CPE 323
MODULE 12
DIRECT MEMORY ACCESS (DMA) CONTROLLER

Aleksandar Milenković

Email: milenka@uah.edu

Web: http://www.ece.uah.edu/~milenka

Overview

This module introduces direct memory access (DMA) transfers and DMA controllers. The module describes direct memory access transfers and how they compare to other software approaches to interfacing peripherals (polling and interrupt). You will learn about hardware and software aspects of MSP430's DMA controller.

Objectives

Upon completion of this module learners will be able to:

- *Describe hardware and software aspects of direct memory access transfers*
- *Utilize DMA controllers to interface input/output peripherals*

Contents

1 Introduction

So far we have discussed two software techniques for interfacing input/output peripherals: (a) using polling and (b) using interrupt service routines.

With polling, we continually check whether a hardware event has occurred by reading an I/O peripheral register that contains the corresponding flag that indicates whether an event has occurred or not. For example, recall the use of the watchdog timer in the interval mode to toggle the LED1 in the Lauchpad platform as described in Module 09. When the the predefined time interval (1 s) expires, the WDTIFG bit in the SFRIFG1 register is set to a logic 1. Code 1 shows a code snippet that continually tests whether the WDTIFG is set. The statement in line 10 tests the WDTIFG bit in the SFRIFG1 register; the testing is repeated as long as the WDTIFG flag is not set (has value 0). Once the WDTIFG flag is set, the condition in line 10 becomes false, and the statements in lines 11 and 12 are executed to toggle LED1 and clear the flag, respectively. The infinite loop ensures that LED1 is toggled every 1 s because the WDTIFG is set once every second.

```
1   #include <msp430.h>
2
3   void main(void)
4   {
5       WDTCTL = WDT_ADLY_1000;             // 1 s interval timer
6       P1DIR |= BIT0;                      // Set P1.0 to output direction
7
8       for (;;) {
9           // Use software polling
10          while (!(SFRIFG1 & WDTIFG));
11          P1OUT ^= BIT0;
12          SFRIFG1 &= ~WDTIFG;         // Clear bit WDTIFG in IFG1
13
14      }
15  }
```

Code 1. Toggling LED1 using WDT and Software Polling on WDTIFG.

Code 2 shows a program that performs the same task using the watchdog timer interval mode interrupt service routine (ISR). By enabling the interrupt, once the WDTIFG is set, the interrupt service routine is entered. Please note that ISR for the watchdog timer is single-sourced interrupt, so the WDTIFG flag is cleared during exception processing in hardware.

```
1   #include <msp430.h>
2
3   void main(void) {
4       WDTCTL = WDT_ADLY_1000;           // 1s interval mode
5       P1DIR |= BIT0;                    // Set P1.0 to output direction
6       SFRIE1 |= WDTIE;                  // Enable WDT interrupt
7
```

```
8       _BIS_SR(LPM0_bits + GIE);         // Enter LPM0 with interrupt
9   }
10  // Watchdog Timer interrupt service routine
11  #pragma vector=WDT_VECTOR
12  __interrupt void watchdog_timer(void) {
13      P1OUT ^= BIT0;                  // Toggle P1.0 using exclusive-OR
14  }
```

Code 2. Toggling LED1 using WDT and Software Polling on WDTIFG.

In this module we will discuss third approach to interfacing I/O peripherals using a direct memory access controller or DMA. The primary role of a DMA is to carry out data transfers without involvement from the processor. DMAs can thus be considered as hardware accelerators for moving data in a computer system.

Things to remember 1-1. Three Approaches to Interfacing I/O Peripherals

Three principal software approaches to interface I/O peripherals in computer systems are as follows:

(a) polling – the CPU is actively polling a peripheral to detect when it is ready before it carries out the requested I/O transfer;

(b) Interrupt Service Routine – an interrupt is requested and the CPU carries out the requested I/O transfer inside the corresponding ISR;

(c) Direct Memory Access – a special peripheral called DMA controller carries out the requested I/O transfer without any CPU involvement.

2 Direct Memory Transfers: Basic Principles

A DMA can be used to facilitate the following data transfers:

- Input to memory: from an input peripheral (e.g., serial communication interface receiving data) to a buffer in memory;
- Memory to output: from a buffer in memory to an output peripheral (e.g., serial communication interface transmitting data);
- Input to output: from an input peripheral to an output peripheral; or
- Memory to memory: from a source buffer in memory to a destination buffer in memory.

To utilize a DMA, we need to configure it to carry out desired data transfers. Configuring data transfers involves specifying the starting address of the data source (e.g., RXBUF of an USCI peripheral), the starting address of the data destination (e.g., starting address of a character array in memory that will keep the message received from a USCI), and the number of elements to be transferred (e.g., our message has 20 characters). In addition, we typically configure the type of DMA transfers (bytes, words), how individual data transfers are carried out, and how they are handled in hardware.

Figure 1 shows a register-view of a typical DMA. The Control register is initialized to enable certain types of data transfers, the SourceAddress register is initialized with the starting address of the data source, the DestinationAddress register is initialized with the starting address of the data destination, and the Size register is initialized to specify the number of data items that needs to be transferred.

Once the DMA is initialized and enabled, DMA will carry out data transfers independently from the processor, entirely in hardware. To do this, DMA needs to be able to initiate read and write transactions on the bus. In computer systems discussed so far, only the processor could initiate read and write transaction on the bus. However, if we have a DMA in the system, it can also initiate reads from memory or I/O peripherals or writes to memory and I/O peripherals. Consequently, DMA is a special type of a peripheral – it is initialized by the processor during data transfer setup and thus acts as any other peripheral. However, it also acts independently from the processor when transferring data. When a device can initiate read and write transactions on the bus, it is typically referred to as being a master on the bas – other peripherals and memory have to respond to read or write requests from the DMA.

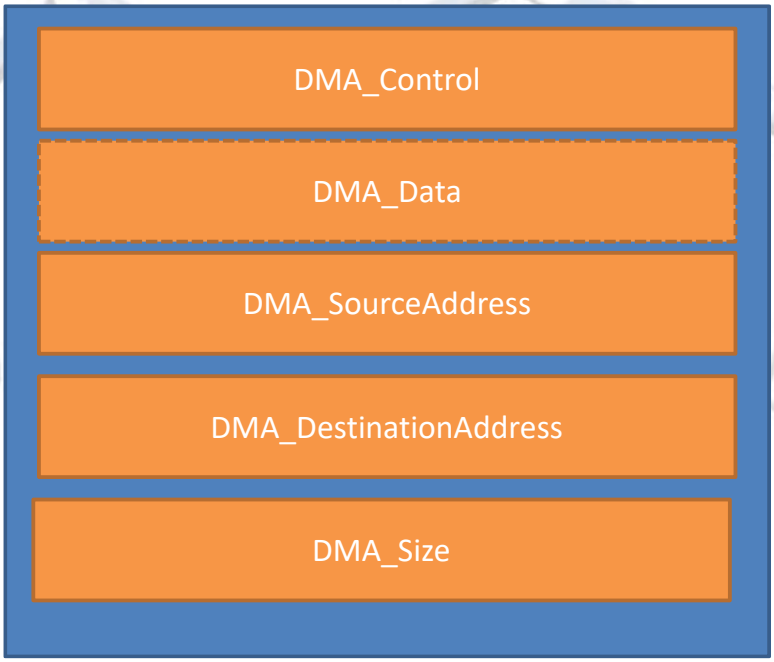

Figure 1. DMA Registers.

`Things to remember 2-1. DMA is a special peripheral`

Unlike other I/O peripherals, a DMA is special type of a peripheral. Like any other peripheral it is initialized by the CPU by writing and reading its registers. However, once it is initialized, the DMA peripheral can initiate reads and writes from I/O peripherals and memory independently from the CPU. Ability to initiate data transfers on the bus makes the DMA special.

Things to remember 2-2. DMA registers

Initializing a DMA transfers involves setting up the source data address (DMAxSA), the destination data address (DMAxDA), and the size of the data block to be transferred (DMAxSZ).

3 MSP430's DMA Peripheral

MSP430 microcontrollers include a DMA with multiple channels (typically 3, but latest microcontrollers can have more). Each channel has its own set of registers and configurable triggers as shown in Figure 2, so multiple independent transfers can take place in the same time window, though only one data transfer can take place at a time on the bus. Each channel has its set of special registers: DMAxSA, DMAxDA, and DMAxSZ. Each DMA channel has its own control register, DMAxCTL. In addition, there is a control register for entire peripheral.

In the following text, we will discuss several specific features of the MSP430's DMA controller.

DMA Addressing Modes. The DMA controller has four addressing modes as follows (Figure 3):

- Fixed address to fixed address
- Fixed address to block of addresses
- Block of addresses to fixed address
- Block of addresses to block of addresses

The addressing mode for each DMA channel is independently configurable. For example, channel 0 may transfer between two fixed addresses, while channel 1 transfers between two blocks of addresses. The addressing modes are configured with the DMASRCINCR and DMADSTINCR control bits. The DMASRCINCR bits select if the source address is incremented, decremented, or unchanged after each transfer. The DMADSTINCR bits select if the destination address is incremented, decremented, or unchanged after each transfer.

Transfers may be byte-to-byte, word-to-word, byte-to-word, or word-to-byte. When transferring word-to-byte, only the lower byte of the source word is transferred. When transferring byte-to-word, the upper byte of the destination word is cleared when the transfer occurs. The control fields DSTBYTE and SRCBYTE in the DMAxCTL control register (x=0, 1, 2, …) select the type of a transfer.

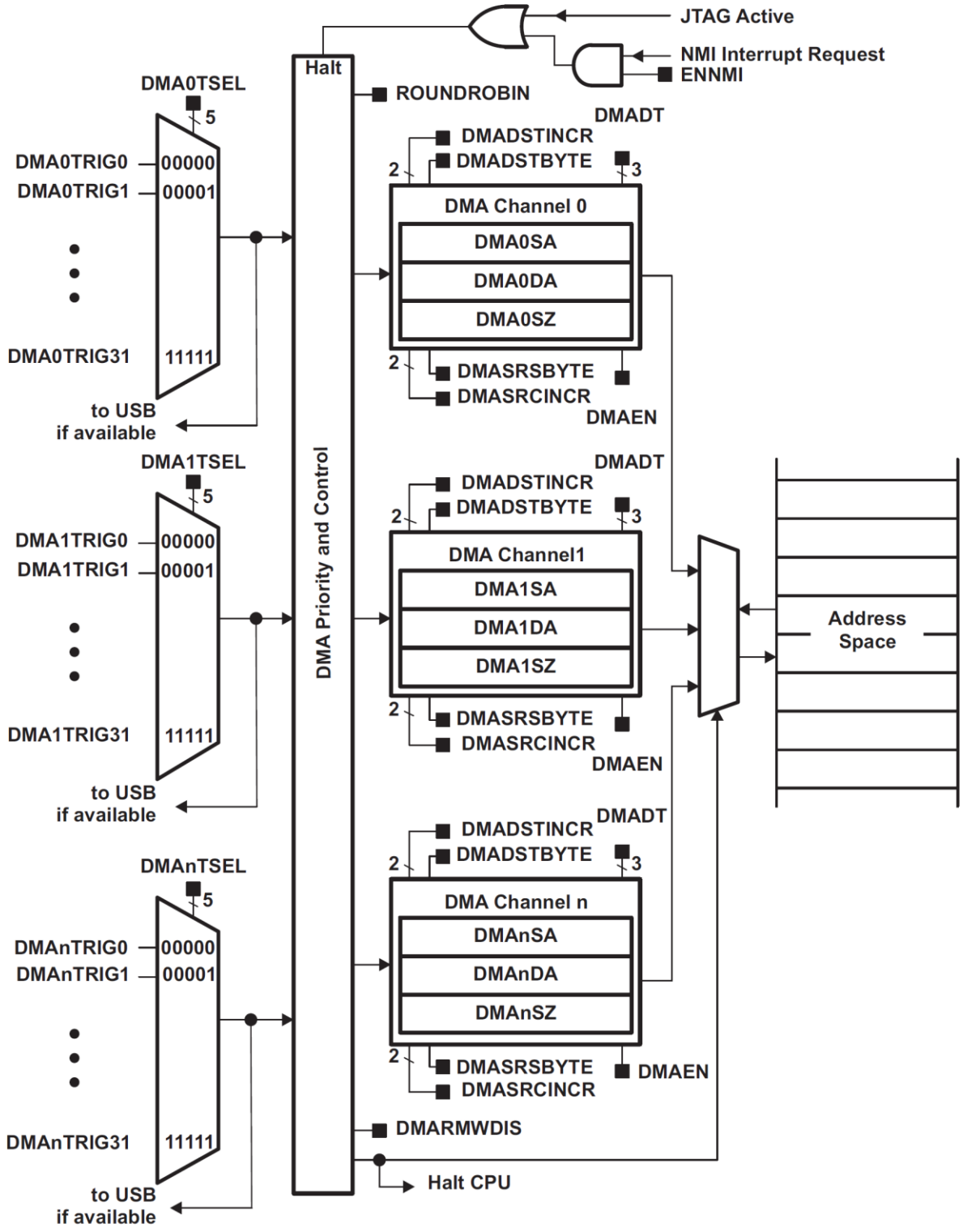

Figure 2. Block diagram of a DMA peripheral with 3 channels (as in MSP430F5529).

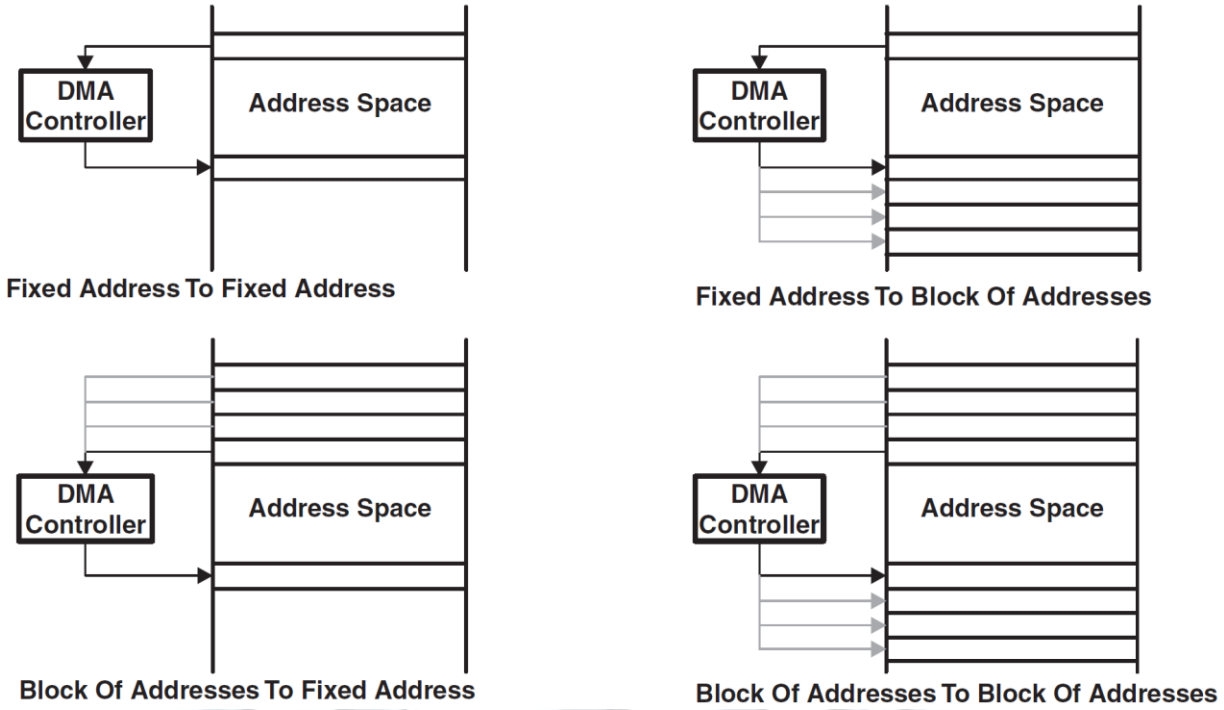

Figure 3. DMA addressing modes.

DMA Transfer Modes. The DMA controller six transfer modes as shown in Figure 4.

Single transfer modes (DMADAT=000b). In single transfer mode, each byte or word transfer requires a separate trigger. The DMAxSZ register defines the number of transfers to be made. The DMADSTINCR and DMASRCINCR bits select if the destination address and the source address are incremented, decremented, or remain fixed after each transfer. If DMAxSZ = 0, no transfers occur.

The DMAxSA, DMAxDA, and DMAxSZ registers are copied into temporary registers. The temporary values of DMAxSA and DMAxDA are incremented or decremented after each transfer. The DMAxSZ register is decremented after each transfer. When the DMAxSZ register decrements to zero, it is reloaded from its temporary register and the corresponding DMAIFG flag is set. When DMADT = {0}, the DMAEN bit is cleared automatically when DMAxSZ decrements to zero and must be set again for a new round of transfer to start. Please note that thanks to these temporary registers, you do not have to re-initialize the DMA registers over-and-over again if you want to repeat data transfer, just setting the DMAEN would suffice.

Repeated single transfer modes (DMADAT=100b). In repeated single transfer mode, the DMA controller remains enabled with DMAEN = 1, and a transfer occurs every time a trigger occurs.

Block transfer mode (DMADT=001b). In block transfer mode, a transfer of a complete block of data occurs after one trigger. When DMADT = {1}, the DMAEN bit is cleared after the completion of the block transfer and must be set again before another block transfer can be triggered. After a block transfer has been triggered, further trigger signals occurring during the block transfer are ignored. The DMAxSA, DMAxDA, and DMAxSZ registers are copied into temporary registers. The temporary values of DMAxSA and DMAxDA are incremented or

decremented after each transfer in the block. The DMAxSZ register is decremented after each transfer of the block and shows the number of transfers remaining in the block. When the DMAxSZ register decrements to zero, it is reloaded from its temporary register and the corresponding DMAIFG flag is set. During a block transfer, the CPU is halted until the complete block has been transferred. The block transfer takes 2 × MCLK × DMAxSZ clock cycles to complete. CPU execution resumes with its previous state after the block transfer is complete.

Repeated block transfer mode (DMADT=101b). In repeated block transfer mode, the DMAEN bit remains set after completion of the block transfer. The next trigger after the completion of a repeated block transfer triggers another block transfer.

Bust-Block transfer mode (DMADT=010b or DMADT=011b). In burst-block mode, transfers are block transfers with CPU activity interleaved. The CPU executes two MCLK cycles after every four byte/word transfers of the block, resulting in 20% CPU execution capacity. After the burst-block, CPU execution resumes at 100% capacity and the DMAEN bit is cleared. DMAEN must be set again before another burst-block transfer can be triggered. After a burst-block transfer has been triggered, further trigger signals occurring during the burst-block transfer are ignored.

The DMAxSA, DMAxDA, and DMAxSZ registers are copied into temporary registers. The temporary values of DMAxSA and DMAxDA are incremented or decremented after each transfer in the block. The DMAxSZ register is decremented after each transfer of the block and shows the number of transfers remaining in the block. When the DMAxSZ register decrements to zero, it is reloaded from its temporary register and the corresponding DMAIFG flag is set.

*Bust-Block transfer mode (DMADT=110b or DMADT=111b).*In repeated burst-block mode, the DMAEN bit remains set after completion of the burst-block transfer and no further trigger signals are required to initiate another burst-block transfer. Another burst-block transfer begins immediately after completion of a burst-block transfer. In this case, the transfers must be stopped by clearing the DMAEN bit, or by an (non)maskable interrupt (NMI) when ENNMI is set. In repeated burst block mode, the CPU executes at 20% capacity continuously until the repeated burst-block transfer is stopped.

DMADT	Transfer Mode	Description
000	Single transfer	Each transfer requires a trigger. DMAEN is automatically cleared when DMAxSZ transfers have been made.
001	Block transfer	A complete block is transferred with one trigger. DMAEN is automatically cleared at the end of the block transfer.
010, 011	Burst-block transfer	CPU activity is interleaved with a block transfer. DMAEN is automatically cleared at the end of the burst-block transfer.
100	Repeated single transfer	Each transfer requires a trigger. DMAEN remains enabled.
101	Repeated block transfer	A complete block is transferred with one trigger. DMAEN remains enabled.
110, 111	Repeated burst-block transfer	CPU activity is interleaved with a block transfer. DMAEN remains enabled.

Figure 4. DMA transfer modes.

DMA Triggers. The trigger source for each DMA channel is independently configured by DMAxTSEL bits. Figure 5 lists common triggers and their operation. Triggers correspond to I/O peripheral events, e.g., a character has been received in UCAxRFBUF (UCAxRXIFG is set) or a UCAxTXIFG is set indicating that the corresponding TXBUF is ready to receive a new character. When DMALEVEL=0, edge sensitive triggers are used – the rising edge of the trigger signal initiates the transfer. Note: in block or burst-block modes only one trigger is required to initiate a transfer of the entire block. When DMALEVEL=1, level-sensitive triggers are used. For proper operation, level-sensitive triggers can only be used when external trigger DMAE0 is selected as the trigger. When DMALEVEL=1, transfer modes selected by DMADT = {0, 1, 2, or 3} are recommended because the DMAEN bit is automatically reset after the configured transfer.

DMA triggers are often device-specific, and thus the corresponding manual should be consulted to determine trigger encoding. For example, for MSP430F5529 microcontroller, the triggers are shown in Figure 6.

Module	Operation
DMA	A transfer is triggered when the DMAREQ bit is set. The DMAREQ bit is automatically reset when the transfer starts. A transfer is triggered when the DMAxIFG flag is set. DMA0IFG triggers channel 1, DMA1IFG triggers channel 2, and DMA2IFG triggers channel 0. None of the DMAxIFG flags are automatically reset when the transfer starts. A transfer is triggered by the external trigger DMAE0.
Timer_A	A transfer is triggered when the TAxCCR0 CCIFG flag is set. The TAxCCR0 CCIFG flag is automatically reset when the transfer starts. If the TAxCCR0 CCIE bit is set, the TAxCCR0 CCIFG flag dies not trigger a transfer. A transfer is triggered when the TAxCCR2 CCIFG flag is set. The TAxCCR2 CCIFG flag is automatically reset when the transfer starts. If the TAxCCR2 CCIE bit is set, the TAxCCR2 CCIFG flag does not trigger a transfer.
Timer_B	A transfer is triggered when the TBxCCR0 CCIFG flag is set. The TBxCCR0 CCIFG flag is automatically reset when the transfer starts. If the TBxCCR0 CCIE bit is set, the TBxCCR0 CCIFG flag does not trigger a transfer. A transfer is triggered when the TBxCCR2 CCIFG flag is set. The TBxCCR2 CCIFG flag is automatically reset when the transfer starts. If the TBxCCR2 CCIE bit is set, the TBxCCR2 CCIFG flag does not trigger a transfer.
USCI_Ax	A transfer is triggered when USCI_Ax receives new data. UCAxRXIFG is automatically reset when the transfer starts. If UCAxRXIE is set, the UCAxRXIFG does not trigger a transfer. A transfer is triggered when USCI_Ax is ready to transmit new data. UCAxTXIFG is automatically reset when the transfer starts. If UCAxTXIE is set, the UCAxTXIFG does not trigger a transfer.
USCI_Bx	A transfer is triggered when USCI_Bx receives new data. UCBxRXIFG is automatically reset when the transfer starts. If UCBxRXIE is set, the UCBxRXIFG does not trigger a transfer. A transfer is triggered when USCI_Bx is ready to transmit new data. UCBxTXIFG is automatically reset when the transfer starts. If UCBxTXIE is set, the UCBxTXIFG does not trigger a transfer.
DAC12_A	A transfer is triggered when the DAC12_xCTL0 DAC12IFG flag is set. The DAC12_xCTL0 DAC12IFG flag is automatically cleared when the transfer starts. If the DAC12_xCTL0 DAC12IE bit is set, the DAC12_xCTL0 DAC12IFG flag does not trigger a transfer.
ADC10_A	A transfer is triggered by an ADC10IFG0 flag with the ADC10IE0 bit reset. A transfer is triggered when the conversion is completed and the ADC10IFG0 is set. Setting the ADC10IFG0 with software does not trigger a transfer. The ADC10IFG0 flag is automatically reset when the ADC10MEM0 register is accessed by the DMA controller.
ADC12_A	A transfer is triggered by an ADC12IFG flag with the corresponding ADC12IE bit reset. When single-channel conversions are performed, the corresponding ADC12IFG is the trigger. When sequences are used, the ADC12IFG for the last conversion in the sequence is the trigger. A transfer is triggered when the conversion is completed and the ADC12IFG is set. Setting the ADC12IFG with software does not trigger a transfer. All ADC12IFG flags are automatically reset when the associated ADC12MEMx register is accessed by the DMA controller.
MPY	A transfer is triggered when the hardware multiplier is ready for a new operand.
Reserved	No transfer is triggered.

Figure 5. DMA triggers and their operation.

TRIGGER	CHANNEL		
	0	1	2
0	DMAREQ	DMAREQ	DMAREQ
1	TA0CCR0 CCIFG	TA0CCR0 CCIFG	TA0CCR0 CCIFG
2	TA0CCR2 CCIFG	TA0CCR2 CCIFG	TA0CCR2 CCIFG
3	TA1CCR0 CCIFG	TA1CCR0 CCIFG	TA1CCR0 CCIFG
4	TA1CCR2 CCIFG	TA1CCR2 CCIFG	TA1CCR2 CCIFG
5	TA2CCR0 CCIFG	TA2CCR0 CCIFG	TA2CCR0 CCIFG
6	TA2CCR2 CCIFG	TA2CCR2 CCIFG	TA2CCR2 CCIFG
7	TB0CCR0 CCIFG	TB0CCR0 CCIFG	TB0CCR0 CCIFG
8	TB0CCR2 CCIFG	TB0CCR2 CCIFG	TB0CCR2 CCIFG
9	Reserved	Reserved	Reserved
10	Reserved	Reserved	Reserved
11	Reserved	Reserved	Reserved
12	Reserved	Reserved	Reserved
13	Reserved	Reserved	Reserved
14	Reserved	Reserved	Reserved
15	Reserved	Reserved	Reserved

TRIGGER	CHANNEL		
	0	1	2
16	UCA0RXIFG	UCA0RXIFG	UCA0RXIFG
17	UCA0TXIFG	UCA0TXIFG	UCA0TXIFG
18	UCB0RXIFG	UCB0RXIFG	UCB0RXIFG
19	UCB0TXIFG	UCB0TXIFG	UCB0TXIFG
20	UCA1RXIFG	UCA1RXIFG	UCA1RXIFG
21	UCA1TXIFG	UCA1TXIFG	UCA1TXIFG
22	UCB1RXIFG	UCB1RXIFG	UCB1RXIFG
23	UCB1TXIFG	UCB1TXIFG	UCB1TXIFG
24	ADC12IFGx(2)	ADC12IFGx(2)	ADC12IFGx(2)
25	Reserved	Reserved	Reserved
26	Reserved	Reserved	Reserved
27	USB FNRXD	USB FNRXD	USB FNRXD
28	USB ready	USB ready	USB ready
29	MPY ready	MPY ready	MPY ready
30	DMA2IFG	DMA0IFG	DMA1IFG
31	DMAE0	DMAE0	DMAE0

Figure 6. DMA triggers for MSP430F5529.

DMA Channel Priorities. The default DMA channel priorities are channel 0 to channel 7. If multiple triggers happen simultaneously or are pending, the channel with the highest priority completes its transfer first. The DMA channel priorities are configurable using ROUNDROBIN control bit. When this bit is set, the channel that completes a transfer becomes the lowest priority.

Stopping DMA transfers. There are two ways to stop DMA transfers in progress. A single, block, and burst-block transfer may be stopped by a NMI if ENNMI is set in register DMACTL4. A burst-block transfer may be stopped by clearing the DMAEN bit.

DMA Interrupts. DMA transfers cannot be interrupted by system interrupts. NMIs (non-maskable interrupts) can interrupt the DMA controller if the ENNMI control bit is set. ISRs can be interrupted by DMA transfers. Consequently, if that is not acceptable, DMA interrupts should be disabled when executing ISRs.

Each DMA channel has its own DMAIFG flag tha tis set in any mode when the corresponding DMAxSZ register counts to 0. If the corresponding DMAIE bit and GIE bit are set, an interrupt request is generated. Similar to other multi-sourced ISRs, the ISR for DMA can take advantage of DMAIV register to reduce overhead insider the ISR. Channel 0 has the highest priority interrupt.

4 DMA Registers

Figure 7 shows a format of the DMACTL0 register that specifies triggers for channels 0 and 1. Registers DMACTL1, DMACTL2, and DMACTL3 are specifying triggers for other channels. These are DMA-wide control registers. Figure 8 shows the format of DMACTL4. Figure 9 shows the format of a channel specific control register DMAxCTL. Figure 10 shows the format of the DMAxSA register. Please note that the address registers are 32-bit long, thus supporting transfers in MSP430 microcontroller with extended ISA (20-bit addresses).

15	14	13	12	11	10	9	8
Reserved			DMA1TSEL				
r0	r0	r0	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
Reserved			DMA0TSEL				
r0	r0	r0	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

Bit	Field	Type	Reset	Description
15-13	Reserved	R	0h	Reserved. Always reads as 0.
12-8	DMA1TSEL	RW	0h	DMA 1 trigger select. These bits select the DMA transfer trigger. See the device-specific data sheet for number of channels and trigger assignment. 00000b = DMA1TRIG0 00001b = DMA1TRIG1 00010b = DMA1TRIG2 ⋮ 11110b = DMA1TRIG30 11111b = DMA1TRIG31
7-5	Reserved	R	0h	Reserved. Always reads as 0.
4-0	DMA0TSEL	RW	0h	DMA 0 trigger select. These bits select the DMA transfer trigger. See the device-specific data sheet for number of channels and trigger assignment. 00000b = DMA0TRIG0 00001b = DMA0TRIG1 00010b = DMA0TRIG2 ⋮ 11110b = DMA0TRIG30 11111b = DMA0TRIG31

Figure 7. DMACTL0. This register selects triggers for channels 0 (bits 4-0) and 1 (bits 12-8). DMACTL1 selects triggers for channels 2 and 3. DMACTL2 selects triggers for channels 4 and 5. DMACTL3 selects triggers for channels 6 and 7.

Bit	Field	Type	Reset	Description
15-3	Reserved	R	0h	Reserved. Always reads as 0.
2	DMARMWDIS	RW	0h	Read-modify-write disable. When set, this bit inhibits any DMA transfers from occurring during CPU read-modify-write operations. 0b = DMA transfers can occur during read-modify-write CPU operations. 1b = DMA transfers inhibited during read-modify-write CPU operations
1	ROUNDROBIN	RW	0h	Round robin. This bit enables the round-robin DMA channel priorities. 0b = DMA channel priority is DMA0-DMA1-DMA2 - ..... -DMA7. 1b = DMA channel priority changes with each transfer.
0	ENNMI	RW	0h	Enable NMI. This bit enables the interruption of a DMA transfer by an NMI. When an NMI interrupts a DMA transfer, the current transfer is completed normally, further transfers are stopped and DMAABORT is set. 0b = NMI does not interrupt DMA transfer. 1b = NMI interrupts a DMA transfer.

Figure 8. DMACTL4.

15	14	13	12	11	10	9	8
Reserved	DMADT			DMADSTINCR		DMASRCINCR	
r0	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
DMADSTBYTE	DMASRCBYTE	DMALEVEL	DMAEN	DMAIFG	DMAIE	DMAABORT	DMAREQ
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

Bit	Field	Type	Reset	Description
15	Reserved	R	0h	Reserved. Always reads as 0.
14-12	DMADT	RW	0h	DMA transfer mode 000b = Single transfer 001b = Block transfer 010b = Burst-block transfer 011b = Burst-block transfer 100b = Repeated single transfer 101b = Repeated block transfer 110b = Repeated burst-block transfer 111b = Repeated burst-block transfer
11-10	DMADSTINCR	RW	0h	DMA destination increment. This bit selects automatic incrementing or decrementing of the destination address after each byte or word transfer. When DMADSTBYTE = 1, the destination address increments/decrements by one. When DMADSTBYTE = 0, the destination address increments/decrements by two. The DMAxDA is copied into a temporary register and the temporary register is incremented or decremented. DMAxDA is not incremented or decremented. 00b = Destination address is unchanged. 01b = Destination address is unchanged. 10b = Destination address is decremented. 11b = Destination address is incremented.
9-8	DMASRCINCR	RW	0h	DMA source increment. This bit selects automatic incrementing or decrementing of the source address for each byte or word transfer. When DMASRCBYTE = 1, the source address increments/decrements by one. When DMASRCBYTE = 0, the source address increments/decrements by two. The DMAxSA is copied into a temporary register and the temporary register is incremented or decremented. DMAxSA is not incremented or decremented. 00b = Source address is unchanged. 01b = Source address is unchanged. 10b = Source address is decremented. 11b = Source address is incremented.
7	DMADSTBYTE	RW	0h	DMA destination byte. This bit selects the destination as a byte or word. 0b = Word 1b = Byte
6	DMASRCBYTE	RW	0h	DMA source byte. This bit selects the source as a byte or word. 0b = Word 1b = Byte
5	DMALEVEL	RW	0h	DMA level. This bit selects between edge-sensitive and level-sensitive triggers. 0b = Edge sensitive (rising edge) 1b = Level sensitive (high level)
4	DMAEN	RW	0h	DMA enable 0b = Disabled 1b = Enabled
3	DMAIFG	RW	0h	DMA interrupt flag 0b = No interrupt pending 1b = Interrupt pending
2	DMAIE	RW	0h	DMA interrupt enable 0b = Disabled 1b = Enabled
1	DMAABORT	RW	0h	DMA abort. This bit indicates if a DMA transfer was interrupt by an NMI. 0b = DMA transfer not interrupted 1b = DMA transfer interrupted by NMI
0	DMAREQ	RW	0h	DMA request. Software-controlled DMA start. DMAREQ is reset automatically. 0b = No DMA start 1b = Start DMA

Figure 9. DMAxCTL. Channel x control register (x=0, 1, 2, …).

31	30	29	28	27	26	25	24
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0
23	22	21	20	19	18	17	16
Reserved				DMAxSA			
r0	r0	r0	r0	rw	rw	rw	rw
15	14	13	12	11	10	9	8
DMAxSA							
rw	rw	rw	rw	rw	rw	rw	rw
7	6	5	4	3	2	1	0
DMAxSA							
rw	rw	rw	rw	rw	rw	rw	rw

Bit	Field	Type	Reset	Description
31-20	Reserved	R	0h	Reserved. Always reads as 0.
19-0	DMAxSA	RW	undefined	DMA source address. The source address register points to the DMA source address for single transfers or the first source address for block transfers. The source address register remains unchanged during block and burst-block transfers. There are two words for the DMAxSA register. Bits 31-20 are reserved and always read as zero. Reading or writing bits 19-16 requires the use of extended instructions. When writing to DMAxSA with word instructions, bits 19-16 are cleared.

Figure 10. DMAxSA. Please note that registers are 32-bit long to support extended architectures (20-bit address).

5 Demo Programs

In this section we will consider there implementations of a program that sends a time message every second over a UART link to the workstation as shown in Figure 11. This program will run on an MSP-EXP430F5529LP Launchpad and utilize UCA1 channel. This channel is multiplexed with the debug interface, so no additional serial connections are needed. We will provide three implementations that will utilize polling, ISR, and DMA.

Figure 11. CCStudio Terminal Showing Time Messages.

Code 3 shows a program implementation using polling. The program initializes the watchdog timer in the interval mode and the USCI_A1 as described in the header. In the main loop of the program, we wait for the WDTIFG bit to be set (line 71). Clearly the while statement in line 71

will execute hundreds of thousands times in a second, before the WDTIFG becomes set. Once the bit is set, we increment the variable keeping track of time, toggle LED1, and prepare the message to be sent over UCA1 using sprintf library function. Then the program enters a for loop where we send one-by-one character of the time message (the message should have 29 characters). Please note that inside the for loop, we use polling to check whether the UCA1TXBUF register is ready to receive a new character (line 77). So, the execution time of this portion of the main loop will be limited by the baud rate set in the program. We can roughly estimate time needed to transmit the message as follows: the total number of characters is 30, each character has 10 bits (start, 8-bit data, stop), so the total number of bits sent is 300. With the 115,200 bps baud rate, the time can be estimated to ~2.6 ms. So, the majority of time, the CPU will spend waiting on WDTIFG in line 71.

```
1  /*-------------------------------------------------------------------------------
2   * File:          Module12_D1_ts.c
3   *
4   * Function:      Sends a greeting message via a serial comm channel using polling.
5   *
6   * Description:   This program sends a time message every second as follows:
7   *                "Elapsed time is <sec> s"
8   *                via a serial communication interface, UCA1.
9   *                It toggles LED1 every second.
10  *
11  * Clocks:        ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO
12  * Board:         MSP-EXP430F5529 (Launchpad)
13  *                Communication is carried out via USCI, channel A1
14  *                that is multiplexed with the debug interface.
15  *                Baud rate: low-frequency (UCOS16=0);
16  *                1048576/115200 = ~9.1 (0x0009|0x01)
17  *
18  * Instructions: Set the following parameters in Terminal/putty/MobaXterm
19  * Port: COMx
20  * Baud rate: 115200
21  * Data bits: 8
22  * Parity: None
23  * Stop bits: 1
24  * Flow Control: None
25  *
26
27  *        MSP430F5529
28  *     -----------------
29  * /|\ |             XIN|-
30  *  |  |                | 32kHz
31  *  |--|RST         XOUT|-
32  *     |                |
33  *     |    P4.4/UCA1TXD|------------>
34  *     |                | 115200 - 8N1
35  *     |    P4.5/UCA1RXD|<------------
36  *     |            P1.0|----> LED1
37  *
```

```
 * Input:     None
 * Output:    Message displayed in Terminal/putty/MobaXterm
 * Author:    A. Milenkovic, milenkovic@computer.org
 * Date:      October 2022
 *-------------------------------------------------------------------------------*/
#include <msp430.h>
#include <stdio.h>

# define LEN 30
const char msg_header[17] = "Elapsed time is: ";
char msg[LEN];                          // Header of the message
unsigned int sec=0;                     // Variable for keeping time in seconds

// Channel A1 is multiplexed through JTAG
void USCIA1_setup(void) {
    P4SEL |= BIT4 + BIT5;   // Set USCI_A1 RXD/TXD to receive/transmit data
    UCA1CTL1 |= UCSWRST;    // Set software reset during initialization
    UCA1CTL0 = 0;           // USCI_A1 control register
    UCA1CTL1 |= UCSSEL_2;   // Clock source SMCLK

    UCA1BR0 = 0x09;         // 1048576 Hz  / 115200 lower byte
    UCA1BR1 = 0x00;         // upper byte
    UCA1MCTL |= UCBRS0;     // Modulation (UCBRS0=0x01, UCOS16=0)
    UCA1CTL1 &= ~UCSWRST;   // Clear software reset to initialize USCI state machine
}

void main(void) {
    int i;

    WDTCTL = WDT_ADLY_1000;         // WDT intrval mode, 1000 ms period
    P1DIR |= BIT0;                  // Set P1.0 to be output
    USCIA1_setup();                 // Initialize UART
    for(;;) {
        while(!(SFRIFG1&WDTIFG));      // wait for WDT
        sec++;
        P1OUT ^= BIT0;                  // Toggle LED1
        SFRIFG1 &= ~WDTIFG;             // Clear WDTIFG
        sprintf(msg, "%s%6u s\n\r", msg_header, sec);
        for(i=0; i<LEN; i++) {
           while(!(UCA1IFG&UCTXIFG));   // Wait for a TXBUF to be ready
           UCA1TXBUF = msg[i];          // TXBUF <= msg[i]
        }
    }
}

```

Code 3. Module12_D1_ts.c: polling implementation.

Code 4 shows a program implementation using ISRs. The main program initializes the watchdog timer (interval mode, 1s) and enables the interrupt by setting the WDTIE bit in the SFRIE1 register. The USCI_A1 is initialized by the USCIA1_setup() procedure. The main loop is organized as an infinite loop. The first line puts the CPU into LPM0 (low-power mode 0). The ISR of the watchdog timer will enable that the CPU resumes program execution upon exiting the

WDT_ISR(). The time variable is incremented (line 74), LED1 toggled (75), and the time message prepared using sprintf (line 76). Next, we enable the interrupt from USCI_A1 when the transmit buffer is ready and go back to the sleep mode LPM0. Please note that the next interrupt from the WDT will enable exiting the sleep mode. While in sleep mode, the USCIA1TX_ISR() will be executed every time the UCA1TXBUF is ready. Once the entire message is sent (i=LEN), the interrupts from the transmit side are disabled and the local index is set back to 0. Please note that the local index variable *i* is declared as static.

```
1  /*-------------------------------------------------------------------------------
2   * File:          Module12_D2_ts.c
3   *
4   * Function:      Sends a greeting message via a serial comm channel using ISRs.
5   *
6   * Description:   This program sends a time message every second
7   *                "Elapsed time is <sec> s"
8   *                via a serial communication interface, UCA1.
9   *                It toggles LED1 every second.
10  *
11  * Clocks:        ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO
12  * Board:         MSP-EXP430F5529 (Launchpad)
13  *                Communication is carried out via USCI, channel A1
14  *                that is multiplexed with the debug interface.
15  *                Baud rate: low-frequency (UCOS16=0);
16  *                1048576/115200 = ~9.1 (0x0009|0x01)
17  *
18  * Instructions: Set the following parameters in Terminal/putty/MobaXterm
19  * Port: COMx
20  * Baud rate: 115200
21  * Data bits: 8
22  * Parity: None
23  * Stop bits: 1
24  * Flow Control: None
25  *
26
27  *        MSP430F5529
28  *     -----------------
29  * /|\ |            XIN|-
30  *  |  |                | 32kHz
31  *  |--|RST         XOUT|-
32  *     |                |
33  *     |   P4.4/UCA1TXD|------------>
34  *     |                | 115200 - 8N1
35  *     |   P4.5/UCA1RXD|<------------
36  *     |           P1.0|----> LED1
37  *
38  * Input:     None
39  * Output:    Message displayed in Terminal/putty/MobaXterm
40  * Author:    A. Milenkovic, milenkovic@computer.org
41  * Date:      October 2022
42  *-------------------------------------------------------------------------------*/
43 #include <msp430.h>
44 #include <stdio.h>
```

```
45
46  # define LEN 30
47  const char msg_header[17] = "Elapsed time is: ";
48  char msg[LEN];                        // header of the message to be printed every
49  second
50  unsigned int sec=0;                   // variable for keeping time in seconds
51
52
53  // Channel A1 is multiplexed through JTAG
54  void USCIA1_setup(void) {
55      P4SEL |= BIT4 + BIT5;   // Set USCI_A1 RXD/TXD to receive/transmit data
56      UCA1CTL1 |= UCSWRST;    // Set software reset during initialization
57      UCA1CTL0 = 0;           // USCI_A1 control register
58      UCA1CTL1 |= UCSSEL_2;   // Clock source SMCLK
59
60      UCA1BR0 = 0x09;         // 1048576 Hz  / 115200 lower byte
61      UCA1BR1 = 0x00;         // upper byte
62      UCA1MCTL |= UCBRS0;     // Modulation (UCBRS0=0x01, UCOS16=0)
63      UCA1CTL1 &= ~UCSWRST;   // Clear software reset to initialize USCI state machine
64  }
65
66  void main(void) {
67
68      WDTCTL = WDT_ADLY_1000;         // WDT intrval mode, 1000 ms period
69      P1DIR |= BIT0;                  // Set P1.0 to be output
70      USCIA1_setup();                 // Initialize UART
71      SFRIE1 |= WDTIE;                // Enable WDT interrupt
72      for(;;) {
73        _BIS_SR(LPM0_bits + GIE);     // enter LPM0, enable interrupts
74        sec++;                        // increment time
75        P1OUT ^= BIT0;                // Toggle LED1
76        sprintf(msg, "%s%6u s\n\r", msg_header, sec);
77        UCA1IE |= UCTXIE;             // Enable transmit interrupt
78     }
79  }
80
81  #pragma vector = WDT_VECTOR
82  __interrupt void WDT_ISR(void) {
83      __bic_SR_register_on_exit(CPUOFF);   // exit LPM mode
84  }
85
86  #pragma vector = USCI_A1_VECTOR
87  __interrupt void USCIA1TX_ISR (void) {
88      static int i=0;
89      UCA1TXBUF = msg[i];         // TXBUF <= msg[i]
90      i++;
91      if (i == LEN) {
92          UCA1IE &= ~UCTXIE;      // disable interrupts
93          i = 0;                  // reset i
94      }
95  }
96
```

Code 4. Module12_D2_ts.c: ISR implementation.

Code 5 shows a program implementation using DMA, channel 0. The main program initializes the watchdog timer in the interval mode, the USCI_A1 (line 81), and the DMA, channel 0 (line 82). The WDT ISR is enabled and the CPU enters the sleep mode, LPM0. The WDT_ISR() is entered every second, the time variable is incremented (line 89), LED1 is toggled (line 90), the time message prepared (line 91), and DMA transfer is enabled (line 92). The transfer of the time message is carried out by the DMA with no intervention from the CPU.

To initialize the DMA controller, the following steps are carried out. The trigger for channel 0 is when the UCA1TXBUF is ready. This trigger is encoded in the control register DMACTL0 and it is set to 10101b (21) as shown in line 68. DMA0SA contains the starting address of the time message (msg) (line 70). DMA0DA contains the address of UCA1TXBUF (line 71). DMA0SZ is set to length of the message (LEN). The control register for channel 0 is set as follows (line 73): DMA single transfer (DMADT=0000b – default), byte-to-byte transfer (DMASBDB), increment the source address (`DMASRCINCR_3`), fixed the destination address (default), and DMA level trigger.

```
1  /*-------------------------------------------------------------------------------
2   * File:         Module12_D3_ts.c
3   *
4   * Function:     Sends a greeting message via a serial comm channel using DMA.
5   *
6   * Description:  This program sends a time message every second
7   *               "Elapsed time is <sec> s"
8   *               via a serial communication interface, UCA1.
9   *               It toggles LED1 every second.
10  *
11  * Clocks:       ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO
12  * Board:        MSP-EXP430F5529 (Launchpad)
13  *               Communication is carried out via USCI, channel A1
14  *               that is multiplexed with the debug interface.
15  *               Baud rate: low-frequency (UCOS16=0);
16  *               1048576/115200 = ~9.1 (0x0009|0x01)
17  *
18  * Instructions: Set the following parameters in Terminal/putty/MobaXterm
19  * Port: COMx
20  * Baud rate: 115200
21  * Data bits: 8
22  * Parity: None
23  * Stop bits: 1
24  * Flow Control: None
25  *
26
27  *        MSP430F5529
28  *     -----------------
29  * /|\ |            XIN|-
30  *  |  |                | 32kHz
31  *  |--|RST         XOUT|-
32  *     |                |
33  *     |   P4.4/UCA1TXD|------------>
```

```
 *     |               | 115200 - 8N1
 *     |   P4.5/UCA1RXD|<------------
 *     |           P1.0|----> LED1
 *
 * Input:     None
 * Output:    Message displayed in Terminal/putty/MobaXterm
 * Author:    A. Milenkovic, milenkovic@computer.org
 * Date:      April 2022
 *-------------------------------------------------------------------------------*/
#include <msp430.h>
#include <stdio.h>

# define LEN 30
const char msg_header[17] = "Elapsed time is: ";
char msg[LEN];                          // header of the message to be printed every
second
unsigned int sec=0;                     // variable for keeping time in seconds


// Channel A1 is multiplexed through JTAG
void USCIA1_setup(void) {
    P4SEL |= BIT4 + BIT5;   // Set USCI_A1 RXD/TXD to receive/transmit data
    UCA1CTL1 |= UCSWRST;    // Set software reset during initialization
    UCA1CTL0 = 0;           // USCI_A1 control register
    UCA1CTL1 |= UCSSEL_2;   // Clock source SMCLK

    UCA1BR0 = 0x09;         // 1048576 Hz  / 115200 lower byte
    UCA1BR1 = 0x00;         // upper byte
    UCA1MCTL |= UCBRS0;     // Modulation (UCBRS0=0x01, UCOS16=0)
    UCA1CTL1 &= ~UCSWRST;   // Clear software reset to initialize USCI state machine
}


void DMA_setup(void) {
    DMACTL0 = DMA0TSEL_21;         // DMAREQ, software trigger, trigger: UCA1TX is
ready
    DMA0SA = msg;                  // DMA0SA gets source block address
    DMA0DA = &UCA1TXBUF;           // DMA0DA gets the address of UCA1TXBUF
    DMA0SZ = LEN;                  // DMA0SZ gets the length of the string
    DMA0CTL = DMASRCINCR_3 + DMASBDB + DMALEVEL; // src-inc, byte-to-byte, level
trigger
}

void main(void) {

    WDTCTL = WDT_ADLY_1000;         // WDT interval mode, 1000 ms period
    P1DIR |= BIT0;                  // Set P1.0 to be output
    USCIA1_setup();                 // Initialize UART
    DMA_setup();                    // Initialize DMA
    SFRIE1 |= WDTIE;                // Enable WDT interrupt
    _BIS_SR(LPM0_bits + GIE);       // Enter LPM0, enable interrupts
}

#pragma vector = WDT_VECTOR
__interrupt void WDT_ISR(void) {
```

```
89      sec++;                          // Increment time
90      P1OUT ^= BIT0;                  // Toggle LED1
91      sprintf(msg, "%s%6u s\n\r", msg_header, sec);
92      DMA0CTL |= DMAEN;               // Enable DMA
93  }
94
```

Code 5. Module12_D2_ts.c: DMA implementation.

6 Exercises

Q1.

Describe main registers of a DMA channel and how they should be set.

Q2.

DMA transfers a message that consists of 20 characters received over USCI_A1 (UCA1TXBUF). The message should go into a RAM memory buffer that starts at the address 0x0800. How would you initialize DMA to carry out this transfer? Explain.

Q3.

Describe pros and cons of using DMA transfers over ISRs. Explain.

Q4.

You need to transfer a lookup table with constants from the flash memory, `unsigned int flash_lookup[64]`, into a buffer in RAM memory, `unsigned int ram_lookup[64]`. Is it possible to use a DMA channel to perform this transfer? If yes, explain how you would initialize the DMA.