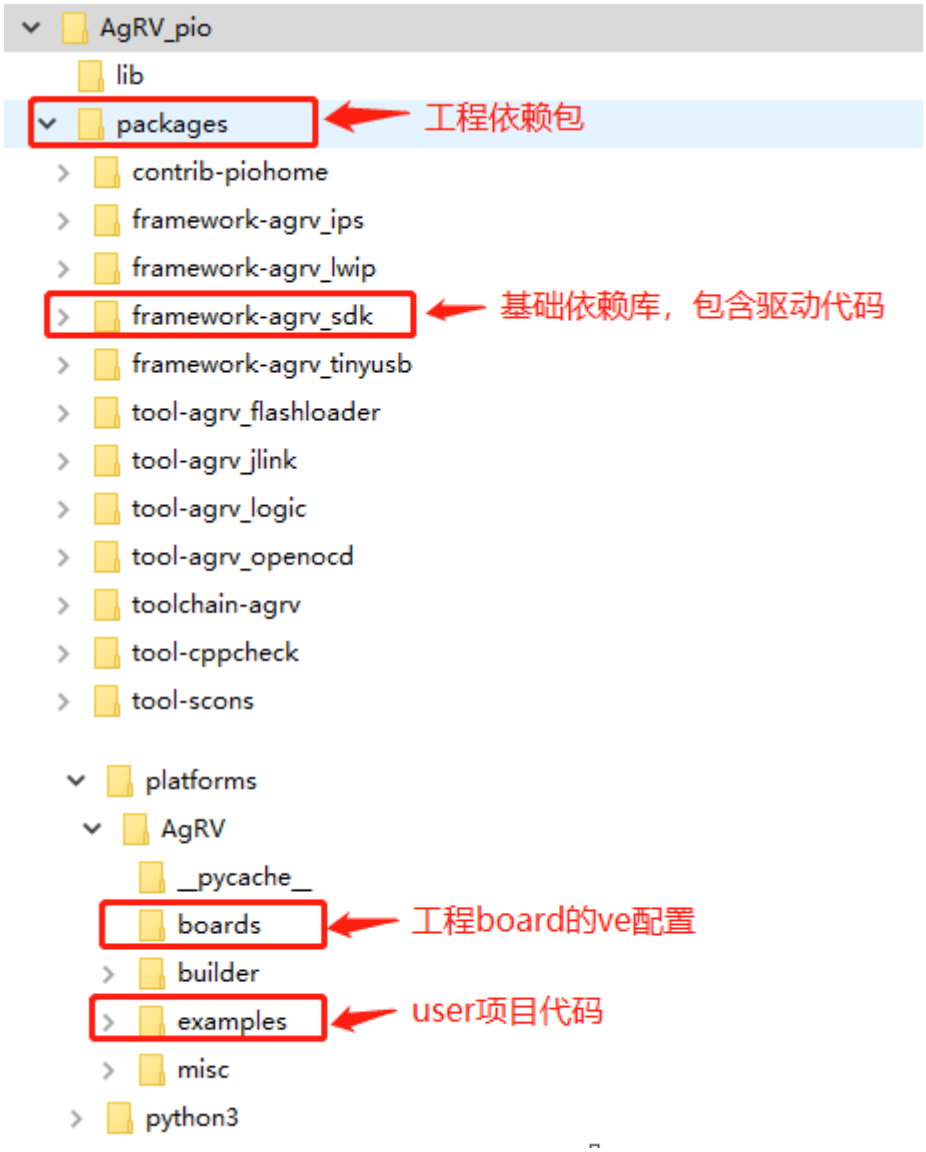
AGM 使用入门

目录:

阅读本文前，请参照《AGM MCU IDE 使用记录.pdf》，先搭建好 AGM 运行环境。

一、了解工程目录结构：

SDK 解压后的目录结构如下：



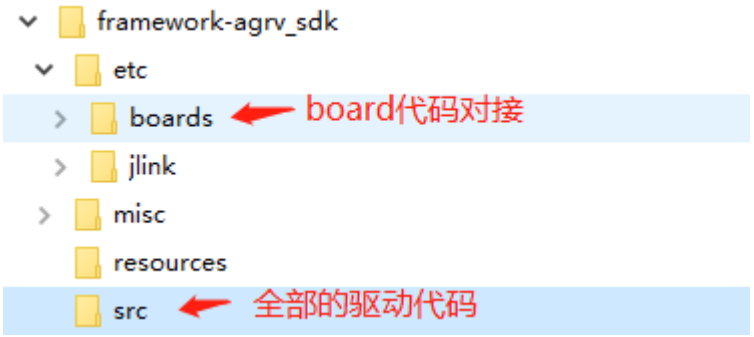
以上红色标出的几个目录，是在后续编写代码时常用的几个目录。

其中：

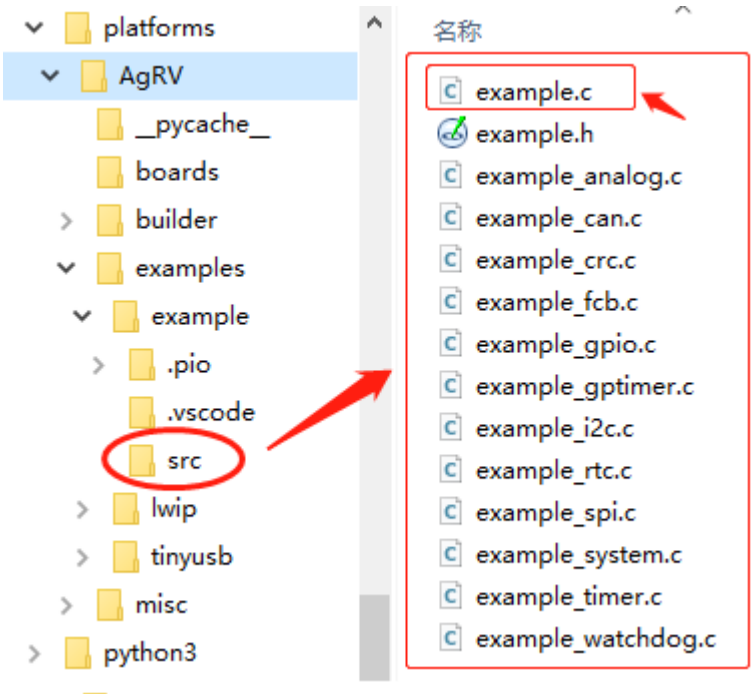
framework-agrv_sdk 是整个 SDK 的基础支撑部分。包含全部的驱动代码、board 代码、输出重定向代码。该 framwork 是用户在创建自己工程时必须依赖的 framwork。

除了 agrv_sdk，还有几个已经集成进 SDK 的 framwork，包括：ips/lwip/usb。用户可根据自己需求决定是否使用它们。

展开 framework-agrv_sdk 的目录，可以看到：



再看 examples 下，是全部的调用驱动的样例代码：



除了以上代码部分，还有两处重要的配置文件：

1. agrv2k_x0x.ve 芯片配置：

   位于路径：AgRV_pio\platforms\AgRV\boards。

   该配置文件中，配置芯片 fpga 中映射出来的各个 IO 端口。

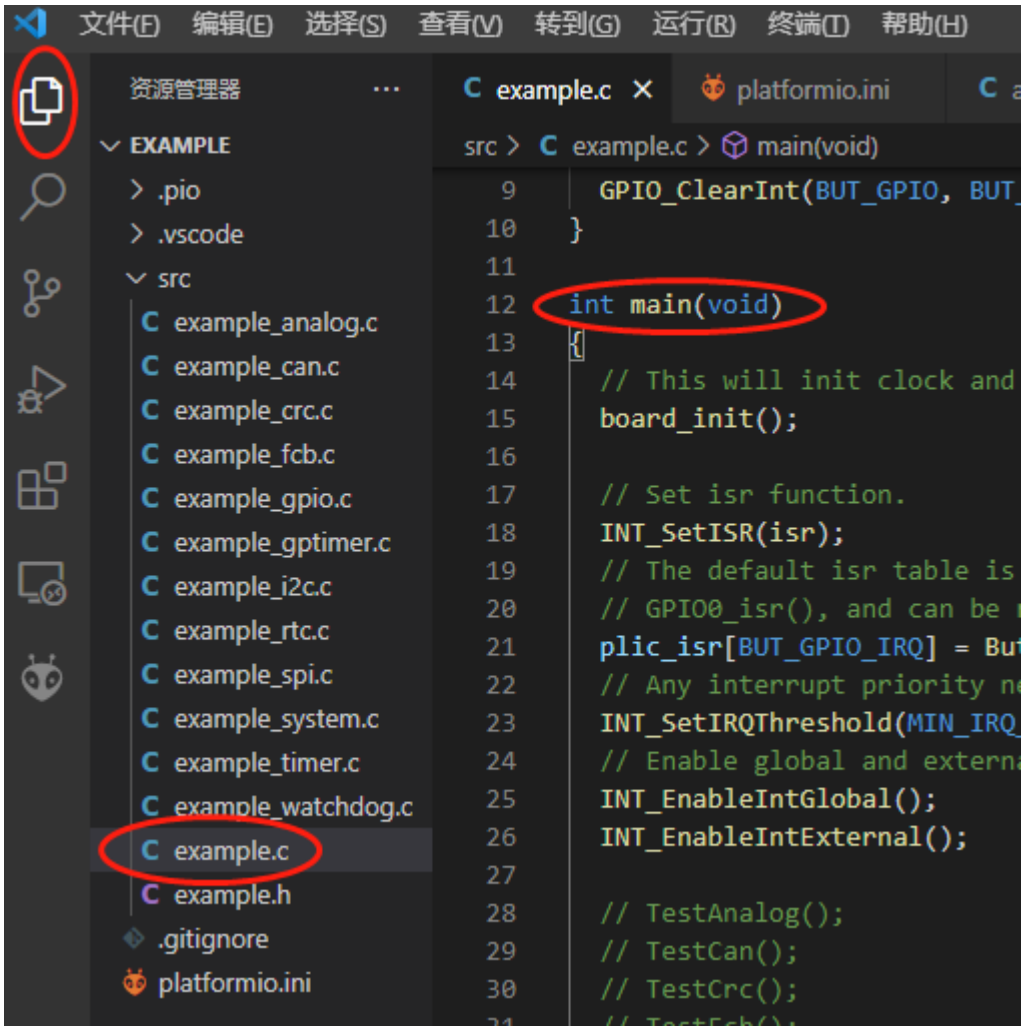
2. platformio.ini 工程配置：

   位于路径：AgRV_pio\platforms\AgRV\examples\example

   该文件是工程配置文件，里边定义 IDE 的编译/烧录/仿真及工程宏的选项。

二、初识 IDE 中的项目：

看完目录结构，再来看 Demo 工程。



AGM 是基于 VSCode 来搭建的环境，编辑、调试、烧录也都基于此。

在 Demo 里，example.c 中的 main 总领了全部的驱动测试入口；

board_init 函数中，会进行时钟初始化、GPIO 初始化、log 串口初始化；

INT_SetISR 是设置系统总中断入口（注：RISC 和 ARM 中断管理是不同的）

INT_EnableInt_xxxx 是使能子级中断；

在 board_init 中初始化完串口后，代码中的 printf，将重定向到该串口输出。具体使用哪个串口输出，是通过 ve 配置中的 logger_if 字段配置的。

这里先有个整体印象，后续会继续介绍。

三、认识 platformIO：

在开始正式的编码和配置前，先了解下 platformIO 是什么。

在前边安装完 VSCode 后，首先下载的插件就是 platformIO。就是说，platformIO 是 VScode 的一款第三方插件，对于开发者来说，它也是基于 VSCode 之上运行的一套 IDE 环境。

platformIO 是一套开放的 Iot 集成环境平台，允许芯片厂商通过配置和对接，方便的实现芯片开发环境。

也就是说，VSCode + platformIO + 芯片厂商对接 = 该芯片的 IDE 编译环境。

在嵌入式开发中，大家比较熟悉的 IDE 可能会有 Keil、IAR。而上边构建出来的 IDE 环境就是类似 Keil、IAR 的一整套环境。

依托功能强大的 VSCode 和 platformIO，芯片厂商可以通过少量的工作就可以构建出功能比 Keil 和 IAR 更强大的 IDE，这是芯片供应商的福音。

而 AGM 的开发环境，就是这么建立的一套 IDE 环境。

既然是基于 platformIO 平台，那么项目中大多数的配置就是围绕 platformIO 来展开的。后边会逐步讲到。

更多信息：

platformIO 的定位是新一代的 IoT 集成开发环境。它是基于 VSCode 的一款插件。VSCode 这款强大的文本编辑器辅以 PlatformIO 插件就可以化身为一款强大的 MCU 开发环境，支持绝大多数流行的单片机平台。

我们知道，嵌入式 Iot 开发中，最让人不舒服的就是不同厂家的芯片要使用不同的集成开发环境。例如：STM32 要使用 Keil 或 IAR，Arduino 默认使用自家 Arduino 开发环境，ESP32 要使用 linux 环境或者在 windows 下部署 eclipse 再用交叉编译。那么，有没有一个 IDE 可以大一统起来，集成大多数常用的芯片和模块的开发任务，只要配置完成之后就一劳永逸的呢？没错，那就是 PlatformIO。PlatformIO 试图整合起目前所有主流的硬件平台：TI/ST/EspressIf/Intel/Silicon/... 等，并且提供更便利的接口和更友好的交互，以提高开发效率。

关于 platformIO，有兴趣的可以去官网获取更多的信息：https://platformio.org/

前边说到，每一款芯片在 PlatformIO 中需要配置，按照 PlatformIO 的格式配置后，才能被正常使用（platformio.ini 有大量的标准的控制选项，可进入官网查看）。

官方配置：https://docs.platformio.org/en/latest/projectconf/index.html#projectconf

在实际使用中，除了官方标准配置（编译、烧录）外，芯片方也会在这个开放平台上自定义一些自己特有的配置项。

四、工程配置：

大致了解完项目结构，接下来了解工程配置。（fpga 不在本文介绍）

两部分工程配置：**platformio.ini 和 agrv2k_103.ve**

其中:

platformio.ini 配置 工程的编译、烧录、仿真的选项；

与 IDE 相关。

agrv2k_103.ve 配置 芯片系统时钟、芯片 IO 引脚映射 等与 fpga 相关的属性；

该文件中的配置是要被写入到 flash 中的，做为该芯片独特的配置。

platformio.ini 配置项：

从名字就可以看出，platformio.ini 是应用到 platformIO 的配置。

基于 platformIO 的项目，都必须配置一个 platformio.ini 文件，文件中对必要的配置字段进行赋值，从而告诉 platformIO 如何对该项目进行编译、烧录和仿真。

除了这些标准的配置字段，芯片厂商还可以定义自己的配置字段，然后在编译时自主使用。

先看 Demo 中的配置项：

boards_dir = boards //指定 board 对应的工作路径（用于代码编译的 path）

board = agrv2k_103 //使用 boards_dir 路径下的哪个硬件版本

//注：boards_dir 和 board 共同组成了完整 path。

//如：以上对应路径\packages\framework-agrv_sdk\etc\boards\agrv2k_103。

framework = agrv_sdk, agrv_lwip //使用工程中的哪些库

//工程中默认带的库有：sdk、lwip、tinyUSB、Ips。

//使用多个库时名字中间加逗号隔开。

program = agm_loraGtw //目标工程名

board_logic_ve = D:\AgRV_pio\platforms\AgRV\boards\agrv2k_103.ve

//工程使用到的 ve 配置文件（要烧写到 flash 中的）

//如果用相对路径，是相对于 platform.ini 的文件路径（即：工程路径）

//注：当前版本必须用全路径（暂不能使用相对路径）

src_dir = user //参与编译的 c 文件基目录（路径相对于工程路径）

include_dir = user //参与链接的 h 文件基目录（路径相对于工程路径）

src_filter = "-<*> +<*.c> +<print/*.c> " //参与编译的 c 文件路径列表

//*用于通配，+增加，-去除。路径基于上边 src_dir 的基路径

src_build_flags = -Iuser -Iuser/print //头文件路径列表

//-I 后边是一个个文件夹，各项之间用空格来分开

logger_if = UART0 //芯片串口输出 log 用的串口号（对应代码中 printf 函数）

//串口的波特率在用户程序中初始化串口时设置。

//注意：UART0 同时是代码烧录的指定串口

monitor_port = COM3 //platform monitor 功能对接的端口，usb 的话填 usb 口

monitor_speed = 57600 //monitor 的速度（如：串口的 115200/57600/...）

//monitor 功能是把配置端口收到的信息重映射到 VSCode 终端窗口输出

//更多用法参考：pio device monitor — PlatformIO latest documentation

upload_port = COM3 //串口烧录时 PC 端接的串口号（usb 烧录的话填 usb 口）

upload_protocol = jlink-openocd //烧录固件的工具，串口要填： serial

//upload 烧录程序时使用的方式和端口号

debug_tool = jlink-openocd　　//jtag 的工具，目前只能选 jlink-openocd

debug_speed = 10000　　//jlink 的数据速度

//jlink 在线跟踪时的设置

以上配置项在后续使用中会陆续介绍。

更多的配置项，可自行参考 platformio 官方文档：
https://docs.platformio.org/en/latest/projectconf/index.html#projectconf

agrv2k_103.ve 配置项：

AGM 芯片是支持 fpga 的，这个.ve 文件就是配置 fpga 芯片管脚定义的。

如下图，该 ve 是通过 platform.ini 配置的。这里关联后，通过命令来烧录配置项时，才能把 ve 文件中的配置烧录到 flash 中去。

```
framework = agrv_sdk
program = agm_example

board_logic_ve = D:\AgRV_pio\platforms\AgRV\boards\agrv2k_103.ve
```

ve 文件内容：

SYSCLK 100　#系统时钟频率，M 为单位
HSECLK 25　　#外部时钟频率，M 为单位

UART0_UARTRXD PIN_69　#串口 0 的收引脚 ----目前被用于 log 输出。
UART0_UARTTXD PIN_68　#串口 0 的发引脚

GPIO6_2 PIN_23　#IO_Button1
GPIO6_4 PIN_24　#IO_xxxx

……

……

关于 GPIOx_y 的说明：

对于 100pin 的芯片，芯片内共有 10 组 GPIO（GPIO0 – GPIO9），每组 8 个 IO（0~7）。所以，可用的对外映射到 PIN 的 GPIO 共有 80 个。

这里的 GPIOx_y 表示的是第 x 组第 y 个 IO。

除了 GPIO，更多配置定义参考文档《AGRV2K_逻辑设置.pdf》

PIN_XX 对应芯片的引脚，每一个 GPIOx_y 可以配置映射到任意一个引脚。

64pin 和 100pin 的封装，模拟输出引脚在对应上是不同的，因而，使用时最好在各自的模板上修改，不要混用。

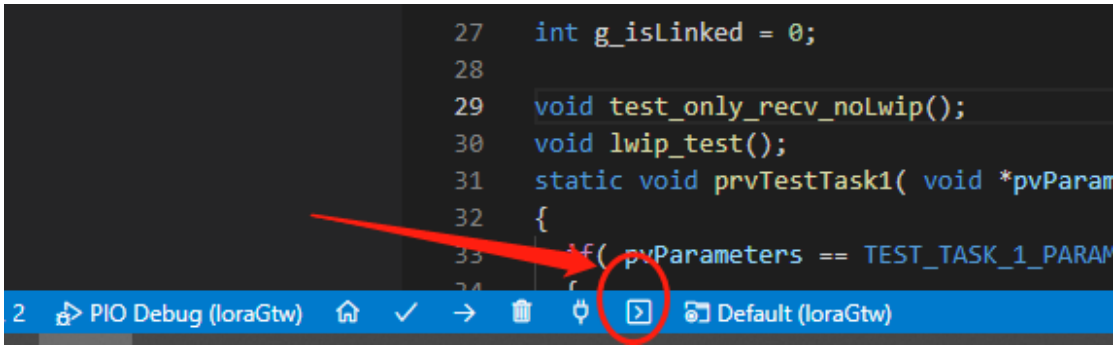
五、编译程序：

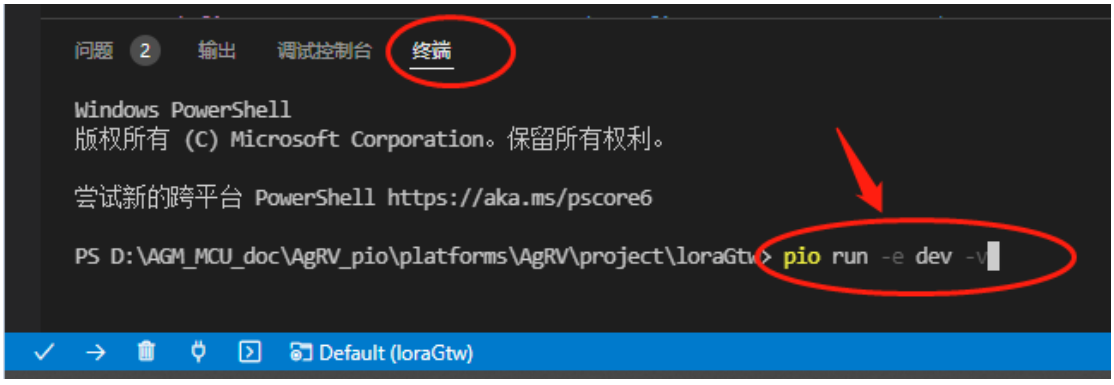
程序编译可以通过 3 种方式：命令行、pio 下栏按钮、pio 左栏按钮；

1. 命令行方式：

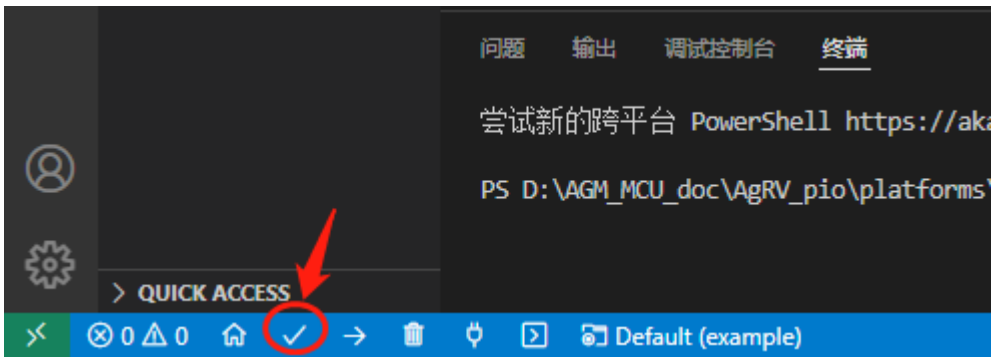
在“终端”通过命令行进行的工程编译。命令：pio run -e dev –v

使用如图：



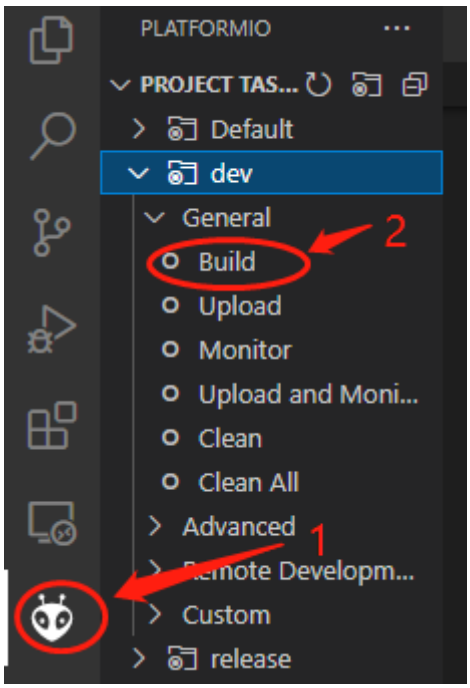


2. Pio 下栏按钮：

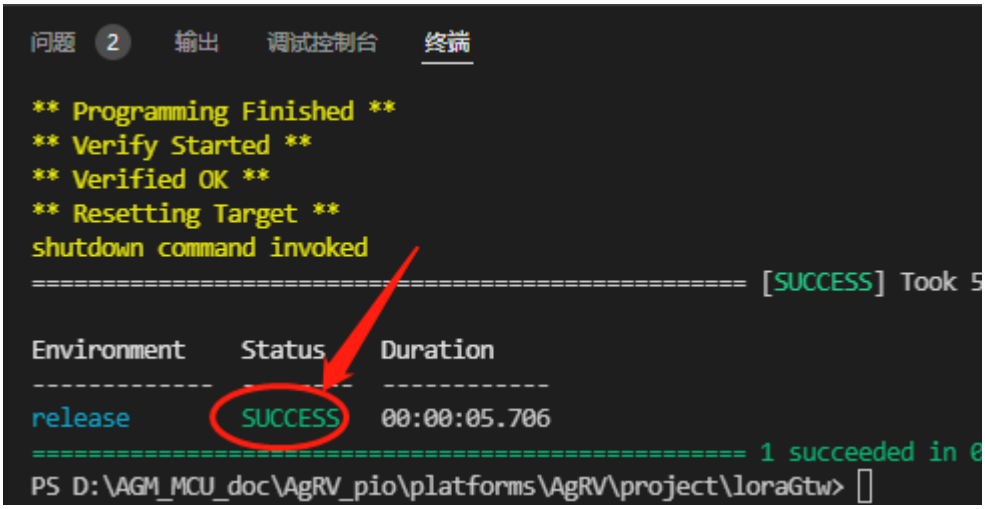


这里的编译和 1 中的命令一样。编译后，会自动弹出终端窗口。

3. Pio 左栏按钮：



不管使用上边的哪种方式，编译和烧录成功时，会有 success 的提示如下：



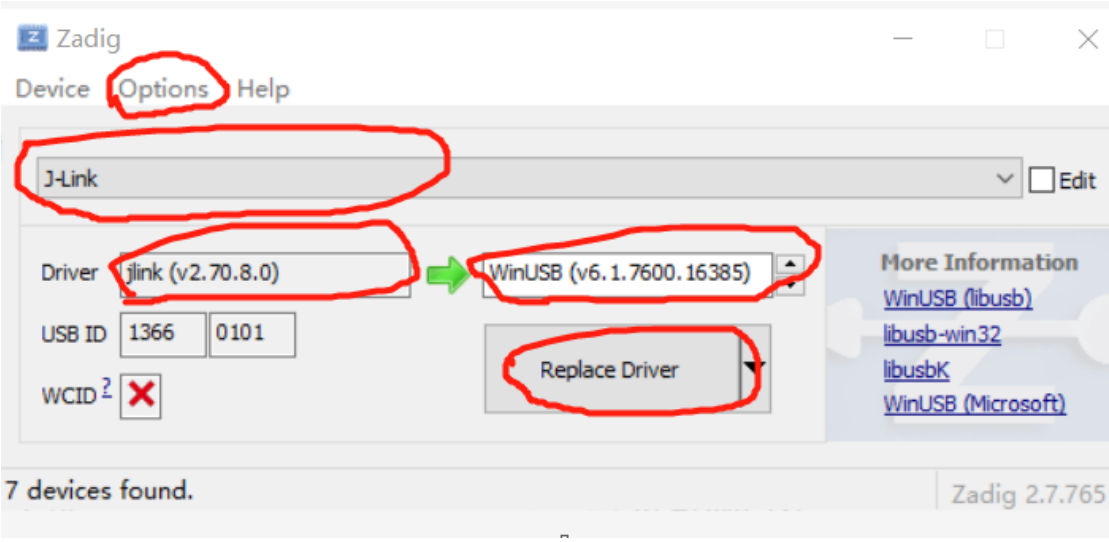
六、烧录程序（和 ve 配置）：

烧录程序一般采用两种方式：jlink 和串口。

Jlink 烧录：

1. 在首次烧录前，需要先安装插件 zadig-2.7.exe。

该文件位于 SDK 解压后的根目录下。安装方式参下图：



该插件安装完毕后，才能正常烧录。

2. 烧录配置：

如果使用 jlink 来烧录程序，必须在 platform.ini 文件中，配置成 jlink 烧录。

配置方法：

修改 upload_protocol 项，使：**upload_protocol = jlink-openocd**

注意，使用 jlink 烧录时，无需配置 upload_port 项。

3. 烧录方法：

烧录方法和上边的编译相似，也是三种方式：命令方式、pio 左边和下边按钮。

烧录命令：pio run -e release -t upload

Pio 下边和左边按钮，紧邻"编译"按钮，不再赘述。

4. 烧录下载 ve 文件：

除了烧录下载程序固件外，AGM 由于支持 fpga，还多出 ve 配置的下载。

ve 配置，就是前边章节中描述的 ve 文件中的配置项。

在烧录 ve 配置时，只支持一种方式：命令方式。

烧录命令：pio run -e release -t logic

烧录到 flash 的 ve 文件，**必须**在 platform.ini 中配置 board_logic_ve 项。如：

board_logic_ve = D:\AgRV_pio\platforms\AgRV\boards\agrv2k_103.ve

该 ve 文件的路径可以使用相对路径或绝对路径。相对路径是相对于工程路径的（即：platformio.ini 所在的路径）。

其中，烧录 ve 配置文件和烧录固件是互不影响的，修改谁烧录谁。

5. 烧录结果提示：

在烧录固件或者 ve 配置完成时，会有 SUCCESS 提示（同编译时相仿）。

如果烧录失败，会有红色 Error 信息给出对应的失败原因。

对于"Error connecting DP: cannot read IDR"，这个报错可以通过重新上电芯片来解决。

串口烧录：

1. 串口烧录前，要先使芯片进入烧录模式；

   进入烧录模式的方法：boot1 接地， boot0 接高。

2. 在 platform.ini 的配置里，配置成串口烧录并指定 PC 使用的串口号；

   配置方法：

   修改 upload_protocol 项，使：**upload_protocol = Serial**

   修改 upload_port 项，试：**upload_port = COMx （x 是编号）**

   还有配置烧录时的波特率，是在[env:serial] 中定义：upload_speed = 115200

   烧录时，PC 使用这个配置的波特率来烧录，MCU 会自动适配速率。

   （烧录程序和 ve 时，MCU 端固定使用 UART0）

3. 烧录程序和烧录 ve；

   串口烧录和 jlink 烧录时相似（可参照上边 jlink 烧录）

   串口烧录支持：命令方式。（按钮方式 未验证）

   烧录程序的命令：pio run -e serial -t upload

   烧录 ve 的命令：pio run -e serial -t logic

   其中，烧录 ve 配置文件和烧录固件是互不影响的，修改谁烧录谁。

4. 烧录成功后的反馈；

   同 jlink 烧录相似，成功也会有 SUCCESS 提示；

   烧录失败会有红色 FAIL 提示错误原因。

总结下几个命令：

编译：pio run -e dev -v

串口烧录 ve 配置：pio run -e serial -t logic

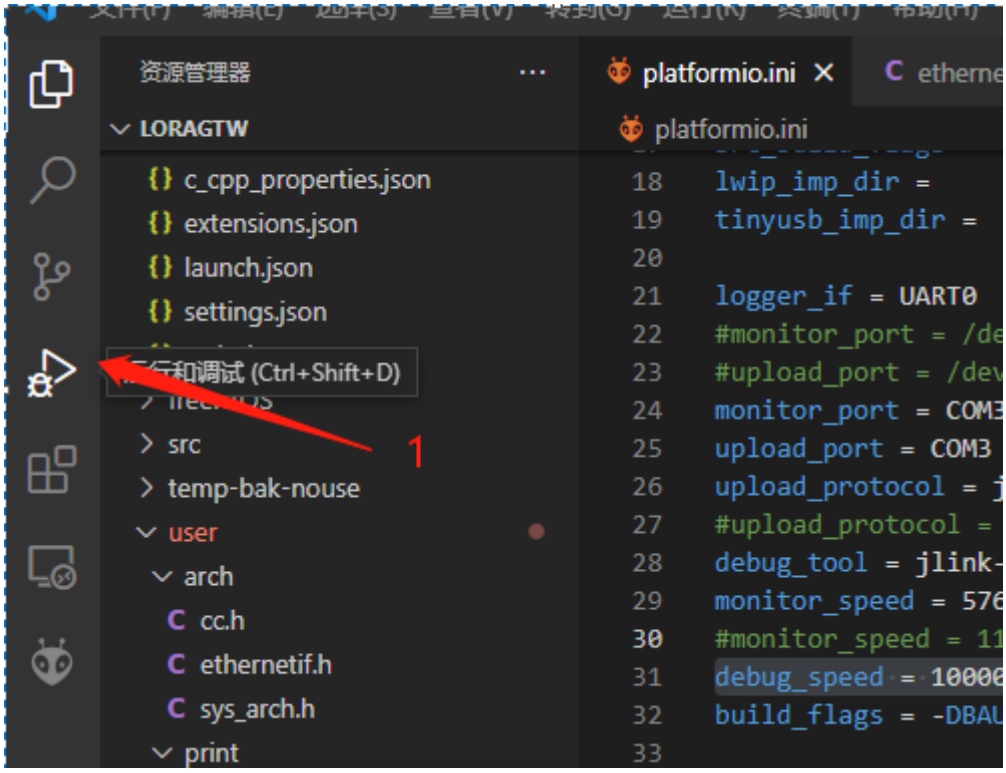
串口烧录 code：pio run -e serial -t upload

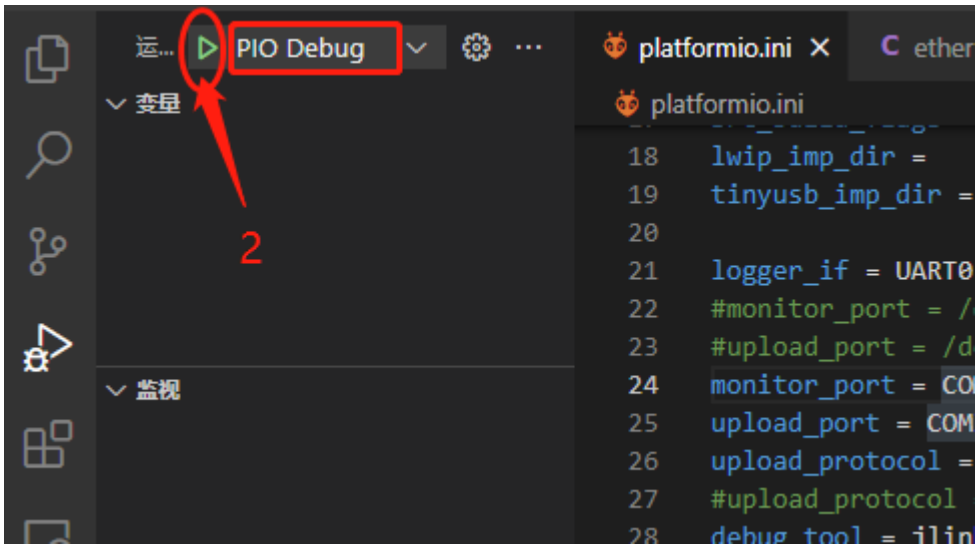
jtag 烧录 ve 配置: pio run -e release -t logic

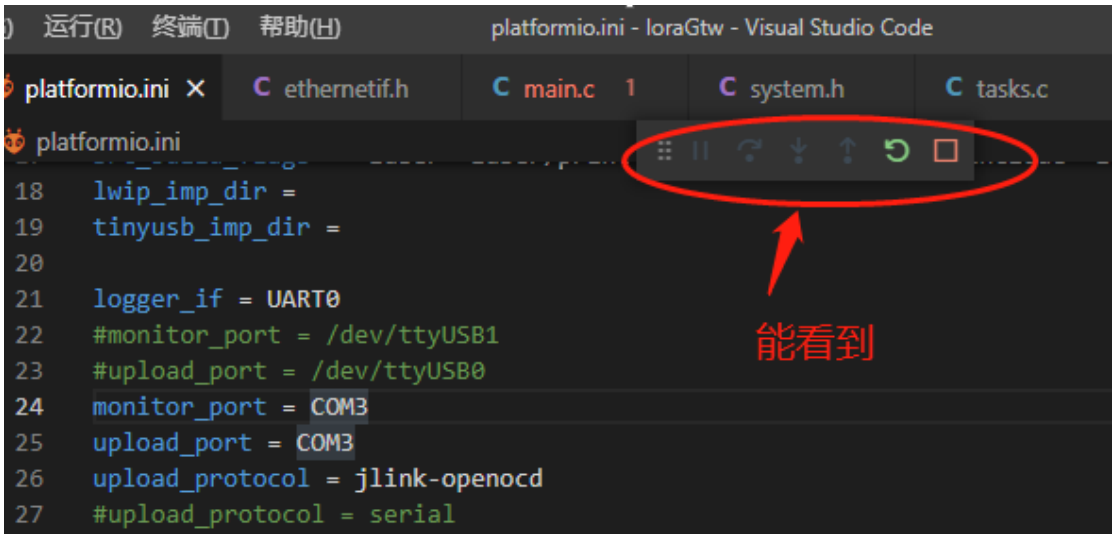
jtag 烧录 code: pio run -e release -t upload

七、jlink 仿真:

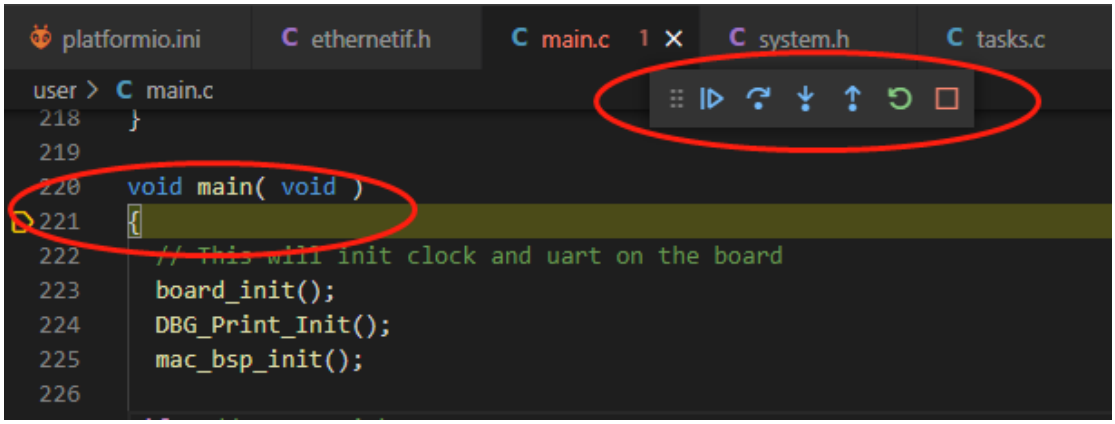
烧录完成后，如果要仿真跟踪代码时，可在 jlink 下按以下步骤启动:







正常运行起来后，这个样子：



接下来就可以单步程序了。debug 的快捷键和 VS 一致。

八、进入开发：

前边已经了解过 SDK 工程的目录结构，这里简单介绍几个常用驱动的使用。

举例 gpio 的使用：

如果我们在 pin3 脚接 led 灯，要控制亮灯 （高为亮），则需要做：

1. ve 配置中设置: GPIO4_5 PIN_3
2. 代码定义三个宏：

```
#define LED_G_GPIO GPIO4
#define LED_G_GPIO_MASK APB_MASK_GPIO4
#define LED_G_GPIO_BITS (1 << 5)
```

3. 代码中调用：

```
SYS_EnableAPBClock(LED_G_GPIO_MASK);
GPIO_SetOutput(LED_G_GPIO, LED_G_GPIO_BITS);
GPIO_SetHigh(LED_G_GPIO, LED_G_GPIO_BITS);
```

4. 烧录 ve，编译代码并烧录；

OVER。

关于 gpio 的使用说明：

1. 芯片可用的对外 gpio 分为 10 组（0~9），每组 8 个 (0~7)；

   ve 配置中的 GPIOx_y：

   GPIO4_5 对应第 4 组的第 5 个 IO；GPIO0_1 对应第 0 组的第 1 个 IO...

   ve 配置中需要写明 GPIOx_y 映射到哪个引脚 PIN_z。

2. 在代码中如何对应该 GPIO：

   仍以 GPIO4_5 举例：

   在代码里定义 #define LED_G_GPIO GPIO4 和 LED_G_GPIO_BITS (1 << 5)，则对应该 GPIO 的唯一地址。

   在函数调用时，就可以使用：

   GPIO_SetOutput(LED_G_GPIO, LED_G_GPIO_BITS);

   来操作该 IO 口。

举例 log 输出：

通过串口烧录 mcu 程序时，是固定使用 mcu 的 uart0。建议 mcu 在运行时也使用 uart0 来输出 log。工程代码中已经重新定向，在代码中 printf 函数将通过 uart0 来输出。

1. 在 ve 中配置输出 Log 的串口：

   logger_if = UART0

2. 在 ve 中设置串口 0 的映射引脚，如：

```
UART0_UARTRXD PIN_69
UART0_UARTTXD PIN_68
```

3. 在程序的 board_init()函数中，确认有 UART0 的初始化：

在该初始化中，有设置串口的波特率、位宽、停止位、奇偶位。

如果不想使用 UART0 输出 log，要改用 UART1。那么在 ve 配置中对应修改：

logger_if = UART1

UART1_UARTRXD PIN_xx
UART1_UARTTXD PIN_xx

即可。

关于系统宏的定义：

在代码里的宏，在 VSCode 中通过鼠标停顿在上边，一般可以看到其定义值。

这些宏，除了代码中定义的外，还有"系统预制宏"也可以被代码中使用（编译时使用）。

系统宏，有些是直接定义在 platform.ini 中的 build_flags 字段中；有些则是在 platform.ini 中定义后再由 main.py 经过转换拼接，才最终使用，比如上边截图中的 LOGGER_UART。

在 platform.ini 中的 build_flags 字段中的宏，这个容易理解，无需多描述。

另外的宏，比如上边截图中的 LOGGER_UART，即没有在代码中定义，也没在 build_flags 中定义。它的使用方式：

a. 在 platform.ini 定义 logger_if = UART0

b. 在 agrv_sdk.py 中引用该字段 logger_if，并判定是 RTT 还是 UART，

如果是 UARTx，则定义 LOGGER_UART 为 x。

c. 这样，logger_if = UART0 时，在 CPP 中就可以认为存在宏：#define LOGGER_UART 0

九、增加编译目录：

在开发中，往往会按照功能来划分模块，并用多个目录来存储。

这里说明如何建立多个目录来进行编译。

1. 如果新增文件在原有路径，则会被自动关联编译进去；
2. 如果新增一个目录文件，则要把该目录加入到编译选项中；

   如果该目录存放 C 文件：在 src_filter 中增加该目录

   如果该目录存放 h 文件：在 src_build_flags 中增加该目录

举例：

在项目里新增一个文件夹 testFolder 到 user 目录下，里边有.c 和.h，要全部编译进去。

原先的 src_filter 和 src_build_flags 对应如下：

src_filter = "-<*> +<*.c> +<print/*.c> "

src_build_flags = -Iuser -Iuser/print

那么，增加 testFolder 后要变为：

src_filter = "-<*> +<*.c> +<print/*.c> **+<testFolder/*.c>**"

src_build_flags = -Iuser -Iuser/print **-Iuser/testFolder**

注意：在 src_filter 和 src_build_flags 中，都可以使用相对路径。他们的相对路径是相对于 src_dir/ include_dir 定义的那个路径。

另外，*是通配符，如果不让某个 C 进入编译，则用：-<testFolder/nowork.c>

十、开发中的注意项：

关于芯片 flash 大小：

不管所选型号的 flash 是多大，请注意最后 100K 是留给 fpga 使用的。

如果使用的芯片是 256K 的 flash，那么就是 156K 程序+100Kfpga，用户程序不能超过 156K。如果超过 156K 编译是可以通过的，但烧录后会冲掉 ve 配置部分。

ve 配置被冲掉后，程序运行会表现出各种异常（连系统时钟初始化都跑不过）。

flash 的大小是在 agrv2k_103.json 中定义的。flash 起始地址是 0x80000000，ram 是 0x20000000。