EIE3105: ARM Programming – ADC (with DMA)

Dr. Lawrence Cheung

Semester 2, 2021/22

Topics

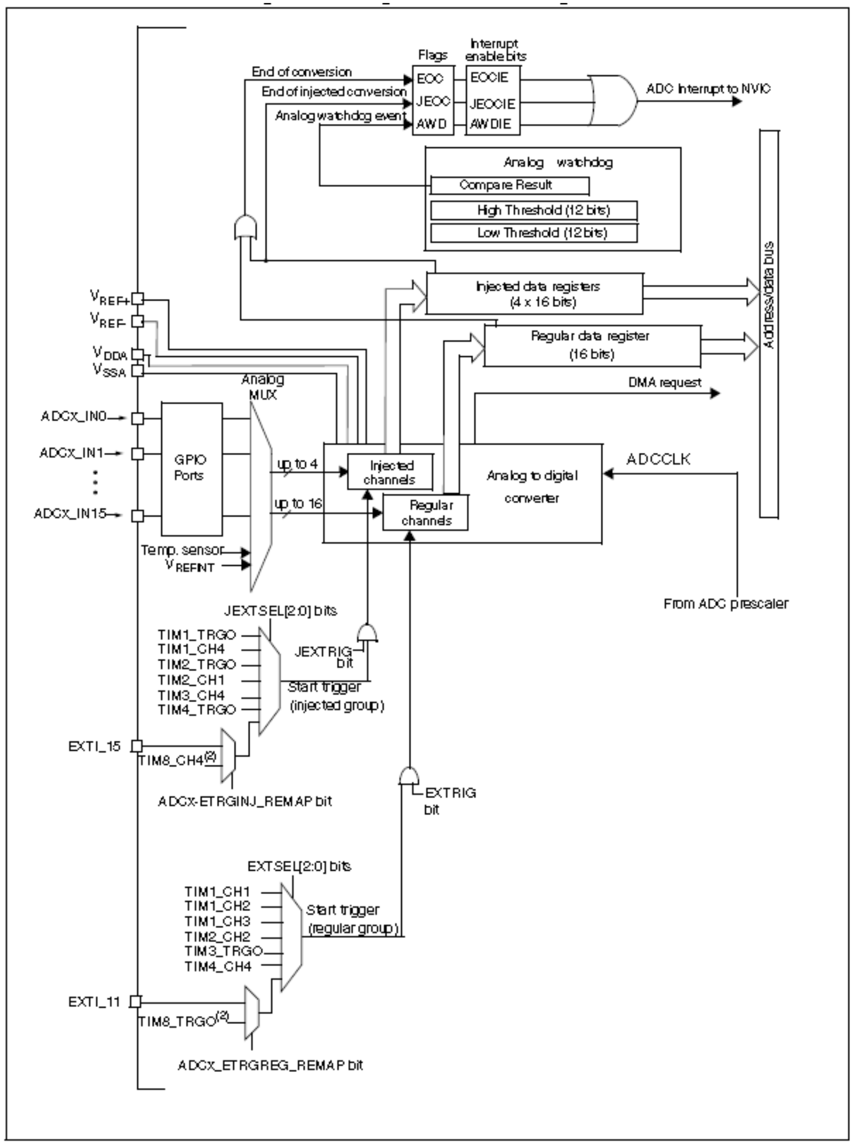
- ADC
- DMA
- ADC with DMA for Multiple Channels

ADC

- Analog-to-Digital Converter
- Function: convert an analog input voltage to a digital value.
- Specification
  - Number of ADCs: 2 (ADC1 and ADC2)
  - Number of channels: 16 (shared by ADC1 and ADC2)
    - One channel for one analog input
  - Number of sources: 16 external and 2 internal sources
    - Internal source: built-in temperature sensor
  - Conversion voltage range: 0 to 3.6 V
  - Conversion time: 1 μs at 56 MHz

ADC

- Block diagram



ADC

- Pin definitions (alternative functions)
  - ADC12: for ADC1 and ADC2

Pin Function	Pin	Pin Function	Pin
ADC12_IN0	PA0	ADC12_IN8	PB0
ADC12_IN1	PA1	ADC12_IN9	PB1
ADC12_IN2	PA2	ADC12_IN10	PC0
ADC12_IN3	PA3	ADC12_IN11	PC1
ADC12_IN4	PA4	ADC12_IN12	PC2
ADC12_IN5	PA5	ADC12_IN13	PC3
ADC12_IN6	PA6	ADC12_IN14	PC4
ADC12_IN7	PA7	ADC12_IN15	PC5

ADC

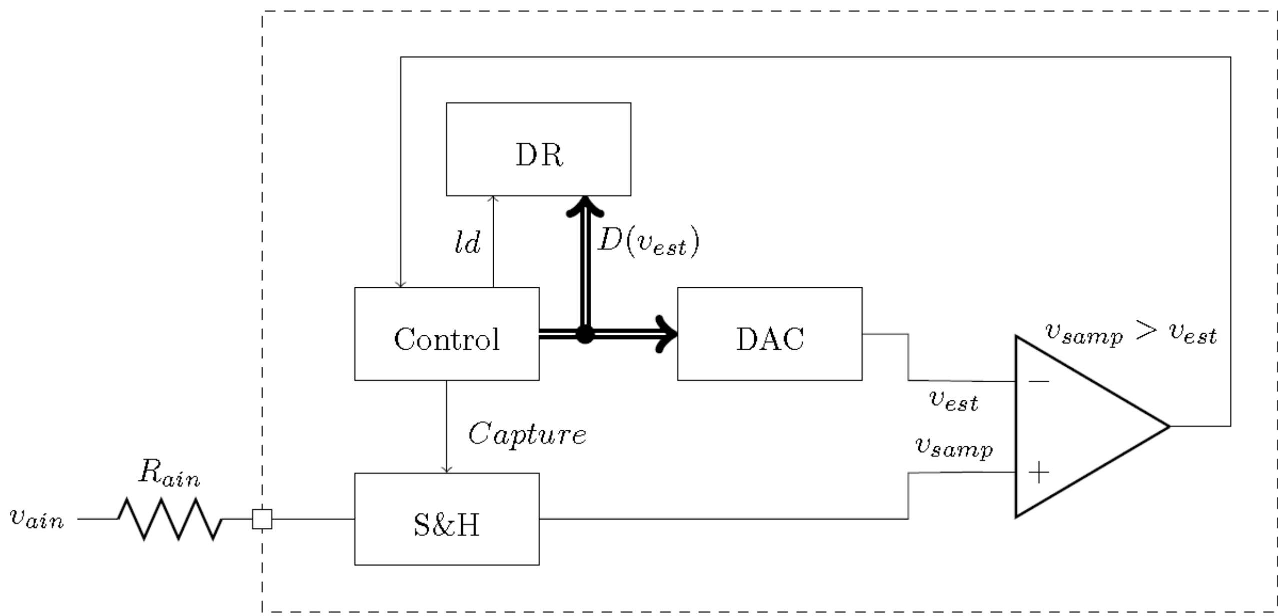
- Two basic modes of operations
  - Single conversion
  - Continuous conversion
- Single conversion: Once the ADC is triggered, it converts a single input and store the result in its data register (DR).
  - The trigger may either come from software or signal (e.g., a timer)
- Continuous conversion: the ADC starts another conversion as soon as it finishes one.

ADC

- Operations of a conversion
  - Capture a sample of input voltage.
  - Generate a sequence of digital approximations.
  - Check each by converting the approximation to an analog signal.
  - Compare it with the input voltage.
  - Store the best approximation into DR.
- For N-bit approximation, the process requires N iterations.

ADC

- For 12 bits of accuracy, it takes at least 14 cycles of the ADC clock (a multiple of the system clock)
  - Two extra cycles for sampling
- Block diagram



ADC

- Example 1:
  - Get an analog input (PA0) from a potentiometer.
  - Show the input voltage through the serial port.
  - Show the input voltage through the brightness of a LED (PA6).
  - Single channel, continuous conversion
- Program Files
  - PinMap.h: initialize pins and functions used in this example.
  - init.c: initialize ADC, PWM and USART2.
  - main.c: main program

ADC

- PinMap.h

```
// Pin Usage
// Function       **  Pin Name ** Board Pin Out
// ADC1 CH0       **  PA0      ** A0
// TIM3 CH1 PWM   **  PA6      ** D12

//ADC1_0 PA0, channel 0, ADC1
#define ADC1_0_RCC_GPIO  RCC_APB2Periph_GPIOA
#define ADC1_0_GPIO      GPIOA
#define ADC1_0_PIN       GPIO_Pin_0

//PWM Tim3 Ch1 PA6, show the brightness of a LED
#define TIM3_CH1_PWM_RCC_GPIO  RCC_APB2Periph_GPIOA
#define TIM3_CH1_PWM_GPIO      GPIOA
#define TIM3_CH1_PWM_PIN       GPIO_Pin_6

void ADC1_1channel_init(void);
void TIM3_PWM_CH1_init(void);
void USART2_init(void);
void USARTSend(char *pucBuffer, unsigned long ulCount);
```

ADC

- init.c

```
#include "stm32f10x.h"                    // Device header
#include "PinMap.h"

void ADC1_1channel_init(void) {
    //PCLK2 is the APB2 clock */
    //ADCCLK = PCLK2/6 = 72/6 = 12MHz*/
    RCC_ADCCLKConfig(RCC_PCLK2_Div6);

    GPIO_InitTypeDef GPIO_InitStructure;
    // Configure I/O for  ADC
    //no need to set default is input floating
    RCC_APB2PeriphClockCmd(ADC1_0_RCC_GPIO, ENABLE);
    GPIO_InitStructure.GPIO_Pin = ADC1_0_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_Init(ADC1_0_GPIO, &GPIO_InitStructure);
```

ADC

```
/* Enable ADC1 clock so that we can talk to it */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
/* Put everything back to power-on defaults */
ADC_DeInit(ADC1);

/* ADC1 Configuration */
ADC_InitTypeDef  ADC_InitStructure;
/* ADC1 and ADC2 operate independently */
ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
/* Disable the scan conversion so we do one at a time */
ADC_InitStructure.ADC_ScanConvMode = DISABLE;
ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
/* Start conversion by software, not an external trigger */
ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
/* 12-bit conversions: put them in the lower 12 bits of the result */
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
/* Say how many channels would be used by the sequencer */
ADC_InitStructure.ADC_NbrOfChannel = 1;
```

ADC

```
    // define regular conversion configuration
    ADC_RegularChannelConfig(ADC1,ADC_Channel_0,1,ADC_SampleTime_239Cycles5);
    /* Now do the setup */
    ADC_Init(ADC1, &ADC_InitStructure);
    /* Enable ADC1 */
    ADC_Cmd(ADC1, ENABLE);

    /* Enable ADC1 reset calibration register */
    ADC_ResetCalibration(ADC1);
    /* Check the end of ADC1 reset calibration register */
    while(ADC_GetResetCalibrationStatus(ADC1));
    /* Start ADC1 calibration */
    ADC_StartCalibration(ADC1);
    /* Check the end of ADC1 calibration */
    while(ADC_GetCalibrationStatus(ADC1));
}
```

ADC

```
void TIM3_PWM_CH1_init(void) {
   RCC_APB2PeriphClockCmd(TIM3_CH1_PWM_RCC_GPIO, ENABLE);
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO,ENABLE);

   GPIO_InitTypeDef GPIO_InitStructure;
   // Configure I/O for Tim3 Ch1 PWM pin
   GPIO_InitStructure.GPIO_Pin = TIM3_CH1_PWM_PIN;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
   GPIO_Init(TIM3_CH1_PWM_GPIO, &GPIO_InitStructure);

   //Tim3 set up
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
```

ADC

```
    TIM_TimeBaseInitTypeDef timerInitStructure;
    timerInitStructure.TIM_Prescaler = 144-1;  //1/(72Mhz/1440)=0.2ms
    timerInitStructure.TIM_CounterMode = TIM_CounterMode_Up;
    timerInitStructure.TIM_Period = 5000-1;
    timerInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;
    timerInitStructure.TIM_RepetitionCounter = 0;
    TIM_TimeBaseInit(TIM3, &timerInitStructure);
    TIM_Cmd(TIM3, ENABLE);

    //Enable Tim3 Ch1 PWM
    TIM_OCInitTypeDef outputChannelInit;
    outputChannelInit.TIM_OCMode = TIM_OCMode_PWM1;
    outputChannelInit.TIM_Pulse = 100-1;
    outputChannelInit.TIM_OutputState = TIM_OutputState_Enable;
    outputChannelInit.TIM_OCPolarity = TIM_OCPolarity_High;
    TIM_OC1Init(TIM3, &outputChannelInit);
    TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Enable);
}
```

ADC

```
void USART2_init(void) {
    //USART2 TX RX
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO, ENABLE);

    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    //USART2 ST-LINK USB
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2,ENABLE);
```

ADC

```
    USART_InitTypeDef USART_InitStructure;
    USART_InitStructure.USART_BaudRate = 9600;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl =
    USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

    USART_Init(USART2, &USART_InitStructure);
    USART_Cmd(USART2, ENABLE);
}
```

ADC

```
void USARTSend(char *pucBuffer, unsigned long ulCount)
{
    // Loop while there are more characters to send.
    while(ulCount--)
    {
      USART_SendData(USART2, *pucBuffer++);
      /* Loop until the end of transmission */
      while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET)
          {
          }
     }
}
```

ADC

```
void RCC_ADCCLKConfig(uint32_t RCC_PCLK2)
```

- Configure the ADC clock (ADCCLK)
  - RCC_PCLK2: the ADC clock divider
    - PCLK2: APB2 clock (72 MHz)
    - RCC_PCLK2_Div2: ADC clock = PCLK2/2 (36 MHz)
    - RCC_PCLK2_Div4: ADC clock = PCLK2/4 (18 MHz)
    - RCC_PCLK2_Div6: ADC clock = PCLK2/6 (12 MHz)
    - RCC_PCLK2_Div8: ADC clock = PCLK2/8 (9 MHz)
  - Possible ADC clock frequency: 0.6 MHz to 14 MHz
    - Thus RCC_PCLK2_Div6 is appropriate.

ADC

```
void ADC_DeInit(ADC_TypeDef* ADCx)
```

- De-initialize the ADCx peripheral registers to their default reset values.
  - ADCx: x = 1, 2 or 3 (3 is not available in STM32F103)

ADC

- ADC initialization

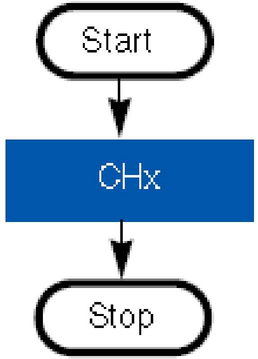
```
typedef struct
{
   uint32_t ADC_Mode;
   FunctionalState ADC_ScanConvMode;
   FunctionalState ADC_ContinuousConvMode;
   uint32_t ADC_ExternalTrigConv;
   uint32_t ADC_DataAlign;
   uint8_t ADC_NbrOfChannel;
} ADC_InitTypeDef;
```

ADC

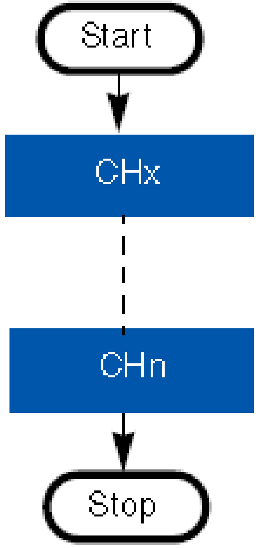
- ADC_Mode = configure the ADC to operate in independent or dual mode.
  - ADC_Mode_Independent
  - ADC_Mode_RegInjecSimult
  - ADC_Mode_RegSimult_AlterTrig
  - ADC_Mode_InjecSimult_FastInter1
  - ADC_Mode_InjecSimult_SlowInter1
  - ADC_Mode_InjecSimult
  - ADC_Mode_RegSimult
  - ADC_Mode_FastInter1
  - ADC_Mode_SlowInter1
  - ADC_Mode_AlterTrig

ADC

- Independent mode (ADC_Mode_Independent)
  - Single channel, single conversion

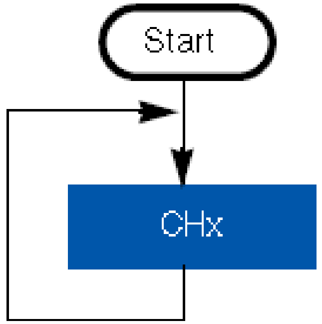


  - Multi-channel, single conversion
    - More than one channel
    - Will be described later

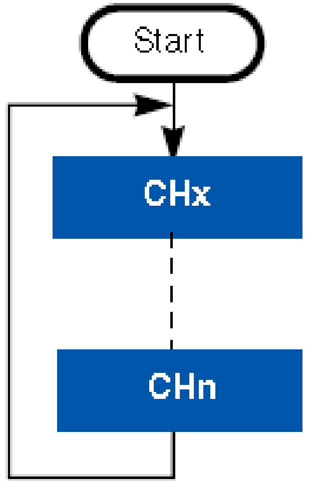


ADC

– Single channel, continuous conversion

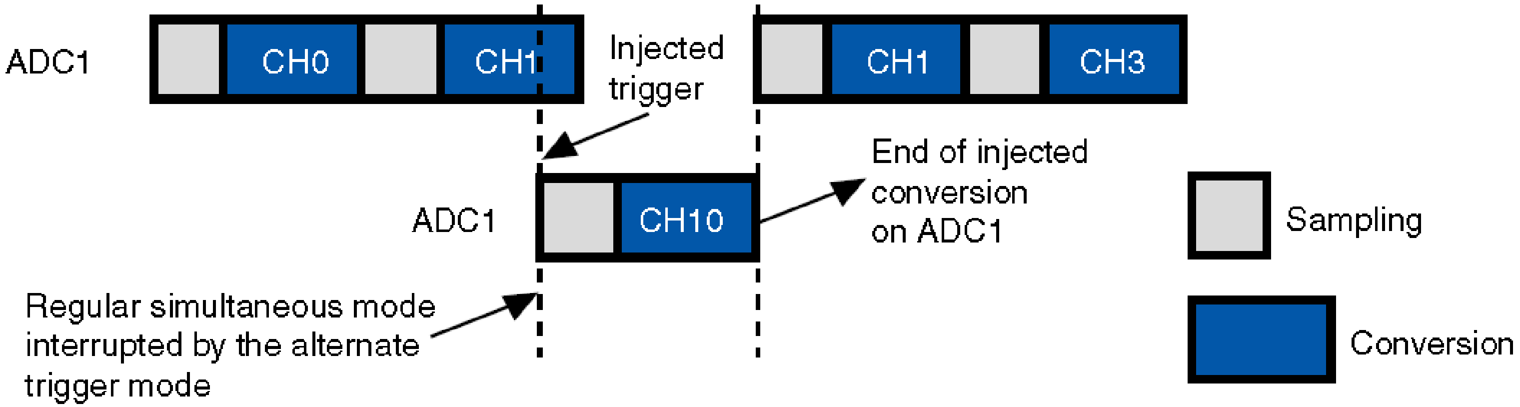


– Multi-channel, continuous conversion



ADC

- Injected conversion mode
  - This mode is used when conversion is triggered by an external event or by software.

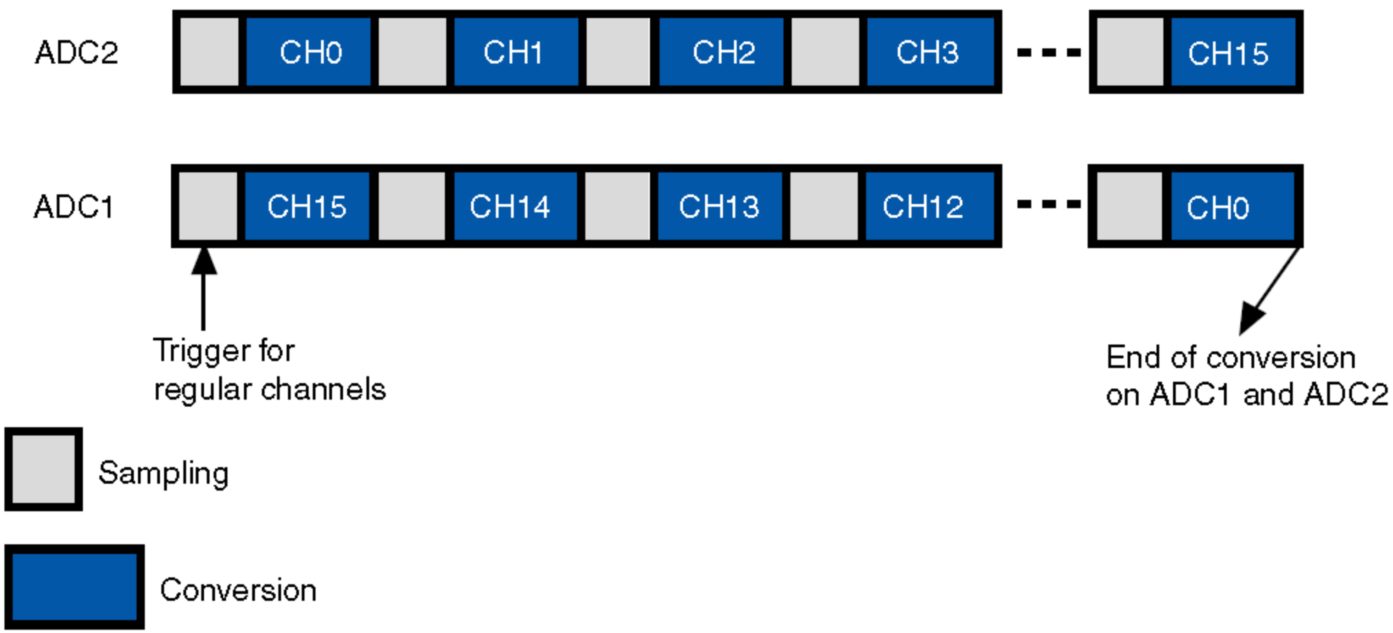


ADC

- Dual mode
  - ADC1: master; ADC2: slave
  - ADC1 and ADC2 triggers are synchronized internally.

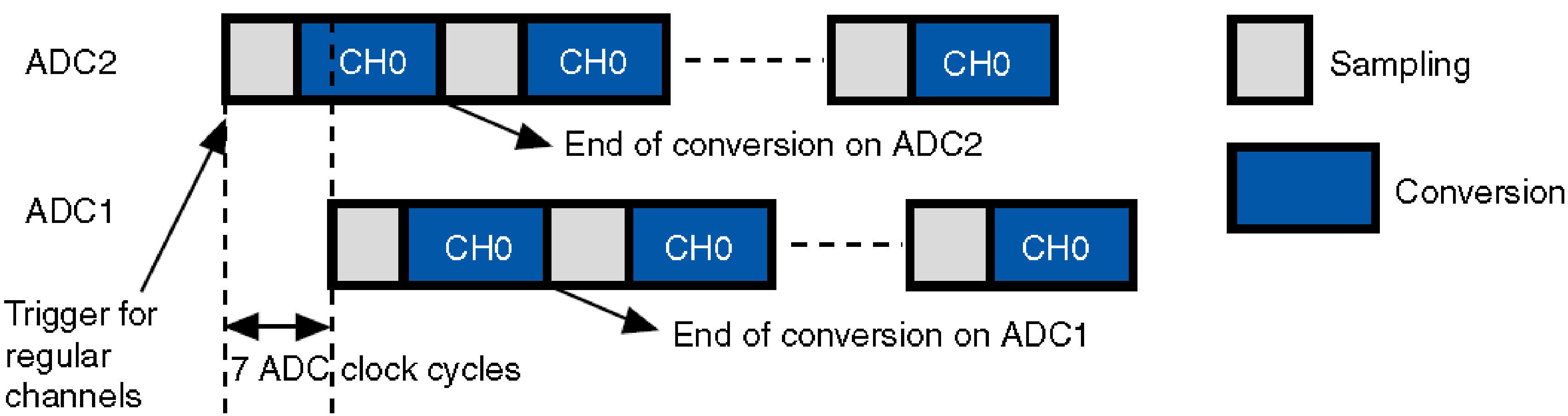
ADC

– Dual regular simultaneous mode (ADC_Mode_RegSimult)
  - Perform two conversions simultaneously.



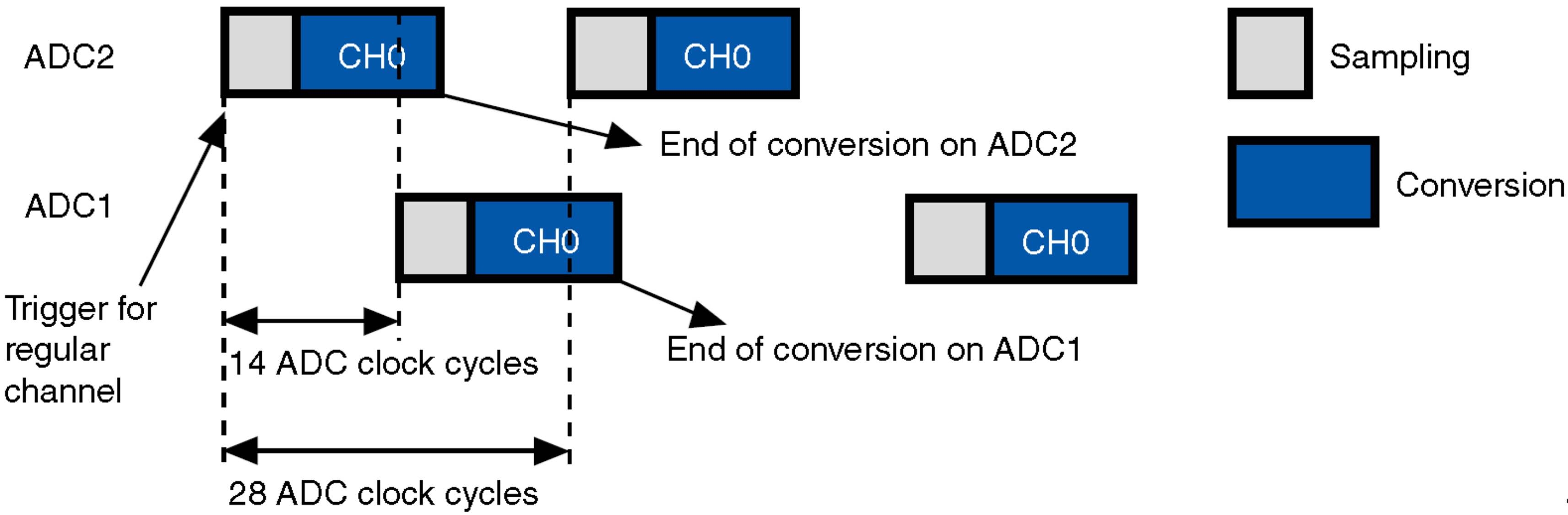
ADC

- Dual fast interleaved mode (ADC_Mode_FastInter1)
  - The conversion of one channel
  - ADC1 and ADC2 convert the selected channel alternatively with a period of 7 ADC clock cycles.
  - Avoid the overlap



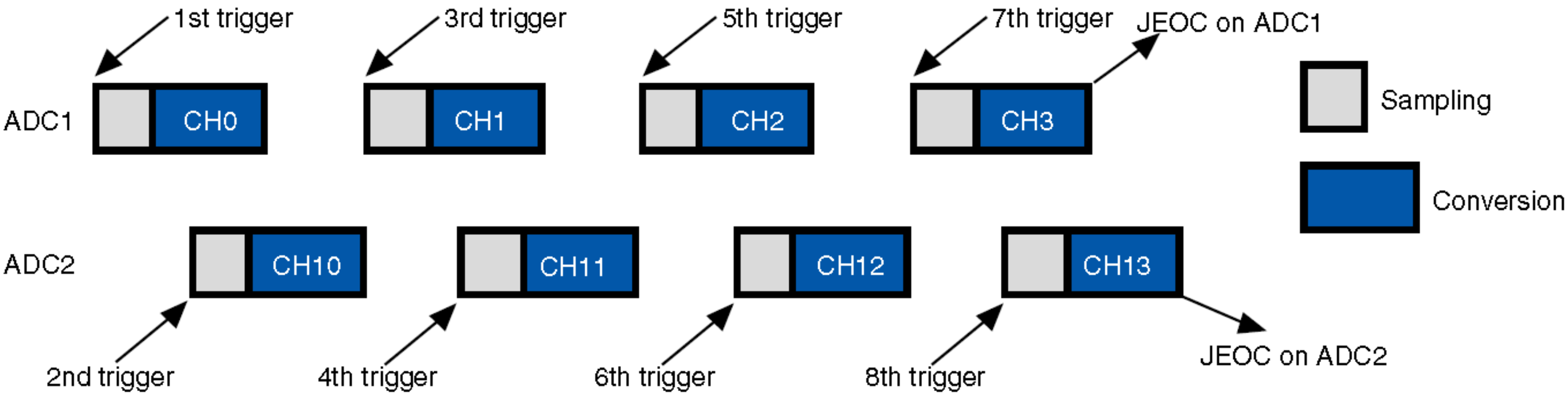
ADC

- Dual slow interleaved mode (ADC_Mode_SlowInter1)
  - Change from 7 to 14 ADC clock cycles.



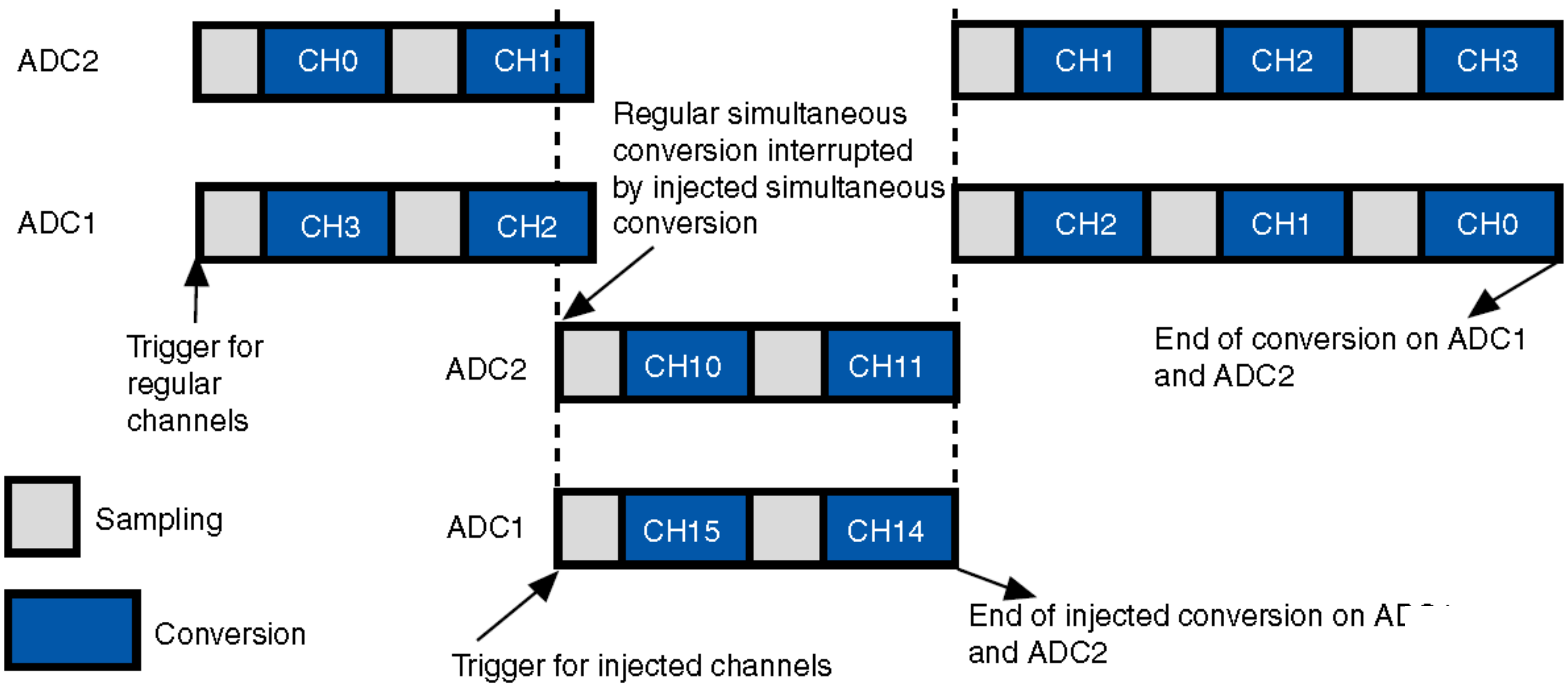
ADC

– Dual alternate trigger mode (ADC_Mode_AlterTrig)
  - ADC1 and ADC2 alternatively convert the injected channels on the same external trigger.



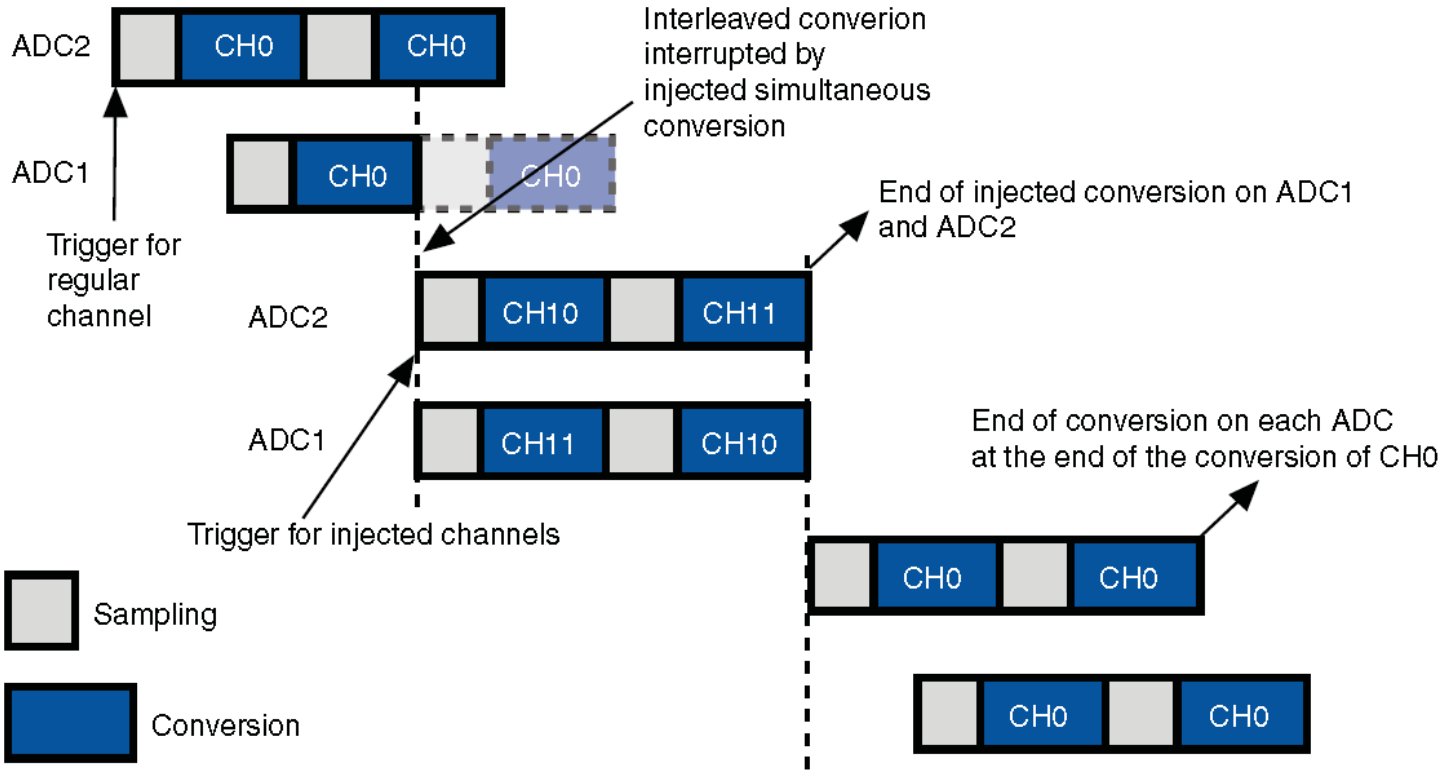
ADC

– Dual combined regular/injected simultaneous mode (ADC_Mode_RegInjecSimult)
  • Regular simultaneous mode that allows injection.



ADC

– Dual combined: injected simultaneous + interleaved mode (ADC_Mode_InjecSimult_FastInter1, ADC_Mode_InjecSimult_SlowInter1)



ADC

- Summary
  - ADC_Mode_Independent = Independent mode
  - ADC_Mode_RegInjecSimult = Combined regular simultaneous + injected simultaneous mode
  - ADC_Mode_RegSimult_AlterTrig = Combined regular simultaneous + alternate trigger mode
  - ADC_Mode_InjecSimult_FastInter1 = Combined injected simultaneous + fast interleaved mode
  - ADC_Mode_InjecSimult_SlowInter1 = Combined injected simultaneous + slow interleaved mode

ADC

- ADC_Mode_InjecSimult = Injected simultaneous mode only
- ADC_Mode_RegSimult = Regular simultaneous mode only
- ADC_Mode_FastInter1 = Fast interleaved mode only
- ADC_Mode_SlowInter1 = Slow interleaved mode only
- ADC_Mode_AlterTrig = Alternate trigger mode only

ADC

- ADC_ScanConvMode = specify whether the conversion is performed in Scan (multi-channels) or Single (one channel) mode.
  - ENABLE: Multi-channels
  - DISABLE: Single channel
- ADC_ContinuousConvMode = specify whether the conversion is performed in Continuous or Single mode.
  - ENABLE: Continuous
  - DISABLE: Single

ADC

- ADC_ExternalTrigConv = define the external trigger used to start the analog to digital conversion of regular channels.
  - ADC_ExternalTrigConv_T1_CC1 = Timer 1 CC1 event (CC1 = Capture/Compare Register 1)
  - ADC_ExternalTrigConv_T1_CC2 = Timer 1 CC2 event
  - ADC_ExternalTrigConv_T2_CC2 = Timer 2 CC2 event
  - ADC_ExternalTrigConv_T3_TRGO = Timer 3 TRGO (TRGO = Information to be sent in master mode to slave timers for synchronization)

ADC

- ADC_ExternalTrigConv_T4_CC4 = Timer 4 CC4 event
- ADC_ExternalTrigConv_T1_CC3 = Timer 1 CC3 event
- ADC_ExternalTrigConv_None = SWSTART (start conversion of regular channels, software control bit)
- Some modes are ignored because they cannot be used in STM32F103.

ADC

- ADC_DataAlign = specify whether the ADC data alignment is aligned to left or right (12-bit data into a 16-bit register).
  - ADC_DataAlign_Left
  - ADC_DataAlign_Right
- ADC_NbrOfChannel = specify the number of ADC channels that will be converted using the sequencer for the regular channel group.
  - Value: 1 to 16

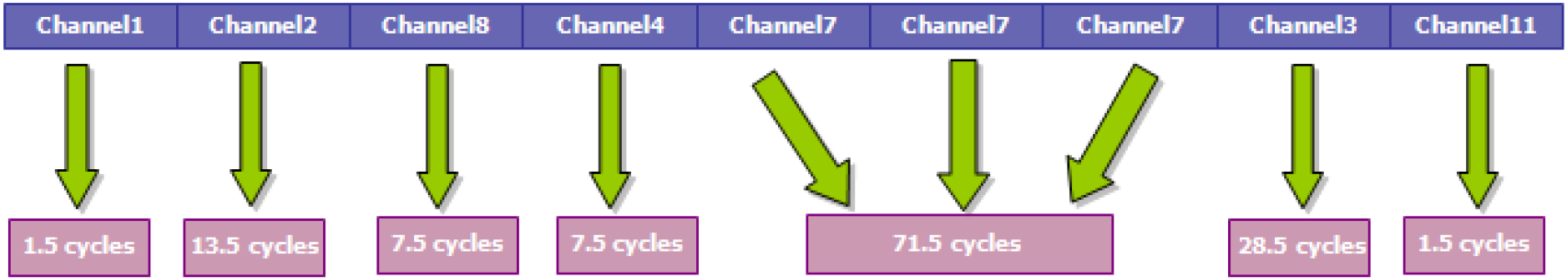
ADC

```
void ADC_RegularChannelConfig(ADC_TypeDef* ADCx,
uint8_t ADC_Channel, uint8_t Rank, uint8_t
ADC_SampleTime);
```

- Configure for the selected ADC regular channel its corresponding rank in the sequencer and its sample time.
  - ADCx: x = 1, 2 or 3 (3 is not available in STM32F103)
  - ADC_Channel: ADC_Channel_0 to ADC_Channel_17 (ADC Channel 0 to ADC Channel 17)
  - Rank: 1 to 16

ADC

- ADC_SampleTime
  - ADC_SampleTime_1Cycles5: Sample Time = 1.5 cycles
  - ADC_SampleTime_7Cycles5: Sample Time = 7.5 cycles
  - ADC_SampleTime_13Cycles5: Sample Time = 13.5 cycles
  - ADC_SampleTime_28Cycles5: Sample Time = 28.5 cycles
  - ADC_SampleTime_41ycles5: Sample Time = 41.5 cycles
  - ADC_SampleTime_55Cycles5: Sample Time = 55.5 cycles
  - ADC_SampleTime_71Cycles5: Sample Time = 71.5 cycles
  - ADC_SampleTime_239Cycles5: Sample Time = 239.5 cycles



ADC

```
void ADC_Init(ADC_TypeDef* ADCx, ADC_InitTypeDef*
ADC_InitStruct)
```

- Initialize the ADCx peripheral according to the specified parameters in the ADC_InitStruct.
  - ADCx: x = 1, 2 or 3 (3 is not available in STM32F103)

```
void ADC_Cmd(ADC_TypeDef* ADCx, FunctionState
NewState)
```

- Enable or disable the specified ADC peripheral.
  - ADCx: x = 1, 2 or 3 (3 is not available in STM32F103)
  - NewState: ENABLE or DISABLE

ADC

- After STM32F103 is powered on, it is recommended to run ADC self-calibration. This calculates error collection codes for capacitors and reduces overall error in the result.

```
void ADC_ResetCalibration(ADC_TypeDef* ADCx)
```

  – Reset the selected ADC calibration register.
    - ADCx: x = 1, 2 or 3 (3 is not available in STM32F103)

ADC

```
FlagStatus ADC_GetResetCalibrationStatus(ADC_TypeDef*
ADCx)
```

- Get the status of the selected ADC reset calibration register.
    - ADCx: x = 1, 2 or 3 (3 is not available in STM32F103)
    - Return value: SET or RESET

```
void ADC_StartCalibration (ADC_TypeDef* ADCx)
```

- Start the selected ADC calibration process.
    - ADCx: x = 1, 2 or 3 (3 is not available in STM32F103)

ADC

```
FlagStatus ADC_GetCalibrationStatus(ADC_TypeDef*
ADCx)
```

- Get the status of the selected ADC calibration.
  - ADCx: x = 1, 2 or 3 (3 is not available in STM32F103)
  - Return value: SET or RESET

ADC

- main.c

```
#include "stm32f10x.h"                    // Device header
#include "PinMap.h"
#include "stdio.h"
#include "misc.h"

int main(void) {

   char buffer[50] = {'\0'};
   int adc_value;

   USART2_init();
   ADC1_1channel_init();
   TIM3_PWM_CH1_init();

   // start conversion (will be endless as we are in continuous mode)
   ADC_SoftwareStartConvCmd(ADC1, ENABLE);
```

ADC

```
while(1) {
  while( ADC_GetFlagStatus( ADC1, ADC_FLAG_EOC ) == RESET )
  {
  }
  adc_value = ADC_GetConversionValue(ADC1);
  TIM_SetCompare1(TIM3, adc_value);
  sprintf(buffer, "%d\r\n", adc_value);
  USARTSend(buffer, sizeof(buffer));
}
```

ADC

```
void ADC_SoftwareStartConvCmd(ADC_TypeDef* ADCx,
FunctionState NewState)
```

- Enable or disable the selected ADC software start conversion.
  - ADCx: x = 1, 2 or 3 (3 is not available in STM32F103)
  - NewState: SET or RESET

ADC

```
FlagStatus ADC_GetFlagStatus(ADC_TypeDef* ADCx,
uint8_t ADC_FLAG)
```

– Check whether the specified ADC flag is set or not.
  - ADCx: x = 1, 2 or 3 (3 is not available in STM32F103)
  - ADC_FLAG: specify the flag to check
    – ADC_FLAG_AWD: Analog watchdog flag
    – ADC_FLAG_EOC: End of conversion flag
    – ADC_FLAG_JEOC: End of injected group conversion flag
    – ADC_FLAG_JSTRT: Start of injected group conversion flag
    – ADC_FLAG_STRT: Start of regular group conversion flag

ADC

```
uint16_t ADC_GetConversionValue(ADC_TypeDef* ADCx)
```

– Return the last ADCx conversion result data for regular channel.
    - ADCx: x = 1, 2 or 3 (3 is not available in STM32F103)

DMA

- Consider the following program codes to read data from a peripheral by repeatedly waiting for a status flag:

```
for (i = 0; i < N; i++) {
        while(flagBusy);
        buf[i] = peripheralRegister;
}
```

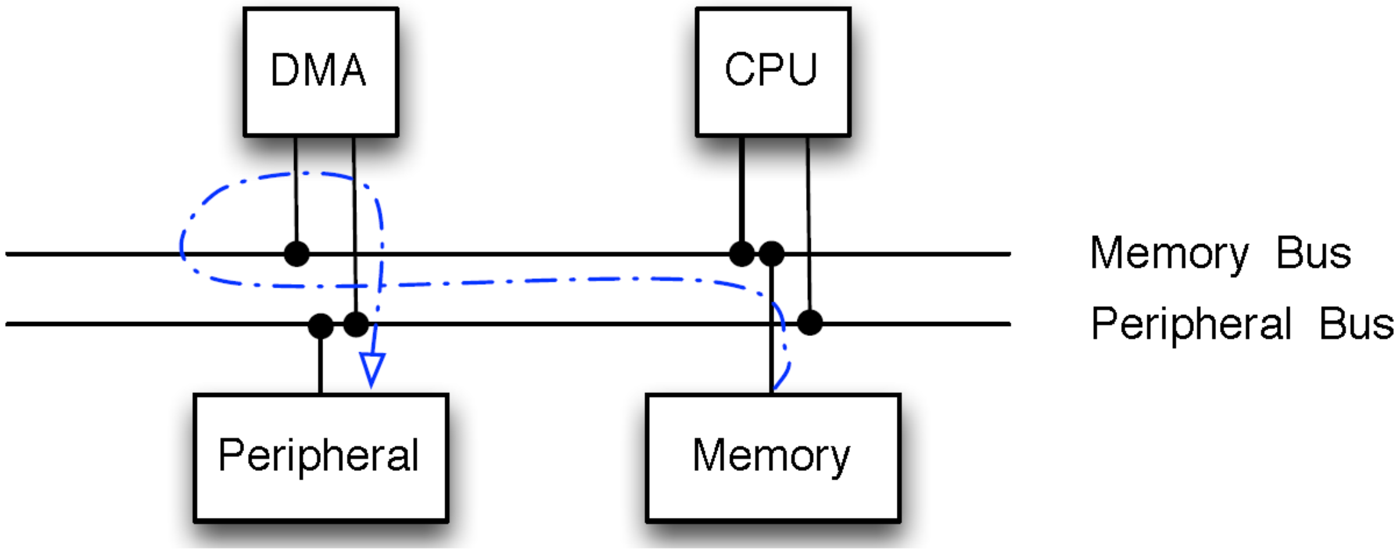
  – This approach is called software polling.

DMA

- It has three limitations:
  1. The processor is tied up during the transfer and cannot perform other tasks.
  2. The actual transfer rate is lower than the maximum one.
  3. It is difficult to achieve tight timing bounds. For example, audio streaming depends up on the data samples to be transferred at a constant rate.

DMA

- Direct Memory Access (DMA) can solve this problem.
- DMA is implemented in processors with dedicated hardware devices.
- These devices share the memory bus and peripheral buses with CPU.



DMA

- This architecture guarantees that the CPU will not be starved.
- Only a fraction of STM32 instructions directly access RAM memory, the rest simply pull instructions from FLASH which uses a different bus.
- DMA1 channel 1 supports ADC1.

ADC with DMA for Multiple Channels

- Example 2:
  - Get three analog inputs (PA0, PA1 and PA4) from three potentiometers.
  - Show the input voltage through the serial port.
  - Show the input voltage through the brightness of three corresponding LEDs (PA6, PA7 and PB0).
  - Multi-channels, continuous conversion
- Program Files
  - PinMap.h: initialize pins and functions.
  - init.c: initialize ADC, DMA, PWM and USART2.
  - main.c: main program

ADC with DMA for Multiple Channels

- PinMap.h

```
#ifndef _PINMAP_H
#define _PINMAP_H
// Pin Usage
// Function      **  Pin Name ** Board Pin Out
// ADC1_IN0      **  PA0      ** A0
// ADC1_IN1      **  PA1      ** A1
// ADC1_IN4      **  PA4      ** A2
// TIM3 CH1 PWM  **  PA6      ** D12
// TIM3 CH2 PWM  **  PA7      ** D11
// TIM3 CH3 PWM  **  PB0      ** A3

// ADC1_IN0      **  PA0      ** A0
#define ADC1_0_RCC_GPIO  RCC_APB2Periph_GPIOA
#define ADC1_0_GPIO      GPIOA
#define ADC1_0_PIN       GPIO_Pin_0
```

ADC with DMA for Multiple Channels

```
// ADC1_IN1      **  PA1      ** A1
#define ADC1_1_RCC_GPIO  RCC_APB2Periph_GPIOA
#define ADC1_1_GPIO      GPIOA
#define ADC1_1_PIN       GPIO_Pin_1

// ADC1_IN4      **  PA4      ** A2
#define ADC1_4_RCC_GPIO  RCC_APB2Periph_GPIOA
#define ADC1_4_GPIO      GPIOA
#define ADC1_4_PIN       GPIO_Pin_4

#define ADC1_0_1_4_GPIO  GPIOA

// TIM3 CH1 PWM  **  PA6      ** D12
#define TIM3_CH1_PWM_RCC_GPIO  RCC_APB2Periph_GPIOA
#define TIM3_CH1_PWM_GPIO      GPIOA
#define TIM3_CH1_PWM_PIN       GPIO_Pin_6
```

ADC with DMA for Multiple Channels

```
// TIM3 CH2 PWM  **  PA7      ** D11
#define TIM3_CH2_PWM_RCC_GPIO  RCC_APB2Periph_GPIOA
#define TIM3_CH2_PWM_GPIO      GPIOA
#define TIM3_CH2_PWM_PIN       GPIO_Pin_7

// TIM3 CH3 PWM  **  PB0      ** A3
#define TIM3_CH3_PWM_RCC_GPIO  RCC_APB2Periph_GPIOB
#define TIM3_CH3_PWM_GPIO      GPIOB
#define TIM3_CH3_PWM_PIN       GPIO_Pin_0

#define ARRAYSIZE 3
#define ADC1_DR    ((uint32_t)0x4001244C)

//Function prototypes
void ADC1_1channel_init(void);
void ADC1_3channels_init(void);
void DMA1_init(void);
void TIM3_PWM_CH1_init(void);
void USART2_init(void);
void USARTSend(char *pucBuffer, unsigned long ulCount);
#endif
```

ADC with DMA for Multiple Channels

- init.c

```
#include "stm32f10x.h"                  // Device header
#include "PinMap.h"
volatile uint16_t ADC_values[ARRAYSIZE];

void ADC1_3channels_init(void) {
   ADC_InitTypeDef  ADC_InitStructure;
   //PCLK2 is the APB2 clock */
   //ADCCLK = PCLK2/6 = 72/6 = 12MHz*/
   RCC_ADCCLKConfig(RCC_PCLK2_Div6);

   GPIO_InitTypeDef GPIO_InitStructure;
   // Configure I/O for  ADC, no need to set, default is input floating
   RCC_APB2PeriphClockCmd(ADC1_1_RCC_GPIO, ENABLE);
   GPIO_InitStructure.GPIO_Pin = ADC1_0_PIN | ADC1_1_PIN | ADC1_4_PIN;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
   GPIO_Init(ADC1_0_1_4_GPIO , &GPIO_InitStructure);
```

ADC with DMA for Multiple Channels

```
/* Enable ADC1 clock so that we can talk to it */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
/* Put everything back to power-on defaults */
ADC_DeInit(ADC1);

/* ADC1 Configuration */
/* ADC1 and ADC2 operate independently */
ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
/* Enable the scan conversion to convert multiple channels */
ADC_InitStructure.ADC_ScanConvMode = ENABLE;
/* Continuous conversions */
ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
/* Start conversion by software, not an external trigger */
ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
/* 12-bit conversions: put them in the lower 12 bits of the result */
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
/* Say how many channels would be used by the sequencer */
ADC_InitStructure.ADC_NbrOfChannel = 3;
```

ADC with DMA for Multiple Channels

```
    // define regular conversion configurations
    ADC_RegularChannelConfig(ADC1,ADC_Channel_0,1,ADC_SampleTime_239Cycles5);
    ADC_RegularChannelConfig(ADC1,ADC_Channel_1,2,ADC_SampleTime_239Cycles5);
    ADC_RegularChannelConfig(ADC1,ADC_Channel_4,3,ADC_SampleTime_239Cycles5);
    /* Now do the setup */
    ADC_Init(ADC1, &ADC_InitStructure);
    /* Enable ADC1 */
    ADC_Cmd(ADC1, ENABLE);

    //enable DMA for ADC
    ADC_DMACmd(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));
}
```

ADC with DMA for Multiple Channels

```
void DMA1_init(void) {
    //enable DMA1 clock
   RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
   //create DMA structure
   DMA_InitTypeDef  DMA_InitStructure;
   //reset DMA1 channe1 to default values;
   DMA_DeInit(DMA1_Channel1);
   //channel will not be used for memory to memory transfer
   DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
   //setting circular mode
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
   //medium priority
   DMA_InitStructure.DMA_Priority = DMA_Priority_High;
   //source and destination data size word=32bit
   DMA_InitStructure.DMA_PeripheralDataSize=DMA_PeripheralDataSize_HalfWord;
   DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
   //automatic memory destination increment enable.
   DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
   //source address increment disable
   DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
```

ADC with DMA for Multiple Channels

```
 //Location assigned to peripheral register will be source
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;
//chunk of data to be transfered
DMA_InitStructure.DMA_BufferSize = ARRAYSIZE;
//source and destination start addresses
DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&ADC1->DR;
 //(uint32_t)ADC1_DR;
DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)ADC_values;
//send values to DMA registers
DMA_Init(DMA1_Channel1, &DMA_InitStructure);
// Enable DMA1 Channel Transfer Complete interrupt
DMA_ITConfig(DMA1_Channel1, DMA_IT_TC, ENABLE);
DMA_Cmd(DMA1_Channel1, ENABLE); //Enable the DMA1 - Channel1
NVIC_InitTypeDef NVIC_InitStructure;
//Enable DMA1 channel IRQ Channel */
NVIC_InitStructure.NVIC_IRQChannel = DMA1_Channel1_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
}
```

ADC with DMA for Multiple Channels

```
void TIM3_PWM_CH1_init(void) {
   RCC_APB2PeriphClockCmd(TIM3_CH1_PWM_RCC_GPIO, ENABLE);
   RCC_APB2PeriphClockCmd(TIM3_CH2_PWM_RCC_GPIO, ENABLE);
   RCC_APB2PeriphClockCmd(TIM3_CH3_PWM_RCC_GPIO, ENABLE);
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO,ENABLE);

   GPIO_InitTypeDef GPIO_InitStructure;
   // Configure I/O for Tim3 Ch1 PWM pin
   GPIO_InitStructure.GPIO_Pin = TIM3_CH1_PWM_PIN;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
   GPIO_Init(TIM3_CH1_PWM_GPIO, &GPIO_InitStructure);

   // Configure I/O for Tim3 Ch2 PWM pin
   GPIO_InitStructure.GPIO_Pin = TIM3_CH2_PWM_PIN;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
   GPIO_Init(TIM3_CH2_PWM_GPIO, &GPIO_InitStructure);
```

ADC with DMA for Multiple Channels

```
// Configure I/O for Tim3 Ch3 PWM pin
GPIO_InitStructure.GPIO_Pin = TIM3_CH3_PWM_PIN;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init(TIM3_CH3_PWM_GPIO, &GPIO_InitStructure);

//Tim3 set up
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

TIM_TimeBaseInitTypeDef timerInitStructure;
timerInitStructure.TIM_Prescaler = 144-1;  //1/(72Mhz/1440)=0.2ms
timerInitStructure.TIM_CounterMode = TIM_CounterMode_Up;
timerInitStructure.TIM_Period = 5000-1;
timerInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;
timerInitStructure.TIM_RepetitionCounter = 0;
TIM_TimeBaseInit(TIM3, &timerInitStructure);
TIM_Cmd(TIM3, ENABLE);
```

ADC with DMA for Multiple Channels

```
TIM_OCInitTypeDef outputChannelInit;
//Enable Tim3 Ch1 PWM
outputChannelInit.TIM_OCMode = TIM_OCMode_PWM1;
outputChannelInit.TIM_Pulse = 1-1;
outputChannelInit.TIM_OutputState = TIM_OutputState_Enable;
outputChannelInit.TIM_OCPolarity = TIM_OCPolarity_High;
TIM_OC1Init(TIM3, &outputChannelInit);
TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Enable);

//Enable Tim3 Ch2 PWM
outputChannelInit.TIM_OCMode = TIM_OCMode_PWM1;
outputChannelInit.TIM_Pulse = 1-1;
outputChannelInit.TIM_OutputState = TIM_OutputState_Enable;
outputChannelInit.TIM_OCPolarity = TIM_OCPolarity_High;
TIM_OC2Init(TIM3, &outputChannelInit);
TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Enable);
```

ADC with DMA for Multiple Channels

```
    //Enable Tim3 Ch1 PWM
    outputChannelInit.TIM_OCMode = TIM_OCMode_PWM1;
    outputChannelInit.TIM_Pulse = 1000-1;
    outputChannelInit.TIM_OutputState = TIM_OutputState_Enable;
    outputChannelInit.TIM_OCPolarity = TIM_OCPolarity_High;
    TIM_OC3Init(TIM3, &outputChannelInit);
    TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Enable);
}

void USART2_init(void) {
    //USART2 TX RX
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO,
ENABLE);

    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
```

ADC with DMA for Multiple Channels

```
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    //USART2 ST-LINK USB
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2,ENABLE);

    USART_InitTypeDef USART_InitStructure;

    USART_InitStructure.USART_BaudRate = 9600;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl =
    USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

    USART_Init(USART2, &USART_InitStructure);
    USART_Cmd(USART2, ENABLE);
}
```

ADC with DMA for Multiple Channels

```
void USARTSend(char *pucBuffer, unsigned long ulCount)
{
    //
    // Loop while there are more characters to send.
    //
    while(ulCount--)
    {
        USART_SendData(USART2, *pucBuffer++);
        /* Loop until the end of transmission */
        while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET)
        {
        }
    }
}
```

ADC with DMA for Multiple Channels

```
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
```

- Enable the AHB peripheral clock for DMA1.

```
void DMA_DeInit(DMA_Channel_TypeDef* DMAy_Channelx)
```

  - De-initialize the DMAy Channelx registers to their default reset values.
    - DMAy: y = 1 or 2
    - Channelx: 1 to 7 for DMA1 and 1 to 5 for DMA2

ADC with DMA for Multiple Channels

- DMA initialization

```
typedef struct {
   uint32_t DMA_PeripheralBaseAddr;
   uint32_t DMA_MemoryBaseAddr;
   uint32_t DMA_DIR;
   uint32_t DMA_BufferSize;
   uint32_t DMA_PeripheralInc;
   uint32_t DMA_MemoryInc;
   uint32_t DMA_PeripheralDataSize;
   uint32_t DMA_MemoryDataSize;
   uint32_t DMA_Mode;
   uint32_t DMA_Priority;
   uint32_t DMA_M2M;
} DMA_InitTypeDef;
```

ADC with DMA for Multiple Channels

- DMA_PeripheralBaseAddr = specify the peripheral base address for DMAy Channelx.
  - Two possible choices only: &ADC1->DR, &ADC2->DR
- DMA_MemoryBaseAddr = specify the memory base address for DMAy Channelx.
  - Use to store your data
  - You should create an uint16_t array to store your data
  - In this example, it is ADC_values.

ADC with DMA for Multiple Channels

- DMA_DIR = specify if the peripheral is the source or destination.
  - DMA_DIR_PeripheralSRC: Data is transferred from the peripheral to the memory.
  - DMA_DIR_PeripheralDST: Data is transferred from the the memory to the peripheral.
- DMA_BufferSize = specify the buffer size, in data unit, of the specified channel.
  - In this example, it is ARRAYSIZE.

ADC with DMA for Multiple Channels

- DMA_PeripheralInc = specify whether the peripheral address register is incremented or not.
  - DMA_PeripheralInc_Enable
  - DMA_PeripheralInc_Disable
  - In this example, it is DMA_PeripheralInc_Disable because ADC1->DR is unchanged but the channel number is changed.
- DMA_MemoryInc = specify whether the memory address register is incremented or not.
  - DMA_MemoryInc_Enable, DMA_MemoryInc_Disable
  - In this example, it is DMA_MemoryInc_Enable because the memory address must be incremented to fill in the array ADC_values.

ADC with DMA for Multiple Channels

- DMA_PeripheralDataSize = specify the peripheral data width.
  - DMA_PeripheralDataSize_Byte
  - DMA_PeripheralDataSize_HalfWord
  - DMA_PeripheralDataSize_Word
  - In this example, it is DMA_PeripheralDataSize_HalfWord for 12-bit conversion.

ADC with DMA for Multiple Channels

- DMA_MemoryDataSize = specify the memory data width.
  - DMA_MemoryDataSize_Byte
  - DMA_MemoryDataSize_HalfWord
  - DMA_MemoryDataSize_Word
  - In this example, it is DMA_MemoryDataSize_HalfWord for 12-bit conversion.

ADC with DMA for Multiple Channels

– DMA_mode = specify the operation mode of the DMAy Channelx.
  - DMA_Mode_Circular: After the last transfer, the data register is automatically reloaded with the initially programmed value.
  - DMA_Mode_Normal: no DMA request is served after the last transfer.
  - In this example, it is DMA_Mode_Circular because the conversion is processed repeatedly.

ADC with DMA for Multiple Channels

- DMA_priority = specify the software priority for the DMAy Channelx.
  - DMA_Priority_VeryHigh
  - DMA_Priority_High
  - DMA_Priority_Medium
  - DMA_Priority_Low
  - In this example, it is DMA_Priority_High.

ADC with DMA for Multiple Channels

- DMA_M2M = specify if the DMAy Channelx will be used in memory-to-memory transfer.
  - DMA_M2M_Enable
  - DMA_M2M_Disable
  - In this example, it is DMA_M2M_Disable because it is from the peripheral to the memory.

ADC with DMA for Multiple Channels

```
void DMA_Init(DMA_Channel_TypeDef* DMAy_Channelx,
DMA_InitTypeDef* DMA_InitStruct)
```

- Initialize the DMAy Channelx according to the specified parameters in the DMA_InitStruct.
  - DMAy: y = 1 or 2
  - Channelx: 1 to 7 for DMA1 and 1 to 5 for DMA2

ADC with DMA for Multiple Channels

```
void DMA_ITConfig(DMA_Channel_TypeDef* DMAy_Channelx,
uint32_t DMA_IT, FunctionState NewState)
```

- Enable or disable the specified DMAy Channelx interrupts.
- DMA_IT: specify the DMA interrupts sources to be enabled or enabled.
  - DMA_IT_TC: Transfer complete interrupt mask
  - DMA_IT_HT: Half transfer interrupt mask
  - DMA_IT_TE: Transfer error interrupt mask
  - In this example, it is DMA_IT_TC.
- NewState: Enable or disable

ADC with DMA for Multiple Channels

```
void DMA_Cmd(DMA_Channel_TypeDef* DMAy_Channelx,
FunctionState NewState)
```

- Enable or disable the specified DMAy Channelx.
- NewState: Enable or disable

ADC with DMA for Multiple Channels

- main.c

```
#include "stm32f10x.h"                    // Device header
#include "PinMap.h"
#include "stdio.h"
#include "misc.h"

volatile uint32_t status = 0;
extern volatile uint16_t ADC_values[ARRAYSIZE];

int main(void) {

   char buffer[50] = {'\0'};

   USART2_init();
   ADC1_3channels_init();
   TIM3_PWM_CH1_init();
   DMA1_init();
```

ADC with DMA for Multiple Channels

```
    // start conversion (will be endless as we are in continuous mode)
    ADC_SoftwareStartConvCmd(ADC1, ENABLE);

    while(1) {
      while(!status);
      sprintf(buffer, "ch0=%d ch1=%d ch4=%d\r\n", ADC_values[0],
          ADC_values[1], ADC_values[2]);
      USARTSend(buffer, sizeof(buffer));
      TIM_SetCompare1(TIM3, ADC_values[0]);
      TIM_SetCompare2(TIM3, ADC_values[1]);
      TIM_SetCompare3(TIM3, ADC_values[2]);
      status = 0;
    }
}
```

ADC with DMA for Multiple Channels

```
void DMA1_Channel1_IRQHandler(void)
{
  // Test on DMA1 Channel1 Transfer Complete interrupt
  if(DMA_GetITStatus(DMA1_IT_TC1))
  {
     status=1;

    //Clear DMA1 interrupt pending bits
    DMA_ClearITPendingBit(DMA1_IT_GL1);
  }
}
```

ADC with DMA for Multiple Channels

```
void DMA_GetITStatus(uint32_t DMAy_IT)
```

- Check whether the specified DMAy Channelx interrupt has occurred or not.
    - DMAy_IT_GLx: DMAy Channelx global interrupt
    - DMAy_IT_TCx: DMAy Channelx transfer complete interrupt
    - DMAy_IT_HTx: DMAy Channelx half transfer interrupt
    - DMAy_IT_TEx: DMAy Channelx transfer error interrupt
    - In this example, it is DMA1_IT_TC1.

ADC with DMA for Multiple Channels

```
void DMA_ClearITPendingBit(uint32_t DMAy_IT)
```

- Clear the DMAy Channelx's interrupt pending bits.
    - DMAy_IT_GLx: DMAy Channelx global interrupt
    - DMAy_IT_TCx: DMAy Channelx transfer complete interrupt
    - DMAy_IT_HTx: DMAy Channelx half transfer interrupt
    - DMAy_IT_TEx: DMAy Channelx transfer error interrupt
    - In this example, it is DMA1_IT_GL1 to make sure all interrupt pending bits are clear.

Reference Readings

- http://www.longlandclan.yi.org/~stuartl/stm32f10x_stdperiph_lib_um
- Chapter 10, 12 and 14 – *Discovering the STM32 Microcontroller*, Geoffrey Brown, 2012
- RM0008 Reference Manual (STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 320bit MCUs)
- AN3116 Application note (STM32 ADC modes and their applications)
- Datasheet – STM32F103x8, STM32F103xB

End