Ústav radioelektroniky
Vysoké učení technické v Brně

Programování mikrokontrolérů pomocí vyšších jazyků

Mikroprocesorová technika a embedded systémy

Přednáška 5

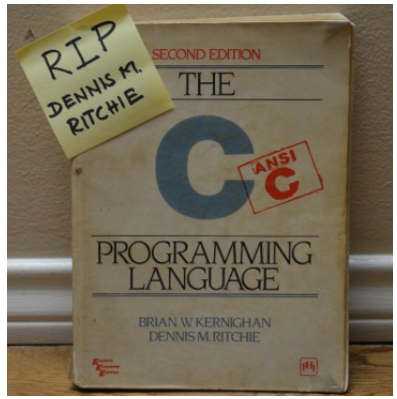
doc. Ing. Tomáš Frýza, Ph.D.

říjen 2012

Obsah přednášky

Obsah přednášky

Jazyk C



Obrázek: Dennis MacAlistair Ritchie (1941–2011); D.M.Ritchie, Brian Kernighan. *The C Programming Language.* 1978.

C89 Programovací jazyk C byl poprvé standardizován americkou společností ANSI (American National Standards Institute) v roce 1989 – ANSI C – označení C89.

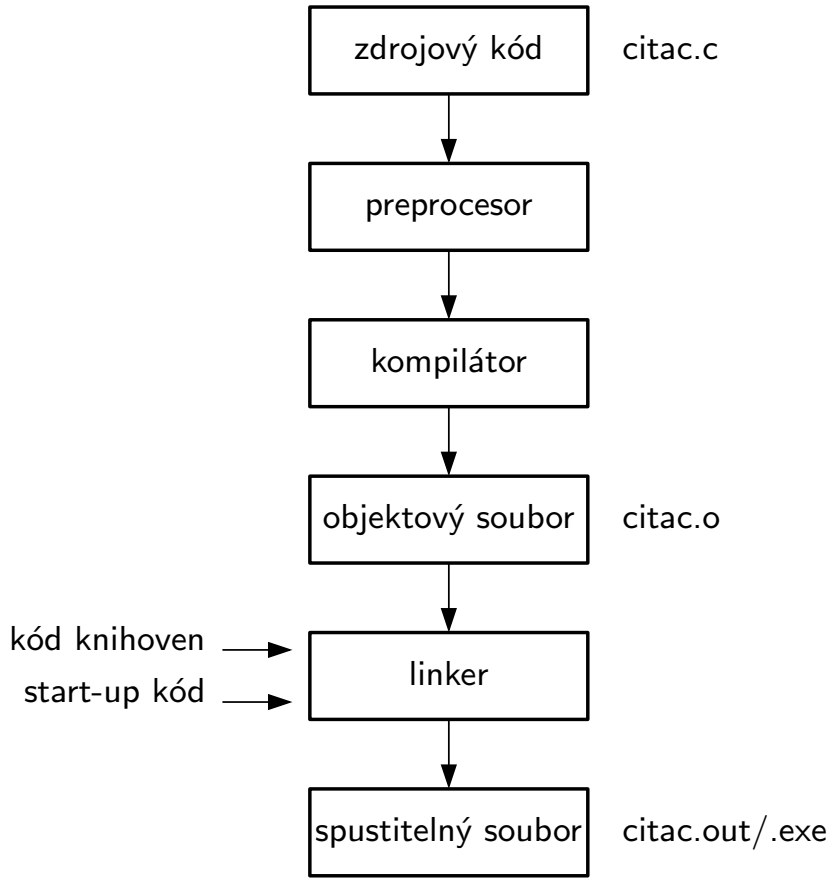
- ANSI C definuje strukturu/syntaxi jazyka i standardní knihovny.

C90 Standard byl v roce 1990 převzat také mezinárodní společností ISO (International Organization for Standardization) – označení C90.

C99 V roce 1994 započala práce na revizi standardu, která vyústila ve standard C99.

- V současné době (říjen 2012) je platná verze s označením ISO/IEC 9899:2011; poslední korekce ISO/IEC 9899:2011/Cor 1:2012 z 2012-07-13.

Kompilace a linkování



Obrázek: Postup překladu aplikace v jazyce C.

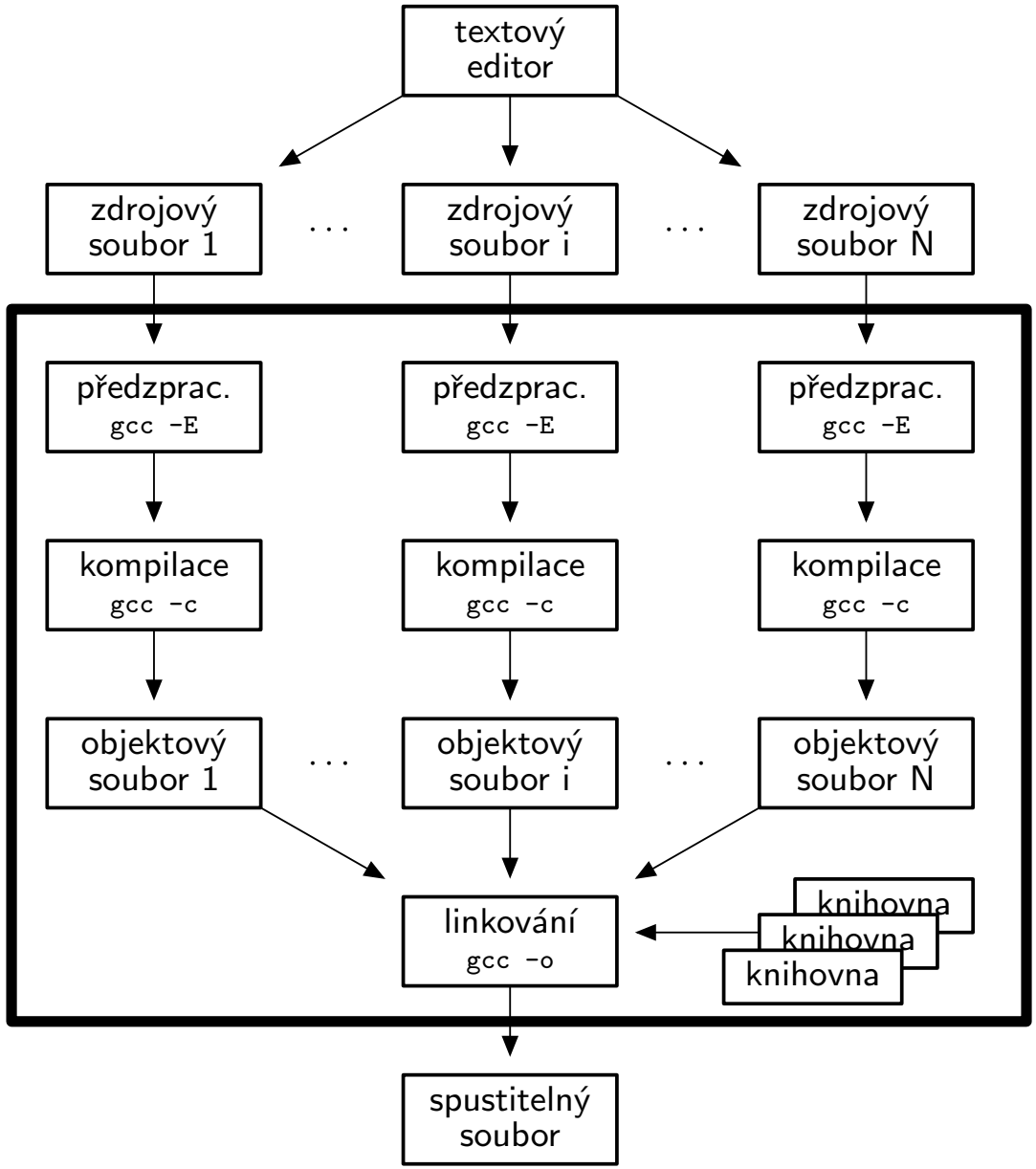
- Podstatou programování v jazyce C je využití programů ke konverzi zdrojových souborů na spustitelné, které obsahují strojový kód aplikace pro cílový procesor.
- Implementace se skládá ze dvou kroků: kompilace, linkování.

Kompilátor konvertuje zdrojový kód do přechodného kódu (objektový soubor).

Linker kombinuje všechny potřebné objektové soubory a vytváří spustitelný kód. – není tak nutné při tvorbě kódu vždy kompilovat všechny soubory.

- Objektový soubor obsahuje přeložený zdrojový kód, není to však kompletní program.
- Kromě přeložených částí použitých knihoven (pouze ty rutiny, které kód využívá) linker vkládá do výsledného programu také *start-up* kód – funguje jako mezičlánek mezi programem a použitým operačním systémem. (Stejný objektový soubor na stejném hardwaru (např. PC) nejde spustit stejně pod Linuxe i Windows.)

Kompilace a linkování

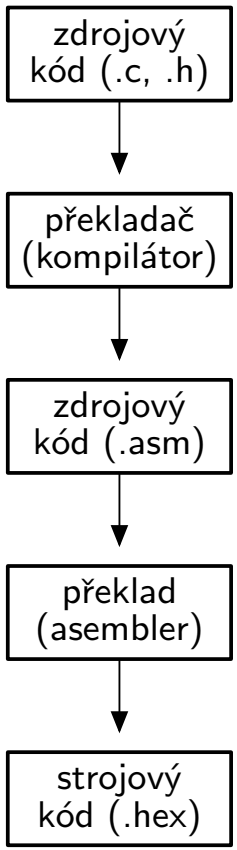


Obrázek: Postup překladu aplikace s více zdroji v jazyce C.

Softwarové prostředky

- Windows – obecně:
  - Microsoft Visual Studio
    `http://www.microsoft.com/cze/msdn/vstudio/`,
  - C++ Builder
    `http://www.embarcadero.com/products/application-development`,
  - Dev-C++; využívá volný překladač GCC – GNU Compiler Collection
    `http://www.bloodshed.net/devcpp.html`,
  - ...
- Windows – AVR:
  - AVR Studio + WinAVR
    `www.atmel.com/microsite/avr_studio_5/` + `http://winavr.sourceforge.net/`,
  - CodeVisionAVR
    `http://www.hpinfotech.ro/html/cvavr.htm`,
  - IAR Embedded Workbench
    `http://www.iar.com/`,
  - Keil
    `http://www.keil.com/`,
  - ...
- Linux:
  - gcc-avr, binutils-avr, avr-libc, avrdude, ...
  - Eclipse (IDE), eclipse-cdt (C/C++ plugin)
    `http://www.eclipse.org/cdt/`,
  - ...

Proces překladu do strojového jazyka u AVR



Obrázek: Zjednodušený překlad zdrojového kódu z jazyka C.

```
1                    LDI   R28, 0x5F     // (!)
2                    LDI   R29, 0x04     // (!)
3                    OUT   0x3E, R29     // (!)
4    ...             OUT   0x3D, R28     // (!)
5    DDRB = 0xFF ;   SER   R24                    SER   R24
6                    OUT   0x17, R24             OUT   0x17, R24
7    temp = 0x03 ;   LDI   R24, 0x03             LDI   R24, 0x03
8                    STS   0x0060, R24   // (!)
9    PORTB = temp ;  OUT   0x18, R24             OUT   0x18, R24
10   while( 1 )      IN    R24, 0x18     // (!)
11       PORTB--- ;  SUBI  R24, 0x01             DEC   R24
12                   RJMP  PC-0x0003             RJMP  PC-0x0002
```

Pozn.: `STS k, Rr` – přímé uložení hodnoty registru do datové paměti SRAM ($0 \leq r \leq 31$, $0 \leq k \leq 65\,535$). Store Direct to Data Space.

Proces překladu do strojového jazyka u AVR – pokračování

```
1  ;    Adresa      Stroj. kod     Instrukce              Popis funkce
2  ;  ----------------------------------------------------------------------
3       0x0047         E5CF        LDI   R28, 0x5F     ; load immediate
4       0x0048         E0D4        LDI   R29, 0x04     ; load immediate
5       0x0049         BFDE        OUT   0x3E, R29     ; out to I/O location
6       0x004A         BFCD        OUT   0x3D, R28     ; out to I/O location
7       0x004B         EF8F        SER   R24           ; set register
8       0X004C         BB87        OUT   0x17, R24     ; out to I/O location
9       0x004D         E083        LDI   R24, 0x03     ; load immediate
10      0x004E         93800060    STS   0x0060, R24   ; store direct to data space
11      0x0050         BB88        OUT   0x18, R24     ; out to I/O location
12      0x0051         B388        IN    R24, 0x18     ; in from I/O location
13      0x0052         5081        SUBI  R24, 0x01     ; subtract immediate
14      0x0053         CFFC        RJMP  PC-0x0003     ; relative jump
```

Tabulka: Zápis ve formátu Intel HEX.

```
...
:10009000D4E0DEBFCDBF8FEF87BB83E080936000ED
:0800A00088BB88B38150FCCF3E
```

Formát Intel HEX

Tabulka: Zápis strojového kódu ve formátu Intel HEX.

```
...
:10   0090   00   D4E0 DEBF CDBF 8FEF 87BB 83E0 8093 6000   ED
:08   00A0   00   88BB 88B3 8150 FCCF   3E
```

- Význam jednotlivých bytů ve formátu Intel HEX:
  - **:** Symbol identifikující začátek řádku.
  - **10** Počet datových bytů v jednom řádku v šestnáctkové soustavě, tj. 0x10 = 16.
  - **0090** Adresa ve Flash pro uložení prvního datového bytu; pozor, indexováno po bytech, tj. $0x48 \cdot 2 = 0x90$.
  - **00** Typ kódu; hodnota 00 až 05. U AVR odpovídá 00 identifikátoru 64 kB paměť. str.
  - **...** Strojový kód instrukcí v pořadí little-endian (LSB byte uložen dříve).
  - **ED** Kontrolní součet jednoho řádku bez symbolu ":". Jednotlivé byty se sečtou; z výsledku se použije jen LSB a k němu se dopočítá dvojkový doplněk.

Kontrola: $08 + 00 + A0 + 00 + 88 + BB + 88 + B3 + 81 + 50 + FC + CF = \$5C2 \Rightarrow C2 \Rightarrow 3E$.

Pozn.: Dvojkový doplněk, viz reprezentace záporných hodnot v mikroprocesorové technice.

Otázka

Jak souvisí velikost souboru .hex s velikostí přeložené aplikace?

Elementární zásady jazyka C

- Příkaz končí středníkem.
- Těla funkcí, posloupnosti příkazů u podmínek, cyklů, apod. se sdružují do složených závorek.
- Řetězce se uvozují uvozovkami.
- Komentáře se píší do bloku uvozeném `/* ... */`. Většina překladačů podporují též řádkové komentáře `//`.
- Rezervovaná slova nelze použít k jinému účelu než je určeno: `for`, `return`, `switch`, `case`, `if`, `else`, `char`, `int`, `float`, `unsigned`, `void`, ...
- Aplikace se skládá z funkcí. Funkce `main()` vždy.
- Deklarace funkce:

```
1    výstupní_hodnota název_funkce( vst_param_0, vst_param_1, ... ){
2        ...          // tělo funkce
3    }
```

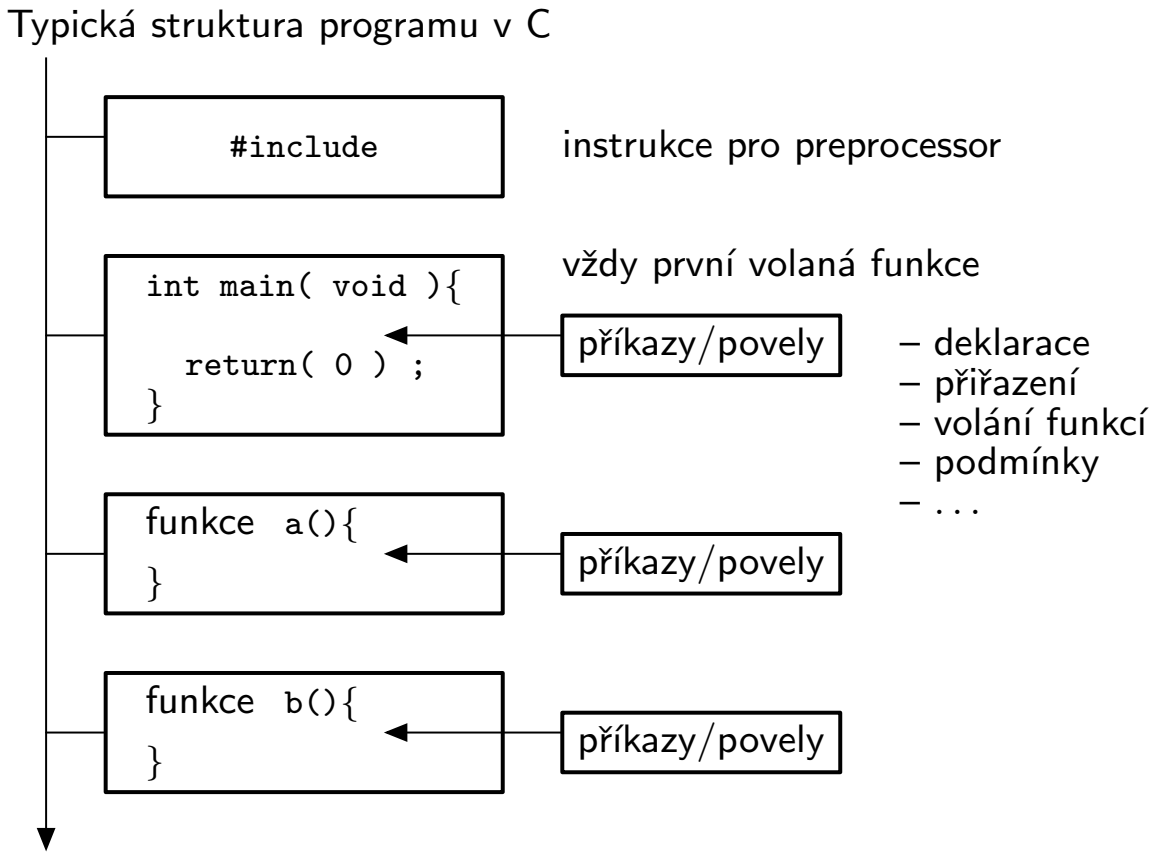
- Prototyp funkce deklaruje název funkce, typ výstupní hodnoty, počet vstupních operandů, typ jednotlivých vstupních operandů.

```
1    int imax( float*, int ) ;
```

Prototyp funkce

```
1  /* Hlavičkové soubory */
2  #include <avr\io.h>                  // hlavičkový soubor mikrokontroléru
3
4
5  /* Prototypy funkcí */
6  void setup( void ) ;                 // prototyp funkce setup
7  int key_pressed( void ) ;            // prototyp funkce key_pressed
8
9
10 /* Globální proměnné */
11 char temp = 0 ;                      // deklarace 8bitové proměnné temp
12
13
14 /* Hlavní funkce */
15 int main( void ){
16     setup() ;                        // volání funkce setup
17     for( ;; ) {                      // nekonečná smyčka
18         if( key_pressed() ){
19         ...                          // tělo podmínky
20         }
21     }
22     return( 1 ) ;                    // výstupní hodnota funkce = 1
23 }
24
25
26 /* Funkce pro nastavení kontrolních registrů */
27 void setup( void ){                  // nastavení I/O portů
28     ...                              // tělo funkce
29 }
30
31
32 /* Funkce pro zjištění stisknutého tlačítka */
33 int key_pressed( void ) {            // vrací kód stisknutého tlačítka
34     char key ;                       // deklarace lokální proměnné
35     ...                              // tělo funkce
```

Struktura zdrojového kódu v jazyce C



Obrázek: Anatomie programu v jazyce C.

Tvorba "čitelného" kódu – dobrá programátorská praxe

- Používání smysluplných názvů proměnných/funkcí.
- Psaní výstižných komentářů. Používání prázdných řádků pro oddělení deklarační a výkonné části funkce, jeden řádek=jeden příkaz.
- C je *free-form* formát.

```
1  int main( void ) { int rank; rank
2  =
3  1
4  ;printf(
5      "BMPT is #%d for me\n",
6  rank);return( 0 );}
```

```
1  int main( void ){                                    // hlavní funkce
2      int rank = 1 ;                                   // deklarace lokální proměnné
3
4      printf( "BMPT is #%d for me\n", rank ) ;         // tisk textu
5
6      return( 0 ) ;                                    // návrat z funkce
7  }
```

Pozn. Funkce `main` vrací hodnotu `int` a zpravidla nemá vstupní parametry, tj. `void`. Pokud má vstupní parametry, zápis je následující:

```
1  int main( void ){                        // hlavní funkce bez parametrů
2      ...
3  }
4
5  int main( int argc, char *argv[] ){      // hlavní funkce s parametry
6      ...                                  // argc — počet vstupních parametrů
7  }                                        // argv — pole vstupních parametrů
```

Deklarace proměnných

- Proměnná je identifikována názvem a typem:

```
1     char temp ;             // znaménkové číslo, 8 bitů
2     unsigned int ii,jj ;    // neznaménková/kladná čísla, 16 bitů
3     float f = 3.14 ;        // reálné číslo s plovoucí řádovou čárkou, 32 bitů
```

- Před použitím je nutné proměnnou deklarovat. Podle dostupnosti se jedná o:
  - globální – alokována při překladu mimo funkce, platí stále, dostupná pro všechny funkce,
  - lokální – alokována ve stacku (zásobník) nebo heap (hromada) při spuštění funkce; dostupná uvnitř funkce, po opuštění funkce ztrácí hodnotu (uvolní paměť), neplatí pro `static`.

```
1  int productSum( int a, int b ){
2      static int sum = 0 ;          // inicializace proměnné při prvním volání
3      int product ;                 // lokální proměnná
4
5      product = a * b ;
6      sum = sum + product ;
7
8      return( sum ) ;               // proměnná (i hodnota) zůstane zachována pro
9                                    // další volání
10 }
```

Typy proměnných v jazyce C

Tabulka: Vybrané typy a velikosti proměnných v jazyce C.

Typ proměnné	Velikost [b]	Číselný rozsah
`char`	8	-128 až 127
`unsigned char`	8	0 až 255
`signed char`	8	-128 až 127
`int`	16	$-32\,768$ až $32\,767$
`unsigned int`	16	0 až $65\,535$
`float`	32	$\pm 1,175 \cdot 10^{-38}$ až $\pm 3,402 \cdot 10^{38}$

Aritmetické a bitové operandy

Tabulka: Aritmetické operace v jazyce C.

Operace	Operand
Násobení	*
Dělení	/
Dělení modulo (celočíselné)	%
Sčítání	+
Odečítání	-
Inkrementace	++
Dekrementace	--

- Zkrácený zápis:
  - `a+=3 ;  // a=a+3`
  - `b-=2 ;  // b=b-2`
  - `c*=5 ;  // c=c*5`
  - `d/=a ;  // d=d/a`

Tabulka: Bitové operace v jazyce C.

Operace	Operand
Jednotkový doplněk	$\sim$
Bitový posuv doleva	$\ll$
Bitový posuv doprava	$\gg$
Logický součin AND	&
Exkluzivní součet EX-OR	$\wedge$
Logický součet OR	$\vert$

- Zkrácený zápis:
  - `a|=3 ;`
  - `b&=2 ;`
  - `c^=5 ;`
  - `d<<=2 ;`

Podmínky v jazyce C

- Syntaxe podmínky pomocí `if`:

```
1   if( a == N ){          // je-li a = N, vykonej následující kód
2       ;                  // podmíněná část kódu
3   }
4   else{                  // je-li a různé od N, vykonej následující kód
5       ;                  // podmíněná část kódu
6   }
7
8   ...
9
10  if( a == 10 && b <= 5 ){
11      ...                // vykonej pokud a = 10 a současně b < 6
12  }
13  else if( a > 15 || b > 15 ){
14      ...                // vykonej pokud a > 15 nebo b > 6
15  }
```

Tabulka: Relační operandy v jazyce C.

Operace	Operand
Rovno	==
Je různý od	!=
Menší než	<
Menší nebo rovno	<=
Větší než	>
Větší nebo rovno	>=
Logické AND	&&
Logické OR	||

Podmínky v jazyce C

- Syntaxe podmínky pomocí `switch`:

```
1   switch( key ){      // testování hodnoty key
2       case 65:        // key = 65 (ASCII kód "A")
3           ...         // kód vykonej pouze je-li key = 65
4           break ;     // konec switch
5       case 72:        // key = 72 (ASCII kód "H")
6           ...         // kód vykonej pouze je-li key = 72
7           break ;     // konec switch
8       default:        // key = libovolná jiná hodnota
9           ...         // tělo podmínky
10  }
```

Cyklus

- Syntaxe cyklu `for`:

```
for( počáteční_hodnota; podmínka_opakování; změna ){
    ...             // opakuj od "počáteční_podmínka" pokud
    ...             // platí "podmínka_opakování"; s každým
    ...             // opakováním změň hodnotu o "změna"
}

...

for( i = 0; i < 8; i++ ){              // opakuj 8krát
    ...                                // i = 0, 1, 2, ..., 6, 7
}

for( i = 80; i >= 10; ii = i-10 ){     // opakuj 8krát
    ...                                // i = 80, 70, ..., 20, 10
}
```

- Syntaxe cyklu `while`:

```
while( podmínka ){
    ...                 // opakuj dokud je podmínka splněna
}

...

a = 10 ;                // opakuj 10krát
while( a > 0 ){         // opakuj dokud je podmínka splněna
    ...                 // tělo cyklu
    a-- ;               // a = a - 1
}
```

Nekonečný cyklus – podmínka vždy splněna

- Nekonečná smyčka:

```
1    while( 1 ){      // podmínka opakování je vždy splněna; je rovna 1
2        ...          // tělo cyklu
3    }
4    for( ;; ) {      // podmínka opakování je vždy splněna
5        ...          // tělo cyklu
6    }
```

- Prázdná nekonečná smyčka – např. aplikace s přerušením:

```
1    while( 1 ) ;     // prázdná nekonečná smyčka
2    for( ;; ) ;      // prázdná nekonečná smyčka
```

Ukazatele, nepřímé adresování

- Ukazatel (pointer) je proměnná, jejíž hodnota je adresa paměti.

Např. `ptr = &temp ;`

  - `ptr` "ukazuje" na hodnotu proměnné `temp`,
  - `ptr` je proměnná, `&temp` je konstanta (adresa).

- Operátor nepřímého adresování `*`:
  - Operátor `*` zpřístupňuje hodnotu, na kterou ukazuje ukazatel. (Hodnota uložená na adrese ukazatele.)

  Např. `val = *ptr ;`

- Zápis `val = temp ;` je tedy totožný s posloupností dvou příkazů:

```
ptr = &temp ;
val = *ptr ;
```

- Deklarace ukazatele obsahuje typ proměnné, na kterou ukazuje:

Např. `int *pi ; // ukazatel na proměnnou typu int`

Pole

- Pole umožňují uchování několika prvků stejného typu v přehledném tvaru.
- Deklarace obsahuje typ prvků, název pole a počet prvků:

Např. `float debts[20] ; // proměnná debts obsahuje 20 prvků typu float`

- První prvek: `debts[0]` – poslední prvek: `debts[19]`
- Příklad naplnění při deklaraci:

  `float temp[N] = {3.14, 5.2, -1.3} ;`

- Přístup k prvkům pole pomocí cyklu:

```
1   #define N 3     // bez středníku
2                   // VŽDY používat symbolické jméno, konstantu
3   float temp[N] = {3.14, 5.2, -1.3} ;  // deklarace jednorozměrného pole
4
5   ...
6   for( i = 0; i < N; i++ )
7       printf( "%d: %5.2f\n", i,temp[i] ) ;
8   ...
9   // 0:  3.14
10  // 1:  5.20
11  // 2: -1.30
```

Pole

- Přístup k prvkům pole pomocí ukazatele:

```
1   #define N 3                                  // počet prvků pole
2   float temp[N] = {3.14, 5.2, -1.3} ;          // deklarace + naplnění pole
3   float *ptr ;                                 // deklarace ukazatele
4   ...
5   ptr = temp ;                                 // ekvivalent ptr = &temp[0] ;
6   for( i = 0; i < N; i++ )
7       printf( "%d: %5.2f\n", i,*ptr++ ) ;
8   ...
9   // 0:  3.14
10  // 1:  5.20
11  // 2: -1.30
```

Obsah přednášky

Překlad zdrojových modulů

Pozn.: Popisuje překlad v prostředí Linux – obdobně též ve Windows.

- Nechť aplikace obsahuje dva zdrojové soubory `f1.c` a `f2.c`:

```c
int main( void ){                   // zdrojový soubor f1.c
    int i ;

    i = square( 10 ) ;
    printf( "i = %d\n", i ) ;

    return( 0 ) ;
}
```

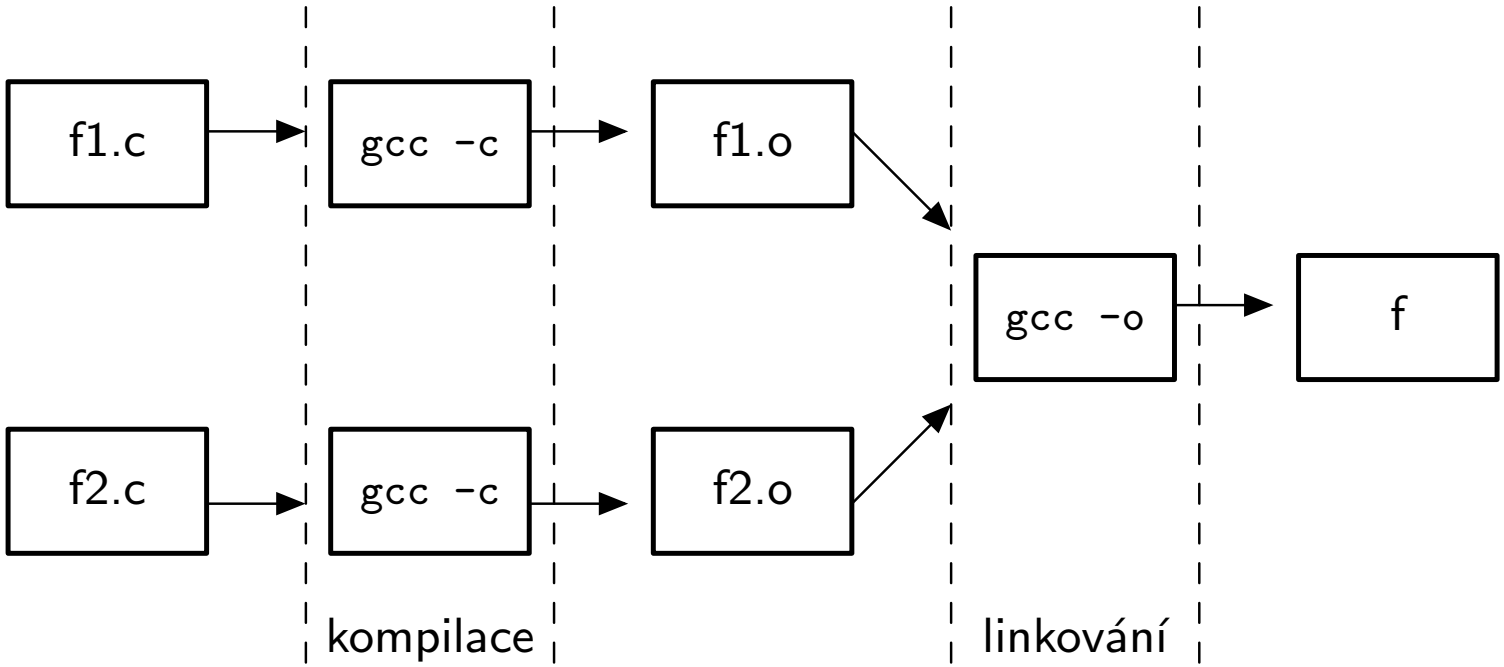
```c
double square( double f ){          // zdrojový soubor f2.c

    return( f*f ) ;
}
```

Tabulka: Základní přepínače překladače gcc.

Přepínač	Funkce
`gcc -c`	Kompilace a vytvoření objektového souboru.
`gcc -S`	Kompilace a vytvoření souboru v asembleru.
`gcc -o run`	Kompilace a vytvoření spustitelného souboru `run`.
`gcc -E`	Spuštění pouze preprocesoru – zpracovává pouze řádky s #.
`gcc -g`	Generuje ladící informace pro použití s GDB debugerem.

Překlad závislých souborů



Obrázek: Postup překladu aplikace v jazyce C pomocí gcc.

Překlad zdrojových modulů

- Oddělený překlad souborů `f1.c` a `f2.c`:
  - `gcc -c f1.c`
    Upozornění při překladu: `warning: incompatible implicit declaration of built-in function 'printf'`
  - Špatný (resp. chybějící) prototyp funkce `printf` – knihovna `stdio.h`.

Pozn.: Užitečný přepínač `-Wall` – zobrazí veškeré upozornění při překladu.

  - `gcc -c -Wall f2.c`
- Následné linkování objektových souborů:
  - `gcc f1.o -o f`
    Chyba překladu: `undefined reference to 'square'`
- Zdrojové soubory musí mít definované prototypy všech používaných funkcí:

```c
#include <stdio.h>                  // PŘIDÁNO; zdrojový soubor f1.c
#include "f2.h"                     // PŘIDÁNO

int main( void ){
    int i ;

    i = square( 10 ) ;
    printf( "i = %d\n", i ) ;

    return( 0 ) ;
}
```

```c
extern double square( double f ) ;  // VYTVOŘENO; hlavičkový soubor f2.h
```

Překlad zdrojových modulů

- Správný překlad a linkování souborů `f1.c` a `f2.c`:
  - `gcc -c -Wall f1.c`
  - `gcc -c -Wall f2.c`
  - `gcc f1.o f2.o -o f`
    Vytvoření spustitelného souboru `f`.
- Spuštění vytvořeného programu: `./f`

```
i = 100
```

Překlad zdrojových modulů

- Ukázka přidání matematické knihovny – výpočet druhé odmocniny:

```
1  #include <stdio.h>                  // zdrojový soubor f1.c
2  #include <math.h>                   // PŘIDÁNO; knihovna pro sqrt
3  #include "f2.h"
4
5  int main( void ){
6      int i ;
7
8      i = square( 10 ) ;
9      printf( "i = %d\n", i ) ;
10
11     i = sqrt( i ) ;                 // PŘIDÁNO
12     printf( "i = %d\n", i ) ;       // PŘIDÁNO
13
14     return( 0 ) ;
15 }
```

- Nutný přepínač pro knihovnu `-l` a označení knihovny; v tomto případě `-lm`:
  - `gcc -c -Wall f1.c`
  - `gcc -c -Wall f2.c`
  - `gcc f1.o f2.o -o f -lm`
- Spuštění vytvořeného programu: `./f`

```
i = 100
i = 10
```

Překlad zdrojových modulů s pomocí souboru `Makefile`

- Pro efektivní překlad vzájemně závislých souborů se standardně využívá soubor s názvem `Makefile`. Překlad se spustí příkazem `make`, který využívá právě tohoto souboru v aktuálním adresáři.

Syntaxe `cíl : závislost1 závislost2 ...`

Pozn.: Cíl nemusí být jen název souboru, ale také příkaz – viz níže.

- Jednotlivé příkazy MUSÍ být uvozeny tabulátorem!

```
1  ### f závisí na f1.o f2.o          skriptový soubor Makefile
2  f : f1.o f2.o
3  ——→gcc f1.o f2.o —o f —lm
4
5  ### f1.o závisí na f1.c f2.h
6  f1.o: f1.c f2.h
7  ——→gcc —c —Wall f1.c
8
9  ### f2.o závisí na f2.c
10 f2.o: f2.c
11 ——→gcc —c —Wall f2.c
```

- Spuštění překladu: `make` (použije první cíl ze souboru) nebo `make f` (zde je ekvivalentní)

```
gcc -c -Wall f1.c
gcc -c -Wall f2.c
gcc f1.o f2.o -o f -lm
```

- Spuštění vytvořeného programu: `./f`

```
i = 100
i = 10
```

Využívání maker v souboru `Makefile`

```
1  ##############################################
2  #
3  BINDIR  = /usr/bin
4  CFLAGS  = -c -Wall
5  LDFLAGS = -lm
6  OFILES  = f1.o f2.o
7  CC      = $(BINDIR)/gcc
8  LD      = $(BINDIR)/gcc
9  RM      = rm -f
10 PROG    = f
11
12 ##############################################
13 #
14 f : $(OFILES)
15     $(LD) $(LDFLAGS) $(OFILES) -o $(PROG)
16
17 f1.o : f1.c f2.h
18     $(CC) $(CFLAGS) f1.c
19
20 f2.o: f1.c
21     $(CC) $(CFLAGS) f2.c
22
23 ##############################################
24 #
25 clean:
26     $(RM) $(OFILES)
```

- Spuštění překladu: `make` (použije první cíl ze souboru) nebo `make f`

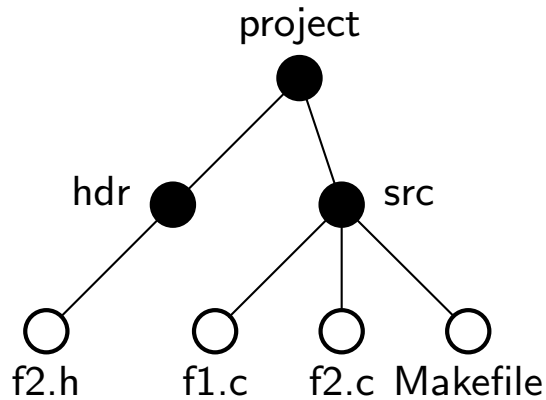
```
gcc -c -Wall f1.c
gcc -c -Wall f2.c
gcc f1.o f2.o -o f -lm
```

- Odstranění objektových souborů: `make clean`

```
rm -f f1.o f2.o
```

Organizace zdrojových souborů

- Vzhledem k tomu, že aplikace zpravidla obsahuje větší množství zdrojových souborů, je vhodné je organizovat od odlišných adresářů.
- Separujeme zdrojové soubory a soubory, které vkládáme pomocí `#include`:
  src/ zdrojové a objektové soubory, `Makefile`,
  hdr/ hlavičkové soubory. Pro překlad přidat přepínač `-I`.



Obrázek: Organizace souborů v projektu.

```
1  ##################################################
2  #
3  BINDIR  = /usr/bin
4  CFLAGS  = -c -g -Wall -I../hdr     # PŘIDÁNO
5  LDFLAGS = -g -lm
6  OFILES  = f1.o f2.o
7  CC      = $(BINDIR)/gcc
8  LD      = $(BINDIR)/gcc
9  RM      = rm -f
10 PROG    = f
11 ...
```

Obsah přednášky

Specifika překladače AVR-GCC a knihovny avr-libc

- Volný překladač GCC (GNU Compiler Collection), uzpůsobený pro aplikace s AVR.
- AVR Studio: konkrétní typ mikrokontroléru (např. ATmega16) je definován při vytvoření projektu – překladači předán jako parametr. Do kódu se tak vkládá vždy stejný hlavičkový soubor `io.h`, který následně odkáže na konkrétní definiční soubor.
- V definičním souboru jsou jména registrů definovány VELKÝMI písmeny (pozor, jazyk C je case sensitive).
- Plnění kontrolních registrů:

```
1  #include <avr\io.h>  // hlavičkový soubor mikrokontroléru
2  ...
3      DDRD = 0xF0 ;     // směrový registr portu B: polovina výst., polovina vst.
4
5      // nastavení předděličky 1024 pro časovač 0
6      TCCR0 |= ( 1<<CS02 ) | ( 1<<CS00 ) ;
7      /*
8       *  +------+-----------+--------+
9       *  | fcpu | N: CS02:0 |  tovf  |  tovf = 1/fcpu * N * (256-default value)
10      *  +======+===========+========+
11      *  | 120k |     0: 000 |   ---  |
12      *  |      |     1: 001 |   2.1m |
13      *  |      |     8: 010 |  17.1m |
14      *  |      |    64: 011 | 136.5m |
15      *  |      |   256: 100 | 546.1m |
16      *  |      |  1024: 101 | 2.185s |
17      *  +------+-----------+--------+
18      */
19 }
```

Vybrané funkce AVR-GCC a knihovny avr-libc

- Testování hodnoty bitu v KONTROLNÍM registru (obdoba instrukcí `SBIS`, `SBIC`):

```
1   // je-li bit "b" v registru "reg" roven 1, vykonej následující kód
2   // reg - libovolný řídicí registr
3   if( bit_is_set( reg, b )){
4       ...
5   }
6
7   // je-li bit "b" v registru "reg" roven 0, vykonej následující kód
8   // reg - libovolný řídicí registr
9   if( bit_is_clear( reg, b )){
10      ...
11  }
```

- Cyklus s testováním bitu v KONTROLNÍM registru:

```
1   // opakuj, pokud je bit "b" v registru "reg" roven 1
2   // reg - libovolný řídicí registr
3   loop_until_bit_is_set( reg, b ){
4       ...
5   }
6
7   // opakuj, pokud je bit "b" v registru "reg" roven 0
8   // reg - libovolný řídicí registr
9   loop_until_bit_is_clear( reg, b ){
10      ...
11  }
```

Obsluha přerušení

- Pro využití přerušení v jazyce C je nutné vložit hlavičkový soubor `interrupt.h`.
- Není nutné definovat zásobník; respektive udělá to překladač. Rovněž není nutné znát konkrétní adresy vektorů přerušení.
- Obsluhy přerušení představují vždy makra `ISR` (anglicky: Interrupt Service Routine) se vstupním parametrem, který identifikuje zdroj přerušení:

```
1  #include <avr\io.h>          // hlavičkový soubor mikrokontroléru
2  #include <avr\interrupt.h>   // hlavičkový soubor pro přerušení
3
4  ISR( INT0_vect ){            // obsluha externího přerušeni INT0
5      ...                      // kód obsluhy přerušení
6  }
7
8  ISR( ADC_vect ){             // obsluha přerušení A/D převodníku
9      ...                      // kód obsluhy přerušení
10 }
11
12 int main( void ){
13     ...                      // kód hlavní funkce
14     sei() ;                  // globální povolení přerušení
15     ...
16 }
```

Parametry makra obsluhy přerušení `ISR(.)` pro ATmega16

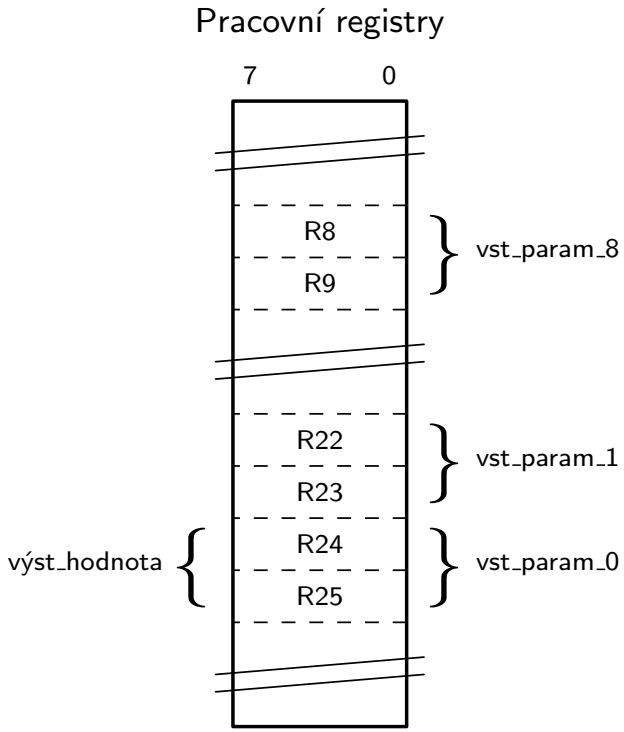
Č.	Adresa	Parametr v C (GCC)	Popis přerušení
1	0x0000		Externí reset, připojení napájení, ...
2	0x0002	INT0_vect	Externí požadavek na přerušení 0.
3	0x0004	INT1_vect	Externí požadavek na přerušení 1.
4	0x0006	TIMER2_COMP_vect	Časovač/čítač 2 – shoda komparace.
5	0x0008	TIMER2_OVF_vect	Časovač/čítač 2 – přetečení.
6	0x000A	TIMER1_CAPT_vect	Časovač/čítač 1 – zachycení.
7	0x000C	TIMER1_COMPA_vect	Časovač/čítač 1 – shoda s komparátorem A.
8	0x000E	TIMER1_COMPB_vect	Časovač/čítač 1 – shoda s komparátorem B.
9	0x0010	TIMER1_OVF_vect	Časovač/čítač 1 – přetečení.
10	0x0012	TIMER0_OVF_vect	Časovač/čítač 0 – přetečení.
11	0x0014	SPI_STC_vect	Dokončení sériového přenosu SPI.
12	0x0016	USART_RXC_vect	USART – kompletní příjem dat.
13	0x0018	USART_UDRE_vect	USART – prázdný datový registr.
14	0x001A	USART_TXC_vect	USART – kompletní vyslání dat.
15	0x001C	ADC_vect	ADC – dokončení A/D převodu.
16	0x001E	EE_RDY_vect	EEPROM – komunikace připravena.
17	0x0020	ANA_COMP_vect	Změna výstupu analogového komparátoru.
18	0x0022	TWI_vect	Událost na sériové sběrnici I2C.
19	0x0024	INT2_vect	Externí požadavek na přerušení 2.
20	0x0026	TIMER0_COMP_vect	Časovač/čítač 0 – shoda komparace.
21	0x0028	SPM_RDY_vect	Uložení do programové paměti připraveno.

Kombinace jazyka symbolických adres a C

- Programování ve vyšším jazyce je pohodlnější, rychlejší, kód je snáze přenositelný mezi platformami; zpravidla zabírá více místa v paměti, proto je i jeho výkon pomalejší.
- Běžná praxe je tvorba aplikace v jazyce C (platí bezezbytku u více-bitových procesorů), pouze exponované části programu, kde je nutné mít přehled o počtu instrukcí, rychlosti výkonu, či konkrétní funkci se tvoří v JSA.
- Proto lze zdrojové kódy kombinovat:
  (1) překladač vloží konkrétní instrukce pomocí funkce `asm()`

```
1    asm( "LDI  R16, 255\n DEC  R16\n BRNE  PC-1\n" ) ;
2    asm( "SEI" ) ;
3        // při kompilaci zdrojového kódu se vloží následující instrukce:
4        // LDI  R16, 255
5        // DEC  R16
6        // BRNE PC-1
7        // SEI
```

Kombinace jazyka symbolických adres a C



Obrázek: Předávání vstupních parametrů/výstupní hodnoty mezi kódem v jazyce C a JSA.

(2) předávání parametrů mezi funkcí v JSA a hlavní aplikací v C pomocí předem určených registrů. Překopírování hodnot parametrů/návratové hodnoty do/z registrů zajišťuje překladač:

- možnost použít až devět vstupních parametrů funkce,
- první vstupní parametr vždy v registrovém páru R25:R24,
- ...
- devátý vstupní parametr: R9:R8,
- návratovou hodnotu funkce musí programátor před ukončením rutiny uložit do registrového páru R25:R24.

Prototyp `extern int add_compl( int, int ) ;`

Volání `c = add_compl( a, b ) ;`

- 16bitová hodnota proměnné `a` je dostupná v registrech R25:R24,
- `b` @ R23:R22,
- výstupní hodnotu musíme uložit do registrů R25:R24 @ `c`.

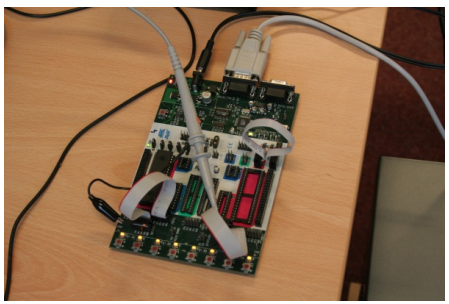
Obsah přednášky

Vývoj aplikací, simulátory, emulátory

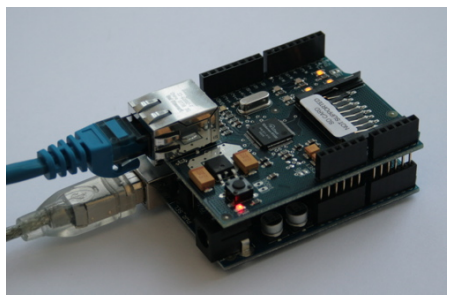
- Postup při vývoji aplikací: vytvoření zdrojového kódu aplikace a její odladění v simulátoru, příp. emulátoru; hardwarovou část je vhodné vyzkoušet na vývojové desce.
- Simulátor:
  - simulace chování mikrokontroléru na odlišném zařízení (typ. PC),
  - simulátor musí obsahovat převod zdrojového kódu do strojového jazyka požadovaného mikrokontroléru,
  - spouštěcí a ladící programy: krokování programu, breakpointy, . . .,
  - omezené možnosti simulace okolního prostředí (tlačítka, přerušení, . . .),
  - nepracuje v reálném čase: je možné získat informaci o době výkonu programu v počtech cyklů hodinového signálu.
- Emulátor:
  - propojení PC + emulační koncovka s cílovým mikrokontrolérem,
  - obslužný software (na PC) obsahuje stejné náležitosti jako u simulátoru + odlišné spouštěcí a ladící nástroje,
  - umožňuje monitorování cílového mikrokontroléru během ladění (obsah registrů, zásobníku, . . .),
  - je možné plně odladit novou aplikaci včetně např. vazby na okolí,
  - pracuje v reálném čase.

Vývoj aplikací, vývojové desky

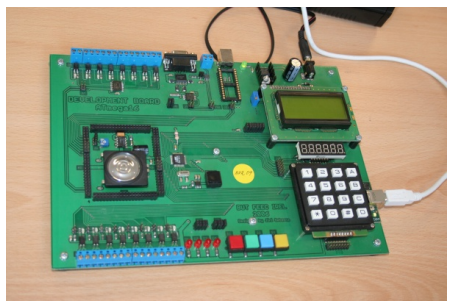
- Vývojová deska:
  - např.: STK500, Arduino, . . .
  - hardwarové zařízení umožňující odladění aplikace včetně připojení základních periferií (LED, tlačítka, LCD, relé, . . .),
  - může obsahovat patice pro odlišné mikrokontroléry, pájivé pole, . . .,
  - v závislosti na aplikaci, není potřeba vytvářet finální zapojení před odladěním – univerzální deska usnadňuje/urychluje vývoj nové aplikace.



(a)



(b)



(c)

Obrázek: Vývojové desky: (a) STK500, (b) Arduino+Ethernet shield, (c) ATmega16 Development Board.

Obsah přednášky

Kombinace JSA a C

- Pevně definované pozice pro vstupní parametry (16bit.) a návratovou hodnotu (jediná).
- Pokud 8bitové operandy – využije se jen jeden registr; druhý – prázdný.
- Před ukončení funkce v JSA MUSÍ programátor uložit návratovou hodnotu do určených registrů, tj. R25:24.

Otázka

Co dělat, potřebujeme-li více návratových hodnot (příp. více bitových)?

- Návrat pomocí `RET`.
- Nutné direktivy v JSA:
  - `.global`
  - `.func`, `.endfunc`

Kombinace JSA a C

```c
#include <avr\io.h>          // hlavičkový soubor mikrokontroléru; soubor f1.c

extern int add_compl( int, int ) ;  // prototyp funkce z jiného souboru

int main( void ){
    int a = 0x0A05 ;         // a_r = 10, a_i = 5
    int b = 0x1403 ;         // b_r = 20, b_i = 3
    int c ;

    c = add_compl( a, b ) ;  // volání funkce v JSA
    while( 1 ) ;             // nekonečná smyčka

    return( 1 ) ;            // výstupní hodnota funkce = 1
}
```

```
/*                                                          soubor f1.sa
 * extern int add_compl( int x, int y ) ;
 *      a = R25:R24 (= 0x0A05 : a_r = 10, a_i = 5)
 *      b = R23:R22 (= 0x1403 : b_r = 20, b_i = 3)
 * return = R25:R24 (= 0x1E08 : c_r = 30, c_i = 8)
 */

#include <avr\io.h>          // hlavičkový soubor mikrokontroléru

    .global add_compl        // definování globálního parametru
    .func   add_compl        // začátek funkce
add_compl:
    ADD R25, R23             // tělo funkce (součet komplexních čísel)
    ADD R24, R22

    RET                      // návrat z funkce/podprogramu; výsledek v R25:24
    .endfunc                 // konec funkce
```

Podmíněné větvení programu v JSA a v C

- Podmíněné větvení programu slouží k řízení běhu programu. "Pokud je splněna podmínka něco vykonej."
- Mikrokontroléry AVR obsahují 25 různých instrukcí pro podmíněné větvení programu.
- Použité instrukce:
  - `SBIS PIND, 0x00` – přeskoč následující instrukci, pokud je pin č. 0 na portu D roven hodnotě 1.
- Ukázka jiné instrukce podmíněného skoku:
  - `DEC const`
  - `BRNE loop` – pokud je `const` různé od nuly skoč na návěští `loop` (využívá nulový příznakový bit).
- Použití podmínek pro překladač AVR-GCC:
  - `if( bit_is_set( PINA, 0 ))` jestliže pin č. 0 na portu A je roven 1,
  - `if( bit_is_clear( PINC, 1 ))` jestliže pin č. 1 na portu C = 0.

Podmíněné větvení programu v JSA a v C

```
1  .include <m16def.inc>     ; definiční soubor mikrokontroléru ATmega16
2  .def temp = R16           ; temp = registr R16
3
4  reset:
5      CLR  temp             ; nuluj všechny bity v registru temp
6      OUT  DDRD, temp       ; port D = vstupní
7      SER  temp             ; nastav všechny bity v registru temp
8      OUT  DDRB, temp       ; port B = výstupní
9  loop:
10     SBIS PIND, 0x00       ; pokud pin 0 na portu D = 1 přeskoč následující instr.
11     INC  temp             ; inkrementuj obsah registru temp
12     SBIS PIND, 0x01       ; pokud pin 1 na portu D = 1 přeskoč následující instr.
13     DEC  temp             ; dekrementuj obsah registru temp
14     OUT  PORTB, temp      ; pošli obsah temp na port B
15     RJMP loop             ; skok na návěští loop
```

```
1  #include <avr\io.h>       // hlavičkový soubor mikrokontroléru
2
3  int main( void ){         // hlavní funkce aplikace
4      char temp = 0 ;       // deklarace lokální 8bitové proměnné temp = 0
5      DDRD = 0x00 ;         // port D = vstupní
6      DDRB = 0xFF ;         // port B = výstupní
7
8      while( 1 ){           // nekonečná smyčka
9                            // pokud pin 0 na portu D = 0, inkrementuj temp
10         if( bit_is_clear( PIND, 0 )) temp++ ;
11                           // pokud pin 1 na portu D = 0, dekrementuj temp
12         if( bit_is_clear( PIND, 1 )) temp-- ;
13         PORTB = temp ;    // pošli obsah proměnné temp na výstupní port B
14     }
15     return( 1 ) ;
16 }
```