

Descrierea softului de test pentru Proiect 2

AT Mega 164 20.000MHz

Cuprins 1. Necesitatea softului de test.....	Error! Bookmark not defined.
2. Folosirea CodeWizardAVR	2
3. Descrierea fişierelor din softul de test v.1.3	3
4. Alte observaţii şi sugestii de scriere a programului	8
5. Debugging-ul la un program embedded	8

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 primeşte un caracter pe portul serial UART0 (la 9600bps, 8,N,1) trimite înapoi caracterul imediat următor în codul ASCII (de exemplu, dacă primeşte “a” trimite “b”)
- cînd primeşte caracterul “?” trimite numărul versiunii programului din *defs.h* - clipeşte LED-ul atasat la portul D.6
- la apăsări repetate ale butonului conectat la portul D.5, frecvenţa cu care clipeşte 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, funcţia *delay_ms* calculează întârzierea pe baza frecvenţei ceasului, deci a cuarţului folosit – dacă se schimbă cuarţul, trebuie actualizat acest parametru. De asemenea, se poate configura ce opţiuni să 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 după punctul zecimal). *Atenţie!* dacă se exclud aceste opţiuni, ele nu trebuie apelate în program, căci compilatorul nu va da erori, dar de afişat nu se va afişa nimic.

Tot aici se defineşte 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 inițializare a variabilelor stocate în EEPROM (doar dacă este cazul)
- .obj .rom .a .i .map .asm etc fişiere temporare de lucru (pot fi ignorate)

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

Codul propriu-zis se află în fişierele .c și .h. Pentru proiecte de complexitate mai mare de cca. 1-200 de linii este recomandată gruparea funcțiilor în fişiere după rolul îndeplinit de acestea. Fiecare fişier este creat și adăugat în proiect folosind meniul Project->Configure->Files->Add.

În cazul unui proiect mai mare, pentru a identifica ușor în ce fişier este fiecare funcție este recomandabilă botezarea funcțiilor având ca prefix numele fişierului.

De exemplu funcția Init_initController() se vede după nume că este în fişierul init.c. Funcțiile fara prefix sînt în main.c.

În limbajul C este nevoie ca funcțiile să aibă prototipuri (declararea funcției, separat de scrierea ei propriu-zisă). Varianta aleasă aici este includerea acestora în fişierul func.h (indiferent în ce fişiere sînt definite funcțiile propriu-zise); prin adăugarea acestui fişier în proiect compilatorul are la dispoziție toate prototipurile.

Este recomandabilă gruparea tuturor definițiilor în fişiere .h; în cazul nostru scopul este îndeplinit de fişierul defs.h. Tot acolo este recomandabil să se definească versiunea programului. De fiecare dată cînd aveți o versiune (intermediară) funcțională și doriți să treceți mai departe prin adăugarea sau schimbarea unor facilitati, salvați întreaga structură de fişiere în versiunea curentă, eventual într-o arhivă cu numele versiunii, și continuați prin incrementarea numărului de versiune. În acest fel puteți reveni ulterior, mai ales dacă de la o versiune încolo apare un anumit bug.

De asemenea, este aproape obligatoriu ca programul să poată fi interogată (pe serială) despre numărul versiunii, așa cum se vede în exemplu (la primirea caracterului ?). *Introduceți această facilitate și în programul vostru.* Dacă programul nu merge cum trebuie, primul lucru e să verificați numărul versiunii – este o greșală des întîlnită să încărcăți alt fişier .hex decît cel dorit !

2. Folosirea CodeWizardAVR

Compilatorul CodeVision pune la dispoziție acest tool extrem de util pentru scrierea automată a părții de inițializare a diferitelor registre¹: timere, întreruperi, serială, watchdog, afișaj LCD etc. *Atenție!* cînd se rulează CodeWizard, după ce se setează toți parametrii doriți, este recomandabilă folosirea opțiunii File>Program Preview, din care se vor copia inițializările pentru perifericele dorite (*doar cele folosite și definite de voi în acel moment*).

Dacă folosiți opțiunea File->Generate, Save and exit, se suprascrie întregul proiect curent, implicit inițializînd tot ce nu ați definit cu 0.

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

3. Descrierea fișierelor din softul de test v.1.3

Se vor descrie linie cu linie fișierele .c și .h din proiect.

Fișierul main.c

Este fișierul principal al programului; conține funcția `main()` și câteva apeluri specifice microcontrollerului, pe care implicit CodeWizard le pune în acest fișier.

Liniile 8-15

Se includ fișierele de tip header. Între `<>` sînt headerurile de sistem, căutate în directorul de *include*-uri configurat în CodeVision, iar între `""` sînt fișierele .h create de voi (din directorul cu proiectul). Nu uitați să includeți fișierul `mega164.h` pentru a defini toate registrele pentru acest procesor (este recomandată vizualizarea acestui fișier prin căutarea 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

Această parte relativ lungă este generată de CodeWizard deci nu necesită intervenția utilizatorului, și nu vom intra în detalii asupra sa.

Scopul ei este definirea funcțiilor seriale ca întreruperi (dacă se bifează *Rx Interrupt* și *Tx Interrupt* în Wizard). În acest caz, se definesc rutinele de servire (ISR – *interrupt service routine*) pentru întreruperile:

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

folosind cuvîntul-cheie *interrupt*. În cazul în care nu se folosește varianta pe întreruperi, această parte nu apare, și se includ funcțiile din librării (care sînt mult mai puțin eficiente, consumînd mult timp de așteptare; este o particularitate a acestui compilator că se preferă a doua variantă). Faptul că se folosesc funcțiile bazate pe întreruperi și nu cele din librării este “semnalat” de compilator prin definițiile `_ALTERNATE_PUTCHAR_` și `_ALTERNATE_GETCHAR_`

Rezultatul este că se pot folosi funcțiile standard de I/O din C (funcțiile de bază sînt `putchar()`, `getchar()` iar funcțiile derivate din acestea sînt `printf()`, `scanf()` și celelalte asemenea). În plus, variabila globală `rx_counter0` conține numărul de caractere recepționate, și poate fi interogată de programul utilizator. În cazul unui program pe PC, aceste funcții folosesc terminalul (ecranul și tastatura). În 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) și conectarea microcontrollerului la portul serial al PC-ului, se obține aproximativ același rezultat: tot ce scrie programul din microcontroller apare pe ecranul terminalului, și tot ce scrie utilizatorul este trimis către microcontroller. Deci, e ca și cînd placa noastră cu uC ar avea tastatură și ecran !

Existența acestor funcții face mult mai ușor *debugging-ul*, și permite comunicarea ușoară între utilizator și microcontroller, de exemplu pentru a seta parametri, a da comenzi, etc. Este posibilă în acest mod și folosirea unui *Bootloader*, un program rezident în Flash care permite încărcarea soft-ului principal fără a mai fi nevoie de un programator extern conectat la pinii de ISP. Folosind ISP se încarcă o singură dată *Bootloader-ul*, după care softul se încarcă de oricâte ori direct pe serială, prin comunicarea cu un program *Pcloader* care rulează pe PC și știe să comunice cu *Bootloader-ul*.

Dacă procesorul are 2 porturi seriale (de exemplu, AT MEGA 164) atunci variabilele și funcțiile 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 funcțiile `putchar1()`, `getchar1()` etc dacă doriți.

Liniile 165-168

Se definește rutina de servire a intreruperii timerului 1 (ISR – *Interrupt Service Routine*). Inițializarea acestuia se face în `init.c`. Numărătorul timer-ului este incrementat la fiecare impuls de la ieșirea prescalerului respectiv; când are loc evenimentul “*output compare match A*” (valoarea din numărator ajunge la valoarea presetată de utilizator în registrul OCR1A) procesorul generează o intrerupere, în cazul nostru la 1 secundă, care apelează automat această funcție. În cazul nostru singura acțiune efectuată este inversarea stării LED-ului – simbolul tilda (~). Definirea portului LED-ului este făcută în `defs.h`.

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

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

adică, funcția `nume_functie()` va fi apelată când are loc întreruperea TIM1_COMPA. Găsiți în secțiunea despre întreruperi din datasheet, respectiv din help-ul CodeVision, lista tuturor întreruperilor disponibile pe fiecare uC din familia AVR.

Liniile 173-181

Începutul funcției `main()`; aici se vor pune inițializările; se face activarea intreruperilor care implicit sînt dezactivate. La sfîrșitul acestei secțiuni se intra în bucla infinită care reprezintă functionarea propriu-zisă 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 funcția `main()` să fie prea mare, ci aceasta să cheme alte funcții.

Linia 183

Se apelează în buclă (folosind definiția din `defs.h`) instrucțiunea `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 sistem intern care resetează procesorul după un anumit timp (configurabil, dar de obicei nu mai mare de 2 secunde) în cazul nedorit când programul se blochează. Blocarea poate interveni din vina programatorului (o situație neprevăzută care introduce programul într-o buclă din care nu poate ieși), sau din cauze externe, de exemplu un glitch pe sursa de alimentare.

Watchdogul conține un temporizator a cărui expirare duce la generarea semnalului RESET pentru procesor. Logica de watchdog este realizată într-o porțiune a procesorului care nu este afectată de program. În timpul funcționării normale, programul (programatorul) *trebuie* să apeleze funcția de resetare a watchdogului la intervale oricît de mici, dar nu mai mari decît intervalul maxim la care a fost setat să expire temporizatorul. Trebuie avut în vedere watchdogul în situațiile în care se așteaptă input de la utilizator; nu este permisă nici în acest caz depășirea intervalului de temporizare; dacă se așteaptă de exemplu date pe termen nedefinit (cum ar fi așteptarea ca utilizatorul să tasteze ceva) intervalul de așteptare trebuie partiționat în intervale de watchdog.

Concluzie: dacă watchdog-ul este activ și nu apelați instrucțiunea `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

Existența unui caracter primit pe seriala de la utilizator este detectată în variabila `rx_counter0`. Caracterul va fi citit cu `getchar()`. Se introduce valoarea citită în `temp` întrucît nu se poate apela această funcție 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 ricoșa, a sări pe rînd între mai multe pozitii). Ideea este că un contact electromecanic se poate închide și deschide de mai multe ori în primele milisecunde corespunzătoare apăsării și eliberării butonului, datorita imperfectiunii sistemului mecanic și a defectelor microscopice de pe suprafețelor 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 cîteva MIPS) încît sa detecteze, în mod eronat, ca utilizatorul a apăsât tasta de mai multe ori. Pentru a preveni aceasta posibilitate, în cazul detectării ca s-a apăsât tasta, se va aștepta un timp de ordiul zecilor de ms și 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 așteptare în bucla, în acest timp procesorul nu mai executa alte instructiuni); se pot imagina și 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 și intrarea în procesor.

De asemenea, se introduce o bucla *while* suplimentară, pentru a aștepta ca userul sa elibereze tasta, similar cu acțiunea mouse-ului în windows, cînd nu apăsarea, ci eliberarea butonului este ceea ce duce la îndeplinirea acțiunii. Nu este obligatoriu să folosiți această funcție. Observați că, în așteptarea 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 initializarea timerului în *init.c* (**vezi mai jos – *init.c***), respectiv valorile impartite la 4 ($4C40h / 4 = 1310h$).

Observatie: aceasta rutina este “quick and dirty” și am lasat-o special așa 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 inițializari 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
```

și 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 și în *main.c*; aceste buguri sînt greu de “prins”;
- dacă sînt multe valori numerice, programul este greu de citit și inteles, mai ales dupa ce a trecut un timp de la scrierea sa.

Excepție: acele valori numerice de inițializare ale registrelor (mai ales în *init.c*), care, dupa cum se vede, sînt foarte numeroase, și apar *numai o dată* în program, se pot lasa ca atare.

Fișierul *init.c*

Cea mai mare parte a codului generat de CodeWizard este mutat manual aici, în functia de inițializare *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 biți. “1” înseamnă ieșire iar “0” intrare. Inițializarea cu 1 sau 0 are efecte majore la nivel intern (în procesor), practic se comuta blocurile corespunzatoare funcțiilor de intrare și iesire, întrucît procesorul permite utilizatorului să folosească orice pin de port în orice direcție. 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 funcție 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 convenție, 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 și pin; astfel, cînd pinul este lasat în gol, starea sa va fi precis definita ca “1”. Dacă însă se aplică din exterior 0 logic, rezistența 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 și cazul acestui exemplu, în care butonul este legat între port și masa; cînd butonul este apăsăat, 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 biți este în ordinea PORTx = 0bB₇B₆B₅B₄B₃B₂B₁B₀ Cîteva exemple de folosire:

DDRA = 0b11110000 inițializează pinii PORTA.7..4 ca ieșiri și 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 și “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 ȘI cu simbolul &)

if (PINx && 0b00000010) condiția este îndeplinită cînd pe pinul X.1 se aplică “1” din exterior; trebuie să fie 1 căci 1 AND 1 =0, orice altă combinație dă 0, iar ceilalți biți ne-testați sînt 0 tocmai pentru a nu putea rezulta 1 pe pozițiile respective.

Atenție! 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 obțineți efectul dorit!

Liniile 44-89 și 104-106

Sînt inițializate 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 și toate modurile în care poate opera fiecare timer, deci nu trebuie sa ne sperie numărul mare de biți cu denumiri și funcții 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 biți 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 și WGM12, adica 4 și 3, din TCCR1B). Acesta are ca efect divizarea frecvenței ceasului cu valoarea din prescaler înmulțită cu cea din OCR1A;
- în acest moment se va genera o intrerupere (bitul OCIE1A, adica bitul 4, din TIMSK).

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

Modul de calcul al registrului OCR1A:

Frecvența ceasului de 20MHz divizată cu prescalerul de 1024 este 19531 Hz; s-a ales factorul de divizare prin prescaler maxim (1024) întrucât dorim să aprindem LEDul cu o frecvență cât mai mică (1Hz) și un timer are doar 16 biți deci factorul maxim de divizare în timer este de $2^{16} = 65536$; rezultă că dacă nu am avea prescaler, cu acest timer nu am putea genera o frecvență mai mică de $20\text{MHz}/65536 = 305\text{Hz}$. Observăm de asemenea că valoarea de 19531 nu încapă pe 8 biți, deci nu s-ar putea folosi un timer pe 8b, ci doar 16 biți (decât dacă am face divizări suplimentare în rutina de întrerupere a timerului).

În cazul nostru, divizăm frecvența de 19531Hz cu 19531 pentru a obține 1Hz.
 19531 (zecimal) = $0x4C41$ (hexazecimal), numărăm de la 0 deci doar până la $0x4C40$ deci, $\text{OCR1AH} = 0x4C$, $\text{OCR1AL} = 0x40$

Liniile 114-118

Se inițializează portul serial (USART0 - *Universal Synchronous/Asynchronous Receiver Transmitter*) în modul asincron (UART = standardul RS232, existent pe PC), viteza de 9600bps, 8 biți de date, fără bit de paritate, 1 bit de stop. Notatia standard este 9600,8,N,1. Nu este necesar nici un calcul, căci CodeWizard face toate calculele pentru acest modul. Este totuși indicată citirea secțiunii despre USART din datasheet pentru a înțelege cum funcționează acest port.

Liniile 138-139

Se inițializează Watchdog-ul. Sintaxa este specială (același registru este scris de 2 ori, cu 2 valori diferite; aparent a 2-a instrucțiune anulează efectul primei, dar nu este cazul pentru acest registru). Se folosește conceptul de *Timed Access*, adică scrierea unei succesiuni de valori într-un registru într-un interval de timp dat, situație des întâlnită în cazul inițializării sistemelor critice, pentru a evita pornirea acestora dintr-o eroare. Dacă Watchdog-ul este pornit accidental într-un program care nu apelează instrucțiunea *wdr*, acel program nu va mai funcționa, procesorul resetându-se tot timpul!

Observație: Pentru a obține punctajul maxim trebuie să folosiți watchdog-ul în programul vostru. Aceasta este una din “*recommended programming practices*”. Piața este plină de produse ieftine cum ar fi switch-uri de rețea și alte automatizări care se blochează (și se deblochează manual printr-un simplu reset), situație care nu ar trebui să apară dacă s-ar folosi în mod corect un watchdog.

Fișierul defs.h

Aici se pun toate *#define*-urile din program cu caracter global. Este recomandabil să se definească și numărul versiunii. Toate constantele numerice se vor defini aici cu nume simbolice. Se observă că inclusiv 1 și 0 sunt definite ca TRUE și FALSE.

Observație: o parte din definițiile de aici nu sunt folosite (ele provin dintr-un proiect mai mare). Ceea ce se definește și nu se folosește nu ocupă memorie, întrucât definițiile sunt interpretate de pre-procesor, nu de compilator.

Fișierul funct.h

Aici se definesc prototipurile funcțiilor folosite. Dacă ar fi și alte fișiere în afară de *init.c*, s-ar include și funcțiile din acestea. Reamintim că în limbajul C prototipurile reprezintă declarațiile funcțiilor, fără

corpul acestora. Se folosesc atît pentru a întări verificarea parametrilor (dacă este apelată o funcție cu alți parametri decît s-a declarat în prototip, se generează o eroare), cît și pentru a putea folosi funcții care se cheama reciproc (în care caz, indiferent în ce ordine sînt declarate, una din funcții va fi folosită înainte de a fi declarată, și fara prototip se obține o eroare de tip “funcție nedeclarată”).

4. Alte observații și 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, puteți include funcții de setare a variabilei respective pe serială, astfel încît să o puteți 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 doriți ca anumite variabile să nu se șteargă la dispariția alimentării (de exemplu, ora de alarmare la un ceas) ele trebuie declarate cu cuvîntul cheie *eprom* înainte, de exemplu *eprom 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 interfața 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 afișată cu *printf()*, fie în buclă, fie la primirea unui caracter special pe serială, așa cum în softul de test vedeți că la primirea caracterului “?” se trimite numărul versiunii. Puteți face ca, de pildă la primirea caracterului d de la *debug*, să se trimită un șir mai lung cu toate variabilele relevante.

2) Folosirea osciloscopului

Puteți folosi osciloscopul pentru a vizualiza funcționarea anumitor părți ale programului. De exemplu, dacă înaintea executării unui grup de instrucțiuni se face *PORTX.Y=1* și 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 aceluia 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 când se dorește ca partea de debugging să fie activă.

Toate aceste aspecte fac ca, cel puțin în cazul acestui proiect, să recomand cu căldură realizarea practică urmată de debugging, în loc să vă consumați timp cu simularea în Proteus, cum au ales să facă unele echipe. Experiența anilor trecuți 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 pierzându-se timp cu o etapă care nu era, de la început, necesară. În plus, este util să vă învățați cu depanarea în domeniul *embedded*, pentru care nu există, de obicei, simulatoare.