RA6M4

CPK-RA6M4 评估板入门

要点

CPK-RA6M4 是用于 RA6M4 单片机的评估板套件。该套件可通过灵活配置软件包 (FSP) 和 e² studio IDE，对 RA6M4 MCU 群组的特性进行无缝评估，并对嵌入系统应用程序进行开发。本文档是《瑞萨 RA MCU 基础知识》的配套文档，旨在将该指南中有关硬件操作的部分在 CPK-RA6M4 评估板上进行实现。

在使用本文档之前，推荐您先学习《瑞萨 RA MCU 基础知识》，以了解更多关于 RA MCU 的基础知识以及其他相关知识，这样有助于您在本文所述的硬件实操中更快上手。

开发环境：

e² studio：2021-04 版

FSP：v3.1.0

目录

1. 首次使用瑞萨 CPK -RA6M4 评估板

本章介绍首次使用瑞萨 CPK-RA6M4 评估板时要进行的设置。

1.1 导入 BSP（板级支持包）文件

本节介绍如何在 e² studio 上生成基于 FSP v3.1.0 的 BSP 文件。

Setp 1. 文件准备

请准备好以下文件（随本文档一同下载，文件名为 BSP File_CHN_3.1.0.rar）：

- .xml 文件:

“Renesas##BSP##Board##ra6m4_cpk####3.1.0.xml”
“Renesas##BSP##Board##ra6m4_cpk####3.1.0##configuration.xml”

- .pack 文件:

“Renesas.RA_board_ra6m4_cpk.3.1.0.pack”

Setp 2. e² studio 准备

打开 e²studio，依照以下顺序找到需要放入文件的“modules”和“pack”文件夹：
“帮助” - “关于 e² studio” - “安装细节” - “e² studio support area”右侧链接 - “internal” - “projecgen” - “RA”。

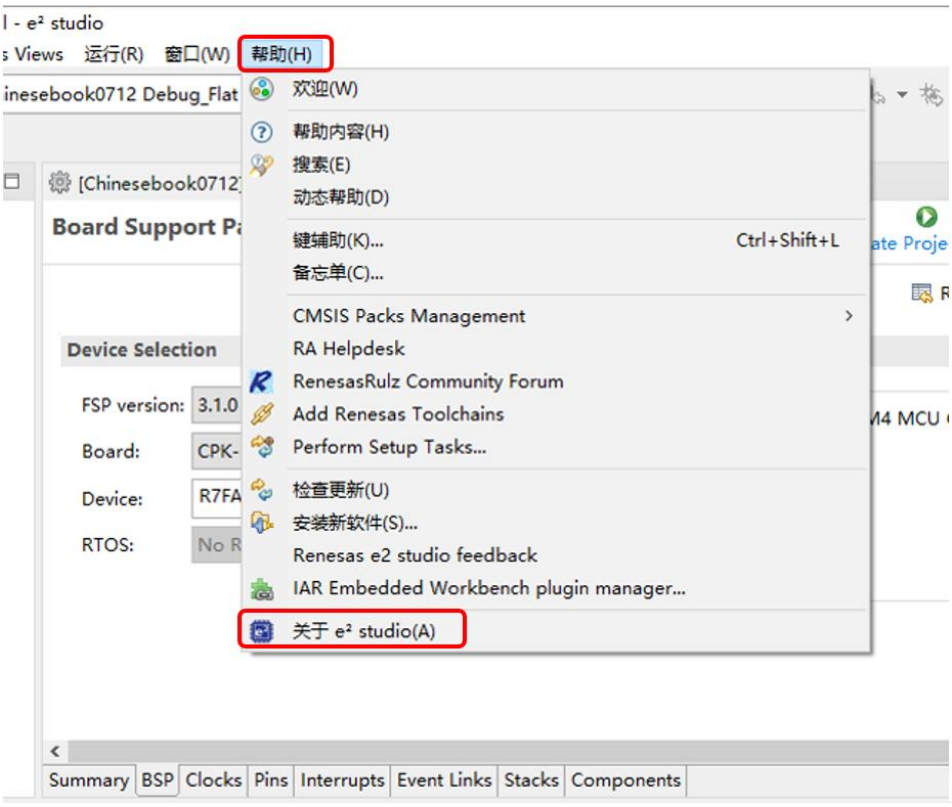


图 1-1 打开 e² studio 在“帮助”出下拉找到“关于 e² studio”

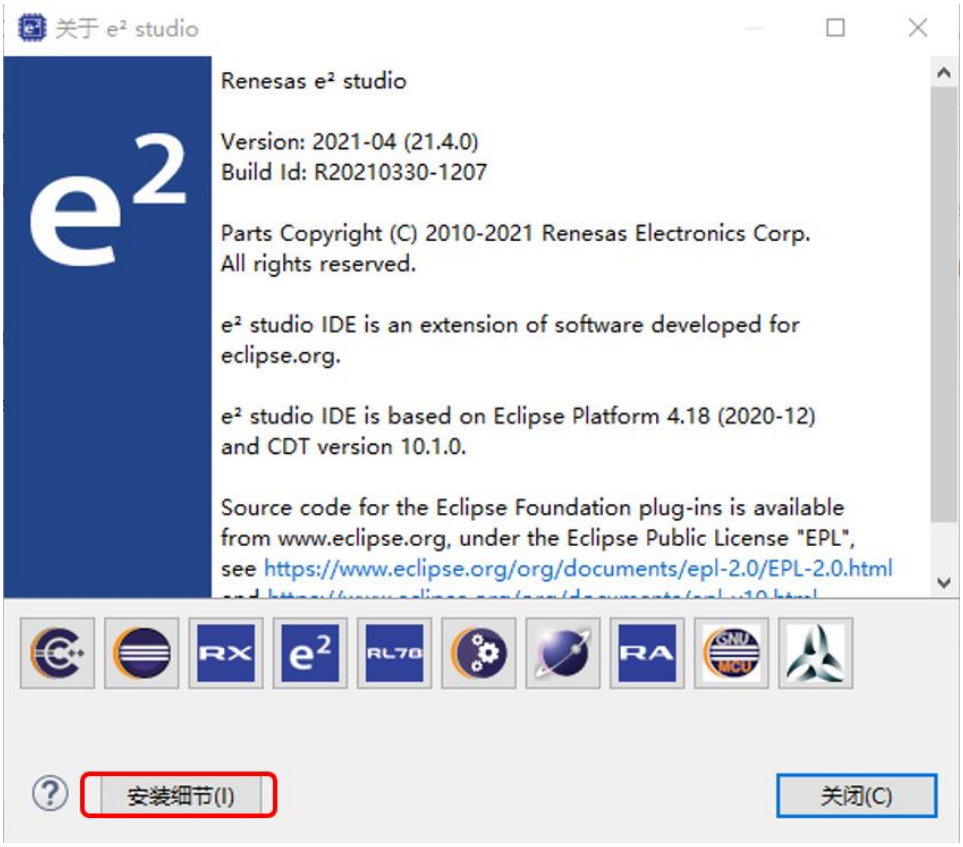


图 1-2 打开 e² studio 的安装细节

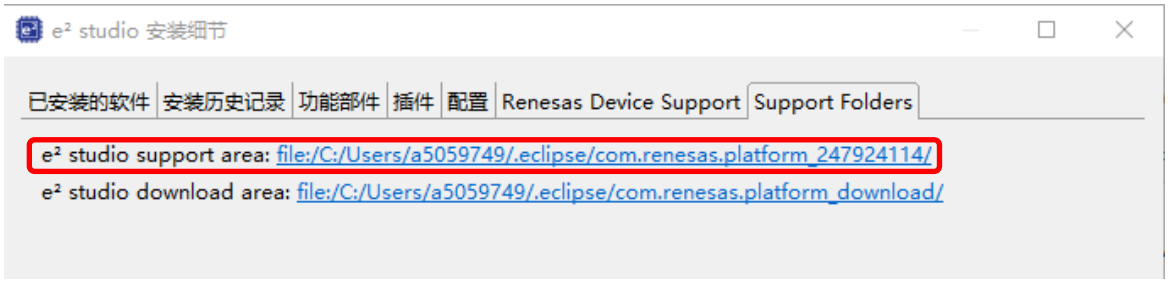


图 1-3 点击“e2 studio support area”右侧链接

Setp 3.

将两个.xml 文件拷贝到 modules 文件夹中，将.pack 文件拷贝到 pack 文件夹中。

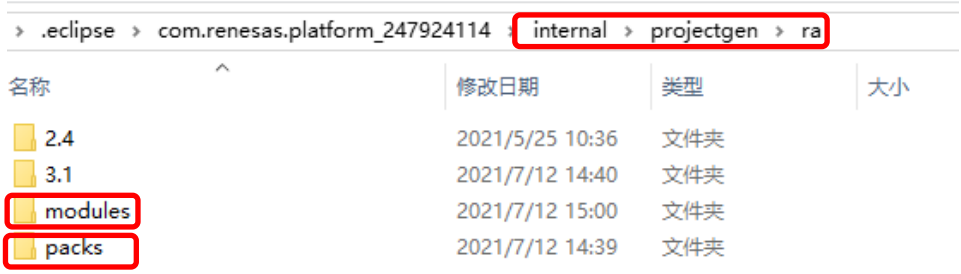


图 1-4 modulus 与 packs 文件夹位置

2. 下载并测试示例

本章内容基于《瑞萨 RA MCU 基础知识》中的章节 **7.2 下载并测试示例** 所作。

首先，从操作系统的“Start”（开始）菜单打开 e² studio。如果系统提示您输入工作区的位置可以从网站（**https://www.renesas.com/ra-book**）下载项目。下载后，将其导入工作区。

在我们将程序下载到评估板并运行之前，需要创建一个调试配置。单击“*Debug*”（调试）符号 旁边的小箭头，然后从下拉列表框中选择“*Debug Configurations*”（调试配置）。

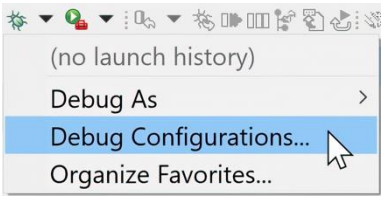


图 2-1：要开始调试，请从下拉列表框中选择“Debug Configurations”（调试配置）

在出现的窗口中，突出显示左侧树形视图中“*Renesas GDB Hardware Debugging*”（Renesas GDB 硬件调试）下的 *MyRaProject Debug_Flat*。如果您为此项目使用了其他名称，请选择您使用的名称。

选择项目后，将为“Debug Configurations”（调试配置）打开一个新屏幕，其中显示相关的所有选项（请参见图 *2-2*）。由于仅用于测试目的，因此无需在此处进行任何更改，只需单击底部的“*Debug*”（调试），调试器随即启动。显示“*Confirm Perspective Switch*”（确认透视图切换）对话框后，选择“*YES*”（是）。

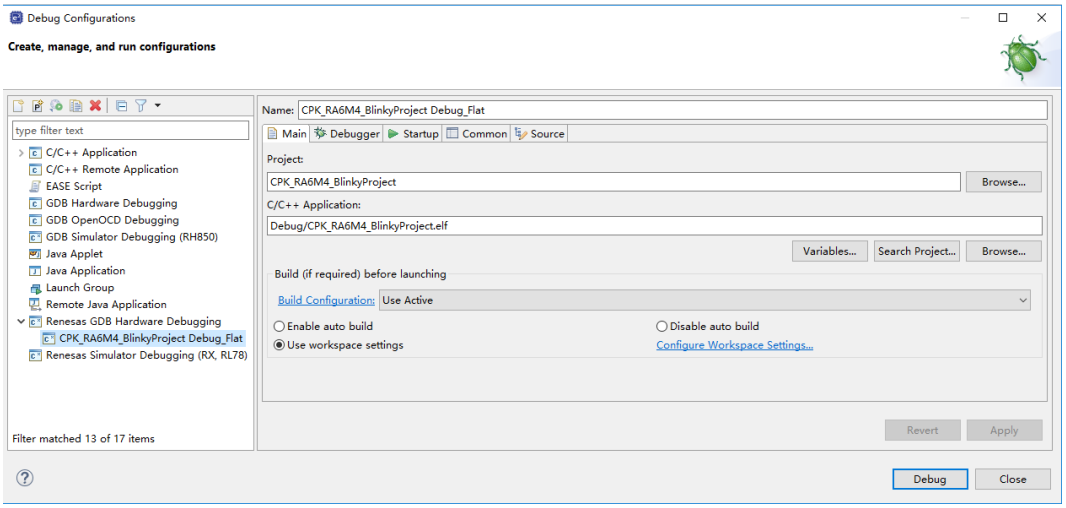


图 2-2：在“Renesas GDB Hardware Debugging”（Renesas GDB 硬件调试）下选择项目后，无需在出现的窗口中进行任何更改。

可能会出现另一个名为“*J-Link® Firmware update*”（J-Link® 固件更新）的对话框，要求为板上调试器安装新的固件版本。强烈建议单击“*Yes*”（是）来允许更新。

根据 Windows 工作站的安全设置，可能会出现一个显示安全警报的对话框窗口，通知您“Windows Defender Firewall has blocked some features of E2 Server GDB on all public and private networks.”（Windows Defender 防火墙已阻止所有公用和专用网络上的 E2 Server GDB 的某些功能。）要继续操作，请

允许 E2 Server GDB 在专用网络上通信。为此，请选中相应的复选框，然后单击“*Allow access*”（允许访问）。

打开“Debug”（调试）透视图后（请参见图 2-3），调试器会将程序计数器设置为程序的入口点，即复位处理程序。单击“*Resume*”（恢复）按钮 ，程序将运行到 `main()` 函数内 `hal_entry()` 调用行中的下一个停止处。再次单击“*Resume*”（恢复），程序将继续执行，评估板上红色 LED3（用户 LED）开始闪烁。

最后一步是单击“*Disconnect*”（断开连接）按钮 ，断开调试器与开发板的连接，以停止程序的执行。现在，您已确定 e² studio 的安装可与评估板配合使用，接下来为 RA 系列单片机编写第一个程序。这将是下一章的主题。

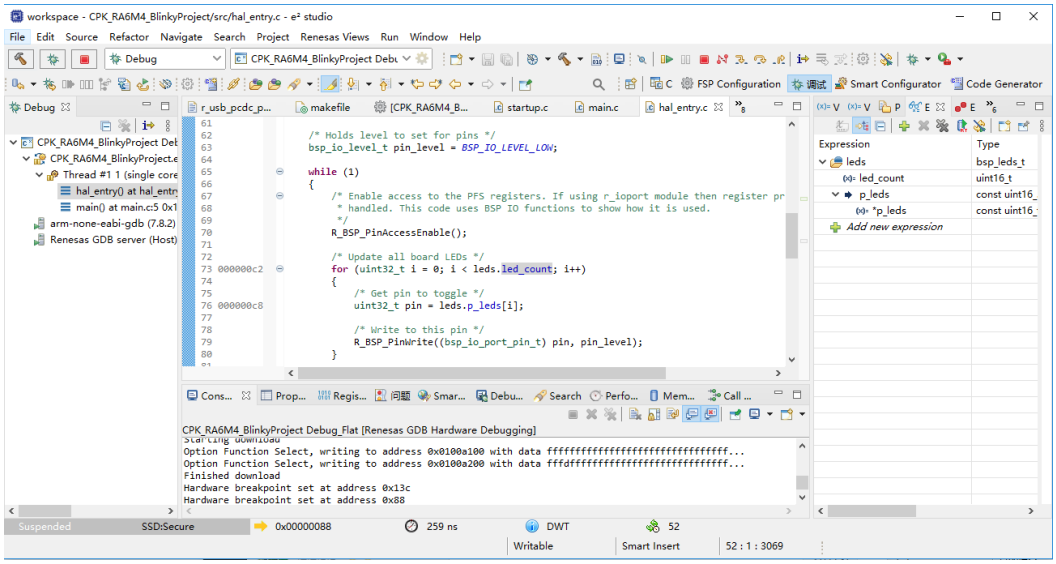


图 2-3：e² studio 的“Debug”（调试）透视图

本章要点：

- 为了将程序下载到任何硬件并进行调试，需要先创建一个调试配置。
- 加载后，调试器会将程序计数器设置为入口点。下一个停止处将位于 main() 中。

3. Hello World! – Hi Blinky!

本章内容基于《瑞萨 RA MCU 基础知识》中的章节 **8 Hello World! – Hi Blinky!** 所作。

您将在本章中学到以下内容：

- 如何从头开始为 CPK-RA6M4 评估板创建项目。
- 如何在 FSP 配置器中更改灵活配置软件包的设置。
- 如何编写代码以切换 CPK 上的用户 LED。
- 如何下载和测试程序。

大多数编程语言新手曾编写的第一个程序（现在仍是）就是将字符串“Hello World”输出到标准输出设备的程序。对我而言，就是在编辑器中键入“Writeln (‘Hello World’)”，如同我开始学习 Pascal 一样。从那时起，我用其他几种语言编写了类似的代码行，主要是为了对新开发环境的安装进行完整性检查。

20 世纪 80 年代末，当我开始编写嵌入式系统时，没有可以接收字符串的屏幕。那么，如何指示处理器发出正常工作的信号？在这些应用中几乎找不到 LED，因此，必须要切换仅有的 I/O 引脚之一并用示波器观察波形。这些年来，LED 成为了一种商品，我们在电路板上放置了大量的 LED，将它们的闪烁作为新的“Hello World”。

这也是本章的目标：切换 RA6M4 系列器件的评估板 (CPK) 上的 LED。您将（几乎）从头开始编写代码，使用配置器创建一个新项目，采用灵活配置软件包 (FSP) 的 API，最后下载、调试并运行代码。这项练习将各个操作步骤集中到一起。

CPK-RA6M4 可以轻松连接外部硬件，因为大多数引脚均可通过 MCU 引脚访问区域中的公头引脚插针或电路板的系统控制和生态系统访问区域中的生态系统连接器进行访问。由于 RA6M4 系列 MCU 是 RA 产品家族 MCU 的 RA6 系列的超集器件，因此可以评估该系列的大多数功能，并随后将结果应用于该系列的较小同级产品。图 3-1 所示为电路板的框图，其中突出显示了主要元件。

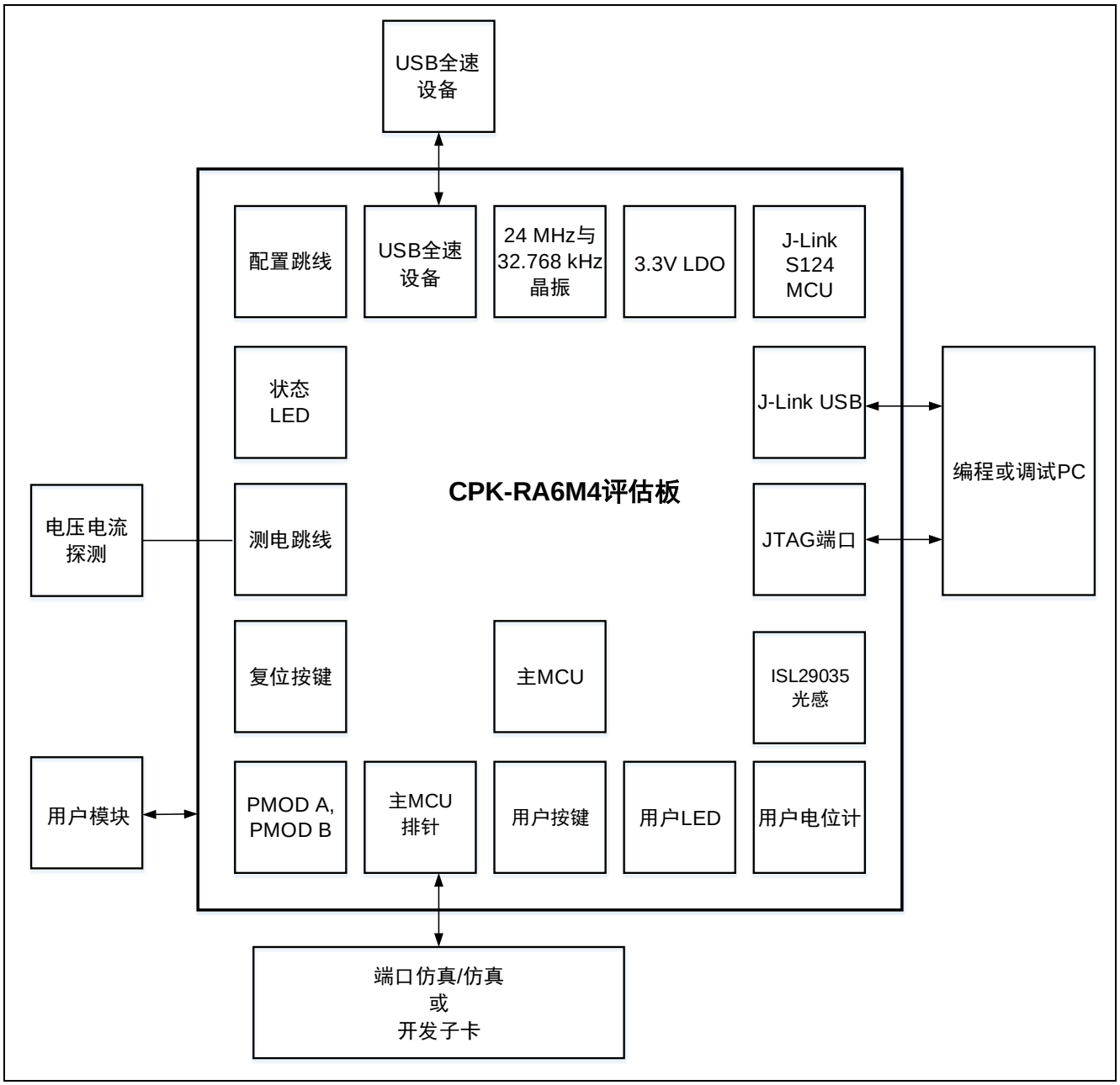


图 3-1：CPK-RA6M4 评估板的框图

3.1 使用项目配置器创建项目

如果尚未启动 e² studio，请从 Windows® 工作站的“Start”（开始）菜单中打开 e² studio。开发环境启动并运行后，请关闭“*Welcome*”（欢迎）屏幕（如果它在显示），因为它会挡住其他窗口。

由于在 e² studio 中为单片机编写新程序始终需要创建一个项目，因此这是您需要执行的第一步。为此，请转到“*File → New → Renesas C/C++ Project*”（文件 → 新建 → *Renesas* C/C++ 项目），或者在“Project Explorer”（项目资源管理器）视图中单击鼠标右键，然后选择“*New → Renesas C/C++ Project*”（新建 → *Renesas* C/C++ 项目）。两种方式都将打开一个对话框，询问要使用的模板。在左侧栏中选择 *Renesas RA*，再从主窗口中选择“*Renesas RA C/C++ Project*”（瑞萨 RA C/C++ 项目）。然后单击“*下一步*”。

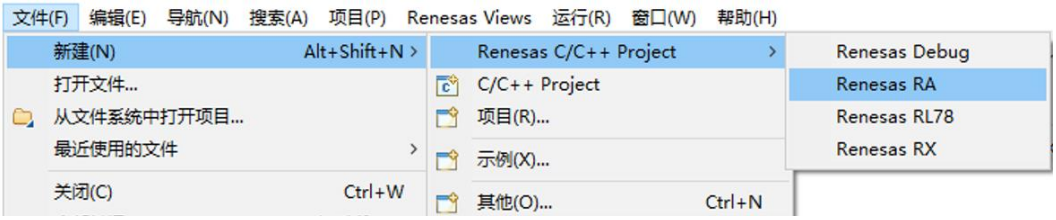


图 3-2：第一步是调用项目配置器

出现“*Project Configurator*”（项目配置器）后，为项目命名，接受项目的默认位置（将作为 e² studio 工作区），或将其更改为您偏好的文件夹。单击“*Next*”（下一步），转到“*Device and Tools Selection*”（器件和工具选择）屏幕。

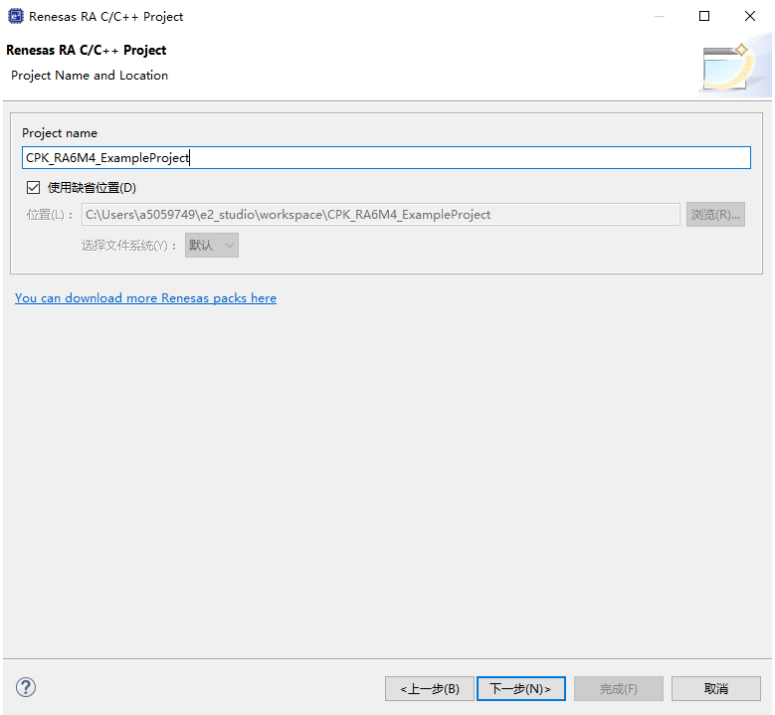


图 3-3：项目配置器的第一个屏幕主要询问项目的名称和位置

在“*Device Selection*”（器件选择）下，查找名为“*FSP Version*”（FSP 版本）的字段：它应显示与之前下载的灵活配置软件包相同的版本。从“*Board*”（电路板）下的下拉列表中选择 *CPK-RA6M4 MCU 评估板 （LQFP144*），因为这是我们用于小型“Hello World”程序的硬件。该列表通常将包含 RA 产品家族的评估板以及“*Custom User Board*”（定制用户板）条目，并通过为所选 FSP 版本安装的 Renesas CMSIS 包文件创建。验证 *R7FA6M4AF3CFB* 是否在“*Device*”（器件）旁显示，它应该已经自动插入。如果未显示，请浏览下拉列表，直到发现为止。在“*Toolchains*”（工具链）框中，验证是否列出了 *GCC ARM® Embedded, 9.2.1.20191025* 或更高版本，以及“*Debugger*”（调试器）框中是否已选择 *J-Link® Arm*。这些字段应预先填入。如果未预先填入，请修改相应项以匹配上面给出的值。

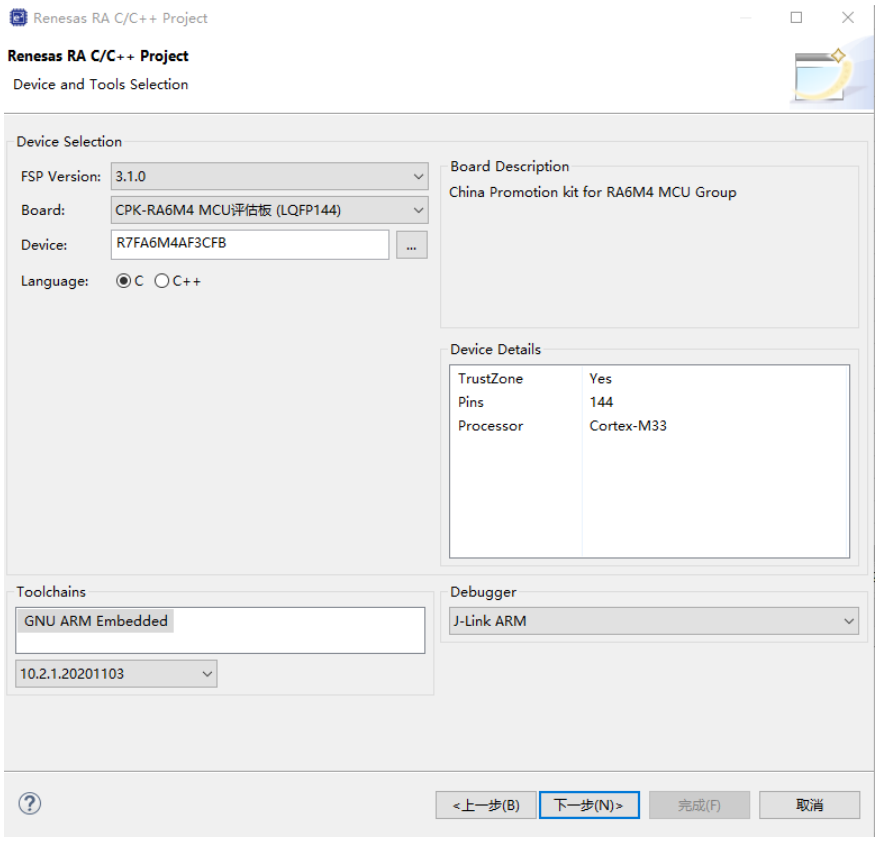


图 3-4：可以在此页面中选择项目的电路板和器件

如果一切正常，请单击“*Next*”（下一步），打开“*Project Type Selection*”（项目类型选择）屏幕。在此处，可以选择您的项目应当为所谓的“扁平化项目”（即无需 TrustZone® 隔离即可立即执行的项目）、包含安全启动代码和其他安全代码的安全 TrustZone 项目，还是包含与安全项目一起使用的非安全代码的非安全 TrustZone 项目。对于本章中的练习，选择“*Flat (Non-TrustZone) Project*”（扁平（非 TrustZone）项目），然后单击“*Next*”（下一步）继续操作。

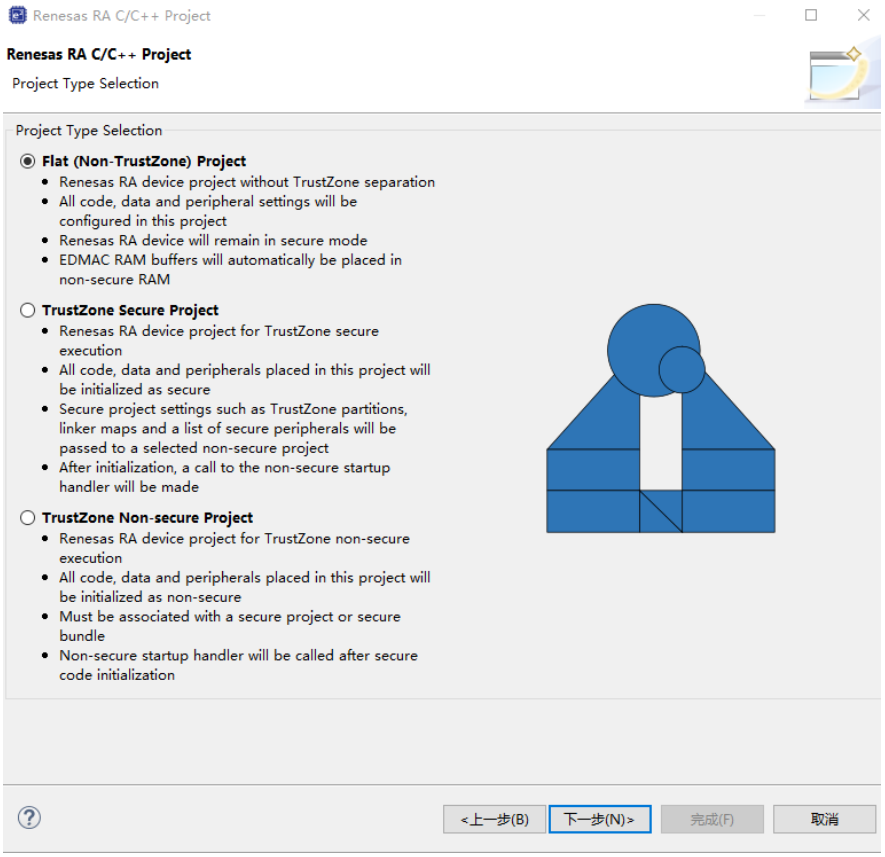


图 3-5：使用“Project Type Selection”（项目类型选择）屏幕，可以在 TrustZone 和非 TrustZone 项目之间进行选择

下一页是“*Build Artifact and RTOS Selection*”（构建工件和 RTOS 选择）屏幕，可以在其中设置构建的类型。只有在上一个窗口中选择了非 TrustZone 器件或为扁平化项目选择了 TrustZone 器件时，才会显示此屏幕。可用的选项包括用于创建独立 ELF（可执行和可链接格式）可执行文件的“*Executable*”（可执行文件）、用于创建目标代码库的“*Static Library*”（静态库）以及用于创建配置为与静态库一起使用的应用程序项目的“*Executable using an RA Static Library*”（使用 RA 静态库的可执行文件）。在页面右侧的下拉列表中，为项目选择可选的实时操作系统 (RTOS)。

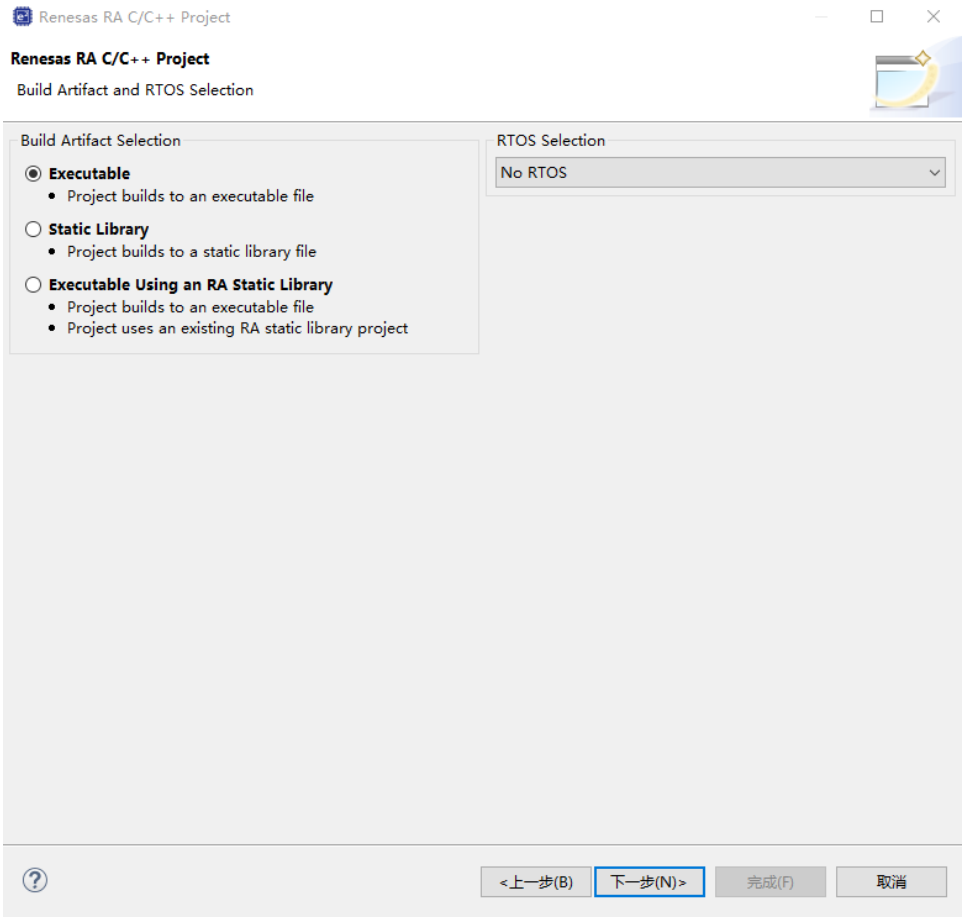


图 3-6：我们要编译一个没有 RTOS 的可执行项目

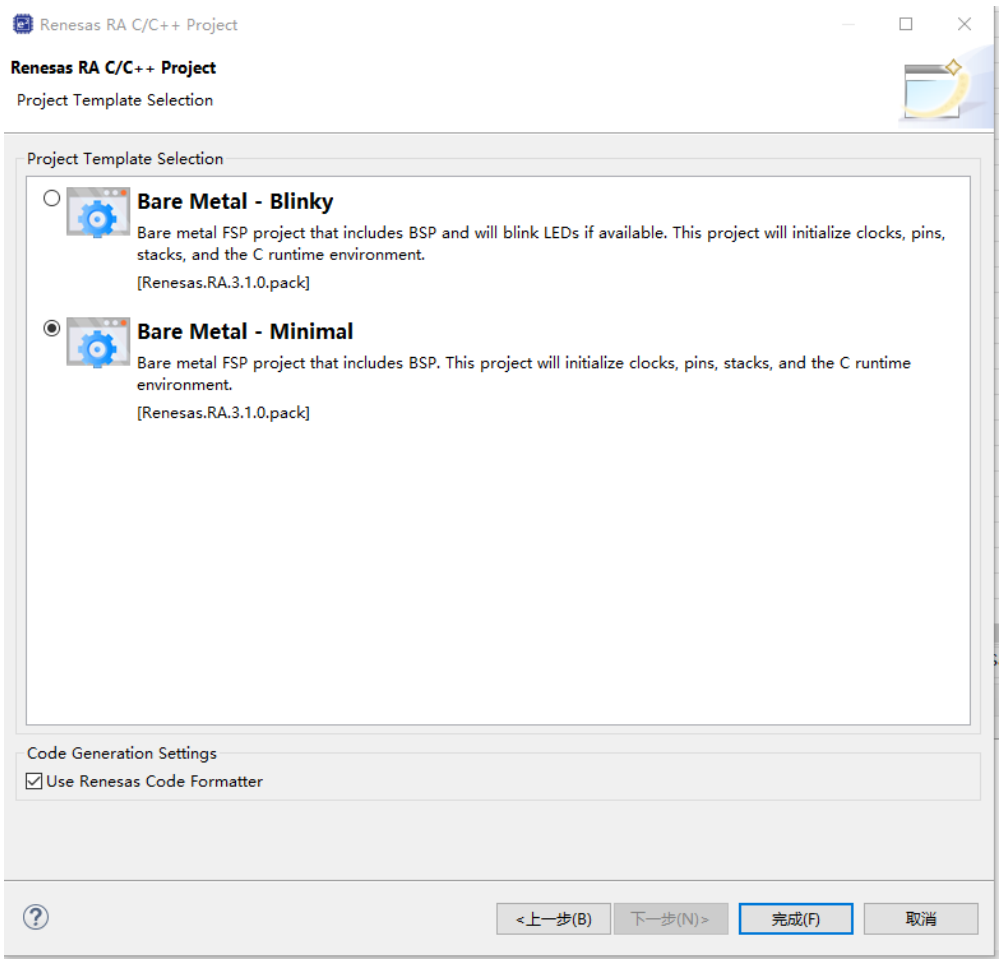


图 3-7：“Project Template Selection Page”（项目模板选择页面）将显示适合项目最初内容的模板

对于小型动手实验，请选择“*Executable*”（可执行文件）和“*No RTOS*”（无 RTOS），然后单击“下一步”。

这将打开“*Project Template Selection*”（项目模板选择）页面，可以在其中选择初始项目内容的模板。项目模板可能包含多个条目；至少包括适合所选电路板/器件组合的板级支持包。有些模板甚至包括一个完整的示例项目，但“Project Configurator”（项目配置器）将仅显示与您在先前屏幕上所做选择匹配的模板。在本例中，选择“*Bare Metal – Minimal*”（裸机 – 最小化）条目，以加载评估板的板级支持包。单击“*完成*”。完成项目的配置。

“*Project Configurator*”（项目配置器）将关闭并在最后一步中创建项目所需的所有文件。完成此后处理后，将出现一个对话框，询问您是否要打开“*FSP Configuration*”（FSP 配置）透视图。选择“*Open Perspective*”（打开透视图）。

3.2 使用 FSP 配置器设置运行环境

FSP 配置器启动后，将为您提供项目的只读摘要和所选软件组件的简短概述。此外，它还提供了快捷方式，可方便地访问 YouTube™ 上的瑞萨 RA 频道、Renesas.com 上的瑞萨设计与支持页面（可在其中访问文档、知识库和 Renesas Rulz 论坛）以及硬盘上的 FSP 用户手册。

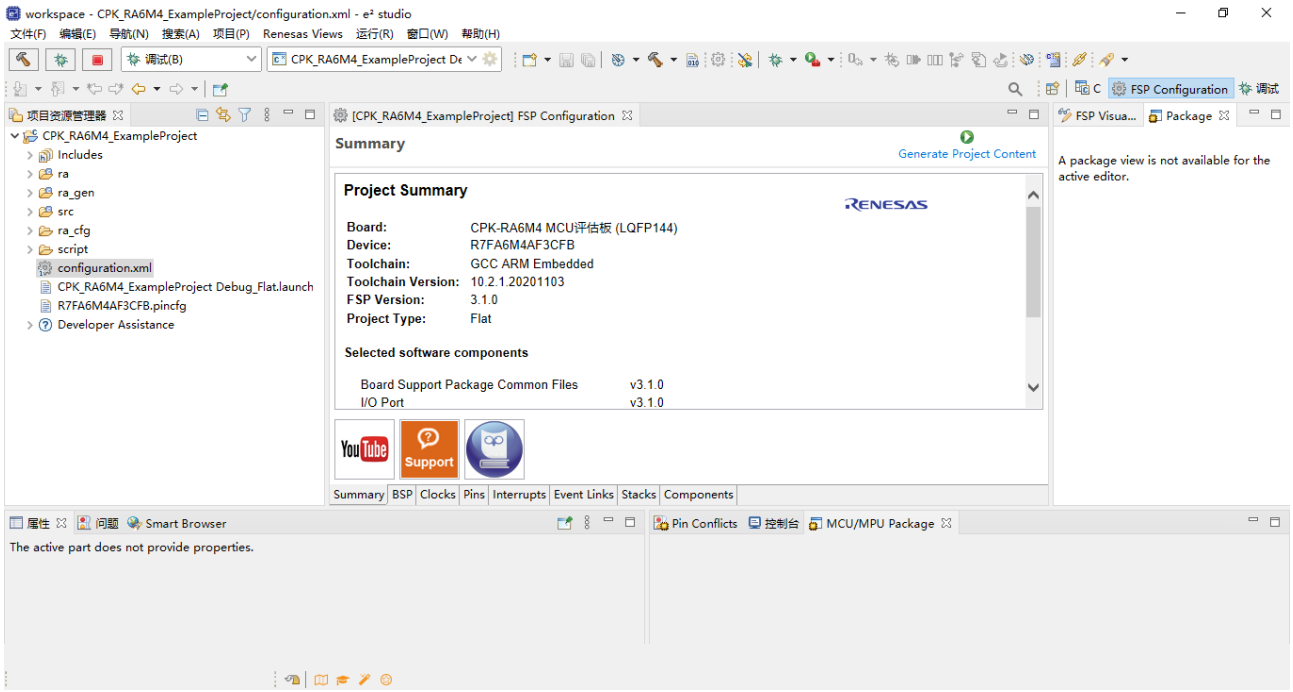


图 3-8：e² studio 内部的“FSP Configuration”（FSP 配置）透视图

在以下名为 *BSP* 的选项卡中，可以查看和编辑设置的多个方面，例如电路板和器件选择。在此选项卡的属性视图中，可以为板级支持包进行其他设置，例如，主堆栈的大小或 MCU 的某些安全功能。在之后的“*Clocks*”（时钟）选项卡中，可以为您的项目分配初始时钟配置。任何潜在的问题都将以红色突出显示，将鼠标悬停在突出显示的位置上将出现有关冲突或设置不完整的说明。

第四个选项卡“*Pins*”（引脚）涵盖了 RA MCU 的引脚分配。可以根据端口或外设列出引脚。如果设置不兼容或缺失，则配置器右侧的“*Package View*”（封装视图）会显示器件的封装，突出显示所配置的引脚并标记错误。“*Problems*”（问题）视图以及“*Pin Conflicts*”（引脚冲突）视图中也会显示这些内容。这样，便可将可能的错误减少到最低限度。

接下来是“*Interrupts*”（中断）选项卡。可以在此处指定用户定义的（即非 FSP）驱动程序如何使用单片机的中断控制器单元 (ICU)，以及将哪个中断服务程序 (ISR) 与 ICE 事件（中断）相关联。此外，还可以在此处查看分配的所有 ICU 事件的完整列表，包括由在配置器的“*Stacks*”（栈）视图中创建的 FSP 模块实例生成的 ICU 事件。

“*Event Links*”（事件链接）选项卡具有类似作用。可以在此处指定驱动程序如何在 RA 项目中使用事件链接控制器 (ELC)，并且可以声明此类驱动程序可能通过一组外设功能产生一组 ELC 事件或使用一组 ELC 事件。

需要花费大部分时间的页面为“*Stacks*”（栈）页面，可以在其中创建 RTOS 线程和内核对象，以及 FSP 软件栈。可以添加不同的对象和模块，并且可以在“*Properties*”（属性）视图中修改其属性。所有这些

对象和模块都将自动插入，直到降至需要用户干预的程度为止。在这种情况下，一旦鼠标悬停在模块上，便会将需要注意的模块标记为红色，同时给出必要设置或问题的说明。如果问题解决，模块将恢复为标准颜色。“*Stacks*”（栈）视图本身以图形方式显示各种栈，可让您轻松跟踪不同的模块。在我们的示例中，仅显示了一个具有一个模块的线程：在 r_ioport 上使用 g_ioport I/O 端口驱动程序的 HAL/通用线程。它是由项目配置器自动插入的，允许我们仅用几行代码便可编写让 LED 闪烁的程序。

最后一个选项卡的名称是“*Components*”（组件），其中显示了不同的 FSP 模块并可对模块进行选择。它还列出了可用的 RA CMSIS 软件组件。不过，最好通过“*Stacks*”（栈）页面在当前项目中添加或删除模块，因为还可以在其中进行配置。

对于我们的项目，无需在 FSP 配置器中进行任何更改，因为项目配置器已经为我们进行了所有必要的设置。最后，需要创建基于当前配置的附加源代码。单击 FSP 配置器右上角的“*Generate Project Content*”（生成项目内容）按钮。此操作将从 FSP 中提取所需文件，将其调整为在配置器中进行的设置，然后将其添加到项目中。

3.3 编写前几行代码

获取所有自动生成的文件之后，接下来查看创建的内容。IDE 左侧的“*Project Explorer*”（项目资源管理器）列出了当前包含的所有内容。*ra_gen* 文件夹保存通道号等配置集。*src* 目录包含一个名为 *hal_entry.c* 的文件。这是稍后要编辑的文件。请注意，尽管在 *ra_gen* 文件夹中有一个名为 *main.c* 的文件，但用户代码必须转到 *hal_entry.c* 中。否则，如果您在 FSP 配置器中进行修改并重新创建项目内容，则更改会丢失，因为每次单击“*Generate Project Content*”（生成项目内容）时，都将覆盖该文件。

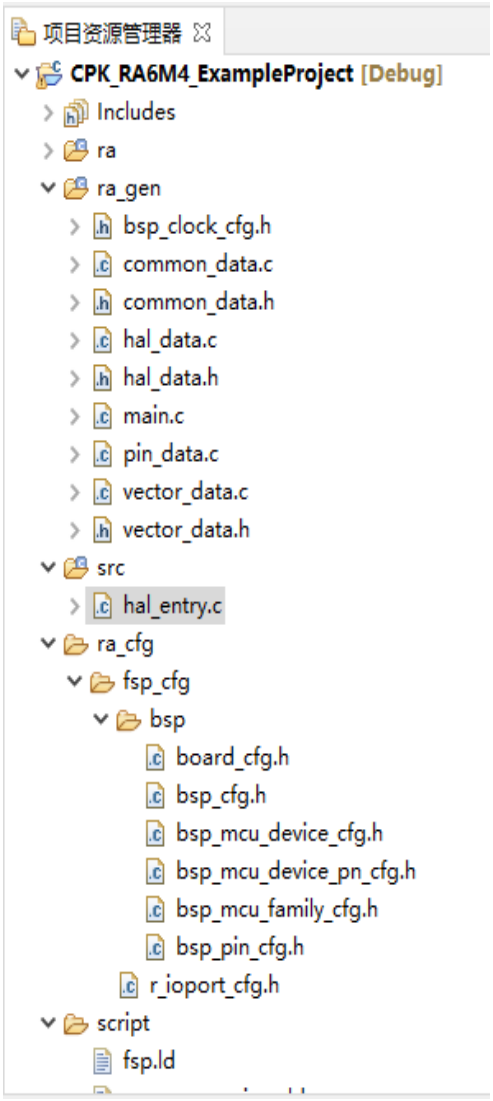


图 3-9：FSP 配置器创建所需文件后的项目树

该项目还包含几个名称中带有“ra”或“fsp”的目录，其中包含 FSP 的源文件、包含文件和配置文件。通常的规则是，不得修改这些文件夹（和子文件夹）的内容。其中包含由配置器生成的文件，在此所做的任何更改都将在下次生成或刷新项目内容时丢失。用户可编辑的源文件是直接位于 *\src* 文件夹或您添加的任何其他文件夹的根目录中的文件。

接下来，为 RA 产品家族单片机编写第一个真实源代码。计划是在 CPK-RA6M4 评估板上的用户 LED（红色） 闪烁。因此您必须通过添加代码来点亮和熄灭 LED 以及实现延时循环。如何实现？

实际上有两种选择：一种是通过接口函数来使用 API，另一种是使用 BSP 实现函数。

如果查看文件 *ra_gen\common_data.c* 中的代码，则会发现 I/O 端口驱动程序实例 `g_ioport` 具有以下定义：

```
const ioport_instance_t g_ioport = {.p_api = &g_ioport_on_ioport,
                                    .p_ctrl = &g _ioport_ctrl,
                                    .p_cfg = &g_bsp_pin_cfg, };
```

`g_ioport_on_ioport` 是一个结构体，用于声明端口可能执行的操作，将分配给 `g_ioport` 实例的 API 指针。将鼠标悬停在该结构体上，可以轻松查看其中的内容，此结构体显示了其成员之一 (`.pinWrite`) 是指向引脚写入函数的指针。

因此，要点亮 LED，可以写入：

```
g_ioport.p_api->pinwrite (&g_ioport_ctrl, pin, BSP_IO_LEVEL_LOW);
```

但这意味着实际上需要知道 用户 LED 连接到哪些 I/O 端口，以及有多少个用户 LED 可用。为此，我们可以阅读电路板的文档或仔细检查原理图以找到正确的端口。或者，也可以只依靠 FSP。创建类型为 `bsp_leds_t` 的结构体（在 *board_leds.h* 中声明）并为其分配在 *board_leds.c* 中定义的全局 BSP 结构体 `g_bsp_leds` 即可解决问题。这两个文件均位于项目的 *ra\board\ra6m4_cpk* 文件夹内。因此，以下两行代码足以获取有关评估板上 LED 的信息：

```
extern bsp_leds_t g_bsp_leds;
bsp_leds_t Leds = g_bsp_leds;
```

现在，可以使用 LED 结构体来访问电路板上的所有 LED，并使用以下语句点亮红色 LED（将端口设置为低电平将点亮 LED，将端口设置为高电平则将熄灭 LED）：

```
g_ioport.p_api->pinWrite(&g_ioport_ctrl,
                         leds.p_leds[BSP_LED_LED3],
                         BSP_IO_LEVEL_LOW);
```

此语句后需要有第二条语句，用于将其引脚设置为高电平以熄灭 用户 LED。
最后，需要提供一段延时以使 LED 以用户友好的方式切换。为此，可以再次调用 BSP API：

```
R_BSP_SoftwareDelay(1, BSP_DELAY_UNITS_SECONDS);
```

`R_BSP_SoftwareDelay` 函数的第一个参数是要延迟的单位数，而第二个参数是指定的基本单位，在本例中为秒。其他选项包括毫秒和微秒。

最后，由于我们想无限期地运行程序，因此必须围绕代码创建一个 `while(1)` 循环。

目前，还需要执行的操作是将以下代码行直接输入到 *hal_entry.c* 文件中的函数签名之后，替换 `/* TODO: add your own code here */` 行。对于由项目配置器和 FSP 配置器插入的其他代码，请保持不变。单片机需要借助这些代码来正常运行。

```
extern bsp_leds_t g_bsp_leds;
bsp_leds_t Leds = g_bsp_leds;

while (1)
{
    g_ioport.p_api->pinWrite(&g_ioport_ctrl,
                             Leds.p_leds[BSP_LED_LED3],
                             BSP_IO_LEVEL_LOW);

    R_BSP_SoftwareDelay(1, BSP_DELAY_UNITS_SECONDS);

    g_ioport.p_api->pinWrite(&g_ioport_ctrl,
                             Leds.p_leds[BSP_LED_LED3],
                             BSP_IO_LEVEL_HIGH);

    R_BSP_SoftwareDelay(1, BSP_DELAY_UNITS_SECONDS);
}
```

编写代码时，始终可以使用 e² studio 的自动完成功能。只需按下 <Ctrl>-<Space>，便会出现一个窗口，显示结构体或函数可能的补全代码。如果单击一个条目，它会被自动插入代码中。

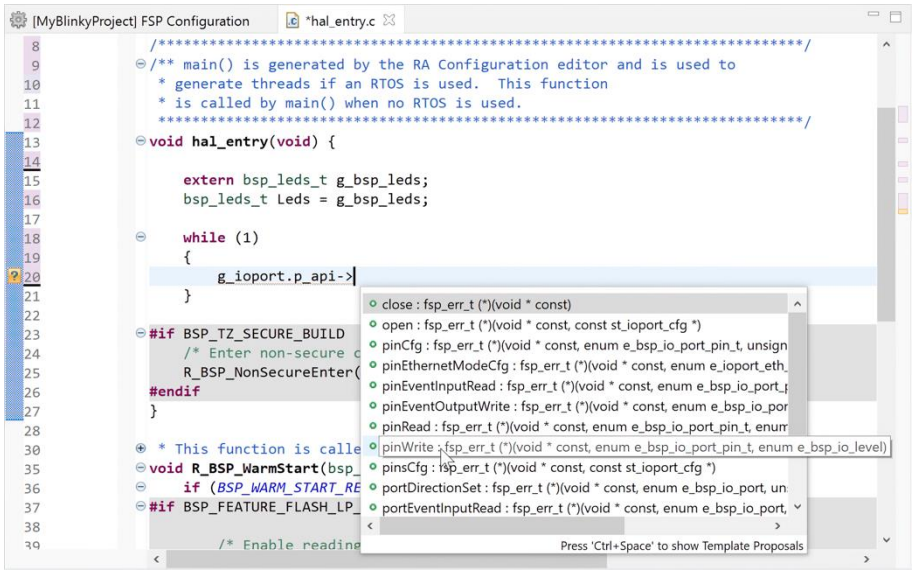


图 3-10：在变量或函数上按下 <Ctrl>-<Space> 将激活 e² studio 的代码补全功能

编写程序时，另一个有用的工具是“Developer Assistance”（开发人员帮助），可以从“Project Explorer”（项目资源管理器）中访问此工具。在使用 FSP 配置器配置了项目的软件堆栈之后，此工具将为您快速了解应用程序代码提供支持。要访问“Developer Assistance”（开发人员帮助），请先在“Project Explorer”（项目资源管理器）中展开项目，此工具随即显示。显示工具后，进一步展开树，直到看到堆栈模块及其 API。选择要使用的 API，然后将对该 API 的调用拖放到源文件中。

现在轮到您进行操作：请将上面的代码行输入到项目的 *hal_entry.c* 文件中。为此，展开项目的 *src* 文件夹，然后双击上述文件。此操作会在编辑器中将其打开。如果您不想自己输入所有内容，也可以从本手册的网站 (**www.renesas.com/ra-book**) 下载完整的项目。

3.4 编译第一个项目

输入所有内容后，便可随时编译程序。编译有两种不同的配置：调试和发布。调试配置将包含调试程序所需的所有信息，例如变量和函数名，并且还将关闭编译器的某些优化，例如循环展开。这会使调试更加容易，但会增大代码大小、减慢代码执行速度。发布配置将从输出文件中除去所有这些信息，并开启完全优化，从而减小代码大小、加快代码执行速度，但是，您再也无法执行查看变量等操作（除非您知道它们在存储器中的地址）。

对于第一个测试，可以采用调试配置（也是默认配置）。要编译项目，单击主菜单栏  上的“*build*”（编译）按钮，编译过程随即开始。如果一切正常，编译将以 0 个错误和 0 个警告结束。如果存在编译时错误，则需要返回代码，仔细检查是否正确输入了所有内容。如果未正确输入所有内容，请相应地更改代码。为了让您更轻松地定位错误，编译器的反馈将直接插入编辑器窗口（如果可能）。

程序编译成功后，会创建输出文件 *CPK_RA6M4_ExampleProject.elf*，需要先将其下载到处理器，然后才能运行和调试该文件。

3.5 下载和调试第一个项目

下一步是在评估板 (CPK) 上实际运行程序。现在需要将评估板连接到 Windows® 工作站：将电路板随附的 USB 线缆的 micro-B 端插入系统控制和生态系统访问区域右下角的 USB 调试端口 J11，将另一端插入 PC 上的空闲端口。LED1 应点亮，表示电路板已通电。如果该评估板支持开箱即用，则预编程的演示会运行，表明一切都按预期运行。Windows 操作系统可能会显示一个对话框，指示正在安装 J-Link® 板上调试器的驱动程序，此过程应自动完成。此外，还可能会出现一个窗口，询问是否更新 J-Link® 调试器。强烈建议允许进行此更新。

下载：

要下载程序，必须先创建一个调试配置。单击“Debug”（调试）符号 旁边的小箭头，然后从下拉列表框中选择“*Debug Configurations*”（调试配置）。

在出现的窗口中，突出显示“*Renesas GDB Hardware Debugging*”（瑞萨 GDB 硬件调试）下的 *CPK_RA6M4_BlinkyProject Debug_Flat*。由于项目配置器已经进行了所有必要的设置，因此无需在此对话框中进行任何更改。只需单击窗口右下角的“*Debug*”（调试）。此操作会启动调试器，将代码下载到 CPK 上的 RA6M4 MCU，并询问您是否要切换到“*Debug Perspective*”（调试透视图）。请选择“*Switch*”（切换）。“*Debug Perspective*”（调试透视图）将打开，并且程序计数器将设置为程序的入口点，即复位处理程序。此调试配置仅需要创建一次。下次只需单击“*Debug*”（调试）符号 便可启动调试器。

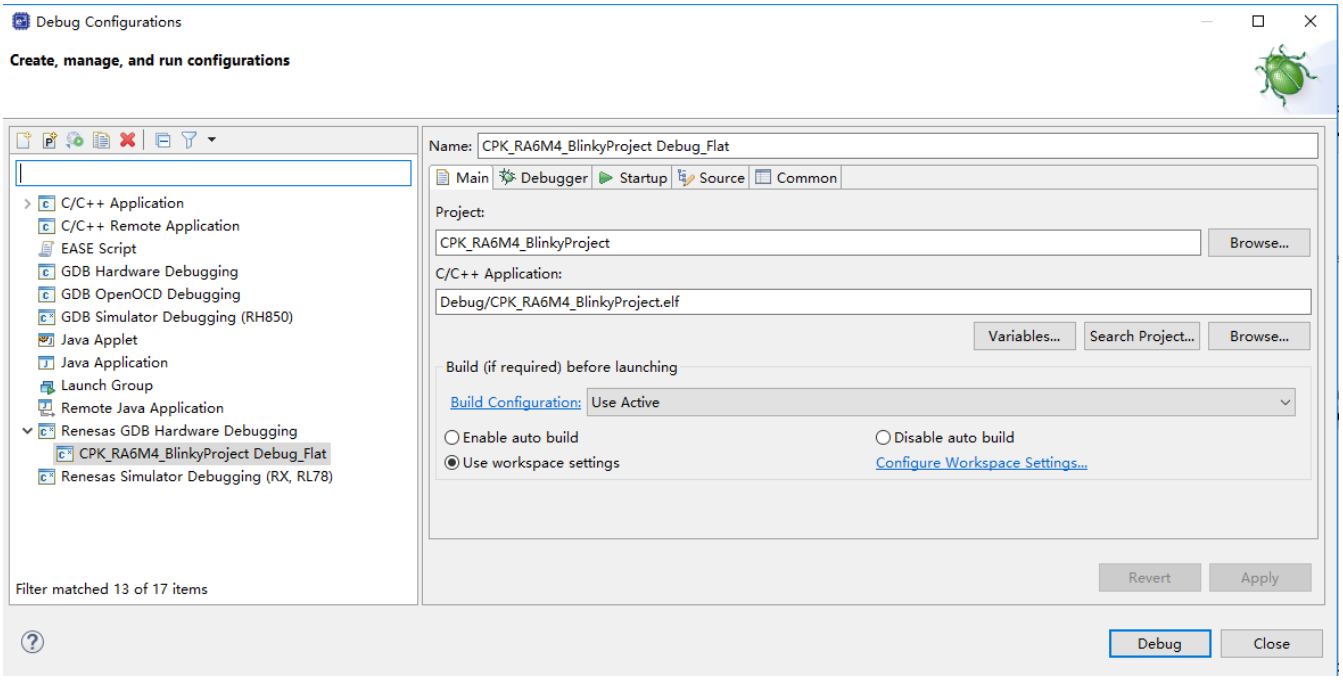


图 3-11：选择 MyBlinkyProject Debug_flat 后，无需在其他选项卡上进行任何更改

运行：

单击“*Resume*”（恢复）按钮，下一个停止处将处于 `main()` 中调用 `hal_entry()` 的位置。再次单击该按钮，程序将继续执行，且用户 LED（红色）将按预期的 1 秒时间间隔闪烁。

观察结果：

如果一切正常，单击主菜单栏上的“*Suspend*”（暂停）按钮。这将停止执行程序但不会将其终止。在编辑器视图中，激活文件 *hal_entry.c* 的选项卡，然后右键单击包含对端口的写操作的其中一行；在出现的菜单中，选择“*Run to line*”（运行至指定行）。执行将恢复，程序将在单击的行处停止。现在来看一下右侧包含变量的视图。您将看到列出的 `Leds` 结构体。将其展开，浏览和分析不同的字段。调试较大的项目时，此视图会派上用场。

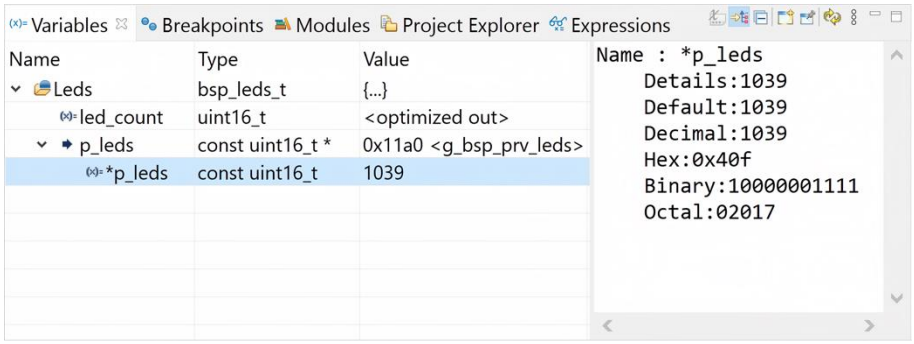


图 3-12：变量及其值可以在“Variables”（变量）视图中进行检查

最后一步是单击“*Terminate*”（终止）按钮，结束调试会话，以停止程序的执行。

恭喜！

您已经掌握了 RA 产品家族单片机的第一个程序！

本章要点：

- 项目配置器将创建新项目所需的所有文件和设置。
- FSP 配置器允许编程人员基于图形用户界面轻松配置 FSP 和运行环境。
- 调试配置是调试项目的必需步骤。它会自动创建，只需要激活即可。
- 实现所需功能仅需要很少的代码行。

4. 使用实时操作系统

本章内容基于《瑞萨 RA MCU 基础知识》中的章节 **9 使用实时操作系统** 所作。

您将在本章中学到以下内容：

- 什么是线程、信号量和队列，以及如何使用它们。
- 如何在 e² studio 中向程序添加线程和信号量。
- 如何在 RTOS 控制下通过按钮切换用户 LED 的状态。

上一章中的练习已经利用了瑞萨 RA 系列单片机 (MCU) 灵活配置软件包 (FSP) 的很大一部分。在本章中，您将使用 FreeRTOS™ 实时操作系统创建一个小型应用程序，利用线程控制 LED 并利用信号量实现与按钮的同步。您将亲身体验到这实际上仅需要几个步骤。

我们将从头开始创建完整的项目，因此如果您没有进行过之前的实验，请不必担心。

4.1 线程、信号量和队列

在我们实际深入进行此练习之前，需要定义将在本章和下一章中使用的一些术语，以确保我们能够达成共识。

首先，需要定义术语“线程”。如果您更习惯于“任务”这个表达方式，只需把线程看作是一种任务。有些人甚至互换使用这两个短语。当使用实时操作系统 (RTOS) 时，单片机上运行的应用程序将拆分为几个较小的半独立代码块，每个代码块通常控制程序的一个方面。这些小片段称为线程。一个应用程序中可以存在多个线程，但是在任何给定时间都只能有一个线程处于活动状态，因为 RA 系列单片机是单核器件。每个线程都有自己的堆栈空间，如果需要安全的上下文，则可以将其置于 MCU 的安全侧。每个线程还分配有优先级（相对于应用程序中的其他线程），并且可以处于不同的状态，例如运行、就绪、阻塞或暂停。在 FreeRTOS™ 中，可以通过调用 `eTaskGetState()` API 函数来查询线程的状态。线程间信号传输、同步或通信是通过信号量、队列、互斥、通知、直接任务通知或者流和消息缓冲区来实现的。

信号量是 RTOS 的资源，可用于传输事件和线程同步（以产生者－使用者方式）。使用信号量允许应用程序暂停线程，直到事件发生并发布信号量。如果没有 RTOS，就需要不断地轮询标志变量或创建代码来执行中断服务程序 (ISR) 中的某个操作，这会在相当长的一段时间内阻塞其他中断。使用信号量可快速退出 ISR 并将操作推迟到相关线程。

FreeRTOS 提供计数信号量和二进制信号量。尽管二进制信号量由于只能采用两个值（0 和 1）而非常适合实现任务之间或中断与任务之间的同步，但是计数信号量的计数范围可涵盖 0 到用户在 FSP 配置器中创建信号量期间指定的最大计数。默认值为 256，可支持设计人员执行更复杂的同步操作。

每个信号量都有两个相关的基本操作：`xSemaphoreTake()`（将使信号量递减 1）和 `xSemaphoreGive()`（将使信号量递增 1）。这两个函数有两种形式：一种是可以从中断服务程序内部调用（`xSemaphoreTakeFromISR()` 和 `xSemaphoreGiveFromThread()`）的形式，另一种则是上述可以在线程的正常上下文中调用的形式。

我们需要讨论的最后一个术语是队列，即使在本练习中不使用队列，下一章的练习中也会使用。报文队列是线程间通信的主要方法，它允许在任务之间或中断与任务之间发送消息。消息队列中可以有一条或多条消息。

数据（也可以是指向更大缓冲区的指针）会复制到队列中，即，它不存储引用而是消息本身。新消息通常置于队列的末尾，但也可以直接发送到开头。接收到的消息将从前面开始删除。

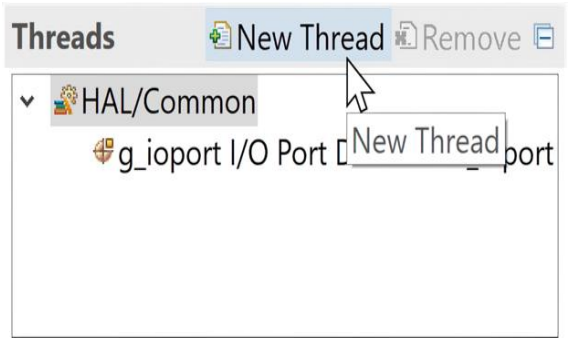
允许的消息大小可在设计时通过 FSP 配置器指定。默认项大小为 4 个字节，默认队列长度（表示队列中可存储的项数）为 20。所有项的大小必须相同。FreeRTOS 中的队列数没有限制；惟一的限制是系统中可用的存储空间。使用 `xQueueSend()` 函数将消息放入队列中，并通过 `xQueueReceive()` 从队列中读取消息。与信号量一样，函数有两种版本：一种可以从线程的上下文调用，另一种可以从 ISR 内部调用。

4.2 使用 e2 studio 将线程添加到 FreeRTOS 中

接下来的练习也是基于 CPK-RA6M4 评估板。这次，我们将使用电路板下方的用户按钮 S1 向应用程序传输事件，应用程序将切换红色 LED 的状态进行响应。为实现目标，我们将使用 FreeRTOS，事件的处理将在线程内进行，并通过信号量进行通知。

第一步是使用项目配置器创建一个新项目。首先，转到“*File → New →“Renesas RA C/C++ Project*”（文件 → 新建 → *Renesas C/C++* 项目）。单击“*Next*”（下一步）并在出现的屏幕上输入项目名称，例如 *CPK_RA6M4_RtosProject*。再次单击“*Next*”（下一步）。此操作将转到“*Device and Tools Selection*”（器件和工具选择）窗口。首先，选择一个电路板。选择 *CPK-RA6M4* 并将相应的器件设置为 *R7FA6M4AF3CFB*（如果尚未列出）。查看工具链：它应显示为 *GCC Arm® Embedded*。单击“*Next*”（下一步）继续操作。

在当前出现的屏幕中，可以在非 TrustZone® 与安全和非安全 TrustZone 项目之间进行选择。保持“*Flat (Non-TrustZone) Project*”（扁平化（非 TrustZone）项目）处于选中状态，然后单击“*Next*”（下一步）。随即出现“*Build Artifact and RTOS Selection*”（构建工件和 RTOS 选择）窗口。保持设置不变，即在“*Build Artifact Selection*”（构建工件选择）下选择“*Executable*”（可执行文件），在“*RTOS Selection*”（RTOS 选择）下选择 *FreeRTOS*。单击“*Next*”（下一步），转到下一个名为“*Project Template Selection*”（项目模板选择）的屏幕。在此，选择“*FreeRTOS – Minimal – Static Allocation*”（FreeRTOS – 最小化 – 静态分配）。



最后，单击“*Finish*”（完成），在配置器生成项目后，e² studio 将询问您是否切换到“*FSP Configuration*”（FSP 配置）透视图。透视图出现后，直接转到“*Stacks*”（堆）选项卡。该选项卡将在“Threads”（线程）窗格中显示“*HAL/Common*”（HAL/通用）线程的单个条目，其中包含 I/O 端口的驱动程序。单击窗格顶部的“*New Thread*”（新线程）图标（请参见图 4-1 添加新线程）。

图 4-1：在 FSP 配置器出现之后，将仅显示一个线程。选择“New Thread”（新线程）按钮，添加另一个线程

现在，在“*Properties*”（属性）视图中更改新线程的属性：将“*Symbol*”（符号）重命名为 *led_thread*，将“*Name*”（名称）重命名为 *LED Thread*。其他属性保持默认值。在“*LED Thread Stack*”（LED 线程堆）窗格中，单击“*New Stack*”（新线程）按钮图标，选择“*Driver → Input → External IRQ Driver on r_icu*”（驱动程序 → 输入 → r_icu 上的外部 IRQ 驱动程序）（请参见图 4-2）。

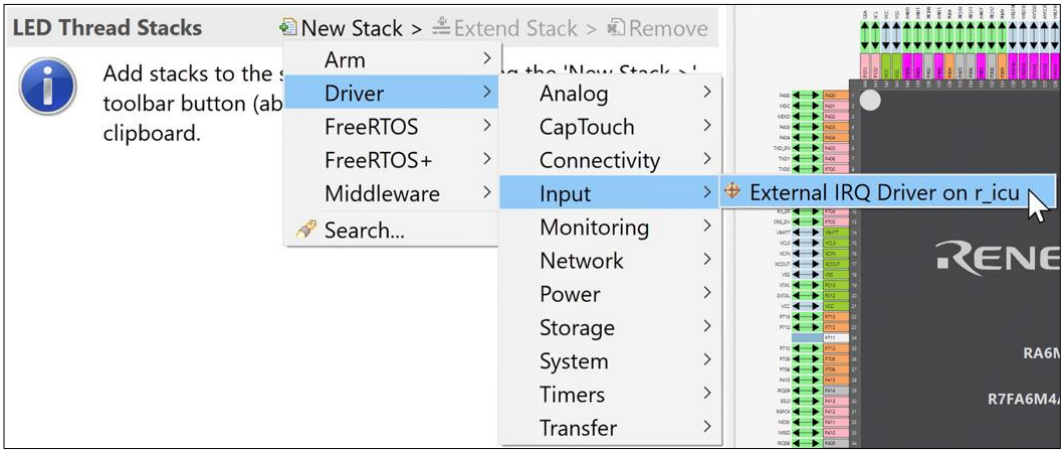


图 4-2：添加新驱动程序只需单击几下鼠标

此操作将为外部中断添加驱动程序。查看新驱动程序的“*Properties*”（属性）：

首先，请确保“*Channel*”（通道）为 *0*，因为 S1 所连引脚连接到 IRQ00。出于相同的原因，将名称更改为 *g_external_irq00* 或您喜欢的任何名称。

为中断分配优先级 12，启动期间 FSP 将不会允许该中断。也可以选择任何其他优先级，但开始时最好选择优先级 12，因为即使在较大的系统中，也很少会遇到中断优先级冲突。请注意，优先级 15 是为系统时钟节拍定时器 (systick) 保留的，因此不应被其他中断使用。

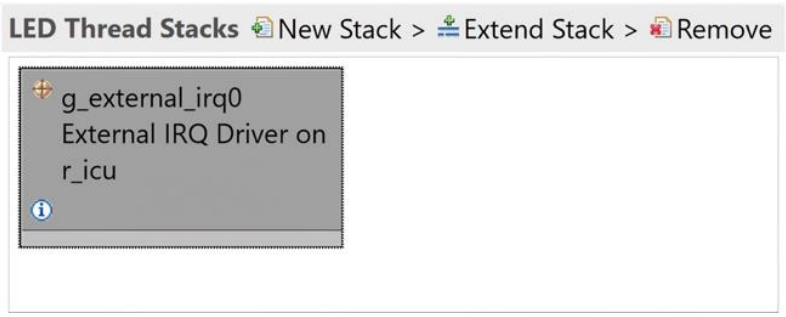


图 4-3：堆元素的灰色条表示此驱动程序是模块实例，只能由另一个 FSP 模块实例引用

将“*Trigger*”（触发器）从“*Rising*”（上升）更改为“*Falling*”（下降）以捕捉按钮激活操作，并将“*Digital Filtering*”（数字滤波）从“*Disabled*”（禁用）更改为“*Enabled*”（启用）。始终将“*Digital Filtering Sample Clock*”（数字滤波采样时钟）设置为 *PCLK / 64*。这将有助于对按钮去抖。最后，用 *external_irq00_callback* 替换 *Callback* 行中的 *NULL*。每次按下 S1 都会调用此函数。在稍后创建应用程序时，我们将为回调函数本身添加代码。图 4-4 给出了必要设置的摘要。

Property	Value
Common	
Module g_external_irq00 External IRQ Driver on r_icu	
Name	g_external_irq00
Channel	0
Trigger	Falling
Digital Filtering	Enabled
Digital Filtering Sample Clock (Only valid when Digital Filtering is Enabled)	PCLK / 64
Callback	external_irq00_callback
Pin Interrupt Priority	Priority 12
Pins	
IRQ00	P105

图 4-4：应用程序所需的 IRQ 驱动程序的属性

现在，只需要执行几个步骤，即可编译和下载程序。下一步是添加信号量。

为此，请在“*LED Thread Objects*”（LED 线程对象）窗格中单击“*New Object*”（新对象）按钮。如果看到的不是此窗格，而是“*HAL/Common Objects*”（HAL/通用对象）窗格，则突出显示“*Threads*”（线程）窗格中的“*LED Thread*”（LED 线程），随即将显示此窗格。添加一个*二进制信号量*，我们需要在按下按钮时通知 *LED 线程*。将信号量的“*Symbol*”（符号）属性更改为 *g_s1_semaphore*，并将“*Memory Allocation*”（存储器分配）保留为“*Static*”（静态）。现在，FSP 配置器中的“*Stacks*”（堆）选项卡的外观应类似于图 4-5。

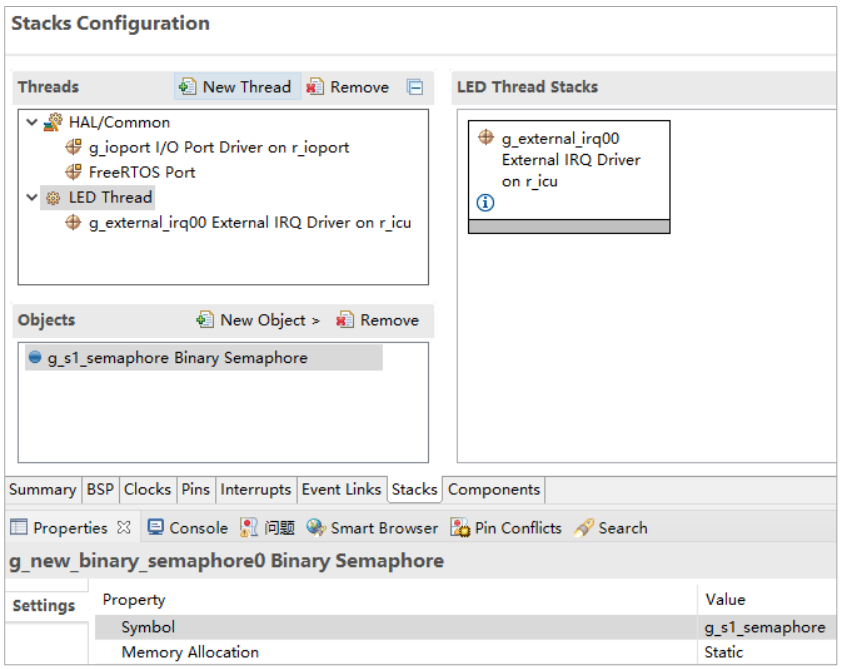


图 4-5：这是添加 LED 线程和信号量后“Stacks”（堆）选项卡应呈现的外观

FSP 配置器中的最后一步是将 S1 连接的 I/O 引脚配置为 IRQ00 输入。为此，请激活配置器中的“*Pins*”（引脚）选项卡，展开“*Ports → P1*”（端口 → P1），然后选择 *P105*。在 CPK-RA6M4 评估板上，这是 S1 连接的端口。在右侧的“*Pin Configuration*”（引脚配置）窗格中，为其指定*符号名称* *SW1*，并确保其他设置与图 4-6 中的设置相同。通常，配置器应该已为您完成了相关设置。如果没有完成，请相应调整。请注意，右侧的封装查看器将突出显示引脚 103/P105，这样便可获得引脚位置的图形参考。

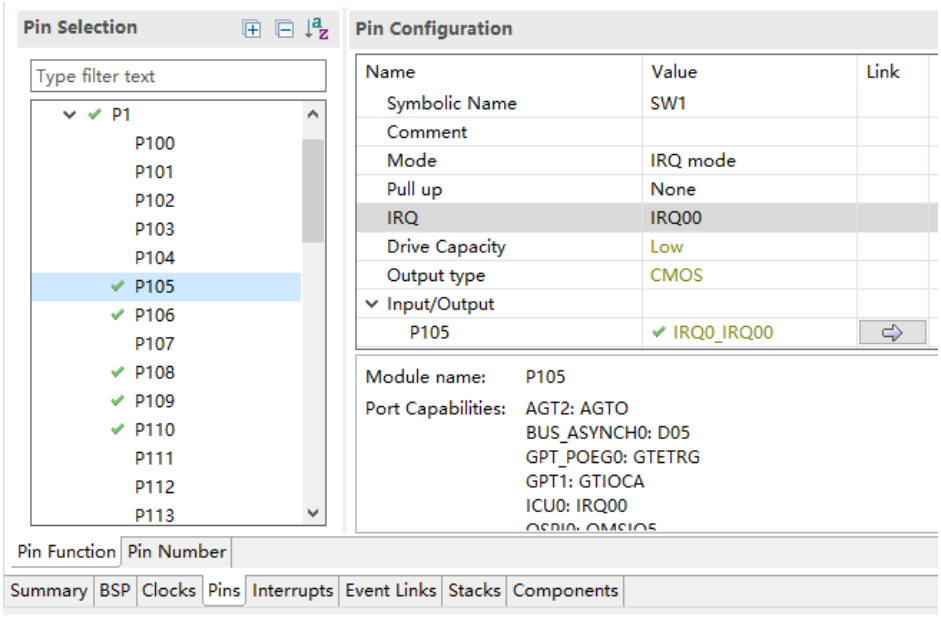


图 4-6：应已为 IRQ00 正确配置了端口 P105

完成此操作后，即完成了配置器中的设置。保存更改，然后单击其顶部的“*Generate Project Content*”（生成项目内容）图标以创建必要的文件、文件夹和设置。

需要执行的最后一项任务是添加初始化 `Leds` 结构体所需的代码，编写几行代码来切换 LED 并读取信号量，然后创建将设置信号量的回调函数。可以在本章末尾查看完整代码。

由于我们正在使用 *LED 线程*处理按钮和切换 LED 的状态，因此本次需要将相关代码添加到 *led_thread_entry.c* 文件中。在“*Project Explorer*”（项目资源管理器）中双击文件名以在编辑器中将其打开。如果未显示文件，请展开项目文件夹，然后展开 *src* 目录。与第 3 章中的练习一样，为 LED 添加结构体并对其进行初始化。需要定义用户 LED（红色）所连 I/O 引脚的电平的另一个变量。将其命名为 `led_level`。该变量的类型需要采用 `ioport_level_t`，并且应初始化为 `IOPORT_LEVEL_HIGH`（在 CPK-RA6M4 上，“高”电平对应于“开启”）。

下一步将是打开并启用连接到板上 S1 的 IRQ00。为此，请使用 IRQ FSP 驱动程序的打开和使能功能。完成后，初始化即完成。

```
g_external_irq00.p_api->open(g_external_irq00.p_ctrl,
                             g_external_irq00.p_cfg);
 g_external_irq00.p_api->enable(g_external_irq00.p_ctrl);
```

在 while(1) 循环内部，需要添加一些语句并删除 vTaskDelay(1); 语句。先使用函数调用将 led_level 的值写入用户 LED（红色）的 I/O 引脚的输出寄存器，然后执行相关语句切换该引脚的电平。有几种方法可以实现这一点。自行实现，回顾第 3 章的练习或查看本章结尾的代码。不要忘记 e² studio 的智能手册功能，它会提供很大帮助！

While(1) 循环中的最后一条语句是调用 xSemaphoreTake()，将信号量的地址和常量 portMAX_DELAY 作为参数。后一个参数将通知 RTOS 无限期地暂停线程，直到从 IRQ00 中断服务程序调用的回调函数中释放信号量为止。

最后要执行的操作是添加回调函数本身。该函数应尽可能短，因为它将在中断服务程序的上下文中执行。编写此函数十分简单：只需转到“Project Explorer”（项目资源管理器）中的“*Developer Assistance → LED Thread → g_external_irq00 External IRQ Driver on r_icu*”（开发人员帮助 → LED 线程 → r_icu 上的 g_external_irq00 外部 IRQ 驱动程序），然后将所出现列表末尾的回调函数定义拖放到源文件中。

```
void external_irq00_callback(external_irq_callback_args_t *p_args);
```

在回调函数内，添加以下两行代码：

```
FSP_PARAMETER_NOT_USED(p_args);
xSemaphoreGiveFromISR(g_sw1_semaphore, NULL);
```

第一行中的宏将告知编译器回调函数不使用参数 p_args，从而避免编译器发出警告，而第二行中的宏则在每次按下按钮 S1 时释放信号量。注意，必须使用 give 系列函数的中断保存版本，因为此函数调用发生在 ISR 的上下文内。此调用的第二个参数是 *pxHigherPriorityTaskWoken。如果可能有一个或多个任务由于信号量发生阻塞并等待该信号量变为可用状态，并且其中一个任务的优先级高于发生中断时执行的任务，则此参数将在调用 xSemaphoreGiveFromISR() 后变为 true。在这种情况下，应在退出中断之前执行上下文切换。由于在我们的示例中，没有其他任务依赖于此信号量，因此可以将此参数设置为 NULL。

完成所有代码编写后，单击“*Build*”（编译）图标（“锤子”），编译项目。如果编译后存在错误，请返回程序，借助“*Problems*”（问题）视图中显示的编译器反馈修复问题。

如果项目编译成功，请单击“*Debug*”（调试）图标旁的小箭头，选择“*Debug Configurations*”（调试配置），然后展开“*Renesas GDB Hardware Debugging*”（瑞萨 GDB 硬件调试）。选择 *MyRtosProject Debug_Flat*，或者为项目指定的名称，然后单击“*Debug*”（调试）。这样便可启动调试器。如果您需要更多相关信息，请回顾第 3 章中的相关部分。调试器启动并运行后，单击“*Resume*”（恢复）两次。现在程序正在执行，每次按下 CPK 上的 S1 时，用户 LED（红色）都应切换状态。

最后一点：在实际应用中，应执行错误检查以确保程序正确运行。为了清楚和简洁起见，本示例中将其省略。

```
 #include "led_thread.h"

 void led_thread_entry(void *pvParameters)
 {
FSP_PARAMETER_NOT_USED (pvParameters);
extern bsp_leds_t g_bsp_leds;
bsp_leds_t Leds = g_bsp_leds;

uint8_t led_level = BSP_IO_LEVEL_HIGH;

g_external_irq00.p_api->open(g_external_irq00.p_ctrl,
                             g_external_irq00.p_cfg);
g_external_irq00.p_api->enable(g_external_irq00.p_ctrl);

while (1)
    {
        g_ioport.p_api->pinWrite(&g_ioport_ctrl
                                 Leds.p_leds[BSP_LED_LED3],led_level);

        if (led_level == BSP_IO_LEVEL_HIGH)
        {
            led_level = BSP_IO_LEVEL_LOW;
        }
        else
        {
            led_level = BSP_IO_LEVEL_HIGH;
        }

        xSemaphoreTake(g_sw1_semaphore, portMAX_DELAY);
}
 }

 /* callback function for the SW1 push button; sets the semaphore */
 void external_irq00_callback(external_irq_callback_args_t *  p_args)
 {
   FSP_PARAMETER_NOT_USED(p_args);
   xSemaphoreGiveFromISR(g_sw1_semaphore, NULL);
 }
```

恭喜！

您已成功完成本练习！

本章要点：

- 通过使用全面的 API，可以轻松使用 FSP 的各个函数。
- FSP 将处理大多数与用户代码无关的内容。
- 使用 FreeRTOS™ 十分简单，因为 FSP 配置器的使用非常直观，添加线程和信号量也相当轻松。

5. 使用“灵活配置软件包”通过 USB 端口发送数据

本章内容基于《瑞萨 RA MCU 基础知识》中的章节 **10 使用“灵活配置软件包”通过 USB 端口发送数据**所作。

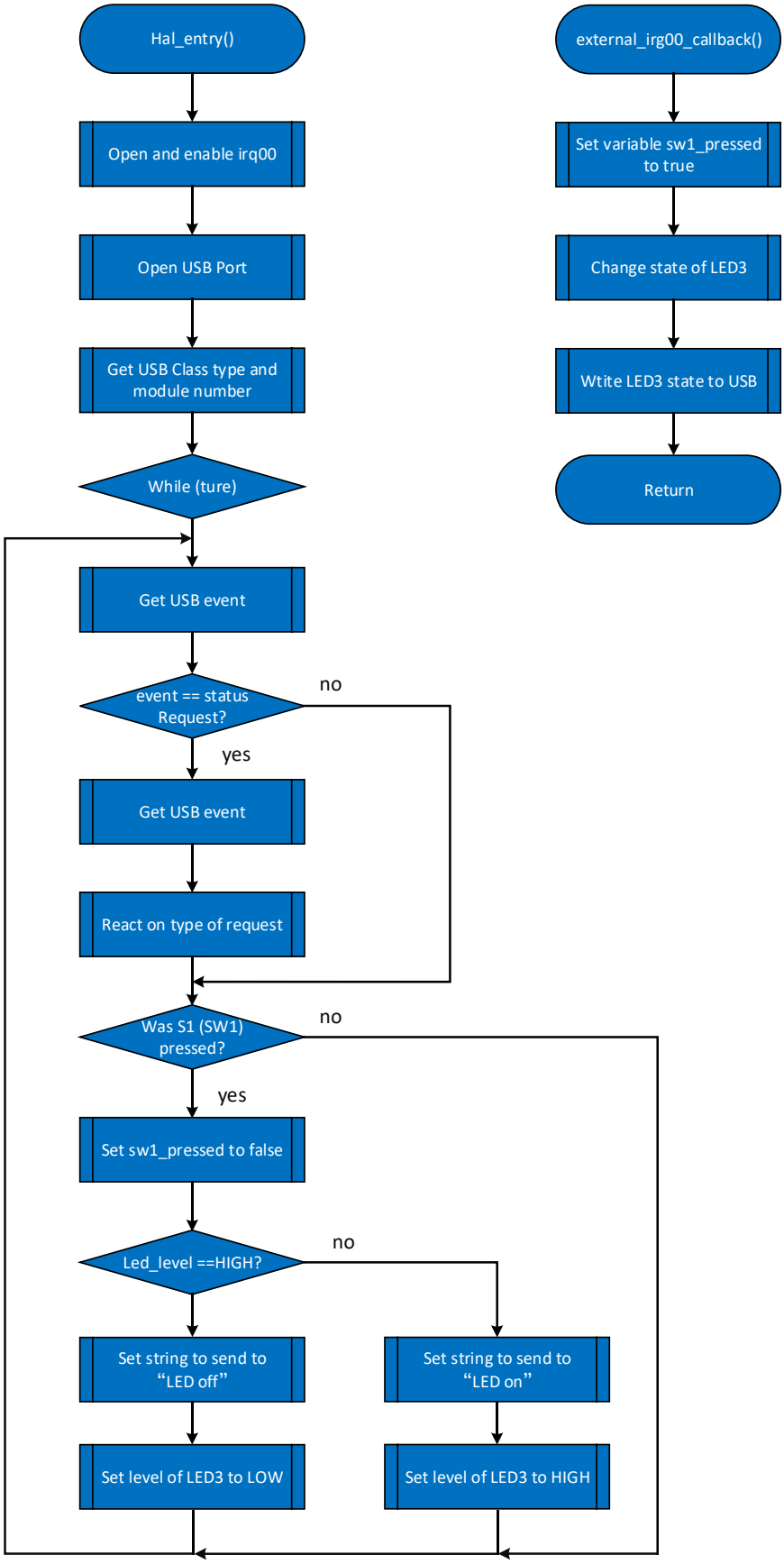
您将在本章中学到以下内容：

- 如何使用 RA 产品家族微控制器的“灵活配置软件包”的中间件来设置 USB 传输。
- 如何在主机工作站上接收 MCU 发送的数据。

在本部分，我们将使用瑞萨 RA 产品家族微控制器的“灵活配置软件包”(FSP) 的 USB 中间件，在每次按下用户按钮 S1 时，将用户 LED（红色） 的当前状态作为文本字符串通过 USB 端口发送到 Windows® 工作站。与第 4 章不同的是，我们在此实验中不使用实时操作系统和信号量，而使用全局变量来指示按钮开关已激活和红色 LED（红色）的状态已更改。

LED 状态（ON 或 OFF）更新、USB 端口的写操作，以及保存按钮按下时的信息的全局变量更新将在 IRQ00 的回调例程中完成。端口的写操作将触发 USB 传输，将 LED 的相关信息发送给主机。返回到 `hal_entry()` 函数内部的无限循环后，将处理 USB 事件，并通过将全局变量设置为“false”和将下一个字符串及下一个 LED 电平分配给各自的变量来准备 LED 状态的下次更新。***错误!未找到引用源。*** 详细描绘了该程序的流程和中断回调函数的流程。

该端口的大部分设置将在 FSP 配置器的图形界面中完成，应用程序程序员只需完成极少的编程工作。在执行该练习中的编程任务时，可再次体验到 FSP 给用户提供的便利，即便在构建如 USB 之类的复杂通信系统时也非常方便。


Hal_entry()
Open and enable irq00
Open USB Port
Get USB Class type and module number
While (ture)
Get USB event
event == status Request?
no
yes
Get USB event
React on type of request
Was S1 (SW1) pressed?
no
yes
Set sw1_pressed to false
Led_level ==HIGH?
no
Set string to send to “LED off”
Set string to send to “LED on”
Set level of LED3 to LOW
Set level of LED3 to HIGH
external_irg00_callback()
Set variable sw1_pressed to true
Change state of LED3
Wtite LED3 state to USB
Return

5.1 使用 FSP 配置器设置 USB 端口

如果在完成上次练习后已关闭 e² studio，请再次打开并创建一个新项目。到目前为止您应该已经掌握了 RA 的相关知识，这里将不再赘述每个步骤，因为大部分需要执行的任务在之前的实验中已经做过介绍。将新项目命名为 *CPK_RA6M4_USBProject*，在进入 *“Device and Tools Selection ”*（器件和工具选择）屏幕后，选择 *CPK-RA6M4* 作为电路板，我们将再次使用该评估板进行实验。在 *“Project Type Selection ”*（项目类型选择）页面，确保 *“Flat (Non-TrustZone) Project ”*（简单（非 TrustZone）项目）处于启用状态，并确保 *“RTOS Selection ”*（RTOS 选择）下的 *“No RTOS ”*（无 RTOS）条目已激活。最后，在 *“Project Template Selection ”*（项目模板选择）页面上选择 *“Bare Metal – Minimal ”*（裸机 – 最小化），然后单击 *“Finish ”*（完成）。

在项目配置器已创建项目并显示 FSP 配置器后，直接转到 *“Stacks ”*（堆）选项卡。首先，我们需要添加用于连接到用户按钮 S1 的外部中断的模块。在 *“HAL/Common Stacks ”*（HAL/通用堆栈）窗格上，单击 *“New Stack ”*（新堆），然后选择 *“Driver → Input → External IRQ Driver on r_icu”*（驱动程序 → 输入 → r_icu 上的外部 IRQ 驱动程序）。

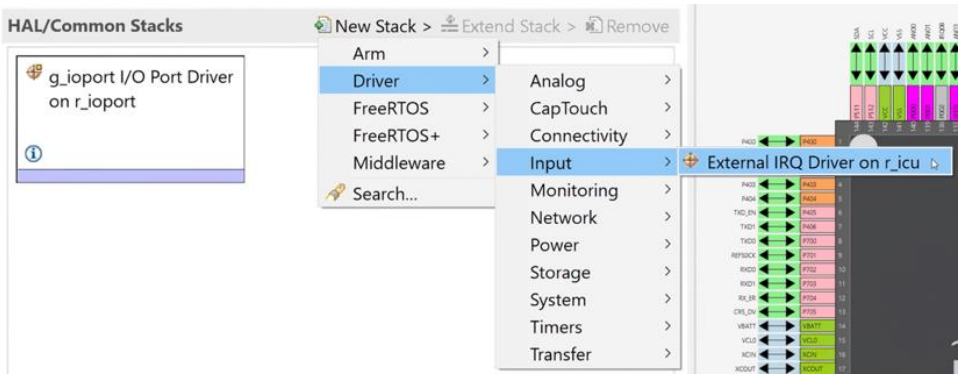


图 5-1：首先添加 S1 中断的驱动程序

在“Properties”（属性）视图中，将中断的 *“Name ”*（名称）修改为 *g_external_irq00*，确保它的 *“Channel ”*（通道）为 0，以作为中断使用的通道。启用 *“Digital Filtering ”*（数字滤波）并将 *“Trigger ”*（触发器）设置为 *“Falling ”*（下降）。这有助于消除开关的抖动。最后，需要提供用于该中断的回调函数的名称：将其命名为 *external_irq00_callback*，并将 *“Priority ”*（优先级）改为 *14*，因为我们要确保 USB 中断的优先级高于按钮（参见图 5-3）。

Property	Value
Parameter Checking	Default (BSP)
Module g_external_irq0 External IRQ Driver on r_icu	
Name	g_external_irq00
Channel	0
Trigger	Falling
Digital Filtering	Disabled
Digital Filtering Sample Clock (Only valid when Digital Filtering is Enabled)	PCLK / 64
Callback	external_irq00_callback
Pin Interrupt Priority	Priority 14
Pins	
IRQ00	P105

图 5-2：以上是 IRQ00 的必要设置

接下来，将 USB 外设通信设备类 (PCDC) 的中间件添加到系统中：创建新堆栈，并选择“Middleware → USB → USB PCDC driver on r_usb_pcdc”（中间件 → USB → r_usb_pcdc 上的 USB PCDC 驱动程序）（参见图 5-4）。

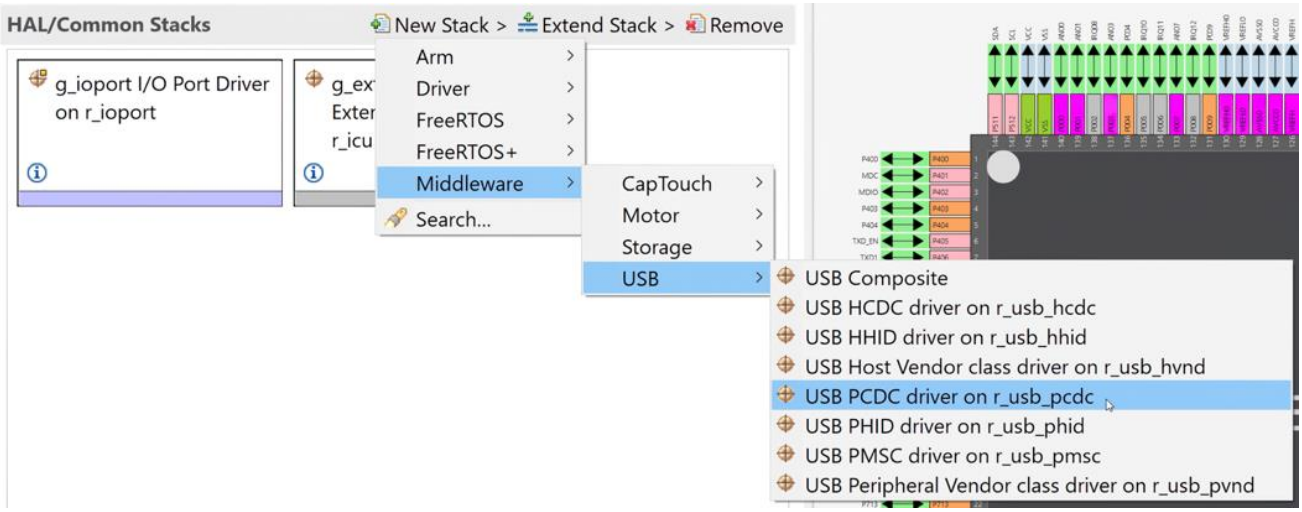


图 5-3：需要将 USB 外设通信设备类驱动程序的中间件添加到系统中

此操作将四个模块添加到项目中：用于全速 USB 端口的实际 PCDC 驱动程序（用于实现应用程序级 USB PCDC 接口），以及 r_usb_basic 上的基本 USB 驱动程序。堆中还显示两个具有粉红色横条的模块。这些模块用于添加可选的直接内存访问控制器 (DMAC) 驱动程序，以传输或接收数据。我们将使用 USB 写入 API 函数直接发送状态消息，因此无需添加它们。关于模块的其他色彩色条的含义，只需记住以下规则：灰色标记仅可由一个其他模块实例引用的模块实例，蓝色标记可由多个其他模块实例（甚至跨多个堆）引用的通用模块实例。通过彩色条中的小三角形，可以展开或折叠模块树。

将 USB 端口作为 PCDC 设备来实现，可以将 USB 端口用作虚拟 COM 端口，从而简化主机端的接收器设置，因为在注册到 Windows® 后，便可通过终端应用程序进行数据通信。这就是我们与评估板进行对话的方式。

添加所有堆栈后，“Stacks”（堆）窗格的外观如图 5-5 所示：

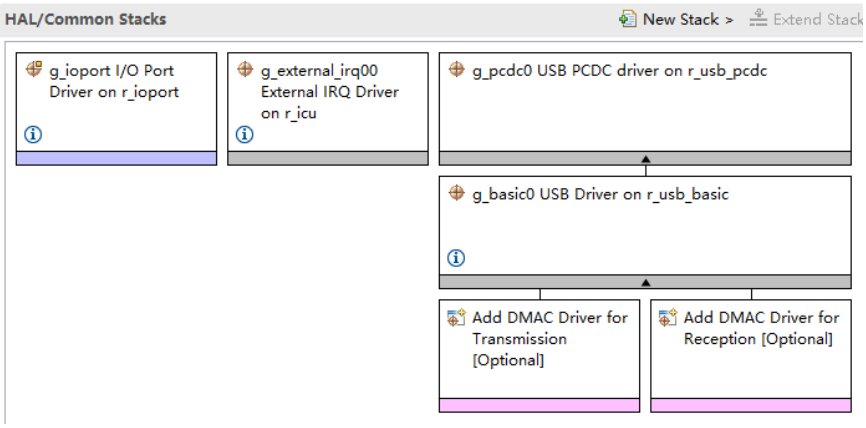


图 5-5：添加 USB 驱动程序后“堆”窗格的外观

只需要对基本 USB 驱动程序的“Properties”（属性）做出两处更改。突出显示“g_basic0 USB Driver on r_usb_basic”（r_usb_basic 上的 g_basic0 USB 驱动程序）模块，并在“Properties”（属性）视图的“Common”（通用）下，将“Continuous Transfer Mode”（连续传输模式）从“Disabled”（已禁用）切换为“Enabled”（已启用），将“DMA Support”（DMA 支持）从“Enabled”（已启用）切换为“Disabled”（已禁用）。记录“g_basic0 USB Driver on r_usb_basic”（r_usb_basic 上的 g_basic0 USB 驱动程序）部分的“USB Descriptor”（USB 描述符）的名称：g_usb_descriptor。稍后将创建一个具有该名称的结构，以描述系统 USB 的功能，因此应记住这个名称。图 5-6 显示了修改后模块的属性。

Property	Value
Common	
Parameter Checking	Default (BSP)
PLL Frequency	24MHz
CPU Bus Access Wait Cycles	9 cycles
Battery Charging	Enabled
Power IC Shutdown Polarity	Active High
Dedicated Charging Port (DCP) Mode	Disabled
Notifications for SET_INTERFACE/SET_FEATURE/CLEA	Enabled
Double Buffering	Enabled
Continuous Transfer Mode	Enabled
DMA Support	Disable
DMA Source Address	DMA Disabled
DMA Destination Address	DMA Disabled
Module g_basic0 USB Driver on r_usb_basic	
Name	g_basic0
USB Mode	Peri mode
USB Speed	Full Speed
USB Module Number	USB_IP0 Port
USB Device Class	Peripheral Communicatio
USB Descriptor	g_usb_descriptor
USB Compliance Callback	NULL
USBFS Interrupt Priority	Priority 12
USBFS Resume Priority	Priority 12
USBFS D0FIFO Interrupt Priority	Priority 12
USBFS D1FIFO Interrupt Priority	Priority 12
USBHS Interrupt Priority	Priority 12
USBHS D0FIFO Interrupt Priority	Priority 12
USBHS D1FIFO Interrupt Priority	Priority 12
USB RTOS Callback	NULL
USB Callback Context	NULL

图 5-6：进行必要更改后的连续传输设置。记录“USB Descriptor”（USB 描述符）的名称

图 5-7 显示了在完成所有更改后“Stacks”（堆）选项卡的外观。

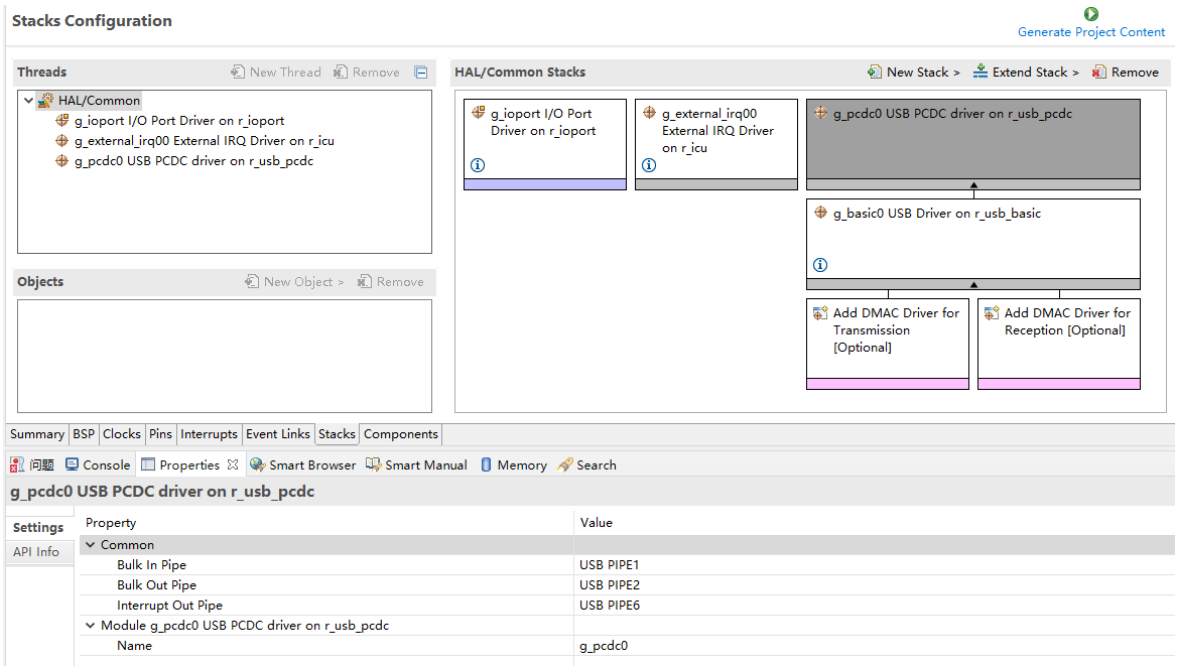


图 5-7：完成所有增加内容和修改后，“Stacks”（堆）选项卡的外观

在 *“Stacks ”*（堆）选项卡中完成所有设置后，现在需要设置 USB 端口的正确操作模式。为此，请切换到 *“Pins ”*（引脚）选项卡，在“Pin Selection”（引脚选择）窗格中，首先展开 *“Peripherals ”*（外设）下拉列表，然后展开 *“Connectivity: USB ”*（连接：USB）列表。在 *“Pin Configuration ”*（引脚配置）窗格中，将 *“Operation Mode ”*（操作模式）从 *“Custom ”*（自定义）修改为 *“Device ”*（器件），作为要使用的模式。注意，输入/输出引脚分配将相应改变。

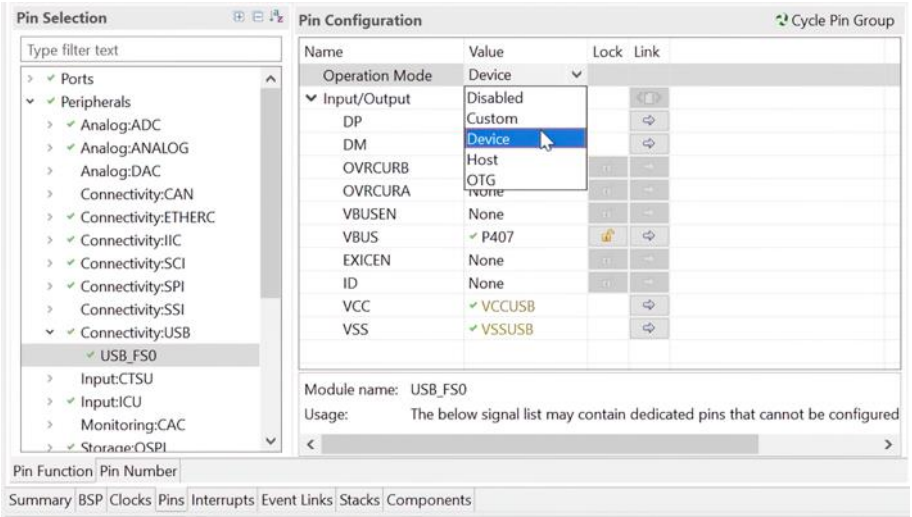


图 5-8：USB 端口将使用器件模式，请进行相应更改

现在还差一步就完成了端口的设置。最后一步是启用 *“USB clock (UCLK)”*（USB 时钟 (UCLK)），用作全速 (FS) USB 模块的工作时钟，并将其设置为所需的 48 MHz 频率。为此，激活 *“Clocks”*（时钟）选项卡，可通过该选项卡配置时钟生成电路。首先启用 USB 时钟，方法为：将靠近窗格底部的 *UCLK* 从 *“Disabled”*（已禁用）更改为 *“Enabled”*（已启用），并选择 *PLL2* 作为源。接下来，启用 *PLL2*，它是 USB 模块的专用 PLL，并选择*高速片上振荡器 (HOCO)* 作为源。将 *PLL2* 的*乘数值*更改为 *30*，以便将 PLL2 的频率更改为 240 MHz。再使用 UCLK 的除数 5，此时 UCLK 的频率已正确设置为 48 MHz。这里，采用 Arm® Cortex®-M33 内核的 RA MCU 系列凸显了巨大的优势：微控制器上还有第二个 PLL，可以将 USB 的时钟频率设置为 48 MHz，同时以最高速度 200MHz 运行 MCU。

如果不确定要更改哪些字段，请参见图 5-5。

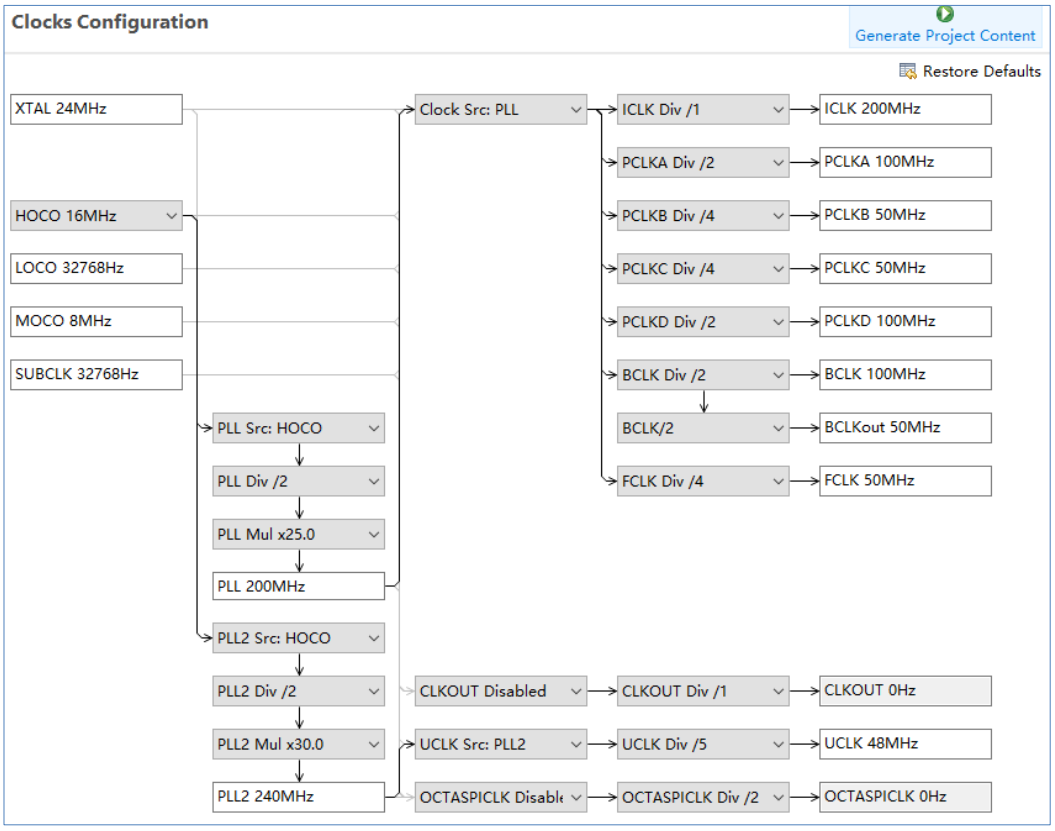


图 5-4：USB FS 需要使用 48 MHz 的时钟，因此需要相应地更改时钟生成电路。必要的更改已突出显示

至此，已经完成了必须在 FSP 配置器中进行的设置。保存配置，然后单击屏幕右上角的 *“Generate Project Content”*（生成项目内容）按钮，以提取文件并创建所需的设置。最后一步，再次切换到 C/C++ 透视图。

5.2 创建代码

现在，添加初始化 USB 端口和执行端口写入所需的代码。由于本练习需要输入大量的内容，建议您通过瑞萨网站从本手册对应的网页下载该实验的解决方案，这样只需按照说明进行操作，而无需手动输入代码。

如果决定自行编写所有内容，首先在 *“Project Explorer ”*（项目资源管理器）中通过双击打开 *hal_entry.c* 文件。为了确保程序正常运行，需要定义多个全局变量。首先，在 *hal_entry()* 函数的头部之前声明 USB 驱动程序状态的枚举。其类型应该是 `usb_status_t`，可以将其命名为 `usb_event`。接下来，添加一个 `usb_setup_t` 类型的结构（在 *r_usb_basic_api.h* 中进行声明），并将其命名为 `usb_setup`。我们稍后将在解码某些 USB 事件时使用该变量，该变量将在 USB 事件循环内进行初始化。

接下来，我们需要一个变量来保存 USB 模块的编号。将其设置为 `uint8_t` 类型，命名为 `g_usb_module_number`，并为其赋值“`0x00`”。最后，声明类型为 `usb_class_t` 的 USB 类类型的结构，将其命名为 `g_usb_class_type`，并为其赋值“`0x00`”。如果要了解我们使用的各种类型的详细信息，请参见《Renesas 灵活配置软件包 (FSP) 用户手册》，该手册可以从 FSP 的 GitHub® 网站下载。

添加这些内容后，此部分代码现在应如下所示：

```
/* Global variables for the USB */
usb_status_t usb_event;
usb_setup_t usb_setup;

uint8_t g_usb_module_number = 0x00;
usb_class_t g_usb_class_type = 0x00;
```

我们自己的代码也依赖于一些静态全局变量。请添加到 USB 全局变量下方：

```
/* Global variables for the program */
static char send_str[12] = { "LED on\n\r" };
static volatile uint8_t sw1_pressed = false;
static uint8_t led_level = BSP_IO_LEVEL_HIGH;
```

命名为 `send_str` 的字符数组用于保存我们要通过 USB 发送的文本。将其初始化为“`LED on\n\r`”，因为将 LED3 切换到“ON”后，将首次使用该变量。下一个变量为 `sw1_pressed`，其类型为 `uint8_t`，并需要声明为 `volatile`，因为其值在用户按钮 S1（SW1） 的回调例程中将更改为 `true`。默认情况下，其值为 `false`，将由 IRQ00 中断的回调例程设置为 `true`，表示已按下该按钮，因此通知主程序该事件已发生。

如果没有将该变量声明为 `volatile`，C 编译器的优化程序可能不会在每次使用该变量时重新读取其值，因此 `hal_entry()` 内部的循环可能无法识别到更改。

第三个变量用于保存 LED3 的电平，在启动时应该初始化为 BSP_IO_LEVEL_HIGH。每次激活 S1（SW1）时，切换该变量的值。

至此，我们已经声明了所有的全局变量，可以继续编写 hal_entry() 函数内的代码。首先，我们需要一个静态变量，用于保存虚拟 UART 通信端口的设置，如比特率、停止位和数据位的数量以及奇偶校验类型。该变量的类型应该是 usb_pcdc_linecoding_t，建议将其命名为 g_line_coding。将“在此添加您自己的代码”占位符替换为声明。稍后将在 USB 事件处理程序循环中初始化该变量。

```
static usb_pcdc_linecoding_t g_line_coding;
```

接下来，编写代码以打开并启用外部 IRQ00，将其连接到评估板的 S1。与第 4 章一样，使用 IRQ FSP 驱动程序的相应函数：

```
g_external_irq00.p_api->open (g_external_irq00.p_ctrl,
                              g_external_irq00.p_cfg);
g_external_irq00.p_api->enable (g_external_irq00.p_ctrl);
```

启用中断后，需要打开 USB 并获取类类型和模块编号。为此，使用 *r_usb_basic 上的 g_basic0 USB 驱动程序*模块的相关函数，并将控制结构传递给这些函数，将引用传递给配置结构（适用于 Open() 函数）和相关的变量。温馨提示，e² studio 中的代码补全功能和*开发人员帮助*可帮助您编写这些代码行。

```
R_USB_Open (&g_basic0_ctrl, &g_basic0_cfg);
R_USB_ClassTypeGet (&g_basic0_ctrl, &g_usb_class_type);
R_USB_ModuleNumberGet (&g_basic0_ctrl, &g_usb_module_number);
```

中断和 USB 端口的初始化现已完成。接下来编写的所有代码都应该放置在 while(1) 循环内，因为这部分程序将循环执行。首先，我们编写用于获取和处理端口的 USB 相关事件的代码。USB 驱动程序关联多个事件，但为了简洁起见，仅处理 USB_STATUS_REQUEST 事件。如果要全面了解事件处理程序，请参见《灵活配置软件包 (FSP) 用户手册》中的 USB 外设通信设备类 (r_usb_pcdc) 文档。在此，可以找到此类处理程序的代码示例以及流程图。

现在，您的第一个任务是通过调用 R_USB_EventGet() 函数来初始化 usb_event 变量，然后编写处理程序，只有发生 USB_STATUS_REQUEST 事件时才能执行该处理程序。在 if – then – else 结构中，首先设置 USB 端口，然后确定是否请求线路设置。如果是，通过传递 g_line_coding 变量来配置虚拟 UART 设置。

如果否，则查询主机是否要接收 UART 设置。如果是，请将其发送给主机。最后，如果发生事件，在此不进行处理，直接确认。

下面是我们的处理程序版本的完整代码：

```
/* Obtain USB related events */
 R_USB_EventGet (&g_basic0_ctrl, &usb_event);

/* USB event received by R_USB_EventGet */
if (usb_event == USB_STATUS_REQUEST)
{
    R_USB_SetupGet (&g_basic0_ctrl, &usb_setup);

if (USB_PCDC_SET_LINE_CODING == (usb_setup.request_type
                                    & USB_BREQUEST))
{
        /* Configure virtual UART settings */
        R_USB_PeriControlDataGet (&g_basic0_ctrl,
                          (uint8_t*) &g_line_coding,
LINE_CODING_LENGTH);
}
else if (USB_PCDC_GET_LINE_CODING == (usb_setup.request_type
                                       & USB_BREQUEST))
{
        /* Send virtual UART settings back to host */
        R_USB_PeriControlDataSet (&g_basic0_ctrl,
                                  (uint8_t*) &g_line_coding,
                                  LINE_CODING_LENGTH);

}
else if (USB_PCDC_SET_CONTROL_LINE_STATE ==
(usb_setup.request_type
                                                    &
USB_BREQUEST))
{
        /* Acknowledge all other status requests */
        R_USB_PeriControlStatusSet (&g_basic0_ctrl,
USB_SETUP_STATUS_ACK);
}
else
{
}
}
```

可以看到，在处理程序中 LINE_CODING_LENGTH 出现两次。由于我们还没有定义 LINE_CODING_LENGTH 的值，请返回到文件的顶部，并将其定义为无符号值 0x07。

```
#define LINE_CODING_LENGTH          (0x07U)
```

返回到 while(1) 循环中，添加在激活 S1 后更改 LED3 电平的代码，以 sw1_pressed 的 true 值表示。类似于第 4 章中写入的内容，但此时需要将要通过 USB 发送的字符串复制到 send_str 变量，并将 s1_pressed 变量设置为 false：

```
if (sw1_pressed == true)
{
sw1_pressed = false;

if (led_level == BSP_IO_LEVEL_HIGH)
{
        strcpy (send_str, "LED off\n\r");
        led_level = BSP_IO_LEVEL_LOW;
}

else
{
        strcpy (send_str, "LED on\n\r");
        led_level = BSP_IO_LEVEL_HIGH;
}
}
```

最后要添加的代码是用于外部 IRQ00 的回调函数的代码。将其放置在 hal_entry () 函数的括号后面。通过复习第 4 章，了解回调函数的一些详细信息。首先，需要导入 g_bsp_leds 结构，并用其初始化我们的本地 Leds 变量。然后将 sw1_pressed 设置为 true，表示事件已发生，接下来将新值写入引脚寄存器。最后，利用 *r_usb_basic* 模块的 R_USB_Write() API，通过 USB 端口发送该字符串。

```
void external_irq00_callback(external_irq_callback_args_t
*p_args)
{
FSP_PARAMETER_NOT_USED(p_args);
extern bsp_leds_t g_bsp_leds;
bsp_leds_t Leds = g_bsp_leds;

sw1_pressed = true;
g_ioport.p_api->pinWrite (&g_ioport_ctrl,
                          Leds.p_leds[BSP_LED_LED3],
                          led_level);

R_USB_Write (&g_basic0_ctrl,
             (uint8_t*) send_str,
             ((uint32_t) strlen (send_str)),
             (uint8_t) g_usb_class_type);
}
```

还记得 USB 描述符 `g_usb_descriptor` 吗？现在该描述符将发挥作用。USB 需要有关器件、其配置和供应商信息的准确描述。该文件十分复杂，具有长达 484 行代码。有关该描述符的说明，请参阅《FSP 用户手册》的 *r_usb_basic* 部分，有关如何构建该描述符的详细说明，请参见通用串行总线规范 2.0 版 (**http://www.usb.org/developers/docs/**)。

但这里有两个捷径：一个是在本手册的网站上下载本手册练习的源文件（**www.renesas.com/ra-book**）。另一个是使用 FSP 配置器放置在项目 *ra* 目录下的模板。其名称为 *r_usb_pcdc_descriptor.c.template*，可以在“项目资源管理”中转到 *ra* → *fsp* → *src* → *r_usb_pcdc* 文件夹进行访问（参见图 5-6）。将该文件复制到 *hal_entry.c* 所在的 *src* 文件夹中，并将其重命名为 *r_usb_descriptor.c*。修改*供应商 ID* 和*产品 ID*，以便与您自己的产品 ID 相匹配。如果尚未获得这些数据，暂时使用值 `0x045BU` 和 `0x5310U`。到这一步已经完成了要进行的设置和要编写的代码。

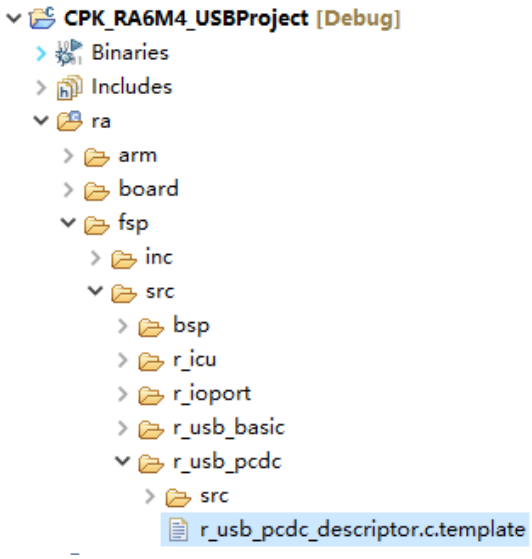


图 5-5：FSP 配置器自动创建 USB 描述符的模板

最后还需要编译项目。第一次执行此操作需要较长时间，因为需要对项目中包含的所有 FSP 模块的代码进行编译。在项目编译完成后，如果没有任何错误和警告，即可连接 CPK-RA6M4 评估板并启动调试会话。打开 *“Debug”*（调试）透视图，双击 *“Resume”*（恢复）以启动程序。作为快速测试手段，按一次 S1，以查看 用户 LED（红色） 是否切换。

5.3 在主机端设置接收器

在程序运行的情况下，将第二根 USB type A 转 Micro-B 电缆连接到评估板的*系统控制和生态系统访问*区域左下方标有 J9 的 USB 端口。将另一端插入 Windows® 工作站，稍等片刻，直到 Windows® 识别该电路板，对其进行枚举并安装驱动程序。

启动终端仿真器程序。在本练习的开发过程中，用到了 Tera Term（可从 **https://ttssh2.osdn.jp/** 下载），它是一款非常实用的工具。在 Tera Term 中，可以看到列出的 CDC 串行端口。在图 5-7 中显示为 COM3，但在其他 PC 上可能有所不同。如果不确定，使用 Windows® 的“Device Manager”（设备管理器）来查找电路板所连接的端口。

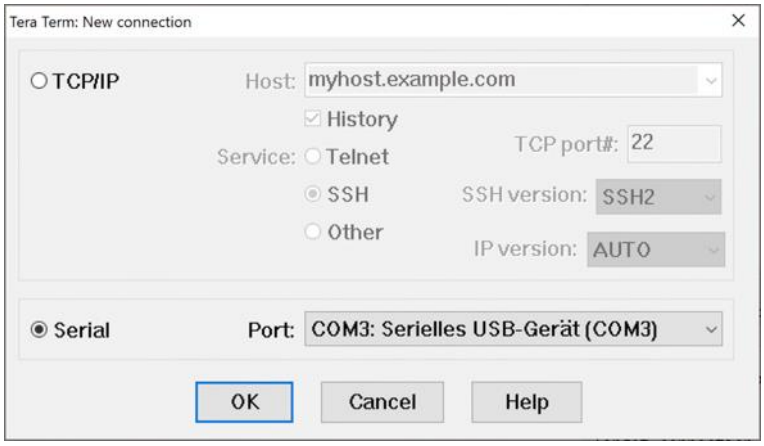


图 5-6：如果 Windows® 正确识别该电路板，它将在 Tera Term 中列为串行连接

如果没有列出该电路板，或者“Device Manager”（设备管理器）指示错误，则驱动程序可能有问题。请参见瑞萨知识库中有关此主题的最新支持条目以解决此问题：
https://en-support.renesas.com/knowledgeBase/18959077。

在已建立连接并运行 Tera Term 的情况下，多次按下 S1，应该可以看到用户 LED （红色）切换状态，其输出到终端的状态如图 5-8 所示。

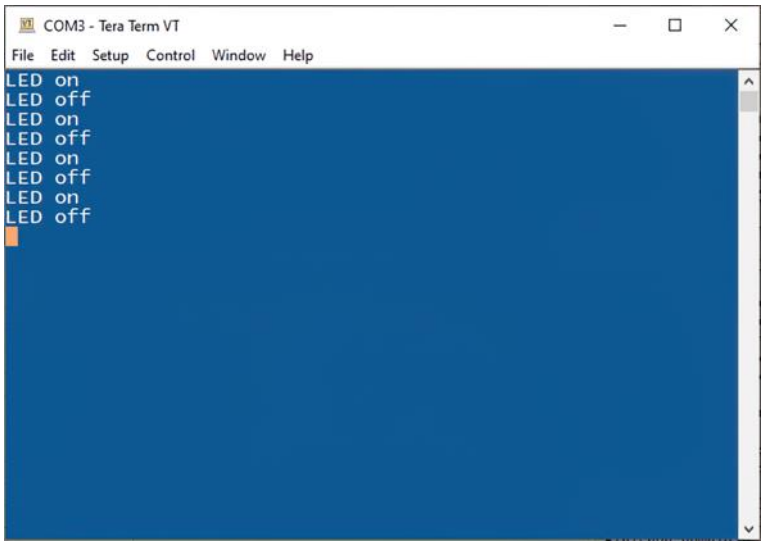


图 5-7：在传输运行的情况下，每次按下 S1 时，终端程序都会显示 用户 LED（红色） 的状态

```
#include “hal_data.h”

FSP_CPP_HEADER
void R_BSP_WarmStart(bsp_warm_start_event_t event);
FSP_CPP_FOOTER

#define LINE_CODING_LENGTH          (0x07U)

/* Global variables for the USB*/
usb_status_t usb_event;
usb_setup_t usb_setup;
```

```
uint8_t g_usb_module_number = 0x00;
usb_class_t g_usb_class_type = 0x00;

/* Global variables for the program */
static char send_str[12] = { "LED on\n\r" };
static volatile uint8_t sw1_pressed = false;
static uint8_t led_level = BSP_IO_LEVEL_HIGH;

/**********************************************************************************************
/
/* main() is generated by the RA Configuration editor and is used to generate threads if
*/ /* an RTOS is used.  This function is called by main() when no RTOS is used.
*/
/**********************************************************************************************
/
void hal_entry(void) {
static usb_pcdc_linecoding_t g_line_coding;

/* open and enable irq00 */
g_external_irq00.p_api->open (g_external_irq00.p_ctrl, g_external_irq00.p_cfg);
g_external_irq00.p_api->enable (g_external_irq00.p_ctrl);

/* Open USB instance */
R_USB_Open (&g_basic0_ctrl, &g_basic0_cfg);

/* Get USB class type */
R_USB_ClassTypeGet (&g_basic0_ctrl, &g_usb_class_type);

/* Get module number */
R_USB_ModuleNumberGet (&g_basic0_ctrl, &g_usb_module_number);

while (true)
{
        /* Obtain USB related events */
        R_USB_EventGet (&g_basic0_ctrl, &usb_event);

        /* USB event received by R_USB_EventGet */
        if (usb_event == USB_STATUS_REQUEST)
        {
            R_USB_SetupGet (&g_basic0_ctrl, &usb_setup);

            if (USB_PCDC_SET_LINE_CODING == (usb_setup.request_type & USB_BREQUEST))
            {
                /* Configure virtual UART settings */
                R_USB_PeriControlDataGet (&g_basic0_ctrl,
                                          (uint8_t*) &g_line_coding, LINE_CODING_LENGTH);
            }
            else if (USB_PCDC_GET_LINE_CODING == (usb_setup.request_type & USB_BREQUEST))
            {
                /* Send virtual UART settings back to host */
                R_USB_PeriControlDataSet (&g_basic0_ctrl,
                                          (uint8_t*) &g_line_coding, LINE_CODING_LENGTH);

            }
            else if (USB_PCDC_SET_CONTROL_LINE_STATE == (usb_setup.request_type
                                                         & USB_BREQUEST))
            {
                 /* Acknowldedge all other status requests */
                 R_USB_PeriControlStatusSet (&g_basic0_ctrl, USB_SETUP_STATUS_ACK);
            }
            else {
```

```
            }
        }

        if (sw1_pressed == true)
        {
            sw1_pressed = false;

            if (led_level == BSP_IO_LEVEL_HIGH)
            {
                 strcpy (send_str, "LED off\n\r");
                 led_level = BSP_IO_LEVEL_LOW;
            }

            else
            {
                strcpy (send_str, "LED on\n\r");
                led_level = BSP_IO_LEVEL_HIGH;
            }
        }
}

#if BSP_TZ_SECURE_BUILD
/* Enter non-secure code */
R_BSP_NonSecureEnter();
#endif
}

/**********************************************************************************************
/
/* callback function for the S1 push button; writes the new level to LED3 and sends it
*/
/* through the USB to the PC
*/
/**********************************************************************************************
/
void external_irq00_callback(external_irq_callback_args_t *p_args)
{
/* Not currently using p_args */
FSP_PARAMETER_NOT_USED(p_args);
extern bsp_leds_t g_bsp_leds;
bsp_leds_t Leds = g_bsp_leds;

sw1_pressed = true;
g_ioport.p_api->pinWrite (&g_ioport_ctrl, Leds.p_leds[BSP_LED_LED3], led_level);

R_USB_Write (&g_basic0_ctrl,
                 (uint8_t*) send_str,
                 ((uint32_t) strlen (send_str)),
                 (uint8_t) g_usb_class_type);
}
```

恭喜！

您已成功完成本练习！也完成了 CPK-RA6M4 评估板的入门操作！

本章要点：

- 使用 FSP 配置器和 USB 中间件便于增加对 USB 端口的支持。
- 要进行 USB 传输，必须具有 USB 描述符文件。

6. 《CPK-RA6M4 评估板入门》的文件列表

文件类型	内容	出现位置	文件名/文件夹名
文件（PDF）	用户手册	-	r01qs0056cc0100-cpk-ra6m4-quickguide.pdf
PACK 文件	用于导入 BSP 的文件	第 1 章	BSP File_CHN_3.1.0.rar
样例程序	基本样例程序	第 2 章	CPK_RA6M4_BlinkyProject.rar
样例程序	LED 闪烁样例程序	第 3 章	CPK_RA6M4_ExampleProject.rar
样例程序	FreeRTOS 样例程序	第 4 章	CPK_RA6M4_RtosProject.rar
样例程序	USB 样例程序	第 5 章	CPK_RA6M4_USBProject.rar

7. 参考文献

《瑞萨 RA MCU 基础知识》
（最新版本请从瑞萨电子网页上取得）

技术信息/技术更新
（最新信息请从瑞萨电子网页上取得）

网站和咨询窗口

RA 产品家族网站

- http://www.renesas.com/ra

瑞萨学院网站

- https://academy.renesas.com

瑞萨 Rulz 中文论坛

- https://japan.renesasrulz.com/rulz-chinese/

修订记录

Rev.	发行日	修订内容	
		页	要点
1.00	2021.07	—	初版发行