武汉芯源半导体有限公司

WUHAN XINYUAN SEMICONDUCTOR CO., LTD

运用 DMA 功能实现高级定时器和 ADC 的同步触发采样

应用笔记

版本号：Rev 1.0

前言

在做 BLDC 电机控制时，采样时刻和定时器产生的 PWM 波形相配合，才能获取准确的采样值，本文介绍了 CW32x030（x 是芯片系列，x=F、A，下同）系列芯片通过运用 DMA 功能实现高级定时器和 ADC 的同步触发采样的功能。

目录

前言 1
1 PWM 输出实现 3
1.1 输出端口的配置 3
1.2 ATIM 的配置 5
1.3 ADC 的配置 7
1.3.1 序列采样 7
1.3.2 DMA 扩展采样 9
2 版本信息 14

1 PWM 输出实现

1.1 输出端口的配置

根据 GPIO 复用功能分配表（完整表格请参阅 CW32x030 用户手册中表 9-2 GPIO 复用功能分配表），选取期望输出互补 PWM 波形的引脚，如本例中 PA08、PA09、PA10、PB13、PB14、PB15，如下表所示：

表 1-1 GPIO 复用功能分配表

引脚名称	复用功能							
	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7
PA08	GPIO		UART1_TXD	BTIM2_TOGN	MCO_OUT	LVD_OUT	GTIM3_ETR	ATIM_CH1A
PA09	GPIO	UART3_TXD	UART1_RXD	I2C1_SCL	BTIM1_TOGP	SPI1_CS	GTIM3_CH1	ATIM_CH2A
PA10	GPIO	UART3_RXD	UART1_CTS	I2C1_SDA	BTIM1_TOGN	SPI1_SCK	GTIM3_CH2	ATIM_CH3A
PB13	GPIO	GTIM2_TOGP	GTIM4_CH3	I2C2_SCL	SPI2_SCK	SPI1_SCK	GTIM1_TOGP	ATIM_CH1B
PB14	GPIO	GTIM2_CH1	GTIM4_CH2	I2C2_SDA	SPI2_MISO	SPI1_MISO	RTC_OUT	ATIM_CH2B
PB15	GPIO	GTIM2_CH2	GTIM4_CH1	BTIM2_TOGP	SPI2_MOSI	SPI1_MOSI	RTC_1Hz	ATIM_CH3B

PA08 和 PB13 组成一对互补输出通道 CH1，PA09 和 PB14 组成一对互补输出通道 CH2，PA10 和 PB15 组成一对互补输出 CH3。

步骤如下：

1. 将相关的 GPIO 设置为输出；
2. 将 GPIO 配置为 ATIM 的比较输出复用功能。

代码如下：

```
void PWM_GPIOConfig(void)

{
    /* PA8   PB13   CH1A   CH1B
       PA9   PB14   CH2A   CH2B
       PA10  PB15   CH3A   CH3B
    */
    GPIO_InitTypeDef GPIO_InitStruct;

    __RCC_GPIOA_CLK_ENABLE();          // 使能 GPIO 的时钟，以便配置 GPIO 的寄存器
    __RCC_GPIOB_CLK_ENABLE();

    GPIO_InitStruct.IT = GPIO_IT_NONE;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pins = GPIO_PIN_8 | GPIO_PIN_9 | GPIO_PIN_10;
    GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
    GPIO_Init(CW_GPIOA, &GPIO_InitStruct);
```

```
    GPIO_InitStruct.Pins = GPIO_PIN_13 | GPIO_PIN_14 | GPIO_PIN_15;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_Init(CW_GPIOB, &GPIO_InitStruct);

    PA08_AFx_ATIMCH1A();        // GPIO 端口功能复用
    PA09_AFx_ATIMCH2A();
    PA10_AFx_ATIMCH3A();
    PB13_AFx_ATIMCH1B();
    PB14_AFx_ATIMCH2B();
    PB15_AFx_ATIMCH3B();
}
```

1.2 ATIM 的配置

ATIM 包含一个 16 位的计数器 / 定时器和 7 个比较单元。7 个比较单元中，有六个具有捕获功能，并且这 6 个捕获 / 比较单元可以成对使用，组成互补输出的功能。

本文以产生一个驱动 BLDC 电机所需的 20kHz 的三路互补输出的 PWM 波形为例，选取 ATIM 的时基信号为 PCLK。

本例中 PCLK 为 64MHz，并通过 ATIM 的预分频器进行 16 分频后，以 4MHz 频率进行计数。

为方便设定 ADC 的采样时间，ATIM 采用中央对齐模式计数，设置 ATIM 的自动重载寄存器（ARR）为 100，则 ATIM 的将先从 0 累加至 99，再从 100 递减至 1，故计数周期为 2 倍的 ARR 寄存器的值，即 PWM 的频率为 20kHz。

通过设置 ATIM 的控制寄存器（CR）的 COMP 位为 1，使得 PWM 以互补的方式输出，CH1A 和 CH1B 的脉宽由通道 1 比较 / 捕获寄存器 A（CH1CCRA）决定，CH1B 的输出脉宽不再由通道 1 比较 / 捕获寄存器 B（CH1CCRB）决定，CH1CCRB 仍可用于设定 CH1B 比较匹配的值。CH2A 和 CH2B，CH3A 和 CH3B 与之类似。

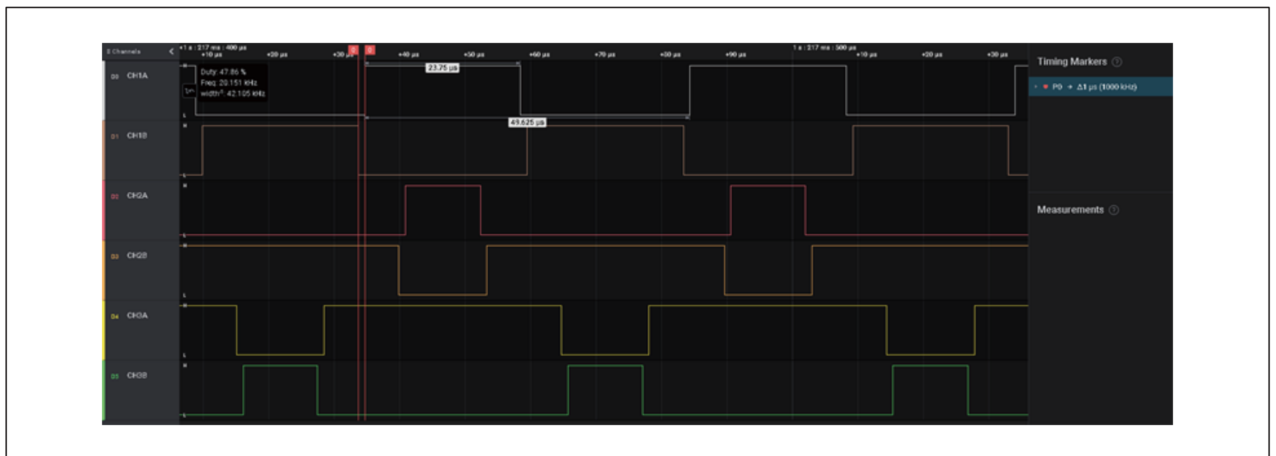
在设置输出 PWM 互补输出时，可以对互补通道加入死区时间，由死区时间寄存器（DTR）控制。

ATIM 配置输出 3 对互补带死区的 PWM 波形，详细配置代码如下：

```
void ATIM_Config(void)
{
    ATIM_InitTypeDef ATIM_InitStruct;
    ATIM_OCInitTypeDef ATIM_OCInitStruct;

    __RCC_ATIM_CLK_ENABLE();          // 开启 ATIM 的配置时钟，以便配置寄存器
    ATIM_InitStruct.BufferState = DISABLE;          // 不使能缓存寄存器
    ATIM_InitStruct.ClockSelect = ATIM_CLOCK_PCLK;          // 选择 PCLK 时钟计数
    ATIM_InitStruct.CounterAlignedMode = ATIM_COUNT_MODE_CENTER_ALIGN;    // 中心对齐
    ATIM_InitStruct.CounterDirection = ATIM_COUNTING_UP;        // 向上计数；中心对齐方式下此项无效
    ATIM_InitStruct.CounterOPMode = ATIM_OP_MODE_REPETITIVE;         // 连续运行模式
    ATIM_InitStruct.OverFlowMask = ENABLE;          // 重复计数器上溢出屏蔽
    ATIM_InitStruct.Prescaler = ATIM_Prescaler_DIV16;          // 16 分频 , 计数时钟 4MHz
    ATIM_InitStruct.ReloadValue = 100;          // 从 0 加到 ARR-1, 再从 ARR 减到 1（20kHz）
    ATIM_InitStruct.RepetitionCounter = 0;          // 重复周期 0
    ATIM_InitStruct.UnderFlowMask = ENABLE;          // 重复计数下溢出屏蔽
    ATIM_Init(&ATIM_InitStruct);          // 配置 ATIM 的基础配置

    ATIM_OCInitStruct.BufferState = DISABLE;          // 比较输出缓冲不使能
    ATIM_OCInitStruct.OCDMAState = DISABLE;          // 比较匹配时不触发 DMA 功能
    ATIM_OCInitStruct.OCInterruptSelect = ATIM_OC_IT_NONE;          // 比较匹配时不产生中断
    ATIM_OCInitStruct.OCInterruptState = DISABLE;          // 比较匹配时中断不使能
    ATIM_OCInitStruct.OCMode = ATIM_OCMODE_PWM2;          // 比较输出采用 PWM2 模式
    ATIM_OCInitStruct.OCPolarity = ATIM_OCPOLARITY_NONINVERT;          // 输出电平不反相
    ATIM_OC1AInit(&ATIM_OCInitStruct);          // 配置 CH1、CH2、CH3 输出比较
    ATIM_OC2AInit(&ATIM_OCInitStruct);
    ATIM_OC3AInit(&ATIM_OCInitStruct);
```

```
    ATIM_OC1BInit(&ATIM_OCInitStruct);
    ATIM_OC2BInit(&ATIM_OCInitStruct);
    ATIM_OC3BInit(&ATIM_OCInitStruct);

    ATIM_SetCompare1A(ATIM_InitStruct.ReloadValue * 50/100);        // 通道 1 输出占空比 50% 的 PWM
    ATIM_SetCompare2A(ATIM_InitStruct.ReloadValue * 25/100);        // 通道 2 输出占空比 75% 的 PWM
    ATIM_SetCompare3A(ATIM_InitStruct.ReloadValue * 75/100);        // 通道 3 输出占空比 25% 的 PWM
    ATIM_PWMOutputConfig(OCREFA_TYPE_SINGLE, OUTPUT_TYPE_COMP, 2);        // 互补输出，插入
    死区，当前 4MHz 时钟，1us 死区，死区计算见用户手册
    ATIM_CtrlPWMOutputs(ENABLE);        // 开启 PWM 输出
    ATIM_Cmd(ENABLE);        // 开启 ATIM
}
```

产生的 PWM 波形如下：



1.3 ADC 的配置

在做 BLDC 电机控制时，需要通过 ADC 采集母线电压、母线电流、相电压、相电流、反相电动势等的变化，因此需要 ADC 能对多个模拟量进行转换。CW32x030 的片上 ADC 具有 13 个外部输入通道，可以满足对采样数量的需求。

1.3.1 序列采样

当需求的采样通道小于等于 4 路时，可以通过 ADC 的序列采样模式实现，并且可以通过 ATIM 的通道 1~3 比较 / 捕获寄存器 B 中任意一个寄存器设定 ADC 的采样时刻，这些操作都可以由硬件自动完成，减轻了 CPU 的工作量。

以采样 AIN0~AIN3 这 4 路输入为例，设定采样时刻为 ATIM 计数达到 ARR 时，其参考代码如下：

```
static void ADC_Config(void)
{
    ADC_InitTypeDef ADC_InitStruct;
    ADC_SerialChTypeDef ADC_SerialChStruct;
    ATIM_OCInitTypeDef ATIM_OCInitStruct;

    __RCC_ADC_CLK_ENABLE();

    ADC_InitStruct.ADC_AccEn = ADC_AccDisable;          // ADC 累加功能不开启
    ADC_InitStruct.ADC_Align = ADC_AlignRight;          // 采样结果右对齐，即结果存于 bit11~bit0
    ADC_InitStruct.ADC_ClkDiv = ADC_Clk_Div4;           // ADC 的 采 样 时 钟 为 PCLK 的 4 分 频， 即
    ADCCLK=16MHz
    ADC_InitStruct.ADC_DMAEn = ADC_DmaDisable;          // DMA 触发 ADC 不使能
    ADC_InitStruct.ADC_InBufEn = ADC_BufDisable;        // 高速采样，ADC 内部电压跟随器不使能
    ADC_InitStruct.ADC_OpMode = ADC_SerialChScanMode;   // 序列扫描转换模式
    ADC_InitStruct.ADC_SampleTime = ADC_SampTime5Clk;   // 设置为 5 个采样周期，须根据实际
    情况调整
    ADC_InitStruct.ADC_TsEn = ADC_TsDisable;            // 内部温度传感器禁止
    ADC_InitStruct.ADC_VrefSel = ADC_Vref_VDDA;         // 采样参考电压选择为 VDDA

    ADC_SerialChStruct.ADC_InitStruct = ADC_InitStruct;     // 序列采样的基本配置项
    ADC_SerialChStruct.ADC_Sqr0Chmux = ADC_SqrCh0;          // 序列 0 配置为外部输入 0
    ADC_SerialChStruct.ADC_Sqr1Chmux = ADC_SqrCh1;          // 序列 1 配置为外部输入 1
    ADC_SerialChStruct.ADC_Sqr2Chmux = ADC_SqrCh2;          // 序列 2 配置为外部输入 2
    ADC_SerialChStruct.ADC_Sqr3Chmux = ADC_SqrCh3;          // 序列 3 配置为外部输入 3
    ADC_SerialChStruct.ADC_SqrEns = ADC_SqrEns03;           // 使能序列 0~3

    ADC_SerialChScanModeCfg(&ADC_SerialChStruct);           // ADC 序列扫描模式初始化

    ADC_ExtTrigCfg(ADC_TRIG_ATIM, ENABLE);          // 配置外部触发 ADC 的启动的触发源为 ATIM
```

```
    /* 修改 ATIM 的部分配置，使能 ATIM 的 CH1B 通道发生比较匹配时，触发 ADC 开始转换 */
    ATIM_ADCTriggerConfig(ATIM_ADC_TRIGGER_CH1B, ENABLE);        // 允许 ATIM 的 CH1B 在比较匹配时触发 ADC 启动

    // 修改 ATIM 的 CH1CR 寄存器，使用 CH1B 的在下计数比较匹配时触发 ADC，注：0~ARR-1 为上计数方向
    // ARR~1 为下计数方向
    REGBITS_MODIFY(CW_ATIM->CH1CR, ATIM_CH1CR_CISB_Msk, ATIM_OC_IT_DOWN_COUNTER << ATIM_CH1CR_CISB_Pos);
    ATIM_SetCompare1B(CW_ATIM->ARR);        // 设置 CH1B 比较匹配的值为 ARR

    ADC_Enable();
}
```

上述方法完全由硬件实现，不需要 CPU 和中断的参与，执行效率非常高，不足的地方是采样通道限制为 4 路。

1.3.2 DMA 扩展采样

如果需要对超过 4 路的模拟量进行采样，则需要结合 DMA 的功能，以实现较少的 CPU 参与。其思路如下：

1. ADC 配置为单通道单次转换，完成转换后硬件触发 DMA；
2. DMA 的 CH1 用于将 ADC 的转换结果传输到 RAM 中，本例中将采样 6 个 ADC 通道，因此传输次数 CNT 为 6，源地址固定为 ADC 的 RESULT0 寄存器，目的地址需要递增；
3. DMA 的 CH2 用于更改 ADC 的采样通道，当 ADC 转换完成后，从 RAM 中取 ADC 的通道配置参数，自动配置 ADC 的寄存器值，因此源地址为 RAM，地址递增，目的地址为 ADC 的通道控制寄存器；
4. DMA 的 CH3 用于再次启动 ADC，因为 ADC 配置为单次转换，当转换完成后，ADC 自动停止转换，所以需要通过 DMA 向 ADC 的转换启动寄存器置位，以再次启动 ADC 转换；
5. DMA 的 CH1 传输完成后，ADC 的 6 路转换也完成了，并且转换结果也被传输到 RAM，可通过 CH1 的传输完成中断，将 DMA 的参数重新配置，就实现了多路 ADC 的循环采样；
6. 通过 ATIM 的比较通道 4，去触发 DMA 的 CH4，向 ADC 的转换启动寄存器置位，启动 ADC。

其参考代码如下：

- ADC 的配置：

```
static void ADC_Config(void)
{
    ADC_InitTypeDef ADC_InitStruct;

    __RCC_ADC_CLK_ENABLE();

    ADC_InitStruct.ADC_AccEn = ADC_AccDisable;          // ADC 累加功能不开启
    ADC_InitStruct.ADC_Align = ADC_AlignRight;          // 采样结果右对齐，即结果存于 bit11~bit0
    ADC_InitStruct.ADC_ClkDiv = ADC_Clk_Div4;           // ADC 的 采 样 时 钟 为 PCLK 的 4 分 频， 即 ADCCLK=16MHz
    ADC_InitStruct.ADC_DMAEn = ADC_DmaEnable;           // ADC 转换完成触发 DMA
    ADC_InitStruct.ADC_InBufEn = ADC_BufDisable;        // 高速采样，ADC 内部电压跟随器不使能
    ADC_InitStruct.ADC_OpMode = ADC_SingleChOneMode;    // 单次单通道采样模式
    ADC_InitStruct.ADC_SampleTime = ADC_SampTime5Clk;   // 设置为 5 个采样周期，须根据实际情况调整
    ADC_InitStruct.ADC_TsEn = ADC_TsDisable;            // 内部温度传感器禁止
    ADC_InitStruct.ADC_VrefSel = ADC_Vref_VDDA;         // 采样参考电压选择为 VDDA

    ADC_Init(&ADC_InitStruct);          // 初始化 ADC 的配置

    CW_ADC->CR1_f.CHMUX = 0;          // 初始采样通道为 AIN0
    ADC_Enable();          // 启用 ADC
}
```

ADC 配置为单次单通道采样，采样完成后可触发 DMA。

- DMA 的配置：

```
uint16_t ADC_ResultBuff[6] = {0};          // 用于存放 6 路采样结果
uint8_t ADC_CR1Array[5] = {0x81, 0x82, 0x83, 0x84, 0x85};          // 通过更改 ADC 的 CR1 寄存器实现 ADC 通道自动切换
uint8_t ADC_Start = 0x01;          // ADC_START 寄存器的配置值

void DMA_Config(void)
{
    // 使用 4 路 DMA 通道：CH1、CH2、CH3、 CH4
    // CH1 将 ADC 单次单通道的采样结果传入 RAM（ADC_ResultBuff[6]）
    // CH2 将 ADC 的 CR1 寄存器的配置值从 RAM（ADC_CR1Array）传入寄存器
    // CH3 将 ADC 的 START 寄存器的配置值从 RAM（ADC_Start) 传入寄存器
    // CH1、CH2、CH3 由 ADC 硬件触发
    // CH4 由 ATIM 硬件触发 , 启动 ADC
    DMA_InitTypeDef DMA_InitStruct = {0};

    __RCC_DMA_CLK_ENABLE();

    DMA_InitStruct.DMA_DstAddress = (uint32_t)&ADC_ResultBuff[0];          // 目标地址
    DMA_InitStruct.DMA_DstInc = DMA_DstAddress_Increase;          // 目标地址递增
    DMA_InitStruct.DMA_Mode = DMA_MODE_BLOCK;          // BLOCK 传输模式
    DMA_InitStruct.DMA_SrcAddress = (uint32_t)&CW_ADC->RESULT0;          // 源地址： ADC 的结果寄存器
    DMA_InitStruct.DMA_SrcInc = DMA_SrcAddress_Fix;          // 源地址固定
    DMA_InitStruct.DMA_TransferCnt = 0x6;          // DMA 传输次数
    DMA_InitStruct.DMA_TransferWidth = DMA_TRANSFER_WIDTH_16BIT;          // 数据位宽 16bit
    DMA_InitStruct.HardTrigSource = DMA_HardTrig_ADC_TRANSCOMPLETE;          // ADC 转换完成硬触发
    DMA_InitStruct.TrigMode = DMA_HardTrig;          // 硬触发模式
    DMA_Init(CW_DMACHANNEL1, &DMA_InitStruct);
    DMA_Cmd(CW_DMACHANNEL1, ENABLE);

    DMA_InitStruct.DMA_DstAddress = (uint32_t)&CW_ADC->CR1;          // 目标地址
    DMA_InitStruct.DMA_DstInc = DMA_DstAddress_Fix;          // 目标地址固定
    DMA_InitStruct.DMA_Mode = DMA_MODE_BLOCK;          // BLOCK 传输模式
    DMA_InitStruct.DMA_SrcAddress = (uint32_t)&ADC_CR1Array[0];          // 源地址
    DMA_InitStruct.DMA_SrcInc = DMA_SrcAddress_Increase;          // 源地址递增
    DMA_InitStruct.DMA_TransferCnt = 0x5;          // DMA 传输次数
    DMA_InitStruct.DMA_TransferWidth = DMA_TRANSFER_WIDTH_8BIT;          // 数据位宽 8bit
    DMA_InitStruct.HardTrigSource = DMA_HardTrig_ADC_TRANSCOMPLETE;          // ADC 转换完成硬触发
    DMA_InitStruct.TrigMode = DMA_HardTrig;          // 硬触发模式
    DMA_Init(CW_DMACHANNEL2, &DMA_InitStruct);
    DMA_Cmd(CW_DMACHANNEL2, ENABLE);
```

```
    DMA_InitStruct.DMA_DstAddress = (uint32_t)&CW_ADC->START;           // 目标地址
    DMA_InitStruct.DMA_DstInc = DMA_DstAddress_Fix;          // 目标地址固定
    DMA_InitStruct.DMA_Mode = DMA_MODE_BLOCK;          // BLOCK 传输模式
    DMA_InitStruct.DMA_SrcAddress = (uint32_t)&ADC_Start;           // 源地址
    DMA_InitStruct.DMA_SrcInc = DMA_SrcAddress_Fix;          // 源地址固定
    DMA_InitStruct.DMA_TransferCnt = 0x5;           // DMA 传输次数
    DMA_InitStruct.DMA_TransferWidth = DMA_TRANSFER_WIDTH_8BIT;           // 数据位宽 8bit
    DMA_InitStruct.HardTrigSource = DMA_HardTrig_ADC_TRANSCOMPLETE;       // ADC 转换完成硬触发
    DMA_InitStruct.TrigMode = DMA_HardTrig;           // 硬触发模式
    DMA_Init(CW_DMACHANNEL3, &DMA_InitStruct);
    DMA_Cmd(CW_DMACHANNEL3, ENABLE);

    DMA_InitStruct.DMA_DstAddress = (uint32_t)&CW_ADC->START;           // 目标地址
    DMA_InitStruct.DMA_DstInc = DMA_DstAddress_Fix;          // 目标地址固定
    DMA_InitStruct.DMA_Mode = DMA_MODE_BLOCK;          // BLOCK 传输模式
    DMA_InitStruct.DMA_SrcAddress = (uint32_t)&ADC_Start;           // 源地址
    DMA_InitStruct.DMA_SrcInc = DMA_SrcAddress_Fix;          // 源地址固定
    DMA_InitStruct.DMA_TransferCnt = 0x1;           // DMA 传输次数
    DMA_InitStruct.DMA_TransferWidth = DMA_TRANSFER_WIDTH_8BIT;           // 数据位宽 8bit
    DMA_InitStruct.HardTrigSource = DMA_HardTrig_ATIM_CH1A2A3A4;          // ATIM 硬件触发
    DMA_InitStruct.TrigMode = DMA_HardTrig;           // 硬触发模式
    DMA_Init(CW_DMACHANNEL4, &DMA_InitStruct);
    DMA_Cmd(CW_DMACHANNEL4, ENABLE);

    DMA_ITConfig(CW_DMACHANNEL1, DMA_IT_TC, ENABLE);          // DMA 的 CH1 传输完成后触发中断
}
```

● ATIM 需要在之前的配置上增加通道 4 的设置，增加的代码如下：

```
{
    // 配置 OC4 比较触发 DMA
    CW_ATIM->MSCR_f.CCDS = 1;          // 允许 ATIM 打开 DMA 总开关
    // 比较通道 4 在向下计数时，比较匹配时触发 DMA
    ATIM_OC4Init(ENABLE, ATIM_OC_IT_DOWN_COUNTER, ENABLE, DISABLE, DISABLE);
    ATIM_SetCompare4(ATIM_InitStruct.ReloadValue);          // 触发时刻
}
```

● DMA 的 CH1 通道传输完成后，将触发中断服务程序，其中断服务程序的代码如下：

```
void DMACH1_IRQHandlerCallBack(void)
{
    CW_DMA->ICR_f.TC1 = 0x00;          // 清中断标志
    CW_ATIM->ICR = 0x00;          // 清 ATIM 的中断标志
    CW_ADC->ICR = 0x00;           // 清 ADC 的中断标志
    CW_ADC->CR1 = 0x80;           // 恢复 ADC 的采样通道为 AIN0
    //-------------------------------------------------
    // DMA.CH1 : 还原为初始设置，即将 ADC 的采样结果存储地址恢复到首位置
    CW_DMACHANNEL1->CNT = 0x10006;          // 传 6 个
    CW_DMACHANNEL1->SRCADDR = (uint32_t)(&CW_ADC->RESULT0);
    CW_DMACHANNEL1->DSTADDR = (uint32_t)(&ADC_ResultBuff[0]);
    CW_DMACHANNEL1->CSR_f.EN = 1;



    //-------------------------------------------------
    // DMA.CH2 : 将下次采样的通道恢复到 AIN1，重置传输次数为 5
    CW_DMACHANNEL2->CNT = 0x10005;          // 传 5 个
    CW_DMACHANNEL2->SRCADDR = (uint32_t)(&ADC_CR1Array[0]);
    CW_DMACHANNEL2->DSTADDR = (uint32_t)(&CW_ADC->CR1);
    CW_DMACHANNEL2->CSR_f.EN = 1;

    //-------------------------------------------------
    // DMA.CH3 : 重置传输次数为 5
    CW_DMACHANNEL3->CNT = 0x10005;          // 传 5 个
    CW_DMACHANNEL3->SRCADDR = (uint32_t)(&ADC_Start);
    CW_DMACHANNEL3->DSTADDR = (uint32_t)(&CW_ADC->START);
    CW_DMACHANNEL3->CSR_f.EN = 1;
```

```
    //-----------------------------------------------
    // DMA.CH4 : 重置传输次数，等待 ATIM 比较匹配后再次触发 DMA 传输
    CW_DMACHANNEL4->CNT = 0x10001;            // 传 1 个
    CW_DMACHANNEL4->SRCADDR = (uint32_t)(&ADC_Start);
    CW_DMACHANNEL4->DSTADDR = (uint32_t)(&CW_ADC->START);
    CW_DMACHANNEL4->CSR_f.EN = 1;
}
```

这种方法可以实现多于4个模拟通道的采样，采样结果自动保存在内存中，并且仅在最后一个通道采样完成后，进入一次中断服务程序对 DMA 的配置进行复位，所以 CPU 的开销是比较小的，而且可以通过 ATIM 的比较通道 4 灵活设置采样时机。

2 版本信息

表 2-1 文档修订信息

日期	版本	变更信息
2022-03-31	Rev 1.0	初始发布