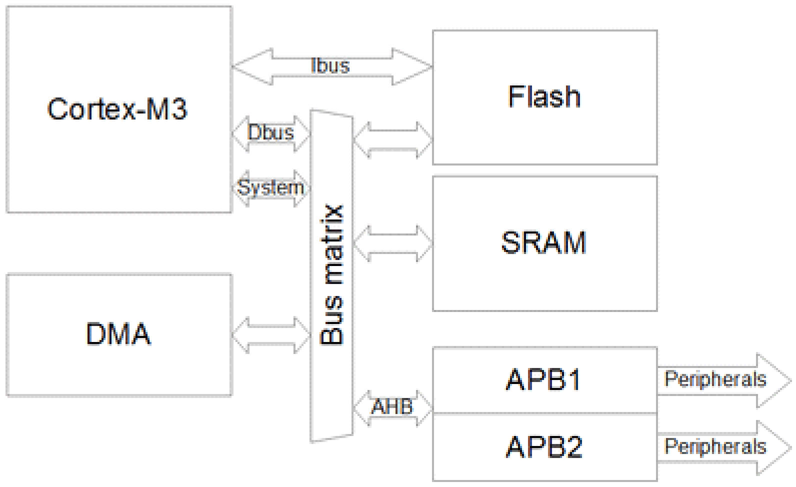
DMA

Friday, March 31, 2017 5:56 PM

Using Direct Memory Access (DMA) in STM32 projects

In many microcontroller projects you need to read and write data. It can be reading data from peripheral unit like ADC and writing values to RAM. In other case maybe you need send chunks of data using SPI. Again you need to read it from RAM and constantly write to SPI data register and so on. When you do this using processor – you loose a significant amount of processing time. In order to avoid occupying CPU most advanced microcontrollers have Direct memory Access (DMA) unit. As its name says – DMA does data transfers between memory locations without need of CPU.



Low and medium density ST32 microcontrollers have single 7 channel DMA unit while high density devices have two DMA controllers with 12 independent channels. In STM32VLDiscovery there ST32F100RB microcontroller with single DMA unit having 7 channels.
DMA can do automated memory to memory data transfers, also do peripheral to memory and peripheral to peripheral. DMA channels can be assigned one of four priority level: very high, high, medium and low. And if two same priority channels are requested at same time – the lowest number channel gets priority. DMA channel can be configured to transfer data in to circular buffer. So DMA is ideal solution for any peripheral data stream.
Speaking of physical DMA buss access it is important to note, that DMA only access bus for actual data transfer. Because DMA request phase, address computation and Ack pulse are performed during other DMA channel bus transfer. So when one DMA channel finishes bus transfer, another channel is already ready to do transfer immediately. This ensures minimal bus occupation and fast transfers. Another interesting feature of DMA bus access is that it doesn't occupy 100% of bus time. DMA takes 5 AHB bus cycles for single word transfer between memory – three of them are still left for CPU access. This means that DMA only takes maximum 40% of buss time. So even if DMA is doing intense data transfer CPU can access any memory area, peripheral at any time. If you look at block diagram you will see that CPU has separate Ibus for Flash access. So program fetch isn't affected by DMA.

Programming DMA

Simply speaking programming DMA is easy. Each channel can be controlled using four registers: Memory address, peripheral address, number of data and configuration. And all channels have two dedicated registers: DMA interrupt status register and interrupt flag clear register. Once set DMA takes care of memory address increment without disturbing CPU. DMA channels can generate three interrupts: transfer finished,

half-finished and transfer error.
As example lets write a simple program which transfers data between two arrays. To make it more interesting lets do same task using DMA and without it. Then we can compare the time taken in both cases.
Here is a code of DMA memory to memory transfer:

```
#include "stm32f10x.h"
#include "leds.h"
#define ARRAYSIZE 800
volatile uint32_t status = 0;
volatile uint32_t i;
int main(void)
{
//initialize source and destination arrays
uint32_t source[ARRAYSIZE];
uint32_t destination[ARRAYSIZE];
//initialize array
for (i=0; i<ARRAYSIZE;i++)
   source[i]=i;
//initialize led
LEDsInit();
//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 be used for memory to memory transfer
DMA_InitStructure.DMA_M2M = DMA_M2M_Enable;
//setting normal mode (non circular)
DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
//medium priority
DMA_InitStructure.DMA_Priority = DMA_Priority_Medium;
//source and destination data size word=32bit
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
//automatic memory increment enable. Destination and source
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
//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)source;
DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)destination;
//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);

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);

//LED on before transfer
LEDToggle(LEDG);
//Enable DMA1 Channel transfer
DMA_Cmd(DMA1_Channel1, ENABLE);
while(status==0) {};
  LEDToggle(LEDB);
  for (i=0; i<ARRAYSIZE;i++)
  {
    destination[i]=source[i];
  }
  LEDToggle(LEDB);

while (1)
 {
  //interrupts does the job
 }
}
```

First of all we create two arrays: source and destination. Size of length is determined by ARRAYSIZE which in our example is equal to 800
We use LED library from previous tutorial – they indicate start and stop transfer for both modes – DMA and CPU. As we see in code first of all we must turn on DMA1 clock to make it functional. Then we start loading settings in to DMA_InitStructure. For this example we selected DMA1 Channel1, so first of all we call *DMA_DeInit(DMA1_Channel1)* function which simply makes sure DMA is reset to its default values. Then turn on memory to memory mode, then we select normal DMA mode (also we could select circular buffer mode). As priority mode we assign Medium. Then we select data size t obe transferred (32-bit word). This need to be done for both – peripheral and memory addresses.
NOTE! if one of memory sizes would be different, say source 32-bit and destination 8- bit – then DMA would cycle four times in 8 bit chunks.
Then we load destination, source start addresses and amount of data to be sent. After load these values using DMA_Init(DMA_Channel1, &DMA_InitStructure). After this operation DMA is prepared to do transfers. Any time DMA can be fired using DMA_Cmd(DMA_Channel1, ENABLE) command.
In order to catch end of DMA transfer we initialized DMA transfer Complete on channel1 interrupt.

```
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);
```

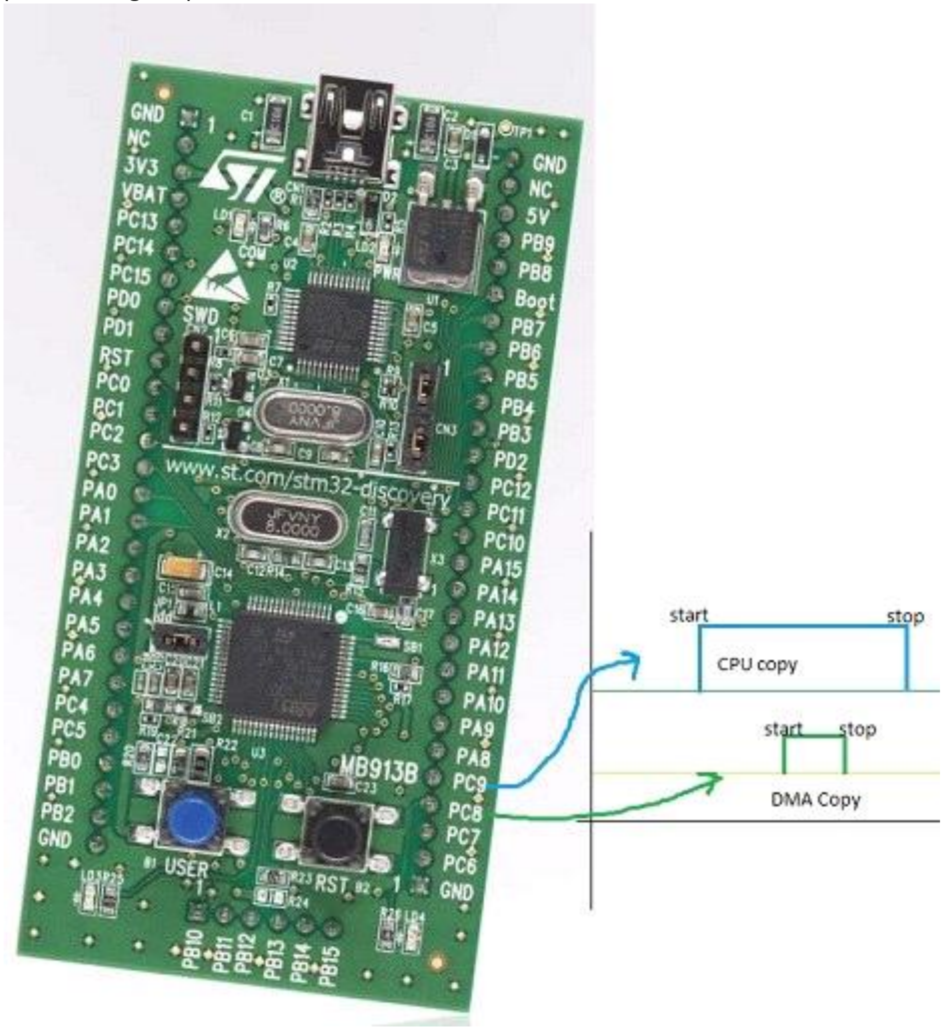
Where we could toggle LED and change status flag giving signal to start CPU transfer test.

```
void DMA1_Channel1_IRQHandler(void)
{
 //Test on DMA1 Channel1 Transfer Complete interrupt
 if(DMA_GetITStatus(DMA1_IT_TC1))
 {
    status=1;
    LEDToggle(LEDG);
  //Clear DMA1 Channel1 Half Transfer, Transfer Complete and Global interrupt pending bits
  DMA_ClearITPendingBit(DMA1_IT_GL1);
 }
}
```

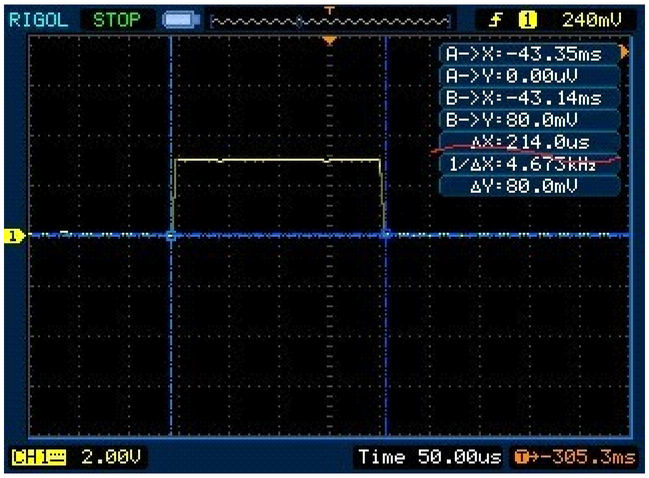
CPU based memory copy routine is simple:

```
//wait for DMA transfer to be finished
while(status==0) {};
  LEDToggle(LEDB);
  for (i=0; i<ARRAYSIZE;i++)
  {
    destination[i]=source[i];
  }
  LEDToggle(LEDB);
```

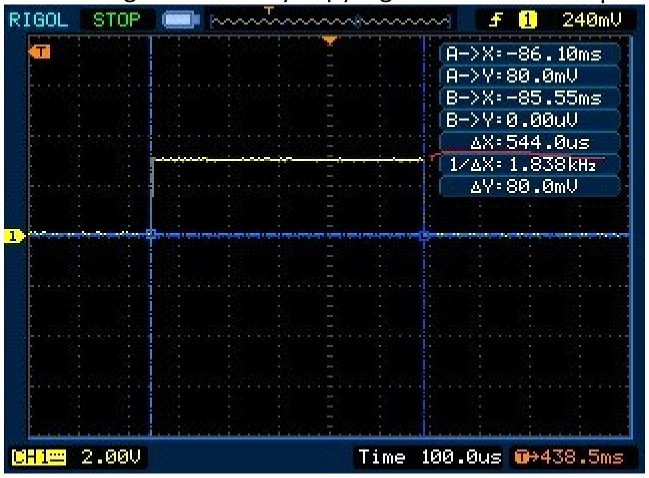
Since LEDG is connected to GPIOC pin 9 and LEDB is connected to GPIOC pin 8 we could track start and end pulses using scope:



So using 800 32-bit word transfer using DMA took 214µs:



While using CPU memory copy algorithm it took 544μs:



This shows significant increase of data transfer speed (more than two times). And with DMA biggest benefit is that CPU is totally unoccupied during transfer and may do other intense tasks or simply go in to sleep mode. Hope this example gives an idea of DMA importance. With DMA we can do loads of work only in hardware level. We will get back to it when we get to other STM32 features like ADC.