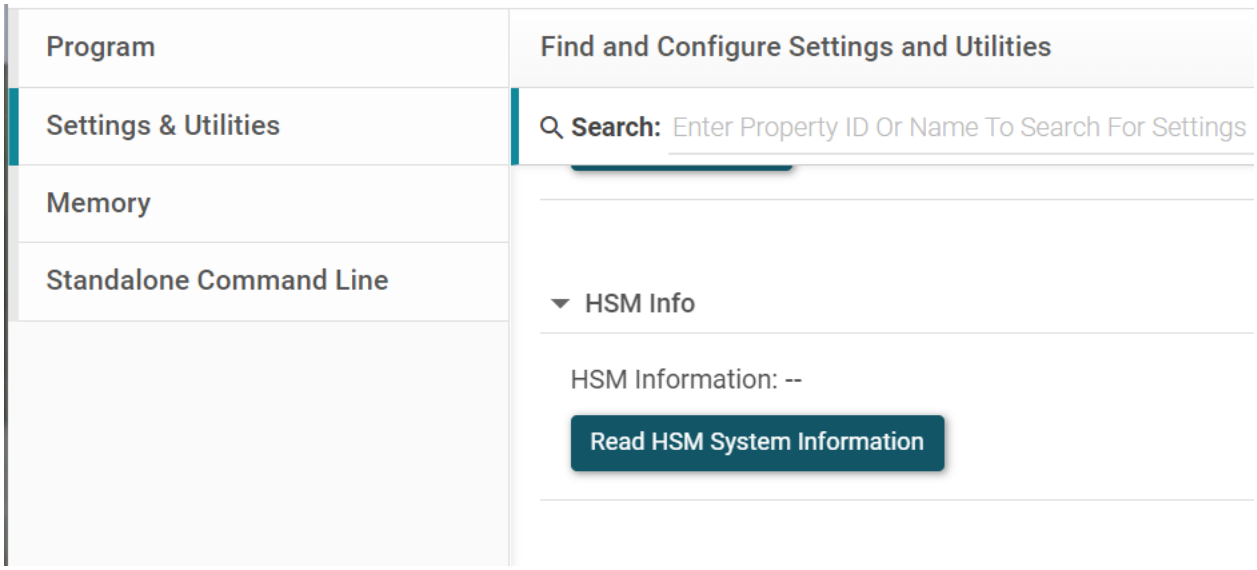
3 CC2340 & CC2745 开发过程中的常见问题

3.1 UniFlash 或 CCS 都烧写程序成功，但无法正常运行

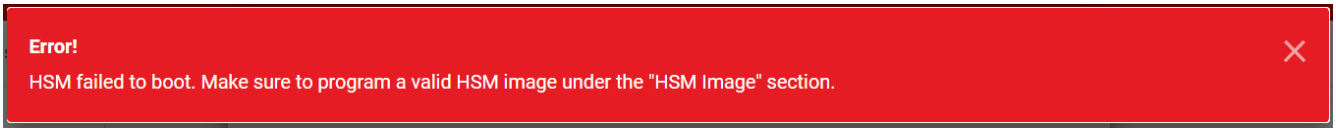
在初次使用 CC2745 时，尤其容易发现的一种问题，使用 TI SDK 所带的官方例程进行 debug 或直接烧录一个预编译的 hex 文件，但程序无法在芯片上正常运行。如，使用 CCS 12.8.1 编译 basic_ble_LP_EM_CC2745R10_Q1_freeertos_ticlang 工程，会报 assert 如下



其原因是 cc2745 空板是需要先烧录 hsm firmware 才能正常初始化的，否则芯片无法正常启动。建议工程师在烧录芯片之前，进行 hsm firmware 的检查。打开 Uniflash→Settings & Utilities→Read HSM System Information



当出现缺失 HSM 固件时，会出现报错信息



解决方法是在 SDK 中找到 HSM firmware 的路径*<SDK path>\bin\hsm\cc27xxx10_hsm_fw_v3.0.0.bin，并输入 Uniflash->Settings& Utilities->HSMImage* 中并点击 Program HSM Image 烧录 HSM firmware

3.2 蓝牙设备的 Tx Power 更改

对于 Tx Power 的更改方法有以下两种：

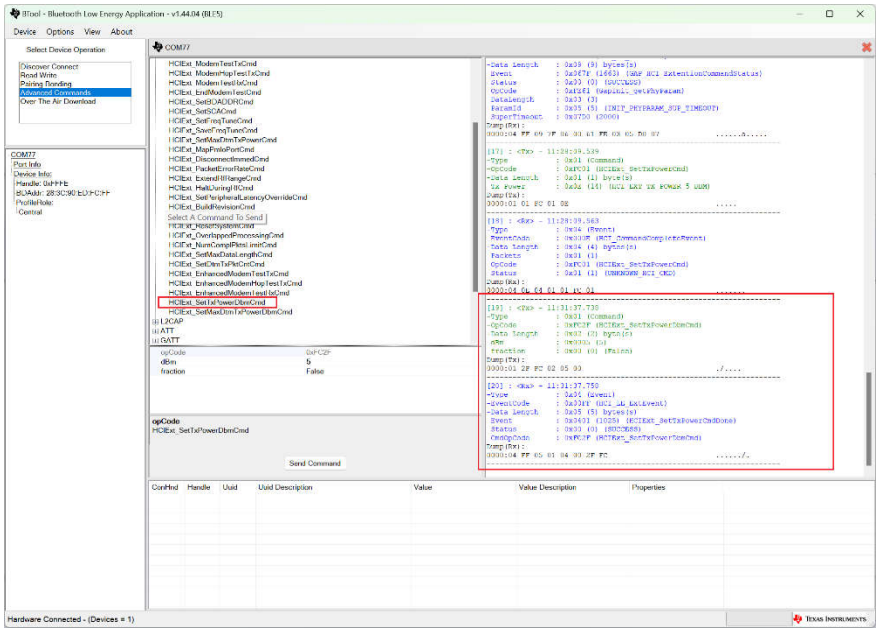
1. 程序运行中更改 HCI_EXT_SetMaxDtmTxPowerDbmCmd()
2. 对于 ADV/CONN 等 BLE 运行的 HCI_EXT_SetTxPowerDbmCmd()

对于第二种情况，BTool 是 TI 提供的一个，客户可以基于 BTool 进行产品验证测试，硬件连接请参考文档 TI 蓝牙产品 BQB RF-PHY 测试及 SRRC 定频测试的配置方法 (Rev. A)， BTool 的使用手册路径为 BTool User Manual。

BTool 路径：<SDK_PATH>\tools\ble\btool

Host_test 路径：<SDK_PATH>\examples\rtos\LP_EM_CC2745R10_Q1\ble\host_test

需要注意的是，BTool 中 HCIExt_SetTxPowerCmd 命令已经不再使用，请使用 Advanced Commands → HCI→ HCIExt_SetTxPowerDbmCmd，更改下方的 dBm 到您要求的数值。



3.3 为什么我的程序会进入 iCall_abort

在调用蓝牙 API 时，一个常见的错误是程序进入 iCall_abort。导致 iCall_abort 的最常见原因有两个。

- 从自定义任务中直接调用蓝牙 API
- 从中断处理函数或回调函数中直接调用蓝牙 API

为避免 iCall_abort，应使用 BLEAppUtil_invokeFunction 这个函数来进行上下文切换（context switching）。BLEAppUtil_invokeFunction 函数的作用是把要调用的 API 的函数指针作为参数送入消息队列，然后在 BLEAppUtil 任务中读取消息队列并执行对应 API。

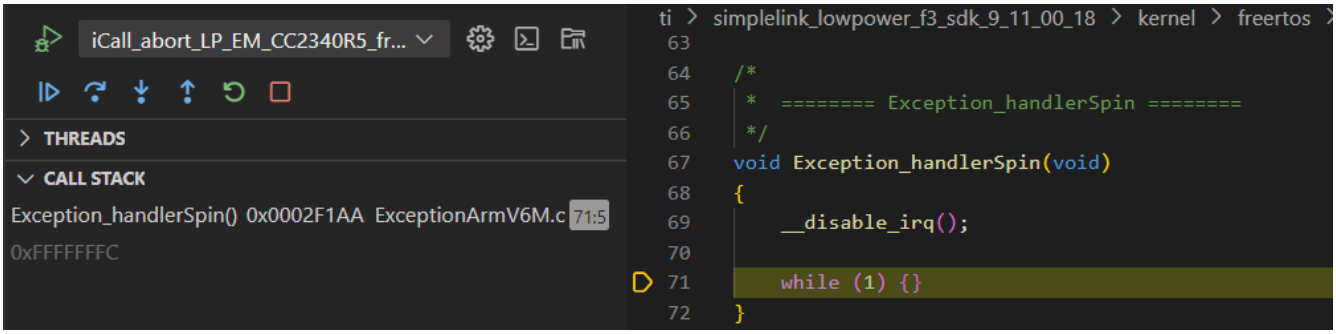
以下是一个典型案例，用户新建了一个任务，并在任务的主函数 myTaskMain 中调用 SimpleGatt_notifyChar4 来发送蓝牙数据。这种场景下就应当使用 BLEAppUtil_invokeFunction 来切换上下文，而不是直接调用 SimpleGatt_notifyChar4：

```
void *myTaskMain(void *arg)
{
    while(1)
    {
        if(sendNotifyTrigger == 1)
        {
            // Call BLE API directly from a non-iCall task will cause iCall_abort
            //SimpleGatt_notifyChar4();
            // Using BLEAppUtil_invokeFunctionNoData to avoid iCall_abort
            BLEAppUtil_invokeFunctionNoData(SimpleGatt_notifyChar4);

            sendNotifyTrigger = 0;
        }
    }
}
```

在使用 BLEAppUtil_invokeFunction 函数时，还有一个注意事项，那就是 BLEAppUtil_invokeFunction 函数的第二个参数 pData 应当指向由 BLEAppUtil_malloc 分配的内存区域，而不可以直接传入全局变量或局部变量的地址。例如下面这种情况：

```
if(sendParamUpdateReqTrigger == 1)
{
    bStatus_t status;
    gapUpdateLinkParamReq_t pParamUpdateReq =
    {
        .connectionHandle = 0,
        .intervalMin = 400,
        .intervalMax = 800,
        .connLatency = 0,
        .connTimeout = 600
    };

    // Send a connection param update request
    BLEAppUtil_invokeFunction((InvokeFromBLEAppUtilContext_t)BLEAppUtil_paramUpdateReq,
                              (char *)&pParamUpdateReq);
}
```

由于直接将 pParamUpdateReq 的地址传入了 BLEAppUtil_invokeFunction，在释放时非常容易造成内存错误（深层原因是在 memory advanced feature 启用时 malloc 的具体实现造成的），从而导致进入 Exception_handlerSpin：



正确的调用方法是创建一个指针变量，并用 BLEAppUtil_malloc 为它分配内存，然后将这个指针作为参数传入 BLEAppUtil_invokeFunction。这样在释放时会被安全释放。

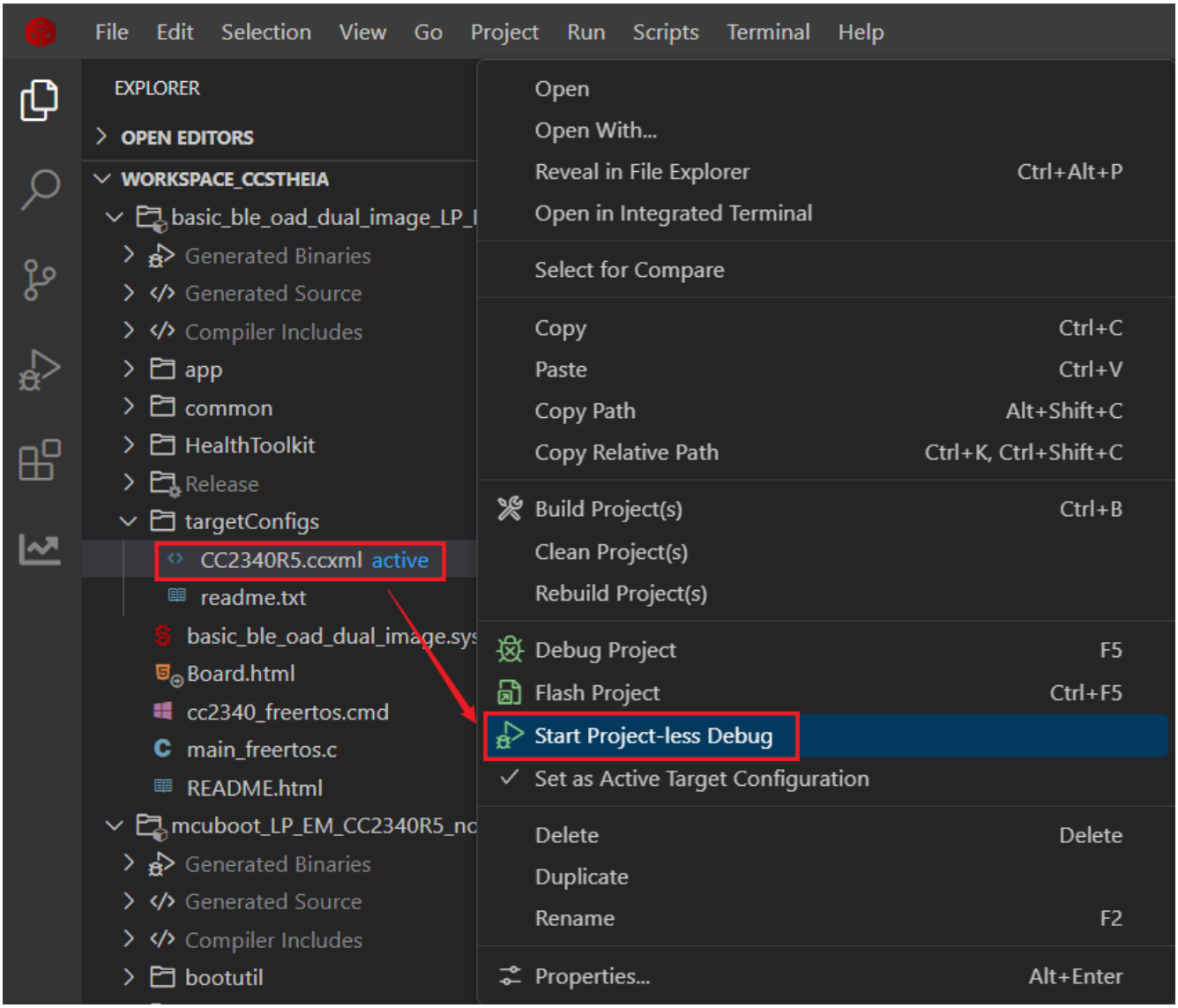
```
if(sendParamUpdateReqTrigger == 1)
{
    bStatus_t status;
    gapUpdateLinkParamReq_t *pParamUpdateReq = BLEAppUtil_malloc(sizeof(gapUpdateLinkParamReq_t));
    pParamUpdateReq->connectionHandle = 0;
    pParamUpdateReq->intervalMin = 400;
    pParamUpdateReq->intervalMax = 800;
    pParamUpdateReq->connLatency = 0;
    pParamUpdateReq->connTimeout = 600;

    // Send a connection param update request
    BLEAppUtil_invokeFunction((InvokeFromBLEAppUtilContext_t)BLEAppUtil_paramUpdateReq,
                              (char *)pParamUpdateReq);
}
```

3.4 如何 Debug 带有 MCUBoot 的工程

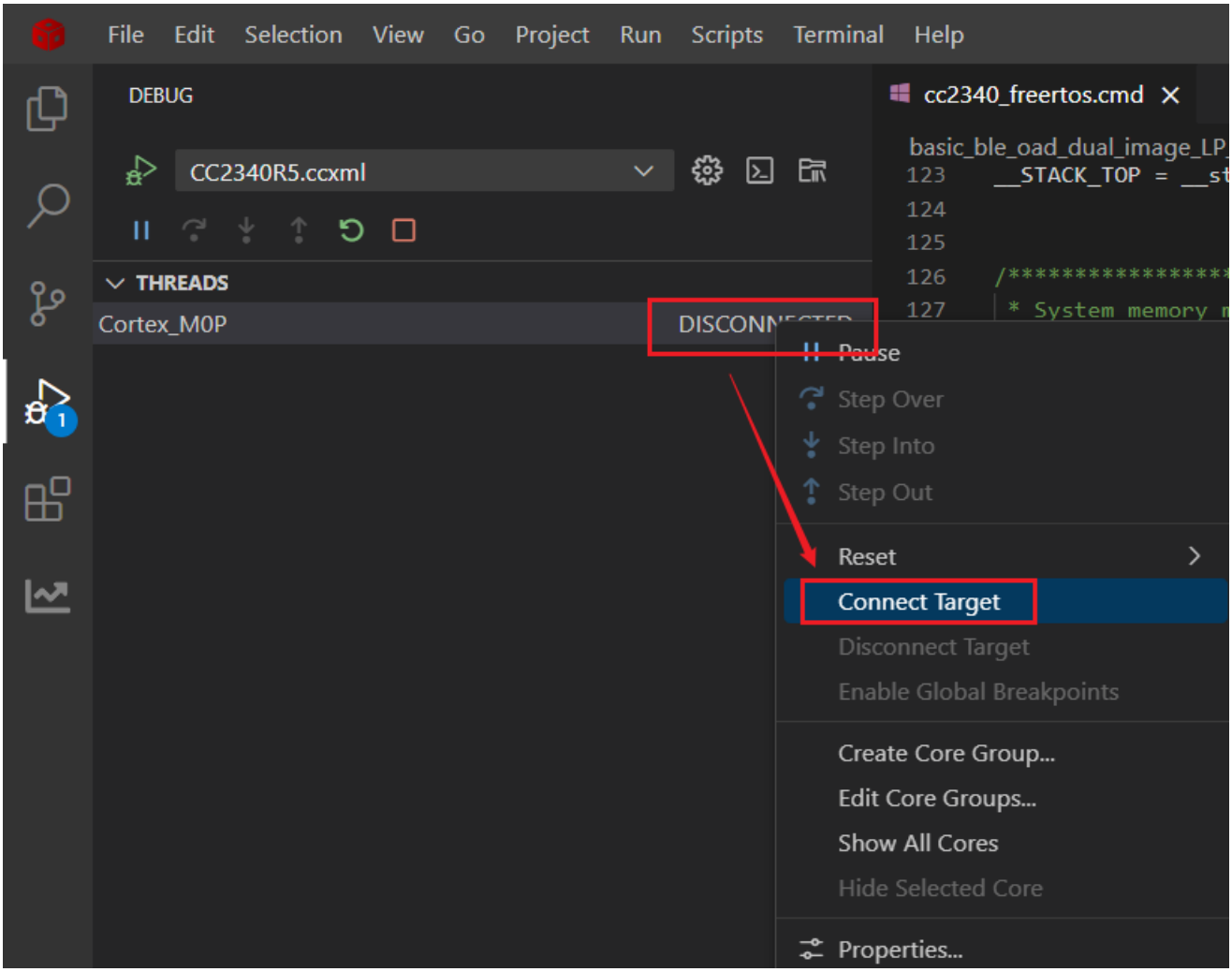
有些程序需要同时烧写多个镜像，例如带有 OTA（空中升级）功能的程序，需要配合 MCUBOOT 一起烧录才能正常运行。这类工程在 CCS 中可以按照以下步骤进行调试。

以 CC2340R5 的 MCUBOOT 和 basic_ble_oad_dual_image 工程为例：

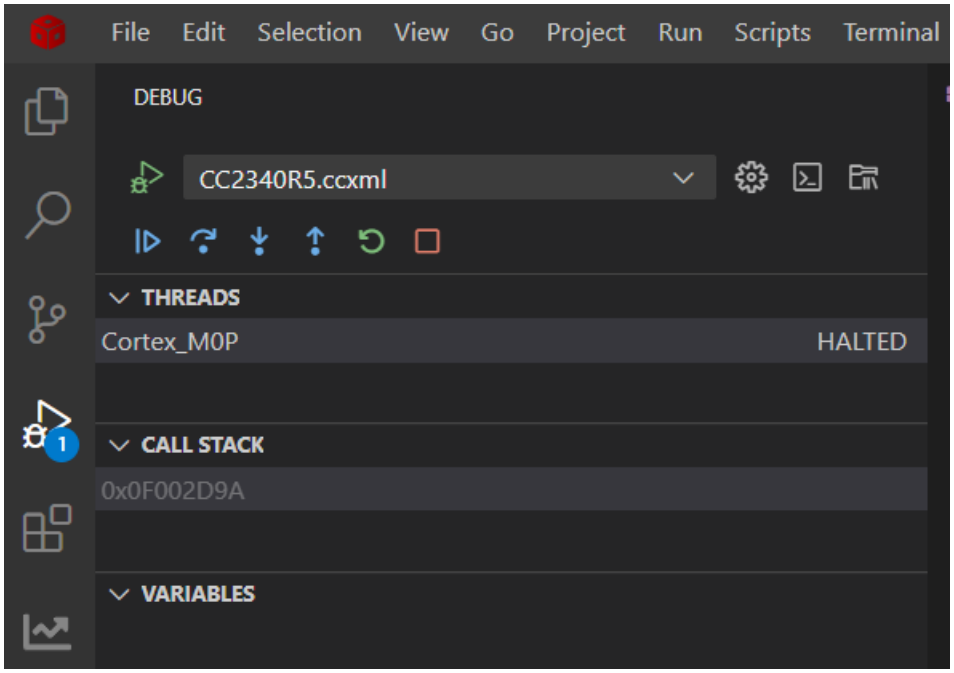
1. 首先用 Uniflash 将 MCUBOOT 和 basic_ble_oad_dual_image 工程同时烧录到 CC2340 中，具体步骤可参考：OAD Target Setup
2. 右键单击 MCUBOOT 工程或 basic_ble_oad_dual_image 工程中的 targetConfigs/CC2340R5.ccxml 文件，在菜单中选择 Start Project-less Debug:



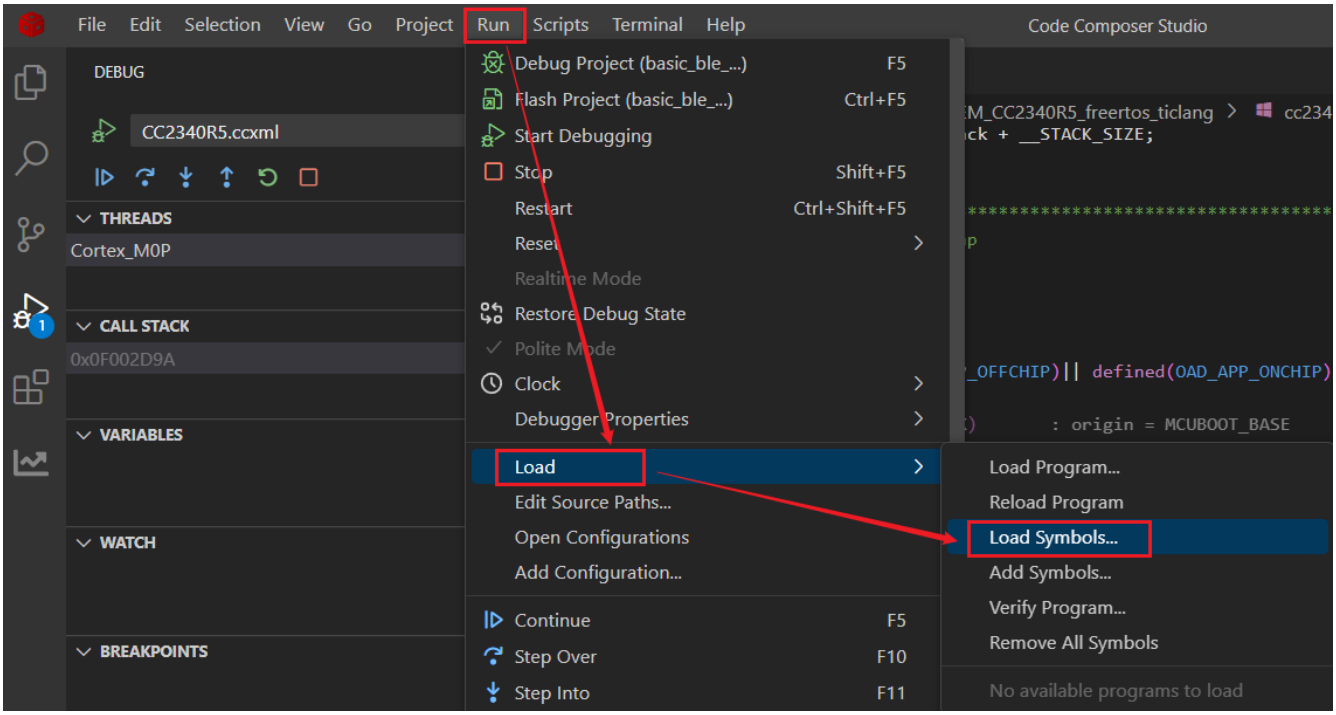
3. CCS 会进入 Debug 界面，展开 THREADS 菜单，可以看到目标芯片当前处在 DISCONNECTED 状态。右键在 DISCONNECTED 处单击，点击菜单中的 Connect Target 连接目标芯片：

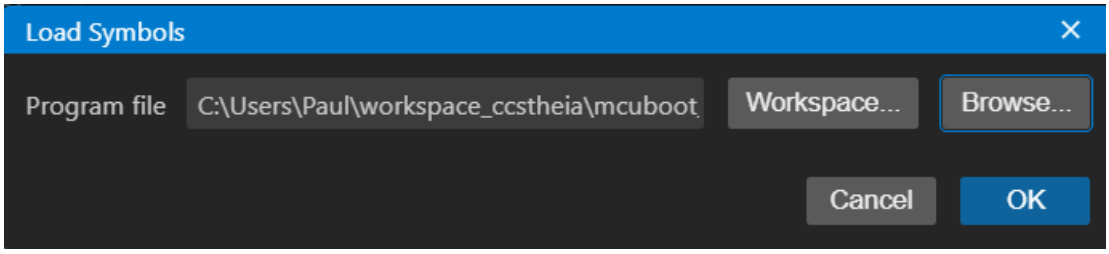


4. 连接成功后可以看到状态变成了 HALTED（挂起状态），同时上边一排调试按钮也被点亮，可以进行 debug 操作了。但此时 CALL STACK 中只能显示当前程序指针所在地址，看不到对应代码，这是因为我们还没有加载程序镜像所对应的 symbol 文件：

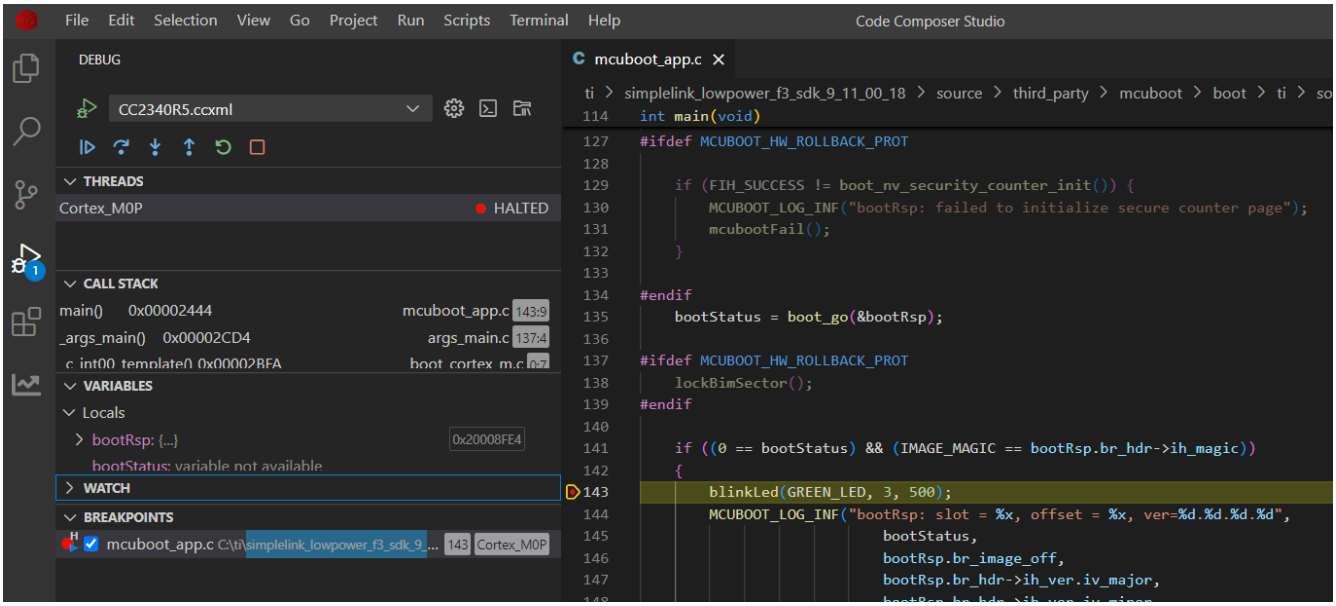


5. 点击 Run→ Load → Load Symbols，然后在弹出的对话框中选择 Browse，找到工程对应的 .out 文件（例如 mcuboot_LP_EM_CC2340R5_nortos_ticlang.out），点击 OK 加载。请注意这里的 symbol 文件必须是 .out 格式，其它格式（hex 或 bin）是不包含 symbol 信息的：

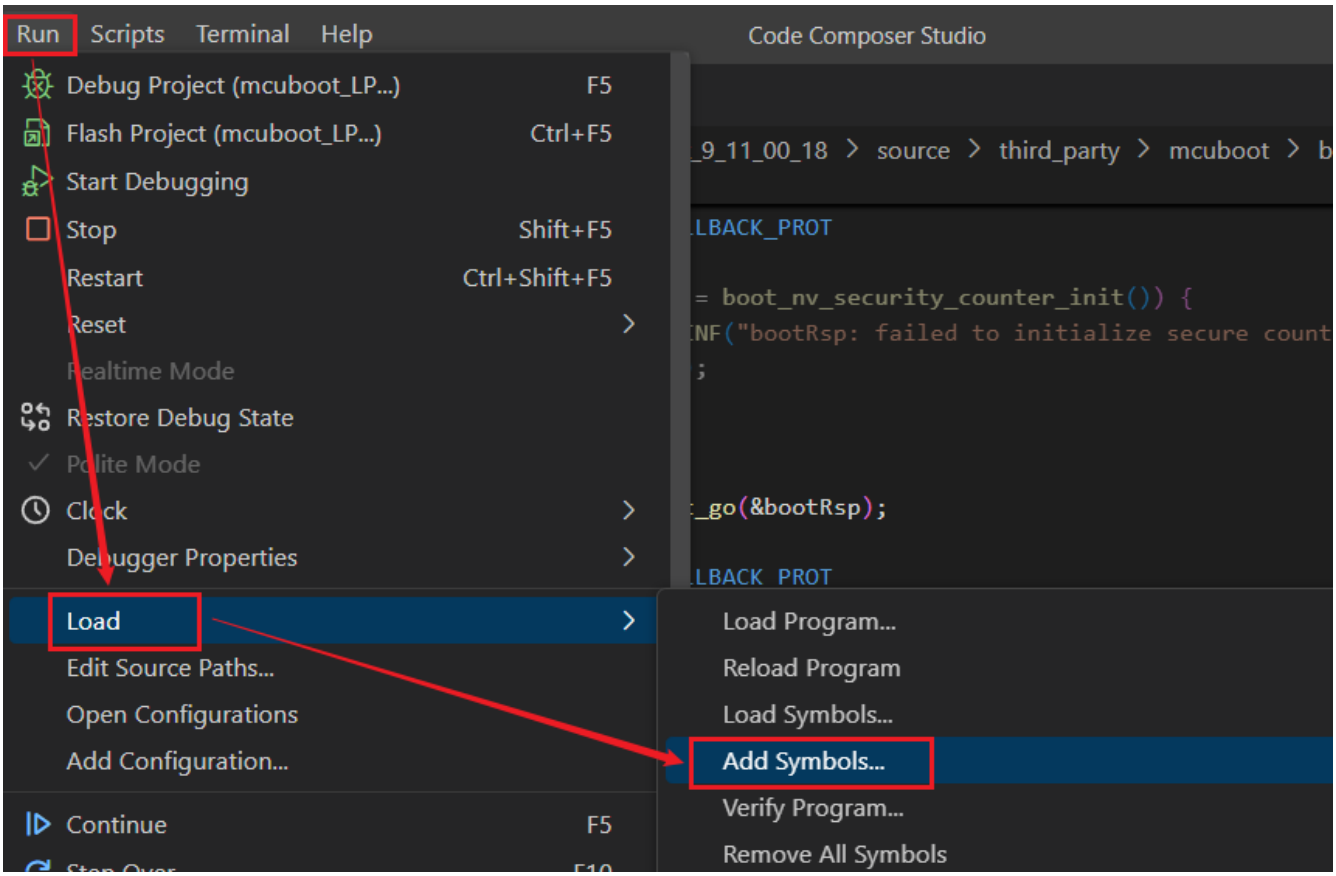


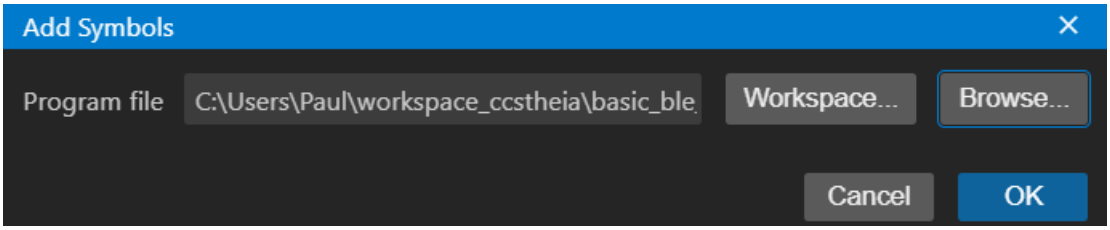


6. 加载 symbol 之后我们就可以在 CCS 中正常 debug 了，例如在程序中添加断点、单步调试、查看变量和寄存器等。请注意加载 symbol 只是加载了工程中各个变量和函数和地址的对应关系，并不会重新烧录程序，因此芯片中运行的仍然是之前 Uniflash 烧录的程序。如果修改了程序，需要重新用 Uniflash 烧写，再按照上述步骤加载 symbol，否则 CCS 中的 symbol 可能与芯片中实际运行的程序对不上：



7. 现在我们仅仅加载了 MCUBOOT 的 symbol，当程序从 MCUBOOT 跳转到 basic_ble_oad_dual_image 之后，还需要添加 basic_ble_oad_dual_image 工程的 symbol 文件。点击 Run→ Load → Add Symbols (注意是 Add 不是 Load) ，将 basic_ble_oad_dual_image_LP_EM_CC2340R5_freertos_ticlang.out 加载到 CCS，就可以同时调试两个工程了：



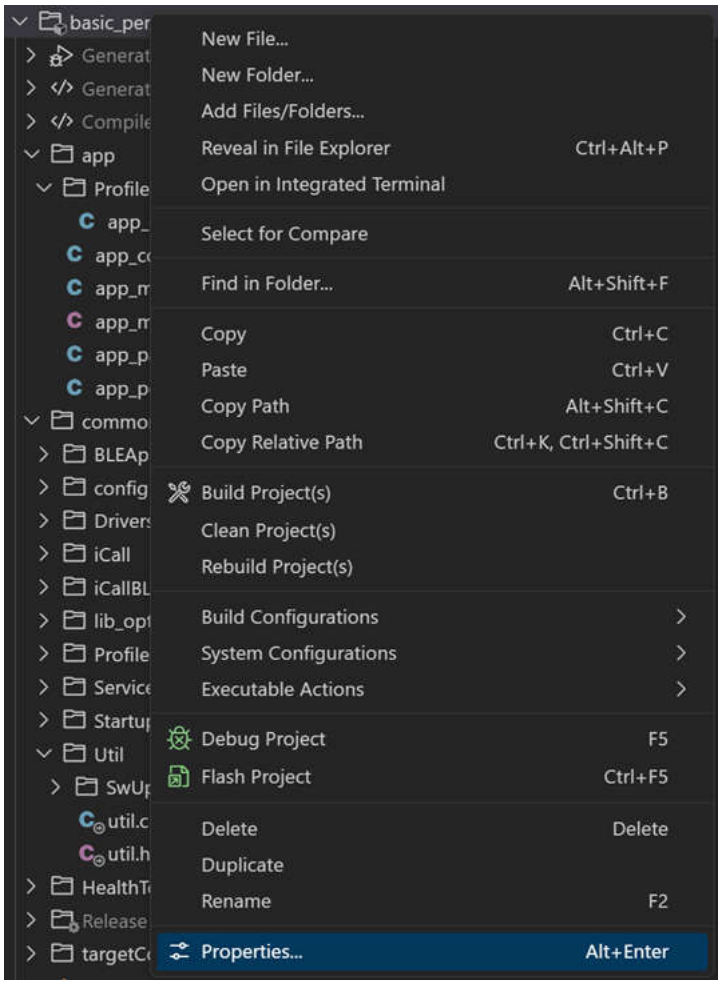


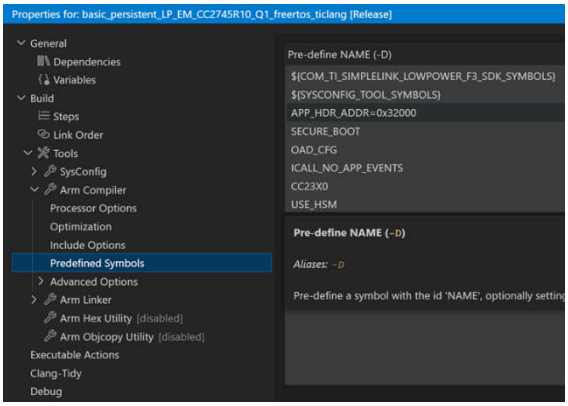
Tips：如果要查看 basic_ble_oad_dual_image 工程所对应的 ROV 信息，则需要先在 Load Symbols 这一步选择 basic_ble_oad_dual_image_LP_EM_CC2340R5_freertos_ticlang.out，再添加 mcuboot_LP_EM_CC2340R5_nortos_ticlang.out。否则由于两个工程配置不同，ROV 中的信息可能无法正常显示。

3.5 在 OAD 应用中调整 APP 所在 flash 位置和尺寸

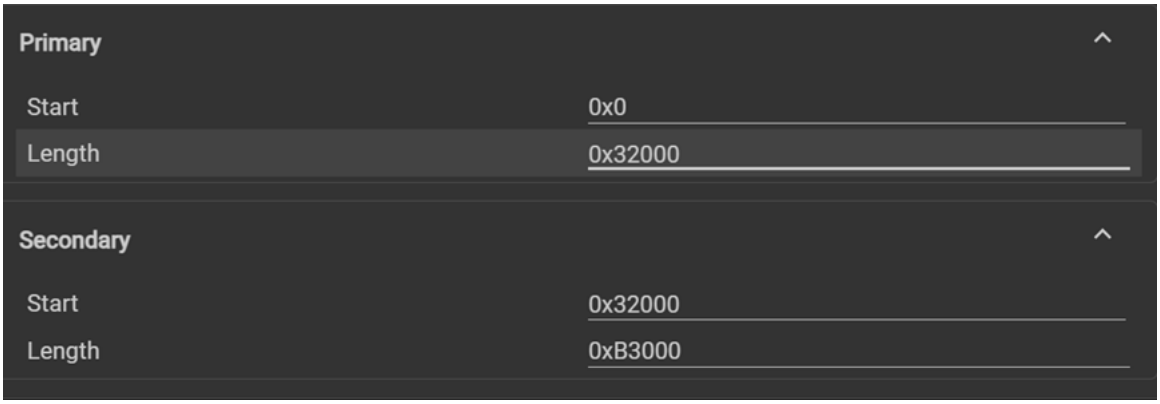
在 basic_ble_oad_onchip 例程中，设置了 basic_persistent 和 oad_onchip 两部分例程，分别烧录于 0x6000 位置和 0x32000 位置。但是工程师可能产生进一步的需求，即需要改变 app 所占据的位置。这一修改的方法如下：

1. 打开 basic_persistant 工程，打开 properties→predefined symbols，修改 APP_HDR_ADDR 到新的 persistent 工程结束地址（一般这也是 basic oad 工程的起始地址）

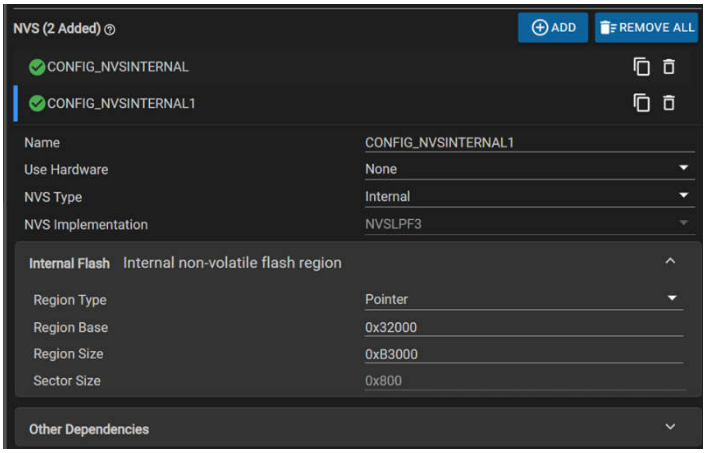




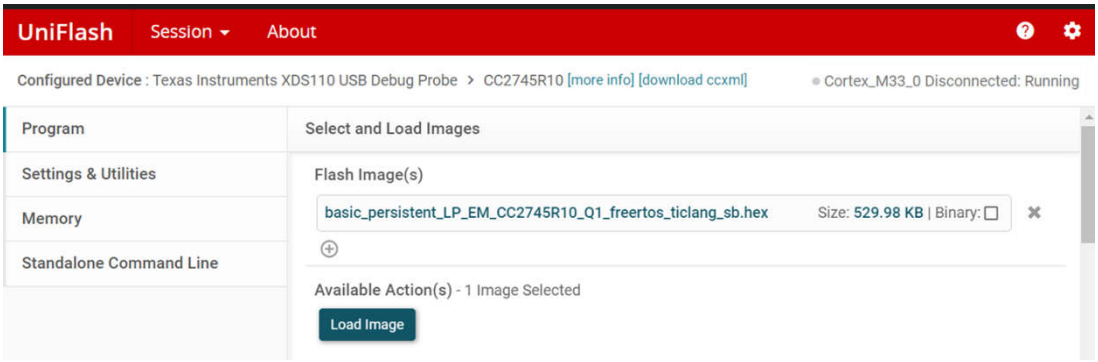
2. 在 basic_persistent.syscfg→Device Configuration→Security Configuration 中找到 primar y 和 secondary image，按照自己的需求修改其起始位置和尺寸



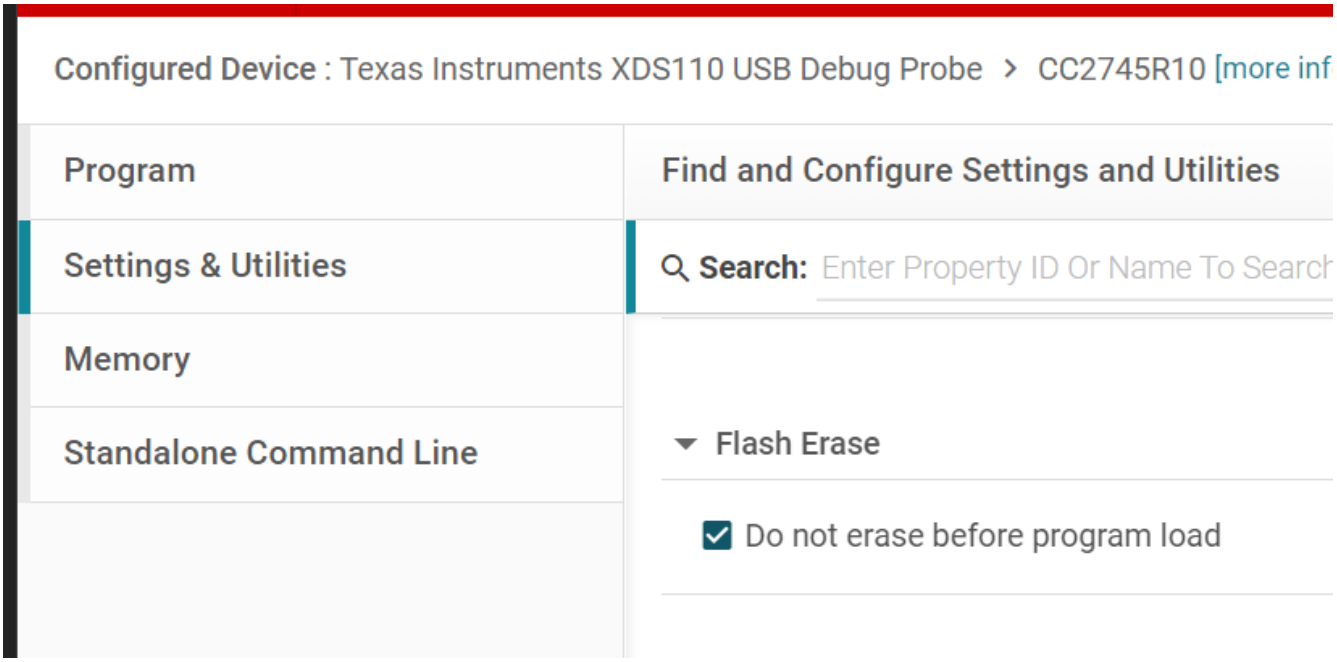
3. 在 basic_persistent.syscfg→NVS 中修改 CONFIG_NVSINTERNAL1 来匹配上面设置的 secondary image。在 basic_ble_oad_onchip 工程中重复 1-3 步骤。编译两个工程，取 basic_ble_persistent_LP_EM_CC2745R10_Q1_freertos_ticlang_sb.hex 和 basic_ble_oad_onchip_LP_EM_CC2745R10_Q1_freertos_ticlang_sb.bin 两个文件。



在 uniflash 中烧录 basic_ble_persistent_LP_EM_CC2745R10_Q1_freertos_ticlang_sb.hex



烧录完成之后修改烧录设置 Settings & Utilities→Flash Erase→勾选 "Do not erase before program load"



从刚才设置的 secondary image 起始位置烧录
basic_ble_oad_onchip_LP_EM_CC2745R10_Q1_freertos_ticlang_sb.bin

3.6 在 .cmd 中自定义 SECTION 并在代码中使用

在一些工程实现中，工程师想要分配一段自己的数据空间来服务于应用，且这段空间是对应于一个特定起始和结束地址的。那么您就可以对 .cmd 的 SECTIONS 段声明新的段空间。

.cmd 文件是将指定的段分配给 Flash 或者 SRAM 的 linker cmd 文件，对于此文件的使用可以参考文档 Linker Command 文件，对于 Linker Command 文件大家最关心的是 Memory 和 Section 这两个模块如何看懂和使用，可以参考两个链接 Memory User Manual 和 Section User Manual 。

```
SECTIONS
{
    .mysection :
{
    __my_section_start__ = ;
*(.my_section)
__my_section_end__ = .;
} > RAM
}
```

并在代码中，extern uint8_t __my_section_start__; 和 extern uint8_t __my_section_end__ ；来进行使用。

3.7 如何使用非初始化变量区

非初始化变量区是指在系统上电或软复位（如 OTA 升级）时不会被清零或重新赋值的内存区域，常用于在程序重启后仍需保留的数据。

要使用初始化变量区定义以及使用变量，需要三步操作：

1. 在上述的.cmd 文件中，在 RAM 的 SECTIONS 中，声明下方的空间，如图所示。

```
.TI.noinit : > SRAM, type = NOINIT
```

```
#if defined(RAM_BASE) && defined(RAM_SIZE)
    .ramVecs        :   > SRAM, type = NOLOAD, ALIGN(256)
    .data           :   > SRAM
    .bss            :   > SRAM
    .sysmem         :   > SRAM
    .stack          :   > SRAM (HIGH)
    .nonretenvar    :   > SRAM
    .TI.noinit      :   > SRAM, type = NOINIT
#endif
```

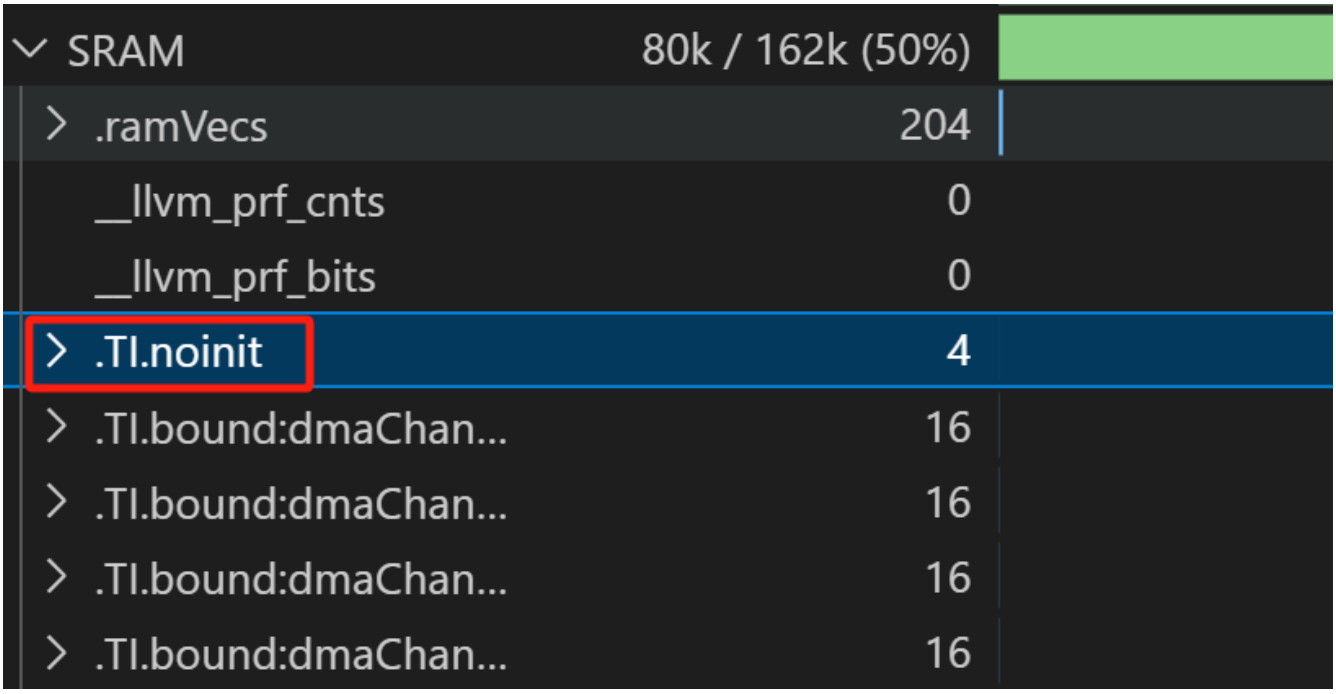
2. 为了将数据放置在非初始化变量区域中，应用程序必须使用编译器属性声明并将其放置在 .TI.noinit 部分中。

```
__attribute__((noinit)) int noinit_global;
__attribute__((noinit,location(0x100))) int noinit_location_global;
```

```
87  #endif //#if defined( HOST_CONFIG ) && ( HOST_CC
88
89
90  __attribute__((noinit)) int noinit_global;
91
92  //*****************************************************
93  //! Functions
94  //*****************************************************
95
96  /*****************************************************
97   * @fn      criticalErrorHandler
98   *
99   * @brief   Application task entry point
100  *
101  * @return  none
```

3. 在您的代码里面使用该变量，编译您的程序，会在 Memory Allocation 中看到对应的定义。

```
186      }
187
188      noinit_global++;
189      MenuModule_printf(line: APP_
190
191      sleep(5);
192      Power_reset();
193
```



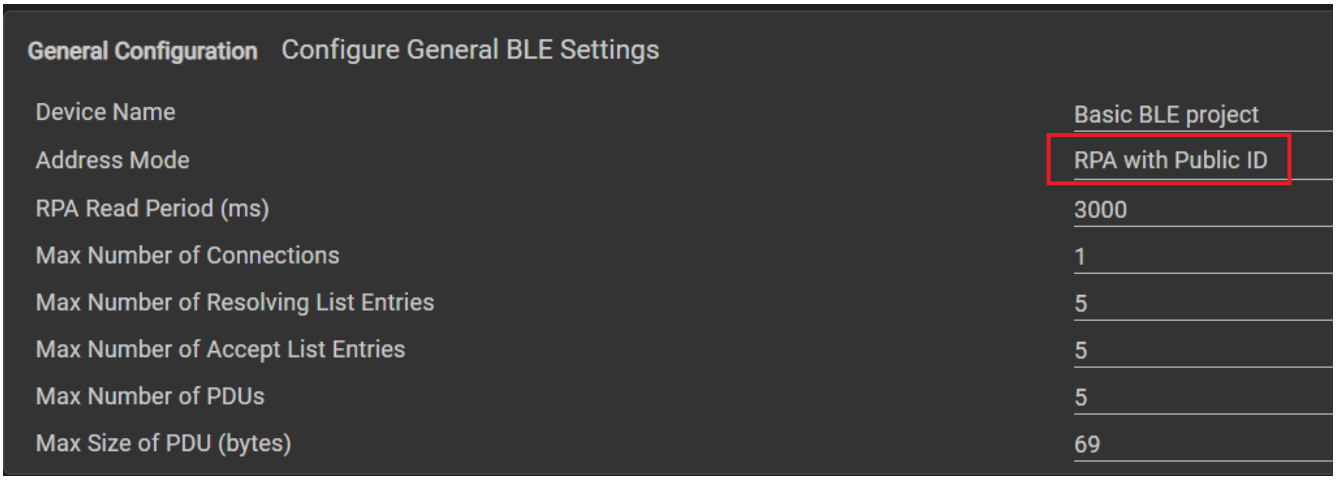
以上图的代码为例。每 5 秒钟，就会软复位，noinit_global 会一直增加，并不会初始化为 0。

请注意：硬件复位（掉电重启 /reset 引脚复位）会使该变量重置。如果需要掉电保持，请使用 NV。

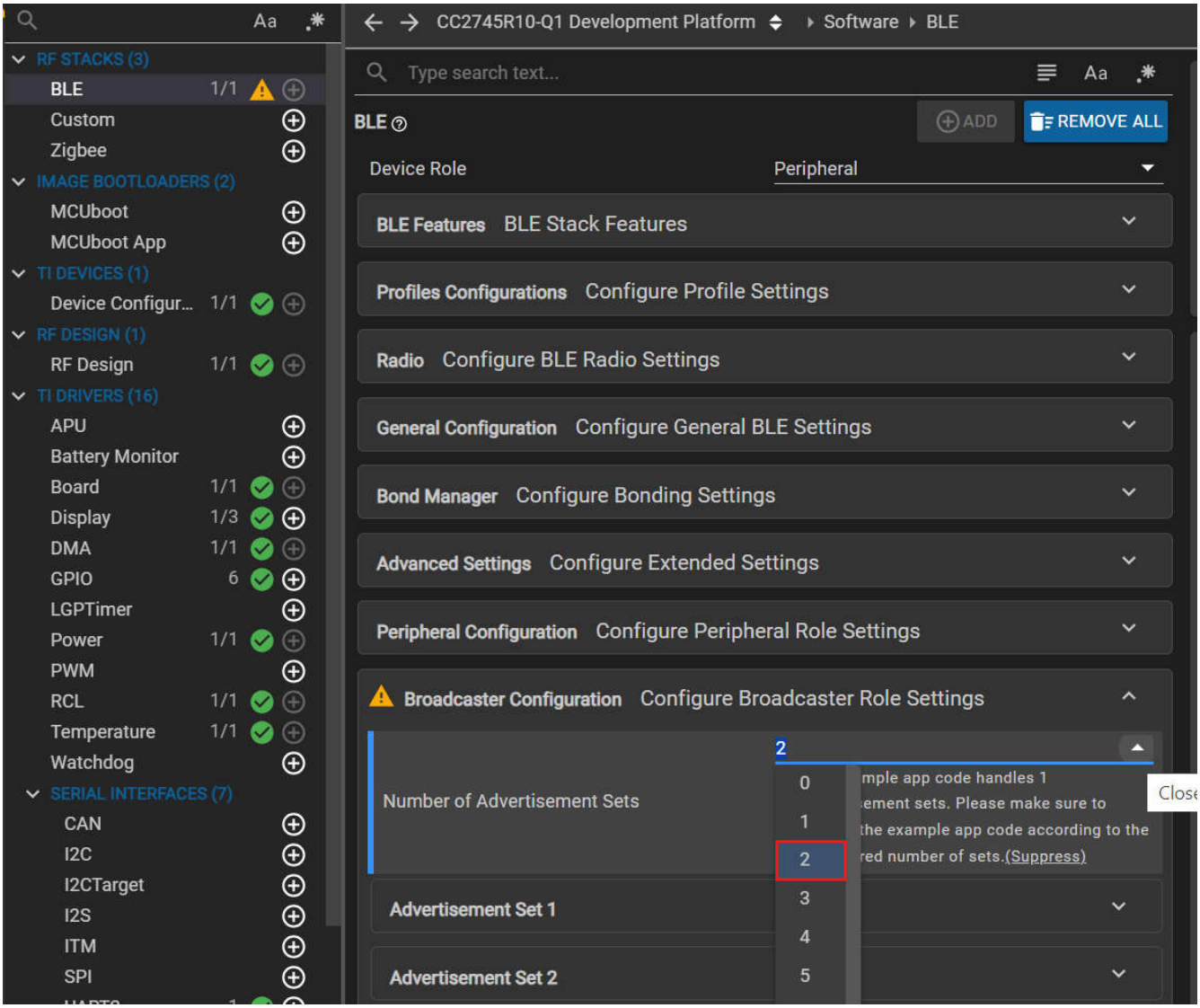
3.8 如何同时发出 Public 地址和 RPA 地址的两路 ADV

在汽车钥匙应用领域，由于 CCC，ICCE 和 ICCOA 等应用规范的存在，经常会有发出多路不同地址类型广播的应用，这里我们介绍如何以 basic_ble 示例程序为基础添加两路广播。

1. 在 Sysconfig → BLE → General Configuration 将 Address Mode 改为 RPA with Public Address



2. Broadcast Configuration 中添加一路广播，此处应忽略 Sysconfig 提示的 Warning，因为我们后面会在代码中添加对第二路广播的处理。

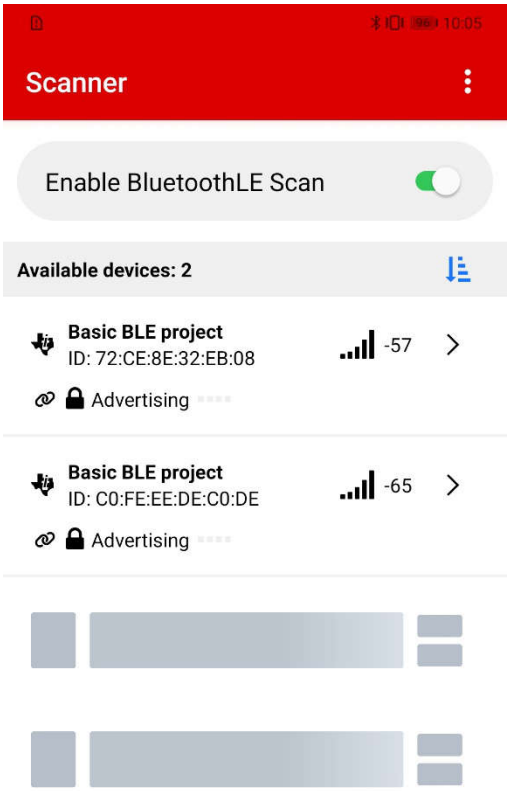


3. 在 app_peripheral.c 中生命第二组 ADV，并按照如下修改 Peripheral_start 函数。

```
uint8 peripheralAdvHandle_2;

const BLEAppUtil_AdvInit_t advSetInitParamsSet_2 =
{
    .advDataLen         = sizeof(advData2),
    .advData            = advData2,

    .scanRespDataLen    = sizeof(scanResData2),
    .scanRespData       = scanResData2,

    .advParam           = &advParams2
};

const BLEAppUtil_AdvStart_t advSetStartParamsSet_2 =
{
    /* Use the maximum possible value. This is the spec-defined maximum for */
    /* directed advertising and infinite advertising for all other types */
    .enableOptions          = GAP_ADV_ENABLE_OPTIONS_USE_MAX,
    .durationOrMaxEvents    = 0
};
```

```
bStatus_t Peripheral_start()
{
    bStatus_t status = SUCCESS;

    status = BLEAppUtil_registerEventHandler(eventHandler: &peripheralConnHandler);
    if(status != SUCCESS)
    {
        // Return status value
        return(status);
    }

    status = BLEAppUtil_registerEventHandler(eventHandler: &peripheralAdvHandler);
    if(status != SUCCESS)
    {
        return(status);
    }

    status = BLEAppUtil_initAdvSet(advHandle: &peripheralAdvHandle_1, advInitInfo: &advSetInitParamsSet_1);
    if(status != SUCCESS)
    {
        // Return status value
        return(status);
    }

    status = GapAdv_setParam(peripheralAdvHandle_2, GAP_ADV_PARAM_OWN_ADDRESS_TYPE, 0x00);
    if(status != SUCCESS)
    {
        return(status);
    }

    status = BLEAppUtil_advStart(handle: peripheralAdvHandle_1, advStartInfo: &advSetStartParamsSet_1);
    if(status != SUCCESS)
    {
        // Return status value
        return(status);
    }

    status = BLEAppUtil_initAdvSet(advHandle: &peripheralAdvHandle_2, advInitInfo: &advSetInitParamsSet_2);
        if(status != SUCCESS)
    {
        return(status);
    }

       status = BLEAppUtil_advStart(handle: peripheralAdvHandle_2, advStartInfo: &advSetStartParamsSet_2);
    if(status != SUCCESS)
    {
        // Return status value
        return(status);
    }

    // Return status value
    return(status);
}
```

4. 编译烧录后，在手机上打开 SimpleLink Connect APP，就能看到两路不同地址的广播了。



3.9 CC2745R10 芯片 E0 和 E1 的区别

CC2745R10-Q1 在 TI store 中有两种可选，分别是 CC2745R10E0WRHARQ1 和 CC2745R10E1WRHARQ1。

这是因为在 CC2745 的研发过程中，有过两个版本的样品，分别是 PG1.0 和 PG2.0。在量产阶段，分别编号为 CC2745R10E0WRHARQ1 和 CC2745R10E1WRHARQ1。二者均是可以进行量产的芯片。但主要的量产芯片是 CC2745R10E1WRHARQ1。

3.10 如何使更多的 DMA channel 来支持我的驱动需求

在一些情况下，工程师想要使用更多的 DMA 通道，例如同时使能 SPI0/1 和 UART0/1 以达成

四路串口同时进行数据传输的目的。在 CC27xx Technical Reference Manual (Rev. A)的 section 19.3.1 中，介绍了 CC2745 所定义的 12 路 DMA 通道。

Table 19-1 lists µDMA channel assignments to peripherals.

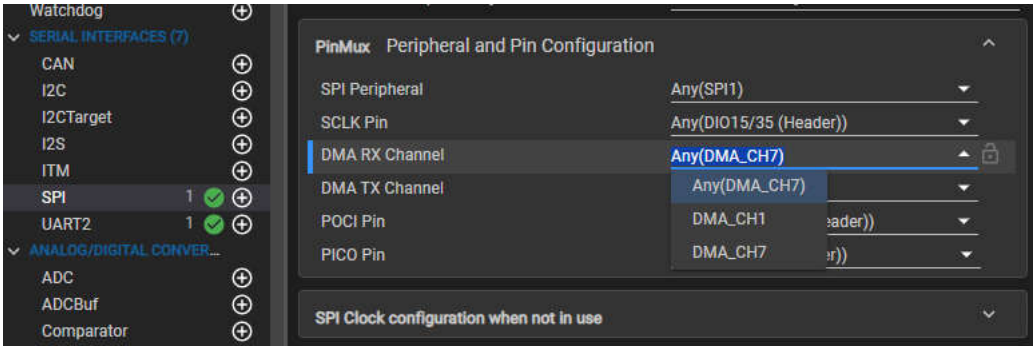
Table 19-1. Channel Assignments

µDMA Channel	Type (1) (2)	DTB (3)	Triggers
0	DCH	YES	SPI0TXTRG, UART1RXTRG
1	DCH	YES	SPI0RXTRG, UART1TXTRG
2	DCH	YES	LRFDTRG, UART0TXTRG
3	DCH	YES	ADC0TRG, UART0RXTRG
4	DCH	YES	LAESTRGA, LRFDTRG
5	DCH	YES	LAESTRGB, ADC0TRG
6	DCH	YES	CANTRGA,SPI1TXTRG
7	DCH	YES	CANTRGB,SPI1RXTRG
8	ECH	YES	ANY
9	ECH	YES	ANY
10	ECH	YES	ANY
11	ECH	YES	ANY

(1) Dedicated µDMA Channel (DCH) type can only be connected to peripherals with dedicated µDMA interface
(2) Event Publisher Channel (ECH) type can only be connected to publishers,

See also Section 4.4.5.1
(3) For DTB output, only SREQ and REQ is considered.

若工程师想要调整某些驱动所对应的 DMA 通道或添加新的通道，可以在工程的 sysconfig 中修改，以 SPI0 为例：



可以看到，如表格所述，SPI 可以被配置为 DMA_CH0/1 或 DMA_CH6/7，符合 sysconfig 中的实际情况。若工程师想要使用 DMA_CH8-11，需注意 CH8-11 为 Event Publisher Channel，只能在有外界 publisher 的情况下使用，不可任意配置为 SPI 或 UART 等驱动使用。

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、与某特定用途的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他安全、安保法规或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。对于因您对这些资源的使用而对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，您将全额赔偿，TI 对此概不负责。

TI 提供的产品受 TI 销售条款)、TI 通用质量指南 或 ti.com 上其他适用条款或 TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。 除非德州仪器 (TI) 明确将某产品指定为定制产品或客户特定产品，否则其产品均为按确定价格收入目录的标准通用器件。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

最后更新日期：2025 年 10 月