

Εργαστήριο 4

05/11/2020

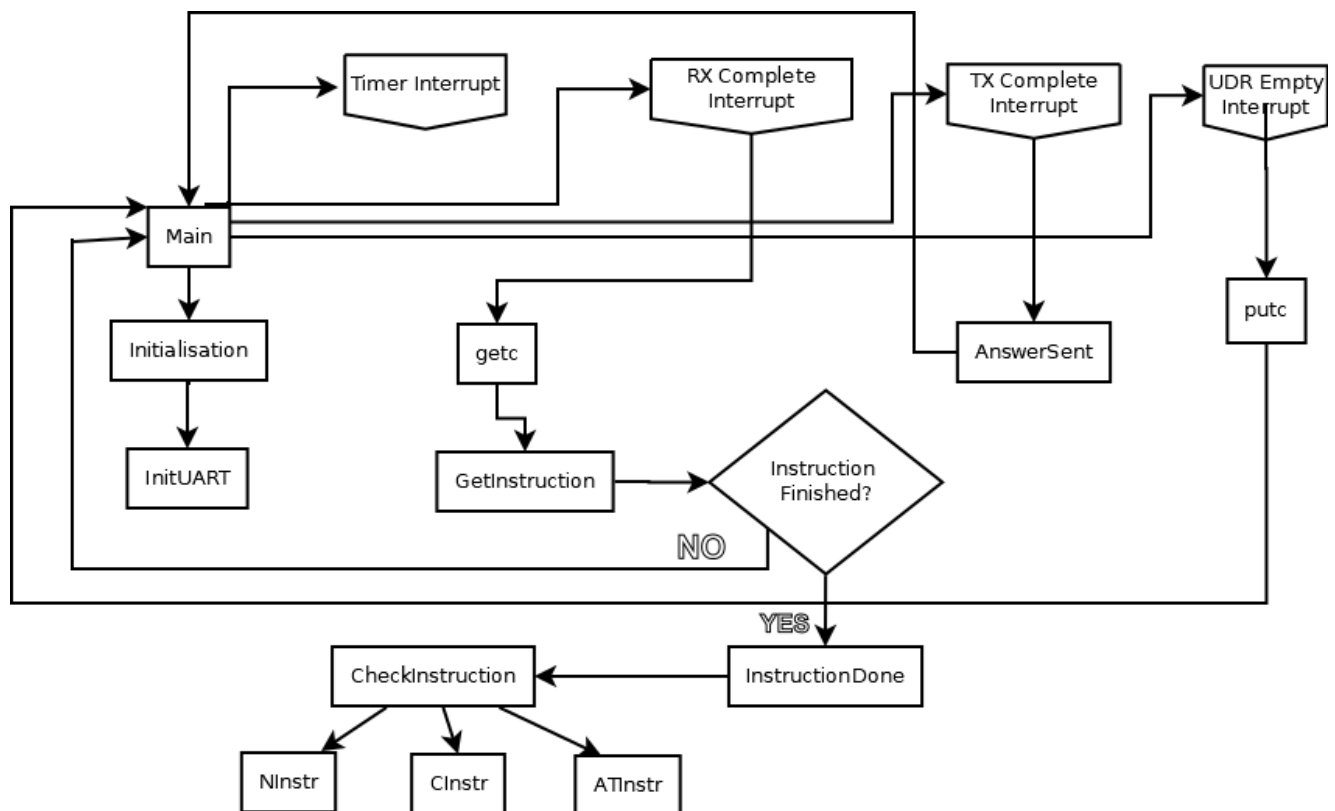
ΚΩΔΙΚΑΣ C ΚΑΙ ASSEMBLY ΣΤΟΝ ATMEL AVR

Ντουνέτας Δημήτρης
AM: 2016030141

Εισαγωγή

Σκοπός του εργαστηρίου είναι η εξοικείωση με τη γλώσσα C, τον Compiler AVR GCC καθώς και το AVRlibc πακέτο για προγραμματισμό Μικροελεγκτών AVR. Επιπλέον, η κατανόηση της χρήση C σε συνδυασμό με γλώσσα assembly για πιο Optimized κώδικες. Τέλος, παρατήρηση της assembly εξόδου από τον C Compiler και πως κάνει χρήση των πόρων του συστήματος. Δημιουργία προγράμματος για επικοινωνία με χρήση USART του ATMEGA16, για παραλαβή εντολών και αποστολή απαντήσεων σε συνδυασμό με οδήγηση με πολυπλεξία στον χρόνο μίας οθόνης 7-segment LED για (έως) οκτώ ψηφία.

Block Diagram Προγράμματος



Αρχικοποίηση του προγράμματος σε Γλώσσα C

Αρχικά το πρόγραμμα ξεκινάει από την συνάρτηση `main` που βρίσκεται στο αρχείο `main.c`. Η `main` συνάρτηση είναι υπεύθυνη για την αρχικοποίηση του προγράμματος και της μνήμης καθώς και τον ατέρμονα βρόγχο πάνω στον οποίο τρέχει το πρόγραμμα. Για γίνει η αρχικοποίηση η `Main` καλεί τη συνάρτηση `initialization`. Ο `Stack Pointer` δεν χρειάζεται να αρχικοποιηθεί αφού τον αρχικοποιεί ο `C Compiler` από μόνος του ώστε να μπορούμε να επιστρέφουμε σωστά από τις ρουτίνες και τις συναρτήσεις που καλούμε.

Η `Initialization` στην συνέχεια καλεί τις `initUART`, `initRam` και `initPorts` συνάρτησεις οι οποίες κάνουν τις αρχικοποιήσεις μνήμης, καταχωρητών, θέσεων μνήμης και `Ports` όπως αναφέρθηκαν στο προηγούμενο εργαστήριο. Συνοπτικά χρησιμοποιούνται 3 σημεία στην μνήμη `SRAM` στα οποία το ένα με όνομα `display_address` είναι το σημείο στο οποίο γράφονται τα `BCD` στοιχεία που θα δείξει η οθόνη. Το δεύτερο είναι το `UARTAnsAddress` όπου γράφεται η αυτοματοποιημένη απάντηση που δίνει ο Μικροελεγκτής μας όταν λάβει ένα ολόκληρο `Instruction` σωστά δηλαδή `OK<CR><LF>`. Τέλος το τρίτο είναι το `InstrAddress` όπου αποθηκεύεται το `instruction` που λαμβάνουμε ώστε να μπορούμε να το διαχειριστούμε κατάλληλα όταν θέλουμε να εκτελέσουμε την εντολή.

```
void initialisation(void){
    //Chip Initializations
    initUART();
    initRam();
    initPorts();
    // Enable interrupts
    sei();
}

int main(void)
{
    initialisation();
    while (1)
    {
    }
}
```

```
void initPorts(){
    // Ports Initialization
    DDRA = 0xFF;    //Port A Initialization as output
    DDRC = 0xFF;    //Port C Initialization as output

    // Enable Compare Interrupt
    TIMSK = (1<<OCIE1A);
    OCR1AH = HIGH(LoopValue);
    OCR1AL = LOW(LoopValue);
    TCCR1B = (1<<CS10)|(1<<WGM12);
}
```

```
void initUART(){

    UBRRL = LOW(UBRRValue);    // load baud prescale
    UBRRH = HIGH(UBRRValue);    // to UBRR

    UCSRB = (1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE); // enable transmitter, receiver
    UCSRC = (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0);    //Set frame format: 8data, 1stop bit
}
```

```

void initRam(){
    // RAM Initialization for display
    uint8_t i;
    for(i=0;i<=7;i++){

        *display_address = i;
        display_address += 1;
    }
    display_address = 0x0070;
    // UART answer Initialization
    *UARTAnsAddress = ASCII_0;
    UARTAnsAddress += 1;
    *UARTAnsAddress = ASCII_K;
    UARTAnsAddress += 1;
    *UARTAnsAddress = ASCII_CR;
    UARTAnsAddress += 1;
    *UARTAnsAddress = ASCII_LF;
    UARTAnsAddress += 1;

    UARTAnsAddress = 0x0080;

    //Instruction Address Initialization
    InstrAddress = 0x0090;

    //Counters Initializations
    RingCounterAddress = 0x0100;
    *RingCounterAddress = 0b00000001;

    DigitRegAddress = 0x0101;
    DigitCounterAddress = 0x0102;
}

```

RAM MAP

0x70	0x71	0x72	0x73	0x74	0x75	0x76	0x77
7segbit7(LSB)	7segbit6	7segbit5	7segbit4	7segbit3	7segbit2	7segbit1	7segbit0(MSB)

Η λέξη που βρίσκεται στη θέση 0x70 είναι το δεξιότερο Digit που θα εμφανιστεί στο 7seg-Screen.

0x80	0x81	0x82	0x83
UARTAnsByte7(O)	UARTAnsByte6(K)	UARTAnsByte5(CR)	UARTAns4(LF)

0x90	0x91	0x92	0x93	0x94	0x95	0x96	0x97	0x98
instrByte0	instrByte1	instrByte2	instrByte3	instrByte4	instrByte5	instrByte6	instrByte7	instrByte8

0x99	0x9A	0x9B
instrByte9	instrByte10	instrByte11

Στις θέσεις ορισμένες ως instrByte εισάγεται διαδοχικά η λέξη της κάθε εντολής που λαμβάνουμε.

0x100	0x101	0x102
RingCounterAddress	DigitRegAddress	DigitCounterAddress

Διαδικασία μετατροπής του προγράμματος από Assembly σε C/AVR.libc assembly mix.

Αρχικά για να τη μεταφορά του προγράμματος από απλή assembly σε μίξη assembly με C γλώσσα χρειάζεται η αλλαγή του επιθέματος του προγράμματος που είναι γραμμένο σε assembly από .asm σε .S . Στη συνέχεια είναι αναγκαία η αλλαγή όλων των Assembly Declarations σε C Declarations.

Για την κλήση μιας συνάρτησης Assembly από τον C κώδικα χρειάζεται να δηλωθεί η συνάρτηση ως .global στον κώδικα assembly.

Τέλος σημαντική προσοχή πρέπει να δοθεί στη χρήση καταχωρητών αφού τους χρησιμοποιεί ο C compiler και έτσι δεν μπορούν πλέον να χρησιμοποιηθούν εύκολα ως μεταβλητές. Καλή πρακτική είναι η αποθήκευση των στοιχείων που θέλουμε να χρησιμοποιήσουμε ως μεταβλητές στην μνήμη και η προσπέλαση αυτών από τη μνήμη όποτε χρειάζεται.

Αναλυτικά ο C compiler κάνει push τους καταχωρητές R18 – R31 στην Stack πριν από την κλήση κάθε εντολής assembly οπότε αυτούς τους καταχωρητές μπορούμε να τους χρησιμοποιήσουμε άφοβα στις συναρτήσεις που γράφουμε στην assembly.

InitUART

Η προσθήκη σε αυτό το εργαστήριο είναι η InitUART όπου φορτώνουμε στον καταχωρητή UBRR την κατάλληλη τιμή που έχουμε υπολογίσει μέσω του Datasheet και σύμφωνα με BAUR Rate 9600 και συχνότητα επεξεργαστή 10Mhz είναι 64. Αυτή η τιμή εισάγεται στον DownCounter του USART ώστε να γίνει το κατάλληλο Prescaling για να λάβουμε το σωστό Baud Rate. Στη συνέχεια, ενεργοποιούμε τον Transmitter και τον Receiver Καθώς και τα Interrupts του βάζοντας 1 στα Bits RXEN, TXEN, RXCIE, TXCIE που βρίσκονται στον καταχωρητή UCSRC.

```
;initialize UART
initUART:
    ldi temp,LOW(UBRRValue) ; Load UBRR value
    ldi temp1,HIGH(UBRRValue)
    out UBRRRL,temp          ; load baud prescale
    out UBRRH,temp1         ; to UBRR

    ldi temp,(1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE) ; enable transmitter, receiver ,
    out UCSRB,temp          ;Receive Complete and Transmit Complete Interrupts
```

Λειτουργία του κύριου προγράμματος και τα Interrupt του USART(USART).

Καθώς το πρόγραμμα έχει ξεκινήσει και βρίσκεται στον ατέρμονα βρόγχο καλούνται τα κατάλληλα interrupts και αυτά χρησιμοποιούμε για να δώσουμε λειτουργικότητα στο πρόγραμμα μας. Αυτή τη στιγμή υπάρχουν 4 Interrupts τα οποία αξιοποιεί το πρόγραμμα για να δείξει στην οθόνη ότι εμείς του δίνουμε ως είσοδο μέσω του UART. Τα Interrupts αυτά είναι:

1. Timer/Counter Compare1 Interrupt
2. USART RX Complete Interrupt
3. USART DATA Registry Empty Interrupt

4. USART TX Complete Interrupt

- Το Timer/Counter 1 Compare A Interrupt είναι υπεύθυνο για το σωστό χρονισμό της οθόνης.

```
ISR(TIMER1_COMPA_vect)
{
    CTC_Timer_Interrupt();
}
```

- Καλεί την συνάρτηση CTC_TIMER_Interrupt η οποία τρέχει στον Assembly.S και μας δίνει το κατάλληλο χρονικό περιθώριο για να ανανεώνουμε την 7segment οθόνη για να δώσουμε τα FPS που θέλουμε.
- Το USART RX Complete Interrupt είναι υπεύθυνο για το σωστό διάβασμα όταν σηκωθεί το Flag RXC το οποίο υποδηλώνει ότι έχουμε λάβει κάποιον χαρακτήρα έτοιμο να διαβάσουμε.

```
ISR(USART_RXC_vect)
{
    RXC_Interrupt();
}
```

Καλεί την συνάρτηση RXC _Interrupt η οποία καλεί την getc, που τρέχει στο Assembly.S αρχείο και λαμβάνει ένα χαρακτήρα και τον αποθηκεύει. Θα εξηγηθεί παρακάτω περαιτέρω.

- Το USART DATA Registry Empty Interrupt είναι υπεύθυνο να ενημερώνει όταν το UDR καταχωρητής είναι διαθέσιμος για εγγραφή ώστε να μην απανογράφουμε χωρίς να έχουν αποσταλεί πρώτα τα δεδομένα που θέλουμε.

```
ISR(USART_UDRE_vect)
{
    UDRE_Interrupt();
}
```

Καλεί την συνάρτηση UDRE_Interrupt η οποία καλεί την putc που τρέχει στο Assembly.S αρχείο και στέλνει έναν χαρακτήρα στο UDR ώστε αυτός να αποσταλεί ορθά από τον USART.

- Το USART TX Complete Interrupt είναι υπεύθυνο για την ενημέρωση της αποστολής ενός χαρακτήρα. Μας ενημερώνει ότι η αποστολή έχει ολοκληρωθεί και ο χαρακτήρας έχει φτάσει στον προορισμό του. Η διαφορά με το DATA Registry Empty Interrupt είναι ότι περιμένει μέχρι το Byte να φύγει εντελώς από τον Shift Register που αποστέλλει σειριακά και έτσι μας

είναι χρήσιμο μόνο σε HALF-Duplex πρωτόκολλα ή όταν θέλουμε να κάνουμε ενέργεια μετά το πέρας της αποστολής. Έτσι το χρησιμοποιούμε για να οριστικοποιήσουμε την αποστολή όλης της απάντησης του Μικροελεγκτή μας.

```
ISR(USART_TXC_vect)
{
    UTXC_Interrupt();
}
```

Καλεί την ρουτίνα AnswerSent η οποία τρέχει στο Assembly.S αρχείο και καθαρίζει τη θέση μνήμης που είναι αποθηκευμένη η εντολή ώστε να είναι έτοιμη να δεχτεί την επόμενη εντολή.

Παρακάτω γίνεται επεξήγηση των ρουτίνων που τρέχουν στην Assembly από τα προηγούμενα εργαστήρια για λόγους πληρότητας.

Η Ρουτίνες getc και putc

Η ρουτίνα getc τραβάει ένα Byte που έχει σταλεί μέσω του USART το αποθηκεύει σειριακά στην κατάλληλη θέση μνήμης από το Reserved σημείο InstrAddress και στη συνέχεια ελέγχει αν έχει ολοκληρωθεί η αποστολή του Instruction σύμφωνα με τους χαρακτήρες <CR> και <LF>.

```
getc:
    /*; For polling
    ; in temp, UCSRA
    ; sbrs temp, RXC
    ; rjmp getc*/
    LDS temp, InstrAddress_pointer    //Load direct from data space
    mov ZH, r1
    mov ZL, temp

    in temp, UDR
    in temp, UDR //; Get Data from USART

    mov temp, r15 //; Added for testing pupposes. R15 is the input now
    st Z, temp
    //;Store Instruction into SRAM
    ldi temp1, ASCII_LF //; If last character was <LF>
    cpse temp, temp1 //; Instruction is all stored and go to InstructionDone
    rjmp .+4
    rcall InstructionDone
    ret
    //Move Instruction pointer 1 position
    LDS temp, InstrAddress_pointer    //Load direct from data space
    SUBI temp, 0xFF //Subtract immediate
    STS 0x0066, temp //Store direct to data space
    ret
```

Η ρουτίνα `putc` αποστέλει ένα χαρακτήρα στον USART. Τραβάει διαδοχικά τους χαρακτήρες από τη θέση μνήμης που είναι γραμμένη η απάντηση μέχρι να τραβήξει τον χαρακτήρα `<LF>` όπου κλείνει το Interrupt UDRE αφού η αποστολή δεν έχει άλλο χαρακτήρα προς μετάδοση.

```
putc:
//;Polling checks
//;in temp, UCSRA
//;sbrs temp, UDRE
//;rjmp putc
lds temp, UARTAnsAddress_pointer
mov ZH,r1
mov ZL,temp
// Move Pointer 1 position
ld AnsReg, Z //;load next character from Answer Address
SUBI temp,0xFF //Subtract immediate
STS UARTAnsAddress_pointer,temp //Store direct to data space

out UDR, AnsReg
out UDR, AnsReg //; Transmit to USART
out TCNT2, AnsReg //; Send to TCNT2 for testing purposes.

ldi temp,ASCII_LF //;If current character is <LF>
cpse AnsReg,temp //; Answer is sent
rjmp .+4 //; else return and wait for next character

ldi temp,(1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE) ; Disable UDRIE Cause Answer sent
out UCSRB,temp
ret
```

Διαχείριση εντολών

Για την διαχείριση των εντολών που λαμβάνουμε μέσω του UART υπάρχουν ξεχωριστές συναρτήσεις που ενεργούν αφού λάβουμε σωστά ολόκληρη την εντολή και ουσιαστικά μας έρθει ο χαρακτήρας CR ακολουθούμενος από τον LF. Τότε πηγαίνουμε στην θέση `InstructionDone` όπου ετοιμάζουμε τους δείκτες για τον έλεγχο της εντολής που μας ήρθε.

```
;Store Instruction into SRAM
GetInstruction:
    st X+,ReceivedByte
    ldi temp,ASCII_LF ; If last character was <LF>
    cpse ReceivedByte , temp ; Instruction is all stored and go to InstructionDone
    reti
    jmp InstructionDone

; After the complete instruction is stored send Instruction Pointer at the beggining
InstructionDone:
    ;Reinitialize X pointer
    ldi XL,LOW(InstrAddress)
    ldi XH,HIGH(InstrAddress)
    jmp CheckInstruction
    reti
```


Η ρουτίνα Checkinstruction ελέγχει τί είδους εντολή μας ήρθε από τα 3 δυνατά σενάρια που έχουμε και στέλνει το πρόγραμμα στον κώδικα για τη διαχείριση της.

```
;Check which instruction is Received
CheckInstruction:
    lds InstructionByte,InstrAddress
    ldi temp,ASCII_A
    cpse InstructionByte,temp ;If first Character is ASCII A
    rjmp PC+2
    jmp ATInstr ; Go to AT instruction
    ldi temp,ASCII_C
    cpse InstructionByte,temp ;If first Character is ASCII C
    rjmp PC+2
    jmp CInstr ; Go to Clear instruction
    ldi temp,ASCII_N
    cpse InstructionByte,temp ;If first Character is ASCII N
    rjmp PC+2
    jmp NInstr ; Go to Number instruction
    jmp Error ;Else go to Error
```

Επειδή θεωρούμε ότι δεν έχουμε λάθος στη μετάδοση και οι εντολές έρχονται πάντα σωστά μπορούμε να απλοποιήσουμε τους ελέγχους που κάνουμε και απλά να ελέγξουμε το πρώτο χαρακτήρα που λάβαμε ο οποίος είναι διαφορετικός για κάθε εντολή.

Έτσι καταλήγουμε σε μια από τις παρακάτω εντολές όπου κάνουμε τις κατάλληλες ενέργειες.

```
;Attention instruction
ATInstr:
    jmp SendOK
;Clear instruction
CInstr:
    rcall StartClear
    jmp SendOK
;Number instruction
NInstr:
    rcall StartClear
    ldi XL,0x8B
    ldi YL,LOW(display_address)
    ldi YH,HIGH(display_address)
    rcall RenewSeg
    jmp SendOK
```

ATTENTION Εντολή

Όταν λάβουμε την εντολή AT<CR><LF> αρκεί να στείλουμε πίσω μια απάντηση OK. Αυτό εκτελεί η SendOK στην οποία ενεργοποιούμε το Interrupt για να στείλουμε την απάντηση.

```
SendOK:
    ldi temp,(1<<UDRIE)|(1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE) ; enable transmitter and receiver
    out UCSRB,temp
    reti
```


CLEAR Εντολή

Όταν λάβουμε την εντολή C<CR><LF> αρκεί να καθαρίσουμε την οθόνη ώστε να μη δείχνει τίποτα και να στείλουμε πίσω την απάντηση OK. Έτσι καλούμε την ρουτίνα StartClear και μετά την SendOK.

```
;Initiate clear of screen
StartClear:
    ldi YL,LOW(display_address) ;Start from the Beggining
    ldi YH,HIGH(display_address)
    ldi temp1,0x68 ;Until you reach the end of reserved memory
    rcall clear7seg ;call routine
    ret

;Clear all screen
clear7seg:
    ldi temp,0x0A ;Put 0x0A in every digit
    st Y+,temp ;Store to displayAddress
    cpse YL,temp1 ;Check if at the end of DisplayAddress
    jmp clear7seg
    jmp Reset_digit_Counter ;Then Reset Screen
    ret
```

NUMBER Εντολή

Όταν λάβουμε την εντολή N239...<CR><LF> αρκεί να καθαρίσουμε την οθόνη ώστε να μη δείχνει τίποτα ύστερα να βάλουμε τα σωστά ψηφία στην οθόνη και να στείλουμε πίσω την απάντηση OK. Καθαρίζουμε πηγαίνουμε στη συνάρτηση RenewSeg και στέλνουμε την απάντηση όπως προηγούμενα.

```
;Renew Segments
RenewSeg:
    ld temp,-X ;Load character from the end
    ldi temp1,ASCII_CR ; until you find CR loop
    cpse temp,temp1
    jmp RenewSeg
LoopRen:
    ld temp,-X ;Load number from the end
    ldi temp1,ASCII_N ;Until character N
    cpse temp,temp1
    jmp PC+3
    ret
    rcall AddNumToSeg ;call routine that stores number to segment address
    jmp LoopRen

;Add number to a segment
AddNumToSeg:
    ld temp,X
    andi temp,0b00001111 ; Mask for clearing the upper 8 Bytes. Turn ASCII to BCD
    st Y+,temp ;Store BCD for Segment
    ret
```

Αυτή η ρουτίνα διαβάζει την αποθηκευμένη εντολή από το τέλος και αφού βρει που ξεκινάνε τα νούμερα που έχουμε ως ορίσματα τα αποθηκεύει 1-1 κατάλληλα στις θέσεις μνήμης ώστε να τα δείχνει σωστά η οθόνη 7seg. Δηλαδή το δεξιότερο νούμερο το αποθηκεύει στο δεξιότερο 7seg και τα υπόλοιπα αντίστοιχα.

Τελικό Αποτέλεσμα

Το πρόγραμμα τρέχει συνεχόμενα χωρίς να σταματάει αφού ως βασικό μέρος έχει έναν ατέρμονα βρόγχο. Διαβάζει εντολές και ενεργεί κατάλληλα καθώς επίσης απαντάει και OK αφού διαβάσει μια σωστή εντολή.

Παρατήρηση

Επειδή το Atmel Studio δεν έχει κατάλληλο περιβάλλον για αποσφαλμάτωση USART επικοινωνίας έχουν γίνει κάποιες αλλαγές στο πρόγραμμα ώστε να είναι δυνατή η προσομοίωση του με STIMFILE και να logάρονται κατάλληλα οι απαντήσεις. Η εισαγωγή των bytes που λαμβάνονται γίνεται με τον Καταχωρητή 15 και έτσι τα διαβάζουμε ως είσοδο στο πρόγραμμά μας. Επίσης οι απαντήσεις δίνονται στον Καταχωρητή TCNT2 αυθαίρετα. Θα μπορούσαμε να χρησιμοποιήσουμε οποιουσδήποτε καταχωρητές δεν χρησιμοποιούμε απλά το Stim File δεν μπορεί να διαβάσει τον UDR ούτε τα PINS RX και TX.

Στην διαδικασία αποσφαλμάτωσης παρατηρήθηκε επίσης ότι κατά τη διάρκεια αποστολής συμβόλου που χρησιμοποιούμε το Flag UDRE για την ενεργοποίηση του Interrupt αυτό το Flag όπως και το RXC χρειάζεται να το προσπελάσουμε και να το γράψουμε 2 φορές και να ανταποκριθεί το USART καθώς και ότι αφού σταλούν δεδομένα χρειάζεται περίπου 180.000 κύκλους για να σηκωθεί το επόμενο Flag και να καλεστεί το interrupt. Αυτό σύμφωνα με τους υπολογισμούς που λαμβάνουμε υπόψιν το Baud Rate θα έπρεπε να συμβαίνει κάθε 10.000 κύκλους χοντρικά. Πιθανόν να είναι σφάλμα που οφείλεται στο Atmel Studio είτε στην τρόπο που θεωρεί το Stim File ρολόι επειδή αυτή η διαφορά μιας τάξης μεγέθους θα μπορούσε να δικαιολογηθεί αν το ρολόι είχε τιμή κοντά στο 1 MHz.