AVR Libc pokračování

interrupt.h

- práce s přerušením
- povolení / zakázání přerušení
  - makro `sei()` = instrukce sei
  - makro `cli()` = instrukce cli
- obslužné funkce – vždy tvar

```
ISR(JMENO_VEKTORU)
{
    // obslužný kod
}
```

  - jména vektorů viz. `interrupt.h`, např. pro ATmega16:

externí přerušení 0	`INT0_vect`
přetečení časovač 1	`TIMER1_OVF_vect`
UART RX	`USART_RXC_vect`
atd.	

- proměnné a přerušení → přerušení je časově asynchronní událost:
  - nelze předávat žádné parametry
  - funkce nemůže vracet hodnotu
  - všechny proměnné používané v přerušeních by měly být deklarovány s modifikátorem `volatile` (pro kompilátor → nelze optimalizovat)
- příklad: LED had na ATmega32

```
/*
bezici had 8 LED na portu B ATmega32
perioda posunu asi 0,52s na flck = 1MHz
blikani pres 16b casovac T1 s preddelickou /8
*/
#include <avr/io.h>
#include <avr/interrupt.h>

typedef unsigned char uchar;
// globalni promenne
volatile uchar Had = 0x01;

// deklarace funkci
void Setup(void);

int main (void)
{
      Setup();
      for(;;);     // nekonecna smycka
      return 0;
}

void Setup(void)
{
      // nastaveni portu
      DDRB = 0xFF;            // cely port jako vystupni
      // nastaveni casovace T1
      TCCR1B = (1<<CS11);     // zapnuti T1 - zdroj osc/8
```

```
        // nastaveni preruseni
        TIMSK = (1<<TOIE1);     // preruseni od preteceni T1
        sei();                  // globalni povoleni preruseni
}

ISR(TIMER1_OVF_vect)
{
        PORTB = Had;
        if (Had == 0x80) {
                Had = 0x01;
        }
        else {
                Had <<= 1;      // posun hada o 1 misto doleva
        }
}
```

- jak byl celý program přeložen (optimalizace -o2, celý program 226 B, ve výpisu chybí startup code 160 B)

```
@00000047: setup {
27:         DDRB = 0xFF;            // cely port jako vystupni
SER     R24                 Set Register
OUT     0x17,R24            Out to I/O location
28:         TCCR1B = (1<<CS11);     // zapnuti T1 - zdroj osc/8
LDI     R24,0x02            Load immediate
OUT     0x2E,R24            Out to I/O location
30:         TIMSK = (1<<TOIE1);     // preruseni od preteceni T1
LDI     R24,0x04            Load immediate
OUT     0x39,R24            Out to I/O location
31:                 sei();      // globalni povoleni preruseni
SEI                         Global Interrupt Enable
RET                         Subroutine return
@0000004F: main
17:       {
LDI     R28,0x5F            Load immediate
LDI     R29,0x08            Load immediate
OUT     0x3E,R29            Out to I/O location
OUT     0x3D,R28            Out to I/O location
18:         Setup();
CALL    0x00000047          Call subroutine
RJMP    PC-0x0000           Relative jump
@00000056: __vector_9
35:       {
PUSH    R1                  Push register on stack
PUSH    R0                  Push register on stack
IN      R0,0x3F             In from I/O location
PUSH    R0                  Push register on stack
CLR     R1                  Clear Register
PUSH    R24                 Push register on stack
36:         PORTB = Had;
LDS     R24,0x0060          Load direct from data space
OUT     0x18,R24            Out to I/O location
37:         if (Had == 0x80) {
LDS     R24,0x0060          Load direct from data space
CPI     R24,0x80            Compare with immediate
BREQ    PC+0x05             Branch if equal
41:                 Had <<= 1;  // posun hada o 1 misto doleva
LDS     R24,0x0060          Load direct from data space
LSL     R24                 Logical Shift Left
RJMP    PC+0x0002           Relative jump
38:                 Had = 0x01;
LDI     R24,0x01            Load immediate
```

```
STS     0x0060,R24      Store direct to data space
POP     R24             Pop register from stack
POP     R0              Pop register from stack
OUT     0x3F,R0         Out to I/O location
POP     R0              Pop register from stack
POP     R1              Pop register from stack
RETI                    Interrupt return
```

- krátké rutiny → lepší napsat v asm (viz. dále)
  - `__attribute__ ((naked)) ISR(vector)` = „holé“ přerušení, úschovu/obnovu registrů nutno napsat ručně

Run-Time model překladače

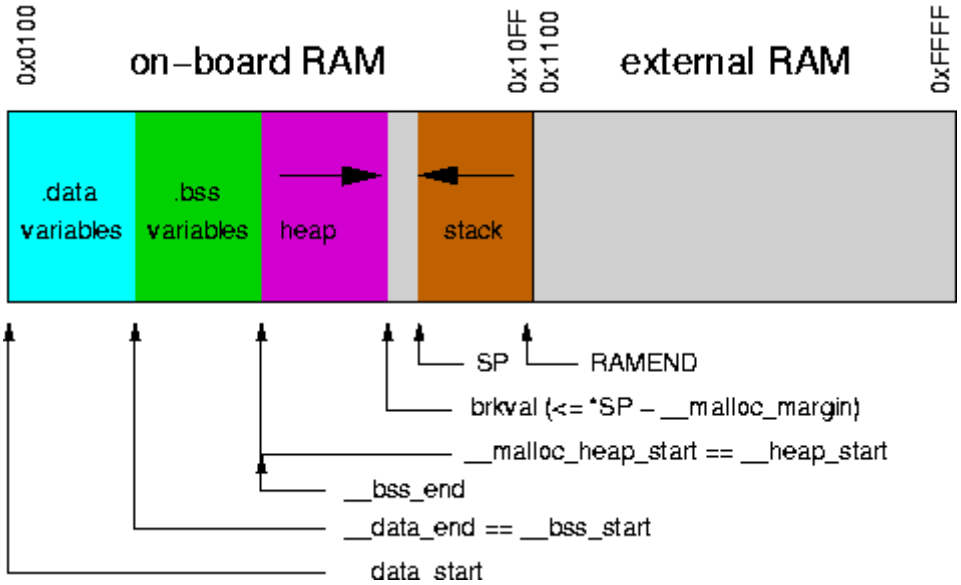
- = jak překladač využívá zdroje CPU
- AVR-GCC jen na MCU s min 128B RAM – nutný stack
  - Všechny lokální proměnné
  - Volání funkcí

Paměťový prostor

- (ROM, RAM, EEPROM) rozdělen do sekcí (sections)
- Nejdůležitější sekce:
  - .text – obsahuje program, dále se dělí
  - .data – inicializované globální a statické proměnné
  - .bss – neinicializované globální a statické proměnné
  - .eeprom – data umístěná do paměti EEPROM na čipu AVR
- Se segmenty lze ručně pracovat, příklad – proměnná v paměti EEPROM

```
int prom __attribute__((section(".eeprom")));
```

- Příklad rozložení paměti s ATmega128



  - Lze změnit

Start-up code

- Automaticky vloženo překladačem před main() – inicializace prostředí (stack, atd.)

- Lze modifikovat – podrobnosti viz. doc. k překladači (section .init)
- Velikost dle typu AVR – cca 200 B

```
JMP     0x00000045      Jump   // tabulka vektoru preruseni
            |
JMP     0x00000045      Jump
CLR     R1              Clear Register      // start-up code
OUT     0x3F,R1         Out to I/O location
LDI     R28,0x5F        Load immediate
LDI     R29,0x08        Load immediate
OUT     0x3E,R29        Out to I/O location
OUT     0x3D,R28        Out to I/O location
LDI     R17,0x00        Load immediate
LDI     R26,0x60        Load immediate
LDI     R27,0x00        Load immediate
LDI     R30,0xF6        Load immediate
LDI     R31,0x00        Load immediate
RJMP    PC+0x0003       Relative jump
LPM     R0,Z+           Load program memory and postincrement
ST      X+,R0           Store indirect and postincrement
CPI     R26,0x60        Compare with immediate
CPC     R27,R17         Compare with carry
BRNE    PC-0x04         Branch if not equal
LDI     R17,0x00        Load immediate
LDI     R26,0x60        Load immediate
LDI     R27,0x00        Load immediate
RJMP    PC+0x0002       Relative jump
ST      X+,R1           Store indirect and postincrement
CPI     R26,0x60        Compare with immediate
CPC     R27,R17         Compare with carry
BRNE    PC-0x03         Branch if not equal
JMP     0x00000065      Jump   // volani main()
JMP     0x00000000      Jump // sem by se spravny prog. nemel nikdy dostat
```

Využití registrů

- Pro svůj běh využívá program jisté skupiny registrů – nelze svévolně „ruční“ manipulaci (např. v asm)
- R0:
  - Lze využít v inline assembleru (v jednom `asm volatile`) jako přechodné úložiště
- R1:
  - Musí být vždy 0
  - Jako r0, ale nakonci nutno vynulovat (crl r1)
- R2 - R17, R28, R29
  - May be allocated by gcc for local data. Calling C subroutines leaves them unchanged. Assembler subroutines are responsible for saving and restoring these registers, if changed. r29:r28 (Y pointer) is used as a frame pointer (points to local data on stack) if necessary
- r18-r27, r30-r31
  - May be allocated by gcc for local data. You may use them freely in assembler subroutines. Calling C subroutines can clobber any of them - the caller is responsible for saving and restoring

Volání funkcí

- CPU – nemá instrukci „call s parametry“ – nutno ručně (resp. překladač):
  1) naplním dohodnutý prostor parametry
  2) call funkce
  3) funkce ví, kde leží parametry a pracuje s nimi
  4) konec funkce – návratová hodnota uložena na dohodnuté místo
  5) v „hlavním programu“ z dohodnutého místa je manipulováno s návratovou hodnotou
  - dohodnuté místo: registry nebo zásobník
- AVR GCC:
- Argumenty zleva doprava do registrů r25 až r8
  - Proměnná = lichý počet B → zaokrouhleno na sudý
  - Nejnižší B parametru = nejnižší registr
  - Pokud registry nestačí – do zásobníku
  - Funkce s PPP (např. printf() – výhradně zásobník)
- Návratová hodnota:
  - 1B do r24 (r25 zůstane volný – zarovnávání)
  - 2B do r25:r24 (LSB první – jako argumenty)
  - 4B do r25 – r22
  - 8B do r25 – r18
- Příklad volání funkce:

```
int funkce(char p1, int p2)
{
      return p1+p2;
}
// uložení parametrů (v pointeru Y) do registrů
LDD     R22,Y+4     // b
LDD     R23,Y+5
LDD     R24,Y+1     // a (note: zarovnávání)
CALL    0x00000047  // volání fce
STD     Y+3,R25     // vyzvednutí návratové hodnoty
STD     Y+2,R24
```

Kombinace AVR-GCC – assembler

- Možnosti:
  - In-line assembler = instrukce přímo v C-kódu
  - Samostatné funkce v asm

Inline assembler

- = využití instrukcí procesoru v C kódu
- Makro `asm volatile(asm-template : output-operand-list : input-operand-list : clobber-list);`
  - chová se cca jako funkce s parametry `input-operand-list` a vracenými hodnotami v `output-operand-list` – funkci volá GCC. Kód je v `asm-template`

- `clobber-list` = seznam registrů přepisovaných (ručně programátorem) assemblerem (info pro překladač)
- kromě `asm-template` lze ostatní vynechat

- U GCC překladačů velmi nepřehledné – jen ukázky
- kód nepodléhá optimalizacím
- Instrukce bez parametrů

```
asm volatile ("nop");
```

- Složitější příklad – náhrada „světelného hada“ viz. výše

```
ISR(TIMER1_OVF_vect)
{
        asm volatile (
                "tst %0             \n\t"
                "brne hop           \n\t"
                "sec                \n\t"
                "hop: rol %0        \n\t"
                "out %2, %0"
                : "=r" (Had) : "0" (Had), "M" (_SFR_IO_ADDR (PORTB))
        );
}
```

%0 %1 %2

  - `\n\t` nutné → v asm 1 instrukce/řádek
  - GCC automaticky kopíruje `Had` do registru `%0` a zpět (`=r`)
  - Bude přeloženo jako:

```
LDS     R24,0x0060
TST     R24
BRNE    PC+0x02
SEC
ROL     R24
OUT     0x18,R24
STS     0x0060,R24
LDS     R24,0x0060
```

- Podrobně na http://www.nongnu.org/avr-libc/user-manual/inline_asm.html

Funkce v ASM volané z C kódu

- Nutno v samostatných souborech (musí být *.S) – jsou překládány GNU-assemblerem (GAS)
  - výstup = standardní object file a pak připojeny k projektu linkerem
- Dokumentace viz. http://www.gnu.org/software/binutils/manual/gas-2.9.1/
- Příklad – funkce:

```
int Secti(char p1, int p2)
{
        PORTB = p1;
        return p1 + p2 + 10;
}
```

  - Soubor neco.c

```
extern int Secti(char p1, int p2);     // definovana v AsmFunkce.S

int main (void)
{
        char a = 15;
        int c;
```

```
        c = Secti(a, 384);
        for(;;);     // nekonecna smycka
        return 0;
}
```

- Soubor AsmFunkce.S

```
#include "avr/io.h"

workreg = 16 // = .def workreg = r16

.global Secti
.func Secti
// realizuje p1(char) + p2(int) + 10, na port B zapise p1, vraci int;
// p1 bude v r25 (0) a r24
// p2 bude v r23 (HIGH(p2)) a r22 (LOW(p2))
// vysledek musi byt v r24 (LOW) a r25 (HIGH)
Secti:
        push   workreg      // r16 nutno uschovat, r18 az r27 nikoli
        // PORTB = p1
        out          _SFR_IO_ADDR(PORTB), 24
        // p1 + p2
        add          24, 22 // soucet nizsich B
        adc          25, 23 // soucet vyssich B
              // vysledek je tam, kde ma byt pro return
        // + 10
        ldi          workreg, 10
        add          24, workreg
        adc          25, 0
        pop workreg
        ret
.endfunc

// pripadne dalsi funkce
// .global DalsiFunkce
// .func ...
```

- Přístup ke globálním proměnným
  - Soubor neco.c

```
volatile unsigned char globProm;
```

  - Soubor AsmFunkce.S

```
#include "avr/io.h"

.extern globProm          // externe globalni promenna

.global Secti
.func Secti
// atd, mohu použít globProm
```