Programování MCU ve vyšších programovacích jazycích

Úvod

- vyšší programovací jazyk = High Level Language (HLL)
- Dříve nemyslitelné, důvody: nedostatek RAM, ROM, malý výkon CPU
- dnes poměrně běžné, používané jazyky:
  - C → absolutně nejčastěji
  - C++ (EC++ – Embedded C++) → jen větší typy (16 bit a výše)
  - Pascal, Basic, atd → také, ale…
- Důvody:
  - rychlost vývoje
  - lehčí přenositelnost na jiné typy CPU (MCU)
- ideální stav:
  - architektura MCU je přizpůsobena požadavkům překladače
  - kód je stejně efektivní jako Assembler
- skutečnost → použití HLL = nárůst kódu (code overhead):
  - nepřizpůsobená architektura 100ky %
  - přizpůsobená 50 – 100% (závisí na programátorovi)
- dále HLL = C

Požadavky HLL na architekturu CPU

vlastnost	komentář	podpora AVR	podpora 8051
několik akumulátorů	žádné přesuny operátorů při ALU operacích	ano (32 GPR)	ne (pouze Acc)
podpora pointerů	dobré nejméně 4 nezávislé (SP, data, cíl, zdroj)	ano (SP, X, Y, Z)	velmi chabá (@R0,1)
bitové adresování	pro bitové proměnné (bool)	ne	ano
přemístitelný zásobník	SP pro plný adresový prostor	ano (16b SP)	omezeně (8b SP)
podpora polí	efektivní přístup na pole přes pointer na začátek pole a offsetu (displacement), podpora ++ a --	ano	ne
efektivní výpočet adres	podpora +- operací s šířkou = šířce adresy	ano (ADIW, MOVW)	jen INC/ DEC DPTR
dostatek zdrojů	RAM – pro operace	RISC, paměti	omezeně

	s čísly > šířka ALU ROM – code overhead výkon CPU	dle typu	(externí paměti, CISC) moderní klony lepší

Kompilátory pro MCU

- nejznámější kompilátory
  - komerční
    - 8051 – Keil, IAR
    - AVR – Keil, IAR
  - free
    - 8051 – SDCC
    - AVR – AVR GCC

Kompilátor HLL a non-PC aplikace

- nutné omezení specifikace HLL (např. dle ANSI C):
  - práce s obrazovkou a klávesnicí (`printf()`, `scanf()` atd.)
- rozšíření specifikace HLL
  - non-PC hardware (porty, periferie)
  - nové datové typy
  - harvardská architektura (program je odděleně od dat)
  - přerušení (obsluha, povolení/zakázání)
- program v assembleru = pouze výkonný kód
- program v HLL =
  - výkonný kód
  - inicializace CPU pro běh programu (Run time Environment, Start-up code)
    - pro ATmega a AVR GCC je to cca 150-200 B kódu

Kompilátor AVR GCC

- velmi kvalitní C kompilátor, GPL licence
- možnost snadno a plně integrovat do AVR studia
- vyhovuje ANSI C (= obsahuje všechny knihovny)

Přístup k HW AVR

- nutno `#include <avr/io.h>`
  v něm pomocí #include definovány jména a adresy registrů v SFR → lze je používat jako běžné proměnné
- bytový přístup na porty:

```
unsigned char  MujPort;

DDRA = 0xff;            // port A = vystupy
PORTA = 0x03;           // zapis Bytu na port A (= bity 0,1 do Log 1)
                        // lepší by bylo PORTA = (1<<PA1) | (1<<PA0)
MujPort = PINA;         // cteni Bytu z portu A, pozor nikoli PORTA!
```

- zápis bitů na porty
  - nahození bitů do 1 (a zbytek do 0)
    ```
    PORTA = (1<<PA0) | (1<<PA3);
    ```
  - nahození bitů do 1 (a zbytek ponechán stávající stav)
    ```
    PORTA |= (1<<PA0) | (1<<PA3);
    ```
  - shození bitů do 0 (a zbytek do 1)
    ```
    DDRA = ~((1<<PA0) | (1<<PA3));
    ```
  - shození bitů do 0 (a zbytek ponechán stávající stav)
    ```
    DDRA &= ~((1<<PA0) | (1<<PA3));
    ```

- větvení na základě stavu bitu `BIT` na adrese `ADR`
  - využívá se maskování a použitím int jako datového typu pro bool v C (`0` → `FALSE`; `!= 0` → `TRUE`)
    ```
    if (ADR & (1<<BIT)) {          // if (BIT == 1)
      // akce
    }

    if (!(ADR & (1<<BIT)) {        // if (BIT == 0)
      // akce
    }
    ```
  - pro cykly stejné – příklad funkce pro odeslání bytu na UART pro ATmega 16 (přes polling – dotazování na stav vysílacího registru – bit UDRE v registru UCSR)
    ```
    void USART_Transmit(uint8_t c)
    {
          // cekani na prazdny buffer
          while (!(UCSR & (1<<UDRE))); // prazdny while
          UDR1 = c;   // odeslani dat
    }
    ```
  - jak překladač přeloží `while (!(UCSR & (1<<UDRE)));`
    ```
    SBIS    0x0B,5          Skip if bit in I/O register set
    RJMP    PC-0x0001       Relative jump
    // další prikaz po while…
    ```

Práce s „velkými“ datovými typy

- celočíselné většinou bez problémů
- u 8b překladačů platí:
  - `char` 8b
  - `int` 16b
  - `long` 32b
- příklad
  ```
  int main (void)
  {
        long Prom1, Prom2, Vysl;

        Prom1 = 12345678;
        Prom2 = 87654321;
        Vysl = Prom1 + Prom2;

        return 0;
  }
  ```

```
4:          Prom1 = 12345678;
LDI     R24,0x4E        Load immediate
LDI     R25,0x61        Load immediate
LDI     R26,0xBC        Load immediate
LDI     R27,0x00        Load immediate
STD     Y+1,R24         Store indirect with displacement
STD     Y+2,R25         Store indirect with displacement
STD     Y+3,R26         Store indirect with displacement
STD     Y+4,R27         Store indirect with displacement
5:          Prom2 = 87654321;
LDI     R24,0xB1        Load immediate
LDI     R25,0x7F        Load immediate
LDI     R26,0x39        Load immediate
LDI     R27,0x05        Load immediate
STD     Y+5,R24         Store indirect with displacement
STD     Y+6,R25         Store indirect with displacement
STD     Y+7,R26         Store indirect with displacement
STD     Y+8,R27         Store indirect with displacement
7:          Vysl = Prom1 + Prom2;
LDD     R18,Y+1         Load indirect with displacement
LDD     R19,Y+2         Load indirect with displacement
LDD     R20,Y+3         Load indirect with displacement
LDD     R21,Y+4         Load indirect with displacement
LDD     R24,Y+5         Load indirect with displacement
LDD     R25,Y+6         Load indirect with displacement
LDD     R26,Y+7         Load indirect with displacement
LDD     R27,Y+8         Load indirect with displacement
ADD     R24,R18         Add without carry
ADC     R25,R19         Add with carry
ADC     R26,R20         Add with carry
ADC     R27,R21         Add with carry
STD     Y+9,R24         Store indirect with displacement
STD     Y+10,R25        Store indirect with displacement
STD     Y+11,R26        Store indirect with displacement
STD     Y+12,R27        Store indirect with displacement
```

  - zde již se začíná objevovat výhoda 32 GPR
  - pro ATmega32 kód 224B

- ale co toto:

```
int main (void)
{
      double Prom1, Prom2, Vysl;

      Prom1 = 12345678;
      Prom2 = 87654321;
      Vysl = Prom1 + Prom2;

      return 0;
}
```

  - pro ATmega32 kód 1638 B

- nebo toto:

```
#include <math.h>

int main (void)
{
      double Uhel = 30, SinUhlu;

      SinUhlu = sin(Uhel*3.141592/180);

    return 0;
```

```
}
```

- pro ATmega32 kód 2742 B

Optimalizace překladače

- u MCU nutné
- u AVR GCC 5 úrovní:
  - O0 – optimalizace vypnuta
  - O1 ÷ O3 – optimalizace (vyšší číslo = vyšší stupeň opt.)
  - Os – opt. na velikost kódu (= O2 které nezvětšují velikost kódu + něco navíc)
- Nelze paušalizovat, vždy nutno vyzkoušet

AVR Libc

- = část C standardních knihoven (ANSI C) pro AVR
- Budou zmíněny jen některé, podrobně viz. http://www.nongnu.org/avr-libc/user-manual/modules.html

stdint.h

- Libc definuje vlastní datové typy

```
typedef signed char int8_t
typedef unsigned char uint8_t
typedef signed int int16_t
typedef unsigned int uint16_t
typedef signed long int   int32_t
typedef unsigned long int uint32_t
typedef signed long long int int64_t
typedef unsigned long long int uint64_t
```

  - `long long` = double word int dle ISO C99– 64b
- pozor: `double` taktéž 32b!

pgmspace.h

- Při práci s konstantními daty v ROM (paměť programu - flash u AVR)
- pokud např. v programu napíšeme

```
char *Text = "Dobry den, jak se mate";
```

  - proměnná je inicializovaná → text musí být v ROM
  - leží v RAM → po resetu je obsah z ROM kopírován do RAM
    - text → 2x
    - zabírá zbytečně RAM
- řešení → MCU mají instrukce pro čtení z ROM (LPM) → data uložíme pouze do ROM → rozšíření jazyka C (u AVR GCC nutno `#include <avr/pgmspace.h>`)
  - datové typy pro proměnné v ROM

```
typedef void PROGMEM          prog_void
typedef char PROGMEM          prog_char
typedef unsigned char         PROGMEM     prog_uchar
typedef int8_t PROGMEM        prog_int8_t
typedef uint8_t PROGMEM       prog_uint8_t
typedef int16_t PROGMEM       prog_int16_t
typedef uint16_t PROGMEM      prog_uint16_t
```

```
typedef int32_t PROGMEM       prog_int32_t
typedef uint32_t PROGMEM      prog_uint32_t
// pointer pro řetězce
#define PGM_P           const prog_char *
// obecný pointer do ROM
#define PGM_VOID_P      const prog_void *
```

- funkce pro práci s ROM

```
void *memcpy_P (void *, PGM_VOID_P, size_t)
char *strcpy_P (char *, PGM_P)
// a několik dalších (ekv. k ANSI C funkcím, ale s _P)

pgm_read_byte_near()    // cteni 1B z ROM (do 64KB)
pgm_read_byte_far()     // cteni 1B z ROM (nad 64KB)
```

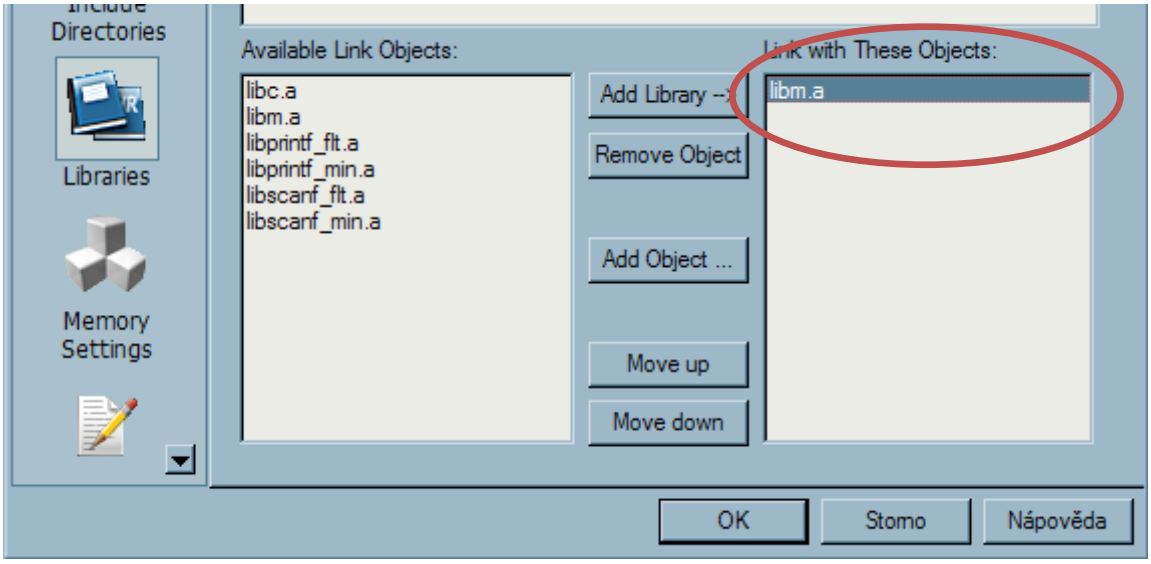
- příklad → řetězec v ROM

```
#include <avr/io.h>
#include <avr/pgmspace.h>

const char TextROM[] PROGMEM = "Dobry den, jak se mate";

int main (void)
{
    char TextRAM[32];
    // kopie textu z ROM do RAM
    strcpy_P(TextRAM, TextROM);
    return 0;
}
```

- poznámka: `const char TextROM[] PROGMEM = "…";`
  - tento formát nutno dodržet, např. `const char *TextROM PROGMEM = "…"` již vede ke kopii řetězce do RAM
  - vždy je nutno se podívat do dokumentace k překladači

math.h

- jako v ANSI C
- nutno přilinkovat matematickou knihovnu libm.a



stdio.h

- AVR nemá vstupní a zobrazovací zařízení → nelze otevřít standardní proudy
  - stdin = klávesnice
  - stdout = obrazovka
  - stderr = obrazovka
- Základem je princip:
  - printf() apod. → používá putchar()
  - scanf() apod.→ používá getchar()
- nutno přesměrovat – např. LCD displej, UART, …

```
#include <stdio.h>
#include <avr/io.h>

static int uart_putchar(char c, FILE *stream);
void init_uart();

static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar, NULL,
_FDEV_SETUP_WRITE);

static int uart_putchar(char c, FILE *stream)
{
	while(!(UCSRA, (1<<UDRE)));
	UDR = c;
	return 0;
}

int main(void)
{
	init_uart();
	stdout = &mystdout;
	printf("10 = %d", 10);

	return 0;
}

void init_uart()
{
	// priklad zakladniho nastaveni
	UBRRH = 0;
	UBRRL = 25;
	UCSRC = (1<<URSEL) | (1<<UCSZ1) | (1<<UCSZ0);
}
```

- pozor: zabírá hodně ROM – příklad výše cca 2kB

stdlib.h

- dynamickou paměť nepoužívat – příliš náročné na zdroje MCU

delay.h

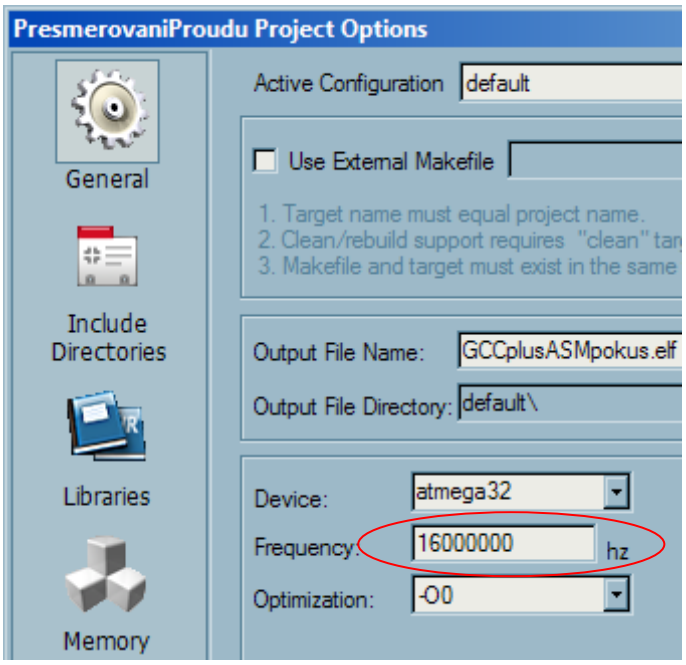
- generování čekacích smyček

```
void _delay_loop_1 (uint8_t __count)
void _delay_loop_2 (uint16_t __count)
```

- čekají `__count` počet strojových cyklů

```
void _delay_us (double __us)
void _delay_ms (double __ms)
```

- čekají `__us` (`__ms`) počet µs (ms)
  - nutno definovat makro `F_CPU` → f_{CLK} MCU v Hz:
    - ručně `#define F_CPU 16000000`
    - v AVR studiu



- `_delay_ms()` využívá `_delay_loop_2()` → max interval je 262.14 ms / `F_CPU` v MHz
- `_delay_us()` využívá `_delay_loop_1()` → max interval je 768 us / `F_CPU` v MHz
- příklad:

```
#include <util/delay.h>

#define F_CPU 16000000    // 16 MHz

int main(void)
{
        _delay_ms(12);     // 12 ms
        return 0;
}
```

string.h

- standardní fce
- pozn.: manipulace s řetězci v ROM je v pgmspace.h