Lecture 5 – Polling and Interrupt

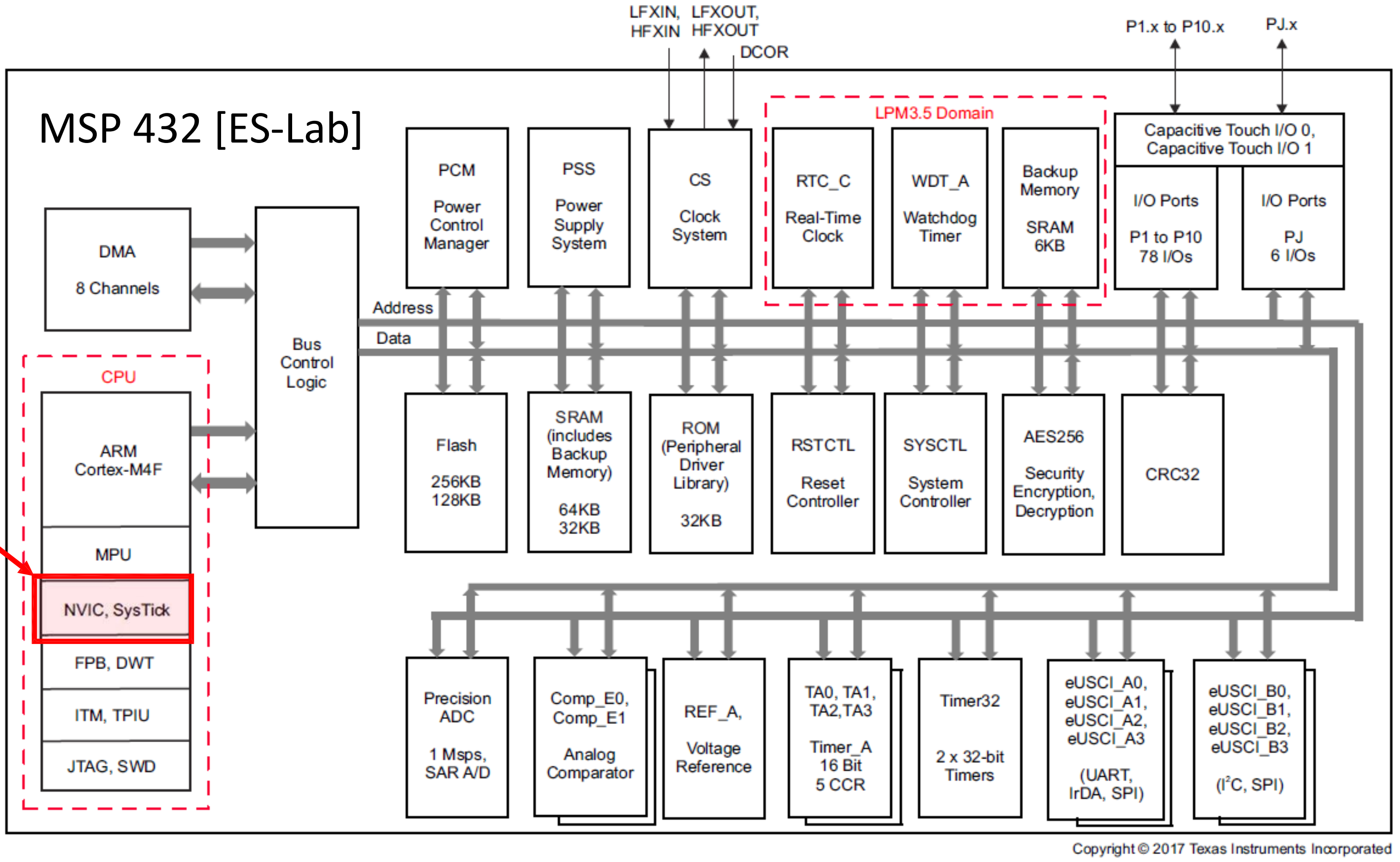
CSE 456: Embedded Systems

Interrupts

Interrupts

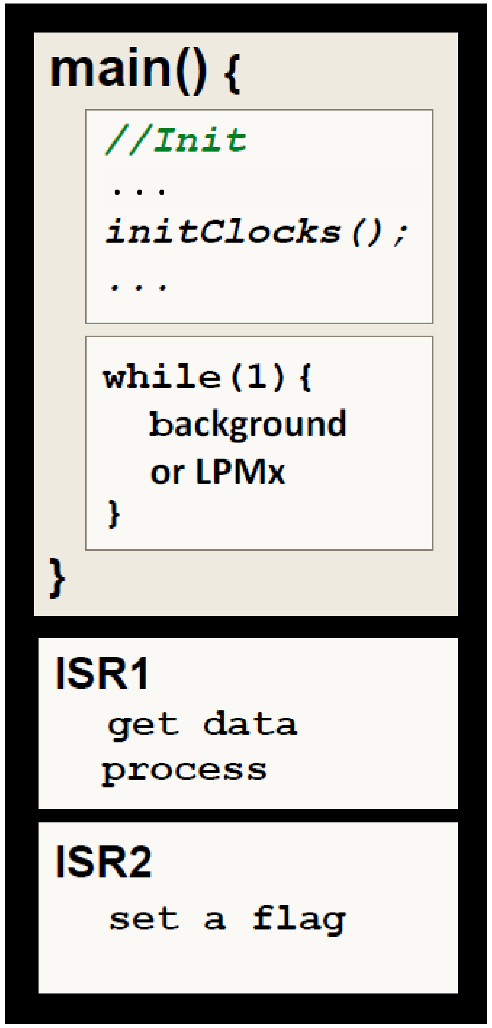
A hardware interrupt is an electronic alerting signal sent to the CPU from another component, either from an internal peripheral or from an external device.

The *Nested Vector Interrupt Controller* (NVIC) handles the processing of interrupts



MSP 432 [ES-Lab]
LFXIN, HFXIN
LFXOUT, HFXOUT
DCOR
P1.x to P10.x
PJ.x
DMA 8 Channels
Bus Control Logic
CPU
ARM Cortex-M4F
MPU
NVIC, SysTick
FPB, DWT
ITM, TPIU
JTAG, SWD
PCM Power Control Manager
PSS Power Supply System
CS Clock System
LPM3.5 Domain
RTC_C Real-Time Clock
WDT_A Watchdog Timer
Backup Memory SRAM 6KB
Capacitive Touch I/O 0, Capacitive Touch I/O 1
I/O Ports P1 to P10 78 I/Os
I/O Ports PJ 6 I/Os
Address
Data
Flash 256KB 128KB
SRAM (includes Backup Memory) 64KB 32KB
ROM (Peripheral Driver Library) 32KB
RSTCTL Reset Controller
SYSCTL System Controller
AES256 Security Encryption, Decryption
CRC32
Precision ADC 1 Msps, SAR A/D
Comp_E0, Comp_E1 Analog Comparator
REF_A, Voltage Reference
TA0, TA1, TA2,TA3 Timer_A 16 Bit 5 CCR
Timer32 2 x 32-bit Timers
eUSCI_A0, eUSCI_A1, eUSCI_A2, eUSCI_A3 (UART, IrDA, SPI)
eUSCI_B0, eUSCI_B1, eUSCI_B2, eUSCI_B3 (I²C, SPI)

Interrupts



```
main() {
    //Init
    ...
    initClocks();
    ...

    while(1){
       background
       or LPMx
    }
}

ISR1
   get data
   process

ISR2
   set a flag
```

System Initialization

- The beginning part of main() is usually dedicated to setting up your system

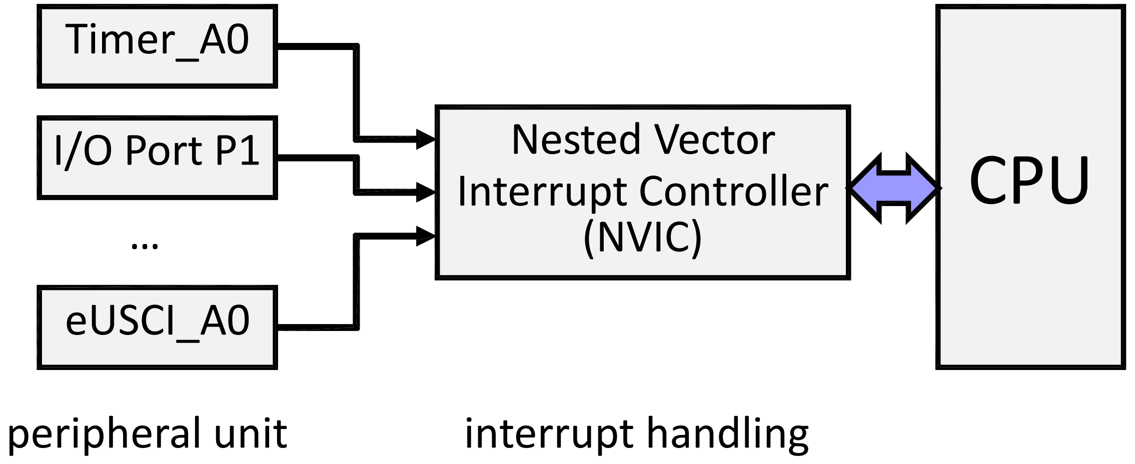
Background

- Most systems have an endless loop that runs 'forever' in the background
- In this case, 'Background' implies that it runs at a lower priority than 'Foreground'
- In **MSP432** systems, the background loop often contains a Low Power Mode (LPMx) command – this sleeps the CPU/System until an interrupt event wakes it up

Foreground

- Interrupt Service Routine (ISR) runs in response to enabled hardware interrupt
- These events may change modes in Background – such as waking the CPU out of low-power mode
- ISR's, by default, are not interruptible
- Some processing may be done in ISR, but it's usually best to keep them short

Processing of an Interrupt



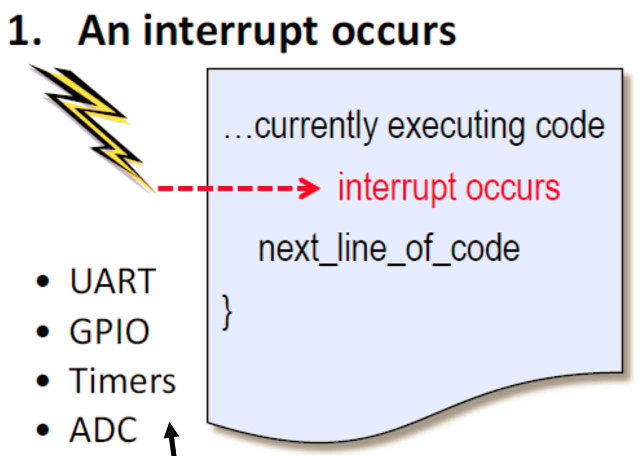
The *nested vector interrupt controller (NVIC)*

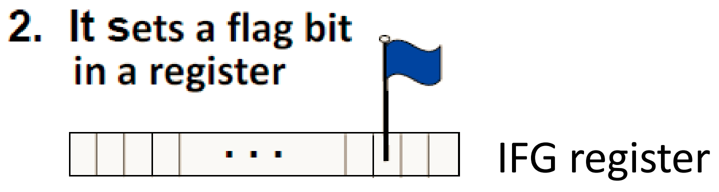
- enables and disables interrupts
- allows to individually and globally *mask interrupts* (disable reaction to interrupt), and
- registers *interrupt service routines* (ISR), sets the priority of interrupts.

Interrupt priorities are relevant if

- several interrupts happen at the same time
- the programmer does not mask interrupts in an interrupt service routine (ISR) and therefore, *preemption of an ISR* by another ISR may happen (interrupt nesting).

Processing of an Interrupt



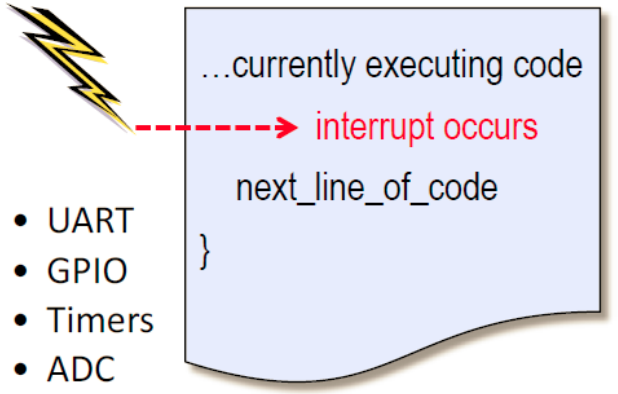


- Most peripherals can generate interrupts to provide status and information.
- Interrupts can also be generated from GPIO pins.

- When an interrupt signal is received, a corresponding bit is set in an IFG register.
- There is an such an IFG register for each interrupt source.
- As some interrupt sources are only on for a short duration, the CPU registers the interrupt signal internally.

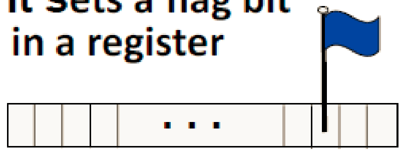
Processing of an Interrupt

1. An interrupt occurs



- UART
- GPIO
- Timers
- ADC
- Etc.

2. It sets a flag bit in a register



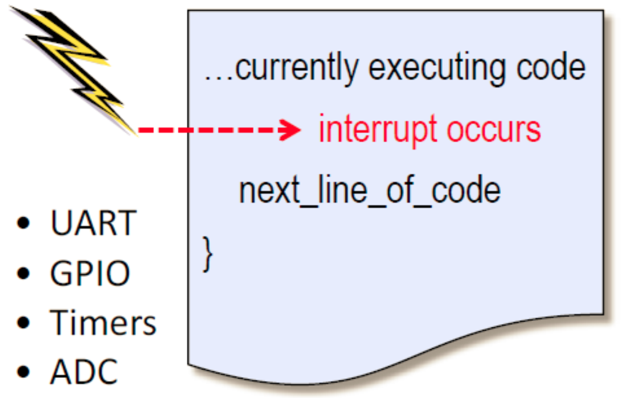
IFG register

3. CPU/NVIC acknowledges interrupt by:

- current instruction completes
- saves return-to location on stack
- mask interrupts globally
- determines source of interrupt
- calls interrupt service routine (ISR)

Processing of an Interrupt

1. An interrupt occurs



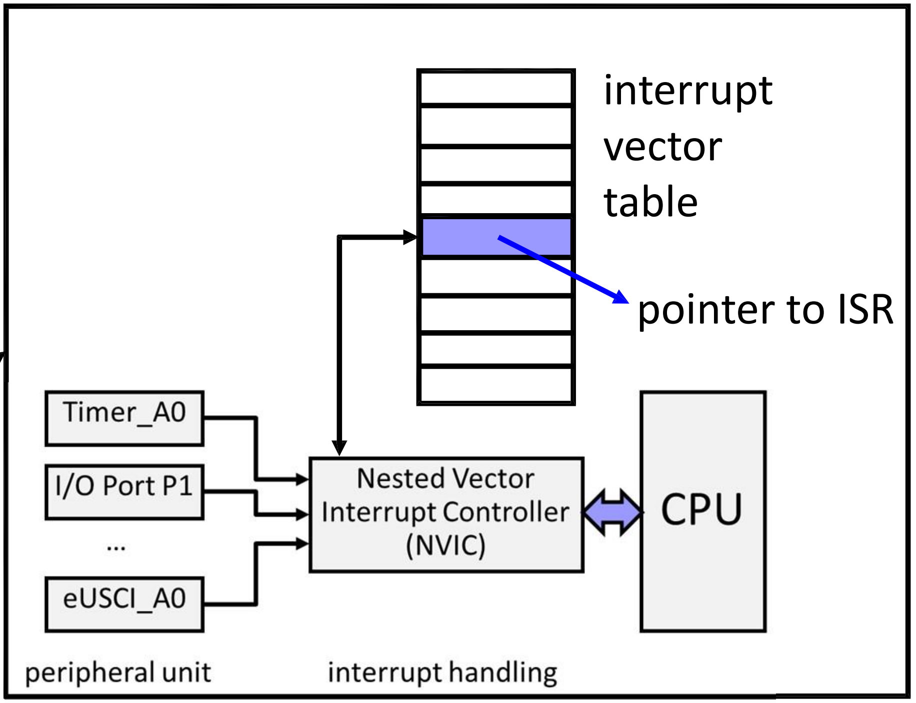
- UART
- GPIO
- Timers
- ADC
- Etc.

2. It sets a flag bit in a register

. . .
IFG register

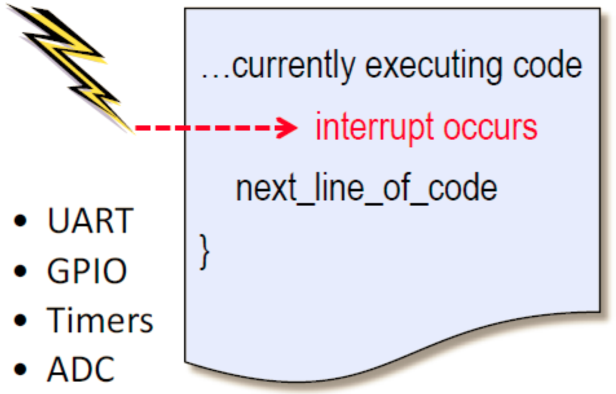
3. CPU/NVIC acknowledges interrupt by:

- current instruction completes
- saves return-to location on stack
- mask interrupts globally
- determines source of interrupt
- calls interrupt service routine (ISR)



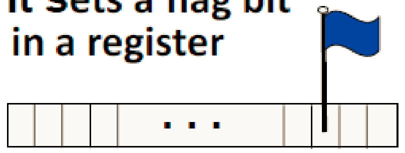
Processing of an Interrupt

1. **An interrupt occurs**



- UART
- GPIO
- Timers
- ADC
- Etc.

2. **It sets a flag bit in a register**



IFG register

3. CPU/NVIC acknowledges interrupt by:

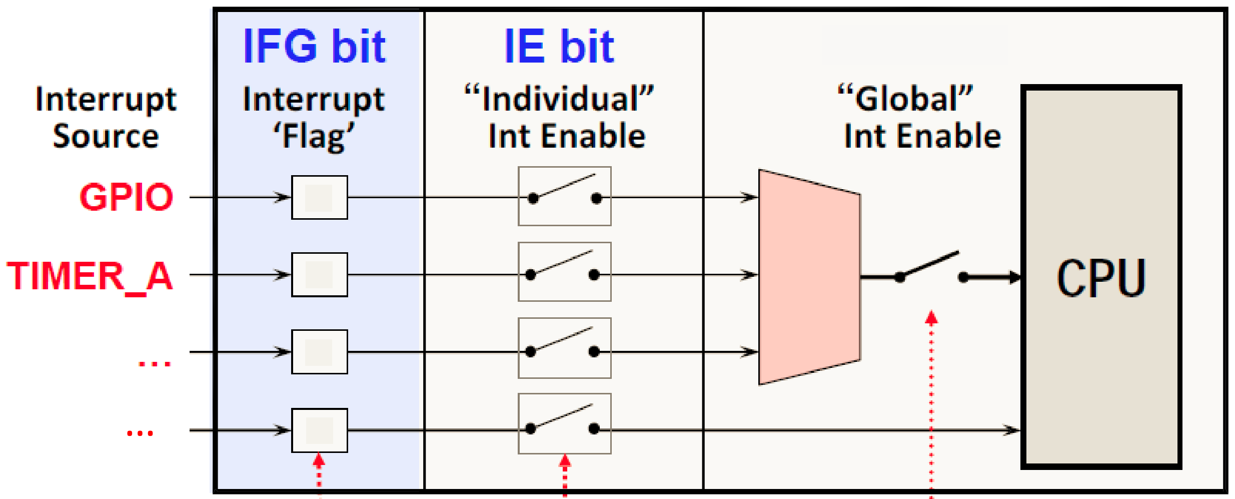
- current instruction completes
- saves return-to location on stack
- mask interrupts globally
- determines source of interrupt
- calls interrupt service routine (ISR)

4. Interrupt Service Routine (ISR):

- save context of system
- **run your interrupt's code**
- restore context of system
- (automatically) un-mask interrupts and
- continue where it left off

Processing of an Interrupt

Detailed interrupt processing flow:



get the interrupt status of the selected pin

clears the interrupt status on the selected pin

Interrupt Flag Reg (IFG)
bit set when int occurs; e.g.
GPIO_getInterruptStatus();
GPIO_clearInterruptFlag();

globally allow / dis-allow the processor to react to interrupts

Global Interrupt Enable
Enables ALL maskable interrupts

Interrupt_enableMaster();
Interrupt_disableMaster();

enable interrupt in the peripheral unit

enable interrupt in the interrupt controller

Interrupt Enable (IE); e.g.
GPIO_enableInterrupt();
GPIO_disableInterrupt();
Interrupt_enableInterrupt();

Example: Interrupt Processing

- *Port 1, pin 1* (which has a switch connected to it) is configured as an *input* with interrupts enabled and *port 1, pin 0* (which has an LED connected) is configured as an *output*.
- When the *switch is pressed*, the *LED output is toggled*.

clear interrupt flag and enable interrupt in periphery

enable interrupts in the controller (NVIC)

enter low power mode LPM3

```
int main(void)
{
    ...
    GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);
    GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN1);

    GPIO_clearInterruptFlag(GPIO_PORT_P1, GPIO_PIN1);
    GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN1);

    Interrupt_enableInterrupt(INT_PORT1);
    Interrupt_enableMaster();

    while (1) PCM_gotoLPM3();
}
```

Example: Interrupt Processing

- *Port 1, pin 1* (which has a switch connected to it) is configured as an *input* with interrupts enabled and *port 1, pin 0* (which has an LED connected) is configured as an *output*.
- When the *switch is pressed*, the *LED output is toggled*.

predefined name of ISR attached to Port 1

get status (flags) of interrupt-enabled pins of port 1

clear all current flags from all interrupt-enabled pins of port 1

check, whether pin 1 was flagged

```
void PORT1_IRQHandler(void)
{
    uint32_t status;
    status = GPIO_getEnabledInterruptStatus(GPIO_PORT_P1);
    GPIO_clearInterruptFlag(GPIO_PORT_P1, status);

    if(status & GPIO_PIN1)
    {
        GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);
    }
}
```

Polling vs. Interrupt

Similar functionality with polling:

```
int main(void)
{
    uint8_t new, old;
     ...
     GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);
     GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN1);
     old = GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN1);

     while (1)
     {
         new = GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN1);
         if (!new & old)
         {
             GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);
         }
         old = new;
     }

}
```

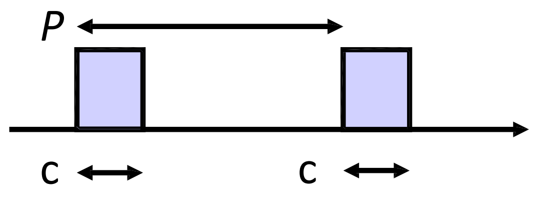
continuously get the signal at pin1 and detect falling edge

Polling vs. Interrupt

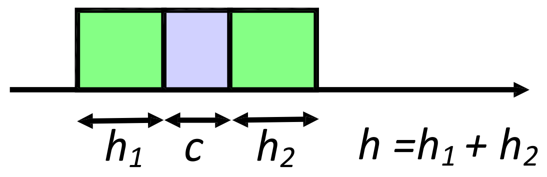
What are advantages and disadvantages?

- We *compare polling and interrupt* based on the utilization of the CPU by using a simplified timing model.
- *Definitions:*
  - *utilization u:* average percentage, the processor is busy
  - *computation c:* processing time of handling the event
  - *overhead h:* time overhead for handling the interrupt
  - *period P:* polling period
  - *interarrival time T*: minimal time between two events
  - *deadline D:* maximal time between event arrival and finishing event processing with D ≤ *T.*

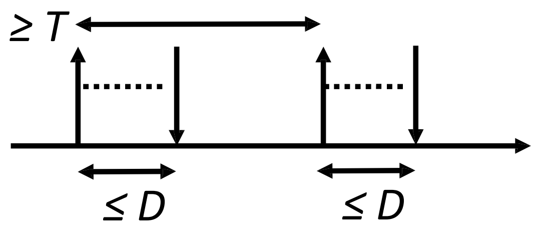
polling



interrupt



events



Polling vs. Interrupts

For the following considerations, we suppose that the interarrival time between events is T. This makes the results a bit easier to understand.

Some relations for *interrupt-based* event processing :

- The average utilization is $u_i = (h + c) / T$.
- As we need at least h+c time to finish the processing of an event, we find the following constraint: $h+c \leq D \leq T$.

Some relations for *polling-based* event processing:

- The average utilization is $u_p = c / P$.
- We need at least time $P+c$ to process an event that arrives shortly after a polling took place. The polling period P should be larger than c. Therefore, we find the following constraints: $2c \leq c+P \leq D \leq T$

Polling vs. Interrupts

Design problem: *D* and *T* are given by application requirements. *h* and *c* are given by the implementation. When to use interrupt and when polling when considering the resulting system utilization? What is the best value for the polling period P?

Case 1: If $D < c + \min(c, h)$ then event processing is not possible.

Case 2: If $2c \leq D < h+c$ then only polling is possible. The maximal period $P = D-c$ leads to the optimal utilization $u_p = c / (D-c)$.

Case 3: If $h+c \leq D < 2c$ then only interrupt is possible with utilization $u_i = (h + c) / T$.

Case 4: If $c + \max(c, h) \leq D$ then both are possible with $u_p = c / (D-c)$ or $u_i = (h + c) / T$.

Interrupt gets better in comparison to polling, if the deadline D for processing interrupts gets smaller in comparison to the interarrival time T, if the overhead h gets smaller in comparison to the computation time c, or if the interarrival time of events is only lower bounded by T (as in this case polling executes unnecessarily).

Clocks and Timers

Clocks

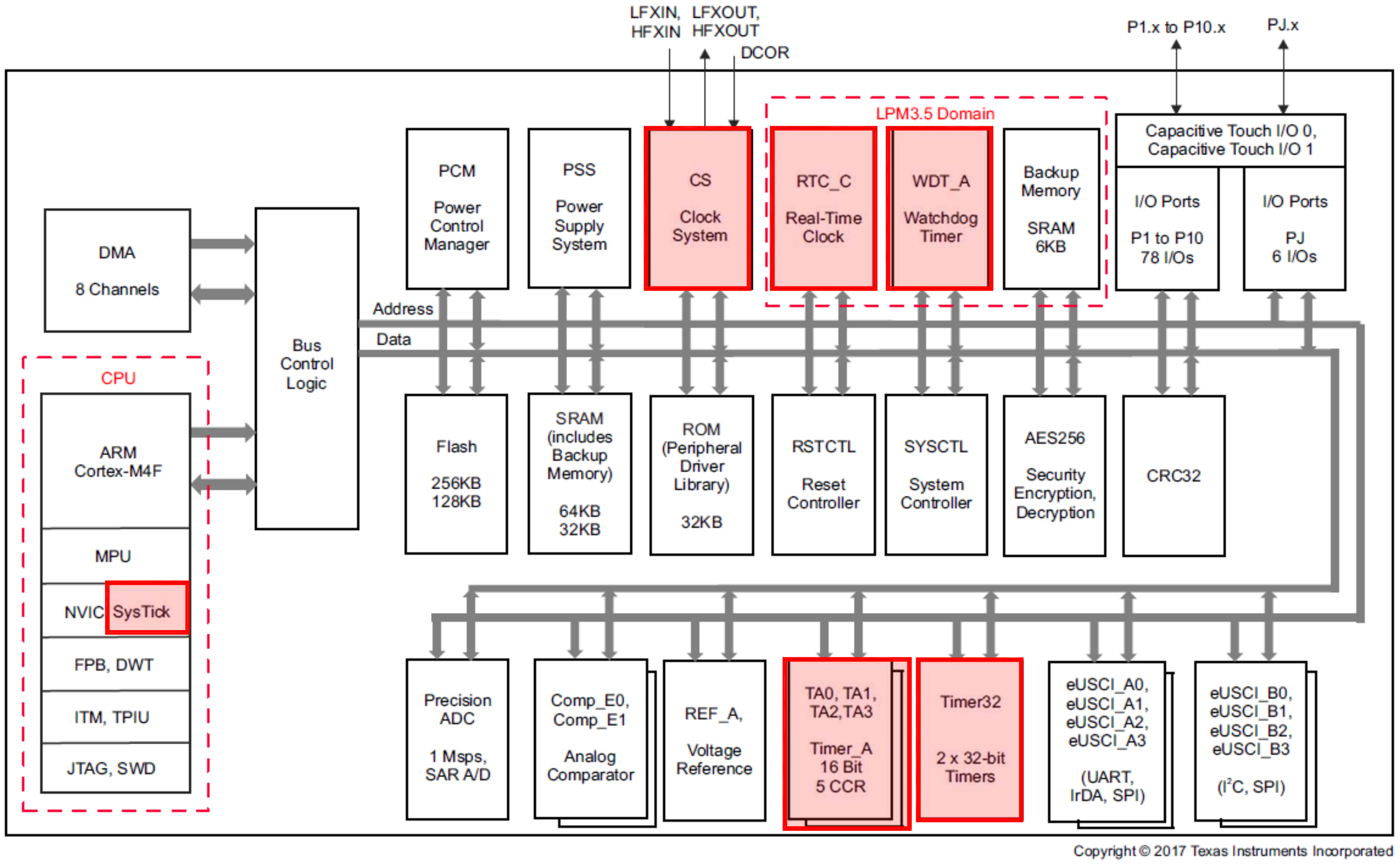
Microcontrollers usually have *many different clock sources* that have different

- frequency (relates to precision)
- energy consumption
- stability, e.g., crystal-controlled clock vs. digitally controlled oszillator

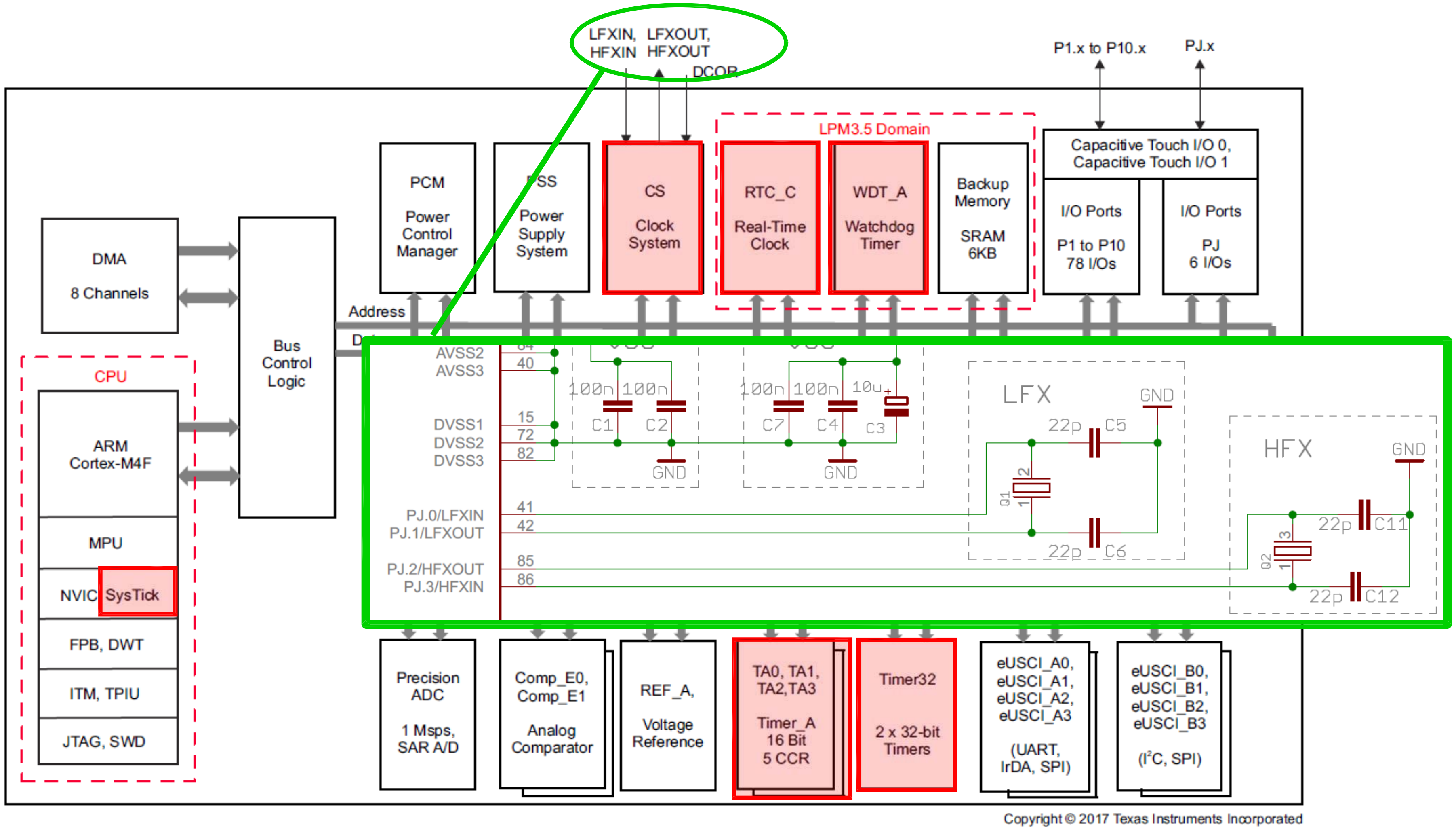
As an example, the MSP432 has the following *clock sources*:

	frequency	**precision**	**current**	**comment**
LFXTCLK	32 kHz	0.0001% / °C … 0.005% / °C	150 nA	external crystal
HFXTCLK	48 MHz	0.0001% / °C … 0.005% / °C	550 µA	external crystal
DCOCLK	3 MHz	0.025% / °C	N/A	internal
VLOCLK	9.4 kHz	0.1% / °C	50 nA	internal
REFOCLK	32 kHz	0.012% / °C	0.6 µA	internal
MODCLK	25 MHz	0.02% / °C	50 µA	internal
SYSOSC	5 MHz	0.03% / °C	30 µA	internal

Clocks and Timers MSP432


LFXIN, HFXIN
LFXOUT, HFXOUT
DCOR
P1.x to P10.x
PJ.x
LPM3.5 Domain
PCM Power Control Manager
PSS Power Supply System
CS Clock System
RTC_C Real-Time Clock
WDT_A Watchdog Timer
Backup Memory SRAM 6KB
Capacitive Touch I/O 0, Capacitive Touch I/O 1
I/O Ports P1 to P10 78 I/Os
I/O Ports PJ 6 I/Os
DMA 8 Channels
Bus Control Logic
Address
Data
CPU
ARM Cortex-M4F
MPU
NVIC
SysTick
FPB, DWT
ITM, TPIU
JTAG, SWD
Flash 256KB 128KB
SRAM (includes Backup Memory) 64KB 32KB
ROM (Peripheral Driver Library) 32KB
RSTCTL Reset Controller
SYSCTL System Controller
AES256 Security Encryption, Decryption
CRC32
Precision ADC 1 Msps, SAR A/D
Comp_E0, Comp_E1 Analog Comparator
REF_A, Voltage Reference
TA0, TA1, TA2,TA3 Timer_A 16 Bit 5 CCR
Timer32 2 x 32-bit Timers
eUSCI_A0, eUSCI_A1, eUSCI_A2, eUSCI_A3 (UART, IrDA, SPI)
eUSCI_B0, eUSCI_B1, eUSCI_B2, eUSCI_B3 (I²C, SPI)
Copyright © 2017 Texas Instruments Incorporated

Clocks and Timers MSP432



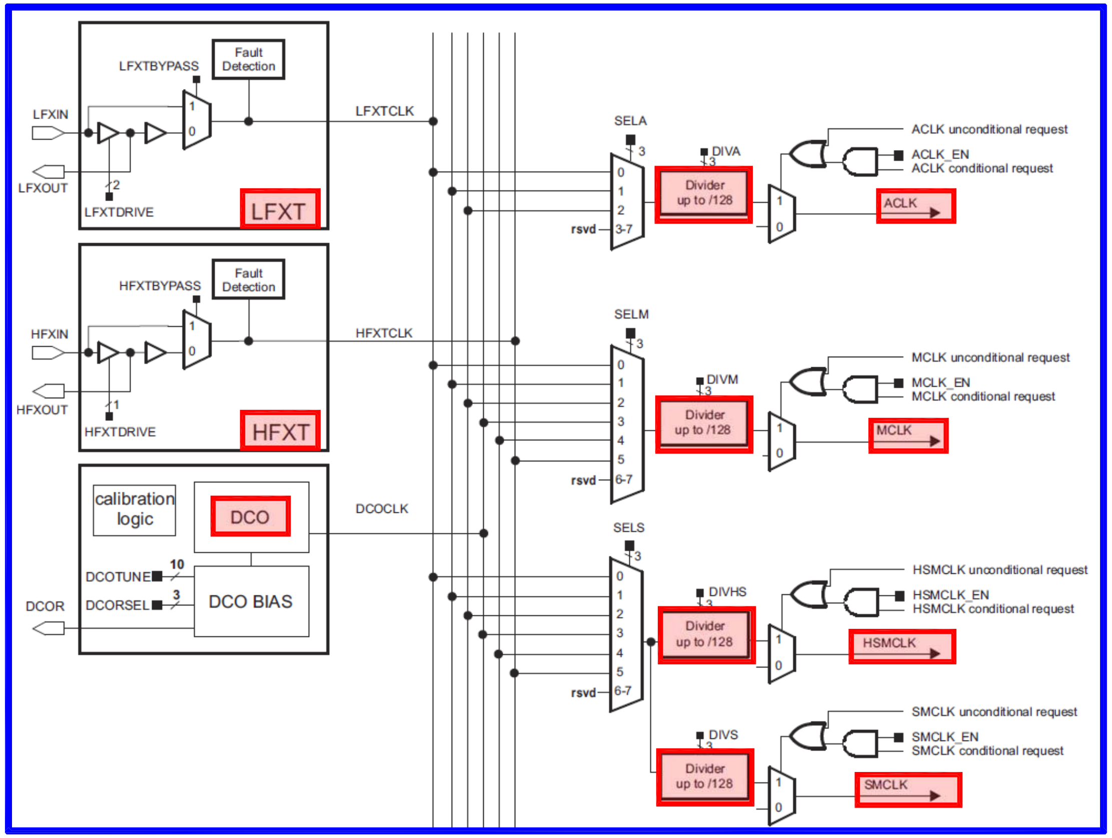
Clocks

From these basic clocks, *several internally available clock signals* are derived.

They can be used for clocking peripheral units, the CPU, memory, and the various timers.

Example MSP432 :

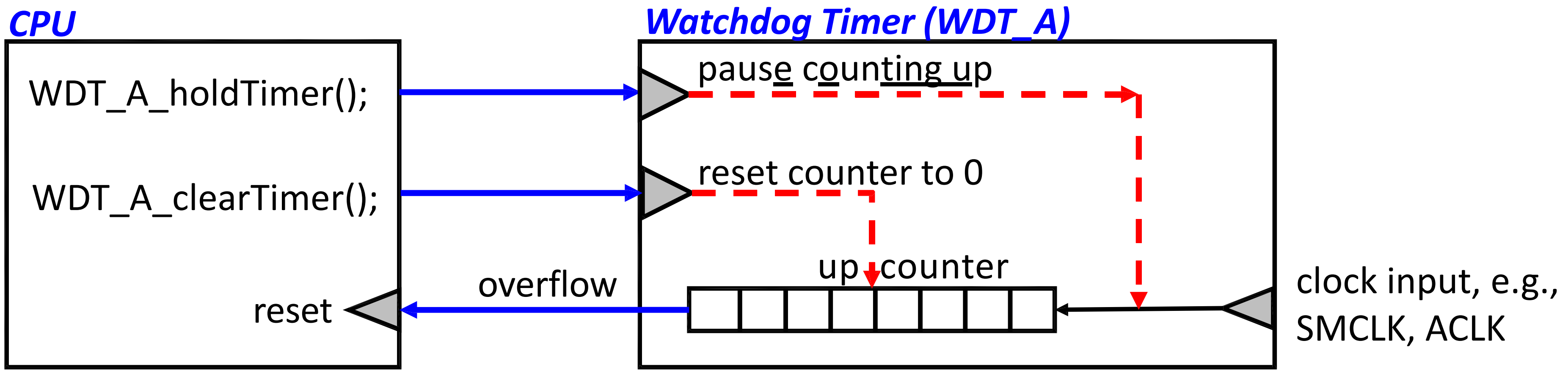
- only some of the clock generators are shown (LFXT, HFXT, DCO)
- dividers and clock sources for the internally available clock signals can be set by software



Watchdog Timer

Watchdog Timers provide system fail-safety:

- If their counter ever rolls over (back to zero), they *reset the processor*. The goal here is to prevent your system from being inactive (deadlock) due to some unexpected fault.
- To prevent your system from continuously resetting itself, the counter should be reset at appropriate intervals.



If the count completes without a restart, the CPU is reset.

SysTick MSP432

- ***SysTick*** is a simple *decrementing 24 bit counter* that is part of the NVIC controller (Nested Vector Interrupt Controller). Its clock source is MCLK and it reloads to period-1 after reaching 0.
- It's a *very simple timer*, mainly used for periodic interrupts or measuring time.

```
int main(void) {
     ...
     GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);

     SysTick_enableModule();
     SysTick_setPeriod(1500000);
     SysTick_enableInterrupt();
     Interrupt_enableMaster();

     while (1) PCM_gotoLPM0();

void SysTick_Handler(void) {
     MAP_GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0); }
```

if MCLK has a frequency of 3 MHz, an interrupt is generated every 0.5 s.

go to low power mode LP0 after executing the ISR

SysTick MSP432

Example for measuring the execution time of some parts of a program:

```
int main(void) {
     int32_t start, end, duration;
     ... SysTick_enableModule();
     SysTick_setPeriod(0x01000000);
     SysTick_disableInterrupt();


     start = SysTick_getValue();

     ... // part of the program whose duration is measured

     end = SysTick_getValue();
     duration = ((start - end) & 0x00FFFFFF) / 3;

     ...
}
```

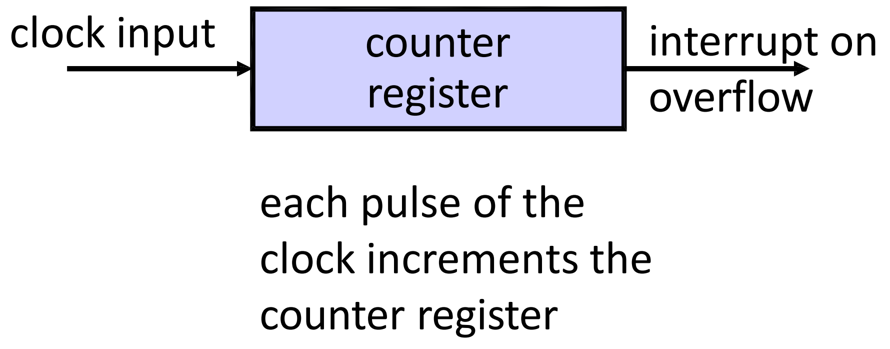
if MCLK has frequency of 3 MHz, the counter rolls over every ~6 seconds;

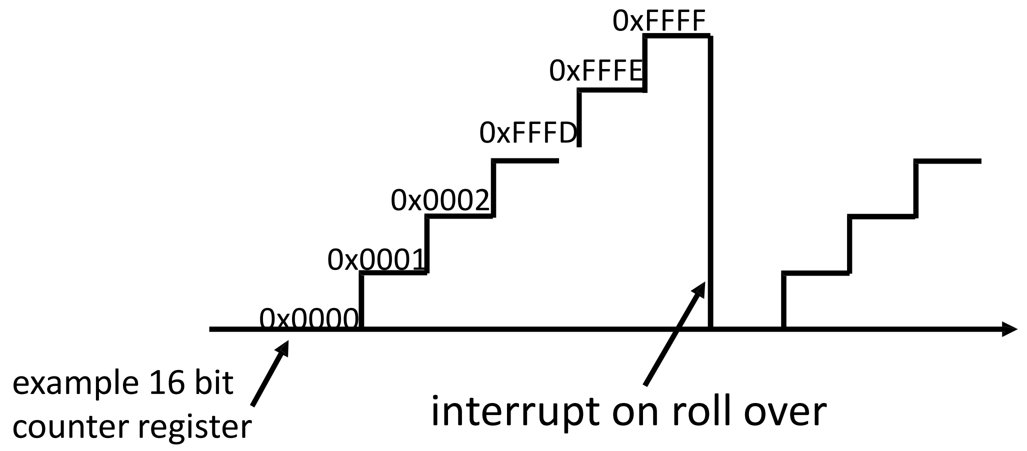
the resolution of the duration is one microsecond; the duration must not be longer than ~6 seconds; note the use of modular arithmetic if end > start; overhead for calling SysTick_getValue() is not accounted for;

Timer

Usually, *embedded microprocessors* have *several* elaborate *timers* that allow to

- *capture the current time* or time differences, triggered by hardware or software events,
- generate interrupts when a *certain time is reached* (stop watch, timeout),
- generate interrupts when *counters overflow*,
- generate *periodic interrupts*, for example in order to periodically execute tasks,
- generate *specific output signals*, for example PWM (*pulse width modulation*).

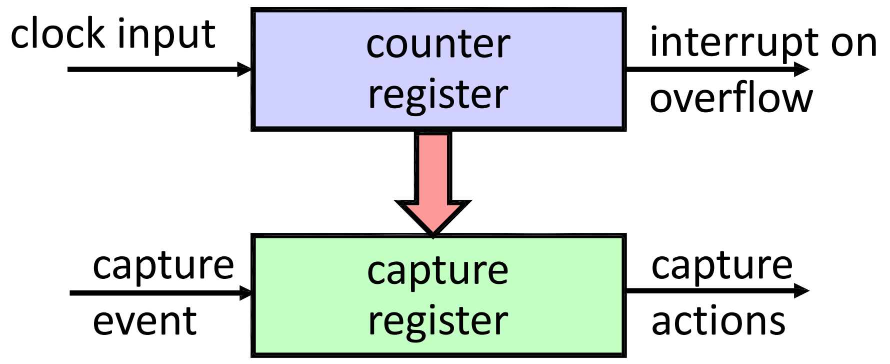




Timer

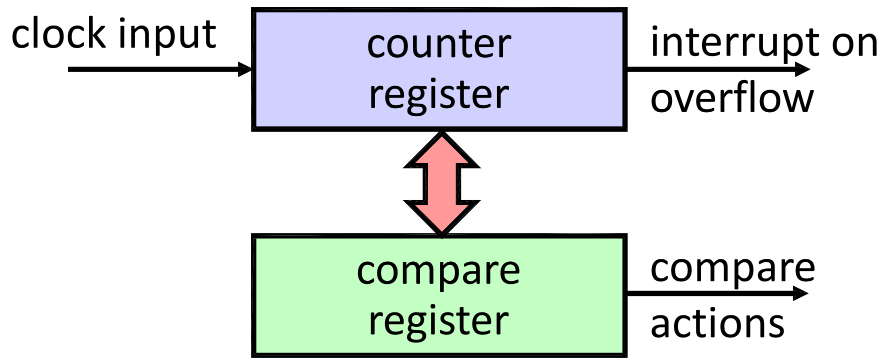
Typically, the mentioned functions are realized via *capture and compare registers*:

capture



- the value of *counter register* is stored in *capture register* at the time of the *capture event* (input signals, software)
- the value can be read by software
- at the time of the capture, further actions can be triggered (interrupt, signal)

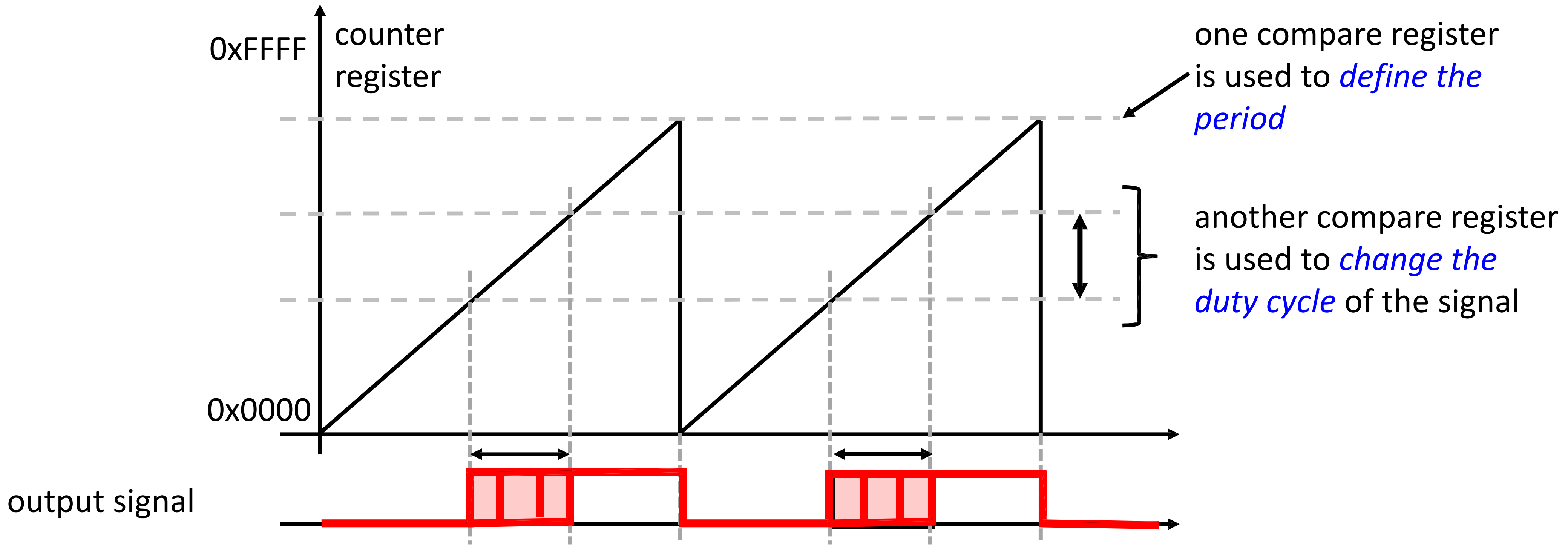
compare



- the value of the *compare register* can be set by software
- as soon as the values of the *counter and compare register are equal*, compare actions can be taken such as interrupt, signaling peripherals, changing pin values, resetting the counter register

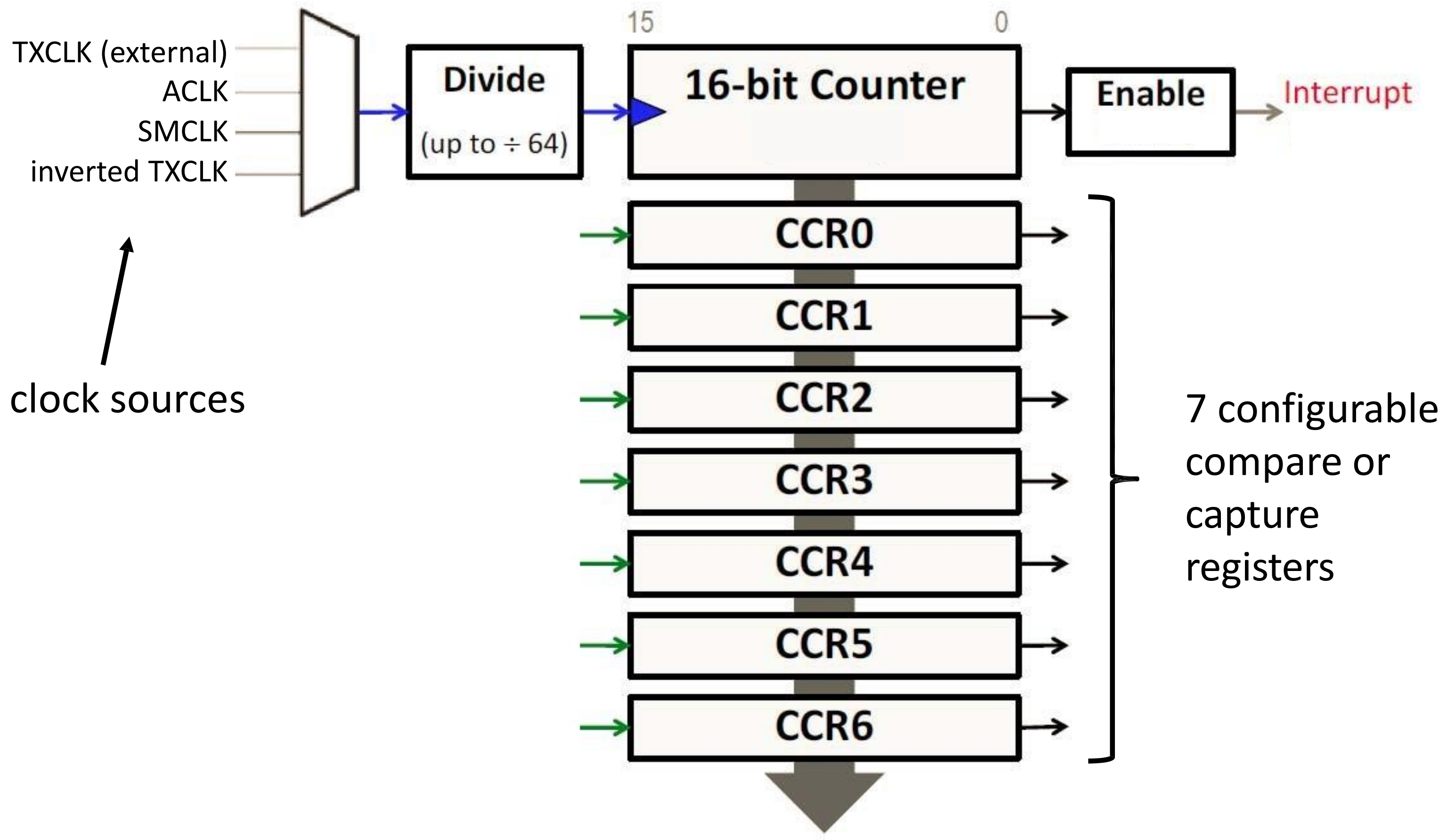
Timer

- *Pulse Width Modulation (PWM)* can be used to *change the average power* of a signal.
- The use case could be to change the speed of a motor or to modulate the light intensity of an LED.


0xFFFF
counter register
one compare register is used to *define the period*
another compare register is used to *change the duty cycle* of the signal
0x0000
output signal

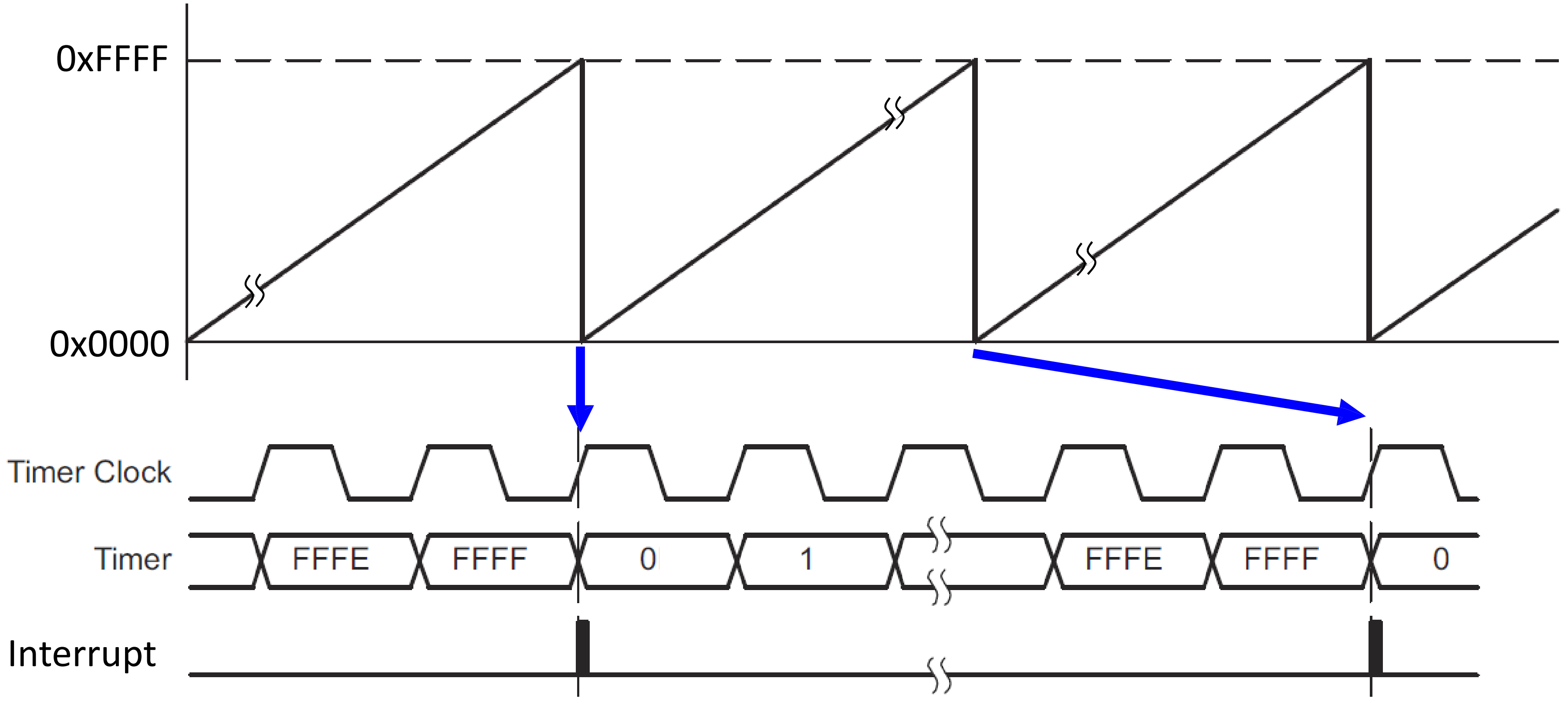
Timer Example MSP432

Example: Configure Timer in "continuous mode". *Goal:* generate periodic interrupts.



Timer Example MSP432

Example: Configure Timer in “continuous mode”. *Goal:* generate periodic interrupts.



Timer Example MSP432

Example: Configure Timer in "continuous mode". *Goal:* generate periodic interrupts.

```
int main(void) {
   ...
   const Timer_A_ContinuousModeConfig continuousModeConfig = {
      TIMER_A_CLOCKSOURCE_ACLK,
      TIMER_A_CLOCKSOURCE_DIVIDER_1,
      TIMER_A_TAIE_INTERRUPT_DISABLE,
      TIMER_A_DO_CLEAR};
   ...


   Timer_A_configureContinuousMode(TIMER_A0_BASE, &continuousModeConfig);
   Timer_A_startCounter(TIMER_A0_BASE, TIMER_A_CONTINUOUS_MODE);
   ...

   while(1) PCM_gotoLPM0(); }
```

clock source is ACLK (32.768 kHz);
divider is 1 (count frequency 32.768 kHz);
no interrupt on roll-over;

configure *continuous mode* of timer instance A0

start counter A0 in continuous mode

so far, nothing happens

only the counter is running

Timer Example MSP432

Example:

- For a *periodic interrupt*, we need to add a *compare register and an ISR*.
- The following code should be added as a definition:

```
const Timer_A_CompareModeConfig compareModeConfig = {
   TIMER_A_CAPTURECOMPARE_REGISTER_1,
   TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE,
   0,
   PERIOD};
...
Timer_A_initCompare(TIMER_A0_BASE, &compareModeConfig);
Timer_A_enableCaptureCompareInterrupt(TIMER_A0_BASE, TIMER_A_CAPTURECOMPARE_REGISTER_1);
Interrupt_enableInterrupt(INT_TA0_N);
Interrupt_enableMaster();
...
```

a first interrupt is generated *after about one second* as the counter frequency is 32.768 kHz

Timer Example MSP432

Example:

- For a *periodic interrupt*, we need to add a *compare register and an ISR*.
- The following *Interrupt Service Routine (ISR)* should be added. It is called if one of the capture/compare registers CCR1 … CCR6 raises an interrupt

```
void TA0_N_IRQHandler(void) {

   switch(TA0IV) {
      case 0x0002: //flag for register CCR1
         TA0CCR1 = TA0CCR1 + PERIOD;
         ...  // do something every PERIOD
      default: break;
   }
}
```

the register TA0IV contains the *interrupt flags* for the registers; after being read, the *highest priority interrupt* (smallest register number) is *cleared automatically*.

the register TA0CCR1 contains the *compare value* of compare register 1.

other cases in the switch statement may be used to handle other capture and compare registers