

Flaviu Nistor, Tudor Orlandea

Sisteme Incorporate în exemple simple

Ediție revizuită și adăugită

Editura Universității "Lucian Blaga"

Sibiu 2019

Flaviu Nistor, Tudor Orlandea

TITLU: Sisteme Incorporate în exemple simple - Ediție revizuită și adăugită

ISBN:

Referenți științifici:

Prof. dr. ing. Ioan P. Mihu

Sl. dr. ing. Beriliu Ilie

Chenar ISBN

Universitatea „Lucian Blaga” din Sibiu

Bd. Victoriei 10, 550024 Sibiu, RO

Cuprins

Prefață	1
1. Introducere	3
1.1. Introducere	3
1.2. Mediul de dezvoltare MPLAB IDE	3
1.2.1. Crearea unui proiect nou	4
1.2.2. Selectarea dispozitivului.....	4
1.2.3. Selectarea header pentru debugging	5
1.2.4. Selectarea dispozitivului pentru debugging	5
1.2.5. Selectarea compilator	6
1.2.6. Selectarea nume proiect.....	6
1.2.7. Adăugarea fișierelor în proiect	7
1.2.8. Setările microcontrolerului.....	8
1.3. Descriere generală - Low Pin Count Demo Board.....	10
1.4. Schema electrică - Low Pin Count Demo Board	11
1.5. Layout - Low Pin Count Demo Board	12
1.6. Lista de materiale - Low Pin Count Demo Board.....	12
1.7. Probleme propuse.....	13
2. Embedded C	14
2.1. Introducere	14
2.2. Sintaxa limbajului C.....	16
2.2.1. Comentarii	16
2.2.2. Directive de pre-procesare.....	17
2.2.3. Variabile	19
2.2.4. Funcții.....	21
2.2.5. Operatori.....	22
2.2.6. Instrucțiuni de control	26
2.3. Programare embedded.....	28

2.3.1. Bucla infinită	28
2.3.2. Întreruperile	29
2.3.3. Operații pe biți.....	29
2.4. Aplicație propusă	30
2.5. Probleme propuse	31
3. Prezentare μ C	32
3.1. Introducere	32
3.2. Caracteristici principale – PIC16F690	32
3.3. Diagrama pinilor și descrierea acestora	34
3.4. Arhitectura microcontrolerului PIC16F690	37
3.5. Harta memorie.....	38
3.6. Probleme propuse	39
4. Pinul de ieșire (Output pin).....	41
4.1. Introducere	41
4.2. Pinul de ieșire	41
4.3. Limitări electrice	46
4.4. Probleme propuse	46
4.5. Aplicație propusă	46
4.6. Model Software	48
4.7. Problemă propusă.....	49
5. Pinul de intrare (Input pin).....	50
5.1. Introducere	50
5.2. Pinul de intrare	51
5.3. Pull-up/Pull-down	56
5.4. Switch Debounce	57
5.5. Probleme propuse	59
5.6. Aplicație propusă	59
5.7. Model software.....	61
5.8. Problemă propusă.....	64

6. Timer 1	65
6.1. Introducere	65
6.2. Descriere Timer 1	65
6.3. Aplicație propusă	70
6.4. Configurarea timer-ului.....	71
6.5. Model software.....	73
6.6. Problemă propusă.....	77
7. Timer 2.....	79
7.1. Descriere Timer 2.....	79
7.2. Aplicații propuse	83
7.3. Configurarea timer-ului.....	85
7.4. Model software.....	86
7.5. Problemă propusă.....	88
8. Servomotor.....	89
8.1. Introducere	89
8.2. Comanda unui servomotor	90
8.3. Aplicație propusă	92
8.4. Model software.....	94
8.5. Problemă propusă.....	96
9. Convertor Analog Numeric.....	97
9.1. Introducere	97
9.2. Descriere ADC pe 10 biți.....	99
9.3. Aplicație propusă	104
9.4. Configurarea ADC	106
9.5. Model software.....	107
10. UART.....	110
10.1. Introducere	110
10.2. Descriere modul UART	110
10.2.1. Blocul de transmisie	113
10.2.2. Blocul de recepție	114

10.2.3. Setarea ceasului (rata de transfer)	115
10.2.4. Regiștri de configurare ai modulului UART	117
10.3. Aplicație propusă	121
10.4. Configurarea modulului UART	121
10.5. Model software.....	123
Anexa 1 - Detalii suplimentare Embedded C	126
Anexa 2 - Programarea microcontrolerului	130
Anexa 3 - Interpretarea oscilogramelor	131
Anexa 4 - Alocarea spațiului de memorie de către compilator.....	134
Anexa 5 - Procesul de build	153
Bibliografie	156

Prefață

Noțiunea de sistem incorporat este folosită tot mai des în zilele noastre în domeniul hardware și software. Un sistem incorporat poate fi definit ca și sistem dedicat, proiectat pentru a fi capabil să realizeze o anumită funcție într-un sistem mai complex, iar pentru această sarcină are nevoie de intrări prin care să citească starea sistemului și de ieșiri pentru controlul unor procese.

Tot mai multe dispozitive folosite zi de zi au la bază un microcontroler. Chiar și un banal filtru de cafea sau un uscător de păr au la baza un astfel de circuit (compus dintr-o parte hardware pe care rulează o aplicație software). Tocmai din acest motiv înțelegerea funcționării unui microcontroler nu ar trebui să lipsească din bagajul de cunoștințe al unui absolvent de electronică, electromecanică sau calculatoare.

Această carte are ca și scop introducerea cititorului în lumea embedded prin niște pași progresivi, împărțiți în zece capitole care tratează noțiunile de bază necesare realizării unor proiecte simple care au la bază un microcontroler.

Cartea dorește să fie un suport pentru orice student care o parcurge și un punct de plecare pentru cei care vor să cunoască lumea embedded în detaliu prin studiu individual ulterior.

Cartea este scrisă de niște foști studenți pentru actualii studenți, într-un format pe care l-au considerat potrivit ca un prim contact al cititorului cu noțiuni despre microcontroler și programarea acestuia.

Dorim să mulțumim foștilor noștri profesori Miha P. Ioan, Ilie Beriliu și Toma Emanoil care nu de puține ori au fost mai mult decât profesori pentru noi în timpul anilor de studiu și un permanent sprijin în cadrul activităților noastre. Mulțumim și companiei Continental și în special lui Sorin Ban pentru sprijinul acordat în vederea tipăririi acestei cărți. Nu în ultimul rând mulțumim tuturor celor care au avut răbdarea necesară pentru a citi materialul și s-au asigurat de forma lui corectă înainte de tipărire.

Sibiu, 21 ianuarie 2012

În cei șapte ani trecuți de la publicarea inițială, am avut bucuria de a primi reacții și păreri pozitive din partea mai multor persoane, fie ei specialiști care au răsfoit materialul sau profesori și studenți ce au folosit cartea ca și suport de curs. Văzând că subiectul e încă de interes și folositor, am decis să republicăm cartea, aducând în același timp și niște completări și revizuiți. Pe lângă o serie de corecturi minore, am modificat și capitolul de prezentare al mediului de programare pentru a descrie o versiune mai nouă a acestuia. De asemenea, am decis să adăugăm două noi capitole ce cuprind informații pentru un nivel mai avansat. Pentru a nu modifica structura inițială a cărții, cu zece capitole gândite pe post de lucrări de laborator, aceste două capitole sunt adăugate ca și anexe, ele servind ca o sursă de informații pentru cei ce doresc să aprofundeze subiectul organizării memoriei în microcontrolere sau al lanțului de compilare (build chain). Sperăm ca aceste informații să ajute pe oricine se află la început de drum în acest domeniu și caută, precum am făcut-o și noi la rândul nostru, mai multe informații despre aceste teme mai puțin discutate.

În continuare suntem recunoscători tuturor celor care ne-au susținut și ajutat la redacarea materialului și la mulțumim tuturor celor ce au citit sau vor citi cartea, cu speranța că informațiile din ea vor deveni o mică parte din vastele lor cunoștințe în domeniu.

Sibiu, 2 octombrie 2019

1. Introducere

1.1. Introducere

Această lucrare se dorește a fi un suport pentru disciplina „Sisteme Incorporate” și prezintă o abordare practică asupra aplicațiilor cu microcontrolere. Pentru a putea parcurge cu ușurință acest material, cititorul are nevoie de cunoștințe de bază în electronică și noțiuni introductive de programare. Materialul cuprinde pașii necesari pentru a porni la drum în acest domeniu, oferind o perspectivă asupra celor două componente majore: electronică și software. Pentru ilustrarea acestor concepte prin proiecte practice am ales folosirea unei plăci de dezvoltare relativ ușor de găsit pe piață și anume placa *Low Pin Count Demo Board*, produsă de firma *Microchip*. Aceasta are la bază un microcontroler pe 8 biți numit PIC16F690.

În prima parte a acestui material vom prezenta uneltele de bază de care avem nevoie pentru a începe un proiect (mediul de dezvoltare, placa și microcontrolerul folosit, bazele limbajului C), trecând apoi la descrierea celor mai uzuale module ale unui microcontroler (porturi GPIO, module timer, ADC). Fiecare capitol este structurat în două părți: prima conține informațiile teoretice necesare, iar a doua conține cerințe practice (cu explicații și fragmente de cod) și probleme propuse.

1.2. Mediul de dezvoltare MPLAB IDE

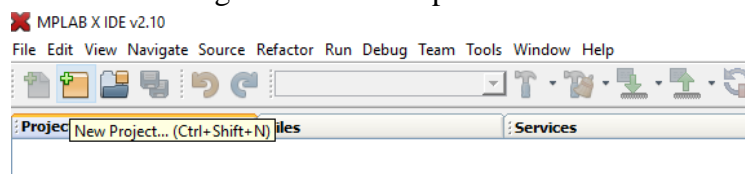
MPLAB IDE este o aplicație PC oferită de către firma *Microchip* și are ca scop facilitarea dezvoltării de cod pentru proiectele care folosesc microcontrolerele acestei firme. Funcționalitatea de bază a mediului de dezvoltare încorporează editare de cod, compilare, suport flashing și debug. Mediul de dezvoltare este disponibil gratuit pe site-ul firmei *Microchip* (www.microchip.com). Pentru realizarea acestui material am folosit MPLAB IDE versiunea X v2.10.

Pentru a putea folosi funcțiile mediului de dezvoltare este necesară crearea unui proiect ce conține sursele aplicației cât și setările microcontrolerului. Pentru a crea un proiect trebuie realizați următorii pași:

1.2.1. Crearea unui proiect nou

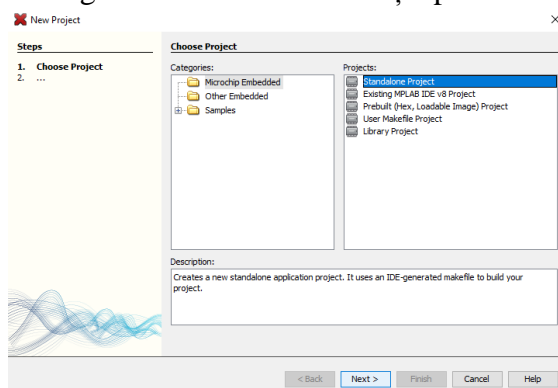
Se va activa butonul **New Project**

Figura 1-1: Creare proiect nou



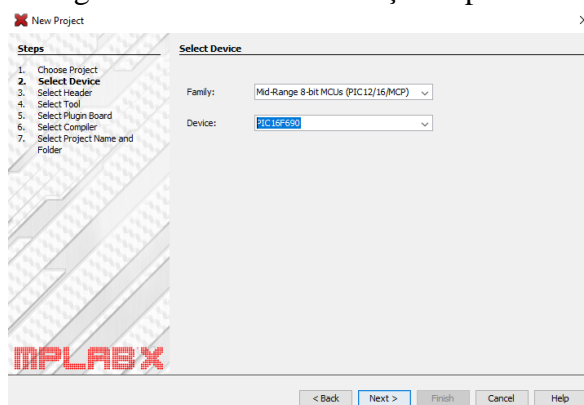
În fereastra deschisă selectați **Microchip Embedded** și în lista din partea dreaptă **Standalone Project**. Se apasă **Next** pentru a trece la următorul pas.

Figura 1-2: Fereastra selecție proiect



1.2.2. Selectarea dispozitivului

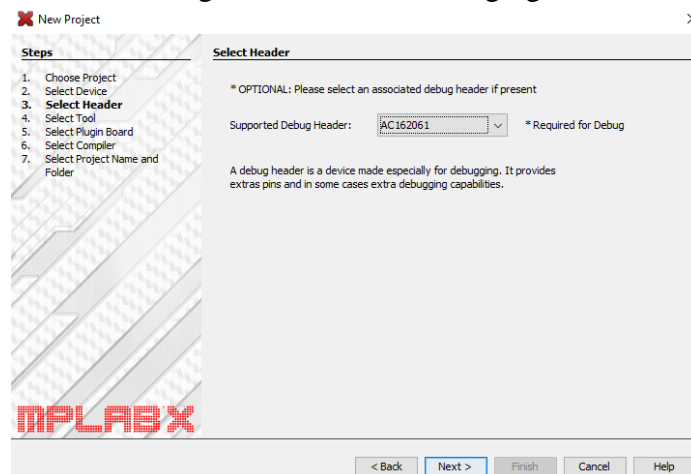
Figura 1-3: Fereastra selecție dispozitiv



Pasul doi din *Project Wizard* este alegerea dispozitivului. Se alege microcontrolerul **PIC16F690** și se apasă **Next**.

1.2.3. Selectarea header pentru debugging

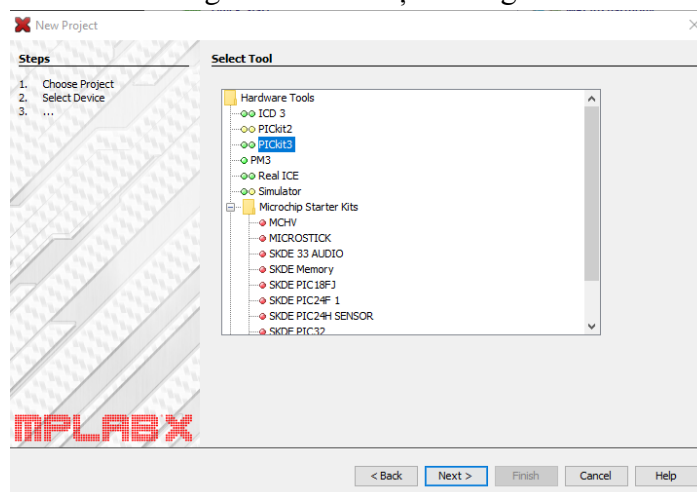
Figura 1-4: Header debugging



Selectați header-ul AC162061 și apăsați butonul **Next**.

1.2.4. Selectarea dispozitivului pentru debugging

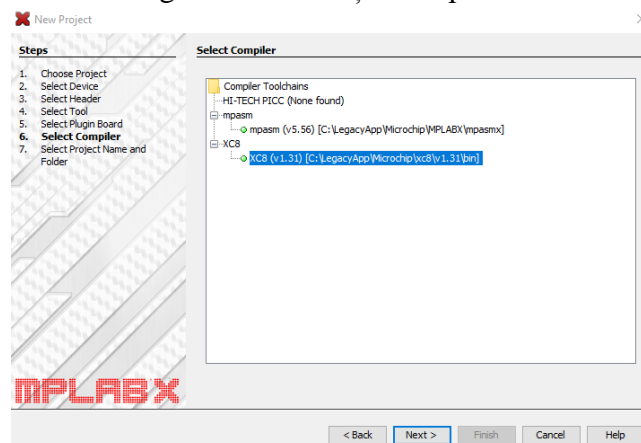
Figura 1-5: Selecție debugger



Selectați **PICKIT3** și apăsați butonul **Next**.

1.2.5. Selectarea compilator

Figura 1-6: Selecție compilator

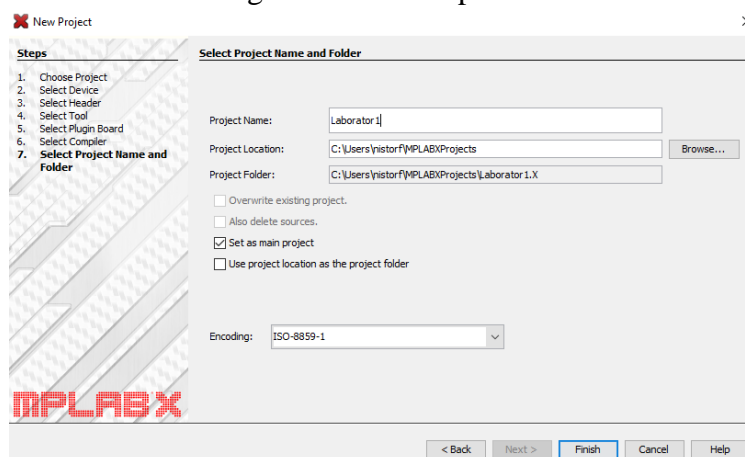


Pentru a putea scrie cod în limbajul C este nevoie de un compilator. Se va selecta compilatorul **XC8** din lista cu compilatoare disponibile și se va apăsa butonul **Next**.

Notă: Compilatorul XC8 se va instala înaintea creării unui proiect nou. Executabilul pentru instalare se găsește gratuit pe site-ul firmei Microchip și necesită o instalare separată față de mediul de dezvoltare Mplab X.

1.2.6. Selectarea nume proiect

Figura 1-7: Nume proiect

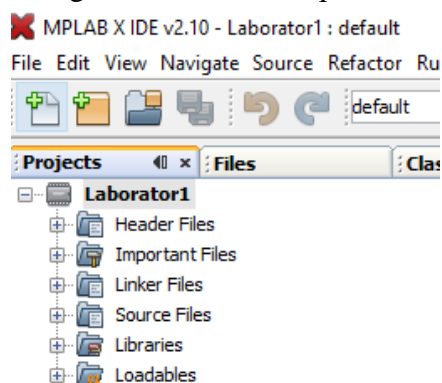


După selecția locației dorite se va introduce numele proiectului (de exemplu *Laborator1*). Se apasă butonul **Finish**.

1.2.7. Adăugarea fișierelor în proiect

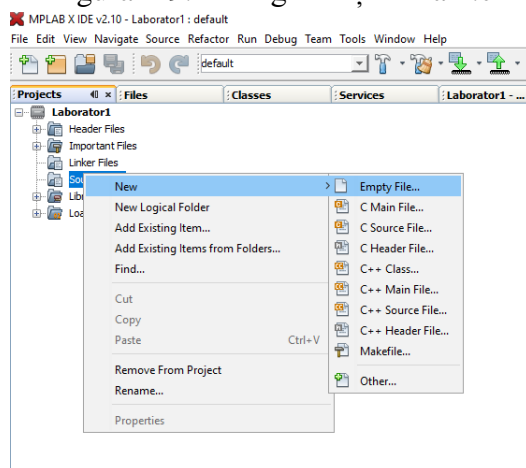
În acest moment proiectul este creat și în partea dreaptă regăsim structura proiectului nou creat.

Figura 1-8: Structură proiect



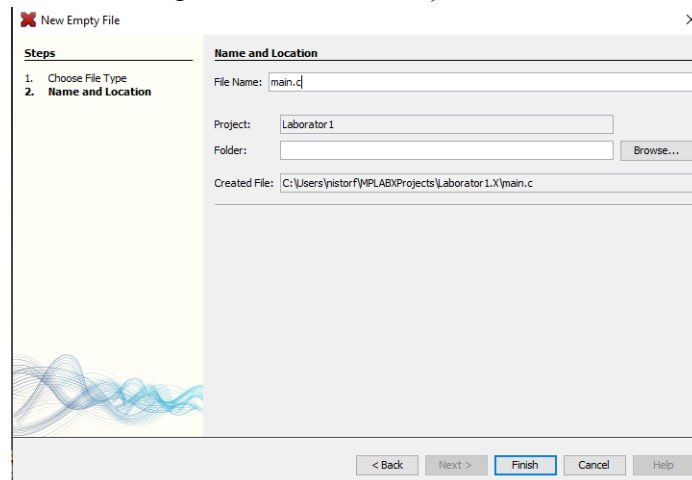
Pentru a adăuga un fișier C care să conțină funcția *main()* și restul aplicației vom face click dreapta pe *Source Files* și vom selecta *Empty File*.

Figura 1-9: Adăugare fișier main.c



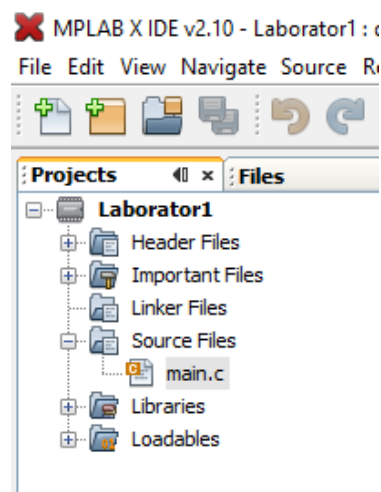
Se va alege numele fișierului. Ca și bună practică se va alege numele *main.c*. Se va apăsa butonul **Finish**.

Figura 1-10: Creare fișier main.c



În acest moment vom regăsi fișierul *main.c* în lista de fișiere sursă a proiectului și prin dublu-click îl vom putea deschide pentru editare.

Figura 1-11: Listă fișiere sursă

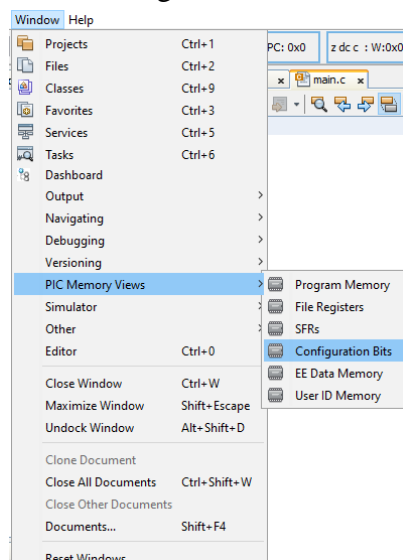


1.2.8. Setările microcontrolerului

Fiecare microcontroler conține un set de regiștri prin care se pot realiza setările de bază ale core-ului. Deși aceștia pot fi modificați direct în cod,

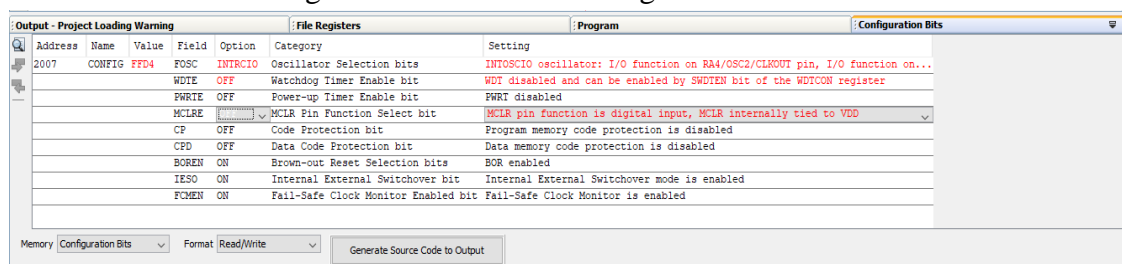
mediul de programare MPLAB ne oferă posibilitatea de a modifica setările de bază direct dintr-o fereastră a meniului. Modificarea acestor setări va fi ultimul pas din pornirea unui proiect și se realizează prin accesarea meniului **Window / Pic memory Views/ Configuration bits**.

Figura 1-12: Configurarea microcontrolerului



Setările prezentate în *Figura 1-13* sunt setările ce vor fi folosite pentru toate proiectele prezentate în acest material.

Figura 1-13: Valori de configurare

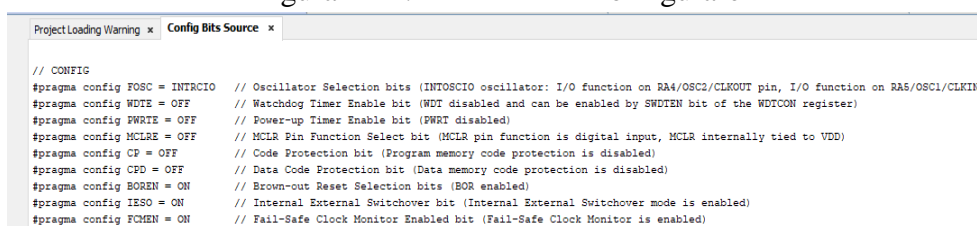


În funcție de versiunea de MPLAB folosită, este posibil să apară diferențe în denumirea setărilor și a categoriilor. În exemplul folosit, singurele categorii ce au fost modificate sunt **Oscillator Selection bits**, **Watchdog Timer Enable bit** și **MCLR Pin Function Select bit**.

Observație 1: Este extrem de important ca aceste setări să fie făcute la începutul fiecărui proiect și cu exact aceleași valori. În caz contrar, este foarte probabil ca proiectele să nu producă rezultatul așteptat.

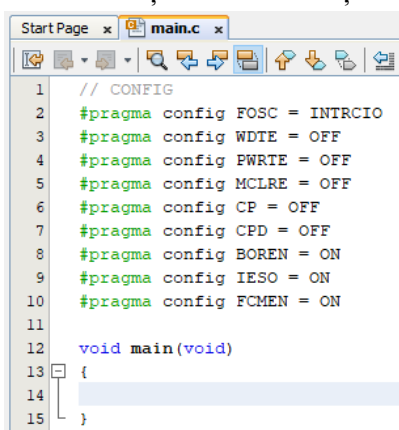
Ulimul pas presupune apăsarea butonului **Generate Source Code to Output** pentru a transla setările realizate în format grafic în linii de cod ce pot fi folosite în cadrul codului sursă. Vom obține următoarea secvență de cod, cea din *Figura 1-14*, pe care o vom copia pentru fiecare lucrare de laborator la începutul fiecărui fișier *main.c*.

Figura 1-14: Cod sursă de configurare



```
// CONFIG
#pragma config FOSC = INTRCIO // Oscillator Selection bits (INTOSCIO oscillator: I/O function on RA4/OSC2/CLKOUT pin, I/O function on RA5/OSC1/CLKIN)
#pragma config WDTE = OFF // Watchdog Timer Enable bit (WDT disabled and can be enabled by SWDTEN bit of the WDTCON register)
#pragma config PWRTE = OFF // Power-up Timer Enable bit (PWRT disabled)
#pragma config MCLRE = OFF // MCLR Pin Function Select bit (MCLR pin function is digital input, MCLR internally tied to VDD)
#pragma config CP = OFF // Code Protection bit (Program memory code protection is disabled)
#pragma config CPD = OFF // Data Code Protection bit (Data memory code protection is disabled)
#pragma config BOREN = ON // Brown-out Reset Selection bits (BOR enabled)
#pragma config IESO = ON // Internal External Switchover bit (Internal External Switchover mode is enabled)
#pragma config FCMEN = ON // Fail-Safe Clock Monitor Enabled bit (Fail-Safe Clock Monitor is enabled)
```

Figura 1-15: Conținut de bază fișier main.c



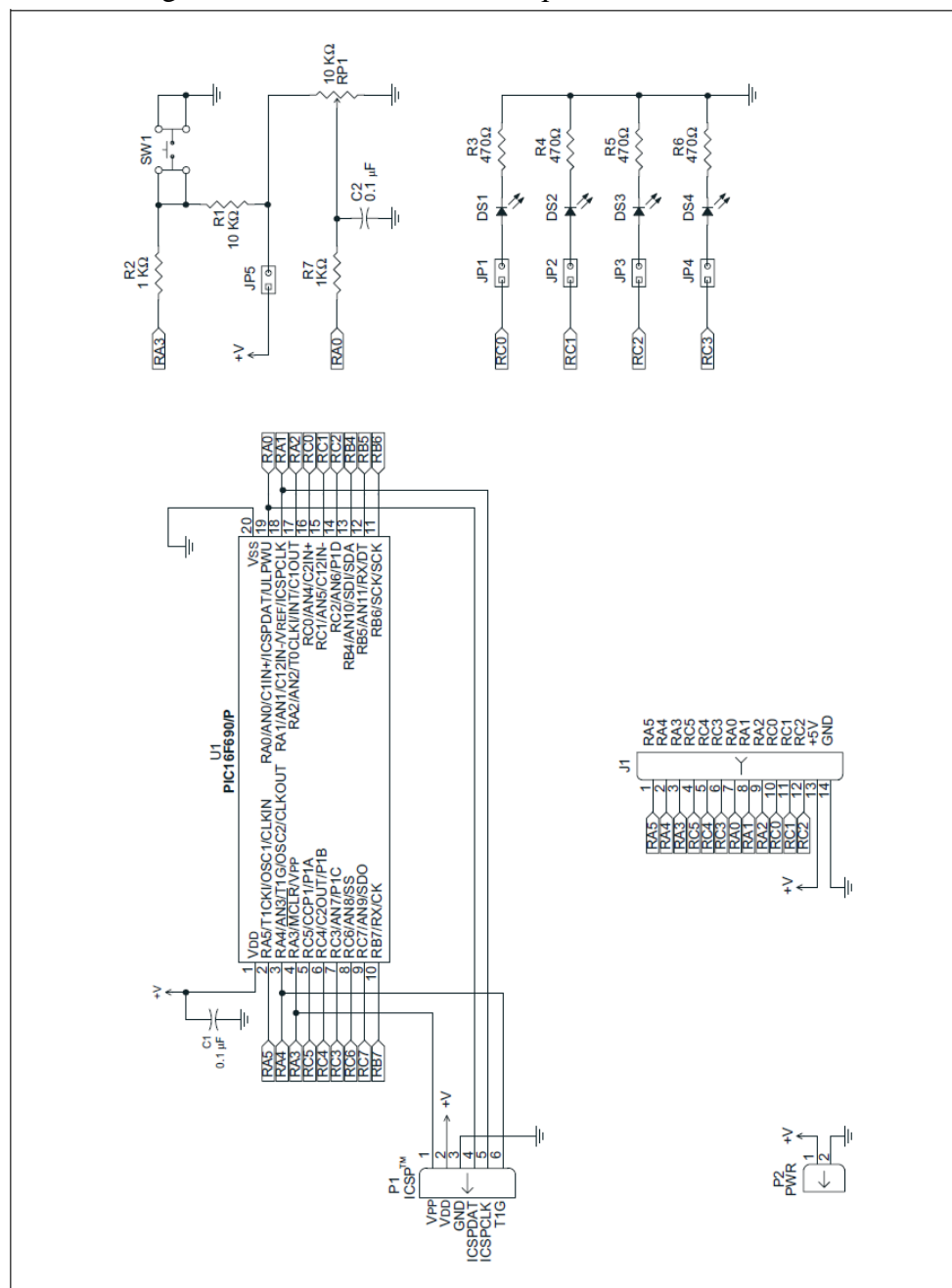
```
1 // CONFIG
2 #pragma config FOSC = INTRCIO
3 #pragma config WDTE = OFF
4 #pragma config PWRTE = OFF
5 #pragma config MCLRE = OFF
6 #pragma config CP = OFF
7 #pragma config CPD = OFF
8 #pragma config BOREN = ON
9 #pragma config IESO = ON
10 #pragma config FCMEN = ON
11
12 void main(void)
13 {
14
15 }
```

1.3. Descriere generală - Low Pin Count Demo Board

Low Pin Count Demo Board este o placă de dezvoltare simplă, pentru microcontrolere cu capsula DIP de 20 pini. Este populată cu PIC16F690, 4 leduri, un push-button și un potențiometrul. Placa de dezvoltare are mai multe puncte de acces pentru pinii microcontrolerului și o zonă dedicată construcției de prototipuri. Programarea microcontrolerului se va face cu ajutorul programatorului PICKIT 2.

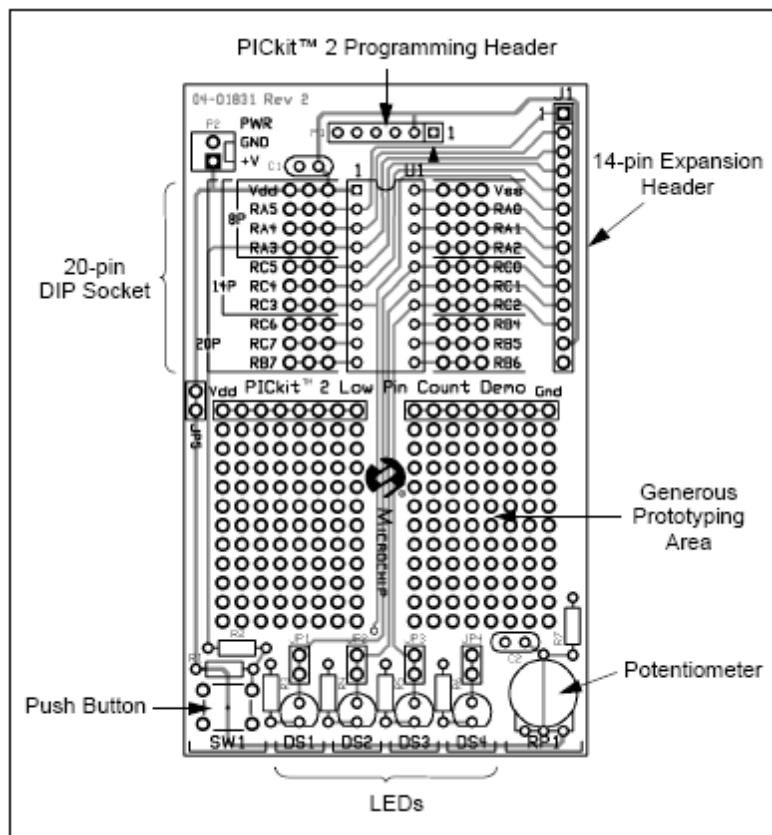
1.4. Schema electrică - Low Pin Count Demo Board

Figura 1-14: Schema electrică a plăcii de dezvoltare [9]



1.5. Layout - Low Pin Count Demo Board

Figura 1-15: Amplasarea componentelor [9]



1.6. Lista de materiale - Low Pin Count Demo Board

Tabel 1-1: Lista de materiale

Nume componentă	Cantitate	Descriere
C1,C2	2	Condensator ceramic THT, 0.1uF, 16V, 5%
R3-R6	4	Rezistor, 470Ω, 5%, 1/8W
R2,R7	2	Rezistor, 1KΩ, 5%, 1/8W
R1	1	Rezistor, 10KΩ, 5%, 1/8W
RP1	1	Potențiometru 10KΩ
DS1-DS4	4	LED, Red
SW1	1	Push buton

U1-Microcontroler	1	20-pin MCU (PIC16F690)
P1	1	Conector, 6 pini, 100 mils
J1	1	Conector, 14 pini, 100 mils
JP1-JP5	5	Jumperi, 2 pini, 100 mils

1.7. Probleme propuse

- Identificați următoarele componente din tabelul 6-1, pe schema electrică a plăcii de dezvoltare Low Pin Count Demo Board cât și pe PCB-ul plăcii de dezvoltare. Care e rolul componentelor: C1, C2, R3-R6, R2, R7, R1, RP1, DS1 - DS4?
- Creați un proiect cu numele ProiectTest într-un director cu același nume și adăugați un fișier main.c care să conțină scheletul unui fișier *.c.
- Realizați următoarele transformări

$$123_{10} = \dots\dots\dots_2$$

$$01100011_2 = \dots\dots\dots_{10} = \dots\dots\dots_{16}$$

$$D6_{16} = \dots\dots\dots_2 = \dots\dots\dots_{10}$$

$$F13B_{16} = \dots\dots\dots_{10}$$

$$11010110_2 = \dots\dots\dots_{10} = \dots\dots\dots_{16}$$

$$54_{10} = \dots\dots\dots_2 = \dots\dots\dots_{16}$$

$$54_{16} = \dots\dots\dots_2 = \dots\dots\dots_{10}$$

- Scrieți tabelul de adevăr pentru următoarele porți logice cu 2 intrări: ȘI, SAU și SAU-EXCLUSIV.

2. Embedded C

2.1. Introducere

Pentru a realiza controlul unui sistem incorporat cu microcontroler, avem nevoie de un cod software care să ruleze pe acesta. Programul trebuie să se folosească de resursele hardware disponibile, colectând informații prin intermediul intrărilor (push-buttons, senzori digitali sau analogici, rețele de comunicație ș.a.), în funcție de acestea controlând ieșirile (leduri, display LCD, motoare ș.a.). Pentru a realiza acest lucru, fiecare procesor oferă un set de instrucțiuni de bază, numite instrucțiuni mașină, cu ajutorul cărora se pot crea aplicații. Deși aplicațiile pot fi scrise direct, folosind instrucțiuni mașină, numărul redus de instrucțiuni existente duce la o complexitate mare a codului. Din acest motiv, majoritatea aplicațiilor sunt scrise în limbaje de programare de nivel înalt, folosindu-se un compilator pentru a transforma codul scris de noi, în limbaj mașină. Cel mai răspândit limbaj folosit în aplicațiile embedded este ANSI C.

Pentru a ne familiariza cu sintaxa limbajului C și cu elementele specifice programării embedded, în cele ce urmează vom analiza și explica un scurt exemplu de cod. Mai multe informații despre limbajul C și particularități ale programării embedded puteți găsi în Anexa 1.

EXEMPLUL 01:

```
#include "SI_L03_ex_01.h"

/* defines of constants and macros */
#define NUMAR_MAGIC 32
#define SUMA(a,b) ((a)+(b))

/* variable definitions */
unsigned int suma_01, suma_02;

/* function declarations */
void functie_01(void);
unsigned int suma(unsigned char b, unsigned int a);

/* function definitions */
void main()
{
    unsigned char numar_01;
    unsigned int numar_02;

    numar_01 = 7;
    numar_02 = 15;
    suma_01 = SUMA(7,NUMAR_MAGIC);
    suma_02 = suma(numar_01, numar_02);

    if(suma_01 > suma_02)
    {
        functie_01();
    }
    else
    {
        suma_01 = suma_02 + NUMAR_MAGIC;
    }

    while(1)
    {
        ;
    }
} /* end main() function */
```

```
void functie_01(void)
{
    ; /* do nothing */
}
```

```
unsigned int suma(unsigned char b, unsigned int a)
{
    unsigned int c;
    c = a + b;
    return c;
}
```

2.2. Sintaxa limbajului C

2.2.1. Comentarii

Folosindu-ne de exemplul anterior, putem începe să analizăm codul și să identificăm componentele principale.

Probabil cel mai la îndemână element al sintaxei C sunt comentariile. Un comentariu este un text introdus în cod pentru a adăuga explicații suplimentare sau pentru a delimita părțile componente ale unui cod. În exemplul anterior se pot observa mai multe comentarii, acestea fiind textele începute cu `/*` și încheiate cu `*/`. Comentariile de această formă se pot întinde pe mai multe rânduri, atâta timp cât sunt cuprinse între cele două delimitatoare:

```
/* tot acest text
este un comentariu pe maimulterânduri */
```

Mai există și posibilitatea de a folosi comentarii de o linie. Acestea sunt începute cu `//` și se încheie unde se termină rândul respectiv:

```
// acestaeste un comentariu de un rând
// pe noulrând, delimitatorulfiindfolosit din nou
```

Se recomandă folosirea comentariilor pentru a adăuga explicații suplimentare asupra funcționalității implementate într-un cod. O documentare bună a codului duce la o înțelegere mai ușoară și mai rapidă în

cazul în care codul este folosit de o altă persoană sau dacă este revizuit după o perioadă mai lungă de timp.

2.2.2. Directive de pre-procesare

Continuăm analiza exemplului cu codul folosit pe primul rând:

```
#include "SI_L03_ex_01.h"
```

Acest rând este o directivă de pre-procesare. Înainte ca un fișier să fie compilat, are loc etapa de pre-procesare. În acest moment, toate directivele de pre-procesare, adică toate rândurile ce încep cu #, sunt înlocuite cu cod C normal. Spre exemplu, directiva *#include* va fi înlocuită cu întreg conținutul fișierului scris între ghilimele. La includerea unui fișier, în loc de ghilimele, se pot folosi și semnele mai mare și mai mic după cum urmează: *<SI_L03_exemplu_01.h>*.

În codul din exemplu se mai pot observa alte directive de preprocesare:

```
#define NUMAR_MAGIC 32
```

Acest tip de instrucțiune (*#define*) duce la înlocuirea numelui simbolic din stânga (*NUMAR_MAGIC*) cu codul din dreapta (*32*). Astfel, codul *a = NUMAR_MAGIC* este echivalent cu *a = 32*. Și atunci, întrebare evidentă este de ce să mai complicăm codul folosind un nume simbolic în locul valorii numerice? Să presupunem că, spre deosebire de exemplul anterior, constanta *32* este folosită în mai multe locuri în cod, în mai multe fișiere. Dacă, după un anumit timp, decidem să schimbăm valoarea *32* cu *77*? În acest moment, va trebui să căutăm prin tot codul locurile unde am folosit numărul *32* și să îl înlocuim cu *77*. Dacă definim un nume simbolic pentru valoarea *32*, precum în exemplu, este suficient să schimbăm valoarea într-un singur loc, pre-procesarea înlocuind apoi în tot codul. Când folosim acest tip de definire spunem despre numele simbolic asignat că este o constantă.

Alt mod pentru a crea constante este prin folosirea calificativului *const*. Explicații despre acest calificativ se pot găsi în Anexa 1. În cele ce urmează vom analiza un scurt exemplu în care sunt folosite cele două tipuri de constante. În imaginile de mai jos putem observa în partea stângă codul C iar în partea dreaptă instrucțiunile în cod mașină rezultate după compilare.

Figura 2-1: Comparație între *#define* și *const*

Pentru lucru cu variabile int - 16 biți		
<pre> /* defines of constants and macros */ #define MACRO 32 /* variable definitions */ unsigned const int constant = 8; /* function definitions */ void main() { unsigned int variable = 7; /* using MACRO*/ sum_01 = variable + MACRO; /* using const*/ sum_02 = variable + constant; while(1); } /* end main function */ </pre>		<pre> 10: /* using MACRO*/ 11: sum_01 = variable + MACRO; 7E2 0876 MOVE 0x76, W 7E3 3E20 ADDLW 0x20 7E4 00F0 MOVWF 0x70 7E5 0877 MOVE 0x77, W 7E6 1803 BTFSC 0x3, 0 7E7 3E01 ADDLW 0x1 7E8 3E00 ADDLW 0 7E9 00F1 MOVWF 0x71 12: /* using const*/ 13: sum_02 = variable + constant; 7EA 3001 MOVLW 0x1 7EB 0084 MOVWF 0x4 7EC 118A BCF 0xa, 0x3 7ED 2002 CALL 0x2 7EE 118A BCF 0xa, 0x3 7EF 00F4 MOVWF 0x74 7F0 118A BCF 0xa, 0x3 7F1 2002 CALL 0x2 7F2 118A BCF 0xa, 0x3 7F3 00F5 MOVWF 0x75 7F4 0874 MOVE 0x74, W 7F5 0776 ADDWF 0x76, W 7F6 00F2 MOVWF 0x72 7F7 0875 MOVE 0x75, W 7F8 1803 BTFSC 0x3, 0 7F9 0A75 INCF 0x75, W 7FA 0777 ADDWF 0x77, W 7FB 00F3 MOVWF 0x73 </pre>
Pentru lucru cu variabile char - 8 biți		
<pre> /* defines of constants and macros */ #define MACRO 32 /* variable definitions */ unsigned char sum_01=0, sum_02=0; /* function definitions */ void main() { unsigned char variable = 7; /* using MACRO*/ sum_01 = variable + MACRO; /* using const*/ sum_02 = variable + constant; while(1); } /* end main function */ </pre>		<pre> 10: /* using MACRO*/ 11: sum_01 = variable + MACRO; 7EE 0873 MOVE 0x73, W 7EF 3E20 ADDLW 0x20 7F0 00F2 MOVWF 0x72 7F1 0872 MOVE 0x72, W 7F2 00F0 MOVWF 0x70 12: /* using const*/ 13: sum_02 = variable + constant; 7F3 3001 MOVLW 0x1 7F4 0084 MOVWF 0x4 7F5 118A BCF 0xa, 0x3 7F6 2002 CALL 0x2 7F7 118A BCF 0xa, 0x3 7F8 0773 ADDWF 0x73, W 7F9 00F2 MOVWF 0x72 7FA 0872 MOVE 0x72, W 7FB 00F1 MOVWF 0x71 </pre>

Din acest exemplu (imaginea de mai sus) este evident că folosirea unei constante prin *#define* produce un cod mai rapid, atât în cazul în care constantele se adună cu variabile care au 16 biți (cazul de sus) cât și în cazul în care se adună cu variabile pe 8 biți (cazul de jos).

Ca și exemplu, pentru lucrul cu variabile pe 8 biți, instrucțiunea C:

sum_01 = variable + MACRO;

este alcătuită din 5 instrucțiuni de asamblare în timp ce instrucțiunea

```
sum_02 = variable + constant;
```

este alcătuită din 9 instrucțiuni de asamblare. Acest lucru se datorează faptului că valoarea constantei *constant* trebuie adusă de la adresa din memorie de unde este salvată, în timp ce valoarea *MACRO* este prezentă în op-codul instrucțiunii. Iată deci că rularea primei variante este aproape de două ori mai rapidă.

În schimb, aceste constante au și dezavantajul de a nu avea un tip de date (ex: *unsigned int*) și deci compilatorul nu poate verifica dacă sunt folosite în mod corespunzător.

În exemplul inițial se mai poate observa un loc unde este folosită directiva *#define*:

```
#define SUMA(a,b) ((a)+(b))
```

Aceasta are același rezultat prezentat anterior, doar că macroul definit primește și parametrii. De exemplu, linia de cod:

```
suma_01 = SUMA(7,NUMAR_MAGIC);
```

va arăta astfel după preprocesare:

```
suma_01 = ((7)+(32));
```

Spunem despre *SUMA* că este un macro precum o funcție (function-like macro) și are ca avantaje o execuție mai rapidă față de o funcție normală.

2.2.3. Variabile

Un alt element de bază al sintaxei C sunt variabilele. O variabilă este, de fapt, un spațiu alocat în memoria volatilă (RAM) ce poate fi accesat de-a lungul aplicației printr-un nume simbolic. Înainte de a putea folosi o variabilă, aceasta trebuie declarată. În exemplul prezentat, sunt declarate mai mult variabile:

```
unsigned int suma_01, suma_02;  
unsigned char numar_01;  
unsigned int numar_02;
```

Pentru a declara o variabilă, trebuie specificat tipul acesteia și un nume simbolic. Variabilele declarate de noi sunt de tip *unsigned int* și *unsigned char*. De fapt, tipul lor este fie *int* fie *char*, calificativul *unsigned* însemnând că acestea nu au semn. Atribuirea unei valori negative unei variabile *unsigned* nu are sens și va produce rezultate neașteptate. Declarând o variabilă de tip *char*, compilatorul va rezerva în memorie un spațiu de 8 biți. Dacă acea variabilă primește calificativul *unsigned*, ea va putea primi valori în intervalul 0 - 255 (0x00 - 0xFF). Aceeași variabilă declarată *signed* va avea valori cuprinse între -128 și 127. O variabilă *signed char* va folosi doar 7 biți pentru a salva valoarea și un bit pentru semn. Astfel, valoarea 0x8F (0b1000_1111) pentru o variabilă fără semn reprezintă 143 în decimal, în timp ce pentru o variabilă cu semn este -113. Prin analogie, aceste reguli se aplică și variabilelor ce ocupă mai mulți biți în memorie.

În tabelul următor sunt prezentate tipurile de date întregi și valorile limită:

Tabel 2-1: Tipuri de date întregi

Tip de date	Număr biți	unsigned range	signed range
char	8 biți	0 - 255	-128 - 127
short	16 biți	0 - 65535	-32768 - 32767
int	16 biți	0 - 65535	-32768 - 32767
long	32 biți	0 - 4294967295	-2147483648 - 2147483647

Observație 1: Calificativul *signed* este unul implicit. Astfel, declarând o variabilă folosind doar tipul de date, de exemplu *char*, este echivalent cu a o declara *signed char*.

Observație 2: Mărimea variabilelor *int* poate să difere în funcție de platforma pentru care scriem codul și de compilatorul folosit, având fie 16 (pentru compilatorul XC8), fie 32 de biți (pentru compilatorul XC32). Pentru a evita rezultatele neașteptate, este recomandată cunoașterea exactă a mărimii variabilelor *int* sau folosirea variabilelor *short* sau *long*. Pentru platforma folosită în acest material, variabilele *int* ocupă 16 biți.

Pentru operații mai complexe există și posibilitatea de a declara variabile cu virgulă mobilă. Aceste tipuri sunt *float* și *double* și mărimea lor este, de regulă, 32 respectiv 64 de biți. Reprezentarea numerelor în virgulă mobilă este mai complexă decât cea a numerelor întregi, motiv pentru care nu va fi explicată în acest material.

2.2.4. Funcții

O funcție este o bucată delimitată de cod ce îndeplinește o sarcină specifică și poate fi executată din mai multe puncte ale aplicației. Funcțiile sunt folosite pentru a împărți codul aplicației în mai multe subrutine generice. Spre exemplu, dacă într-un program avem nevoie să calculăm de mai multe ori rădăcinile unei ecuații de gradul al doilea, în loc să scriem de fiecare dată toate calculele pentru aflarea acestora, vom crea o funcție generică ce primește ca parametri de intrare ecuația și returnează rădăcinile, aceasta urmând să fie chemată din codul principal de câte ori este nevoie.

În exemplul dat, se pot observa trei funcții: *functie_01*, *suma* și *main*. Putem observa pentru *functie_01* și *suma* că acestea apar de mai multe ori în cod. Prima folosire a acestora este momentul în care sunt declarate:

```
void functie_01(void);  
unsigned int suma(unsigned char b, unsigned int a);
```

Se observă că în momentul în care funcțiile sunt declarate, nu este specificat și codul acestora. Declarația unei funcții implică doar stabilirea numelui acesteia, tipului returnat și tipurile datelor primite ca și parametri de intrare. Tipul de date returnat este specificat înainte de numele generic al funcției iar parametri sunt specificați în interiorul parantezelor, după numele funcției. Dacă dorim ca o funcție să primească mai mulți parametri, aceștia vor fi separați prin virgulă.

Pentru prima funcție, numele este *functie_01*, tipul de date returnat este *void*, iar funcția primește ca și parametru de intrare tot tipul *void*. Prin *void* ca tip returnat înțelegem că funcția nu returnează nicio valoare iar *void* ca și parametru de intrare înseamnă că funcția nu primește niciun parametru.

Spre deosebire de *functie_01*, *suma* returnează o variabilă de tip *unsigned int* și primește doi parametri, unul *unsigned char* iar al doilea *unsigned int*.

Al doilea moment în care întâlnim cele două funcții, putem observa că suntem în codul propriu-zis. Aici spunem despre funcții că sunt apelate sau chemate. Chiar dacă încă nu am definit codul funcțiilor, acestea au fost declarate în prealabil și astfel compilatorul poate verifica dacă ele sunt apelate cu tipurile de date corecte. Fără declarațiile de la începutul codului, acest lucru nu ar fi fost posibil, compilatorul returnând erori la întâlnirea funcțiilor.

Al treilea loc în care funcțiile sunt întâlnite este momentul în care acestea sunt definite. Se observă că numele, tipul returnat și cel al parametrilor sunt specificați și aici, fiind identice cu declarația. În cazul în care, din greșeală, apar diferențe între declarație și definiție, compilatorul ne returnează o eroare specificând acest lucru. Definiția unei funcții diferă de declarație prin faptul că este dat și corpul funcției. Codul ce compune corpul unei funcții este cuprins între două acolade `{ }`. În cazul în care avem o funcție care returnează un tip de dată, ultima instrucțiune a funcției trebuie să fie instrucțiunea *return* urmată de o variabilă sau valoare de tipul specificat.

O a treia funcție poate fi observată în cod:

```
void main()
```

Aceasta este o funcție specială, reprezentând locul de unde pornește aplicația. Orice aplicație trebuie să aibă o funcție *main*, aceasta fiind prima care se cheamă când aplicația rulează. Se poate observa că această funcție este direct definită, fără să fie și declarată. Acest lucru este posibil deoarece funcția nu este chemată explicit din alte părți ale programului.

2.2.5. Operatori

Pentru a realiza diversele sarcini ale unei aplicații, limbajul de programare pune la dispoziție un set fix de operatori. Cel mai des folosiți sunt operatorii aritmetici, aceștia fiind foarte asemănători cu operațiile matematice elementare:

Tabel 2-2: Operatori aritmetici

Operație	Simbol	Sintaxă
Atribuire	=	a = b
Adunare	+	a + b
Scădere	-	a - b
Înmulțire	*	a * b
Împărțire	/	a / b
Modulo	%	a % b
pre-incrementare	++	++a
post-incrementare	++	a++
pre-decrementare	--	--a
post-decrementare	--	a--

Atribuirea este cea mai simplă operație disponibilă, prin aceasta asignându-se valoarea din dreapta semnelui egal, variabilei din stânga acestuia. În partea dreaptă putem avea fie o valoare directă, fie o variabilă, fie o expresie compusă din mai multe operații.

Adunarea, scăderea și înmulțirea sunt asemenea operațiilor matematice, cu o singură mențiune: în momentul în care acestea sunt folosite trebuie să se țină cont de mărimea operanzilor. În cazul în care se face, spre exemplu, o adunare între două numere pe 16 biți, rezultatul ar putea depăși limita maximă a unei variabile pe 16 biți. În cazul în care rezultatul nu este salvat într-o variabilă cu dimensiuni mai mari, parte din acesta se va pierde.

Împărțirea pe întregi atribuită tot unui întreg va duce doar la păstrarea părții întregi a rezultatului. Împărțind pe întregi 10 la 3 va avea ca rezultat valoarea 3. Strâns legată de împărțire este și operația modulo, care are același rezultat ca și operația matematică. Astfel, 10 modulo 3 va avea ca rezultat valoarea 1.

Ultimii operatori rămași se comportă asemănător și sunt numiți operatori unari, deoarece se aplică unui singur operand. Folosirea operatorilor de incrementare sau decrementare duce la adunarea sau scăderea valorii 1 din operand iar poziționarea operatorului, post sau pre operand, duce la schimbarea ordinii operațiilor într-o expresie. De exemplu, pre-incrementarea duce mai întâi la modificarea operandului și apoi la folosirea

acestuia în expresie, în timp ce post-incrementarea duce la folosirea operandului în expresie și doar apoi la modificarea valorii acestuia.

Pe lângă operatorii aritmetici, limbajul mai pune la dispoziție și operatori de comparație și operatori logici:

Tabel 2-3: Operatori de comparație și operatori logici

Operație	Simbol	Sintaxă
Egal	==	a == b
nu este egal	!=	a != b
mai mare	>	a > b
mai mic	<	a < b
mai mare sau egal	>=	a >= b
mai mic sau egal	<=	a <= b
Negare	!	! a
ȘI logic	&&	a && b
SAU logic		a b

Operatorii de comparație pot fi folosiți pentru a lua decizii pe parcursul aplicației. Aceștia pot fi folosiți atât pentru a compara variabile cât și constante sau direct expresii. Rezultatul unei operații de comparație este egal cu 1 dacă rezultatul este adevărat sau 0 dacă rezultatul este fals.

Operatorii logici sunt folosiți pentru a înlanțui mai multe expresii de comparație. Un ȘI logic între mai multe expresii are rezultatul 1 doar dacă toate rezultatele acelor expresii sunt 1. Un SAU logic între mai multe expresii va avea rezultatul 0 doar dacă toate rezultatele expresiilor sunt 0.

A treia categorie de operatori este cea a operatorilor pe biți.

Tabel 2-4: Operatori pe biți

Operație	Simbol	Sintaxă
negare pe biți	~	~a
ȘI pe biți	&	a & b
SAU pe biți		a b
XOR pe biți	^	a ^ b
Deplasare la stânga pe biți	<<	a << b

Deplasare la dreapta pe biți	>>	a >> b
------------------------------	----	--------

Primii patru operatori din tabel produc rezultatele operațiilor logice reprezentate dar se întâmplă la nivelul fiecărui bit al operanzilor. Având valoarea pe 8 biți 0b1010_1010, rezultatul unei negări pe biți va fi $\sim(0b1010_1010) = 0b0101_0101$.

Dacă realizăm un ȘI pe biți între valorile:

0b1100_1100 &

0b1010_1010, rezultatul va fi

0b1000_1000.

Dacă realizăm un SAU pe biți între valorile:

0b0100_1100 |

0b1010_0010, rezultatul va fi

0b1110_1110.

Operațiile logice ȘI și SAU sunt des folosite în lucrul cu anumiți biți ai unui registru prin așa numitele măști. De exemplu, dacă dorim să setăm doar bitul 3 de la o anumită adresă, fără a modifica starea celorlalți biți, vom folosi operația SAU:

```
registru = registru | 0b0000_1000;
```

Dacă dorim să ștergem doar bitul 5, fără a modifica valoarea celorlalți biți din registru, vom face o mască cu operația ȘI:

```
registru = registru & 0b1101_1111;
```

În următorul exemplu vom face o mască folosind operația ȘI pentru a verifica doar starea bitului 4 dintr-un registru (care poate fi, spre exemplu, un flag):

```
if((registru & 0b0001_0000) == 0b0001_0000)
{ ... }
```

Operațiile pentru deplasare pe biți au ca rezultat mutarea biților primului operand cu numărul de poziții egal cu al doilea operand. Astfel, valoarea 0b0000_0101 deplasată la stânga cu 3 va avea ca rezultat:

$(0b0000_0101) \ll 3 = 0b0010_1000$.

La folosirea acestor operanzi trebuie ținut cont de mărimea variabilelor. În exemplul anterior, o deplasare cu mai mult de 8 poziții pe o variabilă pe 8 biți va avea ca rezultat 0.

Observație: Deplasarea la stânga sau dreapta cu un anume număr de biți este echivalentă cu înmulțirea sau împărțirea cu puteri ale lui 2.

2.2.6. Instrucțiuni de control

Datorită faptului că o aplicație nu poate rula doar secvențial, avem nevoie de instrucțiuni care să modifice cursul de execuție al unui program. Cele mai uzuale astfel de instrucțiuni vor fi prezentate în continuare.

Instrucțiunile *if* și *else* sunt folosite de regulă în pereche după cum urmează:

```
if( expresie )
{
    ... /* cod_01 */
}
else
{
    ... /* cod_02 */
}
```

În cazul în care expresia din dreptul instrucțiunii *if* este adevărată, codul cuprins între primele două acolade se va executa (*cod_01*). În caz contrar se va executa al doilea bloc (*cod_02*). Instrucțiunile *if else* se pot combina pentru a forma multiple căi de execuție după cum urmează:

```
if( expresie_01 )
{ ... /* cod_01 */ }
else if( expresie_02 )
{ ... /* cod_02 */ }
else
{ ... /* cod_03 */ }
```


Asemănător instrucțiunilor *if else*, este instrucțiunea *switch*, aceasta deosebindu-se însă prin faptul că în loc de o expresie care să determine calea de execuție, este folosită o valoare întreagă.

```
switch( valoare )
{
    case 0: ... /* cod_01 */
        break;
    case 1: ... /* cod_02 */
        break;
    .....
    default: ... /* cod_default */
        break;
}
```

Dacă valoarea din dreptul instrucțiunii *switch* este 0, atunci *cod_01* se va executa. Dacă valoarea este 1, *cod_02* va rula. În cazul în care valoarea nu este printre cele specificate în dreptul instrucțiunilor *case*, se va executa *cod_default*. De observat este faptul că fiecare caz se încheie cu o instrucțiune *break*. Dacă aceasta nu este prezentă, codul va rula în continuare (trecând și prin alte cazuri) până la întâlnirea primului *break* sau până la încheierea blocului instrucțiunii *switch*.

În cadrul instrucțiunilor de control întâlnim și instrucțiuni de buclă precum instrucțiunea *while*.

```
while( expresie )
{
    ... /* cod */
}
```

Această instrucțiune permite executarea codului din blocul delimitat de acolade atâta timp cât expresia din paranteze este adevărată. Dacă la prima trecere prin buclă expresia nu este adevărată, codul nu se va executa niciodată. Expresia este reevaluată de fiecare dată după ce codul dintre acolade a fost executat. Dacă dorim cel puțin o execuție a codului, putem folosi instrucțiunea *do while* după cum urmează:

```
do
{
... /* cod */
} while( expresie );
```

Altă instrucțiune ce poate fi folosită pentru a crea bucle este instrucțiunea *for* după cum urmează:

```
int i;
for(i = 0; i < 10; i++)
{
... /* cod */
}
```

Prima observație în ceea ce privește această instrucțiune este faptul că avem nevoie de o variabilă declarată pentru a realiza bucla. La începutul buclei, variabilei *i* îi este atribuită valoarea 0. Condiția de execuție a buclei este *i < 10* iar după execuția codului, se incrementează variabila de control *i++*. Astfel, codul din interiorul acoladelor se va executa atâta timp cât *i* este mai mic decât 10 și anume de 10 ori.

2.3. Programare embedded

Limbajul C este un limbaj general, menit să ruleze pe orice platformă. Din aceste motive, în funcție de locul unde este folosit, apar anumite particularități.

În aplicațiile create pentru sisteme încorporate se pot observa o serie de particularități ce vor fi prezentate în cele ce urmează.

2.3.1. Buclea infinită

În cazul unui program scris special pentru un PC, nu trebuie acordată mare atenție asupra contextului unde acesta va rula, deoarece sistemul de operare se va ocupa de aceste detalii. În cazul unui sistem încorporat, unde gestiunea aceasta nu este realizată, avem nevoie să introducem în cod diverse mecanisme de control. Buclea infinită este unul din aceste mecanisme și prin folosirea ei se evită rechemarea funcției *main* de fiecare dată după ce execuția a ajuns la capătul acesteia. Spre exemplu, dacă în codul prezentat la începutul capitoului nu am fi adăugat o buclă infinită la capătul funcției *main*, programul ar fi fost reluat la nesfârșit. Dacă, spre exemplu, primele

acțiuni făcute de aplicație în funcția *main* ar fi fost inițializări hardware, acestea ar fi fost și ele reluate de fiecare dată. Este evident că acest lucru nu este un comportament adecvat. Din acest motiv, de regulă, funcționalitatea unui sistem încorporat este realizată în cadrul unui bucle infinite.

```
while( 1 ){ ... /* cod aplicație */ }
```

2.3.2. Întreruperile

O întrerupere este definită ca un mecanism hardware oferit de către platforma pe care rulează aplicația, prin care se întrerupe șirul curent de execuție și se rulează o altă bucată de cod în funcție de anumiți stimuli externi. Spre exemplu, o aplicație ar putea fi întreruptă în momentul în care este apăsat un buton pentru a executa altă bucată de cod ce tratează acest eveniment. Pentru a ne folosi de acest mecanism, trebuie mai întâi să configurăm sursa respectivă de întrerupere, în funcție de platforma folosită, iar apoi să scriem codul ce se va executa la activarea întreruperii.

O rutină de tratare a unei întreruperi este o bucată de cod, asemănătoare unei funcții, care se cheamă prin intermediul unor mecanisme hardware. Această funcție nu primește argumente și nici nu returnează nicio valoare. De obicei se urmărește ca întreruperile să se execute cât mai repede, pentru a deranja cât mai puțin aplicația principală.

Pentru platforma folosită în acest material, avem posibilitatea de a defini o singură rutină de tratare a tuturor surselor de întrerupere după cum urmează:

```
void interrupt isr(void)
```

2.3.3. Operații pe biți

Datorită faptului că mulți regiștri de configurare necesită setarea doar a anumitor biți, în aplicațiile încorporate sunt foarte importante operațiile pe biți. De exemplu, dacă dorim să scriem 1 doar în bitul trei al unui registru pe 8 biți și să lăsăm neschimbate valorile celorlalți biți, în loc să citim registrul și să îl rescriem în funcție de valoarea acestuia, e suficient să realizăm un SAU logic cu valoarea 0b0000_1000. Astfel, toate valorile în afară de cea a bitului trei vor rămâne neschimbate iar bitul trei va lua valoarea 1 indiferent de ce valoare are.

Operațiile pe biți sunt atât de des folosite încât există și denumiri specifice pentru scrierea unui bit cu o anumită valoare. Se spune despre un bit în care se scrie 1 că acesta este setat, iar când scriem 0 că este resetat.

Altă denumire des folosită este aceea de flag. Un flag este un bit, un registru sau o variabilă indicatoare de stare. Dacă spunem că flag-ul unei anumite întreruperi a fost setat, înseamnă că bitul indicator de stare corelat acelei întreruperi a luat valoarea 1.

2.4. Aplicație propusă

Pentru a putea vedea cum se comportă codul din exemplul prezentat, avem opțiunea de a îl rula folosind un debugger simulat de către mediul de dezvoltare MPLAB. Pentru a realiza acest lucru, trebuie urmați pașii:

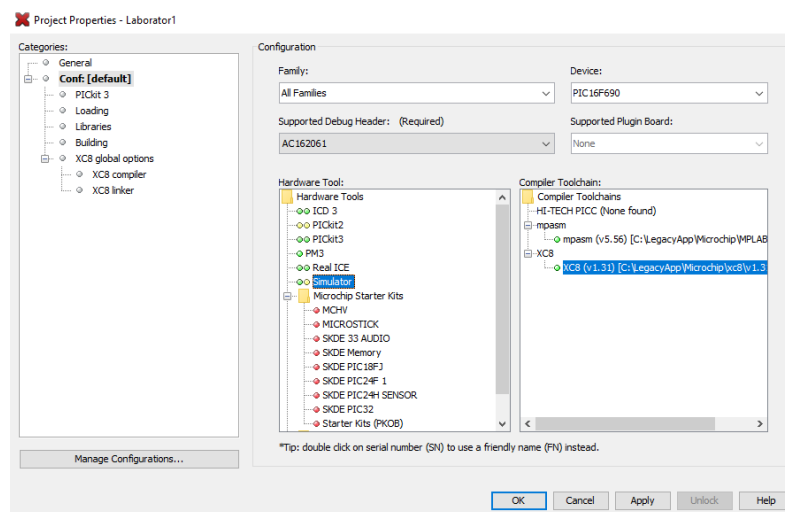
- Creai un proiect și adăugați un fișier ce conține codul prezentat la începutul acestui capitol. Compilați proiectul accesând meniul **RUN** /



Build Main Project sau butonul din bara principală.

- Click dreapta pe numele proiectului (partea stângă a mediului de dezvoltare) și click pe **Properties**. Se va deschide fereastra din figura următoare unde se va selecta ca și unealtă de debug simulatorul. Se va apăsa **Apply** și **OK**.

Figura 2-2: Selecție unealtă debug



- c) Din meniul principal, selectați **Debug**, apoi **Debug main project Tool**.
- d) Pentru a opri execuția unui program, avem posibilitatea de a selecta *breakpoints*. Acestea se adaugă prin click dublu pe o anumită linie de cod. Adăugați un *breakpoint* pe linia 18 (*numar_01 = 7;*).



- e) Din meniul debugger-ului, apăsați butonul **Run** (tasta F5) pentru a porni aplicația.
- f) În partea de jos vom avea mai multe ferestre între care putem alege, una din ele numindu-se **Variables**. Din dreptul butonului **Create new watch** selectați variabilele folosite în codul principal: *numar_01*, *numar_02*, *suma_01* și *suma_02*, cât și cele folosite în funcția *suma*: *a*, *b* și *c*. După selectarea fiecărei variabile, trebuie apăsat butonul **Create new watch**.



- g) Folosind butonul **Step Into** (tasta F7) din panoul debugger-ului, rulați aplicație până la bucla infinită și observați șirul de execuție al acestuia cât și valorile luate de către variabile.
- h) Încercați și celelalte butoane de pe panoul debugger-ului și experimentați acțiunile acestora.

2.5. Probleme propuse

- a) Descrieți ce efect are următoarea linie de cod:

const unsigned char a = 10;

- b) Scrieți rezultatul următoarelor operații cu *a = 3* și *b = 4*:

c = a+++b; a = b = c =
c = ++a+b; a = b = c =

- c) Folosind proiectul oferit ca exemplu, declarați în funcția *main* încă trei variabile de tip *unsigned int* *a*, *b* și *c*. Primele două iau valorile 0xFF00 și 0x0101. Atribuiți variabilei *c* rezultatul sumei dintre *a* și *b*. Ce valoare ia *c*?

3. Prezentare μ C

3.1. Introducere

În cadrul acestui capitol va fi prezentat pe scurt microcontrolerul PIC16F690.

3.2. Caracteristici principale – PIC16F690

a) RISC CPU:

- 35 instrucțiuni single-word.
- Toate instrucțiunile sunt „single-cycle”, exceptând instrucțiunile de salt (program branch) care sunt „two-cycle”.
- Frecvența maximă de funcționare: DC – 20MHz clock input; DC - 200ns ciclul de instrucțiune.
- Memorie program (flash) 4Kx14 words. 4K de cuvinte a câte 14 biți pentru codificarea instrucțiunilor, ceea ce înseamnă o memorie program de 7K.
- Memorie de date (RAM) 256x8 bytes. Adică, 265 de bytes (octeți) a câte 8 biți fiecare, memorie folosită pentru salvarea variabilelor.
- Configurația pinilor compatibilă fie pentru capsulă de 20 pini PDIP, fie pentru SOIC, SSOP și QFN.

b) Periferice digitale:

- Timer 0: 8-bit timer/counter cu pre-scalar pe 8 biți.
- Timer 1: 16-bit timer/counter cu pre-scalar. Numărătorul (counter) poate fi incrementat și în modul Sleep.
- Timer 2: 8-bit timer/counter cu registru de perioadă pe 8 biți, pre-scalar și post-scalar.
- Două module Enhanced Capture/Compare/PWM: Capture pe 16 biți cu rezoluție maximă 12.5ns; Compare pe 16 biți cu rezoluție maximă 200ns; PWM cu rezoluție pe 10 biți și frecvență maximă de 20KHz.
- Comunicare serial sincron prin SPI (Master mode sau Slave mode) și I2C (Master/Slave mode).
- Comunicare serial asincron/sincron prin UART/SCI cu posibilitatea de detecție 9-bit address mode. Suportă modul RS-232, RS-485 și LIN2.0.

- Memorie de date EEPROM 256x8 bytes. 256 de bytes a câte 8 biți pentru salvarea datelor în EEPROM.
- Circuit de detecție Brown-out detection pentru Brown-out Reset (BOR).

c) Periferice analogice:

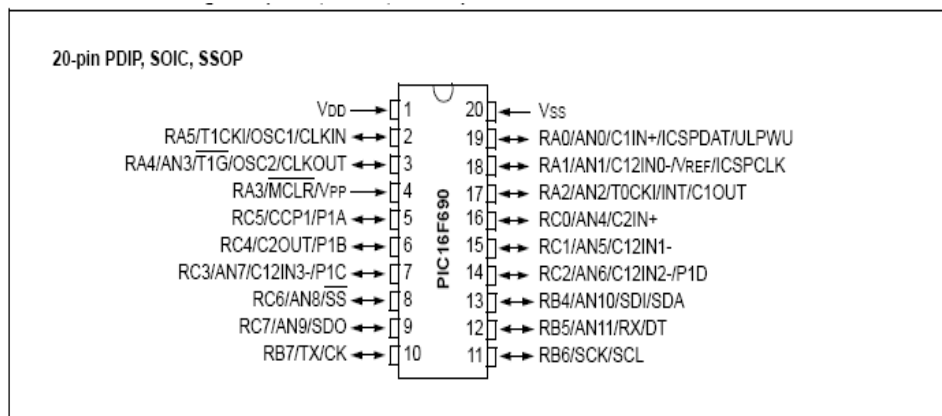
- Convertor analog-digital (A/D) pe 10 biți, pe 12 canale.
- Brown-out Reset.
- Două comparatoare analogice cu tensiune de referință programabilă și intrările selectabile printr-un multiplexor.

d) Caracteristici specifice:

- 100.000 de cicli erase/write pentru memoria de program (flash).
- 1.000.000 de cicli erase/write pentru memoria de date EEPROM.
- Memoria EEPROM menține datele nealterate (data retention) > 40 de ani.
- Programare In-circuit-serial-programming (ICSP) via doi pini.
- Necesită o singură tensiune de alimentare de 5V pentru ICSP.
- Watch Dog Timer (WDT) cu propriul circuit de oscilație on-chip de tip RC.
- Cod de protecție programabil.
- Sleep mode pentru reducerea consumului de energie.
- Diverse surse selectabile pentru oscilator.
- In-circuit-debug (ICD) via doi pini.

3.3. Diagrama pinilor și descrierea acestora

Figura 3-1: Diagrama pinilor pentru capsula 20-pin PDIP, SOIC, SSOP [8]



Placa de dezvoltare *Low Pin Count Demo Board* dispune de un microcontroler PIC16F690 cu capsulă 20-pin PDIP, în tehnologie THT (Through Hole Tehnology).

Tabel 3-1: Descrierea funcționalității pinilor [8]

Nume	Funcție	Tip intrare	Tip ieșire	Descriere
RA0/AN0/C1IN+ /ICSPDAT/ULPWU	RA0	TTL	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	AN0	AN	-	Intrarea 0 A/D.
	CIN+	AN	-	Intrarea pozitivă a comparatorului C1.
	ICSPDAT	TTL	CMOS	ICSP DATA I/O.
RA1/AN1/C12IN0- /V _{REF} /ICSPCLK	ULPWU	AN	-	Intrare de Wake-up Ultra Low-Power.
	RA1	TTL	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	AN1	AN	-	Intrarea 1 A/D.
	C12IN0-	AN	-	Intrarea negativă a comparatorului C1 sau C2.
RA2/AN2/T0CLK /INT/C1OUT	V _{REF}	AN	-	Tensiune de referință externă pentru convertorul A/D.
	ICSPCLK	ST	-	Ceas pentru ICSP.
	RA2	ST	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	AN2	AN	-	Intrarea 2 A/D.
	T0CLK	ST	-	Ceas de intrare pentru Timer0.
RA3/MCLR/V _P	INT	ST	-	Pin pentru întrerupere externă.
	C1OUT	-	CMOS	Ieșirea comparatorului C1.
	RA3	TTL	-	Pin general de intrare ieșire. Activare individuală de pull-up.
	MCLR	ST	-	Pin de RESET general cu pull-up intern.
V _{PP}	V _{PP}	HV	-	Tensiunea de programare.

RA4/AN3/T1G /OSC2/CLKOUT T	RA4	TTL	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	AN3	AN	-	Intrarea 3 A/D.
	T1G	ST	-	Intrare de validare a Timer1.
	OSC2	-	XTAL	Quartz/Rezonator.
	CLKOUT	-	CMOS	Pin de ieșire frecvență $F_{osc}/4$.
RA5/T1CLK/ OSC1/CLKIN	RA5	TTL	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	T1CLK	ST	-	Ceas de intrare pentru Timer1.
	OSC1	-	XTAL	Quartz/Rezonator.
	CLKIN	ST	-	Intrare de ceas extern/ Oscilator RC.
RB4/AN10/ SDI/SDA	RB4	TTL	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	AN10	AN	-	Intrarea 10 A/D.
	SDI	ST	-	Pin intrare SPI.
	SDA	ST	OD	Pin de date intrare/ieșire I ² C TM .
RB5/AN11/RX/ DT	RB5	TTL	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	AN11	AN	-	Intrarea 11 A/D.
	RX	ST	-	Intrarea asincronă EUART.
	DT	ST	CMOS	Pin de date sincron EUART.
RB6/SCK/SCL	RB6	TTL	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	SCK	ST	CMOS	Ceas pentru SPI.
	SCL	ST	OD	Pin de ceas I ² C TM .
RB7/TX/CK	RB7	TTL	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	TX	-	CMOS	Ieșire asincronă EUART.
	CK	ST	CMOS	Pin de ceas sincron EUART.
RC0/AN4/C2I N+	RC0	ST	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	AN4	AN	-	Intrarea 4 A/D.
	C2IN+	AN	-	Intrarea pozitivă a comparatorului C2.
RC1/AN5/C12I N1-	RC1	ST	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	AN5	AN	-	Intrarea 5 A/D.
	C12IN1-	AN	-	Intrarea negativă a comparatorului C1 sau C2.
RC2/AN6/C12I N2-/PID	RC2	ST	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	AN6	AN	-	Intrarea 6 A/D.
	C12IN2-	AN	-	Intrarea negativă a comparatorului C1 sau C2.
	PID	-	CMOS	Ieșire PWM.
RC3/AN7/C12I N3-/PIC	RC3	ST	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	AN7	AN	-	Intrarea 7 A/D.
	C12IN3-	AN	-	Intrarea negativă a comparatorului C1 sau C2.
	PIC	-	CMOS	Ieșire PWM.
RC4/C2OUT/P 1B	RC4	ST	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	C2OUT	-	CMOS	Ieșirea comparatorului C2.
	P1B	-	CMOS	Ieșire PWM.

RC5/CCP1/P1 A	RC5	ST	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	CCP1	ST	-	Intrare de captură/comparare.
	P1A	-	CMOS	Ieșire PWM.
RC6/AN8/SS	RC6	ST	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	AN8	AN	-	Intrarea 8 A/D.
	SS	ST	-	Pin de intrare Slave Select.
RC7/AN9/SDO	RC7	ST	CMOS	Pin general de intrare ieșire. Activare individuală de pull-up.
	AN9	AN	-	Intrarea 8 A/D.
	SDO	-	CMOS	Pin de ieșire date SPI.
V _{SS}	V _{SS}	Power	-	Referință masă.
V _{DD}	V _{DD}	Power	-	Alimentare pozitivă.

Legendă: AN – intrare sau ieșire analogică

TTL – pin de intrare compatibil TTL

HV – tensiune ridicată (high voltage)

CMOS – pin de intrare sau ieșire compatibil CMOS

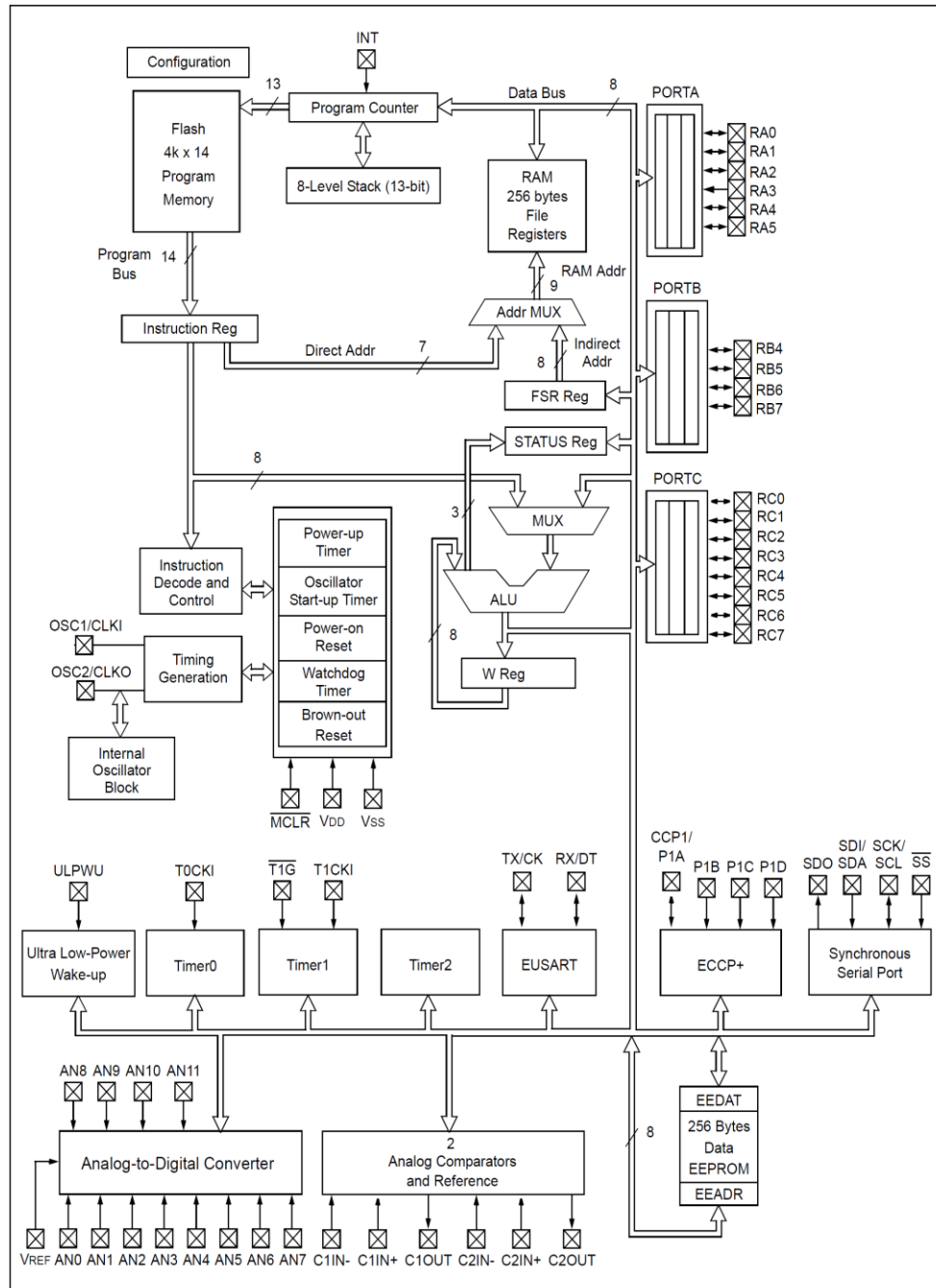
ST – pin de intrare de tip Trigger Schmitt cu nivele logice CMOS

XTAL – cristal

OD – open drain

3.4. Arhitectura microcontrolerului PIC16F690

Figura 3-2: Arhitectura PIC16F690 [8]

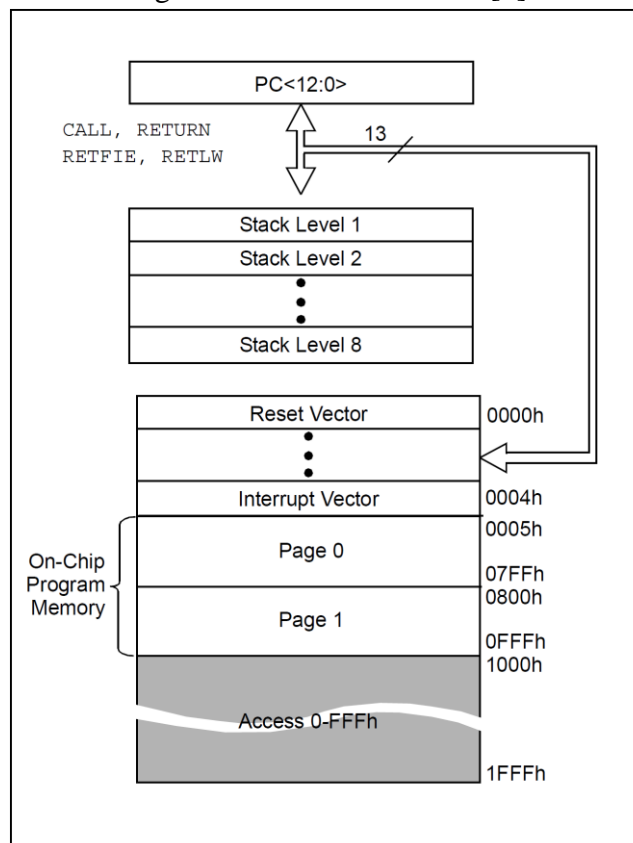


În figura de mai sus se observă:

- Este o arhitectură de tip Harvard.
- Bus-ul de date are o lăţime de 8 biţi şi la el sunt conectate CPU-ul, perifericele (porturile, modulele timer, memoria EEPROM, ADC, USART, etc...) şi memoria RAM.
- Bus-ul de instrucţiuni are o lăţime de 14 biţi şi este situat între memoria Flash (care conţine instrucţiunile codate pe 14 biţi) şi procesor (CPU).
- CPU-ul are o stivă cu o adâncime de 8 cuvinte a câte 13 biţi.
- ALU realizează operaţii aritmetice între 2 operanzi. Primul operand este registrul de lucru W iar al doilea operand poate fi furnizat fie de bus-ul de date, fie direct din conţinutul instrucţiunii.

3.5. Harta memorie

Figura 3-3: Harta memoriei [8]



PIC16F690 are un PC (Program Counter) pe 13 biți, capabil să acceseze locațiile unei memorii de 8k. Memoria Flash conține cuvinte pe 14 biți și are 4K adrese. Memoria totală este de $4K \times 14\text{biți} = 7K\text{Byte}$.

Adresa de reset este 0h, iar vectorul de întreruperi este mapat la adresa 4h.

3.6. Probleme propuse

- a) Căutați în documentația microcontrolerului PIC16F690 adresa și conținutul următorilor regiștri: PORTA, PORTC, TRISB.

Notă: Documentația microcontrolerului se găsește pe site-ul firmei Microchip. Adresa regiștrilor este notată în capitolul Memory Organization. Pentru a afla conținutul unui registru, putem accesa capitolul INDEX din documentație, unde avem referințe directe la descrierea regiștrilor.

- b) Notați numărul și numele pinilor la care sunt conectate ledurile pe placa de dezvoltare (DS1 - DS4). Care regiștri trebuie modificați pentru a seta portul respectiv?

Notă: Pentru a controla un pin digital de intrare/ieșire trebuie modificați doi regiștri iar în cazul pinilor implicit asigurați modulului ADC, trei.

- c) Ce spațiu de memorie ocupă 64 de variabile declarate *char*, împreună cu 32 de variabile declarate *int*?
- d) Câte variabile mai pot fi declarate *float*, într-o memorie de 1kB, dacă deja au fost declarate variabilele de la punctul anterior?

Notă: Un kilobyte este egal cu 1024 de bytes (sau octeți).

- e) În care memorie (de date RAM/program Flash) alocă compilatorul spațiu de memorie în cazul în care declarăm o variabilă? Dar dacă declarăm o constantă?
- f) Câte tipuri de memorie are microcontrolerul PIC16F690? Care este rolul lor?
- g) Ce se întâmplă cu datele salvate într-o memorie RAM, după îndepărtarea tensiunii de alimentare? Dar cu datele dintr-o memorie Flash?

- h) Care din cele două tipuri de memorie RAM/Flash are un consum mai mare de curent? Care credeți că este mai rapidă și de ce?
- i) Dacă declarăm 20 de constante pe 2 octeți (16 biți), câte variabile *int* mai putem declara dacă memoria are 256 de octeți?
- j) Scrieți rezultatul următoarelor operații:

$$0xAE + 0x2A = 0x..... = 0b.....$$

$$0b01001110 + 0b01100010 = 0b..... = 0x.....$$

$$0b00110011 \& 0b00100000 = 0b..... = 0x.....$$

$$0b00001100 | 0xA1 = 0b..... = 0x.....$$

4. Pinul de ieșire (Output pin)

4.1. Introducere

În cadrul acestui capitol va fi prezentat perifericul GPIO PORT (General Purpose Input Output Port). Microcontrolerul PIC16F690 are 3 astfel de porturi: PORTA, PORTB și PORTC. PORT este un grup de k pini asociați informatic unui registru PORTx. Trebuie totuși menționat un aspect important: toate microcontrolerele din familia PIC16F6xx au în arhitectura lor astfel de porturi.

Din punct de vedere al porturilor, trei aspect majore care sunt strâns legate, stau la baza alegerii microcontrolerului pentru aplicația noastră:

- a) Numărul de pini necesari: dacă PIC16F690, care conține trei porturi, nu este suficient din punct de vedere al numărului de pini, vom fi nevoiți să alegem un alt microcontroler.
- b) Spațiu: dacă microcontrolerul dispune de mai multe tipuri de configurații de porturi (și capsule) vom alege configurația cu cei mai puțini pini, care să satisfacă nevoile proiectului. Astfel se poate reduce suprafața ocupată pe PCB.
- c) Tehnologie: dacă tehnologia permite cositorirea capsulei SMD, vom alege acest tip de capsulă în defavoarea capsule THT datorită suprafeței reduse de pe PCB și a unui cost mai mic.

4.2. Pinul de ieșire

Aproape orice pin poate fi asignat mai multor periferice, deci poate avea funcții multiple. Funcția dorită se asignează pinului printr-o configurare hardware corectă. De exemplu, pinul 2 poate avea următoarele funcții:

- RA5: pinul 5 din PORTA. Pin digital de intrare sau ieșire. Nivel TTL ca și pin de intrare și CMOS pentru configurația pin de ieșire.
- T1CKI: pin extern de ceas (clock) pentru TIMER1. Pin digital de intrare.
- OSC2: quartz (XTAL).
- CLKOUT: pin de ieșire pe care se poate vizualiza $F_{osc}/4$ (frecvența de tact/4).

După cum am precizat anterior, pinii GPIO pot fi configurați de intrare sau de ieșire. Această setare se face cu ajutorul registrului TRISx:

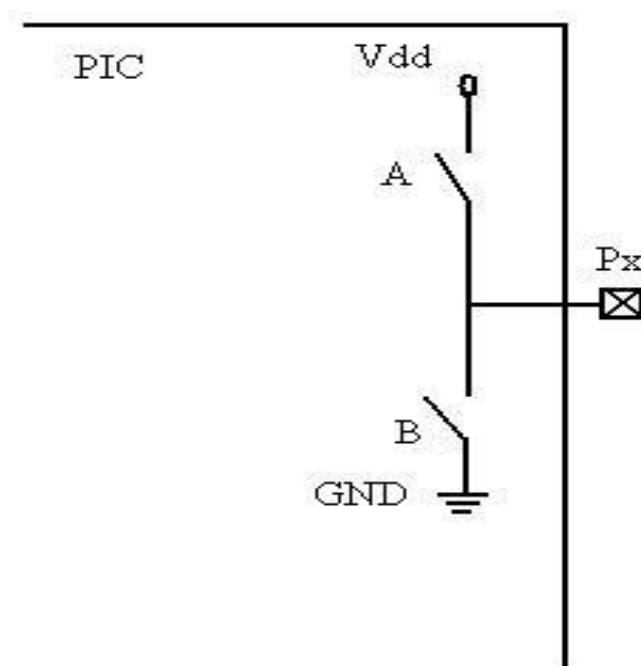
- Dacă bitul K aferent pinului K este setat 1 atunci pinul K va fi pin de intrare.
- Dacă bitul K aferent pinului K este setat 0 atunci pinul K va fi pin de ieșire.

Când pinul K este configurat ca și ieșire, atunci scriind în registrul PORTx la poziția bitului K, vom modifica starea electrică a pinului.

- Scriind 1 în registru, pe pin vom citi cu ajutorul unui multimetru 5V (1 logic).
- Scriind 0 în registru, pe pin vom citi cu ajutorul unui multimetru 0V (0 logic)

Când este configurat ca și pin de ieșire, din punct de vedere electric pinul poate fi echivalat cu următorul circuit tri-state:

Figura 4-1: Echivalarea electrică a unui pin de ieșire (Digital Output)



Vom avea următoarele cazuri:

- Contactul A închis, B deschis: La ieșirea pinului vom avea 1, sau cu alte cuvinte vom citi Vdd cu ajutorul unui multimetru.
- Contactul A deschis, B închis: La ieșirea pinului vom avea 0, sau cu alte cuvinte vom citi 0V cu ajutorul unui multimetru. Pinul e “tras” la masă.
- Contactele A și B deschise: Pinul se află în cea de a treia stare tri-state, sau stare de înaltă impedanță hi-Z.
- Contactele A și B închise: INTERZIS! Se va produce scurt-circuit care duce la distrugerea perifericului sau chiar a microcontrolerului. Această configurație nu este posibilă, deoarece hardware-ul din microcontroler nu permite acest lucru.

PORTC conține 8 pini RC0-RC7. Aceștia pot avea următoarele funcții:

Tabel 4-1: Funcțiile pinilor din PORTC [8]

Nume	Funcție	Tip Intrare	Tip Ieșire	Descriere
RC0/AN4/C2IN+	RC0	ST	CMOS	Pin digital de intrare/ieșire.
	AN4	AN	-	Pin analogic de intrare al A/D. Pinul 4.
	C2IN+	AN	-	Intrarea pozitivă a comparatorului C2.
RC1/AN5/C12IN1-	RC1	ST	CMOS	Pin digital de intrare/ieșire.
	AN5	AN	-	Pin analogic de intrare al A/D. Pinul 5.
	C12IN1-	AN	-	Intrarea negativă a comparatorului C1 sau C2.
RC2/AN6/C12IN2-/P1D	RC2	ST	CMOS	Pin digital de intrare/ieșire.
	AN6	AN	-	Pin analogic de intrare al A/D. Pinul 6.
	C12IN2-	AN	-	Intrarea negativă a comparatorului C1 sau C2.
	P1D	-	CMOS	Pin de ieșire PWM.
RC3/AN7/C12IN3-/P1C	RC3	ST	CMOS	Pin digital de intrare/ieșire.
	AN7	AN	-	Pin analogic de intrare al A/D. Pinul 7.
	C12IN3-	AN	-	Intrarea negativă a comparatorului C1 sau C2.
	P1C	-	CMOS	Pin de ieșire PWM.
RC4/C2OUT/P1B	RC4	ST	COMS	Pin digital de intrare/ieșire.
	C2OUT	-	CMOS	Pinul de ieșire al comparatorului C2.
	P1B	-	CMOS	Pin de ieșire PWM.
RC5/CCP1/P1A	RC5	ST	COMS	Pin digital de intrare/ieșire.
	CCP1	ST	-	Pinul de intrare captură.
	P1A	-	CMOS	Pin de ieșire PWM.
RC6/AN8/SS	RC6	ST	CMOS	Pin digital de intrare/ieșire.
	AN8	AN	-	Pin analogic de intrare al A/D. Pinul 8.
	SS	ST	-	Pin de selecție pentru SPI.
RC7/AN9/SDO	RC7	ST	CMOS	Pin digital de intrare/ieșire.

	AN9	AN	-	Pin analogic de intrare al A/D. Pinul 9.
	SDO	-	CMOS	Pin de ieșire de date pentru SPI.

Legendă: AN – intrare sau ieșire analogică

TTL – pin de intrare compatibil TTL

HV – tensiune ridicată (high voltage)

CMOS – pin de intrare sau ieșire compatibil CMOS

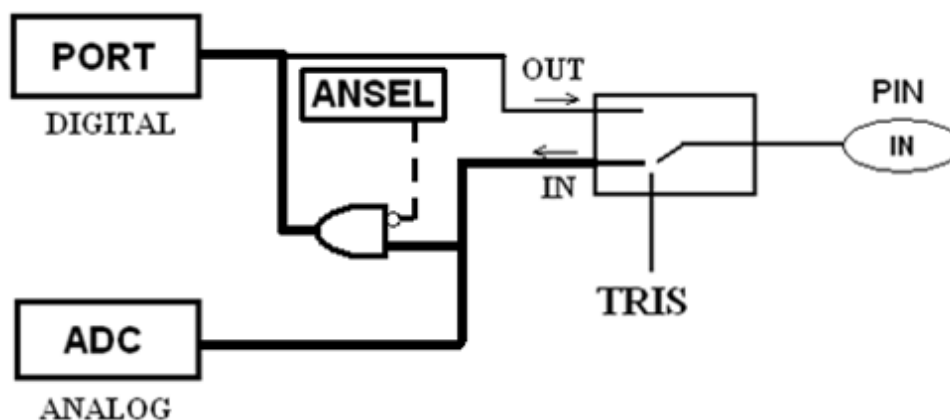
ST – pin de intrare de tip Trigger Schmitt cu nivele logice CMOS

XTAL – cristal

OD – open drain

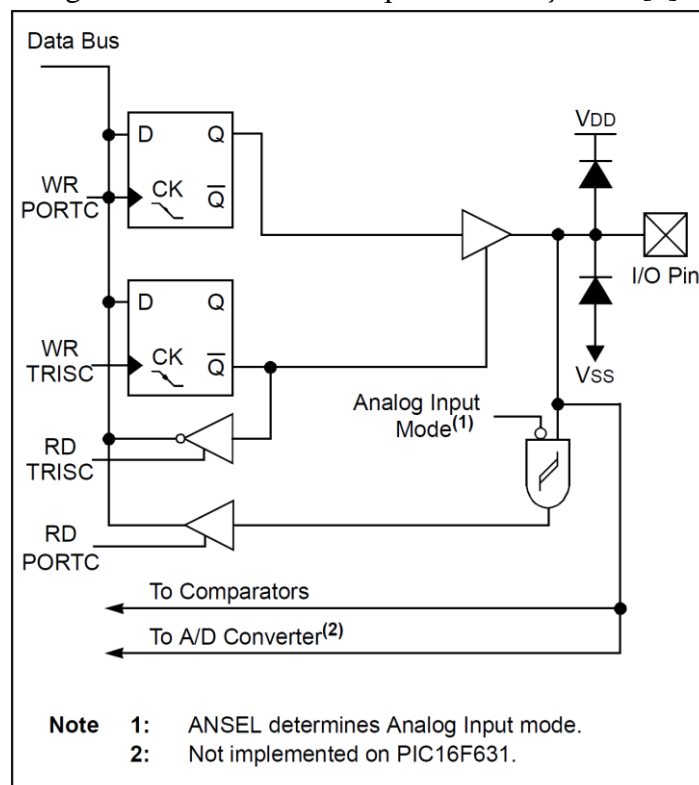
Pinii sunt asignați implicit perifericului ADC (convertorul analog-numeric) mai puțin pinul RC4 și RC5, iar direcția lor este de intrare. Dacă se dorește configurarea pinului ca și pin digital, bitul aferent din registrul ANSEL sau ANSELH (după caz), trebuie scris cu valoarea 0. Astfel semnalul aplicat pe pin va ajunge și la circuitul digital (poarta logică de tip Trigger Schmitt este validată și va avea la ieșire starea logică a semnalului aplicat pe pin). Datorită faptului că sunt 11 pini analogici și fiecare pin are un bit asignat, cei 11 biți nu vor avea loc într-un singur registru de 8 biți. Acesta este motivul pentru care există registrul ANSEL (conține primii 8 biți) și ANSELH (conține restul de biți).

Figura. 4-2: Selecția pin digital/analog



Un pin de ieșire (bitul aferent din registrul TRIS este 0) funcționează ca și pin digital și fără ca bitul aferent din registrul ANSEL să fie 0, dar este recomandat ca și în acest caz, pinul să fie setat cu funcție digitală cu ajutorul registrului ANSEL.

Figura 4-3: Schema bloc a pinilor RC0 și RC1 [8]



Pentru a seta direcția pinilor se utilizează registrul TRISC, iar pentru citirea sau scrierea portului registrul PORTC.

Tabel 4-2: Regiștri asociați cu PORTC [8]

Nume	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
ANSEL	ANS7	ANS6	ANS5	ANS4	ANS3	ANS2	ANS1	ANS0
ANSELH	-	-	-	-	ANS11	ANS10	ANS9	ANS8
CCP1CON	P1M1	P1M0	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0
CM2CON0	C2ON	C2OUT	C2OE	C2POL	-	C2R	C2CH1	C2CH0
CM2CON1	MC1OUT	MC2OUT	-	-	-	-	T1GSS	C2SYNC
PORTC	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0
PSTRCON	-	-	-	STRSYNC	STRD	STRC	STRB	STRA
SRCON	SR1	SR0	C1SEN	C2REN	PULSS	PULSR	-	-
SSPCON	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0
TRISC	TRISC7	TRISC6	TRISC5	TRISC4	TRISC3	TRISC2	TRISC1	TRISC0
VRCON	C1VREN	C2VREN	VRR	VP6EN	VR3	VR2	VR1	VR0

Legendă: x=necunoscut, u=nemodificat, -=bitul se citește ca 0. Biții închiși la culoare nu sunt folosiți de PORTC.

4.3. Limitări electrice

Curent maxim prin pinul Vss = 300mA

Curent maxim prin pinul Vdd = 250mA

Curent maxim absorbit de un pin I/O = 25mA

Curent maxim generat de un pin I/O = 25mA

Curent maxim absorbit de PORTA, PORTB și PORTC = 200mA

Curent maxim generat de PORTA, PORTB și PORTC = 200mA

4.4. Probleme propuse

- a) Studiați cazul în care legăm doi pin de ieșire între ei printr-o sârmă. Ce se întâmplă? Este permis?

Notă: Urmăriți *Figura 4-1* în cazul în care unul dintre pini e legat la masă iar al doilea la Vdd.

- b) Studiați cazul în care legăm un pin de ieșire cu unul de intrare între ei printr-o sârmă. Este permis?
- c) Câte leduri putem lega în paralel la un pin, dacă pentru a lumina este nevoie să fie străbătute de un curent de 7 mA?

Notă: Trebuie considerat curentul maxim generat de un pin.

- d) Câte leduri putem lega la un microcontroler, în funcție de configurație (Catodul sau Anodul legate la pinul portului), dacă pentru a lumina trebuie să fie străbătute de un curent de 10mA?

Notă: Dacă legăm Catodul la pinul portului, ledul se va aprinde când ieșirea acestuia e legată la masă (zero logic). În acest caz trebuie considerat curentul maxim prin pinul Vss.

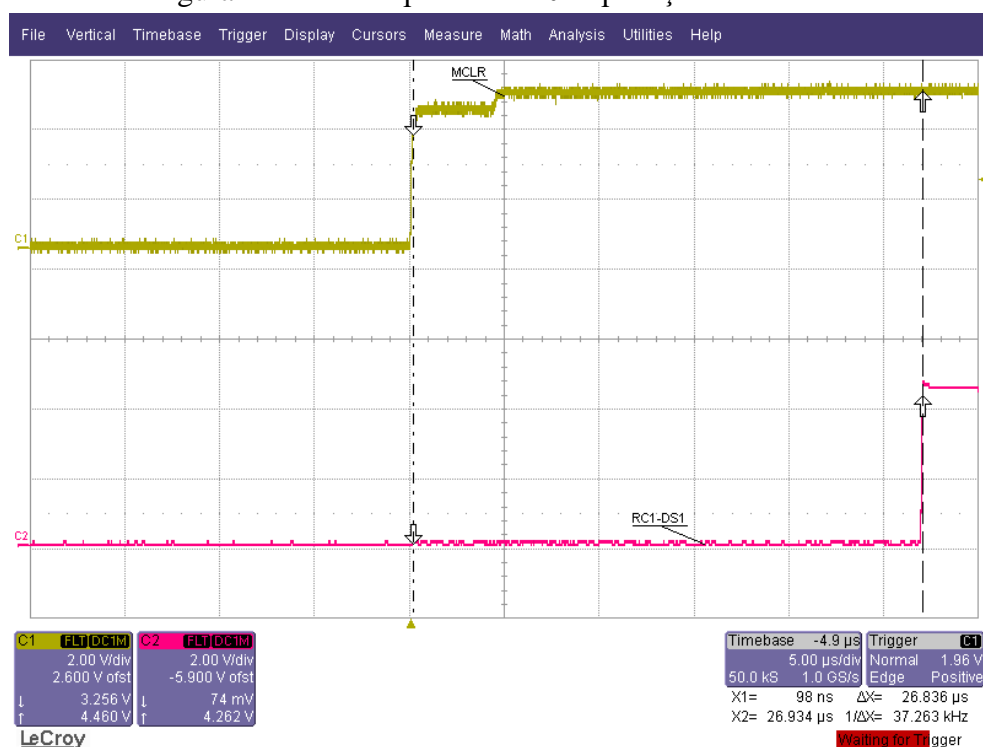
4.5. Aplicație propusă

Scrieți un mic program în care să setați pinii RC0-RC3 pini de ieșire și pinii RC4-RC7 pini de intrare. Salvați valoarea pinilor RC4-RC7 într-o variabilă (care trebuie declarată în prealabil) și aprindeți ledurile legate la pinii RC0 și RC2.

Notă: Pașii necesari scrierii programului în memoria microcontrolerului (flashing) se găsesc în Anexa 2.

În figura de mai jos se poate vizualiza rezultatul așteptat al programului. Microcontrolerul iese din reset (pinul MCLR este 1 logic) și la intrarea în funcția main, pinul RC0 va trece în starea 1 logic, care duce la aprinderea ledului DS1.

Figura 4-4: Setarea pinului RC0 după ieșirea din reset



Notă: Pe parcursul acestui material, revolvarea diverselor aplicații va fi însoțită și de oscilogramme care să exemplifice, într-un mod practic, comportamentul corect și așteptat al microcontrolerului pe care rulează programul dat ca și exemplu (versiunea completă a acestuia). Cei care nu sunt familiarizați cu interpretarea unei oscilogramme pot găsi mai multe detalii în Anexa 3.

4.6. Model Software

```
/* include files */
#include "pic.h"

/* variables */
unsigned char portRead; /* for reading the port C pin values */

/* function declarations */
void init();

/* function definitions */

void main()
{
    init();
    portRead = ??? & ???; /* use mask for reading only RC4 - RC7 pins */
    PORTC = ???; /* set RC0 and RC2 - turn on LEDS */

    while(1)
    {
        ;
    }
}

void init()
{
    ANSEL = ???; /* set RC0 to RC3 as digital pins */
    ANSELH = ???; /* set RC6 and RC7 as digital pins */

    TRISC = ???; /* RC4 to RC7 input. RC0 to RC3 output */
    PORTC = 0x00; /* port C pins reset value */
}
```

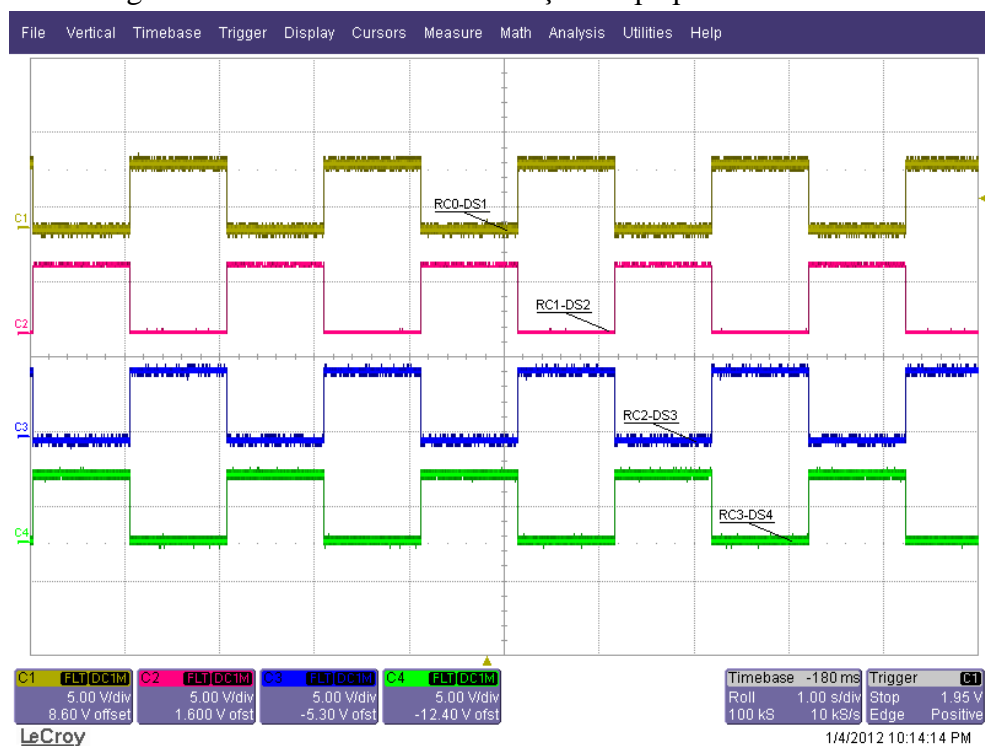
Notă: Modelul software este oferit doar ca punct de plecare. Pentru realizarea aplicației, codul trebuie completat, înlocuind semnele de întrebare cu valorile corecte.

4.7. Problemă propusă

Scrieți un program prin care să setați portul C astfel încât ledurile să afișeze alternative valoarea 0xA și 0x5 într-o buclă infinită.

În *Figura 4-5* se poate vizualiza rezultatul dorit al temei. Într-o buclă infinită în funcția *main*, pinii RC0-RC3 vor indica alternativ valorile 0x5 și 0xA.

Figura 4-5: Alternarea valorii 0x5 și 0xA pe pinii din PORTC



5. Pinul de intrare (Input pin)

5.1. Introducere

În cadrul acestui capitol va fi prezentat în continuare perifericul GPIO PORT (General Purpose Input Output Port).

Toți pinii asigurați perifericelor PORTx, sunt, după reset, pini de intrare. Acest lucru se datorează faptului că, în aplicații, la pinii microcontrolerului pot fi legați senzori, ieșirile digitale ale altor microcontrolere, circuite integrate, etc. Dacă pinii ar fi implicit de ieșire, există riscul să se producă scurt-circuite la punerea sub tensiune a circuitului, care să ducă la distrugerea microcontrolerului, așa cum se va ilustra în figurile de mai jos.

Exemplu: la pinul RB4 se leagă un senzor digital care are starea logică 1 ca în *Figura 5-1*. Presupunem că pinul este implicit de ieșire și starea logică este 0. În acest caz s-ar produce un scurt-circuit care ar duce la distrugerea microcontrolerului până ce aplicația să ruleze pentru a schimba direcția pinului, făcându-l de intrare.

Evident, acest caz poate apărea chiar și dacă pinul este implicit de intrare, printr-o eroare software, dacă pinul RB4 este declarat pin de ieșire în program. Pentru a preveni astfel de situații, legătura electrică corectă este prezentată în *Figura 5-2*. Se observă că rezistența are rolul de a limita valoarea curentului în situația nedorită. Rezistența trebuie aleasă astfel încât valoarea curentului să nu depășească valoarea maximă admisă (25mA pentru un pin digital). În cazul în care pinul este declarat corect, de intrare, rezistența nu modifică funcționarea circuitului, deoarece căderea de tensiune pe ea va fi neglijabilă (curentul absorbit de pinul de intrare este aproape 0) și pe pinul microcontrolerului vom putea măsura 5V.

Întrebare: Întâlnim și pe schema electrică a plăcii de dezvoltare astfel de protecții? Dacă da, unde anume? Ce situație nedorită se evită?

Figura 5-1: Conexiune nedorită

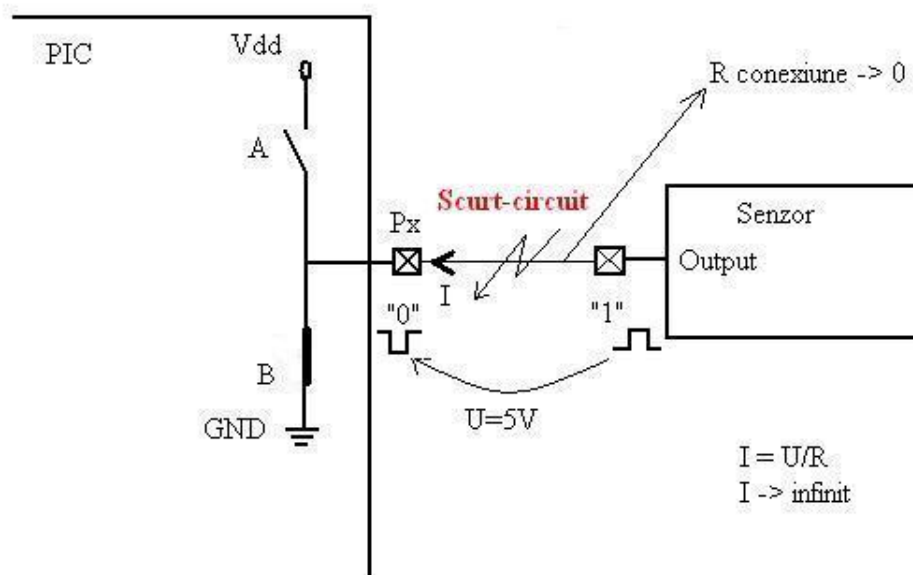
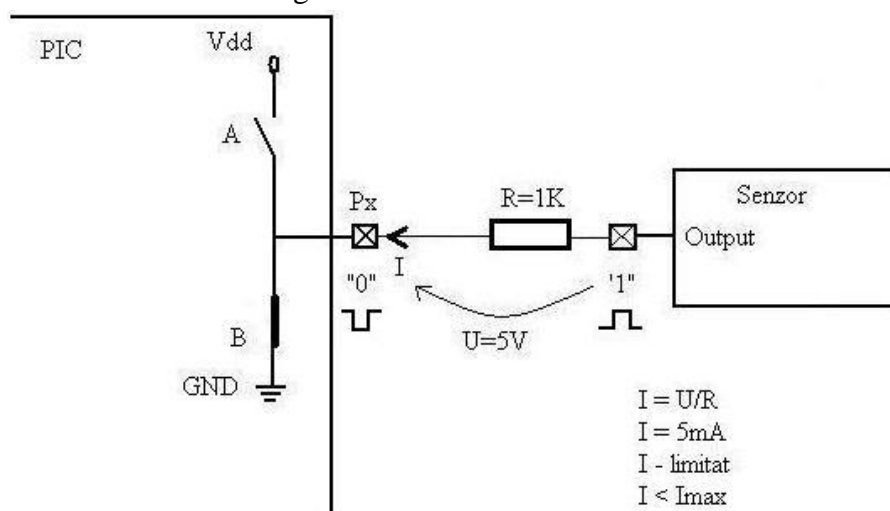


Figura5-2: Conexiune corectă



5.2. Pinul de intrare

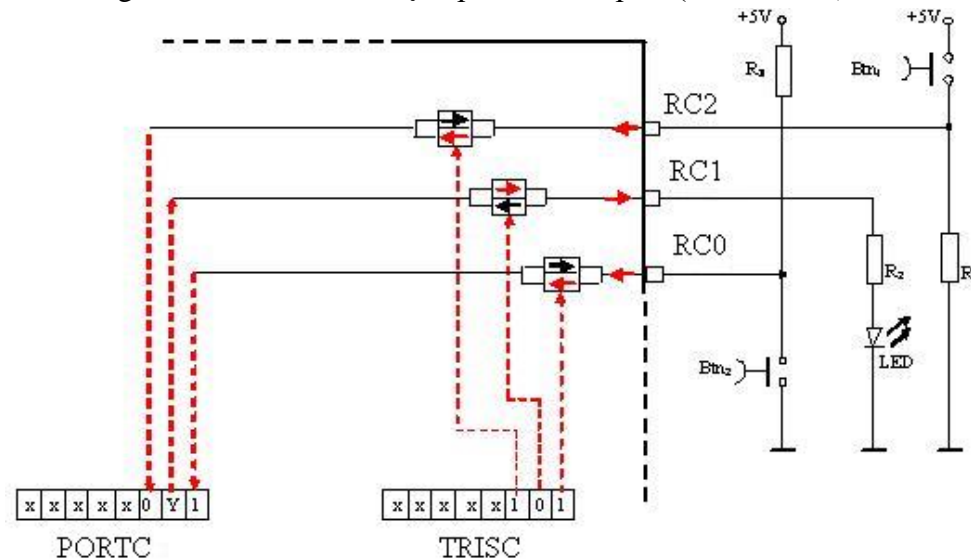
După cum am precizat și în capitolul anterior, pinii GPIO pot fi configurați de intrare sau de ieșire. Aceasta setare se face cu ajutorul registrului TRISx:

- Dacă bitul K aferent pinului K este setat 1 atunci pinul K va fi pin de intrare.
- Dacă bitul K aferent pinului K este setat 0 atunci pinul K va fi pin de ieșire.

Când pinul K este configurat ca și intrare, citind registrul PORTx la poziția bitului K, vom regăsi starea electrică (logică) a pinului de intrare.

- Când în registrul PORTx bitul k are valoarea 1, pe pinul de ieșire va fi o tensiune de 5V, măsurabilă cu un multimetru sau un osciloscop.
- Când în registrul PORTx bitul k are valoarea 0, pe pinul de ieșire va fi o tensiune de 0V, măsurabilă cu un multimetru sau un osciloscop.

Figura 5-3: Setarea direcției pinilor unui port (ex: PORTC) [3]



În exemplul de mai sus întâlnim următoarele situații:

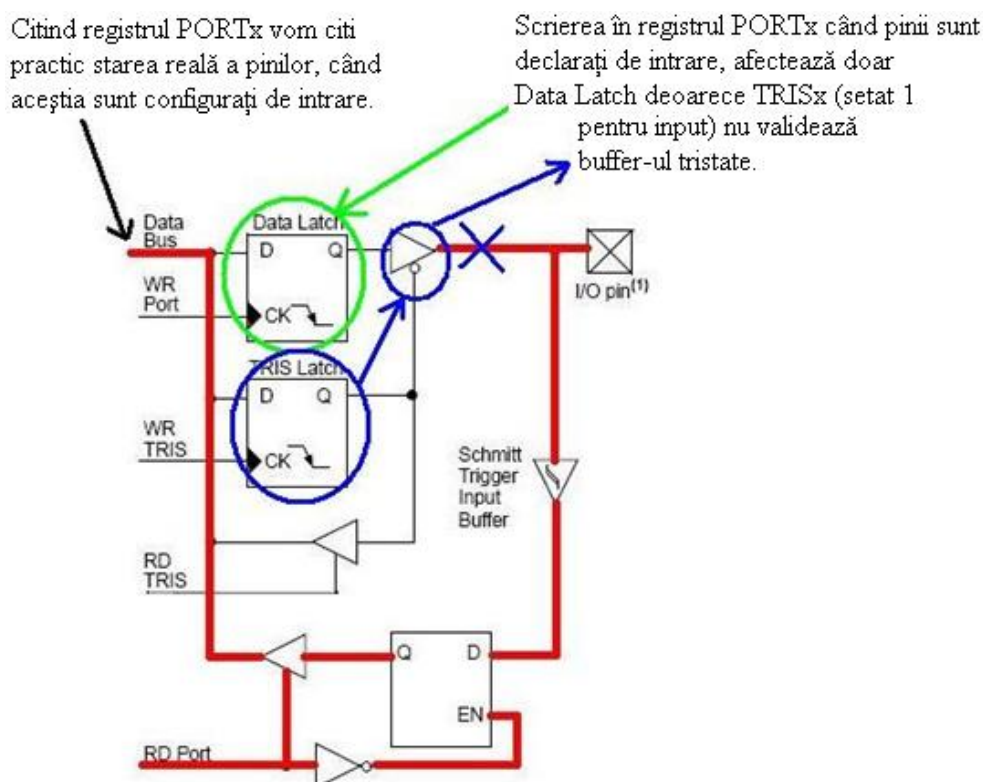
- Pinul RC0 trebuie setat ca și intrare pentru a citi starea butonului Btn2. Acest lucru se face scriind 1 pe poziția bitului 0 din registrul TRISC. Pe poziția bitului 0 din registrul PORTC se poate citi starea logică a pinului.
- Pinul RC1 trebuie setat ca și ieșire pentru a aprinde ledul. Acest lucru se face scriind 0 pe poziția bitului 1 din registrul TRISC. La poziția bitului 1 din registrul PORTC se poate scrie starea logică pe care

dorim să o aibă pinul. Pentru a aprinde ledul trebuie să scriem 1 logic. Pentru stingere vom scrie 0.

- Pinul RC2 trebuie setat ca și intrare pentru a citi starea butonului Btn1. Acest lucru se face scriind 1 pe poziția bitului 2 din registrul TRISC. Pe poziția bitului 2 din registrul PORTC se poate citi starea logică a pinului.

Când pinul K este configurat ca și pin de intrare, operațiile de scriere în registrul PORTx, la poziția bitului K nu vor avea nici un efect (vor fi ignorate). Acest lucru se datorează faptului că scrierea afectează lach-ul, pe când la citire, registrul conține informații despre starea pinilor. Cu alte cuvinte, când pinii sunt declarați de intrare, chiar dacă scriem o valoare în registrul PORTx, la citire, acesta va conține starea reală a pinilor.

Figura 5-4: Accesul de scriere și citire al unui port



Când este configurat ca și pin de intrare, din punct de vedere electric, pinul poate fi echivalat cu o rezistență de valoare foarte mare legată la masă.

Figura 5-5: Echivalarea electrică a unui pin Digital Input

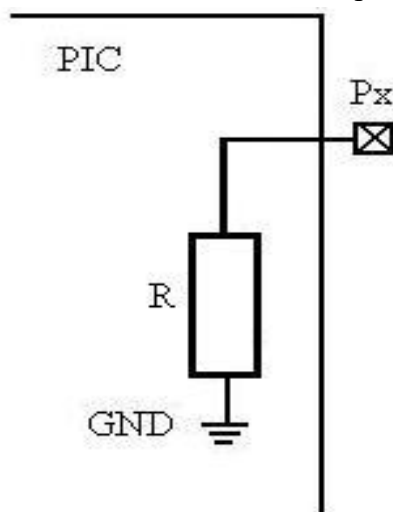
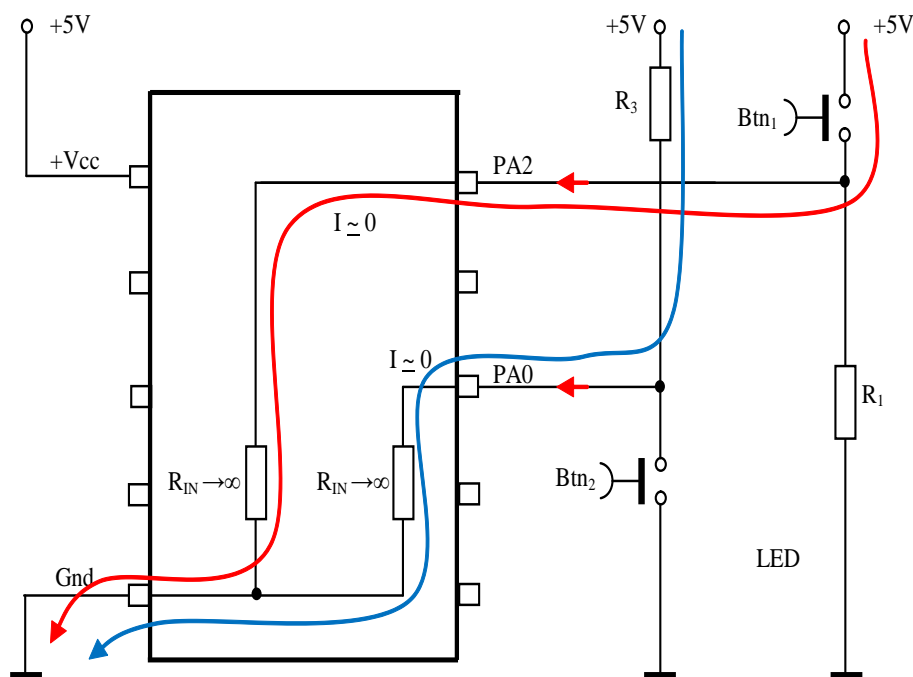


Figura 5-6: Comportamentul electric al pinilor de intrare [3]



După cum se observă în figurile de mai sus, curentul absorbit de un pin de intrare este neglijabil, fiind aproape 0. Cu alte cuvinte, un pin de intrare legat într-un circuit nu modifică funcționarea electrică a acestuia.

PORTA: conține 6 pini - RA0-RA5. Aceștia pot avea următoarele funcții:

Tabel 5-1: Funcțiile pinilor din PORTA [8]

Nume	Funcție	Tip Intrare	Tip Ieșire	Descriere
RA0/AN0/C1IN+/ICSPDAT/ULPWU	RA0	ST	CMOS	Pin digital de intrare/ieșire.
	AN0	AN	-	Pin analogic de intrare al A/D. Pinul 0.
	C1IN+	AN	-	Intrarea pozitivă a comparatorului C1.
	ICSPDAT	TTL	CMOS	Pin de date intrare/ieșire ICSP™
	ULPWU	AN	-	Pin de intrare ultra-low Wake-up.
RA1/AN1/C12IN0-/VREF/ICSPCLK	RA1	ST	CMOS	Pin digital de intrare/ieșire.
	AN1	AN	-	Pin analogic de intrare al A/D. Pinul 1.
	C12IN0-	AN	-	Intrarea negativă a comparatorului C1 sau C2.
	VREF	AN	-	Tensiune externă de referință pentru convertorul ADC.
	ICSPCLK	TTL	-	Ceas pentru ICSP™.
RA2/AN2/T0CLK/INT/C1OUT	RA2	ST	CMOS	Pin digital de intrare/ieșire.
	AN2	AN	-	Pin analogic de intrare al A/D. Pinul 2.
	T0CLK	ST	-	Pin de ceas pentru Timer0.
	INT	ST	-	Pin de întrerupere externă.
	C1OUT	-	CMOS	Pin de ieșire al comparatorului C1.
RA3/MCLR/V _{PP}	RC3	ST	CMOS	Pin digital de intrare/ieșire.
	MCLR	ST	-	Pin de reset cu pull-up intern.
	V _{PP}	HV	-	Tensiune de programare.
RA4/AN3/T1G/OCS2/CLKOUT	RC4	ST	COMS	Pin digital de intrare/ieșire.
	AN3	AN	-	Pin analogic de intrare al A/D. Pinul 2.
	T1G	ST	-	Intrare de validare a Timer1.
	OSC2	-	XTAL	Quartz/Rezonator.
	CLKOUT	-	CMOS	Pin de ieșire frecvență F _{osc} /4.
RA5/T1CLK/OSC1/CLKIN	RA5	ST	COMS	Pin digital de intrare/ieșire.
	T1CLK	ST	-	Ceas de intrare pentru Timer1.
	OSC1	XTAL	-	Quartz/Rezonator.
	CLKIN	ST	-	Intrare de ceas extern / Oscilator RC.

Legendă: AN – intrare sau ieșire analogică

TTL – pin de intrare compatibil TTL

HV – tensiune ridicată (high voltage)

CMOS – pin de intrare sau ieșire compatibil CMOS

ST – pin de intrare de tip Trigger Schmitt cu nivele logice CMOS

XTAL – cristal

OD – open drain

Pinii portului A sunt asigurați implicit perifericului ADC (convertorul analog-numeric) mai puțin pinul RA3 și RA5, iar direcția lor este de intrare. Dacă se dorește configurarea pinului ca și pin digital, bitul aferent din registrul ANSEL trebuie scris cu valoarea 0. Dacă se folosește ca și pin digital, setarea direcției pinilor se face cu ajutorul registrului TRISA, iar pentru citirea sau scrierea portului, registrul PORTA.

Tabel 5-2: Regiștri asociați cu PORTA [8]

Nume	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
ANSEL	ANS7	ANS6	ANS5	ANS4	ANS3	ANS2	ANS1	ANS0
ADCON	ADFM	VCFG	CHS3	CHS2	CHS1	CHS0	GO/DONE	ADON
CM1CON0	C1ON	C1OUT	C1OE	C1POL	-	C1R	C1CH1	C1CH0
INTCON	GIE	PEIE	TOIE	INTE	RABIE	TOIF	INTF	RABIF
IOCA	-	-	IOCA5	IOCA4	IOCA3	IOCA2	IOCA1	IOCA0
PORTC	-	-	RA5	RA4	RA3	RA2	RA1	RA0
OPTION_REG	RABPU	INTEDG	TOCS	TOSE	PSA	PS2	PS1	PS0
T1CON	T1GINV	TMR1GE	T1CKPS1	T1CKPS0	T1OCSEN	T1SYNC	TMR1CS	TMR1ON
SSPCON	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0
TRISC	-	-	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0
WPUA	-	-	WPUA5	WPUA4	-	WPUA2	WPUA1	WPUA0

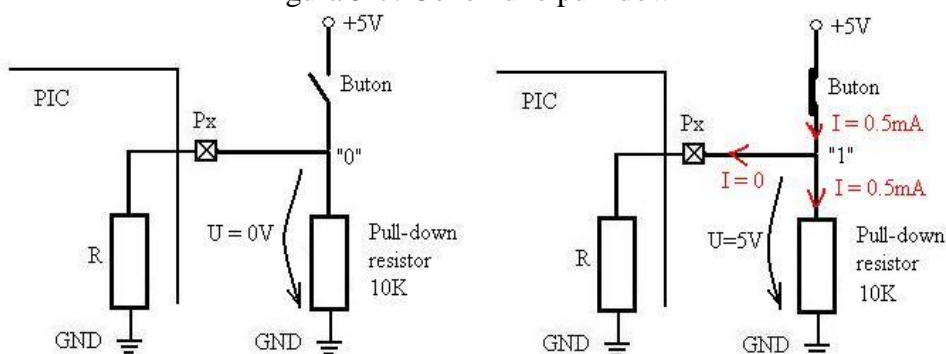
Legendă: x=necunoscut, u=nemodificat, -=bitul se citește ca 0. Biții închiși la culoare nu sunt folosiți de PORTA.

5.3. Pull-up/Pull-down

Dacă intrarea pinului este lăsată în aer, starea logică a pinului poate fi influențată de câmpuri electromagnetice. Pentru a evita această situație nedorită, pinul trebuie legat fie la V_{SS} , fie la V_{DD} . Există două posibilități:

a) Pull-down (legarea la masă)

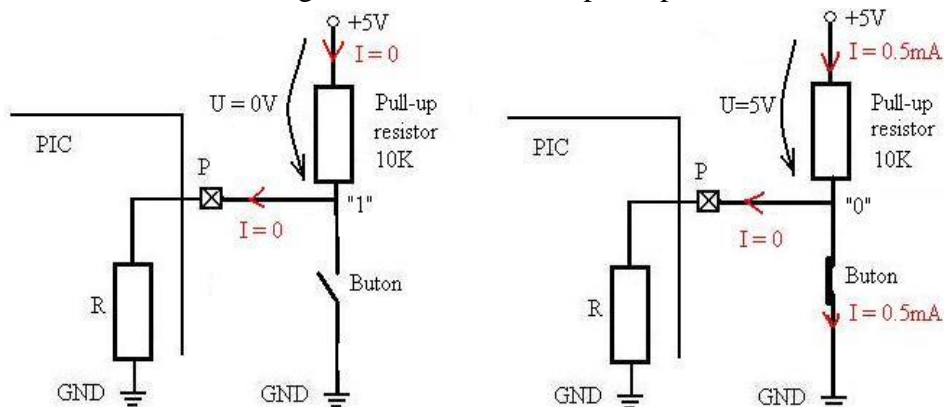
Figura 5-7: Conexiune pull-down



După cum se observă, atâta timp cât butonul nu este apăsă, potențialul pe pin este 0, deci starea logică este 0, pinul nefiind „lăsat în aer”. În momentul în care butonul este apăsă, potențialul pe pin devine +5V, deci starea logică ce va fi citită pe pin este 1 logic. Valoarea rezistenței trebuie să fie de ordinul $K\Omega$.

b) Pull-up (legarea la +5V)

Figura 5-8: Conexiunea pull-up



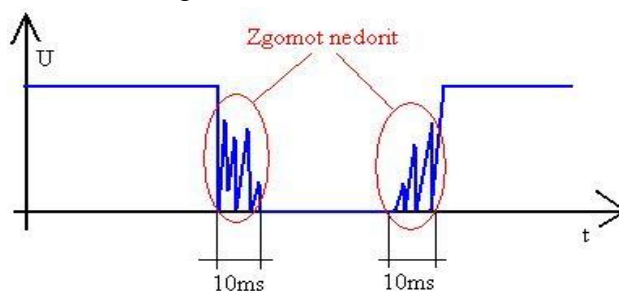
După cum se observă, atâta timp cât butonul nu este apăsat, potențialul pe pin este +5V, deci starea logică este 1, deoarece căderea de tensiune pe rezistență este 0V ($U = I \cdot R$, unde $I = 0$). În momentul în care butonul este apăsat, potențialul pe pin devine 0V, deci starea logică ce va fi citită pe pin este 0 logic. Valoarea rezistenței trebuie să fie de ordinal $K\Omega$, ca și în cazul rezistenței de pull-down.

Trebuie menționat că PORTA și PORTB dispun de conexiune pull-up internă, care poate și activată pentru pinii de intrare cu ajutorul biților din registrul WPUA pentru pinii ce alcătuiesc PORTA sau WPUB pentru pinii ce alcătuiesc PORTB.

5.4. Switch Debounce

Deși sunt foarte des folosite în aplicații datorită costului redus și a simplității, comutatoarele mecanice (push-button) au un mare dezavantaj: sunt foarte „zgomotoase”. Datorită închiderii și deschiderii contactelor apar oscilații și se formează trenuri de impulsuri parazite (Figura 5-9). Problema se numește switch bounce și încercarea de eliminare a acestor impulsuri parazite se numește switch debounce.

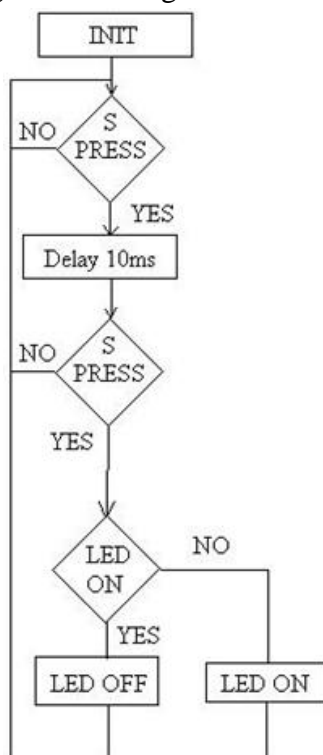
Figura 5-9: Switch bounce



Practic o apăsare fizică a unui push-button este văzută din punct de vedere electric ca o serie de apăsări. Empiric se poate determina durata acestor oscilații, ea fiind în jur de 10ms. Există mai multe soluții, fie hardware fie software, pentru switch debouncing. Cea mai convenabilă și care va fi prezentată în continuare este soluția software.

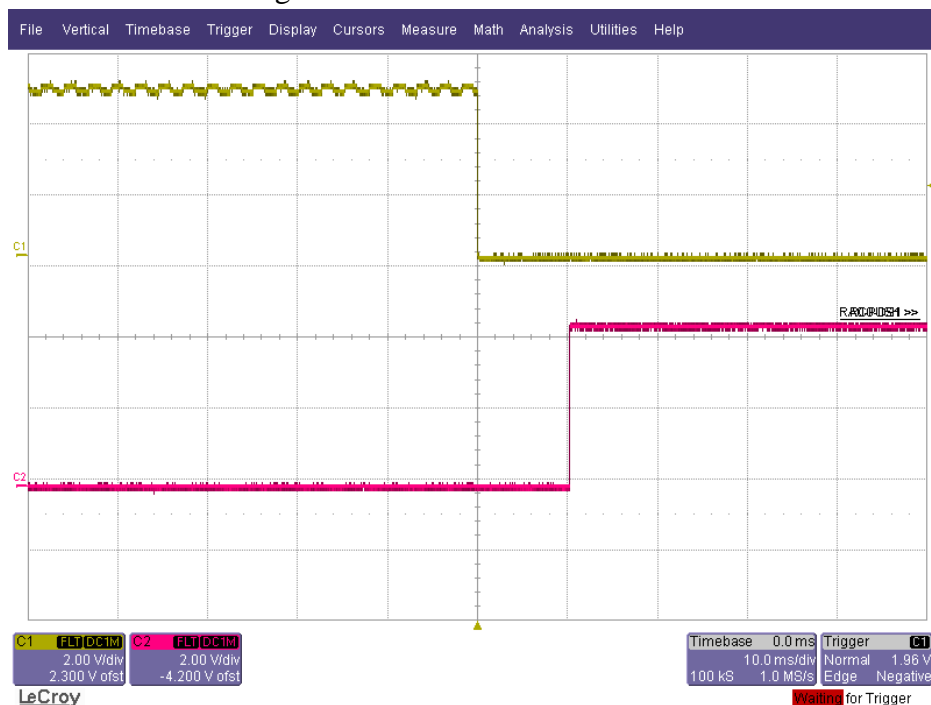
Este prezentat un algoritm pentru schimbarea stării unui led la fiecare apăsare a unui push-button.

Figura 5-10: Algoritm debounce



În figura următoare se poate vedea că după apăsarea push-button-ului conectat la pinul de intrare RA3, pinul RC0 își va schimba starea logică doar după 10ms, întârziere fiind dată de algoritmul prezentat anterior.

Figura 5-11: Debounce software 10ms



5.5. Probleme propuse

- Studiați cazul în care legăm doi pini de intrare între ei printr-o sârmă. Este permis?
- Studiați cazul în care legăm un pin de ieșire cu unul de intrare între ei printr-o sârmă. Ce se întâmplă? Este permis?
- Studiați cazul în care legăm un pin de ieșire cu doi pini de intrare printr-o sârmă. Ce se întâmplă? Este permis?

5.6. Aplicație propusă

Scrieți un mic program în care să setați pinul RA3 pin de intrare și pinii RC0-RC3 pini de ieșire. Schimbați starea ledului conectat la pinul RC0 la fiecare apăsare a butonului conectat la pinul RA3.

În figura de mai jos se poate vedea că la fiecare nouă apăsare a push-button-ului conectat la pinul de intrare RA3, pinul RC0 își va schimba starea logică, lucru care se observă și prin aprinderea sau stingerea ledului DS1.

Figura 5-12: Trecea din starea 0 logic în starea 1 logic la prima acționare a push-button-ului

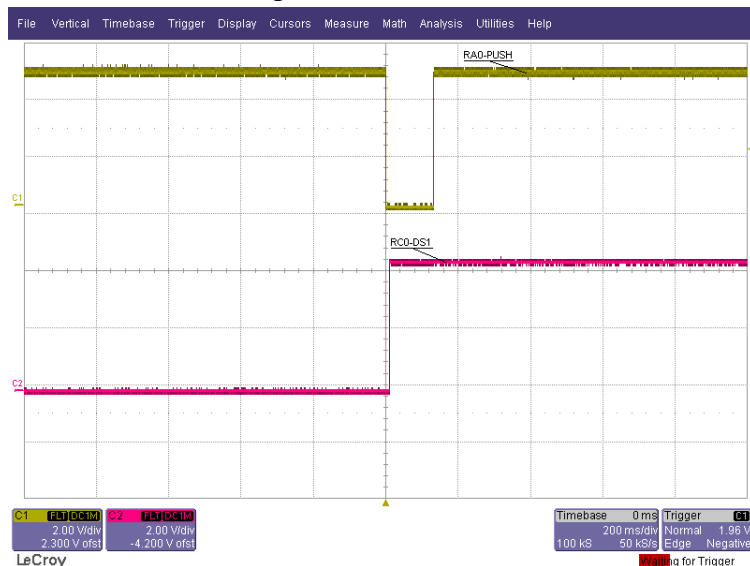
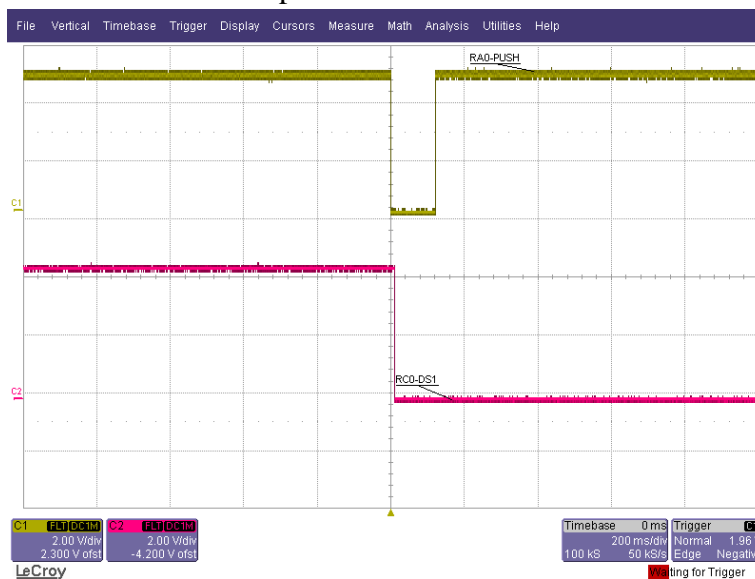


Figura 5-13: Trecea din starea 1 logic în starea 0 logic la o nouă acționare a push-button-ului



5.7. Model software

```
/* include files */
#include "pic.h"

/* constant and macro defines */
#define PUSH    ???
#define LED     ???
#define ON      1
#define OFF     0
#define PRESSED 0

/* function declarations */
void init();
void delayMs(unsigned int ms);

/* function definitions */

void main()
{
    init();
    LED = OFF;
    /* delay before entering infinite loop */
    delayMs(1000);
    while(1)
    {
        /* check if push button has been pressed */
        if(PUSH == ???)
        {
            delayMs(10); /* 10ms delay */
            if(PUSH == ???) /* if push button is still pressed */
            {
                ???
            }
        }
    }
}
```

```

void init()
{
    ANSEL = 0x0F; /* set RC0 to RC3 as digital pins */
    ANSELH = 0x0C ; /* set RC6 and RC7 as digital pins */

    TRISA = ???; /* set all pins on port A as input */
    TRISC = ???; /* RC4 to RC7 input. RC0 to RC3 output */
    PORTC = ???; /* port C pins reset value */
}

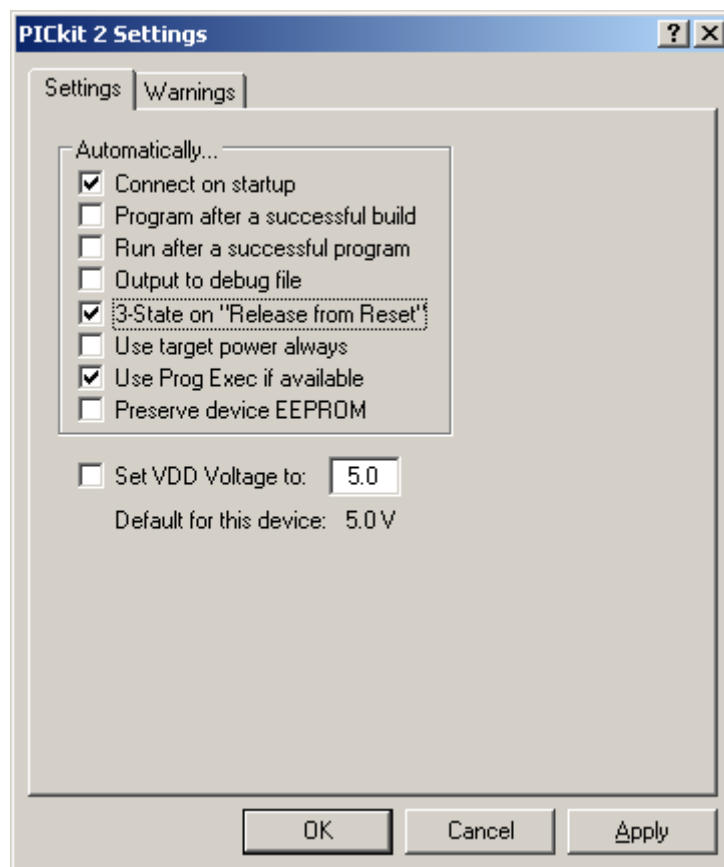
void delayMs(unsigned int ms)
{
    unsigned int i,j;
    for(i = 0; i<ms; i++)
    {
        /* delay for 1 ms - empirically determined */
        for(j=0; j<62; j++)
        {
            ;
        }
    }
}

```

Observație 1: Pinul RA3, pe lângă funcția de intrare-ieșire digitală, mai poate fi folosit și ca pin de reset (MCLR). Pentru a putea folosi push-button-ul de pe placă pentru altă funcție decât cea de reset, trebuie modificați biții de configurare ai microcontrolerului după cum am prezentat în primul capitolul la punctul 1.2.8 (*Pin Function Select bit ia valoarea MCLR pin function is digital input*).

Observație 2: În momentul în care programatorul PICKIT 2 este conectat la placa de dezvoltare, starea electrică a pinului RA3 (MCLR) este controlată din mediul de dezvoltare MPLAB (pentru a putea ține sau scoate din reset microcontrolerul). Datorită faptului că în aplicația noastră alimentarea se face prin PICKIT 2, programatorul nu poate fi deconectat după scrierea programului așa că starea pinul RA3 nu poate fi modificată prin apăsarea push-button-ului. Pentru a putea totuși folosi push-button-ul în timp ce programatorul este conectat trebuie făcută următoarea setare din meniul **Programmer / Settings**: căsuța **3-State on „Release from Reset”** trebuie bifată precum în figura de mai jos:

Figura 5-14: Setarea programatorului pentru folosirea push-button-ului conectat la RA3

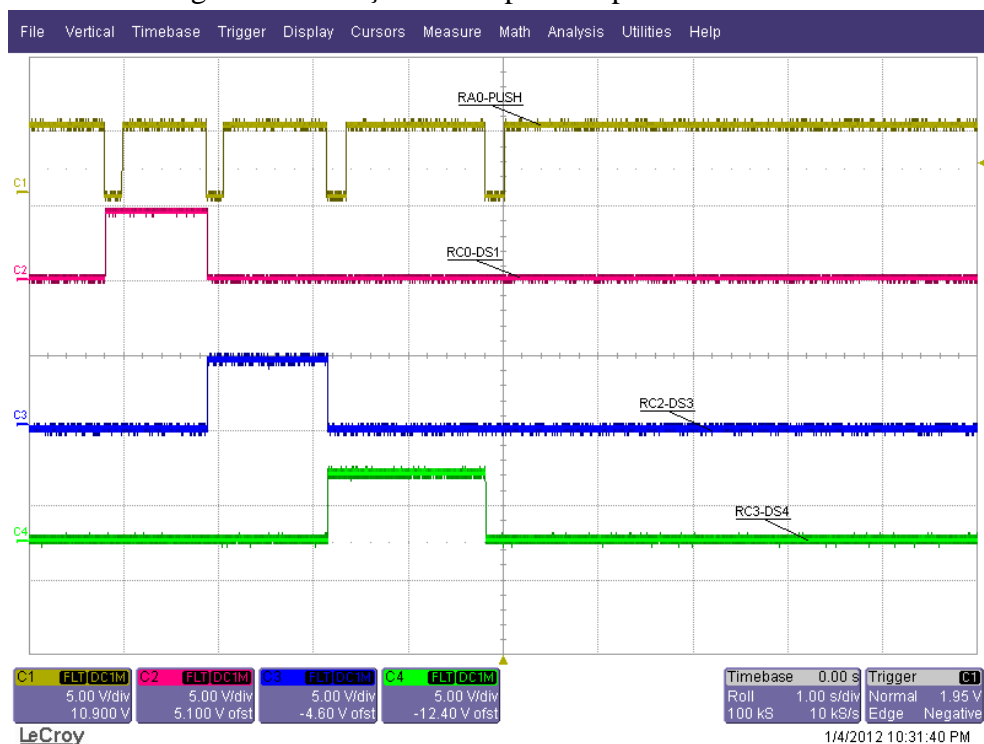


5.8. Problemă propusă

Scrieți un program prin care să aprindeți succesiv ledurile (doar un led aprins la un moment dat), de la DS1 la DS3, la fiecare apăsare nouă a butonului legat la pinul RA3.

În figura de mai jos se poate observa că la fiecare apăsare nouă a push-button-ului conectat la pinul de intrare RA3, pinii RC0-RC3 își vor schimba starea logică succesiv.

Figura 5-15: Acționarea repetată a push-button-ului



6. Timer 1

6.1. Introducere

Pentru realizarea sarcinii de a funcționa în timp real, aplicațiile embedded au nevoie de mecanisme specifice pentru a determina intervale precise de timp. Microcontrolerele oferă astfel de mecanisme încorporate în modulul numit Timer. Cea mai importantă funcție a modulului Timer este aceea de numărător intern (internal counter). Un registru (counter register) este incrementat la intervale fixe, frecvența de incrementare fiind egală cu frecvența la care rulează aplicația (frecvența sistemului) sau este divizată (în funcție de setările modulului) din frecvența sistemului. Pentru a folosi această funcție, putem fie citi registrul intern, determinând din numărul de incremente timpul trecut, fie putem folosi întreruperile hardware generate de către modul în momentul în care contorul atinge o valoare prestabilită.

Microcontrolerul PIC16F690 dispune de 3 module Timer:

- Un timer pe 16 biți (Timer 1).
- Două module de timer pe 8 biți (Timer 0 și Timer 2).
- Modulul Timer 2 dispune și de post-scalare programabilă.
- Oricare din cele trei module poate fi folosit ca sursă de întrerupere.
- Toate cele trei module de timer dispun de pre-scalare programabilă.

6.2. Descriere Timer 1

- Timer pe 16 biți compus din doi regiștri de 8 biți: TMR1H și TMR1L care se pot scrie și citi.
- Pre-scalare programabilă (factor de divizare 1, 2, 4 sau 8).
- Perechea de regiștri se incrementează de la 0x0000 până la 0xFFFF. În momentul în care se atinge valoarea maximă se produce un overflow și se reia incrementarea de la valoarea 0x0000.
- În momentul în care se produce un overflow, modulul de timer poate genera o întrerupere.
- Pentru controlul și indicarea stării întreruperii, modulul are asigurați doi biți: bitul TMR1IE pentru activarea întreruperii și bitul TMR1IF pentru indicarea stării întreruperii.

din registrul de configurație T1CON. Practic, dacă vom folosi un pre-scalar de 1:8, valoarea contorului se va incrementa doar la al 8-lea impuls sosit la intrarea modulului. Astfel, intervalul de timp ce poate fi capturat pe același număr de biți va fi de 8 ori mai mare. O valoare mică a pre-scalarului duce la o precizie mai mare a timpului măsurat dar valoarea maximă pentru generarea întreruperilor este mai mică. În contradicție, un pre-scalar mai mare duce la o precizie mai mică dar aduce beneficiul unui timp mai mare între întreruperi. În funcție de aplicația dorită, se pot alege următoarele valori pentru pre-scalare: 1:1, 1:2, 1:4, 1:8.

În următoarele tabele vom descrie regiștri cei mai uzuali ai modulului.

Tabel 6-1: Descrierea regiștrilor aferenți modulului Timer 1 [8]

Nume	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
CM2CON1	MC1OUT	MC2OUT	-	-	-	-	T1GSS	C2SYNC
INTCON	GIE	PEIE	T0IE	INTE	RABIE	T0IF	INTF	RABIF
PIE1	-	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
PIR1	-	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
TMR1H	Conține cei mai semnificativi 8 biți ai TMR1							
TMR1L	Conține cei mai puțin semnificativi 8 biți ai TMR1							
T1CON	T1GINV	TMR1GE	T1CKPS1	T1CKPS0	T1OSC	T1SYNC	TMR1CS	TMR1ON

Legenda: x=necunoscut, u=nemodificat, -=bitul se citește ca0. Biți închiși la culoare nu sunt folosiți de TIMER1.

INTCON: registrul de configurare al întreruperilor. Se activează întreruperile generale și ale perifericelor.

PIR1: registrul de flag-uri al perifericelor. Conține flag-urile individuale de întrerupere pentru periferice.

PIE1: registrul de activare al întreruperilor pentru periferice. Conține biți individuali de activare a întreruperilor pentru periferice.

CM2CON1: registrul de control al modulului comparator 2. Este folosit pentru activarea funcției „gate control” pentru Timer 1. Această funcționalitate nu este prezentată în această lucrare.

Tabel 6-2: Registrul T1CON - registrul de configurare pentru Timer 1

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
T1GINV ⁽¹⁾	TMR1GE ⁽²⁾	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON
bit 7							bit 0

Legendă: R - bitul poate fi citit; W - bitul poate fi scris; U - bit neimplementat, se va citi 0; n – valoare după POR;
1 - bitul e setat; 0 - bitul e șters; X - valoare necunoscută;

Bit 7 T1GINV: Timer1 Gate Invert bit(1)

1 = Timer 1 se va incrementa când semnalul de „Gate” este 1 logic

0 = Timer 1 se va incrementa când semnalul de „Gate” este 0 logic

Bit 6 TMR1GE: Timer 1 Gate control bit(2)

Dacă TMR1ON este 0:

Bitul TMR1ON este ignorat

Dacă TMR1ON este 1:

1 = Incrementarea Timer 1 este controlată de semnalul de „Gate”

0 = Timer 1 se va incrementa în permanență

Bit 5-4 T1CKPS<1:0>: Timer 1 Input Clock Prescale Select bit

11 = valoare pre-scalar 1:8

10 = valoare pre-scalar 1:4

01 = valoare pre-scalar 1:2

00 = valoare pre-scalar 1:1

Bit 3 T1OSCEN: LP Oscillator Enable Control bit

Dacă INTOSC este activat fără oscilator pe CLKOUT:

1 = Oscilatorul LP e activat ca și sursă de ceas pentru TIMER 1

0 = Oscilatorul LP e dezactivat

Altfel bitul este ignorat

Bit 2 T1SYNC: Timer 1 External Clock Input Synchronization Control bit TMR1CS = 1:

1 = Nu se va sincroniza cu semnalul extern

0 = Se va sincroniza cu semnalul extern

Bit 1 TMR1CS: Timer 1 Clock Source Select bit

1 = Sursa de ceas va fi semnalul extern de pe pinul T1CKI (frontul crescător)

0 = Sursa de ceas va fi semnalul intern $F_{OSC}/4$

Bit 0 TMR1ON: Timer 1 On bit

1 = Timer 1 este activat/pornit

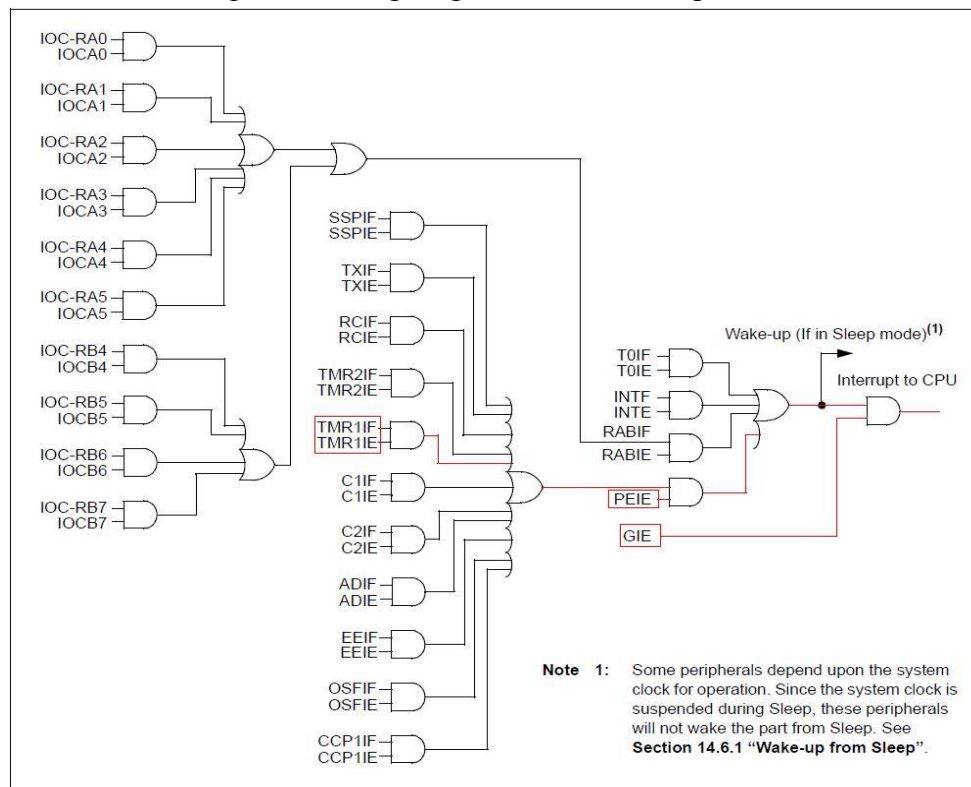
0 = Timer 1 este dezactivat/oprit

Notă 1: Bitul T1GINV inversează logica de „Gate” indiferent de sursa de ceas.

Notă 2: Bitul TMR1GE trebuie setat să folosească fie pinul T1G fie C2OUT ca și sursă de „Gate” pentru Timer 1, după cum este selectat de către bitul T1GSS din registrul CM2CON1.

În ceea ce privește generarea de întreruperi, utilizatorul trebuie să facă următorii pași:

Figura 6-2: Logica generării de întreruperi [8]



Biții GIE și PEIE din registrul INTCON trebuie setați pentru a activa întreruperile generale și ale perifericelor.

Tabel 6-3: Registrul INTCON

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	T0IE	INTE	RABIE ^(1,3)	T0IF ⁽²⁾	INTF	RABIF
bit 7							bit 0

Legendă: R - bitul poate fi citit; W - bitul poate fi scris; U - bit neimplementat, se va citi 0; n - valoare după POR; 1 - bitul e setat; 0 - bitul e șters; X - valoare necunoscută;

Bit 7 GIE: Global Interrupt Enable bit

1 = Activează toate sursele de întrerupere

0 = Dezactivează toate sursele de întrerupere

Bit 6 PEIE: Peripheral Interrupt Enable bit

1 = Activează toate sursele de întrerupere ale perifericelor

0 = dezactivează toate sursele de întrerupere ale perifericelor

Bitul flag TMR1IF din registrul PIR1 trebuie șters înainte de a valida sursa de întrerupere prin setarea bitului de activare a întreruperii. În caz contrar, riscăm ca bitul flag să aibă valoarea 1, iar în momentul activării întreruperii, programul să sară în rutina de tratare a întreruperii, lucru nedorit de noi în acel moment. Aceasta ar trebui să fie o regulă de la care să nu ne abatem niciodată în timpul scrierii programului pentru microcontroler: bitul flag trebuie șters înainte de a activa sursa de întrerupere.

Tabel 6-4: Registrul PIR1

U-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
—	ADIF ⁽⁶⁾	RCIF ⁽³⁾	TXIF ⁽³⁾	SSPIF ⁽⁴⁾	CCP1IF ⁽²⁾	TMR2IF ⁽¹⁾	TMR1IF
bit 7							bit 0

Legendă: R - bitul poate fi citit; W - bitul poate fi scris; U - bit neimplementat, se va citi 0; n – valoare după POR; 1 - bitul e setat; 0 - bitul e șters; X - valoare necunoscută;

Bit 0 TMR1IF: Timer 1 Overflow Interrupt Flag bit

1 = Registrul numărător Timer 1 a atins valoarea maximă

0 = Registrul numărător Timer 1 nu a atins valoarea maximă

Întreruperea Timer 1 trebuie activată prin setarea bitului TMR1IE din registrul PIE1.

Tabel 6-5: Registrul PIE1

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	ADIE ⁽⁶⁾	RCIE ⁽³⁾	TXIE ⁽³⁾	SSPIE ⁽⁴⁾	CCP1IE ⁽²⁾	TMR2IE ⁽¹⁾	TMR1IE
bit 7							bit 0

Legendă: R - bitul poate fi citit; W - bitul poate fi scris; U - bit neimplementat, se va citi 0; n – valoare după POR; 1 - bitul e setat; 0 - bitul e șters; X - valoare necunoscută;

Bit 0 TMR1IE: Timer 1 Overflow Interrupt Enable bit

1 = Activează întreruperea generată de Timer 1

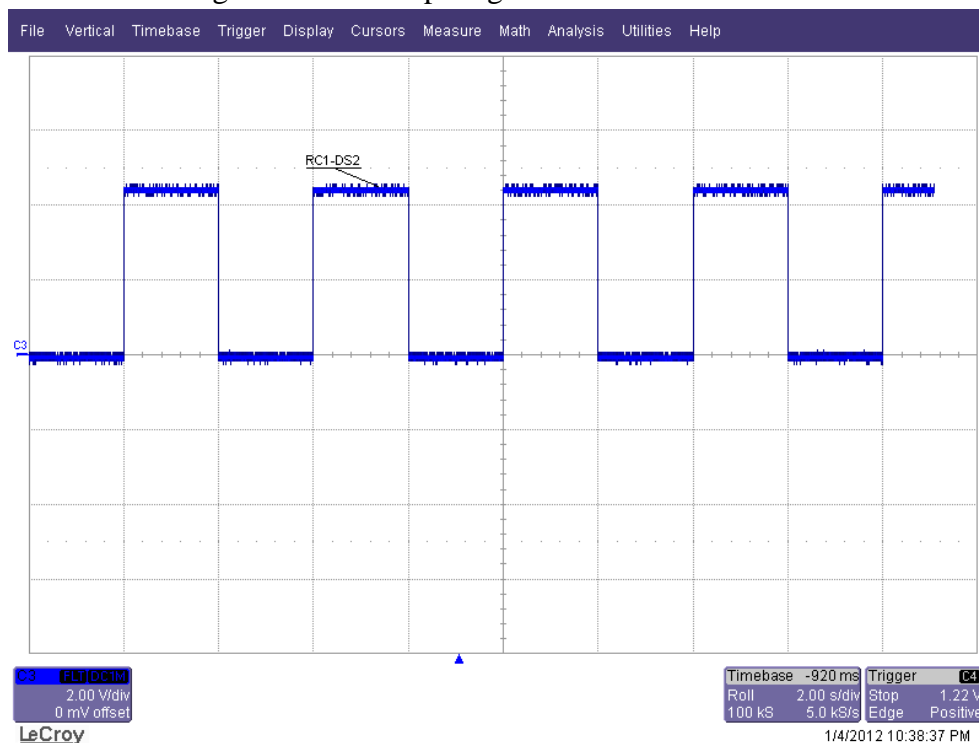
0 = Dezactivează întreruperea generată de Timer 1

6.3. Aplicație propusă

Să se scrie un program care să schimbe starea ledului DS2 de pe placă, la fiecare 2 secunde, folosind ca bază de timp modulul Timer 1.

În *Figura 6-3* este prezentat semnalul dreptunghiular cu factor de umplere 50%, generat prin schimbarea stării logice a pinului RC1 (led DS2) la fiecare 2s.

Figura 6-3: Întrerupere generata la 2s cu Timer1



6.4. Configurarea timer-ului

Timer 1 va fi configurat astfel încât să genereze întreruperi la fiecare 500ms.

În primul rând trebuie calculată durată maximă ce poate fi măsurată pe 16 biți pentru frecvența de oscilator.

$$F_{osc} = 4\text{MHz}$$

$$F_{timer} = F_{osc}/4 = 1\text{MHz}$$

$$T_{osc} = 250\text{ns} (1/F_{osc})$$

$$T_{timer} = 1000\text{ns} = 1\mu\text{s}.$$

Asta înseamnă că pentru un pre-scalar de 1:1 registrul TMR1 (TMR1H+TMR1L) se va incrementa la fiecare 1μs.

$$T_{dorit} = 500\text{ms}$$

$$NR_{\text{incrementari}} = T_{\text{dorit}}/T_{\text{timer}}$$

$$NR_{\text{incrementari}} = 500\text{ms}/1\mu\text{s}$$

$$NR_{\text{incrementari}} = 500\,000$$

Valoarea maximă ce poate fi scrisă în registrul TMR1 este 65535 (0xFFFF), deci perioada de timp maximă ce poate fi măsurată este aproximativ 65.5ms. Vom fi nevoiți să folosim un pre-scalar diferit de 1:1.

Dacă vom folosi un pre-scalar 1:8 T_{timer} va deveni 8 μs , deoarece registrul TMR1 se va incrementa doar la fiecare al optulea impuls de la intrare. Atunci:

$$NR_{\text{incrementari}} = T_{\text{dorit}}/T_{\text{timer}}$$

$$NR_{\text{incrementari}} = 500\text{ms}/8\mu\text{s}$$

$$NR_{\text{incrementari}} = 62\,500 = 0xF424$$

Din numărul de incrementări rezultă valoarea ce trebuie scrisă în TMR1H și TMR1L. Ea este $0xFFFF - 0xF424 = 0xBDB$. Adică timer-ul se va incrementa de la 0xBDB la 0xFFFF, adică de 62 500 de ori. $TMR1H = 0x0B$; $TMR1L = 0xDB$.

După stabilirea valorii pre-scalarului și a registrului TMR1H și TMR1L, vom trece la setarea celorlalți biți din registrul de configurare T1CON:

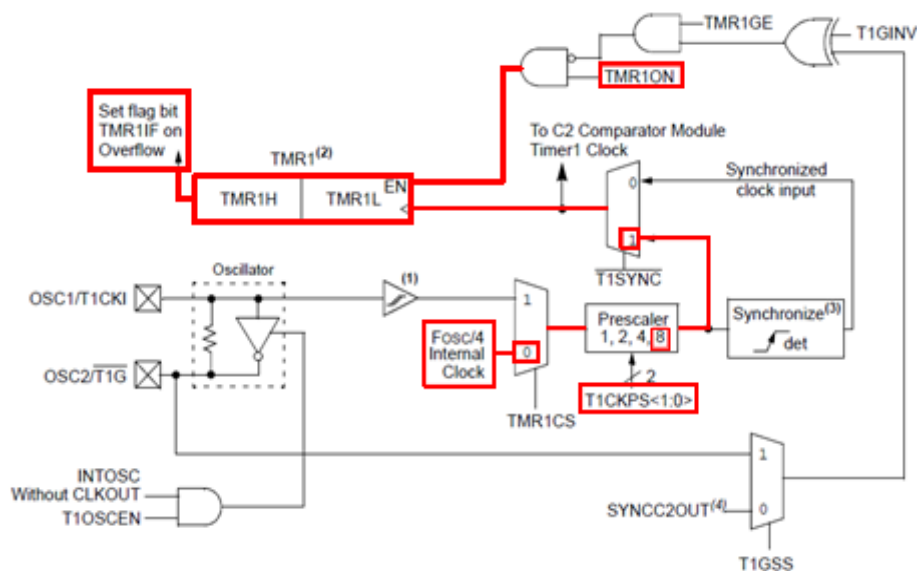
- Bitul 7 și 6 (T1GINV și TMR1GE) vor rămâne zero deoarece nu folosim funcția de „Gate” a timer-ului.
- Bitul 5 și 4 (T1CKPS1:T1CKPS2) sunt biți pentru setarea pre-scalarului. Pentru pre-scalare de 1 la 8 avem nevoie de valoarea 0b11.
- Bitul 3 (T1OSCEN) este folosit doar în cazul în care folosim oscilator intern pentru „low-power”. Pentru configurația folosită, acest bit este ignorat.
- Bitul 2 (T1SYNC) este valid doar dacă modulul folosește un ceas extern pentru configurația folosită, acest bit este ignorat.
- Bitul 1 (TMR1CS) alege sursa ceasului folosit. Pentru a avea ceas intern $F_{\text{osc}}/4$, acesta ia valoarea zero.
- Bitul 0 (TMR1ON) este folosit pentru activarea modulului. Timer 1 este activ/pornit când acesta ia valoarea 1.

Pentru activarea întreruperii, setăm biții GIE și PEIE din registrul INTCON.

În final, trebuie șters bitul de stare (*flag*) TMR1IF și apoi setat bitul TMR1IE pentru activarea întreruperii.

Folosind această configurație, schema bloc a modulului va arăta ca în figura de mai jos.

Figura 6-4: Configurare Timer 1



6.5. Model software

În programul scris de noi va trebui să avem în vedere următoarele:

- Configurarea modulului Timer 1.
- Configurarea pinului RC2 (la care este legat ledul 3) ca pin de ieșire.
- Scrierea rutinei de tratare a întreruperii.

Exemplu de cod:

```
/* include files */
#include "pic.h"

/* constant and macro defines */
#define LED    ???
#define ON     1
#define OFF    0

/* variables */
volatile unsigned int counter;
```

```
/* function declarations */
void init();

/* function definitions */

void main()
{
    init();

    while(1)
    {
        /* switch LED after m seconds */
        if(counter == ???)
        {
            if(LED == ON)
            {
                LED = OFF;
            }
            else
            {
                LED = ON;
            }
            counter = 0; /* reset counter */
        }
    }
}

void init()
{
    ANSEL = ???; /* set RC0 to RC3 as digital pins */
    ANSELH = ???; /* set RC6 and RC7 as digital pins */

    TRISC = ???; /* RC4 to RC7 input. RC0 to RC3 output */
    PORTC = 0x00; /* port C pins reset value */

    /* timer 1 settings */
    TMR1L = ???;
    TMR1H = ???;
    T1CON = ???;
```



```
/* interrupt settings */
GIE = 1; /* global interrupt enable */
PEIE = 1; /* peripheral interrupt enable */
TMR1IF = 0; /* clear TMR1 interrupt flag */
TMR1IE = 1; /* TMR1 interrupt enabled */

/* start TMR1 */
TMR1ON = 1;

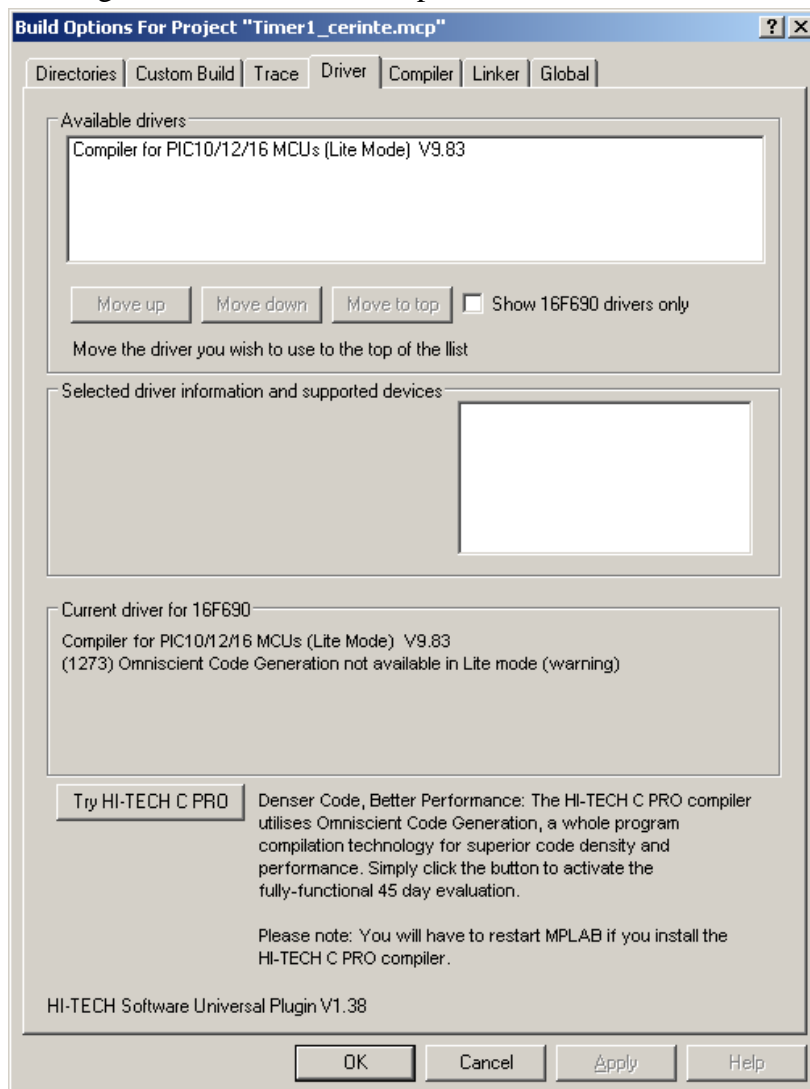
/* variable initializations */
counter = 0;
}

/* Interrupt function */
void interrupt_isr(void)
{
    /* check if TMR1 interrupt enable and flag are set */
    if((TMR1IE == 1) && (TMR1IF == 1))
    {
        TMR1L = ???;
        TMR1H = ???;
        counter++; /* increment counter every 500ms */
        TMR1IF=0; /* clear TMR1 interrupt flag*/
    }
}
```

Observație: Implementând aplicațiile prezentate în această lucrare pe diverse calculatoare s-a observat că, în mediile unde sunt instalate mai multe compilatoare, folosind anumite compilatoare în combinație cu definirea funcției de tratare a întreruperilor duce la un comportament eronat: după ieșirea din reset, în loc să se execute funcția *main*, aplicația sare direct în rutina de tratare a întreruperilor. Dacă, după scrierea programului în microcontroler, aplicația pare să nu ruleze, acest comportament se poate datora compilatorului activ folosit în proiect. Această problemă este prezentă chiar dacă selectăm compilatorul potrivit când creăm proiectul (capitolul 1, punctul 2.3). Pentru a verifica dacă folosim compilatorul potrivit, accesați meniul **Project / Build Options / Project** și în fereastra **Driver**, asigurați-vă că primul compilator din listă este **Compiler for**

PIC10/12/16 MCUs (Lite Mode) Vx.yy precum în figura de mai jos. Folosiți butonul **Move Up** dacă acesta nu e primul.

Figura 6-5: Selectarea compilatorului folosit la **Build**

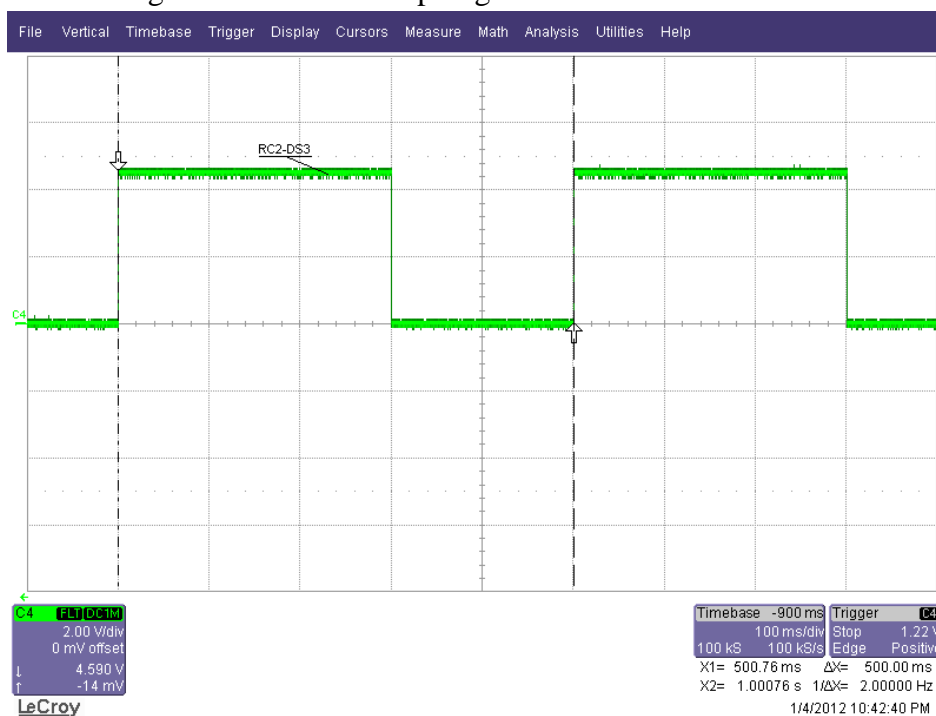


6.6. Problemă propusă

Să se genereze un semnal dreptunghiular cu frecvență 2Hz și factor de umplere 60% pe pinul digital de ieșire conectat la ledul 1 de pe placă, cu ajutorul Timer 1.

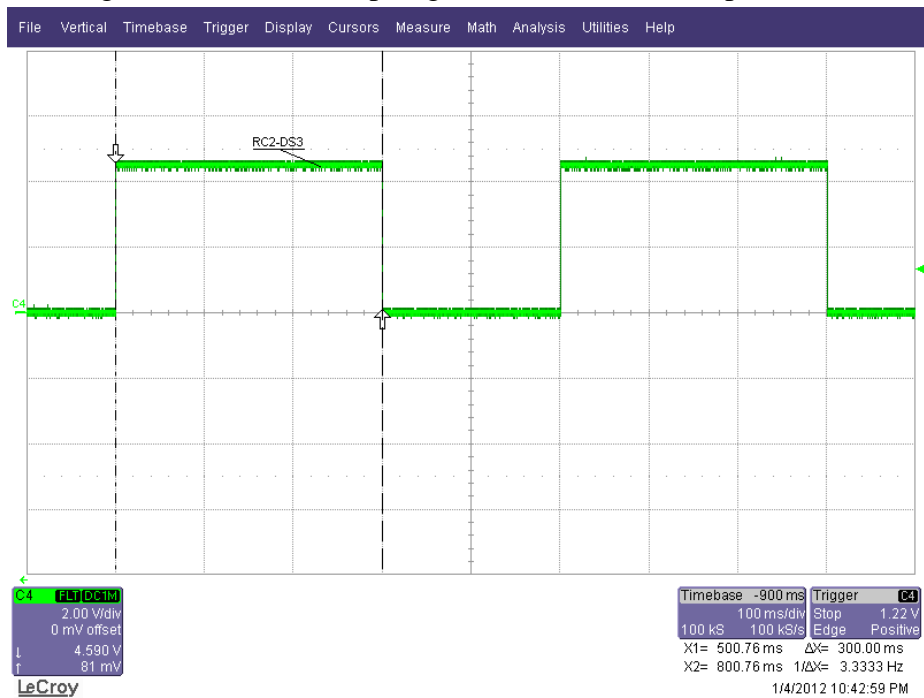
În figurile următoare este prezentat semnalul dreptunghiular cu frecvența 2Hz și factor de umplere de 60% generat cu ajutorul Timer1, pe pinul RC2. În Figura 6-6 este pusă în evidență, între cele două cursoare prezente pe oscilogramă, valoarea perioadei semnalului.

Figura 6-6: Semnal dreptunghiular cu frecventa de 2Hz



În Figura 6-7 este pusă în evidență, între cele două cursoare prezente pe oscilogramă, valoarea pulsului *high* a semnalului (factorul de umplere 60%).

Figura 6-7: Semnal dreptunghiular cu factor de umplere 60%

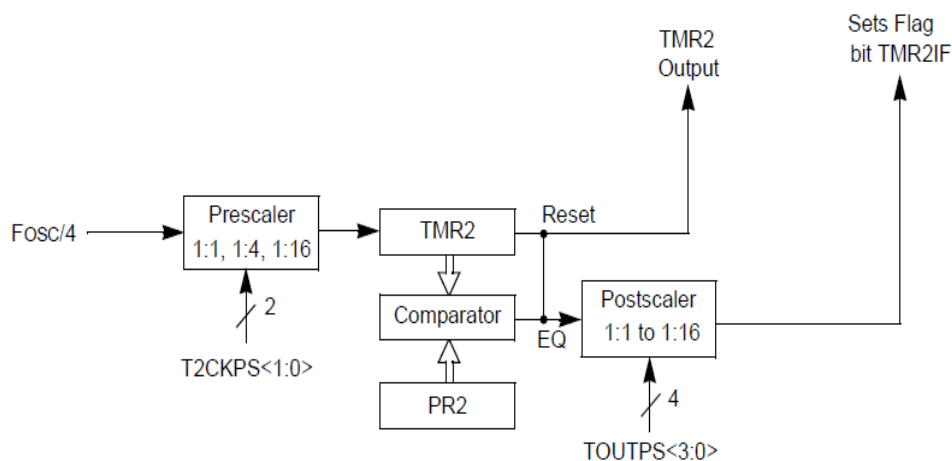


7. Timer 2

7.1. Descriere Timer 2

- Timer pe 8 biți, compus dintr-un registru contor (TMR2) și un registru pentru perioadă (PR2).
- pre-scalar programabil (1:1, 1:4, 1:16).
- post-scalar programabil (1:1 până la 1:16).
- valoarea pre-scalarului și post-scalarului se resetează la orice scriere în regiștri TMR2 sau T2CON.
- registrul TMR2 se incrementează pornind de la valoarea 0x00 până atinge valoarea setată în registrul perioadă PR2. După acest moment, la următorul impuls, incrementarea se reia de la 0x00.
- după reset, registrul PR2 este inițializat cu valoarea 0xFF. În acest registru, înainte de activarea modulului, trebuie scrisa valoarea calculată pentru a măsura perioada de timp dorită.
- când avem condiție de egalitate între contorul TMR2 și valoarea scrisă în PR2, se generează o întrerupere.
- biții asociați întreruperilor sunt: TMR2IE pentru activarea întreruperii; TMR2IF bitul flag al întreruperii; GIE bitul pentru activarea întreruperilor globale; PEIE bitul pentru activarea întreruperilor de la periferice.

Figura 7-1: Schema bloc a Timer2 [8]



Principiul de funcționare al acestui tip de timer este următorul:

Modulul timer conține doi regiștri: unul de incrementare (TMR2) și unul pentru definirea perioadei de timp ce se dorește a fi măsurată (PR2). În registrul de perioadă PR2 se încarcă o valoare dorită de noi (calculată în prealabil). Registrul TMR2 se incrementează după activarea modulului, la fiecare impuls sosit la intrarea timer-ului (cu frecvența $F_{osc}/4$), de la valoarea 0 până va ajunge la valoarea scrisă în registrul de perioadă PR2.

La următorul impuls sosit la intrarea în timer, registrul TMR2 se va reseta la valoarea 0 și va începe din nou să se incrementeze. Dacă circuitul de post-scalare este setat altfel decât 1:1, bitul flag TMR2IF nu se va seta la prima egalitate dintre TMR2 și PR2. Cu alte cuvinte, dacă circuitul de post-scalare este setat 1: n , bitul de întrerupere TMR2IF se va seta doar la a n -a egalitate dintre registrul TMR2 și PR2. Spre exemplu, pentru o setare a post-scalarului 1:6, doar la a 6-a egalitate se va seta flag-ul TMR2IF (TMR2 se incrementează până la valoarea PR2 de 6 ori).

Dacă și bitul TMR2IE (bitul de activare al întreruperii) este setat, se va genera o întrerupere (cu condiția ca și biții GIE și PEIE să fie setați). Microcontrolerul va sări în rutina de tratare a întreruperii, unde bitul TMR2IF trebuie șters, pentru ca o nouă întrerupere să fie posibilă.

Frecvența impulsurilor de intrare în modulul timer este $F_{osc}/4$, unde F_{osc} este frecvența oscilatorului folosit (4MHz în cazul aplicației noastre). Ea poate fi divizată cu ajutorul pre-scalarului setat din registrul de configurare T2CON. Practic, dacă vom folosi un pre-scalar 1:4, valoarea registrului TMR2 se va incrementa doar la al 4-lea impuls sosit la intrarea modulului. Astfel, intervalul de timp ce poate fi măsurat pe același număr de biți va fi de 4 ori mai mare. Valorile posibile pentru pre-scalar sunt 1:1, 1:4, 1:8.

Tabel 7-1: Descrierea regiștrilor aferenți modulului Timer 2 [8]

Nume	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
INTCON	GIE	PEIE	T0IE	INTE	RABIE	T0IF	INTF	RABIF
PIE1	-	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
PIR1	-	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
TMR2	Registru de incrementare pe 8 biți							
PR2	Registru de perioadă a Timer 2							
T2CON	-	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0

Legendă: x=necunoscut, u=nemodificat, -=bitul se citește ca 0. Biții închiși la culoare nu sunt folosiți de TIMER2.

INTCON: registrul de configurare al întreruperilor. Se activează întreruperile generale și ale perifericelor.

PIE1: registrul de activare al întreruperilor pentru periferice. Conține biți individuali de activare a întreruperilor pentru periferice.

PIR1: registrul de flag-uri al perifericelor. Conține flag-urile individuale de întrerupere pentru periferice.

T2CON: registrul de configurare a modulului Timer 2.

Tabel 7-2: Descriere T2CON - registrul de configurare pentru Timer 2

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
bit 7							bit 0

Legendă: R - bitul poate fi citit; W - bitul poate fi scris; U - bit neimplementat, se va citi 0; n – valoare după POR; 1 - bitul e setat; 0 - bitul e șters; X - valoare necunoscută;

Bit 7 Neimplementat: Se citește 0

Bit 6-3 TOUTPS<3:0>: Biții de selecție a post-scalarului

0000 = 1:1 Post-scalar
 0001 = 1:2 Post-scalar
 0010 = 1:3 Post-scalar
 0011 = 1:3 Post-scalar
 1110 = 1:15 Post-scalar
 1111 = 1:16 Post-scalar

Bit 2: TMR2ON: Bit de activare a modulului

1 = TMR2 este activ
 0 = TMR2 este dezactivat/oprit

Bit 1-0 T2CKPS<1:0>: Bit de selecție a pre-scalarului

00 = Pre-scalar 1
 01 = Pre-scalar 4
 1X = Pre-scalar 16

În ceea ce privește generarea de întreruperi, utilizatorul trebuie să facă următorii pași:

Bitul flag TMR2IF din registrul PIR1 trebuie șters.

Tabel 7-4: Registrul PIR1

U-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
—	ADIF ⁽⁵⁾	RCIF ⁽³⁾	TXIF ⁽³⁾	SSPIF ⁽⁴⁾	CCP1IF ⁽²⁾	TMR2IF ⁽¹⁾	TMR1IF
bit 7							bit 0

Legendă: R - bitul poate fi citit; W - bitul poate fi scris; U - bit neimplementat, se va citi 0; n – valoare după POR;
1 - bitul e setat; 0 - bitul e șters; X - valoare necunoscută;

Bit 1 TMR2IF: Bit de stare a întreruperi TMR2

1 = O întrerupere a TMR2 a fost generată

0 = Nici o întrerupere a TMR2 nu a fost generată

Întreruperea timer-ului 2 trebuie activată prin setarea bitului TMR2IE din registrul PIE1.

Tabel 7-5: Registrul PIE1

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	ADIE ⁽⁵⁾	RCIE ⁽³⁾	TXIE ⁽³⁾	SSPIE ⁽⁴⁾	CCP1IE ⁽²⁾	TMR2IE ⁽¹⁾	TMR1IE
bit 7							bit 0

Legendă: R - bitul poate fi citit; W - bitul poate fi scris; U - bit neimplementat, se va citi 0; n – valoare după POR;
1 - bitul e setat; 0 - bitul e șters; X - valoare necunoscută;

Bit 1 TMR2IE: Bit de activare a întreruperii Timer 2

1 = Întreruperea modulului TMR2 este activată

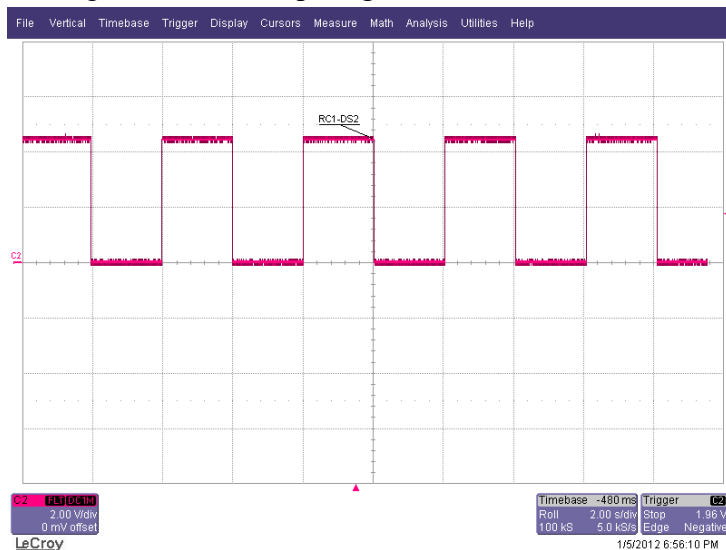
0 = Întreruperea modulului TMR2 este dezactivată

7.2. Aplicații propuse

- Să se scrie un program care să schimbe starea ledului DS2 de pe placă la fiecare 2 secunde, folosind ca și bază de timp Timer 2.

În *Figura 7-3* este prezentat semnalul dreptunghiular cu factor de umplere 50% generat prin schimbarea stării logice a pinului RC1 (led DS2) la fiecare 2s.

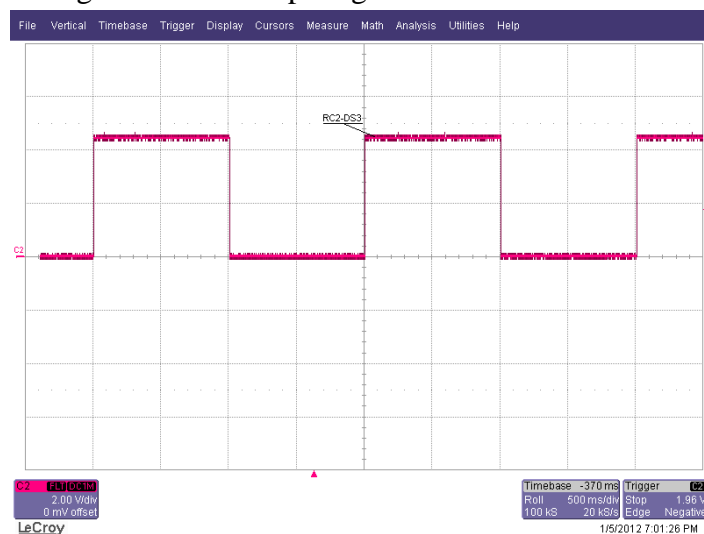
Figura 7-3: Întrerupere generată la 2s cu Timer2



- b) Să se scrie un program care să schimbe starea ledului DS3 de pe placă la fiecare o secundă, folosind ca și bază de timp Timer 2, cu următoarele condiții impuse: post-scalar 1:2 și pre-scalar 1:16.

În Figura 7-4 este prezentat semnalul dreptunghiular cu factor de umplere 50% generat prin schimbarea stării logice a pinului RC2 (led DS3) la fiecare 2s.

Figura 7-4: Întrerupere generată la 1s cu Timer2



7.3. Configurarea timer-ului

Timer 2 va fi configurat astfel încât să genereze întreruperi la fiecare 10ms. În primul rând trebuie calculată durata maximă ce poate fi măsurată pe 8 biți pentru frecvența de oscilator:

$$F_{osc} = 4\text{MHz}$$

$$F_{timer} = F_{osc}/4 = 1\text{MHz}$$

$$T_{osc} = 250\text{ns}$$

$$T_{timer} = 1000\text{ns} = 1\mu\text{s}.$$

Asta înseamnă că pentru un pre-scalar de 1:1 registrul TMR2 se va incrementa la fiecare 1us.

$$T_{dorit} = 10\text{ms}$$

$$NR_{incrementari} = T_{dorit}/T_{timer}$$

$$NR_{incrementari} = 10\text{ms}/1\mu\text{s}$$

$$NR_{incrementari} = 10\,000$$

Valoarea maximă care poate fi scrisă în registrul TMR2 este 255, deci perioada de timp maximă ce poate fi măsurată este aproximativ 256us. Vom fi nevoiți să folosim un pre-scalar diferit de 1:1.

Dacă vom folosi un pre-scalar 1:16, T_{timer} va deveni 16us, deoarece registrul TMR2 se va incrementa doar la fiecare al 16-lea impuls. Atunci:

$$NR_{incrementari} = T_{dorit}/T_{timer}$$

$$NR_{incrementari} = 10\text{ms}/16\mu\text{s}$$

$$NR_{incrementari} = 625 = 0x271$$

Din numărul de incrementări rezultă valoarea ce trebuie scrisă în TMR2. În acest moment valoarea este încă mai mare de 256. În acest caz vom folosi și post-scalarul, astfel încât numărul de incrementări să fie mai mic de 256.

Vom alege un post-scalar 1:5. Adică doar a 5-a egalitate va genera o întrerupere. Atunci:

$$NR_{incrementari} = T_{dorit}/T_{timer}$$

$$NR_{incrementari} = 10\text{ms}/80\mu\text{s}$$

$$NR_{incrementari} = 125 = 0x7D$$

Cu alte cuvinte, pentru a afla valoarea maximă ce poate fi măsurată cu modulul Timer 2, înmulțim valorile maxime pentru pre-scalar și post-scalar (16), cu numărul maxim de incremente al contorului (8 biți = 256 incremente), totul împărțit la frecvența oscilatorului folosit.

$$T_{\max} = (256 * 16 * 16) / (4\text{MHz}/4)$$

Se poate observa că, folosind valorile maxime pentru pre și post scalar, modulul Timer 2 are valoarea maximă cât un timer pe 16 biți ($256 * 16 * 16 = 65536$)

După stabilirea valorilor pentru pre și post scalar, se trece la setarea registrului de control T2CON:

- Bitul 7 bit rezervat.
- Biții 6 până la 3 (TOUTPS3:TOUTPS0) conțin valoarea post-scalarului. Pentru a obține raportul dorit de 1 la 5, trebuie să scriem în acești biți valoarea 0b0100.
- Bitul 2 (TMR2ON) este bitul de activare al modulului. Timer 2 este activat când acesta are valoarea 1.
- Biții 1 și 0 (T2CKPS1-T2CKPS0) setează valoarea pre-scalarului. Pentru un raport de 1 la 16, avem nevoie de valoarea 0b11 sau 0b10.

Pentru activarea întreruperii setăm biții GIE și PEIE din registrul INTCON. La sfârșit trebuie șters bitul TMR2IF și apoi setat bitului TMR2IE pentru activarea întreruperii.

7.4. Model software

În programul scris de noi va trebui să avem în vedere:

- Configurarea modulului Timer 2.
- Configurarea ca și ieșire a pinului RC1 la care este legat ledul DS2.
- Scrierea rutinei de tratare a întreruperii.

Exemplu de cod:

```
/* include files */  
#include "pic.h"
```

```
/* constant and macro defines */  
???
```

```
/* variables */  
???
```

```
/* function declarations */
void init();

/* function definitions */

void main()
{
    init();

    while(1)
    {
        /* switch LED after m seconds */
        if(counter == ???)
        {
            ??? /* insert code here */
        }
    }
}

void init()
{
    /* pin settings */
    ???

    /* timer 2 settings */
    TMR2 = 0x00; /* reset TMR2 counter */
    PR2 = ??? ; /* overflow value */
    T2CON = ??? ; /* TMR 2 settings */

    /* interrupt settings */
    ???

    /* start TMR2 */
    ???

    /* variable intializations */
    counter = 0;
}
```

```
/* Interrupt function */
void interrupt_isr(void)
{
    /* check if TMR2 interrupt enable and flag is set */
    if((???) && (???))
    {
        ??? /* insert code here */
    }
}
```

7.5. Problemă propusă

Descrieți pe scurt un proiect în care poate fi folosit un timer. Ce rol ar avea?
Pe care din cele două module de timer (Timer1 sau Timer2) le-ați folosi și de ce?

8. Servomotor

8.1. Introducere

Multe aplicații embedded presupun comanda unui motor cu scopul de acționare mecanică a unui sistem. În astfel de aplicații, alegerea corectă a motorului constituie de multe ori o sarcină dificilă. Trebuie luate în calcul mai multe aspecte, dintre care cele mai importante, în funcție de aplicație, sunt:

- Gabarit: motorul trebuie să poată fi amplasat în spațiul dedicat produsului. De exemplu, pentru cutiile de viteză automate, ambreiajul este acționat de un motor electric, iar dimensiunile sunt impuse de cerințele clienților (gabaritul cutiei de viteză).
- Cuplu: în funcție de sistemul ce trebuie acționat este nevoie de un anumit cuplu. Fiecare tip de motor are un anumit cuplu în funcție de gabarit, tensiune de alimentare, turație și tip constructiv. Cuplul motorului ales trebuie să satisfacă cerințele proiectului.
- Comanda motorului: comanda electrică a motorului este un criteriu important, deoarece diferă mult de la motor la motor, fiind fie simplă (servomotor), fie foarte complexă (motor brushless).
- Tensiunea de alimentare: ca exemplu, la autoturisme, tensiunea de alimentare este 12V. Un motor cu o tensiune de alimentare diferită de 12V poate introduce în proiect necesitatea unei surse în comutație sau altor soluții hardware, care cresc prețul produsului și complexitatea acestuia.
- Durata de viață: un motor de curent continuu are perii colectoare care, în timp, se deteriorează, pe când un motor brushless nu întâmpină acest impediment. Din acest punct de vedere, un sistem cu motor brushless este indicat în locuri greu accesibile sau în produse cărora li se impune o durată de viață lungă și fără defecte.
- Prețul: prețul motorului este un procent important din costul total al produsului și astfel trebuie să se încadreze în niște limite foarte bine stabilite. Prețul poate varia de la câțiva EURO (servomotoare analogice RC) la sute de EURO (motor de curent continuu cu encoder și cutie reductoare).

Să luăm în continuare un caz particular, o posibilă aplicație. Să plecăm de la ideea că trebuie să realizăm pentru bordul unui automobil un vitezometru

care să pară analogic, adică să aibă o scală și un ac indicator, dar care să fie acționat de un motor electric comandat de sistemul care măsoară viteza autovehiculului. Pentru alegerea motorului trebuie să ținem cont de criteriile de mai sus și am putea lua în calcul câteva tipuri de motoare:

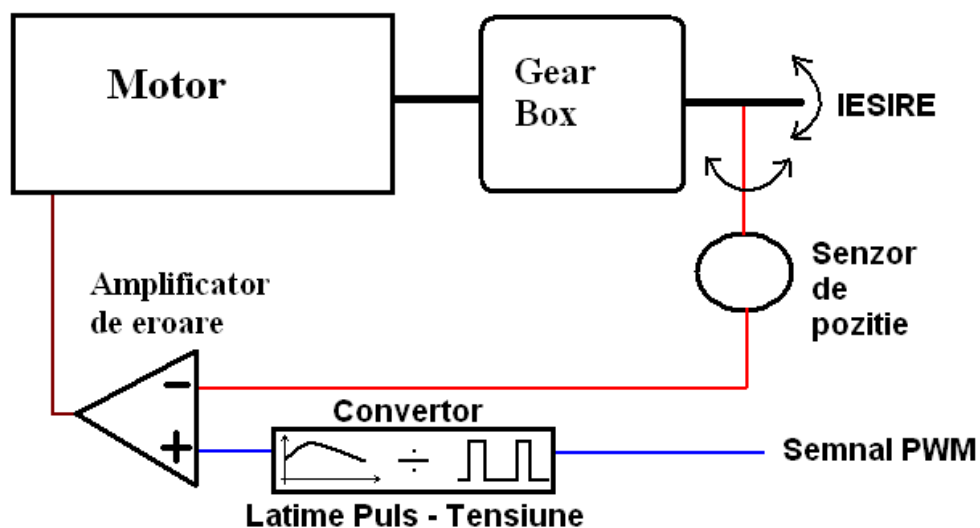
- Motor de curent continuu: din punct de vedere al gabaritului, al cuplului dezvoltat și al prețului poate fi o variantă. Mari dezavantaje ar fi comanda complicată cu buclă închisă (avem nevoie de comandă cu feedback pentru a fi siguri că acul este la poziția dorită) și durata de viață a periiilor.
- Motorul pas cu pas: motorul pas cu pas în conexiune bipolară are de asemenea dezavantajul complexității driverului hardware pentru comandă. În schimb, motorul pas cu pas în conexiune unipolară depășește acest dezavantaj, comanda fiind una relativ simplă. Pentru cuplul necesar acționării acului indicator în această aplicație, motorul are de asemenea un gabarit relativ redus. Durata de viață este foarte mare dar prețul poate fi considerat mare.
- Motorul brushless: nu corespunde din mai multe puncte de vedere. Prețul este ridicat comparativ cu alte tipuri de motoare. Comanda este complexă. În plus, un alt dezavantaj este consumul mai mare de curent.
- Servomotorul: la bază are tot un motor de curent continuu dar de dimensiuni foarte mici. Este cel mai simplu de comandat, deoarece include un sistem de feedback cu potențiomtru iar poziția la care trebuie să se deplaseze rotorul se dă prin comandă PWM pe un singur fir. Include și un reductor, deci cuplul este ridicat comparativ cu gabaritul lui și consumul de curent. Prețul este de asemenea un avantaj, acest tip de motor fiind mai ieftin decât un motor pas cu pas.

8.2. Comanda unui servomotor

Servomotorul este compus din mai multe părți: un motor (DC de regulă), un reductor mecanic, un traductor de poziție (cel mai adesea un potențiomtru), un driver de motor, un amplificator de eroare și un circuit pentru decodificarea poziției dorite.

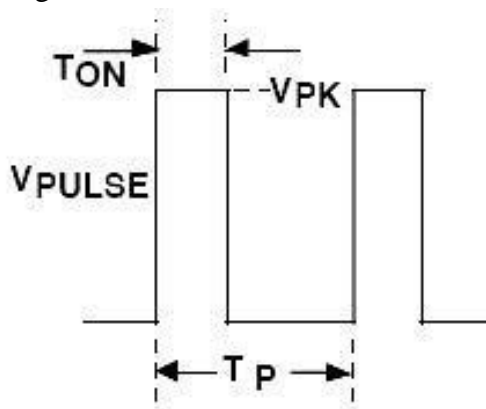
În *Figura 8-1* putem observa schema bloc a unui astfel de sistem.

Figura 8-1: Schema bloc a unui servomotor analogic



Comanda unui servomotor analogic se face cu ajutorul unui semnal PWM (Pulse Width Modulation), generat de către microcontroler.

Figura 8-2: Forma unui semnal PWM



Unde: T_{ON} = durata pulsului high (poate varia)

T_P = perioada semnalului (rămâne constantă)

V_{PK} = amplitudinea semnalului

$DC = T_{ON}/T_P$ (%) - Factor de umplere (Duty Cycle)

Perioada semnalului este 20ms și durata pulsului high variază, în general, între 1ms și 2ms, în funcție de poziția dorită pentru rotor. Durata pulsului

este folosită de către servomotor pentru a determina poziția la care să se rotească.

Semnalul PWM ajunge la intrarea unui circuit care convertește lățimea pulsului high în tensiune. Practic, prin modificarea Duty Cycle a semnalului PWM, se modifică tensiunea de la ieșirea convertorului (care este aplicată la intrarea amplificatorului de eroare).

Senzorul de poziție este un potențiomtru a cărui tensiune de ieșire este proporțională cu poziția absolută a axului. Tensiunea de pe potențiomtru ajunge la una din intrările amplificatorului de eroare, iar la cealaltă intrare se aplică tensiunea de la ieșirea circuitului ce convertește factorul de umplere al semnalului de comandă PWM într-o tensiune analogică. Circuitul se numește convertor *pulse width to voltage* și tensiunea de la ieșire este proporțională cu lățimea pulsului PWM (cu cât lățimea pulsului de la intrare e mai mare cu atât tensiunea va fi mai mare la ieșire). Cu alte cuvinte, la una din intrări vom regăsi poziția curentă, iar la cealaltă poziția dorită.

Amplificatorul de eroare este un amplificator operațional (folosit ca și comparator), care va încerca în permanență să aducă la zero diferența dintre cele două intrări. Ieșirea amplificatorului operațional este o tensiune fie negativă, fie pozitivă, în funcție de diferența celor două tensiuni de la intrare.

Dacă tensiunea este pozitivă, motorul se va roti într-un sens, iar dacă este negativă se va roti în sensul opus. Acest lucru îi permite amplificatorului operațional să reducă diferența dintre cele două tensiuni de la intrare sa, cauzând astfel axul să ajungă la poziția dorită.

8.3. Aplicație propusă

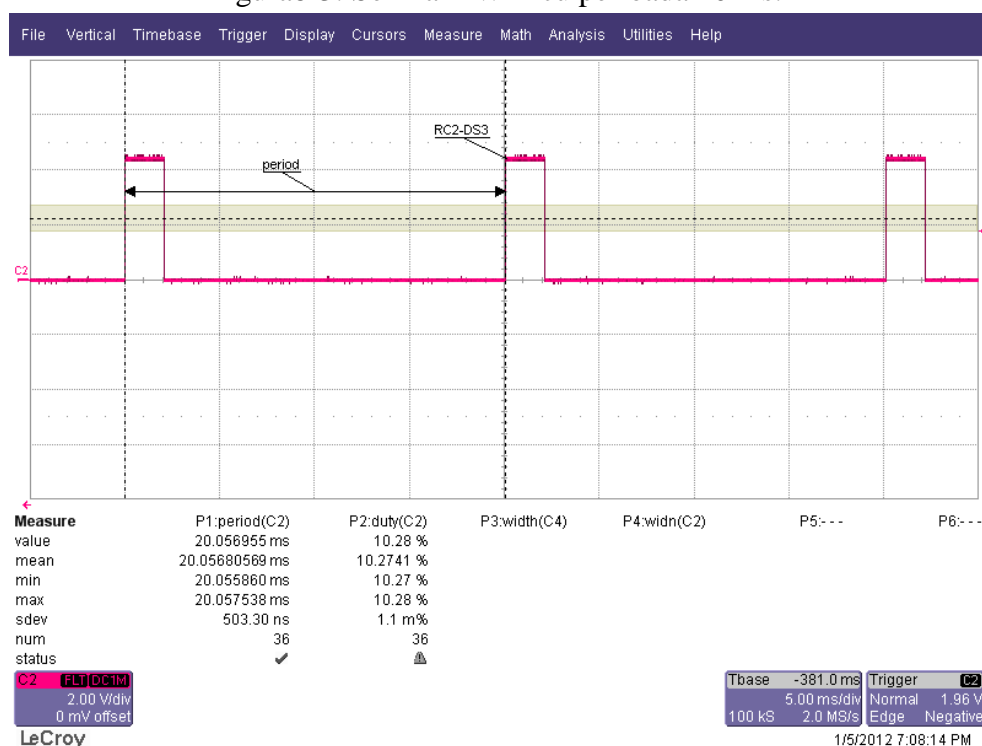
Pentru început stabiliți cu ajutorul unui generator de semnal, lățimea impulsului ce trebuie trimis către servomotor la fiecare 20ms, pentru poziția de 0 grade, 90 de grade și 180 de grade.

	Poziția axului servomotorului		
	0 grade	90 grade	180 grade
Lățimea pulsului [ms]			

Se consideră că aplicația ce trebuie realizată cu ajutorul unui servomotor este un indicator de viteză pentru un autoturism (0Km/h corespunde cu poziția de 0 grade a servomotorului și viteza de 180Km/h cu poziția de 180 de grade a servomotorului). Să se scrie un program în care, cu ajutorul timerelor 1 și 2, microcontrolerul să comande servomotorul astfel încât acesta să indice viteza de 85Km/h. Pinul folosit pentru comandă este pinul la care este conectat și ledul DS4.

În *Figura 8-3* este prezentat semnalul PWM cu perioada 20 ms și cu factor de umplere 10%, generat pe pinul RC3 (led DS4).

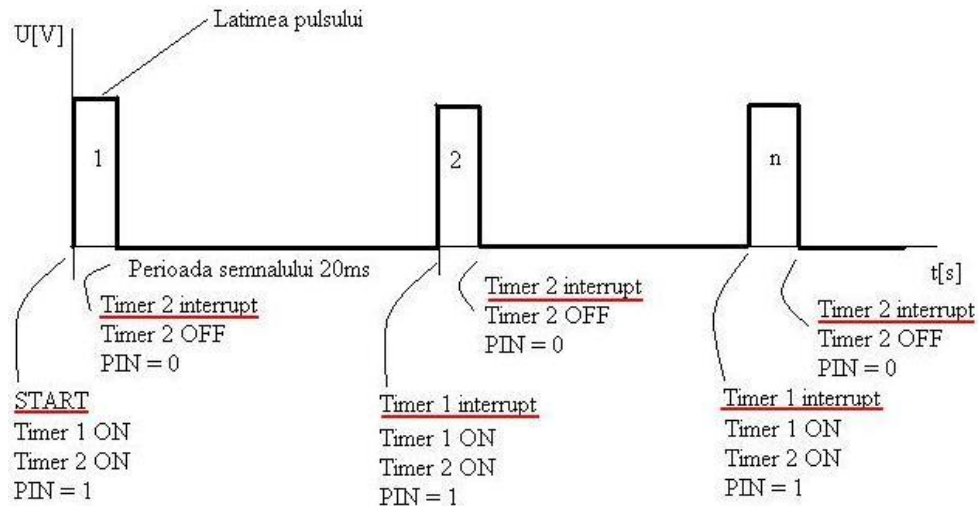
Figura8-3: Semnal PWM cu perioada 20 ms.



8.4. Model software

În programul scris de noi va trebui să avem în vedere, conform figurii de mai jos, următoarele:

Figura 8-4: Program flow



- Configurarea Timer 1 ca și bază de timp de 20ms (perioada constantă a semnalului PWM).
- Configurarea Timer 2 ca și bază de timp pentru lățimea pulsului pozitiv (Duty Cycle).
- Configurarea ca ieșire a pinului la care este legat ledul DS4.
- Scrierea rutinei de tratare a întreruperilor pentru ambele surse de întrerupere.

Exemplu de cod:

```
/* include files */
#include "pic.h"

/* constant and macro defines */
???

/* function declarations */
void init();
```

```
/* function definitions */

void main()
{
    init();

    while(1)
    {
        ; /* further development */
    }
}

void init()
{
    /* pin configuration */
    ???

    /* timer 1 settings */
    ???

    /* timer 2 settings */
    ???

    /* interrupt settings */
    ???

    /* other initializations */
    ???

}

/* Interrupt function */
void interrupt_isr(void)
{
    /* check if TMR1 interrupt enable and flag are set */
    if((TMR1IE == 1) && (TMR1IF == 1))
    {
        ??? /* insert code here */
    }
    /* check if TMR2 interrupt enable and flag are set */
}
```

```

if((TMR2IE == 1) && (TMR2IF == 1))
{
    ??? /* insert code here */
}
}

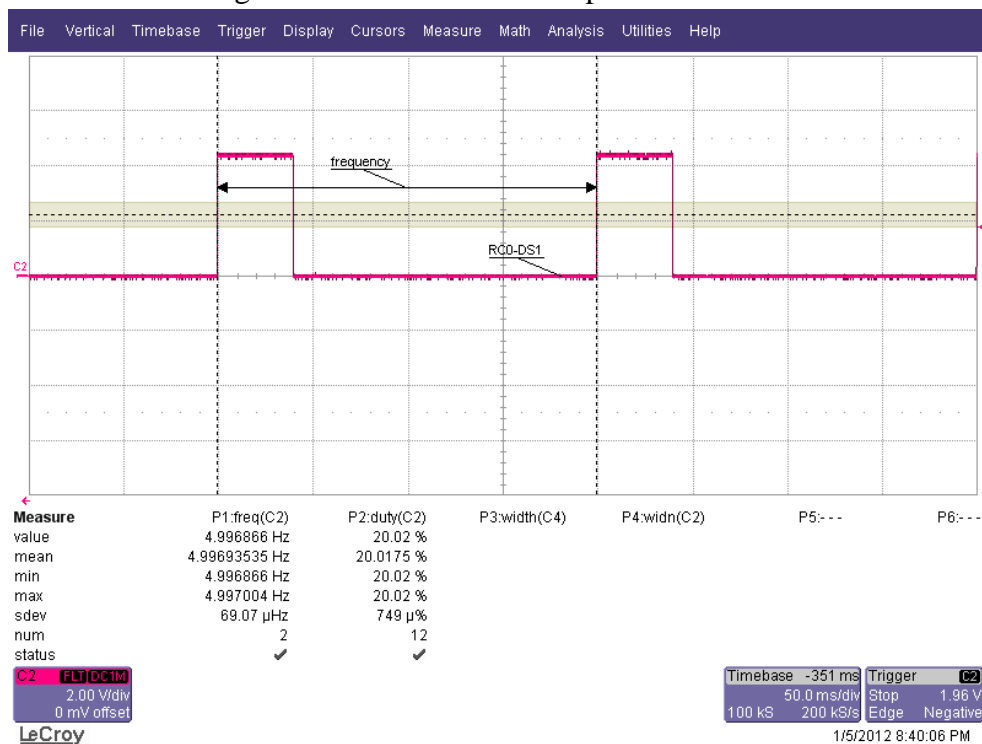
```

8.5. Problemă propusă

Generați un semnal PWM cu frecvență de 5KHz și factor de umplere 20% folosind aceleași 2 periferice (Timer 1 și Timer 2). Pinul pe care semnalul trebuie generat este RC0.

În *Figura 8-5* este prezentat semnalul PWM cu frecvență 5KHz și cu factor de umplere 20% generat pe pinul RC3 (led DS4).

Figura 8-5: Semnal PWM cu perioada 20 ms



9. Convertor Analog Numeric

9.1. Introducere

Conversia analog-numerică reprezintă operația de obținere a unei secvențe numerice de valoare proporțională cu o mărime analogică. În funcție de tipul constructiv, viteză și precizie, un circuit ADC se clasifică în:

ADC cu comparare:

- de tip paralel
- cu tensiune de comparat crescătoare
- cu urmărire
- cu aproximări succesive

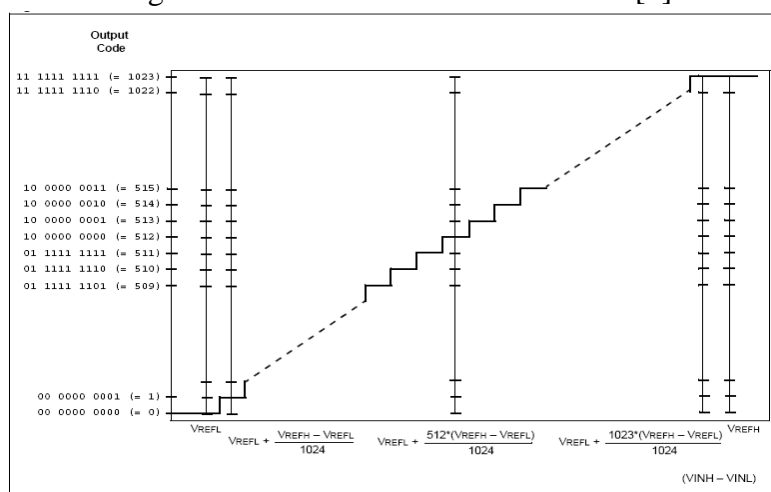
ADC tensiune timp

- cu integrarea unei tensiuni de referință
- cu dublă integrare

ADC tensiune-frecvență

Indiferent de tipul convertorului, mărimea analogică de la intrarea în convertor este discretizată într-un număr de trepte elementare. Acest număr elementar de trepte este dat de numărul de biți pe care se obține rezultatul conversiei și este egal cu 2^N . Cea mai mică valoare convertită diferită de zero este treapta elementară (cuanta): $q = U_R 2^{-N}$

Figura 9-1: Codarea tensiunii de intrare [8]



Două valori posibile consecutive diferă între ele cu q . Cuanta reprezintă, de fapt, chiar rezoluția convertorului.

Pentru o precizie cât mai bună, q trebuie să fie cât mai mic. Cu cât N (numărul de biți pe care se obține rezultatul) este mai mare, pentru aceeași tensiune de referință U_R a convertorului, q va fi mai mic. Inconvenientul care apare la creșterea lui N este creșterea timpului necesar realizării conversiei analog numerice. În practică, de cele mai multe ori, trebuie făcut un compromis între precizie și viteza de realizare a conversiei, în funcție de cerințele aplicației.

O alta modalitate de a îmbunătăți precizia, fără a crește N , este alegerea corespunzătoare a tensiunii de referință U_R . De exemplu, dacă prin natura proiectării aplicației, știm că tensiunea maximă de intrare în convertor este 2.8V, atunci vom impune o tensiune de referință $U_R = 3V$ (cu alte cuvinte, cât mai apropiată de tensiunea maximă de convertit).

Exemplul 1: Dacă avem o tensiune de referință de 5V și un convertor pe 10 biți (1024 de valori posibile), rezoluția este de 4.88mV. Pentru aceeași tensiune de referință, dar de această dată convertorul este pe 8 biți (256 de valori posibile), rezoluția este de 19.53mV.

Exemplul 2: Avem un convertor pe 10 biți, iar tensiunea maximă la intrarea în convertor este de 2.9V. Putem alege o tensiune de referință de 5V, rezoluția fiind în acest caz 4.88mV. Putem îmbunătăți precizia, dacă modificăm tensiunea de referință la o valoare de 3V, rezoluția devenind 2.92mV. Acest lucru va duce și la scăderea timpului necesar eșantionării (în proporții mici).

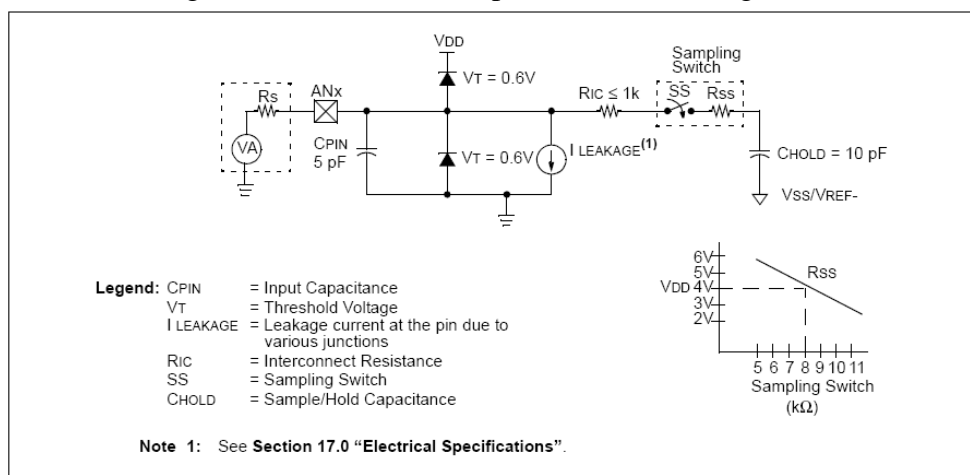
Eșantionarea semnalului analogic se face la intervale de timp bine definite. Cu cât frecvența de eșantionare este mai mare, cu atât se vor pierde mai puține informații din semnalul analogic ce se vrea convertit (vom obține mai multe „mostre”). Timpul minim între două eșantionări T_e trebuie totuși să permită circuitului digital să realizeze conversia analog-numerica. Acest timp depinde în mare măsură de numărul de biți pe care se obține rezultatul conversiei și de tipul constructiv al convertorului.

Conform *teoremei eșantionării*, este necesar ca valoarea minimă a frecvenței de eșantionare să satisfacă relația:

$$f_e \geq 2f_{x\max} \text{ condiția Nyquist.}$$

Unde $f_{x\max}$ este frecvența maximă a spectrului semnalului analogic de intrare U_X , iar f_e este frecvența de eșantionare.

Figura 9-2: Modelul unui pin de intrare analogic [8]



Un pin de intrare analogic are o impedanță foarte mare, pentru a nu modifica funcționarea circuitului în care este conectat. Acest lucru se realizează folosind la intrare un amplificator operațional repetor (prin care se încarcă condensatorul C_{HOLD}). Practic, la închiderea întrerupătorului se realizează eșantionarea, prin încărcarea condensatorului C_{HOLD} până la valoarea tensiunii aplicată pe pinul respectiv. Acest condensator este conectat la intrarea în convertorul analog numeric și el va menține tensiunea constantă pe durata conversiei.

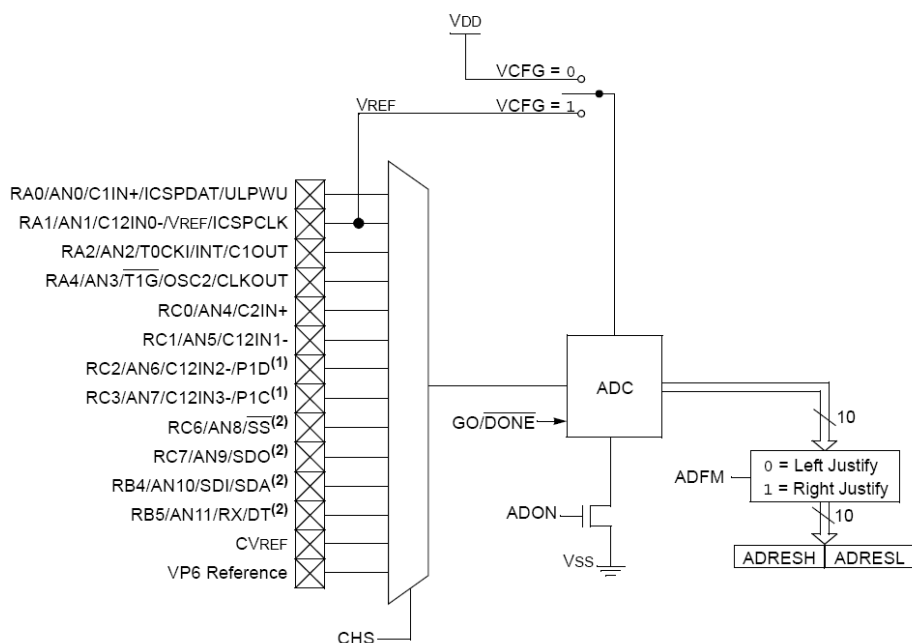
9.2. Descriere ADC pe 10 biți

Microcontrolerul PIC16F690 dispune de un convertor pe 10 biți care are următoarele caracteristici:

- Convertor analog numeric cu aproximări succesive.
- 12 intrări analogice (AN0 – AN11).
- Pini externi pentru tensiunea de referință (selectabil prin software).
- Două tipuri de aliniere a rezultatului conversiei.

În figura de mai jos este prezentată o schemă bloc a convertorului analog numeric.

Figura 9-3: Schema bloc a modului ADC [8]



Modulul ADC are 15 regiștri asociați, care vor fi descriși în continuare. Dintre aceștia, 2 regiștri sunt pentru salvarea rezultatului conversiei și 2 pentru configurarea modului ADC.

Tabel 9-1: Regiștri asociați modului ADC[6]

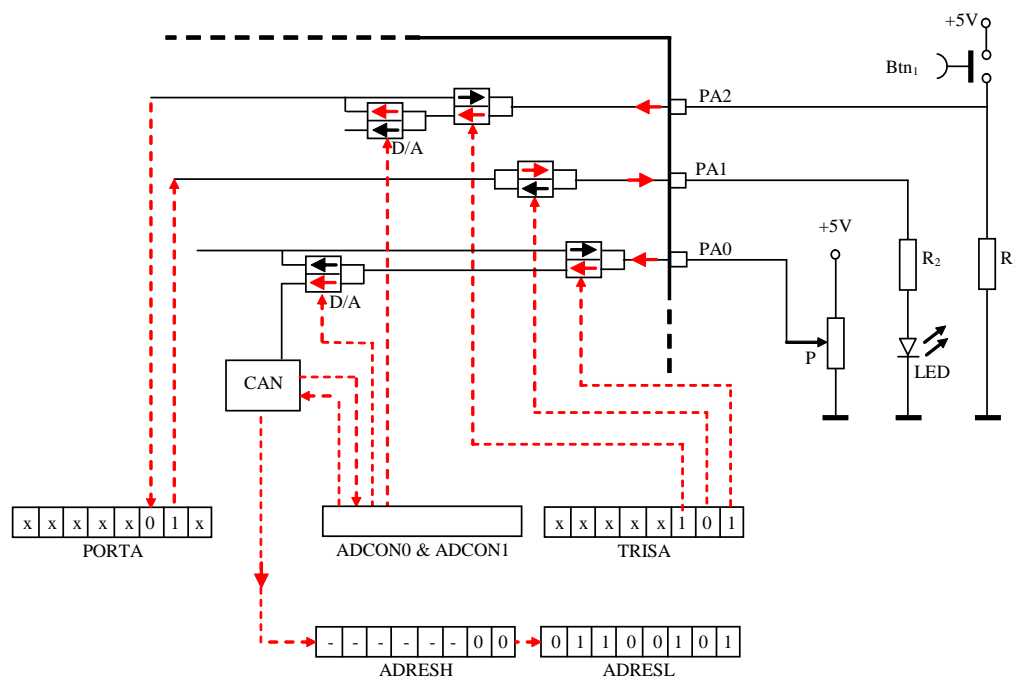
Nume	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
ADCON0	ADFM	VCFG	CH3	CH2	CH1	CH0	GO/DONE	ADON
ADCON1	-	ADCS2	ADCS1	ADCS0	-	-	-	-
ANSEL	ANS7	ANS6	ANS5	ANS4	ANS3	ANS2	ANS1	ANS0
ANSELh	-	-	-	-	ANS11	ANS10	ANS9	ANS8
ADRESH	Rezultatul conversiei A/D. Cel mai semnificativ octet.							
ADRESL	Rezultatul conversiei A/D. Cel mai puțin semnificativ octet.							
INTCON	GIE	PEIE	T0IE	INTE	RABIE	T0IF	INTF	RABIF
PIE1	-	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
PIR1	-	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
PORTA	-	-	RA5	RA4	RA3	RA2	RA1	RA0
PORTB	RB7	RB6	RB5	RB4	-	-	-	-
PORTC	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0
TRISA	-	-	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0

TRISB	TRISB7	TRISB6	TRISB5	TRISB4	-	-	-	-
TRISC	TRISC7	TRISC6	TRISC5	TRISC4	TRISC3	TRISC2	TRISC1	TRISC0

Legendă: x=necunoscut, u=nemodificat, -=bitul se citește ca 0. Biții închiși la culoare nu sunt folosiți de ADC.

Regiștri TRISA, TRISB și TRISC se folosesc pentru alegerea direcției pinilor. Dacă aceștia sunt configurați ca și pini analogici, asigurați modulului ADC, direcția pinilor selectabilă din regiștri de direcție TRISx trebuie să fie input. Regiștri PORTA, PORTB și PORTC se folosesc doar în cazul în care pinii din cele trei porturi sunt setați ca și pini digitali.

Figura 9-4: Asignarea pinilor [3]



Regiștri ADCON0 și ADCON1 sunt regiștri de configurare ai modulului ADC. Cu ajutorul regiștrilor ANSEL și ANSELH se asignează pinii fie modulului ADC, fie portului I/O. Regiștri ADRESH și ADRESL sunt regiștri în care se salvează rezultatul conversiei numărul cuantelor q conținute de tensiunea de convertit. Există 2 moduri de aliniere a rezultatului pe 10 biți.

INTCON, PIR1 și PIE1 sunt regiștri folosiți pentru activarea și lucrul cu întreruperea generată la finalul conversiei analog numerice.

Tabel 9-2: Registrul ADCON0

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	VCFG	CHS3	CHS2	CHS1	CHS0	GO/DONE	ADON
bit 7							bit 0

Legendă: R - bitul poate fi citit; W - bitul poate fi scris; U - bit neimplementat, se va citi 0; n – valoare după POR;
1 - bitul e setat; 0 - bitul e șters; X - valoare necunoscută;

Bit 7 ADFM: Formatul conversiei A/D

- 1 = Aliniat la dreapta
- 0 = Aliniat la stanga

Bit 6 VCFG: Tensiunea de referință

- 1 = Pinul VREF
- 0 = VDD

Bit 5-2 CHS<3:0>: Biții pentru selectarea canalul analogic

- 0000 = AN0
- 0001 = AN1
- 0010 = AN2
- 0011 = AN3
- 0100 = AN4
- 0101 = AN5
- 0110 = AN6
- 0111 = AN7
- 1000 = AN8
- 1001 = AN9
- 1010 = AN10
- 1011 = AN11
- 1100 = CVREF
- 1101 = Referință fixa de 0.6V
- 1110 = Nu este implementat
- 1111 = Nu este implementat

Bit 1 GO/DONE: Bitul de start a conversiei

- 1 = Conversie A/D în execuție. Setarea acestui bit pornește o conversii A/D. Acest bit este șters (devine 0) automat de către hardware la finalul unei conversii.
- 0 = Conversie A/D oprită.

Bit 0 ADON: Bit de activare a modului

- 1 = Modulul ADC este activat
- 0 = Modulul ADC este dezactivat

Tabel 9-3: Registrul ADCON1

U-0	R/W-0	R/W-0	R/W-0	U-0	U-0	U-0	U-0
—	ADCS2	ADCS1	ADCS0	—	—	—	—
bit 7				bit 0			

Legendă: R - bitul poate fi citit; W - bitul poate fi scris; U - bit neimplementat, se va citi 0; n – valoare după POR;
1 - bitul e setat; 0 - bitul e șters; X - valoare necunoscută;

Bit 7 Neimplementat: Se citește '0'.

Bit 6-4 ADCS<2:0>: Bit de selecție a ceasului pentru conversia A/D

000 = FOSC/2

001 = FOSC/8

010 = FOSC/32

X11 = FRC(oscilator intern dedicat = 500kHz max)

100 = FOSC/4

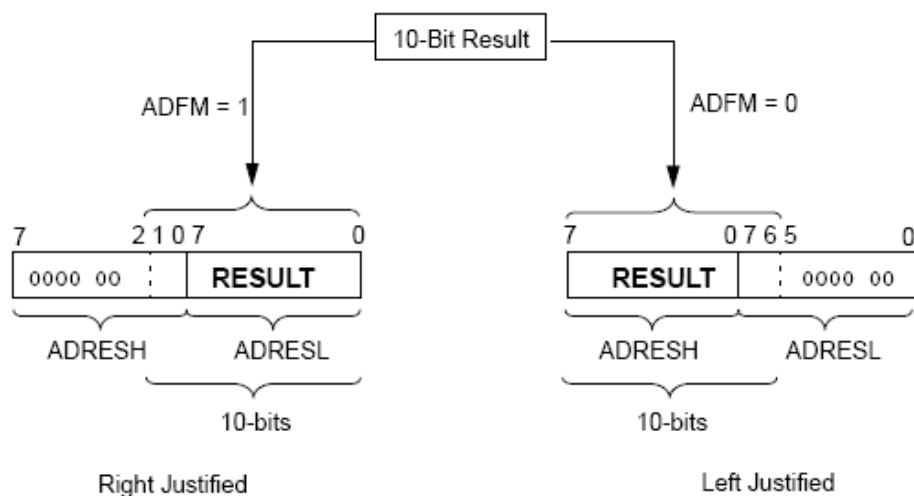
101 = FOSC/16

110 = FOSC/64

Bit 3-0 Neimplementat: Se citește '0'.

Salvarea rezultatului conversiei poate fi făcută în două feluri: *right* sau *left justified*. Dacă alegem modul *right justified*, cei mai puțin semnificativi 8 biți vor fi scriși în ADRESL, iar cei mai semnificativi 2 biți în ADRESH. În cazul selectării modului *left justified*, cei mai puțin semnificativi 2 biți vor fi salvați în ADRESH, iar cei mai semnificativi 8 biți în ADRESL.

Figura 9-5: Alinierea rezultatului conversiei [8]



În ceea ce privește generarea de întreruperi, utilizatorul trebuie să facă următorii pași:

INTCON: registrul de configurare al întreruperilor. Se activează întreruperile generale și a perifericelor prin setarea biților GIE și PEIE.

PIR1: registrul de flag-uri al perifericelor. Conține flag-urile individuale de întrerupere pentru periferice. Se va șterge bitul ADIF (6). Acest bit se va seta automat la finalul conversiei analog numerice.

PIE1: registrul de activare al întreruperilor pentru periferice. Conține biți individuali de activare a întreruperilor pentru periferice. Se va seta bitul ADIE (6), pentru activarea întreruperii generate de modulul ADC.

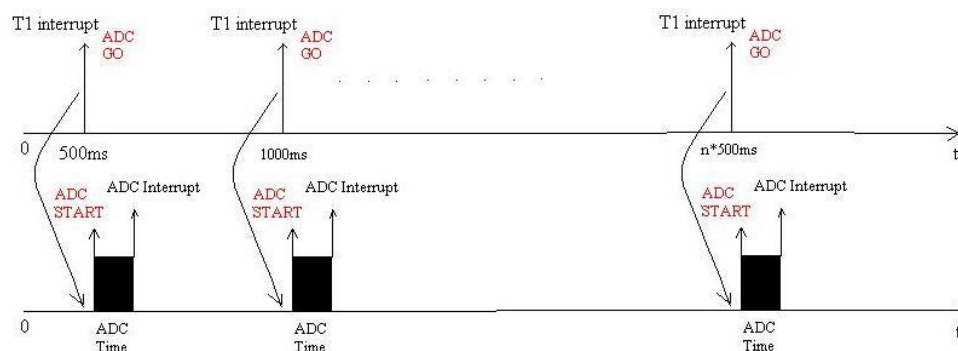
Modul de lucru al convertorului este simplu. După ce setările sunt realizate (se alege pinul de pe care are loc conversia, frecvența de conversie, etc.) procesul de conversie este pornit efectiv cu trigger software prin scrierea bitului GO (bit 2) din registrul ADCON0. La finalul conversiei, bitul GO va fi șters automat de către hardware și va returna în regiștrii ADRSEH și ADRSEL valoarea conversiei, adică o valoare corespunzătoare cu tensiunea de pe pin. Această valoare reprezintă chiar numărul cuantelor q cuprinse în tensiunea de convertit.

Dacă rezoluția conversiei este de 5mV, și valoarea conversiei este 0b000000_1000, atunci tensiunea pe pin este cuprinsă între 40mV și 44.9mV. Aceași regulă (înmulțim numărul de cuante rezultat cu valoarea rezoluției) o aplicăm și pentru valoarea conversiei: 0b010000_1000. Tensiunea pe pin este cuprinsă între 1320mV și 1324.9mV.

9.3. Aplicație propusă

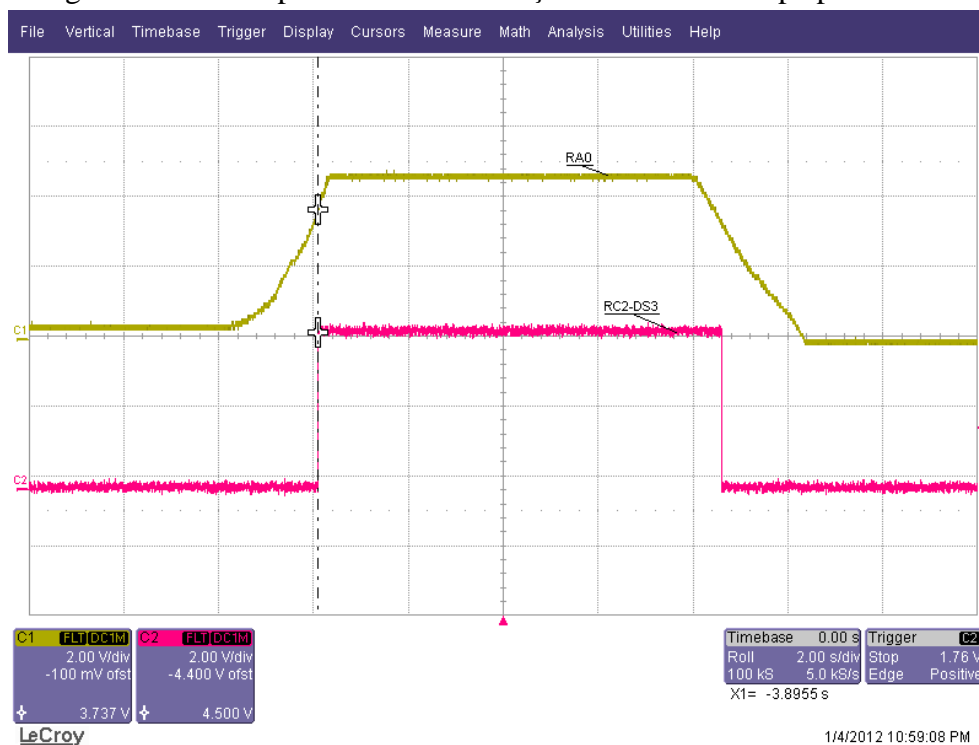
- Să se scrie un program care să aprindă ledul DS3 de pe placă, dacă tensiunea electrică la intrarea pinului de conversie analog numerică AN0 este mai mare de 3V.
- Folosind Timer 1, conversia analog numerică se va realiza la fiecare 500ms.
- Întreruperea modulului ADC trebuie generată la finalul fiecărei conversii.
- Sursa de trigger software a conversiei: setarea bitului GO (prezentată la punctul 9.4).
- Tensiunea de referință să provină de la pinii V_{DD} și V_{SS} .

Figura 9-6: Diagramă cerințe



În *Figura 9-7* este prezentată starea ledului DS3 pentru o tensiune la intrarea convertorului ADC care depășește pragul de 3V (moment în care pinul RC2 devine 1 logic) și care, după un timp, revine sub pragul de 3V (pinul revine și el în starea de 0 logic).

Figura 9-7: Starea pinului RC2 în funcție de tensiunea de pe pinul RA0



9.4. Configurarea ADC

Configurarea modulului ADC presupune următorii pași:

- Selectarea pinilor analogici AN0-AN11:

Deoarece dorim să citim tensiunea aplicată pe pinul AN0, acesta trebuie setat ca și pin de intrare analogic. Bitul ANS0 din registrul ANSEL are valoarea 1, adică AN0 pin analogic (buffer-ul digital de intrare este oprit și orice citire din PORTx va returna valoarea 0 la poziția bitului asignat pinului). Trebuie avut grijă ca și direcția pinului din registrul de direcție TRISA să fie de input. Bitul 0 din acest registru va avea valoarea 1.

- Selectarea sursei tensiunii de referință ADC:

Această tensiune trebuie să fie cel puțin egală cu valoarea maximă ce se vrea a fi convertită. Deoarece tensiunea pe pinul AN0 este aplicată cu ajutorul unui potențiomtru conectat la 5V (deci poate varia între 0V și 5V), tensiunea de referință trebuie să fie 5V. În această aplicație va fi chiar tensiunea de alimentare a modulului ADC de pe pinii VDD și VSS. Setarea se face cu ajutorul bitului VCFG din registrul ADCON0<6>.

Observație: Trebuie precizat că tensiunea de referință poate fi aplicată și pe pinul extern V_{REF+} din mai multe motive. Unul este legat de creșterea preciziei și a fost prezentat în paragrafele anterioare. Un alt motiv important este legat de faptul că în multe aplicații, temperatura de lucru a circuitului poate varia mult, astfel variind și tensiunea de referință. În astfel de situații sunt folosite circuite externe de stabilizare a tensiunii (mai puțin influențate de variația cu temperatura) pentru asigurarea tensiunii de referință a circuitului ADC.

- Selectarea tactului pentru conversie. Biții ADCS<0:2> din registrul ADCON1:

Tactul T_{AD} al modulului analogic ADC controlează timpul de conversie. O conversie completă necesită 12 perioade T_{AD} . Perioada unei conversii analog-numerice se poate configura cu ajutorul biților ACSD<2:0> și trebuie să aibă o valoare minimă de 4.6 μs (pentru 5V). Astfel vom alege ADCS2 = 1 și ACSD<1:0> = 0b00.

- Alegerea formatului rezultatului conversiei (ADFM din registrul ADCON0):

Există două moduri de format. Cel utilizat în această aplicație este de tip *right justify*, adică bitul ADFM = 1.

- Activarea modulului:

Activarea modulului se face prin setarea bitului ADON din registrul ADCON0 (bitul 0). Următorul pas ar fi declanșarea în software a conversiei prin setarea bitului GO, bitul 2 din ADCON0. În momentul scrierii acestui bit, modulul ADC va începe conversia analog numerică a tensiunii regăsite pe pinul analogic de intrare. La finalul conversiei modulul ADC va returna în regiștri ADRSEH și ADRSEL o valoare corespunzătoare cu tensiunea de pe pin (numărul cuantelor q conținute de tensiunea de convertit).

- Activarea întreruperilor, atât pentru Timer 1 cât și pentru modulul ADC:

Trebuie setați biții GIE și PIE din registrul INTCON, biții TMR1IE și ADIE din registrul PIE1. Flag-urile ambelor surse de întrerupere trebuie șterse și anume biții TMR1IF și ADIF din registrul de flag-uri PIR1.

9.5. Model software

În programul scris de noi va trebui să avem în vedere:

- Configurarea Timer 1
- Configurarea modulului ADC
- Configurarea ca și ieșire a pinului la care este legat ledul DS3
- Scrierea rutinei de tratare a întreruperii.

Exemplu de cod:

```
/* include files */
#include "pic.h"

/* constant and macro defines */
???

#define THRESHOLD 3000 /* threshold for turning the LED ON */
#define ADC_TO_mV 5 /* rounded from 4.88 */

/* variables */
volatile unsigned int ADCValue, lastADCValue;
volatile unsigned int milliVolts;

/* function declarations */
void init();
```

```
/* function definitions */
```

```
void main()
```

```
{
```

```
    init();
```

```
    while(1)
```

```
{
```

```
    /* check if last ADC value is different from current read - only if true  
       execute code */
```

```
    if(ADCValue != lastADCValue)
```

```
    {
```

```
        milliVolts = ADCValue * ADC_TO_mV; /* convert from ADC value to  
                                           mV */
```

```
        /* turn LED ON if greater than set threshold */
```

```
        ???
```

```
        * save last ADC value */
```

```
        lastADCValue = ADCValue;
```

```
    }
```

```
}
```

```
}
```

```
void init()
```

```
{
```

```
    /* pin configuration */
```

```
    ???
```

```
    /* timer 1 settings */
```

```
    ???
```

```
    /* ADC configuration */
```

```
    ADCON0 = 0x80; /* right justify; Vref = Vdd; AN0; module turned off */
```

```
    ADCON1 = 0x40; /* Fosc/4 */
```

```
    ADON = 1; /* turn module on */
```

```
    /* interrupt settings */
```

```
    ???
```

```
/* start TMR1 */
TMR1ON = 1;

/* initialize variables */
lastADCValue = 0;
ADCValue = 0;
milliVolts = 0;

}

/* Interrupt function */
void interrupt isr(void)
{
    /* check if TMR1 interrupt enable and flag are set */
    if((TMR1IE == 1) && (TMR1IF == 1))
    {
        TMR1L = ???;
        TMR1H = ???;

        ADCON0 |= 0x02; /* set GO bit to start ADC conversion */

        TMR1IF = 0;          /* clear TMR1 interrupt flag*/
    }
    else if((ADIE == 1) && (ADIF == 1))
    {
        ADCValue = (ADRESH << 8) + ADRESL;
        ADIF = 0; /* clear ADC ISR flag */
    }
}
```

10. UART

10.1. Introducere

UART (Universal Asynchronous Receiver Transmitter) este o interfață de comunicare asincronă serială. Fiecare dintre participanți are propriul circuit de generare a ceasului necesar comunicării și astfel protocolul este asincron (nu există semnal de ceas transmis între participanți). Se mai numește și SCI (Serial Communications Interface).

UART este un protocol de comunicare pe 2 fire, nu necesită master și comunicarea poate fi inițiată simultan de oricare dintre noduri. Acest mod se numește full-duplex. Protocolul poate fi folosit pentru comunicația dintre două microcontrolere aflate la câțiva metri distanță (dacă se folosește cablu torsadat). Ambii participanți la comunicare trebuie să aibă aceleași setări (număr de biți, frecvența semnalului de ceas sau paritatea).

10.2. Descriere modul UART

În funcție de implementarea hardware, modulul UART poate avea diferite caracteristici. Cele mai importante la PIC16F690 sunt:

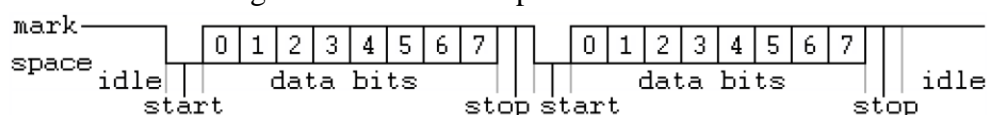
- Datele transmise pot avea o lungime de 8 sau 9 biți, în funcție de setările făcute.
- Datele ce se doresc a fi transmise sunt „împachetate” de către hardware pentru a corespunde cu protocolul, fiind adăugați un bit de start și unul de stop.
- Modulul dispune de pini dedicați de transmisie și de recepție.
- Întreruperi dedicate atât pentru transmisie cât și pentru recepție.
- Un buffer de transmisie și unul de recepție (de tip FIFO cu „adâncime” 2).
- Modulul dispune de biți de stare care indică eroarea de format și eroarea de recepție a unui nou mesaj (când buffer-ul de recepție este plin).
- Când bitul SPEN este setat, pinul RX este configurat ca și pin de intrare, indiferent de starea bitului din registrul TRIS, chiar dacă blocul de recepție nu este activat.

Trebuie adăugat că în unele implementări hardware (nu este cazul microcontrolerului PIC16F690) există și un bit de paritate (pară sau impară)

care face parte (opțional) din formatul mesajului trimis. Acest bit de paritate este calculat și „împachetat” alături de biții de date în formatul mesajului de către participantul care trimite datele. La recepție, acest bit este „despachetat” și comparat cu valoarea de paritate calculată la recepție. În cazul în care diferă, se generează eroare de paritate.

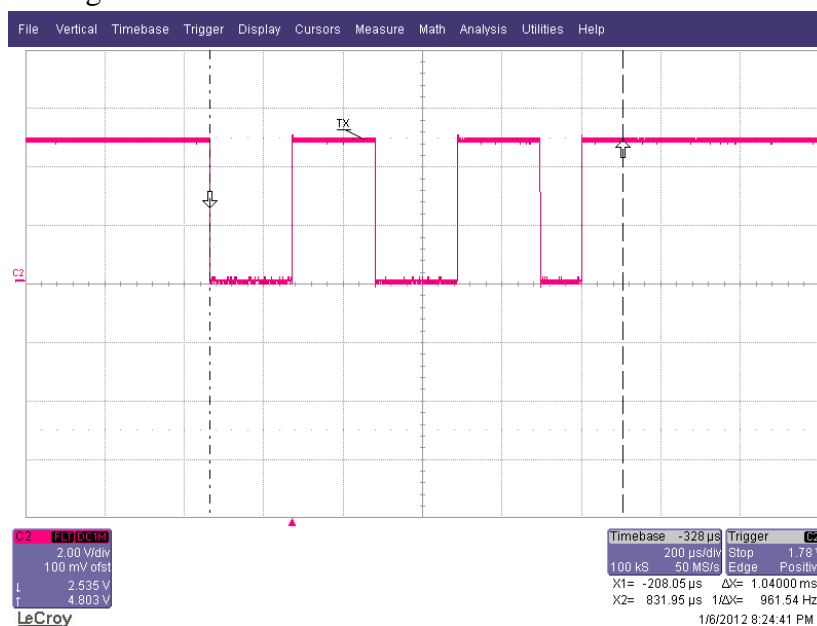
Există și implementări hardware unde, în funcție de setări, putem alege unul sau doi biți de stop.

Figura 10-1: Formatul protocolului UART



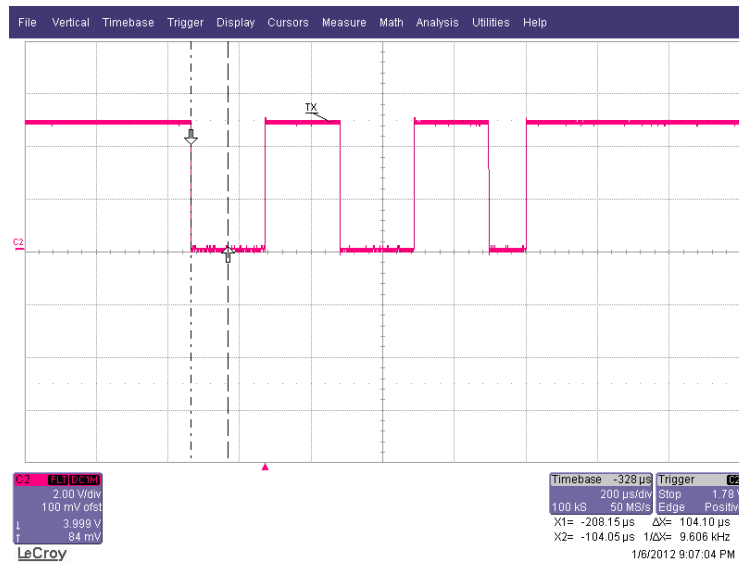
În figurile de mai jos se poate observa un mesaj cu data 0x66 (0b01100110) și cu o rată de transfer de 9600 bit/secundă (adică 140.1 μ s pentru fiecare bit). Mesajul conține un bit de start, primul bit (cu o lățime de 140.1 μ s) ce va fi 0 logic, 8 biți de date (cu o lățime de 8*140.1 μ s și valoarea 0x66) și un bit de stop (cu o lățime tot de 140.1 μ s) ce va fi 1 logic. Lățimea totală a mesajului va fi 1041 μ s (adică 10 biți).

Figura 10-2: Valoarea 0x66 trimisă cu 9600 bit/secunda



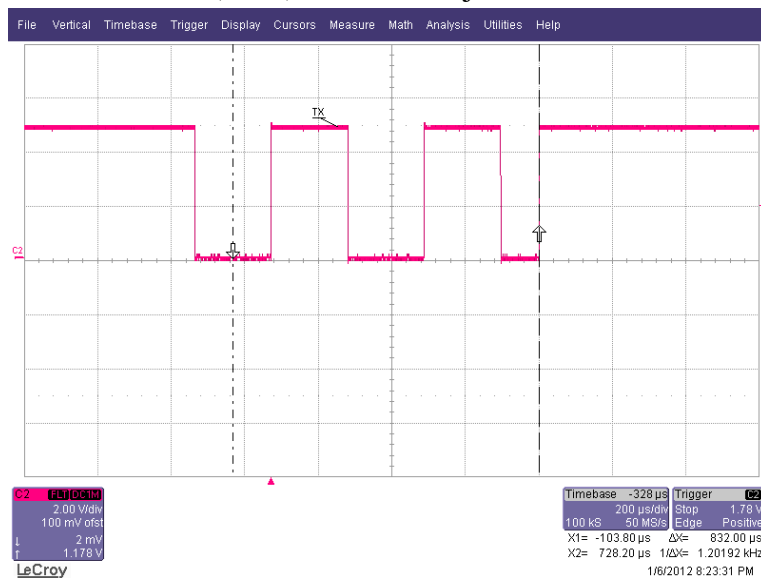
Se poate observa în continuare modul în care datele sunt împachetate în formatul mesajului. Primul este bitul de start, cuprins între cursoare.

Figura10-3: Bit de start pentru un mesaj trimis cu 9600 bit/secunda



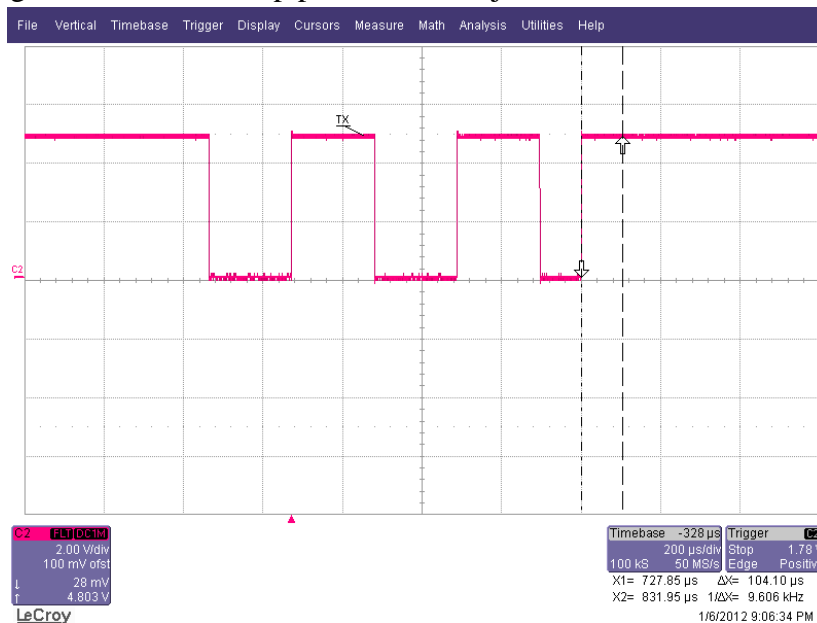
Urmează datele, adică valoarea 0x66 (0b01100110), cu cel mai puțin semnificativ bit trimis primul. Data va avea o lățime de 832.8 μ s, adică 8 biți a câte 104.2 μ s.

Figura 10-4: Datele (0x66) într-un mesaj trimis cu 9600 bit/secunda



Ultimul bit împachetat în formatul mesajului este bitul de stop care are valoarea de 1 logic. În figura următoare bitul de stop este cuprins între cursoare și are o lățime de 104.1 μ s.

Figura 10-5: Bit de stop pentru un mesaj trimis cu 9600 bit/secunda



10.2.1. Blocul de transmisie

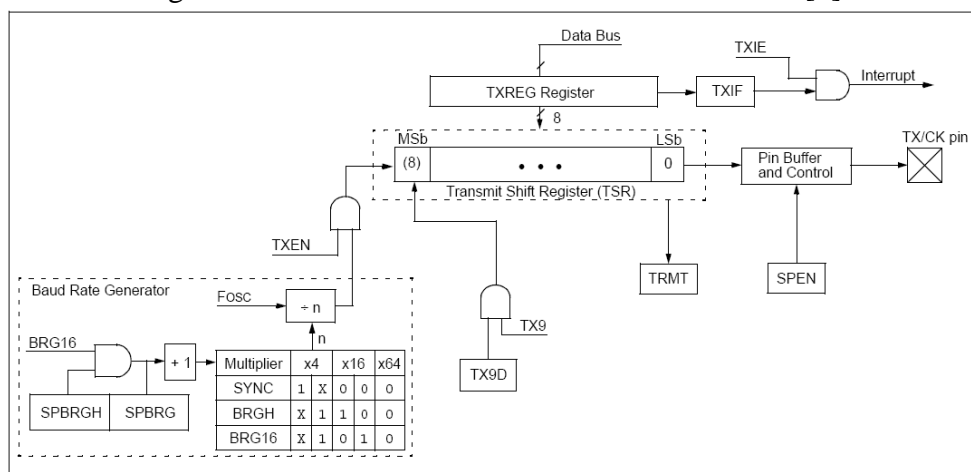
Modulul UART are un pin dedicat de transmisie notat TX. Pe acest pin datele se vor transmite în mod serial, bit după bit, de la cel mai puțin semnificativ, la cel mai semnificativ bit. Schema blocului de transmisie este prezentată în imaginea de mai jos. După ce setările necesare au fost făcute, transmisia începe în momentul în care utilizatorul va scrie în registrul TXREG. Se poate observa că datele scrise în registrul de transmisie TXREG sunt mutate hardware (nu de către o instrucțiune scrisă în software) în Transmit Shift Register, unde vor fi și „împachetate” alături de biții de start și stop, moment în care transmisia va începe și bitul de stare TXIF se va seta.

După scrierea datelor în registrul TXREG se va inițializa trimiterea datelor. Primul bit trimis pe bus-ul TX este bitul de START. Vor urma biții de date (8 sau 9 în funcție de setări) și mesajul se va încheia cu bitul de STOP. Pinul de TX va reveni la finalul transmisiei în starea de așteptare IDLE (va avea starea 1 logic).

Blocul de transmisie este activat prin setarea bitului TXEN. Pentru a transmite date în format de 8 biți, bitul TX9 trebuie să fie 0. Pinul TX (RB7) nu are și funcție analogică, așa că prin activarea modulului (setarea bitului SPEN din registrul RCSTA), pinul va fi asignat modulului UART și va fi pin de ieșire, indiferent de setările făcute în registrul TRISB (la poziția bitului 7).

Dacă se dorește folosirea întreruperilor, bitul TXIE trebuie setat în registrul PIE1, cât și biții PEIE și GIE, din registrul INTCON.

Figura 10-6: Schema blocului de transmisie UART [8]



10.2.2. Blocul de recepție

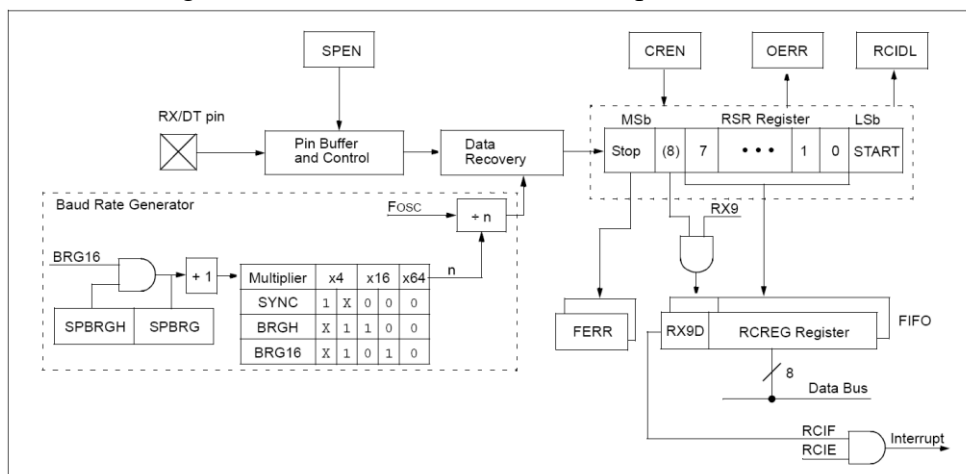
Modulul UART are un pin dedicat de recepție notat RX. Pe acest pin datele se vor recepționa în mod serial, bit după bit, de la cel mai puțin semnificativ, la cel mai semnificativ bit. Schema blocului de recepție este prezentată în imaginea de mai jos. După ce setările necesare au fost făcute, recepția poate începe dacă celălalt participant la comunicație va transmite date. După ce datele „împachetate” sunt recepționate, ele vor fi „despachetate” (se îndepărtează bitul de start și cel de stop) și se vor regăsi în buffer-ul de recepție RSR. Biții de date sunt mutați de către hardware în registrul RCREG (de tip FIFO cu adâncime 2). Chiar dacă primul mesaj recepționat nu este citit, un al doilea poate fi recepționat, deoarece FIFO-ul are două intrări disponibile. Dacă un al treilea mesaj sosește, acesta nu va fi mutat în FIFO (deoarece este plin) și va rămâne în Shift Register. Mesajele ulterioare nu vor mai fi recepționate până nu se citește RCREG.

Când datele sunt mutate (automat de către hardware) după recepție din buffer în registrul RCREG, bitul de stare RXIF va fi setat. Dacă și bitul RXIE este setat, o întrerupere va fi generată.

Blocul de recepție este activat prin setarea bitului SPEN și CREN. Pentru a recepționa date în format de 8 biți, bitul RX9 trebuie să fie 0. Pinul RX (RB5) are și funcție analogică AN11, așa că pe lângă activarea modulului (setarea bitului SPEN din registrul RCSTA) și activarea blocului de recepție (setarea bitului CREN din registrul RCSTA), pinul trebuie asignat modulului UART (devine pin digital) prin scrierea bitului ANSELH cu 0 logic.

Dacă se dorește folosirea întreruperilor, bitul RXIE trebuie setat în registrul PIE1, cât și biții PEIE și GIE, din registrul INTCON.

Figura 10-7: Schema blocului de recepție UART [8]



Blocul de recepție dispune și de biți de stare care să indice posibilele erori de comunicare. Unul din biți este bitul FERR, care indică o posibilă eroare de format a mesajului recepționat. Al doilea bit de stare, numit OERR, indică o eroare de umplere a FIFO de recepție (mai mult de trei mesaje au fost recepționate fără ca registrul RCREG să fie citit).

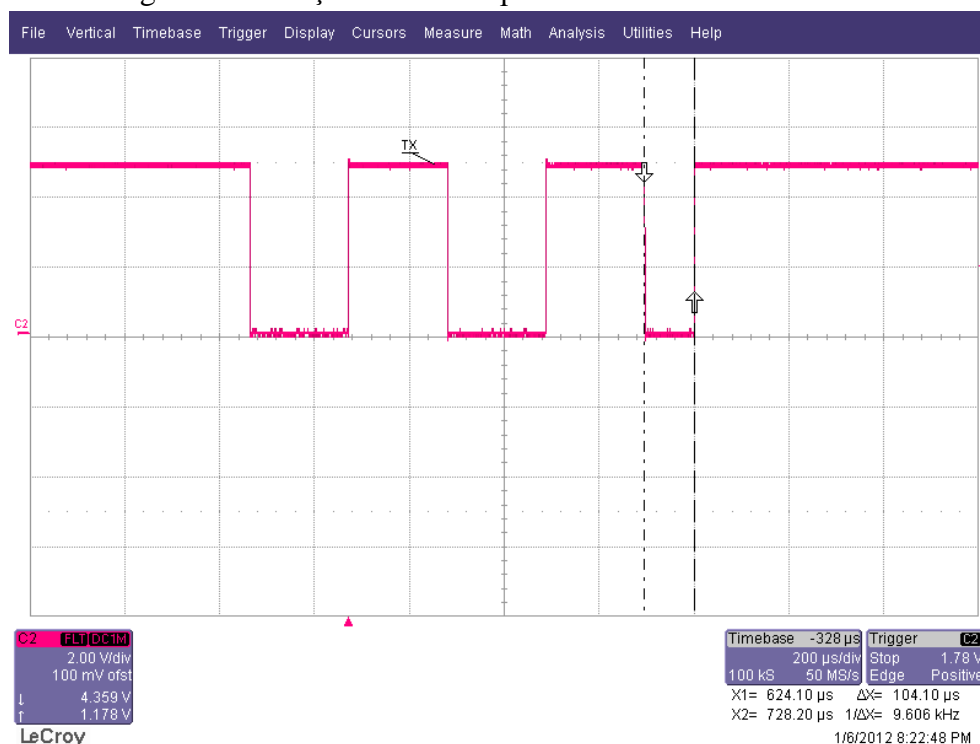
10.2.3. Setarea ceasului (rata de transfer)

Pentru o comunicare corectă între doi participanți, ambii trebuie să aibă aceeași rată de transfer, adică să transmită la fel de mulți biți pe secundă.

Aceasta este și unitatea de măsură, biți pe secundă. În limba engleză rata de transfer se numește baud rate.

Exemplu: Dacă rata de transfer este 9600, atunci, într-o secundă, vor fi trimiși 9600 de biți, iar lățimea unui bit este 104.1 μ s.

Figura 10-8: Lățimea unui bit pentru o rată de transfer 9600



Rata de transfer poate fi generată cu ajutorul a doi regiștri: SPBRGH și SPBRG, fie pe 8 biți, fie pe 16 biți, în funcție de valoarea biților BRG16 și BRGH. Dacă bitul BRG16 este 0, atunci blocul de generare a ceasului folosit pentru rata de transfer va funcționa ca și un timer pe 8 biți și doar registrul SPBRG este folosit. Dacă bitul BRG16 este setat, atunci ambii regiștri vor fi folosiți și se vor comporta ca un timer pe 16 biți.

În documentația microcontrolerului este dată, în funcție de setările făcute și de frecvența oscilatorului, valoarea ce trebuie scrisă în registrul SPBRG pentru rata de transfer dorită.

Biții folosiți pentru setarea ceasului sunt:

- SYNC: cu care selectăm dacă modulul funcționează asincron sau sincron.
- BRGH: cu care alegem multiplicatorul (16 sau 64).
- BRG16: cu care alegem 8 sau 16 biți pentru generarea ceasului.

Tabel 10-1: Biții de configurare ai ratei de transfer [8]

Biți de configurare			Mod de funcționare UART	Formulă calcul
SYNC	BRG16	BRGH		
0	0	0	8 bit/ Asincron	$F_{osc}/[64*(n+1)]$
0	0	1	8 bit/ Asincron	$F_{osc}/[16*(n+1)]$
0	1	0	16 bit/ Asincron	
0	1	1	16 bit/ Asincron	$F_{osc}/[4*(n+1)]$
1	0	X	8 bit/ Asincron	
1	1	X	16 bit/ Sincron	

Pentru calculul valorii registrul SPBRG pentru o anumită rată de transfer se pot folosi formulele din Tabelul 10-1.

10.2.4. Regiștri de configurare ai modulului UART

Tabel 10-2 Regiștri modulului UART [8]

Nume	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
INTCON	GIE	PEIE	T0IE	INTE	RABIE	T0IF	INTF	RABIF
PIE1	-	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
PIR1	-	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
RCSTA	Registrul de recepție a datelor							
TXREG	Registrul de transmisie a datelor							
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
TXSTA	CSRC	TX9	TXEN	SYNC	SENDER	BRGH	TRMT	TX9D
BAUDCTL	ABDOVF	RCIDL	-	SCKP	BRG16	-	WUE	ABDEN
SPBRG	Registrul pentru calculul ratei de transfer. Cei mai puțin semnificativi 8 biți.							
SPBRGH	Registrul pentru calculul ratei de transfer. Cei mai semnificativi 8 biți.							
TRISB	TRISB7	TRISB6	TRISB5	TRISB4	-	-	-	-
ANSELH	-	-	-	-	ANS11	ANS10	ANS9	ANS8

Legendă: x=necunoscut, u=nemodificat, -=bitul se citește ca 0. Biții închiși la culoare nu sunt folosiți de UART.

Regiștri INTCON, PIE1 și PIR1 sunt regiștri asociați întreruperilor pentru transmisie sau recepție ce pot fi generate de modulul UART. Acești regiștri au fost prezentați și în capitolele anterioare. În cadrul acestui capitol nu se vor folosi sursele de întrerupere ale modulului. Regiștri TRISB și ANSELH sunt și ei regiștri ce au fost prezentați anterior și sunt folosiți pentru configurarea direcției și modulului digital/analog a pinilor.

Registrul TXREG este registrul de transmisie. Dacă modulul este activ și corect inițializat, fiecare scriere în TXREG va genera o transmisie a datelor scrise. Registrul RCREG este registrul de recepție. Datele recepționate pe pinul RX se vor regăsi în registrul RCREG de unde pot fi citite. Regiștri SPBRG și SPBRGH sunt regiștri ce conțin valoarea pre-scalarului ce intră în formula de calcul a ratei de transfer. Dacă bitul BRG16 din registrul BAUDCTL este setat 0 logic atunci în calculul formulei va intra doar valoarea scrisă în registrul SPBRG.

$$\text{Rata transfer} = F_{\text{osc}} / [64 * (\text{SPBRGH}:\text{SPBRG} + 1)]$$

În continuare vor fi prezentați regiștri de configurare ai modulului pentru modul asincron și transfer pe 8 biți. Unii biți pot avea funcții și pentru modul de lucru sincron dar, dacă se dorește folosirea acestui mod, trebuie consultată documentația microcontrolerului pentru informații complete asupra acestor biți.

Tabel 10-3: Descriere registrului TXSTA [8]

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN ⁽¹⁾	SYNC	SENDB	BRGH	TRMT	TX9D
bit 7						bit 0	

Legendă: R - bitul poate fi citit; W - bitul poate fi scris; U - bit neimplementat, se va citi 0; n – valoare după POR; 1 - bitul e setat; 0 - bitul e șters; X - valoare necunoscută;

Bit 7 CSRC: Bit de selecție a ceasului

În mod asincron acest bit este ignorat

Bit 6 TX9: Bit pentru activarea transmisiei bitului 9

1 = Transmisie pe 9 biți

0 = Transmisie pe 8 biți

Bit 5 TXEN: Bit de activare a blocului de transmisie⁽¹⁾

1 = Blocul de transmisie e activ

0 = Blocul de transmisie e dezactivat

Bit 4 SYNC: Bit de selecție a modului de funcționare

1 = Mod sincron

0 = Mod asincron

Bit 3 SENDB: Transmisia caracterului "Break"

1 = Următoarea transmisie va fi un mesaj de tip "Break" (bitul este șters automat de către hardware după transmisia mesajului)

0 = Nu se va transmite un mesaj de tip "Break"

Bit 2 BRGH: Bit de selecție a ratei de transfer ridicată

1 = Viteză ridicată

0 = Viteză scăzută

Bit 1 TRMT: Bit de stare a registrului de transmisie TSR

1 = TSR este plin (conține un mesaj ce urmează a fi transmis)

0 = TSR este gol (poate fi scris pentru a iniția o transmisie)

Bit 0: TX9D: Bitul 9 de date

Conține valoarea celui de al-9-lea bit. Poate fi bit de adresă, date sau paritate

⁽¹⁾SREN/CREN suprascrie TXEN în mod sincron

Tabel 10-4: Descriere registrului RCSTA [8]

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

Legendă: R - bitul poate fi citit; W - bitul poate fi scris; U - bit neimplementat, se va citi 0; n - valoare după POR; 1 - bitul e setat; 0 - bitul e șters; X - valoare necunoscută;

Bit 7 SPEN: Bit de activare a portului serial

1 = Portul serial este activ (pinii RX și TX sunt asigurați portului serial)

0 = Portul serial nu este activ (este ținut în Reset)

Bit 6 RX9: Bit pentru activarea recepției bitului 9

1 = Recepție pe 9 biți

0 = Recepție pe 8 biți

Bit 5 SREN: Bit de activare a unei singure recepții

Bitul este ignorat în mod asincron

Bit 4 CREN: Bit de activare a recepției în mod continuu

În mod asincron:

1 = Blocul de recepție este activ

0 = Blocul de recepție este inactiv

Bit 3 ADDEN: Activarea detecției bitului de adresă

În mod asincron cu transmisie pe 9 biți (RX9 = 1)

1 = Detecția adresei este activată

0 = Detecția adresei este inactivă. Al-9-lea bit este recepționat și poate fi folosit ca și

bit de paritate (în software).

În mod asincron cu transmisie pe 8 biți (RX8 = 1) bitul este ignorat

Bit 2 FERR: Eroare de format

1 = O eroare de format a fost detectată

0 = Nici o eroare de format nu a fost detectată

Bit 1 OERR: Eroare de Overrun

1 = O eroare de Overrun a fost detectată

0 = Nici o eroare de Overrun nu a fost detectată

Bit 0: RX9D: Bitul 9 de date

Conține valoarea celui de al-9-lea bit. Poate fi bit de adresă, date sau paritate

Tabel 10-5: Descriere registrului BAUDCTL [8]

R-0	R-1	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
ABDOVF	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN
bit 7							bit 0

Legendă: R - bitul poate fi citit; W - bitul poate fi scris; U - bit neimplementat, se va citi 0; n - valoare după POR;
1 - bitul e setat; 0 - bitul e șters; X - valoare necunoscută;

Bit 7 ABDOVF: Bit de stare a circuitului de detecție automată a ratei de transfer

1 = Timer-ul circuitului a ajuns la valoarea maximă (Overflow)

0 = Timer-ul circuitului nu a ajuns la valoarea maximă

Bit 6 RCIDL: Bit de stare a blocului de recepție

În mod asincron:

1 = Recepția este în așteptare (nu se recepționează un mesaj)

0 = Un bit de start s-a recepționat și recepția este în curs

Bit 5: Neimplementat: Bitul se citește '0'

Bit 4 SCKP: Bit de selecție a polarității sincrone

În mod asincron:

1 = Datele transmise pe pinul TX sunt negate

0 = Datele transmise pe pinul TX nu sunt negate

Bit 3 BRG16: Generatorul ratei de transfer pe 16 biți

1 = Generatorul ratei de transfer folosește 16 biți (SPBRGH:SPBRG)

0 = Generatorul ratei de transfer folosește 8 biți (SPBRG)

Bit 2 Neimplementat: Bitul se citește '0'

Bit 1 WUE: Bit de activare a wake-up

În mod asincron

1 = Blocul de recepție așteaptă un front descrescător

0 = Blocul de recepție funcționează în mod normal

Bit 0 ABDEN: Bit de activare a circuitului de detecție automată a ratei de transfer

1 = Circuitul este activ

0 = Circuitul nu este activ

10.3. Aplicație propusă

Să se scrie un program care să trimită pe pinul TX, într-o buclă infinită, toate valorile decimale de la 0 la 100. Pinul RX va fi legat printr-o sârmă de pinul TX (loopback mode) iar la recepția valorii 90 decimal, ledul DS1 de pe placă trebuie aprins, indicând recepția. Transferul trebuie să fie realizat pe 8 biți și cu o rata de transfer de 9600baud (9600 biți/secunda).

10.4. Configurarea modulului UART

Funcția de configurare a modulului UART trebuie să îndeplinească următorii pași:

- a) Setarea corectă a ratei de transfer. În paginile de catalog este dată formula de calcul a ratei de transfer. În Tabelul 10-1 se găsește formula de calcul în funcție de biții SYNC, BRGH și BRG16. Pentru cerința de mai sus alegem următoarea configurație:
 - SYNC = 0 pentru mod asincron
 - BRG16 = 1 pentru generarea ratei de transfer se folosesc 16 biți
 - BRGH = 0 pentru viteză scăzută

Formula de calcul rezultată este: $Rat\ de\ transfer = F_{osc} / [16 * (n + 1)]$.

Ținând cont de faptul că F_{osc} este 4MHz și că rata de transfer dorită este 9600, valoarea ce trebuie scrisă în registrul SPBRG este 25.

- b) Alegerea modului asincron. Acest lucru se face prin scrierea bitului SYNC din registrul TXSTA cu valoarea 0. Chiar din setarea ratei de transfer acest pas a fost realizat.
- c) Selecția transmisiei și recepției pe 8 biți. Bitul TX9 din registrul TXSTA și RX9 din RCSTA trebuie scriși cu valoarea 0.
- d) Activarea blocului de transmisie prin setarea bitului TXEN din registrul TXSTA. Direcția de ieșire se selectează din registrul TRISB.
- e) Activarea blocului de recepție prin setarea bitului CREN din registrul CREN. Pentru ca datele ce ajung la pinul RX (RB5/AN11/RX) să ajungă și fizic în registrul de recepție, acest pin trebuie „legat” intern la registrul. Pinul trebuie să aibă funcție digitală pentru ca acest lucru să se

întâmplă. După reset pinul are funcție analogică (AN11) și funcțiile sale digitale sunt blocate. Prin scrierea cu valoarea 0 a bitului ANS11 din registrul ANSELH, pinul devine digital și biții care ajung la pinul RX se vor regăsi și în registrul RCREG.

- f) Activarea modulului UART prin scrierea bitului SPEN din registrul RCSTA. În acest moment orice scriere în registrul TXREG va iniția o transmisie de data pe pinul TX. Se vor transmite într-o buclă toate valorile decimale între 0 și 100.

Figura 10-9: Primele 5 mesaje trimise



Dacă pinul TX este legat cu pinul RX printr-o sârmă, datele transmise pe pinul TX prin scrierea în registrul TXREG vor fi recepționate pe pinul RX și se pot citi din registrul RCREG. Bineînțeles că între scrierea în registrul TXREG și citirea datelor din registrul RCREG trebuie un timp de așteptare necesar realizării fizice a transmisiei. Acest timp se poate elimina prin folosirea întreruperii la recepție și citirea datelor în rutina de tratare a întreruperii. În acest fel ne asigurăm că datele sunt citite doar după ce recepția este completă. În cerință și exemplul dat nu se folosește întreruperea la recepție. Atâta timp cât nu s-a recepționat valoarea 90, pinul

RC0 trebuie să rămână 0 logic. Acest lucru se poate observa în figura următoare. La recepția mesajului cu valoarea 90, pinul își schimbă starea în 1 logic.

Figura 10-10: Recepția mesajului cu valoare 90



10.5. Model software

În programul scris de noi va trebui să avem în vedere:

- Scrierea valorii n în registrul SPBRG pentru rata de transfer 9600.
- Selectarea modului asincron și formatului pe 8 biți.
- Activarea blocului de transmisie TX și cel de recepție RX.
- Activarea portului serial.
- Scrierea datelor în registrul TXREG pentru a iniția transmisia.
- Citirea datelor din registrul RCREG pentru detectarea mesajului cu valoarea 90.

Exemplu de cod:

```

/* include files */
#include "pic.h"

/* macro*/
#define LED   ???

/* function declarations */
void init();

/* function definitions */
void main()
{

    unsigned int i, j;    /* local variables*/
    init();

    while(1)              /* infinit loop*/
    {
        for (i = 0; i<101; i++)    /* loop to send 101 messages*/
        {

            TXREG = i; /* send all values from 0 to 100*/
            for (j = 0; j<100; j++); /* small delay between messages*/
            if (RCREG == 90)/* read the received data and compare it to 90*/
            {
                LED = ???;    /* turn on DS1*/
            }
        }
    }
}

void init()
{

    /* Pin settings for LED's*/
    ANSEL = 0x0F; /* set RC0 to RC3 as digital pins */
    TRISC = 0xF0; /* RC4 to RC7 input. RC0 to RC3 output */
    PORTC = 0x00; /* port C pins reset value. LED's off*/
}

```

```

/* Baud setting */
/* SYNC = 0; BRG16 = 1; BRGH = 0 - find the SPBRG value in the
   datasheet (Table 12-5)
   /           /           /
   /           /           ---- low speed
   /           -----16 bit Baud Rate Generator is used
   -----Asyncon mode */

BAUDCTL = 0x08; /* BRG16 = 1 */
SPBRG = ?????; /* this value is from the datasheet or using the correct
                formula*/

/* TX settings */
TRISB &= 0x7F; /* TX/RB7 output */
TXSTA = 0x20; /* TX enabled; asyncon mode (SYNC = 0); 8-bits;
               BRGH = 0 */

/* RX settings */
ANSELH = 0x07; /* set RB5 as digital pins. This is analogic by default*/
TRISB |= 0x20; /* RX/RB5 input */
RCSTA = 0x10; /* RX enabled; 8-bits */

/* Turn on UART*/
RCSTA |= 0x??; /*UART enabled*/
}

```

Anexa 1 - Detalii suplimentare Embedded C

Anexa 1 prezintă câteva elemente ale limbajului C ce nu au fost explicate în Capitolul 2. Acestea sunt descrise pe scurt, sub forma de listă, având ca scop familiarizarea cititorului cu cât mai mult elemente ale limbajului de programare.

Const este un calificativ care, atribuit definiției unei variabile, o transformă pe această într-o constantă, adică devine read-only. Precum o variabilă normală, constanta creată va avea un tip, un nume generic și o adresă, diferența fiind că, o dată atribuită o valoare, aceasta nu mai poate fi modificată.

```
const unsigned int a = 10;
```

Pe parcursul codului se poate folosi noua constantă cu numele ei generic, atribuirea unei alte valori fiind imposibile. În aplicațiile embedded o constantă va fi stocată în memoria de tip read-only (ROM, Flash) și nu în RAM precum în cazul variabilelor.

Volatile este la rândul său un calificativ care se atribuie variabilelor și care semnalează compilatorului să nu optimizeze accesele la acea variabilă. În aplicațiile embedded sunt dese momentele în care se citesc regiștri de stare care se pot schimba independent de aplicația care rulează pe microcontroler. Considerăm exemplul următor:

```
while(reg_read == 0)  
{ ;}
```

Presupunem că registrul pe care îl citim în bucla anterioară este zero la început, primind o valoare diferită doar după un anumit timp (ex: se setează un bit pentru un mesaj recepționat). Dacă variabila *reg_read* nu este declarată ca fiind *volatile*, compilatorul ar putea optimiza codul și în loc să citească valoarea de la adresa registrului de fiecare dată, o va citi o singură dată, la început și aceasta fiind 0, bucla va deveni infinită. Declarând variabila ca fiind *volatile* forțează compilatorul să aducă de fiecare dată valoarea din memorie, în cazul nostru, din registrul pe care dorim să îl citim.

Extern este un cuvânt cheie ce se atribuie variabilelor și funcțiilor în contextul proiectelor formate din mai mult fișiere sursă. În cazul proiectelor complexe, este imposibil ca tot codul să fie în același fișier *.c. Din acest motiv, se pot folosi mai mult fișiere *.c și *.h în care să se scrie codul aplicației iar pentru a putea interacționa între ele, se folosesc variabilele și funcțiile extern.

Dacă declarăm o variabilă ca fiind *extern* (*extern unsigned int a;*) într-un fișier test.h, aceasta va putea fi folosită în orice fișier *.c care include test.h. Astfel, mai multe module software pot interacționa între ele prin valorile salvate în variabila *a*. În mod asemănător, declarăm o funcție ca fiind extern într-un fișier *.h, apoi o definim într-un fișier *.c. Incluzând fișierul *.h (în care funcția a fost definită), putem apela funcția respectivă din mai multe module.

Static este, în mare măsură, opusul lui extern. Declarând o variabilă sau funcție ca fiind static le transformă în membri locali ai modulului. Un membru local nu poate fi accesat în afara fișierului unde a fost declarat (*static unsigned int b;*). Pe lângă acest înțeles, static mai este folosit și în cadrul funcțiilor la declararea variabilelor. Considerăm exemplu:

```
void function (void)
{static unsigned int a = 0;
  a++;
}
```

În cadrul funcției prezentate anterior am declarat o variabilă static locală acelei funcții. O astfel de variabilă are ca și proprietate de bază faptul că, după ce apelul funcției s-a încheiat, ea își păstrează valoarea. În exemplul anterior, la al doilea apel al funcției, variabila *a* va avea valoarea 1 ea fiind inițializată cu valoarea 0 doar la primul apel.

Un **vector** (tablou sau array) este o structura de date în care se păstrează informație de același tip în locații succesive de memorie. Pentru a accesa diversele locații de memorie e suficient să indexăm vectorul respectiv.

```
unsigned int arr[10];
for(i=0;i<10;i++)
{
  arr[i] = i;
}
```

Codul prezentat anterior declară un vector numit *arr* de 10 locații pentru variabile *unsigned int*. Indexarea se face folosind parantezele pătrate *arr[i]*. Vectorii sunt un mod foarte eficient de a ține date ce se vor folosi în bucle *for* sau *while*. Dacă în exemplul anterior nu am fi folosit un vector, codul rezultat ar fi fost mult mai complex, cu 10 declarații de variabile și 10 inițializări diferite.

Pointerul este unul dintre cele mai puternice mecanisme ale limbajului C și unul dintre motivele pentru care limbajul se folosește în aplicațiile embedded. În cel mai simplu mod spus, un pointer este o variabilă ce conține o adresă spre un alt element. Pentru a declara un pointer, trebuie să specificăm tipul de date spre care el ne trimite: *unsigned int * ptr*. Această linie de cod declară un pointer către o variabilă *unsigned int*. Dimensiunea lui *ptr* nu este egală cu cea a variabilei spre care ne trimite (*int* - 16 biți) ci are o mărime ce depinde de platforma pe care rulăm codul. În exemplul următor putem observa cum sunt folosiți pointerii:

```
unsigned int * ptr;  
unsigned int a = 10;
```

```
ptr = &a;  
*ptr = 20;
```

ptr este declarat ca pointer la *unsigned int* și *a* este o variabilă *unsigned int* care primește valoarea 10. *ptr* nu este inițializat la declarare, dar el primește adresa lui *a* prin folosirea operatorului *&*. Pentru a modifica valoarea spre care *ptr* ne trimite, folosim operatorul *** precum în linia **ptr = 20;*. După execuția acestui cod, în *ptr* vom găsi adresa lui *a* iar în *a* vom avea valoarea 20.

Pentru a demonstra modul în care pointerii sunt folosiți în aplicațiile embedded, vom considera un exemplu în care dorim să citim informațiile dintr-un registru de 8 biți aflat la adresa 0x1000 și apoi să scriem în acest registru valoarea 0xAA.

```
volatile unsigned char * ptr_reg;  
unsigned char reg_read = 0;
```

```
ptr_reg = (unsigned char *) 0x1000;  
reg_read = *ptr_reg;  
*ptr_reg = 0xAA;
```

În exemplul anterior se observă cum *ptr_reg* a fost declarat pointer către un *volatile unsigned char* deoarece ne va trimite la un registru (*volatile*) de 8 biți (*unsigned char*). Spre deosebire de primul exemplu în care am folosit operatorul *&* pentru a afla adresa dorită, în cazul de față am scris direct adresa dorită în *ptr_reg*, fără a folosi alt operator. Apoi, pentru a citi și scrie la adresa respectivă, am folosit operatorul ***.

Struct este un cuvânt cheie folosit pentru a crea noi structuri de date. Presupunem că, în aplicația noastră, dorim să folosim un calendar și avem nevoie de variabile care să păstreze data unei zile. În loc să declarăm trei variabile pentru fiecare dată, vom crea un nou tip folosind structuri:

```
struct data
{
    unsigned int anul:
    unsigned char luna:
    unsigned char ziua:
}
```

După ce am construit noul tip de dată putem să declarăm variabile de acest tip după cum urează:

```
struct data test;
test.anul = 2012;
test.luna = 1;
test.ziua = 15;
```

În codul anterior se poate observa cum declarăm o variabilă numită *test* de tip *data*. Apoi, folosind operatorul *.*(punct) accesăm fiecare membru al acelei variabile pentru a o inițializa cu o nouă valoare.



Dat fiind faptul că lucrarea de față nu are ca temă limbajului C, prezentarea noastră se va opri aici. Pentru o mai bună înțelegere a elementelor de programare în acest limbaj, recomandăm consultarea materialelor dedicate acestui subiect.

Anexa 2 - Programarea microcontrolerului

Anexa 2 va prezenta scrierea codului (fișierul *.hex) în memoria microcontrolerului.

După scrierea codului în mediul de dezvoltare MPLAB și compilarea acestuia, vom obține un fișier *.hex care conține codul mașină al programului. Acesta trebuie scris în memoria program (Flash) a microcontrolerului, de unde va fi rulat instrucțiune cu instrucțiune.

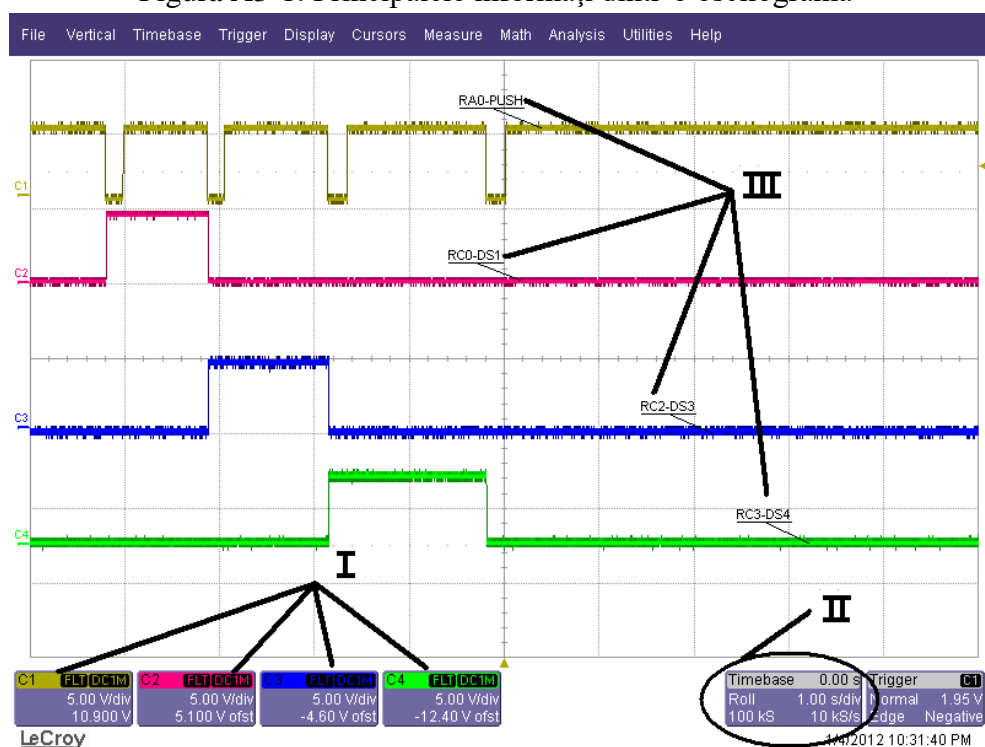
Trebuie urmați următorii pași:

- a) Selectați meniul **Programmer / Select Programmer / PICkit 2**. Se va deschide în bara de tool-uri un nou câmp, **PICkit 2 Program Toolbar** care conține 9 butoane, fiecare având câte o funcție.
- b) Apăsați primul buton din partea stângă, cel de scriere a memoriei, numit **Program the target device** . Așteptați ca scrierea să se finalizeze. În urma acestui proces, codul mașină se va regăsi în memoria Flash a microcontrolerului.
- c) Scoateți microcontrolerul din starea de reset (în care a fost introdus de programatorul PICkit 2 în timpul scrierii memoriei) prin apăsarea butonului 7 din partea stângă **Bring target MCLR to Vdd** . Din acest moment, microcontrolerul va începe execuția programul regăsit în memoria sa Flash pornind de la adresa 0x0.

Anexa 3 - Interpretarea oscilogramelor

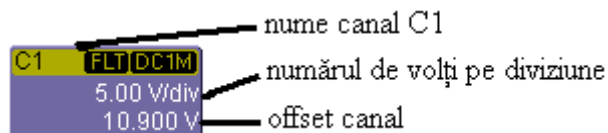
Privind următoarea oscilogramă, este firesc să ne întrebăm: Cum putem interpreta informații conținute de aceasta, și mai mult, unde regăsim aceste informații?

Figura A3-1: Principalele informații dintr-o oscilogramă



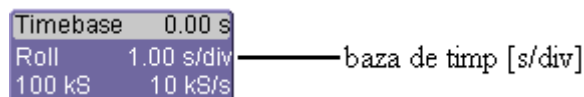
Evident că răspusul la această întrebare îl vom regăsi în imaginea de mai sus. Pentru o mai bună interpretare am notat cu I, II și III următoarele informații:

I: reprezintă cele 4 canale ale osciloscopului: C1 (Galben), C2 (Roșu), C3 (Albastru) și C4 (Verde). Acestea pot fi vizibile sau nu, în funcție de numărul de semnale ce trebuie măsurat. Pentru a calcula amplitudinea semnalului trebuie să cunoaștem numărul de volți pe diviziune (pe verticală). Această informație se regăsește în căsuța canalului respectiv:



Se poate observa ușor că semnalul de pe canalul 1, dat ca exemplu în oscilograma de mai sus, are o amplitudine de 5V (o diviziune înmulțit cu 5V/div).

II: reprezintă informațiile despre baza de timp. Cu alte cuvinte, ce perioadă de timp este cuprinsă într-o diviziune (pe orizontală).

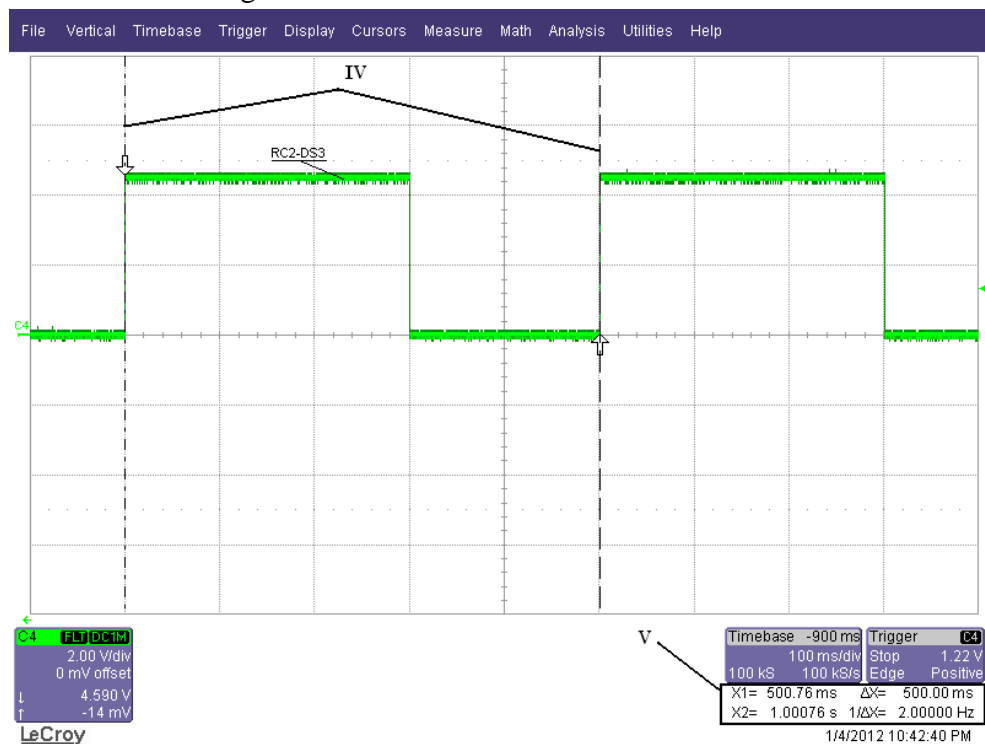


Ce ne spune informația de mai sus? Într-o diviziune aveam o secundă. Astfel putem folosi această informație pentru a calcula frecvența unui semnal.

III: reprezintă eticheta canalului. Este o opțiune a osciloscopului folosit care ne ajută să înțelegem foarte ușor ce pin (sau ce semnal) a fost aplicat pe sonda canalului x. În imaginea de mai sus putem observa că C1 reprezintă semnalul de pe pinul RA0, C2 semnalul de pe pinul RC0 și așa mai departe.

În unele cazuri, mai ales pentru a afla lățimea unui puls sau perioada exactă a unui semnal, se pot folosi cursoare verticale, care permit stabilirea cu exactitate a perioadei de timp cuprinsă între ele, după cum se poate observa în figura următoare.

Figura A3-2: Folosirea cursorilor verticale



IV: cursorile verticale.

V: informațiile despre cele două cursori, după cum urmează:

X1= 500.76 ms	ΔX= 500.00 ms	— perioada de timp cuprinsă între cursori
X2= 1.00076 s	1/ΔX= 2.00000 Hz	— frecvența unui semnal de perioadă ΔX

— poziția relativă a cursorilor față de timpul 0

Anexa 4 - Alocarea spațiului de memorie de către compilator

A4.1 Introducere

În cadrul acestui capitol adițional va fi prezentat modul în care compilatorul alocă spațiul de memorie. În programarea embedded, pe lângă cunoștințele de limbaj C, pentru a putea scrie aplicații care să folosească cât mai eficient resursele hardware ale microcontrolerului pe care rulează, e util să înțelegem cum folosește compilatorul aceste resurse.

A4.2 Tipuri de memorie

Este important să înțelegem de ce în cadrul arhitecturii unui sistem încorporat este nevoie de două tipuri de memorie: o memorie nevolatilă și o memorie volatilă. Din punct de vedere comercial folosirea unui singur tip de memorie ar fi soluția ideală, dar din motive tehnice care vor fi enunțate în continuare, acest lucru nu este posibil.

Memoria volatilă (RAM) are ca principală proprietate faptul că datele stocate în ea se pierd (se șterg) în cazul îndepărtării tensiunii de alimentare. Din acest punct de vedere, această memorie nu poate fi folosită pentru stocarea aplicației (am avea un produs care va funcționa doar până la prima întrerupere a alimentării). Din start sesizăm nevoia unui alt tip de memorie.

Memoria nevolatilă (FLASH) are ca principală proprietate faptul că datele stocate în ea nu se pierd (nu se șterg) în cazul îndepărtării tensiunii de alimentare. Din acest punct de vedere memoria FLASH este folosită pentru stocarea aplicației (aplicația unui termostat care rămâne “fără baterie” va funcționa corect după înlocuirea bateriilor). O altă proprietate importantă a unei memorii FLASH este tipul de operații pe care le suportă. O astfel de memorie poate fi scrisă și citită la nivel de byte, dar poate fi ștearsă doar la nivel de bloc.

Să încercăm următorul exercițiu de imaginație:

Presupunem că alocarea spațiului de memorie a compilatorului pentru variabilele a și b se face în memoria FLASH, la adrese consecutive, în același bloc. Avem următoarea secvență de cod:

Figura A4-1: Cod exemplu variabile

```
1 // declarare variabile
2 char a;
3 char b;
4 void main (void)
5 {
6     a = 3;
7     b = 6;
8     modifica_a( 6 );
9 }
10 void modifica_a (char input)
11 {
12     a = input;
13 }
14
```

La începutul funcției main, celor două variabile li se atribuie valorile 3 și 6. Pentru acest lucru vor avea loc două operații de scriere pe byte în memoria FLASH. În momentul apelului funcției de modificare a valorii variabilei a, noua valoare (6) nu poate fi scrisă pur și simplu peste vechea valoare (3). Procesul hardware este automat și “invizibil” programatorului, dar el constă într-o ștergere a locației de memorie și o nouă operație de scriere. Operație de ștergere poate avea loc doar la nivel de bloc, ceea ce ar duce la pierderea valorii lui b (care ar deveni 0) sau a oricărei alte variabile asignate în acel bloc.

Din acest punct de vedere memoria FLASH nu poate fi folosită pentru alocarea spațiului de memorie unei variabile, fapt ce impune existența memoriei RAM.

Memoria RAM poate fi citită, scrisă și ștersă la nivel de byte ceea ce o face perfectă pentru lucru cu variabile. De menționat că un alt avantaj față de memoria FLASH în acest context este și viteza de accesare mult mai mare în cazul memoriei RAM, ceea ce duce la creșterea performanței sistemului.

Ambele memorii sunt resurse limitate așa că folosirea lor într-un mod cât mai eficient este foarte importantă (de exemplu declararea variabilelor cu o dimensiune optimă).

Tabel A4-1: Comparație memorii FLASH și RAM

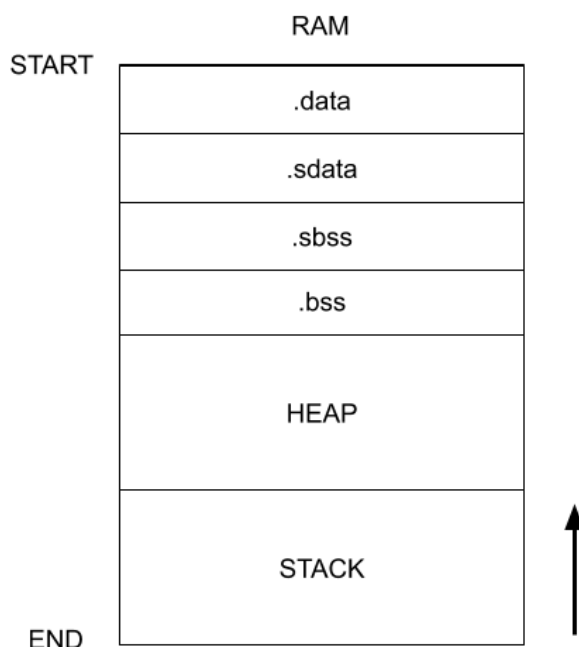
Tip memorie	Citire	Scriere	Ștergere	Destinație spațiu de memorie
RAM	byte	byte	byte	variabile
FLASH	byte	byte	bloc	aplicație

Notă: Deși anterior s-a făcut afirmația că memoria RAM nu poate fi folosită pentru stocarea aplicației, nu trebuie înțeles că din această memorie nu poate avea loc execuția aplicației. În cazul arhitecturilor performante, unde viteza de accesare a memorie Flash reprezintă un motiv de scădere a performanței, execuția aplicației poate avea loc și din memoria RAM (mult mai rapidă). Este evident că aplicația trebuie copiată mai întâi din memoria Flash în memoria RAM printr-o rutină software (această rutină executându-se din memoria Flash), apoi codul putând fi rulat din RAM.

A4.3 Alocarea spațiului de memorie de către compilator

Pentru a putea folosi cât mai eficient resursele hardware, este important să înțelegem cum și mai ales de ce compilatorul împarte spațiu de memorie RAM în diferitele secțiuni. În continuare vor fi prezentate, într-un mod general, secțiunile memoriei RAM, pentru zona de variabile, și felul în care acestea sunt folosite de către compilator.

Figura A4-2: Secțiunile memoriei RAM



Este necesar să precizăm de la început faptul că primele patru secțiuni din figura A4-2 sunt folosite de compilator pentru maparea variabilelor globale, adică variabile declarate în afara corpului unei funcții.

Prima secțiune, care de obicei are originea egală cu adresa de start a memoriei RAM, este numită `.data` (Data Segment). Această secțiune este folosită pentru alocarea de spațiu pentru variabile globale care sunt inițializate cu o valoare diferită de zero și au o dimensiune (în byte) mai mare decât parametrul n (unde n este parte din directiva de compilare `-Gn`).

Imediat după segmentul `.data` (cu aliniament corect) este mapată secțiunea pentru variabile “mici”, numită `.sdata` (Small Data Segment). În acest segment este alocat spațiu pentru variabilele globale care sunt inițializate cu valoare diferită de zero și au o dimensiune (în byte) egală sau mai mică decât parametrul n (unde n este parte din directiva de compilare `-Gn`).

În cazul în care variabilele nu sunt inițializate sau valoarea de inițializare este zero, devine clar că aceste variabile nu sunt mapate în una din cele două secțiuni prezentate anterior. Acesta este motivul pentru care există următoarele două secțiuni.

Secțiunea `.sbss` (Small Block Started by Symbol) este folosită pentru alocarea spațiului de memorie pentru variabile globale care nu sunt inițializate sau sunt inițializate cu valoarea zero și au o dimensiune (în byte) egală sau mai mică decât parametrul n (unde n este parte din directiva de compilare `-Gn`).

Ultima secțiune este numită `.bss` (Block Started by Symbol) și este folosită pentru alocarea spațiului de memorie pentru variabile globale care nu sunt inițializate sau sunt inițializate cu valoarea zero și cu o dimensiune (în byte) mai mare decât parametrul n (unde n este parte din directiva de compilare `-Gn`).

Tabel A4-2: Diferențe între secțiunile de memorie

Nume secțiune	Utilizare	Dimensiune variabilă (byte)	Tip adresare
<code>.data</code>	Variabile globale inițializate cu valoare diferită de zero	$> n$ (<code>-Gn</code>)	absolută
<code>.sdata</code>	Variabile globale inițializate cu valoare diferită de zero	$\leq n$ (<code>-Gn</code>)	relative
<code>.sbss</code>	Variabile globale inițializate cu valoarea zero	$\leq n$ (<code>-Gn</code>)	relative
<code>.bss</code>	Variabile globale inițializate cu valoarea zero	$> n$ (<code>-Gn</code>)	absolută

Folosind informațiile de mai sus, vom analiza secvența de cod din figura A4-3 pentru a verifica maparea corectă a variabilelor globale folosite. Vom

folosi și fișierul *.map generat în timpul procesului de compilare, fișier în care putem găsi secțiunile folosite de către compilator cât și elementele de cod alocate în aceste secțiuni.

Ținând cont că parametrul de compilare n este egal cu 2, putem trage următoarele concluzii, confirmate și de informațiile găsite în fișierul *.map:

- variabilele a și b neinițializate (sau inițializate implicit cu valoarea 0) vor fi mapate în secțiunea pentru variabile "mici" *.sbss*

897	.sbss	0xa000000c	0x4 build/default/production/main.o
898		0xa000000c	a
899		0xa000000e	b

- variabila c , la rândul ei neinițializată (sau inițializată implicit cu valoarea 0), va fi asignată în secțiunea pentru variabile "mari" *.bss*

1215	.bss	0xa0000174	0x4
1216	.bss	0xa0000174	0x4 build/default/production/main.o
1217		0xa0000174	c

- variabilele d și e , inițializate cu valorile 3 respectiv 5 (diferite de zero), vor fi alocate în secțiunea pentru variabile "mici" *.sdata*

874	.sdata	0xa0000000	0x4 build/default/production/main.o
875		0xa0000000	d
876		0xa0000002	e

- variabila f , inițializată cu valoarea 7 (diferită de zero), va fi alocată în secțiunea pentru variabile "mari" *.data*

1211	.data	0xa0000170	0x4
1212	.data	0xa0000170	0x4 build/default/production/main.o
1213		0xa0000170	f

Figura A4-3: Secvență de cod exemplu

```

10 char a = 0;
11 short int b = 0;
12 long int c = 0;
13
14 char d = 3;
15 short int e = 5;
16 long int f = 7;
17
18 const char parametru_1 = 4;
19 const short int parametru_2 = 20;
20 const long int parametru_3 = 12;
21 /*
22  *
23  */
24 int main(void) {
25
26     a = 7;
27     c = 8;
28     printf("Hello World.\n");
29     /* write any code*/
30
31     /* #####*/
32     return (EXIT_SUCCESS);
33 }

```

Lista cu posibilele secțiuni din memoria RAM folosită de compilator pentru alocarea de memorie pentru variabilele globale folosite în aplicație poate fi regăsită și în fișierul *.map:

Figura A4-4: Lista secțiunilor memoriei RAM

section	address	length [bytes]	(dec)	Description
.sdata	0xa0000000	0xc	12	Small init data
.sbss	0xa000000c	0x8	8	Small uninit data
.data	0xa0000014	0xd4	212	Initialized data
.bss	0xa00000e8	0x88	136	Uninitialized data

Notă 1: În exemplul de mai sus putem observa și declararea unor constante (*parametru_1*, *parametru_2* și *parametru_3*), care pot fi considerate read only. Este evident faptul că ele nu vor fi declarate în niciuna dintre secțiunile memoriei RAM (resursă limitată) din moment ce valoarea lor nu se va schimba în timpul execuției aplicației. Din acest motiv compilatorul va

aloca spațiu de memorie în secțiunea *.rodata* (read only data), parte a memoriei Flash.

1732	<i>.rodata</i>	0x9d000a24	0x18
1733	<i>.rodata</i>	0x9d000a24	0x18 build/default/production/main.o
1734		0x9d000a24	parametru_1
1735		0x9d000a26	parametru_2
1736		0x9d000a28	parametru_3

Nota 2: În funcție de compilatorul folosit, fișierul.map și secțiunile implicite pot diferi față de acest exemplu. De asemenea, este bine de știut că putem declara și alte secțiuni de memorie în afara celor implicite.

A4-4 Adresarea variabilelor

Este important de precizat de ce există secțiuni pentru variabile “mici”, mai precis secțiunea *.sdata* și *.sbss*. Fără a intra prea mult în detalii, diferența dintre variabilele mapate în cele două tipuri de secțiuni (pentru variabile “mici” sau “mari”), este dată de modul de accesare. Variabilele mapate în secțiunea *.data* sau *.bss* sunt accesate prin adresare absolută (se folosește adresa completă a variabilei) pe când variabilele din secțiunea *.sdata* și *.sbss* sunt accesate prin adresare relativă (relativ la un registru de lucru al corelului specific fiecărei arhitecturi), adică adresarea se face prin offset față de adresa de bază a secțiunii.

Avem următoarea secvență de cod și parametrul $n = 2$ (-G2):

```

24  char a = 0;
25  long int c = 0;
26
27  int main(void) {
28
29      a = 7;
30      c = 8;

```

Variabila *a* este declarată pe 8 biți (variabilă ”mică”) și inițializată cu valoarea zero, astfel că va fi alocată în secțiunea *.sbss*. Adresarea ei se va face relativ. În figură de mai jos se poate observa că accesul de scriere are loc relativ față de registrul de lucru GP (la care se adaugă un offset).

Variabila *c* este declarată pe 32 biți (variabilă ”mare”) și inițializată cu valoarea zero, astfel că va fi alocată în secțiunea *.bss*. Adresarea ei se va face în mod absolut. În următoarea figură se poate observa că accesul de scriere are loc folosindu-se adresa completă stocată în registrul de lucru V0.

Este nevoie de o instrucțiune suplimentară pentru adresarea absolute, de unde putem deduce faptul că adresarea relativă este mai rapidă și aduce beneficii în ceea ce privește timpul de execuție al aplicației.

```

5  !      a = 7;
⇒ 0x9D00031C: ADDIU V0, ZERO, 7
7  0x9D000320: SB V0, -32748(GP)
8  !      c = 8;
9  0x9D000324: LUI V0, -24576
10 0x9D000328: ADDIU V1, ZERO, 8
11 0x9D00032C: SW V1, 12(V0)

```

Există totuși și o limitare datorită faptului că offsetul este parte din opcode-ul instrucțiunii și are alocat 16 biți (din numărul total de biți pe care este codificată o instrucțiune). Astfel secțiunile *.sdata* și *.sbss* au o dimensiune maximă de 64kB, ceea ce în cazul aplicațiilor moderne se poate dovedi insuficientă, astfel că utilizarea acestor secțiuni trebuie făcută având în vedere acest neajuns.

Notă: În cazul microcontrolerului PIC16F690 (care stă la baza lucrărilor de laborator prezentate în capitol anterior), din motive ce țin de arhitectură, secțiunile *.sdata* și *.sbss* folosite pentru variabile ”mici” nu sunt implementate. Cu alte cuvinte, compilatorul nu le vede, deci nu le folosește.

A4-5 Inițializarea variabilelor

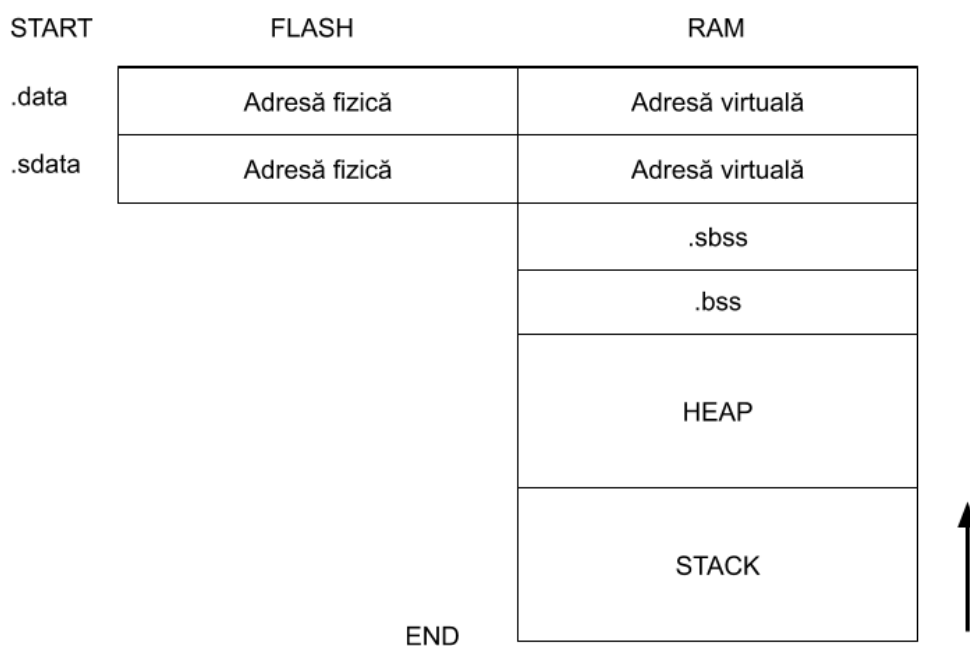
După ce am explicat motivul pentru care există secțiuni diferite pentru variabile ”mici” și ”mari”, e important să facem același lucru și în cazul secțiunilor pentru variabile inițializate sau neinițializate. Pentru simplificare, în continuare se va face referire doar la secțiunile *.data* și *.bss*, deși toate explicațiile se aplică și secțiunilor *.sdata* și *.sbss*.

Am menționat anterior faptul că variabilele sunt mapate în memoria RAM, memorie volatilă care își pierde conținutul în momentul îndepărtării tensiunii de alimentare. Cum pot totuși să aibă variabilele inițializate

valoarea dorită în momentul primei utilizări de către aplicație, sau la a doua sau a n-a rulare?

Acest lucru este posibil pentru că valoarea de inițializare a acestor variabile este stocată în memoria FLASH (nevolatilă), într-o secțiune care reflectă memoria fizică a variabilei. În timpul rulării aplicației, referința către aceste variabile se va face folosind adresa virtuală, parte a memorie RAM (secțiunea *.data*), după cum este ilustrat în următoare.

Figura A4-5: Maparea variabilelor în memoria fizică:



Valoarea de inițializare fiind salvată în memoria FLASH, aceasta nu se va pierde după îndepărtarea tensiunii de alimentare. Până la prima utilizare a valorii de către aplicație (funcția *main*) este clar că valoarea de la adresa fizică trebuie copiată în memoria virtuală. Din fericire, la crearea unui proiect, mediul de dezvoltare ține automat cont de acest aspect și va include în aplicația finală și această rutină (ca parte a rutinei de inițializare *_crt0*), care se va executa înainte de saltul în funcția *main()*, ce conține codul sursă scris de către utilizator.

Trebuie menționat că o variabilă declarată și inițializată cu o valoare diferită de zero va "ocupa" spațiu atât în memoria FLASH cât și memoria RAM. Practic cu cât vom avea mai multe variabile inițializate cu valoare diferită de zero cu atât poate crește și dimensiunea aplicației. Totodată și timpul de la momentul ieșirii din reset a microcontrolerului până la execuția primei instrucțiuni din funcția *main()* va fi influențat de rutina software care copiază valoarea variabilelor la adresele lor virtuale (memoria RAM).

În cartea [6] "Expert C Programming: Deep C Secrets, Prentice Hall 1994" Peter van der Linden, referindu-se la secțiunea *.bss*, afirmă: "Unii oameni preferă să își amintească de această secțiune ca și 'Better Save Space' (mai bine salvează spațiu). Din moment ce această secțiune conține variabile care nu au încă o valoare anume, nu e necesară stocarea niciunei valori. Dimensiunea necesară secțiunii *.bss* în timpul rulării aplicației este conținută de fișierul **.obj*, dar față de alte secțiuni nu ocupă spațiu în acest fișier."

În practică și variabilele neinițializate se consideră egale cu zero la începutul execuției funcției *main()*. Acesta este motivul pentru care variabilele neinițializate (sau inițializate cu valoarea zero) sunt mapate în *.bss*. Ar fi neproductiv să stocăm în memoria FLASH (zonă de adresă fizică) valoarea zero după care în timpul execuției să copiem această valoare în memoria RAM.

Zona de memorie asignată secțiunii *.bss* conține după reset valori considerate "gunoi" (garbage), astfel că în timpul inițializării sistemului, până la saltul în funcția *main()*, memoria RAM este scrisă cu valoarea zero (inițializată) de o rutină parte a bibliotecilor compilatorului. Astfel vom avea valoarea implicită zero în cazul variabilelor neinițializate. În cartea [7] "A new virtual memory implementation, University of California, Berkeley, 1986, McKusick & Karels", autorii menționează: "În cazul sistemelor cu OS, inițializarea secțiunii *.bss* după reset se face folosind o tehnică numită zero-fill-on-demand."

În următoarele două figuri se poate observa diferența dintre conținutul unei memorii RAM imediat după reset și după rutina de inițializare (scriere cu zero).

Figura A4-6: Memoria RAM după reset

address	0	4	8	C
ESD:40000000	????????	????????	00000000	00000005
ESD:40000010	????????	????????	????????	????????
ESD:40000020	????????	????????	????????	????????
ESD:40000030	????????	????????	????????	????????
ESD:40000040	7EBEADF1	A7D7FEC8	????????	????????
ESD:40000050	????????	????????	????????	????????
ESD:40000060	????????	????????	????????	????????
ESD:40000070	????????	????????	????????	????????

Figura A4-7: Memoria RAM după inițializare

address	0	4	8	C
ESD:40000000	00000000	00000000	00000000	00000000
ESD:40000010	00000000	00000000	00000000	00000000
ESD:40000020	00000000	00000000	00000000	00000000
ESD:40000030	00000000	00000000	00000000	00000000
ESD:40000040	00000000	00000000	00000000	00000000
ESD:40000050	00000000	00000000	00000000	00000000
ESD:40000060	00000000	00000000	00000000	00000000
ESD:40000070	00000000	00000000	00000000	00000000

A4-6 Variabile locale

Față de variabilele globale, care sunt declarate în afara corpului oricărei funcții, variabilele locale sunt acele variabile declarate (chiar și inițializate) în cadrul corpului unei funcții (vezi capitolul 2). În cazul acestor variabile, compilatorul nu va folosi pentru asignare nici una din secțiunile prezentate anterior, pentru că aceste variabile nu ”există” până la momentul primei utilizări (”există” doar pe durata execuției funcției în care este declarată). Lor nu le este atribuită o adresă fixă în una din secțiunile memoriei RAM în timpul compilării, ci se folosește stiva (mai multe în următorul subcapitol).

Dacă analizăm următoarea secvență de cod, putem concluziona că variabila *global_var* declarată în afara oricărei funcții este o variabilă globală neinițializată care va fi alocată de către compilator în secțiunea *.bss* și va avea o adresă fixă pe durata completă a rulării aplicației.

Variabila *local_var*, declarată în funcția *main()*, este o variabilă locală funcției și ”există” doar pe durata execuției funcției.

Variabila *local_var1*, declarată în funcția *fct_1()*, este o variabilă locală funcției și ”există” doar pe durata execuției funcției (de la saltul din funcția *main()* și până la execuția instrucțiunii de *return*).

Ca să se stocheze și să se manipuleze valorile acestor variabile se folosește o adresa din stivă, adresă care se eliberează după execuția funcției. Fișierul *.map nu va conține nicio informație despre variabilele locale (în momentul compilării adresa folosită nu este cunoscută).

Figura A4-8: Declaraire variabile locale

```
10  int global_var;    // variabila globala
11
12  void fct_1()
13  {
14      int local_var1; // variabila locala
15      /*
16       */
17  }
18
19  int main(void) {
20
21      int local_var; // variabila locala
22
23      global_var = 9;
24      fct_1();
25      return (EXIT_SUCCESS);
26  }
```

O variabilă locală neinițializată nu trebuie considerată ca fiind egală cu zero. Ea va avea valoarea egală cu cea stocată anterior pe stivă la aceeași adresă, valoarea considerată "garbage". Din motive de performanță, în limbajul C compilatorul nu va face inițializare cu zero a variabilelor de tip auto, deci nici a variabilelor locale. În exemplul de mai jos putem observa o execuție eronată a codului software datorită variabilei neinițializate *local_var1*. Dacă condiția din cadrul funcției *fct_1()* nu este îndeplinită, funcția nu va returna valoarea zero, ci o valoare aleatoare. Asta face ca în funcția *main()* rezultatul să fie interpretat greșit și o eroare să fie semnalată. Pentru a corecta această posibilă eroare software, variabila locală trebuie inițializată cu valoarea zero.

Figura A4-9: Exemplu software variabilă locală

```
10  int interrupt_counter;    // variabila globala
11
12  int fct_1()
13  {
14      int local_var1;    // variabila locala
15
16      if (interrupt_counter != 1)
17      {
18          local_var1 = 1;
19      }
20      /*
21       */
22      return local_var1;
23  }
24
25  int main(void) {
26
27      if(fct_1())
28      {
29          printf("ERROR");
30      }
31      else
32      {
33          printf("TEST PASSED");
34      }
35
36      return (EXIT_SUCCESS);
37  }
38
```

Notă: Trebuie menționat cazul particular în care variabila locală este declarată folosindu-se calificativul static. În acest caz variabila locală își păstrează valoarea și între apelurile funcției, deci "există" pe toata durata execuției aplicației. Acest lucru este posibil deoarece în acest caz particular compilatorul folosește o adresă fixă parte a secțiunii *.data* sau *.bss* dacă variabila este inițializată cu o valoare diferită de zero sau nu. În fișierul *.map numele variabilei nu va fi regăsit deoarece la momentul compilării variabila încă nu "există" dar are totuși adresa rezervată. În cele două exemple de mai jos se poate observa diferența între declararea variabilei locale cu calificativul static sau fără.

Figura A4-10: Variabilă locală

```
11 void fct_1()
12 {
13     int local_var = 100;    // variabila locala
14
15     printf("variabila locala egala cu %d\n", local_var);
16     local_var ++;
17 }
18
19 int main(void) {
20     __XC_UART=1;
21
22     fct_1();
23     fct_1();
24     fct_1();
25
26     return (EXIT_SUCCESS);
27 }
```

```
variabila locala egala cu 100
variabila locala egala cu 100
variabila locala egala cu 100
```

Figura A4-11: Variabilă locală statică

```
11 void fct_1()
12 {
13     static int local_var = 100;    // variabila locala
14
15     printf("variabila locala egala cu %d\n", local_var);
16     local_var ++;
17 }
18
19 int main(void) {
20     __XC_UART=1;
21
22     fct_1();
23     fct_1();
24     fct_1();
25
26     return (EXIT_SUCCESS);
27 }
```

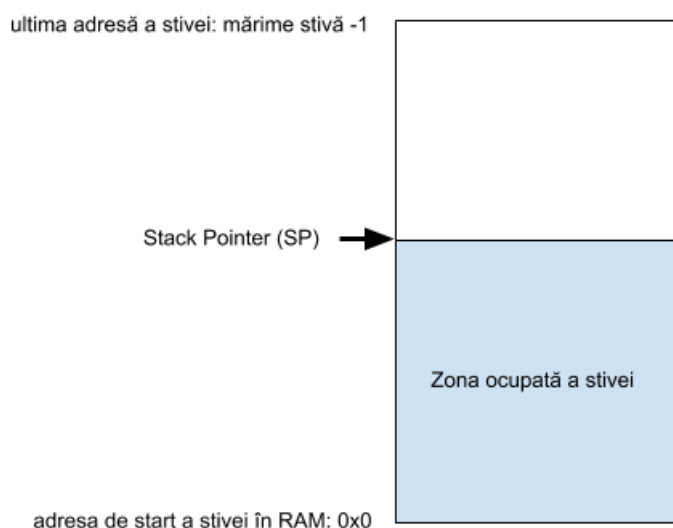
```
variabila locala egala cu 100
variabila locala egala cu 101
variabila locala egala cu 102
```

A4-7 Stack și heap

Precum secțiunile menționate anterior, Stiva (Stack) și Heap-ul sunt alte două zone de memorie create implicit, alocate de către linker în mod static în momentul build-ului. Orice proiect are configurată o dimensiune implicită pentru aceste două zone iar în aplicații mai complexe dimensionarea acestora cade în responsabilitatea proiectantului, mărimea aleasă putând avea consecințe asupra întregului sistem. Dacă aceste secțiuni sunt prea mari, se va face risipă de memorie iar dacă sunt prea mici, pot apărea erori neașteptate. Pentru a înțelege aceste fenomene, e important să știm rolul fiecărei secțiuni.

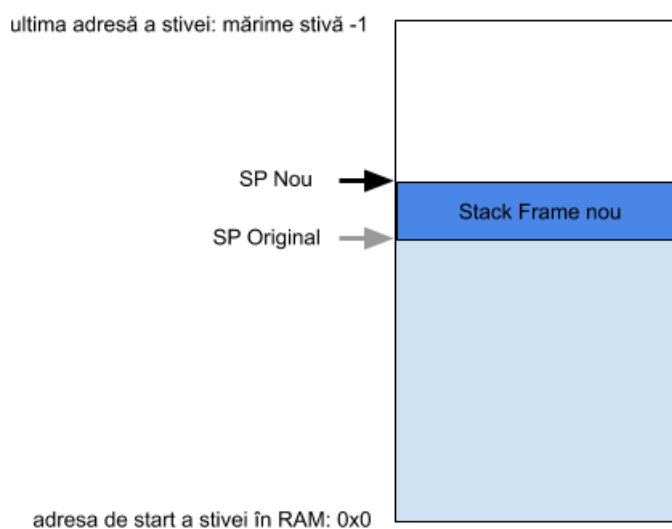
Stiva poate fi privită ca un loc de stocare temporară a datelor în timp ce sunt rulate blocuri de cod (funcții). În această zonă de memorie se vor păstra, pe lângă variabilele locale menționate anterior, și parametrii de apel ai funcțiilor, adrese de revenire, contextul în momentul întreruperilor sau regiștrii de lucru (work registers). Gestionarea stivei este de tip LIFO (Last In First Out – Ultimul Intrat Primul Ieșit), ceea ce înseamnă că datele care au fost stocate ultimele vor fi eliberate primele. Acest mecanism este realizat prin intermediul unui Stack Pointer (SP) care este, de fapt, un registru dedicat ce conține tot timpul adresa de memorie a ultimului byte folosit din stiva. În figura de mai jos se poate observa acest mecanism:

Figura A4-12: Spațiul de memorie al stivei



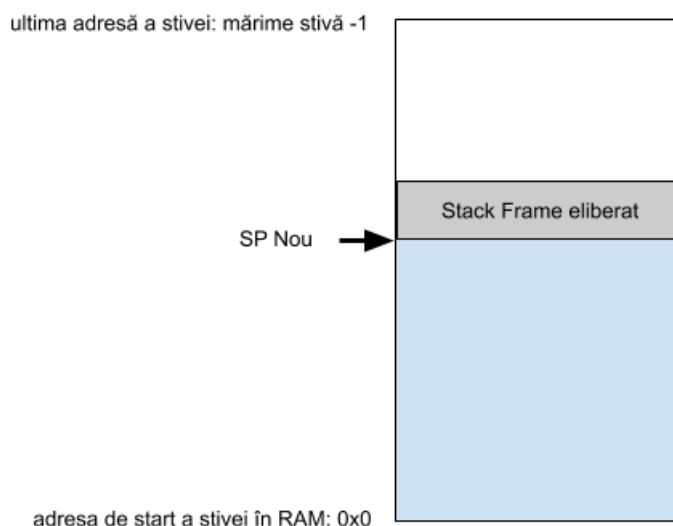
În momentul în care se apelează o nouă funcție, se va scrie un nou bloc de date pe stivă pornind de la adresa curentă din SP, urmând ca valoarea din SP să fie modificată cu dimensiunea blocului adăugat. Un astfel de bloc se numește Stack Frame iar modul de adăugare al acestuia pe stivă se poate vedea în figura următoare:

Figura A4-13: Stiva după un apel nou de funcție



După ce funcția ajunge la final prin instrucțiunea de "return" (explicită pentru funcțiile ce returnează o valoare sau implicită pentru cele ce returnează *void*), Stack Frame-ul folosit de aceasta va fi eliberat iar SP-ul va fi modificat să trimită către adresa anterioară apelului funcției. De observat că memoria eliberată nu este și ștearsă (pentru a nu pierde timp inutil), ci doar SP este mutat, datele scrise anterior rămânând nemodificate până la următoarea scriere. Datele de după SP și până la finalul stivei sunt considerate "garbage".

Figura A4-14: Eliberarea stivei la revenire din funcție



Tot acest mecanism de scriere și citire pe stivă este ”invizibil” în momentul în care scriem codul în C, dar acesta va fi realizat automat de către compilator prin inserarea de cod în limbaj de asamblare ce se va executa în momentul apelului și revenirii din fiecare funcție. Din acest motiv, un apel de funcție implică mai mult timp ”pierdut” decât ar părea la prima vedere (pe lângă branch se mai efectuează și operațiunile de copiere pe stivă).

Înțelegând astfel modul de funcționare al stivei, e ușor de observat că dacă spațiul alocat acesteia este prea mare, aplicația va avea o zonă de memorie ce nu va fi folosit vreodată. Mai periculos decât risipirea resurselor cu o stivă prea mare este însă cazul în care stiva e prea mică. În acest caz riscăm ca în momentul în care se apelează mai multe funcții în cascadă, SP-ul să ajungă să adreseze memorie în afara spațiului stivei, astfel rescriind alte date ale sistemului și generând erori greu de găsit și reprodus. Din acest motiv nu este recomandată folosirea funcțiilor recurente (care se apelează pe sine) în sistemele incorporate. Spre deosebire de aplicațiile pe PC, în sistemele incorporate nu există un mecanism implicit de verificare a SP-ul și de alertare atunci când acesta depășește zona alocată (faimoasa eroare de Stack Overflow).

Asemănător cu Stiva, Heap-ul este o zonă de memorie folosită pentru alocare dinamică a memoriei. În aplicațiile de PC, alocarea dinamică este

foarte des întâlnită datorită resurselor abundente, dar în sistemele incorporate ea este mai puțin întâlnită. În mare parte, se pot crea aplicații complexe fără a folosi acest tip de alocare, motiv pentru care nici nu vom insista prea mult asupra acestui subiect. Cum am menționat la începutul acestui subcapitol, precum dimensiunea stivei, și dimensiunea Heap-ului e stabilită înaintea build-ului, acest spațiu putând fi folosit în timpul rulării aplicație pentru a stoca temporar date sau pentru a partaja date între module software. Deoarece limbajul C nu are prevăzut un mecanism implicit pentru alocarea dinamică, putem folosi funcții dedicate din biblioteca standard `<stdlib.h>`. În cazul în care vrem să utilizăm alocarea dinamică este necesară includerea acestei biblioteci în proiect prin directiva `#include "stdlib.h"`. Pentru alocare de memorie se va folosi funcția `malloc()` iar pentru eliberarea acesteia se va folosi funcția `free()`. Este foarte important ca aceste două funcții să fie apelate mereu în tandem, astfel evitându-se scurgerile de memorie. Dacă doar alocăm memorie fără a o elibera, spațiul din heap se va termina la un moment dat, rezultând în suprascrierea zonelor de memorie din alte secțiuni și la erori mult mai greu de descoperit ca în cazul depășirii stivei. Un exemplu de folosire al alocării dinamice în heap poate fi observat în exemplul de mai jos:

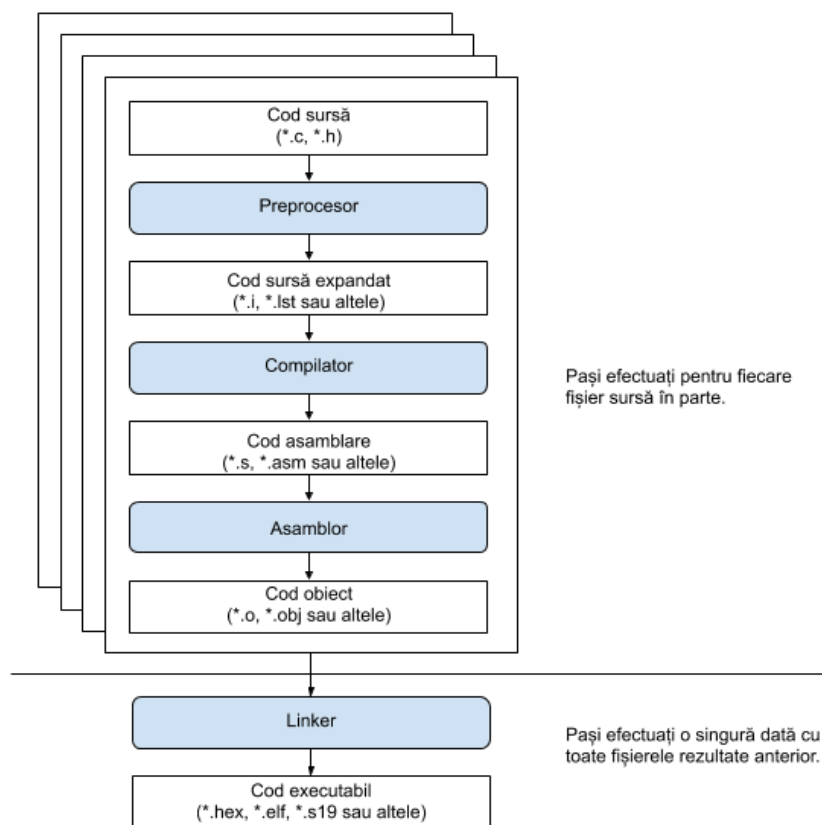
Figura A4-15: Exemplu de alocare dinamică

```
1  #include <stdlib.h>
2
3  void test_func()
4  {
5      unsigned int i;
6      unsigned char * addr_base; // pointer catre un octet
7
8      addr_base = (unsigned char *) malloc(64);
9      // se aloca o zona de 64 de octeti din heap
10     // addr_base va contine adresa de start a acestei zone
11     // functia malloc returneaza pointer catre void (void *)
12     // motiv pentru care avem nevoie de cast la pointer catre char
13
14     for(i=0; i<64; i++)
15     {
16         addr_base[i] = 100+i;
17         // deoarece addr_base este un pointer,
18         // putem sa il adresam ca pe un vector
19     }
20
21     // in acest moment, in zona din heap rezervata initial,
22     // vom avea valori consecutive de la 100 pana la 163
23
24     free( (void *) addr_base);
25     // dupa ce nu mai folosim memoria, aceasta trebuie eliberata
26     // functia free primeste ca parametru un pointer la void,
27     // motiv pentru care avem nevoie din nou de cast
28 }
29
30
```

Anexa 5 - Procesul de build

Pe parcursul acestui material am menționat cum compilăm și rulăm un program din mediu de dezvoltare dar nu am explicat cu exactitate ce se întâmplă în momentul în care facem un build. Pentru o mai bună înțelegere a sistemelor incorporate și a limbajului C, este important să cunoaștem și pașii prin care codul scris de noi ajunge să ruleze în microcontroler. În momentul în care apăsăm buton build, declanșăm o serie de acțiuni ce transformă fișierele sursă (.c și .h) într-un fișier executabil (.hex) ce poate fi programat în memoria microcontrolerului (de unde va fi rulat apoi). Acest șir de acțiuni este realizat de către o suită de programe numite adesea “build toolchain” sau “toolsuit” iar pașii unui build sunt prezentați în figura de mai jos:

Figura A5-1: Pașii procesului de build



Pașii lanțului de build prezentați în figura anterioară sunt: Preprocesorul, Compilatorul, Asamblorul și Linker-ul. Fiecare din acești pași e realizat de către un program complex (sau mai multe), create special pentru arhitectura pe care urmează să ruleze programul. Deși se pot spune foarte multe despre fiecare din aceste programe, o înțelegere de bază a funcționalității acestora e suficientă pentru a ne ajuta în munca zilnică în domeniul sistemelor incorporate.

Primul pas al lanțului de build este preprocesorul iar acesta poate fi privit, în cel mai simplu mod, ca un înlocuitor de text. Preprocesorul nu ține cont de sintaxa limbajului, de nume de funcții și variabile sau de alocări de memorie. Scopul său principal este să parcurgă toate fișierele proiectului și să înlocuiască toate directivele de preprocesare (rândurile ce încep cu “#”) cu cod C. De exemplu, o constantă numerică definită prin directiva *#define* va fi înlocuită în toate fișierele unde este folosită. Alt exemplu este înlocuirea directivei *#include* cu întreg conținutul fișierului ce se dorește a fi inclus. După înlocuirea tuturor directivei de preprocesare, fișierele vor fi salvate cu o nouă extensie și vor fi folosite în următorul pas.

Al doilea pas al procesului de build este compilarea. Fișierele rezultate după preprocesare conțin doar cod C iar în acest pas vor fi transformate din limbajul de nivel înalt în limbaj de asamblare specific platformei pe care urmează să ruleze. E important de înțeles că fiecare microcontroller (sau procesor) are un anumit set de instrucțiuni și diferă de altele din punct de vedere arhitectural. Pentru a putea avea un singur limbaj de nivel înalt în care să scriem programe pentru oricare dintre aceste platforme, avem nevoie de un program care să ”traducă” acest limbaj general în unul specific arhitecturii folosite, acest program fiind compilatorul.

În pasul de compilare se verifică corectitudinea codului scris: sintaxa, simboluri, nume definite ș.a.. Dacă nu apar erori de interpretare (compilatorul ”înțelege” toate instrucțiunile pe care le-am scris) se va crea codul în limbaj de asamblare. Din nou, deoarece codul rezultat este dependent de platforma pentru care se compilează, același cod în C, compilat pe platforme diferite, va fi diferit. Compilatorul ține cont de arhitectura microcontrollerului/procesorului pe care va rula codul și va folosi instrucțiunile și structura de regiștri ai acestuia. De exemplu, o simplă instrucțiune de citire dintr-o variabilă va fi transformată în multiple instrucțiuni de adresare și citire în/din regiștrii interni ai microcontroller-ului.

În al treilea pas al procesului de build, cel al asamblorului, codul în limbaj de asamblare rezultat din pasul anterior va fi transformat în cod obiect, specific microcontrolerului. Instrucțiunile sunt codate conform arhitecturii procesorului iar toate referințele la memorie sunt relative, motiv pentru care ansamblul de fișiere rezultate se numește și cod relocabil. Cu acest pas se încheie procesarea individuală a fișierelor sursă originale, rezultând câte un fișier cu extensia `.o` sau `.obj` (sau asemănător) pentru fiecare fișier sursă din proiect.

Ultimul pas, cel al linker-ului, are rolul de a ”împacheta” toate fișierele obiect create anterior, alături de alte biblioteci externe (dacă este cazul), pentru a forma programul executabil final. Acum sunt realizate alocările de memorie (împărțirea în zone de memorie `.text`, `.bss`, `.data` ș.a.) și referințele sunt transformate din relative în absolute. În cazul în care anumite simboluri nu pot fi găsite (nu au fost definite) sau spațiul de memorie (Flash sau RAM) nu este suficient, link-ul va returna erori specifice (diferite față de cele din pașii anteriori). Dacă nu apar erori, se va crea fișierului ”executabil” (`.hex`, `.elf`, `.s19` sau asemănător) ce va fi scris în memoria microcontrolerului pentru a fi rulat.

Procesul de build se încheie în momentul în care a fost obținut fișierul executabil, mai departe fiind necesare alte unelte pentru scrierea acestuia în memoria microcontrolerului. Cunoașterea pașilor procesului de build este utilă în practică atât pentru identificarea diverselor erori apărute ”la compilare” cât și pentru a folosi tehnici avansate de optimizare sau organizare, creând astfel aplicații cât mai performante. Un ultim aspect de notat despre procesul de build este acela referitor la denumirea uzuală folosită în practică. Deși corect este folosirea termenului de build (nu cred că cineva folosește traducerea în română construire), de foarte multe ori când spunem compilare sau eroare de compilare ne referim la întreg build-ul, nu doar la al doilea pas din proces. Folosirea aceluiași termen pentru a ne referi la două lucruri diferite este nefericită (probabil și noi fiind ”vinovați” de această practică în materialul de față) dar este un aspect ce nu poate fi schimbat, astfel că va trebui să înțelegem situația și să deducem din context dacă termenul compilare se referă la între build-ul sau doar la pasul de compilare.

Bibliografie

- [1] Ioan P. Mihu, „Dispozitive și Circuite Electronice, Volumul I,II”, ISBN 973-95604-0-4, Ed. Alma Mater, Sibiu, 2004
- [2] Ioan P. Mihu, „Procesarea Numerică a Semnalelor. Noțiuni Esențiale”, ISBN 973-632-195-1, Ed. Alma Mater, Sibiu, 2005
- [3] Ioan P. Mihu, „ANSI-C pentru microcontrolere”, Note de curs, Sibiu, 2011
- [4] Ilie Beriliu, „Microcontrolere Aplicații”, ISBN 978-973-739-578-8, Ed. Universitatea „Lucian Blaga” Sibiu, 2008
- [5] Ilie Beriliu, „Cu un PIC mai deștept”, Note curs, 2005
- [6] Peter van der Linden, „Expert C Programming: Deep C Secrets”, ISBN 978-0131774292, Ed. Prentice Hall, 1994
- [7]
- [8] PIC16F631/677/685/687/689/690 Data Sheet (DS41262E), Microchip
- [9] Low Pin Count Demo Board User’s Guide (DS51556A), Microchip
- [10] *** <http://ww1.microchip.com/downloads/en/DeviceDoc/50001686J.pdf>
- [11] *** http://ww1.microchip.com/downloads/en/DeviceDoc/MPLAB_XC8_C_Compiler_User_Guide_for_PIC.pdf
- [12] *** http://publications.gbdirect.co.uk/c_book/
- [13] *** <http://microchip.com/>