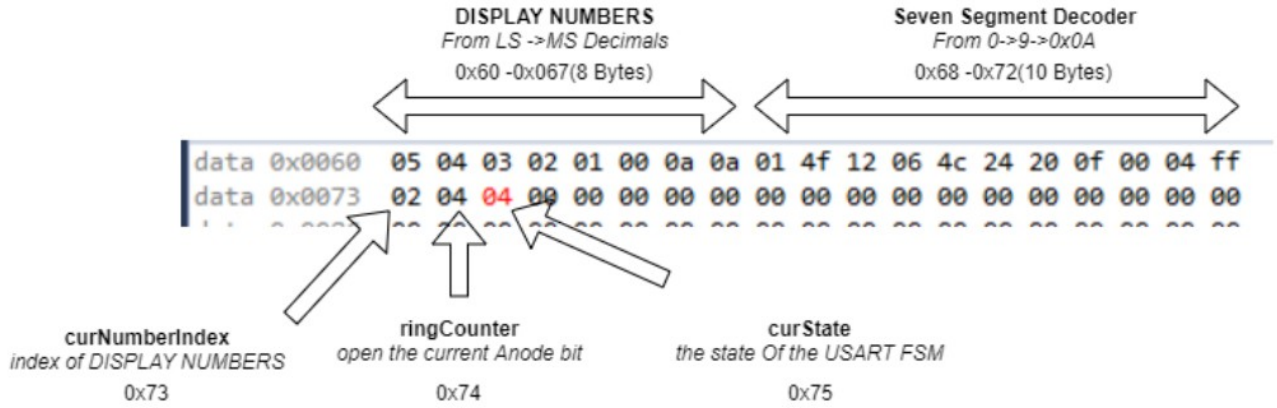
# Θεόδωρος Μπάρκας 2016030050 LAB5

## Σκοπός άσκησης

Στη Συγκεκριμένα άσκηση σκοπός ήταν η πλήρης μετατροπή του κωδικά μας σε C κρατώντας όμως καλή επαφή με τους πόρους της συσκευής. Έτσι κρατήσαμε ίδιο Memory Map με την προηγούμενου υλοποίηση επομένως γνωρίζουμε ακριβώς τις θέσεις RAM που χρειάζονται για τη σωστή εκτέλεση του προγράμματος.




## Μετατροπές άσκησης

Δημιουργήσαμε ένα κομμάτι κωδικά για κάθε Interrupt κώδικας ο οποίος τρέχει μεσώ της *ISR(TIMER0_OVF_vect)* και της *ISR(USART_RXC_vect)* .Ο τρόπος προσπέλασης της μνήμης στη C έτσι ώστε να εξυπηρετεί τον σκοπό μας (δηλαδή να έχουμε επίγνωση των θέσεων μνήμης που χρησιμοποιούμε ) είναι με δημιουργία Pointer (unsigned char *memloc) και διάβασμα/γράψιμο των περιεχόμενων με τους εξής τρόπους:

- (unsigned char value=*memloc)
- είτε με τη μορφή πινάκων ως (unsigned char value=memloc[ i ])

## ISR(TIMER0_OVF_vect)

ISR(TIMER0_OVF_vect)

- Περνούμε από τη μνήμη τον `curNumberIndex` και τον `ringCounter`
- Εκτελούμε τη λογική του προγράμματος αλλάζοντας το `ringCounter` και τον `curNumberIndex` .
- Ενημερώνουμε το Display με τον εξής τρόπο

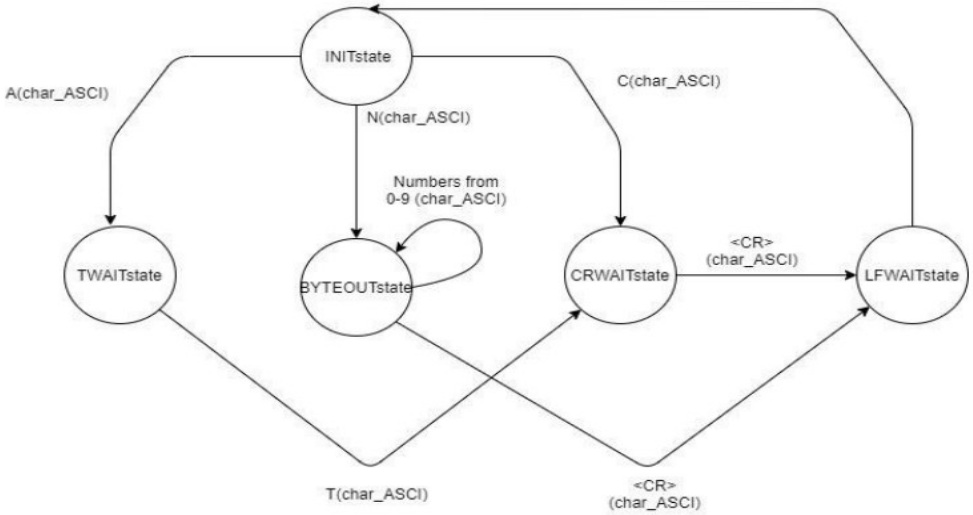
```
//display changes
PORTC=0x00;//close display
curNumber=numberArray[curNumberIndex];//getting number
PORTA=ssdArray[curNumber];//setting up SSD
PORTC=ringCounter;//open display's right anode
```

- Ενημερώνουμε τις θέσεις μνήμης που αποθηκεύονται τα `curNumberIndex` και `ringCounter.`

## ISR(TIMER0_OVF_vect)

ISR(TIMER0_OVF_vect)

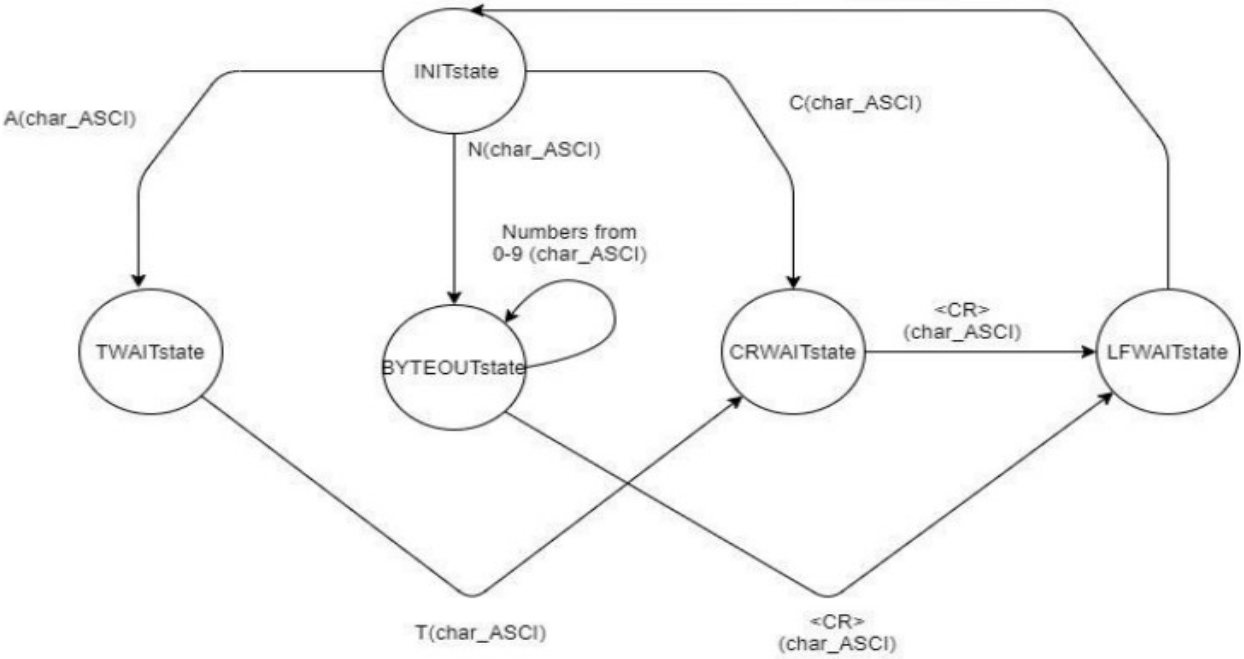
- Πήραμε το Address της μνήμης που αποθηκεύει το State της FSM στην τιμή
  `unsigned char *stateMem`




- Διαβάζουμε το UDR έτσι ώστε να πάρουμε την τιμή που εισήρθε άλλα και έτσι ώστε και να κλείσει το Interrupt Flag που ενεργοποιήθηκε.
  TESTING-SIMULATION
  Για να περάσουμε τα ορίσματα καθώς δεν μπορούμε κάνουμε χρίση της UART μέσω Simulation αλλά ούτε και να αλλάξουμε απευθείας τον καταχωρηθώ UDR , θέτουμε στο Stimuli file τον R15 ως την είσοδο του UDR.
  Κατόπιν περνάμε την τιμή της R15 που προσομοιώνει τον UDR στη θέση μνήμης RAM 0x0076 μέσω Assembly .
  Η χρήση γλωσσάς Assembly ήταν αναγκαία για να επεξεργαστούμε τον συγκεκριμένο Register που θέτει το Stimulifile ως είσοδο (R15) και να περάσουμε δεδομένα από αυτόν σε μια μεταβλητή της C(στη περίπτωση μας στο input).

```
//For simulation purposes
unsigned char *simData =0x76;
__asm__(
"ldi XH,0x00\n"
"ldi XL,0x76\n"
"st X,r15\n"
"nop\n"
);
input=*simData;
```

- Η FSM υλοποιήθηκε μεσώ ενός Switch Statement στην C και η λογική της παραμένει παρόμοια με αυτή της Assembly και γίνεται και πολύ εύκολα αντιληπτή μεσώ του παρακάτω διαγράμματος καταστάσεων.




- Όταν είμαστε στο LFWAITstate και άρχετε το <LF> καλούμε τη συνάρτηση Polling που καλούσαμε και στην Assembly και δημιουργήσαμε στο προηγούμενο εργαστήριο στη C.

```
case LFWAITstate:       //LF Just Came
    USART_Transmit_OK();
    *stateMem=INITstate;
  break;
```

- Επίσης δημιουργήθηκαν δυο νέες συναρτήσεις οι οποίες όμως έχουν ακριβός ίδια λειτουργιά με τις αντίστοιχες Assembly συναρτήσεις των προηγουμένως εργαστηρίων :

```
//numberArray[0 to 8] = 0x0A ->which means closed screens after SSD_decode
void clearAllNumbers()
```

Καλείται όταν δεχόμαστε είσοδο <C> <N> και είμαστε στην INITstate

```
//add input to start of numberArray and shifts all the other numbers
void add_number_UDR(unsigned char input)
```

Καλείται όταν δεχόμαστε είσοδο <number 0 to 9> or not<CR> και είμαστε στην BYTEOUTstate

*Τα States ορίζονται με DEFINE στην αρχή του κωδικά.

```
#define INITstate 0x00
#define TWAITstate 0x01
#define BYTEOUTstate 0x02
#define CRWAITstate 0x03
#define LFWAITstate 0x04
```

# Testing

Το Testing δεν αλλάζει και παραμένει ίδιο με τα προηγούμενα εργαστήρια.

Παρατηρούμε παρόλα αυτά στον μεταφρασμένο κωδικά(Dissasembly) την σχετικά μεγαλύτερη και συχνότερη χρίση του stack που κάνει ο Compiler σε σχέση με τις προηγούμενες υλοποιήσεις μας.