Embedded Systems

3. Hardware Software Interface

© Jan Beutel / Lothar Thiele

Computer Engineering and Networks Laboratory
Do you Remember?
High-Level Physical View

- **nRF51822**
  - 16MHz Cortex-M0
  - 16kB RAM, 256kB Flash
  - BLE and NRF radio

- **Push button**

- **Power supplies and battery charger**

- **μUSB port**

- **RF power amplifier**

- **STM32F405**
  - 168MHz Cortex-M4
  - 196kB RAM, 1MB Flash

- **10DOF IMU**
  - 3-axis accelerometer
  - 3-axis gyro
  - 3-axis magnetometer
  - Pressure sensor

- **Expansion port**

- **Power switched by nRF51 (VCC)**

- **Motor driver**

- **SPI/I2C/GPIO/PWM**

- **Crazyflie 2.0 system architecture**
High-Level Physical View

- **Always ON power domain**
  - RF power amplifier
  - nRF51822
    - 16MHz Cortex-M0
    - 16kB RAM, 256KB Flash
    - BLE and NRF radio
  - Power supplies and battery charger
    - +5V
    - USB Data to STM32
  - µUSB port

- **Power switched by nRF51 (VCC)**
  - 10DOF IMU
    - 3-axis accelerometer
    - 3-axis gyro
    - 3-axis magnetometer
    - Pressure sensor
  - STM32F405
    - 168MHz Cortex-M4
    - 196kB RAM, 1MB Flash
  - Expansion port
  - Motor driver
  - EEPROM

Crazyflie 2.0 system architecture
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
Remember ... ?

Crazyflie 2.0 system architecture

Always ON power domain

- RF power amplifier

10DOF IMU
- 3-axis accelerometer
- 3-axis gyro
- 3-axis magnetometer
- Pressure sensor

Power switched by nRF51 (VCC)

- STM32F405
  - 168MHz Cortex-M4
  - 196kB RAM, 1MB Flash

- nRF51822
  - 16MHz Cortex-M0
  - 16kB RAM, 256KB Flash
  - BLE and NRF Radio

- Power supplies and battery charger

- Expansion port

- Motor driver

- EEPROM

- Push button

- +5V

- μUSB port

USB Data to STM32

Charge/VBAT/VCC

- UART

- I2C

- SPI/I2C/GPIO/PWM

- PWM
MSP432P401R (ES-Lab)
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 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/CAS = row/column address select)
DRAM – Typical Access Process

1. Bus Transmission

2. Precharge and Row Access
DRAM – Typical Access Process

3. Column Access

4. Data Transfer and Bus Transmission
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”

Programming (=writing) to logical “0”

Reading

“Quantum tunneling” Drains charge from FG

“Hot-electron injection” Traps charge in FG

Turn on low Vt or High Vt?

Detect $I_{on}$ to read 0 or 1

Programming via hot electron injection

12 V

0 V

floating gate

200 Å

SOURCE

DRAIN

drain-source resistance

$V_{th}$
erased

$V_{th}$
programmed

$V_{read}$
NAND and NOR Flash Memory

<table>
<thead>
<tr>
<th>Cell Array &amp; Size</th>
<th>NAND</th>
<th>NOR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1" alt="NAND Diagram" /></td>
<td><img src="image2" alt="NOR Diagram" /></td>
</tr>
<tr>
<td>Cross-section</td>
<td><img src="image3" alt="NAND Cross-section" /></td>
<td><img src="image4" alt="NOR Cross-section" /></td>
</tr>
<tr>
<td>Features</td>
<td>Small Cell Size, High Density, Low Power</td>
<td>Fast random access</td>
</tr>
<tr>
<td></td>
<td>Mass Storage</td>
<td>Code Storage</td>
</tr>
</tbody>
</table>
Example: Reading out NAND Flash

**Selected word-line (WL):** Target voltage ($V_{\text{target}}$)

**Unselected word-lines:** $V_{\text{read}}$ is high enough to have a low resistance in all transistors in this row

![Diagram]

- SSL
- Unselected WLs
- Selected WL
- Unselected WLs
- GSL

- Drain-source resistance
- Gate voltage

- $V_{\text{th, erased}}$
- $V_{\text{target}}$
- $V_{\text{th, programmed}}$
- $V_{\text{read}}$
Storage

Memory Map
Available memory:

- The processor used in the lab (MSP432P401R) has built in 256kB flash memory, 64kB SRAM and 32kB ROM (Read Only Memory).

Address space:

- The processor uses 32 bit addresses. Therefore, the addressable memory space is 4 GByte ($= 2^{32}$ Byte) as each memory location corresponds to 1 Byte.
- The address space is used to address the memories (reading and writing), to address the peripheral units, and to have access to debug and trace information (memory mapped microarchitecture).
- The address space is partitioned into zones, each one with a dedicated use. The following is a simplified description to introduce the basic concepts.
Example: Memory Map in MSP432 (ES-Lab)

**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
Example: Memory Map in MSP432 (ES-Lab)

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
Example: Memory Map in MSP432 (ES-Lab)

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

Table 6-21. Port Registers (Base Address: 0x4000_4C00)

<table>
<thead>
<tr>
<th>REGISTER NAME</th>
<th>ACRONYM</th>
<th>OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port 1 Input</td>
<td>P1IN</td>
<td>000h</td>
</tr>
<tr>
<td>Port 2 Input</td>
<td>P2IN</td>
<td>001h</td>
</tr>
<tr>
<td>Port 1 Output</td>
<td>P1OUT</td>
<td>002h</td>
</tr>
<tr>
<td>Port 2 Output</td>
<td>P2OUT</td>
<td>003h</td>
</tr>
</tbody>
</table>

Schematic of LaunchPad as used in the Lab:

- LED1 is connected to Port 1, Pin 0

How do we toggle LED1 in a C program?
Example: Memory Map in MSP432 (ES-Lab)

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). See lab session.

```c
//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
Example: Memory Map in MSP432 (ES-Lab)

Memory map:

- Hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit.
- 0x3FFFF address difference = $4 \times 2^{16}$ different addresses → 256 kByte maximal data capacity for Flash Main Memory.
- 0x0FF address difference = 2^9 different addresses → 229 Byte = 512 MByte.
- Used for program, data and non-volatile configuration.
Example: Memory Map in MSP432 (ES-Lab)

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

- 0x FFFF address difference = $2^{16}$ different addresses → 64 kByte maximal data capacity for SRAM Region
- Used for program and data.
Input and Output
Device Communication

Very often, a processor needs to *exchange information with other processors* or devices. To satisfy various needs, there exists many different *communication protocols*, such as

- **UART** (Universal Asynchronous Receiver-Transmitter)
- **SPI** (Serial Peripheral Interface Bus)
- **I2C** (Inter-Integrated Circuit)
- **USB** (Universal Serial Bus)

As the principles are similar, we will just explain a representative of an asynchronous protocol (**UART**, no shared clock signal between sender and receiver) and one of a synchronous protocol (**SPI**, shared clock signal).
Input and Output

UART Protocol
Remember?

**low power CPU**
- enabling power to the rest of the system
- battery charging and voltage measurement
- wireless radio (boot and operate)
- detect and check expansion boards

**higher performance CPU**
- sensor reading and motor control
- flight control
- telemetry (including the battery voltage)
- additional user development
- USB connection

---

**UART:**
- communication protocol (Universal Asynchronous Receiver/Transmitter)
- exchange of data packets to and from interfaces (wireless, USB)
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:

  - There exist many variations of this simple scheme.
UART

- The receiver runs an *internal clock* whose frequency is an exact multiple of the expected bit rate.
- When a *Start bit* is detected, a counter begins to count clock cycles e.g. 8 cycles until the midpoint of the anticipated Start bit is reached.
- The clock counter counts a further 16 cycles, to the middle of the first *Data bit*, and so on until the *Stop bit*. 
UART with MSP432 (ES-Lab)
UART with MSP432 (Lab)
Input and Output

Memory Mapped Device Access
Memory-Mapped Device Access

**eUSCI_A0 Registers (Base Address: 0x4000_1000)**

<table>
<thead>
<tr>
<th>REGISTER NAME</th>
<th>OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>eUSCI_A0 Control Word 0</td>
<td>00h</td>
</tr>
<tr>
<td>eUSCI_A0 Control Word 1</td>
<td>02h</td>
</tr>
<tr>
<td>eUSCI_A0 Baud Rate Control</td>
<td>06h</td>
</tr>
<tr>
<td>eUSCI_A0 Modulation Control</td>
<td>08h</td>
</tr>
<tr>
<td>eUSCI_A0 Status</td>
<td>0Ah</td>
</tr>
<tr>
<td>eUSCI_A0 Receive Buffer</td>
<td>0Ch</td>
</tr>
<tr>
<td>eUSCI_A0 Transmit Buffer</td>
<td>0Eh</td>
</tr>
<tr>
<td>eUSCI_A0 Auto Baud Rate Control</td>
<td>10h</td>
</tr>
<tr>
<td>eUSCI_A0 IrDA Control</td>
<td>12h</td>
</tr>
<tr>
<td>eUSCI_A0 Interrupt Enable</td>
<td>1Ah</td>
</tr>
<tr>
<td>eUSCI_A0 Interrupt Flag</td>
<td>1Ch</td>
</tr>
<tr>
<td>eUSCI_A0 Interrupt Vector</td>
<td>1Eh</td>
</tr>
</tbody>
</table>

- **Configuration of Transmitter and Receiver must match**; otherwise, they can not communicate.
- Examples of configuration parameters:
  - transmission rate (baud rate, i.e., symbols/s)
  - LSB or MSB first
  - number of bits per packet
  - parity bit
  - number of stop bits
  - interrupt-based communication
  - clock source

buffer for received bits and bits that should be transmitted

in our case: bit/s
Transmission Rate

**Clock subsampling:**
- The clock subsampling block is complex, as one tries to match a large set of transmission rates with a fixed input frequency.

**Clock Source:**
- SMCLK in the lab setup = 3MHz
- Quartz frequency = 48 MHz, is divided by 16 before connected to SMCLK

**Example:**
- Transmission rate 4800 bit/s
- 16 clock periods per bit (see 3-26)
- Subsampling factor = $\frac{3 \times 10^6}{(4.8 \times 10^3 \times 16)} = 39.0625$
Software Interface

Part of C program that prints a character to a UART terminal on the host PC:

```c
... static const eUSCI_UART_Config uartConfig = {     EUSCI_A_UART_CLOCKSOURCE_SMCLK, // SMCLK Clock Source     39, // BRDIV = 39, integral part     1, // UCxBRF = 1, fractional part * 16     0, // UCxBRS = 0     EUSCI_A_UART_NO_PARITY, // No Parity     EUSCI_A_UART_LSB_FIRST, // LSB First     EUSCI_A_UART_ONE_STOP_BIT, // One stop bit     EUSCI_A_UART_MODE, // UART mode     EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION}; // Oversampling Mode GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P1, GPIO_PIN2 | GPIO_PIN3, GPIO_PRIMARY_MODULE_FUNCTION); //Configure CPU signals UART_initModule(EUSCI_A0_BASE, &uartConfig); // Configuring UART Module A0 UART_enableModule(EUSCI_A0_BASE); // Enable UART module A0 UART_transmitData(EUSCI_A0_BASE,'a'); // Write character ‘a’ to UART ...
```

Data structure `uartConfig` contains the configuration of the UART.

Use `uartConfig` to write to `eUSCI_A0` configuration registers.

Start UART.

Base address of A0 (0x40001000), where A0 is the instance of the UART peripheral.
Replacing `UART_transmitData(EUSCI_A0_BASE,'a')` by a *direct access to registers*:

```c
volatile uint16_t* uca0ifg = (uint16_t*) 0x4000101C;
volatile uint16_t* uca0txbuf = (uint16_t*) 0x4000100E;
...
// Initialization of UART as before
...
while (!(*uca0ifg >> 1) & 0x0001));
*uca0txbuf = (char) 'g'; // Write to transmit buffer
...
```

Table 22.18. UCAxIFG Register Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-4</td>
<td>Reserved</td>
<td>R</td>
<td>0h</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>UCTXIFG</td>
<td>RW</td>
<td>1h</td>
<td>Transmit interrupt flag. UCTXIFG is set when UCAxTXBUF empty. 0h = No interrupt pending 1b = Interrupt pending</td>
</tr>
</tbody>
</table>

- Wait until transmit buffer is empty
- Write character ‘g’ to the transmit buffer
- Shift 1 bit to the right
- Expression is ‘1’ if bit UCTXIFG = 0 (buffer not empty).
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
- 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:
- system clock SCLK
- writing data output: MOSI or MISO
- reading data input in the middle of bit:
SPI (Serial Peripheral Interface Bus)

Two examples of bus configurations:

Master and multiple independent slaves

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.
Interrupts

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
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

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

2. It sets a flag bit in a register
   - 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

2. It sets a flag bit in a 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

```
... currently executing code

interrupt occurs

next_line_of_code
```

- 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)
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:**

1. **Interrupt Enable (IE):**
   - `GPIO_enableInterrupt();`
   - `GPIO_disableInterrupt();`
   - `Interrupt_enableInterrupt();`

2. **Global Interrupt Enable:**
   - `Interrupt_enableMaster();`
   - `Interrupt_disableMaster();`

3. **Interrupt Flag Reg (IFG):**
   - `GPIO_getInterruptStatus();`
   - `GPIO_clearInterruptFlag();`

4. **Get the interrupt status of the selected pin:**

5. **Clears the interrupt status on the selected pin:**

6. **Enable interrupt in the peripheral unit:**

7. **Enable interrupt in the interrupt controller:**

8. **Globally allow / disallow the processor to react to interrupts:**

**Interrupt Enable:**

- **“Global” Int Enable:**
  - Enables ALL maskable interrupts

**Interrupt Source:**

- **GPIO**
- **TIMER_A**
- ...
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*.

```c
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.

```c
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:

```c
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 $D \leq T$.  

![Diagram of polling and interrupt](image.png)
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 $u_p = \frac{c}{D-c}$.

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

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

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 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 (ES-Lab) has the following *clock sources*:

<table>
<thead>
<tr>
<th>clock source</th>
<th>frequency</th>
<th>precision</th>
<th>current</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFXTCLK</td>
<td>32 kHz</td>
<td>0.0001% / °C... 0.005% / °C</td>
<td>150 nA</td>
<td>external crystal</td>
</tr>
<tr>
<td>HFXTCLK</td>
<td>48 MHz</td>
<td>0.0001% / °C... 0.005% / °C</td>
<td>550 µA</td>
<td>external crystal</td>
</tr>
<tr>
<td>DCOCLK</td>
<td>3 MHz</td>
<td>0.025% / °C</td>
<td>N/A</td>
<td>internal</td>
</tr>
<tr>
<td>VLOCLK</td>
<td>9.4 kHz</td>
<td>0.1% / °C</td>
<td>50 nA</td>
<td>internal</td>
</tr>
<tr>
<td>REFOCLK</td>
<td>32 kHz</td>
<td>0.012% / °C</td>
<td>0.6 µA</td>
<td>internal</td>
</tr>
<tr>
<td>MODCLK</td>
<td>25 MHz</td>
<td>0.02% / °C</td>
<td>50 µA</td>
<td>internal</td>
</tr>
<tr>
<td>SYSOSC</td>
<td>5 MHz</td>
<td>0.03% / °C</td>
<td>30 µA</td>
<td>internal</td>
</tr>
</tbody>
</table>
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 (ES-Lab):**

- 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

General Aspect of Timers
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*).

![Diagram of timer components](image)
Timer

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

**capture**

- clock input → counter register → interrupt on overflow
- capture event → capture register → capture actions

- 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**

- clock input → counter register → interrupt on overflow
- compare register → compare actions

- 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
Clocks and Timers

System Tick
SysTick MSP432 (ES-Lab)

- **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.

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

```c
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;
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*).

![Diagram](image-url)
Timer

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

**Capture**
- The value of the *counter register* is stored in the *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.

![Diagram of PWM](image)

- One compare register is used to *define the period*.
- Another compare register is used to *change the duty cycle* of the signal.

```
0x0000
0xFFFF
```

- counter register

- output signal
**Timer Example MSP432 (ES-Lab)**

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

- TXCLK (external)
- ACLK
- SMCLK
- inverted TXCLK

Clock sources:

- Divide (up to ÷ 64)
- 16-bit Counter
- Enable
- Interrupt
- CCR0
- CCR1
- CCR2
- CCR3
- CCR4
- CCR5
- CCR6

7 configurable compare or capture registers.
Timer Example MSP432 (ES-Lab)

**Example:** Configure Timer in “continuous mode”. **Goal:** generate periodic interrupts.
Example: Configure Timer in “continuous mode”. Goal: generate periodic interrupts, but with configurable periods.

```c
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;

so far, nothing happens
only the counter is running

configure *continuous mode*
of timer instance A0

start counter A0 in
continuous mode
Example:

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

```c
#define PERIOD 32768
```

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

```c
const Timer_A_CompareModeConfig compareModeConfig = {
    TIMER_A_CAPTURECOMPARE_REGISTER_1,
    TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE,
    0,
    PERIOD};

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

    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();
...```
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.

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

```
CPU
WDT_A_holdTimer();
WDT_A_clearTimer();

Watchdog Timer (WDT_A)
reset counter to 0
pause counting up
up counter
overflow

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

clock input, e.g., SMCLK, ACLK
Reality Check – Where does this all matter?
Miniature Low-Power Wireless Sensors

- Static, low-rate sensing (120 sec)
- Simple scalar values: voltages, resistivity, digital sensors
- 4-5 years operation (~150 μA avg. power)
- ~0.1 Mbyte/node/day
- 10+ years experience, ~3’764’408’638 data points
Many Independent Sensors Form a Network

Sensor Measurements → Wireless Network → Data Management
Wireless sensors cover the Hörnligrat from first couloir up to Eisloch.
Challenge: The Physical Environment

- Strong daily variation of temperature
  - $-30^\circ C$ to $+40^\circ C$
  - $\Delta T \leq 20^\circ C$/hour
- Impact on
  - Fatigue, energy availability, timing, SOFTWARE, ...

![Graph showing temperature variation over time for different nodes.](image)
Impact of Environmental Extremes

- Tight timing control increases energy efficiency
- Software testing in a climate chamber
- Validation of correct function for years