Descrierea softului de test pentru Proiect 2

M. Stanciu – 2018

AT Mega 164   20.000MHz

Cuprins

1. Necesitatea softului de test

Softul de test pus la dispoziţia studenţilor are rolurile:

- de a familiariza studentii cu programarea unui microcontroller AVR, prin exemplificarea modului de acces la registrele şi resursele hardware ale microcontrollerului
- de a ilustra aspectele specifice compilatorului/mediului de dezvoltare CodeVisionAVR: fişierele necesare proiectului, opţiunile, crearea automată a codului de iniţializare, etc
- de asemenea, încărcarea acestui soft pe o machetă permite testarea funcţionării machetei.

Funcţiile softului de test sînt:

- cînd primeste un caracter pe portul serial UART0 (la 9600bps, 8,N,1) trimite înapoi caracterul imediat urmator în codul ASCII (de exemplu, dacă primeşte “a” trimite “b”)
- cînd primeste caracterul “?” trimite numărul versiunii programului din *defs.h*
- clipeşte LED-ul atasat la portul D.6
- la apăsari repetate ale butonului conectat la portul D.5, frecvenţa cu care clipeste LEDul se modifică, pe rînd, între valorile 1Hz şi 4Hz.

Realizarea proiectului (.prj)

Ilustrăm în cele ce urmează folosirea compilatorului CVAVR versiunea 2. Se deschide compilatorul şi se încarcă acest fişier din meniul File -> Open (selecţie tip fişier PRJ). Acest fişier conţine informaţiile necesare editării şi compilării proiectului: fişierele sursă, opţiunile de compilare, tipul de procesor din cadrul familiei AVR, etc.
Pentru a vizualiza aceste opţiuni, verificaţi meniul Project->Configure->C Compiler. Acolo sînt setate toate valorile specifice machetei noastre. De exemplu, functia delay_ms calculeaza întîrzierea pe baza frecventei ceasului, deci a cuarţului folosit – dacă se schimbă cuarţul, trebuie actualizat acest parametru.
De asemenea, se poate configura ce optiuni sa suporte funcţiile printf() şi scanf(). Pentru a economisi memorie se poate exclude suportul pentru reprezentarea unor tipuri (long sau float) precum şi a notatiei de genul %3d (reprezentare pe 3 cifre) respectiv %3.2f (reprezentare cu 2 cifre dupa punctul zecimal). *Atenţie!* dacă se exclud aceste optiuni, ele nu trebuie apelate în program, căci compilatorul nu va da erori, dar de afişat nu se va afişa nimic.
Tot aici se defineste sub ce formă compilatorul va genera codul executabil. Pentru multe microcontrollerele (nu numai AVR) se foloseşte formatul INTEL HEX care este practic analogul fişierului EXE produs de un compilator pentru Windows.
Prin urmare, un proiect este format din următoarele fişiere (în Windows Explorer este recomandat să deselectaţi opţiunea de ascundere a extensiilor fişierelor, care de multe ori este setată implicit)

- .prj   fişierul de definiţie a proiectului
- .c .h   fişierele sursă C
- .hex   fişierul rezultat în urma compilării, care va trebui încărcat în uC
- .eep   fişierul de iniţializare a variabilelor stocate în EEPROM (doar dacă este cazul)

- .obj .rom .a .i .map .asm etc fişiere temporare de lucru (pot fi ignorate)

Observaţie importantă: CodeVisionAVR include şi funcţii de programare a uC-ului, astfel încît după ce se execută opţiunea Project->Build All să fie automat apelat programatorul şi să se încarce fişierul .hex rezultat în cip. În cazul nostru însă, nu vom folosi această opţiune, căci nu avem programator (dispozitiv dedicat), ci încărcăm fişierul prin serială folosind programul separat AVRBuster; mai multe detalii în documentul despre bootloader şi PC-loader. Dacă apare fereastra de programare, se va închide, şi din Project->Configure se va seta să nu se mai apeleze opţiunea "program the chip" după compilare.

Codul propriu-zis se află în fişierele .c şi .h. Pentru proiecte de complexitate mai mare de cca. 1-200 de linii este recomandata gruparea funcţiilor în fişiere dupa rolul indeplinit de acestea. Fiecare fişier este creat şi adaugat în proiect folosind meniul Project->Configure->Files->Add.

În cazul unui proiect mai mare, pentru a identifica usor în ce fişier este fiecare funcţie este recomandabilă botezarea funcţiilor avînd ca prefix numele fişierului.
De exemplu funcţia Init_initController() se vede după nume că este în fişierul init.c. Funcţiile fara prefix sînt în main.c.

In limbajul C este nevoie ca funcţiile sa aiba prototipuri (declararea funcţiei, separat de scrierea ei propriu-zisă). Varianta aleasa aici este includerea acestora în fişierul func.h (indiferent în ce fişiere sînt definite funcţiile propriu-zise); prin adaugarea acestui fişier în proiect compilatorul are la dispoziție toate prototipurile.

Este recomandabilă gruparea tututor definiţiilor în fişiere .h; în cazul nostru scopul este îndeplinit de fişierul defs.h. Tot acolo este recomandabil să se defineasca versiunea programului. De fiecare dată cînd aveţi o versiune (intermediară) funcțională şi doriți să treceți mai departe prin adăugarea sau schimbarea unor facilitati, salvati intreaga structură de fişiere în versiunea curentă, eventual într-o arhivă cu numele versiunii, şi continuati prin incrementarea numărului de versiune. In acest fel puteti reveni ulterior, mai ales dacă de la o versiune încolo apare un anumit bug.
De asemenea, este aproape obligatoriu ca programul sa poată fi interogat (pe seriala) despre numărul versiunii, aşa cum se vede în exemplu (la primirea caracterului ?). *Introduceţi această facilitate şi în programul vostru.* Dacă programul nu merge cum trebuie, primul lucru e să verificaţi numărul versiunii – este o greşeală des întîlnită să încărcaţi alt fişier .hex decît cel dorit !

2. Folosirea CodeWizardAVR

Compilatorul CodeVision pune la dispozitie acest tool extrem de util pentru scrierea automată a părţii de iniţializare a diferitelor registre[1]: timere, întreruperi, serială, watchdog, afişaj LCD etc. *Atenţie!* cînd se rulează CodeWizard, dupa ce se seteaza toţi parametrii doriţi, este recomandabila folosirea optiunii File->Program Preview, din care se vor copia iniţializările pentru perifericele dorite (*doar cele folosite şi definite de voi în acel moment*).
Dacă folosiţi opţiunea File->Generate, Save and exit, se suprascrie întregul proiect curent, implicit iniţializînd tot ce nu aţi definit cu 0.

3. Descrierea fişierelor din softul de test v.1.3

Se vor descrie linie cu linie fişierele .c şi .h din proiect.

Fişierul main.c

[1] În română: registru – registre, nu „regiștri". De asemenea, traducerea „registry" din Windows prin „regiștrii" este o dovadă de necunoaștere a limbajului tehnic.

Este fişierul principal al programului; conține functia main() şi cîteva apeluri specifice microcontrollerului, pe care implicit CodeWizard le pune în acest fişier.

Liniile 8-15
Se includ fişierele de tip header. Între <> sînt headerele de sistem, căutate în directorul de *include*-uri configurat în CodeVision, iar între “” sînt fişierele .h create de voi (din directorul cu proiectul). Nu uitati sa includeti fişierul mega164.h pentru a defini toate registrele pentru acest procesor (este recomandata vizualizarea acestui fişier prin cautarea sa în directorul de instalare a CodeVision).
Atenție ! fișierele de tip .h se include doar în acest fel, nu și în CodeVision Project -> Configure -> Input files. Acolo se selectează doar fișiere de cod .c

Liniile 17-155
Aceasta parte relativ lungă este generată de CodeWizard deci nu necesită intervenţia utilizatorului, şi nu vom intra în detalii asupra sa.
Scopul ei este definirea funcţiilor seriale ca întreruperi (dacă se bifează *Rx Interrupt* şi *Tx Interrupt* în Wizard). In acest caz, se definesc rutinele de servire (ISR – *interrupt service routine)* pentru înteruperile:

```
interrupt [USART_TXC] void usart0_tx_isr(void)
interrupt [USART_RXC] void usart0_rx_isr(void)
```

folosind cuvintul-cheie *interrupt*. In cazul în care nu se foloseste varianta pe interuperi, aceasta parte nu apare, şi se includ funcţiile din librării (care sînt mult mai putin eficiente, consumînd mult timp de aşteptare; este o particularitate a acestui compilator că se prefer a doua variantă). Faptul ca se folosesc funcţiile bazate pe întreruperi şi nu cele din librarii este “semnalat” de compilator prin definiţiile _ALTERNATE_PUTCHAR_ şi _ALTERNATE_GETCHAR_
Rezultatul este ca se pot folosi funcţiile standard de I/O din C (funcţiile de baza sînt *putchar(), getchar()* iar funcţiile derivate din acestea sînt *printf(), scanf()* şi celelalte asemenea). In plus, variabila globală *rx_counter0* contine numărul de caractere receptionate, şi poate fi interogată de programul utilizator.
In cazul unui program pe PC, aceste funcţii folosesc terminalul (ecranul şi tastatura). In cazul programului pe microcontroller, ele folosesc portul serial al acestuia. Prin folosirea pe PC a unui program de Terminal (fie cel inclus în CodeVision, fie altul cum ar fi HyperTerminal) şi conectarea microcontrollerului la portul serial al PC-ului, se obtine aproximativ acelasi rezultat: tot ce scrie programul din microcontroller apare pe ecranul terminalului, şi tot ce scrie utilizatorul este trimis catre microcontroller. Deci, e ca şi cînd placa noastră cu uC ar avea tastatură şi ecran !
Existenta acestor funcţii face mult mai usor *debugging-ul*, şi permite comunicarea uşoară între utilizator şi microcontroller, de exemplu pentru a seta parametri, a da comenzi, etc. Este posibila în acest mod şi folosirea unui *Bootloader*, un program rezident în Flash care permite incarcarea soft-ului principal fara a mai fi nevoie de un programator extern conectat la pinii de ISP. Folosind ISP se încarcă o singura data *Bootloader*-ul, dupa care softul se incarca de oricite ori direct pe seriala, prin comunicarea cu un program *Pcloader* care ruleaza pe PC şi stie sa comunice cu *Bootloader*-ul.

Dacă procesorul are 2 porturi seriale (de exempul, AT MEGA 164) atunci variabilele și functiile corespunzătoare vor avea sufixele 0 și 1. În porțiunea exemplificată nu a fost inclusă decît partea pentru UART0. Copiați din codul produs de Code Wizard initializarea UART1 și folosiți functiile *putchar1()*, *getchar1()* etc dacă doriți.

Liniile 165-168
Se defineşte rutina de servire a intreruperii timerului 1 (ISR – *Interrupt Service Routine*). Iniţializarea acestuia se face în init.c. Numărătorul timer-ului este incrementat la fiecare impuls de la ieşirea prescalerului respectiv; cînd are loc evenimentul “*output compare match A*” (valoarea din numărator ajunge la valoarea presetată de utilizator în registrul OCR1A) procesorul genereaza o intrerupere, în cazul nostru la 1 secundă, care apelează automat aceasta funcţie. In cazul nostru singura acțiune efectuată este inversarea starii LEDului – simbolul tilda (~). Definirea portului LED-ului este făcută în defs.h.

Observaţi că rutinele de întrerupere sînt funcții C definite cu o sintaxă specială:

```
interrupt [TIM1_COMPA] void nume_functie(void)
```

adică, funcţia *nume_funcţie()* va fi apelată cînd are loc întreruperea TIM1_COMPA. Găsiţi în secţiunea despre întreruperi din datasheet, respectiv din help-ul CodeVision, lista tututor întreruperilor disponibile pe fiecare uC din familia AVR.

Liniile 173-181
Începutul funcţiei main(); aici se vor pune intializarile; se face activarea intreruperilor care implicit sînt dezactivate. La sfîrsitul acestei sectiuni se intra în bucla infinită care reprezinta functionarea propriu-zisa a programului (să nu uităm că, fiind un program pentru uC, nu are sens să "se termine" precum un program pentru PC, căci nu ar avea cine să preia controlul la terminarea sa).
Nu este recomandabil ca functia main() sa fie prea mare, ci aceasta sa cheme alte funcţii.

Linia 183
Se apeleaza în buclă (folosind definiţia din defs.h) instructiunea *wdr* a procesorului, definită ca *wdogtrig()* în defs.h, care înseamnă "*watchdog reset*". *Watchdog*-ul, literalmente un "cîine de paza", este un accesoriu extrem de util unui sistem cu microprocesor; din fericire AVR are unul inclus pe cip, în caz contrar, pentru alte procesoare, fiind utilă includerea în sistem a unui cip special de watchdog.
Watchdog-ul este un system intern care resetează procesorul dupa un anumit timp (configurabil, dar de obicei nu mai mare de 2 secunde) în cazul nedorit cînd programul se blochează. Blocarea poate interveni din vina programatorului (o situatie neprevazuta care introduce programul într-o bucla din care nu poate iesi), sau din cauze externe, de exemplu un glitch pe sursa de alimentare.
Watchdogul conține un temporizator a carui expirare duce la generarea semnalului RESET pentru procesor. Logica de watchdog este realizata intr-o porțiune a procesorului care nu este afectata de program. In timpul functionarii normale, programul (programatorul) *trebuie* sa apeleze funcția de resetare a watchdogului la intervale oricît de mici, dar nu mai mari decît intervalul maxim la care a fost setat sa expire temporizatorul. Trebuie avut în vedere watchdogul în situatiile în care se asteapta input de la utilizator; nu este permisa nici în acest caz depășirea intervalului de temporizare; dacă se asteaptă de exemplu date pe termen nedefinit (cum ar fi aşteptarea ca utilizatorul sa tasteze ceva) intervalul de aşteptare trebuie partiționat în intervale de watchdog.
Concluzie: dacă watchdog-ul este activ şi nu apelaţi instrucţiunea wdogtrig() timp de 2 secunde, avînd ca efect resetare watchdogului, acesta va reseta procesorul. Efectul vizibil va fi că procesorul se va reseta la fiecare circa 2 secunde, aparent fără motiv.

Liniile 184-190
Existenta unui caracter primit pe seriala de la utilizator este detectata în variabila *rx_counter0*. Caracterul va fi citit cu *getchar()*. Se introduce valoarea citita în *temp* întrucît nu se poate apela aceasta funcţie de 2 ori (s-ar citi caractere diferite). Orice caracter cu excepția "?" întoarce caracterul n+1 folosind *putchar()*. Scrierea unui șir la terminal se face cu *printf()*.

Liniile 193-1999
Citirea tastei se face folosind o procedura speciala numita *debouncing*. (*to bounce*= a ricoşa, a sări pe rînd între mai multe pozitii). Ideea este că un contact electromecanic se poate închide şi deschide de mai multe ori în primele milisecunde corespunzatoare apăsarii şi eliberarii butonului, datorita imperfectiunii sistemului mecanic şi a defectelor microscopice de pe suprafetelor metalice care vin în contact. De aceea, în anumite situatii, este posibil ca programul sa treaca suficient de des printr-o bucla (cîteva milisecunde pot reprezenta un timp foarte lung pentru un procesor care ruleaza la cîțiva MIPS) încît sa detecteze, în mod eronat, ca utilizatorul a apăsat tasta de mai multe ori. Pentru a preveni aceasta posibilitate, în cazul detectării ca s-a apăsat tasta, se va aştepta un timp de ordiul zecilor de ms şi se va mai verifica o dată; dacă și a doua oară starea logică este aceeași, tasta se consider apăsată.

Metoda folosita în program nu este cea mai eficienta (functia *delay_ms()* este cu aşteptare în bucla, în acest timp procesorul nu mai executa alte instructiuni); se pot imagina şi alte metode de debouncing software. De notat ca la unele sisteme debouncing-ul se face prin hardware, prin adăugarea unor bistabili sau a altor circuite între tastă şi intrarea în procesor.
De asemenea, se introduce o bucla *while* suplimentară, pentru a aştepta ca userul sa elibereze tasta, similar cu acţiunea mouse-ului în windows, cînd nu apăsarea, ci eliberarea butonului este ceea ce duce la îndeplinirea acţiunii. Nu este obligatoriu să folosiţi această funcţie. Observaţi că, în aşteptarea eliberării, trebuie apelat wdogtrig(), altfel o apăsare mai lungă ar duce la resetarea uC-ului !

Liniile 203-206
La fiecare apăsare de tasta, se schimbă între 2 seturi de valori ale registrelor timerului 1, modificînd conditia care genereaza intreruperea de timer. Valorile folosite aici sînt aceleasi ca cele de la intializarea timerului în init.c (**vezi mai jos – init.c**), respectiv valorile impartite la 4 (4C40h /4 = 1310h).
Observatie: aceasta rutina este "quick and dirty" şi am lasat-o special aşa pentru a va arata varianta în care *nu trebuie scris* un program. Mai precis, se recomanda ca *niciodată* în program sa nu apara comparatii sau ințializari cu valori numerice. Toate valorile numerice se vor aloca unor constante cu nume sugestive, tipic va apre ceva de genul urmator în defs.h:
```
#define TIMER_OCR1_MSB 0x4C
```
şi apoi în program se vor folosi numai aceste constante. Folosirea valorilor numerice direct în program are numai dezavantaje:
- dacă trebuie modificata valoarea în init.c, este usor sa se "uite" sa se modifice şi în main.c; aceste bug-uri sînt greu de "prins";
- dacă sînt multe valori numerice, programul este greu de citit şi inteles, mai ales dupa ce a trecut un timp de la scrierea sa.
Excepţie: acele valori numerice de iniţializare ale registrelor (mai ales în init.c), care, dupa cum se vede, sînt foarte numeroase, şi apar *numai o dată* în program, se pot lasa ca atare.

Fişierul init.c

Cea mai mare parte a codului generat de CodeWizard este mutat manual aici, în functia de iniţializare Init_initController() care va fi apoi apelată la începutul lui main().

Liniile 24-37
Fiecare port are 2 registre asociate:
- registrul DDRx (x=A,B,C…) specifica "directia" fiecarui pin electric al portului X; sînt 8 pini corespunzatori celor 8 biţi. "1" înseamnă ieşire iar "0" intrare. Iniţializarea cu 1 sau 0 are efecte majore la nivel intern (în procesor), practic se comuta blocurile corespunzatoare funcţiilor de intrare şi iesire, întrucît procesorul permite utilizatorului să folosească orice pin de port în orice direcţie. Trebuie avut grija ca dacă un pin de port este legat la masa sau la Vcc, sa *NU* fie definit ca "output" întrucît s-ar obtine efectul unui scurt-circuit. De aceea, registrul DDRx se seteaza *numai* cu schema machetei în fata!
- registrul PORTx are semnificatie diferita în funcţie de sensul fiecarui pin:
    - dacă sensul pinului *n* este de iesire (DDRx.n=1), atunci pinul PORTx.n va genera, în logica pozitiva, nivelul de tensiune corespunzator (1=Vcc, 0=GND). Curentul maxim ce poate fi consumat din fiecare pin este dat în datasheet.
    - dacă sensul pinului *n* este de intrare (DDRx.n=0), aparent nu are sens scrierea unei valori în bitul respectiv. În acest caz însă, prin convenţie, scrierea unui "1" în PORTx.n va avea ca efect activarea unei rezistente interne de pull-up pentru acea intrare. Reamintim (de la CID) ca rezistenta de pull-up este o rezistenta de valoare mare (tipic, 100K) legata între Vcc şi pin; astfel, cînd pinul este lasat în gol, starea sa va fi precis definita ca "1". Dacă însă se aplică din exterior 0 logic, rezistenţa nu contează, datorită valorii mari. De obicei se activeaza pull-up-ul atunci cînd pinul este legat la un contact care se inchide la GND (este şi cazul acestui exemplu, în care butonul este legat între port şi masa; cînd butonul este apăsat, portul devine 0, în rest stă în 1). Implicit, PORTx.n=0 adica nu exista rezistenta de pull-up. Dacă procesorul nu ar dispune de aceasta facilitate, ar trebui adaugata manual o rezistenta externa de 100K catre Vcc.

De remarcat ca scrierea în binar a celor 8 biţi este în ordinea PORTx = $0bB_7B_6B_5B_4B_3B_2B_1B_0$
Cîteva exemple de folosire:

`DDRA = 0b11110000` iniţializează pinii PORTA.7..4 ca ieşiri şi 3..0 ca intrări

`PORTA.5 = 1` seteaza pinul 5 din portul A la valoarea 1 logic
`PORTA |= 0b00100000` are acelasi efect deoarece functia SAU între un bit şi "0" (simbolul |) nu modifica valoarea bitului

`PORTA.5 = 0` seteaza acelasi pin la valoarea 0 logic
`PORTA &= 0b11011111` are acelasi efect (se foloseste functia logica ŞI cu simbolul &)

`if(PINx && 0b00000010) ….` condiţia este îndeplinită cînd pe pinul X.1 se aplică "1" din exterior; trebuie să fie 1 căci 1 AND 1 =0, orice altă combinaţie dă 0, iar ceilalţi biţi ne-testaţi sînt 0 tocmai pentru a nu putea rezulta 1 pe poziţiile respective.

Atenţie! dacă se *scrie* o valoare în portul X, se foloseste sintaxa PORTx. Dacă se *citeste* o valoare, se foloseste sintaxa PINx. Citirea PORTx este permisa, dar are ca efect citirea valorii scrise, nu a valorii aplicate din exterior- deci compilatorul nu vă raportează eroare, dar nu obţineţi efectul dorit!

Liniile 44-89 şi 104-106
Sînt iniţializate registrele timerelor 0,1,2. Pentru detalii complete se va citi sectiunea de timere din datasheet; aceasta este destul de lunga, dar nu ne intereseaza la un moment dat toate registrele şi toate modurile în care poate opera fiecare timer, deci nu trebuie sa ne sperie numărul mare de biţi cu denumiri şi funcţii diferite ! In cazul nostru se foloseste timerul 1 în modul Output Compare A astfel:

- timerul va fi incrementat de ceasul intern al procesorului, divizat cu 1024 de catre prescalerul intern (factorul de divizare este dat de ultimii 3 biţi din registrul TCCR1B, numiti CS12-CS10);
- la atingerea valorii programate de utilizator în registrul OCR1A timerul va fi resetat la 0 (datorita activarii modului CTC – *Clear Timer on Compare match* prin bitii WGM13 şi WGM12, adica 4 şi 3, din TCCR1B). Acesta are ca efect divizarea frecvenţei ceasului cu valoarea din prescaler înmulţită cu cea din OCR1A;
- în acest moment se va genera o intrerupere (bitul OCIE1A, adica bitul 4, din TIMSK).

Registrul OCR1A este pe 16 biţi, dar se acceseaza sub forma a 2 registre pe 8 biţi OCR1AH şi OCR1AL. Reamintim că prescalerul este un tip special de registru folosit ca divizor de frecvenţă mare, de valori puţine şi fixe (tipic puteri ale lui 2). Deci, el aduce frecvenţa cuarţului la o frecvenţă mai mică, care va fi apoi divizată prin registrele timerului, de data asta cu o valoare oarecare, între 1 şi valoarea maximă pe 8 sau 16 biţi a registrului respectiv.

Modul de calcul al registrului OCR1A:
Frecvenţa ceasului de 20MHz divizată cu prescalerul de 1024 este 19531 Hz; s-a ales factorul de divizare prin prescaler maxim (1024) întrucît dorim sa aprindem LEDul cu o frecvenţa cît mai mica (1Hz) şi un timer are doar 16 biţi deci factorul maxim de divizare în timer este de 2^{16} = 65536; rezultă ca dacă nu am avea prescaler, cu acest timer nu am putea genera o frecvenţa mai mică de 20MHz/65536 = 305Hz. Observăm de asemenea ca valoarea de 19531 nu încape pe 8 biţi, deci nu s-ar putea folosi un timer pe 8b, ci doar 16 biţi (decît dacă am face divizari suplimentare în rutina de intrerupere a timerului).

In cazul nostru, divizăm frecvenţa de 19531Hz cu 19531 pentru a obtine 1Hz.
19531 (zecimal) = 0x4C41 (hexazecimal), numărăm de la 0 deci doar pînă la 0x4C40
deci, OCR1AH = 0x4C, OCRA1L = 0x40

Liniile 114-118

Se iniţializează portul serial (USART0 - *Universal Synchonous/Asynchronous Receiver Transmitter*) în modul asincron (UART = standardul RS232, existent pe PC), viteza de 9600bps, 8 biţi de date, fara bit de paritate,1 bit de stop. Notatia standard este 9600,8,N,1. Nu este necesar nici un calcul, caci CodeWizard face toate calculele pentru acest modul. Este totusi indicata citirea sectiunii despre USART din datasheet pentru a intelege cum functioneaza acest port.

Liniile 138-139
Se iniţializează *Watchdog*-ul. Sintaxa este specială (acelasi registru este scris de 2 ori, cu 2 valori diferite; aparent a 2-a instructiune anulează efectul primei, dar nu este cazul pentru acest registru). Se foloseste conceptul de *Timed Access*, adica scrierea unei succesiuni de valori intr-un registru intr-un interval de timp dat, situatie des întîlnită în cazul iniţializarii sistemelor critice, pentru a evita pornirea acestora dintr-o eroare. Dacă Watchdog-ul este pornit accidental intr-un program care nu apeleaza instructiunea *wdr*, acel program nu va mai functiona, procesorul resetîndu-se tot timpul!

Observatie: *Pentru a obtine punctajul maxim trebuie sa folositi watchdog-ul în programul vostru*. Aceasta este una din "*recommended programming practices*". Piaţa este plină de produse ieftine cum ar fi switch-uri de retea şi alte automatizari care se blocheaza (şi se deblocheaza manual printr-un simplu reset), situatie care nu ar trebui sa apara dacă s-ar folosi în mod corect un watchdog.

Fişierul defs.h

Aici se pun toate #*define*-urile din program cu caracter global. Este recomandabil sa se definească şi numărul versiunii. Toate constantele numerice se vor defini aici cu nume simbolice. Se observa ca inclusiv 1 şi 0 sînt definite ca TRUE şi FALSE.
Observatie: o parte din definiţiile de aici nu sînt folosite (ele provin dintr-un proiect mai mare). Ceea ce se defineste şi nu se foloseşte nu ocupa memorie, întrucît definiţiile sînt interpretate de pre-procesor, nu de compilator.

Fişierul funct.h

Aici se definesc prototipurile funcţiilor folosite. Dacă ar fi şi alte fişiere în afară de init.c, s-ar include şi funcţiile din acestea. Reamintim ca în limbajul C prototipurile reprezinta declaratiile funcţiilor, fără corpul acestora. Se folosesc atît pentru a întări verificarea parametrilor (dacă este apelată o funcţie cu alţi parametri decît s-a declarat în prototip, se generează o eroare), cît şi pentru a putea folosi funcţii care se cheama reciproc (în care caz, indiferent în ce ordine sînt declarate, una din funcţii va fi folosita înainte de a fi declarata, şi fara prototip se obtine o eroare de tip "funcţie nedeclarata").

4. Alte observaţii şi sugestii de scriere a programului

În general, orice valoare numerică care ar putea fi schimbată este bine să se definească măcar într-o constantă sau într-o variabilă; în ultimul caz, puteţi include funcţii de setare a variabilei respective pe serială, astfel încît să o puteţi modifica chiar în timpul rulării programului (folosiți funcția *scanf()*; atenție însă la *watchdog!* poate fi mai util să se citească caracter cu caracter folosind getchar(), și să nu se aștepte mai mult de intervalul de *watchdog*)

Dacă doriţi ca anumite variabile să nu se şteargă la dispariţia alimentării (de exemplu, ora de alarmare la un ceas) ele trebuie declarate cu cuvîntul cheie *eeprom* înainte, de exemplu *eeprom unsigned int hour* în loc de *unsigned int hour*.

5. Debugging-ul la un program embedded

În lipsa unui *debugger* extern (care este un circuit dedicat, conectat la interfaţa SPI sau JTAG a procesorului) putem folosi următoarele metode pentru *debugging*:

1) Scrierea unor valori pe portul serial

Orice variabilă care ne interesează poate fi afişată cu *printf()*, fie în buclă, fie la primirea unui caracter special pe serială, aşa cum în softul de test vedeţi că la primirea caracterului “?” se trimite numărul versiunii. Puteţi face ca, de pildă la primirea caracterului d de la *debug*, să se trimită un şir mai lung cu toate variabilele relevante.

2) Folosirea osciloscopului
Puteţi folosi osciloscopul pentru a vizualiza funcţionarea anumitor părţi ale programului. De exemplu, dacă înaintea executării unui grup de instrucţiuni se face PORTX.Y=1 şi imediat după se face PORTX.Y=0, folosind osciloscopul conectat la X.Y se va vedea un impuls pe ecran, fie pentru a verifica că evenimentul respectiv are loc, fie chiar pentru a măsura durata acelui eveniment.

În softul de test, liniile 211-215 arată aceasta. Conectați un osciloscop la pinul D.4 și, setînd Cx corespunzător, identificați cît durează un ciclu dintr-o buclă *for*, precum și durata exactă a instrucțiunii *delay_us(1)* – reamintim că dacă ceasul nu este definit corect în Project -> Configure -> C Compiler, instrucțiunile de temporizare nu vor functiona corect.

Dacă se dorește ca funcțiile de debug de mai sus să nu fie active tot timpul (de exemplu, pentru a nu apărea în programul final ci doar în variantele de test), putem include între directive #ifdef … #endif:

```
for (i=0; i<n; i++) {
     // … anumite instrucțiuni
     #ifdef _debug
          printf("i= %d",i);
     #endif
     // .. alte instructiuni
}
```

iar la începutul programului, se introduce

```
#define _debug
```

și se recompilează cu această directivă doar atunci cînd se dorește ca partea de debugging să fie activă.

Toate aceste aspecte fac ca, cel puţin în cazul acestui proiect, să recomand cu căldură realizarea practică urmată de debugging, în loc să vă consumaţi timp cu simularea în Proteus, cum au ales să facă unele echipe. Experienţa anilor trecuţi arată că un proiect care merge în Proteus este în continuare posibil să nu meargă pe placa reală, din cauza unor limitări ale simulatorului, astfel pierzîndu-se timp cu o etapă care nu era, de la început, necesară. În plus, este util să vă învăţaţi cu depanarea în domeniul *embedded*, pentru care nu există, de obicei, simulatoare.