Introduction to Embedded Systems

3. Hardware Software Interface

Prof. Dr. Marco Zimmerling

NETWORKED EMBEDDED SYSTEMS LAB

Organization

Join ILIAS course:

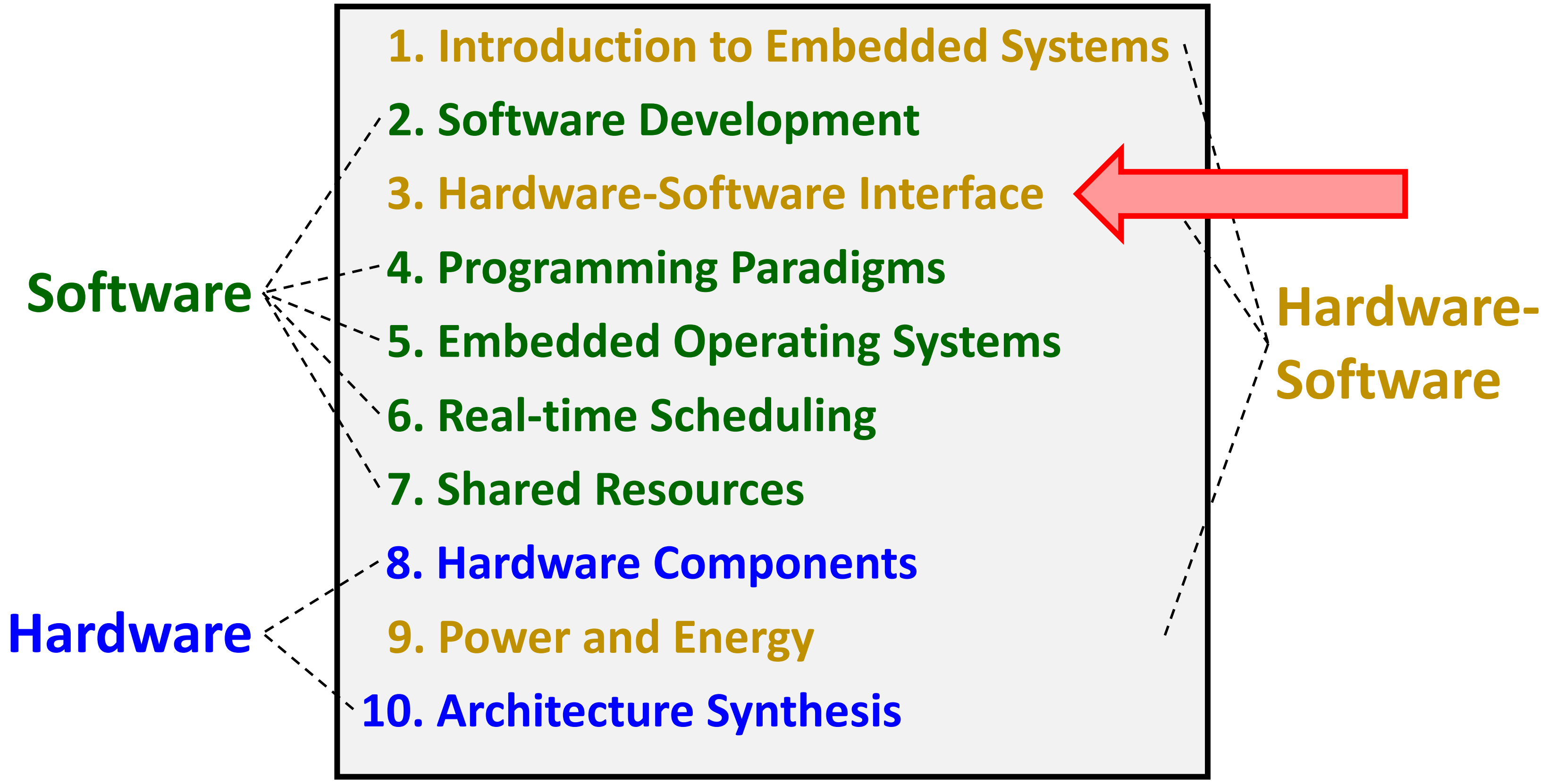
- Login: RZ username + password
- Course password: **es-0x8af**

Exercises:

- From 12:00 to 14:00 in HS 00-038 (German) and Kinohörsaal (English)
- Today (November 8):
  - First exercise sheet released + overview of tasks given during exercise session
- Next week (November 15):
  - Solutions of first exercise sheet presented during exercise session
  - Second exercise sheet released + overview of tasks given during exercise session

Exam: March 4, 2023 from 10:00 to 12:00, room TBD

Where we are …


Software
Hardware
Hardware-Software
1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis

What you will learn …

Hardware-Software Interfaces in Embedded Systems

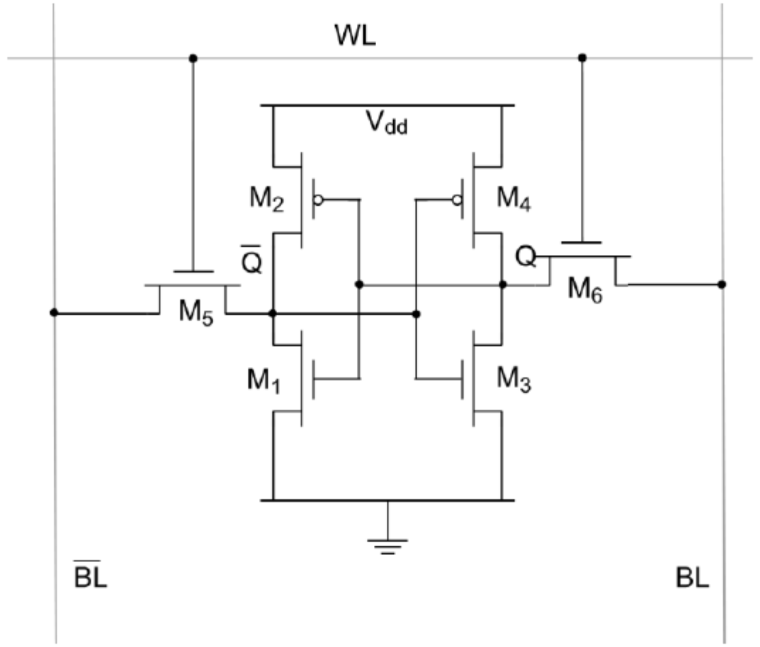
- *Storage*
  - SRAM / DRAM / Flash
  - Memory Map
- *Input and Output*
  - UART Protocol
  - Memory Mapped Device Access
  - SPI Protocol
- *Interrupts*
- *Clocks and Timers*
  - Clocks
  - Watchdog Timer
  - System Tick
  - Timer and PWM

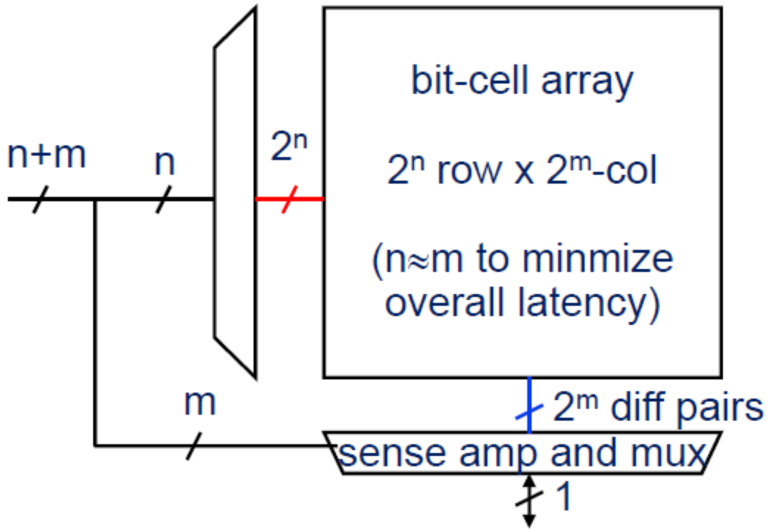
Storage

SRAM / DRAM / Flash

Static Random Access Memory (SRAM)

- *Single bit is stored in a bi-stable circuit*
- *Static Random Access Memory* is used for
  - caches
  - register file within the processor core
  - small but fast memories
- *Read:*
  1. Pre-charge all bit-lines to average voltage
  2. decode address (n+m bits)
  3. select row of cells using 2^n single-bit word lines (WL)
  4. selected bit-cells drive all bit-lines BL (2^m pairs)
  5. sense difference between bit-line pairs and read out
- *Write:*
  - select row and overwrite bit-lines using strong signals





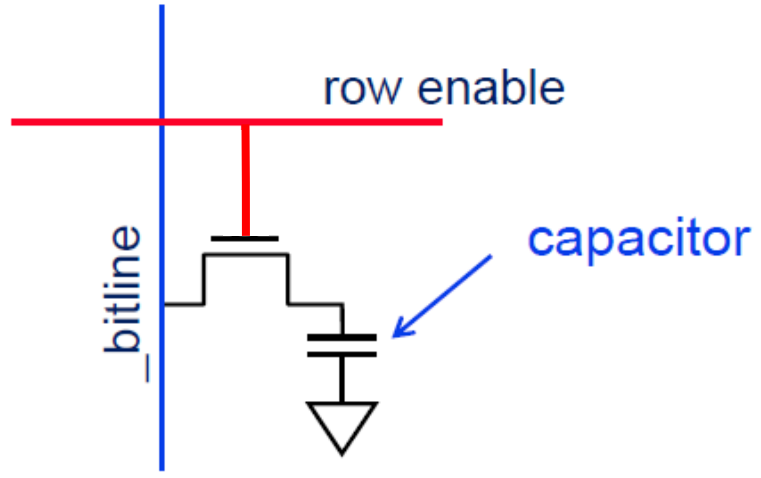
Dynamic Random Access (DRAM)

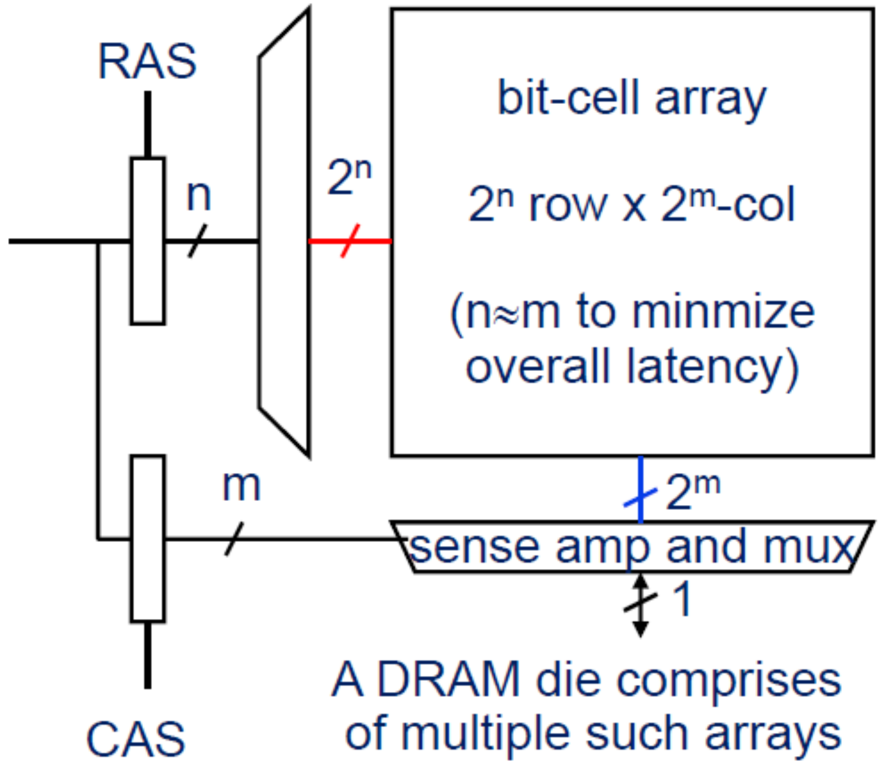
Single bit is stored as a charge in a capacitor

- Bit cell loses charge when read, bit cell drains over time
- Slower access than with SRAM due to small storage capacity in comparison to capacity of bit-line.
- Higher density than SRAM (1 vs. 6 transistors per bit)

DRAMs require *periodic refresh* of charge

- Performed by the memory controller
- Refresh interval is tens of ms
- DRAM is unavailable during refresh




RAS
n
2^n
bit-cell array
2^n row x 2^m-col
(n≈m to minmize overall latency)
m
2^m
sense amp and mux
1
A DRAM die comprises of multiple such arrays
CAS

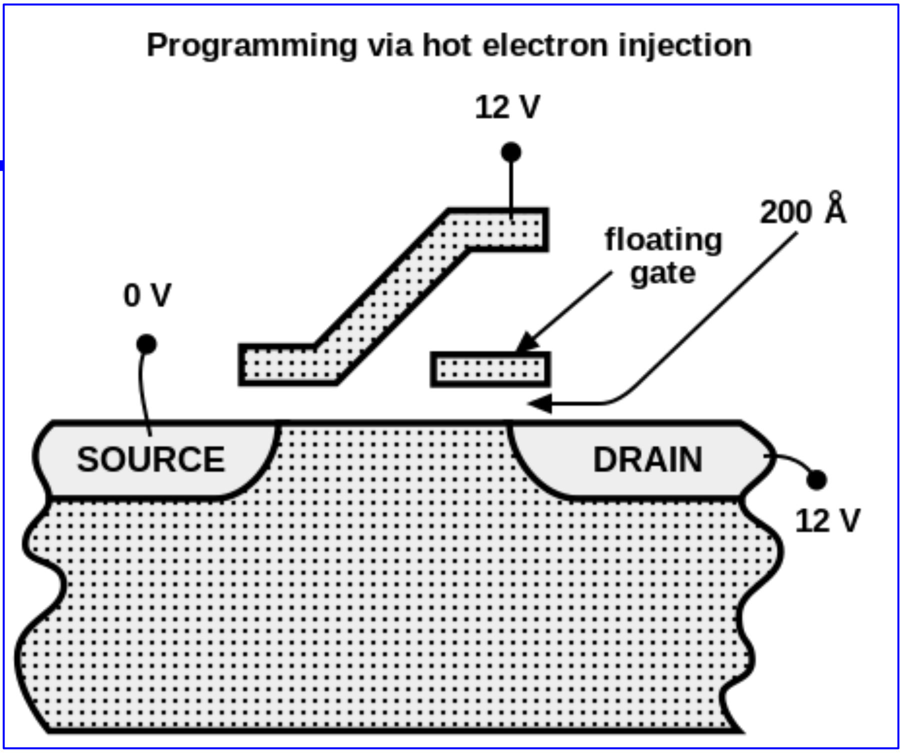
(RAS/CAS = row/column address select)

Flash Memory

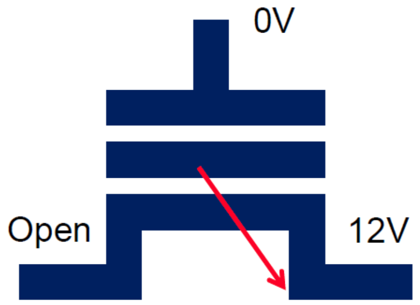
Electrically modifiable, non-volatile storage

Principle of operation:

- Transistor with a second "floating" gate
- Floating gate can trap electrons
- This results in a detectable change in threshold voltage

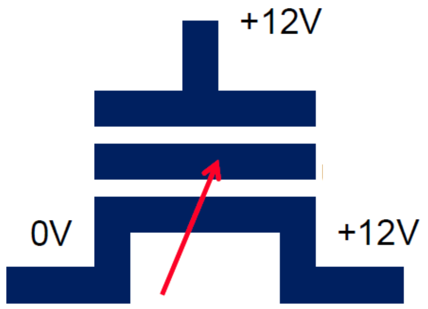


Erasing to logical "1"



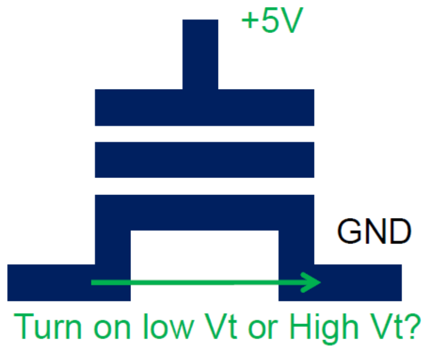
"Quantum tunneling"
Drains charge from FG

Programming (=writing) to logical "0"

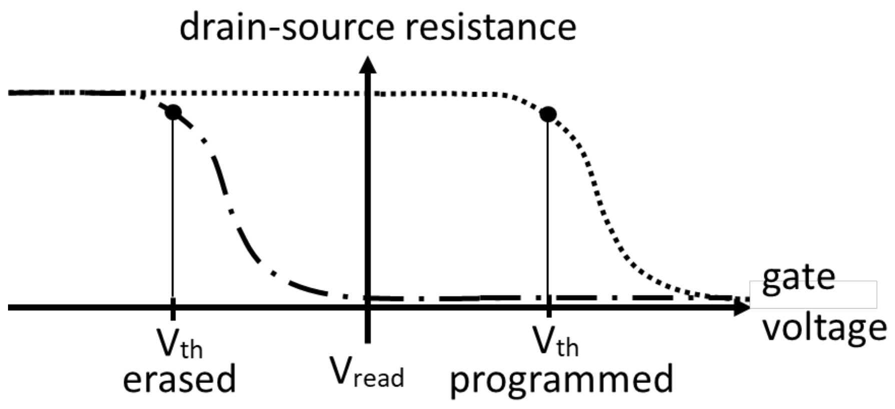


"Hot-electron injection"
traps charge in FG

Reading



Detect I_{on} to read 0 or 1



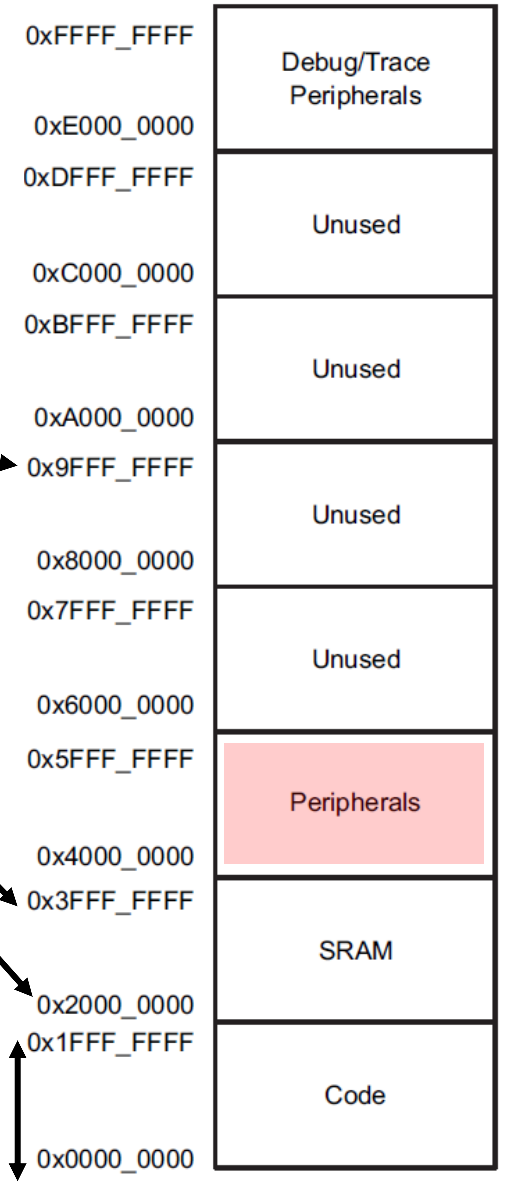
Example: Memory Map in MSP432

Memory map:

hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit

0011 1111 1111
0010 0000 0000

diff. = 0001 1111 .... 1111 →
2^{29} different addresses
capacity = 2^{29} Byte =
512 MByte



Many necessary elements are missing in the sketch below, in particular the configuration of the port (input or output, pull up or pull down resistors for input, drive strength for output).

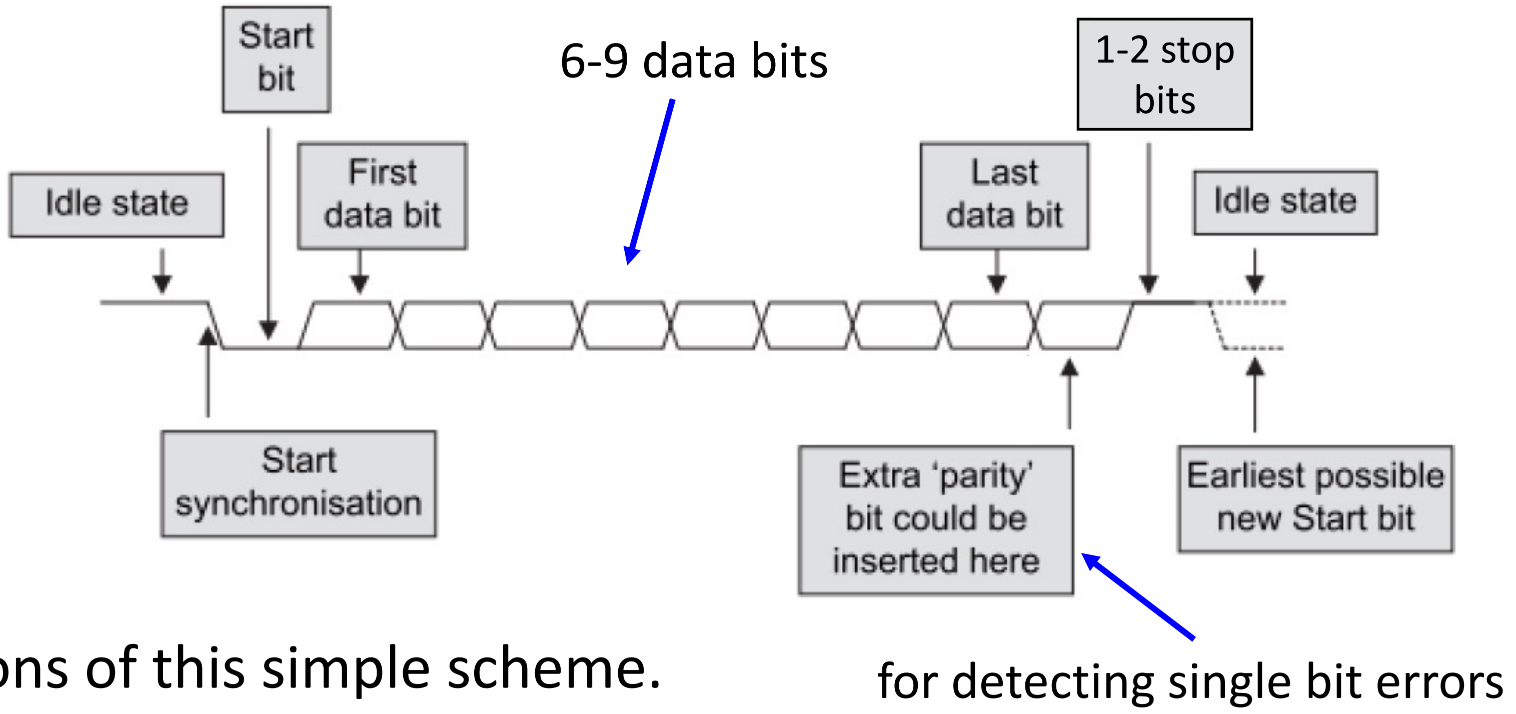
```
…
//declare p1out as a pointer to an 8Bit integer
volatile uint8_t*  p1out;

//P1OUT should point to Port 1 where LED1 is connected
p1out = (uint8_t*) 0x40004C02;

//Toggle Bit 0 (Signal to which LED1 is connected)
*p1out = *p1out ^ 0x01;
```

^ : XOR

Input and Output
UART Protocol

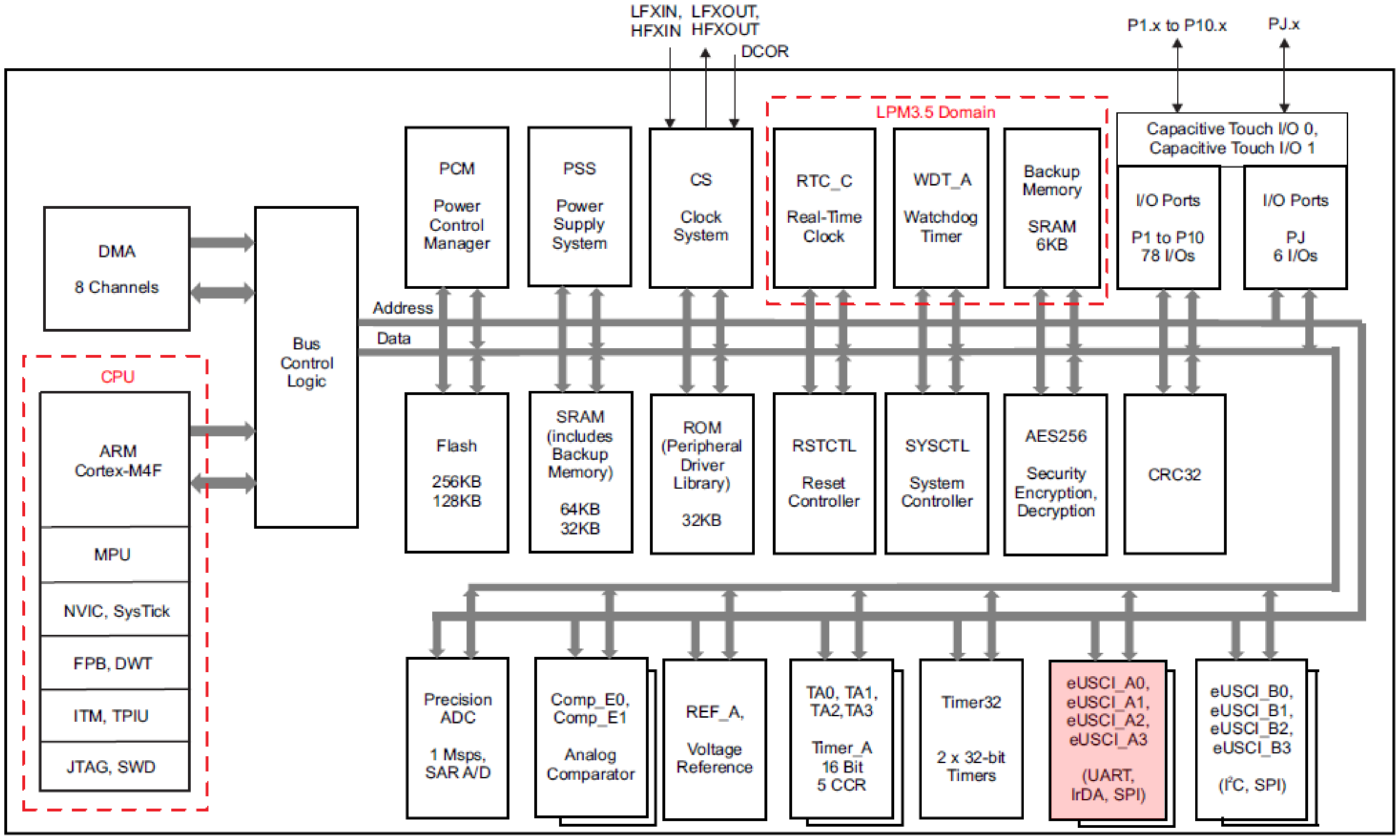
UART

- *Serial communication* of bits via a single signal, i.e. UART provides parallel-to-serial and serial-to-parallel conversion.
- Sender and receiver need to *agree on the transmission rate*.
- Transmission of a serial packet starts with a start bit, followed by data bits and finalized using a stop bit:


Start bit
6-9 data bits
1-2 stop bits
Idle state
First data bit
Last data bit
Idle state
Start synchronisation
Extra 'parity' bit could be inserted here
Earliest possible new Start bit
for detecting single bit errors

- There exist many variations of this simple scheme.

UART with MSP432

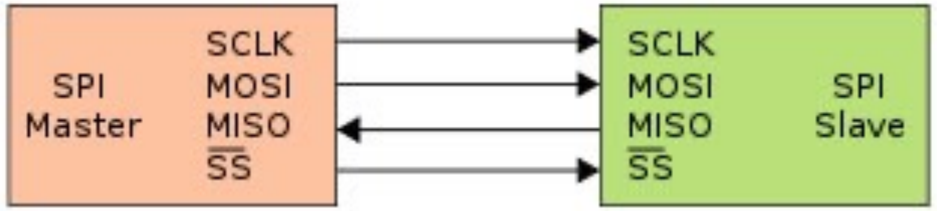


Input and Output

SPI Protocol

SPI (Serial Peripheral Interface Bus)

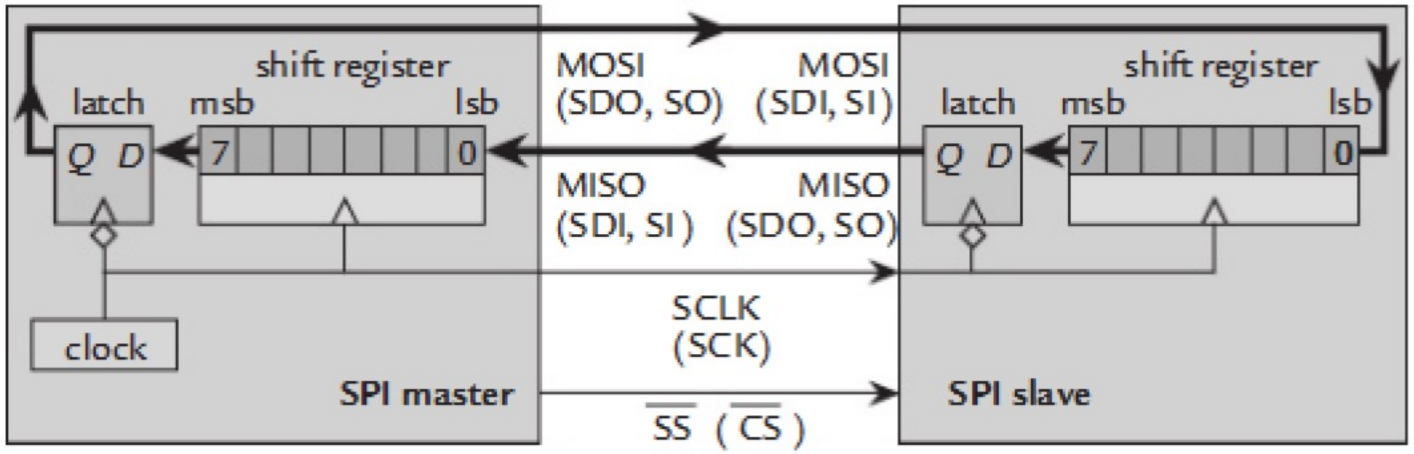
- Typically *communicate across short distances*
- *Characteristics:*
  - 4-wire synchronized (clocked) communications bus
  - supports single master and multiple slaves
  - always full-duplex: Communicates in both directions simultaneously
  - multiple Mbps transmission speeds can be achieved
  - transfer data in 4 to 16 bit serial packets
- *Bus wiring:*
  - MOSI (Master Out Slave In) – carries data out of master to slave
  - MISO (Master In Slave Out) – carries data out of slave to master
  - Both MOSI and MISO are active during every transmission
  - $\overline{\text{SS}}$ (or CS) – signal to select each slave chip
  - System clock SCLK – produced by master to synchronize transfers



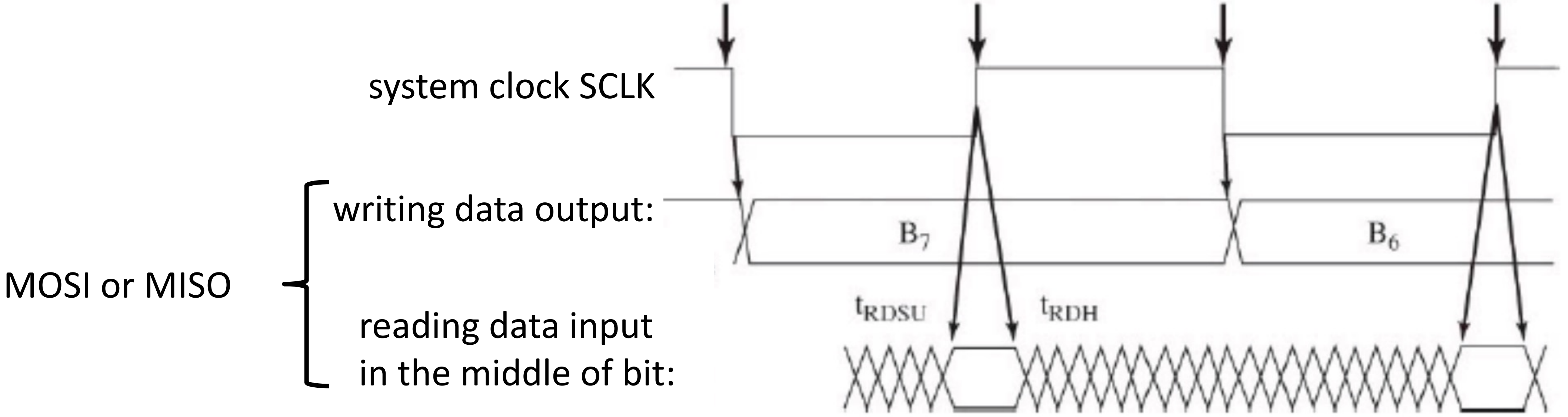
SPI (Serial Peripheral Interface Bus)

More detailed circuit diagram:

- details vary between different vendors and implementations

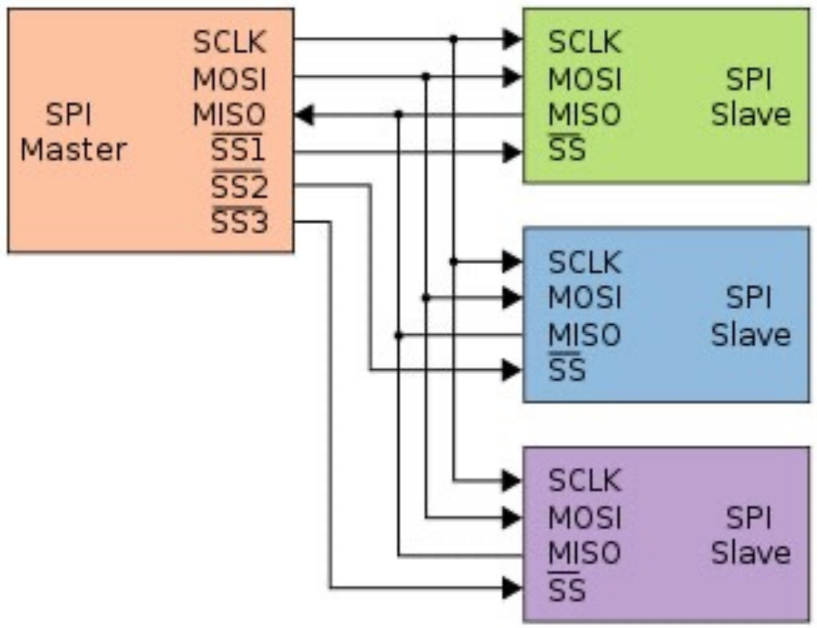


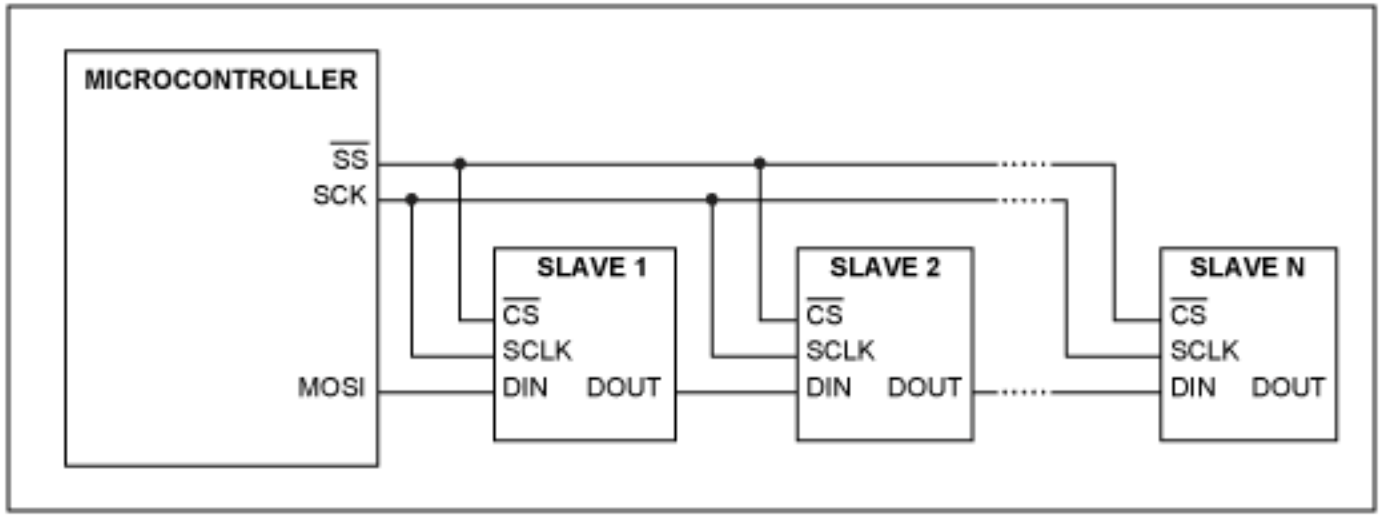
Timing diagram:



SPI (Serial Peripheral Interface Bus)

Two examples of bus configurations:





Master and multiple independent slaves

http://upload.wikimedia.org/wikipedia/commons/thumb/f/fc/SPI_three_slaves .svg/350px-SPI_three_slaves.svg.png

Master and multiple daisy-chained slaves

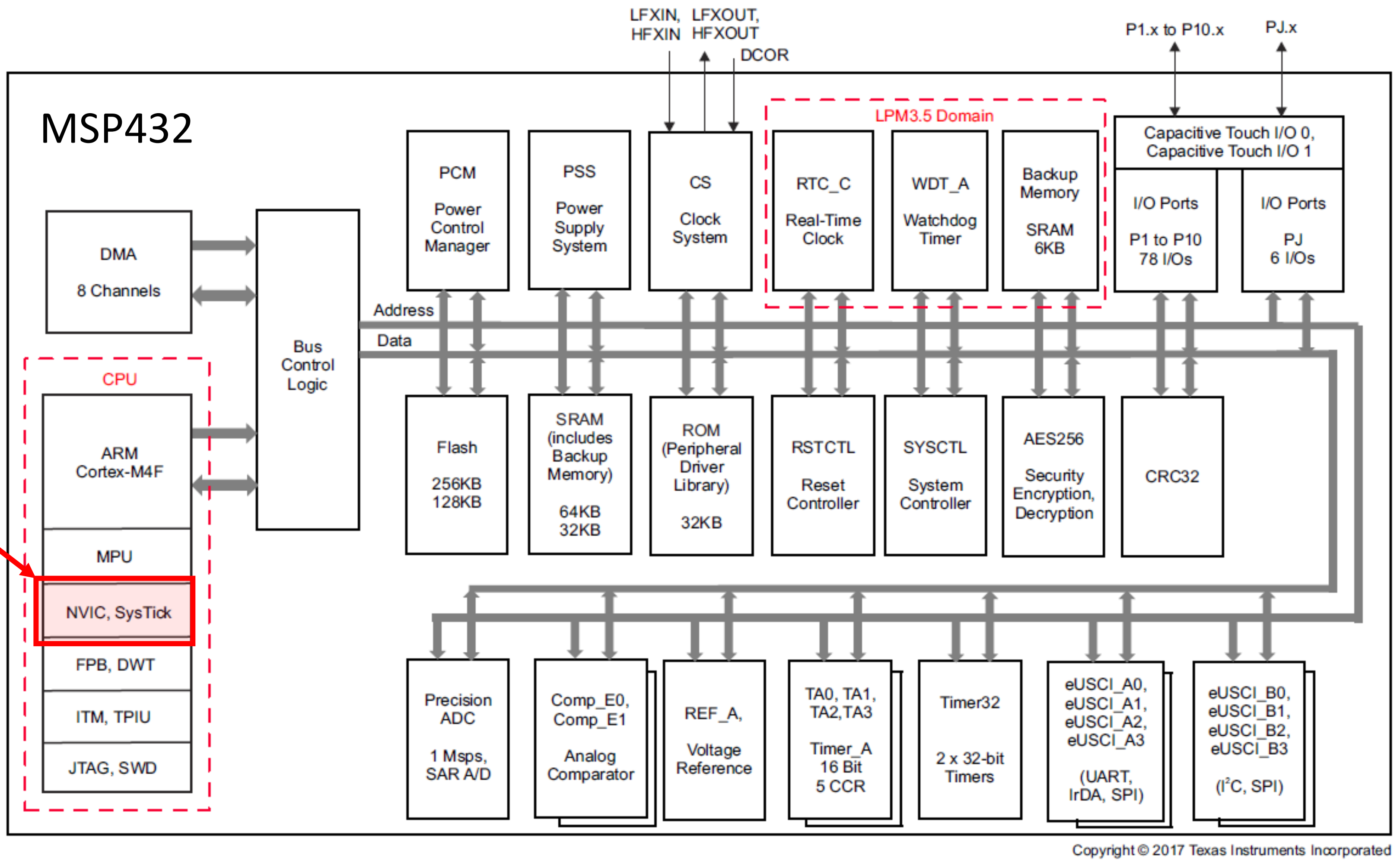
http://www.maxim-ic.com/appnotes.cfm/an_pk/3947

Interrupts

Interrupts

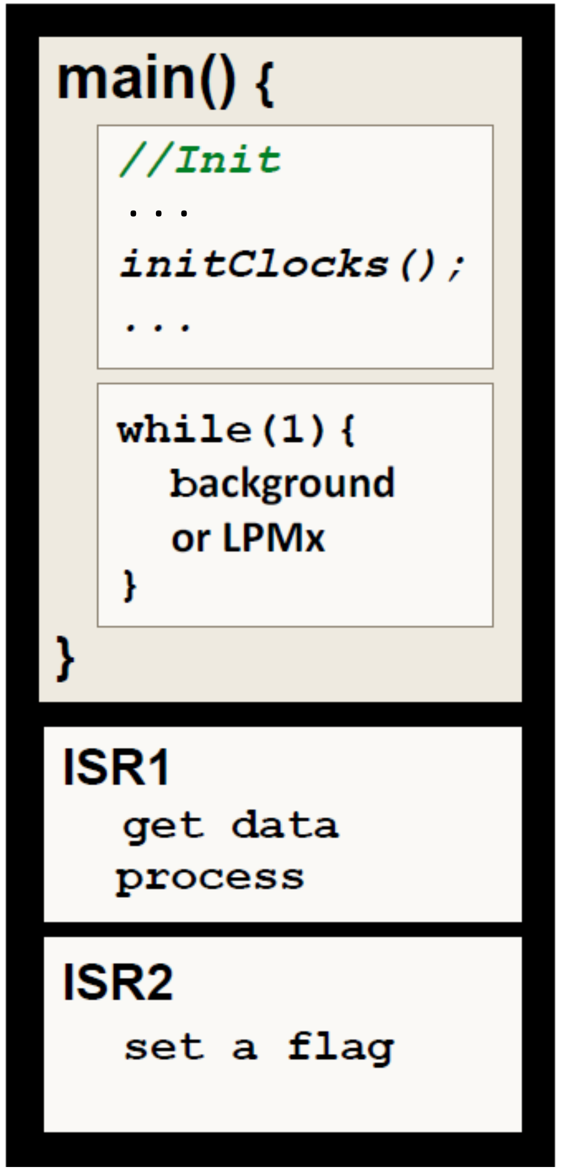
A hardware interrupt is an electronic alerting signal sent to the CPU from another component, either from an internal peripheral or from an external device.

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



LFXIN, HFXIN
LFXOUT, HFXOUT
DCOR
P1.x to P10.x
PJ.x
MSP432
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

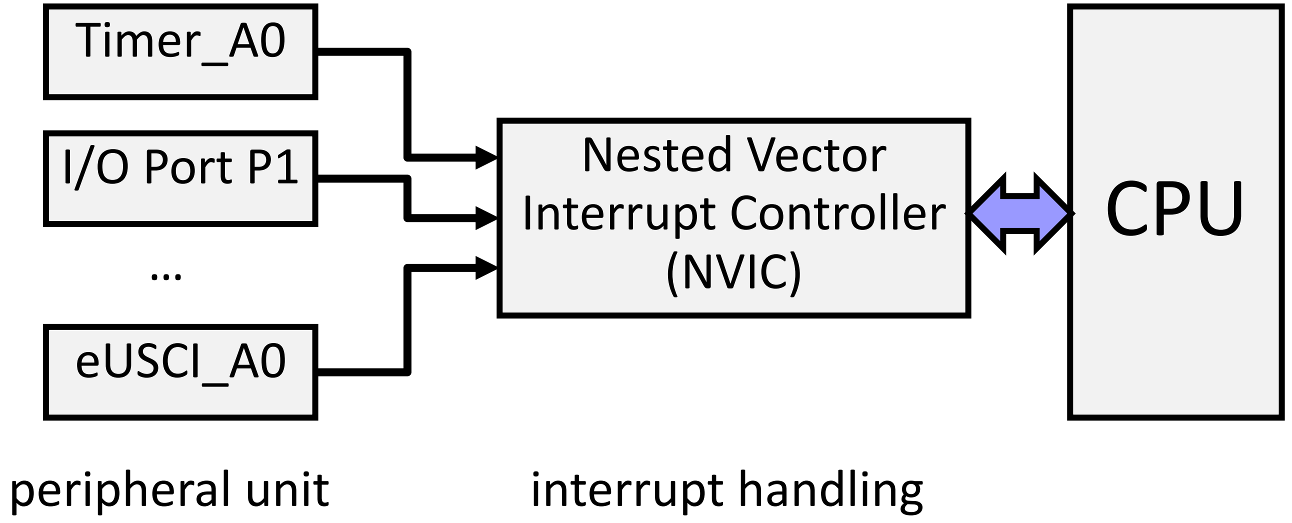
Background

- 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 (MSP432)



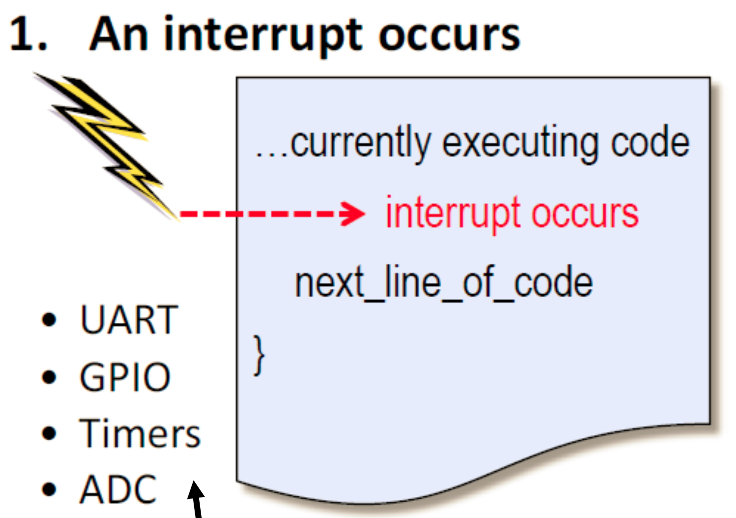
The *vector interrupt controller (NVIC)*

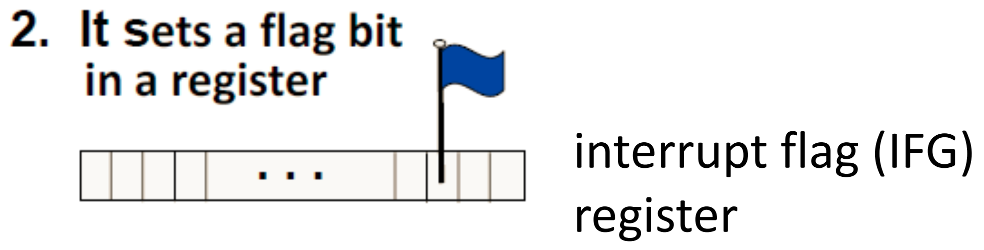
- enables and disables interrupts
- allows to individually and globally *mask interrupts* (disable reaction to interrupt), and
- registers *interrupt service routines* (ISR), sets the priority of interrupts.

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



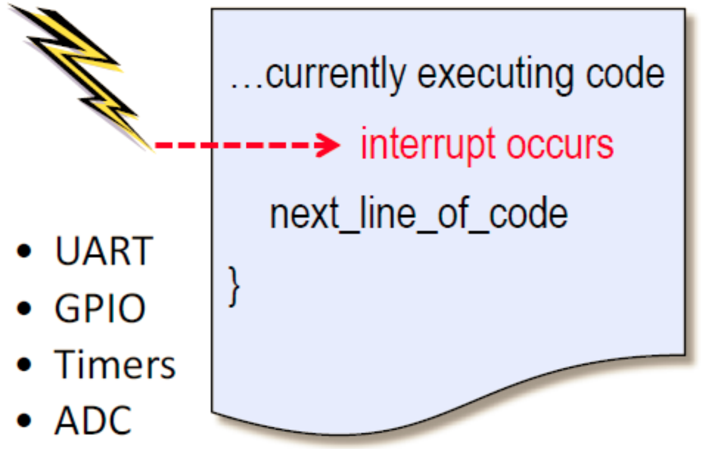

2. It sets a flag bit in a register
. . .
interrupt flag (IFG) register

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

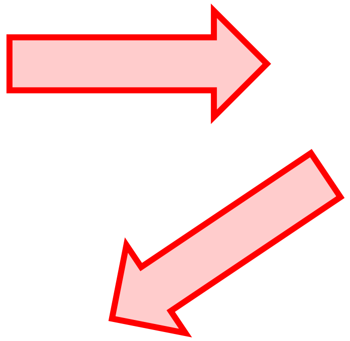
- When an interrupt signal is received, a corresponding bit is set in an IFG register.
- There is such an IFG register for each interrupt source.
- As some interrupt sources are only on for a short duration, the IFG register stores the fact that an interrupt has happened.

Processing of an Interrupt

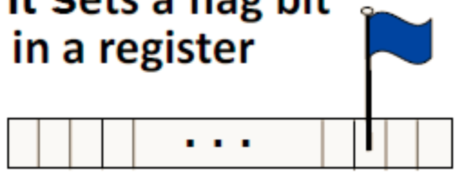
1. An interrupt occurs



- UART
- GPIO
- Timers
- ADC
- Etc.



2. It sets a flag bit in a register



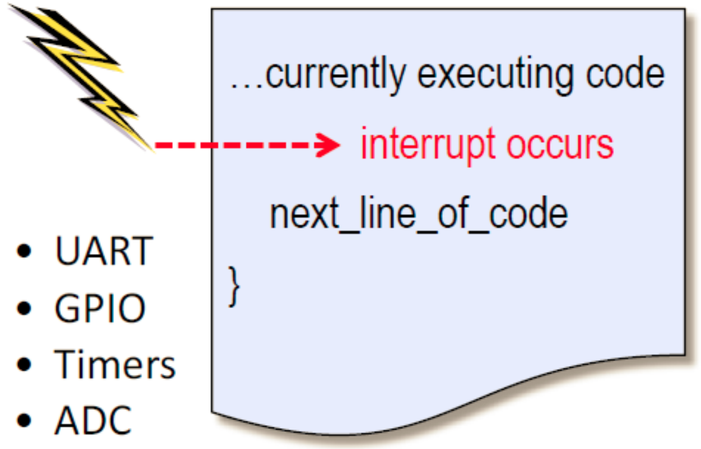
interrupt flag (IFG) register

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)

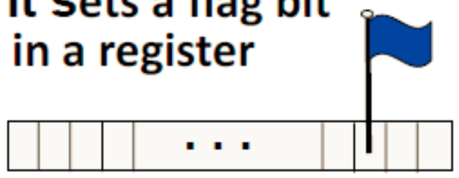
Processing of an Interrupt

1. **An interrupt occurs**



- UART
- GPIO
- Timers
- ADC
- Etc.

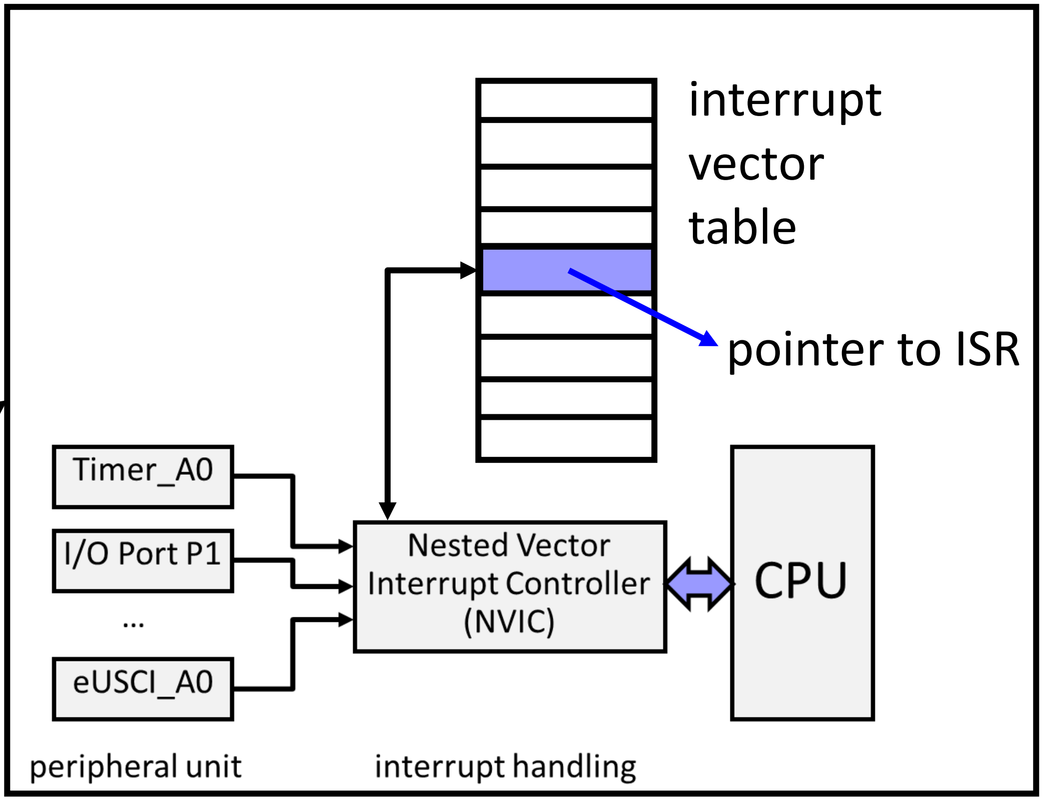
2. **It sets a flag bit in a register**



interrupt flag (IFG) register

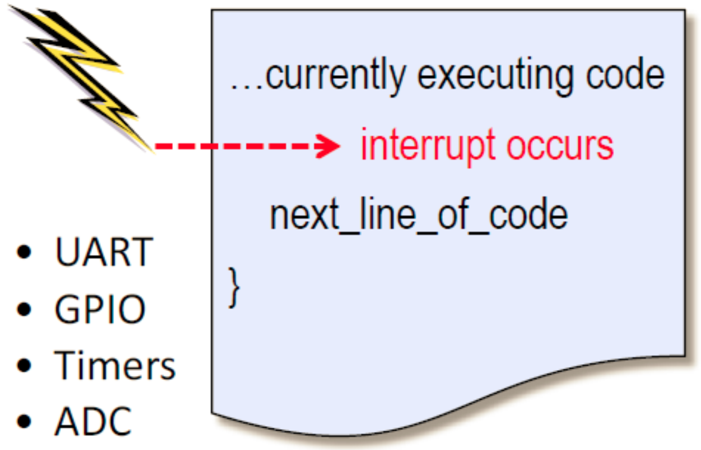
3. CPU/NVIC acknowledges interrupt by:

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



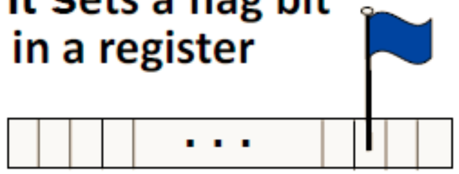
Processing of an Interrupt

1. **An interrupt occurs**



- UART
- GPIO
- Timers
- ADC
- Etc.

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



interrupt flag (IFG) register

3. CPU/NVIC acknowledges interrupt by:

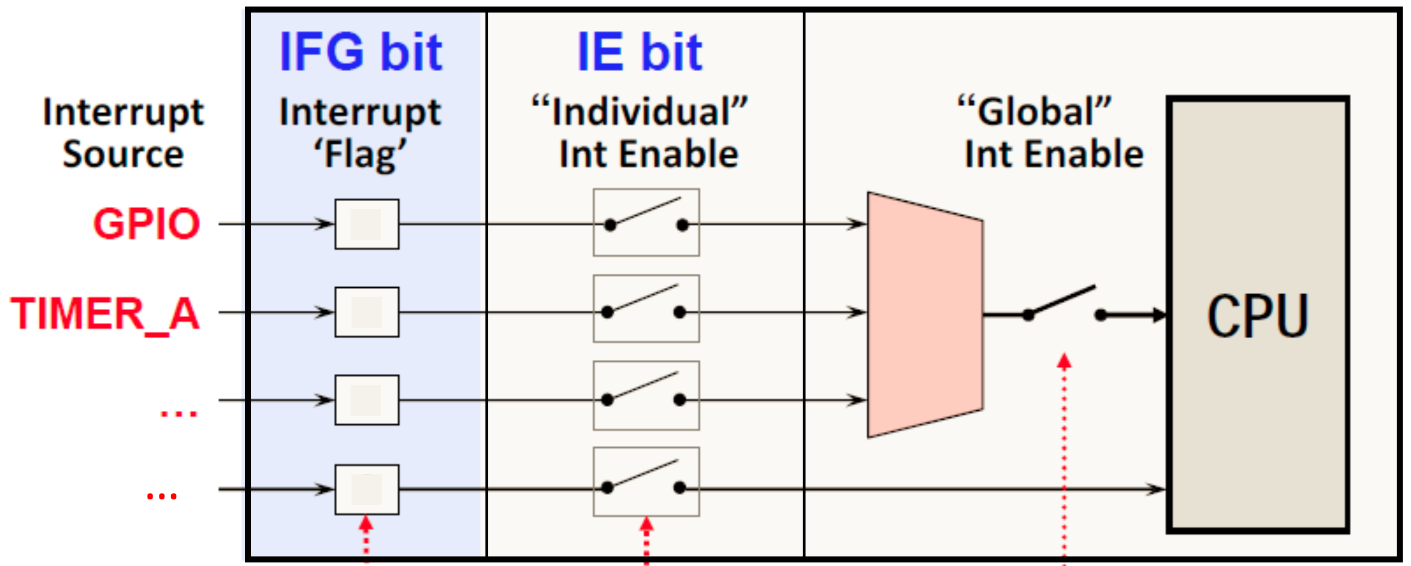
- current instruction completes
- saves return-to location on stack
- mask interrupts globally
- determines source of interrupt
- calls interrupt service routine (ISR)

4. Interrupt Service Routine (ISR):

- save context of system (on stack of the ISR)
- **run your interrupt's code**
- restore context of system (from stack of the ISR)
- (automatically) un-mask interrupts and
- continue where it left off (return-to location)

Processing of an Interrupt

Detailed interrupt processing flow:



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

get the interrupt status of the selected pin

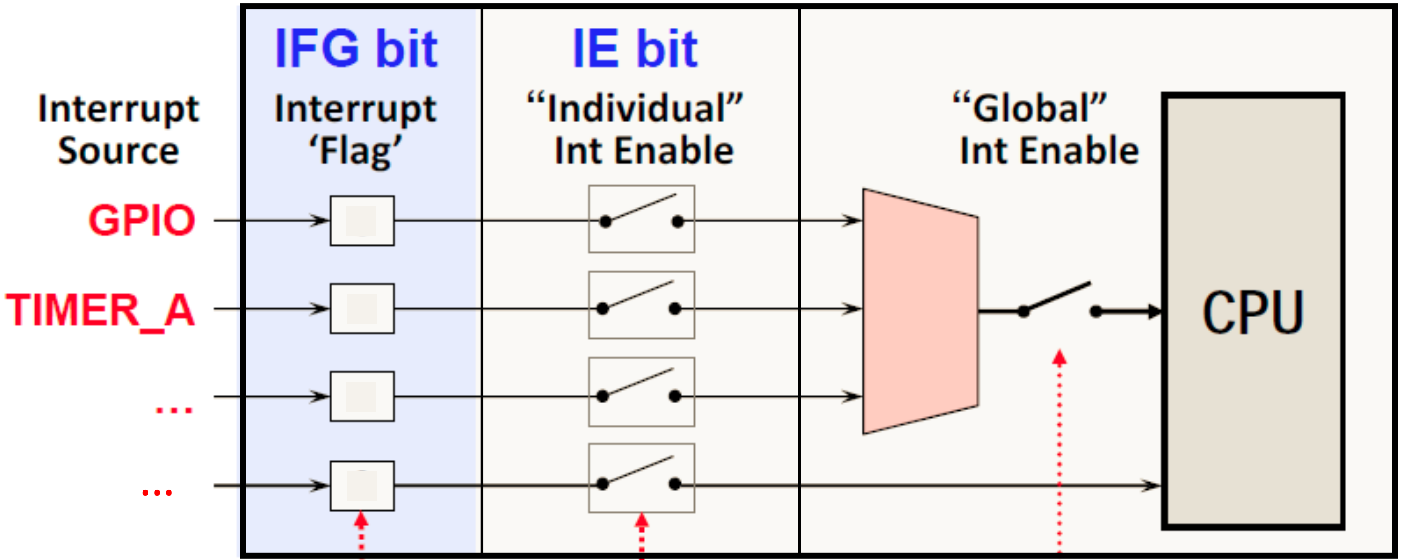
clears the interrupt status on the selected pin

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

Global Interrupt Enable
Enables ALL maskable interrupts
Interrupt_enableMaster();
Interrupt_disableMaster();

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();

Global Interrupt Enable
Enables ALL maskable interrupts

Interrupt_enableMaster();
Interrupt_disableMaster();

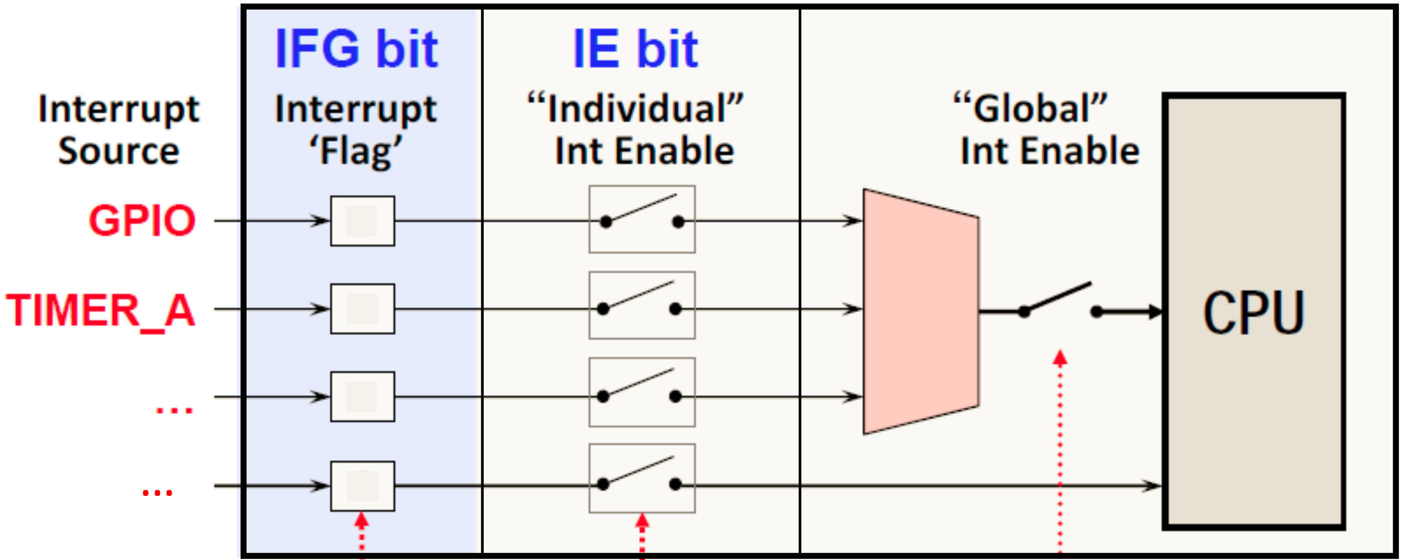
enable interrupt in the peripheral unit

enable interrupt in the interrupt controller

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

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();

enable interrupt in the peripheral unit

enable interrupt in the interrupt controller

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

Global Interrupt Enable
Enables ALL maskable interrupts
Interrupt_enableMaster();
Interrupt_disableMaster();

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

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*.

configure input and output pins

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:

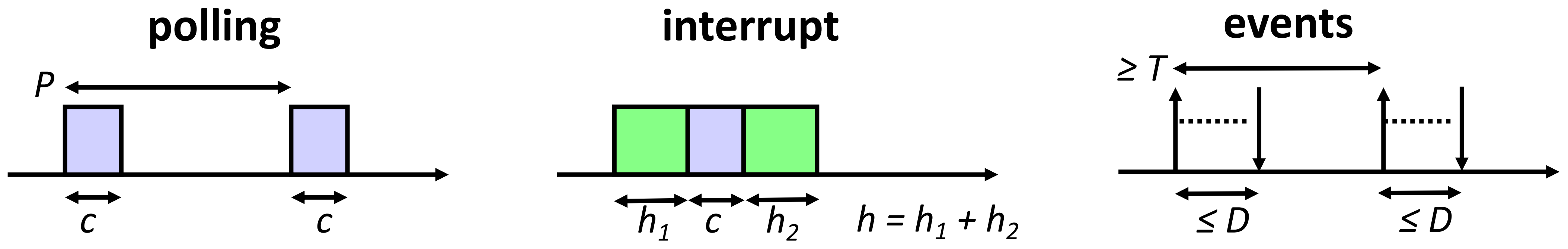
continuously get the signal at pin 1 and detect falling edge

```
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;
     }
}
```

Polling vs. Interrupt

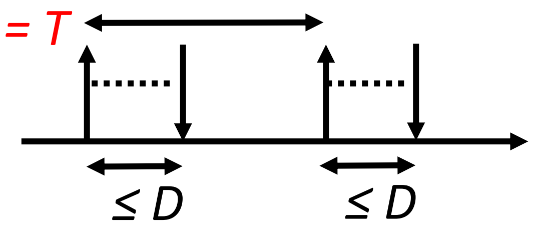
What are advantages and disadvantages?

- 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 for handling the event
  - *overhead h:* time overhead for handling the interrupt
  - *period P:* polling period
  - *inter-arrival time T*: minimal time between two events
  - *deadline D:* maximal time between event arrival and finishing event processing with $D \leq T$.

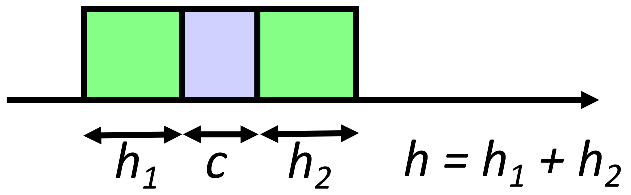


Polling vs. Interrupts

In the following, we suppose that the inter-arrival time between events is exactly T. This makes the results a bit easier to understand.

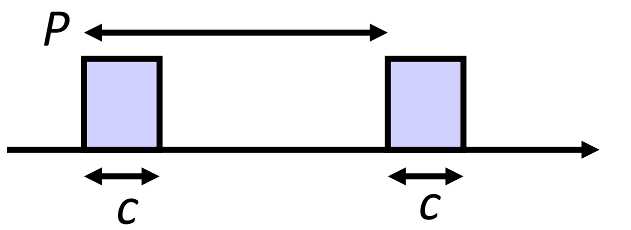


Some relations for *interrupt-based* event processing :



- The average utilization is $u_i = (h + c) / T$.
- As we need $h+c$ 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 most $P+c$ 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 $2c \le D < h+c$ then only polling is possible. The maximal polling period $P = D-c$ leads to the optimal (i.e., minimal) utilization $u_p = c / (D-c)$.

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

Case 4: If $c + max(c, h) \le D$ 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 inter-arrival time T, if the overhead h gets smaller in comparison to the computation time c, or if the inter-arrival time of events is only lower bounded by T (as in this case polling executes unnecessarily).

Clocks and Timers

Clocks and Timers

Clocks

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 oscillator

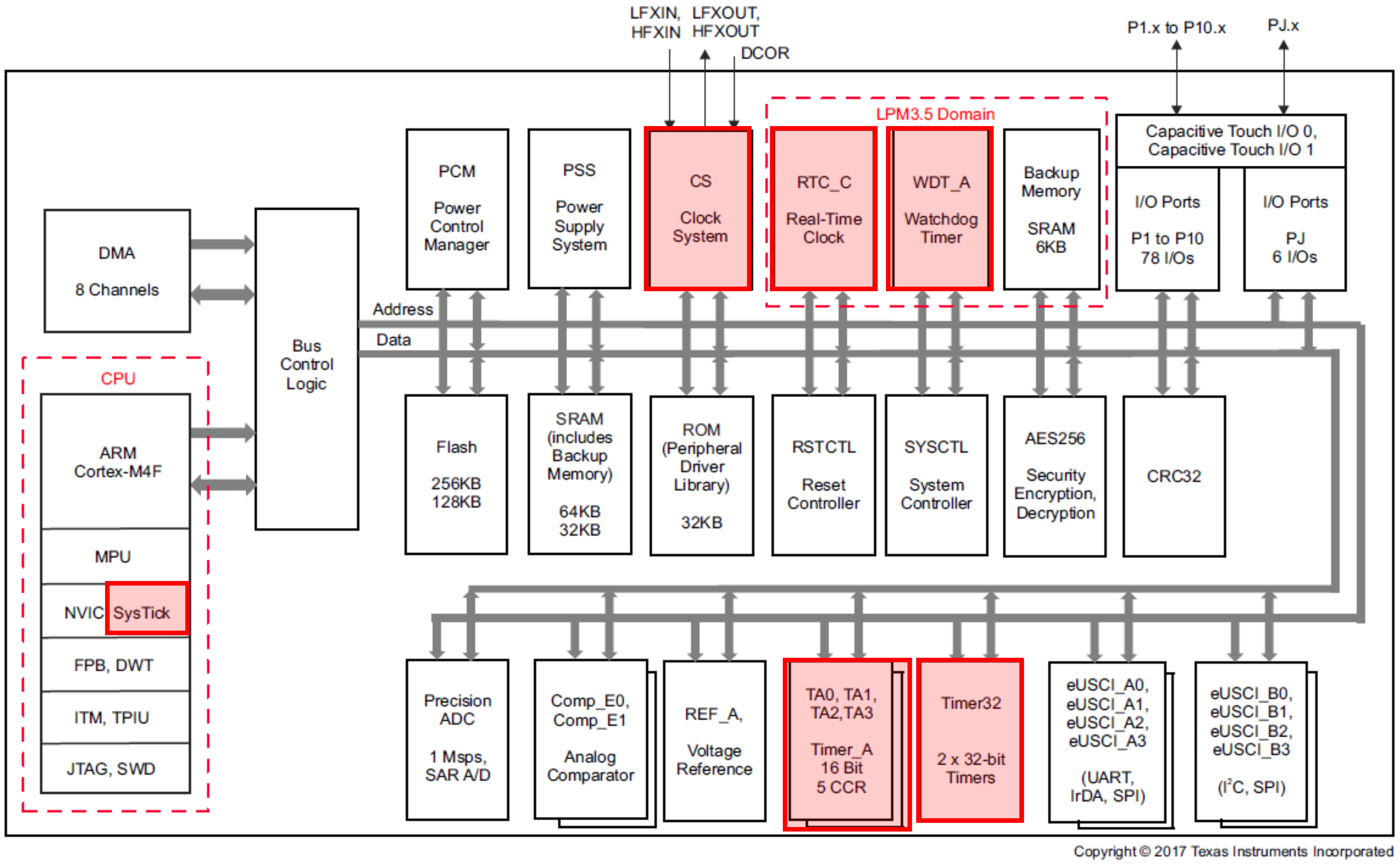
As an example, the MSP432 has the following *clock sources*:

	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

crystal-controlled (more stable)

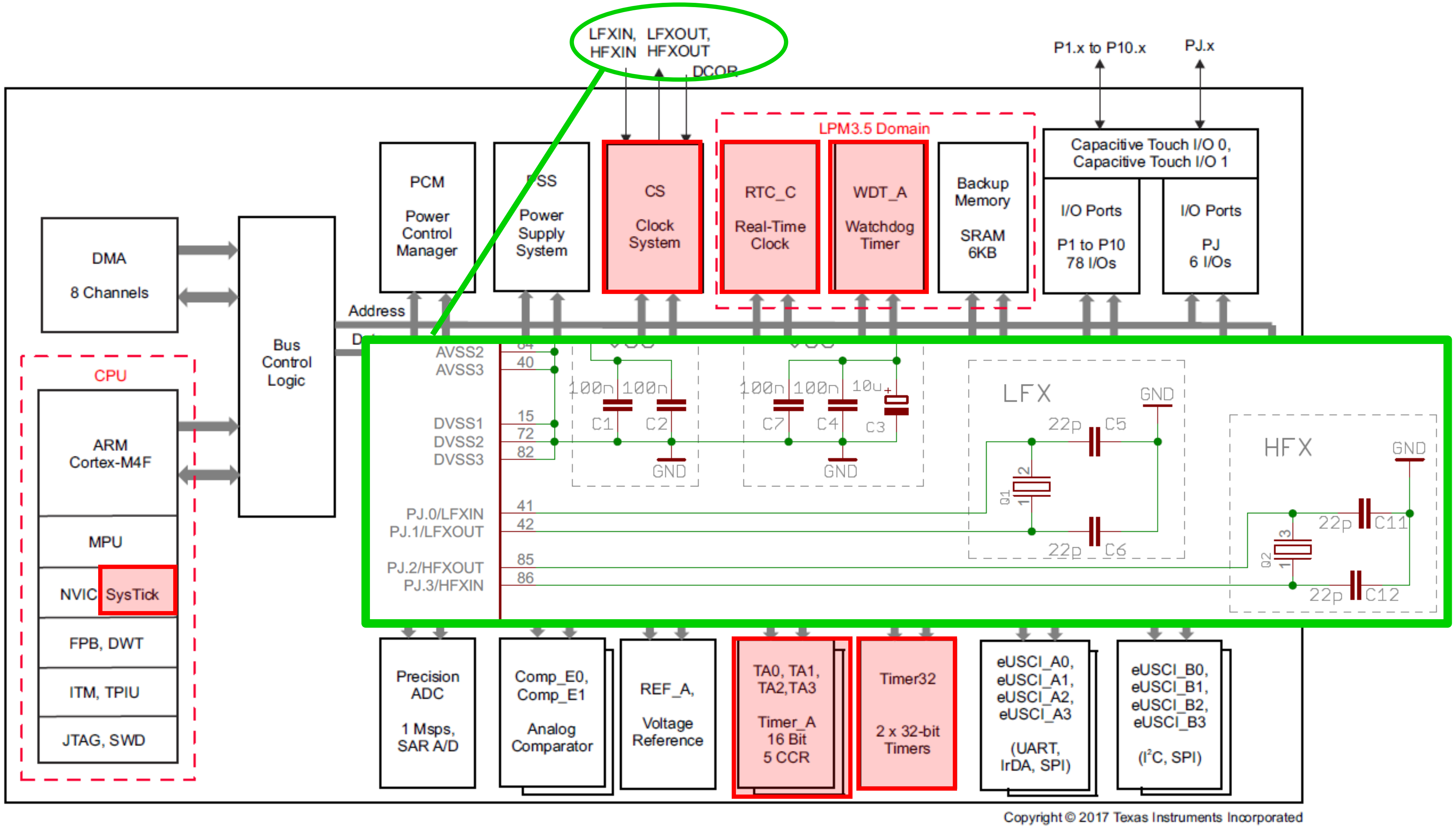
digitally-controlled (less stable)

Clocks and Timers MSP432



LFXIN, HFXIN
LFXOUT, HFXOUT
DCOR
P1.x to P10.x
PJ.x
LPM3.5 Domain
DMA 8 Channels
Bus Control Logic
Address
Data
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
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 (I^2C, SPI)

Clocks and Timers MSP432



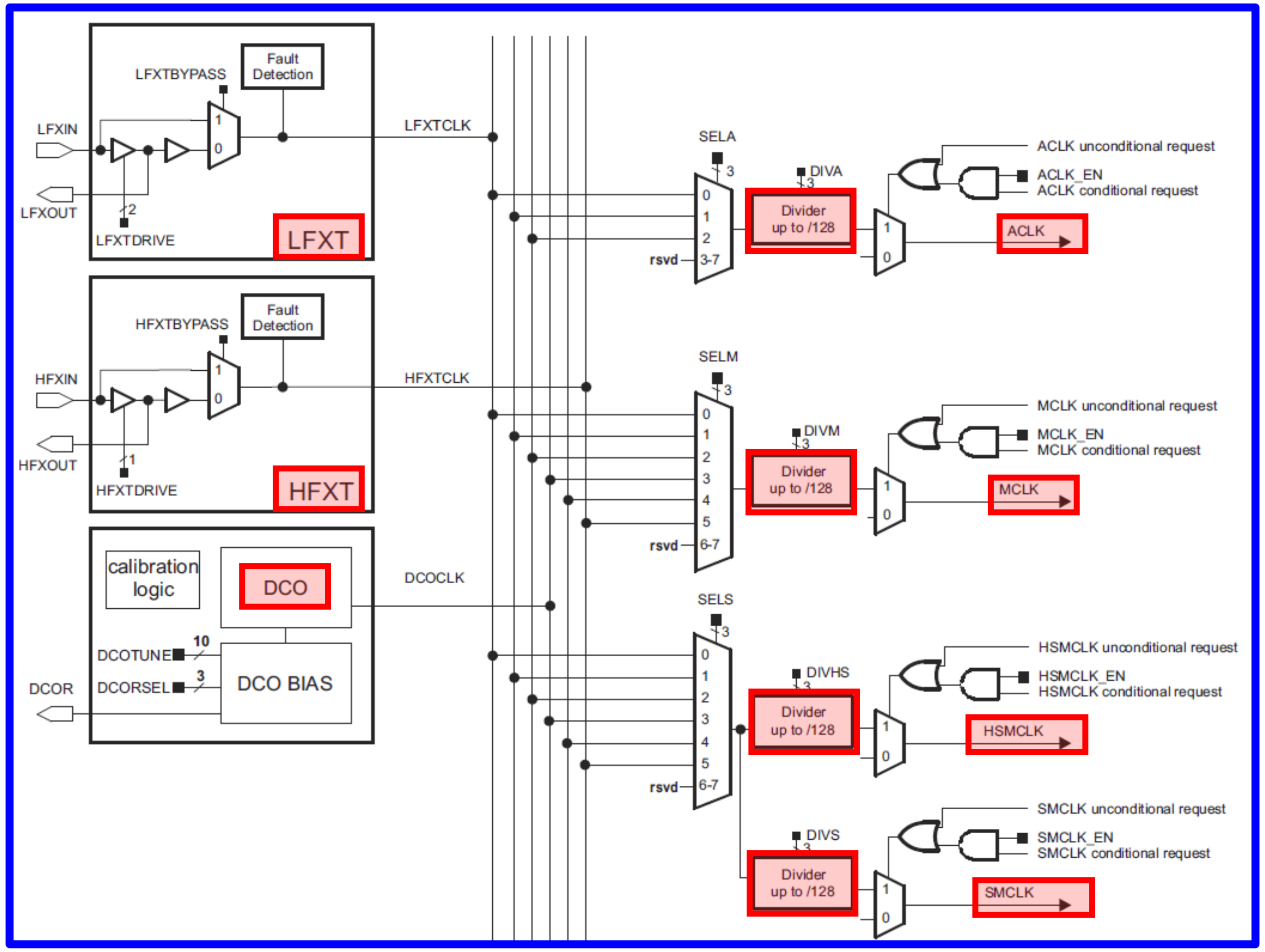
Clocks

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



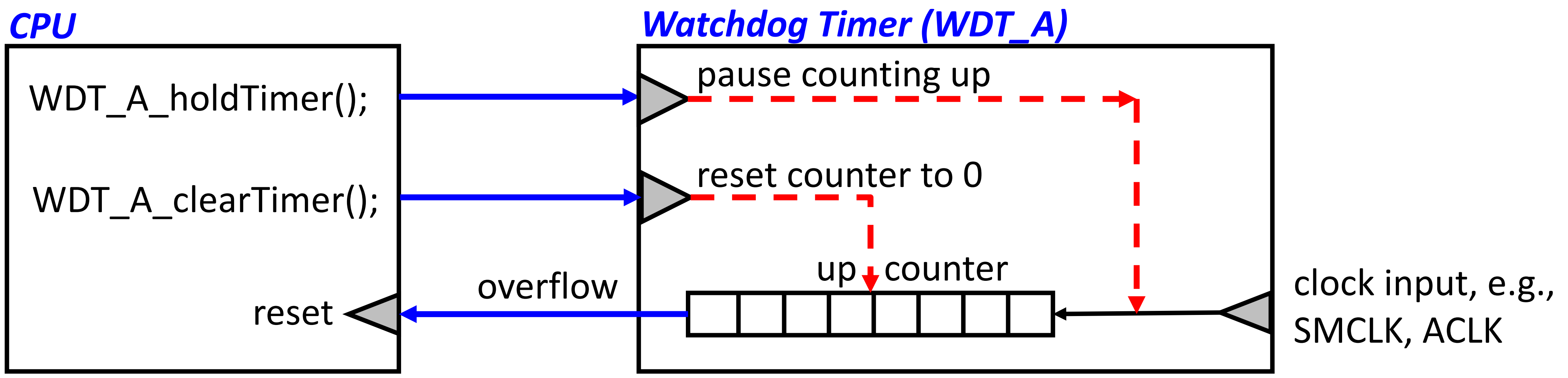
Clocks and Timers

Watchdog Timer

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 an overflow occurs because the counter was not reset to 0, then the CPU is reset.

Clocks and Timers

System Tick

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.

$\longleftarrow$ go to low power mode LPM0 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 ~5.6 seconds as $2^{24} / (3 \times 10^6) = 5.59...$

} the resolution of the duration is one microsecond; the duration must not be longer than ~5.6 seconds; note the use of modular arithmetic if end > start; overhead for calling SysTick_getValue() is not accounted for;

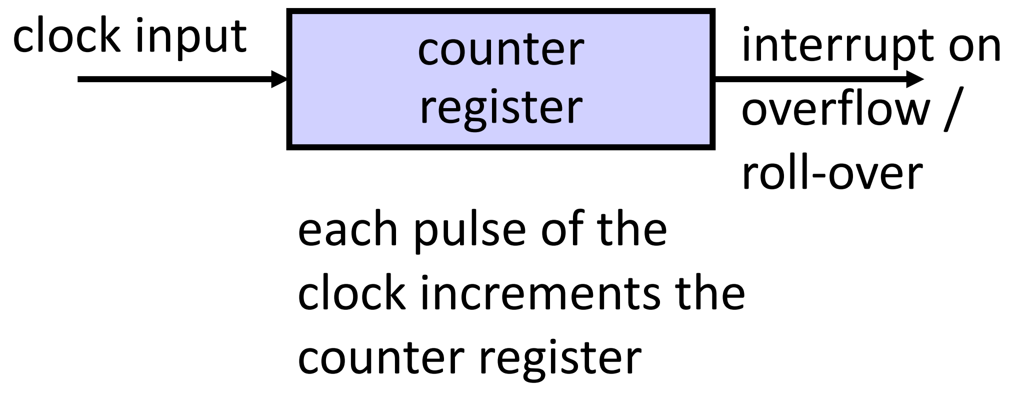
Clocks and Timers

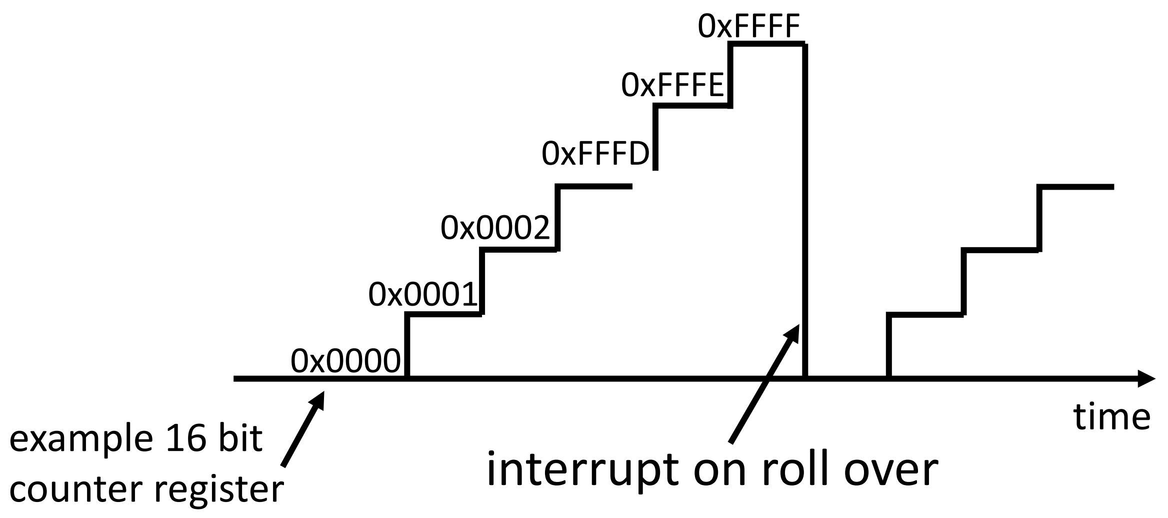
Timer and PWM

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*).

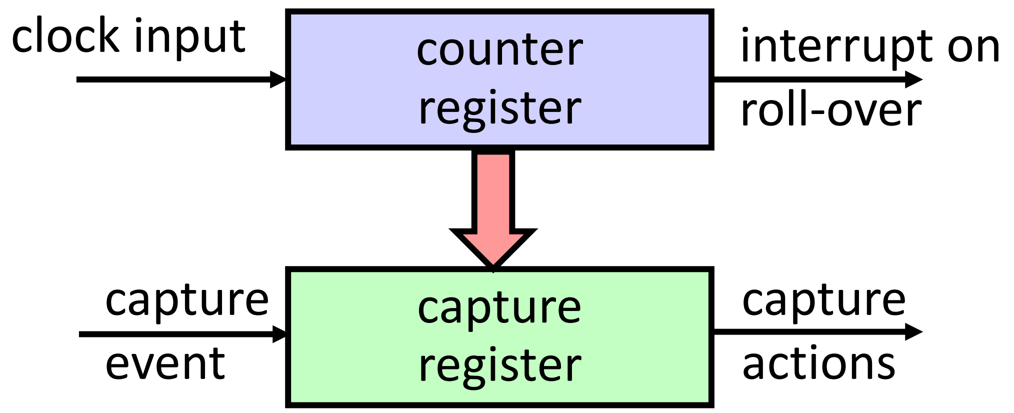

clock input
counter register
interrupt on overflow / roll-over
each pulse of the clock increments the counter register



Timer

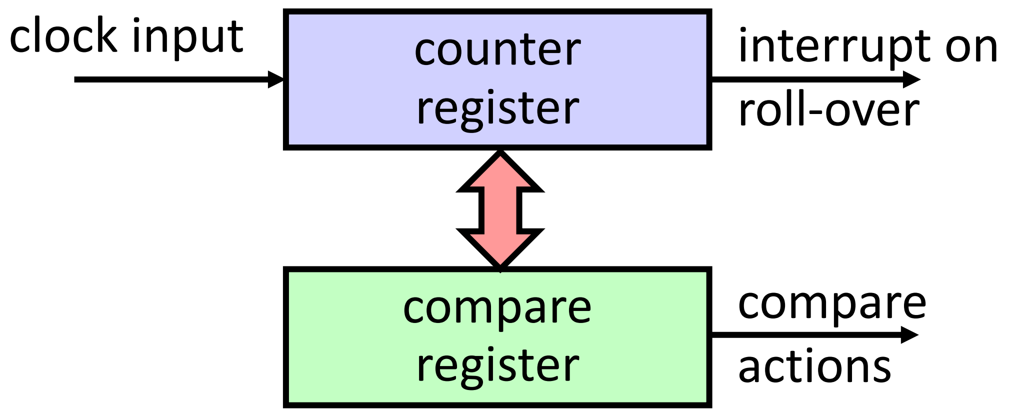
Typically, the mentioned functions are realized via *capture and compare registers*:

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)

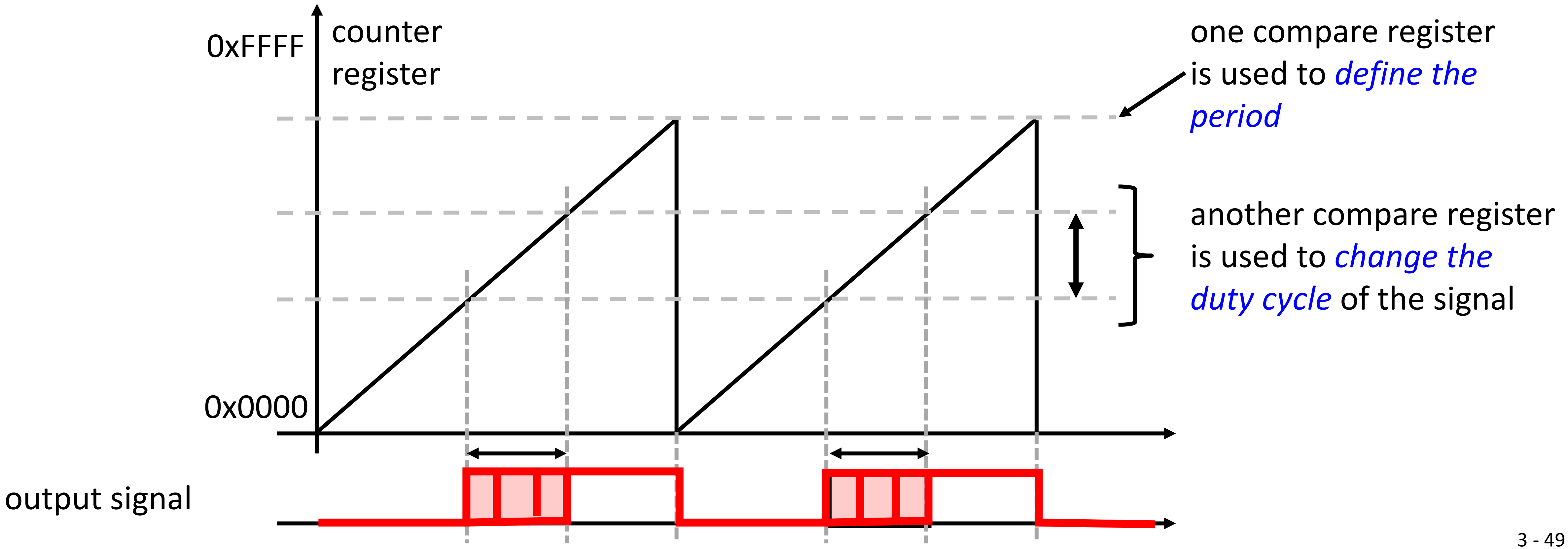
compare



- 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

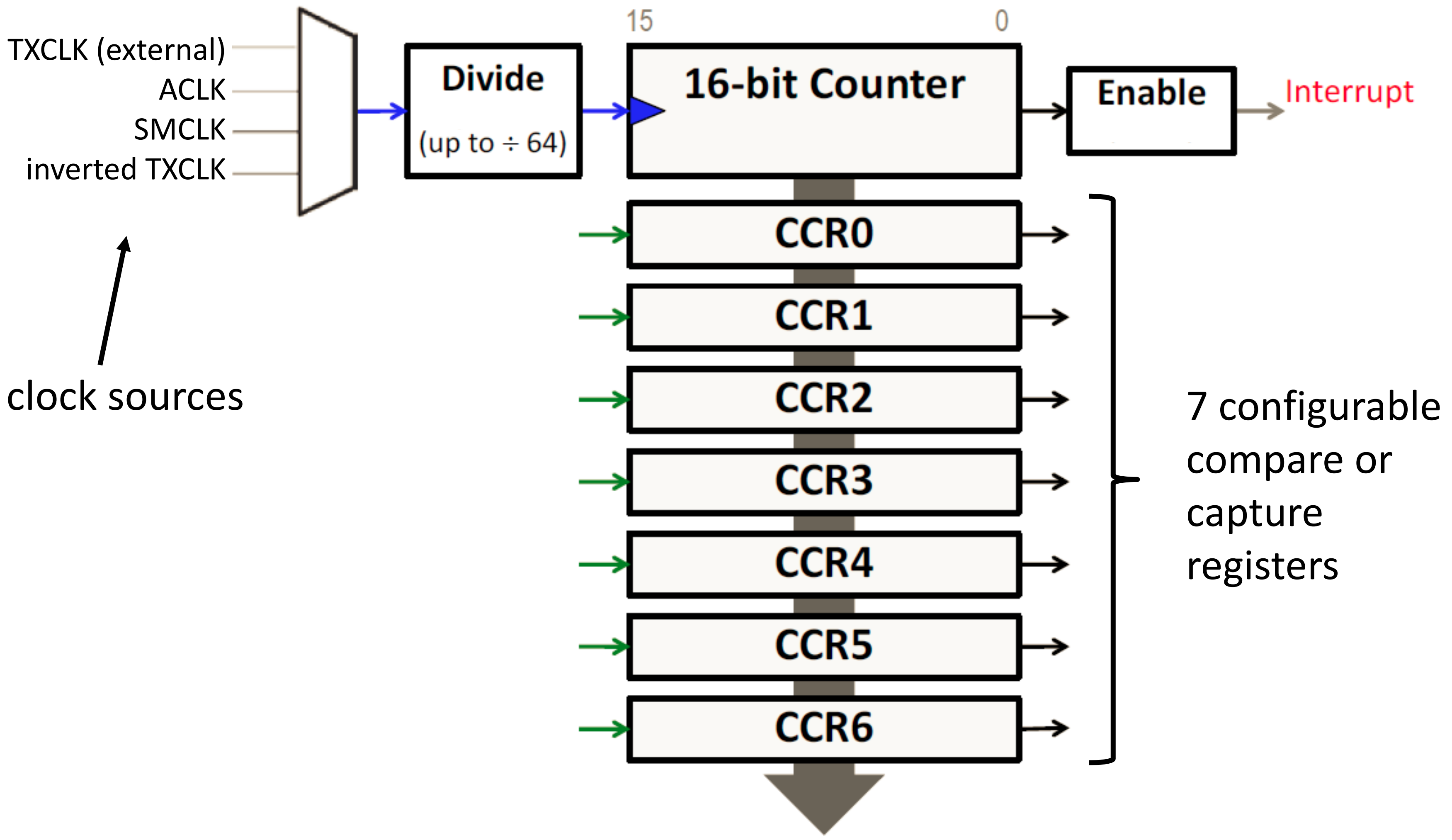
Timer

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

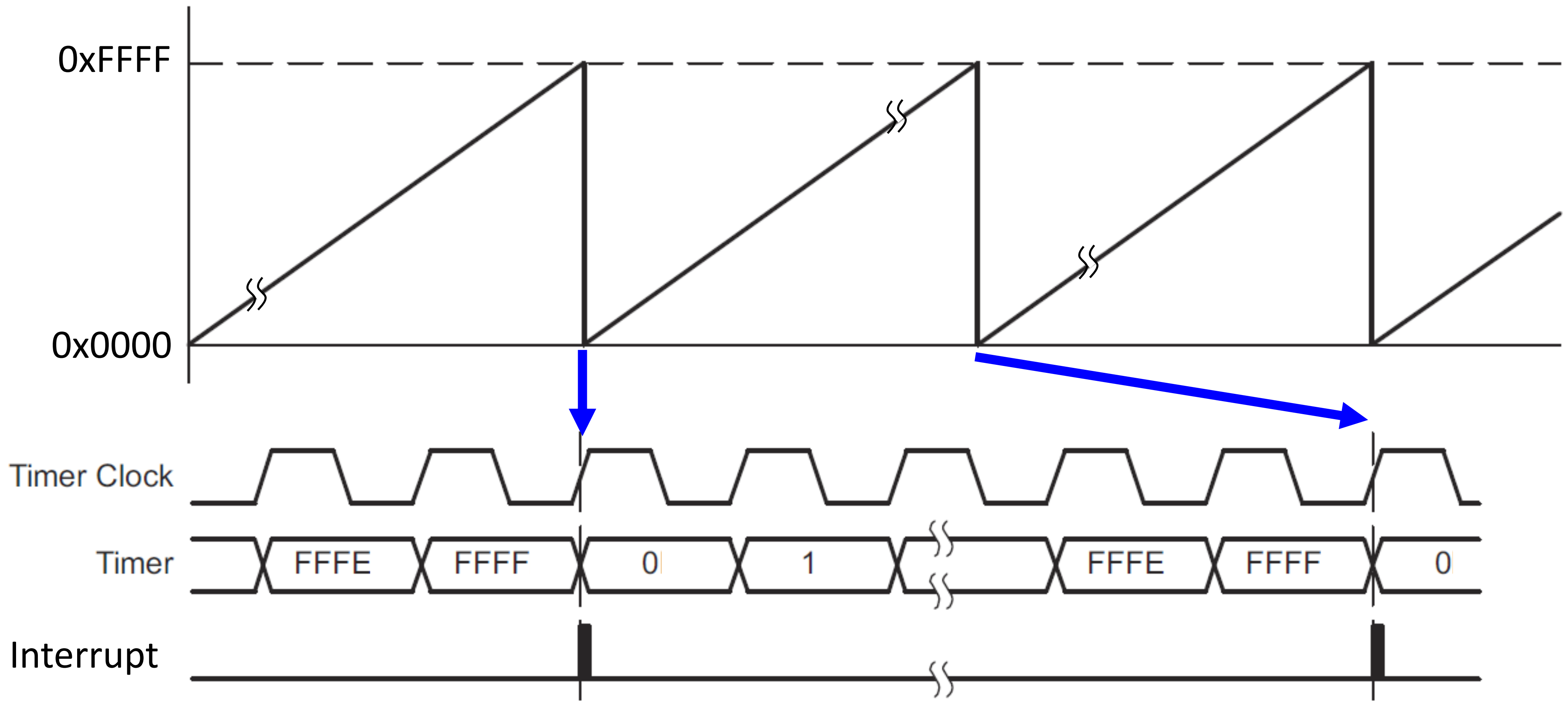
Timer Example MSP432

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, **but** with configurable periods.

```
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:

```
#define PERIOD 32768
```

- The following code should be added to main():

```
   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 CCR0 … 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;
   }
}
```

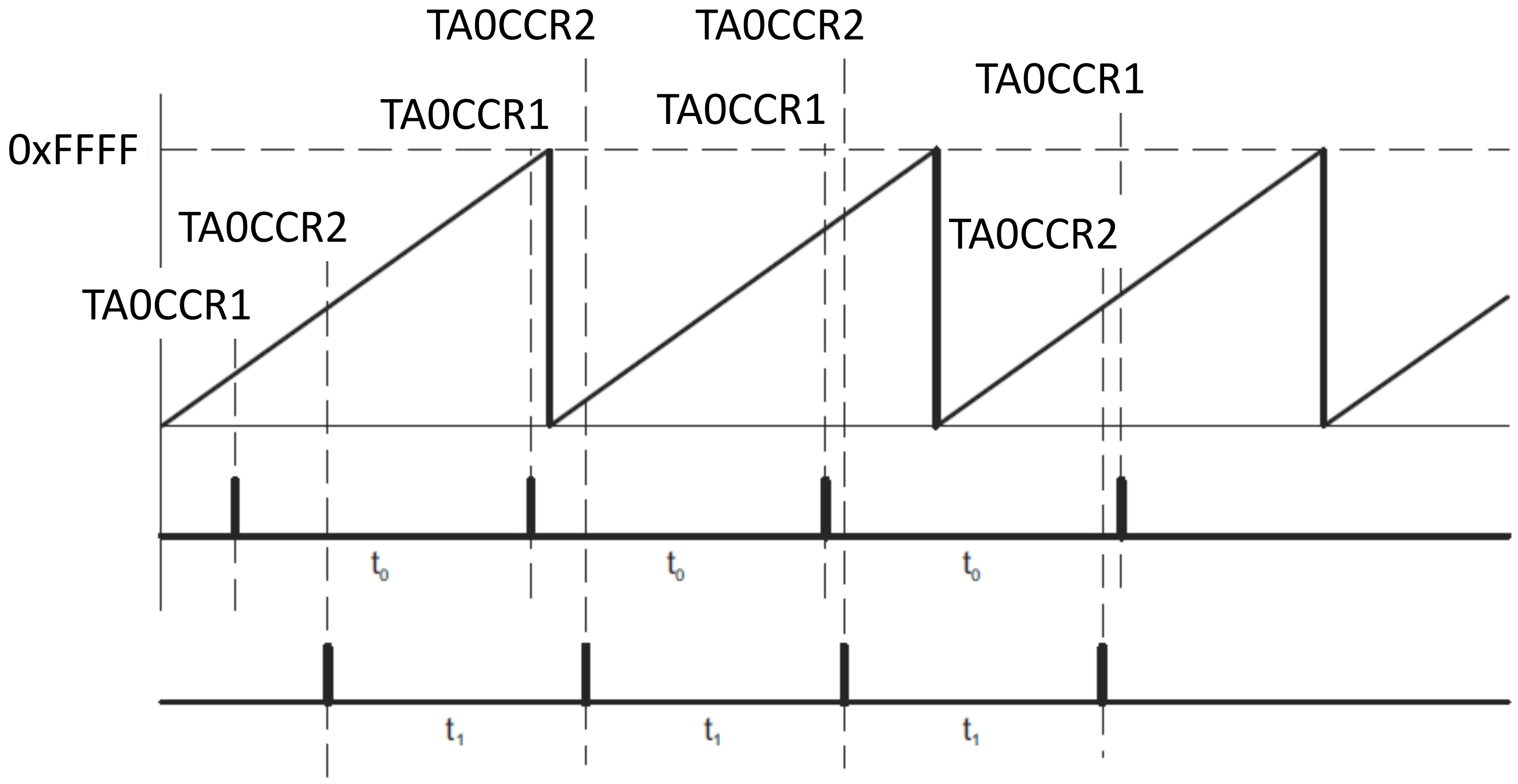
the register TA0IV contains the *interrupt flags* for the registers; after being read, the *highest priority interrupt* (smallest register number) is *cleared automatically*.

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

Timer Example MSP432

Example: This principle can be used to *generate several periodic interrupts* with *one timer*.


TA0CCR2
TA0CCR2
TA0CCR1
TA0CCR1
TA0CCR1
0xFFFF
TA0CCR2
TA0CCR2
TA0CCR1
t_0
t_0
t_0
t_1
t_1
t_1