Kako (ne) programirati STM32: Registri i minimalni toolchain

Lazar Premović - za Beoaviu, sektor za elektroniku i upravljanje 21-02-2022

Rezime

Tema izveštaja je pronalaženje minimalnog skeleta i seta alata potrebnih da bi bilo moguće izvršiti koristan program na STM32 mikrokontroleru. Rezultat je da se minimalni skelet sastoji od tri fajla sa programskim kodom, jednog skript fajla i dva dodatna alata, što je i pokazano na primeru LED indikatora koji se pali i gasi pomoću tajmera.

Pregled fajlova i alata

Kao što je već rečeno, minimalno radno okruženje se sastoji od 4 fajla i dva alata a to su:

- `main.c`: Fajl koji sadrži glavni programski kod u jeziku C.
- `crt.s`: Fajl koji sadrži kod za inicijalizaciju procesora pisan u asembleru.
- `linker.ld`: Fajl koji sadrži instrukcije za povezivač (linker)
- `build.bat`: Fajl koji sadrži skriptu za prevođenje i programiranje mikrokontrolera.
- `GNU Arm Embedded Toolchain`: Prevodilac jezika C za procesore bazirane na ARM arhitekturi.
- `STM32 ST-LINK Utility`: Aplikacija za programiranje STM32 mikrokontrolera korišćenjem ST-LINK programatora.

main.c

Datoteka main.c sadrži glavni programski kod i ovde će biti ukratko objašnjena uloga bitnijih celina koda kao i neko neophodno predznanje za razumevanje navedenog koda.

Periferije, registri, tajmeri i prekidi

STM32F103C8T6 mikrokontroler se sastoji od ARM Cortex M3 jezgra i periferija. Periferije koje su nam trenutno od značaja su GPIO[1] koji nam omogućava da palimo i gasimo LED indikator, tajmer za merenje vremena i NVIC[2] kako bismo mogli da koristimo prekide.

Svaka periferija ima određeni broj registara koji se koriste za njenu konfiguraciju i kasniju komunikaciju sa njom. Na STM32 mikrokontrolerima sve periferije (uključujući i one koje su deo samog jezgra kao što je NVIC) su memorijski mapirane tako da se registrima periferija može pristupiti prostom dodelom vrednosti pokazivaču na odgovarajuću adresu.

Tajmeri na STM32 mikrokontrolerima su jako kompleksni i ovde će ukratko biti reci samo o nekim osnovnim delovima koji su bili potrebni za realizaciju ovog programa. U ovom programu tajmer je podešen da izazove prekid svaki put kada brojački registar dostigne odgovarajuću vrednost, ta vrednost je 9999[3] što nam zajedno sa vrednošću preskalera[4] od 799 daje period tajmera od jedne sekunde[5].

Prekidi nam omogućavaju da napišemo deo koda koji će automatski biti izvršen od strane jezgra kada se određeni uslov ispuni. U ovom programu prekidi se koriste da se svaki put kada tajmer izazove prekid promeni stanje LED indikatora.

[1] General-purpose input/output
[2] Nested Vectored Interrupt Controller
[3] Kako se brojački registar resetuje na 0 ova vrednost nam zapravo daje 10000 vremenskih jedinica između prekida.
[4] Preskaler usporava uvećavanje brojačkog registra proizvoljan broj puta (u ovom slučaju 800).
[5] 800*10000/8000000=1 (Vrednost preskalera pomnožena sa graničnom vrednošću brojačkog registra, podeljena frekvencijom radnog takta jezgra koja je u ovom slučaju 8MHz).

```
#include <stdint.h>
#define REG *(volatile uint32_t *)
#define TIM2 0x40000000
#define RCC  0x40021000
#define GPIOC 0x40011000
#define TIM2_CR1 (TIM2+0x00)
#define TIM2_DIER (TIM2+0x0C)
#define TIM2_CNT (TIM2+0x24)
#define TIM2_PSC (TIM2+0x28)
#define TIM2_ARR (TIM2+0x2C)
#define TIM2_SR (TIM2+0x10)
#define RCC_APB1ENR (RCC+0x1C)
#define RCC_APB2ENR (RCC+0x18)
#define GPIOC_CRH (GPIOC+0x04)
#define GPIOC_ODR (GPIOC+0x0C)
#define NVIC_ISER (0xE000E100)

void tint();
```

U ovom segmentu se pre svega definiše makro `REG` koji služi da kastuje adresu koja sledi u odgovarajući tip pokazivača i da potom taj pokazivač dereferencira. Biblioteka stdint.h je potrebna samo zbog definicije celobrojnog tipa podataka konstantne dužine `uint32_t`. Nakon toga definišu se bazne adrese potrebnih periferija, kao i adrese samih registara kao zbir bazne adrese i pomeraja odgovarajućeg registra.

```
void main(void)
{
    REG NVIC_ISER |=0x10000000;
```

Kako bi omogućili prekide od strane tajmera potrebno je da setujemo odgovarajući bit `ISER` registra NVIC periferije. Prekid od strane Tajmera 2 koristi ulaz 28 (numerisanje počinje od 0) pa je to bit koji setujemo.

```
    REG RCC_APB2ENR |= 0x10;
    REG GPIOC_CRH   &= 0xFF0FFFFF;
    REG GPIOC_CRH   |= 0x00200000;
    REG GPIOC_ODR |=  0x2000;
```

Pre nego što možemo da koristimo GPIO pinove potrebno je da ih konfigurišemo. Da bi koristili bilo koji pin porta C (LED indikator je povezan na pin C13) potrebno je da omogućimo signal takta na tom portu, što možemo učiniti setovanjem potrebnog bita `RCC_APB2ENR` registra. Potom konfigurišemo GPIO pin C13 kao izlazni pin korišćenjem registra `GPIOC_CRH` i postavljamo početno stanje na logičku jedinicu registrom `GPIOC_ODR` (ovo zapravo znači da je dioda ugašena jer je pin C13 zapravo povezan na katodu LED indikatora).

```
    REG RCC_APB1ENR |=0x1;
    REG TIM2_CR1 |= 0x0004; //Only overflow generates an interrupt
    REG TIM2_DIER |= 0x0001; //Update generates an interrupt
    REG TIM2_CNT =0;
    REG TIM2_PSC =799;
    REG TIM2_ARR =9999;
```

Kako bi koristili tajmer opet je neophodno da omogućimo njegov signal takta, zatim podesimo da tajmer generiše prekid i to samo u slučaju prekoračenja koristeći registre `TIM2_CR1` i `TIM2_DIER` (značenje pojedinih bitova se ostavlja kao vežba čitaocu). Na kraju se podesi početna vrednost tajmera, vrednost preskalera i granica prekoračenja.

```
    //Start Timer
    REG TIM2_CR1 |= 0x0001;
    // This should hopefully generate an interrupt
    while(1)
    {
    }
}
```

Kada je sve podešeno preostalo je samo da pokrenemo tajmer, uđemo u beskonačnu while petlju i prepustimo ostatak prekidnoj rutini.

```
__attribute__((optimize("O0"))) void tint()
{
    REG TIM2_SR =0;
    REG GPIOC_ODR ^=  0x2000;
}
```

Ovo je zapravo kod same prekidne rutine, atribut `(optimize("O0"))` je neophodan kako kompajler ne bi uklonio ovu funkciju jer smatra da se nigde ne koristi. U samoj prekidnoj rutini je neophodno da resetujemo indikator da se prekid od strane tajmera desio kako bi se detektovali i naredni prekidi, taj indikator se nalazi u registru `TIM2_SR`. I preostalo je samo da promenimo vrednost GPIO pina C13 primenom XOR operacije na odgovarajući bit.

crt.s

Datoteka crt.s sadrži kratak komad asemblerskog koda koji podešava inicijalno stanje procesora i inicijalizuje odgovarajući ulaz u prekidnu rutinu.

```
.cpu cortex-m3
.thumb
```

Prve dve direktive samo govore asembleru koji je tip procesora i koji set instrukcija da koristi.

```
// end of 20K RAM
.word 0x20005000
.word _reset
.org 0xB0
.word 0x080001ed
```

Ovaj segment popunjava neophodne vrednosti u tabeli prekida kako bi proces mogao da radi. Prve dve vrednosti u tabeli prekida su inicijalna vrednost pokazivača na stek (ovde je postavljamo na kraj radne memorije) i adresa na koju se postavlja registar `PC`[6] pri pokretanju procesora. Nakon toga direktivom .org prelazimo na adresu 0xB0 koja odgovara ulazu za prekid Tajmera 2 i tu upisujemo adresu koju će naša prekidna rutina imati u gotovom programu. Ova Adresa se dobija uvidom u binarni fajl programa alatom `objdump` koji dolazi uz `GNU Arm Embedded Toolchain`, samim tim je neophodno kompajlirati program dva puta, prvi put sa privremenom vrednosti adrese kako bi iz binarnog fajla našli pravu adresu prekidne rutine, zatim privremenu adresu menjamo pravom adresom i ponovo kompajliramo program (kako se ništa nije menjalo osim vrednosti jedne memorijske lokacije sigurni smo da će adresa prekidne rutine ostati ista). Adresa samog ulaza se može naći u dokumentaciji za konkretan mikrokontroler mada je i njeno izračunavanje trivijalno.

```
.org 0x130
.thumb_func
_reset:
    bl main
    b .
```

Nakon konfigurisanja tabele prekida pozicioniramo se odmah iza nje u memoriji (adresa 0x130) i pišemo malu funkciju čiji je jedini posao da skoči na našu main funkciju i da nakon povratka iz main funkcije uđe u beskonačnu petlju. Ova "trampolina" funkcija služi kako bi zaobišli nedostatke asemblera i linkera gde globalne labele[7] (u ovom slučaju labela main) mogu da se nađu samo u instrukcijama dok asemblerske

[6] Registar `PC` sadrži adresu instrukcije koju procesor trenutno izvršava.

[7] Lokalne labele se nalaze u istom fajlu dok se globalne labele nalaze u drugim fajlovima (njih povezuje linker).

direktive rade samo sa lokalnim labelama.

linker.ld

Datoteka linker.ld ima poprilično prost format i služi da na sistemima sa ne uniformnom memorijom (kao što je većina mikrokontrolera) linkeru da informacije o rasporedu memorije kako bi mogao pravilno da rasporedi delove programa. U ovom slučaju se memorija sastoji od radne memorije i fleš memorije u koju se obično smešta kod i u koju se ne može pisati iz programa. Linkeru se daje informacija o nazivu segmenta, privilegijama (da li se u segment može pisati i da li se može izvršavati kod iz tog segmenta0), početnoj adresi segmenta kao i njegovoj veličini. Na osnovu toga linker će smestiti programski kod (tzv. text segment) u fleš memoriju dok bi se globalne promenljive ukoliko postoje našle u radnoj memoriji. Pri pokretanju mikrokontrolera u zavisnosti od boot konfiguracije (podešava se džamperima na samoj pločici) adrese jednog od ova dva segmenta će biti preslikane na adresu 0x0 i odatle će početi izvršavanje programa (u ovom slučaju želimo da preslikamo fleš memoriju na adresu 0x0 jer se tu nalazi naš programski kod).

```
MEMORY
{
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 64K
  RAM (xrw)  : ORIGIN = 0x20000000, LENGTH = 20K
}
```

build.bat

Datoteka build.bat je batch fajl koji obavlja sve potrebne korake kako bi se program kompajlirao i preneo na mikrokontroler. Koraci kroz koje prolazi build.bat su:

1. Asembliranje koda u crt.s
2. Kompajliranje i asembliranje koda u main.c
3. Linkovanje dobijenih objektnih fajlova
4. Kopiranje relevantnih delova izvršnog fajla u binarni fajl
5. Generisanje pregleda izvršnog fajla kako bi mogli da odredimo adresu prekidne rutine.
6. Opciono (ukoliko nije specifikovan argument -np) prenošenje programa na mikrokontroler.

```
arm-none-eabi-as -o crt.o crt.s
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -c -o main.o main.c
arm-none-eabi-ld -T linker.ld -o app.elf crt.o main.o
arm-none-eabi-objcopy -O binary app.elf app.bin
arm-none-eabi-objdump.exe app.elf -D -s > app.objdump
IF NOT "%1" == "-np" (ST-LINK_CLI -P app.bin 0x08000000 -V "while_programming" -Rst)
```

Dok su komande za kompajliranje poprilično standardna stvar činjenica da `STM32 ST-LINK Utility` može da se koristi iz komandne linije je bila prijatno iznenađenje. Sami argumenti nisu preterano komplikovani pa će ovde biti ukratko objašnjeni. Argument -P govori programu da prenese program na mikrokontroler, argumenti koji slede su ime binarnog fajla i adresa od koje se započinje prenos. Argument -V govori programu da tokom programiranja ujedno izvrši i proveru da li je program dobro prenet. I konačno argument -Rst govori programu da nakon prenošenja programa resetuje mikrokontroler.

Zaključak

Nadam se da je sadržaj ovog izveštaja bio informativan ako ništa drugo, i dao vam uvid u malo niže aspekte programiranja i funkcionisanja mikrokontrolera. I nadam se da ćete zbog toga malo više ceniti činjenicu da ne morate ovako da programirate mikrokontrolere (a neko je nekada morao) jer postoje alati koji značajno olakšavaju ovaj proces o kojima će biti više reči u sledećim izveštajima.