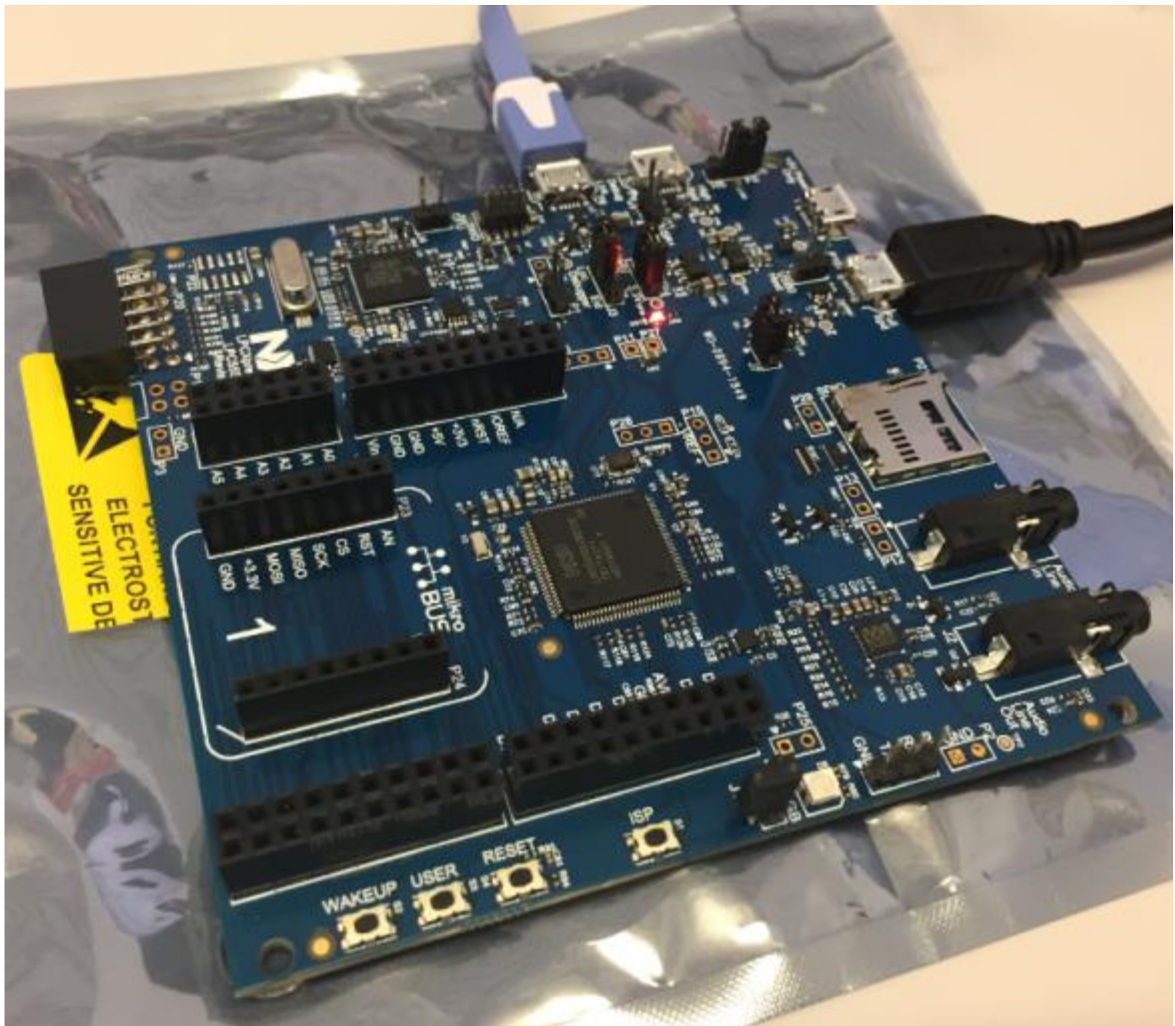
基于恩智浦 LPC55S69-EVK 开发板学习 ARMv8-M 内核的 TrustZone

原文：
https://community.nxp.com/community/general-purpose-mcus/lpc/blog/2019/05/05/trustzone-with-armv8-m-and-the-nxp-lpc55s69-evk

ARM TrustZone 是 Cortex-M33 的一个可选的安全特性，它将提高在微控制器上运行的嵌入式应用程序的安全性，如 LPC55S69-EVK 上的 NXP LPC55S69（双 M33 核）



NXP LPC55S69-EVK 板

与任何事情一样，学习和使用 TrustZone 的特性需要一些时间。ARM 提供了关于 TrustZone 的文档，但要将其应用于实际的开发板或工具链并不容易。
恩智浦 MCUXpresso SDK 附带了三个可用于 lpc55s69-evk 上的 TrustZone 的示例，因此我研究了一下这些示例以了解它的工作原理以及如何在我的应用程序中使用它。

软件和工具

我使用与我之前写的文章(“First Steps with the LPC55S69-EVK (Dual-Core ARM Cortex-M33 with Trustzone)“)相同的配置：

Windows10 系统下 10.3.1 版本的 MCUXpresso IDE（Eclipse 基于 ARM 嵌入式 GNU 工具链）

为 LPC55S69 提供的 MCUXpresso SDK V2.51.

本文介绍的大多数内容都适用于任何其他带有 TrustZone 的 Cortex-M33 环境

ARMV8-M 上的 TrustZone

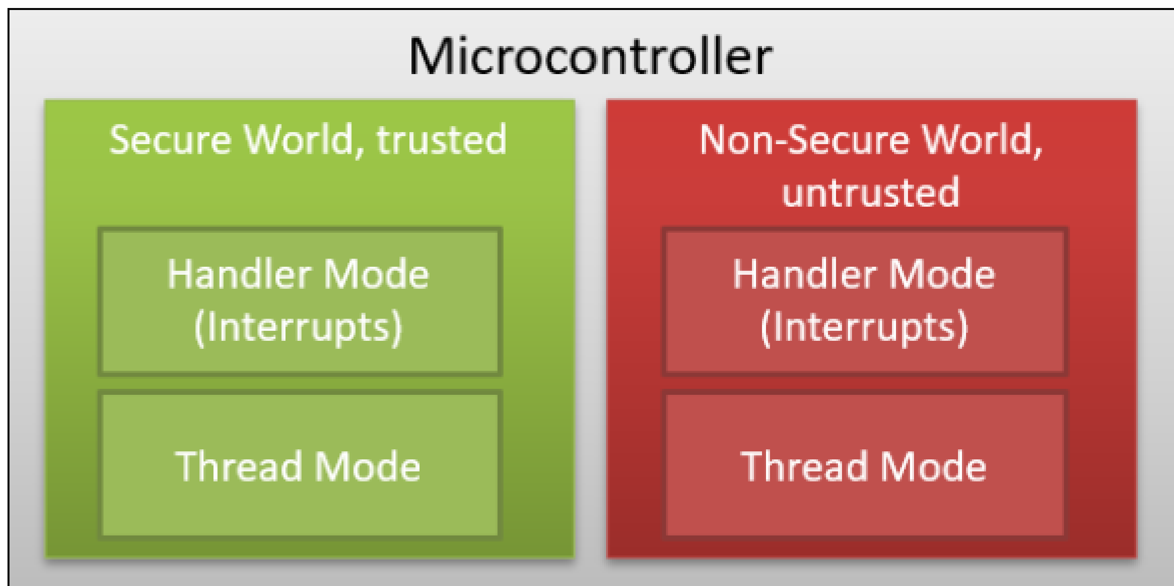
与 ARMV7-M 中的一样，处理器有两种基本模式：

线程模式：此模式通过复位或应用程序运行的常用模式进入。线程模式下的代码可以在特权（完全访问）或非特权（没有限制，例如由 MPU（内存保护单元））下执行。

处理者模式：此模式以特权级别执行，这是中断运行的模式。

TrustZone 保留并扩展了该模型。 ARMV8-M 上 TrustZone 的基本概念是将微控制器上的“不受信任”部分与“受信任”部分分开。有了这个分区，受信任方的 IP 可以得到保护，同时仍然允许“不受信任”的软件在 “不受信任”的一方运行。每个受信任和不受信任的部分可以具有不同的权限，例如某些硬件（GPIO 端口等）只能从受信任的一方访问，而不能从不受信任的一方访问。

我推荐阅读 ARM document about TrustZone.



安全与非安全世界

如果没有 TrustZone，就可以用 MPU 限制内存访问，带有 “安全世界” 和 “非安全世界” 的 TrustZone 概念扩展到“安全”或“受信任”硬件或外设访问。非安全函数只能通过 API 访问安全硬件，API 验证是否允许它通过安全世界访问硬件。所以让安全和不安全的部分一起工作是有办法的。

与使用 MPU 类似，这意味着需要考虑以下几点：：

- 设置内存区域和访问外设的安全权限
- 使用安全和非安全 API 及传输函数
- 保护安全世界限制调试或内存读取的功能（逆向工程）

MPU

ARMV8-M 体系结构的另一个重要变化是，MPU 区域的大小是 32 字节的粒度。在 armv7-m 中，大小必须是 2^n，这是我没弄明白过的，并且这使得它在实际应用中根本不可用（这可能是很少使用 MPU 的原因？）

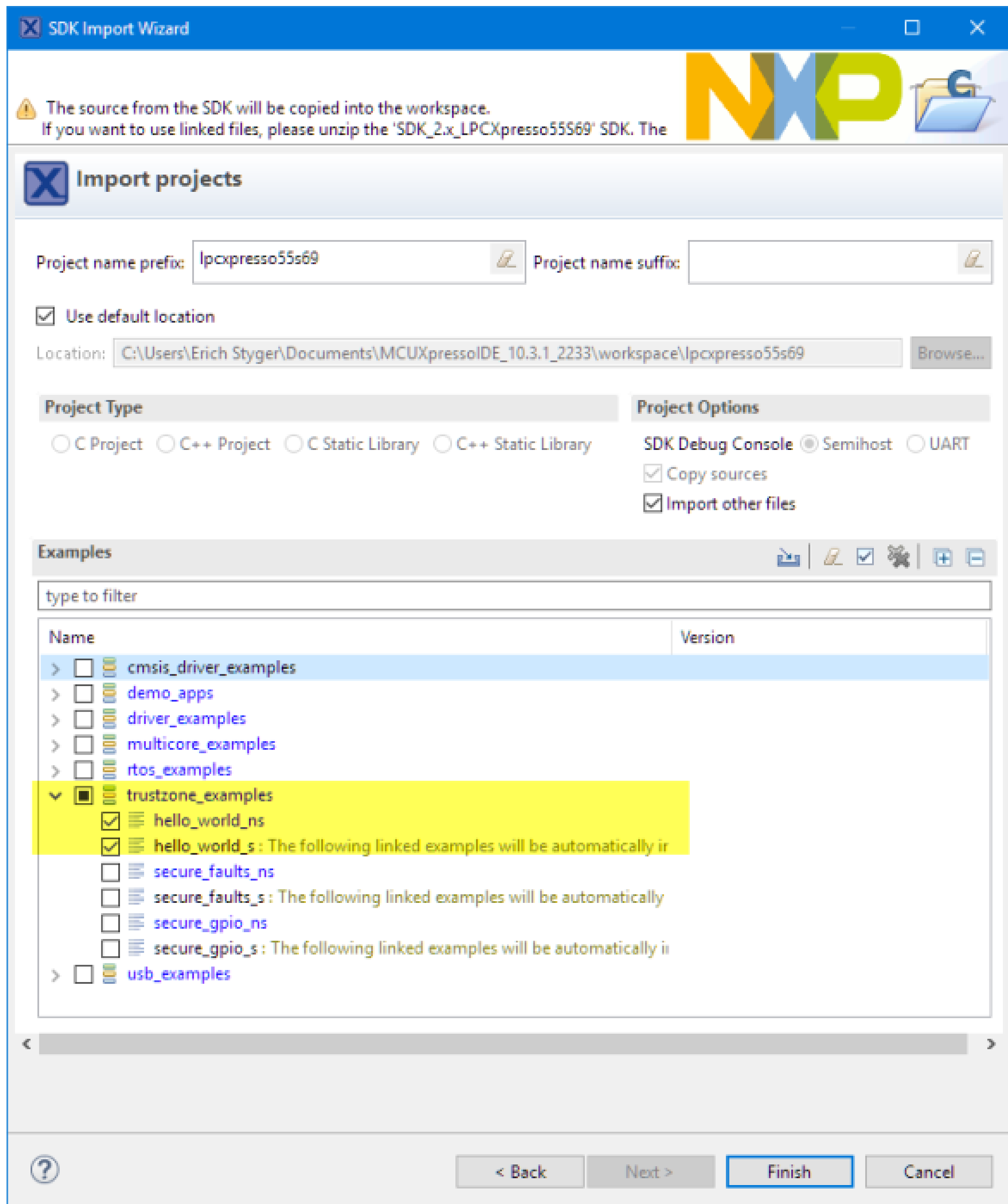
SAU 与 IDAU

因为这一切都不能只在 ARM 核中实现，所以实现 ARM 核的供应商在实现方面需要额外的设置。

- 安全属性单元（SAU）：它位于内核/处理器内部。
  实现定义属性单元（IDAU）：此单元位于处理器外部，SAU 和 IDAU 协同工作，用于授予/拒绝对系统（外围设备、内存）的访问。使用 SAU+IDAU，内存空间分为三种:
- **安全**: 安全世界的代码、堆栈、数据…
- **不安全**: 非安全世界的代码、堆栈、数据…
- **非安全可调用**: 使用安全网关向量表进入安全代码
  最重要（也有点让人困惑）的是，SAU 的设置是第一位的，而 IDAU 是用来让事物变得“不安全”的：
- SAU(安全的) + IDAU(不安全的) => 安全的
- SAU(不安全的) + IDAU(不安全的) => 不安全的
- SAU(非安全可调用) + IDAU(不安全的) ==> 非安全可调用
  或者换言之：默认情况下是安全的，而使用 IDAU，安全级别设置为较低的级别。.

工程

是时候看看一个例子了！NXP MCUXpresso SDK 已经提供了一个示例，展示了如何从安全的区域调用非安全的区域。从“Import SDK 示例”中，我选择了一些例子去演示 TrustZone

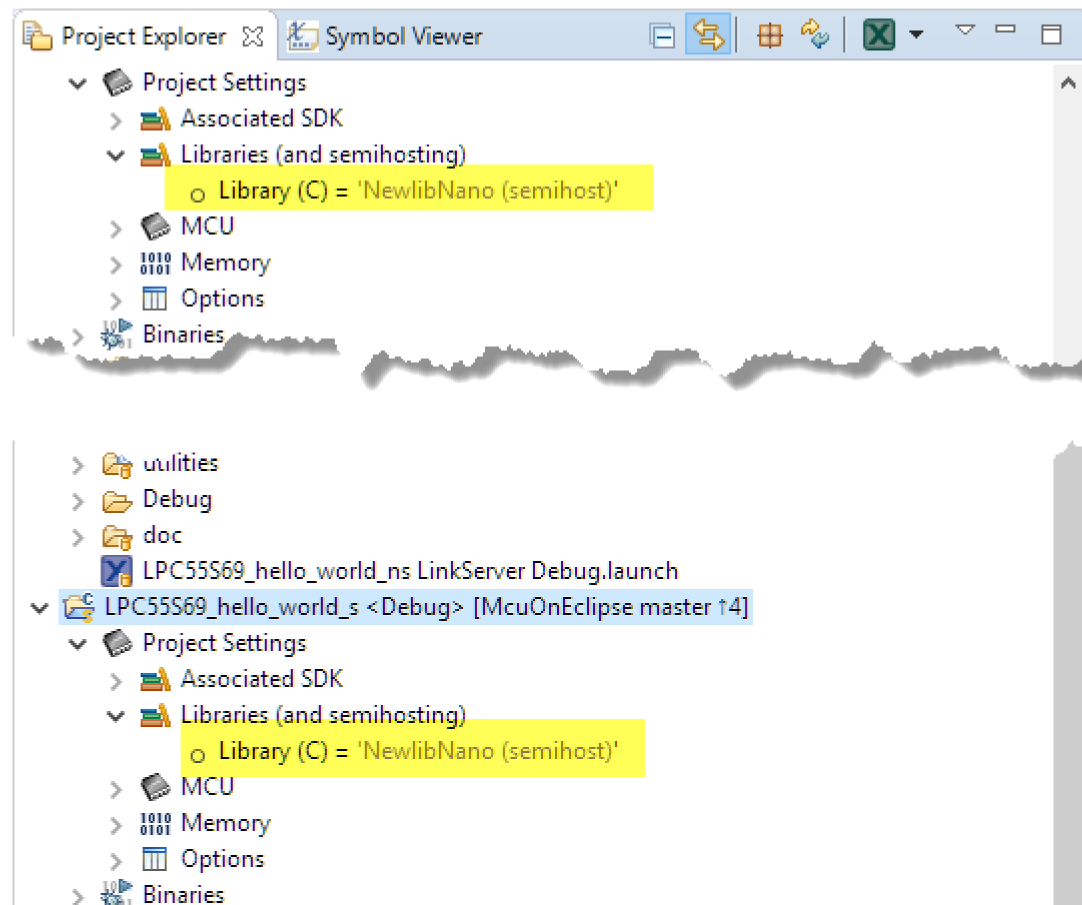


TrustZone 示例

TrustZone 中的“hello_world”示例在安全端执行一些代码，最后将控制权传递给非安全端以执行非安全应用程序。该示例遵循安全引导加载程序的模式，然后调用非安全应用程序启动。我已经调整并复制了本文中讨论的工程，您可以在 GITHUB 上找到它们: https://github.com/ErichStyger/mcuoneclipse/tree/master/Examples/MCUXpresso/LPC55S69-EVK

“ns”（非安全）和“s”安全项目协同工作。使用安全和非安全的应用程序部分并不能使事情变得更简单，而且似乎没有很多关于这个主题的文档。所以我研究了“hello world”这个例子，以便更好地理解它是如何工作的。

我已将两者配置为使用 newlib（nano）somihost 库：：

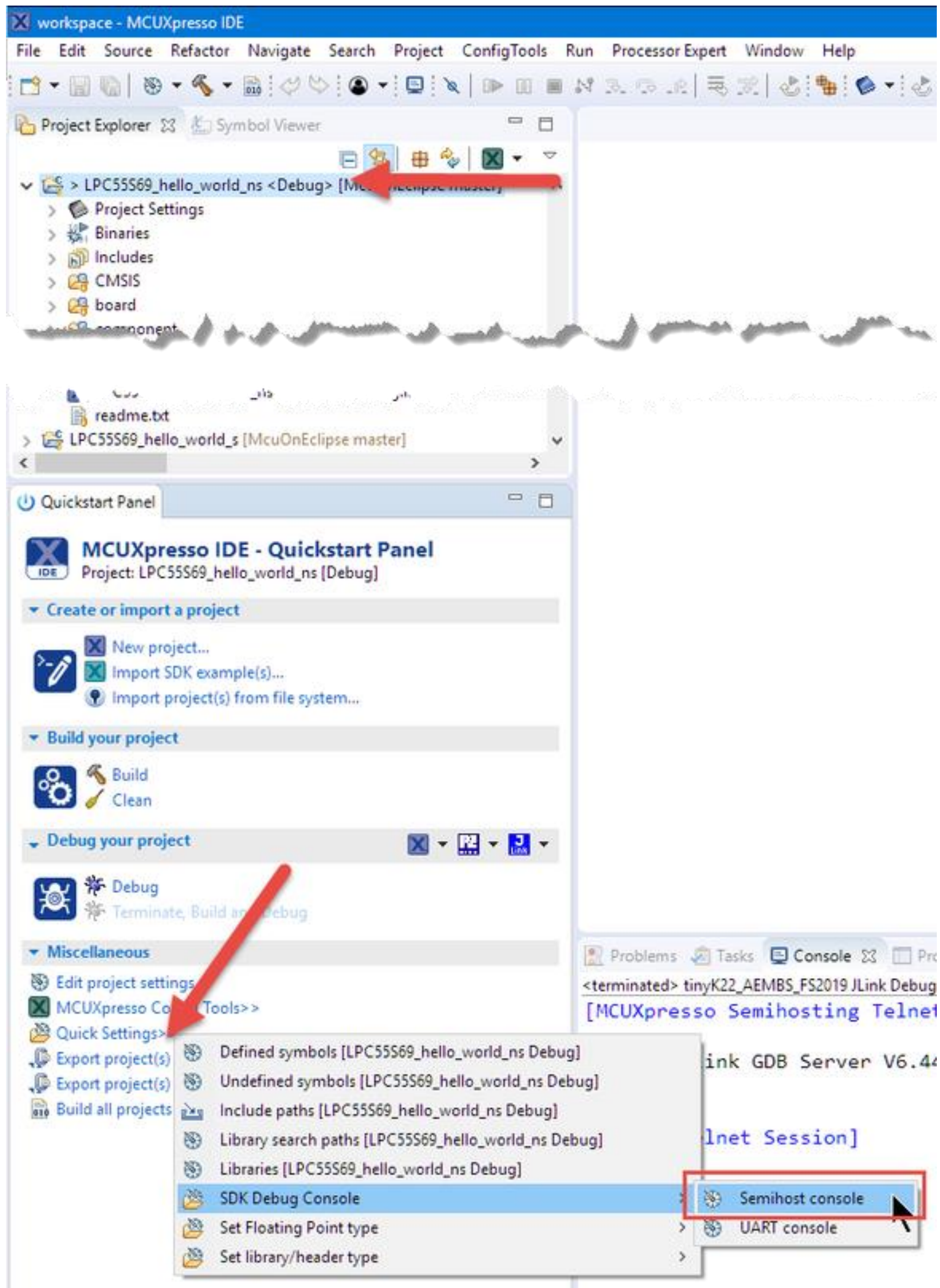


Semihost 设置

对于这两个项目，将 SDK 调试控制台设置为 Semihost Console’:

设置 Semihost Console

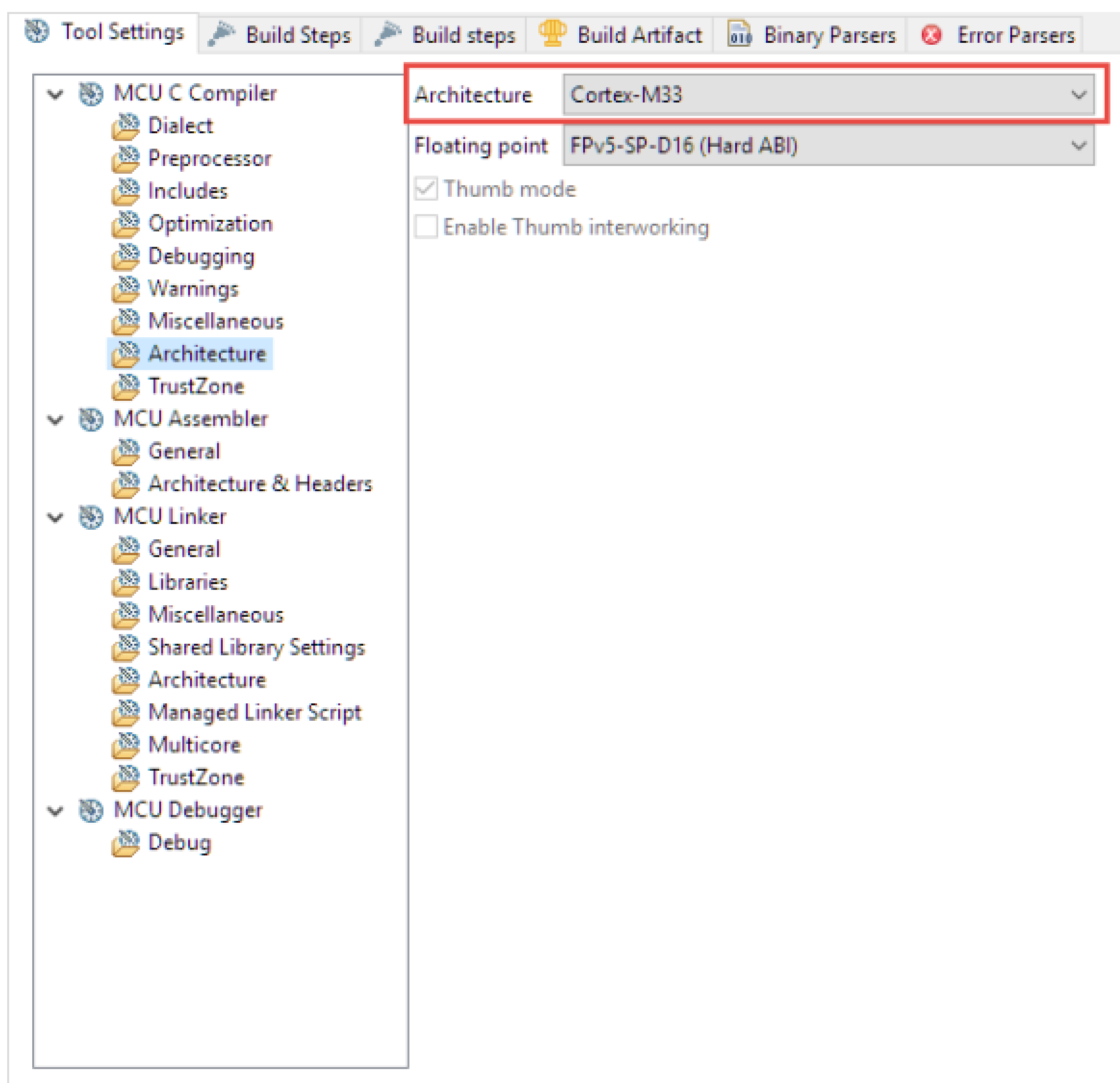
我已经为使用 semihost 控制台配置了安全和非安全项目，但也可以使用真正的 UART。



设置 Semihost Console

我已经为使用 semihost 控制台配置了安全和非安全项目，但也可以使用真正的 UART。

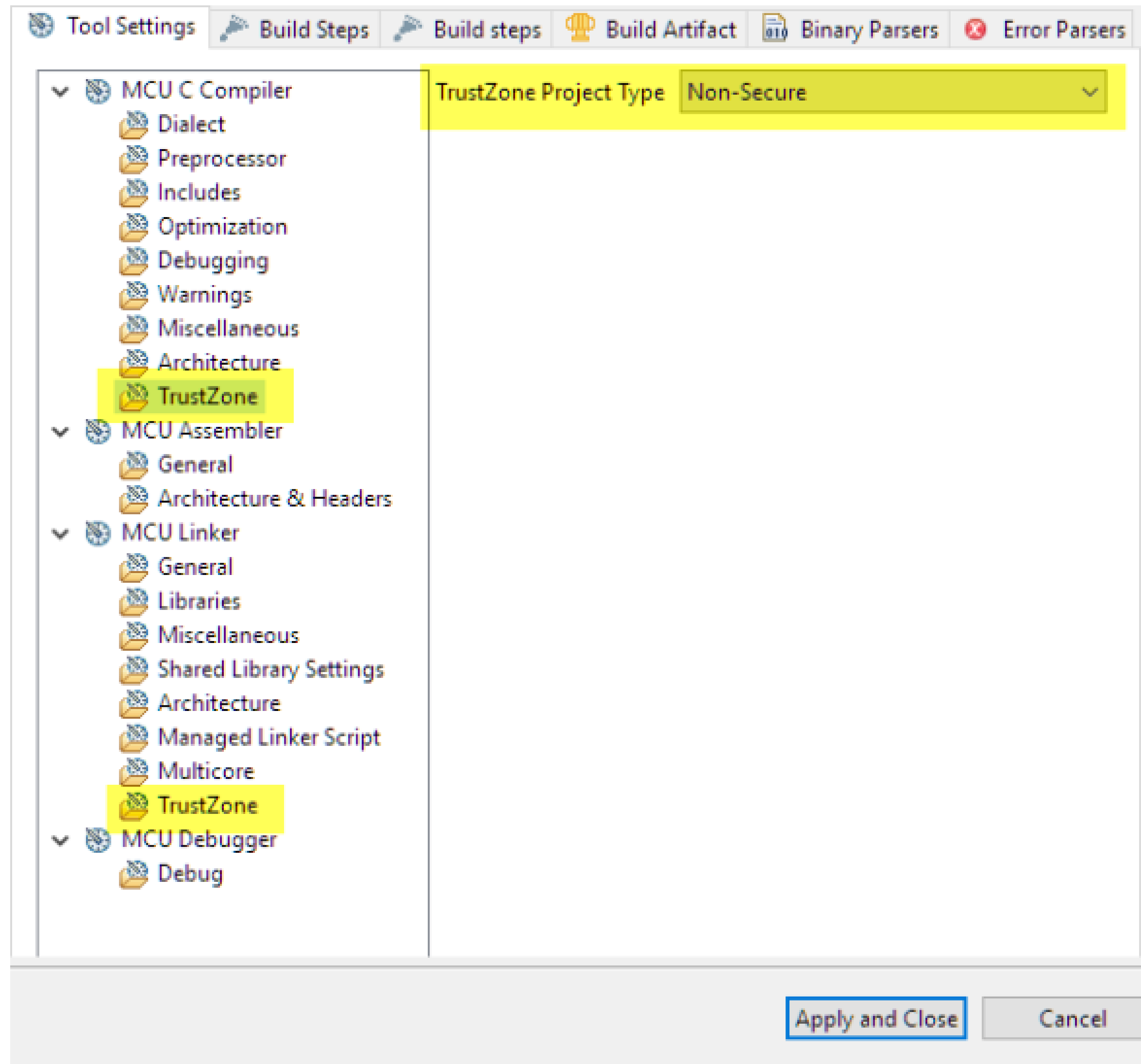
两个项目都配置为使用 Cortex-M33（这是编译器和链接器中的设置）：



M33 架构设置

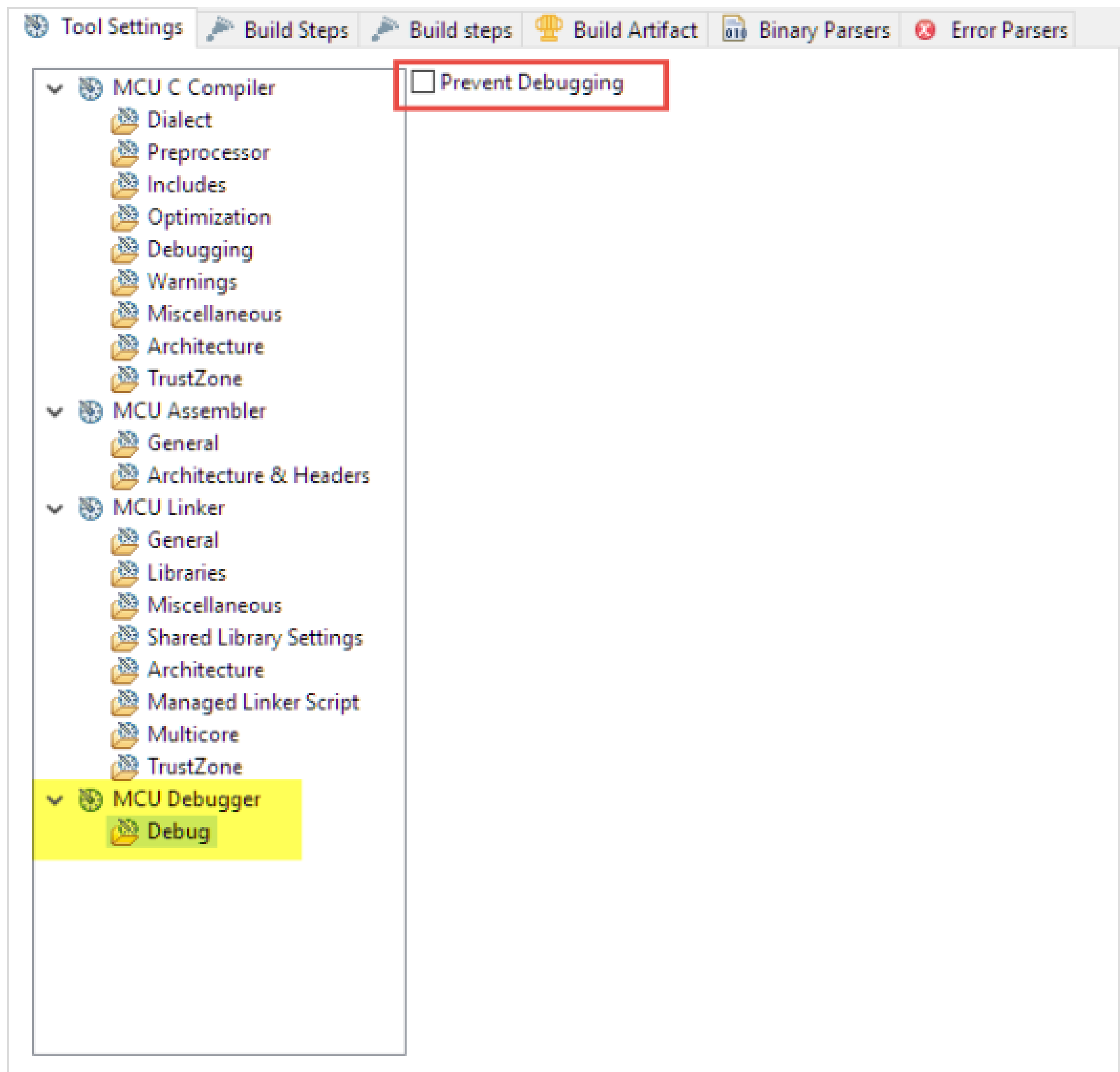
非安全侧

非安全项目在编译器和链接器设置中配置为‘Non-Secure’：



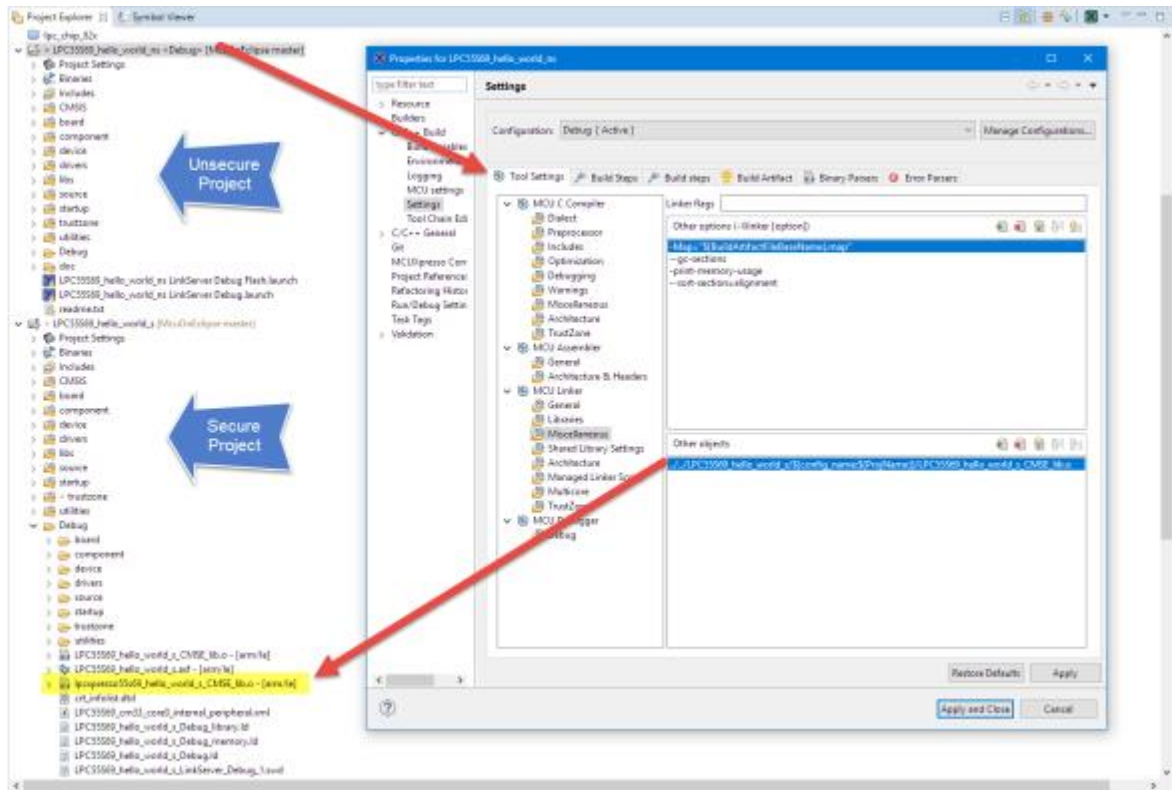
TrustZone 项目设置

有一个阻止调试的设置：



Prevent Debugging

非安全应用链接着安全应用中的一部分对象文件：



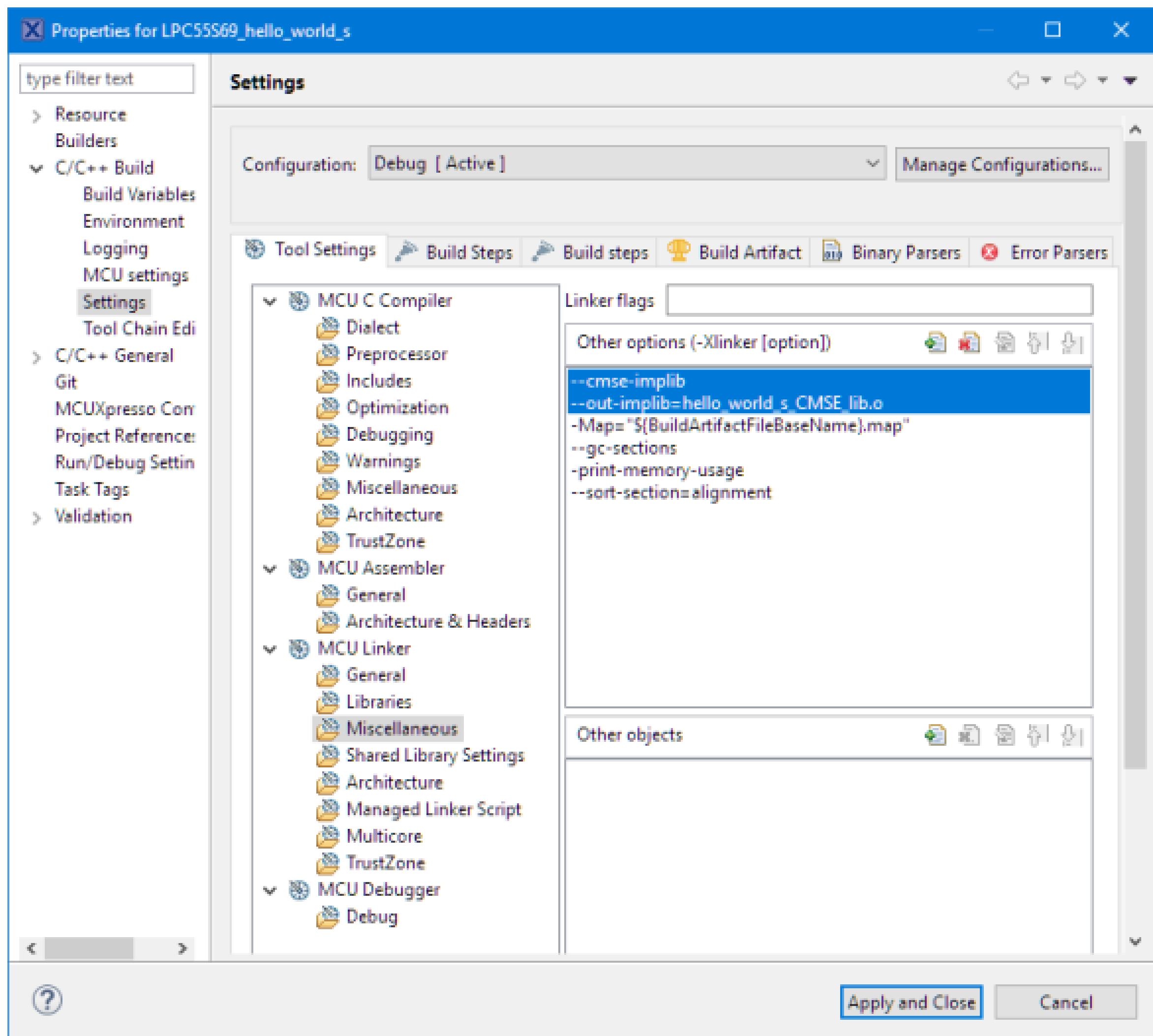
链接 CMSE Lib 对象文件

这意味着必须首先编译“安全”项目。

这是用于‘secure gateway library’的，它是使用 **–cmse-implib** 和 **–out-implib** 链接指令来建立安全项目的

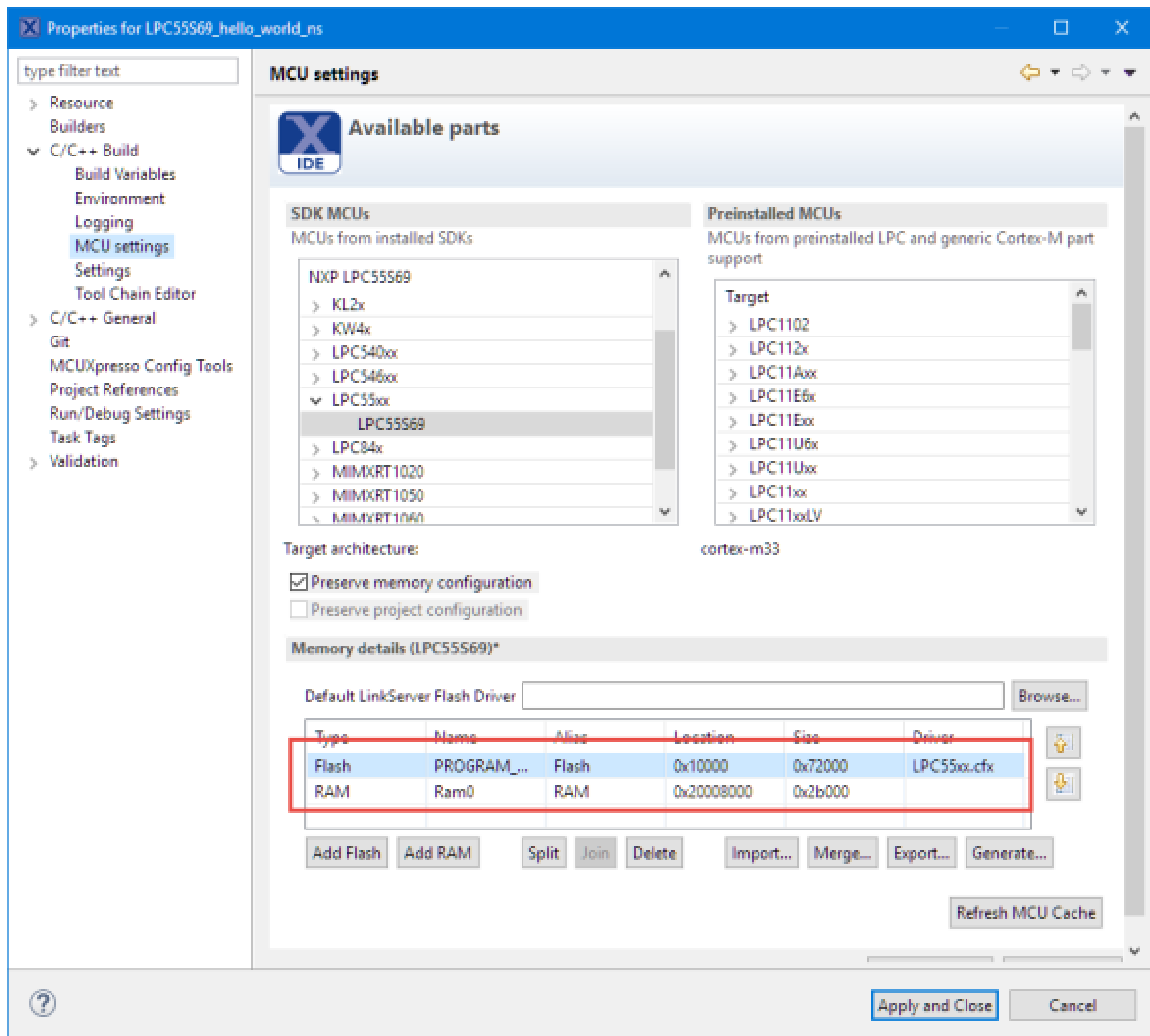
来自 https://sourceware.org/binutils/docs/ld/ARM.html:

‘–cmse-implib’选项要求由“－out implib”和“–inimplib”选项指定的导入库是安全网关导入库，适合根据 ARMV8-M 安全扩展将非安全可执行文件与安全代码链接。



Secure gateway library 链接指令

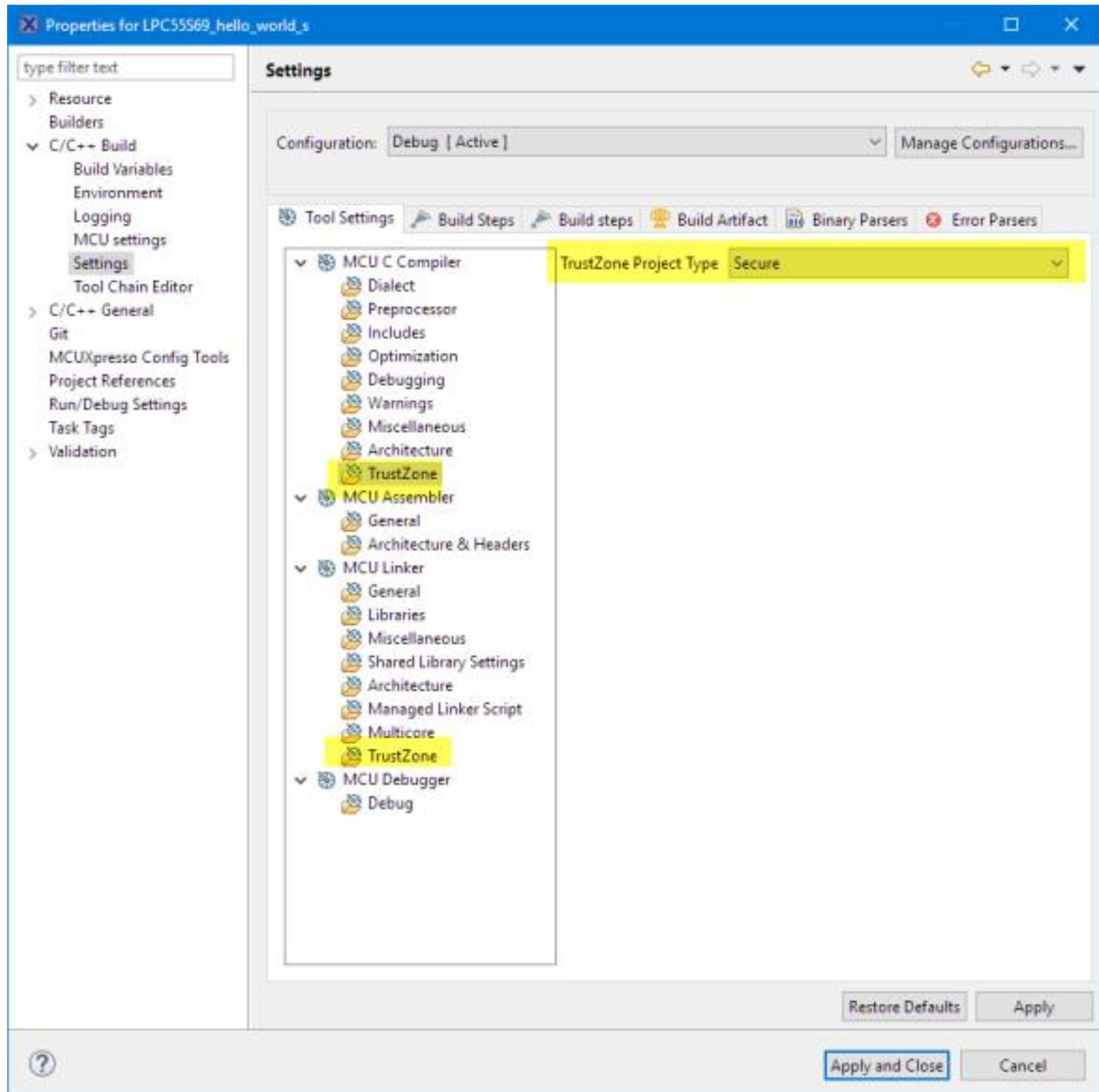
‘hello_world_ns’程序链接到地址 0x10000：向量表和代码放在以下地址:



非安全内存设置

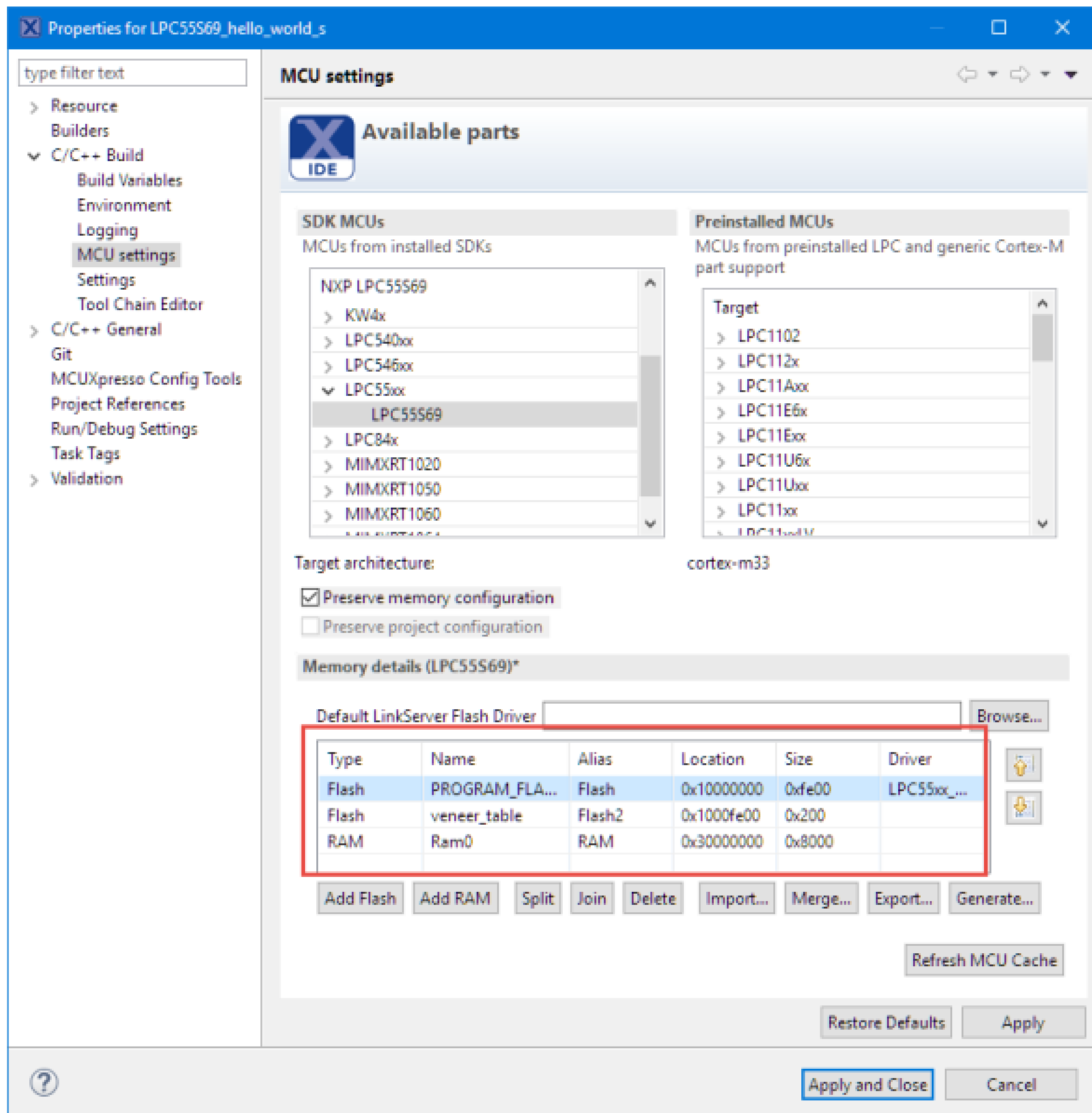
安全应用程序

在安全的一方，将 TrustZone 的编译器和链接器设置设置为‘secure’:



安全连接和编译器设置

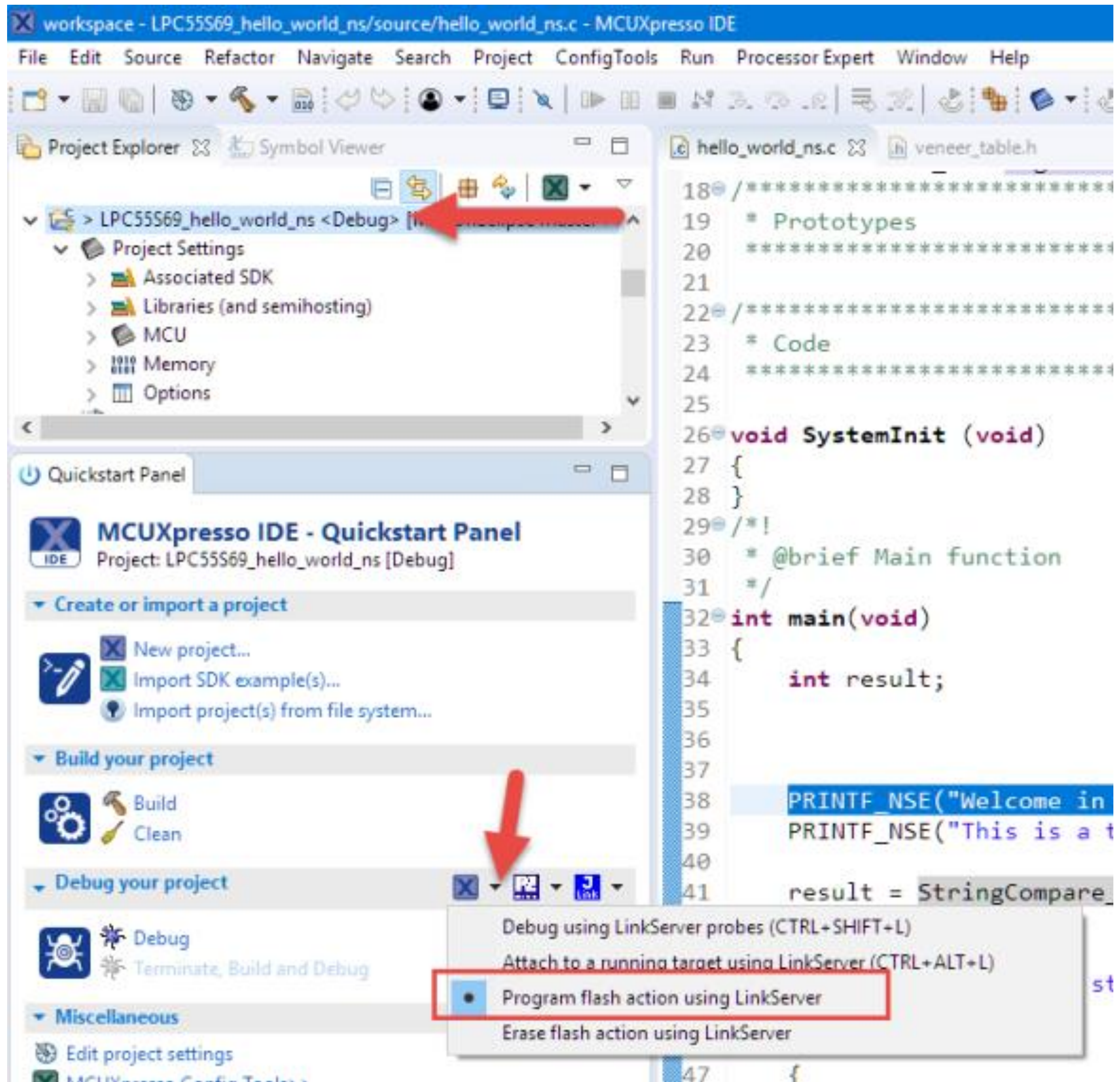
程序和向量表加载在 0x1000'0000，而“veneer”表加载在 0x1000'fe00。稍后再详细介绍…



安全内存分配

调试

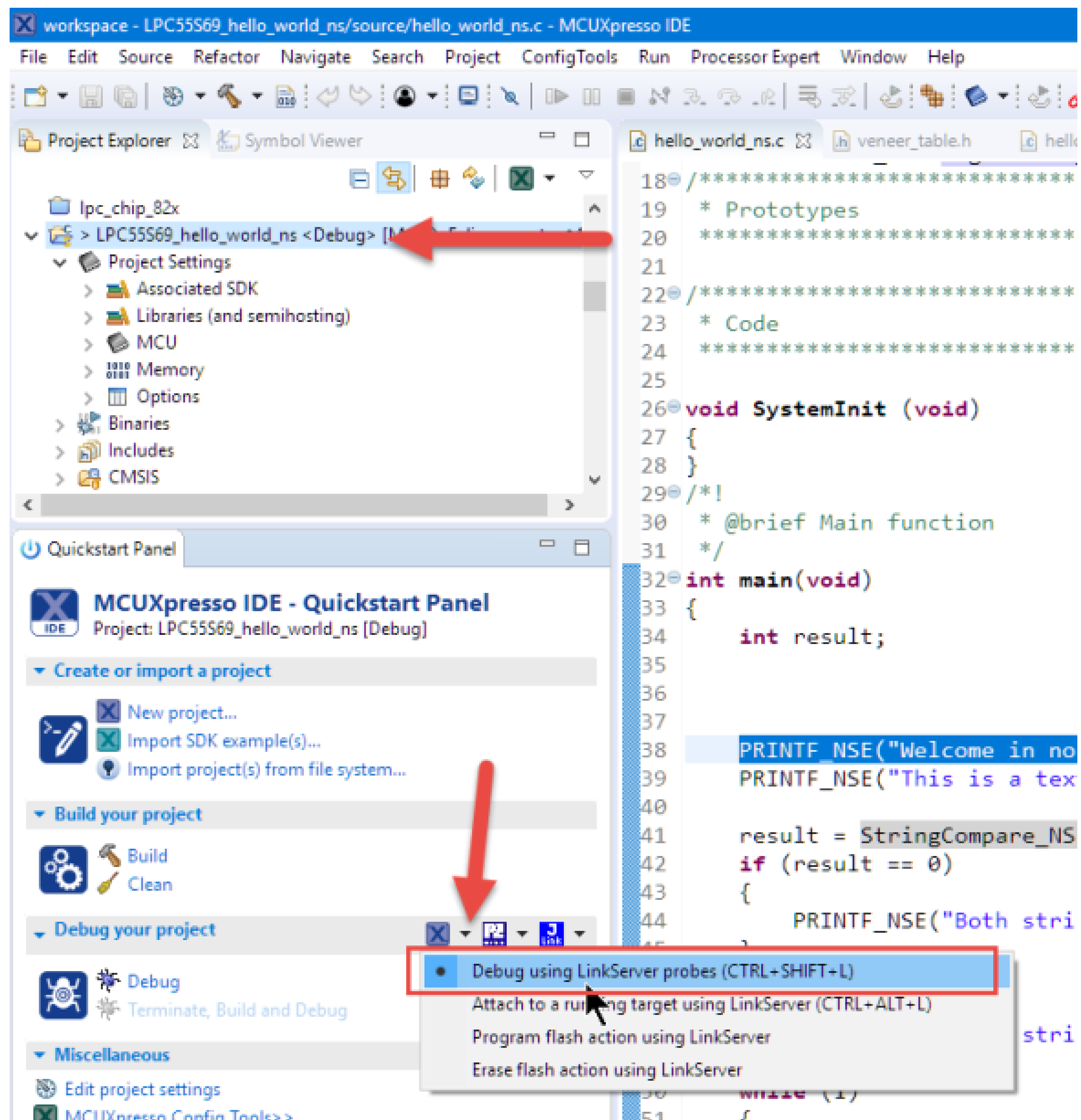
非安全应用能够被烧录到像这样的设备:



到闪存的程序

这基本上就好像新的(非安全的) 应用程序的烧录是使用引导加载程序或类似的方式来更新应用程序一样。

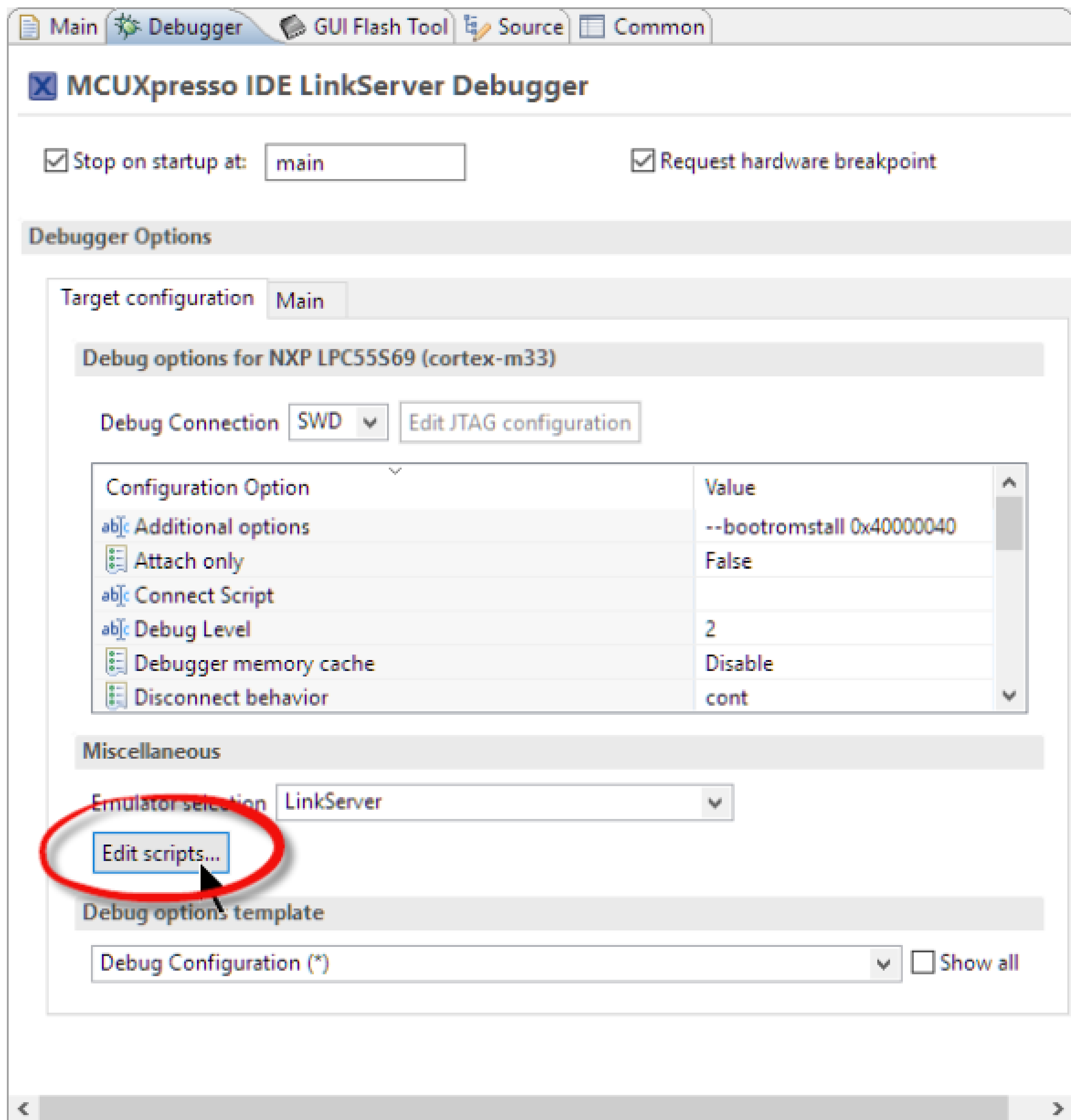
为了能够从安全应用程序中调试第二个（非安全的），我必须在调试器中为它加载符号。现在可以像往常一样调试安全的：



调试安全应用

为了在调试安全应用程序代码时调试非安全应用程序代码，我必须将符号添加到调试器中。我可以通过编辑 debug/launch 配置来做到这一点。双击.launch 文件或使用 Run>Debug Configurations 打开调试配置，然后使用“调试器”选项卡中的‘Edit

Scripts’：

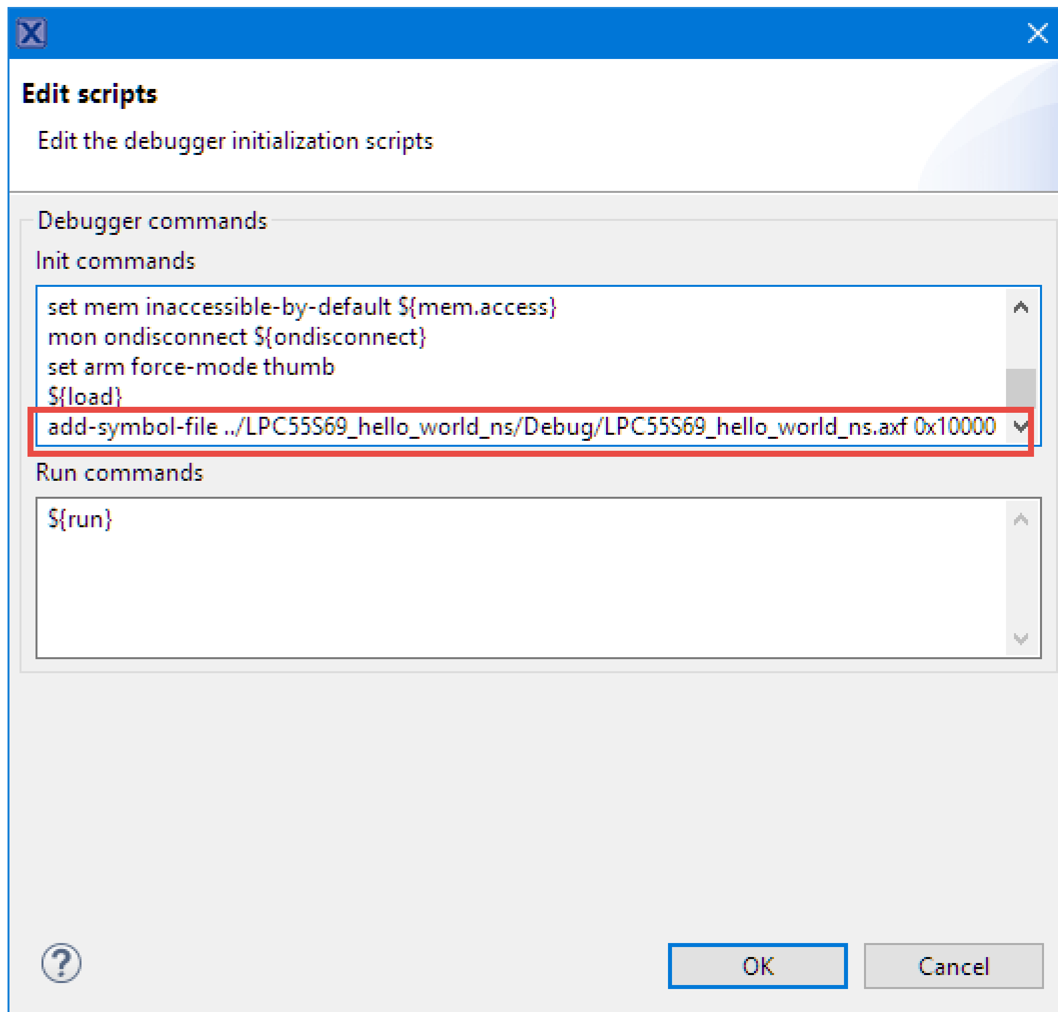


Edit Scripts

添加以下内容以使用 add-symbol-file gdb 命令加载其他项目的符号。根据需要调整路径，我将另一个项目放在同一目录层。

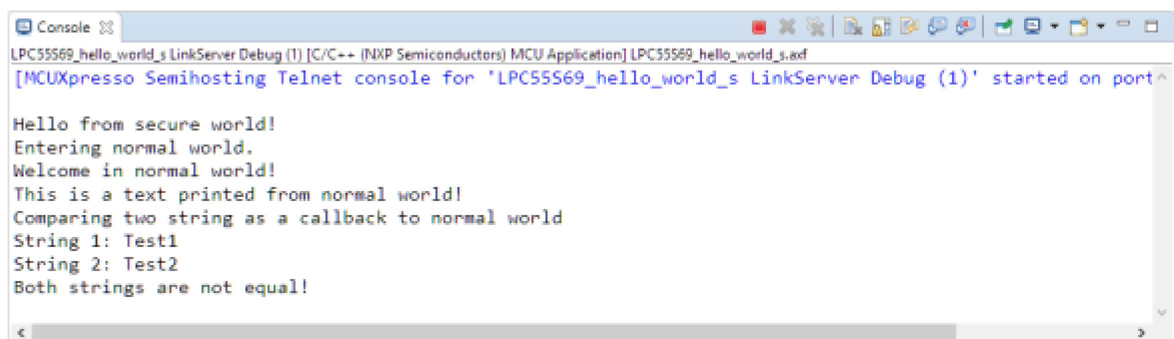
```
add-symbol-file ../LPC55S69_hello_world_ns/Debug/LPC55S69_hello_world_ns.axf 0x10000
```

告诉调试器该应用程序的符号加载在地址 0x10000 处。在${load}命令之后插入:



加载后添加符号

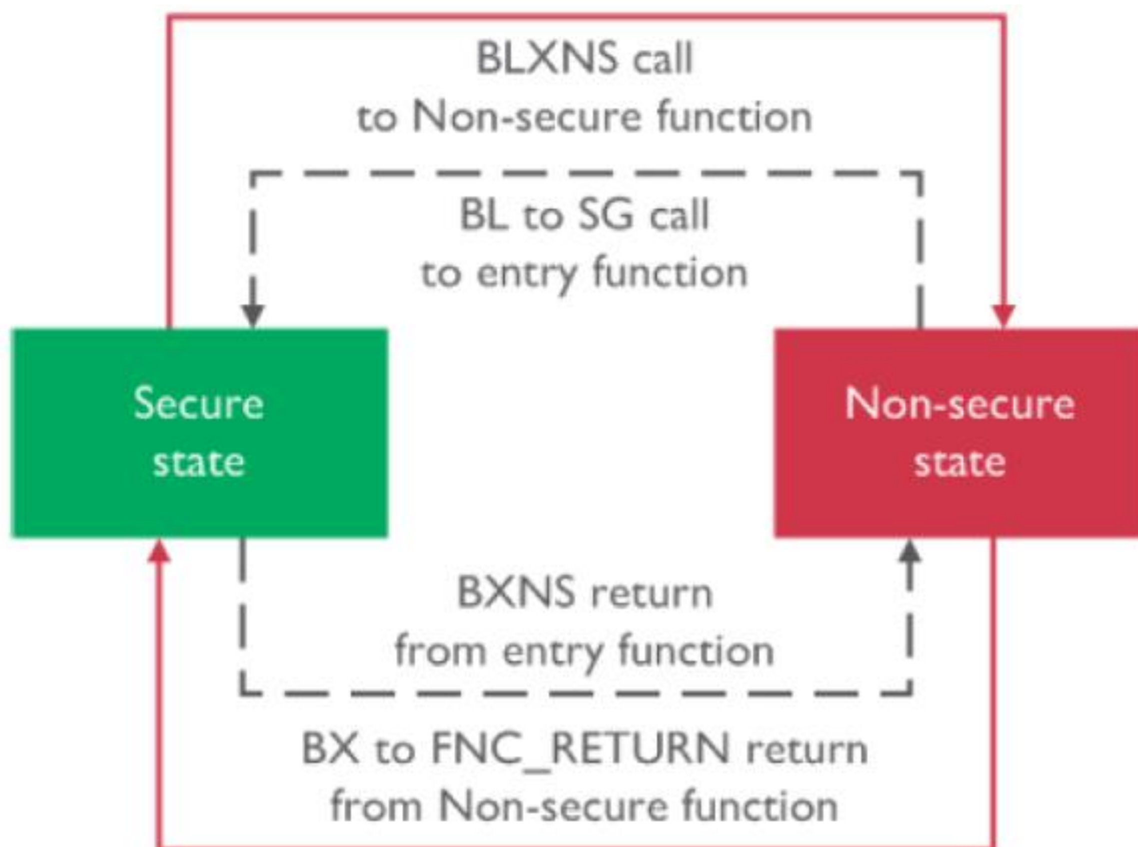
运行应用程序会产生以下输出：



控制台输出

安全状态转换

ARMv8-M 体系结构增加了安全状态之间的转换指令。例如，BLXNX 指令用于从安全世界调用非安全函数：



安全状态转换（来源: ARM，用于 ARMv8-M 的 Trustzone 技术）

从安全世界调用非安全函数

安全应用程序的 main() 函数如下.它可能是引导加载程序的基础，引导加载程序跳转到地址 0x1'0000 处的非安全加载应用程序：

:

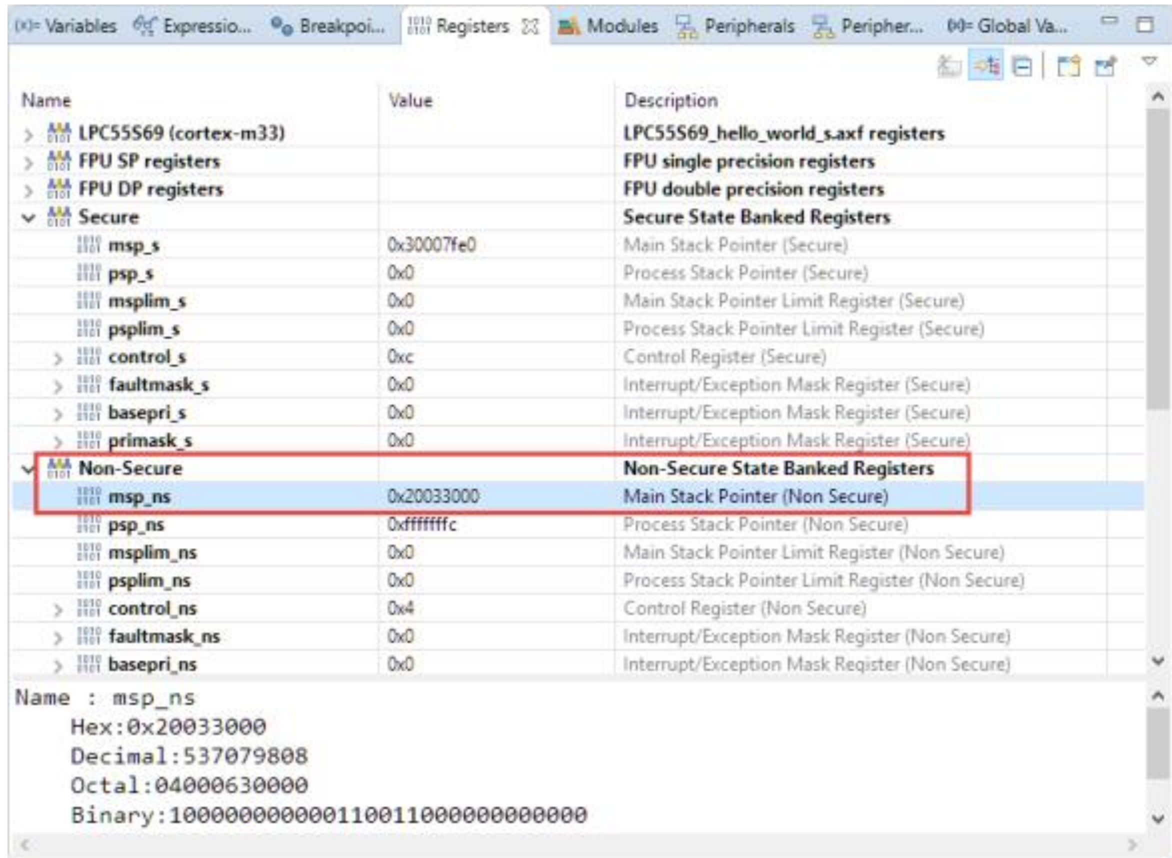
```
1  #define NON_SECURE_START          0x00010000
2
3  /* typedef for non-secure callback functions */
4  typedef void (*funcptr_ns) (void)
5  __attribute__((cmse_nonsecure_call));
6
7  int main(void)
8  {
9      funcptr_ns ResetHandler_ns;
10
11     /* Init board hardware. */
12     /* attach main clock divide to FLEXCOMM0 (debug console)
13 */
14     CLOCK_AttachClk(BOARD_DEBUG_UART_CLK_ATTACH);
15
```

```
16      BOARD_InitPins();
17      BOARD_BootClockFROHF96M();
18      BOARD_InitDebugConsole();
19
20      PRINTF("Hello from secure world!\r\n");
21
22      /* Set non-secure main stack (MSP_NS) */
23      __TZ_set_MSP_NS(*((uint32_t *)(NON_SECURE_START)));
24
25      /* Set non-secure vector table */
26      SCB_NS-&amp;gt;VTOR = NON_SECURE_START;
27
28      /* Get non-secure reset handler */
29      ResetHandler_ns = (funcptr_ns)(*((uint32_t
30  *)((NON_SECURE_START) + 4U)));
31
32      /* Call non-secure application */
33      PRINTF("Entering normal world.\r\n");
34      /* Jump to normal world */
35      ResetHandler_ns();
36      while (1)
37      {
            /* This point should never be reached */
        }
    }
```

这一行:

```
__TZ_set_MSP_NS(*((uint32_t *)(NON_SECURE_START)));
```

加载非安全的 msp（主堆栈指针）。调试器很好地显示了“banked”的安全寄存器和非安全寄存器：



非安全的 MSP

以下是从安全世界中调用非安全世界的操作:

```
ResetHandler_ns();
```

这是一个 cmse_nonsecure_call 属性的函数指针：：

```
/* typedef for non-secure callback functions */
1 typedef void (*funcptr_ns) (void)
2 __attribute__((cmse_nonsecure_call));
```

非安全函数只能使用函数指针从安全世界调用，因此将安全世界与非安全世界分开。在这个函数调用之后，执行了几个汇编指令。它清除函数地址的 LSB，并清除 FPU 单精度寄存器或任何可能包含“机密”信息的寄存器。

最后，它调用库函数 __gnu_cmse_nonsecure_call:

```
Outline   Disassembly   Enter location here
10000718:   ldr     r3, [r3, #0]
1000071a:   str     r3, [r7, #4]
77            PRINTF("Entering normal world.\r\n");
1000071c:   ldr     r0, [pc, #100]  ; (0x10000784 <main+164>)
1000071e:   bl      0x10001f70 <printf>
79            ResetHandler_ns();
10000722:   ldr     r3, [r7, #4]
10000724:   mov     r4, r3
10000726:   lsrs    r4, r4, #1
10000728:   lsls    r4, r4, #1
1000072a:   mov     r0, r4
1000072c:   mov     r1, r4
1000072e:   mov     r2, r4
10000730:   mov     r3, r4
10000732:   vldr    s0, [pc, #84]   ; 0x10000788 <main+168>
10000736:   vldr    s1, [pc, #80]   ; 0x10000788 <main+168>
1000073a:   vldr    s2, [pc, #76]   ; 0x10000788 <main+168>
1000073e:   vldr    s3, [pc, #72]   ; 0x10000788 <main+168>
10000742:   vldr    s4, [pc, #68]   ; 0x10000788 <main+168>
10000746:   vldr    s5, [pc, #64]   ; 0x10000788 <main+168>
1000074a:   vldr    s6, [pc, #60]   ; 0x10000788 <main+168>
1000074e:   vldr    s7, [pc, #56]   ; 0x10000788 <main+168>
10000752:   vldr    s8, [pc, #52]   ; 0x10000788 <main+168>
10000756:   vldr    s9, [pc, #48]   ; 0x10000788 <main+168>
1000075a:   vldr    s10, [pc, #44]  ; 0x10000788 <main+168>
1000075e:   vldr    s11, [pc, #40]  ; 0x10000788 <main+168>
10000762:   vldr    s12, [pc, #36]  ; 0x10000788 <main+168>
10000766:   vldr    s13, [pc, #32]  ; 0x10000788 <main+168>
1000076a:   vldr    s14, [pc, #28]  ; 0x10000788 <main+168>
1000076e:   vldr    s15, [pc, #24]  ; 0x10000788 <main+168>
10000772:   bl      0x10002f32 <__gnu_cmse_nonsecure_call>
80            while (1)
10000776:   b.n     0x10000776 <main+150>
```

__gnu_cmse_nonsecure_函数调用会压栈寄存器并进行更多的寄存器清除，并使用 BLXNS 汇编指令最终进入不安全的世界:

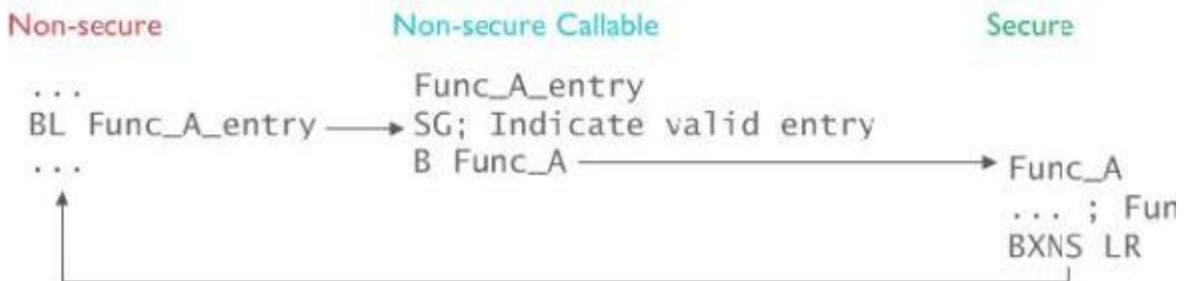
```
Outline  Disassembly  Enter location here
..........  ...
          __gnu_cmse_nonsecure_call:
10002f32:   stmdb    sp!, {r5, r6, r7, r8, r9, r10, r11, lr}
10002f36:   mov      r7, r4
10002f38:   mov      r8, r4
10002f3a:   mov      r9, r4
10002f3c:   mov      r10, r4
10002f3e:   mov      r11, r4
10002f40:   mov      r12, r4
10002f42:   vpush    {d8-d15}
10002f46:   mov.w    r5, #0
10002f4a:   vmov     d8, r5, r5
10002f4e:   vmov     s18, s19, r5, r5
10002f52:   vmov     s20, s21, r5, r5
10002f56:   vmov     s22, s23, r5, r5
10002f5a:   vmov     s24, s25, r5, r5
10002f5e:   vmov     s26, s27, r5, r5
10002f62:   vmov     s28, s29, r5, r5
10002f66:   vmov     s30, s31, r5, r5
10002f6a:   vmrs     r5, fpscr
10002f6e:   movw     r6, #65376      ; 0xff60
10002f72:   movt     r6, #4095       ; 0xfff
10002f76:   ands     r5, r6
10002f78:   vmsr     fpscr, r5
10002f7c:   msr      CPSR_f, r4
10002f80:   mov      r5, r4
10002f82:   mov      r6, r4
10002f84:   blxns    r4
10002f86:   vpop     {d8-d15}
10002f8a:   ldmia.w  sp!, {r5, r6, r7, r8, r9, r10, r11, pc}
          abort:
```

__gnu_cmse_nonsecure_call

所以有很多指令需要执行才能完成转换。

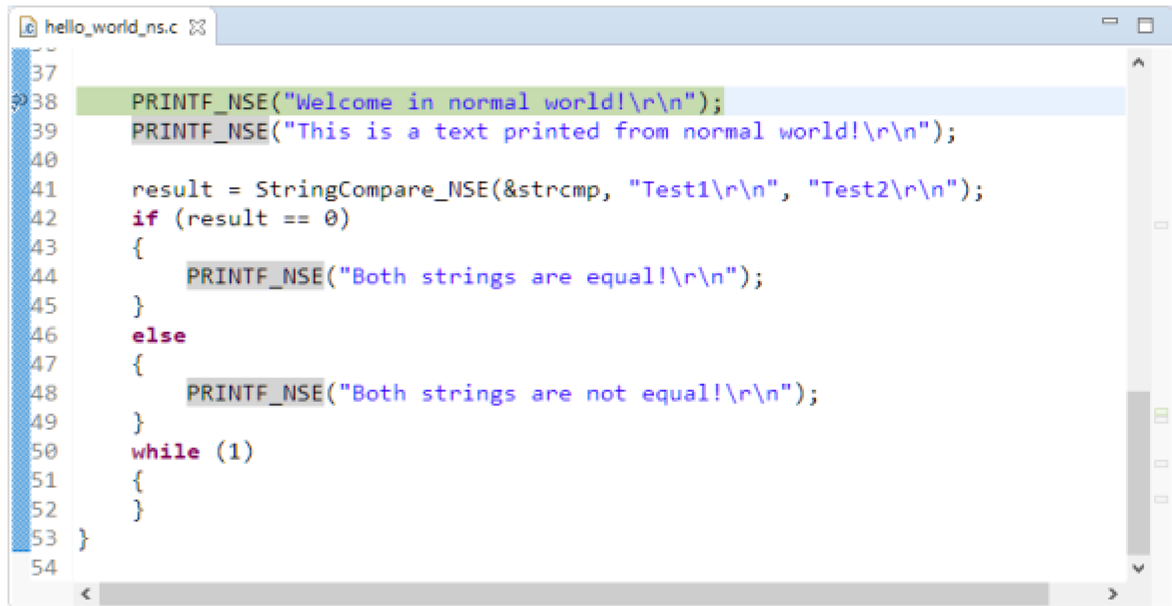
从非安全世界调用安全世界

从非安全端调用安全函数使用中间步骤（非安全可调用）：



从非安全端调用安全函数使用中间步骤（非安全可调用）： (来源: ARM，用于 ARMv8-M 的 Trustzone 技术)

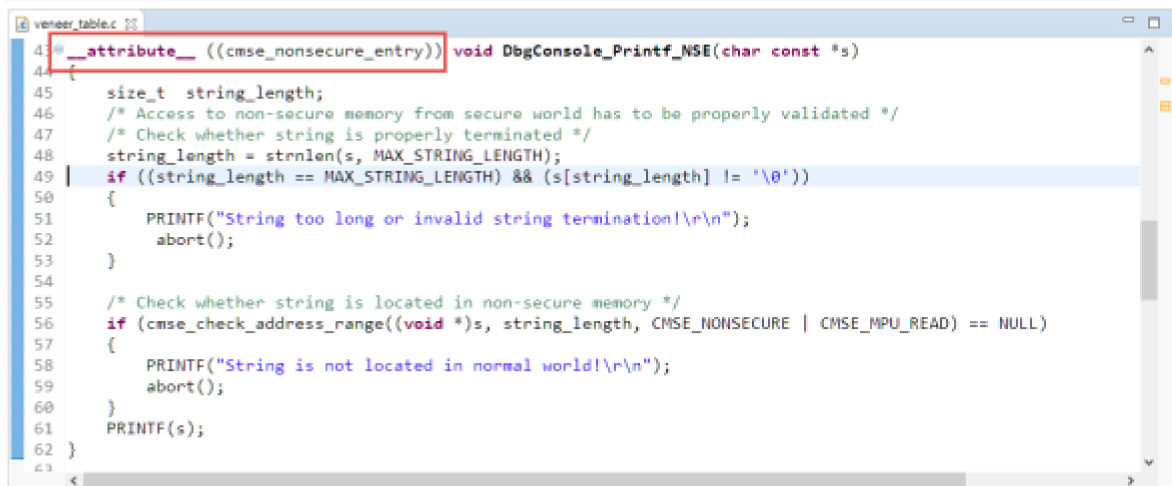
在该示例中，非安全世界正在调用位于安全世界中的 printf 函数（dbgconsole_printf_nse）：：



从非安全世界调用 printf

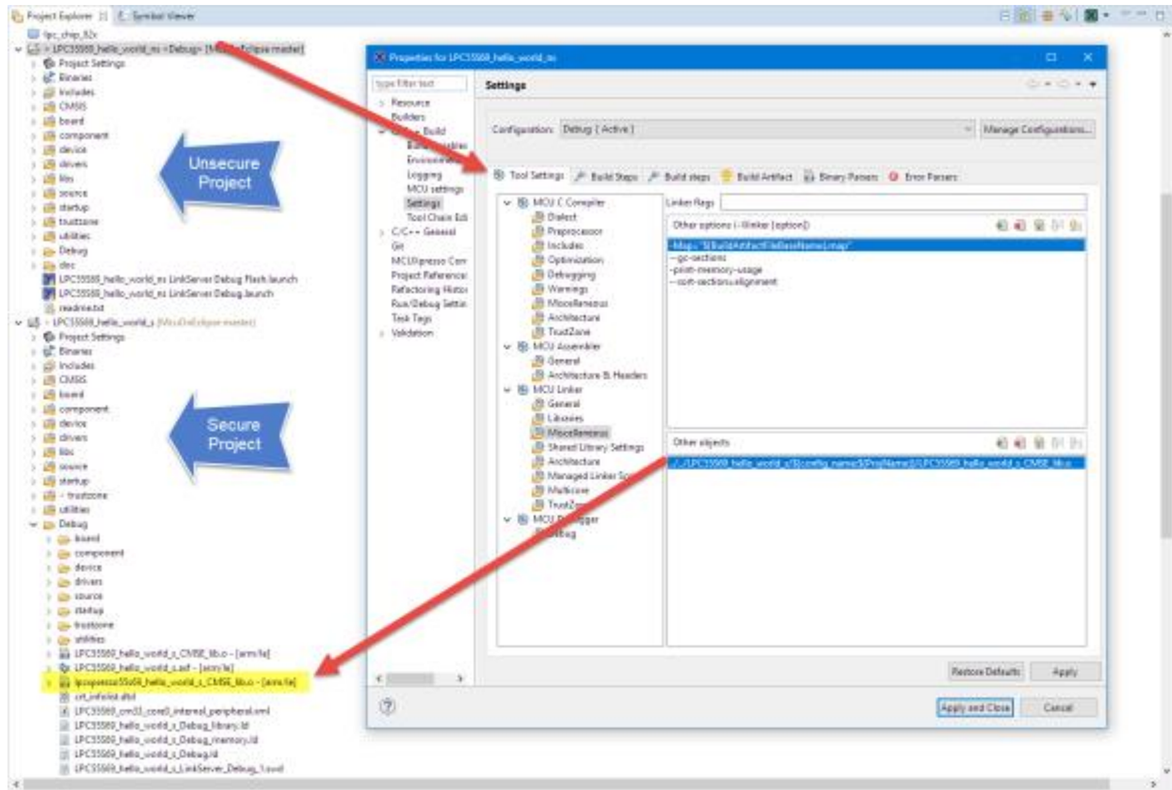
可从非安全世界调用的安全函数必须标记 **cmse_nonsecure_entry** 属性:

CMSE stands for Cortex-M (ARMv8-M) Security Extension



具有 cmse_nonsecure_entry 属性的函数

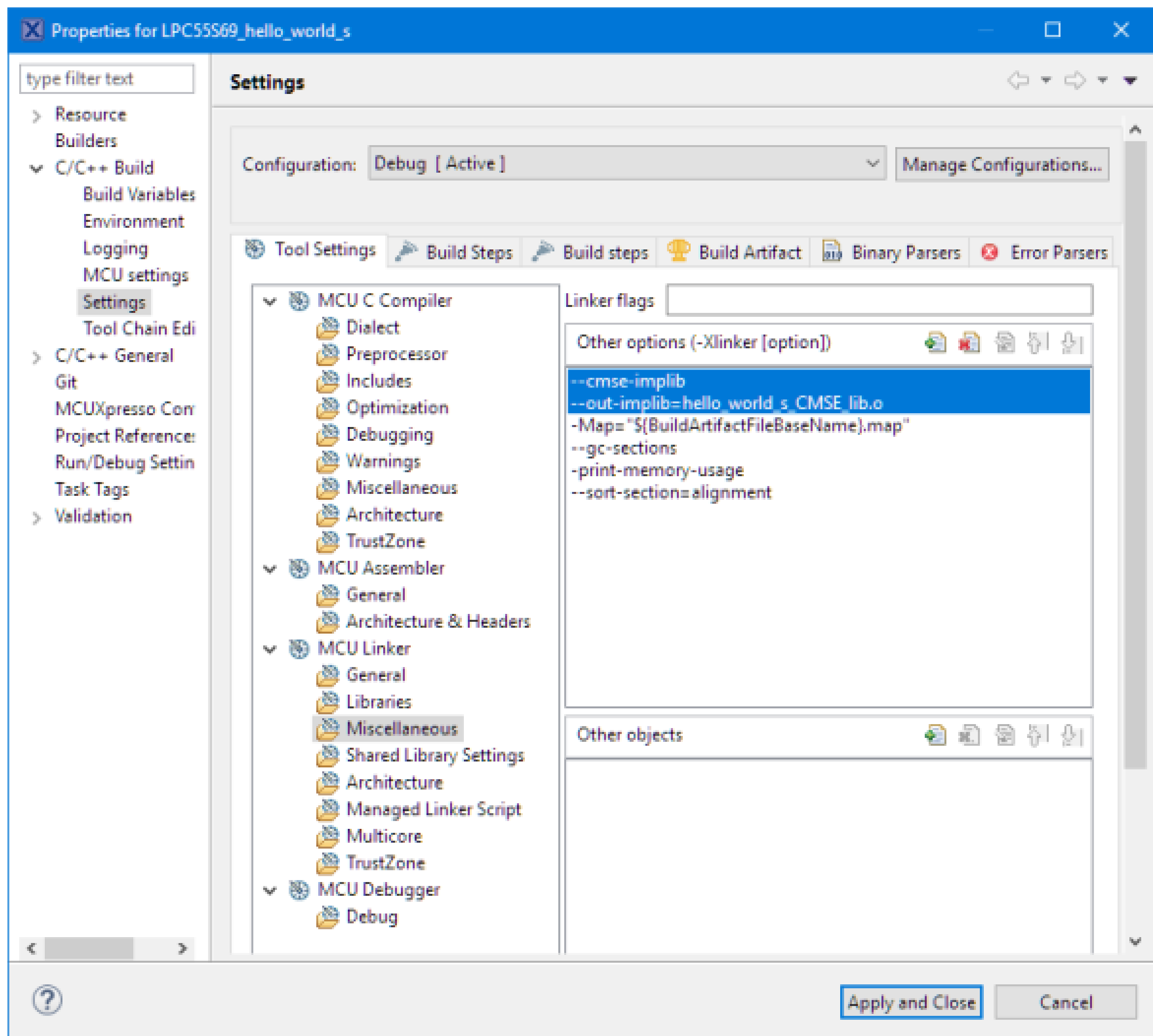
那么，不安全的世界怎么知道如何调用这个函数呢？答案是链接器准备了一切使之成为可能。为此，非安全应用程序必须将目标文件（或“库”）与“veneerr”函数链接:



链接 CMSE Lib Object 文件

这个对象文件 (或库)是用安全端的以下链接器设置的:

```
--cmse-implib --out-implib=hello_world_s_CMSE_lib.o
```



安全网关库链接器命令

因此，让我们看看从非安全世界到安全世界的代码：汇编调用‘veneer’函数：

```
Outline  Disassembly  Enter location here
000102dc:   lsls    r1, r0, #30
000102de:   movs    r1, r0
          main:
000102e0:   push    {r7, lr}
000102e2:   sub     sp, #8
000102e4:   add     r7, sp, #0
000102e6:   ldr     r0, [pc, #48]    ; (0x10318 <main+56>)
000102e8:   bl      0x10ea0 <__DbgConsole_Printf_NSE_veneer>
000102ec:   ldr     r0, [pc, #44]    ; (0x1031c <main+60>)
000102ee:   bl      0x10ea0 <__DbgConsole_Printf_NSE_veneer>
000102f2:   ldr     r2, [pc, #44]    ; (0x10320 <main+64>)
000102f4:   ldr     r1, [pc, #44]    ; (0x10324 <main+68>)
000102f6:   ldr     r0, [pc, #48]    ; (0x10328 <main+72>)
000102f8:   bl      0x10e90 <__StringCompare_NSE_veneer>
000102fc:   mov     r3, r0
000102fe:   str     r3, [r7, #4]
00010300:   ldr     r3, [r7, #4]
00010302:   cmp     r3, #0
```

调用 printf veneer

Veneer 是以一个简单的‘蹦床’函数，它为“非安全可调用”加载地址并对该地址执行 BX 操作：

```
Outline  Disassembly  Enter location here
          __DbgConsole_Printf_NSE_veneer:
00010ea0:   push    {r0}
00010ea2:   ldr     r0, [pc, #8]     ; (0x10eac <__DbgConsole_Printf_NSE_veneer+12>)
00010ea4:   mov     r12, r0
00010ea6:   pop     {r0}
00010ea8:   bx      r12
00010eaa:   nop
00010eac:   subs    r6, #137         ; 0x89
00010eae:   asrs    r0, r0, #32
00010eb0:   strh    r1, [r0, r5]
00010eb2:   cmp     r3, r10
00010eb4:   strb    r2, [r2, r1]
00010eb6:   cmp     r0, r4
00010eb8:   strh    r2, [r2, r1]
00010eba:   strh    r7, [r1, r1]
00010ebc:   movs    r2, #32
00010ebe:   movs    r5, #32
00010ec0:   movs    r0, #115         ; 0x73
```

BX 到非安全可呼叫

“secure non-callable”区域位于“secure world”中，SG 指令是第一个要执行的指令，后面跟着一个分支。

```
Outline  Disassembly
Enter location here

DbgConsole_Printf_NSE:
10003e88:   sg
10003e8c:   b.w      0x10000490 <__acle_se_DbgConsole_Printf_NSE>
10003e90:   movs     r0, r0
10003e92:   movs     r0, r0
10003e94:   movs     r0, r0
```

非安全可调用区域中的 sg 指令

SG（安全网关）指令切换到安全状态，然后 B（分支）指令切换到安全功能本身

```
Outline  Disassembly
Enter location here

__acle_se_DbgConsole_Printf_NSE:
10000490:   push     {r7, lr}
10000492:   sub      sp, #16
10000494:   add      r7, sp, #0
10000496:   str      r0, [r7, #4]
10000498:   mov.w    r1, #1024          ; 0x400
1000049c:   ldr      r0, [r7, #4]
1000049e:   bl       0x10002b3c <strnlen>
100004a2:   str      r0, [r7, #12]
100004a4:   ldr      r3, [r7, #12]
100004a6:   cmp.w    r3, #1024          ; 0x400
100004aa:   bne.n    0x100004c2 <__acle_se_DbgCon
100004ac:   ldr      r2, [r7, #4]
100004ae:   ldr      r3, [r7, #12]
100004b0:   add      r3, r2
100004b2:   ldrb     r3, [r3, #0]
```

执行安保功能

相比于从“secure world”调用不安全的一面，这个的执行速度是相当快的。在 BXNS 返回到非安全状态之前，清除所有寄存器，因为它们可能含有机密信息：

```
Outline    Disassembly    Enter location here
10000500:   vmov.f32        s3, #112         ; 0x3f800000  1.0
10000504:   vmov.f32        s4, #112         ; 0x3f800000  1.0
10000508:   vmov.f32        s5, #112         ; 0x3f800000  1.0
1000050c:   vmov.f32        s6, #112         ; 0x3f800000  1.0
10000510:   vmov.f32        s7, #112         ; 0x3f800000  1.0
10000514:   vmov.f32        s8, #112         ; 0x3f800000  1.0
10000518:   vmov.f32        s9, #112         ; 0x3f800000  1.0
1000051c:   vmov.f32        s10, #112        ; 0x3f800000  1.0
10000520:   vmov.f32        s11, #112        ; 0x3f800000  1.0
10000524:   vmov.f32        s12, #112        ; 0x3f800000  1.0
10000528:   vmov.f32        s13, #112        ; 0x3f800000  1.0
1000052c:   vmov.f32        s14, #112        ; 0x3f800000  1.0
10000530:   vmov.f32        s15, #112        ; 0x3f800000  1.0
10000534:   msr     CPSR_fs, lr
10000538:   push    {r4}
1000053a:   vmrs    r12, fpscr
1000053e:   movw    r4, #65376       ; 0xff60
10000542:   movt    r4, #4095        ; 0xfff
10000546:   and.w   r12, r12, r4
1000054a:   vmsr    fpscr, r12
1000054e:   pop     {r4}
10000550:   mov     r12, lr
10000552:   bxns    lr
10000554:   adds    r5, #124         ; 0x7c
10000556:   asrs    r0, r0, #32
10000558:   adds    r5, #176         ; 0xb0
1000055a:   asrs    r0, r0, #32
          __acle_se_StringCompare_NSE:
1000055c:   push    {r4, r7, lr}
```

在回到非安全区清除寄存器的值

SAU 设置

那么如何配置保护呢？SAU（安全属性单元）被配置为只能在安全侧完成。
这个例子使用了以下安全和非安全的代码和数据区域：

```
1 #define CODE_FLASH_START_NS          0x00010000
2 #define CODE_FLASH_SIZE_NS           0x00062000
3 #define CODE_FLASH_START_NSC         0x1000FE00
4 #define CODE_FLASH_SIZE_NSC          0x200
5 #define DATA_RAM_START_NS            0x20008000
6 #define DATA_RAM_SIZE_NS             0x0002B000
```

在这个例子中，这是在 BOARD_InitTrustZone()中配置的。以下设置为非安全闪存配置一个运行区域

```
1 /* Configure SAU region 0 - Non-secure FLASH for CODE
2 execution*/
3 /* Set SAU region number */
4 SAU->RNR = 0;
5 /* Region base address */
```

```
6  SAU->RBAR = (CODE_FLASH_START_NS & SAU_RBAR_BADDR_Msk);
7  /* Region end address */
8  SAU->RLAR = ((CODE_FLASH_START_NS + CODE_FLASH_SIZE_NS-1)
9  & SAU_RLAR_LADDR_Msk) |
10               /* Region memory attribute index */
11               ((0U >> SAU_RLAR_NSC_Pos) &amp;
   SAU_RLAR_NSC_Msk) |
                 /* Enable region */
                 ((1U >> SAU_RLAR_ENABLE_Pos) &amp;
   SAU_RLAR_ENABLE_Msk);
```

IDAU（实现定义的属性单元）是可选的，旨在提供一个默认的访问内存映射（安全的、非安全的和非安全的可调用），可以被 SAU 覆盖

总结

了解 ARMv8-M 安全扩展的细节可能需要更多的时间，还有更多的细节需要探索，比如安全的外设访问或如何保护内存区域。简而言之，它允许将设备划分为“安全的/受信任的和“不安全的/不受信任的”，并通过添加 MPU 和对外围设备的控制访问将内存映射划分为安全的、不安全的和不安全的可调用。另外还有控制调试级别的功能，以防止逆向工程的发生。

有了 NXP MCUXpresso SDK 和 IDE，再加上 LPC55S69 板，我就有了一个可以用来做实验的工作环境。我喜欢这种方法，基本上非安全应用程序需要知道它在安全环境中运行的事实，除非它想调用“secure world”的函数。我现在使用为 M33 开发的带有 FreeRTOS 端口的 LPC55XX，但我是在“非安全”领域使用它。我的目标是让实时操作系统安全运行。目前还不确定这到底会是什么样子，但如果时间允许的话，这是一个很好的用例，我想在下周进行研究。
“安全”快乐:-)

链接

- 在 GitHub 上的工程文件在这篇文章里: https://github.com/ErichStyger/mcuoneclipse/tree/master/Examples/MCUXpresso/LPC55S69-EVK
- First Steps with the LPC55S69-EVK (Dual-Core ARM Cortex-M33 with Trustzone)
- ARM TrustZone: https://developer.arm.com/ip-products/security-ip/trustzone
- ARM v8-M 架构下的 ARM TrustZone 技术： https://static.docs.arm.com/100690_0101/00/armv8_m_architecture_trustzone_technology_100690_0101_00_en.pdf
- Command Line Programming and Debugging with GDB
- ARMv8-M 安全性拓展: 要求和开发工具: http://infocenter.arm.com/help/topic/com.arm.doc.ecm0359818/ECM0359818_armv8m_security_extensions_reqs_on_dev_tools_1_0.pdf
- ARMv8-M TrustZone 的背景概念: https://www.lobaro.com/using-the-armv8-m-trustzone-with-gcc/

- - -