一　系统公用函数

此函数库包含系统时钟配置函数以及端口时钟开启指令，K60 系统时钟配置，core/system clock 最大 150M,bus clock 75M,flexbus　50M,flash clock 25M，具体时钟配置以及端口时钟，参考 MCG 模块以及 SIM,PORT 模块

函数定义：

void SYS_CLOCK_SET(uint32 X,uint32 div_core,uint32 div_bus,uint32 div_flexbus,uint32 flash);

void PORT_Enable(uint32 X);

void PORT_Disable(uint32 X);

void PORT_Enable_ALL(void);

void PORT_Configuer(uint32 port,uint32 pin,uint8 select);

函数功能介绍

void SYS_CLOCK_SET(uint32 X,uint32 div_core,uint32 div_bus,uint32 div_flexbus,uint32 flash);

参数：X：SYS_CLOCK_150M,SYS_CLOCK_100M，分别表示系统内核时钟为 150M，100M,若要自己配置其他参数，可修改函数内部的 MCG_C5=MCG_C5_PRDIV0(X/16),和 MCG_C6|=MCG_C6_VDIV0(X%16);两条语句参数，具体因子，参考 datesheet　MCG 章节。

div_core:内核分频因子，最高分频后为 150M，值 为 N 表示 N 分频

div_bus：总线分频因子，最高分频后为 75M，值 为 N 表示 N 分频

Flexbus：flexbus 分频因子，最高分频后为 50M，值 为 N 表示 N 分频

Flash:flash 分频因子，最高分频后为 25M，值 为 N 表示 N 分频

void PORT_Enable(uint32 X);

参数：PORTA,PORTB,PORTC,PORTD,PORTE,PORTF

功能：使能某一端口时钟。无论使用哪种模块，只要用到复用引脚必须先使能端口时钟，否则程序无法执行

void PORT_Disable(uint32 X):

参数：PORTA,PORTB,PORTC,PORTD,PORTE,PORTF

功能：关闭端口时钟

void PORT_Enable_ALL(void)：打开所有时钟

void PORT_Configuer(uint32 port,uint32 pin,uint8 select);

参数：port :端口引脚 PORTA,PORTB,PORTC,PORTD,PORTE,PORTF

Pin:端口　0-31

Select:复用功能选择　0-7，具体采用什么功能，参考 datesheet 第十章

二 GPIO

最基本的端口电平控制函数库，输入输出配置，端口中短使能，驱动强度配置，上下拉，开源输出，滤波等功能设定

函数定义：

```
void GPIO_Enable(uint32 port,uint32 pin);
void GPIO_Disable(uint32 port,uint32 pin);
void GPIO_INIT(uint32 port,uint32 pin,uint32 dir);
void GPIO_SET(uint32 port,uint32 pin,uint32 date);
uint32 GPIO_GET(uint32 port,uint32 pin);
void GPIO_OPPSITE(uint32 port , uint32 pin);
void GPIO_DSE(uint32 port ,uint32 pin);
void GPIO_ODE(uint32 port ,uint32 pin);
void GPIO_PFE(uint32 port,uint32 pin);
void GPIO_SER(uint32 port,uint32 pin);
void GPIOPULL_Enable(uint32 port,uint32 pin);
void GPIOPULL_Disable(uint32 port,uint32 pin);
void GPIOPULL_SET(uint32 port ,uint32 pin,uint32 state);
void GPIOINT_Enable(uint32 port,uint32 pin,uint32 mode);
void GPIOINT_Disable(uint32 port,uint32 pin);
void GPIOINT_CLEAR(uint32 port,uint32 pin);
```

函数功能介绍

void GPIO_Enable(uint32 port,uint32 pin);
void GPIO_Disable(uint32 port,uint32 pin);
参数：port(端口):PORTA---PORTF
pin 引脚：0-31
功能：使能和关闭 GPIO 功能，实际和 PORT_Configuer(port,pin,1);这条语句效果相同

void GPIO_INIT(uint32 port,uint32 pin,uint32 dir);
参数：dir: OUTPUT INPUT
功能：初始化端口，设定方向

void GPIO_SET(uint32 port,uint32 pin,uint32 date);
参数：date:数据输出电平 0 或 1
功能：设定端口输出电平，首先要初始化设置输出

uint32 GPIO_GET(uint32 port,uint32 pin);//设置为输入时，获得某一引脚的电平
void GPIO_OPPSITE(uint32 port , uint32 pin);//设置为输出时，引脚电平反向
void GPIO_DSE(uint32 port ,uint32 pin);//设定驱动强度，使用后为 high drive
void GPIO_ODE(uint32 port ,uint32 pin);//设定为 OPEN DRAIN，
void GPIO_PFE(uint32 port,uint32 pin);//设定为 Passive Filter
void GPIO_SER(uint32 port,uint32 pin);//设定为 low slew rate
void GPIOPULL_Enable(uint32 port,uint32 pin);//设定为使能上下拉
void GPIOPULL_Disable(uint32 port,uint32 pin);//设定为去使能上下拉
void GPIOPULL_SET(uint32 port ,uint32 pin,uint32 state);
参数：state:PULL_UP,PULL_DOWN

功能：设置端口的上下拉，若不设置，在端口设置为输入时，默认为高电平

void GPIOINT_Enable(uint32 port,uint32 pin,uint32 mode);

参数：mode: DMA_RISING
DMA_FALLING
DMA_EITHER
LEVEL_LOW
LEVEL_RISING
LEVEL_FALLING
LEVEL_EITHER
LEVEL_HIGH

功能：打开 GPIO 中断，并设置中断模式

void GPIOINT_Disable(uint32 port,uint32 pin);//关闭 GPIO 端口中断

void GPIOINT_Clear(uint32 port,uint32 pin);

功能：清除端口中断标志，一般每次进入中断后要清除标志，具体中断使用，参考中断模块

EXAMPLE:若 PTB16 接一个灯，灯会以 2s 为周期闪烁，单步调试观察比较清楚

```
#include "all.h"

void main(void)
{
    SYS_CLOCK_SET(SYS_CLOCK_150M,1,1,3,5);
    GPIO_INIT(PORTB,16,OUTPUT);
    GPIO_SET(PORTB,16,1);
    SYSDelay_ms(1000);
    GPIO_SET(PORTB,16,0);
    SYSDelay_ms(1000);
    while(1){
        GPIO_OPPSITE(PORTB,16);
        SYSDelay_ms(1000);
    }
}
```

三　中断使用

K60 中断使用方法：

```
//          DisableInterrupts                关闭总中断
//          Enable_IRQ();       中断向量号查 MK60F15.H,对应号
//          INI_Enable();        使能某中断中断服务，比如 GPIOINT_Enable()
//          在 main 函数中写函数入口，查 start_up.s 中的函数名
//         EnableInterrupts
```

函数定义：

```
void Enable_IRQ(int IRQ_NUM);
void Disable_IRQ(int IRQ_NUM);
void SET_IRQ_PRIOR(int IRQ_NUM,int IRQ_PRIOR);
简介：K60 中断源有 256 个，16 个优先级配置，
```

函数功能介绍

void Enable_IRQ(int IRQ_NUM);void Disable_IRQ(int IRQ_NUM);//使能和关闭

参数：IRQ_NUM,中断向量号，查 MK60F15.H,中，有对中断号定义 直接 paste 过来即可

void SET_IRQ_PRIOR(int IRQ_NUM,int IRQ_PRIOR);

参数：IRQ_PRIOR ：优先级，0-15

中断应用函数：当 PTB16 接一个 LED 灯，PTB17 接高电平时，程序运行，灯会以某一频率闪烁

EXAMPLE:

```
#include "all.h"
uint32 i=0;
void delay(uint32 n)
  {
      int x,y;
      for(x=0;x<n;x++)
         for(y=110;y>0;y--);
  }
void main(void)
{
    SYS_CLOCK_SET(SYS_CLOCK_100M,1,1,3,5);
    DisableInterrupts
    PORT_Enable(PORTB);
    GPIO_Enable(PORTB,16);
    GPIO_Enable(PORTB,17);
    GPIO_INIT(PORTB,16,OUTPUT);
    GPIO_INIT(PORTB,17,INPUT);
    GPIOPULL_Enable(PORTB,17);
    GPIOPULL_SET(PORTB,17,PULL_DOWN);

    GPIO_SET(PORTB,16,1);
    Enable_IRQ(INT_PORTB);
    GPIOINT_Enable(PORTB,17,LEVEL_HIGH);
    EnableInterrupts
    while(1){

    }
}
void PORTB_IRQHandler(void)
```

```
{
        GPIOINT_Clear(PORTB,17);
        i++;
        GPIO_OPPSITE(PORTB,16);
          delay(100000);
}
```

四 PIT 定时器

K60 有四个定时器，分别为 PIT0~PIT4,时钟为 BUS 时钟

函数定义：

```
void PIT_INIT(uint32 PIT_NUM,uint32 TIMEOUT,uint32 WAY_DEBUG);
void PIT_Enable(uint32 PIT_NUM);
void PIT_Disable(uint32 PIT_NUM);
uint32 PIT_Read(uint32 PIT_NUM);
void PIT_Reload(uint32 PIT_NUM,uint32 TIMEOUT);
void PITINT_Enable(uint32 PIT_NUM);
void PITINT_Disable(uint32 PIT_NUM);
void PITINT_Clear(uint32 PIT_NUM);
```

功能：

- Timers can generate DMA trigger pulses
- Timers can generate interrupts
- All interrupts are maskable
- Independent timeout periods for each timer

函数功能介绍

```
void PIT_INIT(uint32 PIT_NUM,uint32 TIMEOUT,uint32 WAY_DEBUG);
    参数：PIT_NUM：PIT0 PIT1 PIT2 PIT3
    //      TIMEOUT:装载值
    //      WAY_DEBUG：时钟是否在 DEBUG 下运行，DEBUG_CONTINUE,DEBUG_STOP
    PIT 定时器是递减计数器，若 TIMEOUT=SYS_CLOCK-1，那么定时时间为 1S，
    SYS_CLOCK 为 extern 变量，可直接使用，值为内核时钟频率

void PIT_Enable(uint32 PIT_NUM);//功能：打开定时器开始递减计数

void PIT_Disable(uint32 PIT_NUM);//功能：关闭某一定时器

uint32 PIT_Read(uint32 PIT_NUM);//读取当前定时器的值

void PIT_Reload(uint32 PIT_NUM,uint32 TIMEOUT);//重新装载值

void PITINT_Enable(uint32 PIT_NUM);//PIT 中断使能函数

void PITINT_Disable(uint32 PIT_NUM);//PITPIT 中断去使能函数

void PITINT_Clear(uint32 PIT_NUM);//PIT 中断标志清除
```

EXAMPLE:定时 1s，通过 PTB16 接灯观察

```
#include "all.h"
uint32 i=0;
void delay(uint32 n)
```

```
  {
      int x,y;
      for(x=0;x<n;x++)
        for(y=110;y>0;y--);
  }
void main(void)
{
    SYS_CLOCK_SET(SYS_CLOCK_150M,1,2,3,6);
    DisableInterrupts
    GPIO_INIT(PORTB,16,OUTPUT);
    GPIO_SET(PORTB,16,1);
    PIT_CLOCK_Enable
    PIT_INIT(PIT3,75000000,DEBUG_STOP);
    Enable_IRQ(87);
    EnableInterrupts
    PIT_Enable(PIT3);
    PITINT_Enable(PIT3);
    while(1){
    }
}
void PIT3_IRQHandler(void)
 {
       PITINT_Clear(PIT3);
       DisableInterrupts
       GPIO_OPPSITE(PORTB,16);
       EnableInterrupts
 }
```

五 UART

函数定义：

```
void UART_Enable(uint32 UART_NUM);
void GPIOType_UART(uint32 PORT_R_T);
uint32 UART_INIT(uint32 UART_NUM,uint32 BUS_CLOCK,uint32 BAUD);
uint8 UART_R1(uint32 UART_NUM,uint8 *fp);
uint8 UART_S1(uint32 UART_NUM,uint8 ch);
uint8 UART_RN(uint32 UART_NUM,uint32 lenth,uint8* date);
uint8 UART_SN(uint32 UART_NUM,uint32 lenth,uint8* date);
uint8 UART_SS(uint32 UART_NUM,void *buff);
void UARTINT_Enable(uint32 UART_NUM,uint32 INT_CLASS);
void UARTINT_Disable(uint32 UART_NUM,uint32 INT_CLASS);
```

功能介绍：UART 一共有 6 个模块，分别为 UART0~UART5,其中，UART0,UART1 采用内核时钟，其他采用总线时钟

void UART_Enable(uint32 UART_NUM);

参数：UART0~UART5

功能：使能某一 UART

void GPIOType_UART(uint32 PORT_R_T);

功能：选择某一 UART 模块的引脚，函数内部包括了对时钟的使能

参数：UART0:PORTD_6_7 ，PORTB_16_17 ，PORTA_14_15 ，PORTA_1_2

UART1:PORTE_0_1, PORTC_3_4

UART2:PORTD_2_3

UART3:PORTE_4_5,PORTB_10_11,PORTC_16_17

UART4:PORTE_24_25,PORTC_14_15

UART5:PORTE_8_9,PORTD_8_9

void UART_INIT(uint32 UART_NUM,uint32 BUS_CLOCK,uint32 BAUD);

参数：BUS_CLOCK：如果设定的总线频率是内核频率的一半，就直接输入总线频率，否则若为UART0和UART1，请输入自己设定的内核时钟频率的一半，其他模块直接输入总线时钟频率

BAUD：波特率设定

uint8 UART_R1(uint32 UART_NUM,uint8 *fp);

参数：fp:判断是否收到数据 1表示成功，0表示失败

功能：接收一个字节

返回值：1表示发送成功，0表示失败

uint8 UART_S1(uint32 UART_NUM,uint8 ch);

//发送一个字节，返回值：1表示发送成功，0表示失败

uint8 UART_RN(uint32 UART_NUM,uint32 lenth,uint8* date);

参数：lenth,接受字节长度，date：接收的字节

返回值：1表示发送成功，0表示失败

功能：接受N个字节，

uint8 UART_SN(uint32 UART_NUM,uint32 lenth,uint8* date);

参数：lenth:字符串长度

date:接受数据数组

返回值：1成功，0失败

功能：发送N个字节

uint8 UART_SS(uint32 UART_NUM,void *buff);

功能：发送一个字符串

void UARTINT_Enable(uint32 UART_NUM,uint32 INT_CLASS);

参数：INT_CLASS（中断类型）：ILIE RIE TCIE TIE PEIE FEIE NEIE ORIE

void UARTINT_Disable(uint32 UART_NUM,uint32 INT_CLASS);//关闭中断

EXAMPLE:波特率设置为 115200，用端口 PTA14,PTA15

```
#include "all.h"
uint32 i=0,j=0;
uint32  value=0;
uint8 flag;
uint32 frequence=0;
void main(void)
{
    SYS_CLOCK_SET(SYS_CLOCK_150M,1,2,3,6);
    DisableInterrupts
    PORT_Enable(PORTA);
    GPIOType_UART(PORTA_14_15);
    UART_Enable(UART0);
    UART_INIT(UART0,75000,115200);
    Enable_IRQ(61);
    UARTINT_Enable(UART0,RIE);
    EnableInterrupts
    while(1){
      /*
        value=UART_R1(UART0,&flag);
            if(flag)
             {
                UART_S1(UART0,value+2);
                UART_SS(UART0,"I LOVE YOU");
             }*/
    }
}
void UART0_RX_TX_IRQHandler(void)
 {
     i++;
     DisableInterrupts
     value=UART_R1(UART0,&flag);
            if(flag)
             {
                 UART_S1(UART0,i);
                UART_S1(UART0,value+2);
                UART_SS(UART0,"I LOVE YOU");
             }
     EnableInterrupts
 }
```

六 ADC

函数定义：

```
void ADC_Enable(uint32 ADC_NUM);
void ADCINIT_A(uint32 ADC_NUM,uint32 channel,uint8 accuracy);
uint8 ADC_S_A(uint32 ADC_NUM);
uint8 ADC_S_B(uint32 ADC_NUM);
void ADC_VALUE_A(uint32 ADC_NUM,int *value);
void ADC_VALUE_B(uint32 ADC_NUM,int *value);
void ADC_Disable_A(uint32 ADC_NUM);
void ADC_Disable_B(uint32 ADC_NUM);
void HardWare_ave(uint32 ADC_NUM,char samples_num);
void ADCHardWare_INIT(uint32 ADC_NUM,char trigger_mode,char Pre_trigger,uint32 trigger_select);
void ADCINIT_B(uint32 ADC_NUM,uint32 channel,uint8 accuracy);
void ADCHardWare_configure(uint32 ADC_NUM,uint32 trigger_mode,char sample_ave);
void ADC_Speed(uint32 ADC_NUM,char speed,char sample_time);
void ADC_DIFF_A(uint32 ADC_NUM,uint32 channel,char accuracy);
```

ADC 功能简介：ADC 触发方式包括软件出发，硬件触发，采样有 simple-end ,和 diff 模式以及比较功能，有多位精度选择,以及硬件平均功能,ADC 某些模块包括通道 A 和通道 B,用的时候可以自己选择,具体用法参考 datesheet

函数功能介绍：

void ADC_Enable(uint32 ADC_NUM);

参数：ADC_NUM:ADC0 ADC1 ADC2 ADC3

功能：打开 ADC 时钟

void ADCINIT_A(uint32 ADC_NUM,uint32 channel,uint8 accuracy);

参数：channel:通道选择：

DADP0	DADP1	DADP2	DADP3
AD4	AD5	AD6	AD7
AD8	AD9	AD10	AD11
AD12	AD13	AD14	AD15
AD16	AD17	AD18	AD19
AD20	AD21	AD22	AD23

注意：每个 ADC 通道开启后，应该打开对应的引脚，并使用其引脚配置功能，
用函数 void PORT_Configuer(uint32 port,uint32 pin,uint8 select);不过默认便为 AD 功能，选用即可

accuracy ：精 度:bite_8 bite_10 bite_12 bite_16
分别表示 8 位，10 位，12 位 和 16 位精度

函数功能：初始化通道 A，若选用 A 通道则使用这个函数，此 API 对软件触发采用 A 通道，硬件触发采用 B 通道，故若使用软件触发使用此函数即可

uint8 ADC_S_A(uint32 ADC_NUM)//返回 ADC 状态值

void ADC_VALUE_A(uint32 ADC_NUM,int *value);//取 ADC 转换后的值，函数内部清除标志

void ADC_Disable_A(uint32 ADC_NUM);//关闭 ADC，选择通道 11111 关闭

void ADCHardWare_INIT(uint32 ADC_NUM,char trigger_mode,char Pre_trigger,uint32 trigger_select);

//参数：ADC_NUM:ADC0~3

// triggle_mode：PDB_Trigger，Alternate_Trigger

// Pre_trigger：Pre_A，Pre_B 采用 A 或者 B 通道

```
//      trigger_select:可以配置成PIT ,FTM等,
          External_Trigger     HIGH_SPEED_COM0      HIGH_SPEED_COM1   HIGH_SPEED_COM2
          PIT0_Trigger         PIT1_Trigger         PIT2_Trigger      PIT3_Trigger
          FTM0_Trigger         FTM1_Trigger         FTM2_Trigger      FTM3_Trigger
          RTC_alarm            RTC_seconds          LOW_POWER         HIGH_SPEED_COM3

     void ADCINIT_B(uint32 ADC_NUM,uint32 channel,uint8 accuracy);
     通道B初始化，可结合上一个函数一起使用。
     void ADCHardWare_configure(uint32 ADC_NUM,uint32 trigger_mode,char sample_ave);
     参数：//硬件触发设置：
          //trigger_mode:选择Trigger_once，Trigger_always
          //sample_ave：Sample_4，Sample_8，Sample_16，Sample_32
     uint8 ADC_S_B(uint32 ADC_NUM);//返回ADC状态值
     void ADC_VALUE_B(uint32 ADC_NUM,int *value);//取ADC转换后的值，函数内部清除标志
     void ADC_Disable_B(uint32 ADC_NUM);//关闭ADC，选择通道11111关闭
     void ADCINIT_B(uint32 ADC_NUM,uint32 channel,uint8 accuracy);//配置b通道参数
     void ADC_Speed(uint32 ADC_NUM,char speed,char sample_time);
     参数：//      speed:HIGH_SPEED,NORMAL_SPEED
          //      sample_time:ADCK_24,ADCK_16,ADCK_10,ADCK_6
     功能：设置ADC转换速度和长采样时间
     void ADC_DIFF_A(uint32 ADC_NUM,uint32 channel,char accuracy);
     //简介：差分模式AD采样,只采用A,长采样时间
     //参数：channel:DAD0~4
       //      accuracy:bite9  11  13  16


       EXAMPLE:
//软件触发
#include "all.h"
uint32 i=0,j=0,result=0;
int  VALUE=0;
uint32 frequence=0;
void delay(uint32 n)
  {
      int x,y;
      for(x=0;x<n;x++)
        for(y=110;y>0;y--);
  }
void main(void)
{
    SYS_CLOCK_SET(SYS_CLOCK_150M,1,2,3,6);
    PORT_Enable_ALL();
    ADC_Enable(ADC3);
    ADCINIT_A(ADC3,AD9,bit_16);
    while(1)
     {
         while(!ADC_S_A(ADC3));
         ADC_VALUE_A(ADC3,&VALUE);
         result=VALUE*3300/65536;
```

```
    }
}
//硬件触发:
#include "all.h"
uint32 i=0,j=0,result=0;
int  VALUE=0;
uint32 frequence=0;
void delay(uint32 n)
  {
      int x,y;
      for(x=0;x<n;x++)
        for(y=110;y>0;y--);
  }
void main(void)
{
    SYS_CLOCK_SET(SYS_CLOCK_150M,1,2,3,6);
    PIT_CLOCK_Enable
    PIT_INIT(PIT0,SYS_CLOCK/100,DEBUG_CONTINUE);
    PORT_Enable_ALL();
    ADC_Enable(ADC0);
    ADCHardWare_INIT(ADC0,Alternate_Trigger,Pre_A,PIT0_Trigger);
    ADCINIT_A(ADC0,DADP0,bit_16);
    HardWare_configure(ADC0,Trigger_always,Sample_8);
    PIT_Enable(PIT0);
    while(1)
     {
         while(!ADC_S_A(ADC0));
         ADC_VALUE_A(ADC0,&VALUE);
         result=VALUE*3300/65536;
     }
}
```

差分:

```
#include "all.h"
int32 i=0,j=0,result1=0,result2=0;
int  VALUE=0;
void delay(uint32 n)
  {
      int x,y;
      for(x=0;x<n;x++)
        for(y=110;y>0;y--);
  }
void main(void)
{
    SYS_CLOCK_SET(SYS_CLOCK_150M,1,2,3,6);
    PORT_Enable_ALL();
    ADC_Enable(ADC0);
    ADC_DIFF_A(ADC0,DAD0,bite_16);
    HWREG(ADC_CFG1_BASE +ADC0*(0X1000))|=ADC_CFG1_ADIV(3)|ADC_CFG1_ADICLK(0);
```

```
    while(1)
     {
         while(!ADC_S_A(ADC0));
         ADC_VALUE_A(ADC0,&VALUE);
         result1=VALUE*3300/65536;
     }
}
```

看门狗

具体介绍参考 datesheet,这里只提供使用方法

```
//*********************************************************************
//简介：解锁看门狗
//*********************************************************************
void wdog_unlock(void);
//*********************************************************************
//简介：使能看门狗
//*********************************************************************
void wdog_enable(void);
//*********************************************************************
//简介：去使能看门狗
//*********************************************************************
void wdog_disable(void);
//*********************************************************************
//简介：初始化看门狗，毫秒复位，必须满足 ms>4
//*********************************************************************
void wdog_init_ms(uint32 ms);
//*********************************************************************
//简介：喂狗
//*********************************************************************
void wdog_feed(void);
```

EXAMPLE:若注销 wdog_feed();，则灯会以某一频率闪烁，不注销时灯暗度为正常的一半

```
#include "all.h"
void delay(uint32 n)
  {
      int x,y;
      for(x=0;x<n;x++)
        for(y=110;y>0;y--);
  }
void main(void)
 {
     wdog_init_ms(3000);//定时 3S
     GPIO_INIT(PORTB,16,OUTPUT);
     GPIO_SET(PORTB,16,1);
     delay(100000);
     GPIO_SET(PORTB,16,0);
     delay(100000);
     wdog_enable();
     while(1)
      {
          GPIO_OPPSITE(PORTB,16);
          //wdog_feed();
      }
 }
```

DAC

具体介绍参考 datesheet，这里只提供使用方法

```
//*****************************************************
//DAC 初始化函数
//参数：DAC0.DAC1
//*****************************************************
void DAC_INIT(uint32 DAC_NUM);

//*********************************************************
//输出电压函数    毫伏为单位
//*********************************************************
void DAC_OUT(uint32 DAC_NUM,uint16 mv);
```

EXAMPLE:测量单片机的 DAC 输出引脚，输出约为 1500MV

```
#include "all.h"

void main(void)
 {
      SYS_CLOCK_SET(SYS_CLOCK_150M,1,2,3,6);
     DAC_INIT(DAC1);
          DAC_OUT(DAC1,1500);
          while(1);
 }
```

API of DMA

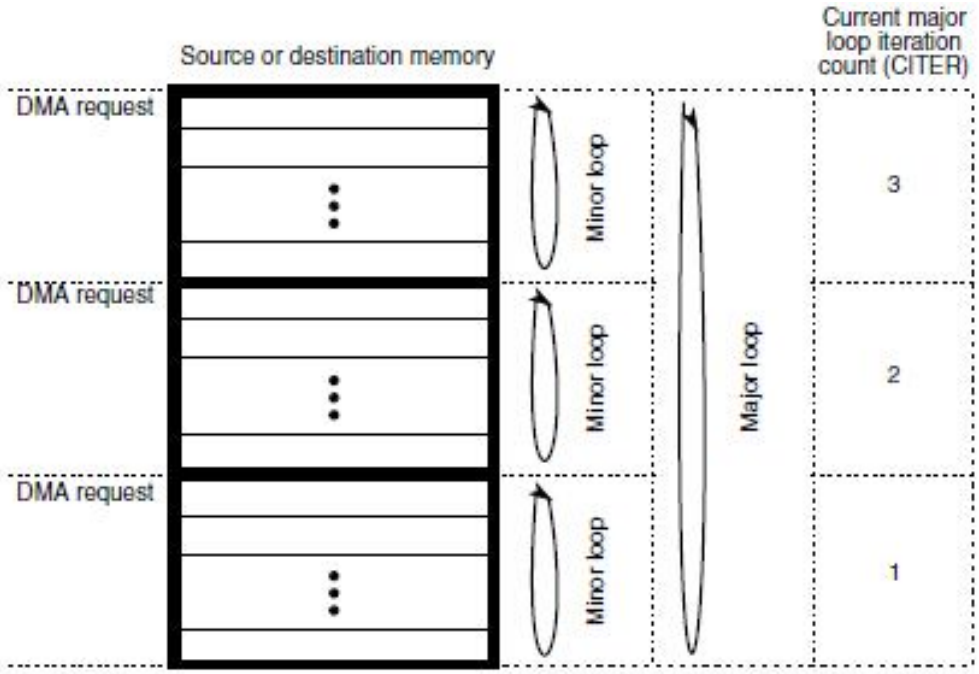


Figure 22-548. Example of multiple loop iterations

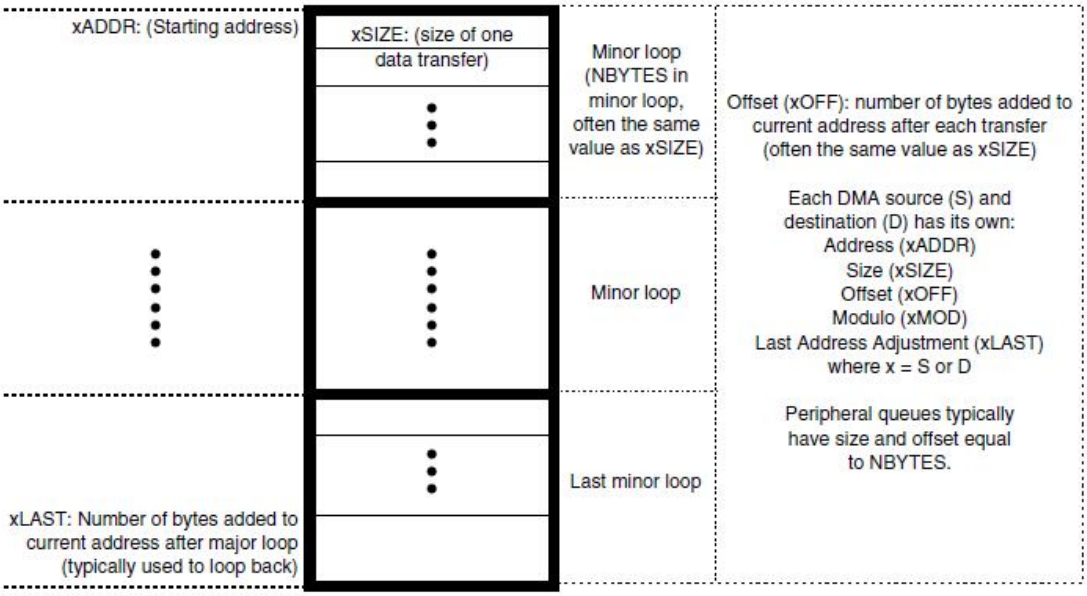
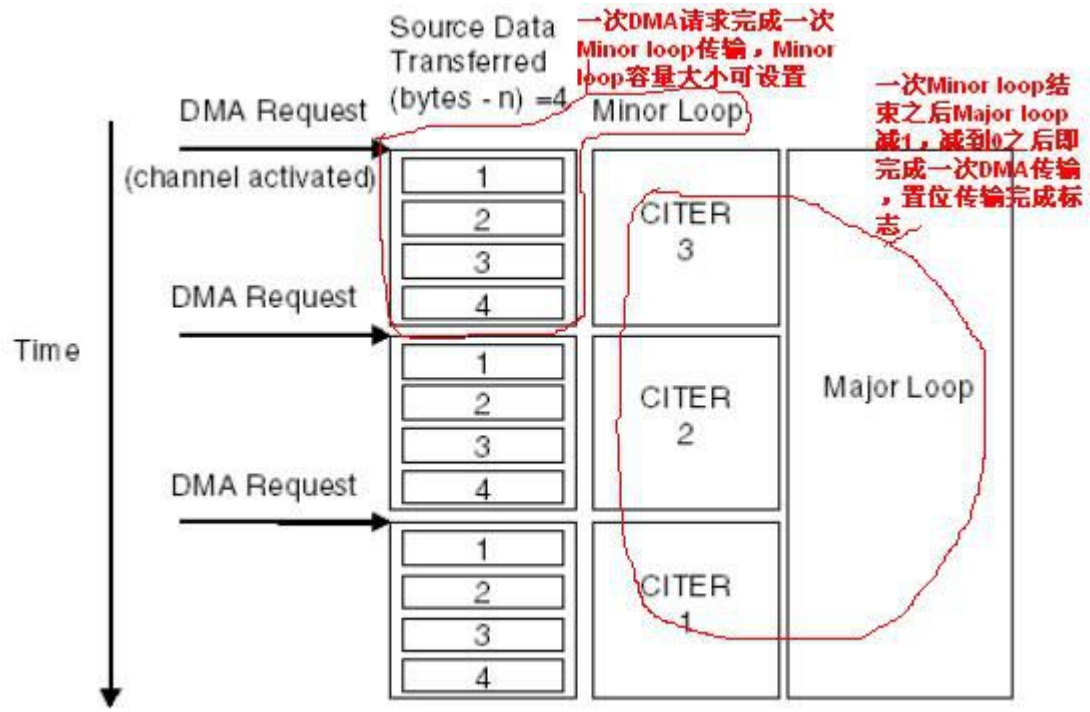


Figure 22-549. Memory array terms



/********** 函数定义 **********/

```
void  DMA_Clear_Int(uint8 CHx);
void  DMA_Init(uint8 DMAMUXx, uint8  CHx, uint32 Source,uint8 Mode);
void DMA_Source(uint8 CHx, uint32 Src_Addr,int32 Src_AddrOffset, uint32 Src_DataSize,
int32 Reset_Src_Addr);
void DMA_Destination(uint8 CHx, uint32 Dest_Addr,int32 Dest_Addr_Offset, uint32
Dest_DataSize, int32  Reset_Dest_Addr);
void  DMA_Int_Enable(uint8  CHx, uint32 Int_Type);
void  DMA_Int_Disable(uint8  CHx);
void  DMA_AutoClose_Enable(uint8  CHx);
void DMA_AutoClose_Disable(uint8 CHx);
void  DMA_Major_Loop_Num(uint8 CHx, uint16  Cycles);
void DMA_Minor_Bytes(uint8 CHx,  uint16  Bytes);
void  DMA_Software_Initiate(uint8  CHx);
void  DMA_Enable();
void DMA_Disable();

void  DMA_Debug_Enable(uint8  CHx);
void  DMA_Debug_Disable(uint8 CHx);
void DMA_Set_Group_Priority(uint8 CHx, uint8 Group0_Priority, uint8 Group1_Priority);
void DMA_Set_Channel_Priority(uint8 CHx, uint8 Priority);
uint8  DMA_Get_Channel_Priority(uint8 CHx);
uint8 DMA_Get_Group_Priority(uint8 CHx);
```

/********** END of 函数定义**********/

/********** 函数功能介绍 **********/

DMAMUX0 对应着通道 0~15
DMAMUX1 对应着通道 16~31

void DMA_Clear_Int(uint8 CHx);
功能:清除中断标志位,用于中断服务函数的第一句
参数: CHx: **DMA_CH0~DMA_CH31**//选择通道 0~31

void DMA_Init(uint8 DMAMUXx, uint8 CHx, uint32 Source,uint8 Mode);
功能:选择通道,触发源,是否周期性触发
参数: DMAMUXx: **DMAMUX0,DMAMUX1**
CHx: **DMA_CH0~DMA_CH31**//选择通道 0~31
Source: //选择触发源,触发源放在DMA.h文件
Mode: **DMA_Normal_ModeDMA_Periodic_Mode**
//是否选择周期触发模式

void DMA_Source(uint8 CHx, uint32 Src_Addr,int32 Src_AddrOffset, uint32 Src_DataSize, int32 Reset_Src_Addr);
功能:选择通道
输入源数据的地址
有符号的源地址偏移.配置源数据地址偏移，即每执行完一次**Src_DataSize**的传输就对源数据地址进行偏移**Src_Addr_Offset**个字节
源数据类型的字节长度,**8BIT,16BIT,32BIT,16BYTE**
有符号的源地址最终偏移.主的计数次数（major iteration count）减到零后，重新调整源地址,原来的源数据地址的偏移

参数: CHx: **DMA_CH0~DMA_CH31**//选择通道0~31
Src_Addr: 源数据的地址,注意是否需要地址符**&**
Src_AddrOffset: **有符号**源地址偏移
配置源数据地址偏移，即每执行完一次**Src_DataSize**的传输就对源数据地址进行偏移**Src_Addr_Offset**个字节
Src_DataSize: **8BIT,16BIT,32BIT,16BYTE**//源数据类型的字节长度
Reset_Src_Addr: **有符号**源地址最终偏移
主的计数次数（major iteration count）减到零后，重新调整源地址,原来的源数据地址的偏移

void DMA_Destination(uint8 CHx, uint32 Dest_Addr,int32 Dest_Addr_Offset, uint32 Dest_DataSize, int32 Reset_Dest_Addr);
*和DMA_Source的配置同理

void DMA_Int_Enable(uint8 CHx, uint32 Int_Type);
功能:使能相应通道的中断,并设置中断类型.中断类型有两种,1.主循环计数器减到零时进入中断,2.

主循环计数器减到一半时进入中断

参数: CHx: **DMA_CH0~DMA_CH31**//选择通道0~31

Int_Type: **DMA_MAJOR,DMA_HALF**

```
void DMA_Int_Disable(uint8 CHx);
```

功能:关闭相应通道的中断

参数: CHx: **DMA_CH0~DMA_CH31**//选择通道0~31

```
void DMA_AutoClose_Enable(uint8 CHx);
```

当主循环计数器减到零时自动关闭DMA的硬件请求,如果调用这条语句会使DMA一直执行
参数: CHx: **DMA_CH0~DMA_CH31**//选择通道0~31

void DMA_AutoClose_Disable(uint8 CHx);
功能: 一直执行DMA传输
参数: CHx: **DMA_CH0~DMA_CH31**//选择通道0~31

void DMA_Major_Loop_Num(uint8 CHx, uint16 Cycles);
//设置主循环的循环次数,即传输**Cycles**个次计数器的字节数
参数: CHx: **DMA_CH0~DMA_CH31**//选择通道0~31
Cycles 有例程,例程很直观

void DMA_Minor_Bytes(uint8 CHx, uint16 Bytes);
功能:设置次计数,即每一次传输数据字节的个数
当个数达到源地址配置的**8BIT/16BIT/32BIT/16BYTE**时,DMA便将数据存在缓冲区
当个数达到目的地址配置的**8BIT/16BIT/32BIT/16BYTE**时,DMA便开始把 缓冲区的数据传输到目的地址
直到传输完**Bytes**个字节
参数: CHx: **DMA_CH0~DMA_CH31**//选择通道0~31
Cycles 有例程,例程很直观

void DMA_Software_Initiate(uint8 CHx);
功能:软件启动(触发)传输
其实如果是**Src_ALWAYS_***(DMA常使能请求源)不调用这句也会触发的
参数: CHx: **DMA_CH0~DMA_CH31**//选择通道0~31

void DMA_Channal_Enable(uint8 CHx);
功能:使能通道,若不使能,则只执行一次副循环,有例程,例程很直观
默认情况下是禁能的,所以一般要调用
参数: CHx: **DMA_CH0~DMA_CH31**//选择通道0~31

void DMA_Channal_Disable(uint8 CHx);
参数: CHx: **DMA_CH0~DMA_CH31**//选择通道0~31

void DMA_Enable();
功能:重新开始DMA模块,默认情况下,是使能的
前面用了**DMA_Disable()**,才需要用到这条语句,重新打开DMA模块
当**DMA_Disable()**被用在配置的第一句时时可以用在配置的最后一句(除了软件触发,就是**DMA_Software_Initiate(DMA_CH1)**这句)

void DMA_Disable();
功能:停止DMA模块,可以用在配置的第一句

void DMA_Debug_Enable(uint8 CHx);
功能:在调试模式下,DMA禁止新通道的开始,正在执行的通道可以完成.
当调用DMA_Debug_Disable(DMA_CH0)时, Channel execution resumes
参数: CHx: DMA_CH0~DMA_CH31//选择通道0~31

void DMA_Debug_Disable(uint8 CHx);
功能:在调试模式下,DMA不停止,在执行
参数: CHx: DMA_CH0~DMA_CH31//选择通道0~31

void DMA_Set_Group_Priority(uint8 CHx, uint8 Group0_Priority, uint8 Group1_Priority);
功能:In group fixed priority arbitration mode
只有两组(DMAMUX0,DMAMUX1),却可以设置4个优先级,怪怪的.
注意:通道之间的优先级不能一样,否则出错
参数: CHx: DMA_CH0~DMA_CH31//选择通道0~31
Group0_Priority: 0~3
Group1_Priority: 0~3

void DMA_Set_Channel_Priority(uint8 CHx, uint8 Priority);
功能:设置通道的优先级
参数: CHx: DMA_CH0~DMA_CH31//选择通道0~31
选择通道(根据触发源, DMAMUXx),CHx:0~31
Priority: 0~15
前提:模式选为fixed-priority固定优先级模式,CR[ERCA] = 0
数值越大,优先级越高
注意:通道之间的优先级不能一样,否则出错

uint8 DMA_Get_Channel_Priority(uint8 CHx);
功能:返回通道CHx的优先级
参数: CHx: DMA_CH0~DMA_CH31//选择通道0~31

uint8 DMA_Get_Group_Priority(uint8 CHx);
功能:返回组别(DMAMUX0,DMAMUX1)的优先级
参数: CHx: DMA_CH0~DMA_CH31//选择通道 0~31

/********** END of 函数功能介绍**********/

/********** DMA 例程 **********/

因为都是我临时写的,后期可能会改动代码,所以截图的代码不正确的,以手写代码为例程

/********** 内存到内存 **********/

/********** 同类型变量的传输 **********/

```
/********** 1个uint8传输到1个uint8 **********/
#include "all.h"
void DMA1_DMA17_IRQHandler()
{
    DMA_Clear_Int(DMA_CH1);
}
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint8  a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint8  temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 1);
    DMA_Minor_Bytes(DMA_CH1, 1);
    DMA_Source(DMA_CH1, (uint32)&a[6], 1, DMA_8BIT, -1);
    DMA_Destination(DMA_CH1, (uint32)&temp, 1, DMA_32BIT, -1);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Int_Enable(DMA_CH1, DMA_CSR_INTMAJOR_MASK);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

1. Read byte from location (uint32)&a[6].

2.Write 8-bits to location (uint32)&temp → first iteration of the minor loop.

3. Major loop complete.

```
/********** 1个uint16传输到1个uint16 **********/
#include "all.h"
void DMA1_DMA17_IRQHandler()
{
    DMA_Clear_Int(DMA_CH1);
}
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint16  a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint16  temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 1);
```

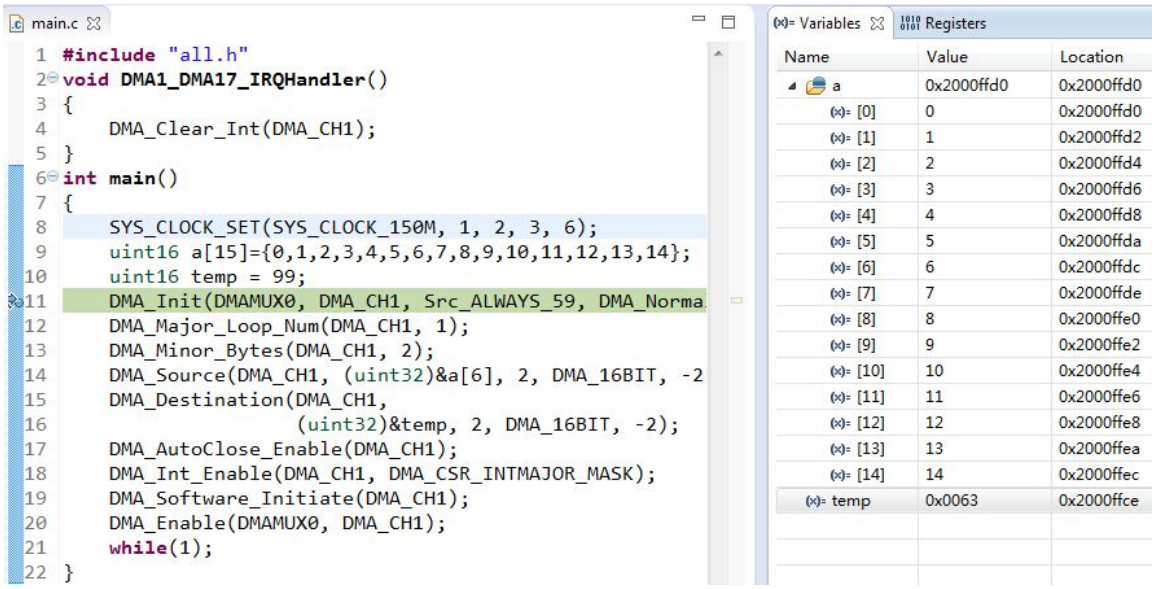
```
    DMA_Minor_Bytes(DMA_CH1, 2);
    DMA_Source(DMA_CH1, (uint32)&a[6], 2, DMA_16BIT, -2);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 2, DMA_16BIT, -2);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Int_Enable(DMA_CH1, DMA_CSR_INTMAJOR_MASK);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

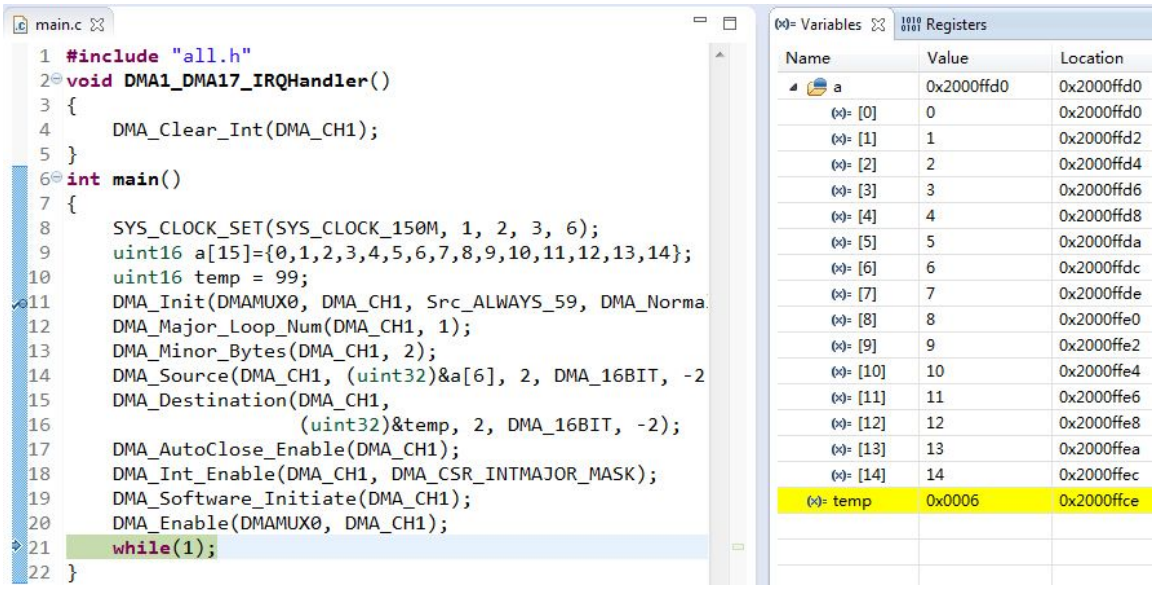
1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffdc), read byte from location (0x2000ffdc + 1).

2. Write 16-bits to location (uint32)&temp(在图中具体是0x2000ffce) → first iteration of the minor loop.

3. Major loop complete.



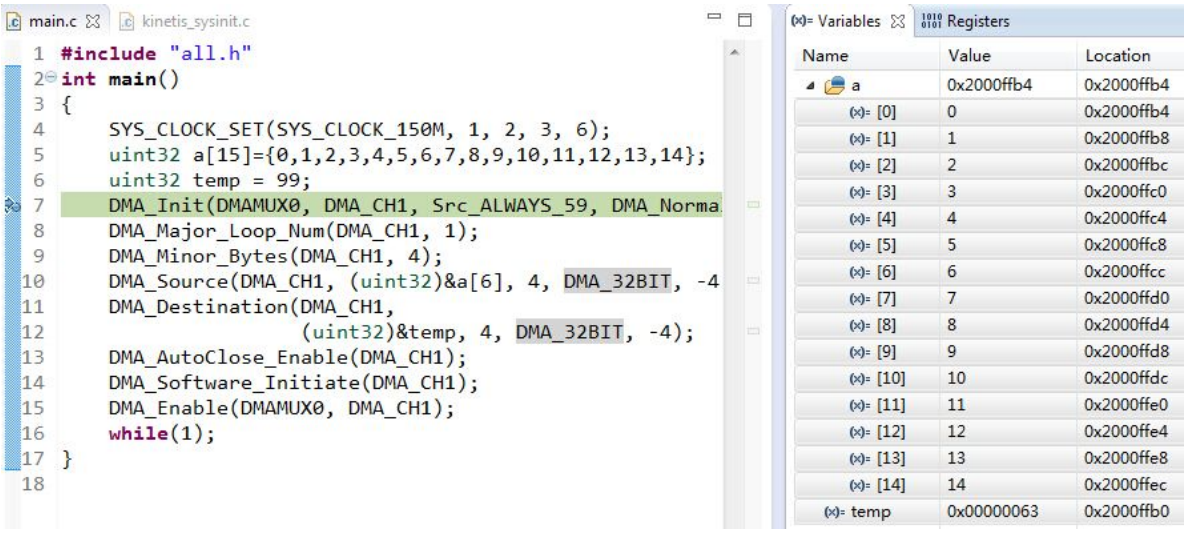
执行后



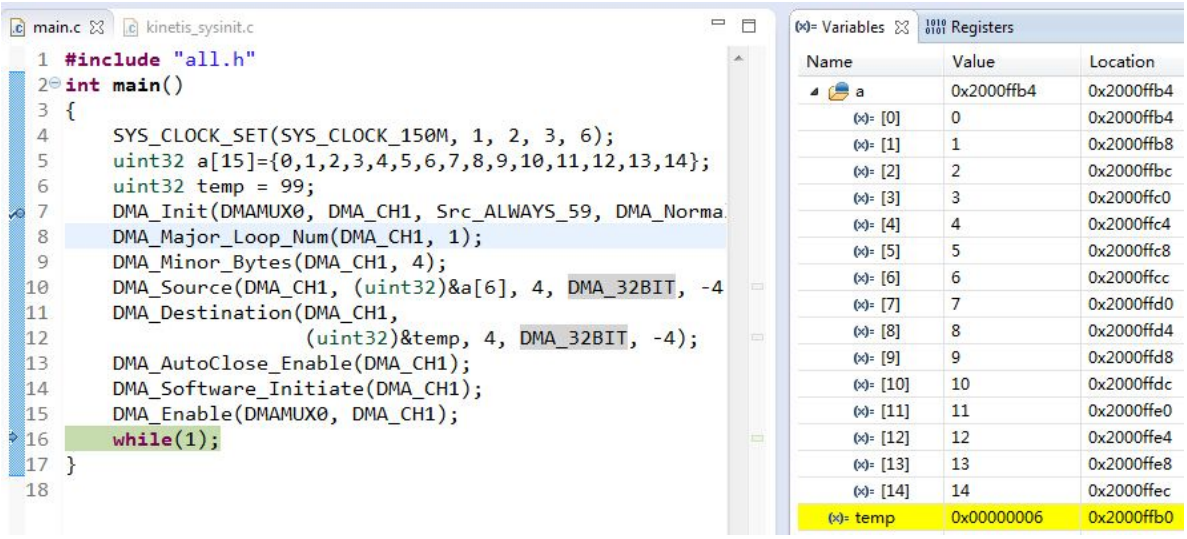
```
/***************** 1个uint32传输到1个uint32 *****************/
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 1);
    DMA_Minor_Bytes(DMA_CH1, 4);
    DMA_Source(DMA_CH1, (uint32)&a[6], 4, DMA_32BIT, -4);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_32BIT, -4);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffcc), read byte from location (0x2000ffcc + 1), read byte from
(0x2000ffcc + 2), read byte from (0x2000ffcc + 3).
2. Write 32-bits to location (uint32)&temp(在图中具体是0x2000ffb0) → first iteration of the minor loop.
3. Major loop complete.



执行后




```
/****************** 2个uint32传输到2个uint32 *****************/
//01
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 1);
    DMA_Minor_Bytes(DMA_CH1, 8);
    DMA_Source(DMA_CH1, (uint32)&a[6], 4, DMA_32BIT, -8);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_32BIT, -8);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

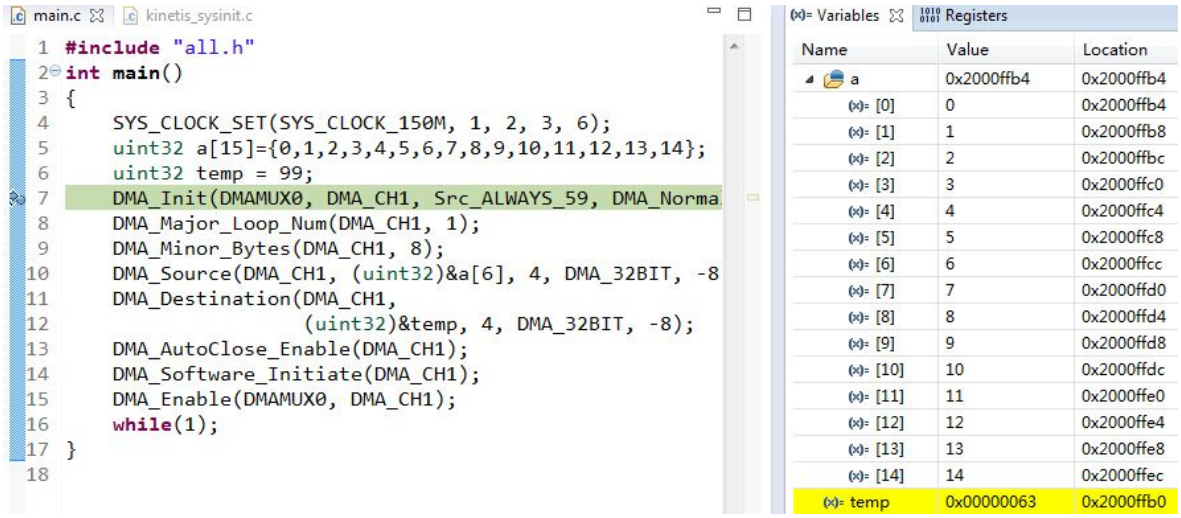
1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffcc), read byte from location (0x2000ffcc + 1), read byte from (0x2000ffcc + 2), read byte from (0x2000ffcc + 3).

2. Write 32-bits to location (uint32)&temp(在图中具体是0x2000ffb0) → first iteration of the minor loop.

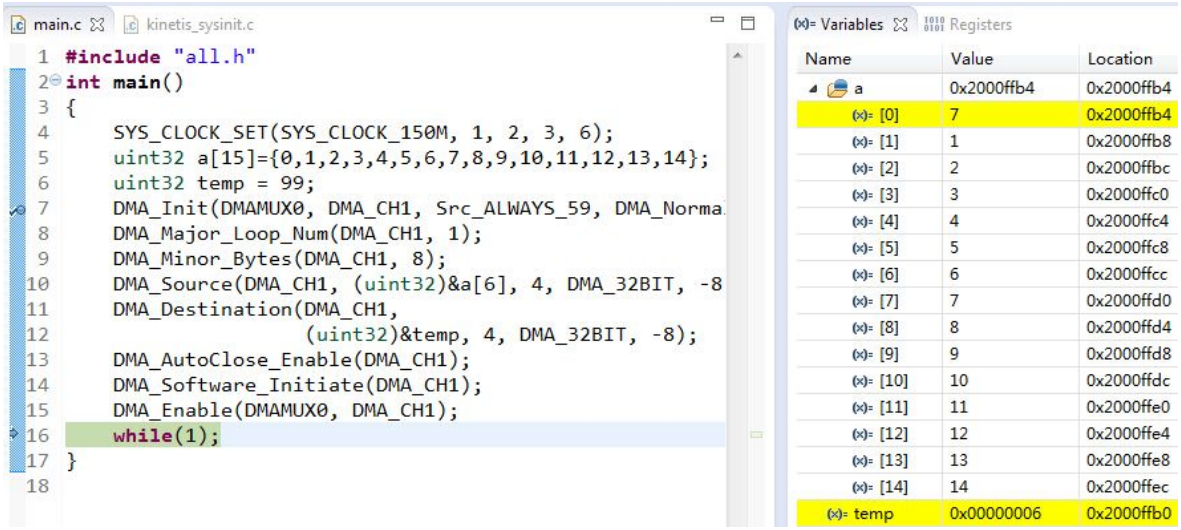
3. Read byte from location (0x2000ffcc + 4), read byte from location (0x2000ffcc + 4), read byte from (0x2000ffcc + 5), read byte from (0x2000ffcc + 6).

4. Write 32-bits to location (0x2000ffb0 + 4)→ second iteration of the minor loop.

5. Major loop complete.



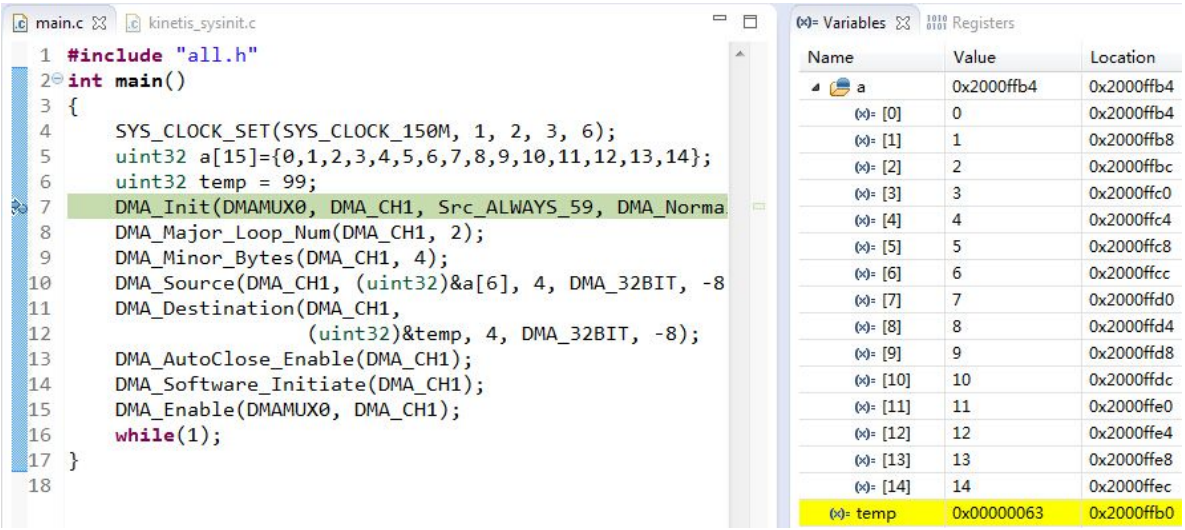
执行后



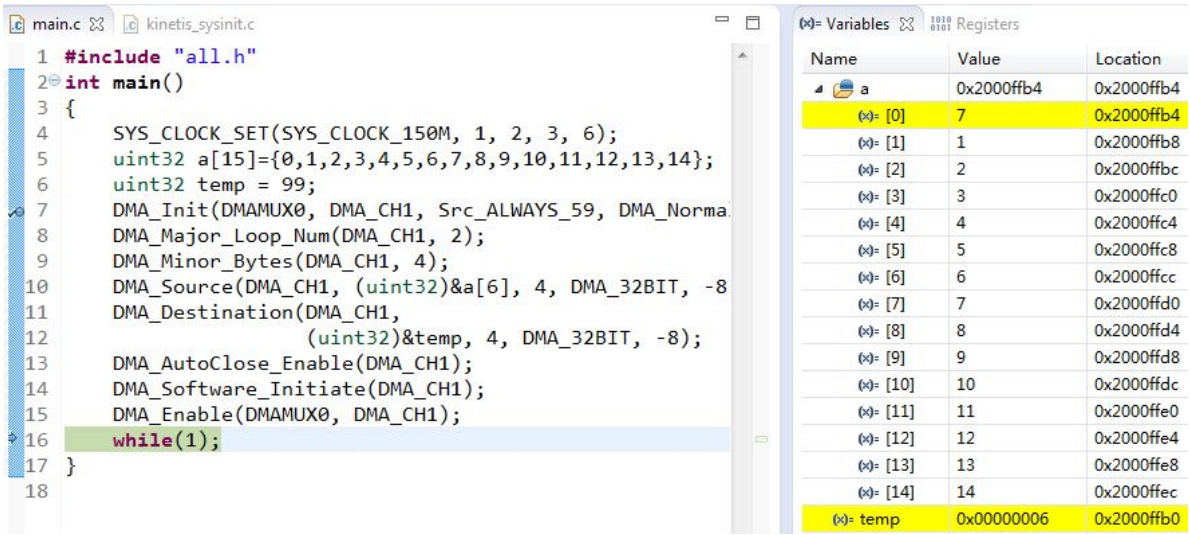

```
//02
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32  temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 2);
    DMA_Minor_Bytes(DMA_CH1, 4);
    DMA_Source(DMA_CH1, (uint32)&a[6], 4, DMA_32BIT, -8);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_32BIT, -8);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffcc), read byte from location (0x2000ffcc + 1), read byte from
(0x2000ffcc + 2), read byte from (0x2000ffcc + 3).
2. Write 32-bits to location (uint32)&temp(在图中具体是0x2000ffb0) → first iteration of the minor loop.
3. Major loop Minus one
d. Read byte from location (0x2000ffcc + 4), read byte from location (0x2000ffcc + 4), read byte from (0x2000ffcc + 5), read byte from (0x2000ffcc + 6).
4. Write 32-bits to location (0x2000ffb0 + 4)→ second iteration of the minor loop.
5. Major loop complete.



执行后




```
//03
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 1);
```

```
    DMA_Minor_Bytes(DMA_CH1, 8);
    DMA_Source(DMA_CH1, (uint32)&a[6], 8, DMA_32BIT, -16);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 8, DMA_32BIT, -16);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

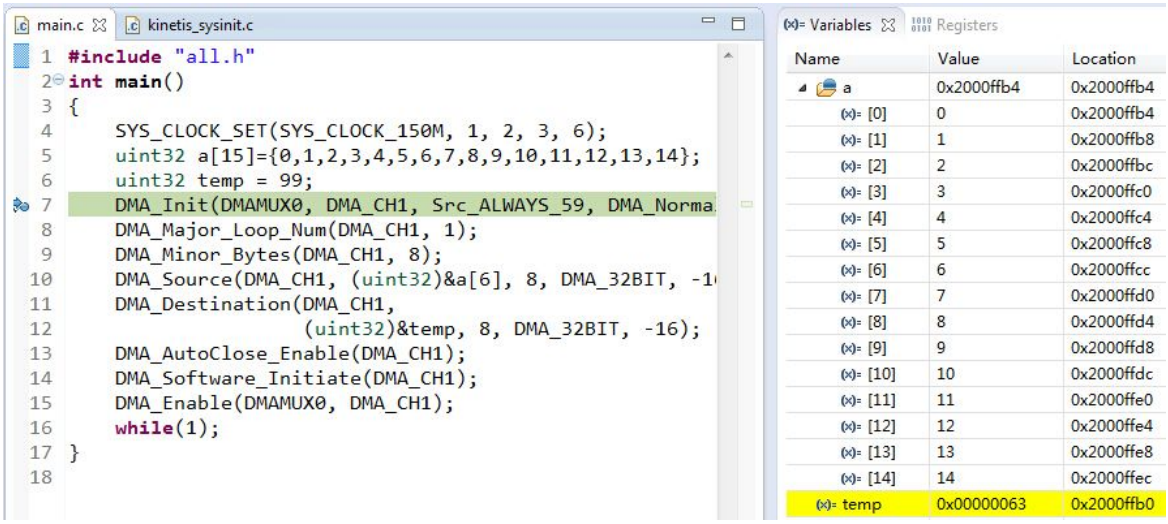
1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffcc), read byte from location (0x2000ffcc + 1), read byte from (0x2000ffcc + 2), read byte from (0x2000ffcc + 3).

2. Write 32-bits to location (uint32)&temp(在图中具体是0x2000ffb0) → first iteration of the minor loop.

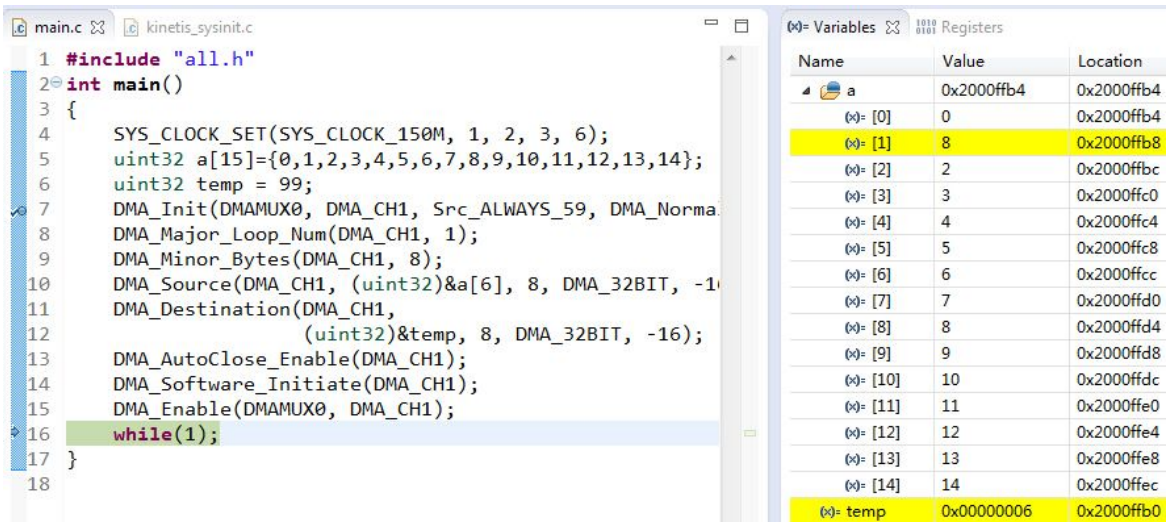
3. Read byte from location (0x2000ffcc + 8), read byte from location (0x2000ffcc + 9), read byte from (0x2000ffcc + 10), read byte from (0x2000ffcc + 11).

4. Write 32-bits to location (0x2000ffb0 + 8)→ second iteration of the minor loop.

5. Major loop complete.



执行后



```
//04
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 1);
    DMA_Minor_Bytes(DMA_CH1, 8);
    DMA_Source(DMA_CH1, (uint32)&a[6], 4, DMA_32BIT, -8);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 8, DMA_32BIT, -16);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffcc), read byte from location (0x2000ffcc + 1), read byte from (0x2000ffcc + 2), read byte from (0x2000ffcc + 3).

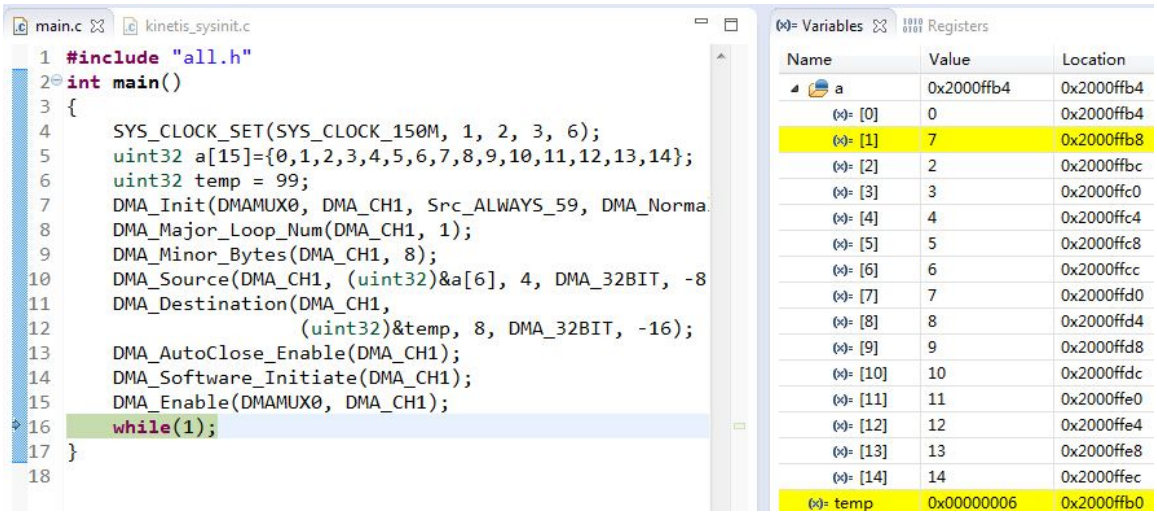
2. Write 32-bits to location (uint32)&temp(在图中具体是0x2000ffb0) → first iteration of the minor loop.

3. Read byte from location (0x2000ffcc + 4), read byte from location (0x2000ffcc + 5), read byte from (0x2000ffcc + 6), read byte from (0x2000ffcc + 7).

4. Write 32-bits to location (0x2000ffb0 + 8)→ second iteration of the minor loop.

5. Major loop complete.

执行后




```
//05
#include "all.h"
int main()
```

```
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 1);
    DMA_Minor_Bytes(DMA_CH1, 8);
    DMA_Source(DMA_CH1, (uint32)&a[6], 8, DMA_32BIT, -16);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_32BIT, -8);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffcc), read byte from location (0x2000ffcc + 1), read byte from (0x2000ffcc + 2), read byte from (0x2000ffcc + 3).

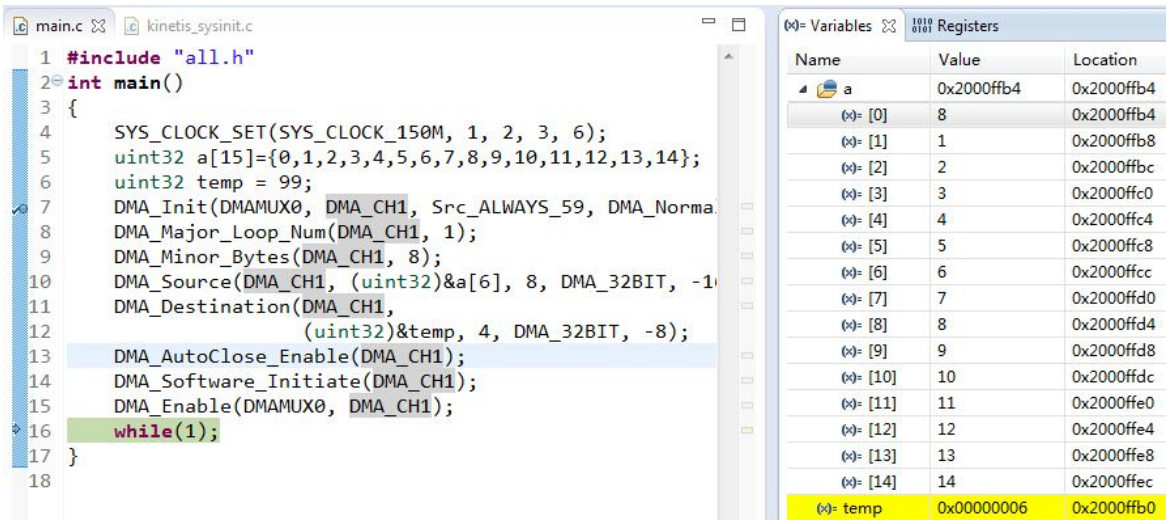
2. Write 32-bits to location (uint32)&temp(在图中具体是0x2000ffb0) → first iteration of the minor loop.

3. Read byte from location (0x2000ffcc + 8), read byte from location (0x2000ffcc + 9), read byte from (0x2000ffcc + 10), read byte from (0x2000ffcc + 11).

4. Write 32-bits to location (0x2000ffb0 + 4)→ second iteration of the minor loop.

5. Major loop complete.

执行后




```
//06
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
```

```
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 2);
    DMA_Minor_Bytes(DMA_CH1, 4);
    DMA_Source(DMA_CH1, (uint32)&a[6], 8, DMA_32BIT, -16);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 8, DMA_32BIT, -16);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffcc), read byte from location (0x2000ffcc + 1), read byte from (0x2000ffcc + 2), read byte from (0x2000ffcc + 3).

2. Write 32-bits to location (uint32)&temp(在图中具体是0x2000ffb0) → first iteration of the minor loop.

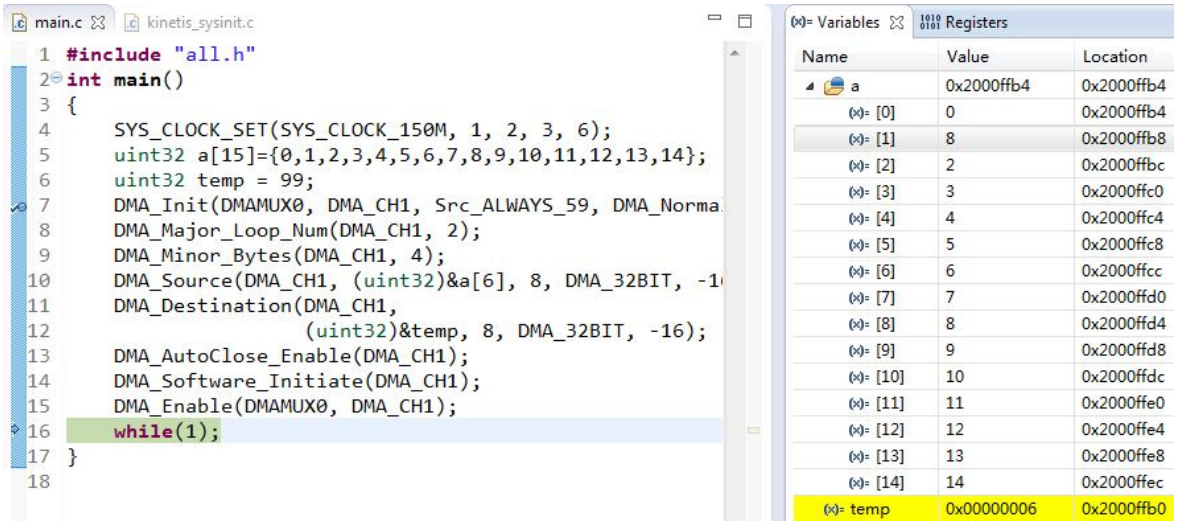
3. Major loop Minus one

4. Read byte from location (0x2000ffcc + 8), read byte from location (0x2000ffcc + 9), read byte from (0x2000ffcc + 10), read byte from (0x2000ffcc + 11).

5. Write 32-bits to location (0x2000ffb0 + 8)→ second iteration of the minor loop.

6. Major loop complete.

执行后




```
//07
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
```

```
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 2);
    DMA_Minor_Bytes(DMA_CH1, 4);
    DMA_Source(DMA_CH1, (uint32)&a[6], 4, DMA_32BIT, -8);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 8, DMA_32BIT, -16);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffcc), read byte from location (0x2000ffcc + 1), read byte from (0x2000ffcc + 2), read byte from (0x2000ffcc + 3).

2. Write 32-bits to location (uint32)&temp(在图中具体是0x2000ffb0) → first iteration of the minor loop.

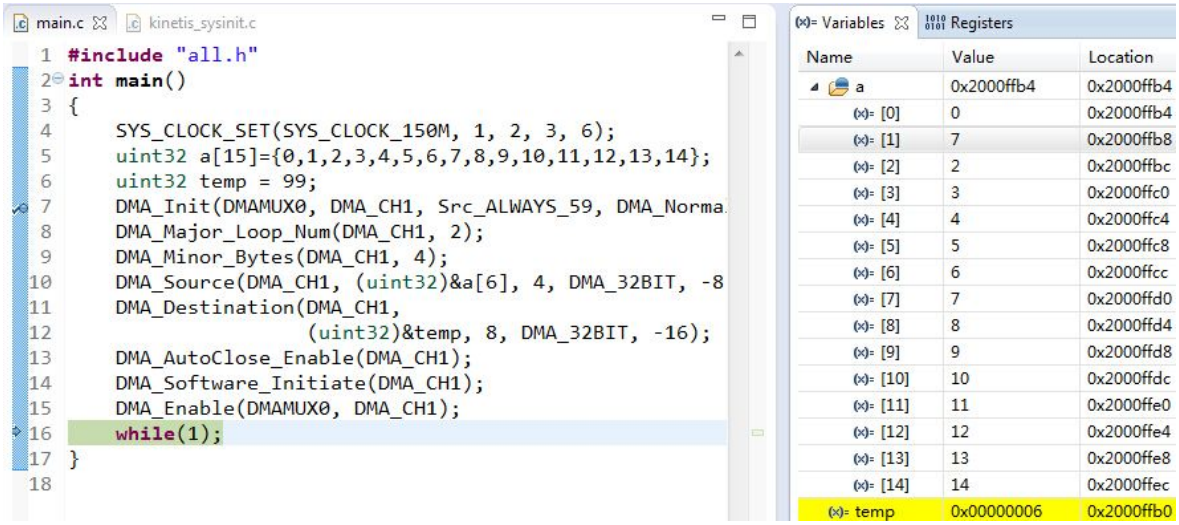
3. Major loop Minus one

4. Read byte from location (0x2000ffcc + 4), read byte from location (0x2000ffcc + 4), read byte from (0x2000ffcc + 5), read byte from (0x2000ffcc + 6).

5. Write 32-bits to location (0x2000ffb0 + 8)→ second iteration of the minor loop.

6. Major loop complete.

执行后




```
//08
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 2);
```

```
    DMA_Minor_Bytes(DMA_CH1, 4);
    DMA_Source(DMA_CH1, (uint32)&a[6], 8, DMA_32BIT, -16);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_32BIT, -8);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffcc), read byte from location (0x2000ffcc + 1), read byte from (0x2000ffcc + 2), read byte from (0x2000ffcc + 3).

2. Write 32-bits to location (uint32)&temp(在图中具体是0x2000ffb0) → first iteration of the minor loop.

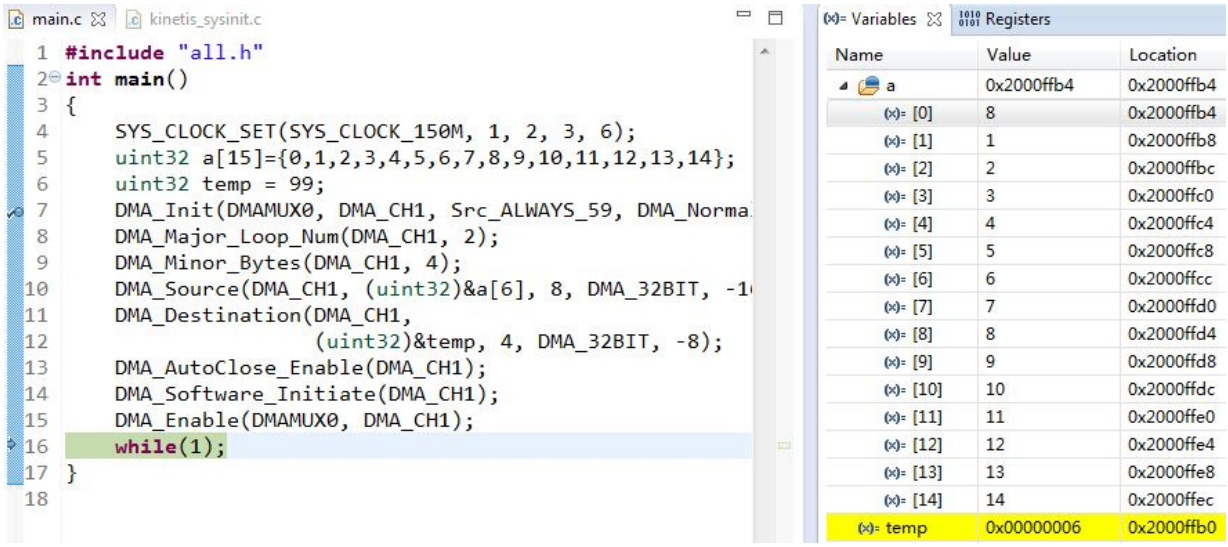
3. Major loop Minus one

d. Read byte from location (0x2000ffcc + 8), read byte from location (0x2000ffcc + 9), read byte from (0x2000ffcc + 10), read byte from (0x2000ffcc + 11).

4. Write 32-bits to location (0x2000ffb0 + 4)→ second iteration of the minor loop.

5. Major loop complete.

执行后



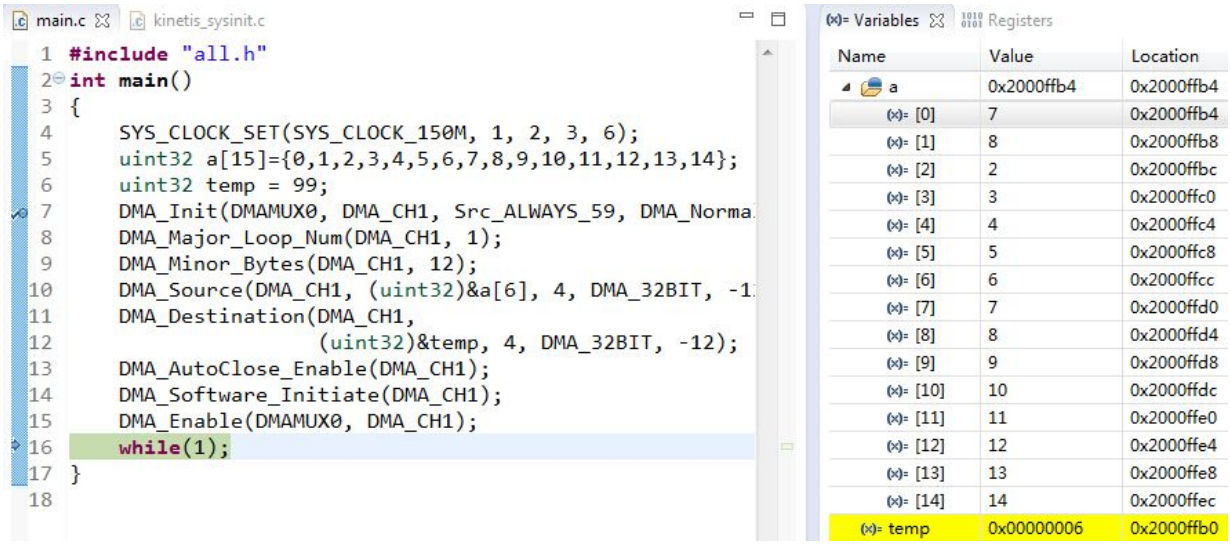

```
/****************** 3个uint32传输到3个uint32 ******************/
//01
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 1);
```

```
    DMA_Minor_Bytes(DMA_CH1, 12);
    DMA_Source(DMA_CH1, (uint32)&a[6], 4, DMA_32BIT, -12);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_32BIT, -12);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffcc), read byte from location (0x2000ffcc + 1), read byte from (0x2000ffcc + 2), read byte from (0x2000ffcc + 3).
2. Write 32-bits to location (uint32)&temp(在图中具体是0x2000ffb0) → first iteration of the minor loop.
3. Read byte from location (0x2000ffcc + 4), read byte from location (0x2000ffcc + 4), read byte from (0x2000ffcc + 5), read byte from (0x2000ffcc + 6).
4. Write 32-bits to location (0x2000ffb0 + 4)→ second iteration of the minor loop.
5. Read byte from location (0x2000ffcc + 8), read byte from location (0x2000ffcc + 9), read byte from (0x2000ffcc + 10), read byte from (0x2000ffcc + 11).
6. Write 32-bits to location (0x2000ffb0 + 8)→ second iteration of the minor loop.
7. Major loop complete.

执行后



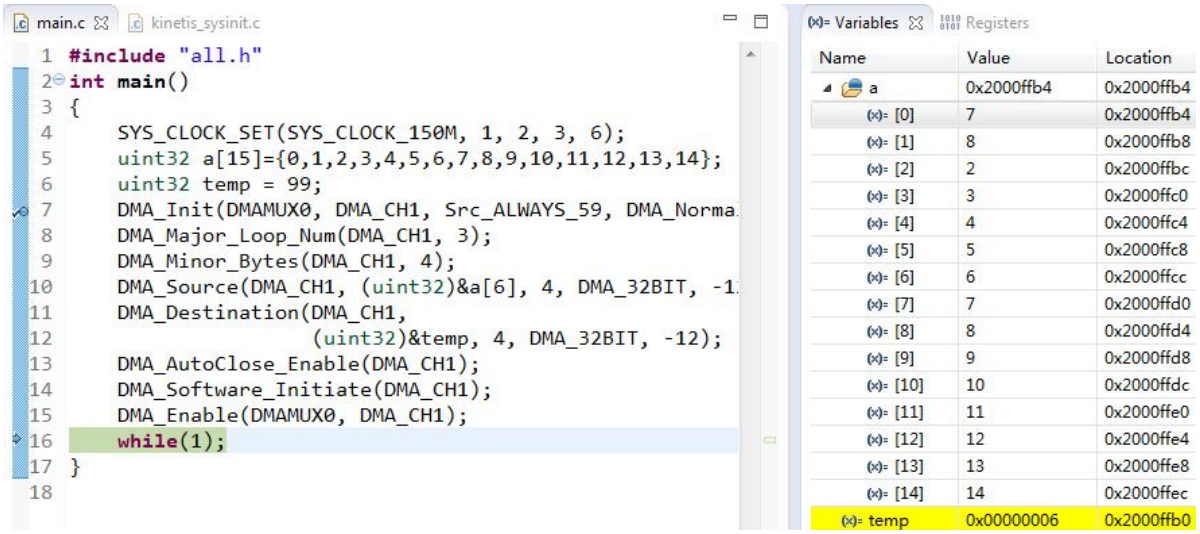

```
//02
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 3);
    DMA_Minor_Bytes(DMA_CH1, 4);
```

```
    DMA_Source(DMA_CH1, (uint32)&a[6], 4, DMA_32BIT, -12);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_32BIT, -12);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffcc), read byte from location (0x2000ffcc + 1), read byte from (0x2000ffcc + 2), read byte from (0x2000ffcc + 3).
2. Write 32-bits to location (uint32)&temp(在图中具体是0x2000ffb0) → first iteration of the minor loop.
3. Major loop Minus one.
4. Read byte from location (0x2000ffcc + 4, read byte from location (0x2000ffcc + 5), read byte from (0x2000ffcc + 6), read byte from (0x2000ffcc + 7).
5. Write 32-bits to location (0x2000ffb0 + 4)→ second iteration of the minor loop.
6. Major loop Minus one again.
7. Read byte from location (0x2000ffcc + 8), read byte from location (0x2000ffcc + 9), read byte from (0x2000ffcc + 10), read byte from (0x2000ffcc + 11).
8. Write 32-bits to location (0x2000ffb0 + 8)→ second iteration of the minor loop.
9. Major loop complete.

执行后




```
/**************** 不同类型变量的传输 *****************/
/***************** 1个uint16传输到1个uint32 *****************/
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint16 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
```

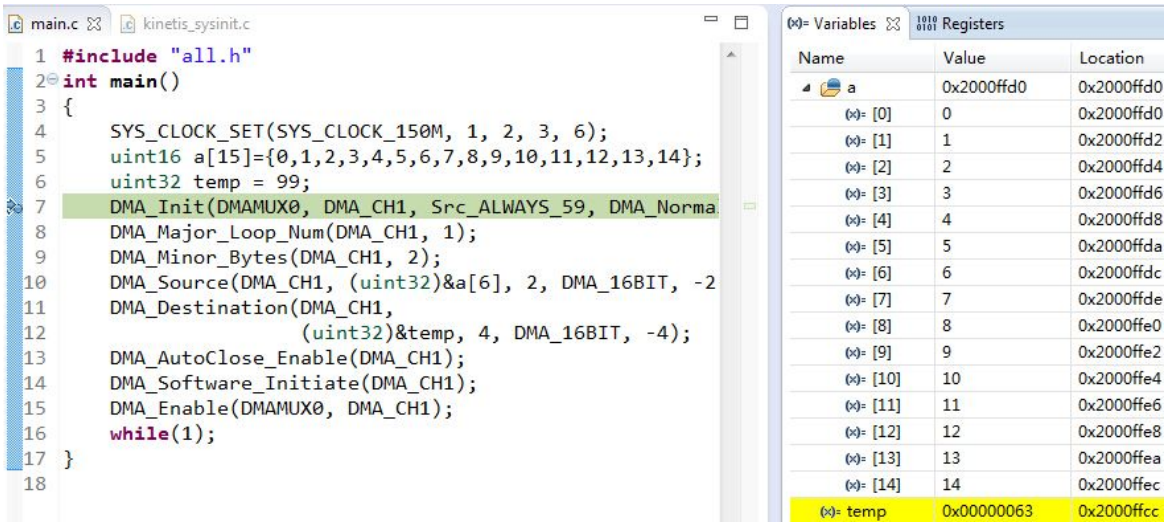
```
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 1);
    DMA_Minor_Bytes(DMA_CH1, 2);
    DMA_Source(DMA_CH1, (uint32)&a[6], 2, DMA_16BIT, -2);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_16BIT, -4);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

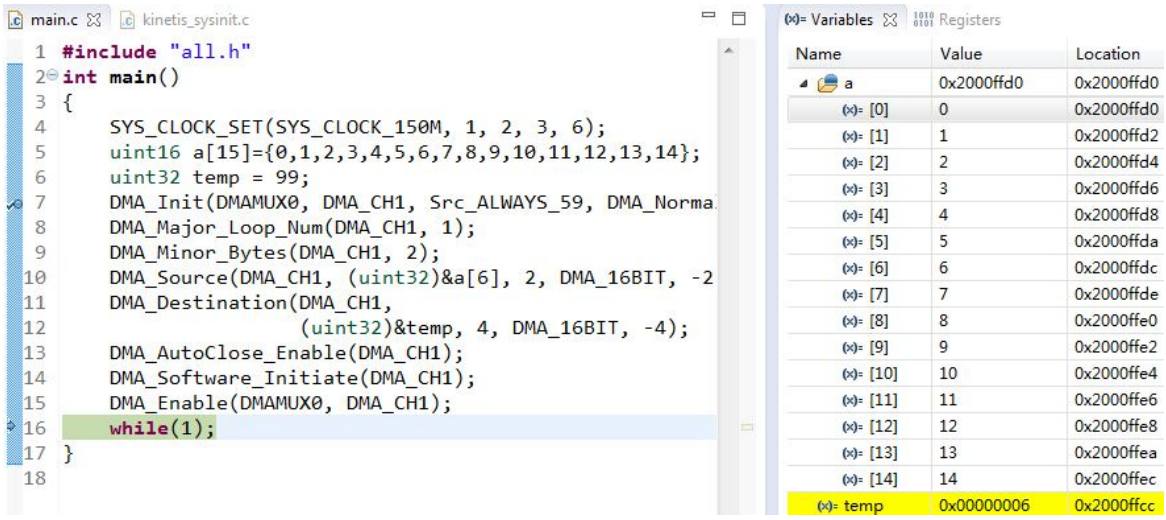
1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffdc), read byte from location (0x2000ffdc + 1).

2. Write 16-bits to location (uint32)&temp(在图中具体是0x2000ffcc) → first iteration of the minor loop.

3. Major loop complete.



执行后



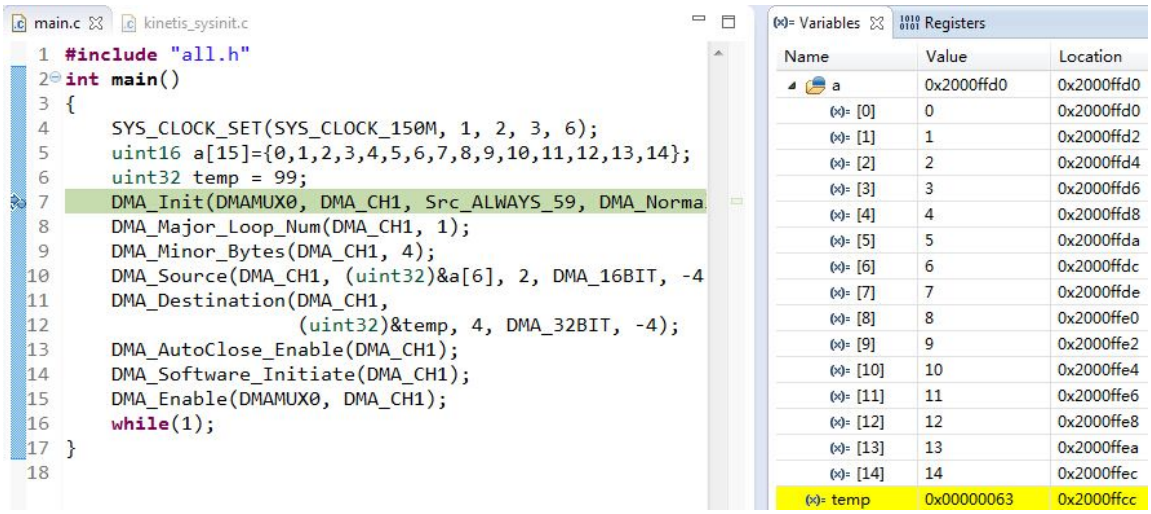
```
/**************** 2 个 uint16 传输到 1 个 uint32 *****************/
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint16 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 1);
    DMA_Minor_Bytes(DMA_CH1, 4);
    DMA_Source(DMA_CH1, (uint32)&a[6], 2, DMA_16BIT, -4);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_32BIT, -4);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

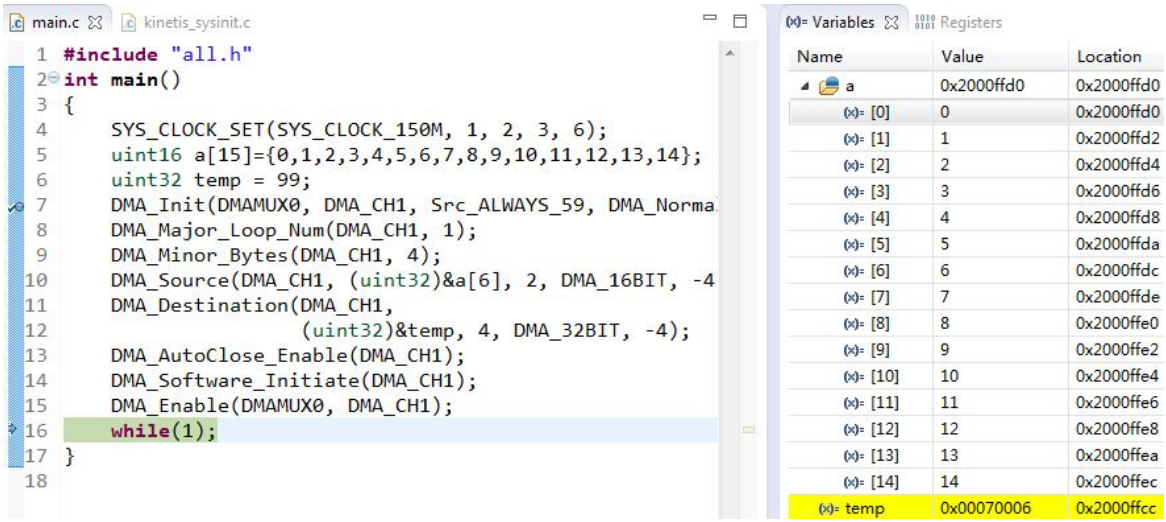
1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffdc), read byte from location (0x2000ffdc + 1), read byte from(0x2000ffdc + 2), read byte from (0x2000ffdc + 3).

2. Write 32-bits to location (uint32)&temp(在图中具体是0x2000ffcc) → first iteration of the minor loop.

3. Major loop complete.



执行后




```
/***************** 2 个 uint16 传输到 2 个 uint32 *****************/
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint16 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 1);
    DMA_Minor_Bytes(DMA_CH1, 4);
    DMA_Source(DMA_CH1, (uint32)&a[6], 2, DMA_16BIT, -4);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_16BIT, -4);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

The source-to-destination transfers are executed as follows:

1. Read byte from location (uint32)&a[6](在图中具体是0x2000ffdc), read byte from location (0x2000ffdc + 1).

2. Write 16-bits to location (uint32)&temp(在图中具体是0x2000ffcc) → first iteration of the minor loop.

3. Read byte from location (0x2000ffdc + 2), read byte from location (0x2000ffdc + 3).

4. Write 16-bits to location (0x2000ffcc + 4)→ second iteration of the minor loop.

5. Major loop complete.

main.c | kinetis_sysinit.c

```c
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint16 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Norma
    DMA_Major_Loop_Num(DMA_CH1, 1);
    DMA_Minor_Bytes(DMA_CH1, 4);
    DMA_Source(DMA_CH1, (uint32)&a[6], 2, DMA_16BIT, -4
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_16BIT, -4);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Software_Initiate(DMA_CH1);
    DMA_Enable(DMAMUX0, DMA_CH1);
    while(1);
}
```

Variables | Registers

Name	Value	Location
a	0x2000ffd0	0x2000ffd0
[0]	0	0x2000ffd0
[1]	1	0x2000ffd2
[2]	2	0x2000ffd4
[3]	3	0x2000ffd6
[4]	4	0x2000ffd8
[5]	5	0x2000ffda
[6]	6	0x2000ffdc
[7]	7	0x2000ffde
[8]	8	0x2000ffe0
[9]	9	0x2000ffe2
[10]	10	0x2000ffe4
[11]	11	0x2000ffe6
[12]	12	0x2000ffe8
[13]	13	0x2000ffea
[14]	14	0x2000ffec
temp	0x00000063	0x2000ffcc

执行后

main.c | kinetis_sysinit.c

```c
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint16 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Norma
    DMA_Major_Loop_Num(DMA_CH1, 1);
    DMA_Minor_Bytes(DMA_CH1, 4);
    DMA_Source(DMA_CH1, (uint32)&a[6], 2, DMA_16BIT, -4
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_16BIT, -4);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Software_Initiate(DMA_CH1);
    DMA_Enable(DMAMUX0, DMA_CH1);
    while(1);
}
```

Variables | Registers

Name	Value	Location
a	0x2000ffd0	0x2000ffd0
[0]	7	0x2000ffd0
[1]	1	0x2000ffd2
[2]	2	0x2000ffd4
[3]	3	0x2000ffd6
[4]	4	0x2000ffd8
[5]	5	0x2000ffda
[6]	6	0x2000ffdc
[7]	7	0x2000ffde
[8]	8	0x2000ffe0
[9]	9	0x2000ffe2
[10]	10	0x2000ffe4
[11]	11	0x2000ffe6
[12]	12	0x2000ffe8
[13]	13	0x2000ffea
[14]	14	0x2000ffec
temp	0x00000006	0x2000ffcc

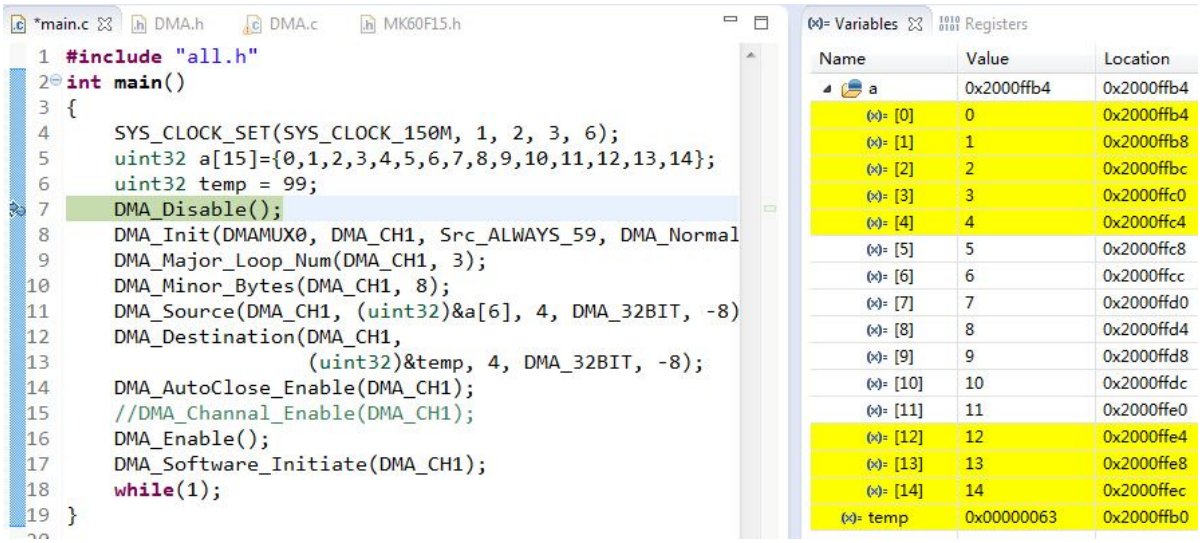
/********** 内存到寄存器 **********/

我感觉原理都是一样的, 不想写

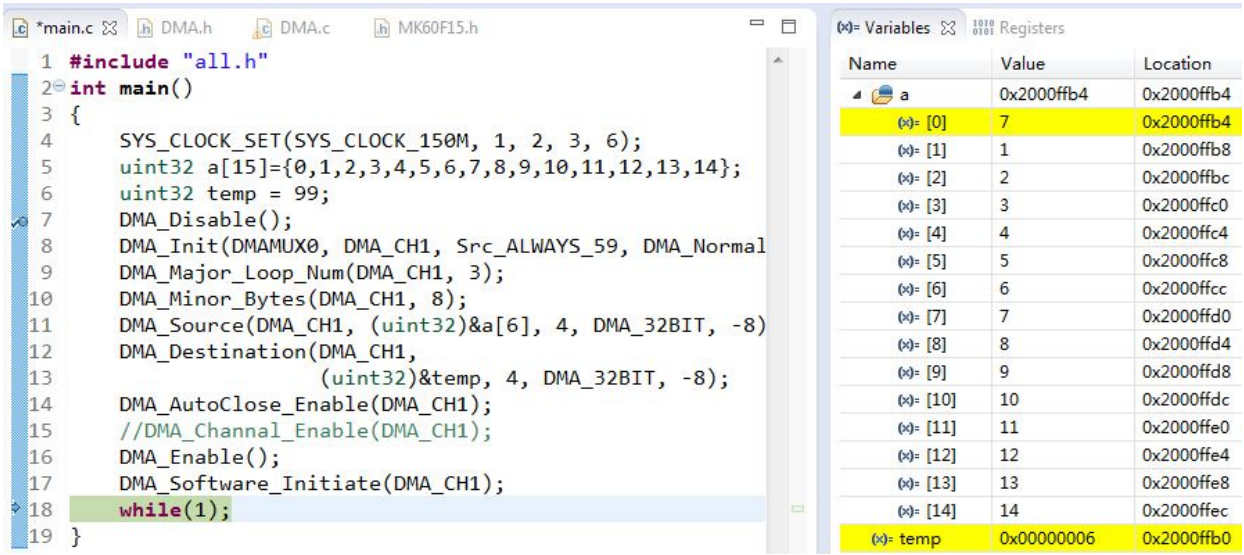
/********** DMA_Channal_Enable(DMA_CH1); *********/

```c
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 3);
    DMA_Minor_Bytes(DMA_CH1, 8);
    DMA_Source(DMA_CH1, (uint32)&a[6], 4, DMA_32BIT, -8);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_32BIT, -8);
```

```
    DMA_AutoClose_Enable(DMA_CH1);
    //DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```



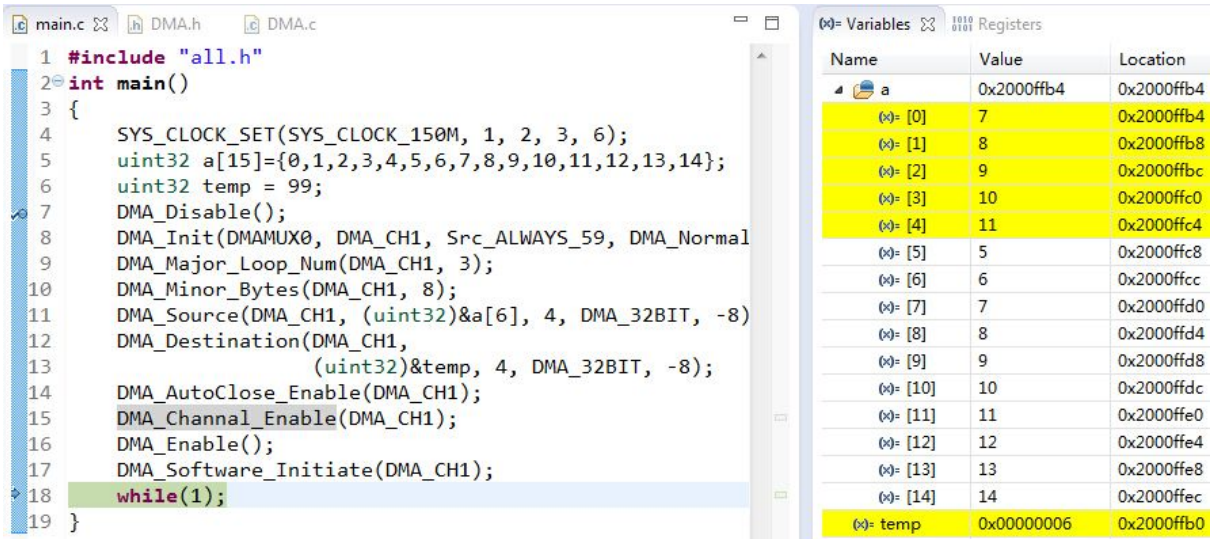
执行后



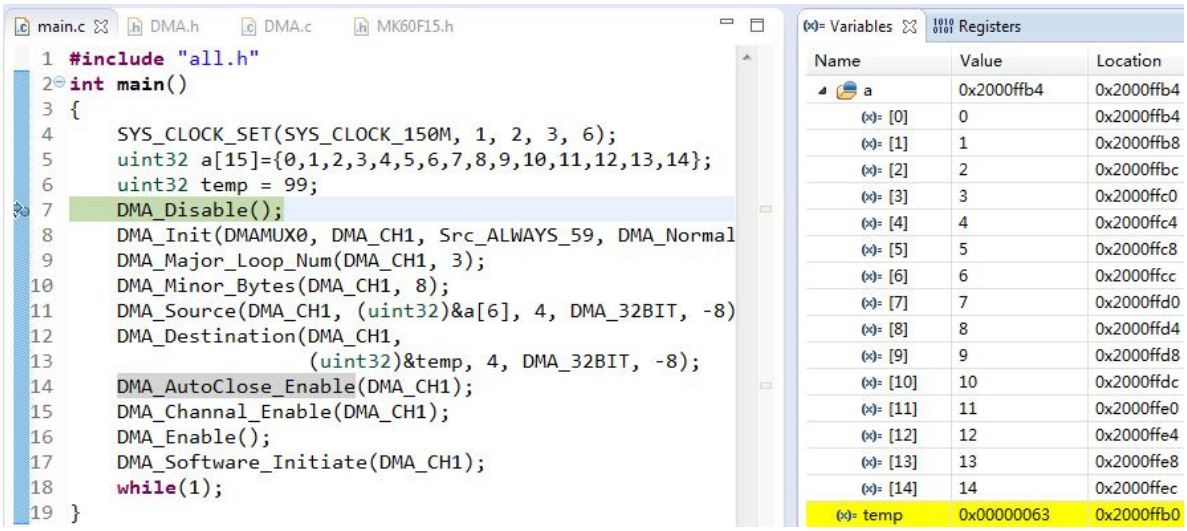
把注释去掉之后

```
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 3);
    DMA_Minor_Bytes(DMA_CH1, 8);
    DMA_Source(DMA_CH1, (uint32)&a[6], 4, DMA_32BIT, -8);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_32BIT, -8);
```

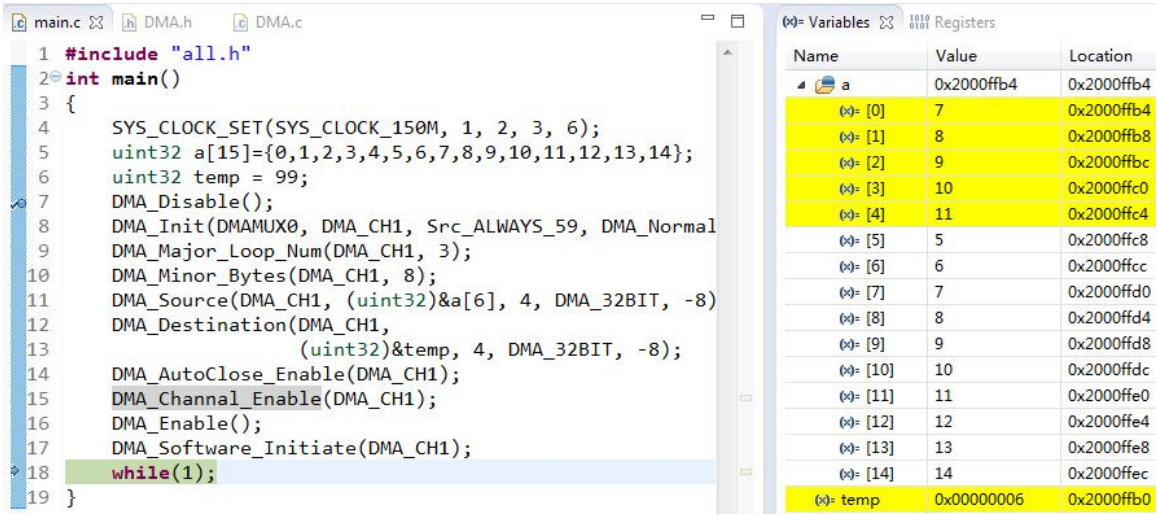
```
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```




```
/********* DMA_AutoClose_Disable(DMA_CH1);********/
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 3);
    DMA_Minor_Bytes(DMA_CH1, 8);
    DMA_Source(DMA_CH1, (uint32)&a[6], 4, DMA_32BIT, -8);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_32BIT, -8);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```



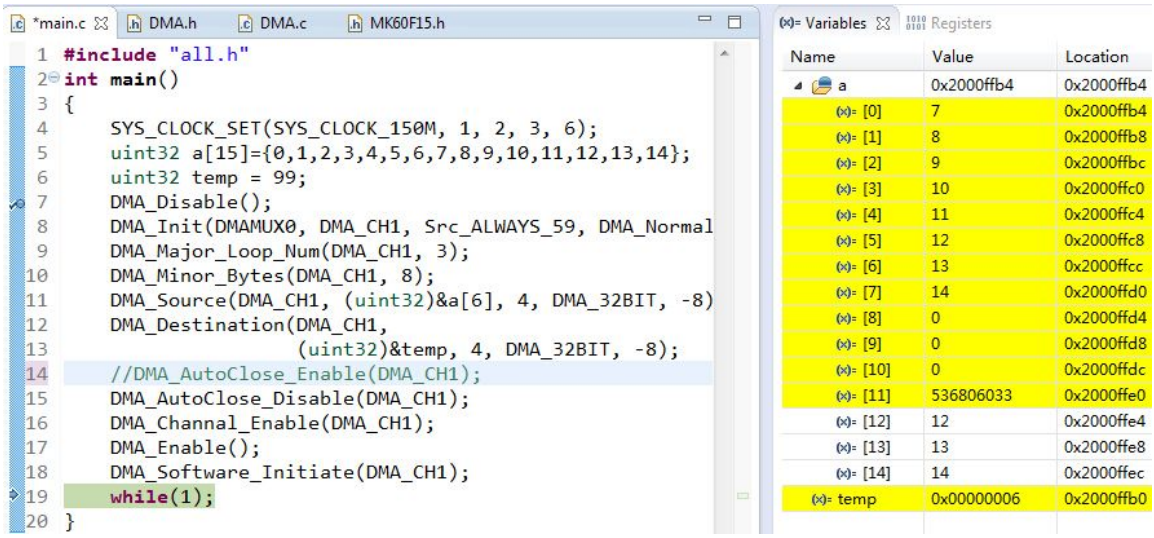
执行后



换成DMA_AutoClose_Disable(DMA_CH1);之后

```
#include "all.h"
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint32 a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint32 temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 3);
    DMA_Minor_Bytes(DMA_CH1, 8);
    DMA_Source(DMA_CH1, (uint32)&a[6], 4, DMA_32BIT, -8);
    DMA_Destination(DMA_CH1,
                    (uint32)&temp, 4, DMA_32BIT, -8);
    //DMA_AutoClose_Enable(DMA_CH1);
    DMA_AutoClose_Disable(DMA_CH1);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
```

}



/********** 中断 **********/

```
#include "all.h"
void DMA1_DMA17_IRQHandler()
{
    DMA_Clear_Int(DMA_CH1);
}
int main()
{
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    uint8  a[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    uint8  temp = 99;
    DMA_Disable();
    DMA_Init(DMAMUX0, DMA_CH1, Src_ALWAYS_59, DMA_Normal_Mode);
    DMA_Major_Loop_Num(DMA_CH1, 1);
    DMA_Minor_Bytes(DMA_CH1, 1);
    DMA_Source(DMA_CH1, (uint32)&a[6], 1, DMA_8BIT, -1);
    DMA_Destination(DMA_CH1, (uint32)&temp, 1, DMA_32BIT, -1);
    DMA_AutoClose_Enable(DMA_CH1);
    DMA_Int_Enable(DMA_CH1, DMA_CSR_INTMAJOR_MASK);
    DMA_Channal_Enable(DMA_CH1);
    DMA_Enable();
    DMA_Software_Initiate(DMA_CH1);
    while(1);
}
```

/********** END of DMA 例程 **********/

PWM

```
void FTM_PWM_Init(uint8 Mode, uint16 Ftmn_CHx_PTnx, uint32 Freq, float Duty);
void FTM_PWM_Set_Freq(uint8 Ftmn, uint8 CHx, uint32 Freq);
void FTM_PWM_Set_Duty(uint8 Ftmn, uint8 CHx, float Duty);
//中断使能函数
void FTM_TimeOut_Int_Enable(uint8 Ftmn, uint8 CHx);
void FTM_TimeOut_Int_Disable(uint8 Ftmn, uint8 CHx)
//中断服务中用到的语句
void FTM_Clear_TimeOut(uint8 Ftmn);
```

Input Capture

```
void FTM_IC_Init(uint16 Ftmn_CHx_PTnx, uint8   cfg);
//中断使能函数
void FTM_Ch_Int_Enable(uint8 Ftmn, uint8 CHx);
void FTM_Ch_Int_Dis(uint8 Ftmn, uint8 CHx);
//中断服务中用到的语句
void FTM_Clear_Channal_Event(uint8 Ftmn, uint8 CHx);
```

/********** END of 函数定义**********/

/********** PWM 函数功能介绍 **********/

Ftmn_CHx_PTnx: //选择相应模块,通道及其引脚

FTM0_CH0_PTC1	FTM1_CH0_PTA12	FTM1_CH0_PTB0
FTM0_CH0_PTA3	FTM0_CH4_PTA7	FTM1_CH1_PTA13
FTM0_CH1_PTA4	FTM0_CH4_PTD4	FTM1_CH1_PTA9
FTM0_CH1_PTC2	FTM0_CH5_PTD5	FTM1_CH1_PTB1
FTM0_CH2_PTA5	FTM0_CH5_PTA0	FTM2_CH0_PTA10
FTM0_CH2_PTC3	FTM0_CH6_PTD6	FTM2_CH0_PTB18
FTM0_CH3_PTA6	FTM0_CH6_PTA1	FTM2_CH1_PTA11
FTM0_CH3_PTC4	FTM0_CH7_PTD7	TM2_CH1_PTB19
FTM0_CH7_PTA2	FTM1_CH0_PTA8	
FTM3_CH0_PTD0	FTM3_CH0_PTE5	
FTM3_CH1_PTD1	FTM3_CH1_PTE6	
FTM3_CH2_PTD2	FTM3_CH2_PTE7	
FTM3_CH3_PTD3	FTM3_CH3_PTE8	
FTM3_CH4_PTC8	FTM3_CH4_PTE9	
FTM3_CH5_PTC9	FTM3_CH5_PTE10	
FTM3_CH6_PTC10	FTM3_CH6_PTE11	
FTM3_CH7_PTC11	FTM3_CH7_PTE12	

void FTM_PWM_Init(uint8 Mode, uint16 Ftmn_CHx_PTnx, uint32 Freq, float Duty);

功能:设置 PWM 的模式是边沿对齐,还是中间对齐，选择 FTM 的哪一个模块,哪个通道及其对应的引脚,设置 PWM 的频率，设置占空比

参数:Mode :　　EPWM_MODE,CPWM_MODE//中间对齐模式,边沿对齐模式
　　Ftmn_CHx_PTnx: 在上面
　　Freq:　　//选择 PWM 的频率

	CPWM	EPWM
建议:	5~ 375000	10~7500000
波形开始失真:	375~2000000	7500000~2500000

　　Duty:　　0~1

void FTM_PWM_Set_Freq(uint8 Ftmn, uint8 CHx, uint32 Freq);
功能:选择 FTM 的哪一个模块,哪个通道
　　设置 PWM 的频率
参数: Ftmn:　　FTM0, FTM1, FTM2, FTM3//选择 FTM 哪一个模块
　　CHx:　　FTM_CH0~FTM_CH7//选择 FTM 的通道,有 0~7

void FTM_PWM_Set_Duty(uint8 Ftmn, uint8 CHx, float Duty);
功能:选择 FTM 哪一个模块,哪个通道
　　设置占空比
参数: Ftmn:　　FTM0, FTM1, FTM2, FTM3//选择 FTM 哪一个模块
　　CHx:　　FTM_CH0~FTM_CH7//选择 FTM 的通道,有 0~7
　　Duty:　　0~1

void FTM_TimeOut_Int_Enable(uint8 Ftmn, uint8 CHx);
功能:使能 PWM 溢出中断, 每个 PWM 周期进入一次中断
参数: Ftmn:　　FTM0, FTM1, FTM2, FTM3//选择 FTM 哪一个模块
　　CHx:　　FTM_CH0~FTM_CH7//选择 FTM 的通道,有 0~7

void FTM_Clear_TimeOut(uint8 Ftmn);
功能:清除 PWM 溢出中断标志位,用在 PWM 的溢出中断服务函数中第一条语句
参数: Ftmn:　　FTM0, FTM1, FTM2, FTM3//选择 FTM 哪一个模块

/********** END of PWM 函数定义**********/

/********** PWM 例程 **********/

现象:先生成一个中间对齐,频率为 100Hz,占空比为 0.5 的方波
　　然后变为频率为 1000Hz,占空比为 0.4 的方波
　　再然后,每个 PWM 周期改变占空比(在 PWM 周期溢出中断服务函数中改变占空比)

```
#include"all.h"
uint32 busclk = 75000000;
uint8 i=3;
void FTM1_IRQHandler()
{
    FTM_Clear_TimeOut(FTM1);
```

```
    FTM_PWM_Set_Duty(FTM1, FTM_CH0, (float)(i)/100);
    i = i + 2;
    if(i>95)i=3;
}
void main(void)
{
      SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
      FTM_PWM_Init(CPWM_MODE, FTM1_CH0_PTB0, 100, 0.5);
      FTM_PWM_Set_Duty(FTM1, FTM_CH0, 0.4);
      FTM_PWM_Set_Freq(FTM1, FTM_CH0,1000);
      FTM_TimeOut_Int_Enable(FTM1, FTM_CH0);
      while(1);
}
```

/********** **END of PWM 例程** **********/

/********** **Input Capture 函数功能介绍** **********/

void FTM_IC_Init(uint16 Ftmn_CHx_PTnx, uint8 cfg);

功能:选择 FTM 的哪一个模块,哪个通道及其对应的引脚
选择触发事件是上升沿,还是下降沿,还是双边沿

参数:Ftmn_CHx_PTnx: 在上面
cfg: **FTM_Rising,FTM_Falling,FTM_Rising_or_Falling** //上升沿触发,下降沿触发,双边沿触发

void FTM_Ch_Int_Enable(uint8 Ftmn, uint8 CHx);

功能:使能 IC 通道事件中断,每捕捉到一次事件(上升沿,下降沿,双边沿)进入一次中断

参数: Ftmn: **FTM0, FTM1, FTM2, FTM3**//选择 FTM 哪一个模块
CHx: **FTM_CH0~FTM_CH7**//选择 FTM 的通道,有 0~7

void FTM_Clear_Channal_Event(uint8 Ftmn, uint8 CHx);

功能:清除 IC 通道事件中断标志,用在 IC 捕捉到相应沿的中断服务函数中第一条语句

参数: Ftmn: **FTM0, FTM1, FTM2, FTM3**//选择 FTM 哪一个模块
CHx: **FTM_CH0~FTM_CH7**//选择 FTM 的通道,有 0~7

/********** **END of Input Capture 函数定义** **********/

/********** **Input Capture 例程** **********/

现象: 测得 **Frequency** 为 10000Hz

```
#include"all.h"
/**********  Input Capture 的变量  **********/
//没函数发生器,所以还没测试 10Hz 以下的,2MHz 以上的.10Kz 到 2MHz 的精度达到 99%
//连接 B0,C2.既可测试,B0 作为 PWM 输出,C2 捕捉
```

```
uint32   current=0, original=0,Freq[10],tick = 0,Frequency;
uint32 FTM_TimerOut_flag = 0,QuTou_QuWei_NUM = 4;
uint32 busclk = 75000000;
uint32 Fre_Frequece_Index=0;                    //用于统计序列
/*****************************************************************/
uint32 average(uint32 p[10]) //功能：去毛刺求平均值
{
    int i = 0,j = 0;
    uint32 sum = 0, temp = 0;
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 10 - i; j++) {
            if (p[j] > p[j + 1]) {
                temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
    for (i = QuTou_QuWei_NUM; i < (10 - QuTou_QuWei_NUM);i++)
        sum = sum + p[i];
    return sum / (10 - 2*QuTou_QuWei_NUM);
}
//the maximum frequency for the channel input signal to be
//detected correctly is system clock divided by 4,
void FTM0_IRQHandler()
{
     //FTM0_SC &= ~FTM_SC_TOF_MASK;       //清除溢出中断标志位
     FTM_Clear_Channal_Event(FTM0, FTM_CH1);
    current = FTM0_C1V;
    // 当两次中断之间的时间差超过了一个计数周期时，需要补加 FTM_CountOut_flag 个 0xFFFF 值
     if(FTM_TimerOut_flag >= 1)
     {
        tick = (0xFFFF - original + (FTM_TimerOut_flag-1)*0xFFFF + current);
        FTM_TimerOut_flag = 0;
    }
    // 若两次中断的响应在一个周期内，则直接用前一次的计数器值减去后一次计数器值
     else
        tick =(current-original);
     Freq[Fre_Frequece_Index] = busclk / tick;
     Fre_Frequece_Index++;
     if(Fre_Frequece_Index >= 10)
     {
       Frequency = average(Freq);
       Fre_Frequece_Index = 0;
     }
     original=current;
}
void main(void)
{
```

```
    SYS_CLOCK_SET(SYS_CLOCK_150M, 1, 2, 3, 6);
    FTM_PWM_Init(EPWM_MODE, FTM1_CH0_PTB0, 10000, 0.5);
    FTM_IC_Init(FTM0_CH1_PTC2, FTM_Falling);
    FTM_Ch_Int_Enable(FTM0, FTM_CH1);
   while(1)
   {
       if(FTM0_SC & FTM_SC_TOF_MASK)
       {
           FTM0_SC &= ~FTM_SC_TOF_MASK;
           FTM_TimerOut_flag++;
       }
   }
}
/******************* END of Input Capture 例程 *********************/
```