AN2112 基于 EC30-EKSTM32 扩展模拟量滤波

CREATE: 2010/08/05
UPDATE: 2010/08/05
GUTTA Ladder Editor Version 1.1
Version 1.1
http://www.plcol.com
http://www.visiblecontrol.com

概述

在 EC30-EKSTM32 的系统块中，可以配置 STM32F103 的自带的 A/D 模数转换器，既 STM32F103 的硬件 ADC1。在系统块的 AD 模块中，可以设定 AIW0、AIW2 ~ AIW30 这 16 个模拟量输入对应的单片机管脚、转换速度、以及数值标定。做好这些设定，便可以在 PLC 程序的主循环 MAIN 中，通过 AIWx 读取对应管脚的模拟量输入。EC30-EKSTM32 系统在进入主循环 MAIN 之前，会根据系统块的配置，依次扫描所有需要转换的管脚，并将转换结果经过处理后，放在 AIW0、AIW2 ~ AIW30 中。

由于这样的工作方式，A/D 实际的采样周期取决于 PLC 主循环 MAIN 的工作时间。若主循环扫描周期长，A/D 采样的周期就长，若主循环扫描周期短，A/D 采样的周期就短。这种工作方式在某些需要精确确定采样周期的场合，并不适合。

例如模拟量数字滤波，就需要有一个稳定的模拟量采样频率。在某段时间内采集多个模拟量数据，做数学处理后，才能得到这段时间的对应最佳结果，供应用程序使用。

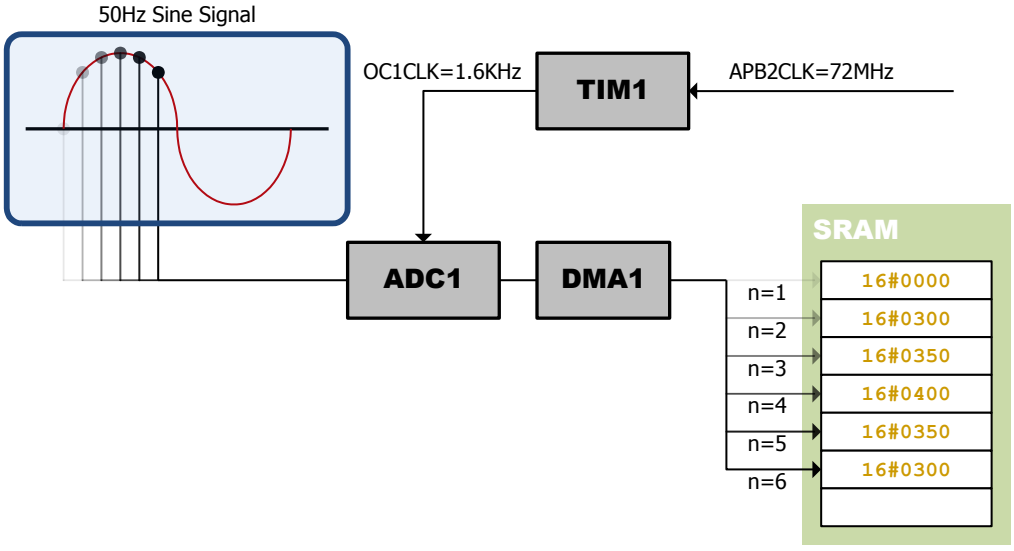
例如电力参数的测量，对于市电的 50Hz 信号（3 相电流或电压），一般需要在一个周期内采样 32 个点。基于这 32 点的采样数据，便可计算出 50Hz 信号的峰峰值、有效值。

可以通过 C 语言二次开发来实现这些驱动。既不配置系统块中的 AD 模块，而是用 C 语言自己初始化 STM32F103 的 A/D 转换硬件 ADC。由于需要周期性的采样，还要初始化一个空闲的定时器 TIM，以及 STM32F103 特有的数据通道 DMA。

实现原理

市电的频率是 50Hz，如果一个周期采样 32 个点的话，我们的采样频率应该是 50*32=1600Hz，即 1 秒钟采样 1600 次。STM32F103 的 A/D 转换最快可达到 1us，即 1 秒钟最快可以采样 100,000 次，这个速度远远高于我们的要求，可以放心使用。为了以后分析数据方便，这里按照 1600Hz 的频率，采样 64 个点。

STM32F103 的 A/D 转换可以通过某个定时器 TIM 的比较捕获信号来触发，转换结果通过 DMA 直接存储到 SRAM 中。



如图所示，TIM1 对 APB2CLK 进行分频后，通过比较捕获器 OC1 产生 1.6KHz 的捕获信号。这个信号用于驱动 A/D 转换器 ADC1。ADC1 转换结束后，驱动 DMA1。DMA1 根据配

置，将 ADC1 的转换结果按时间顺序存放在 SRAM 的连续内存中。DMA1 获得足够多的采样数据后（n=64），停止拷贝数据，等待主循环处理完数据后，方可进行下一组记录。由于使用了 DMA，整个流程不需要中断服务，对 STM32F103 的负载几乎没有影响。

实现代码

C 语言二次开发的介绍请参考 EC30-EKSTM32 手册的第二部分。这里直接介绍实现代码。由于 TIM1、ADC1、DMA1 都需要自己初始化，因此必须响应 EC30-EKSTM32 的 CUSTOM_MODULE_RESET 消息。这个消息将会调用_CUSTOM_MODULE_UNINIT 函数，我们来看看这个函数的实现。

初始化

初始化 RCC

```
    // <InitRCC>
    if (1) {
        RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
        RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1 | RCC_APB2Periph_TIM1,
ENABLE);
        }
    // </InitRCC>
```

我们需要使用 TIM1、ADC1、DMA1，因此必须先使能他们的时钟。

初始化 GPIO

```
    // <InitGPIO>
    if (1) {
        GPIO_InitTypeDef GPIO_InitStructure;

        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
        GPIO_Init(GPIOB, &GPIO_InitStructure);
        }
    // </InitGPIO>
```

在实验板 EC30-EKSTM32-EVAL 上，PB0 管脚连接到电位器 R47 的分压输出。我们通过 PB0 读取电位器的电压，因此必须手动配置 PB0 为模拟量输入模式（GPIO_Mode_AIN）。

初始化 TIM1

```
// <InitTIM1>
if (1) {
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_OCInitTypeDef TIM_OCInitStructure;
    uint32_t TIM_Frequency;

    // Get TIM1 clock frequency
    RCC_ClocksTypeDef RCC_ClocksStatus;
    RCC_GetClocksFreq(&RCC_ClocksStatus);
    TIM_Frequency = RCC_ClocksStatus.PCLK2_Frequency;
    if (TIM_Frequency != RCC_ClocksStatus.HCLK_Frequency)
        TIM_Frequency = TIM_Frequency << 1;

    // Time Base configuration
    TIM_TimeBaseStructInit(&TIM_TimeBaseStructure);
    TIM_TimeBaseStructure.TIM_Period = 100-1;
    TIM_TimeBaseStructure.TIM_Prescaler = TIM_Frequency / (50*32*100);
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);

    // TIM1 channel1 configuration in PWM mode
    TIM_OCStructInit(&TIM_OCInitStructure);
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_Pulse = 50-1;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
    TIM_OC1Init(TIM1, &TIM_OCInitStructure);

    TIM_ITConfig(TIM1, TIM_IT_CC1, ENABLE);
    }
// </InitTIM1>
```

这段代码可以分为 4 个部分。

1. 通过 RCC_GetClocksFreq 函数，得到系统的 APB2CLK 频率。TIM1 使用的时钟源为 APB2CLK 或者为 APB2CLK 的倍频。是否倍频取决于 APB2CLK 是否对 AHBCLK 进行了分频。如果按照 EC30-EKSTM32 的默认配置，APB2CLK 等于 AHBCLK 既 72MHz，因此 TIM1 的输入频率为 72MHz（可在软件 GUTTA Ladder Editor 系统块的 RCC 模块中查看这些时钟频率的配置）。
2. 配置 TIM1 的定时器，这里设置重载寄存器为 100-1=99，分频为 TIM_Frequency / (50*32*100) = 450。既 72MHz 的 APB2CLK 经过 450 分频，调整为 72MHz/450=160KHz。这个频率就是 TIM1 的 CNT 计数频率。由于重载寄存器为 99，故 CNT 总是从 0 计数到

99，然后恢复成 0。CNT 重装的频率为 160KHz/100=1.6KHz。1.6KHz 正是我们期望的 ADC1 采样频率。

3. 配置 TIM1 的比较捕获器 OC1。STM32F103 的每个定时器 TIM 有 4 个独立的比较捕获器。这里我们只使用 OC1，并让其工作在 PWM1 模式。这个比较捕获器不产生真正的输出（TIM_OutputState_Enable）。捕获值我们设置为 50-1=49，这个值没有特殊含义。这个值改变的是 PWM 的占空比，这里不是我们关心的，我们只要求 PWM 的频率是 1.6KHz 即可。只要捕获值在 0 ~ 99 内，OC1 总是能够产生 1.6KHz 的捕获信号，虽然 PWM 的占空比不一样。
4. 使能 OC1 比较捕获中断，进行 A/D 采样的次数统计。正式版本为了不增加 STM32F103 的负载，应该关闭这个统计中断。

初始化 DMA1

```
    // <InitDMA1>
    if (1) {
        DMA_InitTypeDef DMA_InitStructure;

        // DMA1 Channel1 Configuration
        DMA_DeInit(DMA1_Channel1);
        DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)0x4001244C;
        DMA_InitStructure.DMA_MemoryBaseAddr =
(uint32_t)&theADCConvertedValueTab[0];
        DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;
        DMA_InitStructure.DMA_BufferSize = 64;
        DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
        DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
        DMA_InitStructure.DMA_PeripheralDataSize =
DMA_PeripheralDataSize_HalfWord;
        DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
        DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
        DMA_InitStructure.DMA_Priority = DMA_Priority_High;
        DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
        DMA_Init(DMA1_Channel1, &DMA_InitStructure);

        // Enable DMA1 channel1
        DMA_Cmd(DMA1_Channel1, ENABLE);
        }
    // </InitDMA>
```

这里是 DMA 的一个典型配置。

1. DMA1 的输入为 ADC1 的转换结果。
2. DMA1 的输出首地址为 theADCConvertedValueTab 数组，转换结果将保存在这个数组中。
3. DMA1 需要纪录的数据个数为 64 个，数据长度为半字（16 位）。

初始化 ADC1

```
    // <InitADC>
    if (1) {
        ADC_InitTypeDef ADC_InitStructure;

        // ADC1 configuration
        ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
        ADC_InitStructure.ADC_ScanConvMode = DISABLE;
        ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;
        ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T1_CC1;
        ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
        ADC_InitStructure.ADC_NbrOfChannel = 1;
        ADC_DeInit(ADC1);
        ADC_Init(ADC1, &ADC_InitStructure);

        // ADC1 regular channel8 configuration
        ADC_RegularChannelConfig(ADC1, ADC_Channel_8, 1,
ADC_SampleTime_55Cycles5);

        // Enable ADC1 DMA
        ADC_DMACmd(ADC1, ENABLE);

        // Enable ADC1 external trigger
        ADC_ExternalTrigConvCmd(ADC1, ENABLE);

        // Enable ADC1
        ADC_Cmd(ADC1, ENABLE);

        // Enable ADC1 reset calibaration register
        ADC_ResetCalibration(ADC1);
        // Check the end of ADC1 reset calibration register
        while(ADC_GetResetCalibrationStatus(ADC1));

        // Start ADC1 calibaration
        ADC_StartCalibration(ADC1);
        // Check the end of ADC1 calibration
        while(ADC_GetCalibrationStatus(ADC1));
        }
    // </InitADC>
```

这里配置 ADC1 模数转换器。采用 ADC1 的规则转换（相对于注入转换），并且由 TIM1 的比较捕获 OC1 触发转换（ADC_ExternalTrigConv_T1_CC1）。转换数为 1 个即 PB0 对应的 ADC_Chanel_8。转换后触发 DMA 数据传送。

运行 TIM1

```
    // <EnableTIM1Output>
    if (1) {
        // TIM1 counter enable
        TIM_Cmd(TIM1, ENABLE);
        // TIM1 main Output Enable
        TIM_CtrlPWMOutputs(TIM1, ENABLE);
        }
    // </EnableTIM1Output>
```

最后，运行定时器 TIM1，并且使能 PWM 输出。

数据处理

由于 DMA 运行方式为非循环模式（DMA_Mode_Normal），DMA 在获得 64 个采样数据后会停止数据记录。我们可以在 CUSTOM_IO_GET_INPUT 中处理这些采样数据，之后重新使能 DMA 数据记录。

```
void _CUSTOM_IO_GET_INPUT(void) {
    // <CheckDMA>
    if (DMA_GetFlagStatus(DMA1_FLAG_TC1)) {
        DMA_ClearFlag(DMA1_FLAG_TC1);
    // </CheckDMA>

        // <SaveData>
        ++ MD(804);
        if (1) {
            uint32_t lcCnt = 0;
            uint32_t lcSum = 0;
            for (; lcCnt != 64; ++ lcCnt)
                lcSum += (uint32_t)theADCConvertedValueTab[lcCnt];
            AIW(0) = (uint16_t)(lcSum / 64);
            }
        memcpy(&MB(1000), &theADCConvertedValueTab[0], 64*2);
        // </SaveData>

        // <RestartDMA1>
        DMA_Cmd(DMA1_Channel1, DISABLE);
        DMA1_Channel1->CNDTR = 64;
        DMA_Cmd(DMA1_Channel1, ENABLE);
        // </RestartDMA1>
        }
    }
```

CUSTOM_IO_GET_INPUT 消息的处理函数首先通过 DMA_GetFlagStatus 判断 64 个 DMA 数据是否纪录完毕。如果没有完成，等待下一次 CUSTOM_IO_GET_INPUT 消息再做处理，这里直接返回。如果已经完成了，清除完成标志，并且在处理完数据后重新使能 DMA 数据记录。

数据处理首先自加 MD804，统计数据处理的次数。这个统计不是必须的，只是方便我们观察。之后计算 64 个数据的平均值，结果存放在 AIW0 中。同样的为了方便观察，将这 64 个采样数据拷贝到 MW1000 ~ MW1124，这样在 GUTTA Ladder Editor 的状态表中，连线 PLC 后便可以直接观察到这 64 个采样数据。

统计任务

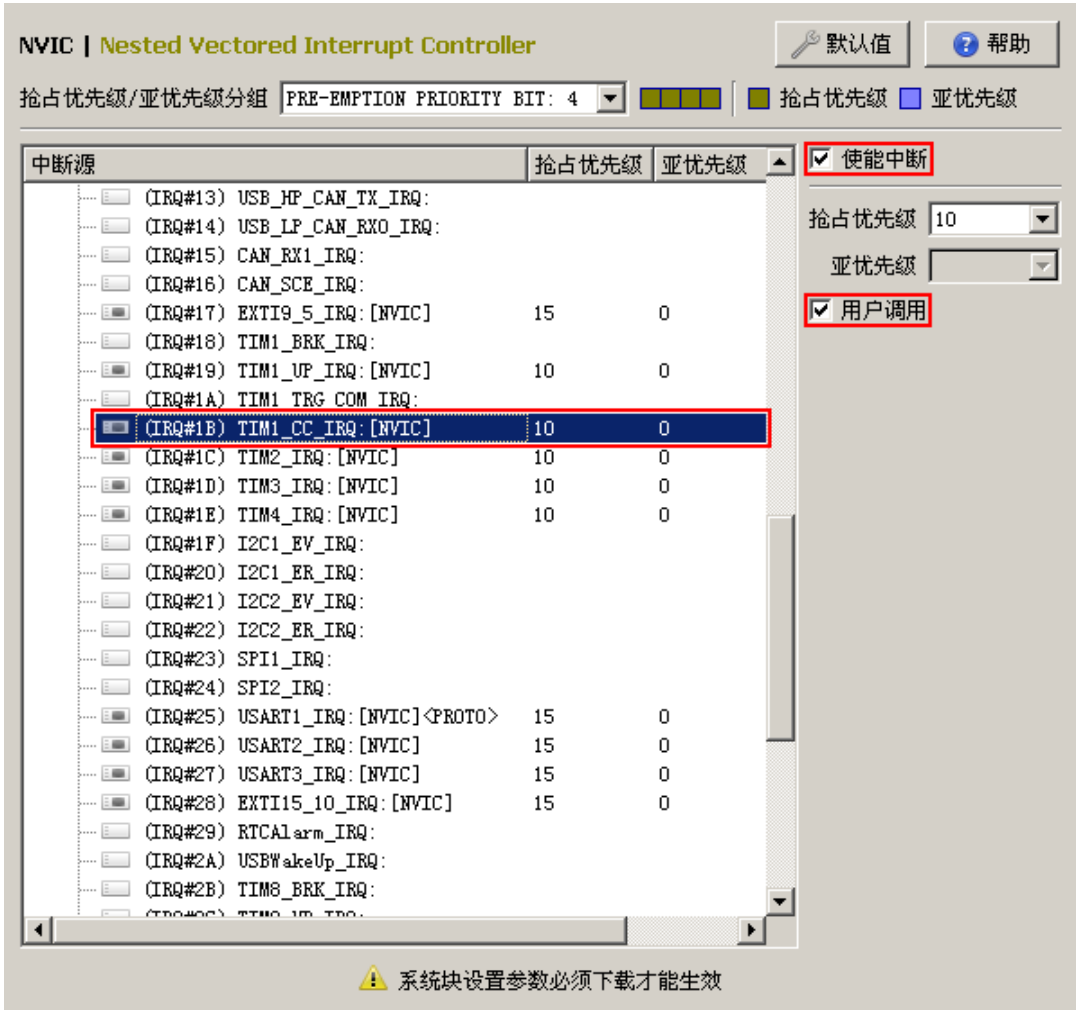
```
void TIM1_CC_IRQHandler(void)
{
    if (TIM_GetITStatus(TIM1, TIM_IT_CC1)) {
        TIM_ClearITPendingBit(TIM1, TIM_IT_CC1);
        ++ MD(800);
        }
}
```

TIM1 比较捕获中断服务程序。如果前面使能了这个中断，这里就必须响应这个中断。我们在这个中断中，自加 MD800，用于统计采样次数。需要注意的是，如果不掉用 TIM_ClearITPendingBit 函数，系统将被这个中断诸塞，造成死机。

测试

下载系统块配置

新建一个 EC30-EKSTM32 项目。由于在二次开发程序中使用了 TIM1 的 CC 中断，故在 GUTTA Ladder Editor 系统块的 NVIC 模块中，需要使能对应的中断 TIM1_CC_IRQ 并使能用户调用。



配置好后，在 GUTTA Ladder Editor 中将这份程序通过串口下载到试验板 EC30-EKSTM32-EVAL。

下载二次开发程序

打开软件 GUTTA Flash Utility。选择二次开发项目生成的程序文件 EC30-EKSTM32.hex，通过串口将这份程序下载到试验板 EC30-EKSTM32-EVAL。

进入连线模式

在软件 GUTTA Ladder Editor 中，进入连线模式，观察变量：

- AIW0　　64 个采样数据的平均值。
- MD800　　采样次数（1600 次每秒）。
- MD804　　DMA 完成次数。
- MW1000 ~ MW1124　　64 个采样数据。

相关下载

模拟量滤波二次开发项目下载：

ProjectForKeil-newlib-AdcSample.zip