

REV - Aplikace embedded systémů v mechatronice
rukověť studenta

Š. Řeřucha, Z. Matěj et al.

verze February 20, 2018

Contents

1	Embedded systémy	1
1.1	Základní konstrukční prvky digitální logiky	1
1.1.1	Logické obvody	2
1.1.2	Popis logických obvodů	3
1.1.3	Příklady základních logických obvodů	4
1.1.4	Stavové automaty	9
2	Mikrokontrolery a periferie	14
2.1	Mikrokontrolery	14
2.1.1	Klasifikace MCU	15
2.1.2	Instrukční sada	16
2.1.3	Architektura a základní operace jádra procesoru	18
2.1.4	Podpůrné obvody	19
2.2	Základní periferní obvody	20
2.2.1	GPIO: obecný vstup a výstup	20
2.2.2	Přerušení	23
2.2.3	Časovače a čítače	25
2.2.4	Konfigurační a jiné paměti	28
2.3	Komunikační metody	29
2.3.1	UART a RS232: sériová linka	29
2.3.2	SPI, I2C: komunikace mezi integrovanými obvody	31
2.3.3	CAN: spolehlivá komunikace mezi samostatnými uzly	32

2.4	Analogové veličiny	33
2.4.1	PWM: pulsně-šířková modulace	33
2.4.2	QENC: rotační enkodér	33
2.4.3	ADC/DAC: převod analog-číslo a zpět	35
3	Přehled jazyka C	37
3.1	O jazyce	37
3.1.1	Historie a standardizace jazyka C	38
3.1.2	Charakteristika jazyka C	39
3.2	Základní struktura jazyka C	42
3.2.1	Základní syntaxe, proměnné a konstanty	42
3.2.2	Operátory	46
3.2.3	Řídící struktury	48
3.2.4	Funkce a moduly	51
3.2.5	Ukazatele a složené datové struktury	53
3.2.6	Další možnosti	60
3.3	Standardní knihovna jazyka C	61
3.4	Podpora jazyka C v embedded	62

Základní konstrukční prvky digitálních systémů

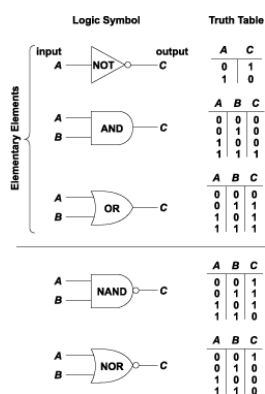
- ▶ procesor obsahuje množství tranzistorů
- ▶ tranzistory tvoří tzv. logické členy, které realizují logické funkce
- ▶ vhodnou kombinací (zpětná vazba) lze vytvořit paměťový člen
- ▶ z primitivních prvků se skládají složitější systémy



1.1.1 Logické obvody

Logické obvody jsou takové elektronické obvody, které pracují s množinou (zpravidla dvou) diskrétních signálových úrovní. Dvě úrovně, označované jako logická nula a logická jedna umožňují reprezentaci binární (booleovské logiky).

Logické členy – hradla



- ▶ NOT - negace:
 $C = \neg A = \bar{A}$
- ▶ AND - logický součin:
 $C = A \cdot B = AB$
- ▶ OR – logický součet:
 $C = A + B$

Funkčně kompletní:

- ▶ **NAND – NOT AND**
 $C = \overline{AB}$
- ▶ **NOR – NOT OR**
 $C = \overline{A + B}$

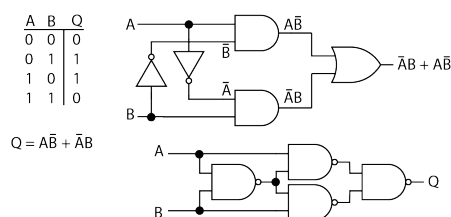


Základním stavebím prvkem logických obvodů jsou tzv. *logické členy* neboli *hradla*. Hradlo je jednoduchý logický člen, který realizuje booleovskou funkci; fyzicky je vstup a výstup realizován právě pomocí napěťových úrovní, které reprezentují logické stavy.

Posledním důležitým formalismem je *Booleovská algebra* (též Booleova algebra). Jedná se o algebraický formalismus (distributivní komplementární svaz), který pracuje s hodnotami 0 a 1, operacemi logický součin (AND), logický součet (OR) a negace (NOT). Systém splňuje axiomy komutativity, distributivity, komplementarity a neutrality ($X + 0 = X$, $X \cdot 1 = 1$). Z těchto axiomů vyplývají ostatní vyjmenované vlastnosti.

Příklad: Exkluzivní součet - XOR

Realizace funkce: pravdivostní tabulka, booleovská rovnice, obvod z obecných hradel, obvod z hradel NAND



Převod mezi intuitivní rovnicí a NAND formou: opakovaně využity De Morganovy zákony.



1.1.3 Příklady základních logických obvodů

Základní obvody

kombinační – výstup je závislý na aktuálním vstupu :

- ▶ logické funkce - booleovská logika
- ▶ aritmetické funkce - pulsčítačka, sčítačka
- ▶ řídicí funkce - multiplexor, dekódér

sekvenční – výstup je závislý na aktuálním vstupu a na celé historii vstupu od iniciálního resetu:

- ▶ klopný obvod
- ▶ čítač
- ▶ stavový automat



Logické obvody třídíme do dvou základních kategorií:

Tzv. *kombinační obvody* jsou takové logické obvody, jejichž výstup je závislý na aktuálním vstupu. Tzn. hodnota výstupů je kombinací vstupních hodnot.

Tzv. *sekvenční obvody* využívají kromě kombinační logiky navíc paměť (obr. 1.1.3), která slouží

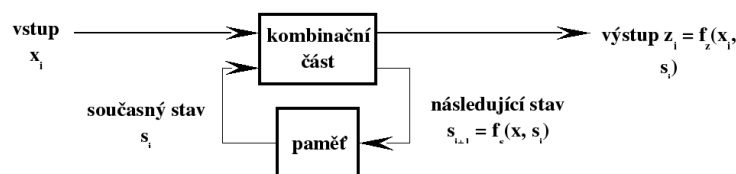
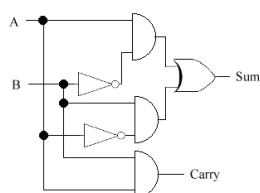


Figure 1.1: Principiální schéma sekvenčních obvodů

k zaznamenání aktuálního stavu. Dále pak koncept diskrétního času, který je tvořen posloupností diskrétních časových okamžiků a zpravidla realizovaný tzv. hodinovým signálem, kdy každý tik – typicky např. každá náběžná hrana, tj. změna $\log. 0 \rightarrow \log. 1$ – reprezentuje jeden tento časový okamžik.

Ideální sekvenční obvod mění svůj stav v právě jednou při každém hodinovém tiky. Stav se mění okamžitě a na základě kombinace aktuálního stavu obvodu a hodnoty vstupů. V praxi tato změna neprobíhá okamžitě, ale dochází k tzv. propagačnímu zpoždění (propagation delay). To mj. ovlivňuje maximální rychlost hodinového signálu, při které obvod korektně funguje a zároveň umožňuje vznik tzv. hazardních (nedefinovaných) stavů obvodu.

Kombinační obvody - pulsčítačka



Vstup:

- ▶ jednobitové hodnoty A, B

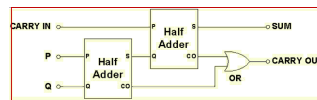
Výstup:

- ▶ součet (Sum):
 $S = \overline{A}B + A\overline{B}$
- ▶ Přenos do vyššího řádu:
 $C = AB$

Kombinační obvody - úplná sčítačka

Vstup:

- ▶ hodnoty A, B
- ▶ přenos z nižšího řádu C_{in}



Výstup:

- ▶ součet (Sum):

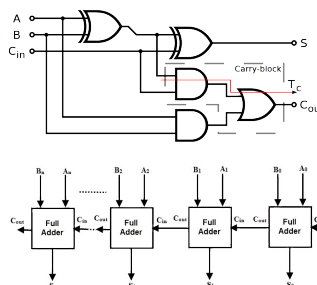
$$S_h = \overline{A}B + A\overline{B}$$

$$S = \overline{S_h}C_{in} + S_h\overline{C_{in}}$$

- ▶ Přenos do vyššího řádu:

$$C_{out} = AB + S_h C_{in}$$

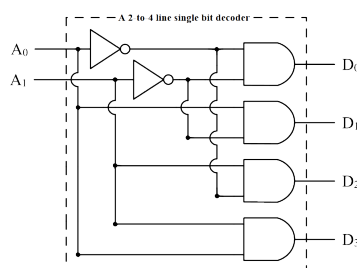
Bloky zřetězeny pro součet vícebitových čísel:



Půlsčítačka (half adder) a *sčítačka* (full adder) jsou typickými příklady *aritmetických kombinačních* obvodů. Sčítání v binární soustavě funguje velmi podobně jako lidem blízké sčítání v desítkové soustavě - sečteme hodnoty ve stejném řádu, pokud je hodnota větší než deset, tak si započítáme jedničku (či více při sčítání více čísel) a tu pak přičteme v o jedna vyšším řádu. Tento mechanismus nazýváme přenos do vyššího řádu (carry). Úplná sčítačka je obvod, který sčítá dvě jedno bitová čísla, přičítá k nim přenos z nižšího řádu a případně tvoří přenos do vyššího řádu. Zřetěžením n sčítaček lze sčítat dva n -bitové operandy. Půlsčítačka je zjednodušenou verzí, která nepřičítá přenos z nižšího řádu. Všimněte si, že úplná sčítačka lze realizovat pomocí dvou půlsčítaček.

Dalšími příklady aritmetických obvodů jsou komparátory (obvody porovnávající dvě čísla), násobičky nebo složitější obvody typu multiply-and-accumulate (MAC).

Řídicí obvody – dekodér



Truth Table

A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Minterm Equations

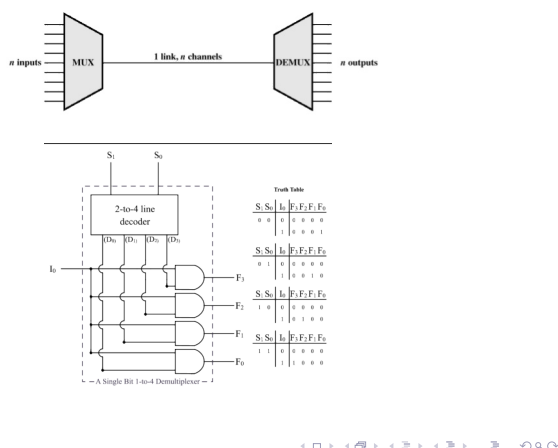
$$\begin{aligned} D_0 &= \overline{A_1} \cdot \overline{A_0} \\ D_1 &= \overline{A_1} \cdot A_0 \\ D_2 &= A_1 \cdot \overline{A_0} \\ D_3 &= A_1 \cdot A_0 \end{aligned}$$

Dekodér je kombinační logický obvod, který na základě kombinační tabulky z kombinace vstupních dat vytváří na výstupu kód jiný. Dekoderů existuje několik druhů. (Např.: binární, BCD)

Jeho protějškem je enkodér, tj. obvod, který na základě log. 1 na jednom až z 2^n vstupů generuje n -bitové kódové slovo.

Variantou je prioritní enkodér, který definuje prioritu vstupů v případě výskytu několika log. 1 na vstupech. Výhodou je definované chování v případě takovéto kolize, nevýhodou je složitější obvodová logika.

Řídicí obvody – multiplexing



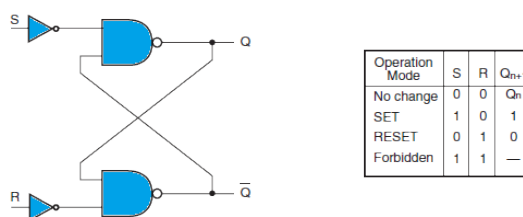
Multiplexor je kombinační obvod, fungující na principu přepínače. Podle nastavení řídicích signálů nastaví na výstup jeden z n vstupních signálů.

Demultiplexor je další příklad kombinačního obvodu, fungující na principu přepínače. Jedná se o opačný princip jako u multiplexeru. Podle nastavení řídicích signálů nastaví na jeden z n výstupních signálů signál vstupní.

Pojem multiplexor/demultiplexor, též mux/demux lze najít i různě jinde v oblasti digitální techniky. Pojem multiplexing se typicky označuje střídání/přepínání více signálů, procházejících jedním médiem. Tzv. časový multiplexing (TDM, Time-division multiplex) obnáší střídání v nějakým způsobem definovaných časových intervalech (oknech) – příkladem je bezdrátový přenos v mobilních sítích GSM, ISDN nebo optické přenosové systémy SONET/SDH. Tzv. frekvenční multiplexing (FDM, frequency-division multiplex) naopak značí souběžný přenos nezávislých signálů, frekvenčně separovaných, po jednom médiu – např. radiové či televizní vysílání nebo WDM technologie (wave-division multiplex) pro přenos více optických frekvencí po jednom optickém vlákne.

Klopné obvody - SR Latch

Díky zpětné vazbě lze vytvořit paměťový obvod:



Vnitřní stav obvodu se změní přivedením 1 na vstup R/S a tento stav "přetrvá" do dalšího podnětu.

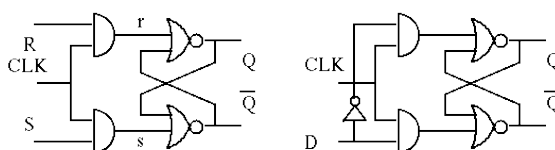
Navigation icons: back, forward, search, etc.

Další důležitou skupinou logických obvodů jsou tzv. *klopné obvody*, jsou realizovány pomocí sady hradel. Klopné obvody jsou elementární realizací paměťové buňky, tj. nabývají jedné z logických hodnot, kterou jsou schopné si "zapamatovat". Jsou triviálním příkladem sekvenčních obvodů.

Klopných obvodů je několik typů a řada realizací. Zde je vyobrazen klopný obvod typu SR (set-reset), realizovaný pomocí hradel NAND ve variantě "latch", tj. stav se mění v okamžiku změny vstupu.

Klopné obvody - odvozené typy

U klopného obvodu typu RS (zde z hradel NOR) lze jednoduchou úpravou dosáhnout synchronizace na hodinový signál:



Klopný obvod typu D navíc odvozuje svůj stav od jednoho vstupu a eliminuje nedefinovaný stav vstupu.

Navigation icons: back, forward, search, etc.

Zde je uveden klopný obvod typu SR (set-reset), realizovaný pomocí hradel NOR (rozdíl v polaritě vstupů), ve variantě "flip-flop" který využívá další hradla a synchronizační hodinový vstup. Změna stavu nastává na základě vstupů pouze v aktivní době hodinového vstupu.

Druhý je uveden klopný obvod typu D (derivační), který svůj stav odvozuje přímo od hodnoty na vstupu. D je asi nejhojněji využívaný klopný obvod.

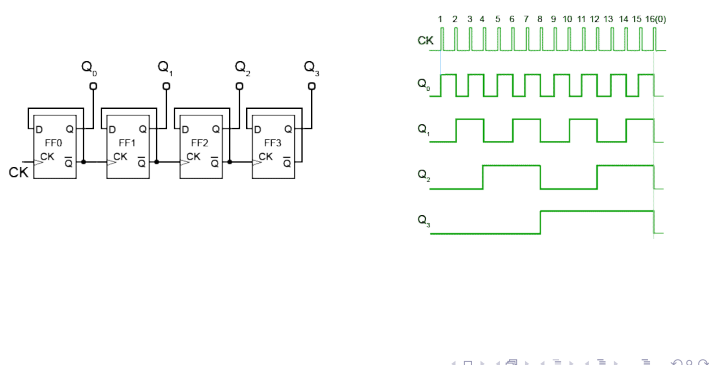
Dalšími variantami JK (obdoba RS s emilovaným nedefinovaným stavem) nebo T (toggle).

Klopné obvody dále bývají vybaveny vstupy "set" (přechod do stavu log. 1) a nebo "clear" (log. 0); tyto vstupy, též označované jako resetovací, slouží k uvedení k.o. do definovaného stavu, typicky právě po resetu. Tyto vstupy mohou být realizovány jako synchronní nebo asynchronní.

Poznámka: překladem termínu "klopný obvod" je "flip-flop", leč tento termín zahrnuje jak variantu "flip-flop" a "latch", tak i některé další.

Čítač

Zřetěžením několika klopných obvodů typu D vznikne čítač:



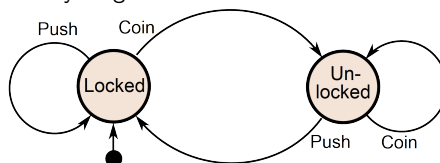
Příkladem základního a hojně využívaného sekvenčního obvodu je *čítač* (counter). Počítá nebo odpočítává kolikrát proběhla určitá událost nebo proces. Například lze čítač použít pro čítání počtu náběžných hran ve zkoumaném signálu.

1.1.4 Stavové automaty

Stavový automat – Finite State Machine

- FSM klíčový koncept pro embedded, nezávisle na platformě
- konečná množina stavů, vstup a výstup, iniciální stav
- přechodová funkce: změna stavu na základě vstupu
- výstupní funkce: změna výstupu na základě stavu, případně vstupu

Příklad: stavový diagram "turniket na mince":



Stavové automaty jsou klíčovým konceptem pro embedded. Jejich fyzická realizace se liší v závislosti na využití technologii – jinak je budete využívat pro programování mikroprocesoru, jinak pro návrhu HW obvodu nebo FPGA aplikaci.

Základní koncepty jsou však stejné: stavový automat je virtuální konstrukce, která obsahuje definovanou množinu stavů, vstupy a výstupy, přechodovou funkci (též *δfunkci*) a v našem kontextu i výstupní funkci.

Je nutné (resp. z praktického hlediska velmi výhodné), aby automat měl definovaný tzv. iniciální (výchozí) stav, tj. stav, do kterého systém přechází po resetu či po zapnutí.

Na uvedeném příkladě je stavový automat definován takto:

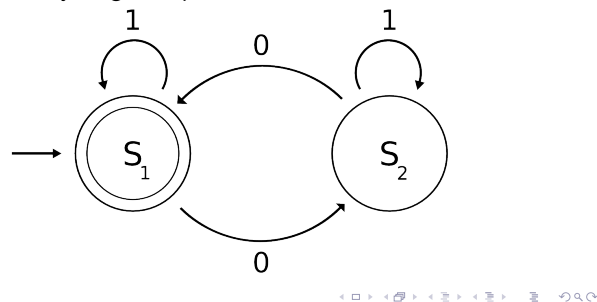
- dva stavy: odemčeno a zamčeno
- iniciální stavem je stav zamčeno
- dva vstupy: vhození mince a stisknutí tlačítka
- jeden výstup: průchozí turniket
- přechodová funkce lze slovně popsat takto:
 - zamčeno + mince → odemčeno
 - odemčeno + tlačítko → zamčeno
 - zamčeno + tlačítko → zamčeno
 - odemčeno + mince → odemčeno
- výstupní funkce pak lze popsat takto:
 - odemčeno + tlačítko → povol průchod turniketem
 - jinak nic

Poznámka: stavové automaty (též konečné automaty) jsou hojně využívaným formalismem v oblasti teorie formálních jazyků. Reprezentace je zde drobně rozdílná: množina vstupů je definována jako tzv. abeceda, čili množina povolených vstupních znaků. Naopak výstupní funkce je definována množinou tzv. akceptujících stavů. Výstup konečného automatu je tedy dvoustavový: pokud je automat buď v jednom z akceptujících stavů nebo nikoliv. Úvahu nad rozdílnou interpretací zanecháváme laskavému čtenáři k úvaze. Nicméně, tento typ automatů lze principiálně využít v oblasti návrhu digitálních systémů, jak je vidno z následujícího příkladu:

Stavový automat typu "Acceptor"

- ▶ rozpoznává vstupní posloupnost
- ▶ neprázdná množina koncových stavů
- ▶ vystupní funkce: akceptace/zamítnutí

Příklad: stavový diagram "parita":



Stejně jako ve zmíněné oblasti formálních jazyků lze využít stavový automat pro rozpoznání vstupní posloupnosti – příkladem může být rozpoznání hesla, správné posloupnosti stisknutých tlačítek, nebo výpočet parity, kterýžto automat je uveden na slajdu.

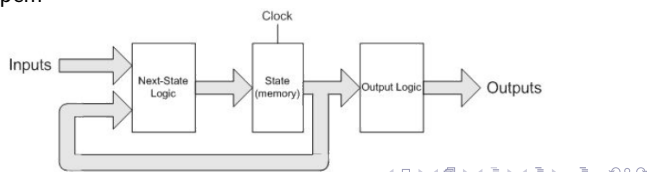
Stavový automat akceptuje, pokud je počet nul na vstupu sudý a je definován takto:

- dva stavy: S1, S2
- iniciální stav: S1 (označeno šipkou)
- vstupní abeceda: 0, 1
- akceptující stav: S1 (označeno dvojitou kružnicí)
- přechodová funkce lze slovně popsat takto:
 - S1, 1 \rightarrow S1
 - S1, 0 \rightarrow S2
 - S2, 1 \rightarrow S2
 - S2, 0 \rightarrow S1

Stavový automat typu "Transducer"

- ▶ generuje výstup na základě změn stavu
- ▶ dva základní koncepty: Moore, Mealy
- ▶ oba využívají tři základní části:
 - ▶ *current state*: realizuje aktuální stav (sekvenční)
 - ▶ *next state*: vyhodnocuje příští stav (kombinační) – realizuje přechodovou funkci
 - ▶ *output logic*: vyhodnocuje výstup (kombinační) – realizuje výstupní funkci

Moore automat: přísně synchronní, výstup asociován se vstupem



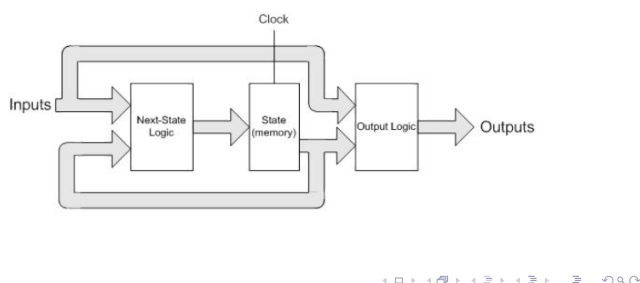
V oblasti návrhu digitálních systémů využíváme dva základní modely stavového automatu: Mealy automat a Moore automat.

Oba dva modely zveme "*transducer type*" (pro nedostatek vhodného českého překladu) a oba se skládají ze tří částí: *current state*, *next state* a *output logic*.

Jak bylo zmíněno dříve, sekvenční obvody obsahují paměť, jejíž obsah definuje aktuální stav obvodu. To stejné platí pro stavový automat, který k tomu využívá část *current state*. Sekvenční obvody jsou synchronizovány nějakým hodinovým signálem (synchronní automat) nebo změnou vstupní hodnoty (asynchronní automat). Při každém tiky či změně stavu pak automat přechází do nového stavu, který je vyhodnocen kombinační částí *next state*. Část *output logic* pak realizuje výstupní funkci automatu.

Mealy automat

- ▶ výstup na základě aktuálního stavu a vstupu vstupů, tj. výstup asociován s přechodem
- ▶ rychlejší díky asynchronní reakci, riziko hazardních stavů
- ▶ menší množství stavů než Moore



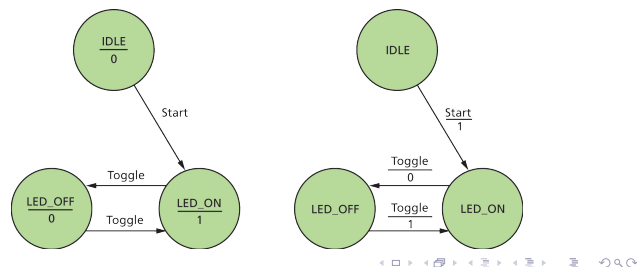
Rozdíly mezi Moore a Mealy automaty se liší pouze v jednom momentu: Mealy automat vyhodnocuje výstup na základě aktuálního stavu a vstupů, zatímco Moore automat pouze na základě

aktuálního stavu. To tedy znamená:

- u Moore automatu je výstup svázaný se vstupem, výstupní funkci lze psát jako $S1 \rightarrow S2$, tj. je přísně synchronní;
- u Mealy automatu je výstup svázaný s *přechodem*, tj. píšeme $S1 \times X \rightarrow S2$; stačí méně stavů, umožňuje asynchronní realizaci, tj. umožní rychlejší odezvu na vstup.

Příklad FSM: LED lampička

- ▶ vstup: tlačítko *toggle*
- ▶ výstup: LED
- ▶ množina stavů: ON, OFF
- ▶ iniciální stav: ON
- ▶ přechodová/výstupní funkce viz diagram:



Na příkladu jednoduché lampičky vidíme příklad triviálního automatu realizovaného oběma způsoby:

Reference

- Wikipedia, hesla Logický obvod, Logický člen, Klopný obvod, Multiplexor, Demultiplexor, Dekodér, Čítač, Boolean Algebra, Finite State Machine

Kapitola 2

Mikrokontrolery a periferie

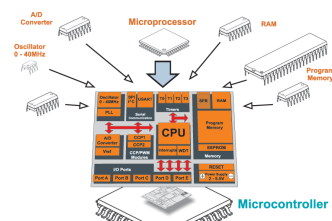
Tato kapitola stručně přibližuje architekturu mikrokontrolerů a výběr typických periférií. V místech, kde jsou některé konkrétní detaily závislé na konkrétní řadě, využíváme jako ilustrační model procesory řady PIC18 (Microchip) zejména z důvodu toho, že s nimi budeme pracovat dále.

Navzdory tomu, že zanedlouho budeme vymezovat rozdíly i společné rysy mikroprocesorů a mikrokontrolerů, označujeme mikrokontroléry jako *procesory*.

2.1 Mikrokontrolery

Jednočipový počítač alias mikrokontroler, MCU, uC

- ▶ jádrem většiny embedded systémů, zpravidla monolitický integrovaný obvod
- ▶ jádro procesoru + paměti pro data a program + obsahuje podpůrné obvody, umožňující samostatnou funkčnost + fixní množství fyzických vstupů a výstupů + periferie



Typické užití: řízení a regulace, komunikační rozhraní, uživatelské rozhraní – analogie zmenšeného stolního PC

Mikrokontrolery, mikrořadiče či jednočipové počítače (v ang. single-chip computers nebo microcontrollers, zkráceně též MCU nebo uC) jsou v principu “jednoduché” počítače, které jsou integrovány společně se sadou základních komponent. Tyto základní komponenty běžně zahrnují paměť pro text programu a pro data, vstupně-výstupní obvody, oscilátor, řadič přerušování a další. Pokročilejší kontrolery pak mohou disponovat dalšími obvody: A/D a D/A převodníky, napěťové komparátory, řadiče sériových sběrnic (UART, IIC, SPI) či řadiče externích periférií (LCD display, PWM).

Pro přiblížení lze mikrokontroler přirovnat ke zmenšené verzi stolního PC. Základní komponenty – procesor, RAM, pevný disk a podpůrné obvody základní desce – představují zmíněné řadič instrukcí, paměť pro data, paměť pro text programu a ovládání fyzického rozhraní. K němu lze připojit externí paměti (stejně jako k PC externí disk), komunikační řadiče (u PC porty COM, USB, Ethernet). Dále lze připojit, a zpravidla je to žádoucí, prvky uživatelského rozhraní: u PC monitor, klávesnici a myš, u mikrokontrolerů např. mechanická tlačítka, LED diody, displeje.

Mikrokontroléry jsou typicky používány pro řadu aplikací s nároky na nízké náklady, spotřebu energie a prostor. Obecně jednodušší architektura mikrokontrolerů dále umožňuje lépe zpracovávat úlohy, které nevyžadují vysoký výpočetní výkon, ale jsou citlivé na přesné časování (např. v oblasti real-time systémů). Speciální podmožinou jsou takzvané číslicové signální procesory (Digital Signal Processor, DSP), které jsou uzpůsobeny pro rychlé zpracování signálů.

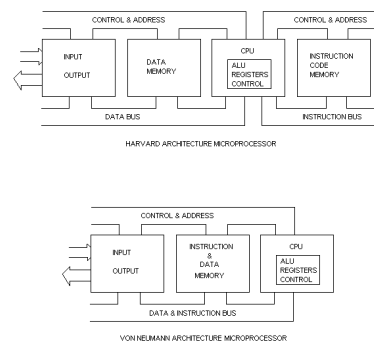
V obecné rovině rozlišujeme mezi mikrokontrolérem a mikroprocesorem v několika ohledech. Mikroprocesor je samostatná procesní jednotka, pro svůj běh vyžaduje další podpůrné obvody. Z toho plyne mimo jiné modularita procesorových systémů, kdy například typ a parametry pamětí či jiných periférií lze volit a zaměňovat v závislosti na cílové aplikaci. Obecně procesorové systémy také poskytují větší výpočetní výkon, jsou univerzálnější a lépe rozšiřitelné. Oproti tomu mikrokontrolér typicky obsahuje většinu komponent integrovanou a proto je schopen samostatného provozu s minimem externích obvodů. V důsledku toho ale nelze paměť a ostatní komponenty nahrazovat bez výměny celého kontroleru. Výhodou je pak právě nízká cena, spotřeba a fyzická velikost; dále pak jednoduchá architektura. Dříve byly procesorové systémy považovány za více univerzální systém a mikrokontrolery naopak silně dedikované pro specifickou aplikaci; dnes se s nástupem výkoných mikrokontrolerů a např. podpůrných grafických procesorů tento rozdíl stírá.

2.1.1 Klasifikace MCU

Běžné mikrokontroléry dnes obsahují něco mezi 16 a několika MB paměti pro data a mezi 64 a několika MB paměti pro kód. Běžný operační kmitočet je v řádu několika MHz, v aplikacích citlivých na spotřebu nebo výkon se často užívá kmitočet výrazně nižší (až 32 kHz) nebo vyšší (500 MHz+). Fyzicky existují kontrolery od miniaturních – s 6 vývody a objemem cca 1 mm³ – až po velké čipy s několika stovkami vývodů. Kromě těchto základních parametrů lze mikrokontroléry (a obecně i procesory) třídit podle několika fundamentálních hledisek: typ architektury, typ instrukční sady, základní adresová šířka datové paměti a fyzické platformy. Dalším tříděním, mimo rozsah tohoto textu, je pak taxonomie podle jednotlivých výrobců a jejich produktových řad.

Architektura MCU

Architektura procesoru



Von Neumann vs. Harvard – společná či oddělená pamět/sběrnice pro data a program

Základní charakteristikou každého mikrokontroleru je jeho architektura. Historicky rozeznáváme dvě základní schémata: *Von Neumannova architektura* a *Harvardská architektura*. V obou případech schéma obsahuje procesní jednotku, vstupně/výstupní zařízení a paměť pro data (datovou paměť) a pro text programu (instrukční paměť). Von Neumannova architektura je typická tím, že využívá jednu paměť zároveň pro data i text programu. Výhodou je jednodušší řešení, které vyžaduje pouze jednu sběrnici a jednu paměť. Harvardská architektura naopak typicky využívá oddělenou instrukční a datovou paměť, včetně nezávislých sběrnic. Toto uspořádání umožňuje například použití různých typů paměti – např. rychlejší pro data a levnější pro kód nebo dvě paměti s různou šířkou slova. Dále oddělené paměti v principu umožňují souběžný a tudíž rychlejší přístup k datům a programu.

Roziřením Harvardské architektury je tzv. Modifikovaná harvardská architektura (Modified-Harvard architecture), která se vyznačuje možností přístupu k instrukční paměti podobným způsobem jako k datové paměti (možností je několik, viz dále). Ve skutečnosti drtivá většina moderních procesorů s Harvardskou архитектурou jsou implementací Modifikované harvardské architektury. Výhodou tohoto přístupu je možnost dynamické změny kódu bez nutného externího přístupu, jak je tomu v případě klasického předchůdce.

Dalším významným rozšířením je tzv. memory-mapped I/O (vstup/výstup mapovaný do paměti). Tato technika využívá jeden společný adresní prostor (a typicky společnou adresovou a datovou sběrnici) pro datovou paměť i pro ostatní vstupně/výstupní periférie. Každý periferní obvod tak má vyhrazen blok adresového prostoru a ve výsledku jsou všechny moduly jednotně programově přístupné.

2.1.2 Instrukční sada

Rozeznáváme dvě základní kategorie: tzv. komplexní instrukční sadu (Complete Instruction Set Computer, CISC) a redukovanou instrukční sadu (Reduced Instruction Set Computer, RISC). Vlastnosti a odlišnosti jsou zhruba následující:

Počítače s komplexní instrukční sadou obecně disponují velkou množinou instrukcí pro řadu základních

i složitých operací, což poskytuje větší možnosti a pohodlí z programátorského hlediska a umožňuje šetřit kapacitu instrukční paměti. Z architektonického hlediska pak procesor jednotlivé instrukce realizuje jako posloupnost takzvaných mikroinstrukcí, díky čemuž zpracování jedné instrukce trvá několik taktů procesoru a navíc se různé instrukce zpracovávají různě dlouho. Příkladem architektur je např. Intel 80x86, Zilog Z80, 8051.

Počítače s redukovanou instrukční sadou naopak disponují omezenou sadou instrukcí, složitější úlohy jsou realizovány jako sekvence jednodušších instrukcí. Tím, že jsou instrukce obecně jednodušší, jejich zpracování je rychlejší a typicky tvá pevně definovaný počet cyklů. Výhodou je možnost zřetěženého zpracování instrukcí (pipeliningu), kdy řadič může během zpracování jedné instrukce připravovat zpracování dalších, což může představovat významné urychlení. Příkladem architektur je např. ARM, Sparc, Atmel AVR, PowerPc, PIC.

V dnešní době je většinou využívána nějaká amorfní kombinace.

Ostatní parametry

Základní adresová šířka datové paměti udává velikost přímo adresovatelného adresního prostoru (např. osm bitů umožní adresovat 256 míst v paměti, 16-bitů pak 65tisíc ap.). Odvozuje se od ní zpravidla i velikost elementární jednotky dat, kterou lze najednou zpracovávat, bitová šířka registrů apod. Typická je 8-bitová architektura pro nejjednodušší zařízení, 16-bitová pro některé pokročilejší platformy a pro signálové procesory, 32-bitová pro průmyslové mikrokontrolery. Pro srovnání, moderní PC dnes využívají 64-bitovou architekturu (Intel i64, AMD x64), ale ještě nedávno byla nejběžnější architektura 32-bitová (IBM x86).

Fyzické platforma zahrnuje několik dílčích aspektů. Fyzicky lze kontroler realizovat jako samostatný integrovaný obvod, dále může být součástí složitějšího logického systému (Xilinx Zynq) nebo může být realizován formou soft-core jednotek pro programovatelné struktury jako jsou hradlová pole (Altera NIOS2, Xilinx MicroBlaze). Dalším aspektem je technologie instrukční paměti, která může být jednorázová či opakovaně přepisovatelná (technologie FLASH).

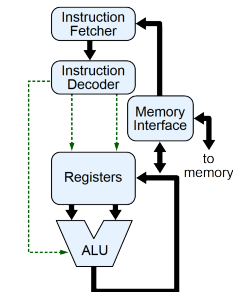
Reference

- Wikipedia, hesla Jednočipový počítač, Architektura počítače

2.1.3 Architektura a základní operace jádra procesoru

Jádro procesoru

- řadič instrukcí - sekvenční automat (IF + ID)
- registry - pracovní paměť
- aritmeticko-logická jednotka - vykonává operace
- řadič sběrnic(e) - umožňuje přístup k datové a programové paměti (memory interface)

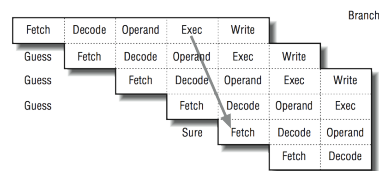


Navigation icons: back, forward, search, etc.

Instrukční cyklus

- Fetch - zpracovávaná instrukce načtena z paměti programu
- Decode - rozpoznán typ instrukce a operandy
- Operand - načteny operandy z datové paměti do registrů
- Execute - vykonána instrukce + zápis výsledku (Write)

Příklad: řetězení zpracování (pipelining)



Navigation icons: back, forward, search, etc.

Mechanismus zpracování instrukcí nejprve vyzvedne instrukci z instrukční paměti. Následně se vyhodnotí jednotlivé operandy a ty předají k dalšímu zpracování. Řadič následně rozloží instrukci podle jejího operačního kódu na mikroinstrukce. Aritmeticko logická jednotka na základě řídicích signálů řadiče provede určené operace vyhodnocení. Řadič provede vyzvednutí výsledků z výstupů vybraných funkčních jednotek a uloží je na místa určené operačním kódem instrukce datové paměti. Dale se ještě vyhodnotí podmínky související s prováděním instrukce a nastaví se adresa na další instrukci, která se má provést.

2.1.4 Podpůrné obvody

Podpůrné obvody

- ▶ datová paměť – energeticky závislá
- ▶ paměť programu (Flash)
- ▶ zásobník – HW nebo SW
- ▶ hodinový signál – reprezentuje diskrétní čas
- ▶ resetovací obvody
- ▶ **speciální funkční registry** – v adresovém prostoru datové paměti, slouží pro řízení periferií
- ▶ konfigurační pojistky – konfigurace obvodů, nutných pro start MCU
- ▶ sběrnice

Hodinové obvody slouží ke generování přesných hodinových signálů nutných ke správné činnosti MCU. Na základě této generované frekvence se provádí jednotlivé úkoly v MCU. Existuje několik typů generátorů hodinového signálu odlišujících se konstrukcí, spotřebou, teplotní závislostí nebo velikostí.

Resetovací obvody jsou podpůrné obvody sloužící k uvedení zařízení či MCU do výchozího stavu. Například po neočekávaných stavech jako jsou poklesy napětí se zpravidla objeví výpočetní chyby v MCU, kterým lze pomocí resetovacích obvodů (hlídajícím pokles napětí) zabránit.

SFR jsou speciální funkční registry (Special Function Register) umístěné v paměti sloužící různým účelům. Lze je rozdělit na registry datové a řídicí. Pomocí nich lze například nastavovat jednotlivé periferie MCU. Tato technika, zvaná *periferie mapované v paměti* jsou snahou o zjednodušení přístupu k jednotlivým vstupním a výstupním zařízením u MCU. Na adresovatelném místě lze nalézt například periferii sloužící k sériové komunikaci UART. Pouhým čtením z přesně definované adresy lze například načítat data vstupující do MCU.

Pojistky - *fuses* slouží u MCU k nastavování základních prvků nutných pro jeho spuštění a činnost. Pojistky patří mezi nejdůležitější konfigurační prvky a jsou různé pro různé typy MCU. Obvykle se nastavuje jedno nebo několik konfiguračních slov.

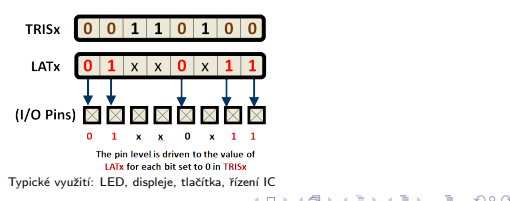
Sběrnice je souhrn jednotlivých signálů aktivních v různých časových okamžicích s různým významem, které mívají různé zdroje. V architektuře MCU se typicky vyskytuje alespoň jedna *systémová sběrnice*, které umožňují komunikaci mezi jádrem procesoru a datovou pamětí, pamětí programu a případně vstupně/výstupními periferiemi.

2.2 Základní periferní obvody

2.2.1 GPIO: obecný vstup a výstup

Paralelní I/O (General-Purpose I/O)

- řízení fyzického výstupu (pinu): jednotlivě či sdružené n-bitové slovo
- výstup: GND / Vcc – logická 0 / 1
- třístavová logika – přepínání vstupu a výstupu
- volitelně: pull-up, open-drain
- SFR TRISx pro směr
- SFR PORTx (ev. LATx) pro čtení/zápis hodnoty



Základním vstupně výstupním systémem procesorů PIC jsou tzv. obecné vstupy a výstupy (General Purpose Input/Output). Tyto jsou připojeny na fyzické vývody procesoru (tzv. piny) a tak umožňují řídit nebo číst elektrické signály (typicky reprezentující logickou hodnotu) propojené s okolními obvody.

Na procesorech PIC jsou vývody organizovány do osmibitových skupin, tzv. portů¹, označených typicky písmeny A-E. Výhodou této organizace je možnost adresovat celou skupinu na jednu např. při čtení nebo zápisu. Celý port se označuje PORTx (například PORTA), jednotlivé piny pak nap Rxn (například RA1);

Poznámka: Jednotlivé modely procesorů v rámci produktové řady konkrétního výrobce dodržují vzájemnou kompatibilitu i napříč variantami s různým počtem pinů. Proto se stává, že jednotlivé porty nejsou obsazeny kompletně. Například z portu E mohou být implementovány pouze signály RE4-RE7.

Obecný vstup/výstup obecně využívá třístavovou logiku. Většina pinů může sloužit jako vstupní nebo výstupní. Režim vstupu/výstupu se nastavuje pomocí registru TRISx, resp. bitu TRISxn (např. TRISB pro celý port B, resp. TRISB0 pro pin RB0). Třemi stavy se rozumí: výstup logická 0, výstup logická 1 a vstup.

Kromě nejjednodušších procesorů je I/O pin řízen třemi položkami ve speciálních funkčních registrech (názvy zde typické pro PIC):

- zmíněný TRISxn – pro nastavení režimu vstupu/výstupu

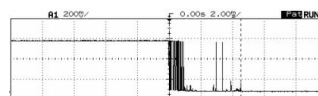
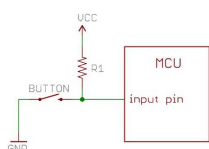
¹Konkrétně na řadách PIC10, PIC12, PIC16 a PIC18, což jsou procesory s osmibitovou architekturou. Na 16-bitových a 32-bitových procesorech jsou často porty právě 16-bitové či 32-bitové.

- **PORTxn** – vstupní registr: tato hodnota je čtená jako vstupní hodnota pokud je pin v režimu vstupu
- **LATxn** – výstupní latch: tato hodnota je hodnotou výstupu, pokud je pin v režimu výstupu

Typickým příkladem periferie ovládané pomocí obecného vstupu a výstupu jsou LED kontrolky nebo segmenty LED displeje. Dále lze řídit rozhraní LCD displeje nebo generovat řídicí signály pro další integrované obvody v rámci jednoho systému.

Paralelní I/O – tlačítka

- ▶ v rozepnutém stavu náhodná hodnota na vstupu
- ▶ pull-up odpor zavede definovanou hodnotu (log. 1)
- ▶ invertovaná sémantika



- ▶ přechodové jevy při stisknutí tlačítka
- ▶ nutno číst vstup opakovaně
- ▶ rozestup typ. ≈ 10 ms

Typickým příkladem vstupní periferie, snímané pomocí obecného vstupu a výstupu jsou tlačítka nebo maticové klávesnice. S těmito se pojí potřeba implementace pull-up odporů – ať již v rámci procesoru nebo mimo něj (viz dále, WPU) – a potřeba vstupní filtrace, tzv. debouncingu.

Pokročilé možnosti GPIO

Pin multiplexing: Vzhledem k omezenému množství pinů procesoru je často obecný vstup/výstup sdílí se vstupy a výstupy dalších periférií. Za zmínku stojí analogové funkce, kdy pin může sloužit např. jako analogový vstup nebo napěťová reference: obecnou vlastností procesorů PIC je to, že analogová funkce je implicitní a proto je nutné ji explicitně vypnout, pokud chceme používat obecný vstup/výstup (typicky registr ANSELx, kde x zde závisí na konkrétním procesoru).

Weak pull-ups (WPU): Pro aplikace, které vyžadují obsluhu např. mechanických spínačů, jsou k dispozici vestavěné pull-up rezistory. Zpravidla se nastavují registrem WPU_n pro jednotlivé piny, dále je pak nutno funkci povolit globálně.

Interrupt on change: Většina procesorů umožňuje asynchronně detekovat změnu hodnoty na vstupním pinu a vygenerovat přerušení. Tento podsystém se může na různých rodinách procesorů jmenovat různě (např. na 16bitových procesorech PIC se nazývá Change notification).

Open-drain výstupy (v. s otevřeným kolektorem): v rozpojeném stavu negeneruje žádný potenciál, v sepnutém stavu propojení k zemi – analogie k mechanickému spínači.

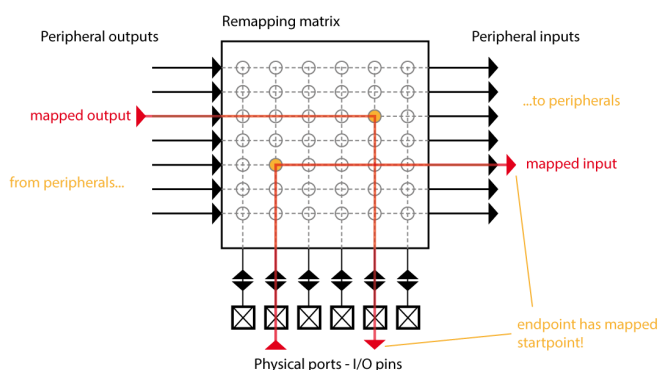


Figure 2.1: Mechanismus přemapování

Přemapování vstupu a výstupu: Obecnou snahou při návrhu jednočipového počítače je poskytnout co největší množství integrovaných periférií. Z logiky věci většina periférií využívá množinu vývodů kontroléru, kterých je ovšem pevně omezený počet. Bežným přístupem je sdílení jednoho vývodu několika periferními obvody, což prakticky výrazně ztěžuje použití více takových periférií v jedné aplikaci. Praktické řešení takovéto situace běžně vede k výrazným změnám návrhu, například změna typu kontroléru nebo doplnění externích periférií. Pokročilejší metodou, jak omezit takovéto kolize při využití vývodů je právě přemapování vývodů.

Přemapování vývodů (Pin Remapping, Peripheral Pin Select aka PPS) je metoda, která umožňuje programově přiřadit jednotlivé vstupní a výstupní funkce periferních obvodů kontroléru jednotlivým fyzickým vývodům (pinům) procesoru. Základní princip je ilustrován na obrázku 2.2.1.

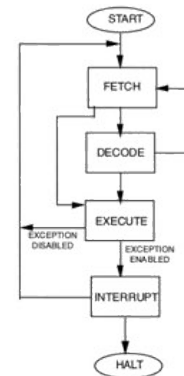
Metoda pracuje s množinou tzv. vstupních a výstupních funkcí, které představují jednotlivé signálové vstupy a výstupy dotčených periférií, dále s pevně danou množinou vývodů kontroléru. Samotné přepínání si lze zjednodušeně představit jako matici, ve které jsou propojeny právě jednotlivé funkce s jednotlivými výstupy.

Protože správné nastavení přemapování je kritické pro korektní běh aplikace, poskytují kontrolery zabezpečení tohoto nastavení proti náhodnému či chybnému přenastavení. Jedna z technik je zamykání nastavení pomocí sekvece konstant, které je nutno zapsat do specifického registru před samotnou změnou nastavení a pro potvrzení po ní. Dále může být tento zamykací registr zabezpečen příznakovým bitem v jiném registru, který umožňuje zakázat a povolit zápis, nebo pomocí pojistky, která umožní pouze jednorázovou změnu nastavení přemapování (mezi dvěma resety).

2.2.2 Přerušení

Přerušení (interrupts)

- ▶ mechanismus pro zpracování asynchronních událostí
- ▶ řadič přerušení – vyhodnotí zdroj, nastaví příznak
- ▶ tabulka vektorů přerušení (IVT)
- ▶ procesor přeruší běh programu, uloží stav procesoru a provede obsluhu přerušení
- ▶ vymazání příznaku, návrat z přerušení
- ▶ atomické operace a maskování přerušení
- ▶ priority přerušení
- ▶ souběh přerušení – odložení, vynechání
- ▶ latence přerušení



Přerušení (interrupt) je technika, která umožňuje procesoru programovou obsluhu asynchronních událostí. Procesor je doplněn tzv. řadičem přerušení, jehož vstupem jsou právě asynchronní signály jednotlivých zdrojů přerušení, tzn. externích obvodů, které signalizují zmíněné asynchronní události. V případě příchodu přerušení (tj. pokud nastane takováto událost) se nejprve zpracuje poslední rozpracovaná instrukce. Následně se na zásobník uloží adresa následující strojové instrukce a vyvolá se obsluha přerušení. Činnost procesoru se po provedení obsluhy vrátí do původního stavu. Historicky koncept vznikl pro obsluhu periférií, které se tímto způsobem dožadují pozornosti procesoru/programu.

Dnes lze přerušení obecně rozdělit na hardwarové a softwarové. Obsluha pro hardwarové přerušení se zavolá pokud nastala nějaká událost u externích periférií. Například začala probíhat komunikace s MCU. Softwarové přerušení slouží především pro služby operačního systému atd. Toto přerušení je vyvoláno speciální instrukcí. Existují také vnitřní přerušení, které nejsou vyvolány externí událostí, ale například interní chybou ALU přímo v procesoru. Tento typ je také označován jako výjimky.

Vyvolání přerušení lze pro některé události zakázat (maskovat). Pokud budou některá přerušení maskována, nebude se při jejich výskytu volat obsluha přerušení. Existuje i skupina přerušení, která nelze zakázat (maskovat), tzv. non-maskable interrupts.

Přerušení lze v mnoha případech také nastavit prioritu. Pokud se například vykonává obsluha přerušení a nastane v průběhu další událost která není maskována, mohou nastat dva případy. V prvním se vyvolá obsluha přerušení s vyšší prioritou než aktuálně obsluhovaná a přerušení se vykoná. Ve druhém případě má nově vyvolaná událost přerušení stejnou nebo nižší prioritu než aktuálně obsluhovaná. V tomto případě se obsluha přerušení provede až po obslužení aktuálního požadavku.

Speciální typ přerušení (SW) je přerušení určené pro debugging MCU. Toto přerušení není dostupné u všech typů MCU a jeho vyvolání závisí i na použitém HW pro programování-ladění MCU.

Konfigurace a obsluha přerušení

Definice obsluhy přerušení – různé metody, platformově závislé:

- ▶ jedna funkce + multiplexing
- ▶ předdefinované názvy procedur
- ▶ plnění tabulky vektorů přerušení

SFR:

- ▶ **intIE** (interrupt enable) – povolí konkrétní přerušení
- ▶ **intIF** (interrupt flag) – indikuje příchod přerušení, příznak nutno vymazat v rámci obsluhy
- ▶ **GIE** (global interrupt enable) – povolení funkce řadiče přerušení
- ▶ **PEIE** (peripheral interrupt enable) – povolení funkce třídy přerušení

◀ ◻ ▶ ◀ ▢ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ 🔍 ↺

Zpravidla i ty nejjednodušší procesory jsou vybaveny řadičem přerušení. Dostupné zdroje přerušení se obecně řadí do dvou skupin: systémová přerušení a přerušení periférií (Peripheral Interrupts).

Systémová přerušení jsou například:

- Detekce poklesu napájení: přerušení je vyvoláno poté, co napětí dočasně poklesne pod určenou mez, program tak může zareagovat na potenciální chybové stavy (chyby paměti, přerušení operací)
- Ladicí přerušení: umožňuje přerušit běh procesoru za účelem ladění programu
- Výjimka při dělení nulou
- Časovače: přerušení je vyvoláno při přetečení čítače (tj. přechod $0xFF \leftarrow 0x00$) nebo při dosažení hodnoty v registru periody (PRx)

Následující periferie typicky mohou generovat přerušení:

- obecný vstup – přerušení je vyvoláno při změně stavu na vstupu
- A/D převodník – přerušení je vyvoláno po ukončení převodu
- UART (či jakýkoliv jiný komunikační modul) – přerušení je vyvoláno po asynchronním přijetí znaku nebo po ukončení odesílání

Klasifikace zdrojů přerušení se může lišit podle typu procesoru, typicky u používaného PIC16F4331 některé časovače generují systémové přerušení a některé přerušení periferie.

Úroveň priority přerušení bývá na procesorech PIC18 většinou jedna nebo dvě (low a high).

Povolení či zakázání přerušení (maskování) se zpravidla provádí jednak pro každý zdroj zvlášť (SFR xxIE, např. TMR1IE), dále pak existují možnosti obecného nastavení maskování: typicky je to globální povolení přerušení (GIE) a povolení přerušení periférií (PEIE).

Na procesoru PIC18F4431 je možno přepínat mezi jedno- a dvou-urovňovým systémem priorit (SFR IPEN). V závislosti na tomto nastavení pak bity GIE a PEIE fungují jako globální a periferní povolení (bez priorit) nebo jako povolení jedné či druhé úrovně (s prioritizací). Vyvolané přerušení vždy nastavuje příznak vyvolání přerušení (interrupt flag, SFR xxIF), který odpovídá konkrétní zdroj přerušení. Příznak vyvolání je potřeba v rámci programové obsluhy přerušení vynulovat, jinak je přerušení zamaskováno.

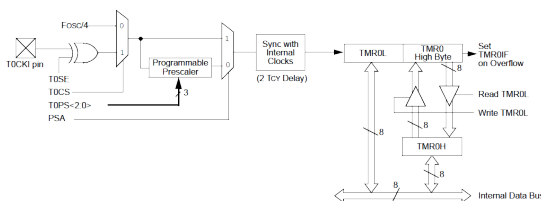
SFR na procesorech PIC pro obsluhu přerušení:

- PIE1, PIE2, PIE3 – peripheral interrupt enable: obsahuje bity povolení pro jednotlivé zdroje přerušení (např. TMR1IE, ADIE)
- PIR1, PIR2, PIR3 – peripheral interrupt flags: obsahuje příznaky vyvolání přerušení
- IPR1, IPR2, IPR3 – interrupt priority level: obsahuje nastavení priorit přerušení
- RCON – obsahuje bit pro povolení priorit přerušení (IPEN), dále sadu bitů, které po resetu procesoru udávají příčinu resetu (např. ztráta napájení, přetečení watchdogu, SW reset).
- INTCON – obsahuje příznaky obecného povolení přerušení (GIE, PEIE), dále nastavení xxIE a xxIF pro některá systémová přerušení
- INTCON2, INTCON3 – např. povolení vestavěných pull-upů, nastavení priorit některých systémových přerušení, nastavení typu přechodu při vnějším přerušení

2.2.3 Časovače a čítače

Časovač/Čítač

- binární čítač, definovaná šířka (8, 16, 32bit)
- zdroj: systémové hodiny, externí hodiny, externí signál
- (a-)synchronní čítání
- pre/post-scaler: čítání každé n -té události
- automatický reset čítače, generování přerušení



Čítač je obecně logický obvod, který zajišťuje ukládání informací o počtu proběhlých událostí, typicky se jedná vzestupná či sestupná hrana vstupního signálu. Vstupem čítačů, které tvoří periferní obvody mikrokontrolerů, je většinou buď periodický hodinový signál nebo neperiodický vnější signál. V prvním případě hovoříme o režimu časovače, neboť čítáním period o známé frekvenci lze v diskretních krocích

odměřovat čas. Protikladem je režim čítače, kdy v rámci definovaného časového kvanta čítáme počet výskytů. Za předpokladu, že je vstupní signál periodický, lze odměřovat jeho frekvenci/periodu.

Čítání obecně může probíhat buď v synchronním nebo asynchronním režimu. Synchronní čítání probíhá ve fázi s hodinovým cyklem zbytku procesoru, což například zajišťuje možnost bezpečně pracovat s čítacím registrem, protože nemůže dojít k situaci, kdy je zároveň vyčítán z procesoru a je do něj zapisováno samotným čítačem. Naopak asynchronní čítání běží nezávisle na taktu systémových hodin procesoru, což umožňuje např. čítat signály s vyšší frekvencí než je hodinový takt nebo čítat události i v případě, kdy je procesor zastaven v režimu spánku.

Typickým zdrojem signálu pro čítače jako periférie mikrokontrolerů je právě hodinový takt procesoru, respektive častěji instrukční takt. Dále může jít o externí oscilátor, externí vstup hodinového signálu nebo obecný signálový vstup. Takt procesoru je čítán synchronně, externí hodinové signály lze čítat asynchronně i synchronně (běžně díky synchronizačnímu obvodu)

Pro čítání asynchronních událostí po předem danou časovou periodu se využívá technika hradlování (gating nebo chip enable), kdy čítač je (de)aktivován hodnotou řídicího vstupního signálu nebo vnitřního registru. Aktivace čítače na definovanou dobu lze přesně vymezit časový usek, během kterého probíhá čítání. Nejjednodušší realizaci si lze představit jako logický součin hodinového vstupu a hradlovacího signálu.

V nejjednodušším (a nejčastějším) případě je při každém příchodu události (hrany) čítač inkrementován, a po dosažení maximální hodnoty pak přeteče. Některé mohou naopak hodnotu čítacího registru snižovat nebo dokonce volit mezi přičítáním/odečítáním. Dále mohou být čítače vybaveny tzv. registrem periody a komparátorem, který jeho hodnotu porovnává s hodnotou v čítacím registru. Shoda hodnot vyvolá signál pro vynulování čítače, efektivně tedy čítač nečítá v plném rozsahu, daném jeho bitovou šířkou, ale v rozsahu daném hodnotou registru periody.

S časovačem/čítačem lze pracovat dvěma základními způsoby: program může buď v pravidelných intervalech číst hodnotu časovače a na vyčtenou hodnotu nějak reagovat. Výrazně obvyklejším způsobem je využití mechanismu přerušení, které je vyvoláno při přetečení hodnoty registru (nebo dosažení hodnoty v registru periody).

Pro rozšíření časovací škály se často využívají děličky hodin (což je v principu opět čítač), zařazené před čítačem (prescaler) nebo za čítačem (postscaler). Nastavení děliček se udává jako poměr 1:n. U prescaleru to znamená, že pouze každá n-tá událost je čítána; u postscaleru pak to, že obsah čítače musí n-krát přetect než dojde k vyvolání přerušení.

Mezi další běžná a méně běžná rozšíření čítačů/časovačů patří:

- přepínání bitové šířky (např. mezi 8 a 16 bity)
- řetězení časovačů (za účelem zvětšení efektivní bitové šířky)
- single shot režim – čítač se dočítá k prvnímu přerušení a poté se zastaví
- filtrace vstupního signálu pro externí vstup s namodulovaným šumem

Obsluha Časovače/Čítače

Obvyklé SFR:

- ▶ TMRx / TMRxH + TMRxL: pracovní registr čítače
- ▶ TMRxIE / TMRxIF: povolení / příznak přerušení
- ▶ PRx (Period Register): hodnota se porovnává s pracovním registrem, v případě shody se pracovní registr nuluje a vyvolá se přerušení
- ▶ TMRxPS, TMRxCKPS, TMRxOUTPS: nastavení pre/post-scaleru, zpravidla bitový kód
- ▶ TMRxCS (clock source): nastavení pre/post-scaleru, zpravidla bitový kód
- ▶ TMRxON: povolení časovače

MCU obvykle obsahuje několik časovačů s různou funkcí; zároveň mohou tyto realizovat časovou základnu pro další periférie (např. PWM).

◀ ◻ ▶ ◀ ▢ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ 🔍 ↺

Typické SFR pro obsluhu časovače/čítače mohou vypadat například následovně:

Registr TMRn reprezentuje stav čítače. Čtením z registru lze získat aktuální stav čítače, zápisem do registru jej lze nastavit.

Registr PRn (Period Register) obsahuje hodnotu, vůči které se hodnota čítače při každé inkrementaci porovnává; v případě shody čítač generuje přerušení – není dostupné pro všechny časovače.

Registr TnCON: obsahují konfiguraci čítače, m.j. tyto nejobvyklejší položky (ale ne všechny čítače podporují všechny možnosti nastavení):

- TMRnON – aktivuje/deaktivuje čítač
- Tn(CK)PS – nastavení prescaleru
- TOUTPS – nastavení postscaleru
- PSA – aktivace prescaleru (na jiných procesor např. znamená přiřazení PS pro watchdog)
- položky pro nastavení zdroje vstupního signálu, které mají asi deset tisíc různých jmen

Konečně speciální funkční registry pro přerušení obsahují položky pro povolení přerušení TMRnIE, příznaky přerušení TMRnIF a možnost nastavení priority TMRnIP.

Hlídací pes

- ▶ bezpečnostní obvod, hlídá korektní běh procesoru
- ▶ čítač se zpravidla dedikovaným oscilátorem
- ▶ program MCU musí čítač pravidelně programově nulovat
- ▶ pokud čítač dosáhne stanovené hodnoty, resetuje MCU
- ▶ předpokládá se, že se MCU někde zapoměl
- ▶ konfigurace typicky pomocí pojistek

Tzv. Watch-dog timer je hojně využívaný obvod určený pro zvýšení spolehlivosti systému. V principu se jedná o samostatný čítač se samostatným zdrojem hodinového signálu (tj. nezávislém na systémových hodinách), který hlídá korektní chod procesoru. Úkolem programátora je v dostatečně častých a pravidelných intervalech čítač nulovat (= nakrmit hlídacího psa). Pokud dojde k přetečení čítače (tj. tento nebyl včas vynulován), vygeneruje hlídací pes resetovací signál a procesor zresetuje. Tímto způsobem lze předcházet chybovým stavům typu nekonečného zacyklení programu, nenadálý výpadek komunikace nebo chybná rekonfigurace hlavního oscilátoru.

2.2.4 Konfigurační a jiné paměti

EEPROM

EEPROM paměť slouží k uložení konfiguračních dat.

Obsluha pomocí SFR:

- ▶ **čtení:** zápisem log. 1 do položky RD registru EECON1 se hodnota na adrese EEADR uloží do EEDATA
- ▶ **zápis:** zápisem log. 1 do položky WR registru EECON1 se hodnota EEDATA uloží do paměti na adresu EEADR
- ▶ zabezpečení zápisu – před zápisem je třeba:
 - ▶ explicitně zápis povolit zápisem log. 1 do položky WREN registru EECON1
 - ▶ zapsat sekvenci 0x55, 0xAA do registru EECON2.

Podobně lze u některých MCU zapisovat do Flash paměti program nebo konfigurační pojistky.

Paměť EEPROM je typický zástupce permanentní paměti, tj. uchovává informaci i po vypnutí napájení číslicového obvodu. Je nasnadě otázka, proč pro konfigurační paměť není použita např. flash paměť, která zpravidla obsahuje kód programu. Důvod je jednoduchý, při podobných parametrech

snese paměť typu EEPROM o řád vyšší počet zápisových cyklů a předpokládá se, že principiálně počet změn konfigurace bude řádově vyšší než počet přeprogramování procesoru.

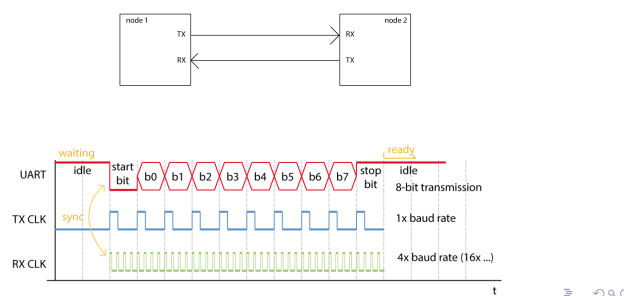
Procesy PIC typicky obsahují malý blok této paměti – jako jednu z periférií řízenou pomocí SFR, čtení a zápis se realizují tabulkovou metodou. Mezi pokročilejší procesory lze stejný subsystém využít i pro zápis do programové paměti Flash a konfiguračních pojistek

2.3 Komunikační metody

2.3.1 UART a RS232: sériová linka

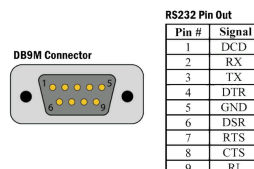
Komunikační protokol UART

- Universal Asynchronous Receiver-Transmitter
- volitelná délka datového slova (7,8,9 bitů)
- volitelně detekční kód – parita
- volitelná délka stop-bitu
- uzly domluvené na taktování a parametrech přenosu



Rozhraní RS-232

- dříve běžná součást PC jako "sériový port"
- využívá UART
- definuje fyzické rozhraní a napěťové úrovně
- standardní datové rychlosti
- definuje metodu synchronizace a dekodování na RX straně



UART (universal asynchronous receiver/transmitter) je obecně nízkoúrovňový komunikační protokol, v kontextu MCU pak logický obvod, který tento protokol realizuje. Nejběžnějším použitím byl

dříve protokol RS232, který bývával standardní výbavou PC (sériový port).

Hlavní rozdíl mezi UART a RS232 je tento: UART definuje komunikační protokol respektive chování komunikačního logického obvodu. Oproti tomu RS232 definuje elektrické rozhraní pro komunikaci mezi dvěma zařízeními (které využívá UART) a jeho parametry (napěťové úrovně, doplňkovou signalizaci ap.).

Protokol UART je v principu typu point-to-point, čili komunikují spolu právě dvě protistrany; existují však rozšíření, která umožňují vícestrannou komunikaci. UART je asynchronní, tedy komunikace není synchronizována hodinovým signálem. Pro přenos dat se využívá jeden signál (vodič) pro každý směr. UART je tedy jednosměrný protokol (simplex), pro realizaci obousměrné komunikace (full-duplex) je nutno využít dvou kanálů, tedy dvou signálů.

Data jsou přenášena po jednotlivých “znacích” (typicky Byte), kde jednotlivé bity jsou přenášeny sériově. Obecné schéma komunikace je zobrazeno na Obr. 2, základní délka každého bitu je daná zvolenou komunikační rychlostí (tzv. baud rate). V klidu je sběrnice ve stavu log. 1 z toho důvodu, aby přijímací strana rozeznala klidový stav a odpojenou sběrnici. Přenos znaku začíná tzv. start-bitem, který má opačnou polaritu než klidový stav. Následuje série datových bitů (typicky 5-9bitů) v pořadí od nejnižšího bitu po nejvyšší (nejběžněji je využíváno 8 bitů). Volitelně je datové slovo doplněno paritním bitem (viz dále). Vysílání znaku je ukončeno tzv. stop-bitem, který má stejnou polaritu jako klidová úroveň. Stop-bit může mít (z historických důvodů) různou délku: 1, 1,5 nebo 2 bity.

Vysílání: Vysílač UART je řízen hodinovým signálem s frekvencí odpovídající základní přenosové rychlosti. Vysílač je jednoduchý stavový automat, který v každém kroku vystavuje jeden bit v pořadí: start-bit, datové bity, (parita,) stop-bit.

Příjem: Příjímač UART je řízen hodinovým signálem s frekvencí odpovídající n -násobku základní přenosové rychlosti (typicky $4\times$ nebo $16\times$). Hodinový signál je sesynchronizován s první sestupnou hranou signálu (přechod z klidového stavu do start-bitu). Logická hodnota je čtena n krát během doby vystavení každého bitu. Hodnota načteného bitu se určuje průměrováním.

Nastavení komunikace: Aby byla komunikace možná, komunikující protistrany musí být “domluvené” na základních parametrech komunikace, kterými jsou:

- přenosová rychlost (baud-rate),
- počet bitů v datovém slově,
- délka tzv. stop-bitu (viz dále),
- využití parity

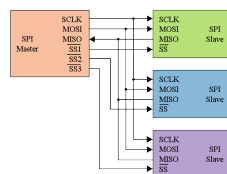
Parita: Jedná se o jednoduchý detekční kód, který umožňuje příjemci detekovat určité (statisticky významné) množství chyb v přenosu. Datové slovo je doplněno o jeden bit, který nabývá hodnoty log. 0 nebo log. 1 tak, aby počet jedniček v celém rozšířeném slově byl sudý (při případ tzv. sudé parity) nebo lichý (při případ tzv. liché parity).

Baud, Baud-rate: Jednotka Baud (Bd) je jednotka přenosové rychlosti. Udává počet symbolů/pulsů přenesených po komunikačním spoji za jednu vteřinu. Jednotka je příbuzná jednotkám, které udávají přenosovou rychlostí (bitů za sekundu, bps), nicméně nemusí úplně přesně korespondovat (např. v případě UARTu díky režii pro start/stop bity).

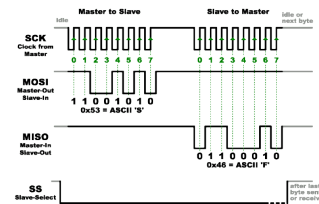
2.3.2 SPI, I2C: komunikace mezi integrovanými obvody

Sběrnice SPI

- ▶ master-slave sériová sběrnice
- ▶ hodinový signál
- ▶ dva datové signály: master in - slave out (MISO) master out - slave in (MOSI)
- ▶ slave-select signály



Typické užití: komunikace mezi IC nebo mezi blízkými komponenty



- ▶ duplexní režim možný
- ▶ korektní nastavení fáze a polarity
- ▶ libovolné napěťové úrovně i délka slova
- ▶ taktování > 10 MHz

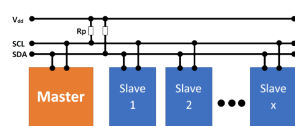
SPI (Serial Peripheral Interface) je sériové periferní rozhraní. Používá se pro komunikaci mezi řídicími mikroprocesory a ostatními integrovanými obvody (EEPROM, A/D převodníky, displeje...). Komunikace je realizována pomocí společné sběrnice. Adresace se provádí pomocí zvláštních vodičů, které při logické nule aktivují příjem a vysílání zvoleného zařízení (piny SS nebo CS).

Klíčové vlastnosti:

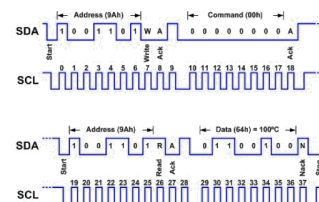
- Synchronní komunikace
- Více než dvě zařízení na jedné sběrnici
- Hodinový signál rozveden všem
- Frekvence komunikace až 70MHz

Sběrnice I2C

- ▶ master-slave sériová sběrnice
- ▶ hodinový signál SCL
- ▶ datový signál SDA
- ▶ externí pull-up pro střídaní zdroje signálu
- ▶ uzly se mohou v roli Master střídat



Typické užití: komunikace mezi IC



- ▶ součástí datového paketu adresa Slave uzlu
- ▶ potvrzování / timeouty
- ▶ taktování 400 KHz

Sběrnice typu I2C (též IIC nebo I²C, v aj. I-square C, Inter-integrated circuit) je dvojvodičová sériová sběrnice, která využívá jeden signál pro hodiny (SCK) a druhý pro data (SDA). Pro adresaci komunikující protistrany se využívá adresace, která je součástí přenášeného datového paketu. Stejně jako SPI se jedná o synchronní sériovou sběrnici, využívá méně vodičů, ale je pomalejší (max. takt cca 400kHz).

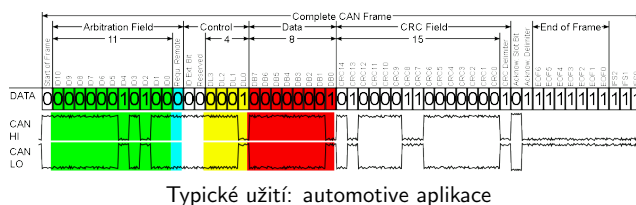
Jeden uzel je vždy řídicí (tzv. *Master*)², ostatní jsou podřízené (tzv. *Slave*). Pro přenos paketu začne řídicí uzel generovat hodinový signál a po datové sběrnici vysílá adresu podřízeného uzlu. Pokud je cílový uzel aktivní, převezme řízení datového signálu a vyšle potvrzovací pulz. Dále pak řídicí uzel generuje hodinový signál a jeden z uzlů vysílá obsah dat (dle směru komunikace) a druhý vysílá potvrzovací pulzy.

Protože se řízení signálu SDA střídá mezi komunikujícími uzly, je výchozí napěťová úroveň realizována pomocí externího pull-up odporu (typ. 1 – 10 kΩ).

2.3.3 CAN: spolehlivá komunikace mezi samostatnými uzly

CAN-bus (Controller Area Network)

- ▶ peer-to-peer sériová sběrnice, 1 diferenciální pár
- ▶ spolehlivé doručení, prioritizace provozu
- ▶ 11-bit (28-bit) CAN-id, 8 bajtů data
- ▶ rychlost až 1 Mbit/s, robustní provoz, optické oddělení uzlů



Typické užití: automotive aplikace

Sběrnice CAN je uvedena pouze pro ukázkou: jedná se o sériovou sběrnici s paketovým přenosem typu peer-to-peer, tj. k jednomu médiu (dvojvodičový diferenciální signál) je připojeno několik nezávislých uzlů. Využívá se protokol pro přístupu k médiu, který zjednodušeně řečeno udává střídání uzlů ve vysílání. Každý paket obsahuje identifikační kód (CAN-id) a až 8 bajtů užitečných dat.

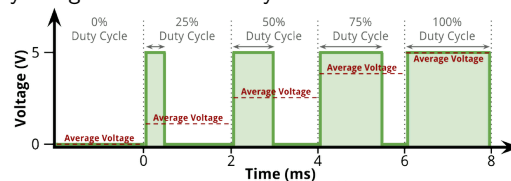
²Uzly se v řídicí roli mohou střídát, jejich synchronizace však není řešena na úrovni samotného protokolu a je na zodpovědnosti návrháře systému.

2.4 Analogové veličiny

2.4.1 PWM: pulsně-šířková modulace

Pulsně-šířková modulace (PWM)

Modulační technika, která umožňuje binárně generovat spojitou veličinu změnou délky aktivní doby pulzu v rámci periody
Zpravidla řízeno čítačem, který je porovnáván s registrem periody a registrem aktivní doby.



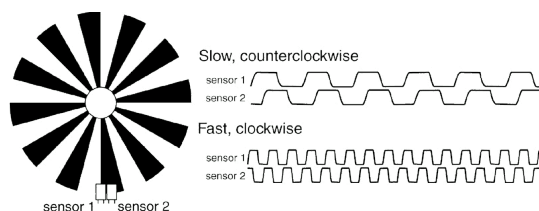
Typické užití: motory, světelné zdroje, výkonové zátěže obecně

Pulsně šířková modulace, neboli PWM (Pulse Width Modulation) je diskrétní modulace pro přenos analogového signálu pomocí dvouhodnotového signálu. Jako dvouhodnotová veličina může být použito například napětí, proud, nebo světelný tok. Signál je přenášén pomocí střídý. Pro demodulaci takového signálu pak stačí dolnofrekvenční propust. Vzhledem ke svým vlastnostem je pulsně šířková modulace často využívána ve výkonové elektronice pro řízení velikosti napětí nebo proudu. Kombinace PWM modulátoru a dolnofrekvenční propusti bývá rovněž využívána jako levná náhrada D/A převodníku.

2.4.2 QENC: rotační enkodér

Kvadraturní (rotační) enkodér

- ▶ elektromechanický/ elektrooptický systém
- ▶ převádí změnu úhlu na kvadraturní signál
- ▶ rozlišení: pulzy na otáčku
- ▶ vzájemná fáze dvojice signálů udává směr pohybu
- ▶ ev. doplněn 3. signálem pro indikaci absolutní polohy
- ▶ řízeno přerušením (A xor B)



Typické užití: mechanické ovládací prvky, úhlové a lineární délkové

senzory

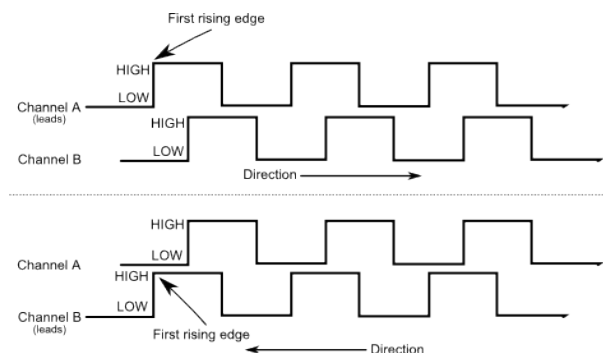


Figure 2.2: Signál z rotačního enkodéru

Rotační enkodér (kvadrurní enkodér, rotary encoder) je elektromechanický prvek, který převádí úhlovou pozici respektive rotační pohyb na analogový či digitální signál. Existují dva základní principy: absolutní a inkrementální.

Absolutní enkodér udává aktuální úhlovou pozici, reprezentovanou například napětovou úrovní (analogový absolutní enkodér) nebo digitálním kódem. Informace o rychlosti a směru otáčení se odvozují derivací pozice.

Inkrementální enkodér produkuje informaci o změně úhlové pozice, reprezentovanou dvojicí fázově posunutých signálů. Ty mají buď analogovou formu (typicky sinusový průběh) nebo digitální (pulsní průběh). Typický průběh výstupu digitálního enkodéru je zobrazen na obr. 2.4.2. Fáze dvojice signálů (na obrázku A a B) je posunutá o $\pi/2$, což umožňuje v digitální formě určit směr pohybu enkodéru; v analogové formě umožňuje navíc ještě přesněji rozlišit relativní úhlovou pozici $\varphi = \arctan(B/A)$.

Typické parametry inkrementálního enkodéru jsou:

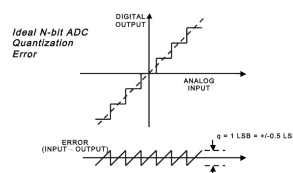
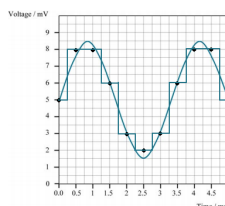
1. počet pulzů na otáčku (pulses per revolution, PPR)
2. maximální rychlost otáčení (otáčky za minutu, RPM)

Oba parametry jsou pevně dané mechanickou konstrukcí enkodéru.

2.4.3 ADC/DAC: převod analog-číslo a zpět

Převodník analog-číslo (ADC)

- ▶ převod spojitě veličiny na digitální reprezentaci
- ▶ diskrétní vzorkování
- ▶ diskrétní kvantizace
- ▶ přenosová funkce



- ▶ kvantizační chyba
- ▶ vzorkovací rychlost
- ▶ vzorkovací teorém
- ▶ filtr pro anti-aliasing

Analogově digitální převodník (převodník analog-číslo, zkráceně A/D, v angličtině i ADC) je elektronický obvod určený pro převod spojitě (neboli analogového) signálu na diskretní reprezentaci (digitální). Důvodem tohoto převodu je umožnění zpracování původně analogového signálu na číslicových počítačích. Mezi nimi v současnosti převažují digitální signální procesory DSP, které jsou právě na zpracování takových signálů specializované. V digitální podobě se také dají signály lépe přenášet, digitální reprezentace v principu není ovlivněna šumem přenosového kanálu.

Převodu analogové veličiny na číslicovou reprezentaci zahrnuje dva důležité kroky: vzorkování a kvantizace.

Vzorkování (sampling) je proces, kdy v čase spojitý průběh veličiny je zaznamenán v periodicky se opakujících časových okamžicích; tzn. jednou za definovanou periodu je zaznamenána okamžitá hodnota.

Kvantizace (quantization) je proces, kdy načtený vzorek (spojitá hodnota) je převeden na výsledný kód. Tím, že rozsah výstupních kódů je konečný, dochází k zaokrouhlení hodnoty na nejbližší diskretní krok – toto zaokrouhlení se označuje jako kvantizační chyba (quantization error).

Základní parametry AD převodníku jsou šířka pásma (bandwidth) zpracovávaného signálu a jeho bitové rozlišení. Dalšími parametry jsou pak napr. linearita převodníku, šumové vlastnosti, přizpůsobení vstupu (impedance), unipolární či bipolární zapojení a jiné.

Šířka pásma udává rozsah frekvencí, které je možné pomocí převodníku korektně zpracovat (viz Nyquistův teorém). V praxi má obvod převodníku konstrukčně danou maximální vzorkovací rychlost, udávanou v počtu vzorků za sekundu (samples per second, sps/Ksps/...).

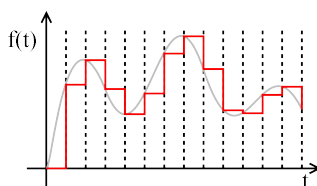
Bitové rozlišení (bitová hloubka) udává počet rozlišitelných diskretních úrovní, tedy výstupních kódů. Ve spojitosti se vstupním rozsahem převodníku udává nejmenší měřitelný krok vstupní veličiny.

Například: Uvažujme převodník, který má rozlišení 10bitů a vstupní rozsah 0-5 V. Převodník je tedy schopen rozeznat 1024 úrovní, nejmenší rozlišitelná hodnota je 4,88 mV.

Vstupní rozsah se v praxi odvozuje buď od napájecího napětí procesoru, od interní napěťové reference (realizované specializovaným obvodem) nebo od externí reference. Výhoda reference je zpravidla přesněji známá hodnota referenčního napětí, která není ovlivněna provozem ostatních částí logického systému.

Převodník číslo-analog(DAC)

- ▶ převádí digitálně reprezentovanou veličinu na analogovou
- ▶ vzorkování, kvantizace, přenos analogické opačným směrem
- ▶ rekonstrukční filtr



Digitálně analogový převodník (zkratky D/A, v angličtině i DAC) je elektronická součástka určená pro převod digitálního (diskrétního, tj. nespojitého) signálu na signál analogový (spojitý). Pro opačný převod signálu se používá A/D převodník. Ideálně navzorkovaný signál.

D/A převodník se dnes používá všude tam kde je třeba z digitálního signálu udělat zpět analogový tedy ve všech přehrávačích (CD, MP3 přehrávač, Minidisc, ...), zvukových kartách v počítačích. V každém moderním digitálním telefonu (ISDN, GSM, UMTS) se také nachází D/A a A/D převodník.

Pro DAC jsou definovány následující důležité veličiny:

- kvantovací krok - nejmenší možná změna výstupní analogové veličiny
- kvantovací hladina - diskrétní úroveň, kterou může výstupní analogová veličina dosáhnout
- počet kvantovacích hladin je dán délkou vstupního kódového slova a způsobem kódování

Reference

- Wikipedia, hesla Instrukční cyklus (Instruction cycle), Hodinový signál, Reset, SFR, Mikrokontroler PIC
- ad zpracování instrukcí: <http://www.fi.muni.cz/usr/pelikan/ARCHIT/TEXTY/PARPROC.HTML>
- ad pojistky: <http://ww1.microchip.com/downloads/en/DeviceDoc/70194F.pdf>
- ad sběrnice: <http://poli.cs.vsb.cz/edu/arp/down/komunikace.pdf>

minimálně pro základní obsluhu procesoru a periférií, v případě "větších" systémů pro interakci s jádrem operačního systému a pro běžně používané rutiny (např. práce s řetězci nebo matematické operace). Je zřejmé, že např. na PC by byla použitelnost jazyka C bez standardní knihovny minimální. Naopak v oblasti *embedded* a zejména u nejmenších, zpravidla osmibitových, kontrolerů se bez standardní knihovny snadno obejdu. Mimo jiné proto, že se vám standardní funkce nevejdou do paměti.

S aplikačním SW to v dnešní době už pochopitelně není tak žhavé; když už, používají se modernější deriváty (např. C++, C#).

3.1.1 Historie a standardizace jazyka C

Historie jazyka C lze letmo shrnout v následujících bodech:

- 1969-71: Jazyk C vznikl v laboratořích AT & T Bell Labs, autory jsou Brian Kernighan and Dennis Ritchie
- late 70s-80s: Jazyk C získává na popularitě, vyvíjí se standardní knihovna jako doplněk k překladači.
- 1990: vznikl první formální standard ANSI C
- 2004+: vzniká sada doporučení pro "Embedded C"
- 12/2011: Zatím poslední standard C11

Původní dialekt jazyka se dle autorů dnes označuje jako K&R C, ale díky zhruba 40leté historii vývoje jazyka C to nebyl zdaleka dialekt poslední. Za tu dobu bylo ustáleno několik standardů:

- K&R C – pre ANSI (1971)
- ANS X3.159-1989 (1989) – známá jako ANSI C, C89 obsahuje i standardní knihovnu C
- ISO/IEC 9899:1990 (1990) – známá jako ISO C, C90 identické s ANSI C standardem
- ISO/IEC 9899:1999 (1999) – C99
- ISO/IEC 9899:2011 (2011) – C11

Pro praktické využití je nutné vědět, že tyto normy existují a že některé vlastnosti jazyka nemusí být podporované ve starších verzích. Norma ANSI C / ISO C stále zůstává standardem nejrozšířenějším a patřívá k dobrým zvyklostem ji respektovat kvůli přenositelnosti (portabilitě).

Pro zajímavost, existuje ještě jedna nezávislá struktura standardů, vážící se k jazyce C se týká operačních systémů typu Unix, kde standardní knihovna koexistuje v těsné návaznosti na jádro OS:

- IEEE 1003.1 (last revision 2004)
POSIX 1 – Unix API

- Single Unix Specification, SUS (1994, 97, 2002) incorporates POSIX.1
- FIPS 151-1, 151-2 – Federal Information Processing Standards more specific than POSIX.1
- System V Interface Description, SVID3 / SVID4
- BSD

Co se týče embedded oblasti, standardy v podstatě neexistují. Jedním z náznaků jsou "Embedded C" technical reports, schválené ISO. Běžně překladače C pro embedded kopírují jednu ze standardních norem včetně (často nekompletní) implementace standardní knihovny, ke kterým jsou přidána rozšíření specifická pro konkrétní platformu. Typickým příkladem rozšíření syntaxe jazyka jsou mechanismy pro obsluhu přerušení nebo nastavení konfiguračních pojistek. V oblasti standardní knihovny jsou často předdefinovány pojmenované struktury a proměnné, usnadňující přístup k SFR nebo sady knihovnických funkcí pro práci s periferiemi. Naopak v některých částech je úkolem uživatele doprogramovat nízkourovňovou podporu pro vyšší knihovnické funkce (typicky standardní vstup-výstup, kde nelze u mikrokontroleru očekávat klávesnici a monitor) nebo jsou vyšší knihovnické funkce natolik paměťově náročné, že je pro některé kontrolery nelze prakticky využít.

3.1.2 Charakteristika jazyka C

Charakteristika języka C l.

- ▶ free-form jazyk, case-sensitive
- ▶ málo klíčových slov, mnoho aritmeticko-logických operátorů

Proměnné

- ▶ jednoduché typy - znak, celé číslo, reálné číslo
- ▶ složené typy heterogenní - struktury
- ▶ složené typy homogenní - pole, řetězce znaků
- ▶ ukazatele - odkaz na proměnou
- ▶ uživatelsky definované typy

[illegible]

Free-form znamená, že není třeba dodržovat počet mezer při odsazení bloku. Case-sensitive znamená, že `Cislo` a `cislo` jsou z hlediska překladače dva různé identifikátory. Z hlediska uživatele/programátora už to tak zřejmé není a proto je vhodné se podobným situacím vyhnout. Podobně problematická může být i záměna nuly a velkého O (`P0` vs. `P0`), jedničky a malého L (`h1` vs. `h1`) nebo umístění podtržítka na konec (`DetailniNazev` vs. `DetailniNazev_`).

Klíčová slova obsahuje jazyk C právě tyto: auto, break, case, char, const, continue, default, do, double, enum, extern, float, for, if, int, long, register, return, short, signed,

sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while. Poznámka na okraj: jedno jsme vám zamlčeli, ale máme k tomu dost dobrý důvod. Když zjistíte, které to je a napíšete nám jej emailem, dostanete jeden bonusový bod do závěrečného hodnocení. Klíčová slova se píší malými písmeny a je velmi vhodné nepoužívat příliš podobné identifikátory (jako např. auto2, If, d0).

Proměnná lze definovat jako prvek, uložený v datové paměti programu, který má tyto tři charakteristiky: jméno (identifikátor), definovaný datový typ a definovanou lexikální viditelnost. Jazyk C je tzv. *staticky typovaný*, čili každá proměnná musí být před prvním použitím definována (případně deklarována) a musí jí být přiřazen konkrétní datový typ¹. Nelze využívat anonymní nebo implicitní proměnné.² Jazyk C je také tzv. *weakly enforced*, což zjednodušeně řečeno znamená, že lze hodnoty konvertovat z jednoho datového typu na druhý implicitně, bez explicitního zadání programátorem. O identifikátorech, proměnných a konstantách více v následující kapitole.

Charakteristika języka C II.

Funkce:

- ▶ C program se skládá z procedur/funkcí – základní modularita
- ▶ deklarace = funkční prototyp
udává název funkce, vstupní argumenty a návratovou hodnotu
- ▶ definice obsahuje hlavičku a posloupnost příkazů
- ▶ funkce vždy definována vně jiné funkce

Na funkce i proměnné se vztahuje tzv. lexikální viditelnost: obé lze použít pouze pokud je již definováno nebo deklarováno.



Charakteristika jazyka C III.

Modularita:

- ▶ funkce podobného účelu jsou sdružovány do modulů.
- ▶ deklarace umožňuje "export" funkcí mezi moduly

Textový preprocesor jazyka C:

- ▶ zpracování zdrojového textu před kompilací
- ▶ možnost spojení zdrojových textů
- ▶ podpora textových maker
- ▶ podmíněná/modulární kompilace



¹Narozdíl od některých interpretovaných jazyků, jako např. PERL nebo PHP.

²Jako např. v jazyce PERL nebo funkcionálních jazycích.

Jazyk C je založený na funkcích. Uvažujeme-li program jako posloupnost příkazů, pak všechny tyto jsou zapouzdřeny do funkcí. Je dobré si pamatovat: příkazy patří dovnitř funkce ale definice a deklarace funkcí patří vně funkce.

Funkce lze přiovnat k funkcím v matematice (pro puntičkáře přesněji řečeno k *zobrazením*): funkce v C má argumenty funkci a vrací návratovou hodnotu. Obojí má definovaný datový typ – definiční obor a obor hodnot. *Procedura* je v kontextu C zvláštním případem funkce, která nepřijímá žádné argumenty (arita = 0) a nevrací návratovou hodnotu.

Pochopení konceptu definice a deklarace je klíčové pro návrh jakéhokoliv mírně složitějšího programu v jazyce C. Tento koncept je úzce svázaný se zmíněnou lexikální viditelností:

Lexikální viditelnost proměnných a funkcí (obecně identifikátorů) zjednodušeně praví, že každý identifikátor je platný od místa, kde je ve zdrojovém kódu definován až do konce bloku příkazů, ve kterém byl definován/deklarován. To v praxi znamená, že například:

- proměnná definovaná v rámci bloku je platná pouze do konce tohoto bloku a ne ve zbytku funkce
- proměnná definovaná na začátku funkce je platná ve dané funkci i v jejích zanořených blocích, ale ne v jiné funkci
- proměnná definovaná mimo funkci je platná ve všech následně definovaných funkcích v rámci modulu (souboru).
- funkce *definovaná* na začátku modulu (souboru) je platná a může být volána kdekoli v rámci tohoto modulu ale ne v modulech vedlejších
- funkce *deklarovaná* na začátku modulu (souboru) je platná a může být volána kdekoli v rámci tohoto modulu za předpokladu, že je *definovaná* v jiném modulu

Deklaraci tedy potřebujeme k volání funkce z jednoho modulu v jiném modulu. K čemu ještě? Typickým příkladem jsou dvě funkce (třeba A() a B()), které obě potřebují volat tu druhou. Z definice lexikální viditelnosti narážíme na problém typu vejce-slepice, tj. funkce A() musí být definována dříve než B() a zároveň funkce B() musí být definována dříve než A(). Tento logický spor lze vyřešit následovně: Nejprve je *deklarována* funkce B(), poté je definována funkce A() a konečně je definována funkce B(), čímž jsou splněna pravidla lexikální viditelnosti. Nutno dodat, že jazyk C nepřipouští vícenásobnou definici stejné funkce.

Preprocesor jazyka C je tím nástrojem, který umožňuje "vlepovat" zdrojové kódy do sebe a případně na základě podmínek zahrnovat nebo vylučovat části kódu z procesu překladač. O tomto více později.

Závěrem je třeba zmínit i co naopak v jazyce C nenajdeme:

- objektově orientovaný přístup – s troškou nadhledu lze některé základní postupy aplikovat i v C
- výjimky, kontrolu meze polí, polymorfismus
- garbage collection – nakládání s pamětí je na zodpovědnosti programátora
- GUI knihovny, datová persistence, lambda kalkul

3.2 Základní struktura jazyka C

“There are only two kinds of programming languages: those people always bitch about and those nobody uses.”

– *Bjarne Stroustrup*

3.2.1 Základní syntaxe, proměnné a konstanty

Tradiční, plně funkční, ukázkový program vypíše na terminál větu "Hello world!", odřádkuje a skončí:

```
#include <stdio.h>

void main(void){
    printf("Hello world!\n");
    getchar();
    return;
}
```

Na prvním řádku je příkaz (též nazýváno direktiva) preprocesoru, který vloží na začátek programu systémový soubor `stdio.h` (Standard I/O), který patří do standardní knihovny jazyka C a obsahuje řadu užitečných definic pro práci se vstupem a výstupem např. na terminál či do souboru.

Druhý řádek obsahuje pouze formátovací prázdný řádek, kterým vizuálně oddělíme dva logické celky od sebe.

Na třetím řádku vidíme funkční prototyp (předpis, hlavičku) funkce `main()`. Klíčové slovo `void` před názvem funkce značí, že funkce negeneruje návratovou hodnotu. Totéž klíčové slovo v kulatých závorkách značí, že funkce nepřijímá žádné parametry. Za funkční hlavičkou se vyskytuje otevírací složená závorka, což značí, že hned za funkčním předpisem následuje *definice* (neboli *tělo*) funkce.

Na čtvrtém řádku program volá funkci `printf()`, která umožňuje tzv. formátovaný výstup na terminál, s parametrem `"Hello world!\n"` (=řetězcová konstanta).

Na pátém řádku program volá funkci `getchar()`, která čte znak z terminálu. Prakticky se tím program pozastaví, dokud uživatel nestiskne (téměř) libovolnou klávesu na klávesnici.

Na šestém řádku program se nachází konstrukce `return`, která explicitně ukončuje funkci – a jelikož ukončuje funkci `main()`, ukončuje i celý program Na posledním řádku se nachází ukončovací složená závorka, která formálně ukončuje definici funkce `main()`.

Bezpečnostní varování: snadno se přihodí, že při kompilaci tohoto jednoduchého programku pojme váš antivir podeření z nekalé činnosti a zablokuje buď samotný program nebo rovnou celé prostředí. Příčina pravděpodobně tkví v tom, že smysluplné aplikace zpravidla řetězec "Hello World" neobsahují, narozdíl od několika mutací různého škodlivého SW, jehož autoři slepě vycházeli z tohoto ukázkového souboru.

Mírně a nepříliš smysluplně rozšířený program může vypadat například takto:

Hello world!

```
#include<stdio.h>

void main(void){
    int i;
    int k = 5;

    printf("Ahoj svete!\n");

    if(k > 1){
        for (i = 0; i < k; i++){
            printf("...%d\n", i);
        }
    }

    return;
}
```

Základní syntaxe C

- ▶ příkazy ukončené středníkem: `a = 42;`
- ▶ bloky příkazů ve složených závorkách: `{...}`
- ▶ jednořádkové komentáře: `// komentar do konce radku`
- ▶ víceřádkové komentáře: `/* komentar od .. do */`
- ▶ identifikátory (názvy proměnných a funkcí)
 - ▶ skládají se z alfanumerických znaků a podtržítka
 - ▶ jsou case-sensitive
 - ▶ nesmí začínat číslem
 - ▶ identifikátory začínající podtržítkem rezervované pro systémové knihovny

Komentáře vypadají ve zdrojovém kódu takto:

```
// add numbers
y = x + 1;

/*
    and now multiply
*/
z = x * y;
```

Pozor, víceřádkové komentáře nelze vnořit do sebe:

```
/* toto je komentar
```


float single precision floating-point, typ 32-bit (sign + 8exp + 23man)

double double precision floating-point, typ 64-bit (1 + 11 + 53)

long double extended precision, typ 64–80-bit

Specialitou je prázdný datový typ **void**, jehož význam bude ozřejmen dále.

Abeceda jazyka C a konstanty

- ▶ **celé číslo:** soustava dle prefixu implicitně typu int:
123 = 0173 = 0x7B = 0b01111011
- ▶ **racionální číslo:** implicitně typu double
1.0, 3.14, 5e6, .84
- ▶ u konstant lze explicitně zadat typ: 123U /*unsigned*/,
123L /*long*/, 3.14f /*float*/
- ▶ **znak**, v apostrofech: 'A' = '\101' = '\x041' = 65
- ▶ speciální znaky:
'\0' /*nulový znak*/,
'\n' '\r' /*konec řádku*/,
'\'' /* apostrof*/
'\\' /* zpetné lomítko*/

A set of small navigation icons typically found in Beamer presentations, including symbols for back, forward, search, and other slide controls.

Řetězcové konstanty

- ▶ uvádí se v uvozovkách, konstanty se implicitně spojují:
"kus textu" = "kus" " textu" // spojení konstant
- ▶ pokud má obsahovat uvozovku, nutno ošetřit zpětným lomítkem
"kdyz \"ano\" znamená \"ne\"" // escape sequence
- ▶ může obsahovat netisknutelné znaky
" radek1 \n radek dva" // na dva řádky
- ▶ může obsahovat i zpetne lomítka, apostrofy jsou OK
"neplet lomítko '/' a backslash!'\\"

A set of small navigation icons typically found in Beamer presentations, including symbols for back, forward, search, and other slide controls.

Příklady

```

int i = 0;           // same as "signed int i"
unsigned int ui;     // "unsigned ui" also valid
long int a = 1L;     // "long a" also valid
long long int u;     // c99 only
char c = 'A';        // signed, range -128..127!
unsigned char uc;    // unsigned, range 0..255
float f = 1.0f;      // single precision
double d = 0.5;      // double precision
long double xxx;     // extended precision
typedef unsigned long int my_type; // uživatelsky datový
my_type udaj;        // "udaj" typu unsigned long

```

V rámci definice lze proměnnou i tzv. inicializovat.

Příklady korektních a nekorektních definic proměnných:

```

int money$owed; (incorrect: cannot contain $)
int total_count; (correct)
int score2; (correct)
int 2ndscore; (incorrect: must start with a letter)
int long; (incorrect: cannot use keyword)

```

3.2.2 Operátory

Jazyk C je bohatý na **operátory**:

- ▶ přiřazení: `=`
- ▶ aritmetické: `+`, `-`, `*`, `/`, `%`,
- ▶ kombinované:
 - `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `~=`, `<<=`, `>>=`
- ▶ relační: `==`, `!=`, `<`, `>`, `<=`, `>=`
- ▶ logické: `!`, `&&`, `||`

Logické a relační operátory pracují s typem `int`:

logická 0 = FALSE = hodnota 0
 logická 1 = TRUE = nenulová hodnota

Pozor: nutno si neplést přiřazení a porovnání

Poznámka: častou chybou je záměna operátorů porovnání a přiřazení, například:

- `i == x` je TRUE pokud `i` je rovno `x`.

- `i = x` je TRUE pokud `x` je nenulové.

Operátory II.

- ▶ bitové: `~, &, |, ^`
- ▶ bitový posun: `<<, >>`
- ▶ inkrement/dekrement: `++, --`
- ▶ podvýraz: `()`
- ▶ podmínkový: `? :`
- ▶ volání funkce: `()`
- ▶ (de)referenční: `*, &, []`
- ▶ sekvenční: `,`
- ▶ typová konverze: `(type)`
- ▶ adresační: `., ->`
- ▶ pomocné: `sizeof()`

Operátory III.

- ▶ je definována priorita operátorů
- ▶ líné vyhodnocení – pokud je po vyhodnocení části výrazu známý výsledek, zbytek podvýrazu se nevyhodnocuje
- ▶ implicitní typová konverze při přiřazení: ořezání, přetečení
- ▶ implicitní typová konverze při vyhodnocení:
`int -> unsigned -> long -> float -> double`
`int i = 2; float f = i / 2;`
- ▶ explicitní TK: `float f=3.14; int i = (int)f;`

3.2.3 Řídicí struktury

Dvoucestné větvení - IF

Syntaxe:

```
if(condition) then expr1; or
if(condition) then expr1; else expr2;
```

Příklady:

```
if (a == 1) b = 2;           if (a == 1) b = 2;
                             else      b = 1;

if (a == 1) { b = 2; }      b = (a == 1) ? 2 : 1;

if (a == 1) {               if (a == 1){
    b = 2;                   b = 2;
};                             c = 1;
                             }else{
(a == 1) ? b = 2 : ;         a = 1;
                             }

if (a = 1) b = 2; // !!!
```

Vícecestné větvení - SWITCH/CASE

Konstrukce `switch()` přijímá celočíselný parametr a vyhodnotí `case`, jehož konstanta odpovídá hodnotě. Pokud takový není definován, vyhodnotí se větev `default`.

```
int c;                        char c;
switch(c){                    switch(c){
    case 1: ...                case 'f': ... //
        break;                 case 'F': ...
    case 2: ...                break;
        break;                 default: ...
    default: ...                break;
        break;                 }
}                                }
```

Poznámka: pokud vynecháte `break`, vyhodnocení pokračuje v další větvi.

Poznámka: parametrem větve `case` je vždy pouze číselná konstanta; tyto se nesmí v jedné konstrukci `switch` opakovat. Znak je pro tyto účely jen číslo.

Cyklus WHILE


Syntaxe: `while(condition) expr1;`

- cyklus probíhá dokud platí *condition*
- *condition* je vyhodnocena dříve než *expr*

```
a = 123; int b;           // prazdny cyklus
                           while(a < 0);

// jednoduchy cyklus
while (a > 0){             // nekonecny cyklus
    // spocitej            while (1){
    b += calculate(a);      b += calculate(a++);
                           }

    // odedcti 1 od a
    a--;                   // neproběhne nikdy
                           while(0 || a < 0);
}
```



Cyklus DO .. WHILE

Syntaxe: `do expr1 while(condition);`

- cyklus probíhá dokud platí *condition*
- *expr* je vyhodnocena dříve než *condition*, *expr* proběhne alespoň jednou

```
a = 32;

do{
    b += calculate(a);
}while (a > 0);
```



Cyklus FOR

Syntaxe: `for(c_start; c_end; c_iter) expr;`

1. na začátku se jednou vyhodnotí `c_start`
2. poté se vyhodnotí `c_end`:
 - ▶ pokud platí, vyhodnotí se `expr` (proběhne iterace)
 - ▶ pokud ne, cyklus končí
3. na vyhodnotí se `c_iter`, pokračuje se bodem 2.

```
// iterate through 0 to 9
for (i = 0; i < 10; i++){
    b = cool_fn(i);
}
```

Řízení cyklu

Příkaz `continue` skočí na
konec cyklu

Příkaz `break` ukončí
provádění cyklu

```
while (a > 0){
    // preskoc cisla
    // delitelna 5
    if(a % 5 == 0) continue;

    // spocitej a odedti 1 od a
    b += calculate(a--);

    // pokud vychazi soucet
    // zaporny, ukonci
    if(b < 0) break;
}
```

3.2.4 Funkce a moduly

Funkce

- ▶ izolovaná posloupnost příkazů
- ▶ fixní nebo variabilní počet argumentů
- ▶ návratová hodnota
- ▶ definice = funkční prototyp + tělo funkce
- ▶ deklarace = pouze funkční prototyp

Na následujícím příkladě lze demonstrovat několik momentů:

```
double pi_times (int num); // declaration - fn prototype
double circ (double r);

void main (void){ // funkce main
    double dia;
    dia = circ(3); // volani fce circ()
    ...
}

double circ (double r){ // definice fce circ()
    return pi_krat( 2 * r ) ;
}

double pi_krat (int num){ // definice fce pi_krat()
    return num * 3.14;
}
```

- deklarace funkcí
- definice funkce `main()`, která volá funkci `circ()`
- definice funkce `circ()`, která volá funkci `pi_krat()`

Nezapomeňte: Funkce je vždy definována vně funkcí.

Modularita jazyka C

- ▶ skupiny funkcí se sdružují do modulů
- ▶ jeden modul = soubor *.c obsahuje:
 - ▶ definice funkcí (vnitřních i veřejných)
 - ▶ definice globálních proměnných (dtto)
 - ▶ definice lokálních maker, datových typů
- ▶ příslušný hlavičkový soubor *.h definuje vnější rozhraní modulu:
 - ▶ deklarace vnějších datových struktur a maker
 - ▶ deklarace vnějších proměnných
 - ▶ deklarace vnějších funkcí

Právě hlavičkové soubory programátorovi zprostředkovávají přístup ke funkcionalita standardní knihovny jazyka C.



Modularita - příklad

```
// circ.h
double pi_times (int num);
double circ (double r);

// main.c
#include "circ.h"

void main (void){
    double dia;
    dia = circ(3);
    ...
}

double pi_krat (int num){
    return num * 3.14;
}
```



Preprocesor jazyka C

- ▶ 1. krok procesu překladač
- ▶ příkazy (direktivy) preprocesoru začínají znakem #
- ▶ zpracuje zdrojový text před překladem, odstraní komentáře
`#include <soubor.h>, #include "muj.h"`
- ▶ provede náhradu maker - symbolické konstanty a makra s parametry
`#define, #undef`
- ▶ podmíněný překlad
`#if #elif #else #endif`
`#ifdef #ifndef`
- ▶ řízení překladače
`#pragma`

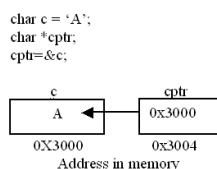
3.2.5 Ukazatele a složené datové struktury

Složené proměnné a jiné příbuzní

- ▶ ukazatel – nenesí hodnotu, ale odkaz na jinou proměnnou
- ▶ pole – složený typ, indexovaná posloupnost (od nuly) položek stejného datového typu
- ▶ řetězec (znaků) – realizován jako pole typu `char`, ukončené nulovým znakem `'\0'`
- ▶ uživatelsky definované datové typy
- ▶ `struct` – složený typ, sdružuje více položek (pojmenovaných) různých typů
- ▶ `enum` – celočíselný výčet, pojmenované konstanty
- ▶ `union` – umožňuje přístup k jednomu paměťovému místu jako k různým datovým typům

Ukazatele (pointers)

- **Ukazatel** je proměnná, která nese odkaz (adresu) jiné proměnné (struktury, funkce)
- typované: "ukazatel na (typ) XXX"
- lze je přetypovat
- rozsah odpovídá architektuře platformy



Ukazatele jsou klíčovým prvkem jazyka C; jejich pochopení je branou k pochopení celého konceptu jazyka!

Navigation icons: back, forward, search, etc.

Ukazatele a operátory

- **referenční operátor &**: &var znamená adresu proměnné var
- **dereferenční operátor ***: *ptr znamená hodnotu, uloženou na adrese ptr

```
int i = 1, j; // i a j jsou typu int
int *p_i;    // pi je ukazatel na typ int
p_i = &i;    // uloz adresu i do p_i
*p_i = 5;    // to stejne jako i = 5;
```

Poznámka: hvězdička v definici ukazatele a dereferenční operátor jsou dvě různé věci!

Navigation icons: back, forward, search, etc.

Práce s ukazateli

S ukazateli lze počítat:

- ▶ porovnávat (==, !=, <, >): `if(p1 == p2){ ...`
- ▶ odčítat (`p1 - p2`)
- ▶ přičítat celé číslo integer: `*(p + n)`, `p++`, `p--`
Pro `int *p ⇒ *(p+n)` ukazuje na n -tý prvek typu `int` za adresou n .

Ukazatel může být argumentem/návratovou hodnotou funkce:

```
int prohod(int *a, int* b); //volani odkazem
int* vetsi(int *a, int* b);
```

Ukazatel může být **prázdný** a/nebo **prázdného typu**:

```
void *ptr;      // prazdny typ
int *pi = NULL; // specielni nulova hodnota ukazatele
```



Pole

- ▶ homogenní posloupnost prvků stejného typu
- ▶ dobře uspořádaná, indexovaná od nuly
- ▶ prvky jsou uloženy v paměti v pořadí za sebou, meze nejsou hlídány překladačem
- ▶ hranaté závorky pro definici i pro přístup k prvku pole

```
int a[10]; // definujeme pole o 10 prvcích
a[1] = 6;  // prirad hodnotu druhého prvku
          // (první is a[0])
a[10] = -1; // tzv. off-by-one error
int prime[3] = {5, 7, 11}; // pole vc. inicializace
```

Pozor! Pole **nejsou primární datatyp**, nelze je přiřazovat:

```
pole1 = pole2 // no f***ing way!!!
```



Pole a ukazatele (!!!)

Pro `int a[3] = 3,5,6; int *pa = &(a[0]);` platí následující:

- ▶ `a[0]` == `*pa` – hodnota 1. prvku
- ▶ `a[1]` == `*(pa + 1)` – hodnota 2. prvku
- ▶ `&(a[n])` == `pa + n` – hodnota *n*. prvku

Proč? Konstrukce `a[0]` se vskutečnosti skládá z:

- ▶ konstantního **ukazatele** a
- ▶ **operátoru** pro přístup k prvku pole []

Proto dále plati:

- ▶ hodnota `a[0]` == `*a` == `*pa` == `pa[0]`
- ▶ hodnota `a[n]` == `*(a + n)` == `*(pa + n)` == `pa[n]`
- ▶ ukazatel `&(a[n])` == `a + n` == `pa + n`



Pole a funkce

- ▶ pole může být argumentem i návratovou hodnotou funkce
- ▶ v obou případech je předává **pouze ukazatel (!)**
- ▶ demonstrace, proč C **nekontroluje velikost polí** – principiálně ani nemůže

```
int max(int *in, int pocet){
    int i, max = in[0];
    for (i = 1; i < pocet; i++)
        if(in[i] > max) max = in[i];
    return max;
}
```

```
int mojedata[7] = {1,5,388,5,3,5,7};
int nejveci = max(mojedata, 7);
```



Řetězce

V jazyce C jsou řetězce realizovány jako pole typu `char`, ukončené nulovým znakem `'\0'`.

```
char pozdrav[6] = "Ahoj!"; // znak navíc kvůli '\0'
char pozdrav2[] = "Zdar!"; // prekladace dosadí delku
char pozdrav3[] = {'c','a','u','\0'}; // po znacích
```

```
pozdrav[2] = 'g'; // pozdrav obsahuje "Agoj!"
```

```
int delka(char *str){ // funkce pro práci s řetězcem
    int i = 0, len = 0;
    while(str[i++] != '\0') len++;
    return len;
}
```

Struktury I.

- obsahuje množinu pojmenovaných položek, obecně různého typu

- **Syntaxe:**

```
struct struct_name { entries } var_name(s);
typedef struct { entries } type_name;

struct { // anonymní struktura, def. proměnné
    int vyska;
    float vaha;
    int vek;
} anna, bara, dana;

struct gf{ // pojmenovaná struktura
    int vyska;
    float vaha;
    int vek; };
struct gf anna, bara, dana;
```

Struktury II.

```
// define new datatype
typedef struct gf{    // definovaný datový typ
    int vyska;
    float vaha;
    int vek;
    char prezdivka[25];
} t_gf;

t_gf anna, bara, dana;    // define variables
t_gf *bff;    // define pointer

// init
t_gf simona = {185, 75.8, 22, "Simi"};
t_gf market = {.prezdivka = "Dracice", .vyska = 168,
               .vaha = 55.2, .vek = 20 };

```

Struktury a operátory

```
// op pro přístup k prvkům:
anna.vek = 21; // staticky
bara.vyska = 170;

bff = &dana;    // pomocí ukazatele
bff->weight = 55;
(*bff).age = 22; /

anna = bara;    // strukturu lze přiřazovat

t_gf cizinky[15]; // pole struktur

cizinky[12].height = 145;

t_gf *bff_cizi;
bff_cizi = &foreigners[10];
bff_cizi->age = 14;

```

Struktury a funkce

```
// volani hodnotou
void print_age (t_gf kamoska){
printf("Vek je %d\\d",kamoska.vek);
}
print_age(bara);

// volani odkazem
void print_age (t_gf *kamoska){
printf("Vek je %d\\d",kamoska->vek);
}
print_age(&market);
}
```

Uniony, výčtové typy

Výčtové typy: celá čísla s pěknými jmény

```
enum semafor { red, green, blue}; // values 0, 1, 2
enum semafor t1; // promenna t1
t1 = red; // prirazeni
int a = red; // tez mozno, slabe typovano
```

Uniony: podobné strukturám, uschovává pouze jeden element

```
union vyska {
    long simpleDate;
    double perciseDate;
} mojevyska ;
mojevyska = 10; // long
mojevyska = 10.34; // double
```

Typové modifikátory

```
const int ci = 5; // nelze přiřadit jinou hodnotu
volatile int a;  // asynchronní přístup k proměnné

a = sizeof(struct gf); // operátor určí delku v bytech

// funkce se tímto zavazuje nemenit předávanou hodnotu
int fce (const char *str){
static int i; // static ve fci udržuje hodnotu
...

// globalní static nebude viditelný vně modulu
static int globalData;
```

3.2.6 Další možnosti

Na co zatím nedošlo

- ▶ detailní specifikace operátorů: bitové
- ▶ doporučená struktura hlavičkových a zdrojových souborů
- ▶ štabní kultura
- ▶ práce s pamětí
- ▶ statická/dynamická 2D pole
- ▶ proces kompilace
- ▶ nedefinované chování
- ▶ ukazatele na funkce

Samostudium - jazyk C

- ▶ <https://www.studytonight.com/c/overview-of-c.php>
- ▶ <https://www.studytonight.com/c/programs/>
- ▶ Herout, P: Učebnice jazyka C
- ▶ Doplnkové materiály k předmětu

3.3 Standardní knihovna jazyka C

Standardní knihovna

Sada knihoven pro typické úlohy:

- ▶ `stdio.h` – core input and output functions
- ▶ `stdlib.h` – numeric conversion functions, pseudo-random numbers generation functions, memory allocation, process control functions
- ▶ `string.h` – string handling functions.
- ▶ `math.h` – common mathematical functions.

```
#include <stdio.h>
printf("Hello world!");
```

```
#include <math.h>
float a = sin (1.5 * M_PI);
```


Standardní knihovna: příklady

```
#include <stdio.h>
int getchar(void); // read character from standard input
int putchar(int c); // put character from standard input
int puts(const char *str); //put string
int printf(const char *format, ...); // formátovaný výstup

#include <string.h>
// copy src to dest (aka dest = src)
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n); // secure
size_t strlen(const char *s); // vrátí délku řetězce

#include <stdlib.h>
int atoi(const char *nptr); // převod řetězce na číslo int
long atol(const char *nptr); // převod řetězce na long
void *malloc(size_t size); // dynamická alokace paměti
void free(void *ptr); // a její uvolnění
```

3.4 Podpora jazyka C v embedded

Podpora C pro embedded I.

- ▶ překladač vždy pro konkrétní platformu (vendor-specific), IDE zpravidla taktéž – podpora datových typů
- ▶ standardní knihovna implementuje část standardních knihoven – např. chybí vlákna, práce s časem
- ▶ standardní funkce často příliš náročné – např. `printf()`
- ▶ některé standardní funkce částeční DIY – standardní I/O
- ▶ doplněny funkce pro řízení periférií – EEPROM, PWM, SPI, UART, ...
- ▶ jazykové konstrukce pro konfigurační pojistky – `#pragma WDT_ON`
- ▶ triviální názvy pro SFR

Kde hledat informace

- ▶ Logické obvody, synaxe jazyka C, standardní knihovna C – ST*G
- ▶ Podpora C pro MCU, knihovní fce pro MCU – manuál k překladači
- ▶ Architektura a periferie MCU – datasheet MCU, family datasheet
- ▶ Zapojení periferií v konkrétním HW – manuál ke kitu



Reference

Studijní materiály

- Herout, Pavel: Učebnice jazyka C. KOPP, České Budějovice, duben 2009, VI. vydání, ISBN 978-80-7232-383-8, 280 stran
<http://www.kiv.zcu.cz/~herout/>
<http://ucebnice.heureka.cz/ucebnice-jazyka-c-1-dil-6-v-herout-pavel/>
- Development of C, by D. Ritchie
<https://www.cs.bell-labs.com/who/dmr/chist.html>
- Prezentace C Introduction, součást materiálů k předmětu REV
- Reference ke standardní knihovně jazyka C:
<http://www.cplusplus.com/reference/clibrary/>
http://www.gnu.org/software/libc/manual/html_mono/libc.html
- Embedded C Technical Report
<http://www.embedded-c.org/>
- Study overnight: jeden z tisíce tutoriálů
<https://www.studytonight.com/c/overview-of-c.php>

Nástroje

- vývojové prostředí DevC++ (STG)
- volně dostupný překladač MinGW:
<https://sourceforge.net/projects/mingw/>
- sériový terminál:
<https://sites.google.com/site/terminalbpp/>