Prekidi

1 Prekidi

1.1 Metoda anketiranja i metoda prekida

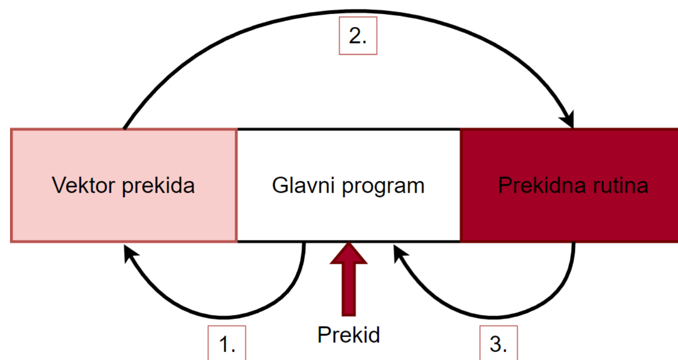
Dva pristupa obrade određenih događaja i generisanja adekvatnih odgovora na događaje jesu *metoda anketiranja (polling)* i *metoda prekida (interrupt)*.

Metoda anketiranja podrazumeva propitivanje da li se određeni događaj desio, i preduzimanje odgovarajuće akcije onda kada se događaj desi. Kako ova metoda zahteva stalnu proveru, utiče na procesorsko vreme i utrošak resursa. Dakle, ukoliko na primer u okviru određenog programa, pored niza različitih funkcija, želimo da izvršimo uključivanje LED dioda porta D kada se na pinu RB2 desi uzlazna ivica, po principu metode anketiranja, ova jednostavna provera stanja pina zahtevaće neprekidno propitivanje da li je na RB2 došlo do uzlazne ivice, a ostatak koda se neće izvršavati sve dok se željeni događaj ne desi. U ovom slučaju CPU (*Central Processing Unit*) je uposlena tako da veliki deo vremena provodi u proveravanju stanja na pinu, dok se ostale funkcionalnosti programa ne izvršavaju - trošenje procesorskog vremena. Usled navedenih nedostataka metode anketiranja, uvodi se metoda prekida.

Metoda prekida podrazumeva kontinualno izvršavanje glavnog dela programa i obaveštenje procesora o određenom događaju od interesa tek kada se on desi. Na ovaj način eliminisano je opterećivanje procesora stalnim propitivanjem da li se određeni događaj desio. U prethodno navedenom primeru, metodom prekida bilo bi omogućeno izvršavanje niza funkcija u okviru programa, do trenutka kada se na pinu RB2 ne dogodi uzlazna ivica. Kada se na pinu RB2 stanje promeni sa logičke 0 na logičku 1, izvršavanje koda se prekida, vrši se odgovarajuća akcija (uključivanje dioda na portu D), nakon koje se izvršavanje koda nastavlja. Akcija koja se izvršava kao odgovor na pojavu određenog događaja od interesa, vrši se u okviru *prekidne rutine (ISR - Interrupt Service Routine)*. Dakle, program se kontinualno izvršava, do trenutka kada se određeni događaj desi. Tada se generiše prekid i procesor se obaveštava o datom događaju. Izvršavanje glavnog programa se privremeno obustavlja, kako bi se izvršila prekidna rutina, a potom se izvršavanje glavnog programa nastavlja.

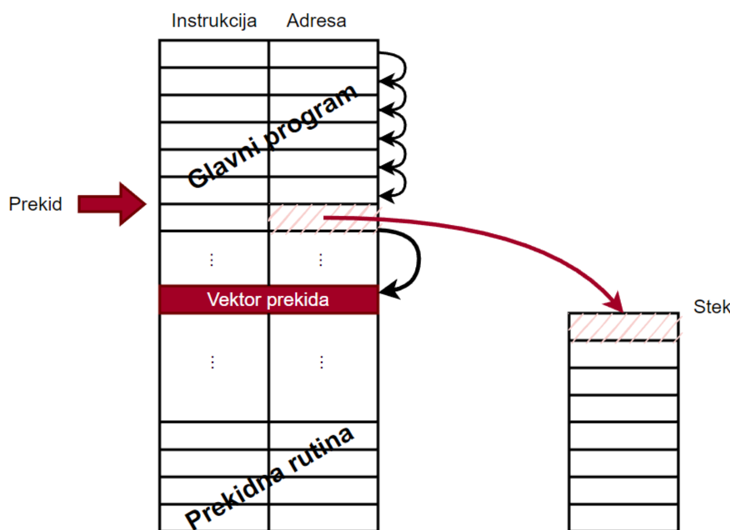
1.2 Obrada pojave prekida

Na slici 1 prikazan je dijagram izvršavanja tri osnovna koraka obrade pojave prekida. Glavni program se izvršava redom - instrukcija po instrukciju, sve do pojave prekida. Kada se generiše signal prekida, vrednost programskog brojača postaje adresa *vektora prekida (interrupt vector)* - u prvom koraku vrši se skok na adresu vektora prekida. Vektor prekida sadrži informaciju o adresi prve instrukcije prekidne rutine i njegova adresa je fiksna za određeni mikrokontroler. U drugom koraku obrade pojave prekida vrši se skok na memorijsku lokaciju prve instrukcije prekidne rutine, a zatim se prekidna rutina izvršava redom. Kada se izvrši kod u okviru prekidne rutine, u trećem koraku, vrši se skok na glavni program i nastavlja se njegovo izvršavanje.



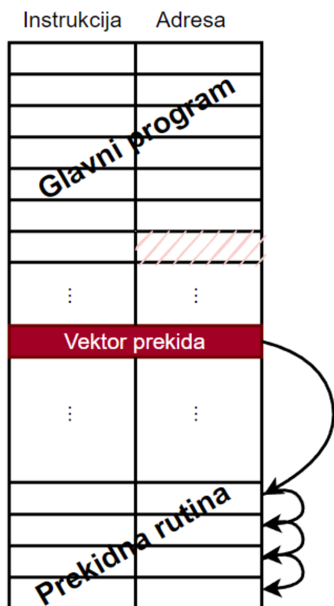
Slika 1: Obrada pojave prekida

Dakle, kada se u toku izvršavanja koda glavnog programa javi prekid i pojavi se potreba za njegovom obradom, izvršavanje glavnog koda se prekida. Kako bi se nakon obrade prekida i izvršavanja prekidne rutine izvršavanje glavnog programa nastavilo, neophodno je na neki način zapamtiti adresu instrukcije do koje se stalo pri izvršavanju glavnog programa. U te svrhe koristi se stek memorija. Kako programski brojač sadrži adresu instrukcije koja se sledeća izvršava, u trenutku pojave prekida ta vrednost se skladišti u okviru steka. Kada je adresa instrukcije glavnog koda sačuvana u stek memoriji, vrednost programskog brojača postaje adresa vektora prekida, i vrši se skok na ovu adresu. Za mikrokontroler PIC18F87K22 adresa vektora prekida je 0x0008 ili 0x0018. Opisani prostupak ilustrovan je na slici 2.



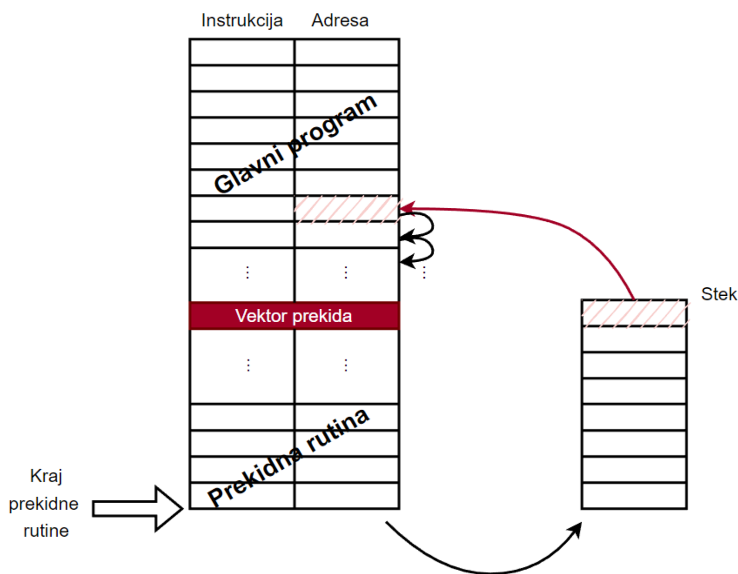
Slika 2: Čuvanje adrese prve sledeće instrukcije na steku i skok na adresu vektora prekida

U sledećem koraku vrednost programskog brojača postaje adresa prve instrukcije prekidne rutine i u memoriji se vrši skok na datu adresu. Kod prekidne rutine izvršava se instrukciju po instrukciju, dakle vrši se adekvatna akcija kao odgovor na pojavu prekida (slika 3).



Slika 3: Skok sa adrese vektora prekida na adresu prve instrukcije prekidne (ISR) rutine, i izvršavanje prekidne rutine

Kada se sa izvršavanjem koda prekidne rutine stigne do kraja, potrebno je vratiti se na izvršavanje glavnog koda, upravo na adresu koja je upisana na stek. Stek memorija radi po LIFO (*Last In First Out*) principu, što znači da je adresa instrukcije glavnog programa koja je bila naredna za izvršavanje u trenutku kada je došlo do pojave prekida, poslednja upisana na stek i prva dostupna za očitavanje. Kada vrednost programskog brojača postane data adresa, vrši se skok na nju i kod glavnog programa nastavlja sa izvršavanjem.



Slika 4: Kraj izvršavanja prekidne rutine, očitavanje adrese instrukcije glavnog koda sa steka i nastavak izvršavanja koda glavnog programa

1.3 Prekidi PIC18F87K22 mikrokontrolera

U mikrokontroleru PIC18F87K22 nalazi se veliki broj izvora prekida od kojih su neki:

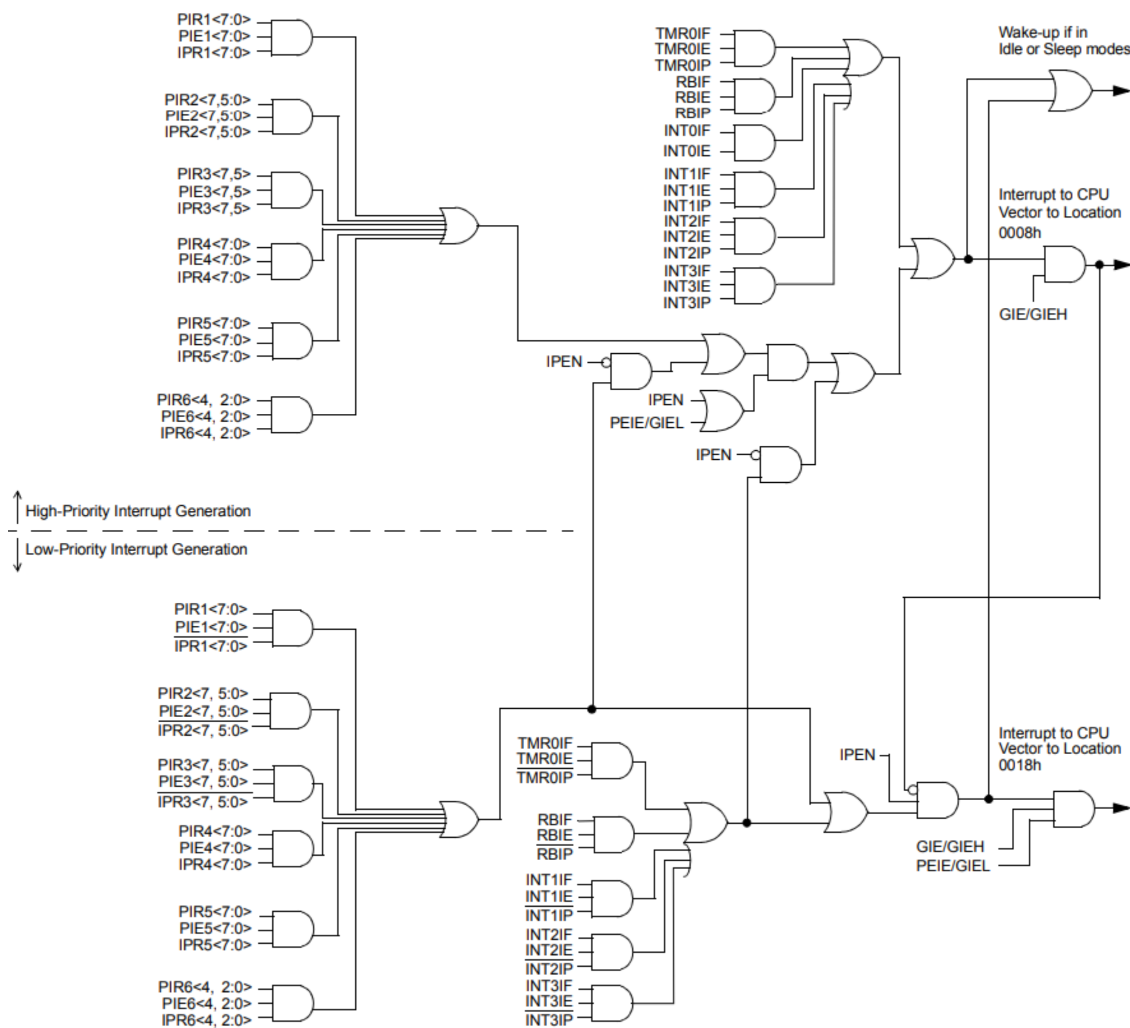
- Tajmersko-brojački - kada dođe do prekoračenja tajmersko-brojačkog registra,
- UART - Kada je neki podatak primljen ili kada je neki podatak poslat,
- Spoljašnji - Na osnovu promene signala (detekcije silazne odnosno uzlazne ivice signala), itd.

Svakom izvoru prekida u ovom mikrokontroleru dodeljuju se tri bita za kontrolu prekida i to su:

- *Enable bit* - bit koji uključuje odgovarajući interapt (IE bit)
- *Flag bit* - bit kojim se dojavljuje da je došlo do prekida (IF bit)
- *Priority bit* - bit prioriteta specificira visok ili nizak prioritet (IP bit)

Na slici 5 prikazana je struktura modula prekida PIC18F87K22 mikrokontrolera. AKo se posmatra samo *i* kolo na koje su povezani bitovi *TMR0IF, TMR0IE* i *TMR0IP* neophodno je da sva tri bita budu na stanju logičke jedinice kako bi se generisao prekid tajmera 0 (TMR0). Za ovaj slučaj to bi značilo da je prekid TMR0 uključen (*TMR0IE=1*), da je visok prioritet (*TMR0IP=1*) i da je bit *TMR0IF* na 1. *Flag bit (TMR0IF)* se automatksi postavlja na stanje jedinice, a programer ga mora softverski obristai. U slučaju da *flag bit* ostane na jedinici po izlasku iz prekidne rutine program bi ponovo dobio prekid i na taj način ostao u beskonačnoj petlji stalnim skokovima u prekidnu rutinu.

Da bi se uključili prekidi neophodno je da bit *INTCONbits.GIE/GIEH* bude na stanju logičke jedinice. Takođe, kako bi se uključili periferalni prekidi (svi prekidi osim spoljašnjih INT0, INT1, INT2, INT3, RB i prekida tajmera 0), neophodno je setovati bit *INTCONbits.PEIE/GIEL*. Periferijski prekidi imaju značaj samo ako prioriteti prekida nisu uključeni.



Slika 5: Logika sistema prekida u okviru PIC18F87K22 mikrokontrolera

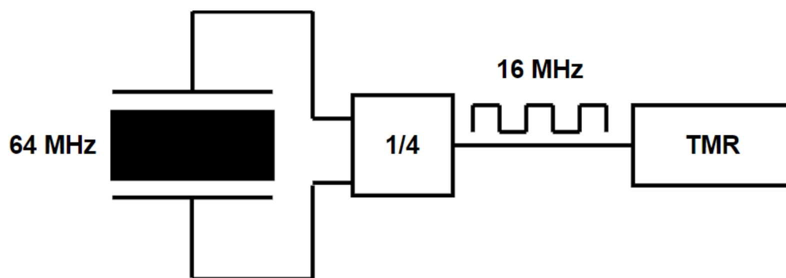
Ova serija mikrokontrolera omogućuje da se svakom od prekida dodeli i prioritet. U slučaju da se dese dva prekida u istom trenutku, prednost će imati prekidi višeg prioriteta. Uključenje ili isključenje prioriteta vrši se setovanjem ili resteovanjem bita *RCONbits.IPEN*. Uključenjem prioriteta prekidi se dele u dve celine: prekidi višeg i prekidi nižeg prioriteta. Na slici 5 u slučaju da je IP bit postavljen da je aktivan na 1, generisani prekid će dojaviti prekid i skok na vektor prekida na adresi 0x0008 (prekid visokog prioriteta). U slučaju da je IP bit aktivan na logičku 0, reč je o prekidu nižeg prioriteta i biće generisan prekid i skok na vektor prekida na adresi 0x0018 (prekid niskog prioriteta). U slučaju da se prioriteti koriste neophodno je uključiti bitove *INTCONbits.GIE/GIEH* i *INTCONbits.PEIE/GIEL* kako bi se globalno uključili prekidi i visokog i niskog prioriteta.

2 Tajmersko/brojački moduli

Tajmerski moduli mikrokontrolera mogu biti konfigurisani tako da rade u jednom od dva moda: *tajmerski mod* i *brojački mod* rada. Tajmerski mod podrazumeva merenje vremena, dok brojački mod omogućava brojanje određenih događaja.

U slučaju modula u tajmerskom režimu, merenje vremena zasniva se na brojanju taktova signala određene frekvencije. Signal takta se na tajmerske module može dovoditi sa oscilatora samog mikrokontrolera ili sa spoljašnjeg stabilnog izvora takta. Ukoliko, na primer, imamo stabilan signal takta sa oscilatora mikrokontrolera frekvencije 64 MHz, neophodno je izvršiti deljenje njegove frekvencije sa 4, kako je zapravo potrebno brojati instrukcije, a za izvršavanje jedne instrukcije, mikrokontroleru je potrebno četiri otkucaja takta. Dakle, na ulaz tajmerskog modula dovodimo povorku četvrtki frekvencije 16 MHz, odnosno, periode $\frac{1}{16}$ μs. Tajmerski modul koji je konfigurisan za rad u n-bitnom modu može da izbroji $2^n - 1$ vrednosti. Ukoliko posmatramo tajmer u 16-bitnom modu, na koji je doveden takt frekvencije 16 MHz, tajmerski modul može izbrojati maksimalno 65535 vrednosti, odnosno, može izmeriti maksimalno $65535 \cdot \frac{1}{16}$ $\mu s = 4095,94$ μs. Na slici 10 ilustrovan je prethodno opisani sistem.

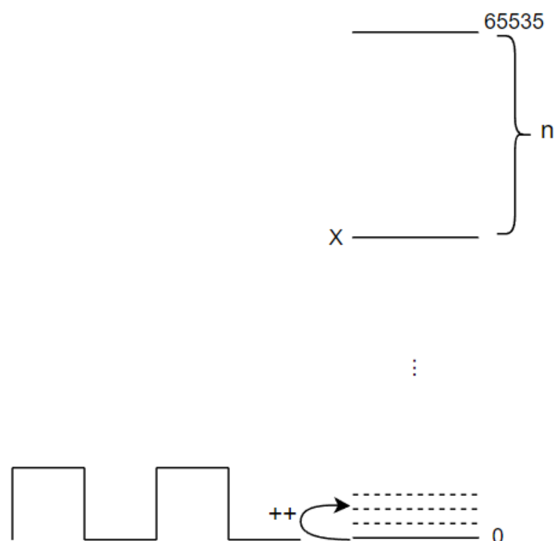
Ukoliko je pri navedenoj konfiguraciji potrebno meriti vreme veće od $4095,94$ μs, neophodna je upotreba *preskalera*. Preskalerom se omogućava deljenje frekvencije signala koji se dovodi na tajmerski modul sa vrednostima stepena dvojke. Na primer, ukoliko se prethodno navedeni signal frekvencije 16 MHz ne dovede direktno na tajmer, već na preskaler u razmeri 1:256, frekvencija signala koji će biti prosleđen tajmeru biće $\frac{16\ MHz}{256} = \frac{1}{16}\ MHz$. Perioda ovakve povorke četvrtki jeste 16 μs, što znači da je maksimalno vreme koje je sada moguće izmeriti datim tajmerskim modulom $65535 \cdot 16$ $\mu s = 1,0485$ s.



Slika 6: Princip rada tajmerskog modula

Za svaki tajmersko/brojački modul mikrokontrolera, pored SFR registara za konfiguraciju samih modula, postoje i registri koji sadrže podatke o izmerenom vremenu, odnosno, o izbrojanim događajima. Iz ovakvih registara moguće je očitavati vrednosti, ali i upisivati vrednosti u njih. U slučaju kada se tajmerskim modulom želi izmeriti vremenski interval tačno određene dužine, u date registre neophodno je upisati vrednost, takvu da brojanjem taktova od date vrednosti do maksimalne vrednosti $(2^n - 1)$, tajmerski modul izmeri zadato vreme. Kada se inkrementiranjem vrednosti registra na svaki otkucaj takta dostigne maksimalna vrednost $2^n - 1$, daljim inkrementiranjem vrednosti za 1 dešava se *overflow*, na osnovu koga je moguće generisati prekid i obavestiti CPU da je prošlo određeno vreme.

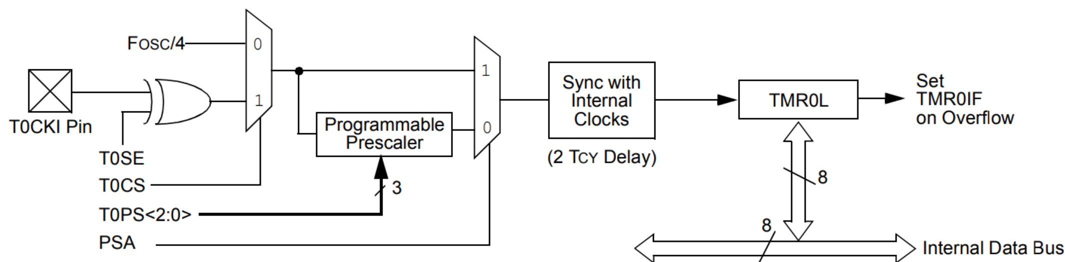
Za prethodno opisan 16-bitni tajmerski modul na čiji se ulaz dovodi signal takta frekvencije 16 MHz, maksimalna vrednost koja može biti upisana u registar jeste 65535 i ona odgovara vremenskom intervalu od oko 4,096 ms. Kako bi se ovakvim tajmerskim modulom izmerilo vreme od 1 ms potrebno je podesiti vrednost odgovarajućeg registra datog tajmera. Kako je frekvencija signala takta 16 MHz, vrednost registra inkrementira se na svakih $\frac{1}{16}\ \mu s$, dakle, u vremenskom intervalu od 1 ms, vrednost registra inkrementiraće se u $n = \frac{1\,ms}{\frac{1}{16}\,\mu s} = 16000$ koraka (slika 7). Vrednost X koju je potrebno upisati u tajmerski registar jeste ona vrednost od koje se u n inkremenata dostiže maksimalna moguća vrednost za dati registar, odnosno u datom primeru: $X = 65535 - 16000 = 49535$ (slika 7). Dakle, u 16-bitni registar biće upisana decimalna vrednost 49535, odnosno, heksadecimalna vrednost 0xC17F.



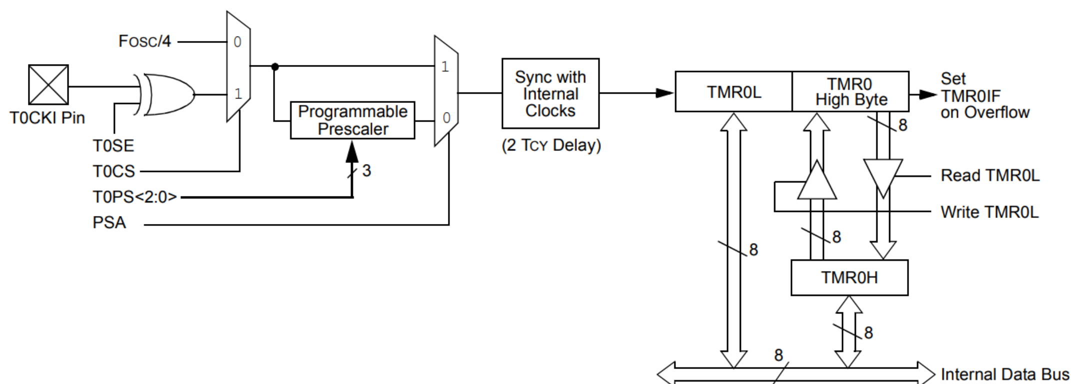
Slika 7: Određivanje vrednosti koju je neophodno uneti u tajmerski registar (X) kako bi tajmer izmerio određeno vreme (n)

2.1 TMR0 modul PIC18F87K22 mikrokontrolera

PIC18F87K22 mikrokontroler sadrži 11 tajmerskih modula od kojih 6 mogu da rade u 16-bitnom i u 8-bitnom modu dok 5 može da radi samo u 8-bitnom modu. Na slici 8 prikazan je tajmerski modul 0 koji radi u osmobitnom režimu rada, dok je na slici 9 prikazan tajmerski modul koji radi u 16-bitnom modu.



Slika 8: Šema TMR0 modula PIC18F87K22 mikrokontrolera u 8-bitnom modu rada



Slika 9: Šema TMR0 modula PIC18F87K22 mikrokontrolera u 16-bitnom modu rada

U slučaju 8-bitnog moda za smeštanje podataka koristi *TMR0L* registar dok se u slučaju rada u 16-bitnom modu koriste zajedno tajmeri *TMR0H* i *TMR0L*. Tajmerski registar broji impulse ili sa spoljašnjeg pina *T0CKI* ili merenjem frekvencije na pinu $\frac{F_{osc}}{4}$. Smer ulaznih impulsa bira se *T0CS* bitom, u slučaju da je na 0 izvor je takt instrukcija ($\frac{F_{osc}}{4}$), odnosno u slučaju da je na 1 izvor takta je *T0CKI* pin. U slučaju da je frekvencija signala tajmerskog modula previsoka moguće je preskalirati u odnosu $\frac{1}{2^n}$ počevši od 1:2 do 1:256. Ako se preskaliranje uključuje neophodno je bit *PSA* postaviti na 0, u suprotnom će preskaliranje biti isključeno.

U slučaju da korisnik želi meriti vreme od 100 ms, ako je tajmer u 16-bitnom modu rada i ako je frekvencija oscilatora 64 MHz, frekvencija instrukcije je 16 MHz. U slučaju da je preskaler isključen maksimalno vreme koje tajmer može meriti iznosi 4,09594 ms ($65535 \cdot \frac{1}{16}\mu$s). Zbog toga se vrednost preskalera mora postaviti na 1:32 pa će frekvencija instrukcija biti 500 kHz. Ovako konfigurisan tajmer može meriti do 131,07 ms ($6535 \cdot 2\mu$s). Pošto se prekoračenje tajmera događa kada se on napuni do vrha neophodno je odrediti od koje vrednosti tajmer mora brojati kako bi prošlo 100 ms. Za ovaj slučaj broj inkremenata tajmera za 100 ms će biti 50000 ($\frac{100\ ms}{2\ \mu s}$), pa će tajmer brojati od 15535 (0x3CAF). Zbog toga se nakon generisanja prekida tajmerskog modula 0 vrednost *TMR0H* treba podesiti na 0x3C, a registra *TMR0L* na 0xAF.

Kako bi se izvršila podešavanja u *MPLAB*-u neophodno je iz liste resursa u *MCC* odabrati *Timers* → TMR0. Na slci 10 je prikazan izgled prozora za podešavanja nakon dodavanja tajmeskog modula.

Slika 10: TMR0 podešavanja u *MCC* alatu

Kako bi tajmer mogao meriti 100 ms, pre svega je uključen tajmer odabirom polja *Enable Timer*, uključen je preskaler 1:32 (selektovanjem i odabirom iz padajućeg menija *Enable Prescaler*), tajmer je postavljen u 16-bitni mod rada (*Timer Mode* padajući meni), a izvor takta je oscilator, odnosno takt instrukcija (*Clock Source* padajući meni). U polje *Requested Period* uneto je vreme 100 ms. Na kraju je uključen *Enable Timer Interrupt* čime je omogućeno generisanje prekida.

Nakon generisanja koda u folderu *mcc_generated_files* nalaze se fajlovi *tmr0.h* i *tmr0.c* za podešavanje tajmera kao i *interrupt_manager.h* i *interrupt_manager.c* za kontrolu prekida. U fajlu *tmr0.c* nalazi se inicijalizaciona funkcija *TMR0_Initialize* koja postavlja tajmer u 16-bitni mod postavljanjem bita *T0CONbits.T08BIT* na 0. Zatim se u registre TMR0H i TMR0L upisuje vrednost 0x3C i 0xAF, redom, kao što je ranije izračunato. Ova 16-bitna vrednost se čuva u promenljivoj *timer0ReloadVal* kako bi se kasnije tajmer resetovao na željenu, početnu vrednost. Nakon toga se briše interapt *flag* bit resetovanjem bita *INTCONbits.TMR0IF* i uključuje se interapt setovanjem bita *INTCONbits.TMR0IE*. Kada se generiše interapt, a kako bi se korisniku omogućilo lakše rukovanje prekidom, generisani kod u *MPLAB* omogućuje korisniku da generiše svoju *callback* funkciju koja će se automatski pozvati kada se prekid desi. Sve što je neophodno je da korisnik napravi svoju funkciju i prosledi pokazivač na nju funkcijom *TMR0_SetInterruptHandler*. Ako ovo korisnik ne uradi, podrazumevana funkcija će biti *TMR0_DefaultInterruptHandler*, koja će se i koristiti u narednim primerima. Na kraju se registar *T0CON* postavlja na vrednost 0x94 čime se podešava preskaler na vrednost 1:32, potvrđuje 16-bitni mod rada tajmera, izvor takta je takt instrukcija, uključuje se preskaler i uključuje se tajmer.

Funkcija *TMR0_StartTimer* pokreće tajmer setovanjem *T0CONbits.TMR0ON* bita.

Funkcija *TMR0_StopTimer* zaustavlja tajmer resetovanjem *T0CONbits.TMR0ON*

bita.

Funkcija *TMR0_ReadTimer* vraća 16-bitni sadržaj registra *TMR0H* i *TMR0L* tj. trenutno skladištenu vrednost u tajmeru 0.

Funkcija *TMR0_WriteTimer* upisuje 16-bitnu vrednost koju prosleđuje korisnik u registre *TMR0H* i *TMR0L*.

Funkcija *TMR0_Reload* postavlja registra *TMR0H* i *TMR0L* na vrednosti specificirane podešavanjima (u ovom slučaju 0x3C i 0xAF).

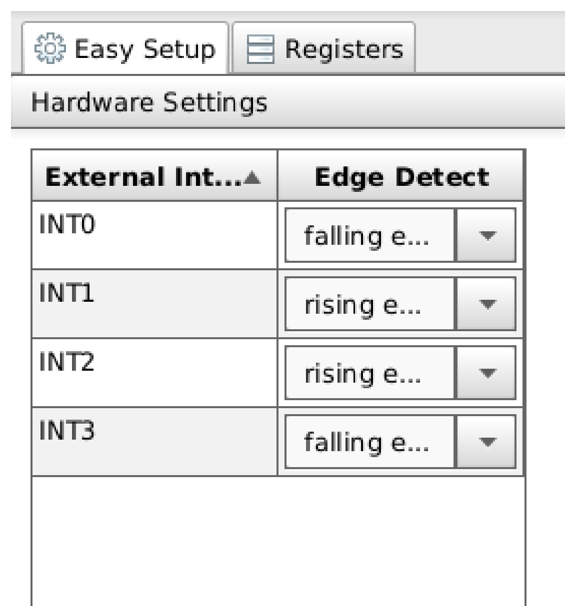
Funkcija *TMR0_ISR* će biti pozvana čim dođe do prekida i ona ima ulogu da obriše interapt *flag* resetovanjem bita *INTCONbits.TMR0IF*, postavlja tajmer na polaznu vrednost za merenje novih 100 ms i na kraju poziva *callback* funkciju. U ovom slučaju poziva se funkcija, definisana na kraju *tmr0.c* fajla, *TMR0_DefaultInterruptHandler*.

U fajlu *interrupt_manager.c* nalazi se *INTERRUPT_Initialize* funkcija koja vrši isključivanje prioriteta resetovanjem *RCONbits.IPEN* bita. Funkcija *__interrupt() INTERRUPT_InterruptManager* predstavlja prekidnu rutinu koja se automatski poziva prilikom generisanja prekida. Ona proverava ko je izvor prekida tj. da li su bitovi *INTCONbits.TMR0IE* i *INTCONbits.TMR0IF* setovani i ako jesu poziva opisanu funkciju *TMR0_ISR*.

3 Spoljašnji prekidi mikrokontrolera PIC18F87K22

Pomoću prekida moguće je detektovati promene na pinovima *INT0, INT1, INT2* i *INT3* koji su povezani na pinove *RB0, RB1, RB2* i *RB3* redom. Moguće je obezbediti detekciju uzlazne odnosno silazne ivice na svakom pinu.

Na slici 11 prikazan je prozor za podešavanje spoljašnjih prekida dobijen u *MCC*-u odabirom iz padajućeg menija *DeviceResources* → *Peripherals* → *EXT_INT*.



Slika 11: Podešavanja spoljašnjih prekida u *MCC* alatu

Odabrano je generisanje prekida *INT0* i *INT3* na silaznu ivicu dok je na pinovima *INT1* i *INT2* podešen prekid na uzlaznu ivicu. Generisani kod se slično kao i kod tajmerskog registra nalazi na putanji u folderu *mcc_generated_files*, fajlovi *ext_int.h* i *ext_int.c* za podešavanje spoljašnjeg prekida kao i *interrupt_manager.h* i *interrupt_manager.c* za kontrolu prekida.

U fajlu *ext_int.c* nalaze se funkcije za svaki od prekida, a u daljem tekstu će se koristiti oznaka *INTx* koja predstavlja generalizaciju za sve četiri mogućnosti prekida *INT0, INT1, INT2* i *INT3*. Funkcija *EXT_INT_Initialize* vrši inicijalizaciju spoljašnjih prekida gde poziva funkciju *EXT_INTx_InterruptFlagClear* koja resetuje bit *INTCONbits.INTxIF*. Zatim poziva funkciju *EXT_INTx_fallingEdgeSet* ili *EXT_INTx_risingEdgeSet* u zavisnosti da li se želi detektovati silazna ili uzlazna ivica na pinu setovanjem ili resetovanjem bita *INTEDGx* u odgovarajućem *INTCON* registru. Na kraju funkcija postavlja podrazumevane callback funkcije i ako korisnik ništa nije specificirao onda se postavlja *INTx_DefaultInterruptHandler*, za čim sledi uključivanje interapta pozivom funkcije *EXT_INTx_InterruptEnable* setovanjem odgovarajućeg *INTxIE* bita. Funkcija *INTx_ISR* se poziva kada dođe do prekida, ova funkcija briše flag pozivom odgovarajuće *EXT_INTx_InterruptFlagClear* funkcije, koja zatim poziva *INTx_CallBack*. Funkcija *INTx_CallBack* se koristi kao umotavajuća funkcija koja poziva odgovarajuću *callback* funkciju koju je definisao korisnik ili u suprotnom poziva *INTx_DefaultInterruptHandler* koja će se koristi kao podrazumevana u narednim primerima.

Kao i u slučaju tajmera *ext_int.c* fajl sadrži funkciju za isključivanje prioriteta, pošto se u ovom slučaju prioriteti ne uključuju i sadrži interapt rutinu koja proverava da li se i koji od spoljašnjih prekida desio proverom bita *INTxIE* i *INTxIF*. U slučaju da se desio neki od prekida, poziva se odgovarajuća *INTx_ISR* funkcija.

4 Primeri

Primer 1: *Realizacija štoperice korišćenjem funkcije __delay_ms*

U listingu koda 1 data je realizacija štoperice. Štoperica prikazuje desete delove sekunde, sekunde i minute. Promenljiva *des* koja predstavlja desete delove sekunde se uvećava svakih 100 ms. Važno je istaći da je realizovana pauza korišćenjem __delay_ms funkcije. Kada promenljiva *des* dostigne vrednost 9, ona se resetuje na 0 (deseta desetinka) i uvećava se promenljiva *sek* koja predstavlja sekunde. Kada promenljiva *sek* dosegne vrednost 59, umesto uvećanja neophodno je promenljivu *sek* podesiti na 0 i promenljivu *min* koja podešava minute uvećati. Sa svakom promenom promenljive *des* šalju se vrednosti preko UART-a.

Listing 1: Realizacija štoperice korišćenjem funkcije __delay_ms

```
1 #include "mcc_generated_files/mcc.h"
2
3 void main(void)
4 {
5     SYSTEM_Initialize();
6
7     uint8_t min=0,sek=0,des=0;
8     printf("min:sek:des\n");
```

```
9       while (1)
10      {
11          printf("%.2hu:%.2hu:%.2hu\n",min,sek,des);
12
13          if(des == 9)
14          {
15              des = 0;
16              if(sek == 59)
17              {
18                  sek = 0;
19                  min++;
20              }
21              else
22                  sek++;
23          }
24          else
25              des++;
26          __delay_ms(100);
27      }
28  }
```

Kada se uporedi ispis vremena u terminalu i realnog vremena na štoperici, može se uočiti odstupanje. Naime, vreme ispisano preko UART-a će proticati "sporije". Razlog leži u tome da pored pauze od 100 ms potrebno je vreme i za slanje podataka preko UART-a kao i za proveru vrednosti. Ovim unošenjem dodatnih pauza javiće se i greška koja se sa svakim pristiglim podatkom uvećava. Jedno, loše, rešenje je smanjivanje pauze kako bi se dobilo $\approx$ 100 ms po svakoj iteraciji. Ovaj način je gotovo praktično neizvodiv jer je jako teško proračunati potrebnu pauzu. Dodatni problem se ogleda u tome da sa svakom izmenom u kodu bi trebalo proračunavati novu pauzu. Drugi način je korišćenjem tajmerskog modula opisanog primerom 2.

Primer 2: *Realizacija štoperice korišćenjem prekidne rutine*

U listingu koda 2 dat je primer korišćenja prekida tajmera 0. Prekidna rutina (*Interrupt Service Routine*) *TMR0_DefaultInterruptHandler*, u fajlu *tmr0.c*, će se pozivati svakih 100 ms. U ovom slučaju prekidna rutina postavljanjem promenljive *flag* na *true* "obaveštava" glavni program da je došlo do generisanja prekida. Važno je istaći da je promenljiva *flag* definisana kao *extern volatile bool*.

Prefiks *extern* daje informaciju kompajleru da će promenljiva biti napravljena u nekom drugom fajlu (u ovom slučaju će to biti u glavnom programu), kako bi promenljiva bila "vidljiva" i u *tmr0.c* fajlu i u *main.c* fajlu.

Pomoću modifikatora *volatile* kompajleru, odnosno optimizeru, se "govori" da ne vrši optimizacije nad deklarisanom promenljivom. Pošto se vrednost promenljive *flag* modifikuje iz prekidne rutine, a pošto kompajler ne može da zna da li će se prekidna rutina izvršiti (da li će doći do prekida), kompajler zaključuje da je vrednost *flag* promenljive *false* i stoga će je proglasiti konstantom. Na ovaj način optimizer štedi RAM memoriju mikrokontrolera. Pošto u slučaju korišćenja prekidne rutine ovakav ishod nije očekivan, kako bi se kompajleru zabranile optimizacije promenljiva se deklariše kao volatile.

Glavni program listinga koda 2 za razliku od listinga koda 1, umesto funkcije *__delay_ms* čeka dok promenljiva *flag* ne postane tačna. Vrednost promenljive *flag* postane tačna kada se pozove prekidna rutina tj. nakon 100 ms. Pošto promenljiva *flag* postane tačna u glavnom programu *while(!flag)* prestaje sa izvršavanjem i

vrednost *flag* se postavlja na *false*. Pošto tajmer meri vreme nezavisno od procesora, postiže se da slanje preko UART-a i provere više ne utiču na merenje vremena. Pa su zbog toga vremena štoperice i ispisa u terminalu usaglašena.

Kako bi se uključili interapti globalno neophodno je pozvati funkciju, koja setuje GIE bit, *INTERRUPT_GlobalInterrupEnable*.

Listing 2: Realizacija štoperice korišćenjem prekidne rutine

```
1 #include "mcc_generated_files/mcc.h"
2
3 volatile bool flag = false;
4
5 void main(void)
6 {
7
8     SYSTEM_Initialize();
9
10    INTERRUPT_GlobalInterruptEnable();
11
12    uint8_t min=0,sek=0,des=0;
13    printf("min:sek:des\n");
14    while (1)
15    {
16        printf("%.2hu:%.2hu:%.2hu\n",min,sek,des);
17
18        if(des == 9)
19        {
20            des = 0;
21            if(sek == 59)
22            {
23                sek = 0;
24                min++;
25            }
26            else
27                sek++;
28        }
29        else
30            des++;
31
32        while(!flag);
33        flag = false;
34    }
35 }
36
37 /***************tmr0.h*****************/
38
39 extern volatile bool flag;
40
41 /***************tmr0.c*****************/
42
43 void TMR0_DefaultInterruptHandler(void){
44     flag = true;
45 }
```

Primer 3: *Detekcija pritiska tastera korišćenjem prekida*

U listingu koda 3 realizovan je primer detekcije uzlazne odnosno silazne ivice signala na pinu mikrokontrolera. Ulazi za ove prekide se u korišćenom PIC mikrokontroleru označavaju sa $INT0$, $INT1$, $INT2$ i $INT3$, a odgovaraju pinovima $RB0$, $RB1$, $RB2$ i $RB3$, redom. U ovom primeru detekcija interapta na pinovima *INT0* i *INT1* postavljena je na uzlaznu ivicu, dok je na pinovima *INT2* i *INT3* postavljena na silaznu ivicu. Prilikom generisanja interapta pozivaju se odgovarajuće

INTx_DefaultInterruptHandler funkcije gde x predstavlja broj od 0 do 3. Kako bi se sačuvali podaci o izvoru prekida napravljena je enumeracija koja skladišti podatke o pritisnutom tasteru i ima polje *NIJEDAN* koje ukazuje da nijedan taster nije pritisnut. U glavnom programu se nalazi mašina stanja koja proverava da li je neki od tastera pritisnut/otpušten i na osnovu toga šalje odgovarajući string preko UART-a na terminal.

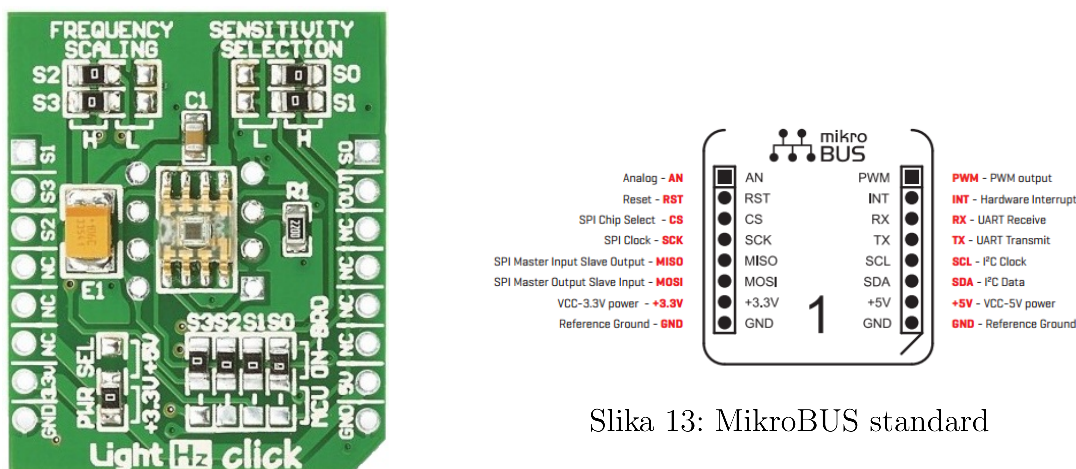
Listing 3: Detekcija pritiska tastera korišćenjem prekida

```
1 #include "mcc_generated_files/mcc.h"
2
3 volatile taster_t taster;
4
5 void main(void)
6 {
7     SYSTEM_Initialize();
8     INTERRUPT_GlobalInterruptEnable();
9     INTERRUPT_PeripheralInterruptEnable();
10
11     while (1)
12     {
13         switch(taster)
14         {
15             case NIJEDAN:
16                 break;
17             case TASTER_RB0:
18                 printf("Uzlazna ivica na tasteru RB0\n");
19                 taster = NIJEDAN;
20                 break;
21             case TASTER_RB1:
22                 printf("Uzlazna ivica na tasteru RB1\n");
23                 taster = NIJEDAN;
24                 break;
25             case TASTER_RB2:
26                 printf("Silazna ivica na tasteru RB2\n");
27                 taster = NIJEDAN;
28                 break;
29             case TASTER_RB3:
30                 printf("Silazna ivica na tasteru RB3\n");
31                 taster = NIJEDAN;
32                 break;
33             default:
34                 printf("Nedefinisano stanje\n");
35         }
36     }
37 }
38
39 /*************ext_int.h****************/
40
41 typedef enum {TASTER_RB0, TASTER_RB1, TASTER_RB2, TASTER_RB3, NIJEDAN}taster_t;
42 extern volatile taster_t taster;
43
44 /*************ext_int.c****************/
45
46 void INT0_DefaultInterruptHandler(void){
47     taster = TASTER_RB0;
48 }
49
50 void INT1_DefaultInterruptHandler(void){
51     taster = TASTER_RB1;
52 }
53
54 void INT2_DefaultInterruptHandler(void){
55     taster = TASTER_RB2;
56 }
57
```

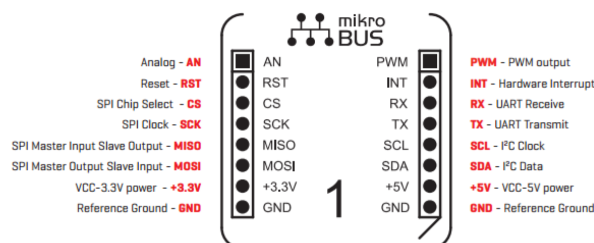
```
58 void INT3_DefaultInterruptHandler(void){
59     taster = TASTER_RB3;
60 }
```

Primer 4: *Rad sa LightHz click pločicom*

U ovom primeru koristi se senzor za merenje inteziteta osvetljenja pod nazivom *LightHz click*. Ova pločica proizvedena je od strane kompanije *Mikroelektronika* i prikazana je na slici 12. *Mikroelektronika* je razvila poseban standard za povezivanje senzorskih modula koji se naziva *MikroBUS* i omogućuje povezivanje preko 1000 različitih senzorskih modula na standardizovani konektor prikazan na slici 13. Ovaj standard objedinjuje najčešće korišćene protokole poput *I2C*, *SPI* i *UART* protokola, kao i analogne senzore i sl. U ovom primeru pločica *LightHz click* povezana je na standardni *mikroBUS* konektor 1, gde se na pinu *INT0/RB0* dobijaju impulsi generisani od strane senzorskog modula. Pomoću pinova *S0* i *S1* vrši se dodatno množenje ako se želi povećati osetljivost senzora, odnosno smanjenje frekvencije pomoću pinova *S2* i *S3* preskaliranjem.



Slika 12: LightHz click pločica



Slika 13: MikroBUS standard

U listingu koda 4 dat je primer gde se pomoću promenljive *freq* broje impulsi (uzlazne ivice signala) na pinu *INT0*. Sa svakom uzlaznom ivicom u fajlu *ext_int.c* poziva se funkcija *INT0_DefaultInterruptHandler* koja promenljivu *freq* uvećava za 1. Paralelno sa brojanjem impulsa tajmerski modul meri vreme 1 s i kada ono protekne poziva se funkcija *TMR0_DefaultInterruptHandler* koja postavlja promenljivu *flag* na *true*. U glavnom programu čeka se da prođe 1 s tj. da se promenljiva *flag* postavi na *true*. Po isteku 1 s vrednost skladištena u promenljivoj *freq* zapravo predstavlja frekvenciju. Nakon čega se vrednost šalje posredstvom UART protokola i ceo ciklus se ponavlja. Poslate vrednosti su u skladu sa standarom prikaza podatka u *Data Visualizer-u*.

Listing 4: Rad sa LightHz click pločicom

```
1 #include "mcc_generated_files/mcc.h"
2
3 volatile bool flag;
```

```
4  volatile uint32_t freq;
5
6  void main(void)
7  {
8      SYSTEM_Initialize();
9      INTERRUPT_GlobalInterruptEnable();
10
11     S0_SetLow();
12     S1_SetHigh();
13     S2_SetHigh();
14     S3_SetHigh();
15
16     while (1)
17     {
18         while(!flag);
19         flag = false;
20         EUSART1_Write(0x03);
21         EUSART1_Write(freq);
22         EUSART1_Write(freq>>8);
23         EUSART1_Write(freq>>16);
24         EUSART1_Write(freq>>24);
25         EUSART1_Write(0xFC);
26         freq = 0;
27     }
28 }
29
30 /****************tmr0.h*****************/
31
32 extern volatile bool flag;
33
34 /****************tmr0.c*****************/
35
36 void TMR0_DefaultInterruptHandler(void){
37     flag = true;
38 }
39
40 /*************ext_int.h****************/
41
42 extern volatile uint32_t freq;
43
44 /*************ext_int.c****************/
45
46 void INT0_DefaultInterruptHandler(void){
47     freq++;
48 }
```