**UART**

UNIVERSAL ASYNCHRONOUS RECEIVER TRANSMITTER

je periferie, která vysílá a přijímá data v sériovém tvaru v asynchronním módu. V klidu je tedy na spojovacích drátech log. 1, přenos začne poklesem do log. 0 – START bit, následuje osm přenášených bitů LSB first, dále log. 1 – STOP bit, a poté je přenosový kanál opět v klidu – log. 1. Každý znak ( 8 bitů) je tedy přenášen samostatně a je ohraničen START a STOP bitem.

Situaci vidíme na následujícím obrázku



Všechny bity trvají stejnou dobu – na obrázku doba **a** . Převrácená hodnota **a** je přenosová rychlost **v** [Bd] . Bity 0 – 7 jsou zobrazeny dvěma čarami, v log. 0 i 1. To je proto, že mohou nabývat obou těchto hodnot podle vysílaného čísla.

Náš procesor má dva obvody UART. Pro studium následující kapitoly budeme potřebovat dva datasheety : náš běžný 33EV32.pdf a UART.pdf, který nalezneme v adresáři ../family ( vzhledem k tomu, že probíráte Linux, je doufám význam ../ jasný ).

V datasheetu 33EV32 máme UART na straně 247, kapitola 21.0 . Na obrázku FIGURE 21-1: je základní blokové schéma modulu.

Baud Rate Generator – zajišťuje požadovanou přenosovou rychlost, tedy délku a jednotlivých bitů

IrDA® - doplněk pro vysílání infrared signálů – např. dálkový ovladač

Hardware Flow Control – ovládá a vyrábí signály RTS, CTS ( pokud netušíte, co to je, tak znovu 2.ročník, první pololetí )

UARTx Receiver - přijímač

UARTx Transmitter – vysílač

Náš procesor má dva moduly UARTu, takže za x dosazujeme 1 nebo 2 .

**Vstupy a výstupy UARTu je nutno přiřadit pomocí PPS . Připomínám, že je nejdříve nutno nastavit PPS, a teprve poté nastavit registry UARTu.**

V datasheetu UART.pdf je popsán detaelně modul UART, jsou zde popsány všechny možnosti pro všechny procesory z rodiny 33EV. Náš procesor neumí úplně všechno, co je v tom to datasheetu popsáno. Na ovládání řídících registrů se proto podívejte do datasheetu 33EV32.pdf . Datasheet UART.pdf je cenný pro pochopení celkové funkce modulu.

**Baud Rate generator**

je blok, který určuje přenosovou rychlost, tedy dobu trvání vysílaného a přijímaného bitu **a** . Vysílač i přijímač mají stejný generátor, to znamená, že vysíláme i přijímáme na stejné rychlosti.

Základem generátoru je čítač, který funguje stejně jako kterýkoli čítač TMR procesoru. Je ovládán registrem UxBRG, který určuje dělící poměr čítače stejně jako registr PR u čítačů TMR. Vstupní hodinový signál je roven frekvenci instrukcí, tedy FOSC / 2 . Dělící poměr je dán číslem v registru UxBRG zvětšeném o 1 ( Pokud je UxBRG == 2 , dělíme třemi ).

Jeden vysílaný bit trvá buď 16 period signálu, který vychází z BaudRateGenerátoru, nebo 4 periody signálu. To nastavujeme bitem BRGH v registru UxMODE . Platí:

16 period na vysílaný bit pro BRGH==0

4 periody na vysílaný bit pro BRGH==1



Vše nejlépe osvětlí příklad.

Předpokládejme, že máme zapnut vnitřní FRC oscilátor s kmitočtem 7.37MHz. Chceme vysílat rychlostí 2400Bd. Zvolíme BRGH==0 , tedy 16 period na 1 vysílaný bit.

Výstupní signál z BaudRate Generátoru musí mít kmitočet 16x2400 = 38400Hz. Vstupní kmitočet do BaudRate Generátoru je 7.37 / 2 = 3,685MHz. Generátor tedy dělí v poměru 3685000 / 38400 = 96 . Do registru UxBRG tedy zapíšeme číslo 95.

Řídící registry UARTu

najdeme v datasheetu 33EV32.pdf počínaje stranou 249 .

**Registr UxMODE** strana 249 datasheetu 33EV32.pdf

Připravil jsem pro vás výtah z datasheetu. Pokud je zde v registru 0 nebo 1, zapíšeme ji do registru. Pokud je tam B, nastavíme bit podle našeho uvážení.



Na význam všech bitů se podívejte do datasheetu, zde jsou jen některé poznámky

**UARTEN** zapnutí celého UARTu. Nejdříve nastavíme registr UxMODE s bitem UARTEN==0 , pak nastavíme úplně všechny další registry, na konci vše zapneme instrukcí

bset UxMODE, #UARTEN . Jediný bit, který je nutno nastavit po bitu UARTEN, je bit pro zapnutí vysílání UTXEN v registru UxSTA

**IREN** - feature pro použití v infrared vysílači a přijímači. V zásadě to se signálem během vysílání log. 1 bliká – dále viz datasheet

**UEN<1:0>** bity zapínají ještě dále signály RTS a CTS. Nebudeme je používat, proto 00 . Pokud to chcete, nastudujte si to v datasheetu

**LPBACK** – loopback, smyčka. Data, která vysíláme, se hned přesunou do vysílací části, je to stejné, jako bychom spojili drátem piny TxD a RxD. Hodí se pro testovací účely .

**ABAUD** – UART umí sám zjistit rychlost znaků, které přijímá. Je to ale poněkud složitější než jen nastavit tento bit do 1, opět viz datasheet.

**URXINV** invertuje přijímaná data ( tedy umožní, aby „klid“ byla log. 0, ne 1 )

**PDSEL<1:0>** - zapnutí paritního bitu

00 – nepřidáváme nic, UART vysílá 8 bitů

11 – UART vysílá 9 bitů , které mu dodáme do vysílacího registru

10 – lichá parita. Programátor dodá do vysílacího registru osm bitů, a UART k nim přidá další jeden paritní bit. Lichá parita znamená buď – nebo:

1. pokud je počet jedniček ve vysílaných osmi bitech lichý, nastaví se paritní bit na 1
2. počet jedniček ve vysílaných devíti bitech ( tedy osm našich + jeden paritní ) je lichý.

Správně jste poznali, že možnosti 1 a 2 dají zcela opačnou hodnotu paritního bitu. Každá firma, která paritu používá, si je zcela jista, že právě její výklad je jediný možný a jediný správný.

**STSEL** dva stop bity znamenají prostě to, že čas pro log.1, která je na místě STOP bitu, je dvakrát delší ( prostě se počká déle před započnutím vysílání dalšího znaku )

**Vysílání znaků**

Díváme se na stranu 15 datasheetu UART.pdf. Znak se UARTem vyšle tak, že ho zapíšeme do vysílacího registru UxTXREG. Znak je hned předán do vysílacího posuvného registru TSR a je vysílán na vývod UxTX ( tento vývod není pacička procesoru, tu je nutno nastavit pomocí PPS ). Vysílací registr UxTXREG ale není jeden registr, je to čtyřúrovňová fronta FIFO. To znamená, že do UxTXREG můžeme zapsat rychle za sebou pět znaků, aniž bychom o ně přišli, budou v pořádku odvysílány ( pět znaků – jeden hned propadne do TSR a je vysílán, další čtyři se uloží do fronty v TXREG ) . Po odvysílání znaku ( nebo znaků ) volá vysílač UARTu přerušení nastavením bitu U1TXIF nebo U2TXIF ( máme dva UARTy ). UART sám doplní START a STOP bity a eventuálně paritní bit.

**Přijímání znaků**

Díváme se na stranu 23 datasheetu UART.pdf . Data přicházejí po vstupu UxRX ( již potřetí upozorňuji, že to není pacička procesoru, je ji nutno nastavit pomocí PPS ). Jsou přijímána v přijímacím posuvném registru UxRSR. Poté, co se do registru „došoupe“ celý znak, je přenesen do přijímacího registru UxRXREG. Z něj si pak přijatá data přečte programátor. RXREG ale není jeden registr, ale čtyřúrovňová fronta FIFO. Znamená to, že UART může přijmout čtyři znaky a teprve potom si je programátor může přečíst, aniž by došlo ke ztrátě přijatého znaku.

Při příjmu může dojít k následujícím chybám:

1. Framming Error – v místě, kde přijímač očekává STOP bit, tedy log. 1, přijde log. 0 . K této chybě dojde při poruše linky mezi vysílačem a přijímačem nebo při nesouhlasu přijímací a vysílací rychlosti. Tuto chybu indikuje bit FERR v registru UxSTA.
2. Parity Error – přišel znak se špatnou paritou. Chyba je indikována samozřejmě pouze tehdy, když jsme nějakou paritu nastavili pomocí bitů PDSEL registru UXMODE. Chybu indikuje bit PERR v registru UxSTA
3. Overrun Error – přišly čtyři znaky, programátor je nepřečetl a nyní přišel další znak. To znamená, že některý znak je ztracen. Chybu indikuje bit OERR v registru UxSTA. Pokud k této chybě dojde, musíme bit OERR vynulovat. Tím ale také ztratíme veškeré nepřečtené přijaté znaky ( přečtěte si to podrobně v datasheetu ). **Při vzniku chyby OERR přestává přijímač přijímat,** dokud bit OERR nevynulujeme.

Při vzniku jakékoli chyby zavolá přijímač UARTu přerušení ( ne to normální přijímací, ale ještě jiné přerušení, které slouží pro obsluhu chyby )

Bity FERR a PERR patří vždy k tomu znaku, který jsme ještě nepřečetli z RXREG. Po přečtení znaku z RXREG se FERR a PERR nastaví na hodnoty, které odpovídají dalšímu připravenému znaku, tedy tomu, který budeme číst z RXREG .

Po přijetí znaku ( nebo znaků ) generuje přijímač požadavek na přerušení nastavením bitu U1RXIF nebo U2RXIF.

**UxSTA:** UARTx STATUS AND CONTROL REGISTER strana 251 datasheetu 33EV32.pdf

v registru jsme doplnili písmenko R – Read Only . Tento bit je tedy pouze ke čtení.



Přerušení od vysílače ovládají bity 15 a 13, tedy mezi nimi je jeden bit na něco jiného ( negace vysílaných bitů )

**Přerušení od UARTu**

Modul generuje přerušení poté, co přijal znak nebo znaky (přerušení od přijímače). Další přerušení modul generuje, pokud nemá co vysílat ( přerušení od vysílače). Požadavkový bit najdeme jako vždy v registru IFS, přerušení je nutno povolit v registru IEC . V obslužném programu pro přerušení je nutno vynulovat bit IF ( U1TXIF U1RXIF U2TXIF U2RXIF ). Podívejte se, v kterých registrech bity jsou. Povolovací bity jsou v odpovídajících registrech IEC .

Další přerušení volá UART, pokud dojde k chybě ( U2EIF U1EIF ) .

Bity si najděte, do zápisníčků si napište, v kterých registrech se nalézají, v linker script file si najděte, jak se přerušení nazývá. Procesor už programujeme půl roku, tak si to najdete sami.

**Příklad**

sestavte program, který bude na pinu 11 procesoru desetkrát za 1 sec vysílat v sériovém kódu písmeno A rychlostí 9600Bd. Použijte UART2 . Předpokládejme FRC oscilátor v základním módu.

Až budete realizovat vlastní program, doporučuji postupovat přesně tak, jak jsem to napsal zde, tedy nejprve „uchodit“ vysílání jednoho písmena a pak pokračovat dál. UART je dost složitý, neuděláte to napoprvé.

Příklad je v programu UART01.s

Nejdříve si nastavíme TRIS, ANSEL, a hlavně PPS

Následně nastavíme Baud Rate Generator. Zvolíme si dejme tomu BRGH = 0, tedy 16 period na vysílaný bit. Pokud nás netlačí jiné okolnosti, volte to takhle, protože pak má UART 16 period hodin na zpracování bitu a pracuje se mu lépe.

Kmitočet, který potřebujeme z BRG, je tedy 9600 x 16 = 110400 Hz. Vstup do BRG je 7.37/2 = 3685000 Hz. BRG tedy dělí 3685000/110400 = 33,38 . To je špatné číslo, musíme ho zaokrouhlit o 0.38 , a to už je docela dost. Proto nastavíme bit BRGH = 1, tedy 4 periody na vysílaný bit

Kmitočet, který potřebujeme z BRG, je tedy 9600 x 4 = 38400 Hz. Vstup do BRG je 7.37/2 = 3685000 Hz.

BRG tedy dělí 3685000/38400 = 95,963 . To už je hezké číslo. Do registru U2BRG uložíme číslo 95 ( o 1 méně než je požadovaný dělící poměr ).

Nastavíme registry U2MODE a U2STA. U registru U2MODE necháme bit 15 v 0. Úplně na konec nastavíme bit UARTEN do 1 a ještě potom bit UTXEN do 1

Nastavíme čítač TMR2 ( vybral jsem si ) tak, aby generoval přerušení 10 x za 1 sec. V obslužném programu pro přerušení od TMR2 budeme vysílat písmeno A, tedy do vysílacího registru U2TXREG zapíšeme ASCII kód písmena A .

Povolíme přerušení od UARTu a od čítače

Povolíme veškerá přerušení bitem GIE.

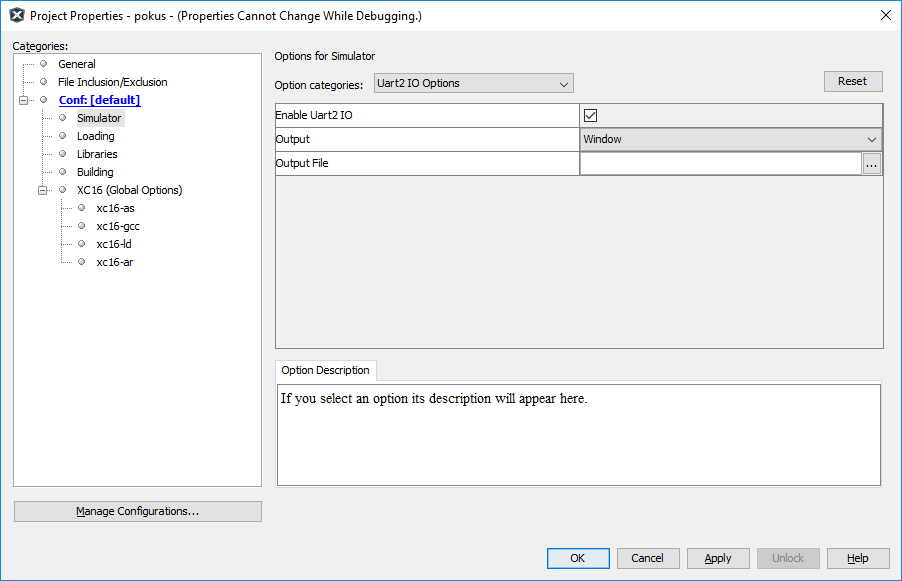
A je to.

V programu si dáme breakpoint do přerušení od TMR2 a mačkáme F5. Samozřejmě, rádi bychom viděli, co opravdu UART dělá.

**Simulace UARTu - vysílání**

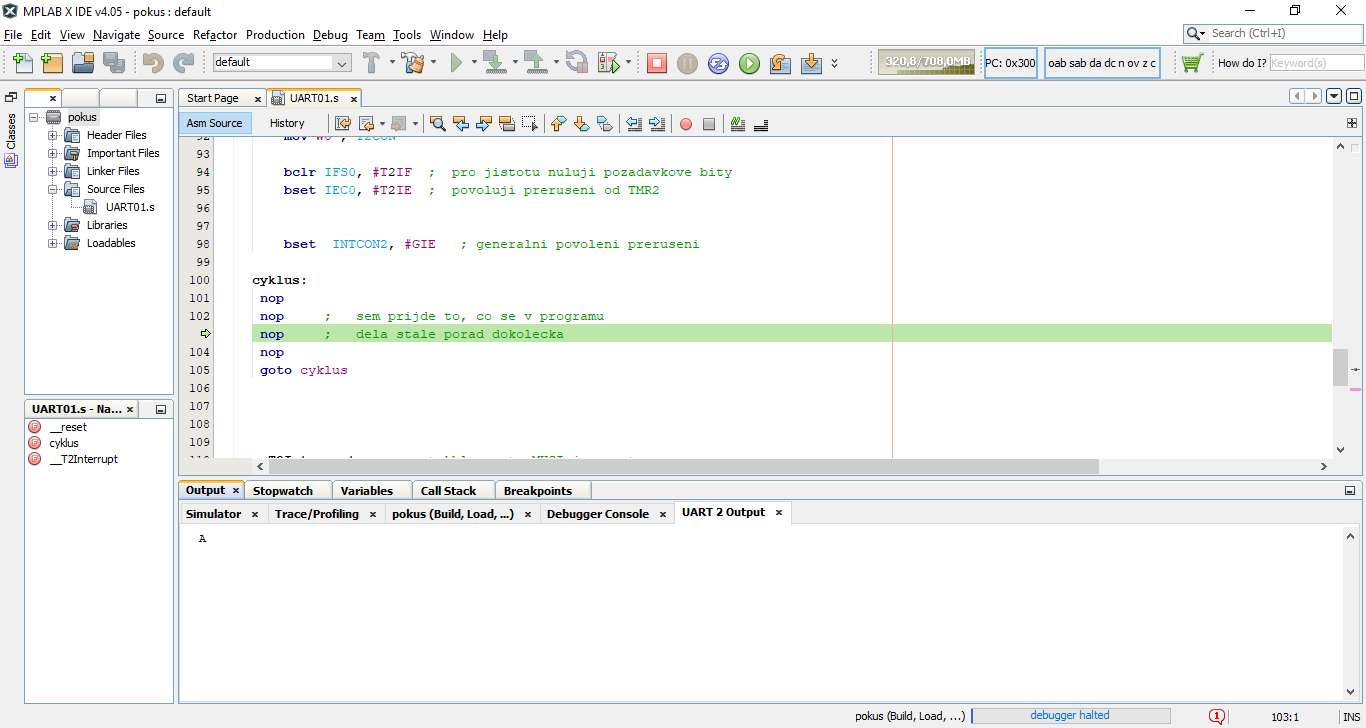
Simulaci výstupu UARTu je nejdříve nutno zapnout.

Cvakneme pravým tlačítkem myši na název projektu, dole properties, dále Simulator – UART options



No a zaškrtneme Enable Uart IO – prostě to zapneme

Musíme znovu pustit debugger . Dole se objeví další okno UART2 output a v něm to, co UART vysílá. Okno se objeví až poté, co UART vyšle první znak .



**Jak poznáme, jestli to opravdu funguje.**

Naprogramujeme si to do bastldesky. Funkci přerušení od čítače poznáme snadno – dáme diodu na RA4 , a vidíme to. Dále si dáme diodu na výstup UARTu. Vidíme kontinuální log. 1, protože pulsy do log. 0 při vysílání jsou tak krátké, že je nevidíme. Řešení je jednoduché. Čítač nastavíme na výrazně delší periodu a taktéž rychlost UARTu nastavíme výrazně menší. Pak uvidíme krátké bliknutí při vysílání. Na vaší laskavé úvaze ponechávám, co znamená „výrazně menší“. Prostě tak, abyste to viděli, a aby UART byl schopen odvysílat znak mezi přerušeními od čítače.

Ale postoupíme dál. Vrátíme rychlosti na původní hodnoty a na PICKITu si pustíme UART\_TOOL . Nyní je tu ovšem problém. Náš UART v procesoru má výstup na pinu 11, UART\_TOOL má vstup na pinu ……… ( najděte si sami. Pustíme si UART\_TOOL, podíváme se na obrázek, najdeme drát, který vede **DO** UartToolu. Pak se podíváme na schéma bastldesky a najdeme si, na který pin procesoru tento drát pokračuje. Pozor ! číslování kontaktů na schématu je matoucí ! Postupujte tak, že si najdete GND, potom „ten drát, co je vedle země“ , „ten drát, co je o jednu pozici od země“ atd. atd. atd. Do svého zápisníčku si napište, kam vstup vede, bude se vám to mnohokrát hodit ). Všechny piny procesoru máme vyvedeny na konektor, takže prostě spojíme drátkem pin 11 a pin, na kterém je vstup UartToolu. No a krásně vidíme, jak nám UART\_TOOL píše písmenka A.

Ještě jedna poznámka: výstup z pinu 11 jsme připojili na nějaký jiný pin procesoru. To samozřejmě dosti nebezpečné. Ale u našeho programu jsou všechny ostatní piny vstupní, takže to nevadí. Pokud spojíte drátem dva výstupní piny, pak samozřejmě procesor hoří.

**Použití přerušení od vysílače UARTu**

máme v programu UART02.s Program vychází z UART.s, takže výstupní pin i rychlost jsou stejné. Program jednou za sekundu vyšle krátký nápis na UART. Nápis je zapsán v paměti programu pomocí PSV. Nápis končí bytem 0 , tím program pozná, že je už na konci nápisu. Psaní nápisu se odstartuje pomocí přerušení od čítače, který nastaví ukazovátko na začátek nápisu a povolí přerušení od UARTu. V přerušení od přijímače se zkontroluje, zda nejsme už na konci ( přečetli jsme 0 ), pokud ne, tak znak zapíšeme do TXBUF a inkrementujeme ukazovátko. Pokud jsme na konci nápisu, zakážeme další přerušení od vysílače UARTu .

**Přijímání dat pomocí UARTu**

máme v programu UART03.s

Navíc oproti předchozím programům má procesor vstup RxD pro UART2 na pinu 10.

Po přijetí znaku zavolá UART přerušení od přijímače. Znak přečteme z přijímacího registru

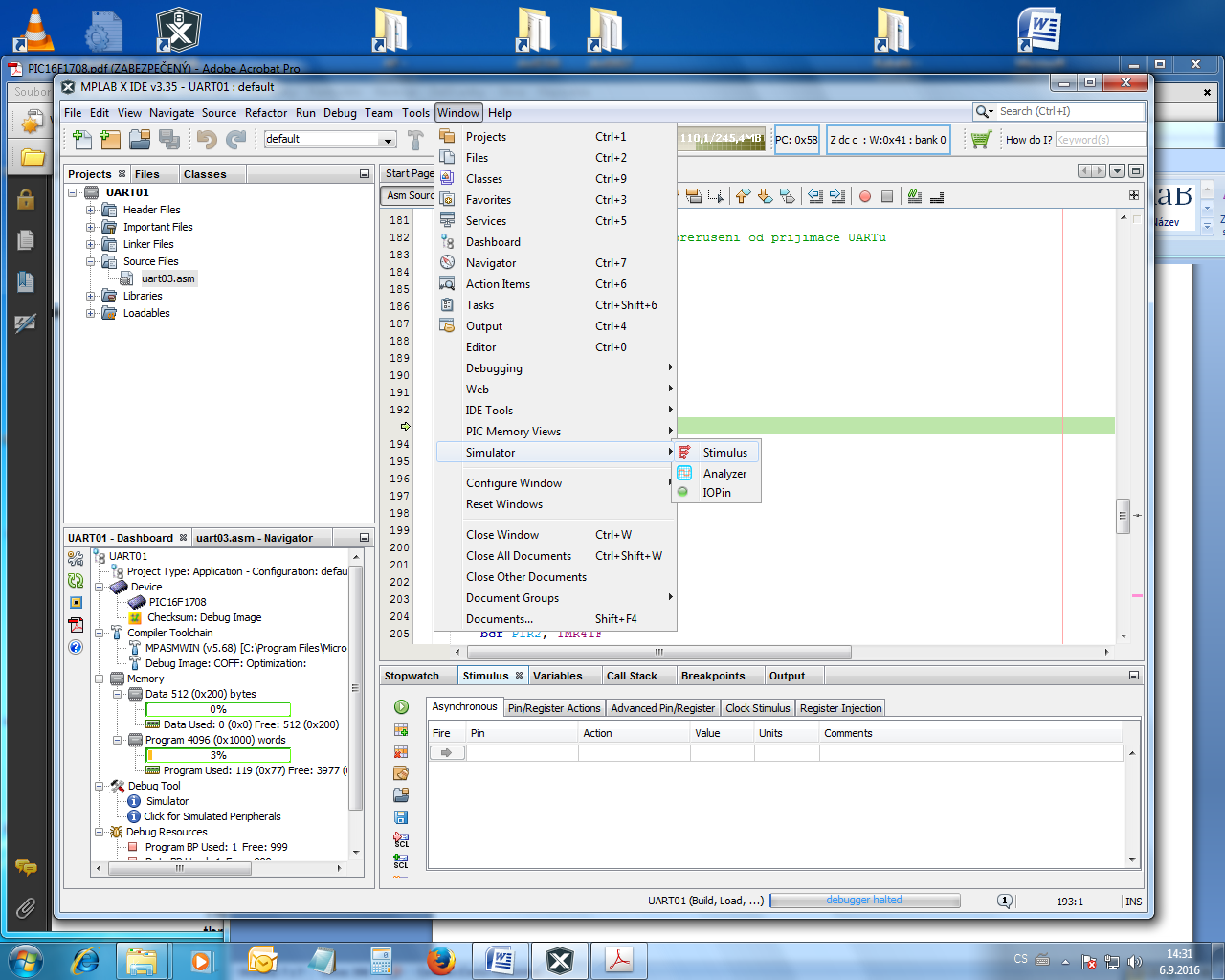
Naprogramoval jsem jenom takovou jednoduchou hrátku – procesor vyšle zpátky to, co přijal, poté znak s ASCII hodnotou o 1 větší, poté znak s ASCII hodnotou o 2 větší. Takže když do procesoru pošleme písmeno D , procesor odpoví DEF . Protože "The transmit buffer is 4 levels deep FIFO" , prostě zapíšeme za sebou do TXREG čtyři znaky a ono se to odvysílá. Očekáváme samozřejmě, že znaky do procesoru budete posílat pomalu, tedy stisknete klávesu, počkáte, stisknete druhou klávesu, počkáte atd. atd. Na PICKITu si vypněte Echo on, aby se vám v přijímacím okně nezobrazoval také vyslaný znak. Náš program havaruje při příliš rychlém vysílání znaků. To si můžete zkusit tak, že několik znaků napíšete do okénka send a pak je cvaknutím myši vyšlete rychle za sebou.

Pin 10 procesoru (vstup UARTu) spojíme s výstupem UARTu na PICKITu. JAko v minulém případě se i tady podíváte na schéma zapojení bastldesky a najdete si, na který pin je připojen výstup PICKITu. Tento výstup pak drátkem spojíte s pinem 10. A můžete se dívat, jak krásně procesor pracuje.

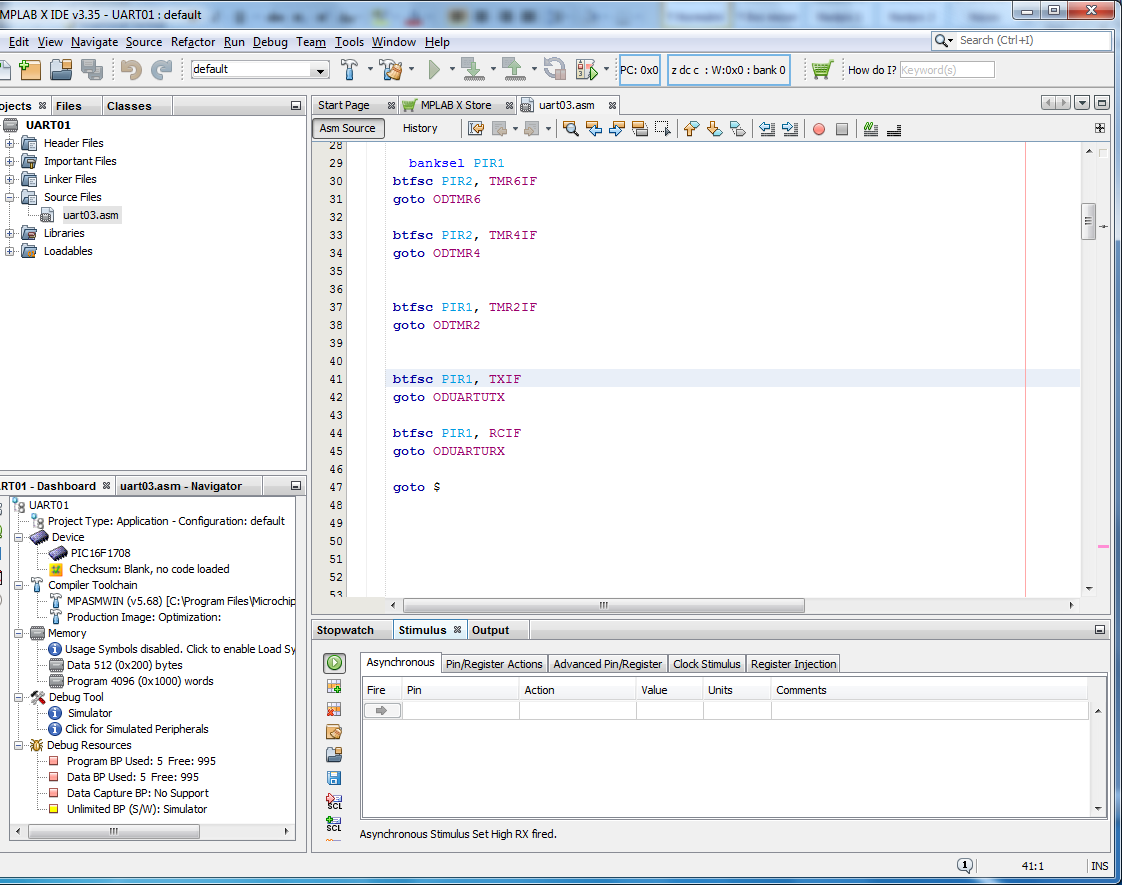
**Simulace přijímače UARTu v MPLABX**

Pustíme si dále v simulátoru okno Window – Simulator – Stimulus

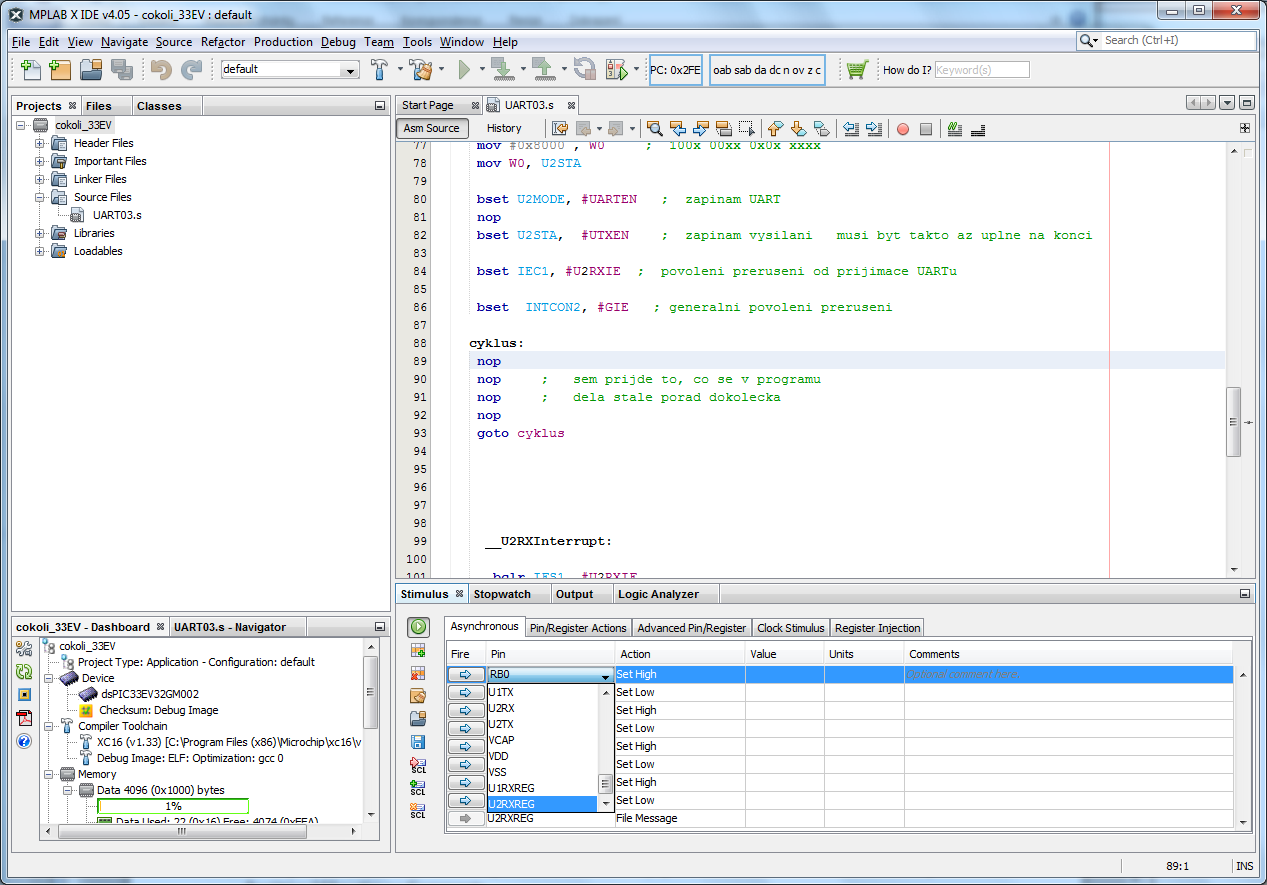
Vysílání znaku do UARTu nasimulujeme pomocí Stimulus (Window – Simulator-Stimulus)



Dole se objeví okno Stimulus. To, co potřebujeme, je Asynchronous



V nám již známém okně jsou řádky. Cvakneme do sloupečku pin, a v rozbalovacím menu úplně dole najdeme U2RXREG ( nebo U1RXREG pro první UART )

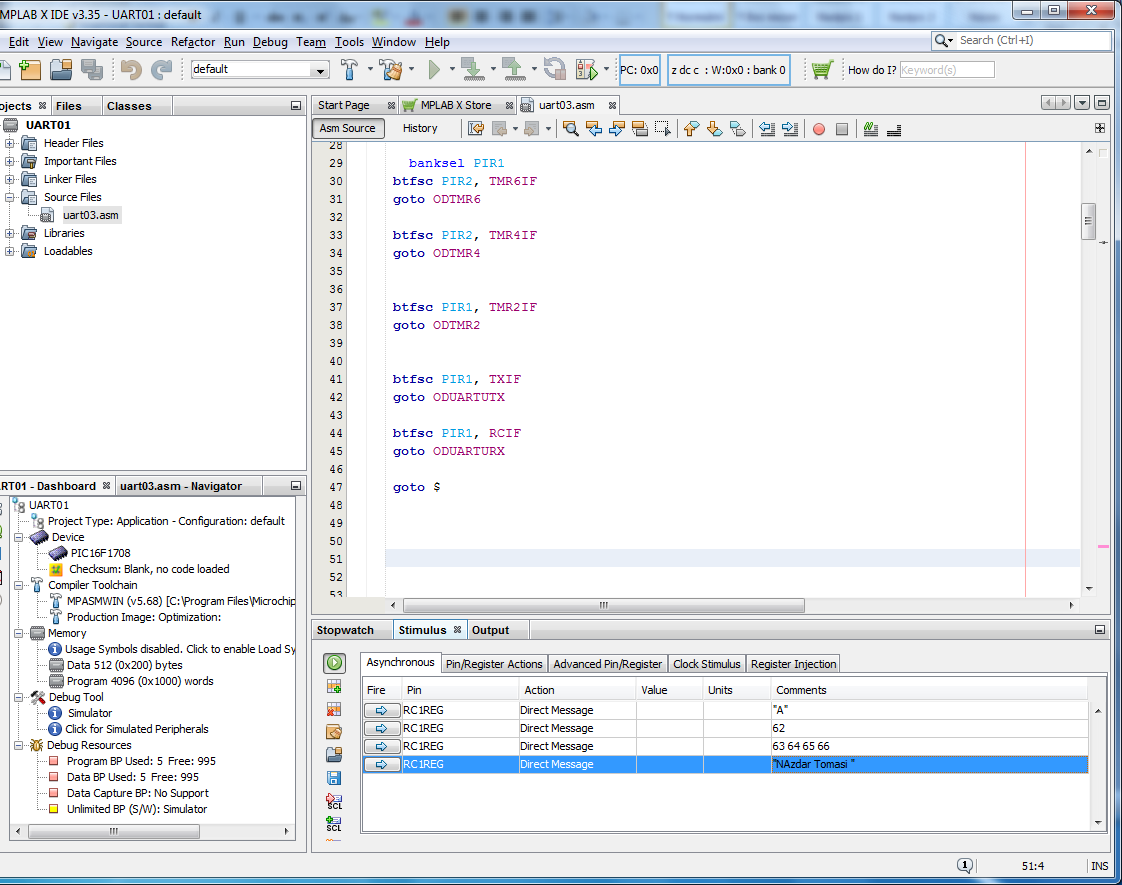


Zvolíme ho.

Můžeme si udělat několik řádek s týmž registrem – U2RXREG

Ve sloupečku Action nastavíme Ditect Message . Ve sloupci Commenst máte napsáno, co máte dělat. Bohužel dost nečitelně. – Type string or hex code . String je samozřejmě v uvozovkách, jak známe z jazyka C. Hexadecimální čísla jsou ve tvaru 56 fa 23 89 11 ff 1f atd, pokud jich dáme víc na řádek, jsou oddělena mezerami. V jednom řádku může být jeden znak nebo více znaků. Zadáme je dvojitým kliknutím na Comments.

Příklad máte zde ( je to pro jiný procesor, vzsílací regsitr je samozžejmě UxRXREG ):



No a dál je práce naprosto stejná, jako byla u nastavování bitů portu. Cvakneme na Fire, a v ten okamžik se nastavená hodnota pošle zvenku do UARTu. Samozřejmě, pokud máme v řádku několik znaků, posílají se postupně a procesor tedy generuje několik přerušení.

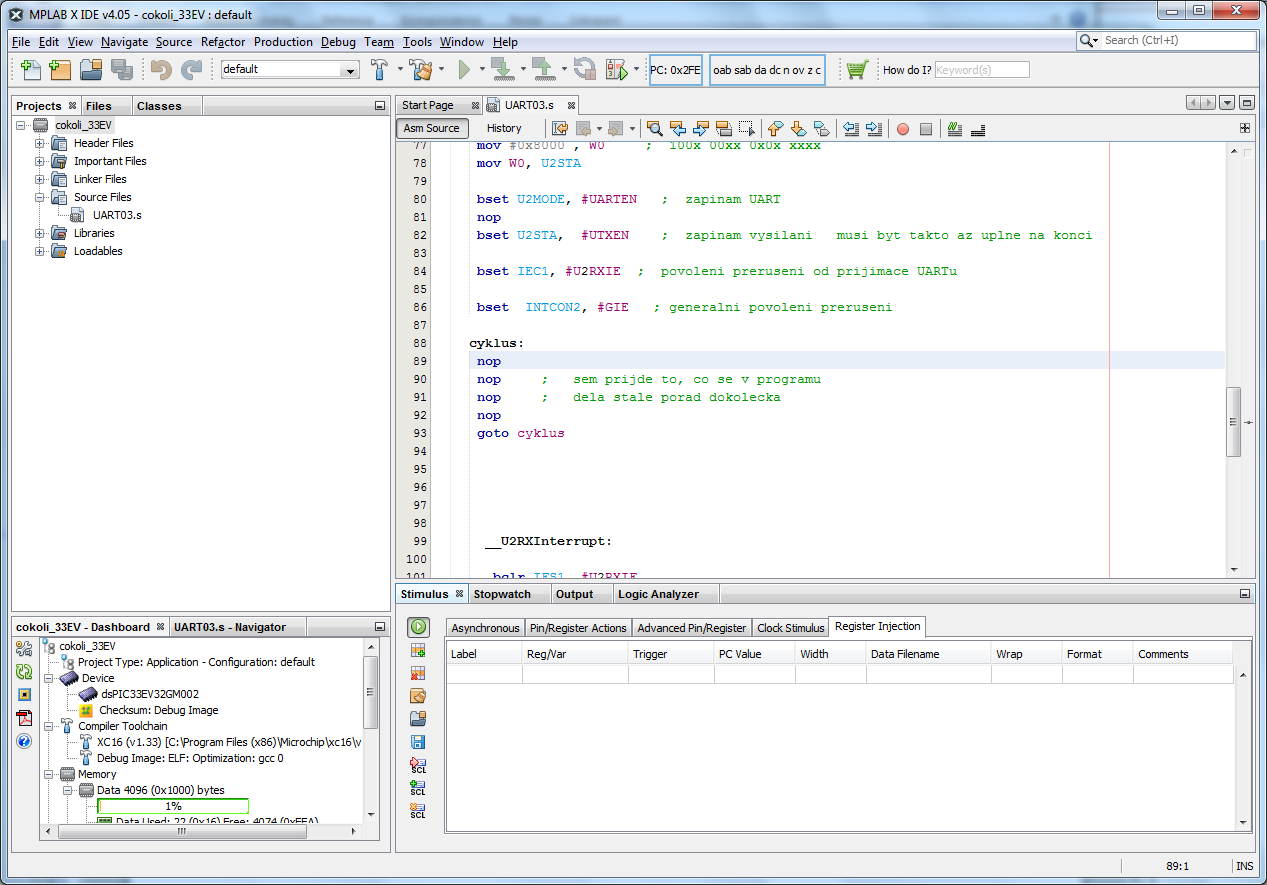
No a pomocí fire posíláme znaky. Debuuger přitom běží !! Vždycky, když cvaknete na Fire, pošlete znak (nebo několik znaků) a program se zastaví na breakpointu. Ve Variables se podíváme na přijímací registr nebo na regsistr W0, tam vidíme, co jsme přijali.

Pokud v programu nemáme breakpoint, tak simulátor běží a běží. Ale funguje ! Takže při cvaknutí na Fire se ve výstupním okně UARTu objeví to, co má. Zkuste to !

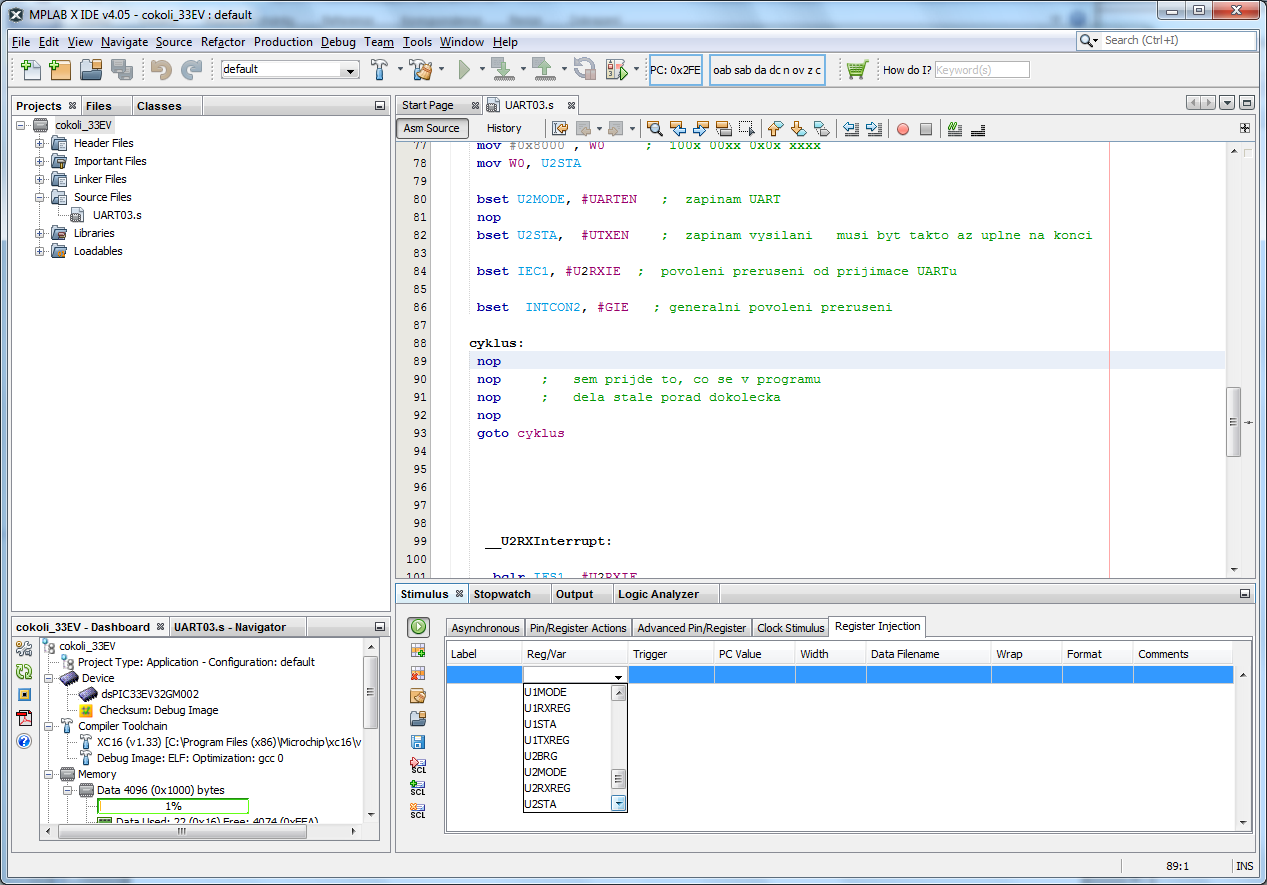
Pokud si „nacvakáte“ pomoví fire mnoho znaků do bufferu, pamatujte, že s nimi simulátor stále počítá a chce je přijímat. Zbavíme se jich tak, že uděláme „close debugger section“ a potom znovu „Debug main project“

Druhá možnost je výrazně krásnější, ale také komplikovanější.

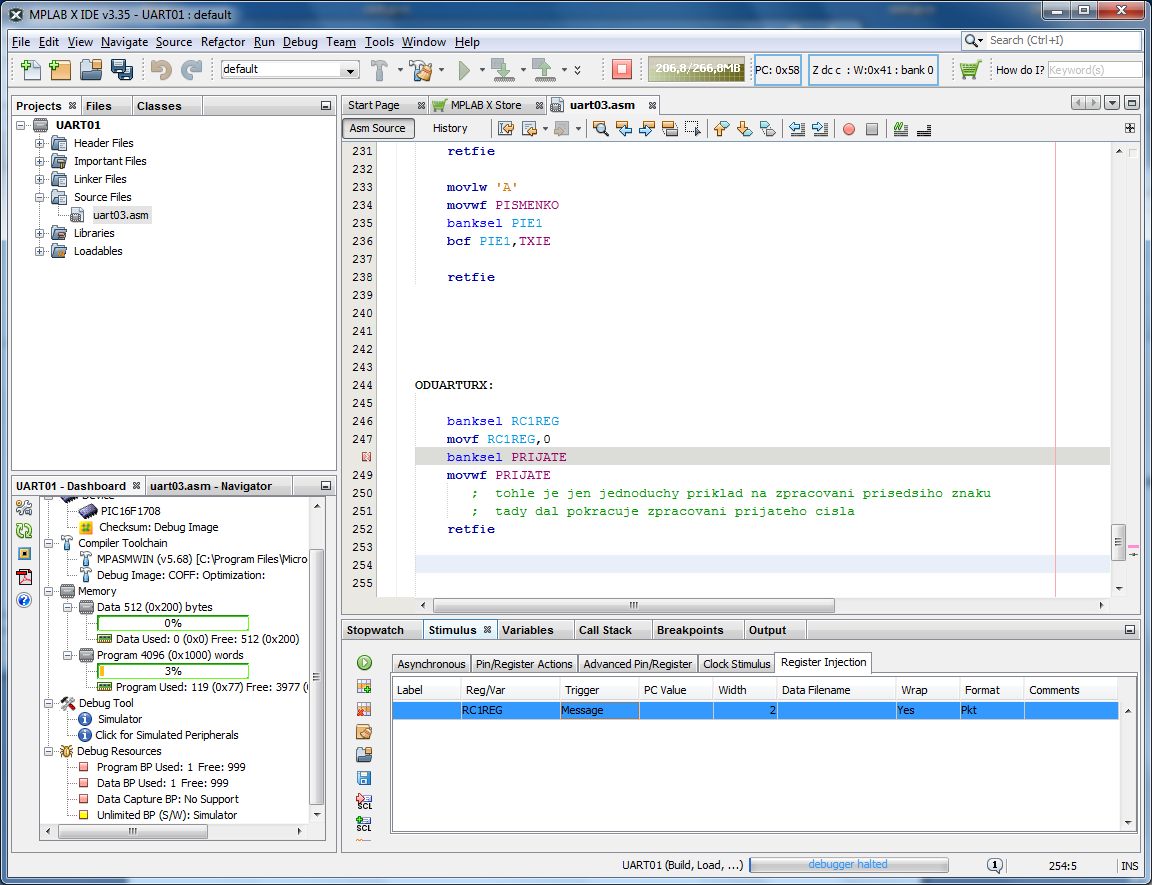
Tentokrát cvakáme na kartu Register Injection



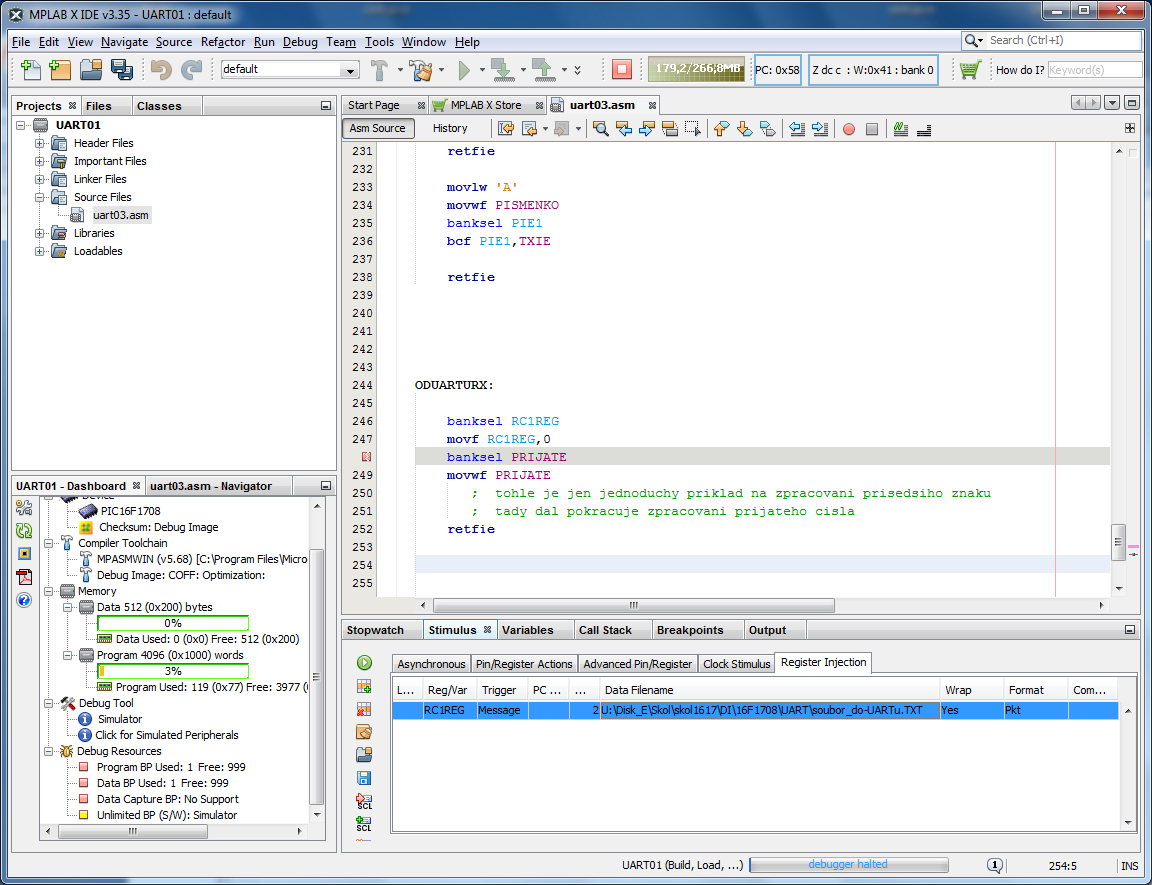
A dále na sloupec Reg/Var a vybereme U2RXREG



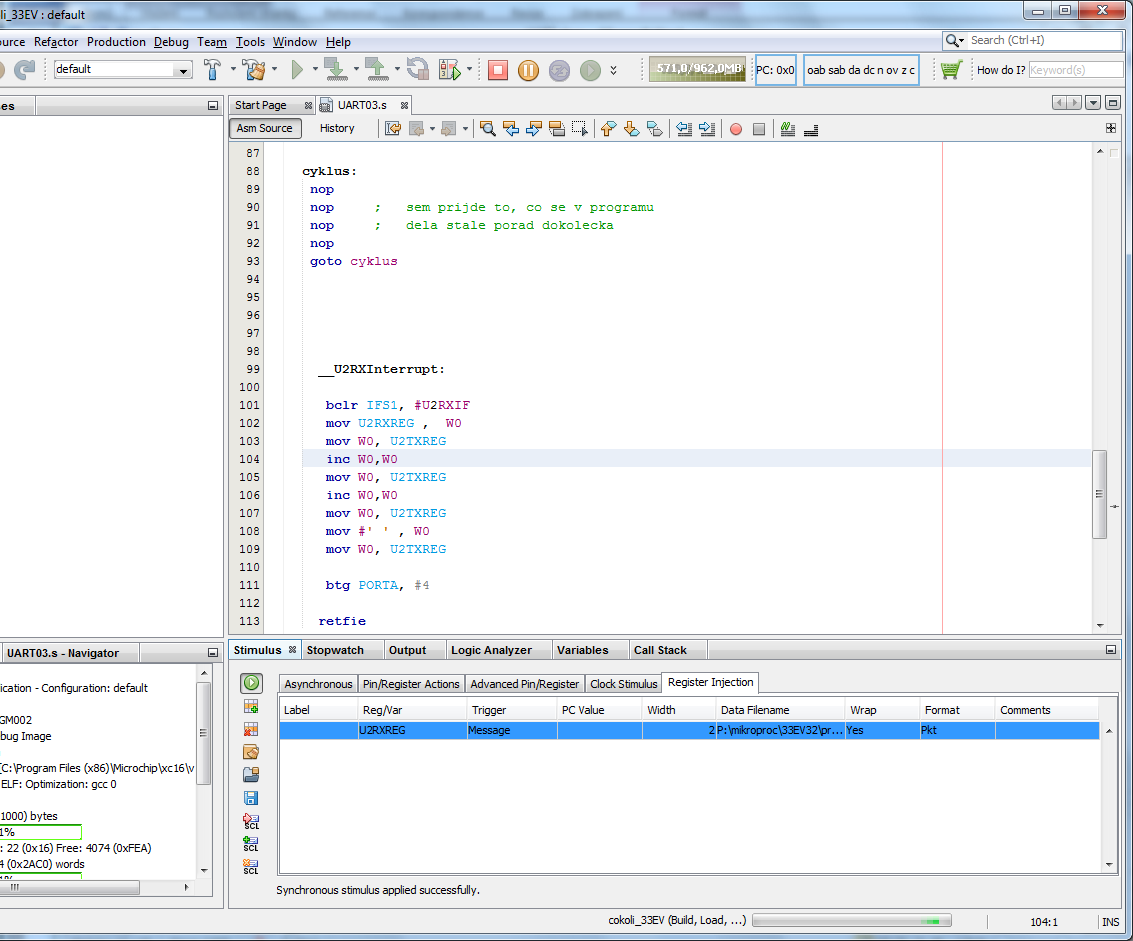
Ve sloupečku trigger nastavíme message



A do „Data filename“ zadáme cestu k file, ve kterém máme příkazy pro UART – tedy hlavně data, která chceme poslat do UARTu.



Dále cvakneme na „apply synchronous stimulus“



Dole v okně se objeví „Synchronous stimulus applied successfully“

Příklad syntaxe file je v

soubor\_do\_UARTu.txt

File je „self-explanatory“ , takže není co řešit.