/elektro|myš.eu/

/ USART II |

/ Příjem po USARTu |

V minulém díle jsme si předvedli jeden ze způsobů odesílání zpráv USARTem. Využili jsme USB-UART převodník, který je součástí mEDBG na Xnano modulech. Na tento příklad přímo navážeme a doplníme ho o schopnost přijímat textové příkazy. Zatímco odesílání šlo snadno realizovat bez přerušení, příjem budete skoro vždy provádět s přerušením. Jistě se najdou velice jednoduché aplikace kdy může program tupě čekat na příchozí znaky, ale daleko častěji budete mít potřebu aby příjem zpráv probíhal současně s jinou prací. Koncepčně je postup úplně stejný jako v tutoriálu o starších AVR. V přerušení budeme ukládat znaky někam do paměti a až po přijetí celé zprávy ji budeme interpretovat. Málokdy budete v amatérské praxi potřebovat něco víc, takže příklad může posloužit i jako šablona do dalších projektů.

Protože tento díl nezaměřujeme na přerušení obecně, budeme s nimi zatím pracovat bez hlubších znalostí. Jen pro přehled napíšu, že moderní AVR mají dvouúrovňové přerušení, tzn máte možnost jednomu přerušení nastavit vyšší prioritu než ostatním. Prioritní přerušení pak může přerušit jiná probíhající přerušení. Co se USARTu týče tak ten má tři samostatné vektory přerušení.

- Přerušení od RXC (Receive Complete) - vyvolá se když USART přijme znak
- Přerušení od DRE (Data register Empty) - vyvolá se když máte místo ve vysílacím bufferu (tedy když můžete USARTu předhodit další data k odeslání)
- Přerušení od TXC (Transmit Complete) - vyvolá se když odvysíláte znak a v bufferu už další není (tedy když vysílání skončilo)

Existují ještě další dvě specifické události, které mohou vyvolat přerušení (obě vedou do vektoru RXC a musíte sami rozpoznat která z nich přerušení vyvolala). S nimi si teď ale nebudeme mást hlavu. Která přerušení chcete povolit si vybíráte v registru **CTRLA**. Stejně jako na starých AVR se přerušení globálně povolují i zakazují příkazy *sei()* a *cli()* a po startu jsou zakázaná (takže je musíte povolit). Názvy všech vektorů přerušení najdete jako obvykle v hlavičkovém souboru. To by nám zatím mělo stačit a můžeme se zabývat samotným příkladem.

/ Příklad |

Jak jsme již řekli, obohatíme náš program z předchozího dílu. Takže ke schopnosti odesílat zprávy pomocí *printf* přidáme schopnost zprávy přijímat a dekódovat. Zprávy budou dvojího typu. Buď pouze textové (v našem případě příkazy *"zapni"* a *"vypni"* LEDku na PB3) a nebo textové s argumentem (ve formátu *"x=nějaké číslo"*, například *"x=35"*). Příkaz bude ukončen jedním ze znaků '\n' (LF) nebo '\r' (CR). K dekódování příkazů s argumentem použijeme funkci *sscanf_P()*, která za nás převede příslušný text na číslo. Předesílám, že zabírá skoro stejné množství paměti jako *printf* a celý náš program se do paměti vleze jen tak tak (což netrápí ty, kteří pracují na "tinách" s větší pamětí). Je tedy jasné co po našem programu chceme, takže pojďme kódit. Pro přehlednost budu uvádět vždy jen příslušné kusy zdrojového kódu, takže kdo bude chtít celý, nechť si ho stáhne.

Nejprve si projdeme inicializaci. Čip taktujeme na 20MHz, tedy na plný výkon. LEDku máme připojenou na PB5 a proto si jej nastavíme jako výstup. UART z mEDBG je připojen na PA1(Tx) a PA2(Rx), nastavíme proto PA1 jako výstup, PA2 jako vstup a remapujeme USART na tyto dva piny (bez remapu vede USART na PB2 a PB3). Následně nastavíme baudrate (o němž byla řeč v předchozím díle). Pak povolíme příjmač i vysílač. Od tohoto okamžiku přebere USART kontrolu nad pinem Tx. Nakonec povolíme přerušení od příjmu a to bitem **RXCIE** (Receive Complet Interrupt Enable) v registru **CTRLA**. Pak už stačí povolit přerušení globálně (funkcí *sei()*) a je vše připraveno.

```
int main(void){
 clock_20MHz(); // taktujeme na 20MHz
 PORTB.DIRSET = PIN5_bm; // PB3 - výstup (LEDka)
 usart_init(); // konfigurace UARTu
 stdout = &mystdout; // přesměrujeme Printf na UART
 sei(); // globální povolení přerušení
// ... část kódu přeskočena

// ... část kódu přeskočena
void usart_init(void){
```

```
 PORTA.OUTSET = PIN1_bm;  // log.1 PA1 (Tx) - neutrální hodnota na UARTu
 PORTA.DIRSET = PIN1_bm; // PA1 (Tx) - je výstup
 PORTA.DIRCLR = PIN2_bm; // PA2 (Rx) - je vstup
 PORTMUX.CTRLB = PORTMUX_USART0_bm; // Remapujeme USART (Tx na PA1 a Rx na PA2)
 USART0.BAUD = BAUDVAL; // nastavit baudrate
 USART0.CTRLB |= USART_TXEN_bm | USART_RXEN_bm; // zapnout přijímač a vysílač
 USART0.CTRLA |= USART_RXCIE_bm; // zapnout přerušení od Rx
}
```

Jakmile dorazí celý znak, vyvolá se rutina přerušení *USART0_RXC_vect*. Úkolem rutiny bude znak zkontrolovat a případně uložit. Jako první vytáhneme přijatý znak z registru **RXDATAL** abychom co nejdříve uvolnili místo pro další data. V registru **RXDATAH** máme k dispozici chybové vlajky. Mezi nimi je například "buffer overflow" (overrun), která značí že jsme nestihli vyčíst předchozí znak nebo "Frame error", která nás informuje o chybném formátu zprávy. My si teď dovolíme chyby příjmu neřešit neboť riziko, že nastanou je v našem případě zanedbatelné. Pokud je budete chtít kontrolovat, musíte si vlajky vyčíst před čtením přijatého znaku. Tedy nejprve **RXDATAH** a až pak **RXDATAL** (výjimkou je 9bit komunikace kdy je pořadí opačné). Teď když máme znak vyčtený, je v přijímacím bufferu místo na další a my máme mezi tím čas data zpracovat. Nejprve znak otestujeme na '\n' a '\r', tedy jestli to není "ukončovací" znak naší zprávy. Pokud není, zkontrolujeme ještě jestli máme znaky kam ukládat. Ukládáme je do pole *rx_string* (přístupného jen rutině přerušení) a přirozeně nechceme aby nám přeteklo ;) Teprve pokud ani jedna z výjimek nenastala si znak uložíme. K orientaci v poli nám slouží počítadlo znaků *cnt*. Pokud přijmeme ukončovací znak (nebo nám zpráva kompletně zaplní pole), budeme to chápat jako konec zprávy a zpracujeme ji. Nejprve celou zprávu zkopírujeme do pole *command* (které je přístupné zbytku programu) abychom s ní mohli později pracovat a zároveň přijímat další znaky. Kopírování můžete provést forcyklem nebo třeba funkcí *memcpy()*. Protože se snažíme rutinu udržet co nejkratší, kopírujeme jen tolik znaků kolik jsme přijali. Za poslední přijatý znak uložíme ještě hodnotu 0 a tím se z našeho pole (*command*) stane řetězec. Pomocí proměnné *num_command* dáme hlavní smyčce vědět, že může příkaz zpracovat. Přirozeně nezapomeneme vynulovat počítadlo znaků (*cnt*), čímž se připravíme na příjem další zprávy. Protože některé terminálové programy odesílají klávesou "enter" dvojici znaků "\r\n" děláme před kopírováním zprávy ještě test zda je zpráva "neprázdná", tedy jestli obsahuje alespoň jeden znak. Pokud je prázdná, tak nenese žádnou informaci, můžeme ji směle ignorovat a vynulováním *cnt* se opět připravit na příjem nové.

```
ISR(USART0_RXC_vect){
 static char rx_string[MAX_STRLEN]; // zde dočasně ukládáme přicházející znaky
 static uint8_t cnt = 0; // počítadlo přijatých znaků
 char znak; // pomocná proměnná
 //err = USART0.RXDATAH; // případná kontrola chyb
 znak = USART0.RXDATAL; // vyčteme přijatý znak
 // není to "konec příkazu" a máme ještě kam data ukládat ? ...
 if((znak!='\r') && (znak!='\n') && (cnt < (MAX_STRLEN-1))){
  rx_string[cnt] = znak; // ulož znak do pole
  cnt++; // inkrementuj počítadlo znaků
 }
 // ... jinak je příkaz celý a je čas ho zpracovat
 else{
  if(cnt>0){ // pokud není příkaz prázdný
   memcpy(command,rx_string, cnt); // zkopírujeme řetězec do pole "command"
   command[cnt]= '\0'; // připojíme na konec ukončovací znak 0
   num_commands++; // dáme vědět do "main" že je tu nový kompletní příkaz
  }
  cnt = 0; // nulujeme počítadlo znaků (abychom mohli přijímat další příkazy)
 }
}
```

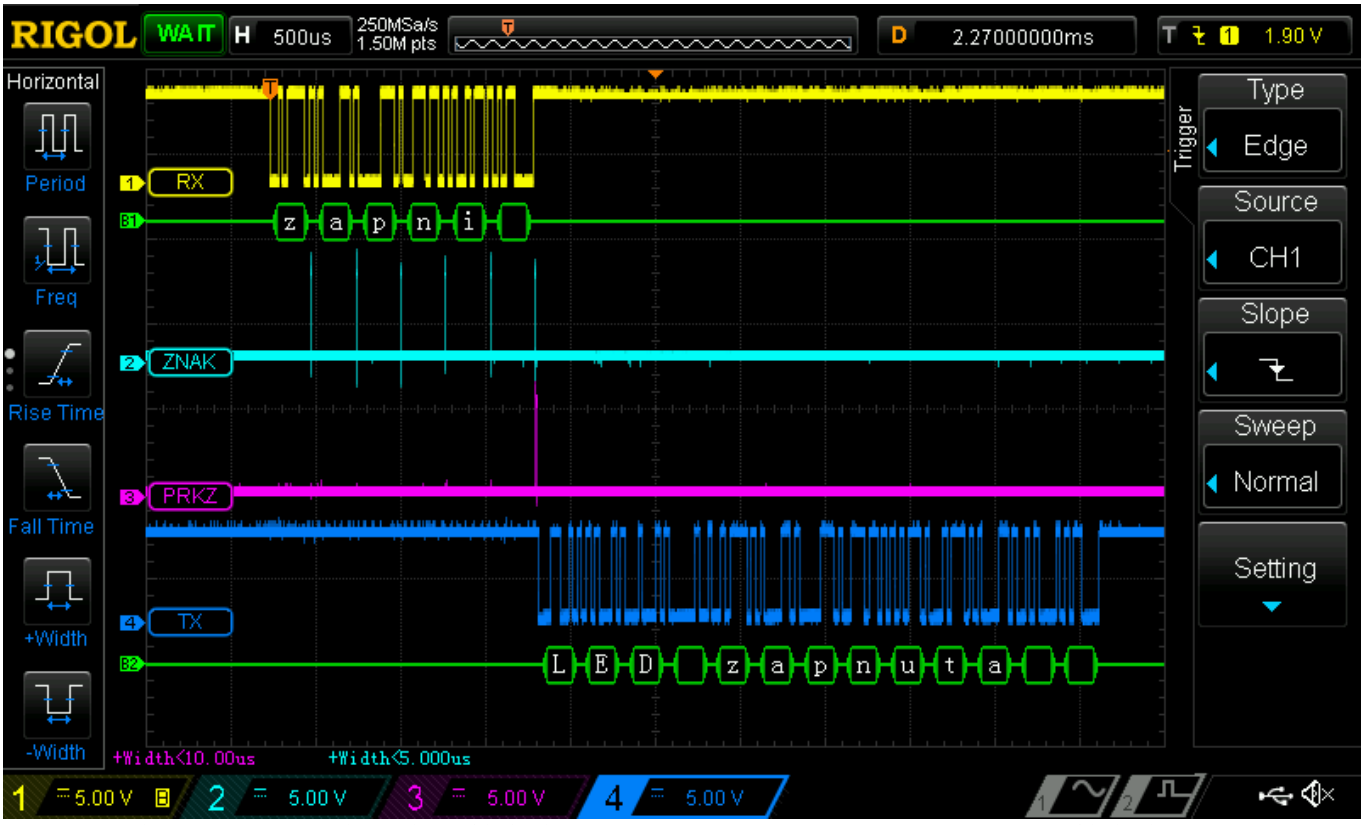
Rutinu přerušení se snažíme držet co nejkratší. Její délka nám může limitovat maximální přenosovou rychlost. Naše rutina trvá v nejhorším případě (přijetí posledního znaku nejdelší možné zprávy) 7.3us. Příjem a uložení jednoho běžného znaku zabere něco málo přes 1us. Co to pro nás znamená rozebereme v závěru tutoriálu.

Nakonec musíme zprávu (příkaz) dekódovat a nějak zužitkovat. Náš program tedy čas od času (neustále) testuje proměnnou *num_commands* a pokud je nastavená, začne řetězec *command* interpretovat. Nejprve otestuje shodu s řetězcem *"zapni"* a *"vypni"*. K tomu využívá funkce *strncmp()*. Pokud shoda nenastane, testuje program zda příkaz vyhovuje masce *"x=%u"* a pokud ano načte číselný argument do dočasné proměnné a odpoví jaká je jeho hodnota. Navíc vypíše i dvojnásobek jako důkaz, že ho interpretoval jako číslo. Pokud ani jedna shoda nenastane, je příkaz nesmyslný a program to dá vědět uživateli příslušnou hláškou. Po zpracování, musíme ještě vynulovat proměnnou *num_commands* abychom mohli zpracovat další příkazy.

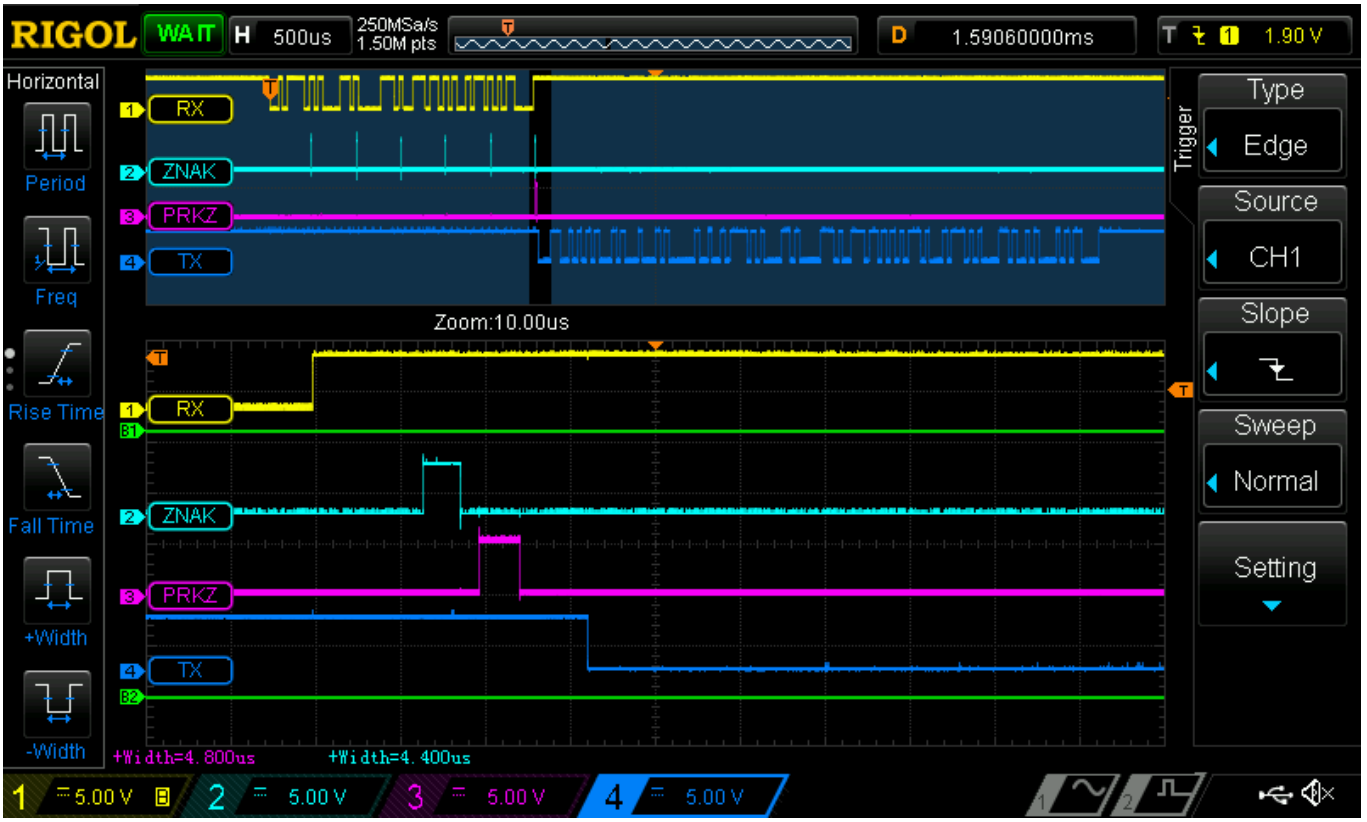
```
 while (1){
  if(num_commands){ // dorazil nový příkaz ?
   // poslal uživatel příkaz "zapni" ?
    if(strncmp(command,"zapni",MAX_STRLEN)==0){
     LED_ON;  // zapni LED a odpověz
    printf_P(PSTR("LED zapnuta\n\r"));
    }
    // poslal uživatel příkaz "vypni" ?
```

```
    else if(strncmp(command,"vypni",MAX_STRLEN)==0){
     LED_OFF;  // zhasni LED a odpověz
     printf_P(PSTR("LED vypnuta\n\r"));
    }
    // poslal uživatel číslo ?
   else if(sscanf_P(command,PSTR("x=%u"),&tmp) > 0){
    // odpověz a předveď že je to opravdu číslo
    printf_P(PSTR("x=%u, 2x=%u\n\r"),tmp, 2*tmp);
   }
   // jiný příkaz...
   else{
    printf_P(PSTR("nerozumim\n\r"));
   }
   num_commands=0; // příkaz zpracován, čekáme na nový
  }
 }
```

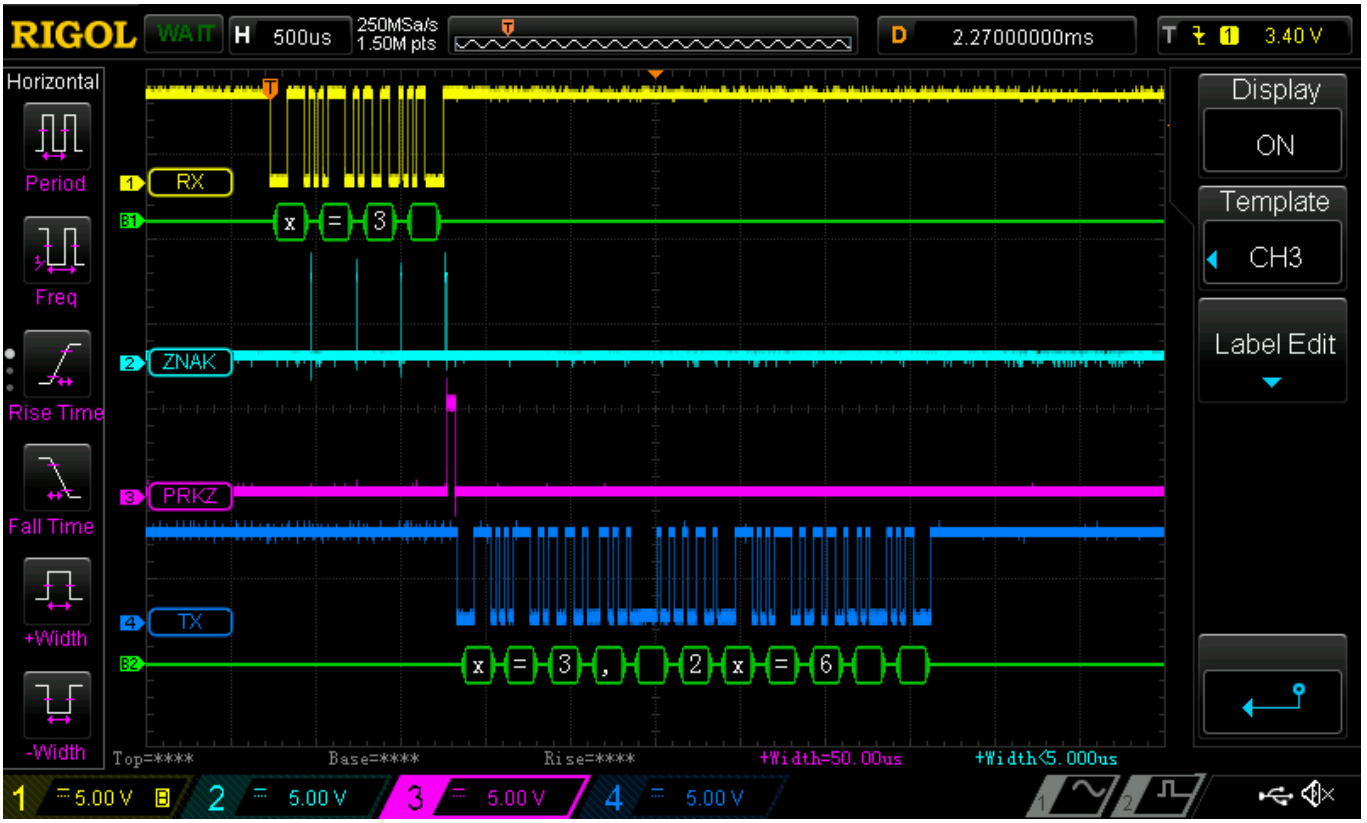
Komunikace by měla probíhat vždy tak, že uživatel nebo program v PC pošle příkaz a počká na odpověď. Tak má záruku, že mikrokontrolér svou práci dokončil a je schopen přijmout a zpracovat další příkaz. Uživatel který ručně píše zprávy do terminálu žádné nebezpečí pro naši aplikaci nepředstavuje. Počítačový program je ale v principu schopen poslat příkazy těsně za sebou, proto je obecně nutné aby čekal na zprávu o dokončení příkazu. Abychom měli představu v jakém časovém měřítku se pohybujeme, prohlédneme si následující oscilogramy.



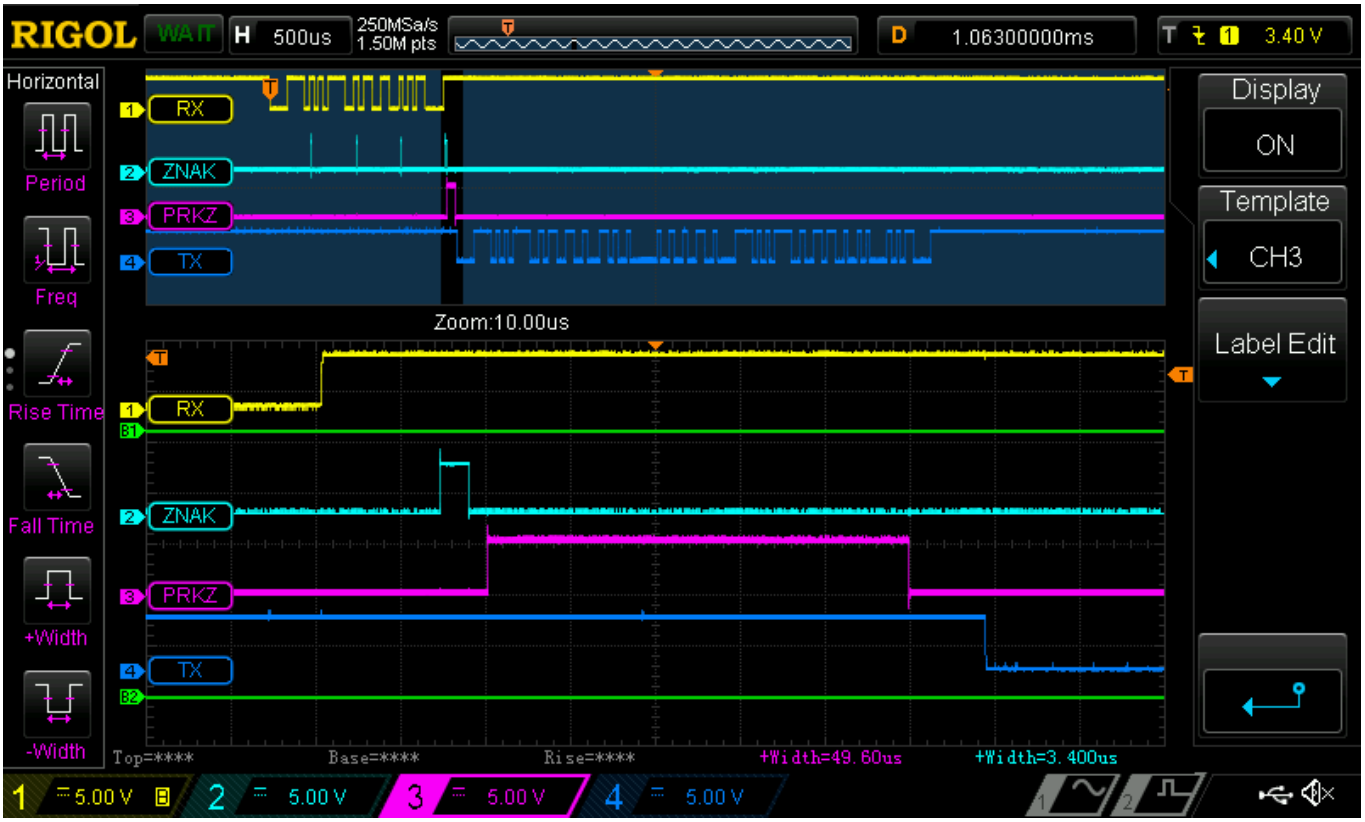
Příjem příkazu "zapni". Světle modrý průběh ukazuje čas trávený v rutině přerušení, červený průběh ukazuje čas potřebný k dekódování příkazu. Z následujícího oscilogramu bude dobře patrné, že dekódování je řádově rychlejší než samotná odpověď do PC (Tmavě modrý průběh)



Na obrázku je vidět, že po přijetí posledního znaku stráví program v rutině přerušení přibližně 4.5us a podobný čas stráví i dekódováním zprávy.



Příjem příkazu s číselným argumentem ("x=3"). Všimněte si že dekódování zprávy trvá o poznání déle (přibližně 50us), ale pořád je řádově kratší jak odpověď do PC.



Detail na dobu potřebnou ke zpracování příkazu (červený průběh). Světle modrá stopa naznačuje že příjem posledního znaku trvá o něco kratší dobu než v případě zprávy "zapni". To je logické, neboť je zpráva o něco kratší.

Jak jsem psal dříve, příjem a uložení jednoho znaku trvá přibližně 1.2us, teoreticky je tedy možné aby náš program přijímal zprávy s baudrate okolo 800kb/s. Přirozeně jen s podmínkou, že nám po posledním znaku zprávy nechá dost času a že zprávy budou korektní (což by mělo být zajištěno). To jsou ale jen detaily, které v drtivé většině amatérských aplikací nebudete řešit. Jen pro rychlý přehled velmi stručně okomentuji použité funkce:

- *printf_P* - modifikovaná verze printf využívající formátovací řetězec z paměti flash (šetří RAM)
- *sscanf_P* - modifikovaná verze sscanf využívající formátovací řetězec z paměti flash (šetří RAM) - něco jako opak printf. Porovnává řetězec s formátovacím řetězcem a pokouší se z něj načíst číselné a jiné datové typy.
- *strncmp* - porovnává dva řetězce s omezením na maximální počet znaků (to je to "n"). Vrací 0 pokud jsou řetězce shodné.
- *memcpy* - kopíruje jedno pole do druhého

| Závěr /

Celý program je na hraně použitelnosti na Attiny416. Zabírá totiž něco přes 92% paměti flash. Na užitečnou činnost by nám v tomto případě zbývalo pár stovek bytů. Je tedy třeba vnímat příklad jen jako demonstrační. Praktické použití může najít jen na "Tinách" s větší pamětí (Attiny816, 1614, 1616 atd.) nebo na moderních Atmega. Snad jste se v relativně komplexním příkladě neztratili a doufám, že jsem vás zase trochu přesvědčil, že přechod na moderní AVR není tak úplně špatný nápad. Těším se na shledanou u dalších dílů.

| Odkazy /

- Attiny416 Datasheet
- Attiny416 Xnano modul
- Tutoriál o printf a scanf
- Přehled o nabídce funkcí ze string.h

Home
| V1.01 2.1.2019 /
| By Michal Dudka (m.dudka@seznam.cz) /