Taller de Sistemas Embebidos
STM32 MCU – Direct Memory Access (DMA)

Información relevante

Taller de Sistemas Embebidos

Asignatura correspondiente a la **actualización 2023** del Plan de Estudios 2020 y resoluciones modificatorias, de Ingeniería Electrónica de FIUBA

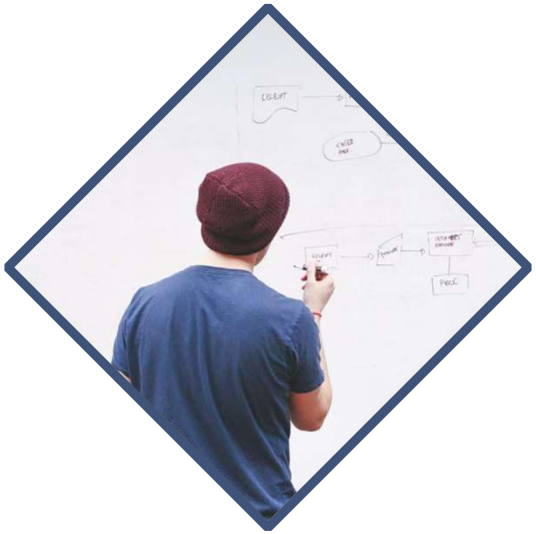
Estructura Curricular de la Carrera

El **Proyecto Intermedio** se desarrolla en la asignatura **Taller de Sistemas Embebidos**, la cual tiene un enfoque centrado en la **práctica propia de la carrera** más que en el desarrollo teórico disciplinar, con eje en la **participación de las y los estudiantes**

Más información . . .

. . . sobre la **actualización 2023** . . . https://www.fi.uba.ar/grado/carreras/ingenieria-electronica/plan-de-estudios

. . . sobre el **Taller de Sistemas Embebidos** . . . https://campusgrado.fi.uba.ar/course/view.php?id=1217



¡Hola!

Soy Juan Manuel Cruz
Taller de Sistemas Embebidos
Consultas a: jcruz@fi.uba.ar

1

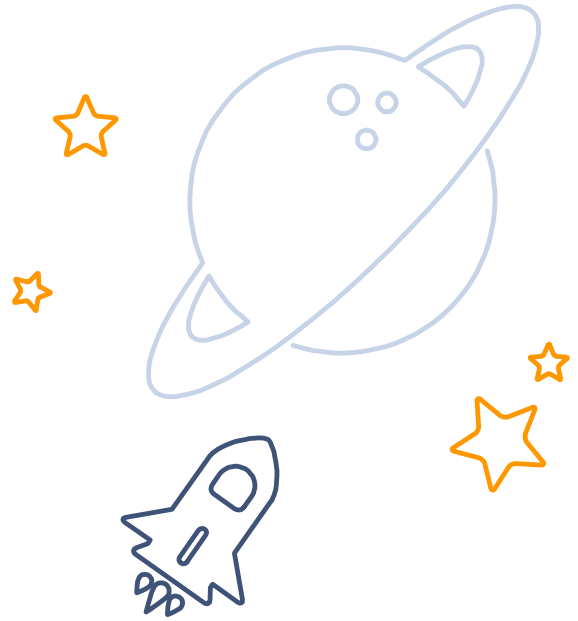
Introducción

Actualización 2023 del Plan de Estudios 2020 y resoluciones . . .

Conceptos básicos

- Referencia:
  - STM32 Arm Programming for Embedded Systems, Using C Language with STM32 Nucleo - Muhammad Ali Mazidi (Author), Shujen Chen (Author), Eshragh Ghaemi (Author)
  - Chaapter 13: DMA Programming
    - Este capítulo examina el concepto de DMA (Acceso Directo a Memoria) y muestra cómo programarlo.
    - Los microcontroladores Arm implementan DMA de forma diferente, según los distintos proveedores y pueden ser muy complejos.
    - Este capítulo es simplemente una introducción.

Solución Adecuada



. . . lo más simple posible, previa determinación del objetivo de excelencia a cumplir, obviamente contando con la documentación debida y recurriendo a la metodología de trabajo adecuada

2

Documentación debida

1er Cuatrimestre de 2024, dictado por primera vez . . .

“

STM32 Arm Programming for Embedded Systems, Using C Language with STM32 Nucleo - Muhammad Ali Mazidi (Author), Shujen Chen (Author), Eshragh Ghaemi (Author)

Chaapter 13: DMA Programming

Concept of DMA

- En las computadoras, a menudo existe la necesidad de transferir una gran cantidad de bytes de datos entre la memoria y los periféricos, como las unidades de disco.
  - En tales casos, usar el microprocesador para transferir los datos es demasiado lento ya que los datos primero deben recuperarse en la CPU y luego enviarse a su destino.
  - Además, el proceso de buscar y decodificar las instrucciones que mueven los datos aumenta la sobrecarga.
  - Por esta razón, en la mayoría de las computadoras y microcontroladores existe un DMAC (Controlador de Acceso Directo a Memoria), cuya función es puentear a la CPU y proporcionar una conexión directa entre los periféricos y la memoria, transfiriendo así los datos lo más rápido posible.

Concept of DMA

- Un problema con el uso de DMA es que sólo hay un conjunto de buses (un conjunto de cada bus: bus de datos, bus de direcciones, bus de control) en una computadora determinada y ningún bus puede dar servicio a dos maestros al mismo tiempo.
  - Los buses pueden ser utilizados por la CPU principal o por el DMAC (Controlador DMA).
  - Dado que la CPU tiene el control principal sobre los buses, debe otorgar permiso a DMAC para usarlos.

Concept of DMA

- ¿Cómo se hace esto?
  - La respuesta es que cada vez que el DMAC necesita usar los buses para transferir datos, envía una señal llamada HOLD a la CPU y la CPU responderá enviando de vuelta la señal HLDA (reconocimiento de retención) para indicarle al DMAC que puede continuar. adelante y use los buses.
  - Mientras el DMAC usa los buses para transferir datos, la CPU se mantiene alejada de los buses y, a la inversa, cuando la CPU usa el bus, el DMA está inactivo.
  - Después de que DMAC termine de transferir los datos, hará que HOLD baje y luego la CPU recuperará el control sobre los buses. Vea la Figura 13-1.

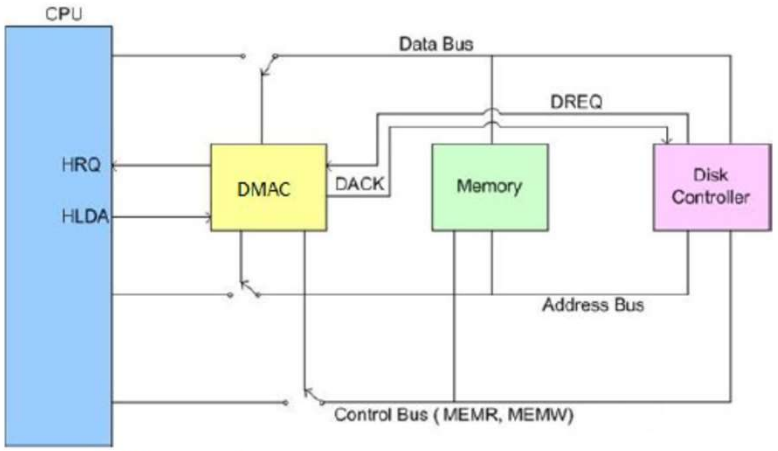


Figure 13- 1: DMA Usage of System Bus

Concept of DMA

- Por ejemplo, si el DMAC va a transferir un bloque de datos desde la memoria a un dispositivo de E/S como un controlador de disco, debe conocer la dirección del comienzo del bloque de datos (dirección del primer byte de datos) y el número de bytes (cuenta) que necesita transferir. Luego seguirá los siguientes pasos:
  - 1. El dispositivo periférico (como el controlador de disco) solicitará el servicio de DMA elevando DREQ (solicitud de DMA).
  - 2. El DMAC pondrá un alto en su HRQ (solicitud de retención), indicando a la CPU a través de su pin HOLD que necesita usar los buses.

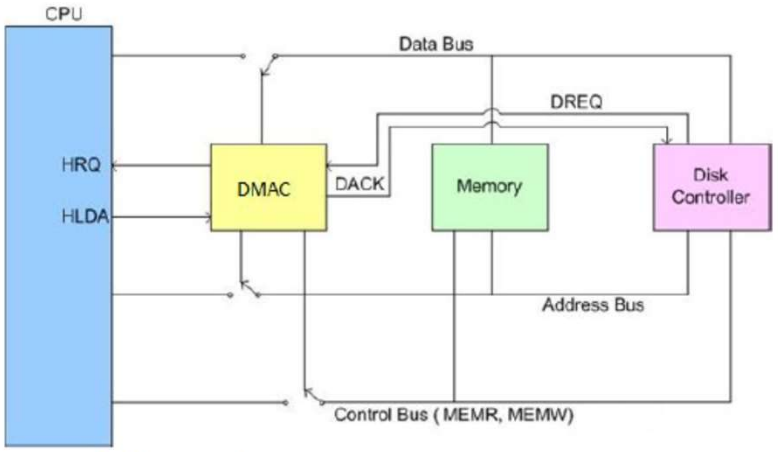


Figure 13- 1: DMA Usage of System Bus

Concept of DMA

- 3. La CPU finalizará el ciclo de bus actual y responderá a la solicitud de DMAC poniendo alto su HLDA (reconocimiento de retención), indicando así al DMAC que puede seguir adelante y utilizar los buses para realizar su tarea. HOLD debe permanecer activo alto mientras DMAC esté realizando su tarea.
- 4. DMAC activará DACK (reconocimiento DMA), que le indica al dispositivo periférico que comenzará a transferir los datos.

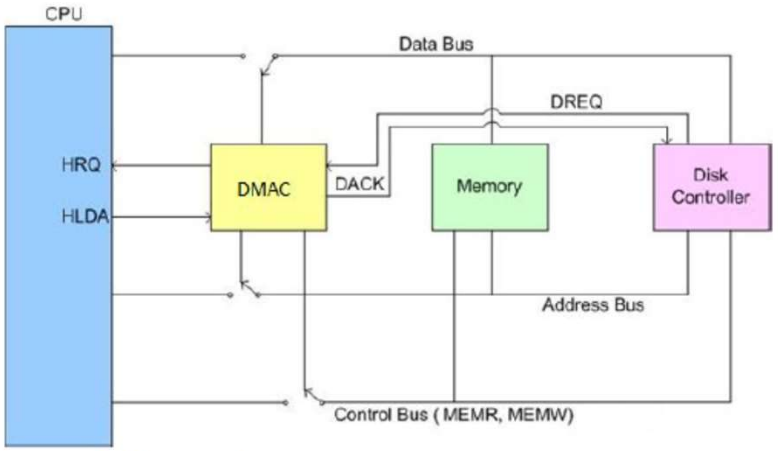


Figure 13- 1: DMA Usage of System Bus

Concept of DMA

- 5. DMAC comienza a transferir los datos de la memoria al periférico poniendo la dirección del primer byte del bloque en el bus de direcciones y activando MEMR, leyendo así el byte de la memoria en el bus de datos; luego cambia la dirección para que sea el registro de datos del periférico y activa MEMW para escribir los datos en el periférico.
  - Luego, DMAC disminuye el contador e incrementa el puntero de dirección y repite este proceso hasta que la cuenta llega a cero y la tarea finaliza.

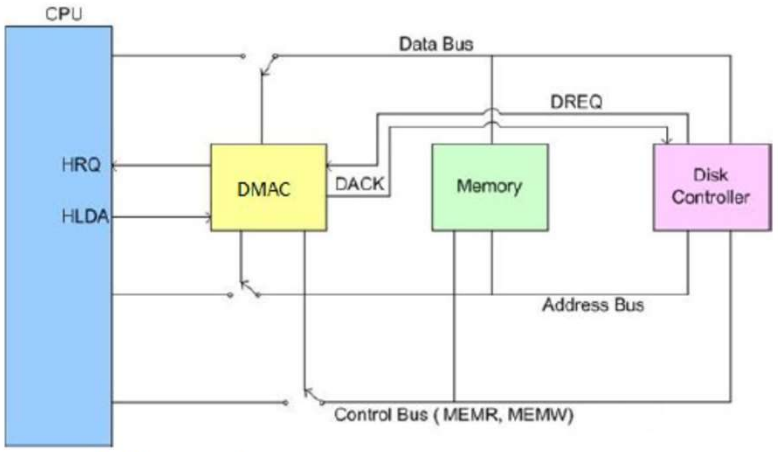


Figure 13- 1: DMA Usage of System Bus

Concept of DMA

- 6. Una vez que el DMA haya terminado su trabajo, desactivará HRQ, indicando a la CPU que puede recuperar el control sobre sus buses.
  - Esta discusión anterior indica que DMAC solo puede transferir datos; a diferencia de la CPU, no puede decodificar ni ejecutar instrucciones.
  - Se podría considerar el DMAC como una especie de CPU sin el circuito lógico decodificador/ejecutor de instrucciones.
  - Para que el DMAC pueda transferir datos, está equipado con señales de bus de direcciones, bus de datos y bus de control.

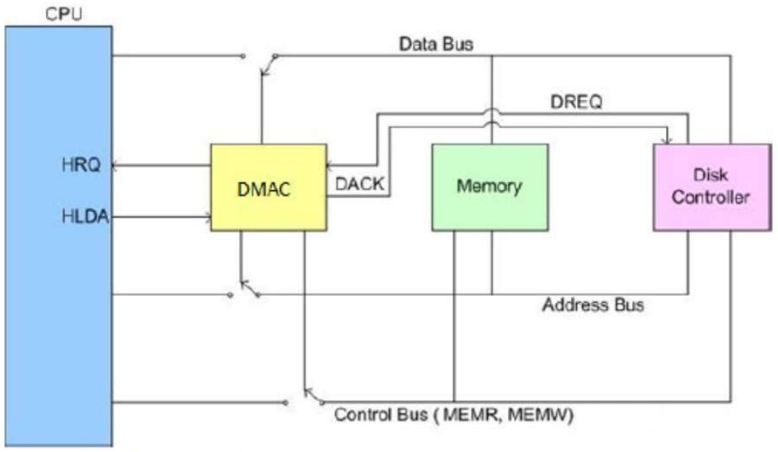


Figure 13- 1: DMA Usage of System Bus

Concept of DMA

- Las señales de intercambio entre el periférico y el DMAC o las señales entre el DMAC y la CPU se describen aquí para aclarar las operaciones.
  - Sus actividades son transparentes para el programador.
  - Como verá en los ejemplos de programación más adelante, una vez que DMAC está configurado y habilitado correctamente, la solicitud de DMA, el reconocimiento y la solicitud de bus, el reconocimiento son manejados por el hardware sin la intervención del software.

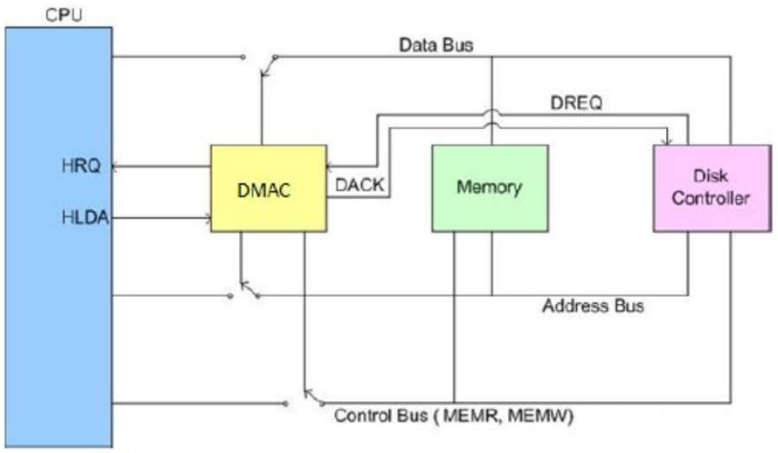


Figure 13- 1: DMA Usage of System Bus

Concept of DMA

- En este ejemplo de controlador de disco, tiene un búfer de datos grande para que DMA complete una operación.
  - Para otros periféricos como UART o ADC, como veremos más adelante, los datos se suelen transferir en uno o dos bytes a la vez.
  - En lugar de otorgarle al DMAC el uso exclusivo del bus para la transferencia del bloque de datos, la CPU permitirá que el DMAC se haga cargo del bus solo cuando la CPU no lo esté usando.
  - De esta manera, el impacto de DMA en el rendimiento de la CPU se puede reducir aún más.

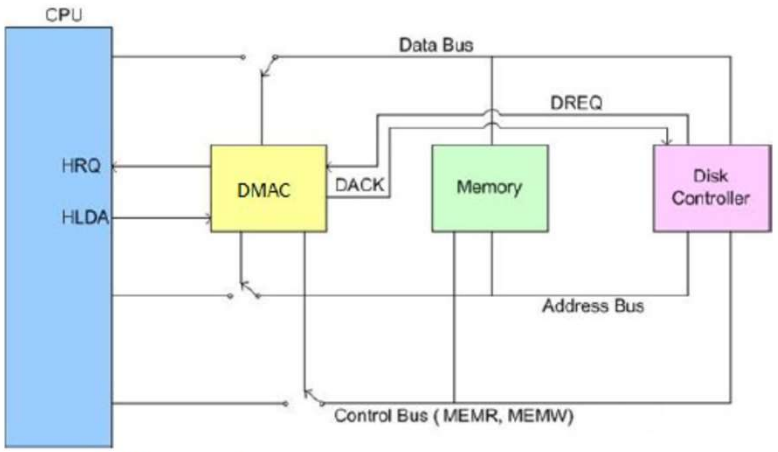


Figure 13- 1: DMA Usage of System Bus

DMA in Microcontrollers

- Como mencionamos, DMA (Acceso Directo a Memoria) es un método para permitir que los periféricos obtengan acceso a la memoria sin involucrar a la CPU.
- Esto generalmente se logra agregando un controlador DMA y un árbitro de bus de memoria al sistema que transfiere datos entre la memoria y los periféricos a través del mismo bus de datos que la CPU lee/escribe en la memoria.
- En el microcontrolador como Arm, el software configura el controlador DMA configurando la dirección de origen, la dirección de destino, el ancho de la palabra de datos y la cantidad de datos a transferir.
- El software también configura el controlador DMA y el periférico para el evento que desencadena la transferencia de datos.
- Dado que la idea de DMA es reducir la participación de la CPU en la transferencia de datos, la interrupción generalmente se utiliza para notificar la finalización de la transferencia de datos.

DMA in Microcontrollers

- La razón principal para realizar DMA es liberar la CPU para realizar otras tareas mientras los periféricos mueven datos dentro y fuera de la memoria.
- En este capítulo, utilizaremos la transmisión de datos UART y ADC para ilustrar cómo funciona DMA.
- Consideremos si necesitamos transmitir 2 Kbytes de datos a través de un UART a 9600 bauds.
  - Usando un método de polling, la CPU esperará a que el registro de datos de transmisión UART esté disponible antes de leer un byte de los datos de la memoria y escribirlo en el registro de datos de transmisión UART.
  - Mientras dura la transmisión de datos de 2 Kbytes, la CPU está completamente ocupada sin realizar ninguna otra tarea.
  - A 9600 baudios, se necesitará un poco más de dos segundos para transmitir 2 Kbytes de datos. Durante estos dos segundos, el sistema no puede realizar otras tareas.

DMA in Microcontrollers

- En el Capítulo 6, introdujimos interrupciones. Al utilizar interrupciones, la CPU podrá realizar otras tareas mientras transmite datos.
  - Cuando el registro de datos de transmisión UART se vacía, genera una interrupción en la CPU.
  - La CPU detiene la tarea actual, guarda los registros actuales de la CPU en la pila y luego ejecuta el controlador de interrupciones.
  - El controlador de interrupciones lee un byte de datos de la memoria y lo escribe en el registro de datos de transmisión UART.
  - La CPU restaura los registros de la pila y reanuda la tarea que fue interrumpida.
  - Todas estas operaciones de CPU incurrieron solo para mover un byte de datos.

DMA in Microcontrollers

- Con 16 MHz de reloj del sistema, se necesitan un par de microsegundos de tiempo de CPU para enviar un byte de datos a la UART.
  - Para dos Kbytes de datos, se necesitan varios milisegundos del tiempo de la CPU.
  - Desde la perspectiva del rendimiento, es varias magnitudes mejor que usar polling, pero podemos hacerlo aún mejor con DMA.
- Con DMA en Arm, el software necesita configurar el controlador DMA configurando la dirección inicial de los datos en la memoria en el registro de dirección de origen, la dirección del registro de transmisión UART en el registro de destino y la cantidad de bytes para transmitir en el registro de conteo de datos.
  - Luego, el software configura el UART para activar la transferencia DMA cuando el registro de datos de transmisión del UART esté vacío.
  - A partir de aquí, el controlador DMA ejecutará por sí solo sin involucrar a la CPU hasta que se envíen todos los bytes.

DMA in Microcontrollers

- Cuando el registro de datos de transmisión UART queda vacío, notifica al controlador DMA.
- El controlador DMA solicita el bus de datos de la memoria al árbitro del bus.
- Cuando se le concede el bus de datos, lee los datos de la memoria utilizando el registro de dirección de origen y escribe los datos en el registro de datos de transmisión UART.
- El controlador DMA incrementará el registro de dirección de origen para que apunte al siguiente byte de datos y disminuirá el registro de cuenta de datos para ver si se enviaron todos los datos.
- Cuando el registro de cuenta de datos llega a cero, el controlador DMA genera una interrupción de transferencia completa para que la CPU sepa que se enviaron todos los datos.

DMA in Microcontrollers

- Todas las operaciones de DMA hasta que se interrumpe la transferencia completa no involucran a la CPU y la CPU puede realizar otras tareas sin ser interrumpida.

Preguntas de revisión

- 1. Verdadero o falso. Cuando el DMA está funcionando, la CPU no está utilizando el bus.
- 2. Verdadero o falso. Cuando la CPU utiliza el bus, el DMA está inactivo.
- 3. Verdadero o falso. Ningún bus puede dar servicio a dos amos al mismo tiempo.

DMA Programming in STM32

- El microcontrolador STM32F446RE tiene dos controladores DMA. Vea la Figura 13-2.
- Cada controlador DMA tiene ocho flujos DMA. Juntos, tienen 16 flujos que pueden realizar operaciones DMA de forma independiente.
- Cada flujo DMA tiene ocho canales.
- Las asociaciones de periféricos a canales/flujos DMA están cableadas y se pueden ver en las Tablas 13-1 y 13-2.

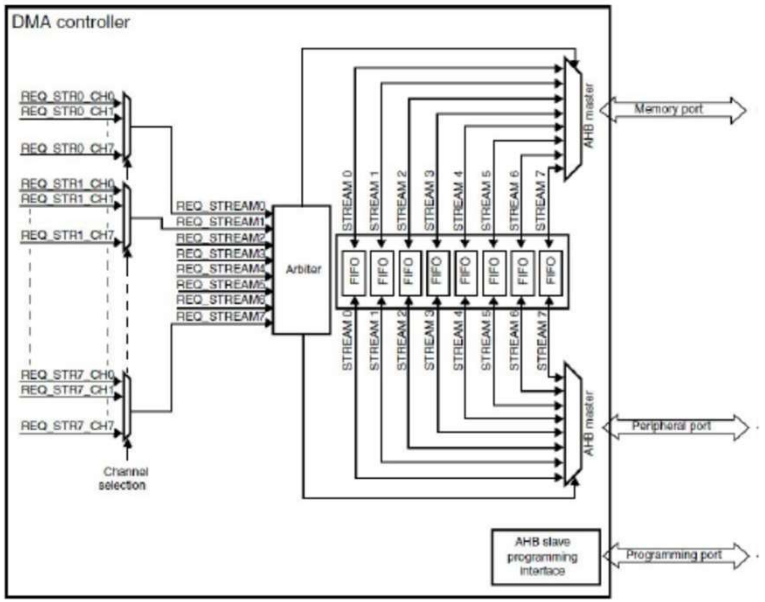


Figure 13- 2: DMA Block Diagram in STM32F4xx

Table 13- 1: DMA1 Mapping in STM32F4xx

DMA Programming in STM32

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX	SPDIFRX_DT	SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX	SPDIFRX_CS	SPI3_TX
Channel 1	I2C1_RX	I2C3_RX	TIM7_UP	-	TIM7_UP	I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1	-	FMPI2C1_RX	TIM4_CH2	-	FMPI2C1_RX	TIM4_UP	TIM4_CH3
Channel 3	-	TIM2_UP TIM2_CH3	I2C3_RX	-	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4	UART5_RX	USART3_RX	UART4_RX	USART3_TX	UART4_TX	USART2_RX	USART2_TX	UART5_TX
Channel 5	-	-	TIM3_CH4 TIM3_UP	-	TIM3_CH1 TIM3_TRIG	TIM3_CH2	-	TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2	-	TIM5_UP	-
Channel 7	-	TIM6_UP	I2C2_RX	I2C2_RX	USART3_TX	DAC1	DAC2	I2C2_TX

Table 13- 2: DMA2 Mapping in STM32F4xx

- Cada stream tiene seis registros y también comparte con otros streams: bits de estado de interrupción y bits de flag de borrado de interrupción en cuatro registros.
- . . .

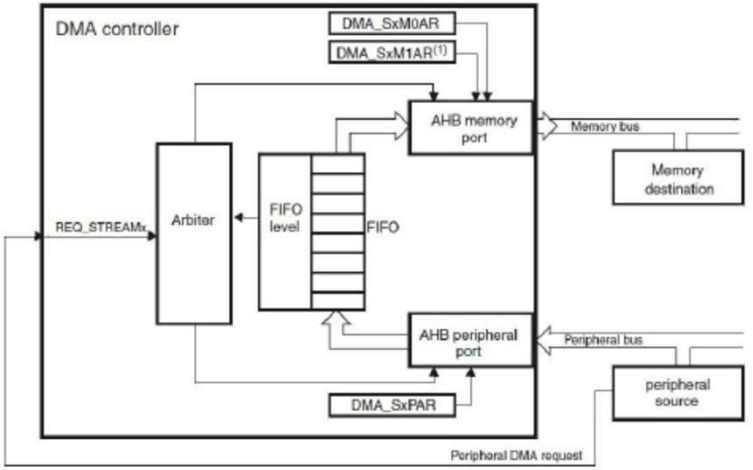


Figure 13-3: Peripheral-to-memory mode

Figure 13-4: Memory-to-peripheral mode

Program 13-1: Send a string to USART2 by DMA

```
/* p13_1.c Send a string to USART2 by DMA
*
* USART2 transmit is on DMA1 Stream6 Channel 4.
* See Reference manual Table 28 RM0390 July 2017.
*
* USART2 Tx signal is connected to pin PA0. To see the output of USART2 * on a PC, you need to use a USB-serial module. Connect the Rx pin of * the module to the PA0 pin of the STM32F446 Nucleo board. Make sure * the USB-serial module you use has a 3.3V interface.
*
* By default, the system clock is running at 16 MHz.
* The UART is configured for 9600 Baud.
* PA0 - USART2 TX (AF8)
*
* This program was tested with Keil uVision v5.24a with DFP v2.11.0
*/

#include "stm32F4xx.h"

void USART2_init(void);
void DMA1_init(void);
void DMA1_Stream6_setup(unsigned int src, unsigned int dst, int len);

int done = 1;

int main (void) {
char alphabets[] = "abcdefghijklmnopqrstuvwxyz"; char message[80];
int i;
int size = sizeof(alphabets);

USART2_init(); DMA1_init();

while (1) {
/* prepare the message for transfer */ for (i = 0; i < size; i++)

message[i] = alphabets[i]; /* Configure PA2 for USART2_TX */
GPIOA->AFR[0] &= ~0x0F00;
GPIOA->AFR[0] |= 0x0700; /* alt7 for USART2 */
GPIOA->MODER &= ~0x0030;
GPIOA->MODER |= 0x0020; /* enable alternate function for PA2 */

/* send the message out by USART2 using DMA */
while (done == 0) {} /* wait until DMA data transfer is done */ done = 0; /* clear done flag */
DMA1_Stream6_setup((unsigned int)message, (unsigned int)&USART2->DR, size);

}
}
/* Initialize UART pins, Baudrate

The USART2 is configured to send output to pin PA2 at 9600 Baud.

*/
void USART2_init (void) {
RCC->AHB1ENR |= 1; /* enable GPIOA clock */
RCC->APB1ENR |= 0x20000; /* enable USART2 clock */

USART2->BRR = 0x0683; /* 9600 baud @ 16 MHz */ USART2->CR1 = 0x0008;
/* enable Tx, 8-bit data */ USART2->CR2 = 0x0000; /* 1 stop bit */
USART2->CR3 = 0x0000; /* no flow control */ USART2->CR1 |= 0x2000;
/* enable USART2 */

USART2->SR = ~0x40; /* clear TC flag */
USART2->CR1 |= 0x0040; /* enable transmit complete interrupt */
NVIC_EnableIRQ(USART2_IRQn); /* USART2 interrupt enable at NVIC */ }

/* Initialize DMA1 controller
* DMA1 controller's clock is enabled and also the DMA interrupt is * enabled in NVIC.
*/

void DMA1_init(void) {
RCC->AHB1ENR |= 0x00200000; /* DMA controller clock enable */ DMA1->HIFCR = 0x003F0000; /* clear all interrupt flags of Stream 6 */
NVIC_EnableIRQ(DMA1_Stream6_IRQn); /* DMA interrupt enable at NVIC
*/

}

/* Set up a DMA transfer for USART2
* The USART2 is connected to DMA1 Stream 6. This function sets up the * peripheral register address, memory address, number of transfers, * data size, transfer direction, and DMA interrupts are enabled.
* At the end, the DMA controller is enabled and the USART2 transmit * DMA is enabled.
*/

void DMA1_Stream6_setup(unsigned int src, unsigned int dst, int len)
{ DMA1_Stream6->CR &= ~1; /* disable DMA1 Stream 6 */
while (DMA1_Stream6->CR & 1) {} /* wait until DMA1 Stream 6 is disabled */ DMA1->HIFCR = 0x003F0000; /* clear all interrupt flags of Stream 6 */ DMA1_Stream6->PAR = dst;
DMA1_Stream6->M0AR = src;
DMA1_Stream6->NDTR = len;
```

DMA Programming in STM32

```
DMA1_Stream6->CR = 0x08000000; /* USART2_TX on DMA1 Stream6 Channel
4 */ DMA1_Stream6->CR |= 0x00000440; /* data size byte, mem incr,
mem-to-peripheral */ DMA1_Stream6->CR |= 0x16; /* enable interrupts
DMA_IT_TC | DMA_IT_TE |

DMA_IT_DME */
DMA1_Stream6->FCR = 0; /* direct mode, no FIFO */
DMA1_Stream6->CR |= 1; /* enable DMA1 Stream 6 */

USART2->SR &= ~0x0040; /* clear UART transmit complete interrupt
flag */ USART2->CR3 |= 0x80; /* enable USART2 transmitter DMA */

}
have a placeholder for error handling code. If the interrupt is from
DMA data transfer complete, the DMA controller is disabled, the
interrupt flags are cleared.

/* DMA1 Stream6 interrupt handler
This function handles the interrupts from DMA1 controller Stream6.
The error interrupts

*/
void DMA1_Stream6_IRQHandler(void)
{

if (DMA1->HISR & 0x000C0000) /* if an error occurred */
while(1) {} /* substitute this by error handling */
DMA1->HIFCR = 0x003F0000; /* clear all interrupt flags of Stream 6
*/
DMA1_Stream6->CR &= ~0x10; /* disable DMA1 Stream 6 TCIE */ }

/* USART2 interrupt handler
* USART2 transmit complete interrupt is used to set the done flag to
signal * the other part of the program that the data transfer is
done. */

void USART2_IRQHandler(void)
{
USART2->SR &= ~0x0040; /* clear transmit complete interrupt flag */
done = 1; /* set the done flag */
```

An Example of DMA from ADC Peripheral to Memory

- En este ejemplo, el ADC se activa periódicamente mediante un temporizador (TIM2 Ch2).
  - Los resultados de la conversión se almacenan en un búfer en la memoria mediante DMA.
  - Cuando el búfer está lleno, el controlador DMA genera una interrupción de transferencia completa para notificar al programa.
  - Una vez configurado, hasta el reconocimiento de la interrupción, todas estas actividades se realizan sin la participación de la CPU.
  - Esto es muy útil para la adquisición de datos. Normalmente, se utilizará el modo de doble búfer para que, mientras los datos del ADC se transfieren a un búfer, la CPU pueda procesar los datos anteriores en el otro búfer.
  - Pero para simplificar, utilizamos solo un búfer. Cuando el búfer está lleno, TIM2, ADC y DMA se detienen mientras la CPU procesa los datos y los envía a través de USART2.

An Example of DMA from ADC Perip Memory

Program 13-2: Acquire data from ADC by DMA

```
/* p13_2.c Acquire data from ADC by DMA
*
* The program sets up the timer TIM2 channel 2 to trigger ADC1 to *
convert the analog input channel 0. The output of the ADC1 is
transferred * to the buffer in memory by DMA. Once the buffer if
full, the DMA is * stopped. That data are converted to decimal ASCII
numbers and sent * to USART2 to be displayed on the console.
*
* A global variable, done, is used by the DMA transfer complete
interrupt * handler to signal the other part of the program that a
buffer full of * data conversion is done.
*
* This program was tested with Keil uVision v5.24a with DFP v2.11.0
*/

#include "stm32f4xx.h" #include "stdio.h"

#define ADCBUFSIZE 64 void USART2_init (void); /* Initialize UART
pins, Baudrate */
void DMA2_init(void); /* Initialize DMA2 controller */
void DMA2_Stream0_setup(unsigned int src, unsigned int dst, int
len); /* set up a DMA transfer for ADC1 */
void TIM2_init(void); /* initialize TIM2 */
void ADC1_init(void); /* setup ADC */

int done = 1;
char adcbuf[ADCBUFSIZE]; /* buffer to receive DMA data transfers
from ADC conversion results */
char uartbuf[ADCBUFSIZE * 5]; /* buffer to hold ASCII numbers for
display */

int main(void) { int i;
char* p;

USART2_init(); DMA2_init(); TIM2_init(); ADC1_init();

while(1) {
done = 0; /* clear done flag */
/* start a DMA of ADC data transfer */
DMA2_Stream0_setup((uint32_t)adcbuf, (uint32_t)&(ADC1->DR),
ADCBUFSIZE); while (done == 0) {} /* wait for ADC DMA transfer
complete */

/* convert the ADC data into decimal ASCII numbers for display */ p
= uartbuf;
for (i = 0; i < ADCBUFSIZE; i++) {

sprintf(p, "%3d ", adcbuf[i]);
p += 4;
}

/* send the ADC numbers through USART2 to the console terminal */
for (i = 0; i < p - uartbuf; i++)
{

while (!(USART2->SR & 0x40)) {} USART2->DR = uartbuf[i]; }
}

}
/* Initialize TIM2
Timer TIM2 channel 2 is configured to generate PWM at 1 kHz. The
output of the timer signal is used to trigger ADC conversion.

/* Initialize ADC
ADC1 is configured to do 8-bit data conversion and triggered by
the rising edge of timer TIM2 channel 2 output.

*/
void ADC1_init(void) {
RCC->AHB1ENR |= 1; /* enable GPIOA clock */
GPIOA->MODER |= 3; /* PA0 analog */
RCC->APB2ENR |= 0x0100; /* enable ADC1 clock */
ADC1->CR1 = 0x2000000; /* 8-bit conversion */
ADC1->CR2 = 0x13000000; /* exten rising edge, extsel 3 = tim2.2 */
ADC1->CR2 |= 0x400; /* enable setting EOC bit after each conversion
*/ ADC1->CR2 |= 1; /* enable ADC1 */

}

*/
void TIM2_init(void) {
RCC->AHB1ENR |= 2; /* enable GPIOB clock */
GPIOB->MODER |= 0x80; /* PB3 timer2.2 out */
GPIOB->AFR[0] |= 0x1000; /* set pin for timer output mode */ RCC-
>APB1ENR |= 1; /* enable TIM2 clock */
TIM2->PSC = 160 - 1; /* divided by 160 */
TIM2->ARR = 100 - 1; /* divided by 100, sample at 1 kHz */ TIM2->CNT
= 0;
TIM2->CCMR1 = 0x6800; /* pwm1 mode, preload enable */ TIM2->CCER =
0x10; /* ch2 enable */
TIM2->CCR2 = 50 - 1;
```

```
}

/* Initialize DMA2 controller
* DMA2 controller's clock is enabled and also the DMA interrupt is *
enabled in NVIC.
*/

void DMA2_init(void) {
RCC->AHB1ENR |= 0x00400000; /* DMA2 controller clock enable */ DMA2-
>HIFCR = 0x003F; /* clear all interrupt flags of Stream 0 */
NVIC_EnableIRQ(DMA2_Stream0_IRQn); /* DMA interrupt enable at NVIC
*/

}

/* Set up a DMA transfer for ADC
* The ADC1 is connected to DMA2 Stream 0. This function sets up the
* peripheral register address, memory address, number of transfers,
* data size, transfer direction, and DMA interrupts are enabled.
* At the end, the DMA controller is enabled, the ADC conversion
* complete is used to trigger DMA data transfer, and the timer
* used to trigger ADC is enabled.
*/

void DMA2_Stream0_setup(unsigned int src, unsigned int dst, int len)
{ DMA2_Stream0->CR &= ~1; /* disable DMA2 Stream 0 */
while (DMA2_Stream0->CR & 1) {} /* wait until DMA2 Stream 0 is
disabled */ DMA2->HIFCR = 0x003F; /* clear all interrupt flags of
Stream 0 */ DMA2_Stream0->PAR = dst;
DMA2_Stream0->M0AR = src;
DMA2_Stream0->NDTR = len;
DMA2_Stream0->CR = 0x00000000; /* ADC1_0 on DMA2 Stream0 Channel 0
*/ DMA2_Stream0->CR |= 0x00000400; /* data size byte, mem incr,
peripheral-to-mem */ DMA2_Stream0->CR |= 0x16; /* enable interrupts
DMA_IT_TC | DMA_IT_TE |

DMA_IT_DME */
DMA2_Stream0->FCR = 0; /* direct mode, no FIFO */
DMA2_Stream0->CR |= 1; /* enable DMA2 Stream 0 */

ADC1->CR2 |= 0x0100; /* enable ADC conversion complete DMA data
transfer */
TIM2->CR1 = 1; /* enable timer2 */
}

/* DMA2 Stream0 interrupt handler

This function handles the interrupts from DMA2 controller Stream0.
The error interrupts
have a placeholder for error handling code. If the interrupt is from
DMA data
transfer complete, the DMA controller is disabled, the interrupt
```

eripheral to

```
flags are
cleared, the ADC conversion complete DMA trigger is turned off and
the timer
that triggers ADC conversion is turned off too.
*/
void DMA2_Stream0_IRQHandler(void)
{
if (DMA2->HISR & 0x000C) /* if an error occurred */ while(1) {} /*
substitute this by error handling */

DMA2_Stream0->CR = 0; /* disable DMA2 Stream 0 */ DMA2->LIFCR =
0x003F; /* clear DMA2 interrupt flags */ ADC1->CR2 &= ~0x0100; /*
disable ADC conversion complete DMA */ TIM2->CR1 &= ~1; /* disable
timer2 */

done = 1; }

/* Initialize UART pins, Baudrate
The USART2 is configured to send output to pin PA2 at 9600 Baud.
This is used to display the ADC conversion results on the console
terminal.

*/
void USART2_init (void) {
RCC->AHB1ENR |= 1; /* enable GPIOA clock */
RCC->APB1ENR |= 0x20000; /* enable USART2 clock */

/* Configure PA2 for USART2_TX */
GPIOA->AFR[0] &= ~0x0F00;
GPIOA->AFR[0] |= 0x0700; /* alt7 for USART2 */
GPIOA->MODER &= ~0x0030;
GPIOA->MODER |= 0x0020; /* enable alternate function for PA2 */

USART2->BRR = 0x0683; /* 9600 baud @ 16 MHz */ USART2->CR1 = 0x0008;
/* enable Tx, 8-bit data */ USART2->CR2 = 0x0000; /* 1 stop bit */
USART2->CR3 = 0x0000; /* no flow control */ USART2->CR1 |= 0x2000;
/* enable USART2 */
```

An Example of DMA from ADC Peripheral to Memory

- Preguntas de revisión
  - 1. Verdadero o falso. El Arm STM32F4xx tiene 2 controladores DMA.
  - 2. En STM32F4xx Arm, ¿cuántas streams podemos tener?
  - 3. Verdadero o falso. STM32F4xx solo puede realizar transferencias de datos de memoria a memoria mediante DMA.
  - 4. En STM32F4xx Arm, ¿cuál es la cantidad máxima de datos que se pueden transferir mediante DMA?
  - 5. En STM32F4xx Arm, ¿por qué usamos datos de tamaño de bytes cuando usamos la transferencia de datos de DMA a UART2?

Referencias

- Programming with STM32: Getting Started with the Nucleo Board and C/C++ 1st Edición - Donal Norris (Author)
- Nucleo Boards Programming with the STM32CubeIDE, Hands-on in more than 50 projects - Dogan Ibrahim (Author)
- STM32 Arm Programming for Embedded Systems, Using C Language with STM32 Nucleo - Muhammad Ali Mazidi (Author), Shujen Chen (Author), Eshragh Ghaemi (Author)

Manos a la obra con el . . .

. . . Proyecto Intermedio

. . . un enfoque centrado en la **práctica propia de la carrera** más que en el desarrollo teórico disciplinar, con eje en la **participación de las y los estudiantes**


Las y los estudiantes preguntarán:
¿en qué lío nos metimos?

¡Muchas gracias!

¿Preguntas?

. . .

Consultas a: jcruz@fi.uba.ar