Návrh systémů IoT

7. Mikrokontroléry II.

Stanislav Vítek
Katedra radioelektroniky
České vysoké učení technické v Praze

Obsah přednášky

V čem se dá ještě programovat?

Serializace

Přístrojové sběrnice

RTOS

Vícevláknové programování

V čem se dá ještě programovat (nejen RPi Pico)?

C/C++

- C/C++ SDK

```
#include "pico/stdlib.h"

const uint LED_PIN = 25;

int main() {
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);
    while (1) {
        gpio_put(LED_PIN, 0);
        sleep_ms(250);
        gpio_put(LED_PIN, 1);
        sleep_ms(1000);
    }
}
```

CircuitPython

```
import time
import board
import digitalio

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

Javascript

- Tiny Javascript runtime Kaluma, github
- Javascript engine (JerryScript)[https://jerryscript.net/]

```
var led = 25;
pinMode(led, OUTPUT);

setInterval(() => {
  digitalToggle(led);
}, 1000);
```

Arduino

- Komunitní port založený na C SDK, dokumentace
- Oficiální port založený na Arm Mbed

```
#define LED 25

void setup() {
  pinMode(LED, OUTPUT);
}

void loop() {
  digitalWrite(LED, HIGH);
  delay(1000);
  digitalWrite(LED, LOW);
  delay(1000);
}
```

Rust

- Rust podporuje RP2040 prostřednictvím HAL
- Existuje řada tutoriálů a knih,
- Porovnání Rust vs. C
- Love: 10 Reasons Not To Use Rust (The Whole Truth)
- Hate: Stop writing Rust

```rust
#![no_std]
#![no_main]

use cortex_m_rt::entry;
use defmt::*;
use defmt_rtt as _;
use embedded_hal::digital::v2::OutputPin;
use embedded_time::fixed_point::FixedPoint;
use panic_probe as _;
use rp2040_hal as hal;

use hal::{
    clocks::{init_clocks_and_plls, Clock},
    pac,
    io::Sio,
    watchdog::Watchdog
};

#[link_section = ".boot2"]
#[used]
pub static BOOT2: [u8; 256] = rp2040_boot2::BOOT_LOADER_W25Q080;
```

```rust
#[entry]
fn main() -> ! {
    let mut pac = pac::Peripherals::take().unwrap();
    let core = pac::CorePeripherals::take().unwrap();
    let mut watchdog = Watchdog::new(pac.WATCHDOG);
    let sio = Sio::new(pac.SIO);

    let external_xtal_freq_hz = 12_000_000u32;
    let clocks = init_clocks_and_plls(
        external_xtal_freq_hz,
        pac.XOSC,
        pac.CLOCKS,
        pac.PLL_SYS,
        pac.PLL_USB,
        &mut pac.RESETS,
        &mut watchdog,
    )
    .ok()
    .unwrap();

    let mut delay = cortex_m::delay::Delay::new(core.SYST, clocks.system_clock.freq().integer());
```

```
    let pins = hal::gpio::Pins::new(
        pac.IO_BANK0,
        pac.PADS_BANK0,
        sio.gpio_bank0,
        &mut pac.RESETS,
    );

    let mut led_pin = pins.gpio25.into_push_pull_output();

    loop {
        led_pin.set_high().unwrap();
        delay.delay_ms(500);
        led_pin.set_low().unwrap();
        delay.delay_ms(500);
    }
}
```

Lua

```
gpio_pin = 25
pico.gpio_set_function (gpio_pin, GPIO_FUNC_SIO)
pico.gpio_set_dir (gpio_pin, GPIO_OUT)
while true do
  pico.gpio_put (gpio_pin, HIGH)
  pico.sleep_ms (300)
  pico.gpio_put (gpio_pin, LOW)
  pico.sleep_ms (300)
end
```

Go

- Kompilátor TinyGo, getting started, YouTube tutoriál

```
package main

import (
    "machine"
    "time"
)

func main() {
    led := machine.LED
    led.Configure(machine.PinConfig{Mode: machine.PinOutput})
    for {
        led.Low()
        time.Sleep(time.Millisecond * 500)

        led.High()
        time.Sleep(time.Millisecond * 500)
    }
}
```

Serializace dat

- Potřeba serializace vzniká vždy, když je třeba data přenášet přes rozhraní, jako je socket, UART nebo rozhraní jako I2C nebo SPI.
- Přenos dat sériovým rozhraním vyžaduje, aby data byla prezentována jako lineární posloupnosti bajtů.
- **Problém:** jak převést libovolný objekt (třeba v Pythonu) na takovou posloupnost a jak následně objekt obnovit.
- Pro dosažení tohoto cíle existuje řada standardů, z nichž pět je pro MicroPython snadno dostupných. Každý z nich má své výhody a nevýhody. Ve dvou případech je cílem zakódovaných řetězců, aby byly čitelné pro člověka a obsahovaly znaky ASCII. V ostatních se skládají z binárních bajtových objektů, kde bajty mohou nabývat všech možných hodnot.

Serializace dat v MicroPythonu

1. ujson, pickle (ASCII, official)
2. ustruct (binary, official)
    - vyžaduje, aby vysílací a přijímací strana sdíleli schéma, délka zprávy je fixní
3. MessagePack (binary, unofficial)
    - umožňuje, aby se za běhu měnila struktura i délka zprávy
4. protobuf (binary, unofficial)
    - délka zprávy se může za běhu změnit, ale struktura nikoli.

Přenos přes nespolehlivé linky

- Uvažujme systém, v němž vysílač periodicky posílá zprávy příjemci prostřednictvím komunikačního spojení.
- Problém s rámováním zpráv vzniká, pokud je toto spojení nespolehlivé, což znamená, že při přenosu může dojít ke ztrátě nebo poškození dat.
- V případě formátů ASCII s oddělovačem může přijímač, jakmile zjistí problém, zahazovat znaky, dokud nepřijme oddělovač, a pak čekat na kompletní zprávu.
- V případě binárních formátů je obecně nemožné provést opětovnou synchronizaci na souvislý tok dat.
    - V případě pravidelných dávek dat lze použít časový limit.
    - V opačném případě je nutná signalizace, kdy přijímač signalizuje vysílači, aby si vyžádal opakování přenosu.

ujson a pickle

- Výhodou ujson je, že řetězce JSON mohou být akceptovány jazykem Python i jinými jazyky.
- Nevýhodou je, že pouze podmnožinu objektových typů jazyka Python lze převést na legální řetězce JSON; jedná se o omezení specifikace JSON.

- Výhodou pickle je, že akceptuje jakýkoli objekt jazyka Python s výjimkou instancí uživatelsky definovaných tříd.
- Extrémně jednoduchý zdrojový kód lze nalézt v oficiální knihovně.
- Vytvořené řetězce jsou nekompatibilní s pickle CPythonu, ale lze je dekódovat v CPythonu pomocí dekodéru MicroPython.

Příklady

```
import pickle
data = {1:'test', 2:1.414, 3: [11, 12, 13]}
s = pickle.dumps(data)
print('Human readable data:', s)
v = pickle.loads(s)
print('Decoded data (partial):', v[3])
```

```
import ujson
data = {'1':'test', '2':1.414, '3': [11, 12, 13]}
s = ujson.dumps(data)
print('Human readable data:', s)
v = ujson.loads(s)
print('Decoded data (partial):', v['3'])
```

Data proměnné délky

- V reálných aplikacích se data, a tedy i délka řetězce, za běhu mění.
- Přijímací proces potřebuje vědět, kdy byl přijat celý řetězec.
- V praxi ujson a pickle nezahrnují do kódovaných řetězců znaky nového řádku.
  - Pokud kódovaná data obsahují nový řádek, je v řetězci escapován.

```
import ujson
data = {'1':b'test\nmore', '2':1.414, '3': [11, 12, 13]}
s = ujson.dumps(data)
print('Human readable data:', s)
v = ujson.loads(s)
print('Decoded data (partial):', v['1'])
```

ustruct

- Binární formát ustruct je efektivní, ale formát sekvence se za běhu nemůže změnit a musí být znám procesu dekódování.
- Záznamy mají pevnou délku. Pokud mají být data uložena v binárním souboru s náhodným přístupem, znamená pevná velikost záznamu, že lze snadno vypočítat offset daného záznamu.

```
import ustruct
fmt = 'iii'  # Record format: 3 signed ints
rlen = ustruct.calcsize(fmt)  # Record length
buf = bytearray(rlen)
with open('myfile', 'wb') as f:
    for x in range(100):
        y = x * x
        z = x * 10
        ustruct.pack_into(fmt, buf, 0, x, y, z)
        f.write(buf)
```

MessagePack

- Z binárních formátů je tento formát nejjednodušší na použití a může být "náhradou" za ujson, protože podporuje stejné čtyři metody dump, dumps, load a loads.
- Aplikace může být zpočátku vyvíjena s protokolem ujson, později se protokol změní na MessagePack.
- Vytvoření řetězce MessagePack lze provést pomocí:

```
import umsgpack
obj = [1.23, 2.56, 89000]
msg = umsgpack.dumps(obj)  # msg is a bytes object
```

- Více v dokumentaci.

Protocol Buffers

- Jedná se o standard společnosti Google popsaný např. v tomto článku.
- Záznamy mají proměnnou délku, přenáší se řetězce a čísla libovolné velikosti.
- Implementace kompatibilní s MicroPythonem je mikro implementace:
  - soubory .proto nejsou podporovány,
  - datový formát však deklaruje kompatibilitu s jinými platformami a jazyky.
- Schéma je tuple definující strukturu datového diktu.
  - Každý prvek deklaruje klíč a jeho datový typ ve vnitřním tuplu.
  - Prvky tohoto vnitřního tuplu jsou řetězce, přičemž prvek 0 definuje klíč pole.
  - Následující prvky definují datový typ pole; ve většině případů je datový typ definován jediným řetězcem.

Datové typy

- Popsány v dokumentaci

1. 'U' Řetězec v kódování UTF8.
2. 'a' Objekt bajtů.
3. 'b' Logická hodnota.
4. 'f' 32bitový float: obvyklé výchozí nastavení MicroPythonu.
5. 'z' Int: celé číslo se znaménkem libovolné délky. Efektivně zakódováno pomocí důmyslného algoritmu.
6. 'd' 64bitový float s dvojnásobnou přesností.
7. 'x' Prázdné pole.

Příklad

```
import minipb

schema = (('value', 'z'),)  # Dict will hold a single integer
w = minipb.Wire(schema)

data = {'value': 0}
data['value'] = 150
tx = w.encode(data)
rx = w.decode(tx)  # received data
print(rx)
```

- Tento příklad zamlčuje skutečnost, že v reálné aplikaci se budou data měnit a délka přenášeného řetězce tx se bude měnit. Přijímací proces musí znát délku každého řetězce.

Příklad vysílající strany

```
import minipb
schema = (('value', 'z'),
          ('float', 'f'),
          ('signed', 'z'),)
w = minipb.Wire(schema)
# Create a dict to hold the data
data = {'value': 0,
        'float': 0.0,
        'signed' : 0,}
while True:
    # Update values then encode and transmit them, e.g.
    # data['signed'] = get_signed_value()
    tx = w.encode(data)
    # Data lengths may change on each iteration
    # here we encode the length in a single byte
    dlen = len(tx).to_bytes(1, 'little')
    send(dlen)
    send(tx)
```

Příklad přijímající strany

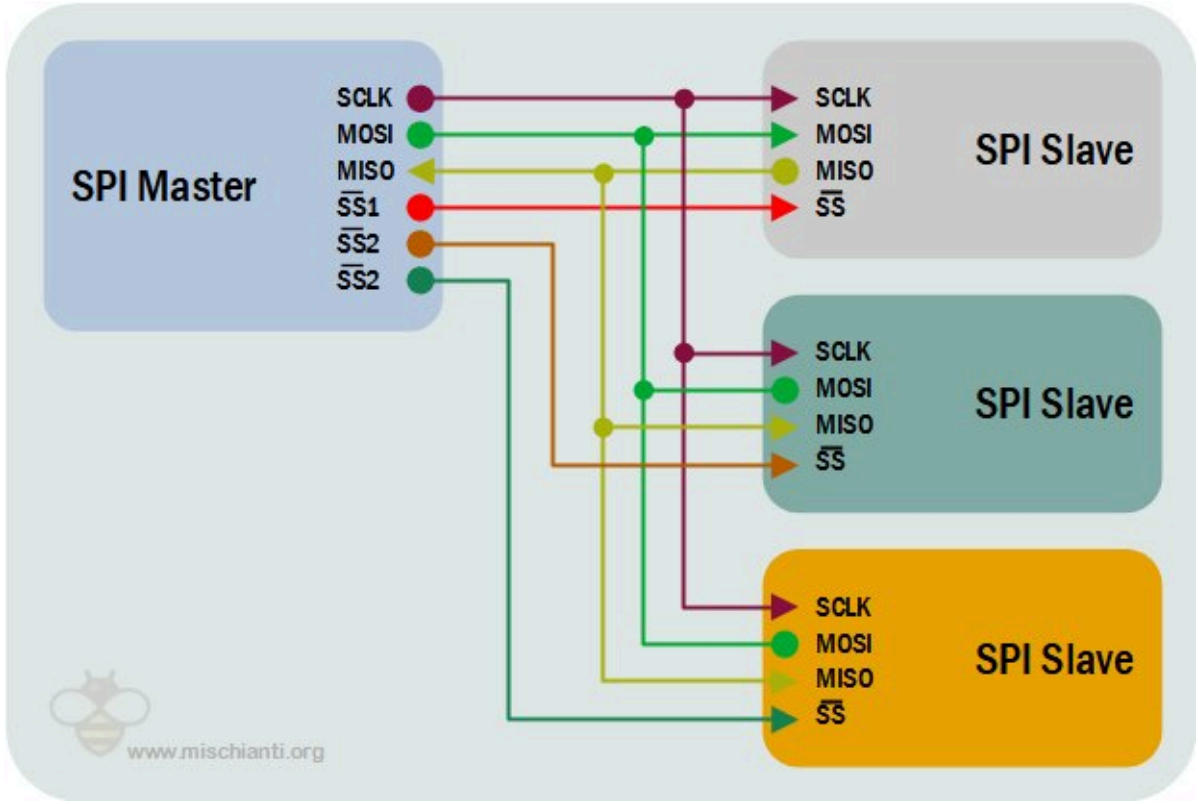
```
import minipb
# schema must match transmitter. Typically both would import this.
schema = (('value', 'z'),
          ('float', 'f'),
          ('signed', 'z'),)

w = minipb.Wire(schema)
while True:
    dlen = receive(1)  # Data length stored in 1 byte
    data = receive(dlen)  # Retrieve actual data
    rx = w.decode(data)
    # Do something with the received dict
```

Přístrojové sběrnice

1. SPI
2. I2C
3. 1-wire

SPI

- Sériová sběrnice, full duplex
- Single master, multiple slave
- MISO, MOSI, SCLK, SS/CS
- Typická rychlost 10Mbps
- Vzdálenost do 10m
- Jednoduchá implementace
  - není třeba adresace
- Nevýhody
  - více zařízení = více drátů
  - bez kontroly dat



SPI v MicroPythonu

- Sběrnice SPI je popsána třídou machine.SPI, SW i HW implementace

```
from machine import SPI, Pin

spi = SPI(0, baudrate=400000)        # Create SPI peripheral 0 at frequency of 400kHz.
                                     # Depending on the use case, extra parameters may be required
                                     # to select the bus characteristics and/or pins to use.
cs = Pin(4, mode=Pin.OUT, value=1)   # Create chip-select on pin 4.

try:
  cs(0)                              # Select peripheral.
  spi.write(b"12345678")             # Write 8 bytes, and don't care about received data.
finally:
   cs(1)                             # Deselect peripheral.
```

SPI - konstruktory a metody

- Hardware SPI

```
class machine.SPI(id, ...)
```

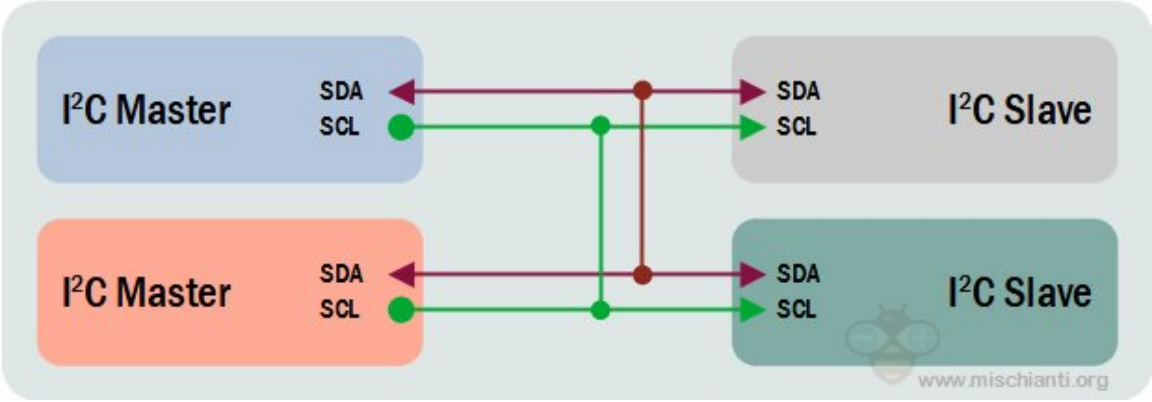
- Software SoftSPI

```
class machine.SoftSPI(baudrate=500000, *, polarity=0, phase=0, bits=8,
                      firstbit=MSB, sck=None, mosi=None, miso=None)
```

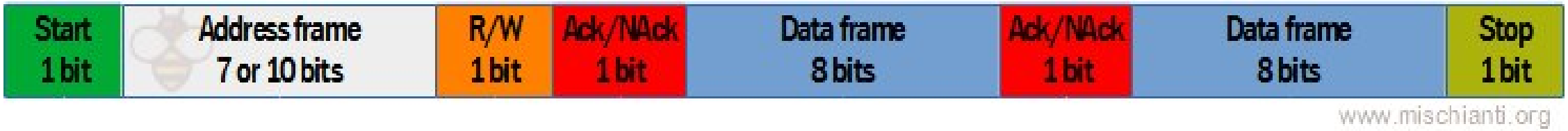
- inicializace init(), deinit()
- zápis read(), readinto()
- čtení write(), write_readinto()

I2C

- Sériová synchronní sběrnice, half duplex
  - multimaster, multislave, dva vodiče: SDA, SCL
  - podporuje rychlosti 100 kbps, 400 kbps, a 3.4 Mbps (příp. 10 kbps a 1 Mbps)
  - spíše kratší vzdálenosti, typicky max. 30 cm



I2C protokol



- START: SDA se přepne HIGH->LOW před tím, než SCL přepne HIGH->LOW
- Stop: SDA se přepne LOW->HIGH poté, co SCL přejde LOW->HIGH
- Adresní rámec: 7 nebo 10bitová adresa slave, se kterým chce master komunikovat.
- Bit pro čtení/zápis: bit určující, zda master posílá data podřízenému zařízení (LOW) nebo od něj požaduje data (HIGH).
- ACK/NACK Bit: Za každým rámcem ve zprávě následuje potvrzovací/nepotvrzovací bit. Pokud byl rámec adresy nebo dat úspěšně přijat, je odesílateli vrácen bit ACK od přijímajícího zařízení.

I2C v MicroPythonu

- Sběrnice I2C je popsána třídou machine.I2C

```
from machine import I2C

i2c = I2C(freq=400000)           # create I2C peripheral at frequency of 400kHz
                                 # depending on the port, extra parameters may be required
                                 # to select the peripheral and/or pins to use

i2c.scan()                       # scan for peripherals, returning a list of 7-bit addresses

i2c.writeto(42, b'123')          # write 3 bytes to peripheral with 7-bit address 42
i2c.readfrom(42, 4)              # read 4 bytes from peripheral with 7-bit address 42

i2c.readfrom_mem(42, 8, 3)       # read 3 bytes from memory of peripheral 42,
                                 #   starting at memory-address 8 in the peripheral
i2c.writeto_mem(42, 2, b'\x10') # write 1 byte to memory of peripheral 42
                                 #   starting at address 2 in the peripheral
```

I2C - konstruktory a metody

- Hardware I2C

```
class machine.I2C(id, *, scl, sda, freq=400000)
```

- Software SoftI2C

```
class machine.SoftI2C(scl, sda, *, freq=400000, timeout=50000)
```

- Inicializace init() a deinicializace deinit()
- Skenování scan()
  - prohledá všechny adresy I2C mezi 0x08 a 0x77 včetně a vrátí seznam těch, které odpovídají
  - zařízení reaguje, pokud po odeslání své adresy (včetně zapisovacího bitu) na sběrnici přitáhne SDA.

I2C - nízkoúrovňové funkce třídy SoftI2C

- start() - START podmínka (SDA přechází na LOW hodnotu, zatímco SCL je HIGH).
- stop() - STOP podmnínka (SDA přechází na HIGH, zatímco SCL je HIGH)
- readinto() - čte data ze sběrnice a ukládá je do bufferu.
  - Počet načtených bajtů odpovídá délce bufferu.
  - Po přijetí všech bajtů kromě posledního se na sběrnici odešle ACK. Po přijetí posledního bajtu lze odesláním NACK nebo ACK rozli3it, zda se budou číst data v pozdějším volání.
- write() - zápis dat z bufferu na sběrnici.
  - Kontroluje, zda je po každém bajtu přijato ACK, a v případě přijetí NACK přeruší zápis zbývajících dat. Funkce vrací počet přijatých ACK.

Standardní I2C operace - čtení

```
I2C.readfrom(addr, nbytes, stop=True, /)
```

- readfrom() - přečte nbytes z periferie zadané pomocí addr.
  - Pokud je stop true, pak je na konci přenosu generována podmínka STOP.
  - Vrací objekt bytes s přečtenými daty.

```
I2C.readfrom_into(addr, buf, stop=True, /)
```

- readfrom_into() - čtení do buf z periferie zadané pomocí addr.
  - Počet přečtených bajtů bude odpovídat délce buf.
  - Pokud je stop true, pak je na konci přenosu generována podmínka STOP.

Standardní I2C operace - zápis

```
I2C.writeto(addr, buf, stop=True, /)
```

- writeto() - zapíše bajty z buf do periferie zadané pomocí addr. Pokud je po zápisu bajtu z buf přijat NACK, zbývající bajty se neodešlou. Pokud je stop true, pak je na konci přenosu generována podmínka STOP, i když je přijat NACK. Funkce vrací počet přijatých ACK.

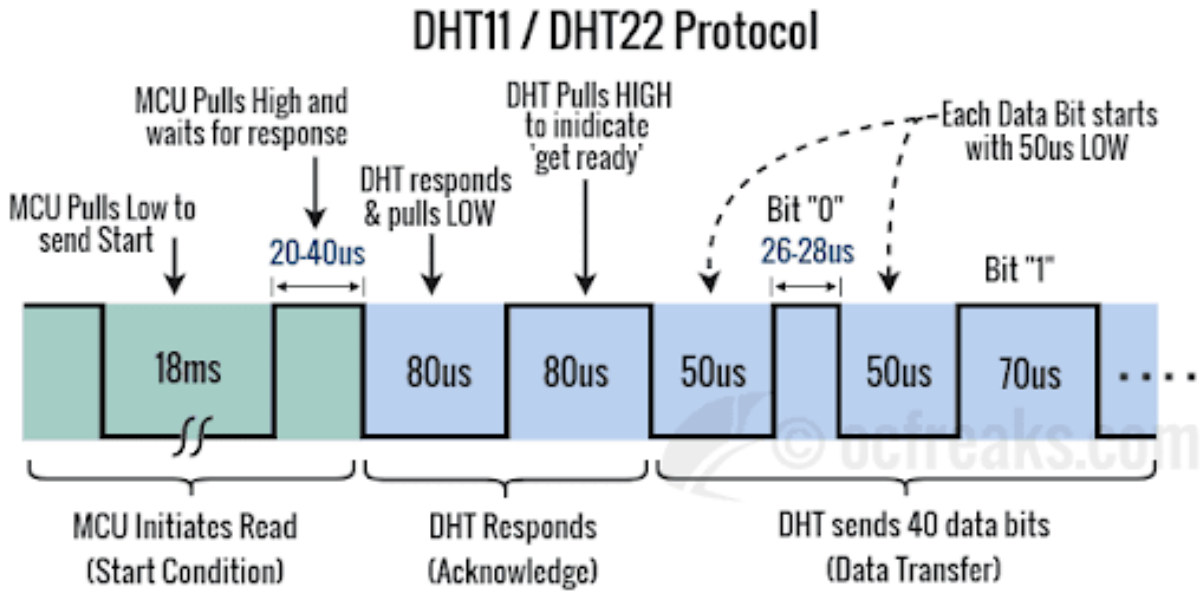
```
I2C.writevto(addr, vector, stop=True, /)
```

- writevto() - zapíše bajty obsažené ve vektoru na periferii zadanou pomocí addr. vektor by měl být tuple nebo seznam objektů s protokolem bufferu. Addr se odešle jednou a pak se postupně vypíší bajty z každého objektu ve vektoru. Objekty ve vektoru mohou mít délku nula bajtů, v takovém případě se na výstupu nepodílejí.

1-wire, DHT11/22

- podobná I²C, jen s nižší datovou propustností a delším dosahem.

```
from dht import DHT22
from machine import Pin

p = Pin(4)

d = DHT22(p)

d.measure()
d.temperature()
d.humidity()
```



Řešení problémů

```
import machine, dht, time, sys

d = dht.DHT22(machine.Pin(4))

while True:
  try:
    d.measure()
    time.sleep(2)
    tem = d.temperature()
    hum = d.humidity()
    print('Temp:', hum, 'C\tHumidity', tem, '%')
    time.sleep(1)
  except OSError as e:
    print("Cant read:", e)
    sys.print_exception(e)
```

Klasifikace IoT zařízení

Vestavný systém

- vykonává omezený soubor specifických funkcí;
- často komunikuje se svým okolím.

RT systém

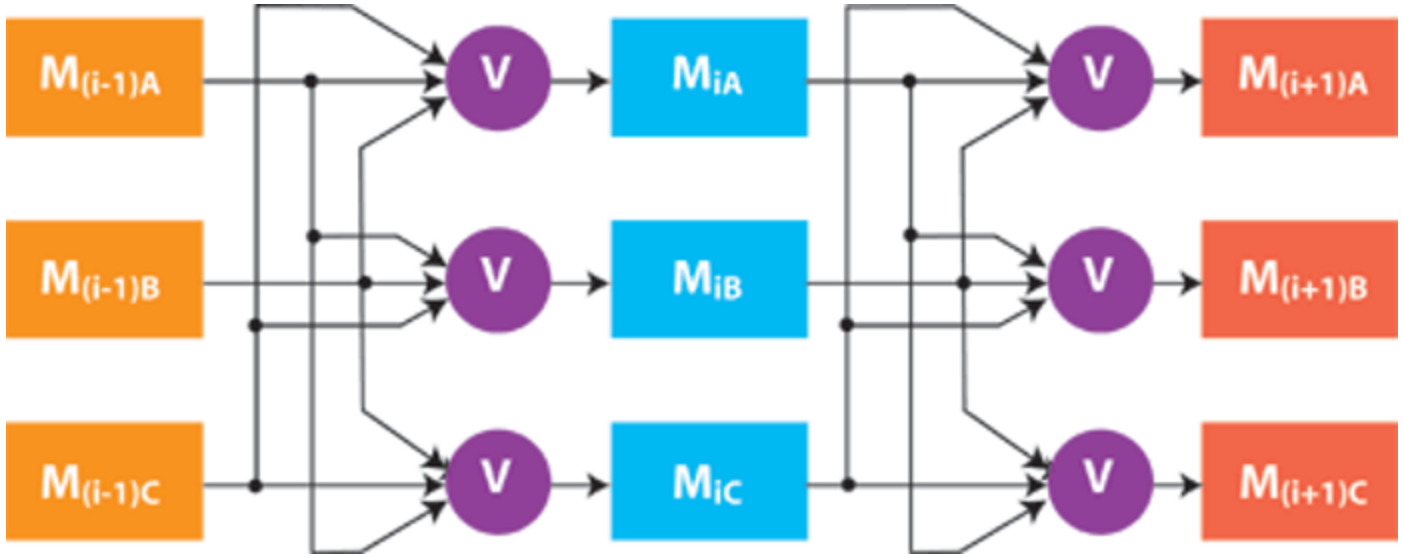
- správnost systému závisí nejen na logických výsledcích, ale také na čase, za který jsou výsledky vytvořeny.

Příklady systémů pracujících v reálném čase

- Real-time vestavné:
  - Systémy kritické z hlediska bezpečnosti: řízení jaderného reaktoru, řízení letu
  - GPS, MP3 přehrávač, mobilní telefon
- Real-time, ale ne vestavné:
  - Plaforma pro burzovní operace
  - Skype, Youtube, Netflix
- Vestavné, ale ne real-time:
  - Domácí termostat, zavlažovací systém
  - Pračka, lednička

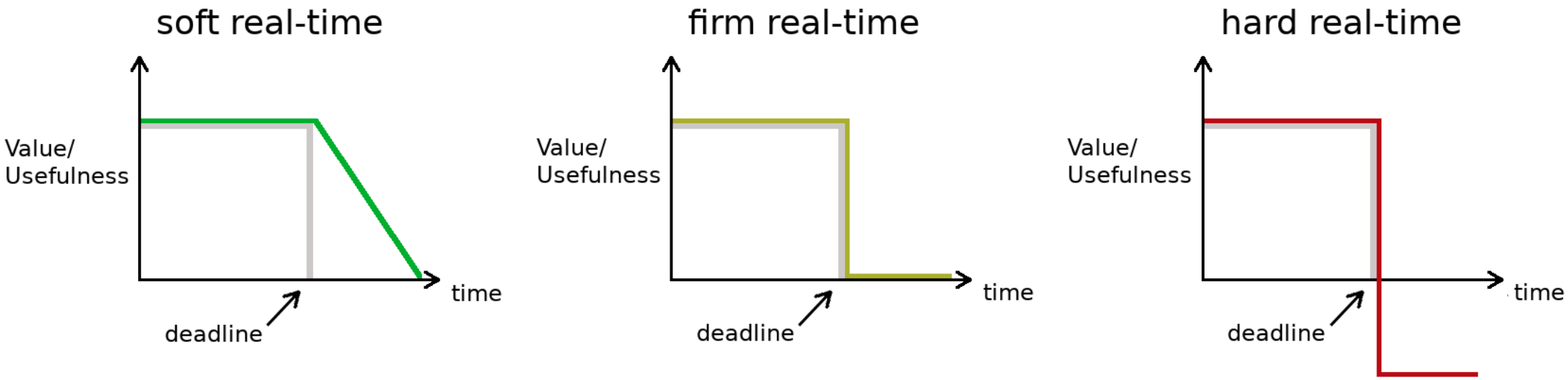
Charakteristiky real-time systémů

- Řízení událostí (reaktivní) vs. řízení podle času
- Požadavky na spolehlivost/odolnost proti poruchám (příklad: trojnásobná modulární redundance)
- Předvídatelnost
- Priority ve víceúlohových systémech



Klasifikace RT systémů

- Hard RT: reakce na vstupy musí přijít v požadovaném termínu - systémy řízení letů.
  - Real RT: navíc velmi krátká odezva, např. systém navádění raket
- Soft RT: termíny jsou důležité, ale systém bude správně fungovat i v případě, že budou termíny občas nedodrženy. Např. systém sběru dat.
- Firm RT: několik zmeškaných termínů nepovede selhání, ale více než několik zmeškaných termínů může vést k úplnému nebo katastrofickému selhání systému.



Klasifikace RT systémů

Statické

- Lze předvídat časy příchodu úkolů
- Možnost statické analýzy (v době kompilace)
- Umožňuje dobré využití zdrojů (nízká doba nečinnosti procesorů)

Dynamické

- Nepředvídatelné časy příjezdu
- Statická analýza (v době kompilace) je možná pouze pro jednoduché případy
- Využití procesoru se dramaticky mění - návrh tak, aby zvládl "nejhorší případ".
- Je třeba se vyvarovat příliš zjednodušujících předpokladů, např. předpokladu, že všechny úlohy jsou nezávislé, i když je to nepravděpodobné.

Klasifikace RT systémů

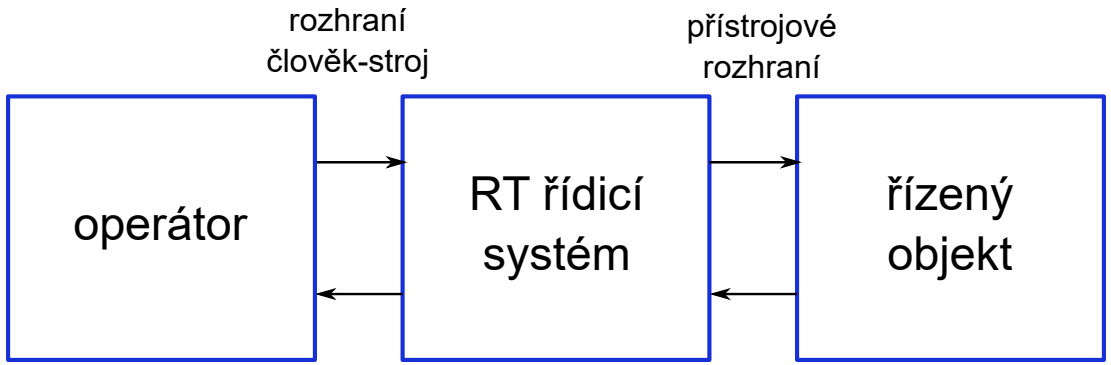
Periodické

- Každá úloha (nebo skupina úloh) se provádí opakovaně s určitou periodou.
- Umožňuje použití některých technik statické analýzy
- Odpovídá charakteristikám mnoha skutečných problémů
- Je možné mít úlohy s termíny menšími, rovnými nebo většími, než je jejich perioda - pozdější jsou obtížně zpracovatelné, vyskytuje se více souběžných instancí úloh

Neperiodické (sporadické, asynchronní nebo reaktivní)

- Vytváří dynamickou situaci
- Časově omezené intervaly příchodu jsou snadněji zvládnutelné
- Systémy s omezenými zdroji nezvládnou neomezené časové intervaly příchodu

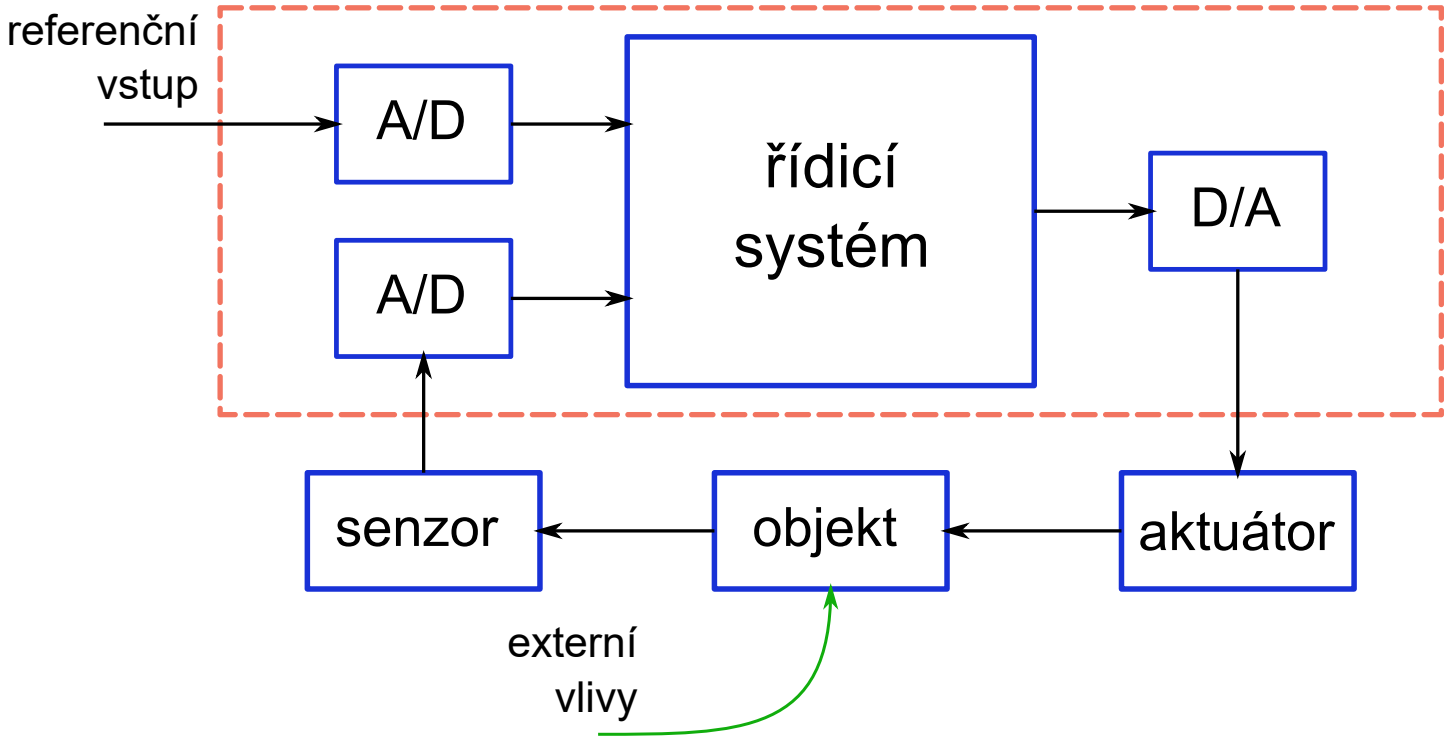
Řídicí systémy



- Rozhraní člověk-stroj: vstupní zařízení (klávesnice), a výstupní zařízení (displej).
- Přístrojové rozhraní: snímače a akční členy, převádějí fyz. signály na dig. data.
- Většina řídicích systémů je v tvrdém reálném čase
- Termíny jsou určeny řízeným objektem, tj. časovým chováním fyzikálního jevu (vstřikování paliva vs. ATM).

Příklad řídicího systému

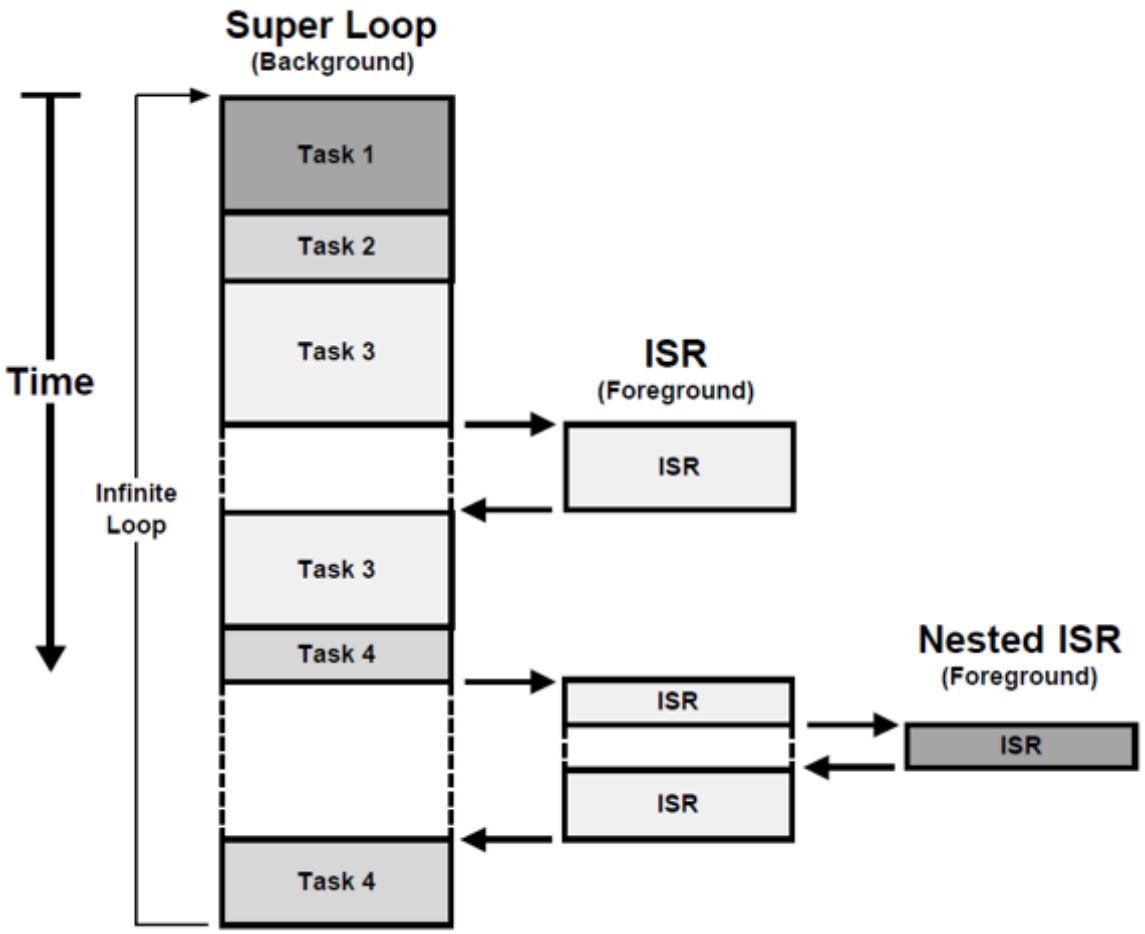
- Jednoduchý systém s jedním senzorem a jedním aktuátorem



Operační systém?

Běh systému

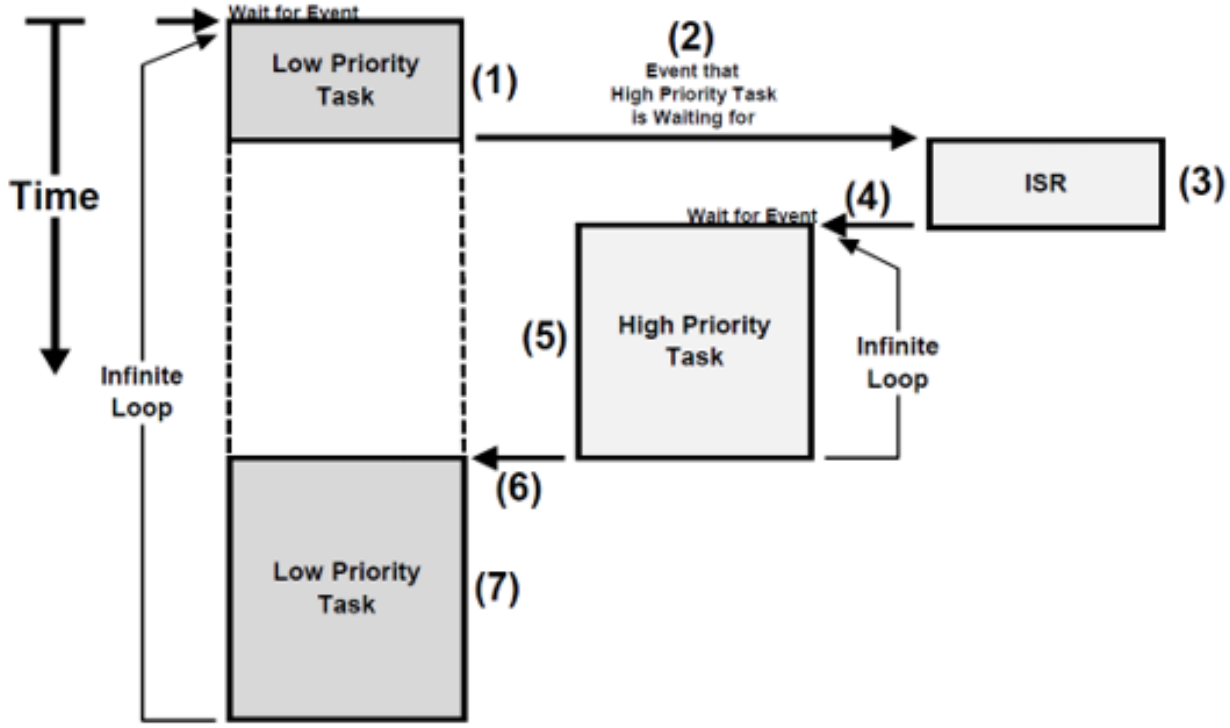
- Úroveň úlohy, úroveň přerušení
- Kritické operace se musí provádět na úrovni přerušení (není dobré)
- Doba odezvy/časování závisí na celé smyčce
- Změna kódu ovlivňuje časování
- Jednoduché, levné systémy



RT systémy

- Časové požadavky různých podnětů/odpovědí
  - architektura systému umožňovat rychlé přepínání mezi zpracovateli podnětů.
- Různé prioritám, neznámé pořadí a různé časové požadavky
  - sekvenční smyčka obvykle nevyhovuje.

RT systémy se proto obvykle navrhují jako procesy spolupracující s jádrem reálného času, které tyto procesy řídí.



Plánovací strategie

- **Nepreemptivní plánování** - jakmile je proces naplánován k provedení, běží až do dokončení nebo dokud není z nějakého důvodu zablokován (např. čekání na vstup/výstup).
- **Preemptivní plánování** - Provádění prováděných procesů může být zastaveno, pokud proces s vyšší prioritou vyžaduje obsluhu.

Plánovací algoritmy

- Round-robin - běžícímu procesu přiděluje kvantum času, po jeho uběhnutí je proces odstaven, předpokládá se konstatní priorita
- Rate monotonic - statické přidělování priorit
- Earliest deadline first - zpracování úloh probíhá na základě mezní doby platnosti procesu

RT operační systémy - RTOS

- RTOS jsou specializované operační systémy, které řídí procesy v RTS.
- Odpovídají za správu procesů a přidělování zdrojů (procesoru a paměti).
- Komponenty
  - **Real-time clock** poskytuje informace pro plánování procesů
  - **Interrupt handler** spravuje nepravidelné žádosti o obsluhu
  - **Scheduler** vybere další proces, který má být spuštěn
  - **Resource manager** alokuje paměť a zdroje (procesor, vlákno, ...)
  - **Dispatcher** spustí proces

Obsluha přerušení v RTOS

- Ovládání se automaticky přenese do předem určeného místa v paměti.
- Toto místo obsahuje instrukci pro skok do rutiny obsluhy přerušení.
- Další přerušení (od stejného zdroje) jsou zakázána
- Přerušení je obslouženo a řízení je vráceno přerušenému procesu.
- Obslužné rutiny přerušení MUSÍ být krátké, jednoduché a rychlé.

Parametry pro výběr vhodného operačního systému

- **Režie** minimální požadavky na paměť, spotřebu a výkon.
- **Přenositelnost** middleware je nezávislý na hardware, obvykle je na různé platformy a rozhraní přenášen pomocí BSP (Board Support Package)
- **Modularita** pevné jádro a další funkce by měly být přidány ve formě doplňků. OS je pak možné přizpůsobit aplikaci a zredukovat potřebné zdroje.
- **Konektivita** podpora různých protokolů, např. Ethernet, Wi-Fi, BLE, IEEE 802.15.4
- **Škálovatelnost** škálovatelnost pro jakýkoli typ zařízení -- vývojáři a integrátoři budou znát pouze jeden OS pro uzly i brány.
- **Spolehlivost** dlouhodobý běh aplikace bez selhání, certifikace pro určité aplikace.
- **Bezpečnost** bezpečné spouštění, podpora SSL a ovladačů pro šifrování.

Spolehlivost -- certifikace

- **DO-178B** pro systémy avioniky
- **IEC 61508** pro průmyslové řídicí systémy
- **ISO 62304** pro zdravotnické zařízení
- **SIL3 / SIL4** pro dopravu a jaderné systémy (Safety integrity level)

MISRA

- Motor Industry Software Reliability Association
- Sada doporučení pro bezpečné programy C a C++
- Cílem je zajistit bezpečnost a robustustnost implementace
- Gudelines nejsou volně dostupné

Operační systém RIOT

- Bezplatný operační systém s otevřeným zdrojovým kódem (LGPLv2.1)
- Programování je v jazyku C/C++ nebo Rust
- Podporuje standardní nástroje jako jsou gcc, gdb nebo valgrid.
- Architektury: AVR, ARM7, Cortex-M0, Cortex-M0 +, Cortex-M3, Cortex-M4, Cortex-M7, ESP8266, MIPS32, MSP430, PIC32, x86.
- Desky: Airfy Beacon, Arduino Due, Arduino Mega 2560, Arduino Zero, Atmel samr21-Xplained Pro, f4vi, mbed NXP LPC1768, Micro::bit, Nordic nrf51822 (DevKit), Nordic nrf52840 (DevKit), Nucleo desky (téměř všechny) a mnoho dalších.

Web: https://www.riot-os.org/

Příklad programu v RIOT OS

```
gpio_t pin_out = GPIO_PIN(PORT_B, 5);
if (gpio_init(pin_out, GPIO_OUT)) {
    printf("Error to initialize GPIO_PIN(%d %d)\n", PORT_B, 5);
    return -1;
}

while(1)
{
  printf("Set pin to HIGH\n");
  gpio_set(pin_out);
  xtimer_sleep(2);

  printf("Set pin to LOW\n");
  gpio_clear(pin_out);
  xtimer_sleep(2);
}
```

Web: https://www.hackster.io/ichatz/control-external-led-using-riot-os-b626da

Mongoose OS

- Framework pro vývoj firmwaru internetu věcí (Apache Licence 2.0 nebo Enterprise)
- Cílem je komplexní řešení pro vývoj a správu
- Nízkopříkonové MCU: ESP32, ESP8266, TI CC3200, STM32
- Cloudové integrace: AWS, Google, Azure, IBM Watson (komerční licence)
- K dispozici je Dashboard mDash (Apache 2.0)
  - stav připojených zařízení - online / offline, verze firmwaru, doba provozu atd.
  - aktualizace firmwaru OTA,
- K dispozici je také mobilní aplikace pro iOS i Android.

Web: https://mongoose-os.com/

Mongoose OS

Příklad čtení dat ze senzoru DHT22

```
#include "mgos.h"
#include "mgos_dht.h"

static void timer_cb(void *dht) {
  LOG(LL_INFO, ("Temperature: %lf", mgos_dht_get_temp(dht)));
}

enum mgos_app_init_result mgos_app_init(void) {
  struct mgos_dht *dht = mgos_dht_create(mgos_sys_config_get_app_pin(), DHT22);
  mgos_set_timer(1000, true, timer_cb, dht);
  return MGOS_APP_INIT_SUCCESS;
}
```

Contiki OS

- Operační systém s otevřeným zdrojovým kódem (BSD licence)
- Poskytuje multitasking, jádro 10kB RAM + 30kB ROM, jazyk C
- Simulátor Cooja umožňuje emulovat celou Contiki síť

Web: https://github.com/contiki-ng/contiki-ng/wiki

Projekt Zephyr

- Malý škálovatelný RTOS, původně pod patronací Linux Foundation (Apache licence)
- Jádro Zephyr je odvozeno od komerčního VxWorks Microkernel Profile.
- MCU: https://docs.zephyrproject.org/latest/boards/index.html
- Raspberry Pi Pico tutorial

Web: https://zephyrproject.org/

Blinky LED

- Zdrojový kód
- Konfigurace specifického boardu

Nucleus

- OS od divize Embedded Software společnosti Mentor Graphics
- Honeywell: systém varování před přiblížením k zemi v aplikaci pro letecký průmysl.
- Garmin: Avionics Navigator CNX80
- ZOLL: automatizovaný externí defibrilátor AED Plus
- MCU: https://www.mentor.com/embedded-software/nucleus/processor-support

---

Update 2023: RTOS převzal Siemens

Mbed OS

- RTOS určený primárně pro procesory ARM (licence Apache 2.0)
- Dostupný zdrojový kód: https://github.com/ARMmbed/mbed-os
- Podporuje BLE, NFC, RFID, LoRa, 6LoWPAN-ND, Thread, Wi-SUN, Ethernet, Wi-Fi, LPWAN
- Propojení s platformou Pelion IoT
- Nástroje: Mbed CLI, Mbed Online Compiler, Mbed Studio
- Desky, Komponenty

Web: https://os.mbed.com/

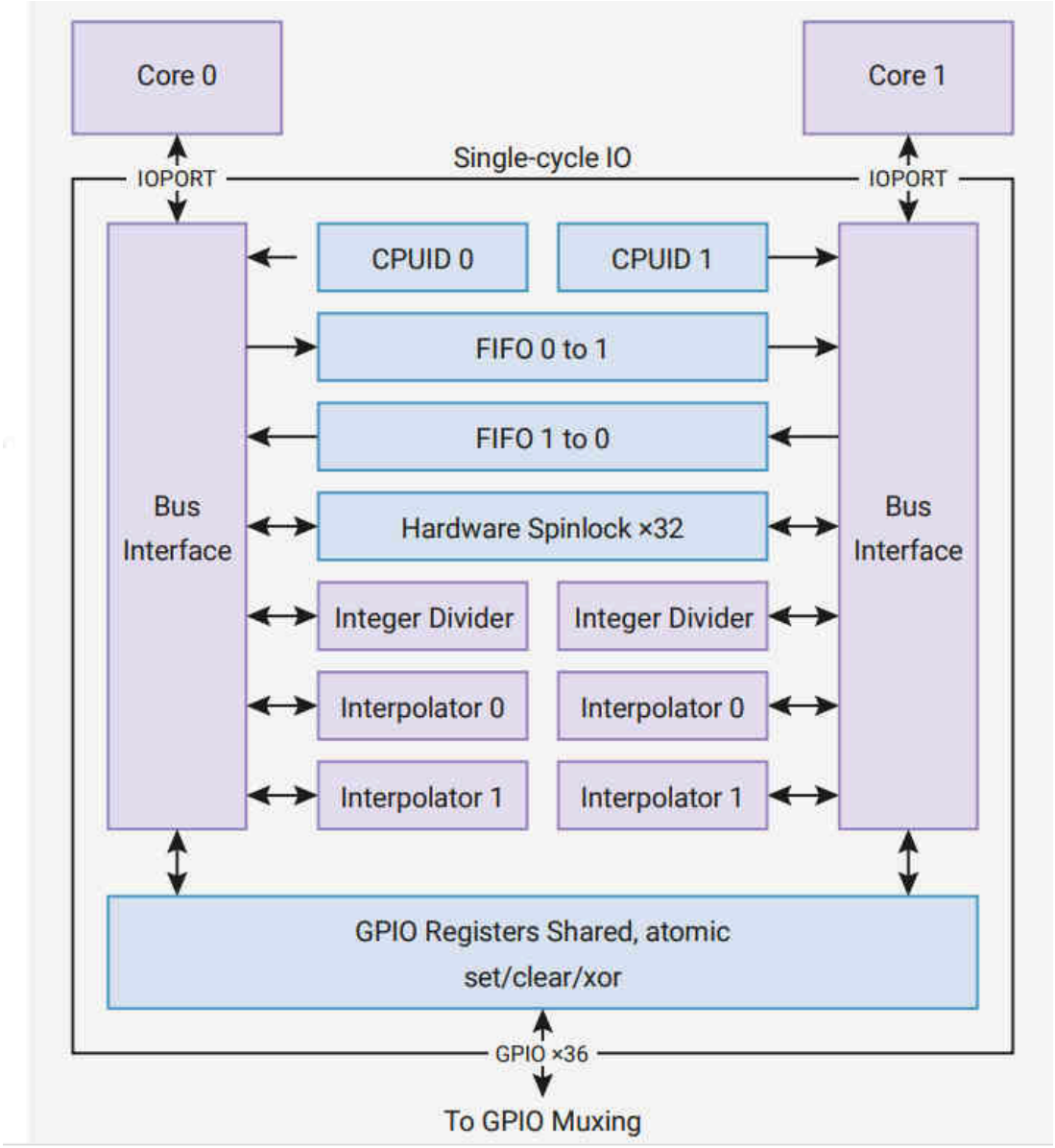
Co MicroPython?

```
import dht, machine, uasyncio as asyncio

async def report_dht_data(d):
  while True:
    await asyncio.sleep(2)
    try:
      wait_ms = d.start()
      await asyncio.sleep_ms(wait_ms)
      d.receive()
    except Exception as ex:
      print("error: {}".format(ex))
      continue

    print("temp: {}°C humi: {}%RH".format(d.temperature(), d.humidity()))

d = dht.DHT22(machine.Pin(4))
loop = asyncio.get_event_loop()
loop.create_task(report_dht_data(d))
loop.run_forever()
```

Využití více jader mikrokontroléru - vlákna

Raspberry Pi Pico

- Mikrokontrolér RP2040 obsahuje dvě jádra, **Core 0** a **Core 1**
- Defaulní mód:
  - **Core 0** vykonává všechny úlohy
  - **Core 1** zůstává ve standby módu
- MicroPython nabízí modul _thread, který umožňuje rozdělit úlohy na jednotlivá jádra.



Komunikace mezi jádry

- Aby spolu jádra mohla komunikovat, je modul Raspberry Pi Pico vybaven dvěma samostatnými FIFO.
- Každé jádro může přistupovat pouze k jedné FIFO, což pomáhá vyhnout se **race condition** nebo současnému zápisu na stejné místo v paměti.
- Modul **_thread** poskytuje synchronizační primitiva, **semafor** a **zámek**.
    - O vytvoření objektu pro uzamčení je k dispozici funkce **allocate_lock()**

Modul _thread

- Kód je automaticky spuštěn na jádře **Core 0**
- Modul **_thread** umožňuje spustit kód na **Core 1**

```
def thread_function():
  # code to be running on Core 1

new_thread = _thread.start_new_thread(thread_function, args, [,kwargs])
```

- **thread_function** je odkaz na standardní funkci jazyka Python, která obsahuje kód nového vlákna. Za ním musí následovat tuple obsahující argumenty funkce a pak nepovinný slovník nebo klíčová slova argumentů funkce.

Příklad

```
from time import sleep
import _thread

def core0_thread(counter = 0):
    while True:
        print(counter)
        counter += 2
        sleep(1)

def core1_thread(counter = 1):
    while True:
        print(counter)
        counter += 2
        sleep(2)

second_thread = _thread.start_new_thread(core1_thread, ())

core0_thread()
```

Komunikace mezi vlákny

```
def core0_thread():
    global run_core_1
    # do something
    # signal core 1 to run
    run_core_1 = True
    # wait for core 1 to finish
    while run_core_1:
      pass

def core1_thread():
    global run_core_1
    while True:
        # wait for core 0 to signal start
        while not run_core_1:
            pass
        # do something
        # signal core 0 code finished

run_core_1 = False

second_thread = _thread.start_new_thread(core1_thread, ())
core0_thread()
```

Sdílení zdrojů

- Někdy musíme velmi pečlivě sledovat, kdo a kdy může mít přístup k některým datům nebo prostředkům, např. k rozhraní SPI.
    - Pokud se obě vlákna pokusí použít nebo aktualizovat stejný prostředek současně, dojde buď k poškození dat, nebo k potenciálnímu pádu části kódu.
- V jednoduchých a dobře definovaných situacích funguje uspokojivě **vlajka**. Když potřebujete flexibilnější ovládání, musíte použít zámek.
    - Zámek umožňuje řídit přístup.
    - Vytvoříme objekt zámku a prostředek může používat pouze jeho vlastník.
    - Každé další vlákno musí počkat, až vlastník zámku zámek uvolní, a teprve potom může jedno z čekajících vláken převzít vlastnictví a získat přístup k prostředku.

Sdílení zdrojů - race condition

```
def core0_thread():
  while True:
    print('A')
    sleep(0.5)

def core1_thread():
  while True:
    print('B')
    sleep(0.5)

second_thread = _thread.start_new_thread(core1_thread, ())
core0_thread()
```

Sdílení zdrojů - použití zámku (mutex)

```
def core0_thread():
  global lock
  while True:
    lock.acquire()
    print('A'); sleep(0.5)
    lock.release()

def core1_thread():
  global lock
  while True:
    lock.acquire()
    print('B'); sleep(0.5)
    lock.release()

lock = _thread.allocate_lock()

second_thread = _thread.start_new_thread(core1_thread, ())
core0_thread()
```

Komunikace mezi vlákny využívající Frontu (Queue)

- Queue() má implementaci bezpečnou pro vlákna se všemi potřebnými zamykacími mechanismy.
  - může v podstatě předat frontu jako argument druhému vláknu.

Core0 thread:

```
q = queue.Queue()
_thread.start_new_thread(second_core_thread, (q) )

while True:
  msg = q.get()
```

Core 1 thread (second_core_thread):

```
def writer(q):
  queue.put(....)
```