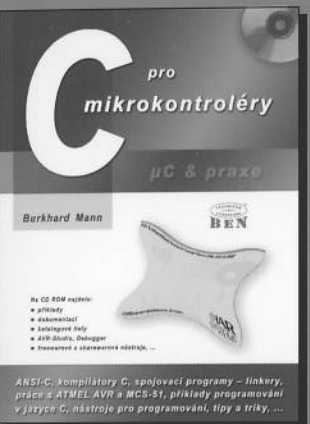
DOPORUČUJEME



C pro mikrokontroléry

Chcete se naučit, jak se programují mikrokontroléry v jazyce C?
Pak je kniha „C pro mikrokontroléry“ pro vás ta pravá.
Po názorném, koncentrovaném úvodu do prvků jazyka „ANSI-C“ jsou popsány zvláštnosti a rozšíření pro aplikace u mikrokontrolérů. Vedle mnoha užitečných tipů a triků se dozvíte, jak musí vypadat dobrý program v C a jak se vyhnete chybám. Typické praktické příklady pro rodiny mikrokontrolérů AVR a '51 od firmy Atmel vzájemně spojují teorii a praxi a uvádějí konkrétní souvislosti mezi architekturou mikrokontroléru a jazykem C.

Autor Burkhard Mann, 280 stran B5 + CD ROM, objednací číslo 121120, MC 499 Kč.

TECHNICKÁ LITERATURA ®
BEN
http://www.ben.cz

Doporučená cena
299 Kč

Objednací číslo
121139

ISBN 80-7300-102-0
9 788073 001025

Mikrokontroléry Atmel AVR Programování v jazyce C

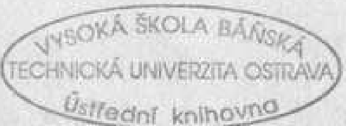
Mikrokontroléry ATMEL AVR

Vladimír Váňa

Programování v jazyce C

Popis a práce ve vývojovém prostředí CodeVisionAVR C

TECHNICKÁ LITERATURA ®
BEN



VYSOKÁ ŠKOLA BÁŇSKÁ
TECHNICKÁ UNIVERZITA OSTRAVA
Ústřední knihovna

157 506 - 4290/2003
přil. CD II-1299,
004

Vladimír Váňa

Mikrokontroléry ATMEL AVR – Programování v jazyce C –

Popis a práce ve vývojovém prostředí CodeVisionAVR C

Bez předchozího písemného svolení nakladatelství nesmí být kterákoli část kopírována nebo rozmnožována jakoukoli formou (tisk, fotokopie, mikrofilm nebo jiný postup), zadána do informačního systému nebo přenášena v jiné formě či jinými prostředky.

Autor a nakladatelství nepřejímají záruku za správnost tištěných materiálů. Předkládané informace jsou zveřejněny bez ohledu na případné patenty třetích osob. Nároky na odškodnění na základě změn, chyb nebo vynechání jsou zásadně vyloučeny.

Všechny registrované nebo jiné obchodní známky použité v této knize jsou majetkem jejich vlastníků. Uvedením nejsou zpochybněna z toho vyplývající vlastnická práva.

Veškerá práva vyhrazena
© Vladimír Váňa, Praha 2003
© Nakladatelství BEN – technická literatura, Věšínova 5, Praha 10

Vladimír Váňa: Mikrokontroléry ATMEL AVR – Programování v jazyce C
BEN – technická literatura, Praha 2003
1. vydání

ISBN 80-7300-102-0

OBSAH

Co najdete na doprovodném CD 7

1 Ještě než začtete 8

2 Popis CodeVisionAVR C 9

3 Vývojové prostředí (IDE) CodeVisionAVR C 11

4 Referenční manuál překladače C CodeVisionAVR 17

4.1 Direktivy preprocesoru a pragmy 17

4.1.1 Preprocesor 17

4.2 Komentáře 23

4.3 Klíčová slova 24

4.4 Identifikátory 24

4.5 Datové typy 24

4.6 Konstanty 24

4.7 Proměnné 26

4.8 Uživatelem definované datové typy 36

4.9 Typové konverze, přetypování 36

4.10 Operátory 38

4.11 Funkce ... 40
4.12 Ukazatele ... 40
4.13 Přístup k I/O registrům ... 42
4.14 Přístup k EEPROM ... 44
4.15 Použití přerušení ... 45
4.16 Využití SRAM ... 46
4.17 Použití externího souboru STARTUP.ASM ... 48
4.18 Využití assembleru ve zdrojovém kódu C jazyka ... 51
4.19 Volání funkcí napsaných v assembleru ... 51
4.20 Využití debugeru AVR studia ... 53
4.21 Zbývající rysy překladače CodeVisionAVR C ... 54
5 Knihovní funkce jazyka C CodeVisionAVR ... 55
5.1 Znakové funkce ... 55
5.2 Standardní I/O funkce ... 56
5.3 Funkce standardní knihovny ... 59
5.4 Matematické funkce ... 60
5.5 Řetězcové funkce ... 63
5.6 BCD konverzní funkce ... 68
5.7 Konverzní funkce Grayova kódu ... 68
5.8 Funkce pro přístup k paměti ... 69
5.9 Funkce pro LCD ... 69
5.10 Funkce sběrnice I2C ... 73
5.11 SPI funkce ... 77
5.12 Funkce pro úsporný režim (Power Management Functions) ... 80
5.13 Funkce časových prodlev, časového zpoždění ... 81
6 Vytváření knihoven ... 83
6.1 Vytvoření vlastní knihovny ... 83
6.2 LCD displej a knihovna pro jeho ovládání z jazyka C ... 86
7 Příklady ... 93
7.1 Program 1 – ovládání LED diod, blikač ... 93
7.2 Program 2 – vyslání nápisu na LCD displej ... 101
7.3 Program 3 – vyslání řetězce znaků na RS232 ... 104
7.4 Program 4 – vstupy z tlačítek ... 110
7.5 Program 5 – maticová klávesnice ... 112

7.6 Program 6 – klávesnice PC ... 121
7.7 Program 7 – voltmetr ... 137
7.8 Program 8 – čítač ... 139
7.8a Program 8 – měřiče kmitočtu ... 145
7.9 Program 9 – hodiny ... 146
7.10 Program 10 – sběrnice MicroWire ... 150
7.11 Program 11 – I²C zápis ... 155
7.12 Program11 – I²C čtení ... 159
7.13 Program 11 – PLL syntezátor kmitočtu řízený I²C ... 162
7.14 Program 12 – SPI ... 170
7.14 Program 13 – PWM ... 178
7.15 Program 14 – USB ... 181
8 Závěrečná poznámka ... 197
9 Příloha – programování v AVR GCC ... 199
Literatura a odkazy na Internetu ... 205
Knihy BEN – technická literatura ... 206

CO NAJDETE NA DOPROVODNÉM CD-ROM

Doprovodné CD-ROM obsahuje všechny informace potřebné pro snadnou práci s knihou. Tyto informace lze rozdělit do logických celků, které se nacházejí v oddělených adresářích:

- adresář **BEN** obsahuje off-line verzi www stránek nakladatelství BEN – technická literatura (aktualizováno k počátku léta 2003), jejichž součástí je počítačová verze tištěného katalogu – Edičního plánu „jaro a léto 2003" a samostatného přehledu naší produkce „BEN 2003".
- adresář **DATASHEET** obsahuje dokumentaci ve formátu PDF vybraných integrovaných obvodů ATMEL, které jsou v knize používány. Pro úplnost jsou zde i některé další periferní obvody mající vztah ke zveřejněným příkladům.

  Najdete zde též samorozbalitelný archiv programu Adobe Acrobat Reader verze 5.0, který slouží k prohlížení PDF souborů,
- adresář **NAPADY** obsahuje skutečné aplikace posbírané na Internetu, které mají sloužit jako inspirace, zejména pro amatérské konstruktéry. Viz též str. 198,
- adresář **PRIKLADY** obsahuje zdrojové i přeložené formy všech programů realizovaných v knize,
- adresář **SW** obsahuje samostatné složky s volně šiřitelnými verzemi nebo demoverzemi vývojového prostředí určeného pro procesory ATMEL AVR.

ATMEL – obsahuje především více verzí vývojového prostředí AVR Studio v3.20, v3.56 a v4.07 Všechny verze pracují pod operačním systémem Windows. Starší verze (3.xx) jsme uvedli proto, že pracují téměř na každém PC s prostředím alespoň Windows 95.
Navíc je na CD program WAVRASM v1.30, který rovněž umožňuje kompletní vývoj programů pro ATMEL AVR v assembleru. Pro čtenáře bude jistě i užitečný ovládací program pro programátor ATMEL AVR ISP 3.30, který je rovněž ve složce ATMEL.

BASCOM – vývojové prostředí včetně překladače z jazyka, který se podobá známému Visual Basicu 6.0. Je produktem firmy MCS Electronics. Omezení je na maximálně 2 kB výsledného kódu (HEX). Výhodou jsou speciální příkazy podporující práci s LCD displeji, komunikaci I²C, 1WIRE atd.

CVAVR – výborným kompilátorem C pro AVR, včetně vývojového prostředí, je CodeVision AVR. Rovněž tento překladač C lze nainstalovat jako součást AVR Studia. Zdarma je jeho školní verze (CodeVisionAVR C Compiler v1.23.5 Evaluation), jejímž jediným omezením je velikost výsledného kódu do 2 kB.

GNU_C – Kompilátor C, který lze nainstalovat jako součást AVR Studia. Na tento překladač není žádné časové omezení nebo omezení velikosti kódu. Je k dispozici zcela zdarma. Pro jeho užití je pouze nutné dodržet licenci GNU.

IAR – obsahuje časově omezená vývojová prostředí firmy IAR. Jedná se především o assembler a překladač z jazyka C/C++. Konkrétně se jedná o IAR Embedded Workbench Evaluation version for Atmel AVR v2.27B a IAR Embedded Workbench Assembler Edition for Atmel AVR v1.50B. Navíc je zde umístěn i produkt IAR MakeApp for Atmel AVR v3.01.

JAVA – klasická Java, ke které jsou přidány knihovny JEPES dánské firmy Mjolner Informatics. Demoverze umožňuje programovat pouze AT90S8515.

PASCAL – ideální prostředek pro programování, jedná se o školní verzi produktu (demo) německé firmy E-LAB Computers. Omezení je na maximálně 4 kB výsledného kódu (HEX), což pro většinu aplikací stačí. V assembleru to představuje cca 6000 řádků kódu.

1

JEŠTĚ NEŽ SE ZAČTETE

Mikrokontroléry ATMEL AVR AT90S si získávají stále větší oblibenost mezi profesionálními i amatérskými konstruktéry vestavěných (emebedded) zařízení. Při vytváření zařízení s mikropočítači či mikrokontroléry je důležitou částí jejich vývoje a konstrukce tvorba jejich programového vybavení. K tomu je ovšem potřeba mít vhodné prostředky – nějaké vývojové prostředí zahrnující mj. překladač z nějakého jazyka do kódu procesoru mikrokontroléru. Neméně důležité je mít schopnost s těmito prostředky pracovat.

Prvními programovacími jazyky byly assemblery. U větších počítačů jako jsou osobní počítače PC, pracovní stanice či mainfraimy se již téměř nepoužívají a tak jediným polem působnosti pro assemblery zůstaly především jednočipové mikropočítače a mikrokontroléry. Pokud potřebujeme vytvořit jednoduchý program pro takový „jednočipák" je použití assembleru ještě únosné. S rozvojem schopností těchto malých počítačů potřebují k jejich využití konstruktéři vytvářet programy poměrně rozsáhlé a složité a jejich tvorba v assembleru se stává již neúnosná.

Proto byly pro jednočipové mikrokontroléry a mikropočítače vytvořeny překladače z vyšších programovacích jazyků. Velké obliby dosáhl zejména jazyk C, což je dané tím, že má nejenom vlastnosti, které očekáváme od vyšších programovacích jazyků, ale i vlastnosti očekávané spíše u assemblerů. Z vyšších programovacích jazyků má jazyk C „nejblíže" k hardwaru. Proto se i u velkých počítačů používá při vytváření operačních systémů.

Používání vyšších programovacích jazyků respektují dokonce i tvůrci procesorů, když navrhují jejich jádro optimalizované pro práci s kompilátory nějakého vyššího jazyka. Příkladem může být např. procesor Chip švédské firmy Imsys optimalizovaný pro jazyk Java.

RISCový procesor mikrokontrolérů ATMEL AVR byl v norském vývojovém centru Nordic VLSI v Trondheimu navržen tak, aby vyhovoval zejména široce používanému jazyku C. Programováním mikrokontrolérů ATMEL AVR v jazyce C se budeme zabývat v této publikaci. Ta je určena především začátečníkům, předpokládá se u nich alespoň základní znalost jazyka C, např. na úrovni středoškolské učebnice jazyka C.

2

POPIS CodeVisionAVR C

Vývojový prostředek **CodeVisionAVR** obsahuje překladač jazyka C, integrované vývojové prostředí IDE a průvodce (wizard), umožňující automatické generování zdrojového kódu pro mikrokontroléry ATMEL AVR.

CodeVisionAVR je program spustitelný pod Windows 95, 98, 2000 a XP.

Jeho překladač jazyka C z větší části, pokud to umožňuje architektura AVR, vyhovuje specifikaci ANSI C a dále má několik rozšíření vyhovující potřebám vestavěných (embedded) systémů, speciálně AVR. Je možné zvolit formát souborů, které budou výsledkem překladu. Takovým formátem může být např. Intel HEX vhodný jako vstup pro programátor mikrokontrolérů AVR nebo formát COFF, který umožňuje ladění, debugging, na úrovni zdrojového kódu pomocí debuggeru Atmel AVR Studia.

Součástí CodeVisionAVR C jsou kromě standardních knihoven jazyka C další knihovny určené pro použití s:

- alfanumerickými LCD moduly,
- sběrnicí I²C,
- teplotním čidlem LM75,
- obvody reálného času PCF8563, PCF8583, DS1302 a DS1307,
- protokolem 1 wire (firma Dallas Semiconductor),
- obvody DS1820, DS1822, DS1621, DS2430 a DS2433 firmy Dallas Semiconductor,
- sběrnicí SPI,
- časové prodlevy,
- podporu Grayova kódování.

Dále obsahuje průvodce automatickou tvorbou kódu, implementující následující funkce:

- nastavení přístupu k externí paměti,
- identifikace zdroje resetu,

- nastavení I/O portů,
- inicializace externích přerušení,
- inicializace čítačů/časovačů,
- inicializace hlídacího obvodu Watchdog,
- inicializace UARTu,
- inicializace analogového komparátoru,
- inicializace převodníku ADC,
- inicializace SPI,
- obsluha I²C a navíc ještě podpora některých obvodů s I²C komunikací,
- sběrnici 1 wire a dále ještě podporu několika obvodů s 1 wire,
- inicializaci modulu LCD displeje.

CodeVisionAVR C je produktem firmy HP InfoTech S.R.L., hlavním autorem překladače je Pavel Haiduc. Instalace tohoto produktu se provádí spuštěním instalačního programu Setup.exe. Tento program je pro školní verzi, která je k dispozici zdarma, umístěn v souboru CVAVRE.zip. Hlavním omezením školní verze je omezení délky vytvářeného kódu. **Všechny příklady v této knize jsou tvořeny a překládány pomocí školní verze**. Dalším omezením školní verze je, že jeho wizard nepodporuje tvorbu kódu pro některé obvody, komunikující s AVR MCU. Toto omezení příliš nevadí, protože jednak jde o obvody u nás příliš nepoužívané, jednak je možné použít obecné knihovny pro I²C či 1 wire.

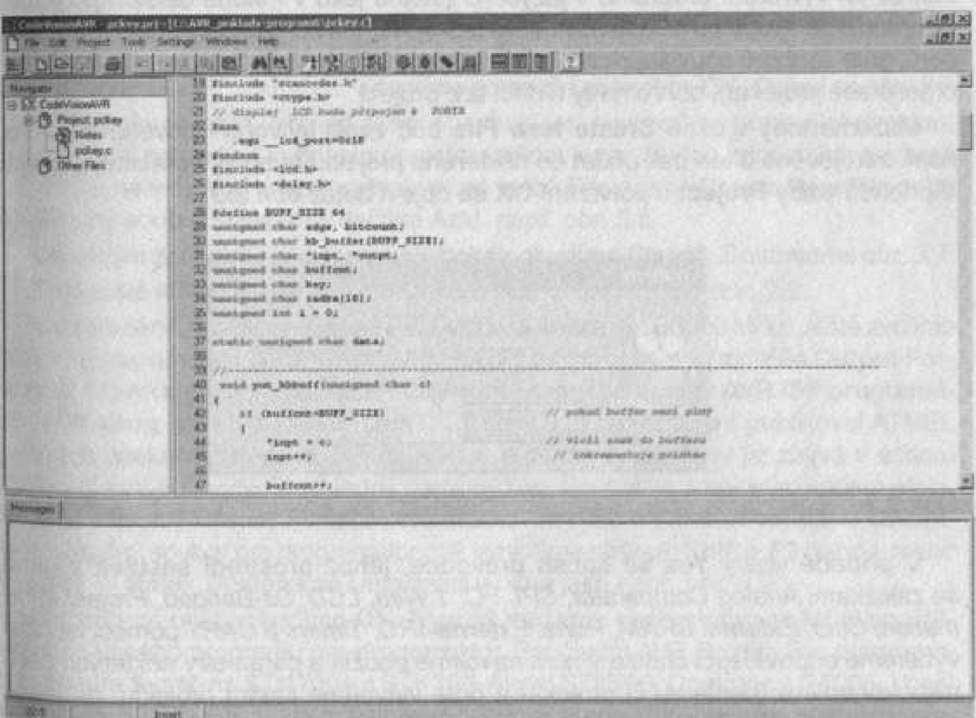
Instalační soubor pro plnou, komerční verzi je umístěn v souboru CVAVR.zip. K provozování této verze je potřeba soubor licence.dat, který lze získat od výrobce programu či distribujících firem po zaplacení 90 $. To jistě není velká částka pro firmu vyvíjející aplikace s mikrokontroléry AVR. (Chceme-li se o nějakém software před zakoupením přesvědčit, že je opravdu dobrý, stačí se obvykle rozhlédnout po internetu, zda nějakému hackerovi stál za námahu, zvláště jde-li o program, u kterého se nepředpokládá tak široké rozšíření jako např. MS Office, hry či samotný OS Windows).

V případě, kdy tvoříme v jazyce C nějakou AVR aplikaci pro nekomerční použití a narazili bychom na omezení délky kódu v CodeVisionAVR můžeme použít free kompilátor GCC pro AVR, který může překládat zdrojové kódy v C bez omezení délkou. Proto jsem zařadil krátký popis použití tohoto překladače, který ve spojení s **AVR Edit** tvoří docela slušný vývojový prostředek s IDE. Rovněž tento free překladač spolu s instalačkami AVR Edit je součástí CD.

3

VÝVOJOVÉ PROSTŘEDÍ (IDE) CodeVisionAVR C

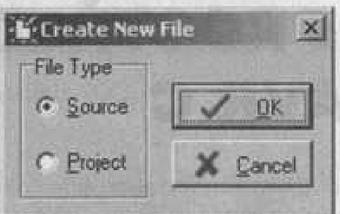
Integrované vývojové prostředí (IDE) je windowsovským programovým prostředím pro editaci zdrojových kódů, překlad, spojování, debugging atd., stejně jako pro spouštění průvodce či programátoru. Je napsáno jako klasický windowsovský program, takže jeho ovládání je intuitivní a řada akcí jako otevření či zavření souboru, jeho editace, tisk či nastavování vlastního prostředí odpovídá běžným konvencím programů běžících pod windows. Zvládne je každý uživatel zvyklý pracovat



Obr. 3.1 *Ovládací a nastavovací prvky v základním okně*

s tímto operačním systémem, stejně jako instalaci programu. Podrobný popis instalace i vlastního IDE je součástí **Uživatelského manuálu**, který najdete např. na CD. Téměř veškeré ovládací a nastavovací prvky a menu najdeme v základním okně *obr. 3.1*.

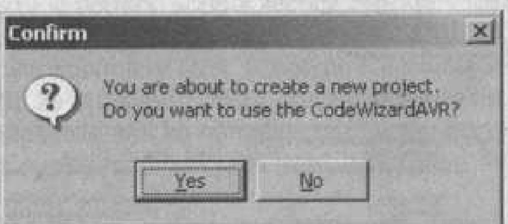
Popíšeme si činnost ovládacích prvků, které jsou specifické pro vývojové nástroje *obr. 3.2*. Výběrem v menu **File → New** se vyvolá okno **Create New File**.



Obr. 3.2

V něm zvolíme, zda chceme vytvořit nový soubor či nový projekt. Protože čtenář je zajisté obeznámen alespoň se základy programování v jazyce C, jen připomeneme, že při vytváření programů v jazyce C (stejně jako v mnoha dalších jazycích) se pracuje se skupinou několika souborů (zdrojové, hlavičkové, obj, knihovních souborů, dále souborů souvisejících s linkováním, používáním programu make i vlastní organizace projektu), dohromady tvořící tzv. projekt.

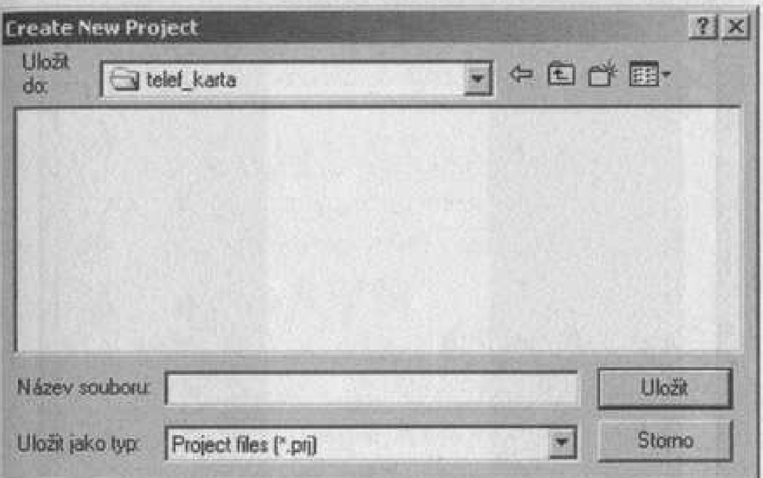
Můžeme tedy v okně **Create New File** buď zvolit vytvoření nového souboru, např. zdrojového a ten pak přidat do některého projektu nebo vytvořit nový projekt. V případě volby **Project** a potvrzení **OK** se objeví dotaz *obr. 3.3*.



Obr. 3.3

V případě volby **Yes** se spustí průvodce, jehož prostředí sestává z okna se záložkami *Analog Comparator, SPI, I²C, 1 Wire, LCD, Bit-Banged, Project Information, Chip, External SRAM, Ports, External IRQ, Timers* a *UART* pomocí nichž si vybereme odpovídající okno a v něm navolíme použití a parametry některých částí mikrokontroléru (periferie) či projektu a poté vytvoříme kostru projektu, do jehož zdrojových kódů doplníme nějaký vlastní kód. Tento postup si ukážeme při tvorbě ilustračních programů v kap. 7.

Další možností je napsat si celý kód sami. V tom případě zvolíme výběr **No**, a pak následuje dotaz na název a umístění projektu *obr. 3.4*.



Obr. 3.4

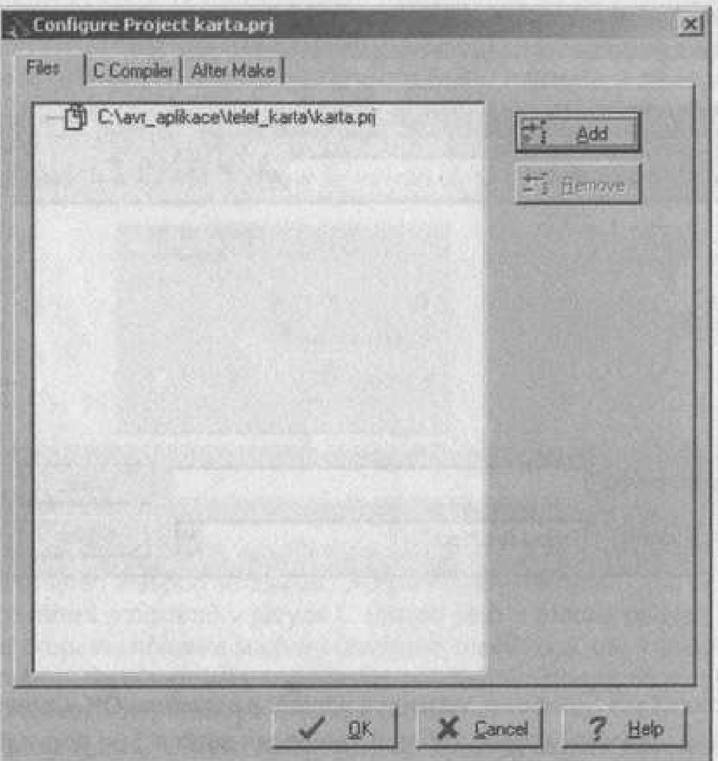
Vyplníme `Název souboru`, vybereme adresář a potvrdíme **OK**. Objeví se *obr. 3.5*.

Náš nově vytvořený projekt zatím obsahuje jen soubor *.prj popisující strukturu a organizaci projektu. Do projektu musíme přidat jeden či více zdrojových souborů *.C. Přidáme tedy buď soubor vytvořený po volbě **File** v okně **Create New File** nebo nějaký jiný soubor stisknutím tlačítka **Add**, např. *obr. 3.6*.

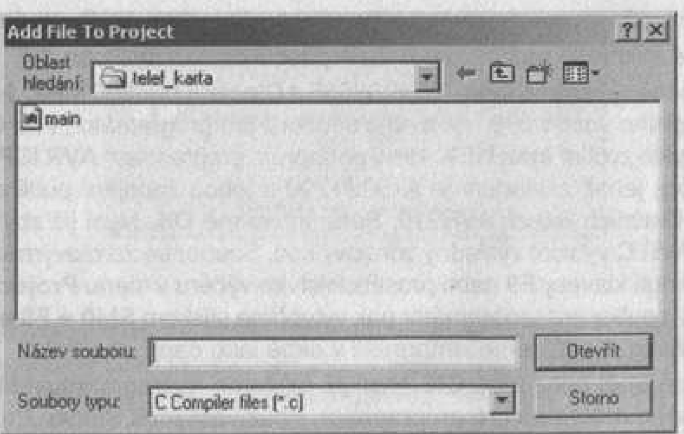
Obvyklým způsobem vybereme soubor a potvrdíme **Otevřít**. Dostaneme *obr. 3.7*.

Poté ještě klikneme na záložku **C Compiler** a dostaneme *obr. 3.8*.

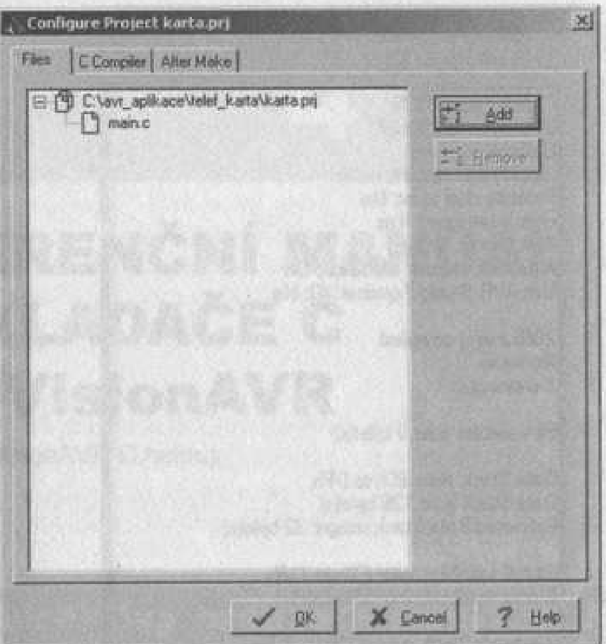
V tomto okně vybereme **Chip**: AT90S8535 a **Clock**: 8,000000 MHz. Ještě zvolíme formát výstupního souboru, tj. vstupního souboru pro programátor, **File Output Format(s):** Obvykle zvolím **Intel HEX**, který podporuje programátor **AVR ISP programátor AVR prog**, jehož základem je AT90S1200 a jehož zapojení publikoval ATMEL ve svých aplikačních listech AVR910. Poté potvrdíme **OK**. Nyní již zbývá v editoru CodeVisionAVR C vytvořit výsledný zdrojový kód. Soubor se zdrojovým kódem přeložíme po stisknutí klávesy F9 nebo prostřednictvím výběru v menu **Project → Compile** a výsledný soubor pro programátor pak vytvoříme stiskem **Shift + F9** nebo v menu **Project → Make**. Dostaneme i informaci v okně jako např. *obr. 3.9*
a samozřejmě opět potvrdíme **OK**. Nyní již můžeme naprogramovat MCU spuštěním příslušného programu pro programátor. Používám AVR Studio. Pro programátory **Kanda Systems STK200+** a STK300, Atmel STK500, Dontronics DT006, Vogel Elektronik VTEC-ISP či MicroTronics ATCPU a Mega2000 lze jako obslužný program programátoru přímo použít CodeVisionAVR C.



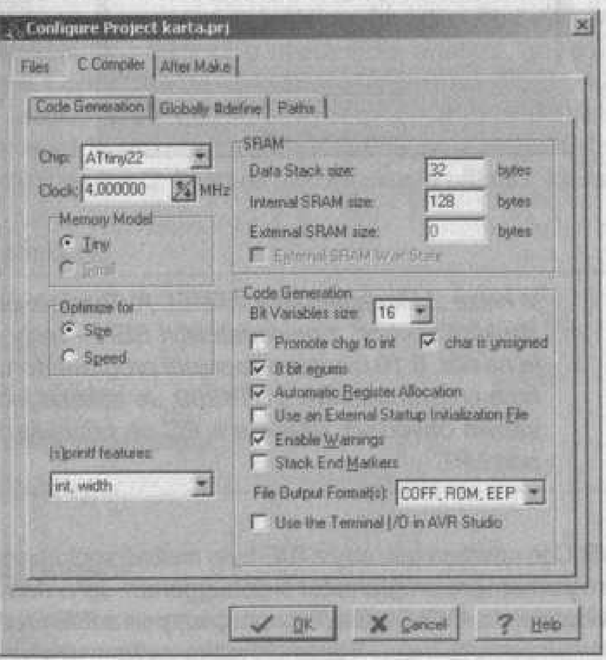
Obr. 3.5



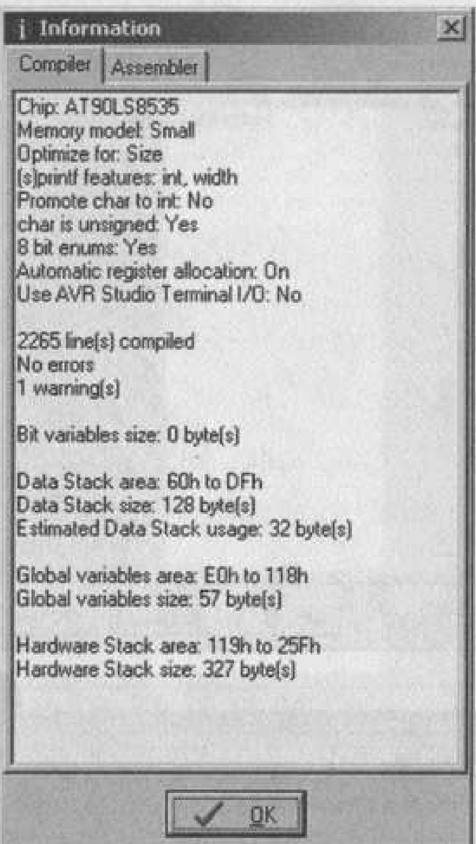
Obr. 3.6



Obr. 3.7



Obr. 3.8



Obr. 3.9

Poznámka: *V knize „Mikrokontroléry ATMEL AVR popis procesorů a instrukční soubor" z nakladatelství BEN – technická literatura je na obr. 8.10 uvedeno zapojení programátoru kompatibilního s programátorem s STK200. Je osazen jediným integrovaným obvodem 74244 a k PC se připojuje přes paralelní port LPT.*

CodeVisionAVR C je navržen tak, aby z IDE bylo možné spolupracovat s dalšími programy, např. programem pro programátor či debuggerem. Jako debugger se předpokládá použití debuggeru z AVR Studia. Tyto programy se z IDE prostředí CodeVisionAVR C vyvolávají pomocí menu **Tools**. Předtím ovšem výběrem v **Settings** musíme nastavit cesty k umístění těchto programů.

4

REFERENČNÍ MANUÁL PŘEKLADAČE C CodeVisionAVR

(podle CodeVisionAVR C helpu)

4.1 Direktivy preprocesoru a pragmy

Překlad zdrojového kódu napsaného v jazyce C se provádí ve skutečnosti ve dvou fázích (krocích). V první fázi se ještě neprovádí vlastní vyhodnocování příkazů jazyka C (tj. lexikální, syntaktická a sémantická analýza, generování výsledného kódu a případně i optimalizace výsledného kódu), ale jde o překlad příkazů tzv. *preprocesoru* jazyka C. Příkazům preprocesoru se říká *direktivy preprocesoru a pragmy*. Teprve až budou všechny zpracovány, může dojít k vlastnímu překladu zdrojového kódu v jazyce C. Existuje několik skupin příkazů, které si v této kapitole probereme:

- vkládání souborů,
- konstanty,
- makropříkazy,
- podmíněný překlad,
- vložení kódu v assembleru,
- pragma direktivy.

4.1.1 Preprocesor

Pomocí preprocesoru jsme schopni modifikovat obsah zdrojového kódu programu v jazyce C. Úkolem preprocesoru není tady překlad zdrojového kódu v jazyce C. Direktivy preprocesoru se nejčastěji používají pro zlepšení přenositelnosti zdrojové-

ho kódu mezi různými překladači a také umožňují zpřehlednit a zčitelnit samotný kód. Jazyk ANSI C definuje následující množinu klíčových slov pro zadávání příkazů preprocesoru:

#define	**#elif**	**#else**	**#endif**
#if	**#ifdef**	**#error**	**#include**
#line	**#unrer**	**#ifndef**	**#pragma**

CodeVisionAVR C překladač podporuje všechny tyto direktivy, i když v dokumentaci (helpu) jsou výslovně uvedené jen některé z nich. Navíc obsahuje ještě direktivy **#asm** a **#endasm**.

Vkládání souborů

Direktiva **#include** se používá k vložení obsahu libovolného souboru (většinou hlavičkového) do našeho zdrojového souboru tak, jako by byl nedílnou součástí zdrojového souboru (u CodeVisioAVR je omezení na max. 16 souborů). Jméno vkládaného souboru se může nacházet buď v špičatých závorkách, nebo v uvozovkách.

Příklad:

```
#include <studio.h>
#include "MojeDekl.h"
```

V prvním případě je vložen hlavičkový soubor `studio.h` nacházející se mezi standardními hlavičkovými soubory jazyka C, většinou se nacházející v podadresáři inc vytvořeném při instalace CodeVisionAVR C. V IDE CodeVisionAVR C je jejich umístění součástí projektu a nastavuje se v menu **Configure Project → C Compiler → Paths**.

V druhém případě je hlavičkový soubor `MojeDekl.h` vložen z aktuálního adresáře a jde o programátorem vytvořený hlavičkový soubor.

Konstanty a makropříkazy

Nejjednodušším příkazem preprocesoru je direktiva **#define** umožňující přiřadit jméno konstantám a řetězcům.

Příklad:

```
#define ALFA 0xff
```

Tento příkaz definuje symbolickou konstantu ALFA pro označení hodnoty 0xff.

Preprocesor jazyka C nahradí před kompilací v zdrojovém textu posloupnost znaků ALFA posloupností znaků 0xff. Makropříkazy mohou mít také parametry, obdobně jako funkce. Rozdíl mezi parametrem funkce a makra spočívá v určení typu. Parametry makropříkazů nejsou definovány s určením typu, protože nejsou vyhodnocovány při překladu a jejich rozvoj probíhá na textové úrovni. Jasnější to bude z následujícího příkladu:


```
#define SUM(a,b) a+b
/* následující úsek kódu bude nahrazen kódem int i=2+3; */
int i=SUM(2,3);
```

Operátory # a ##

Při psaní makropříkazů můžeme použít dva operátory, které mění způsob vyhodnocování parametrů. Při použití operátoru # před jménem parametru při jeho použití v těle makra dojde k převedení hodnoty parametru na textový řetězec.

Příklad:

```
#define PRINT_MESSAGE(t) printf(#t)
/* ......... */
/* následující úsek kódu bude nahrazen kódem printf("ahoj"); */
PRINT_MESSAGE(ahoj) ;
```

Druhým operátorem je dvojice znaků ## sloužící ke spojení dvou parametrů.

Poznámka: *Definice makra může být rozšířena na nový řádek použitím *

Příklad:

```
  #define MESSAGE "Toto je velmi \
dlouhý text ... "
```

Zrušení definice makra

Definici makra lze zrušit pomocí direktivy **#undef**. Tato direktiva se používá, např. když potřebujeme zajistit, aby použití identifikátoru standardní funkce v programu znamenalo volání funkce a nikoli použití makra. Můžeme ji také použít, když potřebujeme znovu jinak definovat existující makro.

Příklad:

```
#undef ALFA
```

Podmíněný překlad

Použitím direktiv preprocesoru můžeme vynechat nebo modifikovat části zdrojového textu. Pro podmíněný překlad, jak se programové modifikaci říká, jsou k dispozici následující direktivy: **#ifdef, #ifndef, #else** a **#endif**.

Jejich syntaxe je:

```
#ifdef alfa
//skupina příkazů 1
#else
//skupina příkazů 2
#endif
```

Je-li 'alfa' definováno jako jméno makra, pak výraz **#ifdef** se vyhodnotí (relační výraz) jako *true* a *skupina příkazů 1* bude zpracována překladačem. V opačném případě bude překladač zpracovávat *skupinu příkazů 2*. Direktiva **#else** a *skupina příkazů 2* jsou nepovinné (jsou volitelné, optional). Když 'alfa' není definováno, direktiva **#ifndef** se vyhodnotí jako true. Zbývající část má stejnou syntaxi jako v případě **#ifdef**.

Pro podmíněný překlad mohou být rovněž použity direktivy **#if, #elif, #else** a **#endif**.

```
#if vyraz1
//skupina příkazů 1
#elif vyraz2
//skupina příkazů 2
#else
//skupina příkazů 3
#endif
```

Bude-li `vyraz1` vyhodnocen jako *true*, bude proveden překlad *skupiny příkazů 1*.

Bude-li `vyraz2` vyhodnocen jako *true*, bude proveden překlad *skupiny příkazů 2*.

V opačném případě se bude překládat *skupina příkazů 3*.

Direktiva **#else** a *skupina příkazů 3* jsou nepovinné (jsou volitelné, optional).

CodeVisionAVR C obsahuje následující předdefinovaná makra:

__CODEVISIONAVR__ číslo verze a revize překladače vyjádřené jako celé číslo, např. pro V1.23.6 to bude číslo 1236.

__LINE__ číslo aktuálního řádku překládaného souboru.

__FILE__ aktuální překládaný soubor.

__TIME__ aktuální čas ve formátu hh:mm:ss.

__DATE__ aktuální datum ve formátu mmm dd yyyy.

_CHIP_ATXXXXX_ kde ATXXXXX je typ obvodu popsaný velkými písmeny, určený výběrem v menu Project → Configure → C Compiler → Chip option.

_MCU_CLOCK_FREQUENCY_ je kmitočet hodin AVR MCU nastavený v menu Project → Configure → C Compiler → Clock option, celočíselným výrazem v Hz.

_MODEL_TINY_ když je program překládán s použitím paměťového modelu TINY.

_MODEL_SMALL_ když je program překládán s použitím SMALL paměťového modelu.

_OPTIMIZE_SIZE_ když je program překládán s použitím optimizace na velikost výsledného kódu.

_OPTIMIZE_SPEED_ když je program překládán s použitím optimalizace na rychlost přeloženého kódu.

_UNSIGNED_CHAR_ když je povolena volba *char unsigned* překladače nebo je *použita #pragma uchar+*.

_8BIT_ENUMS_ když je povolena volba *8bit enums* překladače nebo je **použita #pragma 8bit_enums+**.

Použitím direktivy **#line** se mění hodnoty v makrech **__LINE__** a **__FILE__**.

Její syntaxe je:

```
#line integer_constant ["jmeno_souboru"]
```

Příklad:

```
/* nastaví __LINE__ na 50 a
   __FILE__ na "file2.c" */
#line 50 "file2.c"
/* nastaví __LINE__ na 100 */
#line 100
```

Direktiva #error může být použita k přerušení překladu a zobrazení zprávy o chybě.

Její syntaxe:

```
#error error_message
```

Příklad:

```
#error Toto je chyba!
```

Můžeme použít direktivu #pragma warn k povolení či zakázání upozornění.

Příklad:

```
/* upozornění jsou zakázána */
#pragma warn-
// nějaký kód
/* upozornění jsou povolena */
#pragma warn+
```

Optimalizace překládaného kódu může být zapnuta či vypnuta direktivou **#pragma**. Tato direktiva musí být umístěna na začátku zdrojového kódu.

Příklad:

```
/* zapnutí optimalizace */
#pragma opt+
```

nebo

```
/* vypnutí optimalizace */

#pragma opt-
```

Automatické ukládání a vyvolávání obsahu registrů R0, R1, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31 a SREG, při přerušeních může být zapnuto či vypnuto pomocí direktivy #pragma savereg.

Příklad:

```
/* vypnutí uchovávání obsahu registrů */
#pragma savereg-

/* obsluha přerušení */
interrupt [1] void my_irq(void) {
/* nyní se uchovává pouze obsah registrů jejichž
   úschova je zajištěna v obslužném kódu,
   např R30, R31 a SREG */
#asm
    push r30
    push r31
    in r30,SREG
    push r30
#endasm
/* umístění nějakého kódu v jazyce C */
/*.... */

/* nyní se obnoví obsah SREG, R31 a R30 */
#asm
    pop r30
    out SREG,r30
    pop r31
    pop r30
#endasm
}
/* zapnutí uchovávání obsahu registrů při dalších přerušeních */

#pragma savereg+
```

Přednastavené (default) je automatické uchovávání obsahu registrů při obsluze přerušení.

4.2 Komentáře

Komentáře v Céčku začínají lomítkem a hvězdičkou /* a končí kombinací těchto znaků v opačném pořadí. Všechny znaky mezi těmito kombinacemi znaků budou překladačem ignorovány. Komentáře mohou být několikařádkové. Není možné používat vnořené komentáře (komentář v komentáři).

Příklad:

```
/* Toto je komentář */
/* Toto je
   několikařádkový komentář */
```

V CodeVisionAVR C, podobně jako v C++, mohou komentáře začínat též dvěma lomítky //. Komentář, který po nich následuje, končí na konci řádku.

Příklad:

```
// Toto je také komentář
```

4.3 Klíčová slova

Klíčová slova jsou rezervovaná slova, která mají pro překladač jazyka C speciální význam a nesmí být použita jako identifikátory, tedy jména funkcí, proměnných nebo symbolických konstant. Klíčová slova jsou

break	**bit**	**case**	**char**
continue	**default**	**do**	**eeprom**
else	**enum**	**extern**	**flash**
float	**goto**	**if**	**int**
interrupt	**long**	**register**	**sizeof**
sfrb	**sfrw**	**static**	**struct**
typedef	**union**	**while**	

Klíčová slova **enum**, **register** a **typedef** nejsou zatím využita, ale jsou rezervovaná pro budoucí verze překladače CodeVisionAVR C.

4.4 Identifikátory

Identifikátory jsou jména dávaná proměnným, funkcím, návěštím nebo jiným objektům. Identifikátory mohou obsahovat písmena (A ... Z, a ... z), číslice (0–9) nebo znak podtržítka (_). Prvním znakem nesmí být číslice. Všechna jména v jazyce C jsou citlivá na výšku písmen. Proto např. variable1 není totéž co Variable1.

Délka identifikátorů rozlišovaných CodeVisionAVR C je až 32 znaků.

4.5 Datové typy

Následující seznam obsahuje datové typy podporované CodeVisionAVR C Compiler, jejich velikost a rozsah možných hodnot *tab. 4.1*.

Datový typ bit je podporovaný pouze pro globální proměnné. Použitím direktivy **#pragma uchar+** nebo výběrem v menu **Project → Configure → C Compiler → char → unsigned** se nastaví rozsah typu **char** na 0 až 255.

4.6 Konstanty

KONSTANTA je číslo, znak nebo řetězec znaků, které můžeme použít v programu jako hodnotu. CodeVisionAVR C podporuje následující konstanty:

Konstanty typu **integer** nebo **long integer** mohou být napsány v dekadickém tvaru (např. 1234), v hexadecimálním tvaru s prefixem 0x (např. 0xff) nebo v oktalovém tvaru s prefixem 0 (např. 0777).

Long integer konstanty mající sufix L (např. 99L).

Floating point constanty mající F (např. 1234F).

Tab. 4.1 Velikost a rozsah hodnot

Typ	velikost (Bits)	rozsah
bit	1	0, 1
char	8	–128 až 127
unsigned char	8	0 až 255
signed char	8	–128 až 127
int	16	–32768 až 32767
short int	16	–32768 až 32767
unsigned int	16	0 až 65535
signed int	16	–32768 až 32767
long int	32	–2147483648 až 2147483647
unsigned long int	32	0 až 4294967295
signed long int	32	–2147483648 až 2147483647
float	32	±1,175e-38 až ±3,402e38
double	32	±1,175e-38 až ±3,402e38

Znakové konstanty musí být uzavřeny mezi apostrofy. Např. 'a'.

Řetězcové konstanty musí být uzavřeny v uvozovkách. Např. „Nazdar svete".

Když použijete řetězec mezi uvozovkami jako parametr funkce, bude automaticky tento řetězec považován za konstantu a bude umístěn v paměti FLASH.

Příklad:

```
/ * tato funkce zobrazuje řetěz umístěný v RAM */
void display_ram(char *s) {
/*....... */
}
/ * tato funkce zobrazuje řetěz umístěný v FLASH */
void display_flash(char flash *s) {
/*....... */
}
void main(void) {
/* nelze použít !!! */
```

```
/* protože funkce nepoužívá string */

/*umístěný v RAM, ale string "ahoj svete" */
/* což je konstanta umístěná v FLASH */
display_ram("ahoj svete");

/* lze použít !!! */
/* funkce používá string umístěný v FLASH */
display_flash("ahoj svete");
}
```

Konstanty mohou tvořit až 8rozměrné pole (array). Konstanty jsou ukládány v paměti FLASH, proto je třeba použít klíčové slovo **flash**.

Konstantní výrazy jsou automaticky vyčísleny během překladu.

Příklad:

```
flash int integer_constant=1234+5;
flash char_constant='a';
flash long long_int_constant1=99L;
flash long long_int_constant2=0x10000000;
flash int integer_array1[]={1,2,3};
/*první dva prvky budou 1 a 2,
  zbytek bude 0 */
flash int integer_array2[10]={1,2};
flash int multidim_array[2,3]={{1,2,3},{4,5,6}};
flash char string_constant[]="Tento řetězec je konstanta";
```

Konstanty nemohou být definovány uvnitř funkcí.

4.7 Proměnné

Proměnné jsou paměťová místa, do kterých se ukládají data jistého typu. K těmto paměťovým místům přistupujeme pomocí jmen, abychom mohli přečíst data, která se tam nacházejí nebo tam zapsat nová data. Proměnné (stejně jako funkce či definice typů) musí být deklarovány a zpravidla též definovány. Proměnné mohou být globální (dosažitelné všemi funkcemi v programu) nebo lokální (dosažitelné pouze uvnitř funkce v níž jsou deklarovány).

Pokud nejsou inicializovány v kódu programu, budou globální proměnné automaticky nastaveny na 0.

Lokální proměnné nejsou automaticky inicializovány při spuštění programu.

Příklad:

```
/* deklarace globálních proměnných */
char a;
int b;
/* a inicializace */
long c=1111111;

void main(void) {
/*deklarace lokálních proměnných */
char d;
int e;
long f;

/* a inicializace */
f=22222222;
}
```

Proměnné mohou být skládány v pole mající až 8 dimenzí. Pole je shrnutím více dat stejného typu do jedné proměnné. S jednotlivými prvky polí pracujeme prostřednictvím indexů, které zapisujeme do hranatých závorek za identifikátor pole. První prvek pole má vždy index 0. Prvkům pole (globálních proměnných), které explicitně neinicializujeme, se automaticky přiřadí hodnota 0. Prvky lokálních polí nejsou automaticky inicializovány.

Příklad:

```
/* všechny prvky pole budou 0 */
int global_array1[32];

/* pole je automaticky inicializováno */
int global_array2[]={1,2,3};
int global_array3[4]={1,2,3,4};
char global_array4[]="Toto je retez";

/* pouze první 3 prvky jsou inicializovány
  zbývajících 29 bude nulových 0 */
int global_array5[32]={1,2,3};

/* vícerozměrné pole */
int multidim_array[2,3]={{1,2,3},{4,5,6}};

void main(void) {
int local_array1[3];
char local_array2[7];
/* inicializace */
local_array1[0]=11;
local_array1[1]=22;
local_array1[2]=33;
```

```
/* inicializace lokálního pole */
local_array2[0]='S';
local_array2[1]='t';
local_array2[2]='r';
local_array2[3]='i';
local_array2[4]='n';
local_array2[5]='g';
local_array2[6]=0;
}
```

Jiný způsob inicializace lokálních polí je předveden v následujícím příkladu, Který používá funkci memcpyf:

```
/* obsahuje funkci memory copy */
#include <string.h>

/* ukládá do FLASH hodnoty použité při inicializaci */
flash int init_array[5]={1,2,3,4,5};

void main(void) {
/* deklaruje lokální pole které musí být inicializované */
int local_array[5];

/* inicializace lokálního pole */
/* obsahem init_aray */
memcpyf(local_array,init_array,5*sizeof(int));

}
```

Lokální proměnné, které si mají uchovávat své hodnoty při různých volání funkce musí být deklarované jako static.

Příklad:

```
int alfa(void) {
/* deklaruje a inicializuje statickou proměnnou */
static int n=1;
return n++;
}
void main(void) {
int i;

/* funkce vracející hodnotu 1 */
i=alfa();
```



```
/* funkce vracející hodnotu 2 */
i=alfa();
}
```

Nejsou-li explicitně inicializovány, budou static proměnné automaticky nastaveny na 0.

Proměnné, které jsou deklarované v jiném souboru musí být označeny klíčovým slovem extern.

Příklad:

```
extern int xyz;

/* zahrnuje soubor, který obsahuje
  definici proměnné xyz */
#include <file_xyz.h>
```

Všechny globální proměnné jsou umístěné v části Global Variables paměťového prostoru RAM.

Všechny lokální proměnné jsou ukládány v dynamicky alokovaném prostoru datového zásobníku paměťového prostoru RAM.

Globální proměnné mohou být ukládány i do konkrétních míst v RAM určených pomocí operátoru @.

Příklady:

```
/* celočíselná proměnná "a" je ukládána
  v RAM na adrese 80h */
int a @0x80;

/* struktura "alfa" je ukládána
  v RAM na adrese 90h */
struct x {
         int a;
         char c;
         } alfa @0x90;
```

Bitové proměnné

Bitové proměnné jsou speciální globální proměnné umístěné v registrech R2 až R15. Tyto proměnné jsou deklarovány za použití klíčového slova **bit**.

Příklad:

```
/* deklarace a inicializace */
bit alfa=1; /* bit0 of R2 */
bit beta; /* bit1 of R2 */

void main(void)
{
if (alfa) beta=!beta;

/*........ */
}
```

Alokace paměti pro bitové proměnné se provádí v případě deklarace počínajíc bitem 0 registru R2, pak bit 1 R2, bit 3 R3 ... bit 7 R15.

Může být deklarováno maximálně 112 bitových proměnných. Nejsou-li explicitně inicializovány, budou globální bitové proměnné automaticky nastaveny na 0.

Při vyhodnocování výrazů jsou bitové proměnné automaticky rozšířené na unsigned char.

Struktury

Struktury jsou uživatelem definované kolekce pojmenovaných prvků. Shrnují do jedné proměnné větší množství dat, která navíc mohou být různých datových typů. Členy struktur mohou být jen podporované datové typy nebo ukazatele na ně. Struktury jsou definované pomocí klíčového slova **struct**.

Syntaxe je:

```
[<modifikátor_paměti>] struct jméno_struktury> {
        [<typ> <jméno_proměnné[,jméno_proměnné,...]>];
        [<typ> < jméno_proměnné [,jméno_proměnné,...]>];
        ...
        } [<proměnné_struktury>];
```

Příklad:

```
/* globální struktura umístěná v RAM */
struct ram_structure {
        char a,b;
        int c;
        char d[30],e[10];
        char *pp;
        } sr;
```



```
/* globální konstantní struktura umístěná v FLASH */
flash struct flash_structure {
        int a;
        char b[30],c[10];
        } sf;

/* globální struktura umístěná v EEPROM */
eeprom struct eeprom_structure {
        char a;
        int b;

        char c[15];
        } se;

void main(void) {
/*lokální struktura */
struct local_structure {
        char a;
        int b;
        long c;
        } sl;

/*............. */

}
```

Velikost paměťového prostoru alokovaného pro strukturu se rovná součtu velikostí paměťového prostoru přidělených všem jeho členům.

Pro struktury umístěné v FLASH a EEPROM platí několik omezení. Je to dané tím, že ukazatele musí být vždy umístěné v RAM, proto se nemohou používat u struktur umístěných v FLASH a EEPROM.

Protože Atmel AVRASM Assembler pro jednotlivé byty definované pomocí DB v FLASH ve skutečnosti zabírá 2 byty, překladač CodeVisionAVR C nahrazuje členy typu **char** struktur umístěných v FLASH členy typu **int**.

Ze stejného důvodu to také rozšíří velikost pole prvků typu char, členů takových struktur, na sudou hodnotu.

Následující příklad ukazuje jak inicializovat a přistupovat ke globálním strukturám umístěným v EEPROM:

```
/* globální struktura umístěná v EEPROM */
eeprom struct eeprom_structure {
```

```
        char a;
        int b;
        char c[15];
        } se[2]={{'a',25,"ahoj"},
        {'b',50,"svete"}};

void main(void) {
char k1,k2,k3,k4;
int i1,i2;

/* definuje ukazatel na strukturu */
struct eeprom_structure eeprom *ep;

/* přímý přístup ke členům struktury */
k1=se[0].a;
i1=se[0].b;
k2=se[0].c[2];
k3=se[1].a;
i2=se[1].b;
k4=se[1].c[2];

/* přístup ke členům struktury pomocí ukazatele */

ep=&se; /* inicializuje ukazatel na adresu struktury */
k1=ep->a;
i1=ep->b;
k2=ep->c[2];
++ep; /* inkrementuje ukazatel */
k3=ep->a;
i2=ep->b;
k4=ep->c[2];

}
```

Struktury mohou být parametrem funkcí nebo návratovou hodnotou funkcí jenom prostřednictvím ukazatelů.

Příklad:

```
struct alpha  {
          int a,b,c;
          } s={2,3};

/* definuje funkci */
struct alpha *sum_struct(struct alpha *sp) {
/* člen c=člen a + člen b */

sp->c=sp->a + sp->b;

/* vrací ukazatel na strukturu */
return sp;
}

void main(void) {
int i;
/* s->c=s->a + s->b */
/* i=s->c */
i=sum_struct(&s)->c;

}
```

Ve strukturách nejsou implementovány bitová pole.

Uniony

Union je datový typ, který má několik prvků (složek), avšak programátor může používat vždy jen jeden z nich. Tyto prvky leží v paměti na stejném místě „přes sebe". Velikost proměnné typu union odpovídá velikosti nejvyššího prvku. Oproti strukturám šetří uniony místo v paměti v případě, kdy je potřeba pouze jeden prvek, avšak programátor předem neví který.

Členy unionů mohou být libovolného podporovaného datového typu, pole těchto typů či ukazatelů na ně.

Uniony jsou definované pomocí klíčového slova **union**.

Syntaxe je:

```
union  <jméno_unionu> {
       [<typ>  <jméno_proměnné[,jméno_proměnné,...]>];
       [<typ> < jméno_proměnné  [,jméno_proměnné,...]>];
       ...
       } [<proměnné unionu>];
```

Uniony jsou vždy umístěné v RAM. Ke členům unionu se přistupuje stejně jako ke členům struktury.

Příklad:

```
/* deklarace unionu */
union alpha {
            unsigned char lsb;
            unsigned int word;
            } data;

void main(void) {
unsigned char k;

/* definuje ukazatel na union */
union alpha *dp;

/* přímý přístup ke členům unionu */
data.word=0x1234;
k=data.lsb; /* get the LSB of 0x1234 */

/* přístup ke členům unionu pomocí ukazatele */
dp=&data; /* inicializace ukazatele na adresu unionu */
dp->word=0x1234;
k=dp->lsb; /* dostane LSB z f 0x1234 */
}
```

Uniony mohou být parametrem funkce nebo její návratovou hodnotou jenom prostřednictvím ukazatelů.

Příklad:

```
#include <stdio.h> /* printf */

union alpha {
            unsigned char lsb;
            unsigned int word;
            } data;

/* definice funkce */

unsigned char low(union alpha *up) {
/* vrací LSB slova */
return up->lsb;
}
```



```
void main(void) {
data.word=0x1234;
printf(" LSB %x je %2x",data.word,low(&data));

}
```

Výčtové typy

Výčtové typy jsou datové typy, jejichž možné hodnoty jsou určeny pomocí celočíselných konstant. Používají se jako příznaky, k vyjádření hodnot, které se vyskytují pouze v malém počtu (dny v týdnu) ap. Mohou se využívat k poskytnutí mnemonických identifikátorů množiny celočíselných hodnot. Používá se přitom klíčové slovo **enum**.

Syntaxe:

```
[<modifikátor_paměti>] enum [<jméno_výčtového_typu>] {
    [<jméno_položky[[=hodnota_konstanty], jméno_položky,...]>]}
    [<proměnná>];
```

Příklad:

```
/* výčtové typy - konstanty inicializované takto :
  sunday=0, monday=1, tuesday=2,..., saturday=6 */
enum days {

        sunday, monday, tuesday, wednesday,
        thursday, friday, saturday} days_of_week;

/* výčtové typy - konstanty inicializované takto ::
  january=1, february=2, march=3,..., december=12 */
enum months {
          january=1, february, march, april, may, june,
          july, august, september, october, november, december}
          months_of_year;

void main {
/* proměnná days_of_week inicializovaná celočíselnou /*
/* hodnotou value 6 */

days_of_week=saturday;
}
```

Výčtové typy mohou být umístěné v SRAM nebo EEPROM. K určení umístění do EEPROM se musí použít klíčové slovo **eeprom**.

Příklad:

```
eeprom enum days       {
                       sunday, monday, tuesday, wednesday,
                       thursday, friday, saturday} days_of_week;
```

4.8 Uživatelem definované datové typy

Uživatelem definované datové typy jsou deklarované užitím klíčového slova **typedef**. Již dříve jsme se seznámili s definicí nových datových typů pomocí klíčových slov **enum**, **struct**, **union**. Deklarace **typedef** se od nich liší tím, že nezavádí nový typ, ale nové pojmenování pro existující typ.

Syntaxe:

typedef [<modifikátor_paměti>] <definice typu> <identifikátor>;

Deklarace typedef vytvářejí zkratky pro komplikované typy.

Příklad:

```
/* definice typu */
typedef unsigned char byte;
typedef eeprom struct   {
                        int a;
                        char b[5];
                        } eeprom_struct_type;

/* deklarace proměnné */
byte alfa;

eeprom eeprom_struct_type struct1;
```

4.9 Typové konverze, přetypování

Aritmetické operace jsou definovány pouze na proměnných či konstantách stejného typu. Proto pokud jsou operandy nestejných typů, je nutné je převést na odpovídající typ. Pro takové případy existuje jednoduchá a krátká sada tzv. STANDARDNÍCH KONVERZÍ. Je-li třeba, jsou standardní konverze prováděny překladačem tak, aby byly provedeny před samotným vyhodnocením výrazu. Při standardních konverzích dochází vždy ke konverzi operandu z typu „nižšího" na typ „vyšší". Pojmem „nižší" se rozumí typ s menším rozsahem a „vyšší" pak typ s větším rozsahem hodnot a s vyšší přesností.

Překladač používá následující pravidla:

- pokud je některý z operandů **double**, druhý z operandů se konvertuje na **double** a výsledek je **double**,
- jinak pokud je některý z operandů **float**, druhý operand se konvertuje na **float** a výsledek je **float**,
- jinak pokud je některý z operandů **long int** nebo **unsigned long int**, pak druhý z operandů se konvertuje na tentýž typ a ten je i typem výsledku,
- jinak když některý z operandů je typu **int** nebo **unsigned int**, pak druhý z operandů se konvertuje na tentýž typ a ten je i typem výsledku,
- jinak když jeden z operandů je typu **char** nebo **unsigned char**, pak druhý z operandů se konvertuje na tentýž typ a ten je i typem výsledku.

Tedy typ **char** nebo **unsigned char** mají nejnižší prioritu.

Chceme-li vynutit konverzi hodnoty na námi zvolený typ (odporující těmto pravidlům), musíme použít *přetypování*. Pokud je takové přetypování možné, lze je explicitně provést pomocí přetypovacího operátoru *(typ)*. Přetypování lze použít např. jako ochranu před přetečením u 8bitového sčítání či násobení.

Příklad:

```
void main(void) {
int a,c;
long b;
/* proměnná b typu long integer bude přetypována na
  integer */
c=a+(int) b;
}
```

Poznámka: *Na rozdíl od kompilátorů C pro 16 či 32bitové CPU neplatí pro CodeVisionAVR C, že **char** resp. **unsigned char** jsou automaticky převáděny na **int**. Tento převod je v CodeVisionAVR C nastaven buď v menu **Project → Configure → C Compiler → Promote char to int** nebo užitím **#pragma promotechar+**.*

Příklad:

```
void main(void) {

unsigned char a=30;

unsigned char b=128;

unsigned int c;

/*  zde je vytvořen nesprávný výsledek, protože se provádí
    8bitové násobení jehož výsledek je 8bitový, což způsobí
    přetečení.
    Až po provedení násobení se 8bitový výsledek převede na
    unsigned int */
c=a*b;

/*  tady se přetypuje násobení na 16 bitů,
    vytváří 16bitový výsledek bez přetečení */

c=(unsigned int) a*b;

}
```

Překladač se pro následující operátory (popis operátorů viz následující kapitola) chová odlišně:

+=
-=
*=
/=
%=
&=
|=
^=
<<=
>>=

Pro tyto operátory je výsledek napsán zpět do operandu na levé straně (což musí být proměnná, nikoli konstanta). Proto vždy překladač převádí operátory na pravé straně na typ operandu na levé straně.

4.10 Operátory

Kompilátor CodeVisionAVR C podporuje následující operátory viz *tab. 4.2*:

Tab. 4.2 Přehled operátorů

<table>
<tr><td>+</td><td>sčítání</td></tr>
<tr><td>-</td><td>odčítání</td></tr>
<tr><td>*</td><td>násobení</td></tr>
<tr><td>/</td><td>dělení</td></tr>
<tr><td>%</td><td>modulo, zbytek po celočíselném dělení</td></tr>
<tr><td>++</td><td>prefixová inkrementace</td></tr>
<tr><td>--</td><td>prefixová dekrementace</td></tr>
<tr><td>=</td><td>přiřazení</td></tr>
<tr><td>==</td><td>rovno</td></tr>
<tr><td>~</td><td>bitový doplněk</td></tr>
<tr><td>!</td><td>logická negace</td></tr>
<tr><td>!=</td><td>není rovno</td></tr>
<tr><td><</td><td>menší</td></tr>
<tr><td>></td><td>větší</td></tr>
<tr><td><=</td><td>menší nebo rovno</td></tr>
<tr><td>>=</td><td>větší nebo rovno</td></tr>
<tr><td>&</td><td>bitový logický součin</td></tr>
<tr><td>&&</td><td>konjukce, logický součin</td></tr>
<tr><td>|</td><td>bitový logický součin</td></tr>
<tr><td>||</td><td>disjunkce, logický součin</td></tr>
<tr><td>^</td><td>bitový XOR, exklusivní OR</td></tr>
<tr><td>?:</td><td>podmíněný výraz</td></tr>
<tr><td><<</td><td>bitový posuv doleva</td></tr>
<tr><td>>></td><td>bitový posuv doprava</td></tr>
<tr><td>-=</td><td>odečte operandy a výsledek přiřadí levému operandu</td></tr>
<tr><td>+=</td><td>sečte operandy a výsledek přiřadí levému operandu</td></tr>
<tr><td>/=</td><td>dělí operandy a výsledek přiřadí levému operandu</td></tr>
<tr><td>%=</td><td>vypočte zbytek po celočíselném dělení a přiřadí výsledek levému operandu</td></tr>
<tr><td>&=</td><td>vypočte bitovou konjunkci a výsledek přiřadí levému operandu</td></tr>
<tr><td>*=</td><td>vynásobí aritmetické operandy a výsledek přiřadí pravému operandu</td></tr>
<tr><td>^=</td><td>bitová nonekvivalence a přiřazení výsledku levému operandu</td></tr>
<tr><td>|=</td><td>disjunkce po bitech a přiřazení výsledku levému operandu</td></tr>
<tr><td>>>=</td><td>posunutí bitů doprava a přiřazení výsledku levému operandu</td></tr>
<tr><td><<=</td><td>posunutí bitů doleva a přiřazení výsledku pravému operandu</td></tr>
</table>

4.11 Funkce

Můžeme použít prototyp funkce k *deklaraci funkce*.

Tyto deklarace obsahují informace o parametrech funkce.

Příklad:

```
int alfa(char par1, int par2, long par3);
```

Definice funkce může vypadat nějak takhle:

```
int alfa(char par1, int par2, long par3) {
/* zde jsou napsány nějaké příkazy */
}
```

(definice funkcí podle Kernighana a Ritchie není podporována, podporuje se jen způsob podle ANSI C).

Funkce v CodeVisionAVR C dostávají parametry pomocí zásobníku. **Funkce v CodeVisionAVR C vracejí hodnotu v registrech R30, R31, R22** a **R23** (LSB až MSB) a to v R30 je-li vrácená hodnota typu char či unsigned char, v R30 a R31 pro funkce vracející výsledek typu unsigned int a v R30, R31, R22 a R23 pro funkce vracející long nebo unsigned long.

4.12 Ukazatele

Ukazatel je proměnná, která obsahuje adresu datového objektu nebo funkce. Výhody používání ukazatelů (jako úspora paměti) i nevýhody (častý zdroj chyb způsobených programátorem) zná každý, kdo již alespoň trochu programoval v céčku. Podrobnosti najdeme v řadě učebnic i dalších publikací o C či C++. Ve většině případů jde o publikace o jazyku C implementovaném na počítačích PC či jiných počítačích s von Neumannovskou architekturou. AVR mikrokontroléry však mají **Harvardskou architekturu** s oddělenými adresovými prostory pro data (RAM), program (FLASH) a paměť EEPROM memory. Proto překladač implementuje tři typy ukazatelů.

K proměnným umístěným v RAM se přistupuje pomocí normálních ukazatelů.

Pro přístup ke konstantám umístěným v paměti FLASH, se použije klíčové slovo **flash**. Pro přístup k proměnným umístěným v EEPROM, se použije klíčové slovo **eeprom**. Třebaže tyto ukazatelé odkazují do různých paměťových prostorů, jsou vždy uložené v paměti RAM.

Příklad:

```
/* ukazatel na string umístěný v RAM */
char *ptr_to_ram="Tento řetězec je umístěný v RAM";

/* ukazatel na string umístěný v FLASH */
char flash *ptr_to_flash="Tento řetězec je umístěný v FLASH";

/* ukazatel na string umístěný v EEPROM */
char eeprom *ptr_to_eeprom="Tento řetězec je umístěný v
EEPROM";
```

Pro zlepšení efektivnosti přeloženého kódu implementuje CodeVisionAVR C dva paměťové modely. Paměťový model určuje, jaké ukazatele překladač použije, pokud nepředepíšeme explicitně něco jiného.

Paměťový model TINY používá 8 bitů pro uložení ukazatelů na proměnné umístěné v RAM. Tento paměťový model umožňuje přístup pouze k prvním 256 bytům RAM.

Paměťový model SMALL používá 16 bitů pro uložení ukazatelů na proměnné umístěné v RAM. Tento paměťový model umožňuje přístup až k 65 536 bytům RAM.

Chceme-li mít co nejkratší a nejrychlejší výsledný program, musíme se vždy pokusit použít paměťový model TINY. Ukazatele do paměťových prostorů FLASH a EEPROM vždy používají 16 bitů. Protože ukazatele na paměť FLASH jsou široké 16 bitů, velikost přeloženého binárního kódu nesmí překročit 64 K na což je potřeba pamatovat u MCU řady ATmega.

Ukazatele mohou být uspořádány do polí až 8 dimenzionálních.

Příklad:

```
/* deklarace a inicializace globálního pole ukazatelů na
řetězce umístěné v RAM */
char *strings[3]={"One","Two","Three"};
/* deklarace a inicializace globálního pole ukazatelů na
řetězce umístěné v FLASH
  pozn. Třebaže jsou stringy umístěné v FLASH, pole ukazate-
lů samo je umístěné v RAM */
char flash *messages[3]={"Message 1","Message 2","Message
3"};

/* deklarace několika stringů v EEPROM */
eeprom char m1[]="aaaa";

eeprom char m2[]="bbbb";

void main(void) {
/* deklarace lokálního pole ukazatelů na stringy umístěné v
  EEPROM
  pozn. Ačkoli stringy jsou umístěné v EEPROM,
```

```
   samo pole ukazatelů je umístěné v RAM */
   char eeprom *pp[2];

/* a inicializace pole */
pp[0]=m1;
pp[1]=m2;
}
```

4.13 Přístup k I/O registrům

Překladač používá klíčová slova **sfrb** a **sfrw** k přístupu k I/O registrům AVR MCU s použitím assemblerovských instrukcí IN a OUT.

Příklad:

```
/* definice SFR */

sfrb PINA=0x19; /* 8bitový přístup k SFR */
sfrw TCNT1=0x2c; /* 16bitový přístup k SFR */

void main(void) {
unsigned char a;
a=PINA; /* čtení vstupních pinů PORTA */
TCNT1=0x1111; /* zápis do registrů TCNT1L & TCNT1H */
}
```

Adresy I/O registrů jsou předdefinované v následujících hlavičkových souborech, umístěných v podadresáři ..\INC subdirectory:

tiny22.h
tiny26.h
90s2313.h
90s2323.h
90s2333.h
90s2343.h
90s4414.h
90s4433.h
90s4434.h
90s8515.h
90s8534.h
90s8535.h
mega103.h
mega128.h
mega16.h
mega161.h
mega163.h
mega169.h
mega32.h
mega323.h
mega603.h
mega64.h
mega8.h
mega8515.h
43USB355.h
94k.h

Můžeme pomocí direktivy **#include** zahrnout příslušný soubor na začátek našeho programu. Bitový přístup k I/O registrům je umožněn použitím **bitových selektorů** připojených za jméno I/O registru. Protože bitový přístup k I/O registrům provádějí instrukce CBI, SBI, SBIC a SBIS, adresy registrů musí být v rozsahu 0 až 1Fh pro **sfrb** a v rozsahu 0 až 1Eh pro **sfrw**.

Příklad:

```
sfrb PORTA=0x1b;
sfrb DDRA=0x18;
sfrb PINA=0x19;

void main(void) {
/* nastaví bit 0 Portu A jako výstup */
DDRA.0=1;

/* nastaví bit 1 Portu A jako vstup */

DDRA.1=0;

/* nastavuje bit 0 Portu A jako výstup */
PORTA.0=1;

/* testuje zda bit 1 je vstupním bitem Portu A */
if (PINA.1) { /* nějaký kód */ };

/*....... */

}
```

Ke zlepšení čitelnosti programu můžeme pomocí direktivy **#define** zavést symbolická jména pro bity I/O registrů:

```
sfrb PINA=0x19;
#define alarm_input PINA.2

void main(void)
{
/* testuje zda bit 2 je vstupním bitem Portu A */
if (alarm_input) { /* nějaký kód */ };

/*....... */
}
```

4.14 Přístup k EEPROM

Přístup k vnitřní EEPROM AVR MCU je nastaven použitím globálních proměnných umístěných za klíčové slovo **eeprom**.

Příklad:

```
/* hodnota 1 je uložena do EEPROM při programování chipu MCU */
eeprom int alfa=1;

eeprom char beta;
eeprom long array1[5];

/* řetězec je umístěn do EEPROM při programování chipu MCU */

void main(void) {
int i;

/* ukazatel do EEPROM */
int eeprom *ptr_to_eeprom;

/* píše hodnotu 0x55 do EEPROM - přímý způsob*/
alfa=0x55;

/* nebo nepřímý způsob s použitím ukazatele */
ptr_to_eeprom=&alfa;
*ptr_to_eeprom=0x55;
```



```
/* čte hodnotu z EEPROM - přímý způsob*/
i=alfa;
/* nebo nepřímý způsob s použitím ukazatele */
i=*ptr_to_eeprom;
}
```

Ukazatele do EEPROM jsou vždy 16bitové.

4.15 Použití přerušení

Přístup k AVR přerušovacímu systému je implementován s klíčovým slovem **interrupt**.

Příklad:

```
/* čísla vektorů přerušení obvodu AT90S8515 */

/* volané automaticky při externím přerušení */
interrupt [2] void external_int0(void) {
/* nějaký kód */

}

/* volán automaticky při přetečení TIMER0 */
interrupt [8] void timer0_overflow(void) {
/* nějaký kód */

}
```

Vektory přerušení jsou očíslovány od 1. Překladač automaticky uloží obsah všech použitých registrů při volání obslužných funkcí přerušení a obnoví jejich obsah po návratu z obsluhy přerušení. Assemblerovská instrukce RETI je umístěná na konci funkce obsluhující přerušení.

Funkce pro obsluhu přerušení nemusí vracet hodnotu ani nemusí mít parametry.

Rovněž musíte nastavit odpovídající bity řídicích registrů periferií kvůli konfiguraci přerušovacího systému a povolit přerušení.

Automatické ukládání a obnovování obsahu registrů R0, R1, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31 a SREG, při přerušeních může být zapnuto či vypnuto pomocí direktivy **#pragma savereg**.

Příklad:

```
/* vypnutí úschovy obsahu registrů */
#pragma savereg-
```

```
/* obsluha přerušení */
interrupt [1] void my_irq(void) {
/* nyní uložíme jen obsah registrů použitých dále v obslužné
rutině přerušení,
   např. R30, R31 a SREG */
#asm
   push r30
   push r31
   in r30,SREG
   push r30
#endasm

/* nějaký kód v céčku */
/*.... */

/* nyní obnovíme obsah SREG, R31 a R30 */
#asm
   pop r30
   out SREG,r30
   pop r31
   pop r30
#endasm
}
/* zapneme automatickou úschovu obsahu registrů při dalších
přerušeních */

#pragma savereg+
```

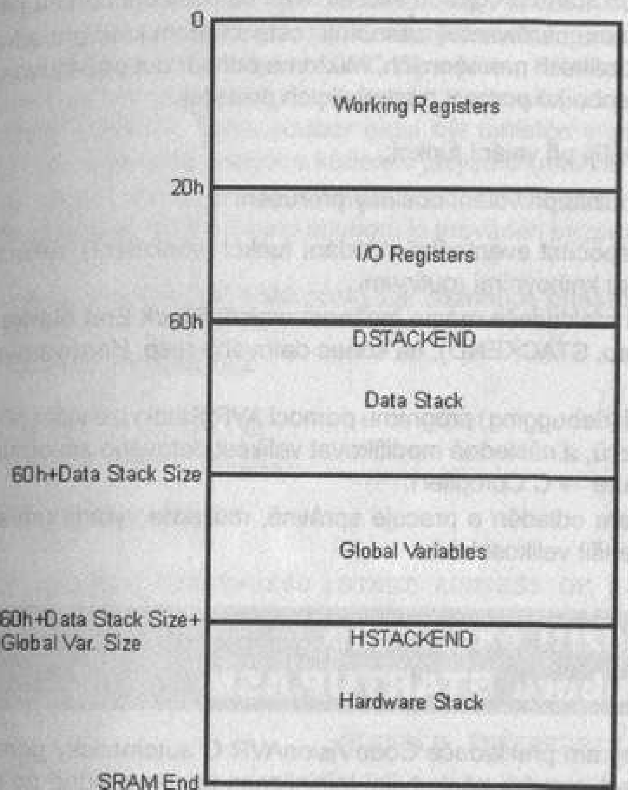
Přednastavené je automatické uchovávání obsahu registrů při přerušeních.

4.16 Využití SRAM

Přeložený program využívá paměť následujícím způsobem *obr. 4.1*.

Prostor pracovních registrů obsahuje 32 × 8 bitových registrů pro obecné použití. Překladač sám používá následující registry: R0, R1, R22, R23, R24, R25, R26, R27, R28, R29, R30 a R31. Dále mohou být některé z registrů R2 až R15 alokovány překladačem pro globální bitové proměnné. Zbývající registry můžeme libovolně používat ve svém programu. Prostor I/O registrů obsahuje 64 adres pro MCU periferie jako je Port Control Registers, Timer/Counters a další I/O funkce. Můžeme tyto registry používat v našich (assemblerovských) programech.

Prostor **datového zásobníku** je používán pro dynamické ukládání lokálních proměnných a k vkládání parametrů funkcí.



Obr. 4.1 Využití paměti

Ukazatel datového zásobníku je implementován pomocí registru Y. Při spuštění programu je datový zásobník inicializován hodnotou 5Fh+ velikost datového zásobníku. Po uložení hodnoty do datového zásobníku, provede se dekrementace zásobníkového ukazatele. Po vyjmutí hodnoty ze zásobníku se provede inkrementace zásobníkového ukazatele.

Při konfiguraci překladače v menu **Project → Configure → C Compiler**, musíme určit velikost prostoru zásobníku, aby nedošlo k přesahu zásobníku do I/O registrového prostoru při provádění programu.

Prostor pro globální proměnné se používá pro statické uchovávání globálních proměnných při provádění programu. Velikost tohoto prostoru může být spočtena jako součet velikostí všech deklarovaných globálních proměnných.

Prostor **hardwarového zásobníku** je používán pro uchování návratových adres funkcí a hodnot registrů R0, R1, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31 a SREG při obsluze přerušení. Registr SP se používá jako ukazatel zásobníku.

Registr SP je při startu programu inicializován na poslední adresu paměti RAM.

Při běhu programu hardwarový zásobník roste směrem k nižším adresám směrem k prostoru globálních proměnných. Můžeme odhadnout požadovanou velikost hardwarového zásobníku pomocí následujících pravidel:

- 2 byty se použijí při volání funkcí;
- 15 bytů je použito při volání obsluhy přerušení.

Musíme také spočítat eventuální skládání funkcí (vhnízdění), rekurzivní volání a využití zásobníku knihovními routinami.

Při konfiguraci překladače máme možnost umístit **Stack End Markers** (řetězce DSTACKEND, resp. STACKEND), na konec datového resp. Hardwarového zásobníku.

Při odlaďování (debugging) programu pomocí AVR Studia lze vidět případné přepsání těchto řetězců, a následně modifikovat velikost datového zásobníku (v menu Project → Configure → C Compiler).

Když je program odladěn a pracuje správně, můžeme vyřadit umístění těchto řetězců a tím zmenšit velikost kódu.

4.17 Použití externího souboru STARTUP.ASM

Pro každý program překladače CodeVisionAVR C automaticky generuje kódovou sekvenci, která provádí následující inicializace bezprostředně po resetu AVR čipu:

1. tabulka skoků vektorů přerušení,
2. zakázání globálního přerušení,
3. zakázání přístupu k EEPROM,
4. zakázání Watchdog časovače,
5. v případě potřeby povolení přístupu k externí RAM a povolení čekacích stavů (wait state),
6. vynulování RAM,
7. inicializace globálních proměnných umístěných v RAM,
8. inicializace ukazatele datového zásobníku = registru Y,
9. inicializace ukazatele zásobníku – registru SP,
10. inicializace registru UBRR, je-li to potřebné.

Automatické generování kódových sekvencí 2 až 7 může být vyřazeno výběrem **Use** v checboxu **External Startup Initialization File** v dialogovém okně vybraném pomocí menu **Project→ Configure → C Compiler**. Překladač C pak zahrne do vytvořeného .asm souboru, kódovou sekvenci z externího souboru, který se musí jmenovat STARTUP.ASM. Tento soubor musí být umístěn v adresáři ve kterém se nachází i hlavní zdrojový soubor s kódem v jazyce C (main C source file).

Můžete si napsat vlastní STARTUP.ASM soubor a tak přidat svému programu některé další vlastnosti. Kód z tohoto souboru je prováděn bezprostředně po resetu MCU čipu.

Základní soubor STARTUP.ASM poskytuje distribuce překladače a je umístěn v adresáři ..\BIN.

Toto je obsah tohoto souboru:

```
;CodeVisionAVR C Compiler
;(C) 1998-2000 Pavel Haiduc

;EXAMPLE STARTUP FILE

    .EQU __CLEAR_START=0X60 ;START ADDRESS OF RAM AREA TO
    CLEAR
    .EQU __CLEAR_SIZE=256   ;SIZE OF RAM AREA TO CLEAR IN
    BYTES

    CLI                     ;DISABLE INTERRUPTS
    CLR R30
    OUT EECR,R30            ;DISBALE EEPROM ACCESS

;DISABLE WATCHDOG
    LDI R30,0x18
    OUT WDTCR,R30
    LDI R30,0x10
    OUT WDTCR,R30

;CLEAR RAM
    CLR R30
    OUT MCUCR,R30           ;MCUCR=0, NO EXTERNAL RAM ACCESS
    LDI R24,LOW(__CLEAR_SIZE)

LDI R25,HIGH(__CLEAR_SIZE)
    LDI R26,__CLEAR_START
    CLR R27
__CLEAR_RAM:
    ST X+,R30
```

```
    SBIW R24,1
    BRNE __CLEAR_RAM

;GLOBAL VARIABLES INITIALIZATION
    LDI R30,LOW(__GLOBAL_INI_TBL*2)
    LDI R31,HIGH(__GLOBAL_INI_TBL*2)
__GLOBAL_INI_NEXT:
    LPM
    MOV R1,R0
    ADIW R30,1
    LPM
    ADIW R30,1
    MOV R22,R30
    MOV R23,R31
    MOV R31,R0
    MOV R30,R1
    SBIW R30,0
    BREQ __GLOBAL_INI_END
    LPM
    MOV R26,R0
    ADIW R30,1
    LPM

MOV R27,R0
    ADIW R30,1
    LPM
    MOV R24,R0
    ADIW R30,1
    LPM
    MOV R25,R0
    ADIW R30,1
__GLOBAL_INI_LOOP:
    LPM
    ST X+,R0
    ADIW R30,1
    SBIW R24,1
    BRNE __GLOBAL_INI_LOOP
    MOV R30,R22
    MOV R31,R23
    RJMP __GLOBAL_INI_NEXT
__GLOBAL_INI_END:
```


Konstanty __CLEAR_START a __CLEAR_SIZE mohou být změněny, aby se mohlo určit, která část v prostoru paměti RAM má být vynulována při inicializaci programu.

Návěští __GLOBAL_INI_TBL musí být umístěné na začátku tabulky obsahující informaci nutnou k inicializaci globálních proměnných umístěných v RAM. Tato tabulka je automaticky generována překladačem.

4.18 Využití assembleru ve zdrojovém kódu C jazyka

Kamkoli do zdrojového kódu C můžeme umístit i kód v assembleru pomocí direktiv **#asm** a **#endasm**.

Příklad:

```
void delay(unsigned char i) {
while (i--) {
      /* kód v assembleru */
      #asm
      nop
      nop
      #endasm
      };
}
```

Assembler lze rovněž použít inline.

Příklad:

```
#asm("sei") /* povolení přerušení */
```

Musíme se však vyhnout užití registrů R0, R1, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30 a R31 protože je používá překladač.

Totéž platí pro registry R2 až R15 jsou-li použity překladačem pro globální bitové proměnné.

4.19 Volání funkcí napsaných v assembleru

Následující příklad ukazuje jak přistupovat v jazyce C k funkcím napsaným v assembleru:

```
// funkce s deklarací v assembleru
// tato funkce vrací a+b+c
```

```
#pragma warn- // toto vyřadí upozornění (warnings)
int sum_abc(int a, int b, unsigned char c) {
#asm
    ldd r30,y+3    ;R30=LSB a
    ldd r31,y+4    ;R31=MSB a
    ldd r26,y+1    ;R26=LSB b
    ldd r27,y+2    ;R27=MSB b
    add r30,r26    ;(R31,R30)=a+b
    adc r31,r27
    ld r26,y       ;R26=c
    clr r27        ;převod unsigned char c na int

    add r30,r26    ;(R31,R30)=(R31,R30)+c
    adc r31,r27
#endasm
}
#pragma warn+ // povolení upozornění (warnings)

void main(void) {
int r;
// nyní voláme funkci a výsledek uložíme do r
r=sum_abc(2,4,6);
}
```

Překladač přebírá parametry funkcí pomocí datového zásobníku.

Nejdříve do něj ukládá celočíselný parametr a, pak b, a nakonec unsigned char parametr c.

Při každém uložení do zásobníku (push) se registrová dvojice Y dekrementuje o velikost parametrů (4 pro long int, 2 pro int, 1 pro char).

Pro vícebitové parametry se nejdříve ukládá MSB. Už jsme se zmínili o tom, že datový zásobník roste dolů. Poté, co jsou všechny parametry funkce uloženy do datového zásobníku, registr Y ukazuje na poslední parametr c, takže funkce může číst jeho hodnotu v R26 použitím instrukce ld r26,y. Parametr b byl uložen před c, takže je na vyšší adrese v datovém zásobníku. Funkce ho přečte pomocí instrukcí ldd r27,y+2 (MSB) a ldd r26,y+1 (LSB). MSB byl uložen nejdřív, takže je na vyšší adrese.

Parametr a byl uložen před b, takže je na vyšší adrese v datovém zásobníku.

Funkce ho přečte pomocí instrukcí ldd r31,y+4 (MSB) a ldd r30,y+3 (LSB).

Funkce vrací návratovou hodnotu v registrech (nejdřív LSB, pak MSB):

- R30 pro char a unsigned char,
- R30, R31 pro int a unsigned int,
- R30, R31, R22, R23 pro long a unsigned long.

Takže naše funkce musí vracet výsledek v registrech R30, R31. Po návratu z funkce překladač automaticky generuje kód pro vrácení prostoru datového zásobníku, který byl použit pro parametry funkce.

Direktiva překladače **#pragma warn-** zabrání překladači generovat upozornění, že funkce nevrací hodnotu. Je to potřeba z toho důvodu, že překladač C neví, co provádí kód v assembleru v těle funkce.

4.20 Využití debuggeru AVR studia

CodeVisionAVR umí spolupracovat s Atmel AVR Studio debuggerem verze 3 a 4.

V případě kdy chceme provádět debugging na úrovni jazyka C pomocí AVR Studia, musíme vybrat volbu **COFF** formátu výstupního souboru v menu **Project → Configure → Assembler**.

AVR Studio Debugger je spuštěn v AVR Studiu použitím menu **Tools → Debugger** nebo tlačítkem Debugger v toolbaru.

Po spuštění AVR Studia musí uživatel vybrat **File → |Open** má-li se zavést soubor COFF, který se bude debugovat.

Zavedený program může být spuštěn pomocí příkazu v menu **Debug → Go**, stisknutím klávesy **F5** nebo stisknutím tlačítka Execute program na toolbaru.

Provádění programu může být kdykoli zastaveno užitím příkazu menu **Debug → Break**, stisknutím **Ctrl+F5** nebo stisknutím tlačítka **Break** na toolbaru.

Pro krokování programem se používají příkazy **Debug → Trace Into** (F11 key), **debug → Step** (F10 key), **Debug → |Step Out** nebo odpovídající tlačítka na toolbaru.

Pro zastavení programu na určité řádce zdrojového kódu se použije menu **Breakpoints → Toggle Breakpoint**, tlačítka **F9** nebo odpovídajícího tlačítka na toolbaru.

V případě sledování programových proměnných uživatel musí vybrat v menu **Watch → Add Watch** nebo stisknutím tlačítka **Add Watch** na toolbaru, a určením jména proměnné ve sloupci **Watch**.

Registry AVR čipu lze prohlížet příkazem menu **View → Registers** nebo stisknutím kláves **Alt+0**.

Registry PC, SP, X, Y, Z AVR MCU a stavové příznaky mohou být zobrazeny příkazem menu **View → Processor** nebo stisknutím kláves **Alt+3**.

Obsahy pamětí FLASH, SRAM a EEPROM můžeme prohlížet po použití příkazu v menu **View → New Memory View** nebo stisknutím kláves **Alt+4**.

I/O registry mohou být prohlíženy po použití příkazu v menu **View → New IO View** nebo stisknutím kláves **Alt+5**.

Chceme-li použít terminálové I/O okno, vybrané příkazem menu **View → Terminal I/O**, pro komunikaci s UARTEem simulovaného AVR MCU, musí být vybrán

souborový formát COFF a **Use the Terminal I/O** AVR Studio check box musí být vybrán v menu **Project → Configure → C Compiler**.

Více informací o použití AVR Studia lze najít v jeho helpu a nebo v [13].

4.21 Zbývající rysy překladače CodeVisionAVR C

Chceme-li vytvořit co nejkratší a co nejrychlejší kód, musíme se řídit následujícími pravidly:

- pokud možno používat unsigned proměnné,
- pokud možno používat malé datové typy jako unsigned char,
- pokud možno používat lokální proměnné místo globálních,
- pokud možno používat TINY paměťový model,
- v překladači mít zapnutý **Code Optimizer**,
- konstantní řetězce ukládat do FLASH (použitím klíčového slova **flash**),
- po skončení debugování našeho programu provedeme znovu překlad s vyřazenou volbou **Stack End Markers**,
- časově náročné části kódu pište v assembleru.

Překladač CodeVisionAVR C má následující omezení:

- nejsou povoleny ukazatele na ukazatele,
- pole struktur nebo unionů mohou být jen jednodimenzionální,
- bitové složky struktur nejsou implementovány. Používají se jen bitové proměnné,
- EVALUATION (free) verze CodeVisionAVR C může vytvářet jen kód s omezenou délkou,
- v EVALUATION verzi nejsou podporovány funkce pro Philips PCF8563, PCF8583, Dallas Semiconductor DS1302, DS1307, 1 Wire Protocol a DS1820/DS18S20, 4x40 znakové LCD.

5 KNIHOVNÍ FUNKCE JAZYKA C CodeVisionAVR

Pomocí direktivy **#include** musíme vložit hlavičkové soubory knihovních funkcí používaných v programu.

Příklad:

```
/* hlavičkové soubory jsou vloženy před použitím funkce */

#include <math.h> // kvůli funkci abs
#include <stdio.h> // kvůli funkci putsf

void main(void) {
int a,b;
a=-99;
/* zde se tyto funkce používají */
b=abs(a);
putsf("Ahoj svete ");
}
```

5.1 Znakové funkce

Prototypy těchto funkcí jsou umístěné v souboru **ctype.h**, umístěném v podadresáři ..\INC. Tento soubor musí být vložen pomocí **#include** před použitím těchto funkcí.

`unsigned char isalnum(char c)`
vrací 1 když c je alphanumerický znak, jinak vrací 0..

`unsigned char isalpha(char c)`
vrací 1 když c je písmeno, jinak vrací 0.

`unsigned char isascii(char c)`
vrací 1 když c je ASCII znak (0 ... 127), jinak vrací 0.

`unsigned char iscntrl(char c)`
vrací 1 když c je řídicí znak (0 ... 31 nebo 127), jinak vrací 0

`unsigned char isdigit(char c)`
vrací 1 když c je dekadická číslice (0 ... 9), jinak vrací 0.

`unsigned char islower(char c)`
vrací 1 když c je malé písmeno (a ... z), jinak vrací 0.

`unsigned char isprint(char c)`
vrací 1 když c je tisknutelný znak (32 ... 127), jinak vrací 0.

`unsigned char ispunct(char c)`
vrací 1 když c je interpunkční znak, jinak vrací 0.

`unsigned char isspace(char c)`
vrací 1 když c je mezera, tabulátor, návrat vozíku, nový řádek, vertikální tabulátor nebo posun stránky, v opačném případě vrací 0.

`unsigned char isupper(char c)`
vrací 1 když c je velké písmeno (A ... Z), jinak vrací 0.

`unsigned char isxdigit(char c)`
vrací 1 když c je hexadecimální číslice, jinak vrací 0.

`char toascii(char c)`
vrací konvertovanou hodnotu, ASCII ekvivalent, parametru c.

`unsigned char toint(char c)`
interpretuje c jako hexadecimální číslici a vrací usigned char v rozsahu 0 až 15.

`char tolower(char c)`
vrací konvertovanou hodnotu parametru c na odpovídající malé písmeno, je-li c velké písmeno, jinak nechává znak c nezměněn.

`char toupper(char c)`
vrací konvertovanou hodnotu parametru c na odpovídající velké písmeno, je-li c malé písmeno, jinak nechává znak c nezměněn.

5.2 Standardní I/O funkce

Prototypy těchto funkcí jsou umístěny v souboru **stdio.h**, umístěném v podadresáři ..\INC. Tento soubor musí být pomocí **#include** vložen před použitím těchto funkcí.

I/O funkce ze standardního jazyka C implementované pro použití v mikrokontrolérech mají určitá omezení.

Nízkoúrovňové I/O funkce jsou:

`char getchar(void)`
vrací znak přijatý UARTem.

`void putchar(char c)`
vysílá znak c pomocí UARTu.

Před použitím těchto funkcí musíme:

- inicializovat přenosovou rychlost **UART Baud rate,**
- povolení vysílače UARTu,
- povolení přijímače UARTu.

Příklad:

```
#include <90s8515.h>
#include <stdio.h>

/* kmitočet krystalu [Hz] */
#define xtal 4000000L

/* přenosová rychlost */
#define baud 9600

void main(void) {
char k;

/* inicializace přenosové rychlosti UARTu */
UBRR=xtal/16/baud-1;

/* inicializace řídicího registru UARTu
  RX & TX povoleny, bez využití přerušení, 8 bitů dat */
UCR=0x18;

while (1) {
      /* přijme znak */
      k=getchar();
      /* a vrací ho nazpátek */
      putchar(k);
      };
}
```

Jiný způsob nastavení přenosové rychlosti UARTu je přes menu **Project → Configure → C Compiler**. Chceme-li používat jiné periferie pro I/O funkce, musíme proto modifikovat funkce **getchar** a **putchar**.

Definice pro tyto funkce jsou přístupné v

`file stdio.h.`

Všechny vysokoúrovňové I/O funkce používají funkce **getchar** a **putchar**.

`void puts(char *str)`

s použitím putchar provádí výstup řetězce (string) str zakončeného nulou (null), umístěného v RAM, a potom ještě odešle znak *nový řádek*.

`void putsf(char flash *str)`

s použitím putchar provádí výstup řetězce (string) str zakončeného nulou (null), umístěného v FLASH, a potom ještě odešle znak *nový řádek*.

`void printf(char flash *fmtstr [, arg1, arg2,...])`

s použitím putchar provádí výstup formátovaného řetězce ve formátu specifikovaném v řetězci **fmtstr**. Řetězec **fmtstr** specifikující formát je konstanta a musí být umístěn v paměti FLASH. Implementace **printf** je redukovanou verzí standardní funkce jazyka C.

Toto omezení bylo provedeno kvůli specifickým potřebám vestavěných (embedded) systémů a dále protože úplná implementace **printf** by byla náročná na potřebné množství paměti.

Jsou implementovány následující specifikátory formátu:

%c provádí výstup následujícího argumentu jako ASCII znak

%d provádí výstup následujícího argumentu jako dekadické číslo **signed int**,

%i provádí výstup následujícího argumentu jako dekadické číslo **signed int**,

%u provádí výstup následujícího argumentu jako dekadické číslo **unsigned int**,

%x provádí výstup následujícího argumentu jako hexadecimální číslo unsigned int s malými písmeny,

%X provádí výstup následujícího argumentu jako hexadecimální číslo unsigned int s velkými písmeny,

%s provádí výstup následujícího argumentu jako řetězce znaků zakončeného null a umístěného v RAM,

%% provádí výstup znaku **%**.

Všechny číselné hodnoty jsou zarovnány napravo a zleva jsou doplněny mezerami. Je-li znak 0 vložen mezi **%** a **d, i, u, x** nebo **X** pak bude číslo zleva doplněno nulami. Je-li vložen znak – mezi **%** a **d, i, u, x** nebo **X** pak číslo se zarovná doleva.

Specifikátor šířky v rozmezí 1 až 9 může být vložen mezi **%** a **d, i, u, x** nebo **X**, aby určoval minimální šířku zobrazovaného čísla.

Zobrazované číslo bude zarovnané doprava. Umístění znaku – před specifikátor šířky způsobí zarovnání čísla doleva.

`void sprintf(char *str, char flash *fmtstr [, arg1, arg2,...])`

tato funkce je shodná s **printf** až na to, že formátovaný text je umístěn v řetězci **(string) str** zakončeném null.

`char *gets(char *str, unsigned char len)`

s použitím **getchar** provádí vstup řetězce znaků **str** zakončeného znakem *nový řádek*. Znak *nový řádek* bude nahrazen 0.

Maximální délka řetězce je **len**. Když je přečteno **len** znaků (znak *nový řádek* se nepočítá do délky **len**), pak je řetězec zakončen 0 a funkce končí.

Funkce vrací ukazatel na str.

`char scanf(char flash *fmtstr [, arg1 address, arg2 address,...])`

s použitím **getchar** provádí vstup a formátování textu, ve formátu specifikovaném v řetězci **fmtstr**. Řetězec **fmtstr** specifikující formát je konstanta a musí být umístěn v paměti FLASH. Tato implementace je redukovanou verzí standardní funkce jazyka C. Toto omezení bylo provedeno kvůli specifickým potřebám vestavěných (embedded) systémů a dále protože úplná implementace **scanf** by byla náročná na potřebné množství paměti. Jsou implementovány následující specifikátory formátu:

%c provádí vstup následujícího argumentu jako ASCII znak,

%d provádí vstup následujícího argumentu jako dekadické celé číslo,

%i provádí vstup následujícího argumentu jako dekadické celé číslo,

%u provádí vstup následujícího argumentu jako dekadické celé číslo (unsigned),

%x provádí vstup následujícího argumentu jako hexadecimální celé číslo (unsigned),

%s provádí vstup následujícího argumentu jako řetězec ukončený znakem null.

Tato funkce vrací počet úspěšně nasnímaných, konvertovaných a uložených polí, nebo –1 v případě chyby.

`char sscanf(char *str, char flash *fmtstr [, arg1 address, arg2 address,...])`

tato funkce je identická se scanf vyjma toho, že se čte formátovaný text z řetězce str zakončeným null, umístěným v RAM.

5.3 Funkce standardní knihovny

Prototypy těchto funkcí jsou umístěny v souboru **stdlib.h**, umístěném v podadresáři ..\INC. Tento soubor musí být vložen pomocí **#include** před užitím těchto funkcí.

`int atoi(char *str)`

převádí řetězec **str** na celé číslo.

`long int atol(char *str)`
převádí řetězec **str** na dlouhé celé číslo.

`void itoa(int n,char *str)`
převádí celé číslo n na řetězec str.

`void ltoa(long int n,char *str)`
převádí dlouhé celé číslo **n** řetězec **str**.

Tato funkce je napsána v jazyce C a její prototyp i zdrojový kód jsou umístěny v souboru **ltoa.h**.

`void ftoa(float n,unsigned char decimals,char *str)`
převádí číslo v pohyblivé řádové čárce n na řetězec str.

Počet číslic je určen pomocí parametru.

Tato funkce je napsána v C a má svůj prototyp i zdrojový kód umístěný v souboru **ftoa.h**.

`void ftoe(float n,unsigned char decimals,char *str)`
převádí reálné číslo v pohyblivé řádové čárce n na řetězec str.

Číslo je vyjádřeno jako mantisa s daným počtem číslic a celočíselnou mocninou dekadického exponentu (např. 12.35e-5).

Tato funkce je napsána v jazyce C a má svůj prototyp i zdrojový kód umístěný v souboru **ftoe.h**.

`float atof(char *str)`
převádí řetězec na číslo v pohyblivé řádové čárce.

Tato funkce je napsána v jazyce C a má svůj prototyp i zdrojový kód umístěný v souboru **atof.h**.

`int rand (void)`
generuje pseudonáhodné číslo v rozmezí 0 až 32767.

`void srand(int seed)`
nastavuje počáteční hodnotu generátoru pseudonáhodných čísel používaného funkcí rand.

5.4 Matematické funkce

Prototypy těchto funkcí jsou umístěny v souboru **math.h**, umístěném v podadresáři ..\INC. Tento soubor musí být vložen pomocí **#include** před použitím těchto funkcí.

`unsigned char cabs(char n)`
vrací absolutní hodnoty bytu **n**.

`unsigned int abs(int n)`
vrací absolutní hodnotu celého čísla **n**.

`unsigned long labs(long int n)`
vrací absolutní hodnotu dlouhého celého čísla **n**.

`float fabs(float n)`
vrací absolutní hodnotu čísla v pohyblivé řádové čárce n

`char cmax(char a,char b)`
vrací větší z bytů **a** a **b**.

`int max(int a,int b)`
vrací větší z celých čísel **a** a **b**.

`long int lmax(long int a,long int b)`
vrací větší z dlouhých celých čísel **a** a **b**.

`float fmax(float a,float b)`
vrací větší z čísel v pohyblivé řádové čárce **a** a **b**.

`char cmin(char a,char b)`
vrací menší z bytů **a** a **b**.

`int min(int a,int b)`
vrací menší z celých čísel **a** a **b**.

`long int lmin(long int a,long int b)`
vrací menší z dlouhých celých čísel **a** a **b**.

`float fmin(float a,float b)`
vrací menší z čísel v pohyblivé řádové čárce **a** a **b**.

`char csign(char n)`
vrací –1, 0 nebo 1 podle toho, zda byte **n** je záporný, nulový nebo kladný.

`char sign(int n)`
vrací –1, 0 nebo 1 podle toho, zda celé číslo **n** je záporné, nulové nebo kladné.

`char lsign(long int n)`
vrací –1, 0 nebo 1 podle toho, zda dlouhé celé číslo **n** je záporné, nulové nebo kladné.

`char fsign(float n)`
vrací –1, 0 nebo 1 podle toho, zda číslo v pohyblivé řádové **čárce n** je záporné, nulové nebo kladné.

`unsigned char sqrt(unsigned int n)`
vrací druhou odmocninu z čísla **n** typu unsigned int.

`unsigned int lsqrt(unsigned long n)`
vrací druhou odmocninu z čísla **n** typu unsigned long int.

`float fsqrt(float n)`
vrací druhou odmocninu z kladného čísla v pohyblivé řádové čárce **n**.

`float floor(float n)`
vrací největší celé číslo ne větší než **n**, toto vrácené číslo je typu float. Zaokrouhluje dolů.

`float ceil(float n)`
vrací nejmenší číslo ne menší než **n**, toto vrácené číslo je typu float. Zaokrouhluje nahoru.

`float fmod(float x, float y)`
vrací zbytek po dělení čísla **x** číslem **y**.

`float modf(float x, float *ipart)`
rozkládá číslo v pohyblivé řádové čárce **x** na celočíselnou a zlomkovou část.

Tato funkce vrací zlomkovou část x jako číslo typu signed float. Celou část x ukládá do ipart.

`float ldexp(float x, int expn)`
vrací $x * 2^{expn}$.

`float frexp(float x, int *expn)`
vrací mantisu a exponent čísla v pohyblivé řádové čárce **n**.

`float exp(float x)`
vrací e^x.

`float log(float x)`
vrací přirozený logaritmus čísla **x** v pohyblivé řádové čárce

`float log10(float x)`
vrací dekadický logaritmus čísla **x** v pohyblivé řádové čárce

`float pow(float x, float y)`
vrací x^y.

`float sin(float x)`
vrací sinus vstupní hodnoty – čísla **x** v pohyblivé řádové čárce, kde úhel je vyjádřen v radiánech.

`float cos(float x)`
vrací cosinus vstupní hodnoty – čísla **x** v pohyblivé řádové čárce, kde úhel je vyjádřen v radiánech.

`float tan(float x)`
vrací tangens – čísla **x** v pohyblivé řádové čárce, kde úhel je vyjádřen v radiánech.

`float sinh(float x)`
vrací hyperbolický sinus vstupní hodnoty – čísla **x** v pohyblivé řádové čárce, kde úhel je vyjádřen v radiánech.

`float cosh(float x)`
vrací hyperbolický cosinus vstupní hodnoty – čísla **x** v pohyblivé řádové čárce, kde úhel je vyjádřen v radiánech.

`float tanh(float x)`
vrací hyperbolický tangens vstupní hodnoty – čísla **x** v pohyblivé řádové čárce, kde úhel je vyjádřen v radiánech.

`float asin(float x)`
vrací arkus sinus vstupní hodnoty – čísla **x** v pohyblivé řádové čárce, který je v rozmezí –PI/2 až PI/2. **x** musí být v rozmezí –1 až 1.

`float acos(float x)`
vrací arkus cosinus vstupní hodnoty – čísla **x** v pohyblivé řádové čárce, který je v rozmezí 0 až PI. **x** musí být v rozmezí –1 až 1.

`float atan(float x)`
vrací arkus tangent čísla **x** v pohyblivé řádové čárce, který je v rozmezí –PI/2 až PI/2.

`float atan2(float y, float x)`
vrací arkus tangent čísla **y/x** v pohyblivé řádové čárce, který je v rozmezí –PI až PI.

5.5 Řetězcové funkce

Prototypy těchto funkcí jsou umístěny v souboru **string.h**, umístěném v podadresáři ..\INC. Tento soubor musí být pomocí **#include** vložen před použitím funkcí.

Manipulace s řetězci byly rozšířeny na zacházení s řetězci umístěnými v pamětech RAM i FLASH.

`char *strcat(char *str1,char *str2)`
připojuje kopii řetězce **str2** na konec řetězce **str1**.

`char *strcatf(char *str1,char flash *str2)`
připojuje kopii řetězce **str2**, umístěného v FLASH, na konec řetězce **str1**.

`char *strncat(char *str1,char *str2,unsigned char n)`
kopíruje maximálně **n** znaků z řetězce **str2** na konec řetězce **str1**. Vrací ukazatel na řetězec **str1**.

`char *strncatf(char *str1,char flash *str2,unsigned char n)`
kopíruje maximálně **n** znaků z řetězce **str2**, umístěného v FLASH, na konec řetězce **str1**. Vrací ukazatel na řetězec **str1**.

char *strchr(char *str,char c)
vrací ukazatel na první výskyt znaku **c** v řetězci **str**, jestliže se **c** nevyskytuje v str, vrací NULL.

char *strrchr(char *str,char c)
vrací ukazatel na poslední výskyt znaku **c** v řetězci **str**, jestliže se **c** nevyskytuje v **str**, vrací NULL.

char strpos(char *str,char c)
vrací index prvního výskytu znaku **c** v řetězci **str**, jinak vrací –1.

char strrpos(char *str,char c)
vrací index posledního výskytu znaku **c** v řetězci **str**, jinak vrací –1.

char strcmp(char *str1,char *str2)
porovnává řetězec **str1** s řetězcem **str2**. Vrací hodnotu, která je
<0 pro str1<str2,
je 0 pro str1=str2,
a je >0 pro str1>str2.

char strcmpf(char *str1,char flash *str2)
porovnává řetězec **str1**, umístěný v SRAM, s řetězcem **str2**, umístěným v FLASH. Vrací hodnotu, která je
<0 pro str1<str2,
je 0 pro str1=str2,
a je >0 pro str1>str2.

char strncmp(char *str1,char *str2,unsigned char n)
porovnává nejvýše n znaků řetězce **str1** s řetězcem **str2**. Vrací hodnotu, která je
<0 pro str1<str2,
je 0 pro str1=str2,
a je >0 pro str1>str2.

char strncmpf(char *str1,char flash *str2,unsigned char n)
porovnává nejvýše **n** znaků řetězce **str1**, umístěného v RAM, s řetězcem str2, umístěného v FLASH. Vrací hodnotu, která je
<0 pro str1<str2,
je 0 pro str1=str2,
a je >0 pro str1>str2.

char *strcpy(char *dest,char *src)
kopíruje řetězec **src** do řetězce **dest**, přičemž končí po dosažení nulového ukončovacího znaku.

char *strcpyf(char *dest,char flash *src)
kopíruje řetězec **src**, umístěný v FLASH, do řetězce **dest**, umístěného v RAM, přičemž končí po dosažení nulového ukončovacího znaku. Vrací ukazatel na řetězec **dest**.

char *strncpy(char *dest,char *src,unsigned char n)
kopíruje až **n** znaků z řetězce **src** do řetězce **dest**, přičemž řetězec **dest** ořezává nebo doplňuje nulami. Cílový řetězec **dest** nemusí být ukončen nulou, když délka řetězce **src** je **n** nebo větší. Vrací ukazatel na řetězec **dest**.

char *strncpyf(char *dest,char flash *src,unsigned char n)
kopíruje až **n** znaků z řetězce **src**, umístěného v FLASH, do řetězce **dest**, umístěného v RAM. Přičemž řetězec **dest** ořezává nebo doplňuje nulami. Cílový řetězec **dest** nemusí být ukončen nulou, když délka řetězce **src** je **n** nebo větší. Vrací ukazatel na řetězec **dest**.

unsigned char strspn(char *str,char *set)
hledá počáteční úsek řetězce **str**, jenž se zcela skládá ze znaků řetězce **set**. Vrací délku počátečního úseku řetězce **str**, jež se zcela skládá ze *znaků řetězce* **set**.

unsigned char strspnf(char *str,char flash *set)
hledá počáteční usek řetězce **str**, umístěného v RAM, jenž se zcela skládá ze znaků řetězce **set**, umístěného v FLASH. Vrací délku počátečního úseku řetězce **str**, jež se zcela skládá ze znaků řetězce **set**.

unsigned char strcspn(char *str,char *set)
snímá řetězec **str** a hledá první úsek, který neobsahuje žádnou podmnožinu z řetězce **set**. Vrací délku počátečního úseku řetězce **str**, jenž se skládá pouze ze znaků neobsažených v řetězci **set**.

unsigned char strcspnf(char *str,char flash *set)
snímá řetězec **str** a hledá první úsek, který neobsahuje žádnou podmnožinu z řetězce **set**, umístěného v FLASH. Vrací délku počátečního úseku řetězce **str**, jenž se skládá pouze ze znaků *neobsažených* v řetězci **set**.

char *strpbrk(char *str,char *set)
snímá řetězec **str** a hledá první výskyt libovolného znaku, který se vyskytuje v řetězci **set**. Vrací ukazatel na první výskyt libovolného ze znaků v řetězci **set**. Jestliže se v řetězci **str** nevyskytuje žádný ze znaků řetězce **set**, vrací NULL.

char *strpbrkf(char *str,char flash *set)
snímá řetězec **str** a hledá první výskyt libovolného znaku, který se vyskytuje v řetězci **set**, umístěného v FLASH. Vrací ukazatel na první výskyt libovolného ze znaků v řetězci set. Jestliže se v řetězci str nevyskytuje žádný ze znaků řetězce **set**, vrací NULL.

`char *strrpbrk(char *str,char *set)`
snímá řetězec **str** a hledá poslední výskyt libovolného znaku, který se vyskytuje v řetězci **set**. Vrací ukazatel na poslední výskyt libovolného ze znaků v řetězci set. Jestliže se v řetězci **str** nevyskytuje žádný ze znaků řetězce **set**, vrací NULL.

`char *strrpbrkf(char *str,char flash *set)`
snímá řetězec **str** a hledá poslední výskyt libovolného znaku, který se vyskytuje v řetězci **set**, umístěného v FLASH. Vrací ukazatel na poslední výskyt libovolného ze znaků v řetězci **set**. Jestliže se v řetězci **str** nevyskytuje žádný ze znaků řetězce **set**, vrací NULL.

`char *strstr(char *str1, char *str2)`
snímá řetězec **str1** na první výskyt podřetězce **str2** v řetězci **str1**. Vrací ukazatel na prvek v řetězci **str1**, kde začíná řetězec **str2**. Jestliže se **str2** nevyskytuje v str1, vrací NULL.

`char *strstrf(char *str1, char flash *str2)`
snímá řetězec **str1** na první výskyt podřetězce **str2**, umístěného v FLASH, v řetězci **str1**. Vrací ukazatel na prvek v řetězci **str1**, kde začíná řetězec **str2**. Jestliže se str2 nevyskytuje v **str1**, vrací NULL.

`char *strtok(char *str1, char flash *str2)`
hledá v jednom řetězci rámce (tokens), které jsou odděleny omezovači definovanými ve druhém řetězci. Tato funkce předpokládá, že řetězec **str1**, umístěný v SRAM, se skládá z posloupnosti N rámců oddělených polemi jednoho nebo více znaků z řetězce oddělovačů **str2**, umístěného v FLASH, přičemž N má nezápornou hodnotu. První volání **strtok** vrací ukazatel na první znak prvního rámce v řetězci str1 a zapisuje NULL do **str1** bezprostředně za vraceným rámcem. Následující volání s prvním argumentem rovným NULL bude pracovat v řetězci **str1** tímto způsobem, dokud nezůstanou žádné rámce.

strtok vrací ukazatel na rámec nalezený v řetězci **str1**. Nulový ukazatel NULL se vrací, když nejsou žádné další rámce.

`unsigned char strlen(char *str)`
používá se při nastavení TINY paměťového modelu. Vrací délku řetězce str (v rozsahu 0–255).

`unsigned int strlen(char *str)`
používá se při nastavení SMALL paměťového modelu. Vrací délku řetězce str (v rozsahu 0–65535).

`unsigned int strlenf(char flash *str)`
vrací délku řetězce **str** umístěného v FLASH.

`void *memcpy(void *dest,void *src,unsigned char n)` pro paměťový model TINY

`void *memcpy(void *dest,void *src,unsigned int n)` pro paměťový model SMALL

Kopíruje blok n bytů z řetězce **src** na **dest**. Překrývají-li **src** s **dest**, chování této funkce není definováno a je třeba použít místo ní funkci **memmove**. Vrací ukazatel na **dest**.

`void *memcpyf(void *dest,void flash *src,unsigned char n)` pro TINY model

`void *memcpyf(void *dest,void flash *src,unsigned int n)` pro SMALL model

Kopíruje n bytů z **src**, umístěného v FLASH, do **dest**. Vrací ukazatel na **dest**.

`void *memccpy(void *dest,void *src,char c,unsigned char n)` pro TINY model

`void *memccpy(void *dest,void *src,char c,unsigned int n)` pro SMALL model

Kopíruje blok n bytů ze **src** do **dest**. Kopírování se zastaví, jakmile nastane jedna ze dvou následujících událostí:

- znak **c** se poprvé kopíroval do **dest**,
- **n** bytů se kopírovalo do **dest**.

Memccpy vrací ukazatel na byte v řetězci **dest**, který bezprostředně následuje za znakem **c**, když se znak **c** zkopíroval, jinak vrací NULL.

`void *memmove(void *dest,void *src,unsigned char n)` pro TINY model

`void *memmove(void *dest,void *src,unsigned int n)` pro SMALL model

Kopíruje n bytů z **src** na dest. I když se zdrojový a cílový blok překrývají, byty se na překrývajících pozicích kopírují správně. Vrací ukazatel na **dest**.

`void *memchr(void *buf,unsigned char c,unsigned char n)` pro TINY model

`void *memchr(void *buf,unsigned char c,unsigned int n)` pro SMALL model

Hledá znak **c** v prvních **n** bytech bloku, na který ukazuje **buf**.
V případě úspěchu vrací ukazatel na první výskyt **c** v **buf**, jinak vrací NULL.

`char memcmp(void *buf1,void *buf2,unsigned char n)` pro TINY model

`char memcmp(void *buf1,void *buf2,unsigned int n)` pro SMALL model

Porovnává prvních **n** bytů bloků **buf1** a **buf2**. Jako znaky typu unsigned char. Protože porovnává byty jako znaky unsigned char, vrací hodnotu:

<0, když **buf1<buf2**,

=0, když **buf1=buf2**,

>0, když **buf1>buf2**.

`char memcmpf(void *buf1,void flash *buf2,unsigned char n)` pro TINY model

`char memcmpf(void *buf1,void flash *buf2,unsigned int n)` pro SMALL model

Porovnává prvních **n** bytů bloků **buf1**, umístěného v RAM a **buf2**, umístěného v FLASH, jako znaky typu unsigned char. Protože porovnává byty jako znaky unsigned char, vrací hodnotu:

<0, když **buf1<buf2**,

=0, když **buf1=buf2**,

>0, když **buf1>buf2**.

`void *memset(void *buf,unsigned char c,unsigned char n)` pro TINY model

`void *memset(void *buf,unsigned char c,unsigned int n)` pro SMALL model

Nastavuje **n** bytů paměťového bloku buf na hodnotu bytu **c**. Vrací ukazatel na buf.

5.6 BCD konverzní funkce

Prototypy těchto funkcí jsou umístěny v souboru **bcd.h**, umístěném v podadresáři ..\INC. Tento soubor musí být vložen pomocí #include před použitím těchto funkcí.

`unsigned char bcd2bin(unsigned char n)`
konvertuje číslo **n** v BCD reprezentaci na jeho binární reprezentaci.

`unsigned char bin2bcd(unsigned char n)`
konvertuje binárně reprezentované číslo **n** na BCD ekvivalent tohoto čísla.

Číslo n musí mít hodnotu v rozmezí 0 až 99.

5.7 Konverzní funkce Grayova kódu

Prototypy těchto funkcí jsou umístěny v souboru **gray.h**, umístěném v podadresáři ..\INC. Tento soubor musí být vložen pomocí **#include** před použitím těchto funkcí.

unsigned char gray2binc(unsigned char n)

unsigned char gray2bin(unsigned int n)

unsigned char gray2binl(unsigned long n)

převádí číslo **n** z jeho reprezentaci v Grayově kódu do jeho binární reprezentace

unsigned char bin2grayc(unsigned char n)

unsigned char bin2gray(unsigned int n)

unsigned char bin2grayl(unsigned long n)

převádí číslo n v binární reprezentaci do jeho reprezentace v Grayově kódu.

5.8 Funkce pro přístup k paměti

Prototypy těchto funkcí jsou umístěny v souboru **mem.h**, umístěného v podadresáři ..\INC. Tento soubor musí být vložen pomocí **#include** před použitím těchto funkcí.

`void pokeb(unsigned int addr,unsigned char data)`
tato funkce ukládá hodnotu bytu do paměti RAM na adresu **addr**.

`void pokew(unsigned int addr,unsigned int data)`
tato funkce ukládá hodnotu datového slova do paměti RAM na adresu **addr**.

LSB je uložen na adresu **addr** a MSB je uložen na adresu **addr+1**.

`unsigned char peekb(unsigned int addr)`
vrací byte, který je umístěn v paměti RAM na adrese **addr**.

`unsigned int peekw (unsigned int addr)`
vrací slovo, které je umístěno v paměti RAM na adrese **addr**.

LSB je čteno z adresy **addr** a MSB je čteno z adresy **addr+1**.

5.9 Funkce pro LCD

LCD funkce jsou navrženy pro snadný přístup programů v jazyce C k alfanumerickým LCD modulům založeným na obvodu HD44780 a jeho ekvivalentech. Tento obvod se stal defacto normou pro moduly LCD.

Prototypy pro tyto funkce jsou umístěné v souboru **lcd.h**, umístěném v podadresáři ..\INC. Tento soubor musí být vložen pomocí **#include** před použitím těchto funkcí.

Před vložením souboru **lcd.h musíme** deklarovat, který port mikrokontroléru AVR bude komunikovat s modulem LCD.

Příklad:

```
/* the LCD module is connected to PORTC */
#asm
    .equ __lcd_port=0x15
#endasm
```

```
/* now you can include the LCD Functions */
#include <lcd.h>
```

Modul LCD komunikující s programem zahrnujícím tuto knihovnu **musí** mít vývody portu propojeny následujícím způsobem:

```
[LCD]      [AVR Port]
RS (pin4) ------ bit 0
RD (pin5) ------ bit 1
EN (pin 6) ------ bit 2
DB4 (pin 11) --- bit 4
DB5 (pin 12) --- bit 5
DB6 (pin 13) --- bit 6
DB7 (pin 14) --- bit 7
```

Rovněž musíme připojit napájení LCD modulu i obvody pro řízení jasu.

Poznámka: ***Velice snadno můžeme vytvořit vlastní knihovnu a hlavičkový soubor, umožňující jiné propojení modulu LCD a portu MCU. Více informací bude uvedeno dále v textu.*** *Do podadresáře ..\INC nakopírujeme soubor LCD1.h a do podadresáře ..\LIB soubor LCD1.lib. Oba soubory jsou umístěny na doprovodném CD. V kódu C pak místo*

```
#include <lcd.h>
```

bude

```
#include <lcd1.h>
```

Nízkoúrovňové LCD funkce jsou

`void _lcd_ready(void)`
čeká, až LCD modul je připraven přijmout data. Tato funkce musí být volána před zapsáním dat do LCD pomocí funkce **_lcd_write_data**.

`void _lcd_write_data(unsigned char data)`
zapisuje byte dat do LCD instrukčního registru. Tato funkce může být použita pro modifikaci konfigurace LCD.

Příklad:

```
/* povoluje zobrazovat kursor */
_lcd_ready();
_lcd_write_data(0xe);
```


`void lcd_write_byte(unsigned char addr, unsigned char data);`
zapisuje byte z LCD generátoru znaků nebo paměti RAM displeje

Příklad:

```
/* LCD uživatelem definované znaky

obvod: AT90S8515
paměťový model: SMALL
Velikost zásobníku: 128 bytes

Zapojení alfanumerického LCD 2 × 16
[LCD] 1 GND- 9 GND
2 +5V- 10 VCC
3 VLC- LCD HEADER Vo
4 RS - 1 PC0
5 RD - 2 PC1

6 EN - 3 PC2
11 D4 - 5 PC4
12 D5 - 6 PC5
13 D6 - 7 PC6
14 D7 - 8 PC7 */

/* LCD je připojen na výstupy PORTC */
#asm
.equ __lcd_port=0x15 ;PORTC
#endasm

/* vloží LCD driver routiny */
#include <lcd.h>

typedef unsigned char byte;

/* tabulka pro uživatelem definovaný znak
šipka ukazující do pravého horního rohu */
flash byte char0[8]={
0b0000000,
0b0001111,
0b0000011,
0b0000101,
0b0001001,
0b0010000,
0b0100000,
```

```
0b1000000};

/* funkce použitá k definici uživatelských znaků */
void define_char(byte flash *pc,byte char_code)
{
byte i,a;
a=(char_code<<3) | 0x40;
for (i=0; i<8; i++) lcd_write_byte(a++,*pc++);
}

void main(void)
{
/* inicializuje LCD na 2 řádky & 16 sloupců */
lcd_init(16);

/* definuje uživatelský znak 0 */
define_char(char0,0);

/* přepíná na zápis do displejové RAM */
lcd_gotoxy(0,0);
lcd_putsf("User char 0:");

/* zobrazuje uživatel definovaný znak 0 */
lcd_putchar(0);

while (1); /* nekonečná smyčka */
}

unsigned char lcd_read_byte(unsigned char addr);
čte byte z LCD generátoru znaků nebo paměti RAM displeje
```

Vysokoúrovňové LCD funkce jsou

`void lcd_init(unsigned char lcd_columns)`
inicializuje LCD modul, vymaže displej a nastaví aktuální pozici (pro „tisk" znaků) na řádek 0 i sloupec 0. Počet sloupců LCD musí být nastaven (např. na 16). Kurzor není zobrazován. Tato funkce musí být zavolána před voláním dalších vysokoúrovňových LCD funkcí.

`void lcd_clear(void)`
vymaže displej a nastaví aktuální pozici (pro „tisk" znaků) na řádek 0 i sloupec 0.

`void lcd_gotoxy(unsigned char x,unsigned char y)`

nastaví aktuální pozici displeje (pro „tisk" znaků) na sloupec **x** a řádek **y**. Řádky i sloupce jsou číslovány od 0.

`void lcd_putchar(char c)`
zobrazuje znak **c** na aktuální pozici displeje

`void lcd_puts(char *str)`
zobrazuje řetězec **str**, umístěný v RAM, na aktuální pozici LCD

`void lcd_putsf(char flash *str)`
zobrazuje řetězec **str**, umístěný v FLASH, na aktuální pozici LCD

Další LCD funkce

další funkce pro zobrazování znaků na LCD se 4 × 40 znaky využívají knihovnu lcd4x40, s hlavičkovým souborem **lcd4x40.h. Tato knihovna je však podporována jen komerční verzí CodeVision AVR C.**

U některých typů AVR MCU jako AT90S8515 či některé ATmega lze připojit **externí paměť**. U těchto MCU můžeme LCD namapovat jako vnější paměť. Tak je např. připojen LCD u startkitů STK200+ a STK300 firmy Kanda Systéme. Funkce pro takto zapojený LCD obsahuje knihovna lcdstk s hlavičkovým souborem **lcdstk.h**.

5.10 Funkce sběrnice I²C

I²C funkce jsou určeny pro snadnou implementaci interface mezi programy v C a různými obvody podporující sběrnici I²C firmy Philips.

Tyto funkce předpokládají, AVR MCU jako master a periferie jako slave.

Prototypy těchto funkcí se nachází v souboru **i2c.h**, umístěném v podadresáři ..\INC. Tento soubor musí být vložen pomocí **#include** před použitím těchto funkcí.

Před vložením souboru **i2c.h**, musíme deklarovat, který port mikrokontroléru a které bity tohoto portu budou sloužit pro komunikaci protokolem I²C.

Příklad:

```
/* I2C bus je připojen k PORTB */
/* signál SDA je bit 3 */
/* signál SCL je bit 4 */
#asm
    .equ  __i2c_port=0x18
    .equ  __sda_bit=3
    .equ  __scl_bit=4
#endasm

/* nyní můžeme vkládat I2C funkce */
#include <i2c.h>
```

Tyto I^2C funkce jsou:

void i2c_init(void)
tato funkce inicializuje I^2C bus. Tuto funkci musíme zavolat před užitím dalších I^2C funkcí.

unsigned char i2c_start(void)
generuje START komunikace. Vrací 1 když je I^2C sběrnice volná nebo 0 v opačném případě.

void i2c_stop(void)
generuje STOP komunikace.

unsigned char i2c_read(void)
čte byte ze sběrnice.

unsigned char i2c_write(unsigned char data)
zapisuje byte dat na sběrnici. Vrací 1 když slave potvrdí signálem ACK příjem tohoto byte, v opačném případě vrací 0.

Příklad přístupu k paměti EEPROM typu AT 24C02 s kapacitou 256 byte:

```
/* I2C bus je připojen na PORTB */
/* signál SDA je bit 3 */
/* signál SCL je is bit 4 */
#asm
    .equ __i2c_port=0x18
    .equ __sda_bit=3
    .equ __scl_bit=4
#endasm

/* nyní můžeme vložit I2C funkce */
#include <i2c.h>

/* deklarace funkce delay_ms */
#include <delay.h>

/* řídicí slovo pro EEPROM */
#define EEPROM_BUS_ADDRESS 0xa0

/* čte byte z EEPROM */
unsigned char eeprom_read(unsigned char address) {
unsigned char data;
i2c_start();
i2c_write(EEPROM_BUS_ADDRESS); //zapisuje řídicí slovo s 0
na konci - write
```



```
i2c_write(address);
i2c_start();
i2c_write(EEPROM_BUS_ADDRESS | 1);//zapisuje řídicí slovo
                                  //s 1 na kon.- read

data=i2c_read();
i2c_stop();
return data;
}

/* zapisuje byte do EEPROM */
void eeprom_write(unsigned char address,unsigned char data)
{
i2c_start();

i2c_write(EEPROM_BUS_ADDRESS);
i2c_write(address);
i2c_write(data);
i2c_stop();

/* 10ms časová prodleva nutná k zakončení operace zápisu */
delay_ms(10);
}

void main(void) {
unsigned char i;

/* inicializace I2C bus */
i2c_init();

/* zapisuje 55h na adresu AAh */
eeprom_write(0xaa,0x55);

/* čte byte z adresy AAh */
i=eeprom_read(0xaa);

/* konec - nekonečná smyčka */
while (1);
}
```

Poznámka: *Sběrnice I²C, má z hlediska univerzálnosti použití i z hlediska standardizace v porovnání s jinými podobnými sběrnicemi největší praktický význam. Pro aplikace z oblasti spotřební elektroniky (audio, video) a telekomunikací mají firmy Philips a Siemens v řadách obvodů PCF, SAA, TDA, TEA a TSA mnoho prvků řízených sběrnicí I²C. Konkrétní způsob komunikace – význam jednotlivých bitů v řídicích slovech, stejně jako význam vlastních přenášených dat či adres najdeme v katalogových listech pro tyto obvody.*

Pro některé vybrané typy těchto obvodů obsahuje CodeVisionAVR C i knihovní funkce uvedené v *tab. 5.1*.

Tab. 5.1 Knihovní funkce periferních obvodů

obvod	Popis obvodu	Hlavičkový soubor
National Semicondustor LM75	Teplotní čidlo, teploměr	**lm75.h**
Dallas Semiconductor DS1621	Teploměr/ termostat	**ds1621.h**
Phillips PCF8563	Obvod hodin reáného času	**pcf8563.h**
Phillips PCF8583	Obvod hodin reálného času	**pcf8583.h**
Dallas Semiconductor DS1307	Obvod hodin reálného času	**ds1307.h**
Dallas Semiconductor	Obvod hodin reálného času	**ds1302.h**

Dalším jednoduchým protokolem podporovaným knihovnami CodeVisionAVR C je protokol **1 wire** firmy Dallas Semiconductor. I zde se předpokládá AVR MCU jako master a periferie jako slave. Prototypy pro funkce používané při implementaci tohoto protokolu jsou umístěny v hlavičkovém souboru **1wire.h**. Navíc pro několik obvodů komunikujících pomocí protokolu 1 wire máme i další knihovny *tab. 5.2*

Tab. 5.2

obvod	Popis obvodu	Hlavičkový soubor
DS1820 / DS1822	Teplotní čidlo	**ds1820.h**
DS2430	EEPROM	**ds2430.h**
DS2433	EEPROM	**ds2433.h**

5.11 SPI funkce

Jednoduchou sériovou sběrnici, označovanou jako SPI (Serial Periphal Interface) najdeme především u mikropočítačů firmy MOTOROLA (řady MC68H05, MC68H11, MC68H16). Nyní je podporována i mikrokontroléry firmy ATMEL. Proto je logické, že její podpora je i součástí knihovny CodeVisonAVR C.

Sběrnice SPI je typicky používána pro připojení periferních obvodů (nebo podřízených MCU) k MCU. Je tvořena trojicí signálů. Hodinový signál SCK (Serial Clock) je generován řadičem (MCU), signály MOSI (Master Out/Slave In) a MISO (Master In/Slave Out) propojují posuvné registry řadiče a podřízeného obvodu do kruhu. Podřízený obvod musí být aktivován signálem $\overline{SS}$ (Slave Select). Vlastní předání dat je jednoduché, operace nad SPI zamění údaje v datových registrech řadiče a podřízeného obvodu.

Prototypy knihovních funkcí jsou umístěny v souboru **spi.h**, umístěném v podadresáři ..\INC. Tento soubor musí být vložen pomocí **#include** před použitím těchto funkcí.

Tyto SPI funkce jsou:

`unsigned char spi(unsigned char data)`
tato funkce vysílá byte a současně přijímá byte.

Před použitím této **SPI** funkce musíme provést konfiguraci **SPI Control Register SPCR** tak, jak je popsána v dokumentaci k příslušnému typu MCU (Atmel Data Sheets). Tato SPI funkce používá „polling" při SPI komunikaci, takže *není potřeba* nastavovat SPI Interrupt Enable Bit SPIE.

Příklad na využití SPI funkce pro komunikaci s AD7896 ADC:

```
/*
  Digitální voltmetr používající
  Analog Devices AD7896 ADC
  Připojený k AT90S8515
  S využitím SPI busu

  MCU: AT90S8515

  Paměťový model: SMALL
  Velikost zásobníku: 128 bytes
  Frekvence hodin: 4 MHz

  Připojení AD7896 k AT90S8515

  [AD7896] [AT9S8515 DIP40]
  1 Vin
```

```
  2 Vref=5V
  3 AGND - 20 GND
  4 SCLK - 8 SCK
  5 SDATA - 7 MISO
  6 DGND - 20 GND
  7 CONVST- 2 PB1
  8 BUSY - 1 PB0

  použijeme alfanumerický LCD 2x16
  připojený na PORTC :

  [LCD] [AT90S8515 DIP40]
  1 GND- 20 GND
  2 +5V- 40 VCC
  3 VLC
  4 RS - 21 PC0

  5 RD - 22 PC1
  6 EN - 23 PC2
  11 D4 - 25 PC4
  12 D5 - 26 PC5
  13 D6 - 27 PC6
  14 D7 - 28 PC7 */

#asm
    .equ __lcd_port=0x15
#endasm

#include <lcd.h> // funkce pro LCD
#include <spih> // funkce pro SPI
#include <90s8515.h>
#include <stdio.h>
#include <delay.h>

// AD7896 referenční napětí [mV]
#define VREF 5000L

// alokace řídicích signálů AD7896 na PORTB
#define ADC_BUSY PINB.0
#define NCONVST PORTB.1

// buffer displeje LCD

char lcd_buffer[33];
```



```
unsigned read_adc(void)
{
unsigned result;
// začátek převodu v módu 1
// (high sampling performance)
NCONVST=0;
NCONVST=1;
// čeká na ukončení převodu
while (ADC_BUSY);
// s použitím SPI čte MSB
result=(unsigned) spi(0)<<8;
// s použitím SPI čte LSB a spojí s MSB
result|=spi(0);
// spočítá napětí v [mV]
result=(unsigned) (((unsigned long) result*VREF)/4096L);
// vrací změřené napětí
return result;

}

void main(void)
{
// initialize PORTB
// PB.0 vstup z AD7896 BUSY
// PB.1 výstup do AD7896 /CONVST
// PB.2 & PB.3 vstupy
// PB.4 výstup (SPI /SS pin)
// PB.5 vstup
// PB.6 vstup (SPI MISO)
// PB.7 výstup do AD7896 SCLK
DDRB=0x92;
// inicializuje SPI do master modu
// nepoužije se přerušení, MSB se přenáší před LSB, clock //
// phase negative
// SCK na nule při idle, clock phase=0
// SCK=fxtal/4
SPCR=0x54;
// AD7896 bude pracovat v módu 1
// (high sampling performance)

// /CONVST=1, SCLK=0
PORTB=2;
// inicializace LCD
lcd_init(16);
```

```
lcd_putsf("AD7896 SPI bus\nVoltmeter");
delay_ms(2000);
lcd_clear();

// čte a zobrazuje vstupní napětí ADC
while  (1)
      {
      sprintf(lcd_buffer,"Uadc=%4umV",read_adc());
      lcd_clear();
      lcd_puts(lcd_buffer);
      delay_ms(100);
      };
}
```

5.12 Funkce pro úsporný režim (Power Management Functions)

Funkce Power Managementu jsou určeny pro uvedení AVR MCU do jednoho z jeho úsporných (nízkospotřebových) režimů.

Prototypy těchto funkcí jsou umístěny v souboru **sleep.h**, který se nachází v podadresáři ..\INC. Tento soubor musí být vložen pomocí **#include** před použitím těchto funkcí.

Funkce pro Power Management jsou:

`void sleep_enable(void)`
tato funkce povoluje úsporný režim.

`void sleep_disable(void)`
tato funkce zakazuje úsporný režim. Používá se k zabránění náhodného přechodu do úsporného režimu.

`void idle(void)`
tato funkce nastavuje AVR MCU do *idle mode*.

Před použitím této funkce musí být volána funkce sleep_enable povolující úsporný režim. V tomto módu je zastaveno CPU, ale čítače/časovače, Watchdog a přerušovací systém nepřestávají pracovat. CPU může být „probuzen", znovu zapnut, prostřednictvím vnějšího či vnitřního přerušení.

`void powerdown(void)`
tato funkce nastaví AVR mikrokontrolér do režimu power-down, režimu „spánku".

Před použitím této funkce musí být zavolána funkce **the sleep_enable** povolující úsporný režim. V tomto režimu je zastaven i externí oscilátor.

AVR může být „probuzen", znovu zapnut, jenom *externím resetem*, Watchdog time-out nebo vnějším přerušením.

`void powersave(void)`
tato funkce nastaví AVR mikrokontrolér do úsporného režimu.

Před použitím této funkce musí být zavolána funkce **the sleep_enable** povolující úsporný režim. Tento mód je obdobou módu power-down s několika odlišnostmi popsanými v dokumentaci k ATMEL AVR MCU.

`void standby(void)`
tato funkce nastaví AVR mikrokontrolér do standby módu. Před použitím této funkce musí být zavolána funkce **the sleep_enable** povolující úsporný režim. Tento mód je obdobou powerdown módu s tím rozdílem, že vnitřní oscilátor zůstává v činnosti.

Před použitím tohoto módu je třeba si ověřit v dokumentaci Atmel Data, že námi použitý typ AVR MCU podporuje tento režim.

`void extended_standby(void)`
tato funkce nastaví AVR mikrokontrolér do rozšířeného standby módu.

Před použitím této funkce musí být zavolána funkce **the sleep_enable** povolující úsporný režim. Tento režim je obdobou úsporného režimu s tím rozdílem, že vnitřní oscilátor zůstává v činnosti. Před použitím tohoto módu je třeba si ověřit v dokumentaci Atmel Data, že námi použitý typ AVR MCU podporuje tento režim.

5.13 Funkce časových prodlev, časového zpoždění

Tyto funkce jsou navrženy pro realizaci časového zpoždění v programech v jazyce C.

Prototypy těchto funkcí jsou umístěny v souboru **delay.h**, nacházejícím se v podadresáři ..\INC. Tento soubor musí být vložen pomocí **#include** před použitím těchto funkcí. **Před voláním těchto funkcí musí být zakázáno přerušení**, jinak mohou být časové prodlevy delší, než máme nastaveno. Rovněž je třeba mít správně nastavený kmitočet hodin AVR MCU v menu **Project → Configure → C Compiler**.

Tyto funkce jsou:

`void delay_us(unsigned int n)`
generuje časovou prodlevu n mikrosekund, n musí být konstantní výraz.

`void delay_ms(unsigned int n)`
generuje časovou prodlevu n milisekund.

Příklad:

```
void main(void) {
/* zakazuje přerušení */
#asm("cli")

/* 100ms časová prodleva */
delay_us(100);

/* povoluje přerušení */
#asm("sei")
/*............. */
}
```

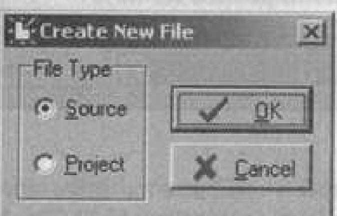

6

VYTVÁŘENÍ KNIHOVEN

6.1 Vytvoření vlastní knihovny

V předchozí kapitole jsme si popsali některé knihovny, které jsou součástí instalace CodeVisionAVR C. V řadě případů se však ukáže potřeba napsat si vlastní knihovnu, např. proto, že potřebujeme zajistit komunikaci AVR MCU s obvodem, který není podporován již nainstalovanými knihovnami, potřebujeme implementovat další komunikační protokol atd. Postup při vytváření vlastní knihovny si ukážeme na příkladu vlastní knihovny, pojmenované např. **MojeLIB**, se dvěma zcela jednoduchými celočíselnými funkcemi **secti(int a, int b)** a **nasob(int a, int b)**.

Nejprve vytvoříme hlavičkový soubor s prototypy našich knihovních funkcí. V menu vybereme **File** → **New** nebo stiskneme tlačítko **New** v toolbaru. Tím se otevře následující okno *obr. 6.1*.



Obr. 6.1

Vybereme **Source** a stiskneme tlačítko **OK**. Tím se otevře okno editoru otevřené pro zdrojový soubor untitled.c, do tohoto editačního okna napíšeme prototypy našich knihovních funkcí:

```
// tato #pragma direktiva zakáže překladači
// generovat upozornění, že funkce byla
// deklarovaná, ale nebyla použita v programu
#pragma used+
```

```
// prototypy knihovních funkcí
int secti(int a, int b);
int nasob(int a, int b);

#pragma used-

// tato #pragma direktiva sdělí překladači, aby

// přeložil/spojil funkce z knihovny MojeLIB.lib
#pragma library MojeLIB.lib
```

Uložíme ho pod novým jménem **MojeLIB.h** do adresáře **..\INC** pomocí výběru menu **File → Save As** *obr. 6.2.*

Obr. 6.2

Nyní vytvoříme již vlastní knihovní soubor obdobným způsobem jako při vytváření hlavičkového souboru.

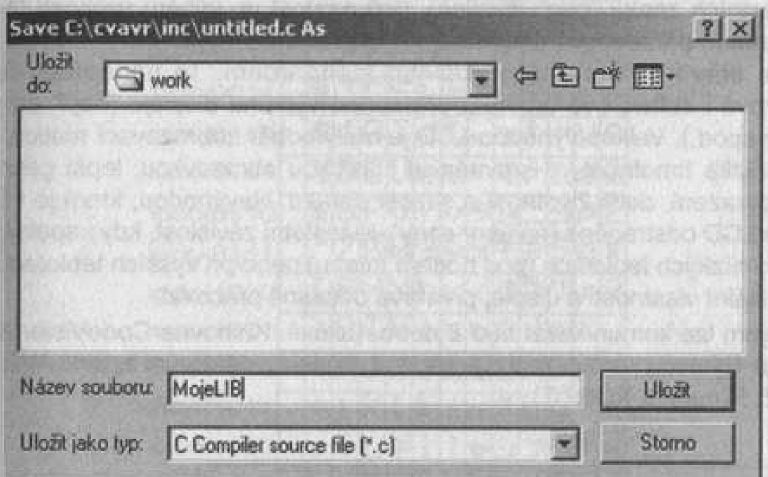
V menu vybereme **File → New** nebo stiskneme tlačítko **New** v toolbaru. Tím se otevře nám již známé okno **Create New File**, ve kterém opět vybereme volbu **Source** a stiskneme **OK**. Tím se otevře okno editoru otevřené pro zdrojový soubor **untitled.c**, do tohoto editačního okna napíšeme definice našich knihovních funkcí:

```
#if funcused secti
int secti(int a, int b) {
return a+b;
```



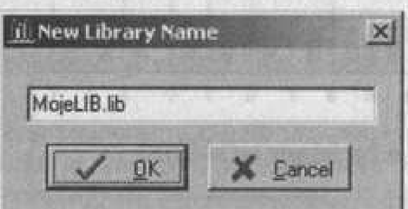
```
}
#endif

#if funcused nasob
int nasob(int a, int b) {
return a*b;
}
#endif
```

Direktivy preprocesoru **#if funcused** a **#endif** určují, že k překladu funkcí a jejich spojení (linkování) dojde jen v případě použití v programu. Uložíme definice funkcí do souboru **MojeLIB.c** do nějakého adresáře pomocí menu **File → Save As** *obr. 6.3.*



Obr. 6.3

Nakonec pomocí výběru menu **File → Convert to Library** uložíme tento soubor pod jménem **MojeLIB.lib** do adresáře **..\LIB** *obr. 6.4.*



Obr. 6.4

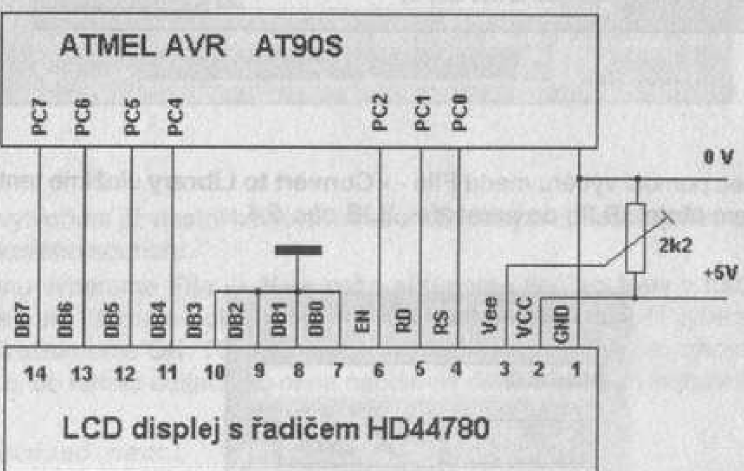
V případě využití této knihovny v programu musíme na jeho začátku uvést

```
#include <MojeLIB.h>
```

6.2 LCD displej a knihovna pro jeho ovládání z jazyka C

Velkého rozšíření mezi konstruktéry jako zobrazovací alfanumerické zařízení dosáhl inteligentní LCD displej s řadičem **HD44780** nebo jeho klon, takže všechny mají shodnou instrukční sadu a shodné ovládání. Tohoto rozšíření dosáhli díky výhodným vlastnostem. Je to především relativně snadné ovládání, možnost definice až osmi vlastních znaků (např. čeština), různorodost ve výběru velikosti (1 × 16 až 4 × 40 znaků) při zachování jednotného ovládání a v neposlední řadě se jejich specifikace stává defacto průmyslovým standardem. To usnadňuje ladění na hardwarové i softwarové úrovni a případnou výměnu displeje (např. za větší, podsvícený apod.). Velkou výhodou LCD je malý odběr zobrazovací matice, malé rozměry a nízká hmotnost ve srovnání s klasickou obrazovkou, lepší geometrie a ostrost zobrazení, delší životnost a stálost obrazu. Nevýhodou, která je však již u některých LCD odstraněna (na úkor ceny), je teplotní závislost, kdy kapalné krystaly při velmi nízkých teplotách (pod bodem mrazu) nebo při vyšších teplotách ztrácejí své fyzikální vlastnosti a displej přestává dočasně pracovat.

S displejem lze komunikovat buď 8 nebo 4bitově. Knihovna CodeVisionAVR C předpokládá 4bitovou komunikaci a propojení displeje s některým z portů AVR MCU podle *obr. 6.5* (např. k PORTC).



Obr. 6.5 Komunikace s LCD displejem pomocí 4bitů dat

Při této 4bitové komunikaci slouží DB4, DB5, DB6 a DB7 pro přenos instrukcí do řadiče LCD a pro zápis či čtení dat do/z paměti kódů znaků (DDRAM) nebo paměti uživatelských znaků (CGRAM). To, zda se zapisují data či instrukce je dáno signálem na RS (0 … vstup instrukce, 1 … Data), signál RD určuje zápis/čtení do LCD a EN znamená **platná data**. Pomocí HD44780 můžeme na displej zobrazovat pouze znakovou sadou uloženou v paměti ROM na řadiči, která je pro řadič vlastní viz *obr. 6.6*.

	0000	0010	0011	0100	0101	0110	0111	1010	1011	1100	1101	1110	1111
xxxx0000			0	@	P	`	p		-	タ	ミ	α	p
xxxx0001		!	1	A	Q	a	q	。	ア	チ	ム	ä	q
xxxx0010		"	2	B	R	b	r	「	イ	ツ	メ	β	θ
xxxx0011		#	3	C	S	c	s	」	ウ	テ	モ	ε	∞
xxxx0100		$	4	D	T	d	t	、	エ	ト	ヤ	μ	Ω
xxxx0101		%	5	E	U	e	u	・	オ	ナ	ユ	σ	ü
xxxx0110		&	6	F	V	f	v	ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111		'	7	G	W	g	w	ァ	キ	ヌ	ラ	g	π
xxxx1000		(	8	H	X	h	x	ィ	ク	ネ	リ	√	x̄
xxxx1001		)	9	I	Y	i	y	ゥ	ケ	ノ	ル	⁻¹	y
xxxx1010		*	:	J	Z	j	z	ェ	コ	ハ	レ	j	千
xxxx1011		+	;	K	[	k	{	ォ	サ	ヒ	ロ	×	万
xxxx1100		,	<	L	¥	l	|	ャ	シ	フ	ワ	¢	円
xxxx1101		-	=	M	]	m	}	ュ	ス	ヘ	ン	£	÷
xxxx1110		.	>	N	^	n	→	ョ	セ	ホ	゛	ñ	
xxxx1111		/	?	O	_	o	←	ッ	ソ	マ	゜	ö	█

Obr. 6.6 Znaková sada řadiče HD44780

Instrukce pro řadič LCD jsou v CodeVision C generovány funkcemi popsanými v kapitole 5.9. Předpokladem je ovšem propojení AVR MCU a LCD displeje podle *obr. 6.5*.

Knihovna je tvořena souborem **LCD.lib** umístěným v adresáři **..\LIB**, příslušný hlavičkový soubor **lcd.h** je umístěn v adresáři **..\INC**. Ukážeme si teď jednoduchý způsob úpravy této knihovny pro jiné propojení LCD displeje s portem mikrokontroléru AVR. Novou knihovnu pojmenujeme např. LCD1. V adresáři **..\LIB** umístíme kopii souboru **LCD.lib** pod novým jménem **LCD1.lib**. Obdobně v podadresáři **..\INC**

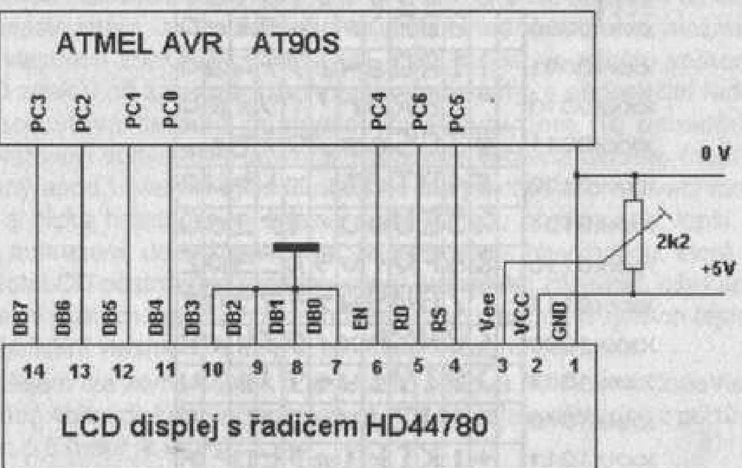
kopii hlavičkového souboru lcd.h pod názvem **lcd1.h**. V novém hlavičkovém souboru **lcd1.h** provedeme jedinou změnu. Najdeme na jeho konci řádku s kódem

```
#pragma library lcd.lib
```

který přepíšeme na

```
#pragma library lcd1.lib
```

Úprava obsahu souboru **LCD1.lib** je závislá na konkrétním propojení LCD s portem. Tyto úpravy si ukážeme pro zapojení *obr. 6.7.*



Obr. 6.7 Zapojení podporované knihovnou E_LAB PASCAL

Poznámka: *Konkrétně jde o zapojení vyhovující knihovně PASCAL od německé firmy E-LAB. Budeme-li mít toto zapojení, můžeme své aplikace s LCD ATMEL AVR programovat jak v CodeVisionAVR C, tak v E-Lab PASCALu bez jakýchkoli zásahů do zapojení AVR MCU a LCD.*

Porovnáme-li obě zapojení, tj. propojení AVR MCU a LCD, vidíme, že se liší tím, že data/instrukce do řadiče LCD jsou přenášena v jednom zapojení dolními čtyřmi bity portu AVR, v druhém zapojení horními čtyřmi bity tohoto portu. Proto na vhodných místech asm kódu v **lcd1.lib** pomocí instrukce SWAP provedeme prohození těchto čtveřic bitů. Další odlišnost obou zapojení je dána tím, ke kterým pinům portu AVR MCU jsou připojeny signály **EN**, **RD** a **RS**. Připojení těchto signálů ošetříme změnou konstant uvedenou na začátku LCD.lib:

```
#asm
    .equ __lcd_direction=__lcd_port-1
    .equ __lcd_pin=__lcd_port-2
    .equ __lcd_rs=0
    .equ __lcd_rd=1
    .equ __lcd_enable=2
    .equ __lcd_busy_flag=7
#endasm
```

na kód (v LCD1.lib) :

```
#asm
    .equ __lcd_direction=__lcd_port-1
    .equ __lcd_pin=__lcd_port-2
    .equ __lcd_rs=5
    .equ __lcd_rd=6
    .equ __lcd_enable=4
    .equ __lcd_busy_flag=3
#endasm
```

Řádky, v nichž se provádějí změny jsou vytištěny tučně. Další změny se již týkají jen přehození horní a dolní čtveřice bitů v portu pro přenos dat/instrukcí řadiče LCD. Protože se ale v instrukcích AVR jako operandy používají byty a nikoli jen jejich poloviny, používá se v kódu **lcd.lib** maskování hodnotou 0xF0 popř. 0x0F. V souboru **lcd1.lib** se tyto masky změní. Tam, kde bylo 0xF0 bude 0x0F, obdobně 0x0F se nahradí 0xF0. Jde konkrétně o tyto úseky kódu:

Kódu v **lcd.lib**

```
void _lcd_ready(void)
{
#asm
    in r26,__lcd_direction
    andi r26,0xf            ;set as input
```

bude v **lcd1.lib** odpovídat

```
void _lcd_ready(void)
{
#asm
```

```
    in r26,__lcd_direction
    andi r26,0xF0                  ;set as input
```

Obdobně další úsek z **lcd.lib**

```
#asm
__lcd_write_nibble:
    andi r26,0xf0
```

nahradí v **lcd1.lib** úsek

```
#asm
__lcd_write_nibble:
    andi r26,0x0F0
```

Dále se změní funkce (z **lcd.lib**):

```
void _lcd_write_data(unsigned char data)
{
#asm
    cbi __lcd_port,__lcd_rd        ;RD=0
    in r26,__lcd_direction
    ori r26,0xf7                   ;set as output
    out __lcd_direction,r26
    in r27,__lcd_port
    andi r27,0xf
    ld r26,y
    rcall __lcd_write_nibble       ;RD=0, write MSN
    ld r26,y
    swap r26
    rcall __lcd_write_nibble       ;write LSN
    sbi __lcd_port,__lcd_rd        ;RD=1
#endasm
}
```

na (v **lcd1.lib**):

```
void _lcd_write_data(unsigned char data)
{
#asm
```



```
    cbi __lcd_port,__lcd_rd            ;RD=0
    in r26,__lcd_direction
    ori r26,0x7f                       ;set as output
    out __lcd_direction,r26
    in r27,__lcd_port
    andi r27,0xf0
    ld r26,y
    swap r26
    rcall __lcd_write_nibble           ;RD=0, write MSN
    ld r26,y
    nop
    rcall __lcd_write_nibble           ;write LSN
    sbi __lcd_port,__lcd_rd            ;RD=1
#endasm
}
```

Předposledním úsekem z **lcd.lib**, který se bude měnit je

```
#if funcused lcd_read_byte
#asm
__lcd_read_nibble:
    sbi __lcd_port,__lcd_enable ;EN=1
    rcall __lcd_delay
    in r30,__lcd_pin                ;read
    cbi __lcd_port,__lcd_enable ;EN=0
    rcall __lcd_delay
    andi r30,0xf0
    ret
#endasm
```

kde se přehodí **0xf0 na 0x0f v lcd1.lib**

```
#if funcused lcd_read_byte
#asm
__lcd_read_nibble:
    sbi __lcd_port,__lcd_enable ;EN=1
    rcall __lcd_delay
    in r30,__lcd_pin                ;read
    cbi __lcd_port,__lcd_enable ;EN=0
    rcall __lcd_delay
    andi r30,0x0F
    ret
#endasm
```

a konečně poslední měněný úsek kódu **lcd.lib**

```
unsigned char lcd_read_byte(unsigned char addr)
{
_lcd_ready();
_lcd_write_data(addr);
_lcd_ready();
#asm
    in r26,__lcd_direction
    andi r26,0xf                    ;set as input
    out __lcd_direction,r26
    sbi __lcd_port,__lcd_rs         ;RS=1
    rcall __lcd_delay
    rcall __lcd_read_nibble         ;read MSN
    mov r26,r30
    rcall __lcd_read_nibble         ;read LSN
    cbi __lcd_port,__lcd_rd         ;RD=0
    swap r30
    or r30,r26
#endasm
}
```

se v **lcd1.lib** přepíše na

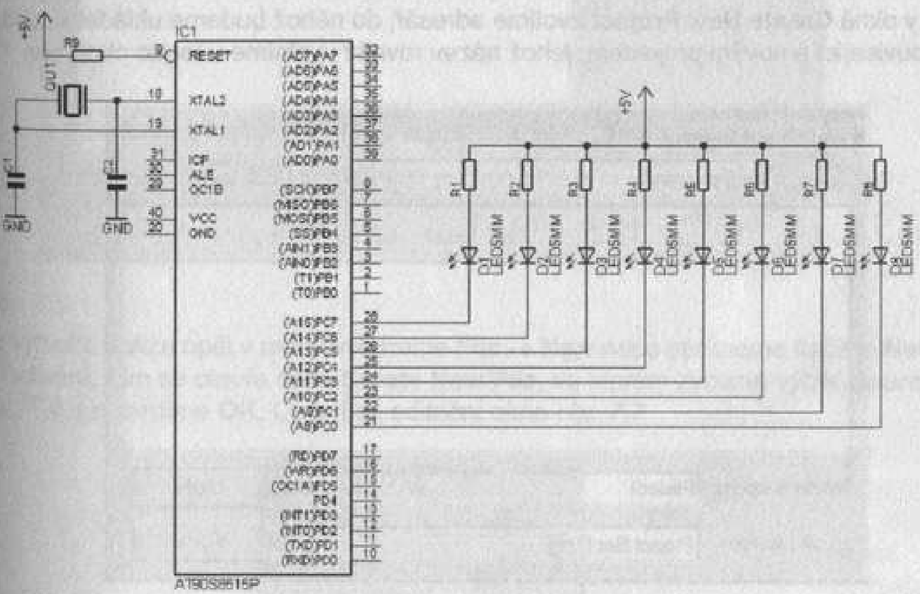
```
unsigned char lcd_read_byte(unsigned char addr)
{
_lcd_ready();
_lcd_write_data(addr);
_lcd_ready();
#asm
    in r26,__lcd_direction
    andi r26,0xf0                   ;set as input
    out __lcd_direction,r26
    sbi __lcd_port,__lcd_rs         ;RS=1
    rcall __lcd_delay
    rcall __lcd_read_nibble         ;read MSN
    mov r26,r30
    swap r26
    rcall __lcd_read_nibble         ;read LSN
    cbi __lcd_port,__lcd_rd         ;RD=0

    or r30,r26

#endasm
}
```

7

PŘÍKLADY

7.1 Program 1 – ovládání LED diod, blikač

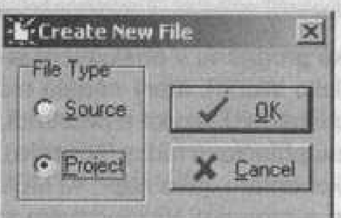
Učebnice programovacích jazyků jako první ukázku programu v příslušném jazyce nejčastěji začínají programem *Nazdar světe* (Hello World), který nedělá nic jiného, než vypsání nápisu na obrazovku a to podle charakteru jazyka a prostředí, pro které je příslušný program určen, nejčastěji v režimu příkazové řádky (konzole), okna či www stránky. V případě programování mikrokontroléru, jako např. ATMEL AVR, je možnost začít ještě jednodušším programem a program *Nazdar světe* si ponecháme až jako druhý program ukazující programování výstupu na displej LCD. V prvním programu si ukážeme pouhé posílání dat na I/O bránu, např. PORTC, ke kterému bude připojeno 8 diod LED.



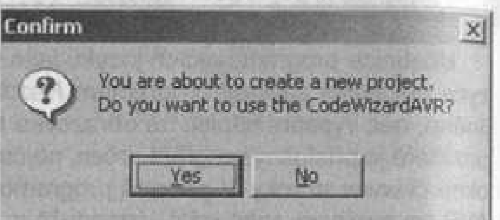
Obr. 7.1 *Principiální zapojení mikrokontroléru s LED – příklad č. 1*

V tomto zapojení bude LED svítit, když PORTC bude na příslušném pinu mít 0, a bude zhasnutá, bude-li tento signál na úrovni 1. Hodnota jednotlivých bitů v bytu poslaném na výstup PORTC pak určí, které LED budou svítit a které nikoli. Použití mikrokontroléru na rozsvícení několika LED bez jakékoli další činnosti není rozhodně typickou aplikací využití MCU, tak si naprogramuje alespoň jednoduchý blikač, např. střídavé rozsvícení sudých a lichých LED po 500 ms. U tohoto prvního programu si ukážeme celý postup při vytváření projektu, u dalších programů si uvedeme již jen zdrojový kód a vysvětlíme si jeho jednotlivé části.

Nejprve vytvoříme nový projekt. V menu vybereme **File** → **New** nebo stiskneme tlačítko **New** v toolbaru. Tím se otevře okno **Create New File**, ve kterém zvolíme výběr **Project** *obr. 7.2* a stiskneme tlačítko **OK**. Poté se objeví okno *obr. 7.3*.

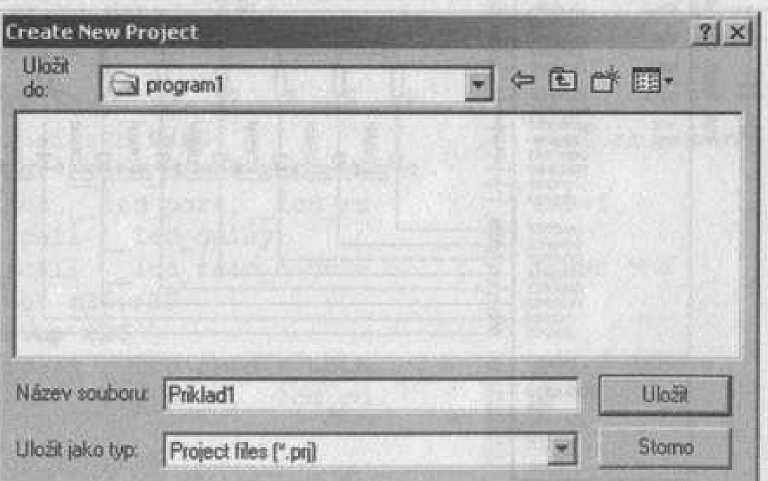


Obr. 7.2



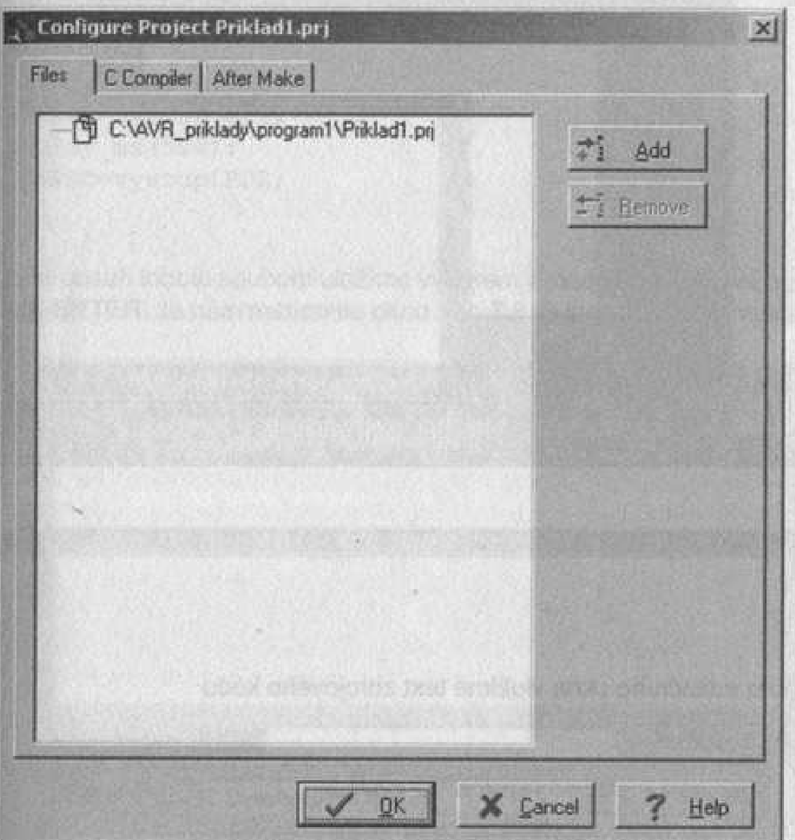
Obr. 7.3

V něm zvolíme volbu **No** (návrh pomocí CodeWizarda si ukážeme později) a v okně **Create New Project** zvolíme adresář, do něhož budeme ukládat soubory související s novým projektem, jehož název rovněž vyplníme v tomto okně *obr. 7.4*



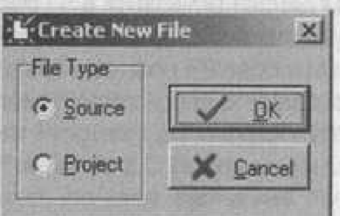
Obr. 7.4

a pak potvrdíme tlačítkem **Uložit**. Následně se objeví nové okno **Configure Project** *obr. 7.5* ve kterém vybereme záložku **C Compiler** a potom zvolíme **Chip** AT90S8515, **Clock** 8 MHz a **File Output Form** Intel Hex a potvrdíme **OK**. Do tohoto nového projektu vložíme soubor se zdrojovým kódem v C. Tento soubor ale musíme nejdříve vytvořit.

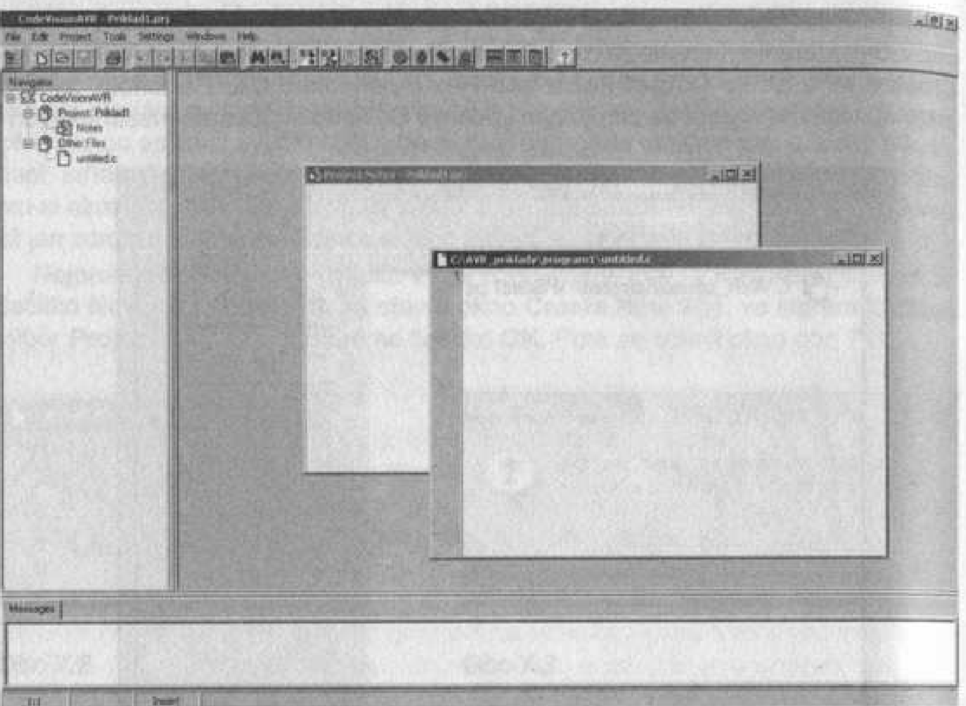


Obr. 7.5

Takže opět v menu vybereme **File** → **New** nebo stiskneme tlačítko **New** v toolbaru. Tím se otevře okno **Create New File**, ve kterém zvolíme výběr **Source** *obr. 7.6* a potvrdíme **OK**. Objeví se editační okno *obr. 7.7*.



Obr. 7.6



Obr. 7.7

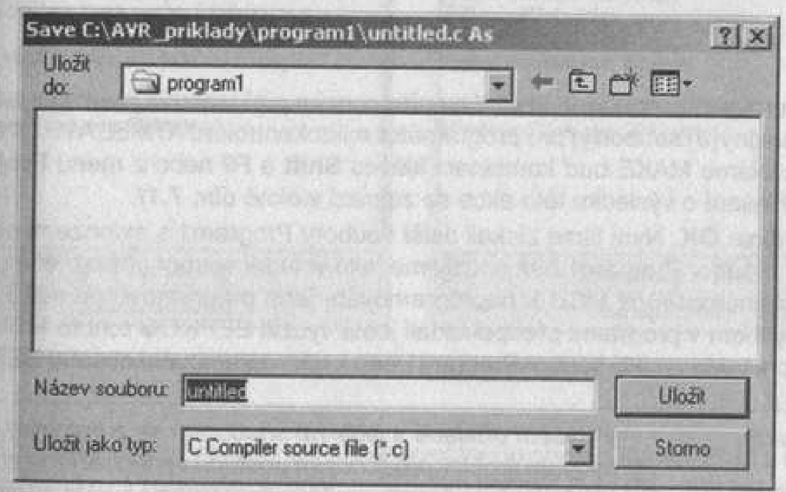
Do tohoto editačního okna vložíme text zdrojového kódu

```
/* program1
blikani LED diod připojených k portu C
memory model SMALL
pouzity MCU AT90S8515
*/

#include <90s8515.h>

#define xtal 8000000

#include <Delay.h>

unsigned char vystupLED1;
unsigned char vystupLED2;
```

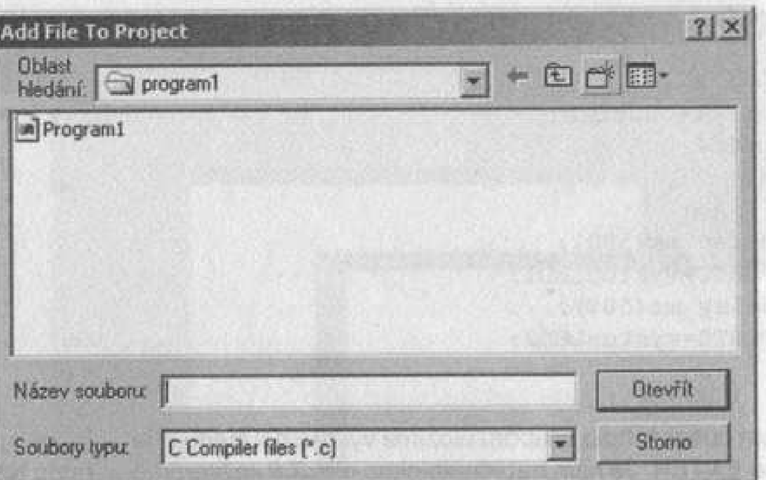


```
void main(void)
{
DDRC=0xff;
vystupLED1=0x55;
vystupLED2=0xAA;
while(1)
   {
      delay_ms(500);
      PORTC=vystupLED1;
      delay_ms(500);
      PORTC=vystupLED2;
   };
}
```

Potom obsah tohoto souboru uložíme výběrem v menu **File** → **Save** a potvrdíme klávesou **ENTER**. Je nám nabídnuto okno *obr. 7.8* ve kterém vyplníme **Název souboru**: Program1 a stiskneme tlačítko **Uložit**. Nyní zbývá tento soubor přidat do projektu. Proto v menu vybíráme **Project** → **Configure Project** a poté v okně **Configure Project** stiskneme tlačítko **Add** a poté v takto vyvolaném okně *obr. 7.9* vybereme soubor Program1.c a stiskneme tlačítko **Otevřít**. Potom ještě potvrdíme **OK** v okně **Configure Project** a vše ještě uložíme výběrem **File** → **Save All**. Nyní již můžeme zavolat překladač klávesou F9 nebo v menu **Project** → **Compiler**. Poté proběhne překlad zakončený informacemi překladače v okně *obr. 7.10* a potvrdíme **OK**. Nyní kromě souboru Program1.prj a zdrojového souboru Program1.c máme další soubory vzniklé při překladu z C do assembleru, tj. máme ještě další soubory Program1 s extenzemi *asm, inc, map, sym* a *vec*. Nyní zbývá ještě provést překlad



Obr. 7.8



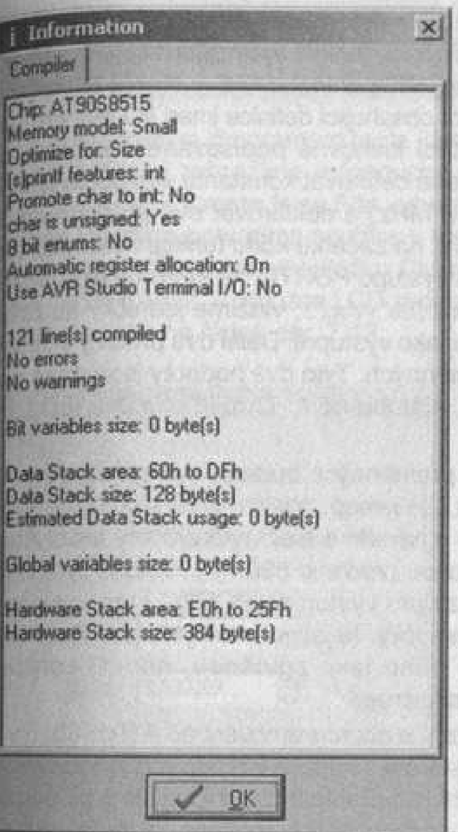
Obr. 7.9

assemblerovského souboru, vzniklého předchozím překladem z jazyka C, a tím vytvořit výsledný(é) soubor(y) pro programátor mikrokontroléru ATMEL AVR. K tomuto účelu zavoláme MAKE buď kombinací kláves **Shift** a **F9** nebo z menu **Project → Make** a hlášení o výsledku této akce se zobrazí v okně *obr. 7.11*.

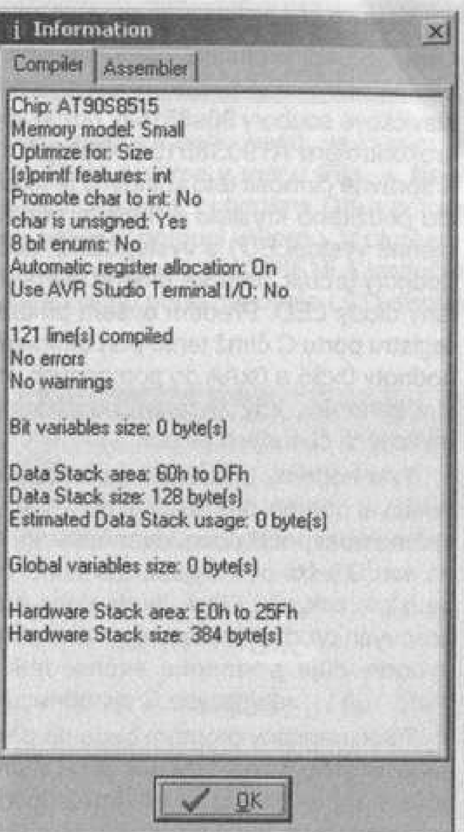
Potvrdíme **OK**. Nyní jsme získali další soubory Program1 s extenzemi *eep*, *err*, *hex* a *lst*. Soubor Program1.hex použijeme jako vstupní soubor obslužného programu programátoru AVR MCU k naprogramování jeho programové paměti FLASH. Pokud bychom v programu předpokládali ještě využití EEPROM tohoto MCU, použili bychom takto vyniklý soubor Program1.eep k naprogramování obsahu EEPROM.

Poznámka: *V našem příkladě s jediným souborem se zdrojovým kódem bychom mohli v předchozím postupu vynechat krok spouštění překladače (např. např. pomocí F9) a provést přímo spuštění **make**. Tím by se provedl jak překlad z C do asm, tak překlad asm. Postup se spouštěním překladu použijeme, máme-li několik zdrojových souborů, takže tyto soubory překládáme jednotlivě a tím šetříme čas při překladu. Takto totiž vždy překládáme jen jeden, právě odlaďovaný soubor. Při provádění Make by se vždy překládaly všechny soubory.*

Naprogramování mikrokontroléru ATMEL AVR teď můžeme spustit z prostředí CodeVisionAVR C, máme-li k dispozici jeden z šesti podporovaných programátorů. Výběr programátoru se provádí výběrem menu **Settings → Programmer**, ovládání



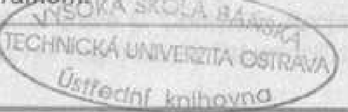
Obr. 7.10



Obr. 7.11

činnosti programátoru provádíme z okna vyvolaného pomocí kláves **Shift** a **F4**, nebo z menu **Tools → Programers**, popř. stisknutím tlačítka **Run the chip programmer** na toolbaru.

Poznámka: *Při výběru typu programátoru máme možnost zvolit jeden z šesti typů. Z nich bude zřejmě nejdostupnější typ označený **Kanda Systems STK200+/300**, který lze realizovat i amatérsky. Je to jednoduchý interface obsahující jediný IO 74LS244, zapojený mezi port pro tiskárnu a programovaný mikrokontrolér. Další možností je použít některý z programátorů podporovaných **AVR Studiem**, popř. známým PonnyProg či jiným programem.*



Zbývá si ještě popsat zdrojový kód našeho prvního, velice jednoduchého programu. Obsahuje jedinou funkci, což je v jazyce C povinná funkce **main**, sloužící jako vstupní bod programu. Na začátku kódu jsou do něj ještě pomocí direktivy **#include** vloženy hlavičkové soubory **90s8515.h**, což je soubor obsahující definice jmen I/O registrů mikrokontroléru AT90S8515 a **Delay.h**, patřící knihovně popisované v kapitole 5. K správné činnosti této knihovny je ještě třeba definovat konstantu xtal podle kmitočtu použitého krystalu (v našem případě 8 MHz) a deklarovat dvě pomocné proměnné vystupLED1 a vystupLED2, do nichž na začátku kódu funkce main vložíme hodnoty určující logické úrovně signálu na výstupu PORTC, ke kterému jsou připojeny diody LED. Předtím ovšem příkazem `DDRC=0xff;` vložíme jedničky do řídicí registru portu C čímž tento port definujeme jako výstupní. Další dva příkazy vkládají hodnoty 0x55 a 0xAA do pomocných proměnných. Tyto dvě hodnoty jsou dané naším zadáním, kdy chceme, abychom na výstupu portu C měli střídavě jedničky na lichých či sudých pinech.

Tyto hodnoty obsažené v pomocných proměnných budeme na výstup portu C dostávat pomocí příkazu `PORTC=vystupLED1;` resp. `PORTC=vystupLED2;`. Abychom zabezpečili požadované blikání 0,5 s, vřadíme mezi tyto příkazy pro výstup na port C ještě příkazy zabezpečující časovou prodlevu 500 ms. Protože požaduje neustálé blikání LEDek, budou oba příkazy pro výstup na PORTC i pro vytvoření časových prodlev vloženy do nekonečné smyčky. Ta je tvořena cyklem **while**, který vyhodnocuje podmínku svého dalšího běhu jako splněnou, neboť jednička v `while(1)` se v jazyce C vyhodnocuje jako **true**.

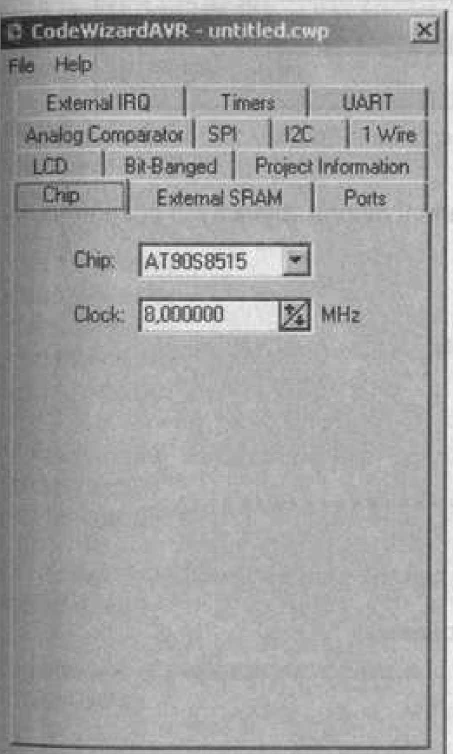
Takto napsaný program bude po přeložení a naprogramování do AT90S8515 vykonávat svou činnost tak, jak jsme si předsevzali, i když je v programu jedna chyba, způsobená snahou o co největší jednoduchost a přehlednost. V kapitole 5 při popisu knihovny Delay jsme se zmínili, že získání požadovaných zpoždění *je třeba zakázat všechna přerušení*. Takže bychom před nekonečnou smyčkou `while(1)` umístili `#asm("cli");`.

Řadě podobných chyb se můžeme vyhnout při návrhu pomocí wizarda.

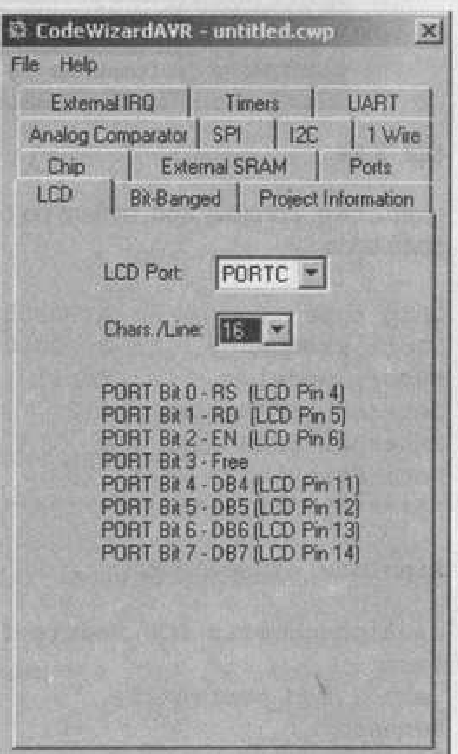
Poznámka: *I takto napsaný program by někteří programátoři napsali trochu jinak. Např. pomocné proměnné by definovali již při deklaraci, či místo proměnných by použili konstanty. Obdobně i v následujících programech jistě najdete místa, která by bylo možné napsat lépe, stručněji, stylem „opravdových programátorů C", což ovšem obvykle nevede k přehlednému a srozumitelnému kódu, kterému se budeme snažit dávat přednost.*

7.2 Program 2 – vyslání nápisu na LCD displej

Naším druhým programem bude již oblíbený program Hello world, ve kterém si na LCD displej zobrazíme jednoduchý nápis. Opět zvolíme v menu **File** → **New** a potom v okně **Create New File** vybereme volbu **Project**, potvrdíme **OK** a potom v okně **Confirm** potvrdíme souhlas s tvorbou projektu pomocí wizardu. V okně wizardu *obr. 7.12* vybereme záložku **Chip** a vybereme obvod AT90S8515 a kmitočet jeho hodin 8 MHz. V záložce LCD zvolíme, ke kterému portu připojíme LCD displej a počet znaků na řádek *obr. 7.13*.

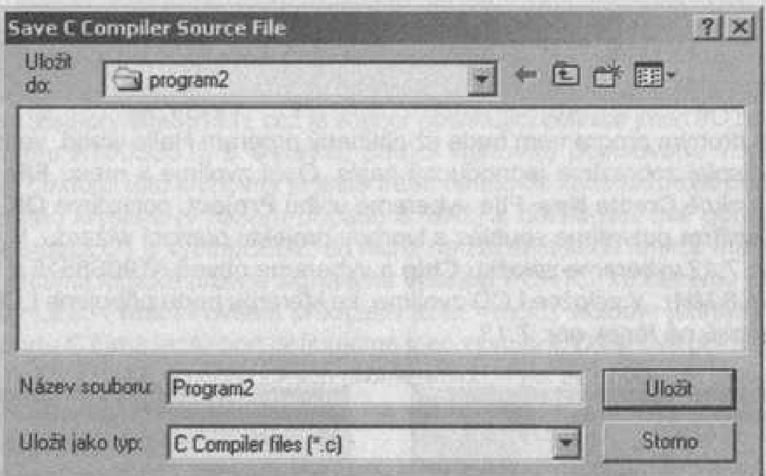


Obr. 7.12



Obr. 7.13

Zvolili jsme PORTC s 16 znaky na řádek. Poté v tomto okně, s hlavičkou **CodeWizardAVR – untitled.cwp vybereme File → Generate, Save a Exit** a v okně *obr. 7.14* zvolíme adresář pro náš druhý projekt, název souboru Program2 a stiskneme **Uložit**. Obdobně budeme volit názvy u dalšího okna **Save C Compiler Project file** i okna **Save untitled.swp as**. Tím získáme soubory Program2.prj, Program2.c a Pro-



Obr. 7.14

gram2.cwp. Zdrojový kód, ovšem po odstranění mnoha zbytečných poznámek, vypadá takto

```
Chip type           : AT90S8515
Clock frequency     : 8,000000 MHz
Memory model        : Small
Internal SRAM size  : 512
External SRAM size  : 0
Data Stack size     : 128
*********************************************/

#include <90s8515.h>

// Alphanumeric LCD Module functions
#asm
.equ __lcd_port=0x15
#endasm
#include <lcd.h>

void main(void)
{

// Port A initialization
PORTA=0x00;
DDRA=0x00;

// Port B initialization
PORTB=0x00;
DDRB=0x00;

// Port C initialization
PORTC=0x00;
DDRC=0x00;

// Port D initialization
PORTD=0x00;
DDRD=0x00;

// Timer/Counter 0 initialization
TCCR0=0x00;
TCNT0=0x00;

// Timer/Counter 1 initialization
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// External Interrupt(s) initialization
GIMSK=0x00;
MCUCR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x00;

// Analog Comparator initialization
ACSR=0x80;

// LCD module initialization
lcd_init(16);

while (1)
  {

    // Place your code here

  };
}
```

Tento kód doplníme dvěma řádky umístěnými před nekonečnou smyčku na konci.

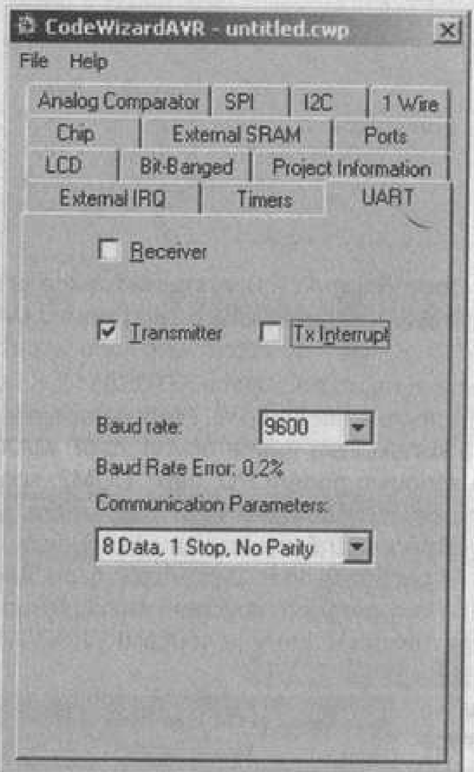
```
lcd_init(16);
lcd_gotoxy(0,0);
lcd_putsf("Ahoj");
while (1)
      {
      // Place your code here

      };
}
```

Nově přidané řádky jsou vytištěny silně. Vidíme, že na začátku zdrojového kódu wizard umístil direktivu vyžadovanou knihovnou LCD a informující, ke kterému portu bude připojen LCD. Teprve potom umístil direktivu s hlavičkovým souborem **LCD.h**. Na začátku hlavní funkce main pak umístil inicializace všech portů PORTA až PORTD, časovačů/čítačů, analogového komparátoru i obsah registrů souvisejících s vnějším přerušením. Veškeré tyto inicializace na začátku funkce main umisťuje wizard i v případě, jako je ten náš, kdy je vpodstatě nepotřebujeme a které bychom při tvorbě zdrojového kódu jen pomocí editoru asi nepoužili. Inicializace LCD displeje funkcí `lcd_init(16)` je ovšem nutná. Příkaz `lcd_gotoxy(0,0);` nastaví aktuální pozici na LCD displeji na první řádek, první sloupec. *Řádky i sloupce se však počítají od nuly.* Následuje příkaz `lcd_putsf("Ahoj");` provádějící již výstup textového řetězce na LCD. Následuje prázdná, nekonečná smyčka, vytvořená wizardem a o níž se předpokládá, že do ní programátor vloží nějaký kód. V našem jednoduchém programu žádný další kód již nepotřebujeme, jediný úkol a to vypsání nápisu na LCD je splněn. Přesto je nekonečná, třeba prázdná, smyčka na konci programu nutností. Museli bychom ji vytvořit i v případě psaní celého kódu jen editorem, bez pomoci wizarda. Překlad z C se nejprve provede do assembleru a i v něm pak bude na konci nekonečná smyčka. Pokud by na konci takováto nekonečná smyčka nebyla, pokračovalo by po vypsání nápisu na LCD prováděním dalších strojových instrukcí. Mikrokontrolér totiž jako kód instrukcí bude chápat i obsah dalších paměťových buněk v programové paměti FLASH, umístěných za námi napsaným programem. Neošetření zakončení programu nějakým definovaným způsobem, jako např. je tato smyčka, by mohlo způsobovat problémy.

7.3 Program 3 – vyslání řetězce znaků na RS232

Jako třetí program si uvedeme další variantu programu Hello world, tentokrát vyslání řetězce znaků UARTem přes rozhraní RS232 třeba do sériového portu počítače PC a zobrazení tohoto nápisu v nějakém terminálovém programu na PC. Vytvoříme projekt Program3 opět pomocí wizarda, kde tentokrát v záložce UART

provedeme volbu přenosové rychlosti, dále zda UART budeme používat jako vysílač, přijímač či budeme UART využívat pro vysílání i příjem. Dále můžeme zvolit, zda budeme při vysílání využívat přerušení. Volba použití přerušení při vysílání však u školní verze CodeVisionAVR C není wizardem podporována, takže bychom si museli takový kód napsat sami *obr. 7.15*.



Obr. 7.15

Potom v menu vybereme **File → Generate**, **Save** a **Exit** a stejným způsobem jako v předchozím programu zvolíme názvy souborů, které potom wizard vytvoří. Třebaže wizard opět vytvoří množství zbytečných inicializací, vytvoří i potřebnou inicializaci pro UART

```
// UART initialization
// Communication Parameters: 8 Data, 1 Stop, No Parity
// UART Receiver: Off
// UART Transmitter: On
// UART Baud rate: 9600
```

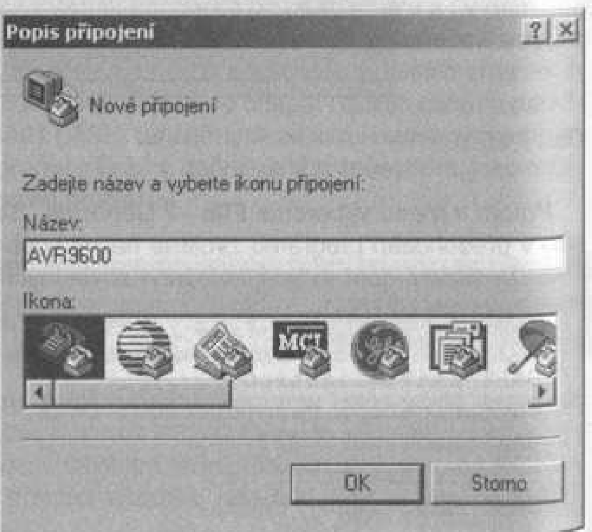
```
UCR=0x08;
UBRR=0x33;
```

My již jen doplníme příkaz pro odeslání řetězce znaků a umístíme ho před nekonečnou smyčku.

```
printf("Ahoj");
while (1)
      {
      // Place your code here

      };
}
```

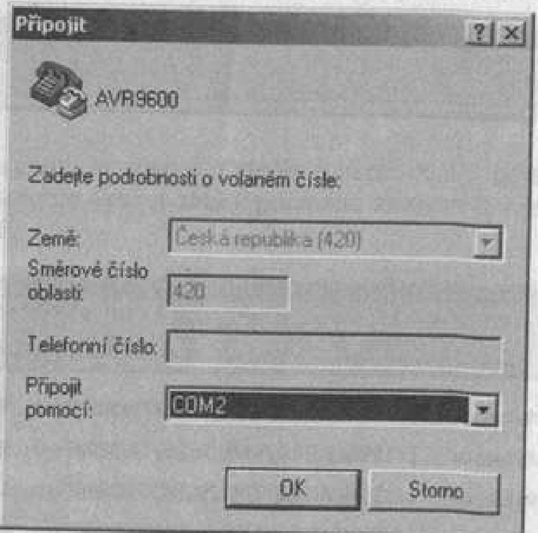
Využili jsme toho, že **u CodeVisionAVR C výstupní funkce printf**, která je součástí knihovny standardních I/O funkcí **provádí výstup pomocí UARTu**. Potřebnou direktivu `#include <stdio.h>` vloží do zdrojového kódu wizard.

Program přeložíme a potom naprogramujeme AT90S8515. K ověření funkčnosti programu spojíme náš AVR mikrokontrolér s PC. Předpokladem je, že mezi vývody UARTu, tj. piny PORTD a konektorem máme obvod, např. MAX232, převádějící úrovně TTL a RS232. Se sériovým portem PC, např. COM2, spojíme náš RS232 konektor kabelem – tzv. nulovým modemem. Vyznačuje se tím, že vodiče signálů RxD a TxD na vysílací a přijímací straně jsou vzájemně propojeny, takže např. vysílaná data z TxD desky s AVR přicházejí do RxD přijímače, tj. počítače PC. Nyní ještě zbývá spustit nějaký terminálový program, abychom mohli přijímat data od AVRky. Můžeme např. použít **Hyperterminál**, který je součástí WINDOWS. Spustíme ho výběrem z menu **Start → Programy → Příslušenství → Komunikace → Hyperterminál**. Program hyperterminál nám navrhne možnost vytvořit *nové připojení obr. 7.16.*

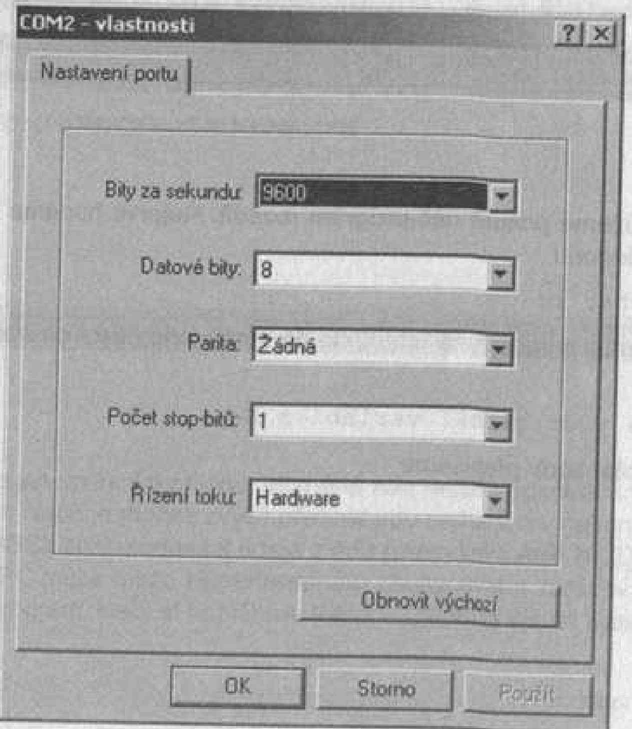


Obr. 7.16

Pojmenovali jsme ho AVR9600 (v C programu jsme totiž zvolili rychlost 9600 Bd) a potvrdíme **OK**. Poté nastavíme port PC, ke kterému máme připojen AVR *obr. 7.17* a nastavíme i rychlost přenosu *obr. 7.18*

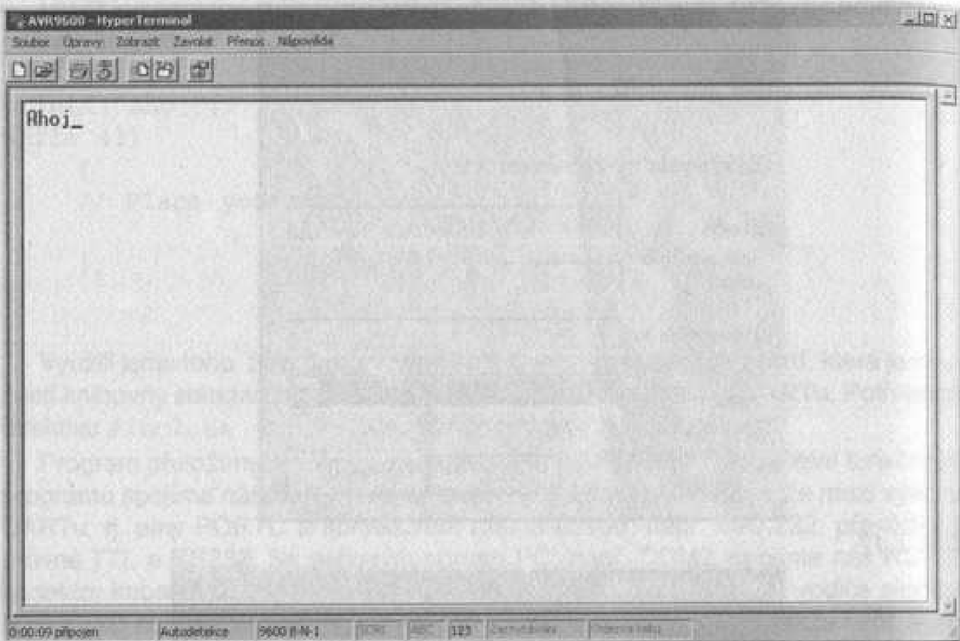


Obr. 7.17



Obr. 7.18

Opět potvrdíme **OK**. Máme-li k PC připojen náš naprogramovaný AVR, můžeme spustit náš program v AVR MCU (*RESETem* nebo připojením napájení) a v *okně* Hyperterminálu uvidíme přijatý text *obr. 7.19*.



Obr. 7.19

Teď se můžeme pokusit náš program rozšířit. Nejprve budeme deklarovat pomocnou proměnnou

```
unsigned char c;
```

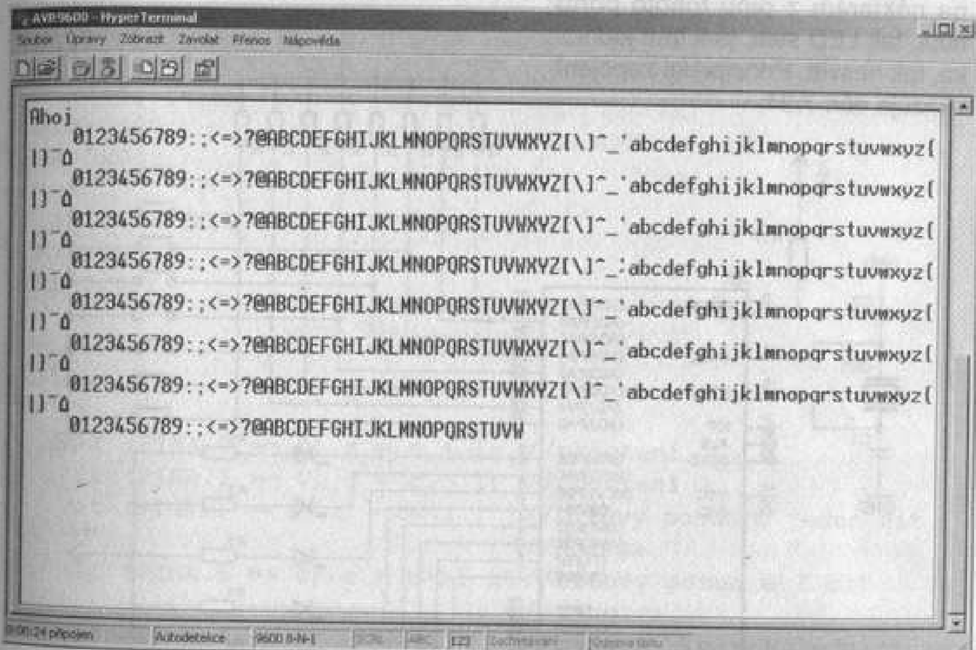
Tuto deklaraci samozřejmě umístíme do správného místa na začátku zdrojového kódu

```
// Declare your global variables here
```

a výkonnou část kódu přepíšeme

```
printf("Ahoj");
printf("\n");
c = '0';
while (1)
      {
      putchar(c);
      c++;
      if (c> 127)
         {
         c = '0';
         printf("\n");
         }
      };
}
```

Činnost programu je zřejmá. Odešle řetězec `Ahoj`, odřádkuje a potom bude v nekonečné smyčce vysílat znaky počínajíc znakem 0 v jejich pořadí v tabulce ASCII *obr. 7.20*.

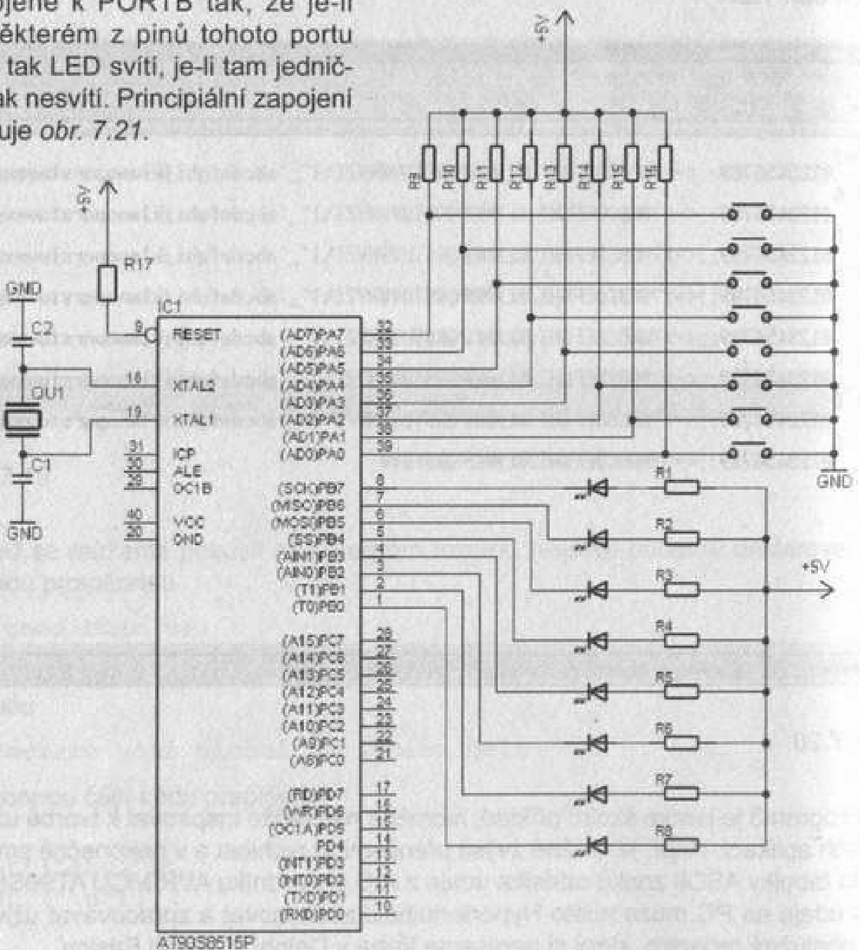


Obr. 7.20

Program3 je jenom školní příklad, nicméně nás může inspirovat k tvorbě užitečnějších aplikací. Např. je možné zvýšit přenosovou rychlost a v nekonečné smyčce místo tabulky ASCII znaků odesílat údaje z A/D převodníku AVR MCU AT90S8535. Tyto údaje na PC může místo Hyperterminálu zachycovat a zpracovávat uživatelský obslužný program, který si napíšeme třeba v Delphi či Visual Basicu.

7.4 Program 4 – vstupy z tlačítek

V programu1, programu2 a programu3 jsme zatím programovali výstup na PORT, LCD displej či jsme výstupní data vysílali sériově UARTem. V dalších programech si ukážeme, jak naopak provádět vstup. Pro školní účely se obvykle používají různé startkity, jako např. STK500, u kterých vstupem budou tlačítka, výstupem diody LED. V dalším programu si ukážeme jak se dá v jazyce C programově obsloužit takový jednoduchý startkit. Předpokládejme, že k PORTA máme připojena tlačítka tak, že v klidovém stavu jsou na PORTA samé jedničky, po stisknutí některého tlačítka se potom na příslušném pinu tohoto portu objeví nula. Jako výstup slouží LEDky připojené k PORTB tak, že je-li na některém z pinů tohoto portu nula, tak LED svítí, je-li tam jednička, tak nesvítí. Principiální zapojení ukazuje *obr. 7.21*.



Obr. 7.21 – Principiální zapojení mikrokontroléru s tlačítky a LED – příklad č. 4

Napíšeme jednoduchý program, inspirovaný ukázkovým programem v assembleru u startkitu STK500. Jeho úkolem je v nekonečné smyčce zjišťovat stav tlačítek a na stisknutí některého z tlačítkem nějak reagovat, např. inkrementací, dekrementací, rotací apod. údaje zobrazovaného diodami LED. Příklad takového programu je např.:

```
// PORTA tlacitka PORTB LEDky
#include <90s8515.h>
#include <delay.h>

// Deklarace globalnich promennych
unsigned char c;

void main(void)
{
// inicializace Port A - vstup
PORTA=0x00;
DDRA=0x00;

// inicializace Port B - vystup
PORTB=0x00;
DDRB=0xFF;

c = 0;
while (1)
    {
    if (PINA.0 == 0) c = c + 1;  //pricteni 1
    if (PINA.1 == 0) c = c - 1;  //odecteni 1
    if (PINA.2 == 0) c = c<<1 ;  //bitovy posun o jeden bit
                                 //doleva
    if (PINA.3 == 0) c = c>>1 ;  //bitovy posun o 1 bit
                                 //doprava
    if (PINA.4 == 0) c = ~c ;    //bitovy doplnek,jednickovy
                                 //doplnek
    if (PINA.5 == 0) c = 255 ;   //nastaveni samych jednicek
    if (PINA.6 == 0) c = 0 ;     //nastaveni samych nul
    PORTB = ~c;  //inver. 0...LED sviti, 1...LED nesviti
    delay_ms(100);
    };
}
```

Činnost tohoto programu je velice primitivní. Nejprve se provede inicializace PORTA jako vstupního a PORTB jako výstupního a nastaví se hodnota pomocné proměnné c na nulu. Potom se v nekonečné smyčce, začínající příkazem `while (1)`, pomocí příkazu `PORTB = ~c;` provádí zobrazování obsahu této pomocné

proměnné. Chci, aby se jedničky v jednotlivých bitech proměnné c projevily svitem LED, nuly naopak zhasnutím LED. Proto, s ohledem na naše zapojení, musíme na výstup PORTB poslat nikoli obsah proměnné c, ale hodnotu vzniklou z obsahu proměnné c negací jednotlivých bitů. Proto jsme použili operátor ~.

Před zobrazením hodnoty proměnné c ale ještě může dojít ke změně této hodnoty, pokud bude stisknuto některé tlačítko. K tomu jsme použili příkazy jako např. `if (PINA.3 == 0) c = c>>1` ; kde PINA.3 snímá logickou úroveň na pinu 3 PORTA a interpretuje ji jako nulu či jedničku. Protože stisk tlačítka se projeví nulou, je prováděn test na stisknutí porovnáním sejmuté hodnoty s nulou. Stisknutí tlačítka se projeví provedením posunutím obsahu proměnné c o jedno místo doprava. Pokud nebude tlačítko stisknuto, neprovede se tento příkaz. Takto se provádí postupně test na stisknutí tlačítek 0, 1, ... 6. Nakonec jsme ještě zařadili časovou prodlevu 100 ms jako jednoduchý způsob ošetření zákmitů tlačítek.

Poznámka: ***Zkušený programátor v jazyce C by v předchozím programu místo příkazu c = c + 1; použil c += 1; místo c = c – 1; použil c –= 1; atd.*** *Předpokládám, že mikrontroléry AVR se bude snažit naučit programovat v jazyce C i začátečník, či programátor běžně používající jiný jazyk a v C programující jen občas. Proto se v této knížce snažím spíše o srozumitelnost kódu, než zhuštěný kód profesionálů v céčku.* ***Viz též Závěrečná poznámka k této knize.***

7.5 Program 5 – maticová klávesnice

V některých aplikacích potřebujeme ke komunikaci s běžící aplikací více než osm tlačítek použitých v předchozím příkladě. Přidání dalších tlačítek by sice bylo možné vyřešit jejich připojením k dalšímu portu MCU a každým z tlačítek tak ovládat úroveň signálu pro jeden bit portu, ale zbytečně se tím ochuzujeme o porty umožňující komunikaci MCU s okolím. Proto při použití více tlačítek je obvyklé jejich uspořádání do matice a připojení této tlačítkové matice k jedinému portu. Programové obsloužení takové klávesnice je však složitější, než obsluha samotných tlačítek. Častým případem je použití 16 tlačítek zapojených do matice 4 × 4. Zdrojový kód obsluhy takové klávesnice najdeme po instalaci *CodeVisionAVR C* v podadresáři **examples** jako příklad **keypad**. Zdrojový kód tohoto programu je příkladem zhuštěného úsporného stylu programování v jazyce C, pro začátečníky velmi obtížně čitelného, až nesrozumitelného. Proto si nejprve ukážeme jednodušší a čitelnější variantu kódu obsluhy klávesnice 4 × 4 (uvedeného jako Program5A na CD), který budeme postupně upravovat a přibližovat se tak stylu céčka, a těmito úpravami nakonec dostaneme kód uvedený v příkladech instalace CodeVisionAVR C (Program5D).

Začneme tedy zdrojovým kódem (Program5A):

```
/*

4x4 Klávesnice
Chip: AT90S8515
Klávesnice 16 tlačítek je spojena do matice takto:

[STK500 PORTD HEADER]   [KEYS]  R1
 1 PD0 -----0----1----2----3-----------o+5V
            |    |    |    |      R2   |
 2 PD1 -----4----5----6----7---------
            |    |    |    |      R3   |
 3 PD2 -----8----9----10---11--------
            |    |    |    |      R4   |
 4 PD3 -----12---13---14---15--------
         D1  |    |    |    |
 5 PD4 -|<|-      |    |    |
         D2       |    |    |
 6 PD5 -|<|------      |    |
         D3            |    |
 7 PD6 -|<|-----------      |   R1..R4=10k..47k
         D4                 |
 8 PD7 -|<|----------------     D1..D4=1N4148

použit alfanumerický displej LCD 2x16
připojený na PORTC takto:

[LCD]
1 GND- 9 GND
2 +5V- 10 VCC
3 VLC- LCD contrast control voltage 0...1V
4 RS - 1 PC0
5 RD - 2 PC1
6 EN - 3 PC2
11 D4 - 5 PC4
12 D5 - 6 PC5
13 D6 - 7 PC6
14 D7 - 8 PC7
*/

//LCD na PORTC
#asm
.equ __lcd_port=0x15
#endasm
```

```
#include <lcd.h>
#include <stdio.h>
#include <delay.h>
#include <90s8515.h>

// kmitočet krystalu 8 MHz
#define F_XTAL 8000000L

// PIND0..3 budou vstupy řádků
#define KEYIN PIND

// PORTD4..7 budou výstupy sloupců
#define KEYOUT PORTD

// konstanty použité při inicializaci čítače
#define INIT_TIMER0 TCNT0=0x100L-F_XTAL/64L/500L

typedef unsigned char byte;
byte column;

//kód klávesy
unsigned keys;

// buffer displeje LCD
char buf[33];

// TIMER 0 interrupt každé 2 ms
interrupt [TIM0_OVF] void timer0_int(void)
{
keys=0;

// reinitialize TIMER0
INIT_TIMER0;

KEYOUT=0b10111111;
column = KEYIN|0xF0;
if (column!=0xFF) keys = ~column|0x10;

KEYOUT=0b11011111;
column = KEYIN|0xF0;
if (column!=0xFF) keys = ~column|0x20;

KEYOUT=0b11101111;
column = KEYIN|0xF0;
if (column!=0xFF) keys = ~column|0x40;

KEYOUT=0b01111111;
column = KEYIN|0xF0;
if (column!=0xFF) keys = ~column|0x80;

}

// test stisku tlačítka, klávesy
unsigned inkey(void)
{
unsigned k;
if (k=keys) keys=0;
return k;
}

void init_keypad(void)
{
DDRD=0xf0;
INIT_TIMER0;
TCCR0=3;
TIMSK=2;
#asm("sei")
}

//hlavní program
main() {
unsigned k;
init_keypad();
lcd_init(16);
lcd_putsf("Test klavesnice");
// snímá stisk kláves a zobrazuje jejich kód
while (1)
      {
      lcd_gotoxy(0,1);
      if (k=inkey())
         {
         sprintf(buf,"Kod tlac.=%Xh        ",k);
         lcd_puts(buf);
         }
      else lcd_putsf("nestisknuto tl.        ");
      delay_ms(250);
      }
}
```

Zdrojový kód začíná poznámkou obsahující popis použitého hardware. Za ní jsou uvedeny hlavičkové soubory knihoven a definice konstant. Vlastní, výkonná část programu `main()` začíná inicializací – volání funkce `init_keypad` provádějící inicializaci **čítače/časovače0** a povolení přerušení příkazem `#asm("sei")`, a inicializace displeje LED `lcd_init(16)`. Poté již program běží v nekonečné smyčce, kde při každém průběhu touto smyčkou zavolá funkci `inkey()`, která vrací hodnotu 0, není-li stisknuta žádná klávesa, jinak vrací kód stisknuté klávesy. V závislosti na vrácené hodnotě funkce `inkey()` se na displeji LCD zobrazuje buď informace, že není stisknuta žádná klávesa, nebo zobrazuje kód stisknuté klávesy v hexadecimálním formátu. Funkce `inkey()` pracuje tak, že po zavolání této funkce vrací hodnotu globální proměnné `unsigned keys`, v případě nenulovosti této hodnoty provádí ještě její vynulování (protože předtím nenulovou hodnotu proměnné `keys` kopíruje do lokální proměnné k a potom vrací hodnotu k, vrací funkce tuto nenulovou hodnotu i po vynulování globální proměnné `keys`). Globální proměnná keys obsahuje kód stisknuté klávesy. Hodnotu této proměnné nastavuje každých 2 ms obslužná funkce přerušení vyvolaného čítačem/časovačem0.

Obslužná funkce přerušení `timer0_int(void)` začíná nastavením globální proměnné `keys` na nulu a reinicializací čítače/časovače0, čímž se zajistí další přerušení po 2 ms. Poté se provede vyslání hodnoty `0b10111111` na PORTD. Tato hodnota obsahuje sedm bitů nastavených na jedničku a jeden bit nastavený na nulu. Následuje sejmutí dat z PORTD. Nebude-li stisknutá žádná klávesa, bude mít sejmutá hodnota KEYIN jedničky ve čtyřech nejnižších bitech 0, 1, 2 a 3. Logickým součtem po bitech s hodnotou 0xF0 se zajistí, že budou jedničky i v horních čtyřech bitech proměnné

```
column = KEYIN|0xF0
```

která proto bude obsahovat hodnotu 0xFF. Proto podmínka `(column!=0xFF)` nebude splněna a proto se neprovede žádný z příkazů (za) měnících hodnotu proměnné `keys` a proto bude hodnota této proměnné nulová. Bude-li naopak stisknuto alespoň jedno z tlačítek umístěných ve sloupci připojeným k pinu PD6, tj. pinu, na němž je po vyslání `KEYOUT=0b10111111` nula, bude hodnota proměnné `column = KEYIN|0xF0` různá od 0xFF a proto po provedení příslušného příkazu **`if`** bude globální proměnná `keys` obsahovat nenulovou hodnotu – kód stisknuté klávesy. Pokud bychom hodnotu této proměnné naplnili přímo hodnotou proměnné `column` dostali bychom potom nenulovou hodnotu proměnné `keys` a tato hodnota by mohla být již kódem stisknuté klávesy. Pokud by **současně** byly stisknuty ještě další klávesy v témže sloupci, dostali bychom jiný kód, z něhož by bylo možné poznat, které klávesy byly stisknuté. Problém by však nastal v případě stisknutí všech čtyř kláves v tomto sloupci – hodnota proměnné `keys` by pak byla nulová, stejně jako v případě, kdy není stisknuta žádná klávesa. Abychom zabránili této chybě, můžeme do proměnné `keys` místo hodnoty proměnné `column` posílat hodnotu `~column` (negace v jednotlivých bitech), takže v případě nestisknutých kláves bude vracet nulu. V případě stisknutých kláves v třetím sloupci budou v proměnné `keys` jedničky na odpovídajících bitech. V případě stisknutí jediné klávesy v sloupci budou tyto kódy 1, 2, 4 a 8. Tím jsme zjistili případné stisknutí kláves ve třetím sloupci a jejich kód.

Obdobným způsobem zjistíme případné stisknutí kláves v jiných sloupcích – postupně budeme na PORTD vysílat hodnoty 0b11011111, 0b11101111 a 0b01111111 a snímat hodnoty na PORTD a sejmuté hodnoty zpracovávat jako v případě třetího sloupce. Protože bychom opět dostali kódy stisknutých kláves 1, 2, 4 a 8, nemohli bychom rozlišit, která z kláves v jednom řádku byla stisknuta. Rozlišení řádku lze provést např. přičtením 0x10, 0x20, 0x40 či 0x80 v závislosti na ošetřovaném sloupci kláves. Kód jednotlivých kláves pak bude viz *tab. 7.1*.

Tab. 7.1 Kódy jednotlivých kláves

81h	41h	21h	11h
82h	42h	22h	12h
84h	44h	24h	14h
88h	48h	28h	18h

V případě, že nám bude stačit ošetření stisku jediné klávesy, bude takové řešení vyhovující. Někdy se ale může hodit mít ošetřené stisknutí několika kláves současně. Jako vrácený kód se pak uvažuje součet kódů jednotlivých kláves. V tom případě by již zatím zavedené kódování nevyhovovalo. Docházelo by k mnoha nejednoznačnostem. Např. stisknutí kláves s kódy 21 a 12 by vracelo hodnotu 33, stejně jako stisknutí kláves s kódy 11 a 22. Proto je výhodnější zvolit např. kódování, při němž v 16bitové proměnné `keys` bude každá z 16 kláves svým stisknutím nastavovat jeden bit této proměnné, tj. každé tlačítko bude mít „svůj" bit v této proměnné. Jednička v tomto bitu bude odpovídat stisknutému tlačítku, nula nestisknutému. Tabulka kódů kláves pak bude *tab. 7.2*.

Tab. 7.2 Tabulka kódů kláves

1h	10h	100h	1000h
2h	20h	200h	2000h
4h	40h	400h	4000h
8h	80h	800h	8000h

Odpovídající část kódu v obslužné funkci přerušení se změní na

```
KEYOUT=0b10111111;
column = KEYIN|0xF0;
if (column!=0xFF) keys = ~column;

KEYOUT=0b11011111;
column = KEYIN|0xF0;
if (column!=0xFF) keys = (unsigned)(~column)<<4;
```

```
KEYOUT=0b11101111;
column = KEYIN|0xF0;
if (column!=0xFF) keys = (unsigned)(~column)<<8;

KEYOUT=0b01111111;
column = KEYIN|0xF0;
if (column!=0xFF) keys = (unsigned)(~column)<<12;
```

Využívá se zde toho, že po provedení příkazu `keys = ~column;` jsou v dolních čtyřech bitech této proměnné jedničky v místech odpovídajících stisknutým klávesám, tedy pro jeden sloupec máme již to, co bychom potřebovali. Abychom totéž dostali i pro zbývající sloupce, ovšem s tím rozdílem, že výsledek budeme umísťovat do jiné čtveřice bitů v proměnné `keys`, provedeme ještě posun o 4, 8 či 12 bitů doleva, takže každý sloupec bude mít svou vlastní čtveřici bitů v proměnné `keys`. Protože tato proměnná je 16bitová, zatímco proměnná `column` jen 8bitová, musíme ještě provést přetypování `(unsigned)`. Výsledný kód pak tvoří Program5B na doprovodném CD.

Stále má však náš kód ještě alespoň dva nedostatky. Jeden je ten, že máme za sebou čtyři téměř stejné úseky kódu, každý z těchto úseků obsluhuje jeden sloupec kláves. Opakování téměř stejných úseků kódu je dobré se raději vyhnout, např. přepsáním těchto úseků na volání funkce. Výhodou je, že případné změny kódu provádíme na jednom místě, program se tak lépe udržuje.

Dalším nedostatkem našeho kódu je, že v případě stisknutí kláves v několika sloupcích, vrací kód jen tlačítka v posledním ze sloupců obsahujících stisknutá tlačítka. Proto provedeme další úpravy našeho kódu. Výsledkem bude další Program5C. Na jeho začátku přidáme definici dalších dvou konstant

```
#define FIRST_COLUMN 0x80
#define LAST_COLUMN 0x10
```

a hlavně přepíšeme funkci obsluhy přerušení.

```
interrupt [TIM0_OVF] void timer0_int(void)
{
static byte column=FIRST_COLUMN;
static unsigned row_data;

// reinitialize TIMER0
INIT_TIMER0;

row_data<<=4; //totéž jako row_data = row_data<<4
              //neboli posun obsahu row_data o 4 místa doleva

// získání skupiny 4 tlačítek v row_data
row_data|=~KEYIN&0xf;

column>>=1; // totéž jako column = column>>1;
            //neboli obsah column se posune o jedno místo doprava

if (column==(LAST_COLUMN>>1))
   {
   column=FIRST_COLUMN;
   keys=row_data;
   row_data=0;
   };

// vybere se další sloupec, protože vstupy jsou v klidovém
// stavu na úrovni 1, provede se bitová inverze proměnné column
KEYOUT=~column;
}
```

Proti předchozím dvěma programům Program5A a Program5B je tady zásadní rozdíl v tom, že při jedné obsluze přerušení se provede zjištění stavu kláves v jediném sloupci, při další obsluze přerušení po 2 ms se zjistí stav kláves v dalším sloupci, po dalších 2 ms obdobně a po dalších 2 ms již v posledním sloupci a přitom se ještě do proměnné `keys` zapíše informace o stisknutých klávesách. Na zjištění stavu klávesnice je tedy potřeba čtyř po sobě následujících obsluh přerušení, tj. 4 × 2 ms. Zároveň je potřeba, aby i po skončení funkce obsluhující přerušení zůstala zachována informace o tom, který sloupec kláves se má ošetřovat při následující obsluze přerušení a rovněž se má zachovat informace o stavu kláves ve sloupcích, které již byly ošetřeny. Proto jsou proměnné `byte column` a `unsigned row_data` deklarovány jako **`static`**. Proměnná `column` po definici obsahuje hodnotu `FIRST_COLUMN` tj. 0b1000000. Příkaz `column>>=1;` provádí posun o jeden bit vpravo, takže při dalších voláních funkce obsluhy přerušení budou hodnoty proměnné `column` 0b01000000, 0b00100000 a 0b00010000 a proto na PORTD budou příkazem `KEYOUT=~column;` posílány hodnoty 0b01111111, 0b10111111, 0b11011111 a 0b11101111, tj. naprosto stejné hodnoty jako v předchozích dvou programech.

Příkaz `row_data<<=4;` tj. posuv obsahu této proměnné o čtyři bity doleva zabezpečí, že při každé ze čtyř po sobě jdoucích obsluh přerušení se informace o stisknutých klávesách v sloupci bude zapisovat do jiné čtveřice bitů v `row_data`. Tento zápis se provádí příkazem `row_data|=~KEYIN&0xf;`. Nahrazuje vlastně příkazy typu

```
column = KEYIN|0xF0;
if (column!=0xFF) keys = ~column;
```

z předchozích dvou programů. Rozdíl je především v tom, že informaci o stisknutých klávesách ukládá do pomocné proměnné `row_data`. Pro začátečníky ještě uvedeme, že příkaz `row_data|=~KEYIN&0xf;` je vlastně jiným zápisem pro

```
row_data=row_data|(~KEYIN&0xf);
```

tedy že si tato proměnná v sobě nese již informaci o stavu kláves v již sledovaných sloupcích klávesnice. Při každém čtvrtém běhu funkce obsluhy přerušení se provádí v důsledku splnění podmínky

```
if (column==(LAST_COLUMN>>1))
```

provádí kód

```
    {
    column=FIRST_COLUMN;
    keys=row_data;
    row_data=0;
    };
```

tj. nastaví se počáteční hodnoty pomocných proměnných `column` a `row_data` a do globální proměnné `keys` se uloží informace o všech stisknutých klávesách, zjištěná při posledních čtyřech prováděních funkce obsluhy přerušení.

Náš kód je teď již téměř stejný, jako kód příkladu z instalace CodeVisionAVR C. Liší se několika drobnostmi v těle funkce obsluhy přerušení

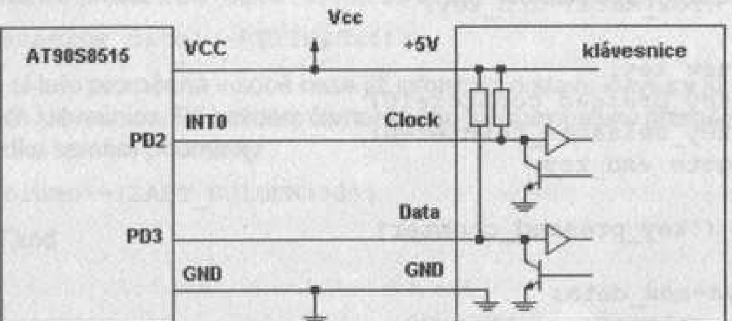
```
interrupt [TIM0_OVF] void timer0_int(void)
{
static byte key_pressed_counter=20;
static byte key_released_counter,column=FIRST_COLUMN;
static unsigned row_data,crt_key;

INIT_TIMER0;
row_data<<=4;
row_data|=~KEYIN&0xf;
column>>=1;

if(column==(LAST_COLUMN>>1))
  {
  column=FIRST_COLUMN;
  if (row_data==0) goto new_key;
  if (key_released_counter) --key_released_counter;
  else
    {
    if (--key_pressed_counter==9) crt_key=row_data;
    else
    {
    if (row_data!=crt_key)
      {
      new_key:
      key_pressed_counter=10;
      key_released_counter=0;
      goto end_key;
      };
    if (!key_pressed_counter)
    {
    keys=row_data;
    key_released_counter=20;
    };
    };
  };
 end_key:;
 row_data=0;
 };
KEYOUT=~column;
}
```

7.6 Program 6 – klávesnice PC

Na předchozím programu jsme si ukázali princip připojení a programového obsloužení maticové klávesnice. V případě, kdy potřebujeme klávesnici s větším počtem kláves může být výhodné použít běžnou klávesnici používanou u počítačů PC. Důsledkem jejího rozšíření je nízká cena a snadnost jejího připojení k mikropočítači či mikrokontroléru. PC AT klávesnice tvoří samostatný konstrukční celek, obsahující obyčejně 102 kláves a mikrokontrolér, který vykonává řadu funkcí. Tou hlavní je neustálé sledování stavu matice spínačů. V případě jakékoli změny zjistí mikrokontrolér souřadnice spínače, který ji způsobil, vybere v této situaci odpovídající SCAN kód, a ten po sériové lince DATA vyšle směrem k zařízení, s nímž klávesnice komunikuje. Je-li některá klávesa stisknuta déle než 0,5 s, vysílá její kód automaticky znovu. Kromě vlastních dat vysílá klávesnice při stisku klávesy i hodinový signál CLK o kmitočtu 10 až 20 kHz. Při sestupné hraně hodinového signálu CLK jsou vysílaná DATA platná. Toho můžeme využít při komunikaci klávesnice s nějakým systémem jako PC, mikrokontrolér AVR apod. Tuto komunikaci včetně obslužného programu napsaného v jazyce C si ukážeme na komunikaci PC klávesnice s mikrokontrolérem ATMEL AT90S8515, který s klávesnicí spojíme podle *obr. 7.22*.

Signál CLK z klávesnice přitom bude vyvolávat vnější přerušení mikrokontroléru AVR. Přerušení vyvolané sestupnou hranou hodinového signálu CLK bude obsluhováno funkcí, která bude snímat data vysílaná klávesnicí. Klávesnice bude

k mikrokontroléru připojena přes její konektor. Zapojení konektoru PC AT klávesnice popisuje *tab. 7.3*.

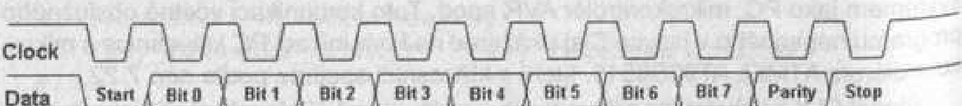


Obr. 7.22 *Spojení klávesnice s mikrokontrolérem*

Tab. 7.3 *Zapojení konektorů klávesnic*

klávesnice PC AT		
Signál	**DIN41524 ,zásuvka 5-pin**	**p-pin Mini DIN PS2**
Clock	1	5
Data	2	1
nezapojen	3	2 , 6
GND	4	3
+5V	5	4
stínění	stínění	stínění

Klávesnice vysílá data sériově vždy po 11 bitech. Nejprve je vysílán Start bit (logická nula), poté 8 datových bitů (první je LSB, poslední MSB), dále parita a nakonec Stop bit (logická jednička) *obr. 7.23*.



Obr. 7.23 *Sériové vysílání dat z klávesnice*

Při obsluze přerušení se při prvním přerušení vynuluje proměnná nesoucí informaci o počtu přijatých bitů, při každém dalším přerušení se tato proměnná inkrementuje. Obsah bitů 1 až 8 (bit 0 byl Start bit) datového vodiče se načítá do jednoho znaku, bit 9 a 10 se ignorují. Poté se provádí dekódování přijatého znaku. Toto řešení s použitím přerušení se dá použít pro jakékoli jednočipové procesory a jeho výhodou je minimální režie, kterou si obsluha klávesnice z celého systému zabere.

Minimální mezera mezi dvěma kódy vyslanými po sobě je 1,2 ms. Tato synchronní sériová komunikace je obousměrná, my však budeme klávesnici používat jen ve směru klávesnice – mikrokontrolér. Kromě kódů kláves vysílá klávesnice i řídicí signály:

FFh přetečení bufferu, klávesnice detekuje chybu,

FEh žádost o zaslání posledního znaku, špatně přijatý znak, parita,

FAh potvrzení – ACK,

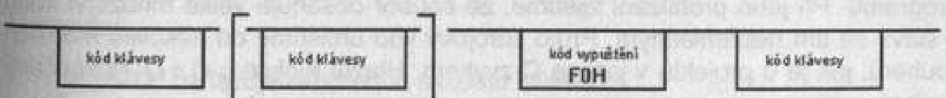
F0h kód uvolnění klávesy,

AAh úspěšný power-on test,

EEh echo – klávesnice odpoví zpět také EEh jako echo – pro test,

00h přetečení bufferu, klávesnice detekuje chybu.

Z hlediska kódování kláves klávesnicí je možné rozdělit klávesy do tří skupin. Jsou to skupina základní, skupina rozšířená a skupina speciální. Do *základní skupiny* patří 83 kláves. Po stisku některé z těchto kláves je vyslán kód této klávesy a po uvolnění této klávesy je vyslán kód uvolnění klávesy a znovu kód uvolněné klávesy. Při delším stisku je kód neustále vysílán až do uvolnění. Kód klávesy v hranatých závorkách (*obr. 7.24*) se vysílá dokud je klávesa stisknuta.



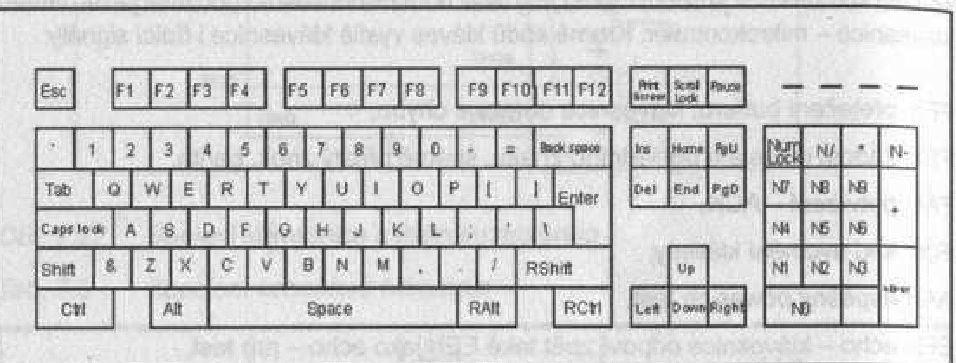
Obr. 7.24 *Posloupnost vysílaných znaků (83 kláves)*

Rozšířená skupina zahrnuje 14 kláves. Tyto klávesy mají kódy shodné s klávesami numerické klávesnice, patřících do základní skupiny, ale před tímto kódem je předřazen kód F8, který je od těchto kláves odlišuje. Dva kódy v hranatých závorkách (*obr. 7.25*) se vysílají dokud je klávesa stisknuta.

Obr. 7.25 *Posloupnost vysílaných znaků pro rozšířenou klávesnici*

Speciální skupina zahrnuje všechny ostatní klávesy. Jedná se o klávesy Pause a Print Screen.

Při současném stisku dvou a více ostatních kláves (základní a rozšířená skupina) jsou kódy vysílány v odpovídajícím pořadí v jakém se jednotlivé akce stisku a uvolnění stanou a tyto kódy se vzájemně neovlivňují, kromě případu, kdy je přerušeno cyklické vysílání kódu klávesy – autorepeat. Rozmístění všech kláves a názvy jejich kódů jsou zřejmé z *obr. 7.26*.



Obr. 7.26 Rozmístění a označení kláves

Kódy kláves jsou uvedeny v *tab. 7.4* (tabulka je použitá z [22]).

Při psaní obslužného programu pro mikrokontrolér AVR AT90S8515 jsme se inspirovali aplikační poznámkou **ATMEL AVR313: Interfacing the PC AT Keyboard**, publikující zdrojový kód interface mezi PC klávesnicí PC a RS-232 napsaný v IAR C. Z tohoto jsme použili jen některé funkce, popř. je upravili pro CodeVisionAVR C. Výsledný zdrojový kód najdeme na doprovodném CD jako zdrojový kód příkladu Program6. Při jeho prohlížení zjistíme, že soubor obsahuje velké množství kódu a stává se tím nepřehledným. Proto zdrojový kód umístíme do několika menších souborů, jak je u projektů v jazyce C zvykem. Hlavní funkce `main()` je součástí souboru `pckey.c`

```
/*********************************************
  vystup znaku z klavesnice na LCD
CodeWizardAVR V1.23.5 Evaluation

Date      : 22.11.2002
Autor     : VLADIMIR VANA
Comments  :
```


Tab. 7.4 SCAN kódy kláves rozšířené skupiny [22]

Tabulka kódů a jmen kódů klávesnice počítače standardu IBM PC											
pořadí	kód	jméno	pořadí	kód	jméno	pořadí	kód	jméno	pořadí	kód	jméno
0	$7E0	uvol.	30	$6FA	N9	60	$000		90	$434	Z
1	$60A	F1	31	$6E2	N•	61	$6AA	=	91	$48A	[
2	$60C	F2	32	$452	Space	62	$000		92	$4BA	\
3	$408	F3	33	$6F6	N-	63	$000		93	$4B6	]
4	$618	F4	34	$708	Sysrq	64	$000		94	$000	
5	$606	F5	35	$6EE	n. lock	65	$438	A	95	$000	
6	$416	F6	36	$6FC	s. lock	66	$464	B	96	$41C	
7	$506	F7	37	$4B0	c. lock	67	$642	C	97	$7C2	Pause
8	$614	F8	38	$4C2	&	68	$446	D	98	$5C0	GR2
9	$402	F9	39	$4A4	'	69	$648	E	99	$000	
10	$612	F10	40	$3ED	PCled	70	$656	F	100	$622	Ralt
11	$6F0	F11	41	$7F4	PCled	71	$468	G	101	$628	Rctrl
12	$40E	F12	42	$4F8	*	72	$666	H	102	$6E2	Del
13	$6B4	↵	43	$4F2	+	73	$486	I	103	$6D2	End
14	$4EC	Esc	44	$682	,	74	$476	J	104	$4F4	PgDn
15	$628	Ctrl	45	$69C	-	75	$684	K	105	$6FA	PgUp
16	$622	Alt	46	$492	•	76	$696	L	106	$6D8	home
17	$624	Shift	47	$494	/	77	$674	M	107	$4E0	Ins
17	$6B2	Rshift	48	$48A	0	78	$462	N	108	$4D6	←
19	$41A	Tab	49	$42C	1	79	$688	O	109	$6E8	→
20	$6CC	Bcksp	50	$63C	2	80	$69A	P	110	$4EA	↑
21	$4E0	N0	51	$44C	3	81	$42A	Q	111	$6E4	↓
22	$6D2	N1	52	$44A	4	82	$65A	R	112	$6B4	N↵
23	$6E4	N2	53	$65C	5	83	$636	S	113	$494	N/
24	$4F4	N3	54	$66C	6	84	$458	T	114	$624	ESH
25	$4D6	N4	55	$47A	7	85	$678	U	115	$4F8	Print
26	$4E6	N5	56	$47C	8	86	$454	V	116	$6FC	Break
27	$6E8	N6	57	$48C	9	87	$63A	W	117	$000	
28	$6D8	N7	58	$000		88	$644	X	118	$000	
29	$4EA	N8	59	$498	;	89	$66A	Y	119	$000	

Legenda:

- kód „GR2" označuje rozšířenou skupinu kláves
- velké písmeno „N" před znakem označuje numerickou část klávesnice
- „pořadí" odpovídá umístění v tabulce ASCII znaků
- kód „$000" označuje, že pro kód ASCII tabulky neexistuje kód klávesy.

```
Chip type           : AT90S8515
Clock frequency     : 8,000000 MHz
Memory model        : Small
Internal SRAM size  : 512
External SRAM size  : 0
Data Stack size     : 128
*********************************************/

#include <90s8515.h>
#include <stdlib.h>
#include "Kb.c"
#include "Serial.c"
#include <ctype.h>
// displej LCD bude připojen k PORTA
#asm
.equ __lcd_port=0x1B
#endasm
#include <lcd.h>
#include <delay.h>

unsigned char radka [16];
unsigned int i = 0;

//---------------------------------------------------------
void main(void)
{

// Port A initialization
PORTA=0x00;
DDRA=0x00;

// Port D initialization
PORTD=0x00;
DDRD=0x00;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: Timer 0 Stopped
TCCR0=0x00;
TCNT0=0x00;

// Timer/Counter 1 initialization
// Clock source: System Clock
```



```
// Clock value: Timer 1 Stopped
// Mode: Normal top=FFFFh
// OC1A output: Discon.
// OC1B output: Discon.
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// External Interrupt(s) initialization
// INT0: On
// INT0 Mode: Rising Edge
// INT1: Off
GIMSK=0x40;
MCUCR=0x03;
GIFR=0x40;

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x00;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
// Analog Comparator Output: Off
ACSR=0x80;

// LCD module initialization
lcd_init(16);
lcd_gotoxy(0,0);
lcd_putsf("test PC key ");
// povolení přerušení
#asm("sei")
init_kb();
lcd_gotoxy(0,1);
lcd_putsf(" ");
lcd_gotoxy(0,1);
while   (1)
      {
```

```
      key = getchar();
      lcd_putchar(key);
      radka[i] = key;
      i = i+1;
      if (i==16)
      {
         lcd_gotoxy(0,0);
         lcd_puts(radka);
         lcd_gotoxy(0,1);
         lcd_putsf(" " );
         lcd_gotoxy(0,1);
         i = 0;
      }
      delay_ms(200);
   };
}
```

Další kód obsahující funkci obsluhy přerušení je součástí souboru `Serial.c`

```
//soubor Serial.c
#include <90s8515.h>

static unsigned char data;

// obsluha vnějšího přerušení Interrupt 0
interrupt [EXT_INT0] void ext_int0_isr(void)
{

  if (!edge) // podprogram spouštěný sestupnou hranou
    {
      if(bitcount < 11 && bitcount > 2) /* Bit 3 až 10 jsou
data. Parity bit, start a stop bits se ignorují.*/
      {
        data = (data >> 1);
        if(PIND & 8)
          data = data | 0x80; // ukládá se '1'
      }

      MCUCR = 3; // nastavení přerušení na vzestupnou hranu
      edge = 1;

   } else { // podprogram spouštěný vzestupnou hranou

      MCUCR = 2; // nastavení přerušení od sestupné hrany
      edge = 0;

      if(--bitcount == 0) // všechny bity přijaty
      {
        decode(data);

        bitcount = 11;
      }
   }

}
```

Tato funkce je stěžejní funkcí celého programu, zabezpečuje příjem sériových dat z klávesnice. Algoritmus její činnosti využívá vzestupnou i sestupnou hranu hodinového signálu CLK vyvolávající vnější přerušení. Ke své činnosti využívá tři proměnné `edge, bitcount` a `data`. Do proměnné `edge` ukládáme 0 jako informaci o tom, že jsme (pomocí `MCUCR = 2;`) nastavili spouštění přerušení INT0 sestupnou hranou signálu na PD2 a do `edge` ukládáme 1, když pomocí `MCUCR = 3;` nastavujeme spouštění přerušení INT0 vzestupnou hranou signálu na PD2. Proto také jsou v našem kódu vždy za sebou příkazy

```
MCUCR = 2;
edge = 0;
```

či

```
MCUCR = 3;
edge = 1;
```

Proměnná `bitcount` slouží jako čítač počítající vzestupné hrany signály CLK. Před začátkem příjmu znaku je vždy nastaven na 11 a při každé vzestupné hraně signálu je dekrementován. Proměnná `data` slouží k ukládání přijímaných dat. Tato proměnná před příjmem znaku obsahuje samé nuly. Pokud dojde k přerušení od vzestupné hrany a přitom je splněna podmínka `if(bitcount < 11 && bitcount > 2)`, tj. obsah čítače hran signálu CLK nás informuje, že na PD3 jsou platná data z klávesnice, zapíše se hodnota z PD3 do nejvyššího bitu proměnné data. Přesněji řečeno, bude se pomocí

```
if(PIND & 8)
data = data | 0x80;
```

zapisovat jen jednička. Pokud bude na PD3 nula, nebude splněna podmínka `PIND & 8` a proto se neprovede příkaz `data = data | 0x80;` zapisující do nejvyššího bitu jedničku. Nebo-li v nejvyšším bitu zůstane 0. Příkaz `data = (data >> 1);` provádí posun obsahu proměnné `data` o jedno místo doprava,

takže po provedení tohoto příkazu je možné do nejvyššího bitu proměnné `data` zapisovat přijatý bit dat z PD3. Po skončení přenosu 11 bitu softwarový čítač `bitcount` dosáhne nuly, takže je splněna podmínka v

```
if(--bitcount == 0)
```

a proměnná `data` obsahuje SCAN kód přijatého znaku. Zavoláním funkce `decode(data);` pak dostaneme ASCII kód přijatého znaku. Příkazem `bitcount = 11;` připravíme softwarový čítač na počítaní přijatých bitů při příjmu dalšího znaku z klávesnice.

Další funkce zabezpečující komunikaci s klávesnicí jsou součástí souboru `Kb.c`

```
//soubor Kb.c

#include <90s8515.h>
#include <stdlib.h>
#include "scancodes.h"
#include <ctype.h>

#define BUFF_SIZE 64
unsigned char edge, bitcount;
unsigned char kb_buffer[BUFF_SIZE];
unsigned char *inpt, *outpt;
unsigned char buffcnt;
unsigned char key;

//-----------------------------------------------------------
void put_kbbuff(unsigned char c)
{
  if (buffcnt<BUFF_SIZE)     // pokud buffer není plný
  {
    *inpt = c;              // vloží znak do bufferu
    inpt++;                 // inkrementuje pointer

    buffcnt++;

    if (inpt >= kb_buffer + BUFF_SIZE) // Pointer wrapping
      inpt = kb_buffer;
  }
}

//-----------------------------------------------------------
void decode(unsigned char sc)
{
```



```
  static unsigned char is_up=0, shift = 0, mode = 0;
  unsigned char i;

  if (!is_up)    // poslední přijatá data byla
                 // identifikátorup-key
  {
    switch (sc)
    {
      case 0xF0 : // identifikátor up-key
        is_up = 1;
        break;

      case 0x12 : // Left SHIFT
        shift = 1;
        break;

      case 0x59 : // Right SHIFT
        shift = 1;
        break;

      case 0x05 : // F1
        if(mode == 0)
          mode = 1; // Enter scan code mode
        if(mode == 2)
          mode = 3; // Leave scan code mode
        break;

      default:
        if(mode == 0 || mode == 3) // při ASCII modu
        {
          if(!shift)   // pokud není shift,
          {  // použije tabulku look-up
            for(i = 0;
            ;unshifted[i][0]!=sc && unshifted[i][0]; i++);
              if (unshifted[i][0] == sc) {
                put_kbbuff(unshifted[i][1]);
            }
          } else {  // při shift
            for(i = 0; shifted[i][0]!=sc && shifted[i][0]; i++);
            if (shifted[i][0] == sc) {
              put_kbbuff(shifted[i][1]);
            }
        }
```

```
    } else{  // Scan code mode
        put_kbbuff(' ');
        put_kbbuff(' ');
      }
    break;
  }
} else {
  is_up = 0; // dvě 0xF0 v řádce nejsou povoleny
  switch (sc)
  {
    case 0x12 :  // Left SHIFT
      shift = 0;
      break;

    case 0x59 :  // Right SHIFT
        shift = 0;
        break;

    case 0x05 :  // F1
        if(mode == 1)
          mode = 2;
        if(mode == 3)
          mode = 0;
        break;
    case 0x06 :  // F2
        //clr();
    break;
  }
 }
}

//-----------------------------------------------------------------
void init_kb(void)
{
  inpt = kb_buffer;
  outpt = kb_buffer;
  buffcnt = 0;

  MCUCR = 2;
  edge = 0;
  bitcount = 11;
}
```



```
//-----------------------------------------------------------------
int getchar(void)
{
   int byte;
   while(buffcnt == 0);          // čeká na data

   byte = *outpt;                // snímá byte
   outpt++;                      // Inkrementuje pointer

   if (outpt >= kb_buffer + BUFF_SIZE) // Pointer wrapping
       outpt = kb_buffer;

   buffcnt--; // Dekrementuje buffer count

   return byte;
}
```

Scankódy jsou součástí hlavičkového souboru scancodes.h:

```
// Unshifted characters
unsigned char unshifted[68][2] = {
0x0d,9,
0x0e,'|',
0x15,'q',
0x16,'1',
0x1a,'z',
0x1b,'s',
0x1c,'a',
0x1d,'w',
0x1e,'2',
0x21,'c',
0x22,'x',
0x23,'d',
0x24,'e',
0x25,'4',
0x26,'3',
0x29,' ',
0x2a,'v',
0x2b,'f',
0x2c,'t',
0x2d,'r',
0x2e,'5',
0x31,'n',
```

```
0x32,'b',
0x33,'h',
0x34,'g',
0x35,'y',
0x36,'6',
0x39,',',
0x3a,'m',
0x3b,'j',
0x3c,'u',
0x3d,'7',
0x3e,'8',
0x41,',',
0x42,'k',
0x43,'i',
0x44,'o',
0x45,'0',
0x46,'9',
0x49,'.',
0x4a,'-',
0x4b,'l',
0x4c,'ř',
0x4d,'p',
0x4e,'+',
0x52,'ć',
0x54,'í',
0x55,' ', //\\
0x5a,13,
0x5b,'¨',
0x5d,' ', // \\
0x61,'<',
0x66,8,
0x69,'1',
0x6b,'4',
0x6c,'7',
0x70,'0',
0x71,',',
0x72,'2',
0x73,'5',
0x74,'6',
0x75,'8',
0x79,'+',
0x7a,'3',
0x7b,'-',
0x7c,'*',
```



```
0x7d,'9',
0,0
};

// Shifted characters
  unsigned char shifted[68][2] = {
0x0d,9,
0x0e,'Š',
0x15,'Q',
0x16,'!',
0x1a,'Z',
0x1b,'S',
0x1c,'A',
0x1d,'W',
0x1e,'"',
0x21,'C',
0x22,'X',
0x23,'D',
0x24,'E',
0x25,'¤',
0x26,'#',
0x29,' ',
0x2a,'V',
0x2b,'F',
0x2c,'T',
0x2d,'R',
0x2e,'%',
0x31,'N',
0x32,'B',
0x33,'H',
0x34,'G',
0x35,'Y',
0x36,'&',
0x39,'L',
0x3a,'M',
0x3b,'J',
0x3c,'U',
0x3d,'/',
0x3e,'(',
0x41,';',
0x42,'K',
0x43,'I',
0x44,'O',
0x45,'=',
```

```
0x46,')',
0x49,':',
0x4a,'_',
0x4b,'L',
0x4c,'Ř',
0x4d,'P',
0x4e,'?',
0x52,'Č',
0x54,'Í',
0x55,'´',
0x5a,13,
0x5b,'^',
0x5d,'*',
0x61,'>',
0x66,8,
0x69,'1',
0x6b,'4',
0x6c,'7',
0x70,'0',
0x71,',',
0x72,'2',
0x73,'5',
0x74,'6',
0x75,'8',
0x79,'+',
0x7a,'3',
0x7b,'-',
0x7c,'*',
0x7d,'9',
0,0
);
```

V tomto programu máme tedy zdrojový kód umístěný v souborech `pckey.c`, `Serial.c`, `Kb.c` a `Scancodes.h`. Náš program je jenom školní, ukázkový program. Vstupní bod tohoto programu, funkce main, je jedinou funkcí souboru `pckey.c`. Program běží v nekonečné smyčce, kde pomocí volání funkce `getchar()` přijímá znaky vysílané klávesnicí a zobrazuje je na znakovém LCD displeji 16 × 2. Při zápisu znaku na 16 pozici v druhé řádce se provede scrolování displeje a nastavení kurzoru na začátek druhé, vymazané řádky. Pokud jde o PC klávesnici, v souboru `pckey.c` se provádí jenom volání funkce `init_kb()` a `getchar()`. Ve zbývajících třech souborech je zase naopak jen kód týkající se klávesnice. To umožní využívat tyto tři soubory i jako součást jiných projektů, ve kterých používáme klávesnici PC jako periferii AVR mikrokontroléru. To je, kromě již zmiňovaného zvýšení přehlednosti a udržovatelnosti, další výhodou rozdělení kódu do více souborů.

7.7 Program 7 – voltmetr

V předchozích programech jsme programovali obsluhu externích periferií jako jsou tlačítka, klávesnice či LCD displej nebo diody LED. Jedinou vnitřní periferií, kterou jsme zatím programově obsluhovali, byl UART v *programu3*. Nyní si ukážeme jak programově obsloužit A/D převodník, který je vnitřní periferií mikrokontroléru AT90S8535. Jeho činnost je ovládána zápisem do jeho registru **ADCSR**. Činnost tohoto převodníku je povolena nastavením nejvýznamnějšího bitu (bit 7) tohoto registru na 1. Pro tento, sedmý bit je používáno označení **ADEN**. Další, nižší bit (bit 6) je označován **ADSC**, jeho nastavením na 1 je nastartován převod A/D. Po dokončení tohoto převodu bude hodnota tohoto bitu rovna 0. Pokud probíhá převod A/D, nemá případné zapsání nuly to tohoto bitu nějaký vliv na převod. Nastavení bitu 3, nazývaného **ADIE**, na jedničku znamená povolení (aktivaci) přerušení **ADC Conversion Complete**. K tomuto přerušení dojde po dokončení A/D převodu. Nejnižší tři bity registru ADCSR určují dělicí poměr předděličky, kterým je dělen kmitočet krystalu mikrokontroléru. Výstupní signál z předděličky je přiveden na hodinový vstup A/D převodníku. Po dokončení A/D převodu je získaná naměřená digitální hodnota uložena v registrech ADCL a ADCH převodníku A/D. Obsah obou registrů vytváří obsah ADCW. Ten je již přímo úměrný měřenému napětí. V našem případě měříme napětí v rozsahu 0 až 5 V. Protože A/D převodník v AT90S8535 je 10bitový, a $2^{10} = 1024$, odpovídá toto číslo 5 V, tj. 5000 mV. Proto musíme naměřený údaj v ADCW vynásobit konstantou 5000/1024 = 4,88. Tomu odpovídá příkaz

```
napeti = (int)(ADCW*4.888);
```

příkazy

```
if (napeti <1000) lcd_gotoxy(1,1);
if (napeti <100) lcd_gotoxy(2,1);
if (napeti <10) lcd_gotoxy(3,1);
```

pak zajišťují umístění naměřené hodnoty na pozici LCD nezávisle na počtu cifer naměřeného napětí – zarovnání doprava. Zdrojový kód voltmetru je velice jednoduchý:

```
/*
na PORTC je připojen LCD displej
nastavení :
- VTarget=5.0V
- ARef=5.0V
- Oscillator=8 MHz

SS měřené napětí 0 až 5V se přes odpor cca 1k připojí
+ na PORTA pin PA0 a - na GND pin
*/
```

```
// I/O definice registru pro AT90S8535
#include <90s8535.h>
#include <delay.h>
#asm
.equ __lcd_port=0x15 ;PORTC
#endasm

#include <lcd.h>
#include <stdlib.h>

#define ADC_VREF_TYPE 0x00

unsigned char pom[5];
unsigned int napeti;

// obsluha přerušení pro ADC
interrupt [ADC_INT] void adc_isr(void)
{
lcd_gotoxy(0,1);
lcd_putsf(" ");
napeti = (int)(ADCW*4.888); // 0.005 kvůli zaokrouhlení
itoa(napeti,pom);
lcd_gotoxy(0,1);
if (napeti <1000) lcd_gotoxy(1,1);
if (napeti <100) lcd_gotoxy(2,1);
if (napeti <10) lcd_gotoxy(3,1);
lcd_puts(pom);

// 20 ms delay
delay_ms(20);
// začátek nového převodu AD
ADCSR|=0x40;
}

void main(void)
{

lcd_init(16);
lcd_gotoxy(0,0);
lcd_putsf("Voltmetr demo");
// ADC inicializace

// ADC Interrupts: On
ADCSR=0x8E;
```



```
// povolení přerušení
#asm("sei")

// výběr vstupu ADC = vstup 0
ADMUX=0;

// začátek prvního převodu, jeho dokončení vyvolá přerušení ADC
ADCSR|=0x40;

// veškerou činnost zajišťuje obsluha přerušení ADC
while (1);
}
```

Kód programu je záměrně velice jednoduchý, stejně jako kód ostatních programů v této knize. Má se tím usnadnit pochopení jeho funkce začátečníkům. Může se však stát základem složitějších projektů doplněných např. o komfortnější obsluhu voltmetru, jeho propojení s PC pomocí RS232 umožňující zpracování naměřených hodnot na osobním počítači. Dokonce lze takto realizovat i osciloskop. Příkladem může být konstrukce A. Jelisejeva na www.telesys.ru včetně zdrojového kódu pro AT90S8535 v IAR C a zdrojového kódu pro PC v Delphi (najdeme i na doprovodném CD v adresáři nápady).

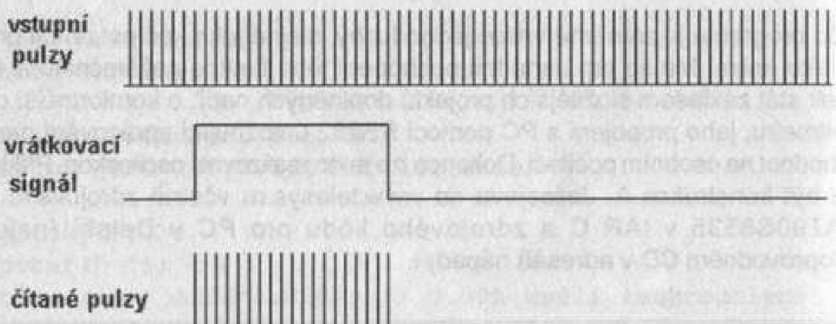
7.8 Program 8 – čítač

Již více než 30 let patří mezi oblíbené konstrukce měřič kmitočtu – čítač. V prvních desetiletích realizovaný zcela hardwarově, číslicovými obvody TTL, později i CMOS či ECL. Nyní již zcela převládají konstrukce čítačů realizovaných jednočipovými mikrokontroléry či mikropočítači, v našich poměrech často s PIC16F84 či jednočipem kompatibilním s x51. V poslední době se objevují i konstrukce s MCU AVR. Bohužel autoři těchto konstrukcí často uveřejňují jen schéma zapojení a obrazec plošných spojů. Pokud jde o program, nabízejí za úplatu prodej naprogramovaného mikrokontroléru.

Existují však i výjimky, viz např. kniha [17 str. 159–178]. Popisující čítač s AT90S2313, včetně zdrojového kódu v assembleru, propojený přes RS232 s PC, na němž pod Windows běží program sloužící jednak jako displej čítače, jednak k ovládání čítače. Výhoda znalosti zdrojového kódu je zřejmá – můžeme si zkontrolovat, že v programu není chyba a můžeme ho dále upravovat. Pro takové účely je výhodné mít zdrojový kód ve vyšším programovacím jazyce. Ukážeme si jednoduchý program v CodeVisionAVR C. Nejprve si však vysvětlíme použitou metodu měření. Tou bude metoda přímá. Její princip je zřejmý z *obr. 7.27*.

Měřený signál prochází zesilovačem/tvarovačem, který vstupní signál přemění na pulzy, které jsou jedním ze vstupních signálů vrátkovacího obvodu. Jeho druhým vstupem je vrátkovací signál, který slouží k řízení vrátkovacího obvodu – určuje kdy

propustí vstupní pulzy na výstup. Bude-li doba, po kterou propouští vstupní pulzy rovna např. 1 s, bude počet pulzů propuštěných na výstup vrátkovacího obvodu roven již kmitočtu vstupního signálu v Hz, v případě délky vrátkovacího signálu 1 ms bude počet pulzů na výstupu vrátkovacího obvodu odpovídat kmitočtu v kHz atd. Proto stačí pulzy na výstupu vrátkovacího obvodu čítat a po skončení čítání zobrazit na displeji, který tak bude zobrazovat kmitočet měřeného signálu. Při klasické realizaci měřiče kmitočtu číslicovými obvody může být vrátkovacím obvodem např. hradlo AND, pulzy jsou pak čítány dekadickými čítači. Po skončení čítání se stav čítačů přenese do nějaké vyrovnávací paměti a poté se stav čítačů vynuluje. Obsah vyrovnávací paměti se přes převodník kódu přenese na displej.



Obr. 7.27 Průběhy pulzů v měřiči kmitočtu pracující přímou metodou

Nyní si ukážeme jak realizovat měřič kmitočtu mikrokontroléru ATMEL AT90S8515. K vrátkování využijeme **čítač/časovač0**, který je součástí mikrokontroléru. Nastavením `TCCR0=0x05;` docílíme nastavení dělicího poměru předděličky 1024. Na jejím vstupu budou hodinové pulzy z krystalového oscilátoru mikrokontroléru o kmitočtu 8 MHz, takže čítač/časovač0 bude čítat pulzy o kmitočtu 8000000/1024 = 7812,5 Hz. Protože tento čítač je 8bitový, bude po načítání 256 pulzů docházet k jeho přetečení, což vyvolá přerušení. Obsluha tohoto přerušení `interrupt [TIM0_OVF] void timer0_ovf_isr(void)` softwarově realizuje vrátkovací obvod s kmitočtem vrátkování 7812,5/1024 což je cca 30,5 Hz. Proto po skončení vrátkování nebude stav čítače čítající vstupní, měřený signál odpovídat přímo kmitočtu v Hz, či jeho dekadických násobcích. Bude však přímo úměrný kmitočtu a proto k získání správného kmitočtu můžeme vypočítat pomocí konstanty přímé úměrnosti. Získáme ji jako převratnou hodnotu kmitočtu vrátkování, tj. přibližně 1/30,5. Přesná hodnota konstanty úměrnosti je 0,032768. Předpokládá ovšem, že hodinový kmitočet mikrokontroléru je nastaven přesně na 8 MHz. Pokud použijeme běžný krystal 8 MHz připojený k oscilátoru, který je součástí MCU dostaneme obvykle kmitočet o něco odlišný od 8 MHz. Místo pracného nastavování kmitočtu tohoto oscilátoru na přesnou hodnotu je snadnější změnit konstantu přímé úměrnosti.

K čítání pulzů odvozených od měřeného signálu použijeme 16bitový **čítač/časovač1**, který je rovněž součástí AT90S8515. Měřený signál, zpracovaný vstupním zesilovačem/tvarovačem, je přiveden na vstup T1 AT90S8515. Ten je v inicializační části programu nastaven tak, že **čítač/časovač1** bude měnit svůj stav s náběžnou hranou signálu na T1. Stav tohoto čítače, tj. údaj o počtu načtených pulzů je součastí registrů `TCNT1L` a `TCNT1H`. Proto při obsluze přerušení **čítače/časovače0** nejprve překopírujeme obsah těchto registrů do pomocných proměnných

```
poc1 = TCNT1L;
poc2 = TCNT1H;
```

a poté tyto registry vynulujeme. Před opuštěním obslužné rutiny příkazem `TCCR1B=0x07;` spustíme čítání měřeného signálu. Protože k dalšímu přerušení dojde za přibližně 1/30,5 sec, budou globální proměnné `poc1` a `poc2` obsahovat hodnoty závislé na měřeném kmitočtu. Příkazem

```
pocitadlo = poc1 + 256*poc2;
```

umístíme v proměnné `počitadlo` hodnotu již přímo úměrnou měřenému kmitočtu. Přesnou hodnotu kmitočtu poskytne příkaz `počitadlo = pocitadlo * 0.03053;` Proměnná `pocitadlo` je globální, takže je přístupná příkazům v nekonečné smyčce `while (1)`, zabezpečujícím zobrazování hodnot naměřeného kmitočtu na LCD připojeném k portu A.

Protože **čítač/časovač1** je 16bitový, tj. může načíst max. 65536 pulzů a vrátkovací kmitočet je cca 30,5 Hz, je maximální měřený kmitočet 65536 × 30,5 tj. cca 1,998 MHz. Výsledný zdrojový kód měřiče kmitočtu do cca **2 MHz** je:

```
MERIC KMITOCTU do 2,0 MHz 16bitovy

Chip type           : AT90S8515
Clock frequency     : 8,000000 MHz
Memory model        : Small
Internal SRAM size  : 512
External SRAM size  : 0
Data Stack size     : 128
**********************************************/


#include <90s8515.h>
#include <delay.h>
#include <stdlib.h>

// LCD displej na PORT A
#asm
.equ __lcd_port=0x1B
#endasm
```

```
#include <lcd.h>        // knihovna pro LCD

float pocitadlo=0;
unsigned int poc1 = 0;
unsigned int poc2 = 0;

unsigned char pom[8];

// obsluha prerušeni - přetečeni Timer 0
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
#asm("cli")            //zákaz přerušeni
TCCR1B=0x00;           //konec čítáni měřeného signálu
TCCR0=0x00;            //konec čítáni vrátkovacího obvodu

poc1 = TCNT1L;         //přečteni hodnoty čitače TIMER1
poc2 = TCNT1H;

TCNT1H = 0;            //vynulovani citace TIMER1
TCNT1L = 0;
TCNT0=0x00;
TCCR1B=0x07;           //začátek čítáni měřeného signálu na T1
TCCR0=0x05;            //děleni kmitočtu hodin 1024 ma
#asm("clv")            //vynulováni příznaku přetečeni
#asm("sei")            //povoleni přerušeni
//
}

void main(void)
{

// Port B initialization - B je vstup
PORTB=0x00;
DDRB=0x00;
// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: Timer 0 Stopped
TCCR0=0x05;
TCNT0=0x00;
// Timer/Counter 1 initialization
// Clock source: T1 pin Rising Edge
// Mode: Normal top=FFFFh
// OC1A output: Discon.
// OC1B output: Discon.
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0x00;
TCCR1B=0x07;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
GIMSK=0x00;
MCUCR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x02;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
// Analog Comparator Output: Off
ACSR=0x80;

// LCD module initialization
lcd_init(16);
lcd_gotoxy(0,0);
lcd_putsf("meric kmitoctu ");
// Global enable interrupts
#asm("sei")

while   (1)
      {
      lcd_gotoxy(0,1);
      lcd_putsf(" ");
      pocitadlo = poc1 + 256*poc2;   //poc1 dolnich 8bitů
                                     //poc2 hornich 8bitů
                                 //16bitoveho citace TIMER1
      //konstanta sloužici k nastaveni přesného kmitočtu
      pocitadlo = pocitadlo * 0.03053;
      if (pocitadlo<10) lcd_gotoxy(5,1);
```

```
    if ((pocitadlo<100)&(pocitadlo>9.99)) lcd_gotoxy(4,1);
    if ((pocitadlo<1000)&(pocitadlo>99.99)) lcd_gotoxy(3,1);
    if ((pocitadlo<10000)&(pocitadlo>999.99)) lcd_gotoxy(2,1);
    if ((pocitadlo<100000)&(pocitadlo>9999.99))
    lcd_gotoxy(1,1);
    if ((pocitadlo<1000000)&(pocitadlo>99999.99))
    lcd_gotoxy(0,1);
    ftoa(pocitadlo,1,pom);
    lcd_puts(pom);
    lcd_gotoxy(9,1);
    lcd_putsf("kHz ");
    delay_ms(200);

    };
}
```

Náš měřič kmitočtu měří do 2 MHz. Čítače mikrokontrolérů AVR však mohou měřit kmitočet až do poloviny kmitočtu hodin, v případě, kdy měřené pulzy jsou synchronní s kmitočtem hodin. Jinak je maximální kmitočet o něco nižší. V našem případě by se tedy dal měřit kmitočet o něco nižší než 4 MHz. Bylo by však nutné buď zvýšit kmitočet vrátkování nebo zvýšit počet bitů čítače měřeného signálu. Protože další vestavěný čítač/časovač již nemáme, můžeme počet bitů tohoto čítače zvýšit již jen softwarově. To lze velice snadno. Jeho zdrojový kód, v CD příloze označený jako *Program8B* se od výše uvedeného zdrojového kódu liší jen tím, že obsahuje další dvě globální proměnné pomocny a poc3 a obsluhu přerušení vyvolaného přetečením **čítače/časovače1**, ke kterému dochází při překročení cca 2000000 pulzů/sec do **čítače/časovače1**. Při obsluze tohoto přetečení

```
interrupt [TIM1_OVF] void timer1_overflow(void) {
pomocny = pomocny + 1;
#asm("clv")
}
```

se zvýší o 1 pomocná proměnná, a tím při obsluze přerušení **čítače/časovače0** i hodnotu pomocného sw čítače pom3. Obsluha přerušení **čítače/časovače0** obsahuje proti výše uvedenému kódu ještě příkazy

```
poc3 = pomocny;
pomocny = 0;
```

Poslední změnou je výpočet kmitočtu v nekonečné smyčce `while (1)`, který nyní bude

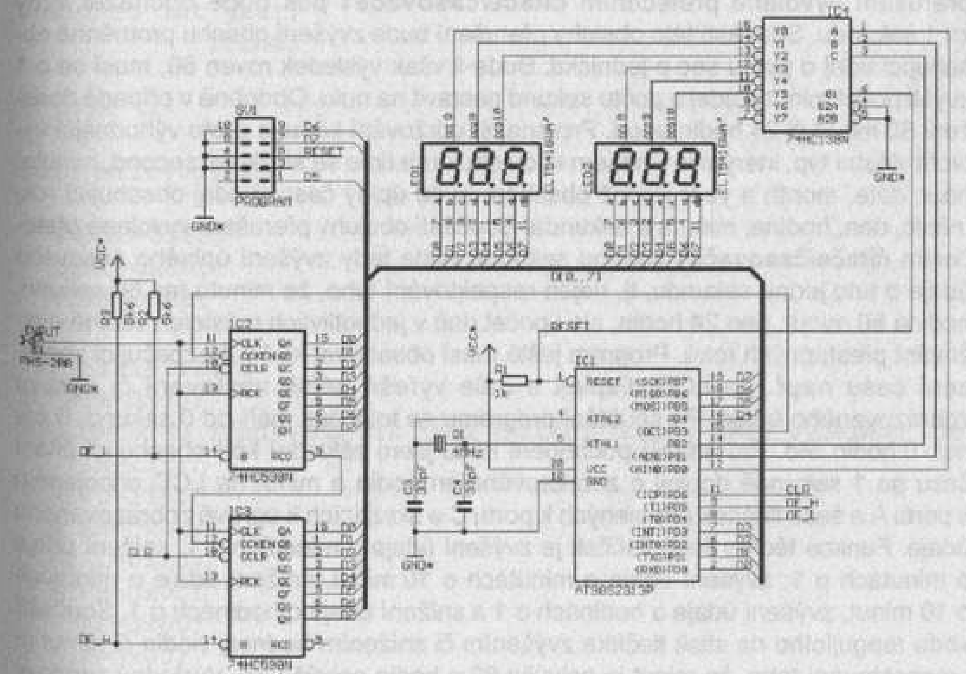
```
pocitadlo = poc1 + 256*poc2
pocitadlo = pocitadlo * 0.03053 + poc3*2000;
```


Od výpočtu v *Program8A* se liší tím, že při kmitočtech nad 2 MHz se ke kmitočtu zjištěnému pomocí 16bitového čítače přidávají právě 2 MHz. Pro nižší kmitočty bude totiž poc3 obsahovat 0, při překročení 2 MHz bude obsahovat 1. Větší číslo než 1 nebude obsahovat, protože **čítač/časovač1** má mezní kmitočet o něco nižší než 1 kmitočet hodin MCU, tj. 4 MHz.

7.8a Program 8 – měřiče kmitočtu

Pokud chceme měřit kmitočty vyšší, než je mezní kmitočet vnitřních čítačů/časovačů MCU, musíme použít vnější čítače s vyšším mezním kmitočtem. Jednou z možností je použít předděličku a tak získat pulzy s kmitočtem, který již náš čítač zvládá a údaj měřený naším AVR čítačem násobit dělicím poměrem předděličky. Tento způsob najdeme např. v již zmiňované knížce D. Matouška [17]. Jinou možností je použít vnější čítače o vyšším mezním kmitočtu přímo k čítání měřených pulzů a AVR MCU použít ke zpracování údaje o stavu těchto čítačů získaného na jejich výstupech a přivedených v paralelním tvaru na porty AVR MCU. To je případ konstrukce, kterou jsem objevil někde na internetu. Autorem je Jesper Hansen, čítač měří do 50 MHz a výsledek zobrazuje na 7segmentovém LED displeji *obr. 7.28*.

Autor uvedl i zdrojový kód v GCC.



Obr. 7.28 Zapojení měřiče kmitočtu do 50 MHz se číslovkami LED podle Jespera Hansena

7.9 Program 9 – hodiny

V předchozím příkladě jsme využili vnitřní čítače/časovače ATMEL AT90S8515 jako základ měřiče kmitočtu. Počítali jsme přitom pulzy měřeného signálu po dobu otevření vrátkovacího obvodu, která při kmitočtu krystalu 8 MHz odpovídá kmitočtu (8000000/1024)/256 tj. přibližně 30,5 Hz. Počet načítaných pulzů jsme potom násobili konstantou, abychom dostali hodnotu odpovídající měřenému kmitočtu. V řadě případů je výhodnější získat z vnitřních čítačů MCU přímo kmitočet, který je nějakým násobkem deseti. V tom případě provádíme přednastavení čítačů tak, aby k jejich přetečení docházelo za požadovanou dobu. Ukážeme si to na příkladě hodin, kdy budeme potřebovat čítat „sekundové tiky". Opět použijeme AT90S8515 s krystalem 8 MHz a k jejich vytvoření použijeme **čítač/časovač1**. Nastavením TCCR1B=0x04 docílíme toho, že tento čítač bude čítat hodinové pulzy z předděličky 256, tj. na jeho vstupu bude kmitočet 8 000 000/256 = 31 250 Hz.

K získání kmitočtu 1 Hz tedy potřebujeme, aby k přetečení čítače/časovače1 došlo vždy po 31 250 pulzech.

Protože však je tento čítač 16bitový, potřebuje k přetečení 65536 pulzů. Proto ho musíme přednastavit na 65 536 – 31 250 = 34 286. Protože 34 286 = 133 × 256 + 238 musí být TCNT1H=133 a TCNT1L=238 na začátku čítání. K volání obsluhy přerušení vyvolané přetečením **čítače/časovače1** pak bude docházet vždy za 1 sekundu. Součástí této obsluhy přerušení bude zvýšení obsahu proměnné obsahující údaj o počtu sec o jedničku. Bude-li však výsledek roven 60, musí se o 1 zvýšit počet minut a údaj o počtu sekund nastavit na nulu. Obdobně v případě dosažení 60 minut či 24 hodin apod. Pro snazší udržování kódu je proto výhodnější vytvořit vlastní typ, kterým je v našem kódu struktura time se složkami second, minute, hour, date, month a year jejichž obsahem bude úplný časový údaj obsahující rok, měsíc, den, hodina, minuta a sekunda. Součástí obsluhy přerušení vyvolané přetečením **čítače/časovače1** každou sekundu bude tedy zvýšení úplného časového údaje o tuto jednu sekundu, tj. nejen respektování toho, že minuta má 60 sekund, hodina 60 minut, den 24 hodin, ale i počet dnů v jednotlivých měsících včetně uvažování přestupných roků. Program ještě musí obsahovat kód zabezpečující zobrazení času např. na LCD displeji a dále vyřešit nějak nastavení či opravu zobrazovaného údaje. Po spuštění programu se totiž čas měří od 0 sekund, 0 minut, 0 hodin atd. Pro snazší pochopení kódu jsem základní kód obsahující čítání času po 1 sekundě doplnil o zobrazování jen hodin a minut na LCD připojeném k portu A a šesti tlačítek připojených k portu C a sloužících k opravě zobrazovaného údaje. Funkce těchto šesti tlačítek je zvýšení údaje o minutách o 1, snížení údaje o minutách o 1, zvýšení údaje o minutách o 10 minut, snížení údaje o minutách o 10 minut, zvýšení údaje o hodinách o 1 a snížení údaje o hodinách o 1. Součástí kódu reagujícího na stisk tlačítka zvýšením či snížením hodnoty hodin či minut je i respektování toho, že minut je nejvýše 60 a hodin nejvýše 24. Výsledný zdrojový kód bude:


```
#include <90S8515.h>
#include <delay.h>
#include <stdlib.h>

// LCD displej na PORT A
#asm
  .equ __lcd_port=0x1B
#endasm
#include <lcd.h> // knihovna pro LCD

unsigned int poc = 0;
unsigned char pom[8];
char not_leap(void);

typedef struct{
unsigned char second;
signed char minute;
signed char hour;
unsigned char date;
unsigned char month;
unsigned int year;
            }time;

  time t;

void main(void)
{
  PORTC=0xFF;
  DDRC=0x00;
  DDRB=0xFF;

// inicializace
TCCR1A=0x00;
TCCR1B=0x04; //zdrojem clk jsou vnitřní hodiny 8MHz vydělené 256
TCNT1H=133;
TCNT1L=238;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
GIMSK=0x00;
MCUCR=0x00;
```

```
// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x80; //

    //prescale the timer to be clock source / 128 to make it
  #asm("sei")
    //set the Global Interrupt Enable Bit
  lcd_init(16);
  lcd_gotoxy(0,0);
  lcd_putsf("hodiny ");
  while(1){
      //obsluha tlačítek nastavujících čas
      if (PINC.0 == 0) t.minute = t.minute + 1;
      if (PINC.1 == 0) t.minute = t.minute - 1;
      if (PINC.2 == 0) t.minute = t.minute + 10;
      if (PINC.3 == 0) t.minute = t.minute - 10;

      if (t.minute==60) t.minute = 0;
      if (t.minute==-1) t.minute = 59;
      if (t.minute>=60) t.minute = t.minute - 60;
      if (t.minute< 0) t.minute = t.minute + 60;

      if (PINC.4 == 0) t.hour = t.hour + 1 ;
      if (PINC.5 == 0) t.hour = t.hour - 1 ;
      if (t.hour>=24) t.hour = t.hour - 24;
      if (t.hour < 0) t.hour = t.hour + 24;

    lcd_gotoxy(0,1);
    lcd_putsf(" ");
    lcd_gotoxy(1,1);
    poc = t.hour;
    itoa(poc,pom);
    lcd_puts(pom);

    lcd_gotoxy(4,1);
    poc = t.minute;
    itoa(poc,pom);
    lcd_puts(pom);

    lcd_gotoxy(7,1);
    poc = t.second;
    itoa(poc,pom);
    lcd_puts(pom);
```



```
    delay_ms(200);
  }
}

interrupt [TIM1_OVF] void counter(void) //overflow interrupt vector
{
#asm("cli")
    if (++t.second==60)
    {
      t.second=0;
      if (++t.minute==60)
      {
        t.minute=0;
        if (++t.hour==24)
        {
          t.hour=0;
          if (++t.date==32)
          {
            t.month++;
            t.date=1;
          }
          else if (t.date==31)
          {
            if ((t.month==4) || (t.month==6) ||
(t.month==9) || (t.month==11))
            {
                t.month++;
                t.date=1;
            }
        }
        else if (t.date==30)
        {
            if(t.month==2)
            {
                t.month++;
                t.date=1;
            }
        }
        else if (t.date==29)
        {
            if((t.month==2) && (not_leap()))
            {
                t.month++;
```

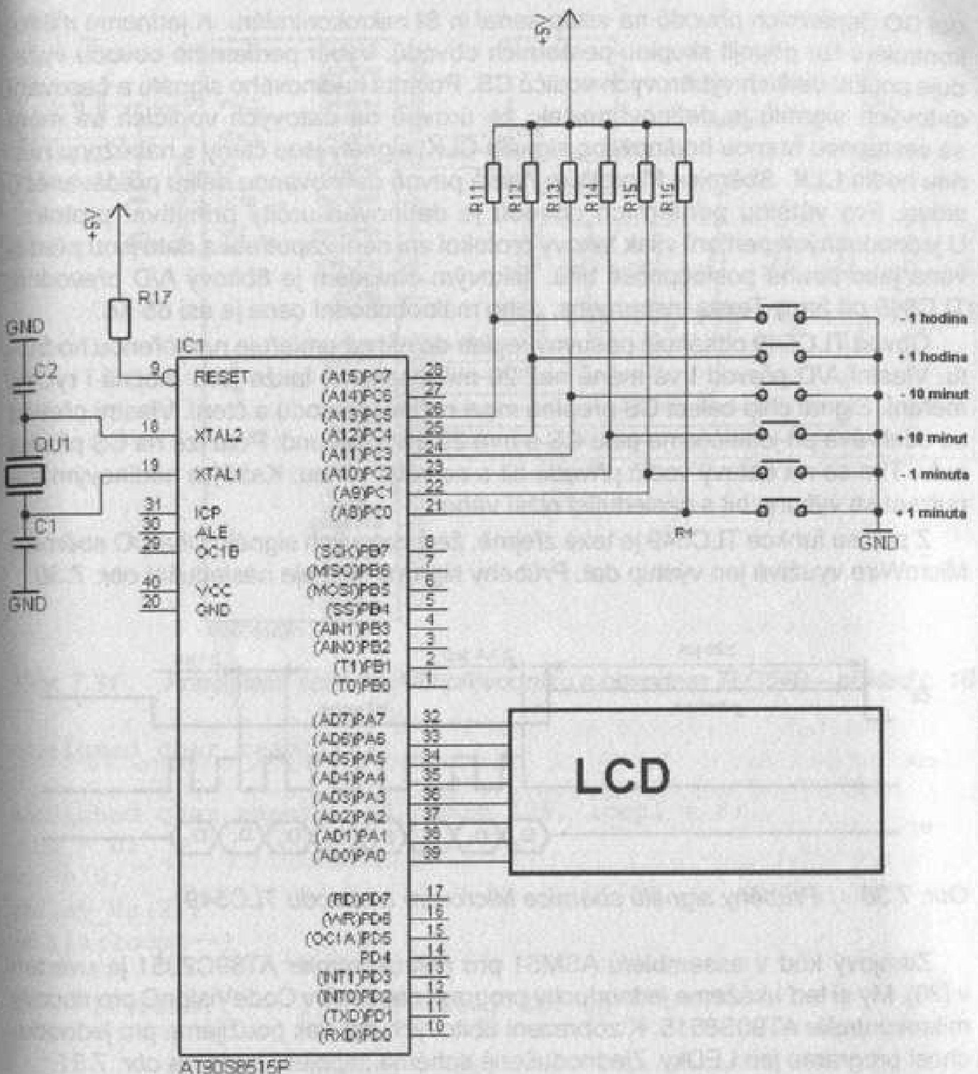
```
                t.date=1;
            }
        }
        if (t.month==13)
        {
            t.month=1;
            t.year++;
        }
    }
  }
}

TCNT1H=133;
TCNT1L=238;
#asm("clv") //vynulování příznaku přetečení
#asm("sei")

}

char not_leap(void) //přestupný rok ?
{
    if (!(t.year%100))
      return (char)(t.year%400);
    else
      return (char)(t.year%4);
}
```

Pro úplnost uvedeme ještě zjednodušené schéma hodin *obr. 7.29*.

7.10 Program 10 – sběrnice MicroWire

V předchozích příkladech jsme si ukázali jednoduché příklady, ve kterých mikrokontrolér AVR komunikuje s okolím prostřednictvím osmibitových paralelních portů, nebo pomocí vestavěných periferií, jako je UART či A/D převodník. Další možností je komunikace pomocí sériové sběrnice. Jako *sériové sběrnice* obvykle označujeme propojovací systémy, které spojují mikrokontrolér s pomocnými obvody v rámci jednoho zařízení. Sériová sběrnice šetří počet vývodů mikrokontroléru a periferních obvodů, zjednodušuje konstrukci. Je typicky tvořena dvojicí signálových vodičů. Jeden přenáší hodinový signál, hrany hodinového signálu definují časové okamžiky, ve kterých jsou na druhém vodiči prezentovány jednotlivé bity přenášených dat.

Pro některé obvody, jako jsou malé paměti EEPROM (např. AT24C02) je použití sériové sběrnice typické. Sériová sběrnice je pochopitelně pomalejší než sběrnice paralelní.



Obr. 7.29 Principiální schéma zapojení hodin s LCD displejem – příklad č. 9

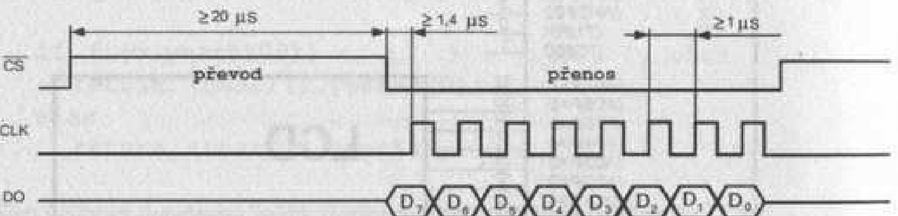
Přítomnost hodinového signálu určuje, že půjde o synchronní přenos. Tento přenos po sériové sběrnici je z obvodového hlediska jednodušší než přenos asynchronní.

Příkladem jednoduché sériové sběrnice je sběrnice **MicroWire** firmy National Semiconductor. Tato sběrnice dovoluje připojit skupinu periferních obvodů k mikrokontroléru. Je tvořena trojicí vodičů CLK, SO/DI a SI/DO. Hodinový signál CLK řídí přenos po dvou datových vodičích. První propojuje výstup mikrokontroléru *serial out* **SO** se vstupy *data in* **DI** periferních obvodů, druhý připojuje výstupy *data*

out **DO** periferních obvodů na vstup *serial in* **SI** mikrokontroléru. K jednomu mikrokontroléru lze připojit skupinu periferních obvodů. Výběr periferního obvodu vyžaduje použití dalších výběrových vodičů **CS**. Polarita hodinového signálu a časování datových signálů je definováno tak, že úrovně na datových vodičích se mění se sestupnou hranou hodinového signálu CLK, signály jsou čteny s náběžnou hranou hodin CLK. Sběrnice MicroWire nemá pevně definovanou délku předávaného slova. Pro většinu periferních obvodů je definován určitý primitivní protokol. U jednoduchých periferií však takový protokol ani není zapotřebí a data jsou předávána jako pouhé posloupnosti bitů. Takovým obvodem je 8bitový A/D převodník TLC549 od firmy Texas Instrumens. Jeho maloobchodní cena je asi 65 Kč.

Obvod TLC549 obsahuje posuvný registr do něhož umísťuje naměřenou hodnotu. Vlastní A/D převod trvá méně než 20 mikrosekund, takže jsou možná i rychlá měření. Signál chip celect CS přepíná mezi režimy převodu a čtení. Vlastní převod se odehrává při jedničce na pinu CS a trvá 20 mikrosekund. Poté lze na CS přivést nulu. Tím se na datový vodič přivede bit s největší váhou. Každým hodinovým impulzem se vysune bit s následující nižší váhou.

Z popisu funkce TLC549 je také zřejmé, že z datových signálů DI a DO sběrnice MicroWire využívá jen výstup dat. Průběhy signálů ukazuje následující *obr. 7.30*.

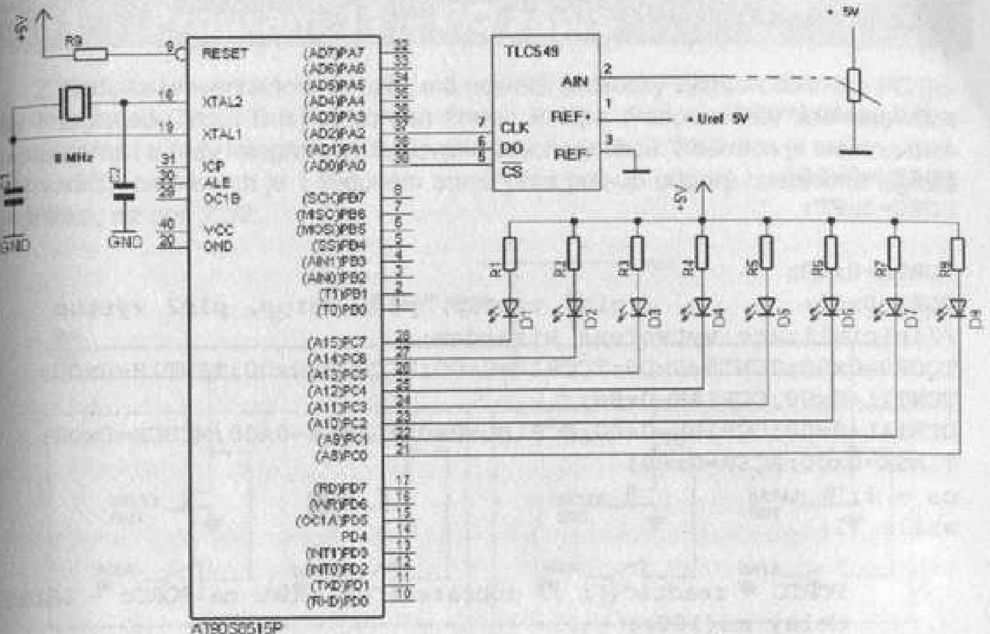


Obr. 7.30 Průběhy signálů sběrnice Microwire u obvodu TLC549

Zdrojový kód v assembleru ASM51 pro mikrokontrolér AT89C2051 je uveden v [20]. My si teď ukážeme jednoduchý program napsaný v CodeVisionC pro riscový mikrokontrolér AT90S8515. K zobrazení 8bitových dat pak použijeme pro jednoduchost programu jen LEDky. Zjednodušené schéma zapojení bude na *obr. 7.31*.

Zdrojový kód může být např.:

```
//TLC549 připojen k PORTA
//CS připojen k PA0, D0 k PA1, CLK k PA2
//výstup na PORTC - např. LEDky
#include <90s8515.h>
#include <delay.h>

#define cs PORTA.0
#define clk PORTA.2
```

Obr. 7.31 Principiální schéma A/D převodníku s obvodem TLC549 – příklad č. 10

```
unsigned char readtlc()
{
unsigned char napeti = 0, Ux = 128, loopi = 8;
clk = 0;
cs = 0;
delay_us(2);
while(loopi--)
    {
    if(PINA.1 == 1)     //čtení dat D0
        {
        napeti = napeti + Ux;
        }
    Ux=Ux/2;
    delay_us(2);
    clk = 1;            //hodinový pulz, náběžná hrana
    delay_us(2);
    clk = 0;            //hodinový pulz,sestupná hrana
    delay_us(2);
    }
cs = 1;
```

```
return napeti;
}

void main(void)
{
PORTC=0x00;
DDRC=0xFF;

PORTA=0x00;
DDRA=0x05;        //pin0 výstup, pin1 vstup, pin2 výstup
//inicializace vytvořená wizardem:
TCCR0=0x00;TCNT0=0x00;TCCR1A=0x00;TCCR1B=0x00;TCNT1H=0x00;
TCNT1L=0x00;OCR1AH=0x00;
OCR1AL=0x00;OCR1BH=0x00;OCR1BL=0x00;GIMSK=0x00;MCUCR=0x00;
TIMSK=0x00;ACSR=0x80;
cs = 1;
while (1)
     {
     PORTC = readtlc(); // zobrazení výsledku na PORTc - LEDky
     delay_ms(100);
     };
}
```

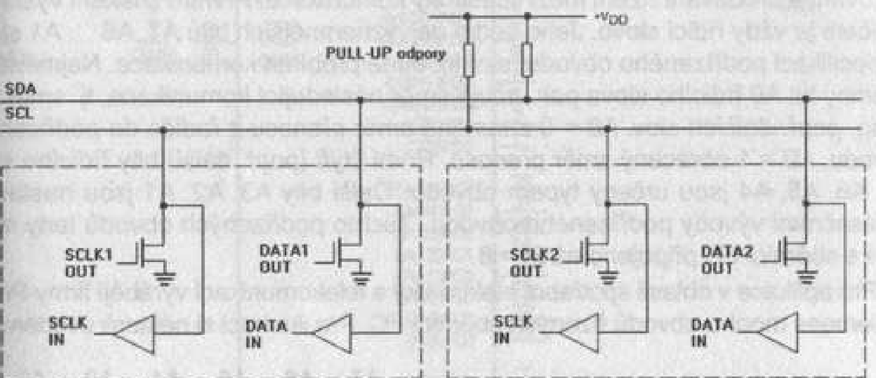
Program běží v nekonečné smyčce `while (1)`, ve které volá funkci `readtlc()` vracející číslo v rozmezí 0 až 255 v závislosti na naměřeném napětí a tuto hodnotu posílá na PORTC s připojenými diodami LED. Činnost funkce `readtlc()` je velice jednoduchá. Příkazem `cs = 1;` se spustí převod A/D. Před přečtením výsledku je třeba nastavit `cs = 0;` a nějakou dobu pomocí `delay_us(2);` počkat na ustálení stavu na CS. Hodinový signál byl přitom na nule. Poté již můžeme pomocí

```
clk = 1;        //hodinový pulz,náběžná hrana
delay_us(2);
clk = 0;        //hodinový pulz,sestupná hrana
delay_us(2);
```

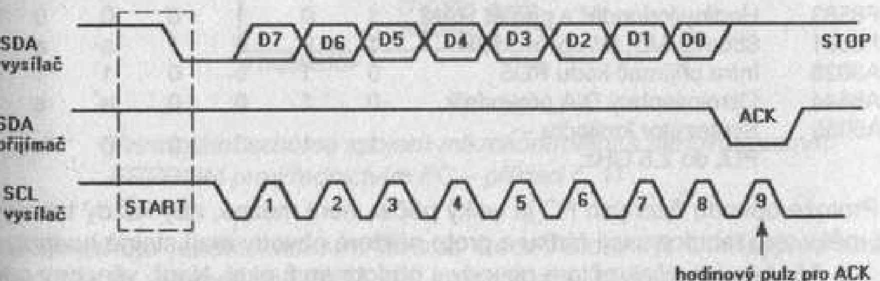
vytvářet hodinové impulzy. Protože na začátku máme proměnnou `loopi` nastavenou na osm a impulzy generujeme ve smyčce `while(loopi--)`, bude počet průchodů touto smyčkou a tedy počet hodinových impulzů právě 8. Při každém průchodu smyčkou bude před vygenerováním hodinového impulzu provedeno zkontrolování úrovně signálu na DO. Bude-li to jednička, přičte se k proměnné **napeti** číslo, odpovídající váze příslušného bitu. Data vysílá TLC549 počínajíc nejvýznamnějším bitem. V tom případě bude tato váha `Ux = 128`. Pro následující bit bude tato váha poloviční, pro další bity čtvrtinová, osminová atd. Toho docílíme příkazem `Ux=Ux/2;` prováděným při každém průchodu smyčkou.

7.11 Program 11 – I²C zápis

Z hlediska univerzálnosti použití, má největší praktický význam sběrnice I²C (Inter-Integrated-Circuit Bus) vytvořená firmou Philips. Podporu této sběrnice však nalezneme i u řady integrovaných obvodů jiných výrobců. Sběrnice je tvořena dvojicí vodičů, na kterých je v klidovém stavu přes pull-up odpory nastavena logická jednička, viz *obr. 7.32*.



Obr. 7.32 Princip zapojení sběrnice I²C



Obr. 7.33 Časování sběrnice I²C

Jeden z vodičů, značený SCL, přenáší hodinový signál. Druhý, značený SDA, slouží k synchronnímu přenosu dat *obr. 7.33*.

Budiče s otevřeným kolektorem umožňují, aby sběrnice byla využívána skupinou rovnoprávných řadičů sběrnice I²C (multimaster konfigurace). Odpovídající arbitrážní protokol, který řeší přidělení sběrnice při současném požadavku více řadičů sběrnice I²C je popsán v dokumentaci k tomuto protokolu, kterou najdete na doprovodném CD. V následujících dvou programech budeme uvažovat nejčastější případ, kdy sběrnici bude řídit jen jeden řadič sběrnice. Bude jím mikrokontrolér

AVR řízený naším programem napsaným v CodeVisionAVR C a využívající jeho knihovní funkce pro I²C.

Komunikaci zahajuje řadič prvkem START – sestupná hrana signálu SDA při jedničce signálu SCL. Dále je vysílán osmibitový znak po vodiči SDA počínaje nejvýznamnějším bitem. Vyslaný znak je potvrzován přijímačem stažením signálu SCL na úroveň nula – signál ACK (acknowledge).

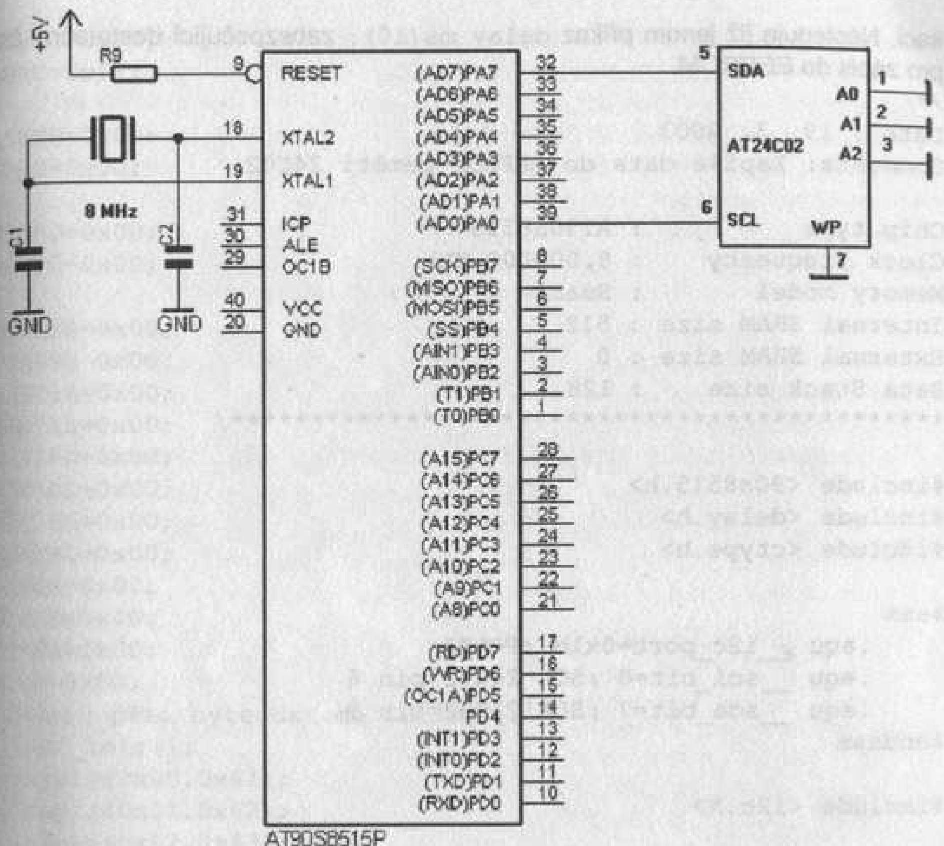
Norma I²C, na rozdíl od MicroWire, definuje i formát přenášených dat, jejich potvrzování a předávání řízení mezi účastníky komunikace. Prvním znakem vyslaným řadičem je vždy řídicí slovo. Jeho sedm nejvýznamnějších bitů A7, A6 ... A1 slouží k specifikaci podřízeného obvodu, s nímž bude probíhat komunikace. Nejméně významný bit A0 řídicího slova pak určuje směr následující komunikace, tj. směr druhého, popř. dalších slov. A0 = 0 znamená směr přenosu z řadiče do podřízeného obvodu, A0 = 1 obrácený směr přenosu. První čtyři (popř. další) bity řídicího slova A7, A6, A5, A4 jsou určeny typem obvodu. Další bity A3, A2, A1 jsou nastaveny adresačními vývody podřízeného obvodu. Těchto podřízených obvodů tedy může být ke sběrnici I²C připojeno až $2^3 = 8$.

Pro aplikace v oblasti spotřební elektroniky a telekomunikací vyrábějí firmy Philips a Siemens mnoho obvodů řízených sběrnicí I²C. Pro ilustraci si některé uvedeme:

		A7	A6	A5	A4	A3	A2	A1
PCF8574	8bitový vstup/výstup	0	1	0	0	a	a	a
PCF8577	64segmentový řadič LCD	0	1	1	1	0	1	0
PCF8578	Řadič bodového LCD	0	1	1	1	1	0	a
PCF8582A	Paměť EEPROM 256 x 8	1	0	1	0	a	a	a
PCF8583	Hodiny/kalendář a paměť RAM	1	0	1	0	0	0	a
PCF8591	8bitový A/D, D/A převodník	1	0	0	1	a	a	a
SAA3028	Infra přijímač kódu RD5	0	1	0	0	1	1	0
TDA8444	Osminásobný D/A převodník	0	1	0	0	a	a	a
TSA5055	Syntezátor kmitočtu – PLL do 2,6 GHz	1	1	0	0	0	a	a

Protože obvodů řízených I²C je velký počet, není možné, aby každý takový obvod měl v této tabulce svoji řádku a proto některé obvody mají stejné hodnoty bitů A7, A6, A5, A4. Jde však přitom obvody s obdobnou funkcí. Např. všechny paměti EEPROM mají hodnoty těchto bitů 1 0 1 0, takže např. AT24C01 či 24C02 jsou řízeny a programovány stejně, jako PCF8582A. Tyto paměti nám také poslouží v následujících dvou příkladech k podrobnějšímu vysvětlení funkce protokolu I²C i snadnosti naprogramování mikrokontroléru AVR jako řadiče I²C pomocí CodeVisionAVR C. Nejprve si ukážeme jednoduchý program umožňující naprogramovat několik byte v paměti EEPROM 24C01 či 24C02. Ta bude s MCU AVR propojena dvěmi vodiči, např. *obr. 7.34*.

Jediným úkolem následujícího programu je naprogramovat do EEPROM 5 znaků abcde tj. 0x61, 0x62, 0c63, 0x64 a 0x65 do buněk o adresách 0, 1, 2, 3 a 4. Pro splnění tohoto úkolu jsme definovali funkci **zápis** mající dva parametry – adresu buňky, do níž chceme zapsat jeden byte dat, a hodnotu těchto dat. Tělo funkce



Obr. 7.34 *Principiální schéma spojení mikrokontroléru s sériovou pamětí EEPROM prostřednictvím I²C – příklad č. 11*

zápis obsahuje jenom knihovní funkce CodeVisionAVR C. Nejprve funkce `i2c_start()` vygeneruje START I2C komunikace. Následuje přesnost tři byte do paměti EEPROM pomocí funkce `i2c_write`. První byte 0xA0 = 10100000 je řídicím slovem. V něm první čtyři bity 1010 informují, že I²C komunikuje s pamětí EEPROM, další tři bity 000 obsahují **adresu obvodu** s nímž se komunikuje. V našem případě jsme připojením pinů A0, A1 a A2 obvodu 24C02 s nulou (GND) zvolili jako tuto adresu 000. Důležité: adresa obvodu daná hardwarovým nastavením pinů A0, A1 a A2 **nemá** vůbec nic společného s adresami buněk paměti EEPROM 0 až 255. Posledním bitem řídicího slova je 0 určující, že další data budou posílána do EEPROM. Druhým bytem přenášeným do EEPROM pomocí příkazu `i2c_write(adresa);` je právě adresa buňky EEPROM, do které chceme zapsat data a třetím přenášeným bitem pomocí `i2c_write(data);` jsou právě data, která zapíšeme do EEPROM. Poslední příkaz `i2c_stop();` ukončuje I²C komuni-

kaci. Následuje již jenom příkaz `delay_ms(10);` zabezpečující dostatečný čas pro zápis do EEPROM.

```
/*
Date : 19. 3. 2003
Comments: Zapíše data do EEPROM paměti 24C02

Chip type           : AT90S8515
Clock frequency     : 8,000000 MHz
Memory model        : Small
Internal SRAM size  : 512
External SRAM size  : 0
Data Stack size     : 128
*********************************************/

#include <90s8515.h>
#include <delay.h>
#include <ctype.h>

#asm
    .equ __i2c_port=0x1b ;PORTA
    .equ __scl_bit=0 ;SCL 24C02 pin 6
    .equ __sda_bit=7 ;SDA 24C02 pin 5
#endasm

#include <i2c.h>

void zapis(unsigned char adresa,unsigned char data)
{
  i2c_start();
  i2c_write(0xA0);
  i2c_write(adresa); //adresa paměti EEPROM
  i2c_write(data);
  i2c_stop();
  delay_ms(10);
}

void main(void)
{
PORTA=0x00;
DDRA=0x01;

PORTB=0x00;
DDRB=0x00;

PORTC=0x00;
DDRC=0xFF;

PORTD=0x00;
DDRD=0x00;

TCCR0=0x00;
TCNT0=0x00;

TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
GIMSK=0x00;
MCUCR=0x00;
TIMSK=0x00;
ACSR=0x80;
//zápis pěti byte dat do EEPROM
  i2c_init();
  zapis(0x00,0x61);
  zapis(0x01,0x62);
  zapis(0x02,0x63);
  zapis(0x03,0x64);
  zapis(0x04,0x65);
while (1)
    {

    };
}
```

7.12 Program11 – I²C čtení

Následující program ukazuje přečtení prvních 32 bytů dat z paměti EEPROM 24C02 připojené k AVR MCU stejně jako v předchozím příkladě. Přečtená data jsou zobrazena na LCD displeji s 2 × 16 znaky, který je připojen k portu PORTC. Základem programu je smyčka, která se proběhne 32×. Při každém průchodu touto smyčkou, která je řízena parametrem `i`, se z paměti EEPROM z adresy `i` přečte obsah na této adrese a pak se zobrazí na LCD. Kód programu je:

```
Comments: Přečte 32 byte dat z EEPROM paměti 24C01
      a zobrazí na LCD popř. LED
Chip type           : AT90S8515
Clock frequency     : 8,000000 MHz
Memory model        : Small
Internal SRAM size  : 512
External SRAM size  : 0
Data Stack size     : 128
*********************************************/

#include <90s8515.h>
#include <delay.h>
#include <ctype.h>
// Alphanumeric LCD Module functions
#asm
  .equ __lcd_port=0x15
#endasm

#asm
   .equ __i2c_port=0x1b ;PORTA
   .equ __scl_bit=0 ;SCL 24C01 pin 6
   .equ __sda_bit=7 ;SDA 24C01 pin 5
#endasm

#include <lcd.h>
#include <i2c.h>

int i;

unsigned char d;
void main(void)
{
PORTA=0x00;
DDRA=0x01;

PORTB=0x00;
DDRB=0x00;

PORTC=0x00;
DDRC=0xFF;

PORTD=0x00;
DDRD=0x00;

TCCR0=0x00;
TCNT0=0x00;
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
GIMSK=0x00;
MCUCR=0x00;
TIMSK=0x00;
ACSR=0x80;

// LCD module initialization
lcd_init(16);
lcd_gotoxy(0,0);
for(i=0;i<32;i++)
{
  i2c_init();
  i2c_start();
  i2c_write(0xA0);
  i2c_write(i);
  i2c_start();
  i2c_write(0xA1);
  d=i2c_read(0);
  i2c_stop();
  lcd_putchar(d);
  //PORTC=d; - použijeme LEDky místo LCD
  //delay_ms(300); - v případě použití LED místo LCD
};

while (1)
      {

      };
}
```

Při každém průchodu smyčkou se pomocí `i2c_init();` a `i2c_start();` provede zahájení I²C komunikace. Poté je do EEPROM pomocí `i2c_write(0xA0)` posláno řídicí slovo 10100000 stejné jako v předchozím příkladě. Mj. znamená, že následující byte bude směřovat do EEPROM. Tímto bytem je adresa i paměti EEPROM. Její přenos zabezpečuje příkaz `i2c_write(i);`. Jeho důsledkem je, že po potvrzení jeho příjmu signálem ACK se provede nastavení čítače adres v EEPROM na adresu i. Poté se provede další start komunikace I²C, nicméně čítač adres zůstane nastaven na adrese i. Poté následuje přenos řídicího slova 0xA1,

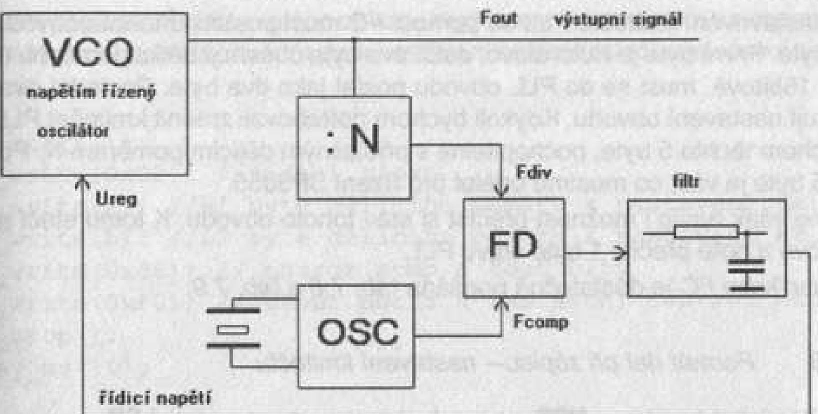
což je 10100001. Protože poslední bit je 1, je tím určen přenos následujícího byte z paměti EEPROM. Tímto bytem je právě obsah paměti EEPROM na adrese i. Ke čtení tohoto obsahu slouží příkaz `d=i2c_read(0);` využívající funkci vracející přečtenou hodnotu. Parametrem této funkce je hodnota potvrzení ACK, což je 0.

7.13 Program 11 – PLL syntezátor kmitočtu řízený I²C

V předchozích dvou příkladech jsme si ukázali že pomocí knihovny I²C z CodeVisionAVR jazyka C lze snadno naprogramovat komunikaci AVR pomocí tohoto protokolu. Ještě snadněji lze v tomto prostředí naprogramovat obsluhu některých integrovaných obvodů pomocí průvodce (wizarda) Konkrétně jde o teplotní čidla LM75 či DS1621 nebo hodiny reálného času PCF8563, PCF 8583, DS1302 a DS0307.

Snadnost naprogramování komunikace I²C v jazyce C CodeVisionAVR můžeme porovnat např. s programováním v asm51 (8bitový A/D a D/A převodník PCF8591, budič LED displeje SAA1064, 8 kanálový 6bitový D/A převodník či DTMF/modem/generátor PCD3312 v [18] či s expandérem portu PCF8574 v [19]. Z předchozího výčtu je zřejmé, ke kterým obvodům můžeme nalézt alespoň nějaký zdrojový kód programového zabezpečení komunikace těchto obvodů s nějakým „jednočipákem" pomocí I²C. I bez hlubší znalosti asm51 můžeme odtud alespoň vyčíst, jak bude vypadat řídicí slovo I²C komunikace takového obvodu a jak posílaná data. Navíc v těchto knížkách najdeme slovní popis I²C komunikace těchto obvodů, tabulky, obrázky průběhů apod., takže s našimi dosavadními znalostmi pro nás nebude obtížné naprogramovat I²C komunikaci mikrokontrolérů ATMEL AVR s těmito obvody. V případě, že nechceme či neumíme naprogramovat komunikaci I²C však můžeme použít jiný typ integrovaného obvodu, např. A/D převodníku, který I²C nepoužívá. Existuje však jistá třída obvodů, dnes snad již téměř bez výjimky ovládaná přes I²C, jejichž představitel mezi výše uvedenými obvody chybí. Jsou to syntezátory kmitočtu, obvody PLL. Proto si jako poslední příklad programování I²C komunikace předvedeme komunikaci právě s PLL obvodem. Zvolil jsem typ SP5055 (dříve MITEL Semiconductor, nyní ZARLINK Semiconductor) zapojený v satelitním tuneru. Nejprve si ukážeme princip činnosti tohoto PLL *obr. 7.35*.

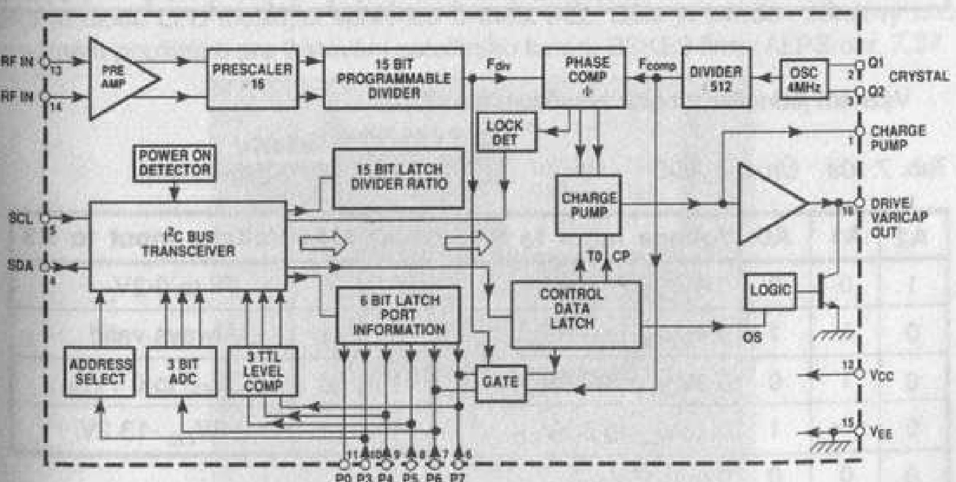
Napětím řízený oscilátor generuje signál o kmitočtu, který v nějakém rozsahu Fmin až Fmax je řízen řídicím, nejlépe stejnosměrným, napětím Ureg. Na výstupu děličky je signál o kmitočtu Fdiv = Fout/N a je ve fázově kmitočtovém detektoru porovnáván s řídicím kmitočtem Fcomp. Rozdíl v kmitočtu/fázi obou porovnávaných signálů se projeví rozdílovým napětím, které doladí VCO tak, aby tento rozdíl byl nulový. V tom případě bude regulační smyčka PLL v ustáleném stavu a výstupní kmitočet Fout bude roven **N. Fcomp**. Z toho je zřejmé, že velikostí dělicího poměru N můžeme nastavovat výstupní kmitočet. Výstupní signál bude mít stabilitu danou stabilitou krystalového oscilátoru OSC, tedy stabilitu větší, než je stabilita samotného přeladitelného oscilátoru.



Obr. 7.35 *Blokové schéma obvodu PLL*

Výstupní kmitočet bude přitom možné volit s krokem Fcomp a protože dělicí poměr N lze měnit číslicově, bude možné kmitočet PLL rovněž řídit číslicově.

Tento popis činnosti PLL je dosti zjednodušený. Teorie PLL i vlastní návrh obvodů je ve skutečnosti dosti složitý. Naštěstí to není našim problémem, neboť ho již vyřešili výrobci integrovaných obvodů PLL. Jedním takovým obvodem je syntezátor kmitočtu SP5055 pracující do 2,6 GHz. Je vyráběný řadou firem pod označením lišícím se jenom písmennou částí kódu. Vnitřní struktura tohoto obvodu je na *obr. 7.36*.



Obr. 7.36 *Vnitřní blokové schéma obvodu SP5055*

Při nastavování kmitočtu PLL se pomocí I²C musí poslat do tohoto obvodu celkem 5 byte. První byte je řídicí slovo, další dva byte obsahují dělicí poměr N. Protože N je 15bitové, musí se do PLL obvodu poslat jako dva byte. Poslední dva byty specifikují nastavení obvodu. Kdykoli bychom potřebovali změnit kmitočet PLL, poslali bychom těchto 5 byte, pochopitelně s příslušným dělicím poměrem N. Poslání těchto 5 byte je vše, co musíme udělat pro řízení SP5055.

Máme však navíc i možnost přečíst si stav tohoto obvodu. K tomu stačí poslat řídicí slovo a poté přečíst 1 byte stavu PLL.

Komunikace I²C je dostatečně popsána *tab. 7.8* a *tab. 7.9*:

Tab. 7.8 Formát dat při zápisu – nastavení kmitočtu

	MSB							LSB		
Address	1	1	0	0	0	MA1	MA0	0	A	**Byte 1**
Programmable divider	0	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	A	**Byte 2**
Programmable divider	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	A	**Byte 3**
Charge pump and test bits	1	CP	T1	T0	1	1	1	OS	A	**Byte 4**
I/O port control bits	P7	P6	P5	P4	P3	X	X	P0	A	**Byte 5**

Tab. 7.9 Formát dat při čtení – čtení stavu PLL

Address	1	1	0	0	0	MA1	MA0	1	A	**Byte 1**
Status byte	POR	FL	I2	I1	I0	A2	A1	A0	A	**Byte 2**

Význam jednotlivých bitů vysvětluje *tab. 7.10*.

Tab. 7.10a Úrovně ADC

A2	A1	A0	Voltage input to P6
1	0	0	$0.6V_{CC}$ to 13.2V
0	1	1	$0{\cdot}45V_{CC}$ to $0{\cdot}6V_{CC}$
0	1	0	$0{\cdot}3V_{CC}$ to $0{\cdot}45V_{CC}$
0	0	1	$0{\cdot}15V_{CC}$ to $0{\cdot}3V_{CC}$
0	0	0	0 to $0.15V_{CC}$

Tab. 7.10b Výběr adres

MA1	MA0	Voltage input to P3
0	0	0V to $0{\cdot}2V_{CC}$
0	1	Always valid
1	0	$0{\cdot}3V_{CC}$ to $0{\cdot}7V_{CC}$
1	1	$0{\cdot}8V_{CC}$ ·13.2V

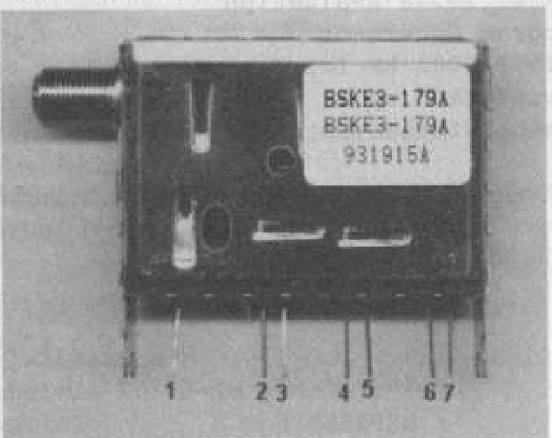
Bližší popis těchto bitů i celého obvodu najdeme v dokumentaci na doprovodném CD, nám však bude stačit vědět, že k nastavení kmitočtu stačí následující sekvence příkazů

```
i2c_start();
i2c_write(0xC2); //řídicí slovo
i2c_write(a); //HI byte dělicího poměru k nastavení kmitočtu
i2c_write(b); //LO byte dělicího poměru k nastavení kmitočtu
i2c_write(0x8E); // charge pump a test bity
i2c_write(0xF0); //output ports a nastavení control bits
i2c_stop();
delay_ms(10);
```

ve které za sebou následuje pět volání funkce `i2c_write` odpovídající pěti řádkům tabulky. Jediné, co budeme měnit, je obsah proměnných **a** a **b** typu **unsigned char** obsahujících dělicí poměr N. K případnému přečtení stavu PLL použijeme

```
i2c_start();
i2c_write(0xC3);
z = i2c_read(0);
i2c_stop();
delay_ms(10);
```

Obsahem proměnné **z** pak bude přečtené stavové slovo obvodu PLL, z něhož je pro nás nejzajímavější druhý nejvíce významný bit – FL. Obsahuje-li jedničku, je smyčka PLL uzavřena a obvod pracuje správně. Při nule není PLL „zavěšen", což je chyba. Toto stavové slovo můžeme poslat na některý z portů MCU a na bit 6 pak připojit diodu LED indikující správnou činnost PLL. Dále je uveden zdrojový kód programu použitého pro testování satelitního tuneru BSKE3 firmy ALPS *obr. 7.37*.



Obr. 7.37 Satelitní tuner použitý v testovacím zapojení

Význam vývodů je:

1 LNC +12/+18 V ... Přepínání vertikální/horizontální polarizace.
2 napájení +5 V.
3 výstup.
4 PORT3 PLL – AFC.
5 +30 V pro ladění.
6 SDA I²C.
7 SCL I²C.

Stačí tedy jen připojit napájení +5 V a +30 V a dále připojit piny 6 a 7 tj. SDA a SCL k mikrokontroléru AVR. Použijeme např. AT90S8515. K PORTC a PORTD tohoto mikrokontroléru připojíme přepínače sloužící k nastavování 15 bitů děličky PLL a tím k nastavení výsledné kmitočty. Komunikaci I²C jsme umístili na PORTA. Signál SCL k bitu 0, SDA k bitu 7 tohoto portu. Signalizaci „zavěšení" PLL bude provádět dioda LED připojená přes předřadný odpor k PORTB, bitu 6 a katodou k GND, takže bude svítit při 1 na výstupu MCU.

Program běží v nekonečné smyčce. Při každém průchodu smyčkou nejprve sejme hodnoty z portů C a D, tj. požadovaný kmitočet. Poté pošle 5 byte do SP5055 a tím provede přeladění satelitního tuneru na požadovaný kmitočet a nakonec přečte stavové slovo SP5055 a při zavěšeném PLL rozsvítí LED. Většina kódu byla přitom vytvořena wizardem. Výpis tohoto kódu:

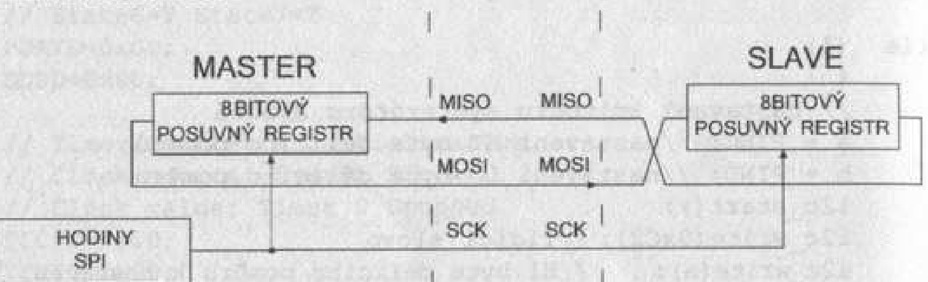
```
/*********************************************
This program was produced by the
CodeWizardAVR V1.23.7a Evaluation
Automatic Program Generator
© Copyright 1998-2002 HP InfoTech s.r.l.
http://www.hpinfotech.ro
e-mail:office@hpinfotech.ro

Project : řízení PLL satelitního tuneru s SP5055
Version :
Date    : 23. 3. 2003
Autor   : V.V.
Company :
Comments:

Chip type         : AT90S8515
Clock frequency   : 8,000000 MHz
Memory model      : Small
Internal SRAM size : 512
External SRAM size : 0
Data Stack size   : 128
*********************************************/

#include <90s8515.h>

// I2C Bus functions
#asm
   .equ __i2c_port=0x1B
   .equ __sda_bit=7
   .equ __scl_bit=0
#endasm
#include <i2c.h>
#include <delay.h>
// Declare your global variables here
unsigned char a,b,z;
void main(void)
{
// Declare your local variables here

// Input/Output Ports initialization
// Port A initialization
// Func0=Out Func1=In Func2=In Func3=In Func4=In Func5=In
// Func6=In Func7=Out
// State0=0 State1=T State2=T State3=T State4=T State5=T
// State6=T State7=0
PORTA=0x00;
DDRA=0x81;

// Port B initialization
// Func0=Out Func1=Out Func2=Out Func3=Out Func4=Out
// Func5=Out Func6=Out Func7=Out
// State0=0 State1=0 State2=0 State3=0 State4=0 State5=0
State6=0 State7=0
PORTB=0x00;
DDRB=0xFF;

// Port C initialization
// Func0=In Func1=In Func2=In Func3=In Func4=In Func5=In
// Func6=In Func7=In
// State0=T State1=T State2=T State3=T State4=T State5=T
```

```
// State6=T State7=T
PORTC=0x00;
DDRC=0x00;

// Port D initialization
// Func0=In Func1=In Func2=In Func3=In Func4=In Func5=In
// Func6=In Func7=In
// State0=T State1=T State2=T State3=T State4=T State5=T
// State6=T State7=T
PORTD=0x00;
DDRD=0x00;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: Timer 0 Stopped
TCCR0=0x00;
TCNT0=0x00;

// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: Timer 1 Stopped
// Mode: Normal top=FFFFh
// OC1A output: Discon.
// OC1B output: Discon.
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
GIMSK=0x00;
MCUCR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x00;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
// Analog Comparator Output: Off
ACSR=0x80;

// I2C Bus initialization
i2c_init();

while  (1)
      {
      // nastaveni kmitočtu syntezátoru SP5055
      a = PINC; //nastavení HI byte dělicího poměru
      b = PIND; //nastavení LO byte dělicího poměru
      i2c_start();
      i2c_write(0xC2); //řídicí slovo
      i2c_write(a);    //HI byte dělicího poměru k nastavení
                       //kmitočtu
      i2c_write(b);    //LO byte dělicího poměru k nastavení
                       //kmitočtu
      i2c_write(0x8E); //charge pump a test bity
      i2c_write(0xF0); //output ports a nastavení control
                       //bits
      i2c_stop();
      delay_ms(10);
      //přečtění stavu PLL SP5055
      i2c_start();
      i2c_write(0xC3);
      z = i2c_read(0);
      i2c_stop();
      delay_ms(10);
      PORTB = z;
      };
}
```

Protože napětím řízený oscilátor pracuje jen v určitém pásmu kmitočtů a dále smyčku PLL ovlivňují ještě např. parametry filtru, FD apod., je fázový závěs PLL „zavěšen" pro dělicí poměr N jen v určitém pásmu. V mém konkrétním případě to bylo rozmezí 3800hex až 3FFFhex, což bylo indikováno svitem LED. Při číslech o něco větších či menších než je toto rozmezí, dioda LED poblikává – obvod se snaží zavěsit. K bliknutí LED dochází i při přepínání významnějších bitů v pásmu zachycení, pak se smyčka ustálí a LED svítí trvale. Pro N dosti odlišná od dělicích poměrů při zavěšeném PLL již LED nesvítí vůbec, stejně jako např. při odpojení 30 V pro ladění VCO či vysazení kmitání VCO způsobeném např. dotykem ruky na jeho rezonanční obvod.

7.14 Program 12 – SPI

Další jednoduchou sériovou sběrnicí, jejíž programové zabezpečení je možné jednoduše uskutečnit v CodeVisionAVR C je sběrnice SPI (Serial Periphal Interface). Tuto sběrnici vytvořila firma Motorola. Je tvořena trojicí signálů. Hodinový signál SCK (Serial Clock) je generován řadičem, signály MOSI (Master Out/Slave In) a MISO (Master In/Slave Out) propojují posuvné registry řadiče a podřízeného obvodu do kruhu *obr. 7.38*.



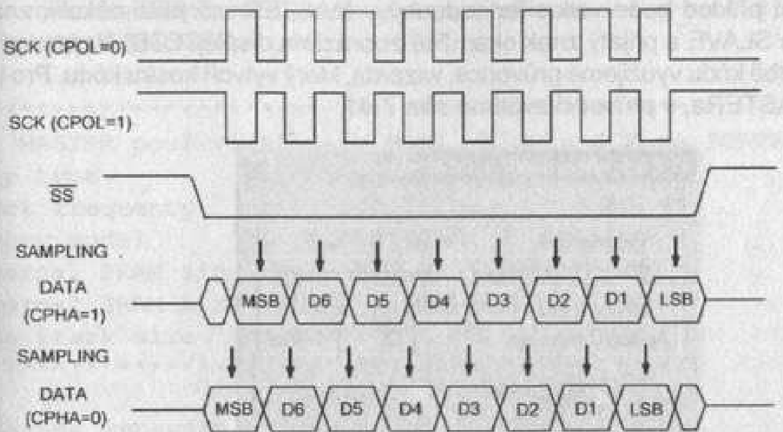
Obr. 7.38 Princip sběrnice SPI

Kromě toho je možné použít ještě další signál, a to $\overline{SS}$ (Slave Select) umožňující připojit k jednomu řadiči více podřízených obvodů. My si ukážeme nejjednodušší případ, tj. komunikaci jednoho řadiče s jedním podřízeným obvodem.

Jak je patrné z *obr. 7.39*, sestává sběrnice SPI v podstatě ze dvou vzájemně propojených posuvných registrů, které jsou taktovány společným hodinovým signálem. Jeho kmitočet přitom může většinou dosahovat až 1 MHz, v případě ATMEL MCU řady AT90S až 1/4 kmitočtu systémových hodin MCU tj. např. pro AT90S8515-8 to jsou 2 MHz. Jako základní atomickou přenosovou operaci nad SPI přitom chápeme vzájemné překopírování obsahu propojených posuvných registrů. Je zřejmé, že tato operace bude vyžadovat celkem osm hodinových taktů SCK. Při plném vytížení kanálu tak můžeme pomocí SPI provozovat duplexní přenos rychlostí až 2000000/8 bytů za sekundu (v případě AT90S8515-8), což je poměrně solidní výkon. Mimo jiné je to také jeden z hlavních důvodů, proč se tato architektura vlastně používá.

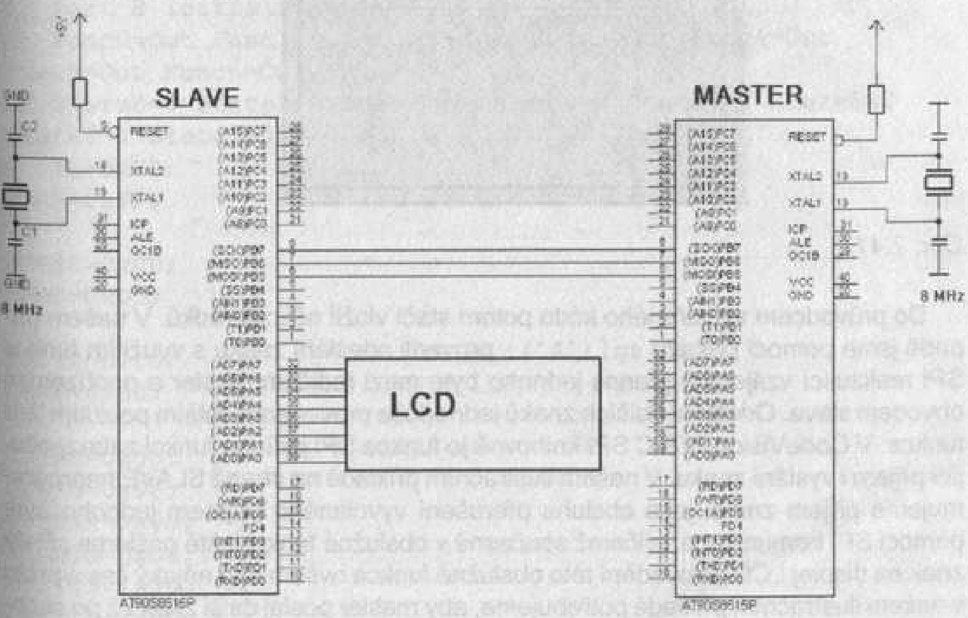
Aby bylo možné připojit i obvody, které neodpovídají jednoznačné volbě polarity hodinového a datového signálu, lze polaritu signálu SCK a polaritu datových signálů MOSI a MISO na straně řadiče nastavit *obr. 7.39*.

Sběrnice SPI byla původně vyvinuta firmou MOTOROLA, která vyrábí řadu periferních obvodů ovládaných sběrnicí SPI jako např. budiče displeje LED MC14489 a MC14499, budiče displeje LCD MC145453 a MC145000/1, hodiny reálného času MC68HC68T1, EEPROM MCM2814, A/D převodníky MC145040 a MC145050, D/A převodník MC144110/1 či obvody PLL MC14515x. Další typy vyrábí i jiné firmy. Takovým obvodem je AD7896. Jeho využití jako mV-metr, připojený přes SPI k AT90S8515, který



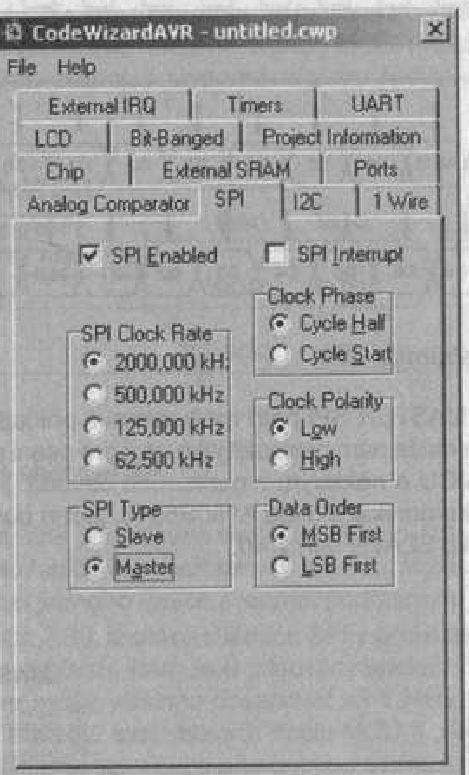
Obr. 7.39 Časování sběrnice SPI

je naprogramován jako MASTER SPI patří mezi vzorové příklady v adresáři examples CodeVisionAVR C a může nám posloužit jako vzor pro psaní programu pro řízení SPI komunikace mezi MCU a některým z periferních obvodů. My si ukážeme SPI komunikaci na případě komunikace dvou AT90S8515, jeden bude naprogramován jako MASTER, druhý jako SLAVE *obr. 7.40*.



Obr. 7.40 Principiální zapojení mikrokontrolérů pomocí sběrnice SPI – příklad č. 12

Ilustrační příklad bude velice jednoduchý – MASTER odvysílá několik znaků, které přijme SLAVE a přijatý znak okamžitě zobrazí na displeji LCD. V obou případech při tvorbě kódu využijeme průvodce, wizarda, který vytvoří kostru kódu. Pro tvorbu kódu MASTERa, v průvodci zvolíme *obr. 7.41*.



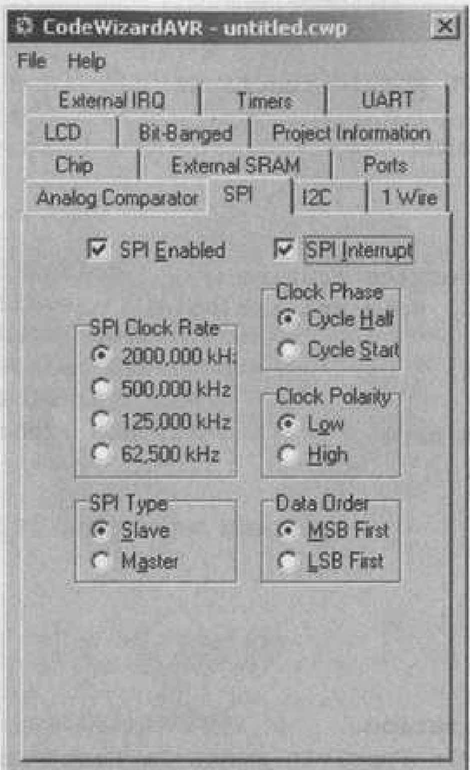
Obr. 7.41

Do průvodcem vytvořeného kódu potom stačí vložit několik řádků. V našem případě jsme pomocí příkazu `spi('A');` provedli odeslání znaku s využitím funkce SPI realizující vzájemný přenos jednoho byte mezi řadičem master a podřízeným obvodem slave. Odeslání dalších znaků jednoduše provedeme dalším použitím této funkce. V CodeVisionAVR C SPI knihovně je funkce SPI jedinou funkcí zabezpečující příjem i vyslání znaku. V našem ilustračním příkladě na straně SLAVE naprogramujeme příjem znaku jako obsluhu přerušení vyvolaného příjmem jednoho byte pomocí SPI komunikace, přičemž současně v obslužné funkci ještě pošleme přijatý znak na displej LCD. Provádění této obslužné funkce ovšem trvá nějaký čas a proto v našem ilustračním příkladě potřebujeme, aby master poslal další znak až po skončení této funkce. Proto na straně řadiče MASTER jsme mezi příkazy realizující odeslání znaku zařadili časové prodlevy pomocí příkazu `delay_us(50);` Výsledný kód na straně MASTERa bude:

```
/*************************************************
SPI MASTER používá signály MOSI, MISO a SCK na PORTB
Chip type           : AT90S8515
Clock frequency     : 8,000000 MHz
Memory model        : Small
Internal SRAM size  : 512
External SRAM size  : 0
Data Stack size     : 128
*************************************************/

#include <90s8515.h>
#include <spi.h>
#include <Delay.h>

void main(void)
{ //inicializace - kód vytvořen wizardem
PORTA=0x00;
DDRA=0xFF;

// Port B initialization
// Func0=Out Func1=Out Func2=Out Func3=Out Func4=Out
Func5=Out Func6=Out Func7=Out
// State0=0 State1=0 State2=0 State3=0 State4=0 State5=0
State6=0 State7=0
PORTB=0x00;
DDRB=0xFF;

PORTC=0x00;
DDRC=0x00;

PORTD=0x00;
DDRD=0x00;

TCCR0=0x00;
TCNT0=0x00;
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
```

```
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
GIMSK=0x00;
MCUCR=0x00;
TIMSK=0x00;

ACSR=0x80;

// SPI initialization
// SPI Type: Master
// SPI Clock Rate: 2000,000 kHz
// SPI Clock Phase: Cycle Half
// SPI Clock Polarity: Low
// SPI Data Order: MSB First
SPCR=0x50;
delay_ms(100);
spi('A');
delay_us(50);
spi('h');
delay_us(50);
spi('o');
delay_us(50);
spi('j');
delay_us(50);
spi(' ');
delay_us(50);
spi('B');
delay_us(50);
spi('E');
delay_us(50);
spi('N');
while (1)
      {
      // Place your code here

      };
}
```

Rovněž kostru kódu na straně podřízeného obvodu SPI vytvoříme pomocí průvodce. Jako volbu použijeme *obr. 7.42*.



Obr. 7.42

Dále ještě vybereme záložku LCD a v ní zvolíme PORTA a pak už jen necháme průvodce vygenerovat kód.

V místě označeném `// Place your code here` v obslužné funkce přerušení doplníme **jediný příkaz** `lcd_putchar(data);` a tím máme vytvořen kód pro SPI komunikaci na straně SLAVE :

```
----------------------------------------------------------------
/*********************************************
SPI SLAVE
Chip type           : AT90S8515
Clock frequency     : 8,000000 MHz
Memory model        : Small
Internal SRAM size  : 512
External SRAM size  : 0
Data Stack size     : 128
*********************************************/
```

```
#include <90s8515.h>

// Alphanumeric LCD Module functions
#asm
.equ __lcd_port=0x1B
#endasm
#include <lcd.h>

// SPI interrupt service routine
interrupt [SPI_STC] void spi_isr(void)
{
unsigned char data;
data=SPDR;
// Place your code here
lcd_putchar(data);
}

void main(void)
{
PORTA=0x00;
DDRA=0xFF;

// Port B initialization
// Func0=In Func1=In Func2=In Func3=In Func4=In Func5=In
// Func6=Out Func7=In
// State0=T State1=T State2=T State3=T State4=T State5=T
// State6=0 State7=T
PORTB=0x00;
DDRB=0x40;

PORTC=0x00;
DDRC=0x00;

PORTD=0x00;
DDRD=0x00;

TCCR0=0x00;
TCNT0=0x00;
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
GIMSK=0x00;
MCUCR=0x00;
TIMSK=0x00;
ACSR=0x80;

// SPI initialization
// SPI Type: Slave
// SPI Clock Rate: 2000,000 kHz
// SPI Clock Phase: Cycle Half
// SPI Clock Polarity: Low
// SPI Data Order: MSB First
SPCR=0xC0;

// Clear the SPI interrupt flag
#asm
  in r30,spsr
  in r30,spdr
#endasm

// LCD module initialization
lcd_init(16);
lcd_putsf("prijem z SPI");
// Global enable interrupts
#asm("sei")
lcd_gotoxy(0,1);
while (1)
      {
      // Place your code here

      };
}
```

Funkce tohoto programu je jednoduchá – program běží v nekonečné (a prázdné) smyčce. Při příjmu znaku ze sběrnice SPI je vyvoláno přerušení, při jeho obsluze se přijatý byte dat pošle na displej LCD.

Tento příklad, kódy na straně MASTER i SLAVE, je opravdu jen **ilustrační**. Nešikovnost našeho kódu spočívá v tom, že sice máme nastaven rychlý přenos 2000000 bitů/sec, ale mezi odesláním znaku čekáme 50 mikrosekund.

SPI je totiž velice jednoduchý, provádí vždy pouze vzájemnou výměnu jednoho byte mezi MASTER a SLAVE. Není zde protokolem definováno např. potvrzení příjmu znaku, jako je tomu např. u I²C. Proto je potřeba si nadefinovat a naprogramo-

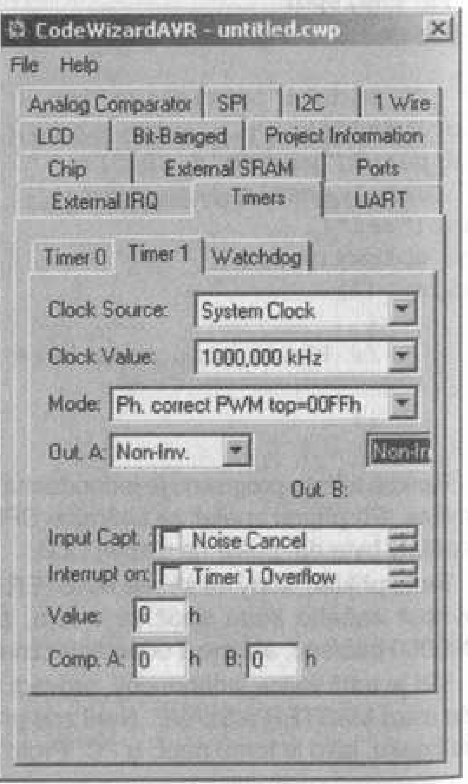
vat doplnění SPI o další funkce. V našem ilustračním příkladě jsme si vlastně vytvořili „komunikační protokol", ve kterém jsme místo potvrzení, tj. nějakého ACK, jen nechali dost času na nějakou činnost na straně příjmu.

Jinou možností by bylo např. nejprve poslat jeden či několik byte s nějakými služebními údaji a teprve potom vlastní data, data přenášet po paketech atd. Takto si můžeme i naprogramovat vlastní LAN.

7.14 Program 13 – PWM

I v tomto příkladu si ukážeme jednoduchost programování pomocí CodeVisionAVR C. Mikrokontroléry ATMEL AVR, stejně jako mikrokontroléry či jednočipové mikropočítače jiných výrobců umožňují generovat pulzně-šířkově modulovaný (PWM) signál. Využívá se přitom vnitřní čítač/časovač, který je třeba nastavit do režimu generování PWM. Střída generovaného PWM signálu je dána hodnotami v registru OCR1. To je vše, co při použití CodeVisionAVR C potřebujeme vědět. Předpokládejme např., že máme generovat PWM signál, jeho střídu budeme ovládat pomocí přepínačů připojených k PORTB a tedy zadávajících do tohoto portu libovolné 8bitové číslo tj. 0x00 až 0xFF.

Pro vytvoření kódu opět použijeme průvodce, wizarda, kde v záložce pro čítač/časovač1 nastavíme *obr. 7.43*.



Obr. 7.43

Celá naše další práce bude jen vložení dvou řádků kódu

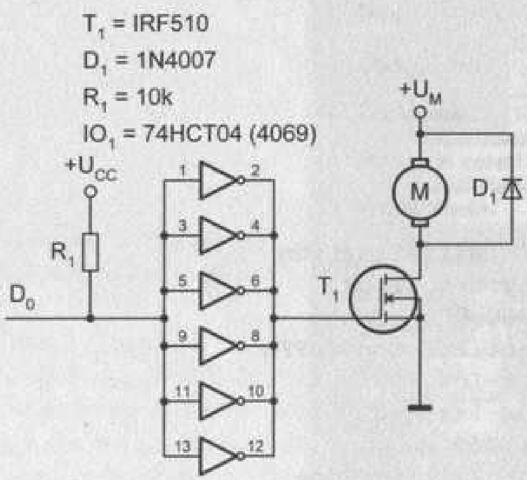
```
pom = PINA;
OCR1AL = pom;
```

do nekonečné smyčky v kódu vytvořeném průvodcem, takže výsledný program má kód:

```
/*********************************************
Comments: výstup bude PD5 - střída tohoto signálu se bude
řídit binární
 hodnotou přivedenou na PORTB

Chip type           : AT90S8515
Clock frequency     : 8,000000 MHz
Memory model        : Small
Internal SRAM size  : 512
External SRAM size  : 0
Data Stack size     : 128
*********************************************/

#include <90s8515.h>

unsigned char pom;
void main(void)
{
PORTA=0x00;
DDRA=0x00;
PORTB=0x00;
DDRB=0x00;
DDRC=0x00;
PORTD=0x00;
DDRD=0x20;
TCCR0=0x00;
TCNT0=0x00;

// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: 1000,000 kHz
// Mode: Ph. correct PWM top=00FFh
// OC1A output: Non-Inv.
// OC1B output: Non-Inv.
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0xA1;
```

```
TCCR1B=0x02;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
GIMSK=0x00;
MCUCR=0x00;
TIMSK=0x00;
ACSR=0x80;

while  (1)
      {
      pom = PINA;
      OCR1AL = pom;
      };
}
```

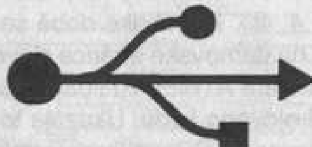
Činnost programu je až směšně jednoduchá. V nekonečné smyčce se neustále snímají hodnoty z PORTB a kopírují se do registru OCR1 čítače/časovače1 a tím se řídí střída generovaného signálu. Takový signál se může použít např. jako řídicí signál v regulátoru otáček stejnosměrného motoru. Využívá se přitom toho, že tento signál má stále stejnou amplitudu a změnou střídy se tedy mění jeho střední hodnota, na níž jsou pak závislé otáčky motoru. Stačí tedy elektromotor napájet přes zesilovač buzený z našeho mikrokontroléru. Příkladem konkrétního zapojení může být zapojení *obr. 7.44* (obrázek je použitý z [18]).



Obr. 7.44 *Jeden z příkladů konkrétního připojení motoru k mikrokontroléru [18]*

7.15 Program 14 – USB

USB rozhraní se během posledních let stalo zcela běžnou součástí spotřební elektroniky připojitelné k počítači a již téměř vytlačilo klasický sériový port RS232 a někde dokonce i paralelní port. Koupit dnes notebook s klasickým sériovým rozhraním začíná být problém a proto jsou nuceni i výrobci a uživatelé speciálních aplikací postupně přecházet na USB. Definice rozhraní USB je provedena ve verzi 1.1 a následně ve verzi 2.0 obsahující již možnost vysokých rychlostí komunikace.



Obr. 7.45 *Symbol rozhraní USB*

Základní parametry rozhraní USB:

- Komunikační rychlost od 1,5 Mbit/s do 480 Mbit/s.
- Komunikační vzdálenost do 5 m.
- Možnost připojení více zařízení.
- Rozhraní poskytuje 5V napájení.
- USB zajišťuje správné přidělování prostředků (IRQ, DMA apod.).

Programové vybavení pro ovládání rozhraní USB odpovídá plně standardu plug & play již několik let. Jsou definovány dvě verze fyzické vrstvy USB 1.1. Pro USB verze 2.0 byla doplněna nejrychlejší vrstva:

- High Speed – 480 Mbits/s.
- Full Speed – 12 Mbits/s.
- Low Speed – 1,5 Mbits/s.

Všechny verze propojení mohou být použity a provozovány současně pro připojení různých typů periferií k jednomu počítači. Uvedené verze se od sebe liší jak provedením kabelu, tak elektrickými parametry rozhraní připojeného zařízení. Protokol USB není rozhodně jednoduchý, stejně jako i jeho hardwarová či softwarová implementace v mikrokontroléru, a dále tvorba odpovídajících driverů pro PC komunikující přes USB s tímto mikrokontrolérem. Základní informace o USB lze získat např. v knize [19].

Implementaci USB na straně mikrokontroléru lze provést

- softwarově
- hardwarově

Softwarovou implementací se zabývá např. článek „Implementace rozhraní USB do mikrokontroléru na úrovni firmware" v časopisu Sdělovací technika 6/2003, nebo článek „Implementace USB do mikrokontroléru SX" uvedený na www.hw.cz a www.mcu.cz a publikující zdrojový kód M. Hetheringtona pro mikrokontrolér SX28AC.

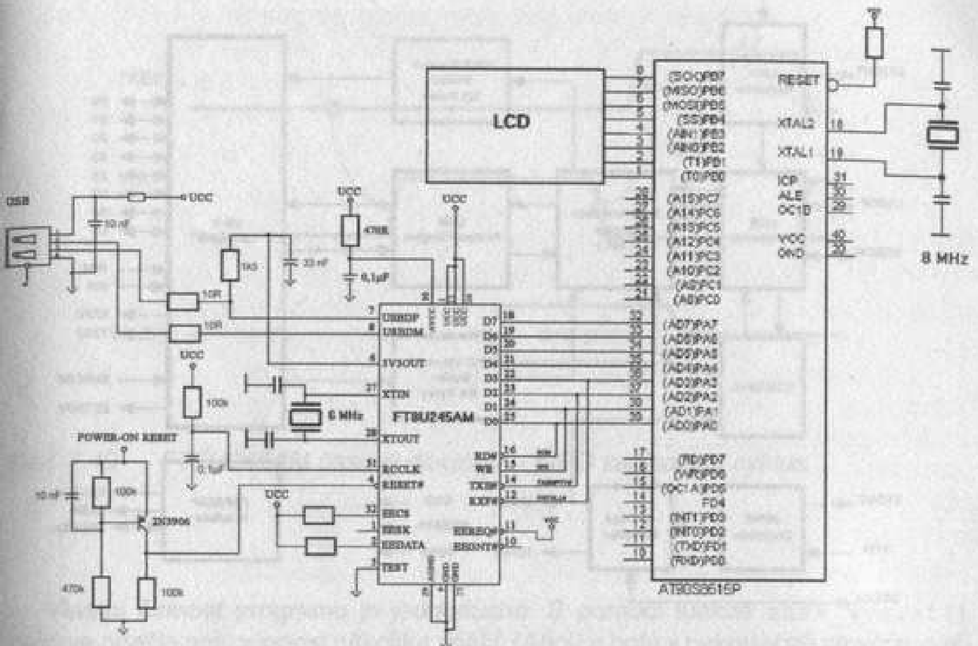
Na těchto servrech se před časem objevil i článek popisující implementaci USB na 8051, což byl sice aprílový žertík ruského konstruktéra, nicméně počtem návštěv byl tento článek dlouhou dobu nejčtenějším, stejně jako mnoha diskuzními příspěvky (včetně mého, kdy jsem upozornil P. T. diskutéry na skutečnost, že na originální ruské stránce je uvedeno datum 1.4. ☺). Po nějaké době se na těchto serverech objevil článek vycházející z informací na domovské stránce slovenského konstruktéra implementujícího USB v mikrokontroléru ATMEL AT90S2313-10 „přetaktovaného" na 12 MHz, ovšem bez uvedení zdrojového kódu. Ukazuje to na potřebu mít možnost jednoduše a lacině implementované USB v mikrokontroléru.

Hardwarová implementace USB je v současné době častější a nemusí být ani příliš drahá. Jednou z možností je použít mikrokontrolér, který kromě řady periferií má hw implementované i USB. Příkladem mohou být výrobky ATMELu AT43USB320A a AT43USB355 obsahující 8bitové RISCové jádro AVR.

Další možností je použít běžný MCU AVR k němu připojit další obvod hw implementující USB. Některé takové USB obvody najdeme popsané v [19]. Další možností je navrhnout si hardware sami – tj. zapojení NRZI kodéry a dekodéry apod. a realizovat je např. pomocí FPGA (Field Programmable Gate Array).

Zatím popisované přístupy vyžadují od vývojáře znalost protokolu USB a nejsou rozhodně příliš jednoduché. Existuje ale i podstatně jednodušší možnost – využít některý z obvodů FTDI. Touto problematikou se bude zabývat podrobně knížka [21]. Proto si ukážeme jen dva jednoduché programy napsané v CodeVisionAVR C – jeden pro vysílání dat z MCU AVR na sběrnici USB (a následný příjem těchto dat počítačem PC) a další program, provádějící výstup znaků přijatých z USB na LCD displej. V obou případech použijeme AT90S8515 komunikující s obvodem FT8U245AM (cena asi 200 Kč), což je konvertor USB/FIFO, komunikující s přenosovou rychlostí až 1 Mbyte/s. Práci si ještě můžeme zjednodušit, když použijeme již hotovou destičku s tímto obvodem, krystalem, několika odpory a kondenzátory vyrobenou jako modul UMP-1.

Dále si uvedeme celkové zapojení, použité pro oba příklady (pro vysílání dat do USB ovšem LCD nepotřebujeme) *obr. 7.46*.

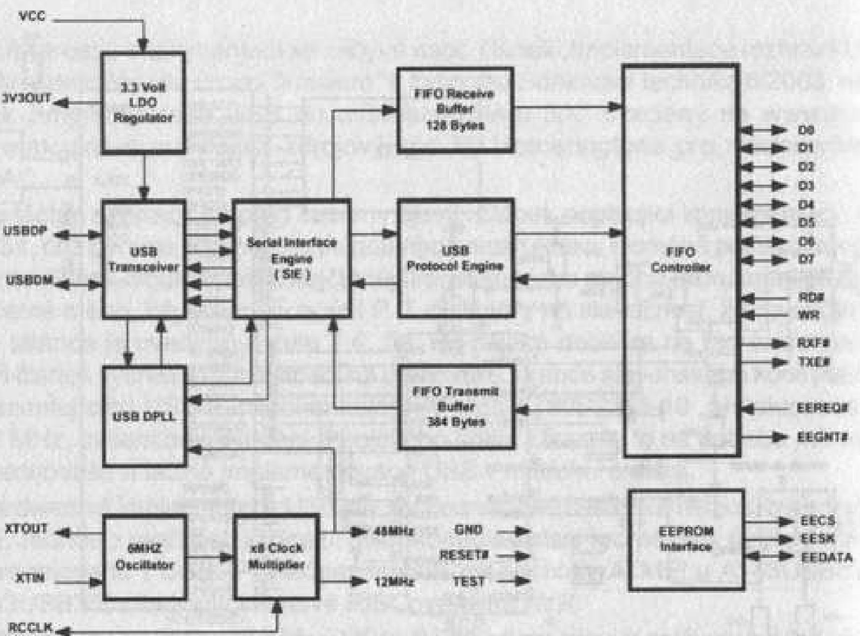


Obr. 7.46 Principiální připojení řadiče FTDI k mikrokontroléru – příklad č. 14

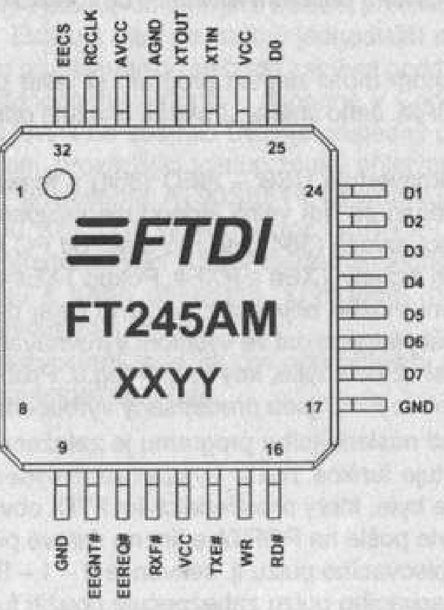
K tomu, abychom mohli napsat program si ještě potřebujeme popsat činnost obvodu FT8U245AM. Jeho vnitřní strukturu ukazuje *obr. 7.47* a zapojení jeho vývodů *obr. 7.48*.

Tento obvod konvertuje USB – FIFO (8bit) s maximální přenosovou rychlostí až 1 Mbyte/s. Přenos se řídí velmi jednoduše vstupními signály RD# a WR, stav vnitřní vyrovnávací paměti (384 byte pro směr od PC k aplikaci a 128 B pro směr k PC) je indikován signály TXE# a RXF#. Pokud TXE#=log.1, je vnitřní vyrovnávací paměť plná a není možné přijmout z periferie další data. RXF#=log.1 signalizuje aplikaci přítomnost platných dat ve výstupní vyrovnávací paměti, z které je koncové zařízení může číst až do chvíle, kdy RXF#=log.0. Průběhy řídicích a datových signálů při zápisu dat do FIFO jsou předepsány výrobcem *obr. 7.49*.

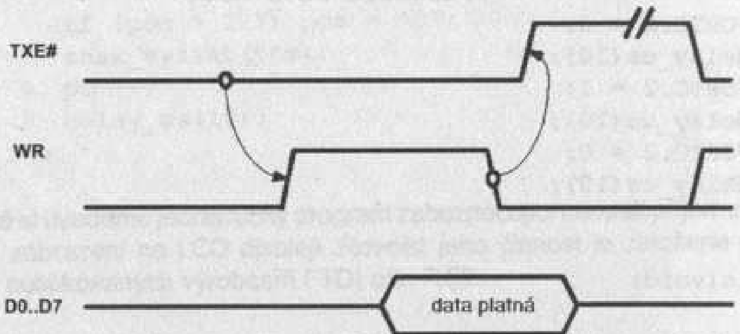
Základ činnosti následujícího programu je založena na těchto průbězích, které v něm implementuje funkce `znak_vyslat(unsigned char znak)`. Parametrem této funkce je byte, který prostřednictvím FTDI obvodu chceme poslat na USB sběrnici. Tento byte pošle na PORTA a tím na datové piny FTDI obvodu. Následuje vygenerování zapisovacího pulzu tj. sekvence 0 – 1 – 0 přes PORTC.2 na pin WR. Délku trvání zapisovacího pulzu zabezpečuje použití funkce `delay_us()`.



Obr. 7.47 *Vnitřní struktura řadiče FTDI*



Obr. 7.48 *Zapojení vývodů řadiče FTDI*



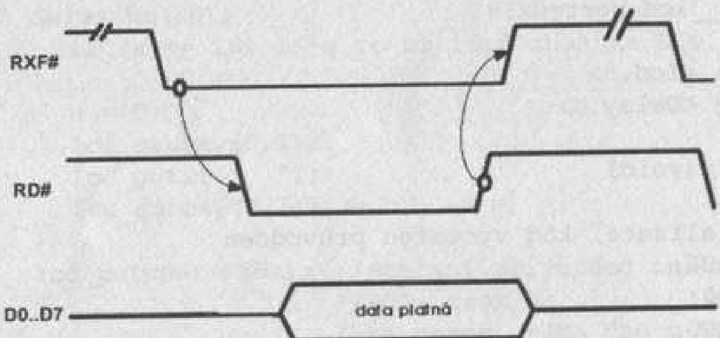
Obr. 7.49 *FT8U245AM časový diagram – FIFO zapisovací cyklus*

Vlastní činnost programu je jednoduchá. S pomocí funkce `znak_vyslat()` nejprve odešle posloupnost několika znaků (Ahoj) a poté v nekonečné smyčce vysílá znaky v pořadí daném ASCII abecedou. Výsledný kód je:

```
/*********************************************
Chip type           : AT90S8515
Clock frequency     : 8,000000 MHz
Memory model        : Small
Internal SRAM size  : 512
External SRAM size  : 0
Data Stack size     : 128
*********************************************/

#include <90s8515.h>
#include <Delay.h>

unsigned char pom;

void znak_vyslat(unsigned char znak)
{
        PORTA = znak;      //poslání vysílaného byte na PORTA tj.
                           //na DATA FTDI
        delay_us(10);      //generování 0 - 1 - 0 na WR tj.
```

```
                    //signálu nová data
    PORTC.2 = 0;
    delay_us(10);
    PORTC.2 = 1;
    delay_us(10);
    PORTC.2 = 0;
    delay_us(10);
}

void main(void)
{
// inicializace provedená průvodcem - wizardem
PORTA=0x00;
DDRA=0xFF;
PORTB=0x00;
DDRB=0x00;
PORTC=0x00;
DDRC=0x0C;
PORTD=0x00;
DDRD=0x00;
TCCR0=0x00;
TCNT0=0x00;
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
GIMSK=0x00;
MCUCR=0x00;
ACSR=0x80;

PORTC.2=0;
delay_ms(10);
znak_vyslat('A');
znak_vyslat('h');
znak_vyslat('o');
znak_vyslat('j');
znak_vyslat(' ');
pom = '0';
while (1)
   {
    if (pom > 127) pom = '0';
    znak_vyslat(pom);
    pom++;
    delay_ms(10);
   };
}
```

Ještě si uvedeme jednoduchý program zabezpečující naopak příjem znaků z USB a jejich zobrazení na LCD displeji. Rovněž jeho činnost je založena na průběhu signálů publikovaných výrobcem FTDI *obr. 7.50.*

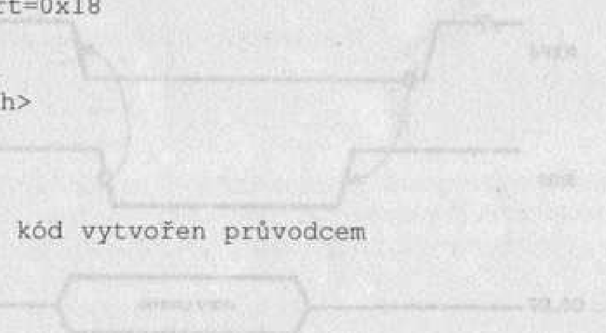


Obr. 7.50 FT8U245AM časový diagram – FIFO čtecí cyklus

Program běží v nekonečné smyčce a přitom kontroluje zda FTDI obvod na pinu RXF# má nulu, tj. signál *data připravena*. V případě, že platí *data připravena* odečte data z datových pinů obvodu FTDI, tato data (tj. jeden byte) a zobrazí je na LCD displeji. Přitom program posílá posloupnost 1 – 0 – 1 na RD#, která z FIFO obvodu FTDI vysouvá přijatá data.

Další činnost programu již jen určuje polohu na LCD, kde se přijatý znak má zobrazit. Výsledný kód je:

```
/**********************************************
Chip type           : AT90S8515
Clock frequency     : 8,000000 MHz
Memory model        : Small
Internal SRAM size  : 512
External SRAM size  : 0
Data Stack size     : 128
**********************************************/

#include <90s8515.h>

// Alphanumerký LCD Modul na portu B
#asm
  .equ __lcd_port=0x18
#endasm
#include <lcd.h>
#include <Delay.h>
int a;
void main(void)
{
// inicializace, kód vytvořen průvodcem
PORTA=0x00;
DDRA=0x00;
PORTB=0x00;
DDRB=0xFF;
PORTC=0x00;
DDRC=0x0C;
PORTD=0x00;
DDRD=0x00;
TCCR0=0x00;
TCNT0=0x00;
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
GIMSK=0x00;
MCUCR=0x00;
ACSR=0x80;
```



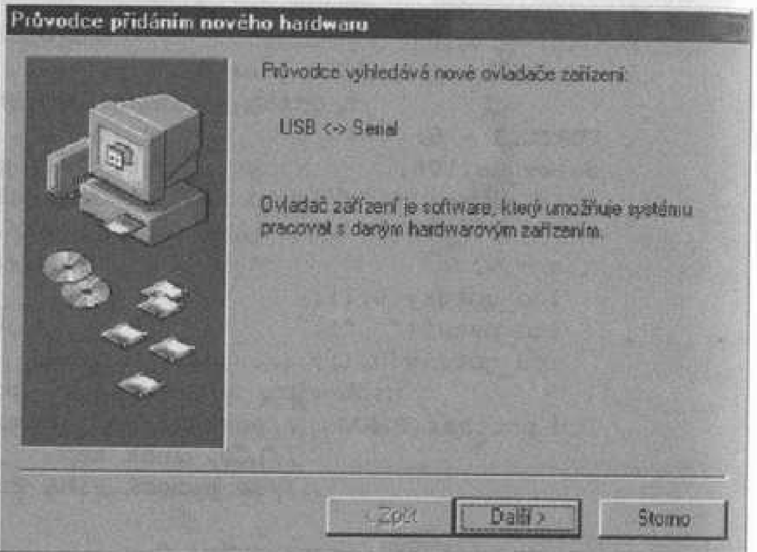

```
// inicializace LCD modulu, použit 2 x 16znaků
lcd_init(16);
lcd_gotoxy(0,0);
lcd_putsf("prijem usb");
lcd_gotoxy(0,1);
PORTC.3 = 1;
a = 0;
while (1)
    {
    PORTC.3 = 0;
    if (PINC.0 == 0) // t.j na RXF# je nula,
      {
      PORTC.3 = 0;
      delay_us(10);
        if (a == 16) //je to už 16tý znak na LCD
      {
        a = 0;
        lcd_gotoxy(0,1);
        lcd_putsf("  ");
        lcd_gotoxy(0,1);
      }
      lcd_putchar(PINA); //poslání přijatého znaku na
                         //LCD, znak se
                         //se pomocí PINA čte z PORTA

      delay_us(10); //generování 0 - 1 - 0 na RD#
      PORTC.3 = 1;
      delay_us(10);
      a = a + 1;

      }

    };
}
```

Ačkoli jsou oba programy funkční, jsou pouze **ilustrační**, tj. ukazují princip a snadnost komunikace USB pomocí obvodů FTDI. Pro skutečné aplikace bychom kód museli doplnit o ošetření stavů, jako je prázdná či plná FIFO, a z jednostranné komunikace vytvořit oboustrannou komunikaci obsahující např. potvrzení o přijmutí dat apod.

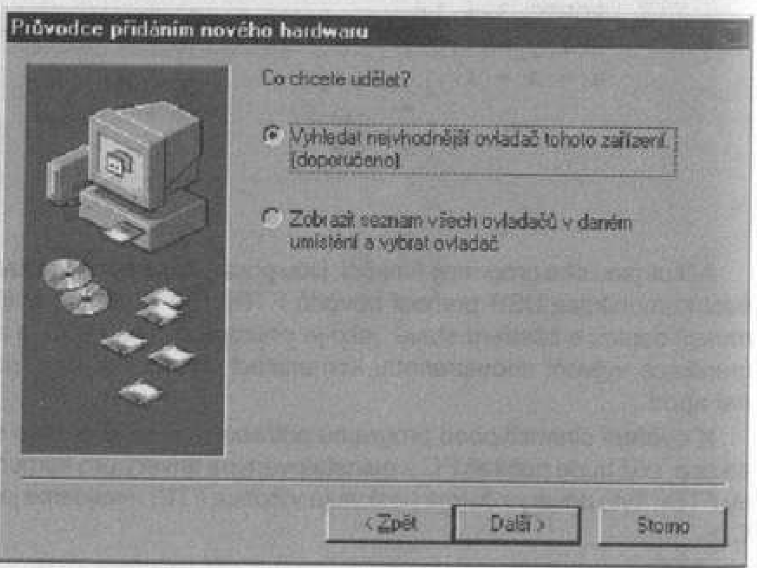
K ověření činnosti obou programů potřebujeme ještě druhou stranu USB komunikace, což bude počítač PC s nainstalovanými drivery pro komunikaci USB s obvody FTDI. Tyto drivery zdarma poskytuje výrobce FTDI a najdeme je i na doprovodném CD.

Pro OS Windows 98/ME/2K/XP jsou k dispozici dva druhy ovladačů – *přímé ovladače* (D2XX) a ovladače VCP (Virtual COM Port). Jednodušší je použití *ovladače VCP*. Nejdříve tento ovladač nainstalujeme. Budeme předpokládat, že máme WIN98. Drivery jsou umístěny v souboru **ftvcpw98p.zip**, takže ho nejdříve „rozzipujeme" (extrahujeme) a takto získané soubory umístíme např. do adresáře FTDI.

Pomocí kabelu a USB konektorů připojíme FTDI obvod k PC. Objeví se následující obrazovka *obr. 7.51* a stikneme `Další`. Poté se objeví *obr. 7.52*.

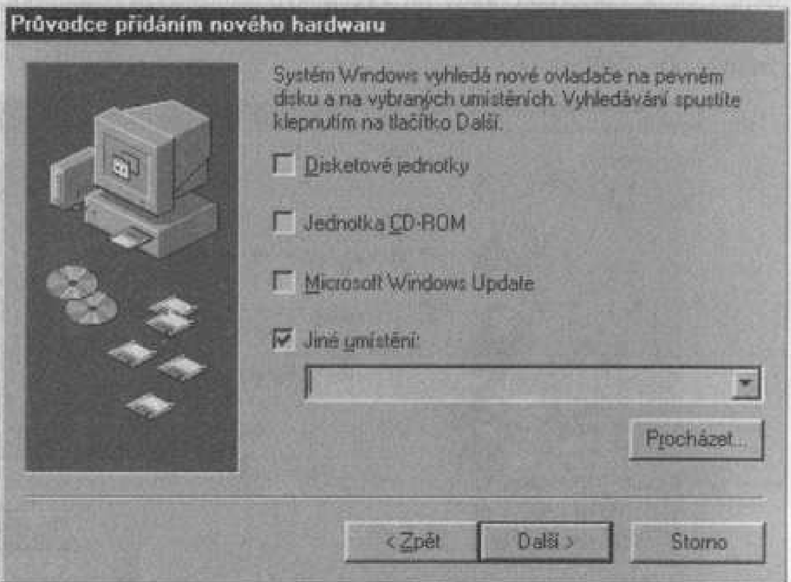


Obr. 7.51

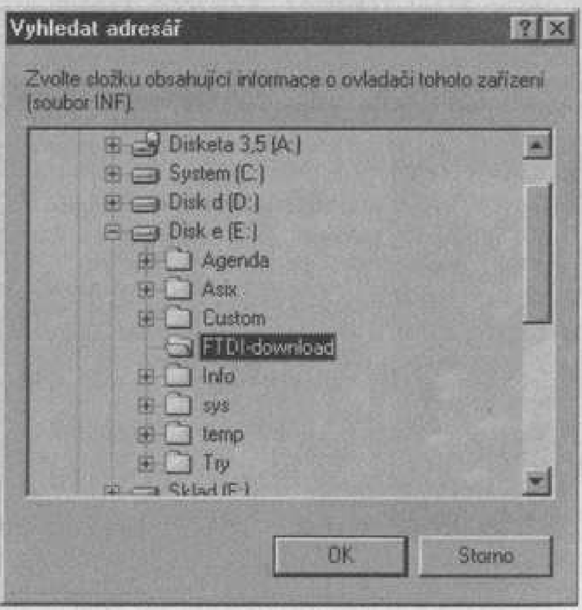


Obr. 7.52

Ponecháme doporučenou volbu a stiskneme `Další`. Objeví se *obr. 7.53*. Opět stiskneme `Další` a následuje dialogové okno, obdobné jako *obr. 7.54*.



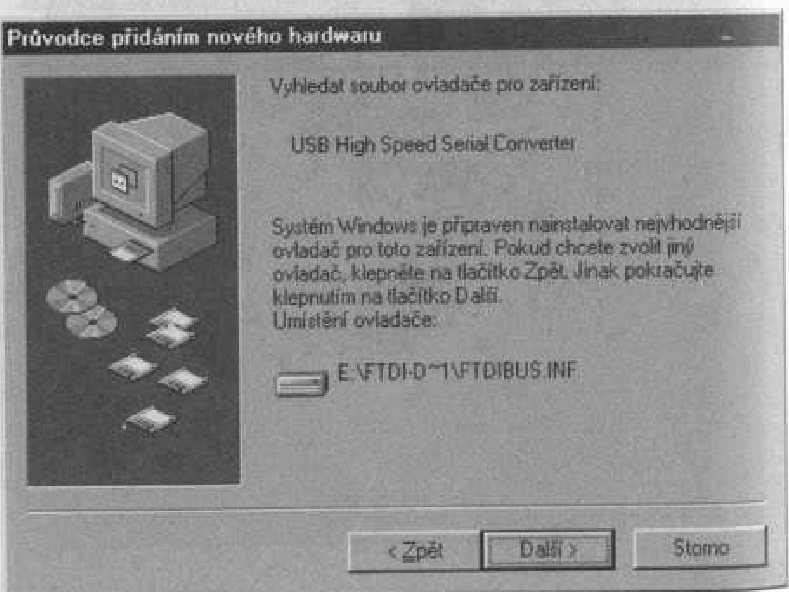
Obr. 7.53



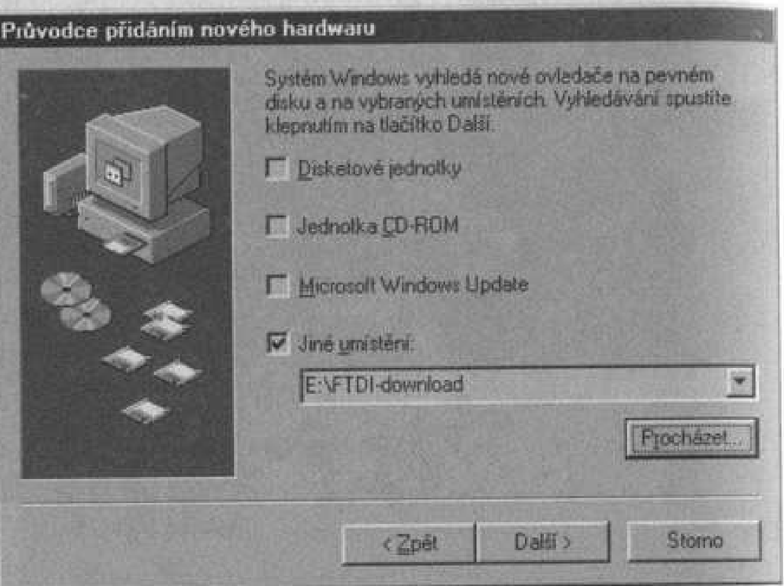
Obr. 7.54

Vybereme adresář, kde máme připraveny drivery a potvrdíme OK. Objeví se další obrazovka *obr. 7.55*.

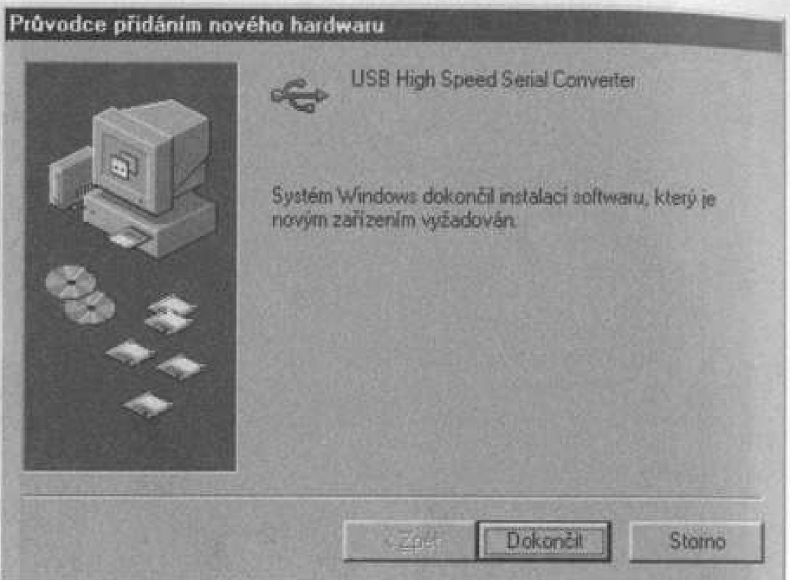
Stiskneme další a následuje *obr. 7.56* a po stisku `Další` se objeví obrazovka *obr. 7.57* a stiskneme `Dokončit`. Nyní se objeví několik podobných dialogů, které se samy zavřou *obr. 7.58*.



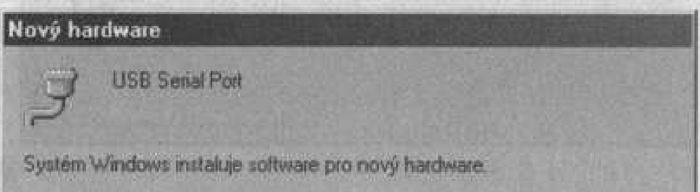
Obr. 7.55



Obr. 7.56

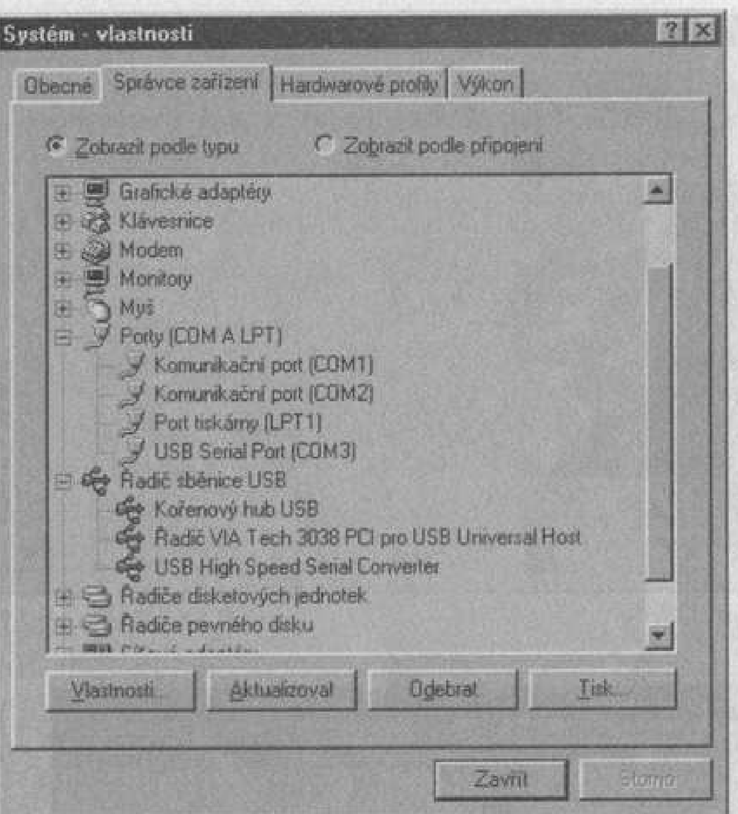


Obr. 7.57



Obr. 7.58

Tím je instalace dokončena. Připojený USB FTDI obvod se bude chovat jako další sériový COM port. Musíme ho ovšem ještě nastavit. Poté již aplikační program, např. Hyper Terminál, bude moci komunikovat po rozhraní RS232. Proto musí být nastaveny parametry číslo COM portu, přenosová rychlost, řízení toku dat, paritu a počet stopbitů. Tyto parametry se nastavují ve správci systému, který vyvoláte jednoduše současným stiskem klávesy WIN a PAUSE. Zvolíte kartu „Správce zařízení" a necháte ji zobrazit podle typu. Poté kliknutím na tlačítko + u řádku Porty rozbalíte submenu Porty, kde je na konci uveden USB serial port *obr. 7.59*.

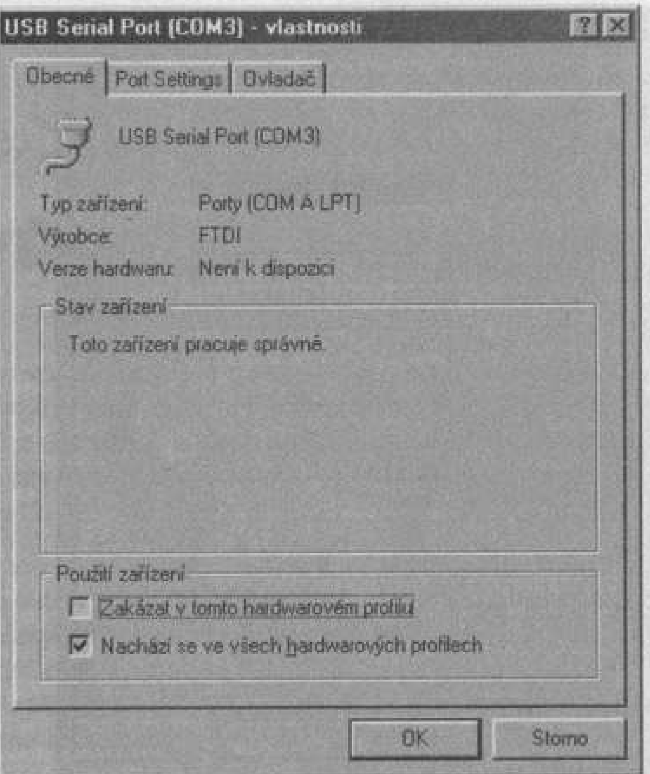


Obr. 7.59

Vyberte myší nebo klávesnicí tento řádek (USB serial port) tak, aby byl označen (např. modře) a stiskněte tlačítko VLASTNOSTI.

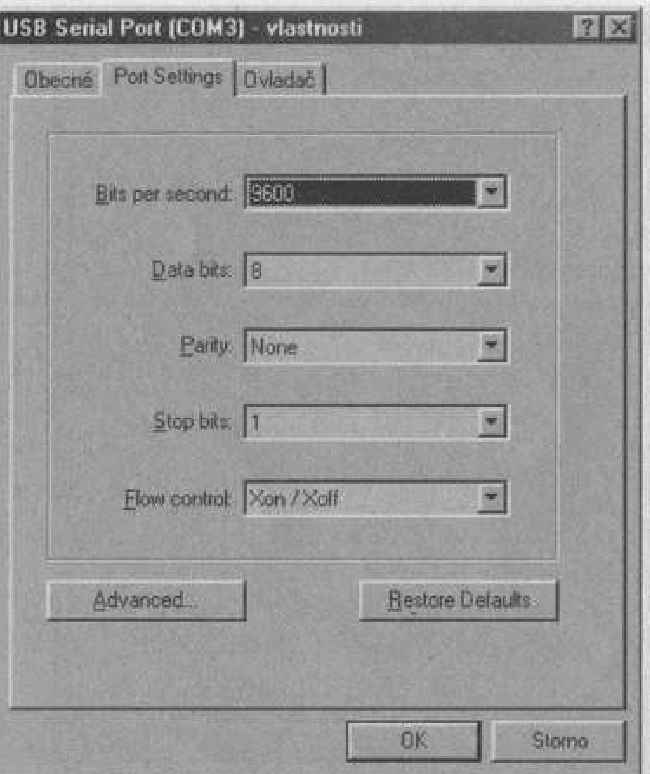
Objeví se *obr. 7.60*. Zvolíme záložku Port Settings a v okně *obr. 7.61*, nastavíme parametry komunikace a potvrdíme OK. Následuje *obr. 7.62*.

Vybereme číslo portu COM a potvrdíme OK.

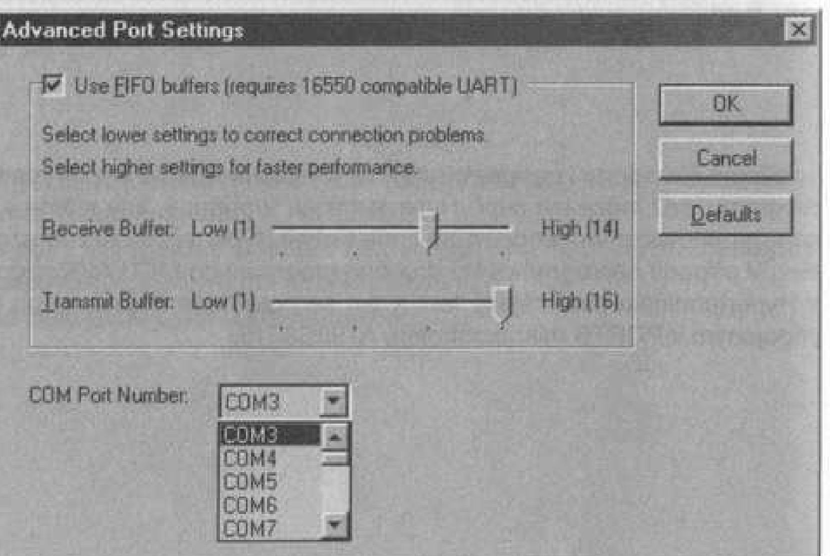


Obr. 7.60

Stejné číslo portu i parametry sériové komunikace ještě nastavíme v aplikačním programu, což může být např. Hyperterminál. V případě, kdy máme v AT90S8515 naprogramován první program, se bude v okně Hyperterminálu zobrazovat vysílaný text. V případě naprogramování druhého programu do MCU AVR naopak můžeme v Hyperterminálu psát nějaký text a ten se poté bude objevovat na displeji LCD připojeným k PORTB mikrokontroléru AT90S8515.



Obr. 7.61



Obr. 7.62

8

ZÁVĚREČNÁ POZNÁMKA

Při přípravě této publikace byl největším problémem rozsah. Původně jsem do ní chtěl zahrnout programování v IAR Embedded Workbench, Imagecraft C, CodeVisionAVR C a GNU GCC a navíc uvést velké množství příkladů a řady praktických aplikací založených na mikrokontrolérech ATMEL AVR řízených programy napsanými v jazyce C.

Bylo by to však velmi náročné nejen na celkový rozsah knihy, ale i náročné časově takové dílo vůbec vytvořit. Naštěstí nakladatelství BEN – technická literatura vydalo překlad knížky [12], věnované programování mikrokontrolérů v jazyce C obecně a v praktické části využívající IAR C. To mi umožnilo koncipovat mou knížku jako knížku pro začátečníky s tím, že **pokročilejší čtenář může další informace nalézt v zmiňované knížce**

„C pro mikrokontroléry".

Doporučuji především kapitoly *Tipy a triky v jazyce C* a ***Dobrý programovací styl v jazyce C***. Možná, že v mé knížce v příkladech najdete něco, co by se dalo napsat lépe. Snažil jsem se začátečníky příliš neodradit zhuštěným stylem psaní typickým pro programy v C vytvořené „opravdovými programátory". Aby programy byly co nejjednodušší a v rozsáhlém kódu se neztratila podstata řešení, neosahují např. ošetření chybových stavů apod.

Budoucnost patří sběrnici USB. Proto jsem na závěr zařadil příklad s obvody FTDI, které jsou již dostupné v tuzemské maloobchodní síti. I já se těším, že co nevidět vyjde knížka pana Matouška „USB prakticky s obvody FTDI", kterou v době uzávěrky této mé knihy nakladatelstství BEN – technická literatura rovněž zpracovávalo.

Internet je skutečně bezedný zdroj informací, jistě nejen pro mne. Kódy skutečných aplikací posbíraných na Internetu, jsem na doprovodném CD umístil do adre-

sáře ***NÁPADY***. Mohou sloužit i jako inspirace zejména pro amatérské konstruktéry, i když jsou některé již staršího data. Pro úplnost zde uvádím alespoň jejich přehled s odkazem na původní zdroj.

- **hodiny řízené DCF77**,
  Jaromír Čechák,
  http://www.stud.feec.vutbr.cz/~xcecha01
  http://www.stud.feec.vutbr.cz/~xcecha01/made/dcf/dcf.htm
- **generátor DTMF**,
- **čítač**,
  Jesper Hansen,
  http://www.myplace.nu/avr/countermeasures/index.htm
  http://www.myplace.nu/avr/index.htm
- **připojení pevného disku HDD** k mikrokontroléru AVR,
- **osciloskop** využívající A/D v AT90S8535 propojeným přes RS232 s PC, na němž běží obslužný program napsaný v Delphi,
  Alexandr Jelisejev,
  http://www.telesys.ru/projects/proj060/index.shtml
  (http://radiotech.by.ru)
- **čtečka telefonní karty** s AT90S8535 od německého konstruktéra (profesionální čtečka čipových karet ChipDriveMicro od německé firmy Towitoko umí přečíst nové telefonní karty Trick českého Telekomu a tvrdí, že používá protokol SLE4436, což je protokol telefonních karet od Siemens),
- **generátor zvuků**.
  http://www.mikrocontroller.com
  http://mikrocontroller.cco-ev.de/en/download.php#M16C

Pozn. red.:

Etika nám velí, abychom se ještě před vydáním CD spojili s autory těchto „nápadů", zda můžeme jejich článek zveřejnit na našem CD. Pokud by některý z autorů nebo zveřejnitelů (serverů) s tímto nesouhlasil, je možné, že se v konečné verzi CD tento „nápad" neobjeví.

Je to proto, že uznáváme právo autorů rozhodovat o tom, kde jejich dílo bude zveřejněno. Oni jediní mají právo s tímto jejich duševním vlastnictvím nakládat, byť je zveřejnili volně někde na Internetu. Dosud však většina z nich souhlasila.

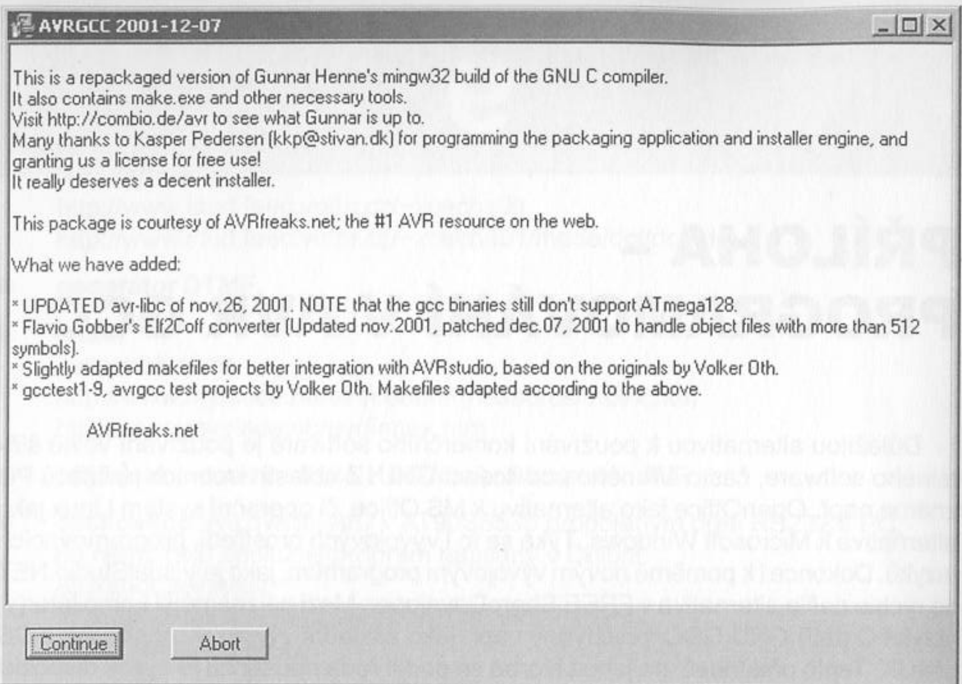
9

PŘÍLOHA – PROGRAMOVÁNÍ V AVR GCC

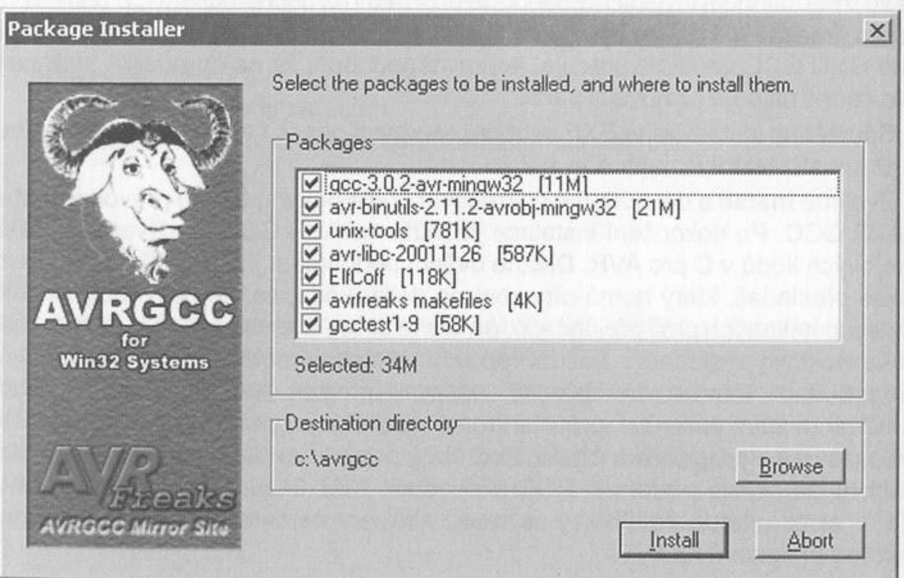
Důležitou alternativou k používání komerčního software je používání volně šiřitelného software, často šířeného pod licencí GNU. Z oblasti osobních počítačů PC známe např. OpenOffice jako alternativu k MS Office, či operační systém Linux jako alternativa k Microsoft Windows. Týká se to i vývojových prostředí, programovacích jazyků. Dokonce i k poměrně novým vývojovým programům, jako je VisualStudio.NET se rychle našla alternativa v FREE SharpDeveloper. Mezi nejznámější kompilátor(y) jazyka C patří GNU GCC, používaný např. jako základní vývojový prostředek v OS LINUX. Tento překladač, na jehož tvorbě se podílí řada nadšenců je nyní k dispozici pro různé platformy, rozličné procesory včetně mikrokontrolérů ATMEL AVR. Domovskou stránkou tohoto překladače pro ATMEL AVR je [4]. Z této stránky si můžeme zdarma stáhnout instalační soubor. Je obsažen i na doprovodném CD pod názvem avrgcc_freaks200112.exe délky cca 8,4 Mbyte. Protože skupina nadšenců na projektu GNU GCC neustále pracuje, je pravděpodobné, že na domovské stránce tohoto céčka najdete již novější verzi.

Spuštěním instalačního EXE souboru se spustí *obr. 9.1* a po potvrzení **Continue** se již spustí vlastní instalér *obr. 9.2*.

Zvolíme **Install** a dál budeme pokračovat podle pokynů tohoto průvodce instalací AVR GCC. Po dokončení instalace již můžeme AVR GCC používat pro překlad zdrojových kódů v C pro AVR. Dlužno ovšem podotknout, že jsme tím sice zdarma získali překladač, který nemá omezení na délku kódu, což je nevýhoda školních, edu verzí některých profi překladačů, na druhé straně lze občas slyšet námitky týkající se efektivity výsledného kódu. Především ale nemáme žádné integrované vývojové prostředí. Tomu se dá *částečně* odpomoci integrací AVR GCC do AVR Studia. Záměrně uvádím *částečně*, protože kromě vlastního jazyka C se musíme ještě naučit pracovat s programem Make. Podrobný popis pravidel sestavení GNU Make souborů lze nalézt v adresáři DOC překladače AVR GCC. Tento přístup je použit v [13]. Je to přístup používaný „opravdovými programátory" odchovanými prací v režimu příkazové řádky.



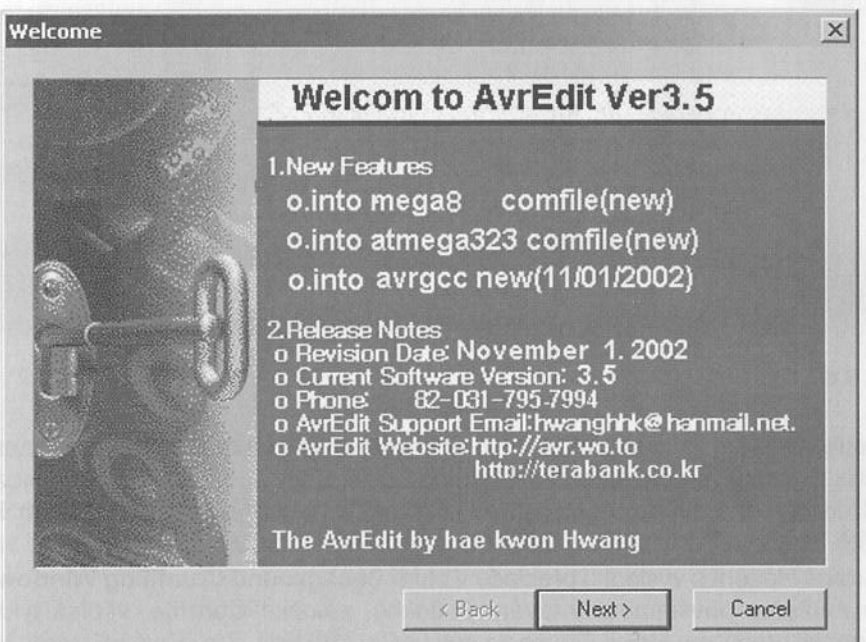
Obr. 9.1



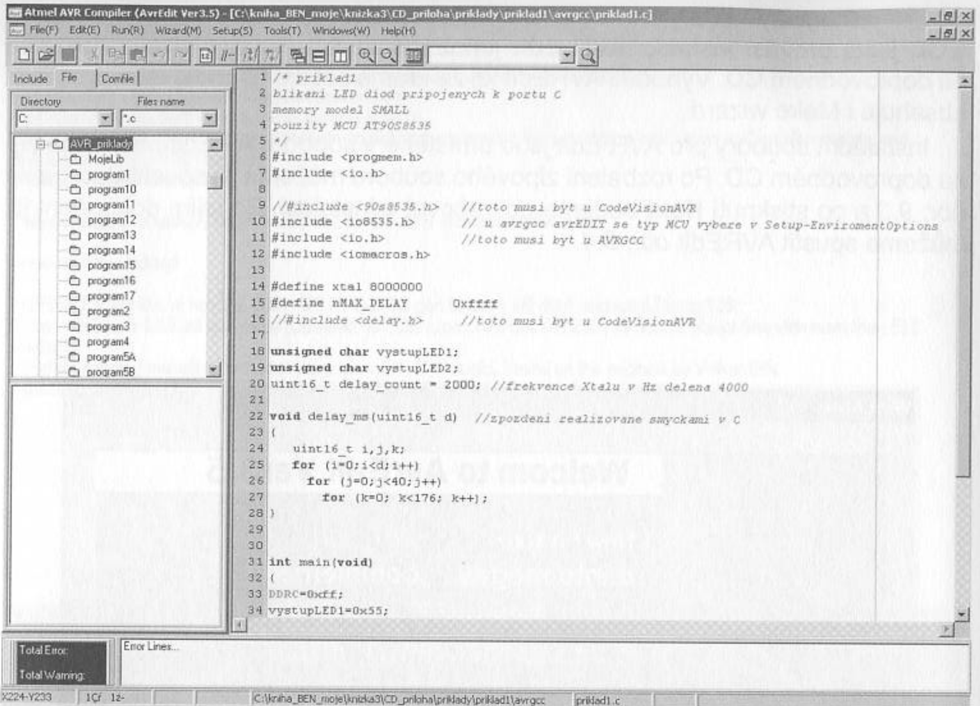
Obr. 9.2

Pro pohodlnější či začínající programátory bude výhodnější, po instalaci AVR GCC ještě provést instalaci **AVR Edit** jehož instalační soubory rovněž najdeme na doprovodném CD. Výhodou AVREdit je, že kromě solidního grafického prostředí obsahuje i Make wizard.

Instalační soubory pro AVR Edit jsou umístěné v souboru AvrEdit3.5English.zip na doprovodném CD. Po rozbalení zipového souboru můžeme již spustit Setup.exe *obr. 9.3* a po stisknutí **Next** budeme pokračovat v instalaci. Po jejím dokončení již můžeme spustit AVREdit *obr. 9.4*.



Obr. 9.3

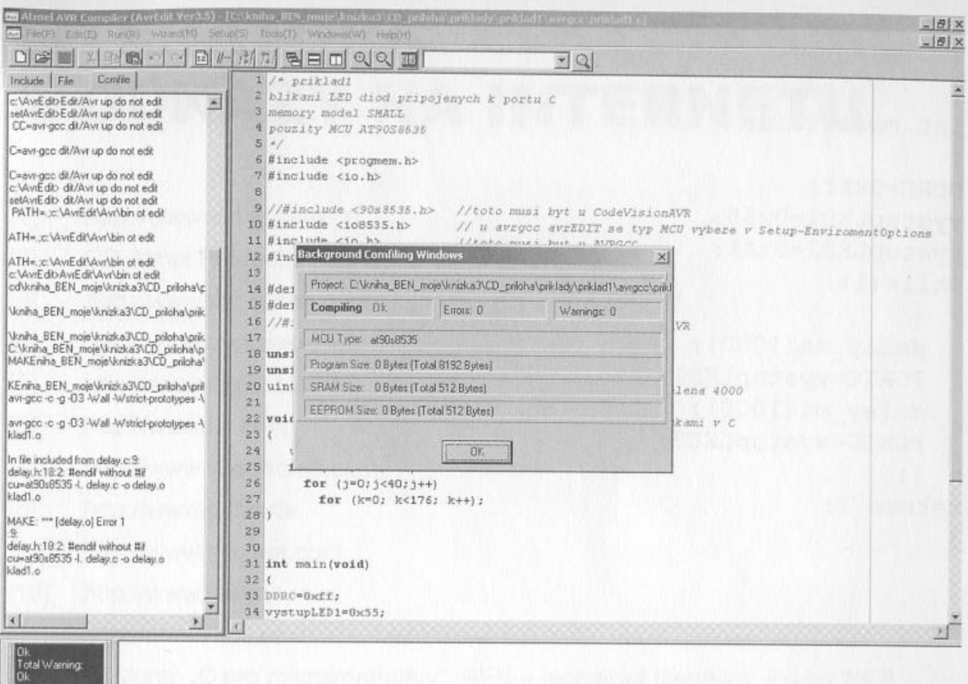


Obr. 9.4

Jeho ovládání je zcela intuitivní a odpovídá zvyklostem ovládání windowsovských programů. V editačním okně napíšeme zdrojový kód v jazyce C. Samozřejmostí je barevné zvýraznění syntaxe editorem. Překlad spustíme tlačítkem **Run** na liště nástrojů. Po úspěšném překladu dostaneme *obr. 9.5*.

Kromě hlášení o výsledku překladu v okně **Background Comfiling Windows** si ještě můžeme povšimnout v levém podokně, záložka **Comfile**, výpisů týkající se činnosti programu Make. To vše za nás udělal AVREdit. Pro srovnání uvedu ještě zdrojový kód z Programu1 v kap. 7 upravený pro AVR GCC:

```
/* priklad1
blikani LED diod pripojenych k portu C
memory model SMALL
pouzity MCU AT90S8535
*/
#include <progmem.h>
#include <io.h>

//#include <90s8535.h> //toto musi byt u CodeVisionAVR
```

Obr. 9.5

```
#include <io8535.h>          //u avrgcc avrEDIT se typ MCU vybere
                              //v Setup-EnviromentOptions
#include <io.h> //toto musi byt u AVRGCC
#include <iomacros.h>

#define xtal 8000000
#define nMAX_DELAY 0xffff
//#include <delay.h> toto musi byt u CodeVisionAVR

unsigned char vystupLED1;
unsigned char vystupLED2;
uint16_t delay_count = 2000; //kmitočet Xtalu v Hz delena 4000

void delay_ms(uint16_t d) //zpozdeni realizovane smyckami v C
{
  uint16_t i,j,k;
  for (i=0;i<d;i++)
    for (j=0;j<40;j++)
```

```
    for (k=0; k<176; k++);
}

int main(void)
{
DDRC=0xff;
vystupLED1=0x55;
vystupLED2=0xAA;
while(1)
  {
  delay_ms(1000);
  PORTC=vystupLED1;
  delay_ms(1000);
  PORTC=vystupLED2;
  };
return 1;
}
```


LITERATURA A ODKAZY NA INTERNETU

[1] http://www.atmel.com

[2] CD Atmel Products, May 2002

[3] http://www.avr-forum.com

[4] http://www.avrfreaks.net

[5] http://www.hpinfotech.ro

[6] http://www.iar.com

[7] http://www.imagecraft.com

[8] http://www.e-lab.de

[9] http://www.mjolner.com

[10] http://www.hw.cz

[11] http://www.mcu.cz

[12] B. Mann: „C pro mikrokontroléry", BEN – technická literatura, Praha 2003

[13] V. Šubrt: „Mikrokontroléry ATMEL AVR – vývojové prostředí", BEN – technická literatura, Praha 2002

[14] V. Váňa: „Mikrokontroléry ATMEL AVR – popis procesoru a instrukční soubor", BEN – technická literatura, Praha 2003

[15] V. Váňa: „Mikrokontroléry ATMEL AVR – Assembler", BEN – technická literatura, Praha 2003

[16] D. Matoušek: „Práce s mikrokontroléry ATMEL AVR", BEN – technická literatura, Praha 2003

[17] D. Matoušek: „Udělejte si z PC ...", 2. díl, BEN – technická literatura, Praha 2002

[18] D. Matoušek: „Práce s mikrokontroléry ATMEL AT89S8252", BEN – technická literatura, Praha 2002

[19] B. Kainka: „USB – Měření, řízení a regulace pomocí sběrnice USB", BEN – technická literatura, Praha 2002

[20] D. Matoušek: Práce s mikrokontroléry ATMEL AT89C2051, BEN – technická literatura, Praha 2002

[21] D. Matoušek: „USB prakticky s řadiči FTDI", BEN – technická literatura, Praha 2003

[22] Z. Rozehnal: Mikrokontroléry Motorola HC11, BEN – technická literatura, Praha 2001