

# Lecture 12: Clocks, Timers, and Flash Memory on the MSPM0

ME/AE 6705

Introduction to Mechatronics

Dr. Jonathan Rogers



# Lesson Objectives

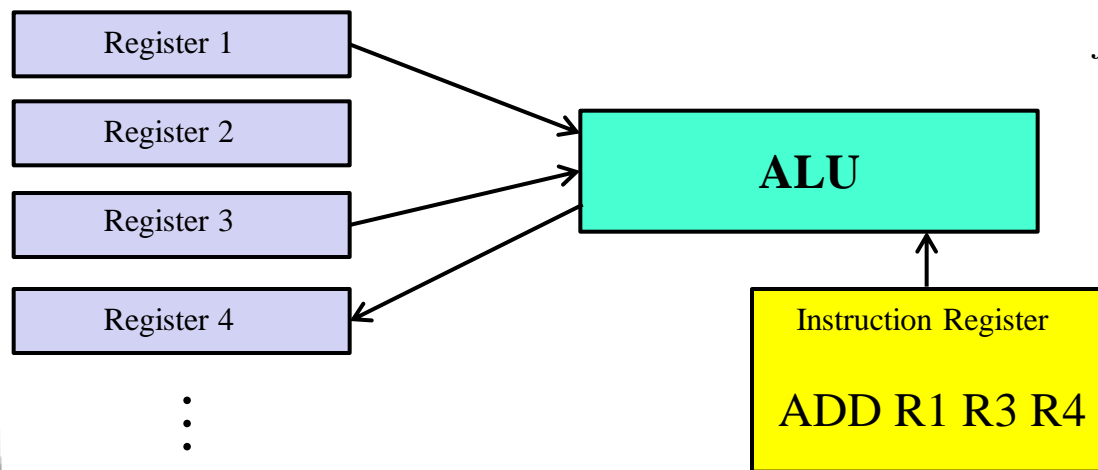
- Understand MCU clocking mechanisms
- Be able to configure the MSPM0 clock at desired frequency
- Understand operation and possible uses of MCU timers
- Be able to configure and use timer-based interrupts
- Understand memory map and hierarchy of MSPM0
- Be able to write to and read from flash memory



# Microprocessor Clocks

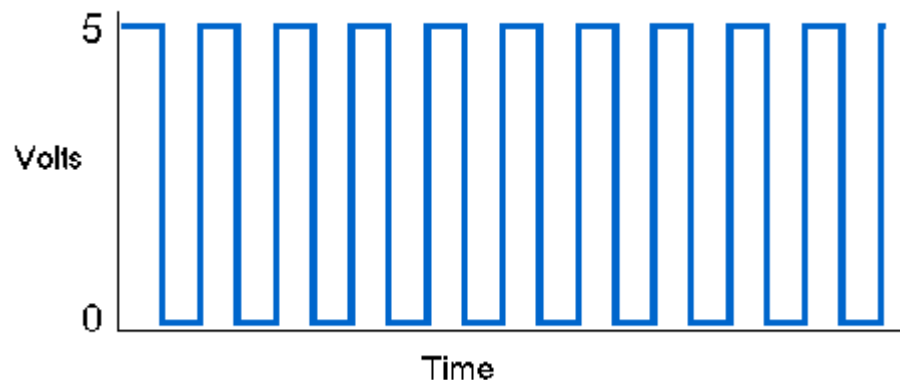
- Microprocessor is really very complex version of finite state machine
  - Machine that moves data, stores data, and performs operations on data
  - Each of these are discrete steps!

*Recall our diagram of CPU execution from Lecture 6.*



# Microprocessor Clocks

- Transitions between each of these “finite states” occurs at each processor clock cycle
  - The faster the clock cycle, the faster these transitions occur → the faster your program runs!
- Clock signals usually defined as a square wave
  - Every time a rising or falling edge occurs, processor is “clocked”



# MCU Clocks

- Processors can't do anything without a clock!
- Two ways to generate clock signals:

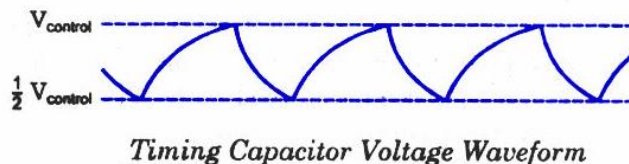


# MCU Clocks

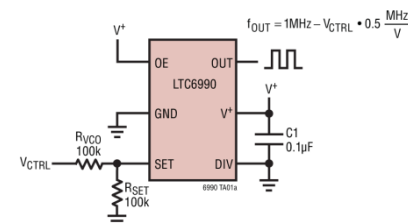
- Processors can't do anything without a clock!
- Two ways to generate clock signals:
  - Crystal oscillator – usually quartz
    - *Piezoelectric material that vibrates at very precise frequency when voltage is applied to it*
    - *Used when very precise clock is needed*
  - Digitally controlled oscillator (DCO)
    - *Circuit that uses capacitor charge rate to generate frequency*
    - *Uses less power than crystal, but less precise*



16 MHz  
quartz crystal



Voltage Controlled Oscillator with 16:1 Frequency Range



# MCU Clocks

- Question: For what applications would you want to use a crystal rather than a DCO?
- Question: For what applications would you want to use a DCO rather than a crystal?



# MSPM0 Clock System

- MSPM0 MCU has DCOs built in (internal clock)
  - System Oscillator (SYSOSC) – nominally 32 MHz
  - Low Frequency Oscillator (LFOSC) – 32 kHz
- Also has ability to use two external crystals
  - Low frequency crystal (LFXT) – 32 kHz
  - High frequency crystal (HFXT) – 4-48 MHz
- During program setup, you can choose to use either SYSOSC or an external clock source
  - Can configure SYSOSC clock rate to discrete set of frequencies (32 MHz, 24 MHz, 16 MHz, or 4 MHz)





# MSPM0 Clock System

- If you don't setup clock, it defaults to SYSOSC operating at 32 MHz
  - This is what all your programs have used so far
- Clock system is setup using Power Module Control Unit (PMCU) registers

SYSOSC configuration

Figure 2-22. SYSOSCCFG Register

31	30	29	28	27	26	25	24
RESERVED							
R/W-X							
23	22	21	20	19	18	17	16
RESERVED						FASTCPEVENT	BLOCKASYNCALL
R/W-X						R/W-1h	R/W-0h
15	14	13	12	11	10	9	8
RESERVED					DISABLE	DISABLESTOP	USE4MHZSTOP
R/W-X					R/W-0h	R/W-0h	R/W-0h
7	6	5	4	3	2	1	0
RESERVED						FREQ	
R/W-X						R/W-0h	

Main clock (MCLK) configuration

Figure 2-23. MCLKCFG Register

31	30	29	28	27	26	25	24
RESERVED							
R/W-X							
23	22	21	20	19	18	17	16
RESERVED	MCLKDEADCH	STOPCLKSTB	USELFCCLK	RESERVED			USEHSCLK
R/W-X	R/W-0h	R/W-0h	R/W-0h	R/W-X			R/W-0h
15	14	13	12	11	10	9	8
RESERVED			USEMFTICK	FLASHWAIT			
R/W-X			R/W-0h	R/W-2h			
7	6	5	4	3	2	1	0
RESERVED		UDIV		MDIV			
R/W-X		R/W-1h		R/W-0h			

*Description of these flags found in MSPM0 Technical Reference Manual.*

# Clock Source Setup Using Driverlib

- Easiest way to configure clock is to use driverlib functions
  - Allows you to select either SYSOSC or external clock (crystal)
  - Allows you to set SYSOSC frequency

```
void DL_SYSCTL_setSYSOSCFreq( DL_SYSCTL_SYSOSC_FREQ freq)
```

Driverlib call to set  
SYSOSC clock frequency

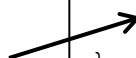
```
#include "ti_msp_dl_config.h"

void main(){

    // Set system oscillator source to 4 MHz (default is 32 MHz)
    DL_SYSCTL_setSYSOSCFreq( DL_SYSCTL_SYSOSC_FREQ_4M ) ;

}
```

*This can be called  
later in program  
with different  
frequency.*



# Clock Source Setup Using Driverlib

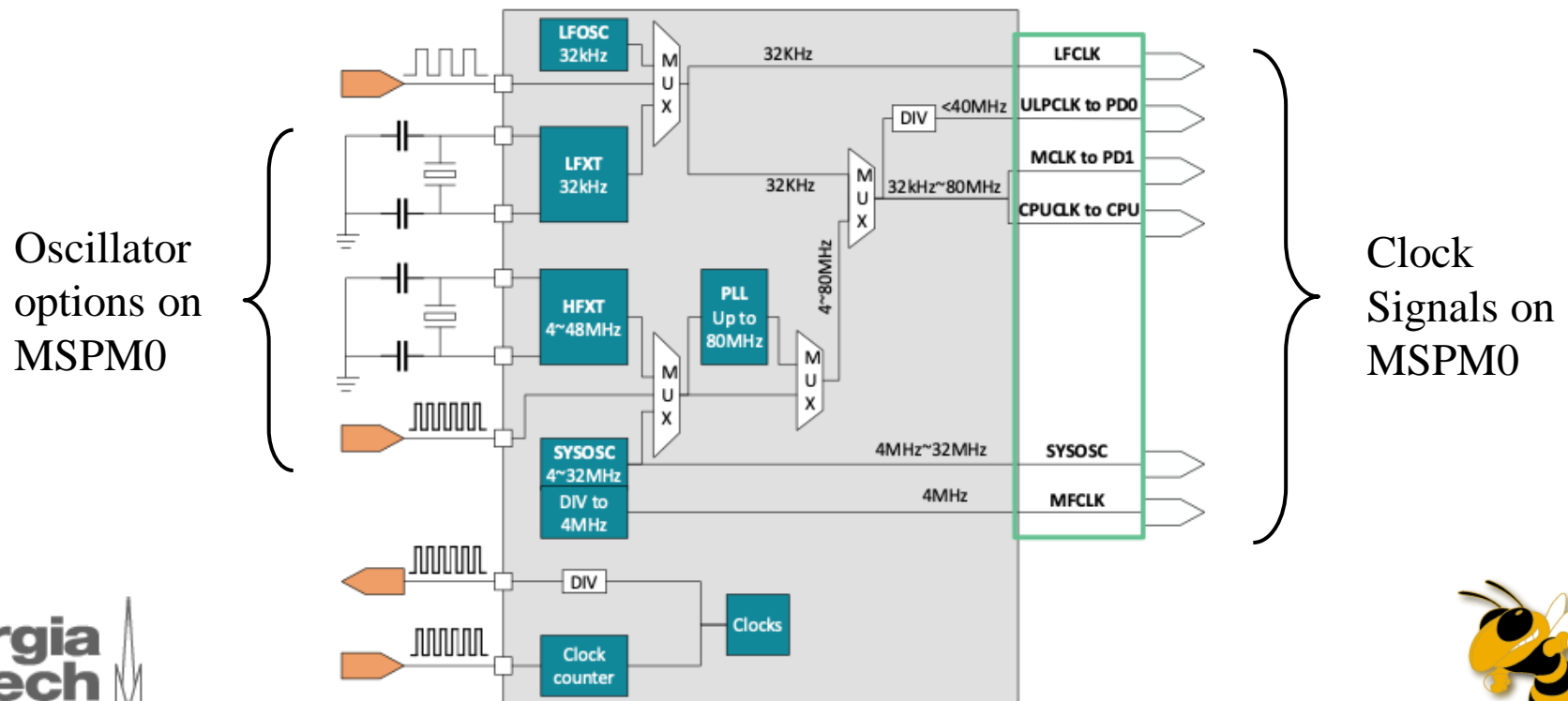
- Previous example showed how to configure SYSOSC as clock source
- If you want to configure board to use one of the external crystals (HFXT, LFXT), driverlib calls also available for this
  - See driverlib manual for example if interested in doing this

*Example code  
to tie LFCLK  
signal to  
LFXT crystal  
oscillator  
source.*

```
static const DL_SYSCTL_LFCLKConfig gLFCLKConfig = {  
    .lowCap = false,  
    .monitor = false,  
    .xt1Drive = DL_SYSCTL_LFXT_DRIVE_STRENGTH_HIGHEST,  
};  
  
...  
DL_SYSCTL_setLFCLKSourceLFXT((DL_SYSCTL_LFCLKConfig *) &gLFCLKConfig);  
DL_SYSCTL_enableExternalClock(DL_SYSCTL_CLK_OUT_SOURCE_LFCLK, DL_SYSCTL_CLK_OUT_DIVIDE_DISABLE);
```

# Clock Source vs Clock Signal

- What we have discussed so far are clock sources
- Clock signals are what MCU actually uses to clock CPU and peripheral devices



# Clock Source vs Clock Signal

- Clock source is “oscillator”
  - Oscillator is running all the time, and we can configure some things about its frequency
  - Just because it runs doesn’t mean we have to tie it into any of the MCU’s clock signals
- We get to choose which clock signals are tied to which oscillator
- We can also choose a clock divider such that **clock** runs at some lower frequency than **oscillator**

$$\text{Clock Signal Freq} = \frac{\text{Oscillator Freq}}{\text{Divider}}$$

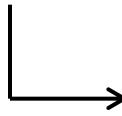
Example: Setting MCLK to run at 1/10 the rate of the SYSOSC

```
void DL_SYSCTL_setMCLKDivider(DL_SYSCTL_MCLK_DIVIDER_10);
```

# Clock Signals

- Main clock signals you need to know about on MSPM0

*Main System Clock  
(used for most  
peripherals)*



*Ultra Low Power Clock*



*Medium  
Freq. Clock*

*Low Freq. clock:  
“slow” peripherals  
like timers, ADC,  
etc.*

Clock Name	Frequency	Source	Direction
CPUCLK	32kHz~80MHz	LFCLK, SYSOSC, HFXT, PLL	CPU
MCLK			PD1
ULPCLK			PD0
SYSOSC	4~32MHz	SYSOSC	PD1/PD0
MFCLK	4MHz	SYSOSC	PD1/PD0
LFCLK	32kHz	LFCLK	PD1/PD0

Note: Driverlib and documentation sometimes refers to “Bus Clock” (BUSCLK). This is same as MCLK under normal power operation.

# Clock Setup

- Example:
  - Set SYSOSC to run at 4 MHz
  - Tie MCLK to SYSOSC with divider of 10
    - *Runs at 400 kHz*
  - Tie UART0 to MCLK (BUSCLK) with divider of 4
    - *Runs at 100 kHz*

```
DL_UART_Main_ClockConfig gUART_0ClockConfig = {  
    DL_UART_CLOCK_BUSCLK,  
    DL_UART_CLOCK_DIVIDE_RATIO_4  
};  
  
int main(void)  
{  
    DL_SYSTCL_setSYSOSCFreq( DL_SYSTCL_SYSOSC_FREQ_4M );  
    . . .  
    DL_SYSTCL_setMCLKDivider(DL_SYSTCL_MCLK_DIVIDER_10);  
    . . .  
    DL_UART_setClockConfig(UART0, &gUART_0ClockConfig);  
}
```

# Clocks: Bringing it All Together

- So to recap:
  - Oscillators: Onboard timing references that can be used by various clock signals
    - *SYSOSC, LFOSC, HFXT, LFXT*
  - Clock Signals: Provide actual clocking to CPU and peripherals
    - *Can be tied to oscillators above (some restrictions – see datasheet)*
    - *MCLK, ULPCLK, LFCLK*
  - Peripheral Clock Sources: Peripheral devices are configured to use a particular clock source
    - *Note: CPU always uses CPUCLK*



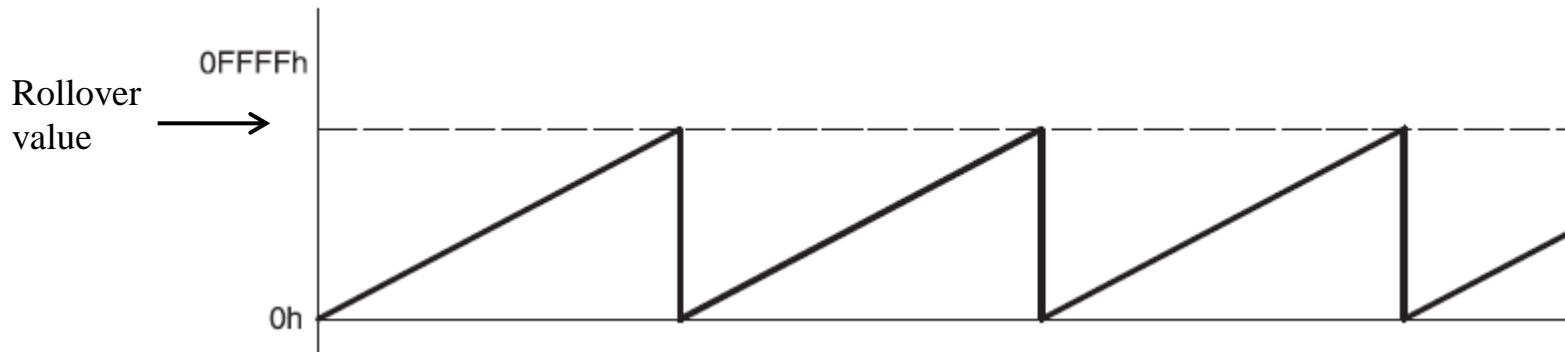
# Timers

- Timers are registers on processor that change value (increment or decrement) every clock cycle
  - Used extensively in MCU programming any time we need to generate a periodic signal
  - Examples:
    - *Pulse Width Modulation (PWM) control*
    - *Time delays*
    - *Other precisely-timed operations*
- MSPM0 has several on-board times. Today we will discuss SysTick and the general purpose timer TIMG.



# Timers

- An up-counting timer starts at zero
  - Increments every clock cycle
  - Upon reaching roll-over value, it resets to zero



- Oftentimes, timer rollover triggers an interrupt, at which we perform some action



# SysTick Timer

- SysTick timer is simplest timer available on MSPM0
- Description:
  - SysTick counts down from value we choose
    - *In 24-bit SysTick Reload Value Register, SysTick->LOAD*
  - Current SysTick value stored in SysTick Current Value Register, SysTick->Val
  - Upon reaching zero:
    - *Can be configured to trigger interrupt*
    - *Value in SysTick->Val gets set back to value in SysTick->LOAD*



# SysTick Timer

- Timer execution:
  - Initialization – set reload value (timer period)

SysTick->LOAD (24-bit reg.)

0xFC2FE8 = 16527336

SysTick->VAL (24-bit reg.)



# SysTick Timer

- Timer execution:
  - On next clock cycle, current value register is initialized to period and begins counting down with every clock cycle

SysTick->LOAD (24-bit reg.)

0xFC2FE8 = 16527336

SysTick->VAL (24-bit reg.)

0xFC2FE8 = 16527336



# SysTick Timer

- Timer execution:
  - Many clock cycles later...

SysTick->LOAD (24-bit reg.)

0xFC2FE8 = 16527336

SysTick->VAL (24-bit reg.)

0x000000 = 0



# SysTick Timer

- Timer execution:
  - After timer reaches zero, on next cycle it resets to reload (period) value and interrupt is triggered (if armed)

SysTick->LOAD (24-bit reg.)

0xFC2FE8 = 16527336

SysTick->VAL (24-bit reg.)

0xFC2FE8 = 16527336

Max period available for SysTick timer  
is 16,777,216 ( $2^{24}$ ).

SysTick Interrupt  
triggered



# SysTick Timer

- In general, we can set the clock source for a timer
- Clock source for SysTick is always MCLK
- Example: Set up SysTick timer using driverlib:

```
int main(void)
{
    DL_GPIO_initDigitalOutput(IOMUX_PINCM50);
    DL_GPIO_clearPins(GPIOB, DL_GPIO_PIN_22);
    DL_GPIO_enableOutput(GPIOB, DL_GPIO_PIN_22);

    // Initializes the SysTick period to 500.00 ms, enables the interrupt, and starts the SysTick Timer
    DL_SYSTICK_setPeriod(16000000);
    DL_SYSTICK_enableInterrupt();
    DL_SYSTICK_enable();

    while (1) {
        __WFI();
    }
}

void SysTick_Handler(void)
{
    DL_GPIO_togglePins(GPIOB, DL_GPIO_PIN_26 | DL_GPIO_PIN_22);
}
```

*Note: SysTick is one of only interrupts for which we do not need to clear interrupt flag in ISR.*





# General Purpose Timer

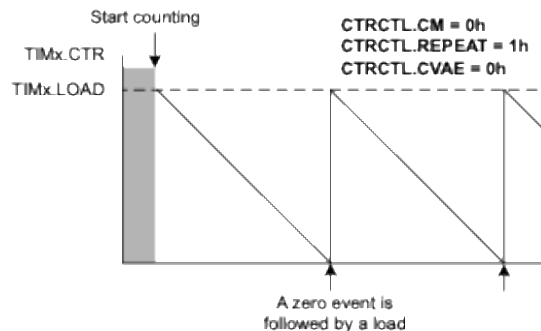
- SysTick is not very flexible
  - It can only be tied to MCLK
  - Max value may not provide us long enough delay for many peripheral functions
- TIMG is a much more useful timer system (and more complicated)
  - Can be tied to other clocks
  - Can generate PWM signals
  - Can be tied to interrupts
  - Four different timers that can be configured separately



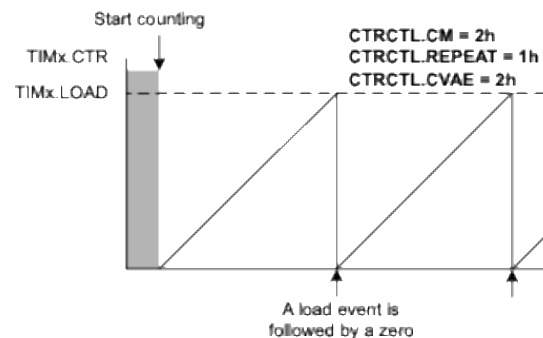
# General Purpose Timer

- TIMG can be sourced from BUSCLK (MCLK), MFCLK, or LFCLK
- Can operate in 3 modes

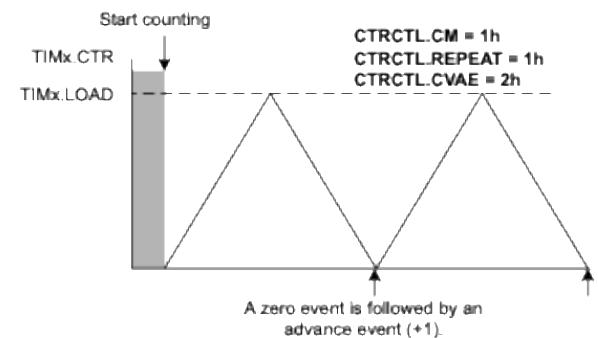
## Down-Counting Mode



## Up-Counting Mode

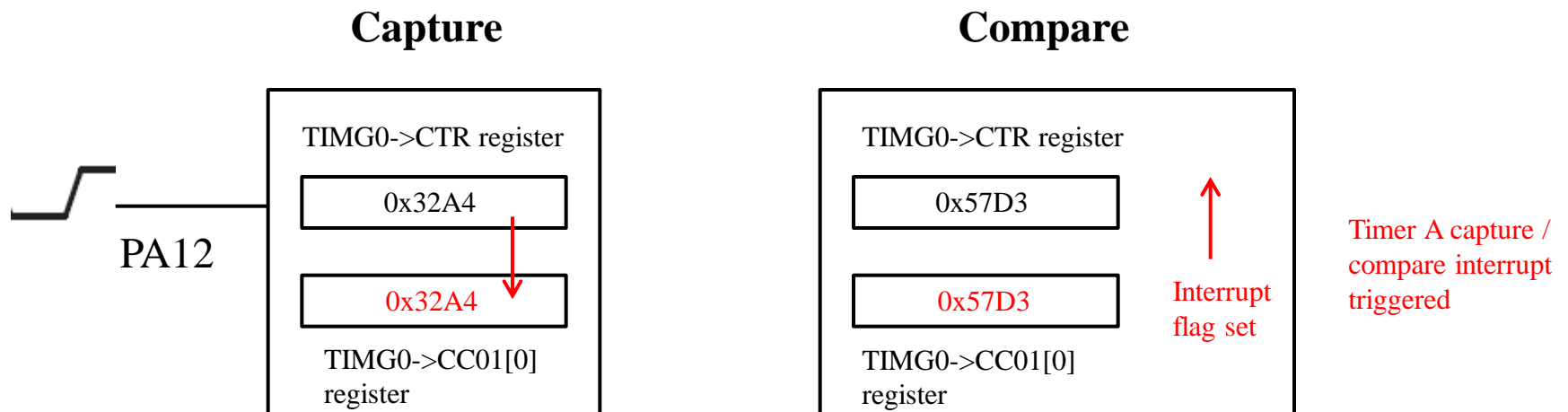


## Up-Down-Counting Mode



# General Purpose Timer

- The GP timer interrupt can be configured to activate when:
  1. The timer rolls over (counter zero)
  2. The timer reaches the value in CC\_01[0], CC\_01[1], CC\_23[0], or CC\_23[1] (compare interrupt)
  3. Capture input pins goes logic high/logic low (capture interrupt)
- Capture/compare – very common use of timers



# General Purpose Timer

- To configure TIMG, you should use structs defined in dl\_timer.h
- Populate these structs, then send pointer to them to driverlib configuration functions

## Struct to configure clock for TIMG

```
typedef struct {  
    /*! Selects timer module clock source */  
    DL_TIMER_CLOCK clockSel;  
    /*! Selects the timer module clock divide ratio  
    @ref DL_TIMER_CLOCK_DIVIDE */  
    DL_TIMER_CLOCK_DIVIDE divideRatio;  
    /*! Selects clock prescaler. Valid range 0-255 */  
    uint8_t prescale;  
} DL_Timer_ClockConfig;
```

## Struct to configure TIMG

```
typedef struct {  
    /*! One shot or Periodic mode configuration. One of  
    * @ref DL_TIMER_TIMER_MODE */  
    DL_TIMER_TIMER_MODE timerMode;  
    /*! Actual period will be period_actual=(period +1)T_TIMCLK  
    * where T_TIMCLK is the period of the timer source clock. */  
    uint32_t period;  
    /*! Start timer after configuration @ref DL_TIMER */  
    DL_TIMER startTimer;  
    /*! Generate intermediate counter interrupt  
    * @ref DL_TIMER_INTERM_INT*/  
    DL_TIMER_INTERM_INT genIntermInt;  
    /*! Counter value when intermediate interrupt should be generated. This member must be set to  
    0 when  
    * @ref genIntermInt == DL_TIMER_INTERM_INT_DISABLED */  
    uint32_t counterVal;  
} DL_Timer_TimerConfig;
```

# General Purpose Timer

- Example: Configure TIMG0
  - Recall MKCLK runs at 4 MHz
  - Prescaler of 255 means timer frequency is  $4 \text{ MHz} / 256 = 15625 \text{ Hz}$
  - Set timer load value to 1562

```
DL_TimerG_ClockConfig gTIMER_OClockConfig = {
    .clockSel = DL_TIMER_CLOCK_MFCLK,
    .divideRatio = DL_TIMER_CLOCK_DIVIDE_1,
    .prescale = 255,
};

DL_TimerG_TimerConfig gTIMER_OTimerConfig = {
    .period = 1562,
    .timerMode = DL_TIMER_TIMER_MODE_PERIODIC,
    .startTimer = DL_TIMER_STOP,
};

int main(void){
    // Turn timer peripheral on
    DL_TimerG_enablePower(TIMG0);

    // Turn on MFCLK
    DL_SYSCCTL_enableMFCLK();

    // Configure timer
    DL_TimerG_setClockConfig(TIMG0, (DL_TimerG_ClockConfig *) &gTIMER_OClockConfig);
    DL_TimerG_initTimerMode(TIMG0, (DL_TimerG_TimerConfig *) &gTIMER_OTimerConfig);
    DL_TimerG_enableInterrupt(TIMG0, DL_TIMERG_INTERRUPT_ZERO_EVENT);
    DL_TimerG_enableClock(TIMG0);
```

Divide by prescaler + 1  
per documentation

*Note – timer must  
be started!*

# General Purpose Timer

```
// Configure interrupts
NVIC_ClearPendingIRQ(TIMGO_INT_IRQn);
NVIC_EnableIRQ(TIMGO_INT_IRQn);

// Start timer
DL_TimerG_startCounter(TIMG0);

while(1){
    __WFI();
}

return 0;
}

void TIMGO_IRQHandler(void){

    // Toggle LED
    DL_GPIO_togglePins(GPIOB, DL_GPIO_PIN_26 | DL_GPIO_PIN_22);

    return;
}
```

- How often (what frequency) is TIMG0 counter incremented?
- How often (what frequency) do we expect to see the rollover interrupt triggered?

# Nonvolatile Memory

- Our current paradigm so far:
  - Use *volatile* memory to store data
  - Use *nonvolatile* memory to store our program
- This means that our program is available every time we power cycle the board, but all of our variable values and other data are wiped clean
- Consider following applications:
  - Autopilot that controls drone but records some flight data
  - MCU that records sensor data for further processing (data acquisition)



# Nonvolatile Memory

- All MCU's have nonvolatile memory to store program
  - However, most still do not allow you to store data in nonvolatile memory – it is for program only
  - One option in such a case is to use an external SD card and write to it periodically
- Recently, MCU's have become available that allow you to write user data to flash
  - Very useful for many applications
  - Not easy to use – takes some care to do so successfully



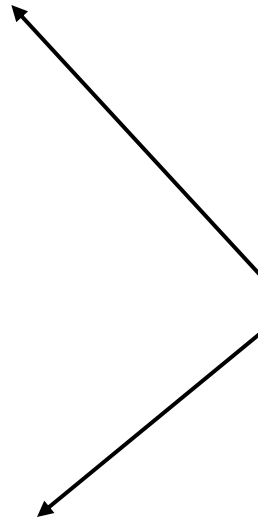


# Memory Map of MSPM0

MEMORY REGION	SUBREGION	MSPM0G3507
Code (Flash)	ECC Corrected	128KB-8B <sup>(1)</sup> 0x0000.0000 to 0x0001.FFF8
	ECC Uncorrected	0x0040.0000 to 0x0041.FFF8
SRAM (SRAM)	Parity checked	0x2010.0000 to 0x2010.7FFF
	Un-checked	0x2020.0000 to 0x2020.7FFF
	Parity code	0x2030.0000 to 0x2030.7FFF
Peripheral	Peripherals	0x4000.0000 to 0x40FF.FFFF
	Flash ECC Corrected	0x4100.0000 to 0x4102.0000
	Flash ECC Uncorrected	0x4140.0000 to 0x4142.0000
	Flash ECC code	0x4180.0000 to 0x4182.0000
	Configuration NVM(NONMAIN) ECC Corrected	512 bytes 0x41C0.0000 to 0x41C0.0200
	Configuration NVM(NONMAIN) ECC Uncorrected	0x41C1.0000 to 0x41C1.0200
	Configuration NVM(NONMAIN) ECC code	0x41C2.0000 to 0x41C2.0200
	FACTORY Corrected	0x41C4.0000 to 0x41C4.0080
	FACTORY Uncorrected	0x41C5.0000 to 0x41C5.0080
	FACTORY ECC code	0x41C6.0000 to 0x41C6.0080
Subsystem		0x6000.0000 to 0x7FFF.FFFF



Non-volatile (flash)  
memory locations



# Flash Memory Organization

- Flash Memory broken up into regions:

Table 6-2. Flash Memory Regions

Flash Memory Region	Region Contents	Executable	Used by	Programmed by
FACTORY	Device ID and other parameters	No	Application	TI only (not modifiable)
NONMAIN (Configuration NVM)	Device boot configuration (BCR and BSL)	No	Boot ROM	TI, User
MAIN (Flash Memory)	Application code and data	Yes	Application	User

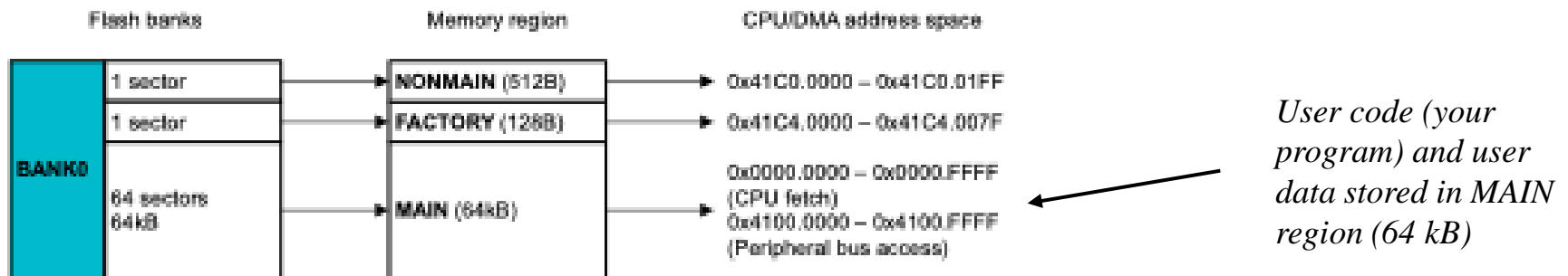
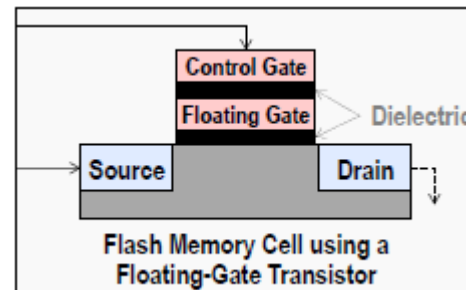


Figure 6-2. Memory Organization Example - Single Bank Configuration

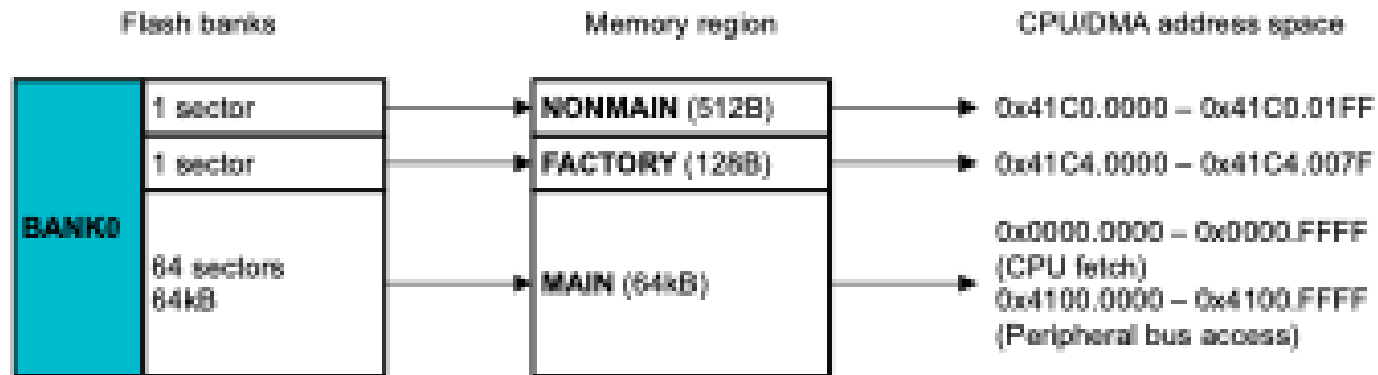


# Flash Memory Terminology

- Flash memory is not byte addressable like RAM
- Memory reads occur 8 bytes at a time
  - “Flash word” size is 64 bits
- Memory writes can only be performed for an entire 1024 byte “sector”
  - So even if you only want to write single byte, you have to write 1024 bytes to an entire sector
  - This is because flash memory works differently than SRAM



# Flash Memory Organization



**Figure 6-2. Memory Organization Example - Single Bank Configuration**

*When writing data to non-volatile memory, we want to write to flash MAIN region, which is memory addresses 0x4100.0000-0x4100.FFFF (total of 64 kB available, which is 64 sectors)*



# Flash Memory Operation

- To write to flash you must perform following steps:
  1. Enable write by “unprotecting” sector
  2. Erase entire sector (1kB)
  3. Program sector (1kB). MSPM0 Driverlib protects sector automatically after every programming operation
- Locking features are implemented to ensure that NVM does not get overwritten by accident
  - This is what protects your program from getting overwritten by itself!
  - Trying to write without unlocking will cause program to crash (enter fault condition)



# Flash Memory Operation – Driverlib

- Easiest way to use flash is through driverlib calls

```
void DL_FlashCTL_unprotectSector(FLASHCTL_Regs  
*flashctl, uint32_t addr, DL_FLASHCTL_REGION_SELECT  
regionSelect);
```

*Unlock sector for writing*

```
void DL_FlashCTL_eraseMemory(FLASHCTL_Regs  
*flashctl, uint32_t address,  
DL_FLASHCTL_COMMAND_SIZE memorySize);
```

*Erase a sector*

```
DL_FLASHCTL_COMMAND_STATUS  
DL_FlashCTL_programMemoryFromRAM( FLASHCTL_Regs  
*flashctl, uint32_t address, uint32_t *data,  
uint32_t dataSize, DL_FLASHCTL_REGION_SELECT  
regionSelect);
```

*Program a sector*



# Flash Memory Write Example

- Example: Place array of 14 chars into flash memory, use Main section, Bank 1, Sector 31

```
#define MAIN_BASE_ADDRESS (0x00001000)

char myString[] = {'F','l','a','s','h',' ',' ','E','x','a','m','p','l','e','\r','\n','\0'}; // 16 bytes

int main(void)
{
    // Unprotect sector
    DL_FlashCTL_unprotectSector(FLASHCTL, MAIN_BASE_ADDRESS, DL_FLASHCTL_REGION_SELECT_MAIN);

    // Erase sector in main memory
    DL_FlashCTL_eraseMemory(FLASHCTL, MAIN_BASE_ADDRESS, DL_FLASHCTL_COMMAND_SIZE_SECTOR);

    // Program sector in main memory. After write operation, sector is automatically protected again.
    uint32_t data_size_in_flash_words = 16*8/32; // Each flash word is 32 bits

    error_flag = DL_FlashCTL_programMemoryFromRAM(FLASHCTL, MAIN_BASE_ADDRESS, (uint32_t*) myString,
    data_size_in_flash_words, DL_FLASHCTL_REGION_SELECT_MAIN);

    ...
}
```

# Flash Memory Read Example

- Can retrieve data just like reading from any other memory location:

```
#define MAIN_BASE_ADDRESS (0x00001000)

int main(void)
{
    // Assume data is already in Main memory from running program last time
    char *string_from_flash = (char*)(MAIN_BASE_ADDRESS + 0x00400000) ;

    myPrint((char*)string_from_flash,16);  // Should print "flash example"
}
```

- NOTE: Flash memory is mirrored to two locations. **Adding 0x00400000 allows us to read data** without having to deal with error correction codes (see ref. manual for more info)
- Trying to read data directly from 0x00001000 will result in fault unless we wrote data originally with ECC



# Flash Example Code

- Let's take a look at an example flash code



# Clock and Timer Example

- Let's take a look at some example code implementing timers

