

AG32 驱动部分的使用

AG32 片内资源列表:

CAN0	--- CAN0_BASE
UART0 ~ 4	--- UART0_BASE
IIC0 ~ 1	--- I2C0_BASE
TIMER0 ~ 1	--- base timer, TIMER0_BASE
GPTIMER0~4	--- advanced timer, GPTIMER0_BASE
MAC0	--- MAC0_BASE
USB0	--- USB0_BASE
watchDog0	--- WATCHDOG0
ADC0 ~ 2	--- ADC0 ---默认 IP 下 3 路, 参考 example_analog;
DAC0 ~ 1	--- DAC0 ---默认 IP 下 2 路
SPI0 ~ 1	--- SPI0

描述项:

1. 时钟的配置;
2. 管脚的配置;
3. GPIO 使用;
4. MTimer 的使用;
5. Base Timer 的使用;
6. GpTimer 的使用;
7. Uart 的使用;
8. IIC 的使用;
9. CAN 的使用;
10. USB 的使用;
11. MAC 的使用;
12. SPI 的使用;
13. ADC/DAC 的使用;
14. WatchDog 的使用;
15. RTC 的使用;
16. 中断说明;
17. 系统休眠 (sleep、stop、stanby);
18. 使用自定义的 logic;
19. 片内 flash 的使用;
20. 其他常用项: 获取芯片 uniqID、堆栈大小的设置;

一、时钟的配置：

AG32 通常使用 HSE 外部晶体（范围：4M~16M）。

AG32 中不需要手动设置 PLL 时钟（时钟树由系统自动配置，无须用户关注）。用户只需在配置文件中给出外部晶振频率和系统主频即可。

配置方式：

在 ve 文件中配置如下：



这里配置的值，会在系统初始化时自动获取并使能。

系统主频的可配置范围：参考 datasheet 中各型号的最高主频（通常是 248M）

外部晶振的可配置范围：4 ~ 16

如果需要使用外部有源晶振，或者使用内部振荡器（5%以内误差），或者 cpld 中需要额外主频输入，请参考文档《AG32 中 cpld 的基础.pdf》中关于时钟的讲解。

二、管脚的配置：

AG32 相比传统芯片（如 ST、GD），很大的一个不同点，是“管脚功能不是定死的”。比如，管脚 2，在传统芯片里，可能是定死用于 uart0_tx。

但在 AG32 中，这个管脚用于什么功能，完全是由用户来自行配置的。用户可以配置成 uart0_tx，也可以配置成 GPIOx_y，还可以配置成 spi_xx，等等。

这种管脚可配置的特性，为 PCB 布线带来了很大的灵活性。并且可以节省大量管脚。

（应用中没用到的外设不去配置，就不会占用管脚）

配置方式，在 ve 文件中一项一项对应即可。

后续的外设驱动中，会分别讲到怎么配置 VE。

那么，对于 mcu 端来说，有哪些信号线的 key 是可用的（比如：uart0 的 tx 信号线是 UART0_UARTTXD），可参考《AGRV2K_逻辑设置.pdf》中的“Function_Pin 列表”的部分。

举例：

1. 配置 GPIO0_1 为 PIN2，则定义：GPIO0_1 PIN_2
2. 配置 UART1_TX 为 PIN3，则定义：UART1_UARTTXD PIN_3
3. 配置 SPI0 的 clk 为 PIN4，则定义：SPI0_SCK PIN_4
4. 配置 CAN0 的 TX 为 PIN5，则定义：CAN0_TX0 PIN_5

注意：除了以上大部分可配置外，也有少量是不可配置的。

不可配置包括：基础类（电源、时钟、地、RESET、BOOT0）、ADC（DAC/CMP）、USB。

其中的 ADC 和 USB 的管脚，如果不接 ADC 和 USB，仍然是可以被用做普通 IO 的。

具体每种封装下管脚的详细定义，请参考文档《AG32_pinout_100_64_48_32_2K.xlsx》。

打开后，如下图：

27	VSS33	GND	GND
28	VDD33	VDD33	VDD33
29	PIN_29	IO ADC IN4 CMP PA4 DAC0	IO
30	PIN_30	IO ADC IN5 CMP PA5 DAC1	IO
31	PIN_31	IO ADC IN6	IO
32	PIN_32	IO ADC IN7	IO
33	PIN_33	IO ADC IN14	IO
34	PIN_34	IO ADC IN15	IO
35	PIN_35	IO ADC IN8	IO
36	PIN_36	IO ADC IN9	IO
37	PIN_37	IO BOOT1	IO
38	PIN_38	IO	IO
39	PIN_39	IO	IO

凡是带有 IO 的，都是可以被配置的管脚。

比如：上图的 PIN_33，如果 ADC 的 channel14 在使用，那这个管脚只能用于这路 ADC。如果这路 adc 没有使能，那 PIN_33 仍然可配置用于其他功能。

三、GPIO 的使用：

可用 GPIO：

AG32 芯片内部可用 gpio 共有 80 个，分为 10 组，每组 8 个。

代码中各组对应为：GPIO0、GPIO1、GPIO2、...

组内各 IO 用 bit 表示：GPIO_BIT0、GPIO_BIT1、GPIO_BIT2、...

使用时，用 组 ID+组内 ID 来标识唯一的 IO。

这里和 ST 是相仿的，ST 分为 GPIOA/GPIOB/GPIOC, PIN_1/PIN_2/PIN_3...

AG32 为：GPIO0/GPIO1/GPIO2..., GPIO_BIT0/GPIO_BIT1/GPIO_BIT2...

对外映射：

程序中使用到的 GPIO 要映射到对外的管脚 PIN。

映射方式，就是在 ve 文件中配置，如下图：

```

5
6  GPIO4_1 PIN_92  # LED1  ← 配置GPIO4_1到92引脚
7  #GPIO4_2 PIN_93  # LED2
8

```

上图的示例，就是把 gpio4_1 映射到管脚 92。

在 AG32 中，必须映射后，代码中操作 gpio 时，才会真正使能到硬件管脚。

这里 GPIOx_y 的角标取值范围：x (0 ~ 9), y (0 ~ 7)

PIN_z 的取值范围：z 小于最大引脚数

在取值范围内，满足限制条件下，任意 GPIOx_y 可以映射到任意 PIN_z。

（“哪些管脚不能被使用”的限制条件，参考文档：AGRV2K_逻辑设置.pdf）

这里配置的 GPIO0_0，等同于代码中的 (GPIO0, GPIO_BIT0)。

样例 1：

用 pin3 引脚接 led 灯，并控制亮灯（高为亮）。

步骤一：

先做 ve 文件中定义引脚映射（gpio 使用 4-1）：

```

5
6 GPIO4_1 PIN_3 # LED ← gpio映射到pin
7

```

步骤二：

定义使用的宏：（也可以不定义，直接在代码中使用）

```

#define LED_GPIO GPIO4 ← group 4
#define LED_GPIO_MASK APB_MASK_GPIO4
#define LED_GPIO_BITS GPIO_BIT1 ← bit1

```

步骤三：

代码中调用：

```

SYS_EnableAPBClock(LED_GPIO_MASK); ← 开时钟
GPIO_SetOutput(LED_GPIO, LED_GPIO_BITS); ← 设置为输出
GPIO_SetHigh(LED_GPIO, LED_GPIO_BITS); ← 置高IO

```

步骤四：

编译并烧录 ve 文件，编译并烧录 code；

驱动开放的 API 包含：

GPIO_SetOutput/GPIO_SetInput ---设置 IO 为输入输出
 GPIO_SetHigh/GPIO_SetLow ---置高置低
 GPIO_Toggle ---高低切换
 GPIO_IntConfig ---配置中断触发方式
 GPIO_EnableInt/GPIO_DisableInt/GPIO_ClearInt ---中断控制
 GPIO_AF_ENABLE/GPIO_AF_DISABLE ---切换 GPIO 模式（如果有复用）
 Gpio 中断函数 SDK 中已经默认指定：GPIOx_isr
 如果要重定向为函数，通过 plic_isr[GPIOx_IRQn] = gpio_xxx_isr 的方式来设置；

样例 2：

用 pin96 接外部按键，处理按键消息；

步骤一：

在 ve 文件中配置 gpio4_5 映射到 pin96；

```

GPIO4_5 PIN_96 # button

```

步骤二：

在测试代码中，编写 IO 初始化，并实现中断函数：

```

void GPIO4_isr()
{
    if (GPIO_IsRawIntActive(GPIO4, GPIO_BIT5)) {
        GPIO_ClearInt(GPIO4, GPIO_BIT5);
        printf("detect key-press\r\n");
    }
}

void TestGpio()
{
    SYS_EnableAPBClock(APB_MASK_GPIO4);
    GPIO_SetInput(GPIO4, GPIO_BIT5);
    GPIO_EnableInt(GPIO4, GPIO_BIT5);
    GPIO_IntConfig(GPIO4, GPIO_BIT5, GPIO_INTMODE_FALLEDGE);
    INT_EnableIRQ(GPIO4_IRQn, GPIO_PRIORITY);
}

```

注：这里的中断函数 GPIO4_isr 无需程序中再次指定。

步骤三：

如果外部电路没有上拉设计，则需要设置芯片内的上拉。

设置方式：

在\platforms\AgRV\boards\agrv2k_x0x\board.asf 文件 中添加（下图红框内容）：

```

if { [info exists USB0_MODE] } {
    alta::tcl_info "USB0_MODE = $USB0_MODE"
    set_config -loc 0 1 3 CFG_PULLUP_ENB 1'b0
    set_config -loc 0 1 3 CFG_PULLDN_ENB 1'b0
}

```

```

set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to PIN_96

```

内容：set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to PIN_96

注意：如果是 cpld 中要实现上拉，这里的 PIN_96 要用 cpld 里的信号名字。

如果设置下拉，则使用以下方式：

```

if { [info exists USB0_MODE] } {
    alta::tcl_info "USB0_MODE = $USB0_MODE"
    set_config -loc 0 1 3 CFG_PULLUP_ENB 1'b0
    set_config -loc 0 1 3 CFG_PULLDN_ENB 1'b0
}

```

```

set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to PIN_96
set_instance_assignment -name CFG_KEEP -to PIN_32 2'b01 -extension

```

内容：set_instance_assignment -name CFG_KEEP -to PIN_32 2'b01 -extension

注意：如果是 cpld 中要实现下拉，这里的 PIN_32 要用 cpld 里的信号名字。

注意：上边一行添加完后，**务必**在后边添加回车换行（保证这行不是文件最后一行）。

步骤四：

编译并烧录 ve 文件，编译并烧录 code；

注意：

如果使用到的引脚是复用引脚，则默认是 IO 功能。使用为特定功能时，需要先设置为复用属性，用函数 `GPIO_AF_ENABLE` 设置（参考各驱动样例代码）。

如果使用到的引脚是特殊的 JTAG 引脚（JNTRST、JTDO、JTDI、JTMS、JTCK），则默认是 JTAG 功能，而不是 IO 功能。这时需要先将 IO 口设置为普通 IO 才能使用。

可使用如下函数来设置为普通 IO：

`SYS_DisableNJTRST()`、`SYS_DisableJTDI()`、`SYS_DisableJTDO()`。

（由于 AG32 默认使用 jtag 的 swd 模式，所以保留 JTMS、JTCK 即可）。

IO 如何输出设置为 PP/OD 模式？

IO 默认输出是 PP 模式。

如果要设置为 OD 模式，则需要在 ve 里定义引脚如下：

`GPIO4_1 PIN_34:OUTPUT: !PIN_34_out_data`

管脚如何配置电流输出驱动能力？

同上边上拉/下拉的设置文件（board.asf 文件），加入：

`set_instance_assignment -name CURRENT_STRENGTH -to PIN_32 16MA`

驱动电流默认为 8MA，支持 4MA/8MA/12MA/16MA。

四、MTimer 的使用：

MTime 是 risc-v 中定义的一个 64 位系统定时器。

在 STM32 中，我们一般用 systick(滴答计时器)作为时基，而在 riscv 中我们用 machine timer(简称 mtime)作为时基。

MTime 中有两个主要寄存器：mtime 和 mtimecmp；

当 mtime 使能后，mtime 寄存器里的值会随着 tick 自增，当自增到 大于等于 mtimecmp 寄存器的值时(无符号比较)，就触发 MTimer 中断。

在移植操作系统时，mtime 一般被用于系统时间片的调度定时。

相关函数：

INT_SetMtime：设置寄存器的值；

INT_SetMtimeCmp：设置比较寄存器的值；

INT_EnableIntTimer：打开 timer 中断；

中断函数 SDK 中已默认指定：void MTIMER_isr()

如果要重定向函数，通过 `clint_isr[IRQ_M_TIMER] = MTIMER_user_isr` 来设置；

如果要设置 1ms 触发一次的连续定时，需要调用：

`INT_SetMtime(0);`

`INT_SetMtimeCmp(SYS_GetSysClkFreq() / 1000); //1ms`

然后在中断里重新计时：

`INT_SetMtime(0);`

完整代码样例请参考 example 部分：


```

void MTIMER_isr(void)
{
    INT_SetMtime(0);
    printf("MTimer int\r\n");
}
void TestMtimer(int ms)
{
    //clint_isr[IRQ_M_TIMER] = MTIMER_isr;
    INT_SetMtime(0);
    INT_SetMtimeCmp(SYS_GetSysClkFreq() / 1000 * ms);
    INT_EnableIntTimer();
}

```

五、Base Timer 的使用：

AG32 中包含 2 个 Base Timer：分别对应 TIMER0 和 TIMER1。

这两个 timer 中，每个又有两组寄存器，每组寄存器可以单独产生定时。

所以，真正可用的普通定时器有 4 个：TIMER0-0、TIMER0-1、TIMER1-0、TIMER1-1。

4 个定时器均可独立设置。

普通定时器特点：

- 定时器支持 16 位和 32 位的设置，

- 支持 3 种类型分频（1 分频，16 分频，256 分频），

- 支持单次定时和循环定时。

驱动 API 函数命名中的 1 和 2，分别对应第一组和第二组寄存器。

如，TIM_Init1 设置的是第一组寄存器，TIM_Init2 设置的是第二组寄存器。

举例：

用 Timer1 的 group2 产生 1s 的循环定时：

```

void TIMER1_isr()
{
    if (TIM_IsRawIntActive2(TIMER1)) {
        TIM_ClearInt2(TIMER1);
        printf("Timer1 g2 INT\r\n");
    }
}
void TestTimer()
{
    SYS_EnableAPBClock(APB_MASK_TIMER1);
    TIM_Init2(TIMER1, 1e6, TIMER_MODE_PERIODIC); //1e6 = 1000000 , 1s
    TIM_EnableInt2(TIMER1);
    TIM_EnableTimer2(TIMER1);
    INT_EnableIRQ(TIMER1_IRQn, TIMER_PRIORITY);
}

```

中断函数 TIMER1_isr 在 SDK 中已经默认指定。

说明:

设置函数: TIM_Init1 <-> TIM_SetLoad1/TIM_SetSize1/TIM_SetMode1/...

中断函数: TIMER0_isr/TIMER1_isr

函数说明:

void TIM_Init1(TIMER_TypeDef *tim, uint32_t timeInUs, TIMER_ModeTypeDef mode)

作用: 启动 Timer0 或 Timer1 的第一个定时器 (TIM_Init2 则启动第二个定时器)。

参数: tim: TIMER0 or TIMER1

timeInUs: 多少 us 触发定时

mode: TIMER_MODE_PERIODIC:循环触发 TIMER_CTRL_ONESHOT:只触发一次

举例:

```
TIM_Init1(TIMER0, 500000, TIMER_MODE_PERIODIC);
```

表示启动 TIMER0 的第一个定时器, 500ms 触发一次定时中断, 循环触发。

除了直接调用 TIM_Init1 来启动一个定时外, 还可以调用各个子函数来启动。

如,

```
TIM_Init2(TIMER0, 500000, TIMER_MODE_PERIODIC)
```

功能等价于:

```
TIM_SetLoad2(TIMER0, SYS_GetPclkFreq() / 1000000 * 500000);
```

```
TIM_SetSize2(TIMER0, TIMER_SIZE_32);
```

```
TIM_SetMode2(TIMER0, TIMER_MODE_PERIODIC);
```

```
TIM_SetPrescaler2(TIMER0, TIMER_PRESCALE_1);
```

```
TIM_EnableInt2(TIMER0);
```

```
TIM_EnableTimer2(TIMER0);
```

以上几个函数中,

TIM_SetPrescaler2 是设置分频,

三个参数可选: TIMER_PRESCALE_1/TIMER_PRESCALE_16/TIMER_PRESCALE_256

分别表示分频数: 1 分频, 16 分频, 256 分频;

TIM_SetSize2 设置计时器位宽,

两个参数可选: TIMER_SIZE_32/TIMER_SIZE_16

表示计数器的 load 的位宽是 32 位还是 16 位。

TIM_SetLoad2 设置触发时间 (以 tick 为单位)

如果定时单位为 ms, 则需要将 tick 转为 ms: $\text{SYS_GetPclkFreq}() / 1000000 * \text{ms}$

中断函数: void TIMER0_isr()

函数说明: 该函数为 TIMER0 的中断函数;

在该函数中需要先查询是第一个还是第二个定时器, 然后再清中断。

该中断函数已默认关联, 不需要程序中来手工设置。

完整代码样例请参考 example 部分。

六、General Purpose Timer 的使用:

AG32 中包含 5 个通用计时器 (GpTimer),

代码中分别对应: GPTIMER0、GPTIMER1、GPTIMER2...

通用定时器可以实现更多功能，包括：计时、生成 pwm、生成任意波形、输入捕获。
5 个定时器均可独立设置。

每个定时器支持 4 个独立通道（channel）：

- 输入捕获
- PWM 输出（边缘或中间对齐模式）
- 单脉冲输出

主要函数：GPTIMER_Init / GPTIMER_OC_Init

1. 用于定时：

用于简单定时，只需要关注一个函数：GPTIMER_Init，
设置好参数后，启动计时即可。

举例：

用 gpTimer1 产生 2 秒一次的定时。

```
void GPTIMER1_isr()
{
    GPTIMER_ClearFlagUpdate(GPTIMER1);
    GPTIMER_EnableCounter(GPTIMER1);
    printf("gpTimer1 INT\r\n");
}

void TestGpTimer()
{
    SYS_EnableAPBClock(APB_MASK_GPTIMER1);
    GPTIMER_InitTypeDef tm_init;
    GPTIMER_StructInit(&tm_init);
    tm_init.Autoreload = 2e6; //2e6 = 2000000, that is 25
    tm_init.Prescaler = 100; //主频为100M时，这里设置为100
    GPTIMER_Init(GPTIMER1, &tm_init);
    GPTIMER_SetOnePulseMode(GPTIMER1, GPTIMER_ONEPULSEMODE_SINGLE);
    GPTIMER_EnableCounter(GPTIMER1);

    INT_EnableIRQ(GPTIMER1_IRQn, TIMER_PRIORITY);
    GPTIMER_EnableIntUpdate(GPTIMER1);
}
```

这里使用到的中断函数 GPTIMER1_isr，已被 SDK 自动设置。

2. 用于 pwm 输出：

用于 pwm 输出时，要设置两个函数：GPTIMER_Init 和 GPTIMER_OC_Init。
GPTIMER_Init 中设置多长时间触发一次 timer；

GPTIMER_OC_Init 中指定 pwm 输出通道及设置 pwm 的占空比；

举例：

用 gpTimer4 在通道 0 上产生 pwm 输出。

```

void TestGpTimerPwm()
{
    SYS_EnableAPBClock(APB_MASK_GPTIMER4);
    GPTIMER_InitTypeDef tm_init;
    GPTIMER_StructInit(&tm_init);
    const uint32_t frequency = 10000;          // In Hz 每秒产生多少次切换
    const float pwm_ratio = 0.3;
    tm_init.Autoreload = SYS_GetPclkFreq() / frequency;
    GPTIMER_Init(GPTIMER4, &tm_init);

    GPTIMER_OC_InitTypeDef oc_init;
    oc_init.OCState = GPTIMER_OCSTATE_ENABLE;
    oc_init.OCMode = GPTIMER_OCMODE_PWM1;
    oc_init.CompareValue = tm_init.Autoreload * pwm_ratio; //设置占空比
    GPTIMER_OC_Init(GPTIMER4, GPTIMER_CHANNEL_CH0, &oc_init); //设置通道
    GPIO_AF_ENABLE(GPTIMER4_CH0);           //GPIO复用为TIMER的channel输出

    GPTIMER_EnableAllOutputs(GPTIMER4);
    GPTIMER_EnableCounter(GPTIMER4);
}

```

除了上述的代码控制外，还需要在 ve 中添加映射关系：

GPTIMER4_CH0 PIN_7 ← 输出的IO映射到管脚

这样的情况下，pwm 才会输出到管脚上。

典型案例：呼吸灯（用 timer+timerPWM 来控制 led 灯逐渐变量逐渐变暗）

3. 输出反向 PWM（带死区）：

样例程序，请参考网盘下“其他文档\驱动样例补充\example_gptimer_pwm_N.c”

4. 输出任意波形：

如果要输出的不是 pwm 的规则波形，而是不规则波形（比如正弦波），则可借助于 DMA 方式来模拟实现。

思路：事先在数组中定义好数据序列，然后通过 dma 每次搬运，作用到输出。

这部分功能，参考例程函数：TestGpTimerDma

这种方式也同样需要管脚映射。

5. 输入捕获：

用于输入捕获时，要设置两个函数：GPTIMER_Init 和 GPTIMER_IC_Init。

样例程序，请参考网盘下“其他文档\驱动样例补充\example_gptimer_capture.c”

七、Uart 的使用：

AG32 可用的 UART 有 4 个，分别对应 UART0、UART1、UART2、UART3；

样例工程中，UART0 被做为输出 log 的串口。其他几个 UART 可供应用使用；

初始化函数:

```
void UART_Init(UART_TypeDef *uart,
               UART_BaudRateTypeDef baudrate,
               UART_LCR_DataBitsTypeDef databits,
               UART_LCR_StopBitsTypeDef stopbits,
               UART_LCR_ParityTypeDef parity,
               UART_LCR_FifoTypeDef fifo)
```

参数说明:

Uart: UART0、UART1、UART2 or UART3

Baudrate: 波特率, 如 115200

Databits/stopbits/parity:

Fifo: 是否开启 16 字节的 fifo 缓冲

收发函数:

```
UART_Send(UART_TypeDef *uart, const unsigned char *p, unsigned int num)
```

```
UART_Receive(UART_TypeDef *uart, unsigned char *p, unsigned int num, unsigned
             int timeout)
```

收函数的 timeout, 是如果收不满 num 个字符, 就等待多少个 tick。可以为 0。

样例 1:

实现 Uart1 的简单收发:

1. 增加 ve 对 uart1 的管脚配置:

```
UART1_UARTTXD PIN_52
UART1_UARTRXD PIN_51
```

2. 代码中实现如下:

```
void TestUart(void)
{
    char txbuf[] = "simple uart1\n";
    char rxbuf[256];

    GPIO_AF_ENABLE(UART1_UARTRXD)
    GPIO_AF_ENABLE(UART1_UARTTXD);
    SYS_EnableAPBClock(APB_MASK_UART1);
    UART_Init(UART1, 115200, UART_LCR_DATABITS_8, UART_LCR_STOPBITS_1,
              UART_LCR_PARITY_NONE, UART_LCR_FIFO_16);

    UART_Send(UART1, txbuf, strlen(txbuf));
    while (1)
    {
        int rLen = UART_Receive(UART1, rxbuf, 12, 0);
        if (rLen > 0)
        {
            UART_Send(UART1, rxbuf, rLen);
        }
    }
}
```

样例 2:

使用接收中断来收取数据。

代码部分可参考以下方式（新增的红框代码）：

```
volatile char isRecv = 0;
char rxbuf[32];
void UART1_isr()
{
    if (UART_IsRawIntActive(UART1, UART_INT_RX)) {
        UART_ClearInt(UART1, UART_INT_RX);
        UART_Receive(UART1, rxbuf, 8, 0); //half: 16/2=8
        isRecv = 1;
    }
}
void TestUart(void)
{
    const char txbuf[] = "uart1 rx INT\n";

    GPIO_AF_ENABLE(UART1_UARTRXD);
    GPIO_AF_ENABLE(UART1_UARTTXD);
    SYS_EnableAPBClock(APB_MASK_UART1);
    UART_Init(UART1, 115200, UART_LCR_DATA_BITS_8, UART_LCR_STOPBITS_1,
              UART_LCR_PARITY_NONE, UART_LCR_FIFO_16);
    UART_EnableInt(UART1, UART_INT_RX);
    UART_SetRxIntFifoLevel(UART1, UART_INT_FIFO_HALF);
    INT_EnableIRQ(UART1_IRQn, UART_PRIORITY);

    UART_Send(UART1, txbuf, strlen(txbuf));
    while (1)
    {
        if (isRecv > 0)
        {
            UART_Send(UART1, rxbuf, 8);
            isRecv = 0;
        }
    }
}
```

中断函数 UART1_isr 在 SDK 中已经默认关联，不用手动设置。

在中断函数中，要判别中断来源再继续操作。

上例中，收 FIFO 因为设置为 16 字节，半数触发时，收到 8 个字节就会触发中断。

如果每来一个字节中断接收一次，可以在 UART_Init 中设置参数为 UART_LCR_FIFO_1，并且不用再调用 UART_SetRxIntFifoLevel 函数。

3. 使用 DMA 收发:

如果要启用 DMA 功能，参考 sdk 中自带的样例。

需要增加 3 个函数:

DMAC_Init: 启动 dma

UART_SetDmaMode: 设置只要收/发 dma，或收发都要 dma

DMAC_Config: 设置 dma 的详细参数。

如果收发都要 dma，则需要调用 2 次 DMAC_Config 来分别设置。

函数 DMAC_Config 的参数说明：

```
void DMAC_Config(  
    DMAC_ChannelNumTypeDef channel, //DMA 通道  
    uint32_t srcAddr, //DMA 数据源地址  
    uint32_t dstAddr, //DMA 数据目标地址  
    DMAC_AddrIncTypeDef srcIncr, //传输后源地址是否自增  
    DMAC_AddrIncTypeDef dstIncr, //传输后目标地址是否自增  
    DMAC_WidthTypeDef srcWidth, //源地址传输数据的字节宽度（可选 8/16/32）  
    DMAC_WidthTypeDef dstWidth, //目标地址传输数据的字节宽度（可选 8/16/32）  
    DMAC_BurstTypeDef srcBurst, //源地址一次传输多少？？  
    DMAC_BurstTypeDef dstBurst,  
    uint32_t transferSize, //传输多少次  
    DMAC_FlowControlTypeDef transferType, //传输方向类型（8 种）  
    uint32_t srcPeripheral, //源地址的外设类型  
    uint32_t dstPeripheral //目标地址的外设类型  
)
```

比如，设置收 DMA，会设置参数如：

```
DMAC_Config(DMAC_CHANNEL1,  
    (uint32_t)&UART3->DR, //串口数据寄存器  
    (uint32_t)rxbuf, //收缓冲 buff  
    DMAC_ADDR_INCR_OFF, //源地址不自增  
    DMAC_ADDR_INCR_ON, //目标地址自增  
    DMAC_WIDTH_8_BIT, //源数据宽度以 8bit 为单位  
    DMAC_WIDTH_8_BIT, //目标数据宽度以 8bit 为单位  
    DMAC_BURST_1,  
    DMAC_BURST_1,  
    0, //传输多少次，如果是 0 则无限制  
    DMAC_PERIPHERAL_TO_MEM_PERIPHERAL_CTRL, //外设到内存的方向  
    UART3_RX_DMA_REQ, //源数据外设类型  
    0 ); //目标数据外设类型
```

设置发的 DMA，会设置参数如：

```
DMAC_Config(DMAC_CHANNEL0,  
    (uint32_t)txbuf, //发缓冲  
    (uint32_t)&UART3->DR, //串口数据寄存器  
    DMAC_ADDR_INCR_ON, //发缓冲自增  
    DMAC_ADDR_INCR_OFF, //寄存器不自增  
    DMAC_WIDTH_8_BIT, //源数据宽度以 8bit 为单位  
    DMAC_WIDTH_8_BIT, //目标数据宽度以 8bit 为单位  
    DMAC_BURST_1,  
    DMAC_BURST_1,  
    dma_count, //要传输的数据量
```

```
DMAC_MEM_TO_PERIPHERAL_DMA_CTRL, //内存到外设的方向
0, //源数据外设类型
tx_dma_req); //目标数据外设类型
```

以上完整代码样例请参考 example 部分。

3. 更多样例，请参考网盘：

- 1). dma 中断：“其他文档\驱动样例补充\example_uart_dmaIrq.c”
- 2). 闲时中断：“其他文档\驱动样例补充\example_uart_rcvIqr.c”

八、IIC 的使用：

AG32 支持两路 I2C，分别对应：I2C0、I2C1；

I2C 是一种简单的双向两线制总线协议，半双工，支持多主从模式。I2C 最大的特点之一就是具有完善的应答机制。

MCU 端是 I2C 的主端。

样例程序参考 example_i2c.c

在使用 I2C 时的流程：

1. Ve 中先配置对应的引脚：

```
I2C0_SDA PIN_36
I2C0_SCL PIN_35
```

2. 代码中时钟使能、中断使能、设置频率；

```
PERIPHERAL_ENABLE(I2C, 0);
INT_EnableIRQ(I2C0_IRQn, I2C_PRIORITY);

I2C_Init(I2C0, frequency);
```

3. 使能 I2C；

```
I2C_Enable(I2C0);
```

3. 收发数据；

IIC 的收过程和发过程，都有对应的应答流程，启动->收/发->结束。

使用中，收发函数会被完整的封装。

请参考例程函数（函数流程可参考，封装请自行调整）：

```
bool I2cReadPROM(uint8_t *mem, bool verify)
bool I2cWritePROM(uint8_t *mem)
```

4. 关闭 I2C：

```
I2C_Disable(I2C0);
```

另外，例程中还使用到了中断函数。当 I2C 准备好时，会触发该中断。

九、CAN 的使用：

AG32 支持 1 路 CAN，对应：CAN0

样例程序参考 example_can.c

在使用 CAN 时的流程：

1. ve 中先配置对应的引脚：

```
CAN0_TX0 PIN_39
CAN0_RX0 PIN_38
```

2. 代码中使能时钟、开中断：

```
PERIPHERAL_ENABLE(CAN, 0);
INT_EnableIRQ(CAN0_IRQn, CAN_PRIORITY);
```

3. 配置参数（参数较多）并开启 can、开启收中断：

```
CAN_Init(CAN0, &init);
CAN_EnableIntRx(CAN0);
```

4. 发送数据：

```
CanTx(can_id, frame_format, CAN_DATA_FRAME, CAN_DATA_LENGTH_8, "test can");
CAN_WaitForTx(CAN0);
```

5. 在中断函数中接收数据：

```
void CAN0_isr()
{
    uint32_t can_ir = CAN_GetIntStatus(CAN0);
    if (can_ir & CAN_IR_TI) { // Tx done
        printf("can tx done\n");
    }
    if (can_ir & CAN_IR_RI) { // Rx done
        CAN_RxMessageTypeDef rx_msg;
        GPIO_Toggle(EXT_GPIO, 0x04);
        CAN_Receive(CAN0, &rx_msg);
```

使用时，请参考样例修改。

十、USB 的使用：

AG32 已经在工程中集成 tinyUSB，可自行关联使用。

usb 使用到的 PIN 脚，是固定的管脚，不能在 ve 中进行改变。

目前支持：单纯 device 端、单纯 host 端、OTG 自动切换主从端；

三种情况要支持的枚举类型，可以在配置头文件中自行配置。

例程位于路径 examples\usb 下：

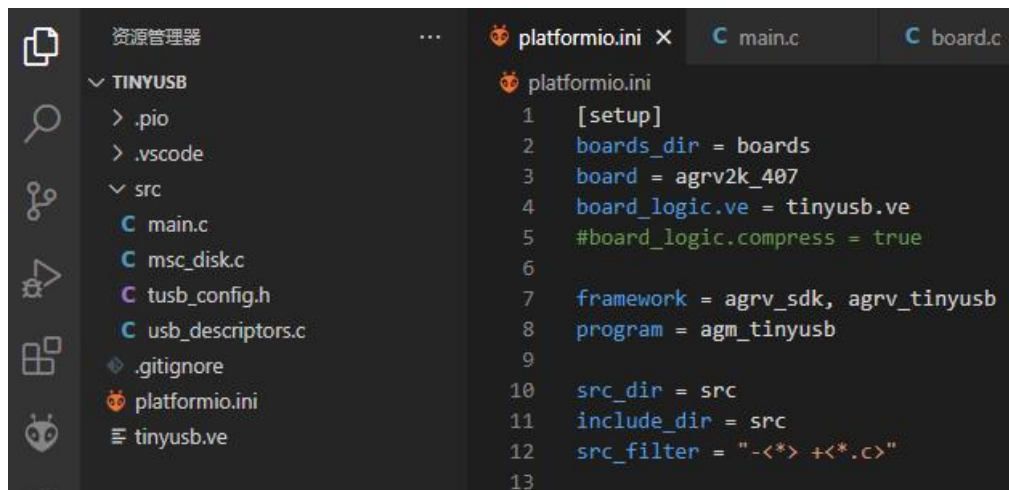
agmWork > AgRV_pio > platforms > AgRV > examples > usb		
名称		修改日期
cdc_msc	← device	2023/10/31 10:40
host_cdc_msc_hid	← host	2023/10/9 13:06
otg_host_hid_device_cdc	← otg	2023/10/9 13:06
tinyusb_lwip		2023/10/9 13:06

简单应用举例（使用 device 样例，其他两个相似）：

在 device 的例程中，usb 被同时枚举为 cdc 和 msc（还支持 HID 和 MIDI）。

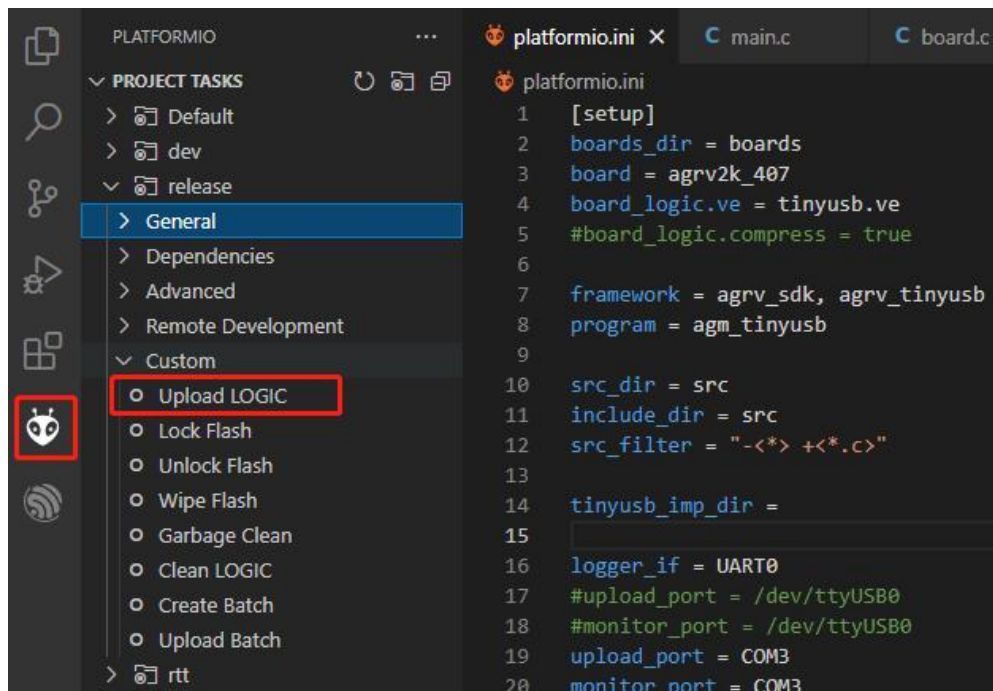
1. vsCode 打开该文件夹工程；

打开后如图：

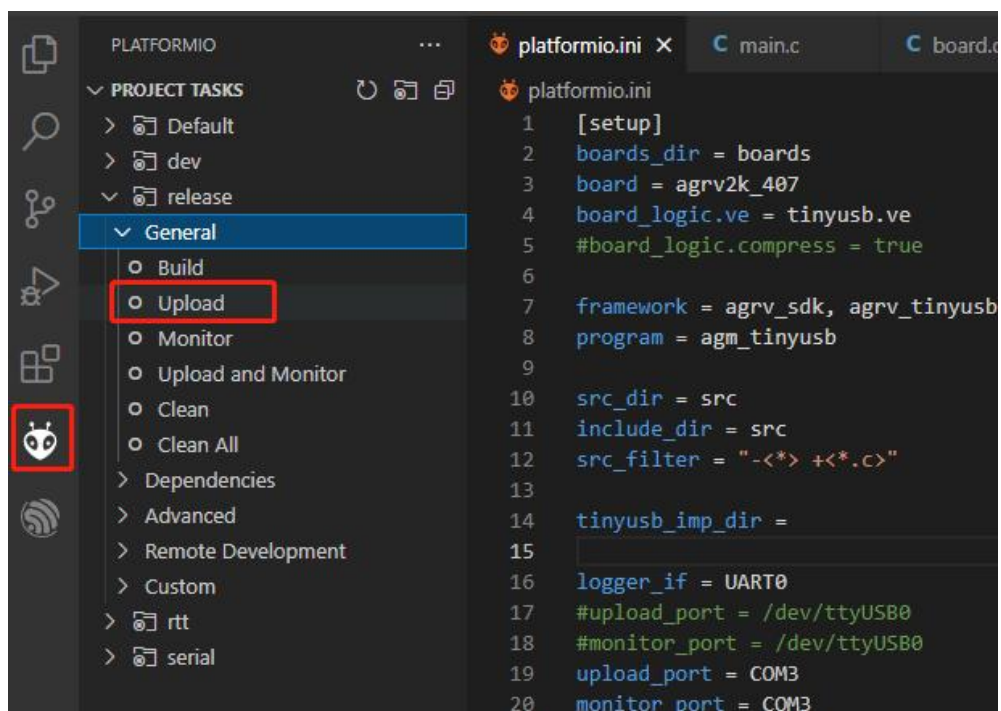


2. 直接烧录 ve 和程序 bin；

烧录 ve：

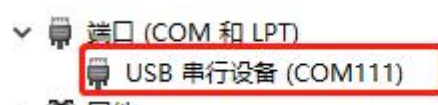


烧录 bin:



3. 上电启动，然后 USB 线连接到电脑（开发板上对应 micro 口的那个 USB 口）
就可以看到 PC 端的 U 盘和 cdc 串口，如下图：

（cdc 串口）



（U 盘）



如果没有显示出来，可以烧录并跟踪程序，看是否出现 `board_init` 失败（板子不同，可能会带来 `board` 初始化的失败）

以上是 demo 验证。

如果要集成到自己的工程（如，要在 example 中使用），需要修改：

1. Platformio.ini 中增加对 `tinyusb` 的引用：

```
framework = agrv_sdk, agrv_tinyusb
program = agm_example
```

注意，引用多个库时，用逗号隔开，并且逗号后边要加空格。

2. ve 文件中增加：

```
#USB0_ID PIN_78
USB0 device
```

3. 代码部分调整：

将 `tinyusb` 下的 `src` 路径文件，修改 `main.c` 后放入 `example` 下的 `src`。

修改点：

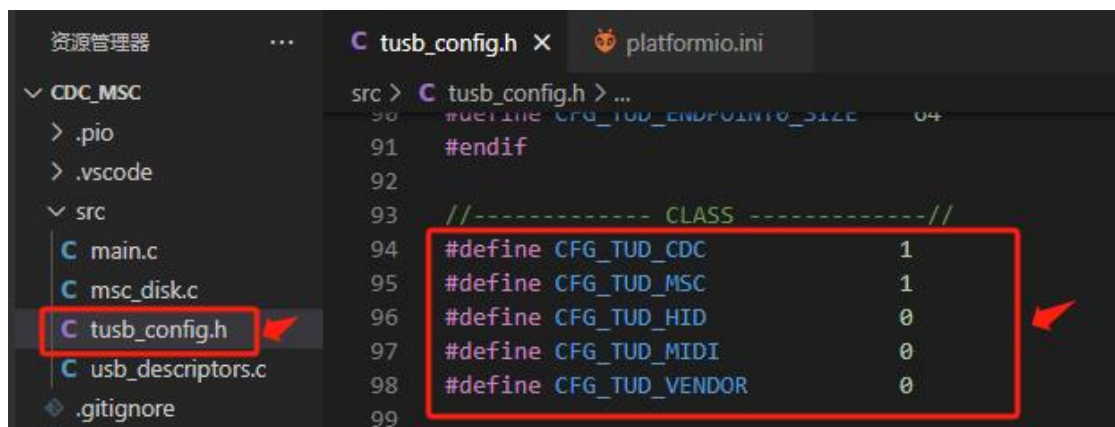
`Main.c` 文件重命名；`main()` 函数重命名；去除 `main()` 中的 `board_init` 函数；

重命名后的 `main` 函数在 `example` 下的 `main()` 中调用；

4. 编译并烧录 `ve` 和代码，即可正常运行。

在例程中，USB 描述符、回调、配置（CDC、HID、MSC、MIDI）均已通过接口开放出来，在 `src` 路径下的 `.c.h` 中。用户可根据自己的需求订制或修改。

可通过头文件中的宏来配置要开放的枚举类型：



更多配置部分及 usb 接口使用详解，可参考 sdk 下 tinyUSB 路径下的文件描述，或者参考 tinyUSB 官方介绍。

十一、MAC 的使用：

AG32 支持 MAC 模块。

支持 RMII/MII 接口。

目前 SDK 中集成了 Lwip2.1.0 版本。在 lwip 样例中，使用了 server 端的功能。


样例使用：

打开样例工程 lwip，

example	2023/5/9 17:41
freeRTOS	2023/2/1 18:28
logic_ip	2023/2/1 18:28
lwip	2023/5/10 11:29
rtthread	2023/5/10 15:56
tinyusb	2023/5/8 16:33

在开发板上测试例程时，步骤：

1. 分别编译并烧录 ve 和 code；
 2. 然后用网线连接 PC 和开发板，并修改 PC 的 IP 地址为 192.168.5.2；
 3. 在 PC 的浏览器上输入：<http://192.168.5.1>
- 此时，可以在网页上看到开发板中展示的画面：



LwIP - A Lightweight TCP/IP Stack

The web page you are watching was served by a simple web server running on top of the lightweight TCP/IP stack [lwIP](#).

LwIP is an open source implementation of the TCP/IP protocol suite that was originally written by [Adam Dunkels of the Swedish Institute of Computer Science](#) but now is being actively developed by a team of developers distributed world-wide. Since it's release, lwIP has spurred a lot of interest and has been ported to several platforms and operating systems. lwIP can be used either with or without an underlying OS.

The focus of the lwIP TCP/IP implementation is to reduce the RAM usage while still having a full scale TCP. This makes lwIP suitable for use in embedded systems with tens of kilobytes of free RAM and room for around 40 kilobytes of code ROM.

More information about lwIP can be found at the lwIP homepage at <http://savannah.nongnu.org/projects/lwip/> or at the lwIP wiki at <http://lwip.wikia.com/>.

移植到自己的板子上时，注意两项配置：

1. 根据自己的板子，可能需要修改的是 phy 地址：

```
uint8_t board_mac_phy_addr(void)
{
    // PHY address depends on how it is physically c
    return 1;
}
```

2. 修改 ve 配置文件中 mac 相关 IO 对应，如：

```
# Pins for RMII
MAC0_RXD0    PIN_59
MAC0_RXD1    PIN_60
MAC0_TXD0    PIN_62
MAC0_TXD1    PIN_63
MAC0_TX_EN   PIN_64
MAC0_CR5     PIN_58
MAC0_RX_ER   PIN_56

# Extra pins for MII
# MAC0_RX_DV   PIN_65
# MAC0_RXD2    PIN_62
# MAC0_RXD3    PIN_61
# MAC0_TXD2    PIN_56
# MAC0_TXD3    PIN_55
# MAC0_COL     PIN_42
# MAC0_RX_CLK  PIN_60 # Do NOT assign to other pins

# MAC reset, MDC, MDIO, and clocks
GPIO5_5      PIN_65 # MAC0_PHYRSTB,
MAC0_MDC     PIN_55
MAC0_MDIO    PIN_54
#MAC0_TX_CLK  PIN_61
MAC0_CLK_OUT PIN_61

# INTB is shared with RXD2 on 32-pin
MAC0_PHY_INTB PIN_57
```

上层部分，使用什么样的网络，则自行配置 lwip。

十二、SPI 的使用：

AG32 支持两路 SPI，分别对应：SPI0、SPI1；

两路是功能对等的。仅支持 SPI-Master 端。

SPI 是一种全双工同步的串行通信，可支持高速数据传输。

采用主-从模式(Master-Slave)的控制方式,通过对 Slave 设备进行片选 (Slave Select) 来控制多个 Slave 设备。

样例程序参考 `example_spi.c`

在例程中,使用了 SPI_FLASH 的 dma 方式。

注意,这里的 SPI 驱动都是针对 SPI_FLASH 的封装,并不用于通用 SPI。

SPI 支持 1 线、2 线和 4 线。

如果是用于通用 SPI 外设(非 FLASH),请参考样例程序 `example_spi_common.c`。

需要注意的是:

由于这里 SPI 底层是针对 FLASH 使用的封装,所以对通用 SPI 外设支持并不全面。

如果要用于普通外设,则该外设必须满足如下时序:

1. SPI 交互时第一段只能是 tx (不能是 rx);

2. 收和发不能同时进行(只能是发完再收);

更详细的使用说明和样例,请参考 demo 和 datasheet。

补充:

从 SDK1.2.4 版本开始,增加了对通用 SPI 的支持。

原 `example_spi_common.c` 中提供的函数:

1. Send: 单纯发送数据,字节数不限制;

2. SendAndRecv: 在一个片选周期内,发送一段数据,再接收一段数据;

其中发送长度最长 4byte,接收长度不限。

(如果发送长度要更长,请自行在 C 驱动中扩展。)

这个版本开始,会在以上基础上,扩展出来两个函数:

3. Recv: 单纯收取数据,字节数现在最长是 16byte; (可扩展)

4. SendWithRecv: 在发送数据的同时来收(双向传输),而不是发完后再收。

收取数据和发送数据的长度等长。现在长度最大是 16byte; (可扩展)

但这两个函数,需要 cpld 的支持,用起来比较费劲。

更详细的使用说明,请参考《AG32 下 spi 的拓展使用.pdf》

十三、ADC/DAC 的使用:

ADC/DAC 包含模拟电路,需要 fpga 部分的支持。

AG32 自带一套 fpga 逻辑(默认 ip),

在默认的 ip 中,支持 3 路 ADC 和 2 路 DAC,1 路比较器 CMP(双通道,可独立运行)。

使用样例 ADC:

在样例代码 `example_analog.c` 中,adc 默认是宏关闭的。可在 `platformio.ini` 中打开该宏:

```
build_flags = -DBAUD_RATE=${setup.monitor_speed} -DIPS_ANALOG_IP
```

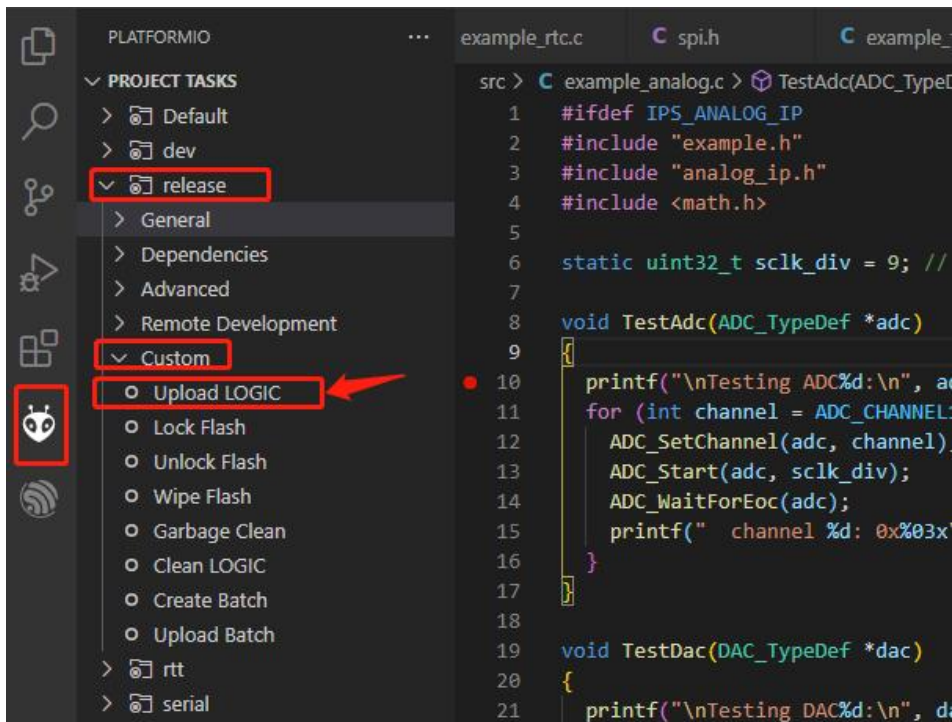
【-DIPS_ANALOG_IP】

同时,使能默认的 ip,在 `platformio.ini` 中配置:

```
#ips_dir = ../ips
ip_name = analog_ip
#logic_dir = logic
```

然后在 main() 函数中放开 TestAnalog() 即可。

注意，第一次打开 ADC/DAC 功能时，需要重新编译烧录一次 ve:



ADC 共有 15 个 channel，每个 channel 均可配置到任一个 ADC 上。

ADC 的简单使用:

参考 TestAdc 函数，ADC 不需要在 ve 里管脚映射，不需要设置 IO 复用。

使用以下 4 个函数即可:

```
ADC_SetChannel(adc, channel);
ADC_Start(adc, sclk_div);
ADC_WaitForEoc(adc);
printf(" channel %d: 0x%03x\r\n", channel - ADC_CHANNEL0, ADC_GetData(adc));
```

如果需要多次转换，则重复调用后 3 个函数。

DAC/比较器/dma 参后续例程。

十四、WatchDog 的使用:

AG32 支持 1 个独立看门狗模块。

WDOG 主要性能:

- 自由运行的递减计数器
 - 看门狗被激活后，则在计数器计数至 0x000 时产生复位
- 默认情况下，在 debug 状态下看门狗是不工作的。

使用逻辑：

1. 为看门狗使能时钟，并使能中断（中断可选）；
 SYS_EnableAPBClock(APB_MASK_WATCHDOG);
 INT_EnableIRQ(WATCHDOG0_IRQn, WDOG_PRIORITY);
2. 启动看门狗，同时设置看门狗时间；
 WDOG_Init(SYS_GetPclkFreq()); // 1 second
3. 定时（或在中断函数中）喂狗；
 WDOG_Feed();
 看门狗中断函数为：void WATCHDOG0_isr();
4. 系统启动后，可以通过查看寄存器，确定是否为看门狗导致的重启；
 if (READ_BIT(SYS->RST_CNTL, SYS_RSTF_WDOG)) { /*reset by Watchdog*/}
 在处理中需要自行清除掉该标记，以避免下次重启时误判。

看门狗中是否使用中断：

1. 如果开启看门狗中断，则中断来了后，必须要在中断里清除标记（清除中断的动作就是喂狗的动作），不然中断函数会一直被调用。
2. 如果关闭看门狗中断，则必须要应用程序在重启时间到来前及时喂狗。
3. 还可以中断函数和应用程序两者一起喂狗。即：开启看门狗中断，然后在中断函数和应用中都调用喂狗函数。
 如果应用中的喂狗周期比中断周期短，则中断函数永远不会被触发。

看门狗的中断时间和重启时间：

在 WDOG_Init 函数中设置的时间，是看门狗中断来一次的时间。而重启的时间，是两个这样的时间（2 倍整）。

比如：WDOG_Init 设置了 5 秒，没启动看门狗中断，当应用中 10 秒没喂狗动作时，系统才被重启。

十五、RTC 的使用：

RTC（Real Time Clock）是个独立的定时器。

RTC 模块拥有一个连续计数的计数器，可进行软件配置，提供时钟日历的功能。RTC 还包含用于管理低功耗模式的自动唤醒单元。

只要芯片的备用电源一直供电，在 mcu 断电情况下 RTC 仍可以独立运行。

RTC 只支持 LSE 作为时钟源（32768）；

支持 3 种中断类型：

1. 秒中断；
2. 溢出中断；
3. 定时中断；

主要寄存器：

RTC 控制寄存器 (RTC_CRH, RTC_CRL)

RTC 预分频装载寄存器 (RTC_PRLH, RTC_PRLl)

RTC 预分频余数寄存器 (RTC_DIVH, RTC_DIVL)

RTC 计数器寄存器 (RTC_CNTH, RTC_CNTL)

RTC 闹钟寄存器 (RTC_ALRH , RTC_ALRL)

以上寄存器都有对应的函数来进行操作。

执行逻辑:

RTC_PRL (预分频装载寄存器) 的值决定 TR_CLK 脉冲产生的周期, RTC_DIV (预分频器余数寄存器) 可读不可写, 当 RTCCLK 的一个上升沿到来, RTC_DIV 的值减 1, 减到 0 后硬件重载为 RTC_PRL 的值同时产生一个 TR_CLK 脉冲, 一个 TR_CLK 脉冲的到来会使 RTC_CNT (计数器寄存器) 的值加 1, 同时产生一个 RTC_Second 中断 (由软件配置是否使能, “秒中断”并不一定是一秒触发一次, 具体是根据 RTC 时钟和 RTC_PRL 的值决定)。当 RTC_CNT 的值溢出后从 0 开始, 并产生一个溢出中断 (由软件配置是否使能)。当 RTC_CNT 等于 RTC_CNTRTC_ALR (闹钟寄存器) 时, 产生一个闹钟中断 (由软件配置是否使能, 可在用在系统待机模式下唤醒系统)。

BKP 备份寄存器

备份寄存器有 16 组 16 位的寄存器 (每组 2 个)。可用来存储 64 个字节数据。

它们处在备份区域, 当 VDD 电源切断, 仍然由 VBAT 维持供电。

当系统在待机模式下被唤醒, 或者系统复位或者电源复位, 它们也不会复位。

一般用 BKP 来存储 RTC 的校验值或者记录一些重要的数据。

可通过以下两个函数接口来读写:

```
RTC_WriteBackupRegister(uint16_t idx, uint16_t value)
```

```
RTC_ReadBackupRegister(uint16_t idx)
```

RTC 常用于三种定时:

1. 秒中断:

RTC 的秒中断功能类似 SysTick 系统滴答的功能。RTC 秒中断功能其实是每计数一次就中断一次。注意, 秒中断并非一定是一秒的时间, 它是由 RTC 时钟源和分频值决定的“秒”的时间, 当然也是可以做到 1 秒钟中断一次。通常通过函数 RTC_SetPrescaler(32768) 来进行设置。

完整代码需要:

```
RTC_Init(board_rtc_source());  
RTC_EnableInt(RTC_FLAG_SEC);  
RTC_SetPrescaler(32768);  
RTC_SetOutputMode(RTC_OUTPUT_CLOCK);
```

2. 溢出中断:

溢出中断是 RTC_CNT 的值溢出时触发的中断。

3. 定时中断:

使用时一般设置秒中断周期为 1s, 用 RTC_CNT 计数器计数。假如 1970 设置为时间起点为 0s, 通过当前时间的秒数计算得到当前的时间。RTC_ALR 是设置闹钟时间, RTC_CNT 计数到 RTC_ALR 就会产生计数中断。

中断函数在 SDK 中已经默认关联, 函数名: RTC_isr

在中断函数中, 需要先判断中断来源, 再进行相应的处理。

十六、中断说明:

RISC-V 系统支持中断嵌套。但 SDK 中推荐（并默认）使用非嵌套中断方式，如果需要嵌套方式，请打开宏 `AGRV_NESTED_INTERRUPT`。

打开宏的方式，请在 `platformio.ini` 里通过 `build_flags = -DAGRV_NESTED_INTERRUPT` 来实现。使能中断可嵌套后，高优先级（数字越大优先级越高）的中断，会打断正在执行中的低优先级的中断。

中断系统被封装在 SDK 的 `interrupt.c` 中，由 `INT_Init()` 函数来完成初始化。

RISC-V 有两套中断向量，分别对应于 PLIC 和 CLINT。目前只有 MTime 是对应到 CLINT 中断，其他都对应于 PLIC 中断。

用户级别的中断设置，都通过函数 `INT_EnableIRQ(uint32_t irq, uint32_t priority)` 来设置使能。中断向量表和中断函数名都已内置定义。从用户角度，只需要设置中断使能即可使用对应的中断函数。

开关系统总中断函数：

`INT_EnableIntGlobal/INT_DisableIntGlobal`

系统中中断向量表及中断函数名，可从 `AltaRiscv.h` 中查看。如：

向量 ID: `TIMERO_IRQn`

中断函数: `TIMERO_isr`

异常和中断都在这里处理。

程序运行中出现异常的解决思路：

如果程序运行中跑飞（进入异常中断函数 `exception_handler`），可以查看几个寄存器：

mepc: 进入异常前的 pc 地址，结合编译 map 可以定位到是从哪个函数飞掉的；

mcause: 是 machine cause register，记录进入异常的原因。

异常列表如下：

0	Instruction address misaligned
1	Instruction access fault
2	Illegal instruction
3	Breakpoint
4	Load address misaligned
5	Load access fault
6	Store/AMO address misaligned
7	Store/AMO access fault
8	Environment call from U-mode
9	Environment call from S-mode
10	<i>Reserved</i>
11	Environment call from M-mode
12	Instruction page fault
13	Load page fault

十七、系统休眠（sleep、stop、stanby）

AG32 支持 3 种休眠方式：sleep、stop、standby。

代码样例参考 `example_system.c`

其中进入 standby 后，有三种唤醒方式：IWDG、RTC(Alarm)、外部 IO。
如果要使用低功耗，在系统进入休眠前，需要先关掉不必要的外设时钟。

十八、使用自定义的 logic:

上边章节“ADC/DAC 的使用”部分，描述了使用默认 logic 的方法。
默认 logic 中只包含了 ADC/DAC/CMP 的功能，如果有额外需求，则需要构建自定义 logic。
在自定义 logic 中，可以编写 cpld，为芯片增加更多的功能支持。
构建的详细流程，参考《AG32 下 fpga 和 cpld 的使用入门.pdf》。

在构建自定义 logic 时，需要 platformio.ini 中设置三项：

```
ip_name = xxxxx_ip  
logic_dir = logic  
board_logic.ve = project_xxx.ve
```

其中，

ip_name 为新建的 user_ip 名字；

logic_dir 为生成的文件夹名称；

board_logic.ve 为生成自定义 ip 时共同使用的 ve 文件；

总结：

三种情况（不用 ADC、仅用 ADC/DAC/CMP、使用更多的 cpld 功能），在 platformio.ini 文件中的配置对比：

1. 如果连默认 ip 都用不到（没有用 ADC/DAC/CMP）：

只需要配置一项：

```
board_logic.ve = project_xxx.ve
```

2. 如果仅用到默认 ip（使用到 ADC/DAC/CMP）：

需要配置两项：

```
board_logic.ve = project_xxx.ve
```

```
ip_name = analog_ip
```

3. 如果要用到自定义 logic（需要更多的 cpld 功能）：

需要配置三项：

```
board_logic.ve = project_xxx.ve
```

```
ip_name = xxxxx_ip
```

```
logic_dir = logic
```

关于自定义 logic，这里仅是配置说明，更多信息参考 cpld 部分的说明。

十九、片内 flash 的使用：

片内 flash 除了烧录 code 和 cpld 的 bin 外，多余出来的空间，可以用于存储信息。

片内 flash 的大小：

AG32 芯片有 256K 和 1M 两种大小，看实际使用的芯片对应的型号。

注意，不管 flash 的大小是哪种，默认情况下最后 100K 都是用来装载 cpld 的。

flash 的起始地址是 0x80000000。默认情况下 code 的 bin 是从这个地方开始装载。
片内 flash 的擦除单位是 4K。读和写都需要 4 字节对齐。

在使用片内 flash 之前，请先确认 code 的 bin 的大小，cp1d 的 bin 的大小。确保自己操作的区域是和两个 bin 是错开的。

如果工程中有使用 bootloader，或者使用“分散加载”，对 flash 的使用会更复杂，请自行安排区域的划分。

代码中如何调用 flash 的读写：

请参考 example\example_flash.c 的样例。

写操作前需要先解锁：FLASH_Unlock();

然后可以擦除：FLASH_Erase(start_addr, erase_len); //需要时，以 sector 为单位擦除

然后写指定长度：FLASH_FastProgram(write_addr, buff, len);

最后再锁定：FLASH_Lock();

如果是读取 flash，则不用解锁，直接按地址读取即可，可通过宏 RD_REG。

注意，读和写都需要 4 字节对齐。

二十、其他常用项：

1. 获取芯片唯一 ID：

通过如下方式来获取：

```
uint32_t id[4];
```

```
FLASH_Unlock();
```

```
FLASH_GetUniqueID(id);
```

```
FLASH_Lock();
```

芯片内部唯一 ID，其实使用的是片内 flash 的唯一 ID。

这里读出来是 16 个 BYTE。

如果这个记录太长，想用 1 个 int 型来记录，可以使用读出来的第三个 int（即：id[2]）。

2. 堆栈大小的设置：

AG32 中默认的栈 stack 大小是 0x1000（即：4K）

如果要改变栈的大小，可以在 platformio.ini 中加入：

```
build_flags = -Wl,--defsym=__stack_size=0x1000
```

而堆的大小，是自动设置的（无须手工设置）。

在 128K 的 sram 里，除去静态/全局变量数组的空间，除去栈 stack 的空间，剩下的都被自动分配为堆 heap 的空间（就是 malloc 可以申请到的空间）。