UNIVERZITET
"DŽEMAL BIJEDIĆ"
U MOSTARU

STM32 Common Bilbioteku za rad sa periferalima

Predmet : Ugrađeni sistemi i sistemi za rad u realnom vremenu

Smjer : Softverski Inženjering

Predmetni profesor : -//-

Student : Armin Smajlagić

Sadržaj

1. Uvod u projekat

Ova biblioteka olakšava rad sa periferalima STM32 ARM Cortex M4 platformama. Često kada želimo da naša aplikacija koristi tajmer kako bi vršili neku radnju u fiksnim intervalima ili da dobijemo informaciju ulaznim signalu sa jedne od GPIO pinova trebamo podešavati registre. Tu dolazi ova biblioteka kao brzo i jednostavno rješenje.

Unutar projekta se većinom konfigurišu registri sistemskih interupta koji nam omogućavaju da reagujemo na vremenske evente u fiksnim intervalima, da spremimo kontekst programa na stek i da nastavimo poslije sa njegovim izvršavanjem ili ipak da prisluškujemo GPIO linije na signale te da reagujemo.

NVIC (Nested Vector Interrupt Controller) interni periferal sadrži interupte koji se mogu enable/disable, odrediti prioritet ili promijeniti stanje. Drugi periferali kao USART za serijsku komunikaciju, SysTick tajmer ili MUX multiplekser za GPIO se spajaju na NVIC koji adekvatno interupta CPU kako bi se dao odgovor tim perfieralima. Čitav projekat se oslanja na okidanje interupta i njihovo adekvatno hendliranje.

Projekat je rađen u Keil qVision i testiran u simuliranom okruženju na STM32F401CDUx uređaju. Implementacija biblioteka je vršena u c programskom jeziku i asembliju. Segment asemblija se implementirao sa inline assembly gdje unutar samog c jezika pišemo asembli korištenjem ključne riječi *__ASM*. Razlog zašto sam koristio asemblij je nemogućnost pristupa specijalnim registrima kao što su MSP, PSP (shadow registri stak pointeri), spremanje konteksta programa i sl.

Projekat sadrži :

- SysTick Handler & Delay
- SysTick Round-Robin CPU Task Scheduler
- Externi Inteupt Handleranje
- Fault Handleranje

2. SysTick tajmer interupt & Delay

2.1 Šta je SysTick i koji su njegovi registri

SysTick je down-counter koji proizvodi fiksne vremenske intervale kako bi se generisao interupt ili napravio delay. Većina, ako ne i svi RTOS i CPU Task Scheduleri koriste SysTick tajmer.

Pri implementaciju SysTick.h modula sam konfigurisao sljedeča 3 registra :

- SysTick_CTRL (memorijska adresa 0xE000E010) – kontrolni i statusni registar
- SysTick_LOAD (memorijska adresa 0xE000E014) – registar reload vrijednosti
- SysTick_VAL (memorijska adresa 0xE000E018) – registar trenutne vrijednosti

2.2 Implementacija SysTick-a

Za implemenetaciju SysTick-a trebamo izvršiti sljedeće korake :

1. Prvo treba da se onemogući SysTick sat i interupt zahtjevi
2. Zatim postaviti reload vrijednost (vrijednost na koju se tajmer vrati nakon underflow-a)
3. Resetuje se counter vrijednost na nulu
4. Odabere se sat externi ili CPU
5. Omogući se SysTick sat
6. Omogući se SysTick interupt request

Inicijalizacija SysTick-a :

```
void Init_SysTick(uint32_t ticks,uint32_t clock, uint32_t priority, uint32_t curr_val, void (*event)()){
  DelayPeriod = 0;  // po zadanom nema delaya
  SysTick->CTRL = 0; // onemoguciti SysTick IRQ i sat
  SysTick->LOAD = ticks - 1; //postavljanje reaload vrijednosti
  NVIC_SetPriority(SysTick_IRQn, priority); // definisanje prioriteta SysTick_IRQ
  SysTick->VAL = curr_val; //resetovanje vrijednosti (trebalo bi biti nula)
  SysTick->CTRL |= clock; // odabir sata
  NVIC_EnableIRQ(SysTick_IRQn); // postavljanje enable bita na 1 za SysTick_IRQ
  func = event; // definisanje eventa kojeg želimo izvršiti na underflow
}
```

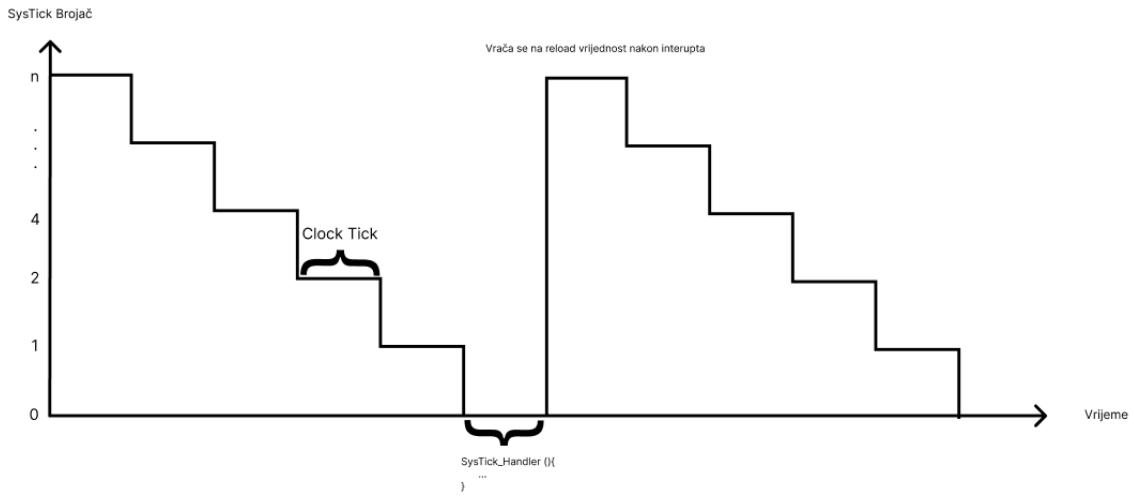
Tajmer delay :

```
void SysTick_Delay(uint32_t period){
  delay = 1; // 1 je za delay enabled
  DelayPeriod = period; // definisanje perioda delaya
  while(DelayPeriod != 0); // delay
}
```

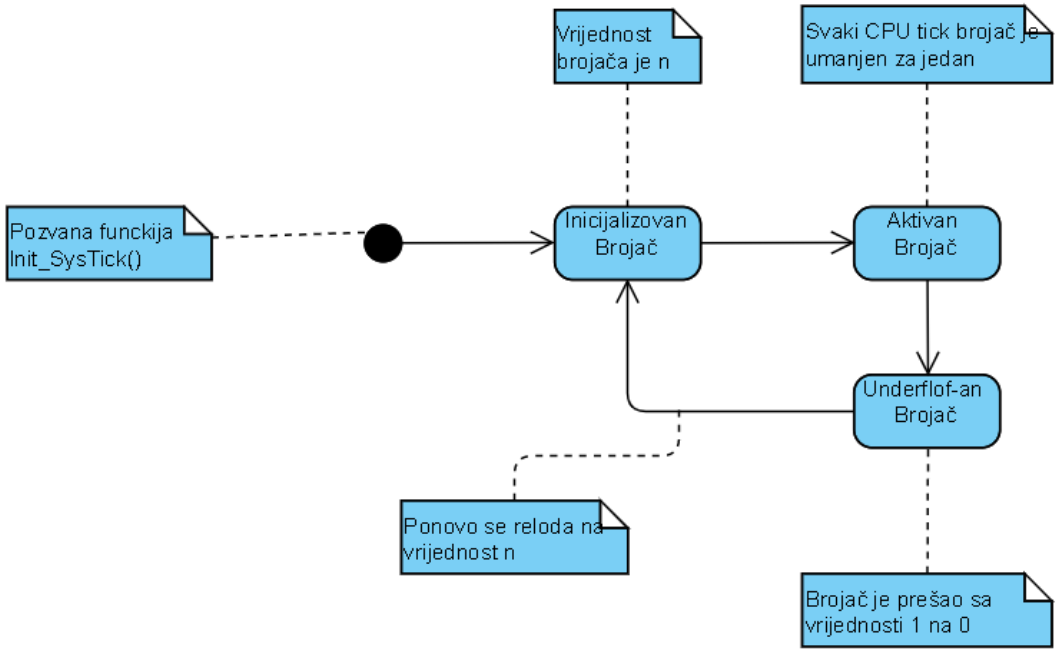
Generalizovana implementacija SysTick_Handlera :

```
void SysTick_Handler(void){
  //koristimo sys tick samo sa delay
  if(delay == 1){
    if(DelayPeriod > 0)
      DelayPeriod--;
  }
  //koristimo SysTick za vršenje neke radnje u fiksnim vremenskim intervalima
  else{
    (*func)();
  }
}
```

2.3 Dijagrami



Slika 1. Prikaz SysTick tajmera i okidanje SysTick_Hanlder() na SysTick underflow



Slika 2. Dijagram stanja SysTick brojača

3. SysTick Round-Robin CPU Task Scheduler

3.1 Šta je SysTick Round-Robin CPU task scheduler

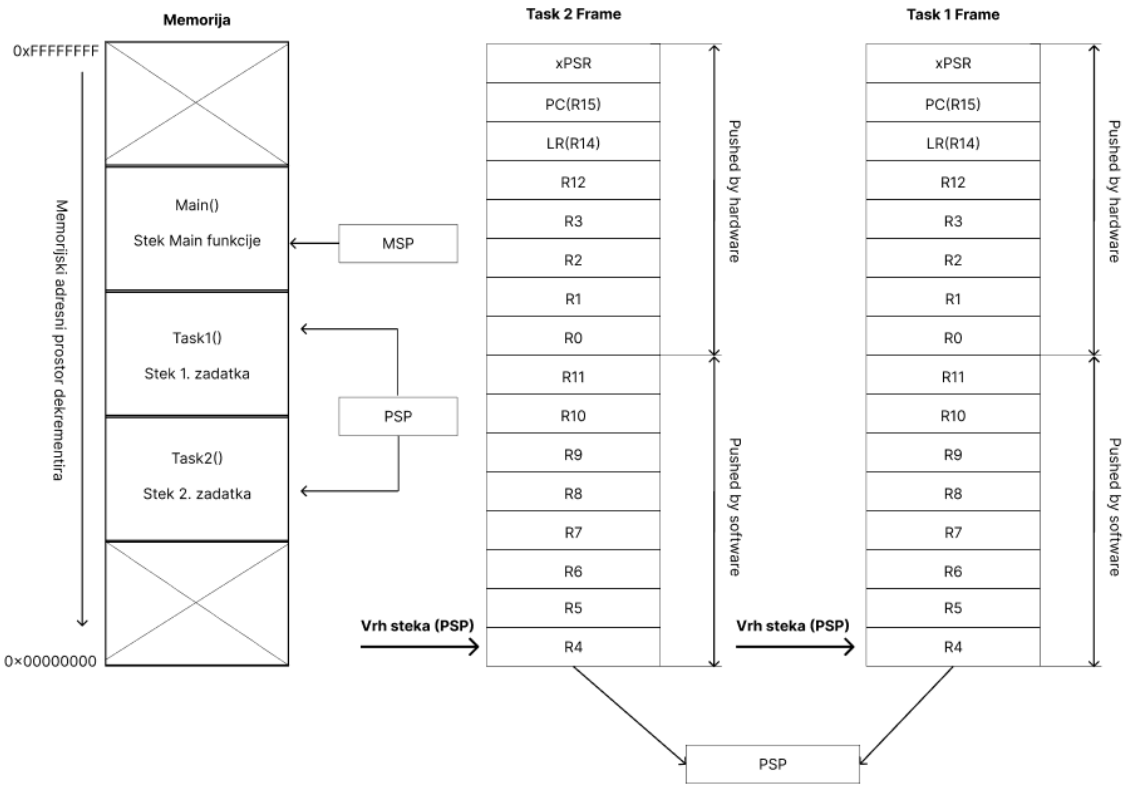
Ovo je program koji raspoređuje zadatke u Round-Robin sekvenci gdje se sa izvršavanja jednog zadatka prelazi na izvršavanje drugog zadataka u fiksnim vremenskim intervalima. Rezultat izvršavanja CPU task schedulera je pseudo-paralelizam izvršavanja zadataka.

Implementacija je vršena u c programskom jeziku, te korištenjem inline-assembly je uključen i asembli. Razlog je nemogućnost pristupu specijalnim registrima kao što su MSP (main stack pointer) i PSP (procces stack pointer) korištenjem asembli instrukcija MRS i MSR za čitanje i pisanje specijalnih registara.

SysTick Round-Robin je scheduling metoda gdje se procesor dodjeljuje svakom zadatku na jednaki vremenski interval kružnim redoslijedom.

Svaki zadatak kojeg želimo raspodijeliti treba da ima svoj okvir koju će da čuva kontekst zadataka (stanje svih registara). Definisanjem stack pointera (shadow PSP) koji pokazuje na vrhu steka (frejm zadatka) možemo se vratiti na taj zadatak ponovno poslije i nastaviti izvršavanje.

Pri prelasku sa jednog zadatka na drugi trebamo spremiti kontekst trenutnog zadataka na stek (stacking) i zamijeniti ga sa kontekstom drugog zadatka (context switch) te kada se ponovo vratimo na isti izvršiti unstacking i spremiti ga u registre. Za pohranu i restore konteksta zadatka koristit ću stek frejm.



Slika 3. Prikaz alokacije memorije za zadatke i context switch

3.2 Implementacija Task Schedulera

Započinjemo implementaciju task schedulera sljedećim nizom koraka :

- Inicijalizacije SysTick-a za mjerenje vremena izvršavanja pojedinih zadataka
- Inicijalizacija steka samog schedulera
- Inicijalizacija defaultnog steka frejma svako od zadataka
- Prelazak sa MSP na PSP (kroz asembli podesiti sp = psp = sp 1. zadatka)
- Započeti izvršavanje prvog zadatka

Odgovornosti inicijalizacije task schedulera preuzima funkcija Scheduler_Init();

```
void Scheduler_Init(void (*task1)(void),void (*task2)(void),void (*task3)(void),void (*task4)(void)){


  enable_processor_faults();

  Task1 = task1;
  Task2 = task2;
  Task3 = task3;
  Task4 = task4;

  init_scheduler_task_stack();

  init_user_tasks_stack();

  SysTick_Init(); // initializing systick clock

  switch_sp_to_psp();

  Task1_Handler();
}
```

U ovoj liniji postavljamo trenutno vrijednost MSP na definisanu adresu na steku (adresa samog schedulera);

```
__attribute__((naked))
void init_scheduler_task_stack(void)
{
  __asm volatile("MSR MSP,%0" : : "r"((SRAM_END) - (5 * SIZE_TASK_STACK)) : );
  __asm volatile("BX LR");
}


__attribute__((naked))
void switch_sp_to_psp(void)
{
  // Initialize the PSP with TASK1 starting value
  __asm volatile("PUSH {LR}"); // preserve LR which connects back to main
  __asm volatile("BL get_psp_value");
  __asm volatile("MSR PSP, R0"); // initialize PSP
  __asm volatile("POP {LR}"); // bring LR back

  // Change the MSP to PSP
  __asm volatile("MRS R0, CONTROL"); // Saving CONTROL Register to R0
  __asm volatile("ORR R0, R0, #0x02"); // Setting the 1st bit
  __asm volatile("MSR CONTROL, R0"); // Load back to CONTROL Register
  __asm volatile("BX LR");
}
```

Postavljanje inicijalnih vrijednosti pojedinih taskova (stanje, handleri, memorija);

```
void init_user_tasks_stack(void)
{
  int i;
  uint32_t *pPSP;

  tasks[0].task_state = TASK_READY_STATE;
  tasks[1].task_state = TASK_READY_STATE;
  tasks[2].task_state = TASK_READY_STATE;
  tasks[3].task_state = TASK_READY_STATE;

  tasks[0].task_handler = Task1_Handler;
  tasks[1].task_handler = Task2_Handler;
  tasks[2].task_handler = Task3_Handler;
  tasks[3].task_handler = Task4_Handler;

  tasks[0].task_frame = T1_STACK_START;
  tasks[1].task_frame = T2_STACK_START;
  tasks[2].task_frame = T3_STACK_START;
  tasks[3].task_frame = T4_STACK_START;

  //...
```

Postavljanje inicijalnih vrijednosti za registre (r0 – r15);

```
for(i=0; i<MAX_USER_TASKS ; i++)
{
  int j;

  pPSP = (uint32_t*)tasks[i].task_frame;

  pPSP--; // xPSR
  *pPSP = (uint32_t)0x01000000; // DUMMY xPSR Should be 0x01000000 because of the T bit

  pPSP--; // PC
  *pPSP = (uint32_t)tasks[i].task_handler;

  pPSP--; // LR
  *pPSP = (uint32_t)0xFFFFFFFD; // Special Return Value

  for(j = 0; j<13; j++)
  {
    pPSP--; // General Purpuse Registers R12-R0
    *pPSP = (uint32_t)0x00;
  }

  tasks[i].task_frame = (uint32_t)pPSP; // Saving the final value of the PSP
}
```

Nakon inicijalizacije započinje izvršavanje 1. zadataka i tajmer broji vrijeme. Nakon određenog vremenskog intervala poziva se SysTick_Handler() koji vrši context switch.

U ovom dijelu moramo izvršiti naredne korake:

- Spremiti novu PSP vrijednost u R0
- Korištenjem spremljene vrijednosti PSP spremiti kontekst trenutnog zadatka
- Odlučiti naredni task
- Restorati kontekst novog odlučenog zadatka
- Postaviti novu vrijednost PSP-a
- Vratiti se na PSP

```
void SysTick_Handler(void){

    /* First push the LR, Because we will use BL instruction */
  __asm volatile("PUSH {LR}");
  /* Save  the context of current tasks */
  //1. Get current running task's PSP value
  __asm volatile("MRS R0, PSP");
  //2. Using that PSP value store SF2(R4 to R11)
  __asm volatile("STMDB R0!, {R4-R11}");
  //3. Save the current value of PSP
  __asm volatile("BL save_psp_value");

  /* Retrieve the context of next task */
  //1. Decide next task to run
  __asm volatile("BL decide_next_task");
  //2. Get its past PSP value
  __asm volatile("BL get_psp_value");
  //3. Using that PSP value retrieve SF2(R4 to R11)
  __asm volatile("LDMFD R0!, {R4-R11}");
  //4. Update PSP
  __asm volatile("MSR PSP, R0");

  tasks[current_task].task_state = TASK_RUNNING_STATE;

  /* POP the LR and go back */
  __asm volatile("POP {LR}");
  __asm volatile("BX LR");

}


//Round-Robin task scheduling
void decide_next_task(void)
{
    int i;
    for(i=0;i<MAX_USER_TASKS;i++){

      tasks[current_task].task_state = TASK_BLOCKED_STATE;

      current_task++;

      if(current_task == 4){
        current_task = 0;
        unblock_user_tasks();
      }

      if(tasks[current_task].task_state == TASK_READY_STATE)
        break;

    }

}
```

Izmjena konteksta u modernim RTOS se vrši unutar PendSV_Handler-a.

```
void PendSV_Handler(void)
{
  //context switch
}
```

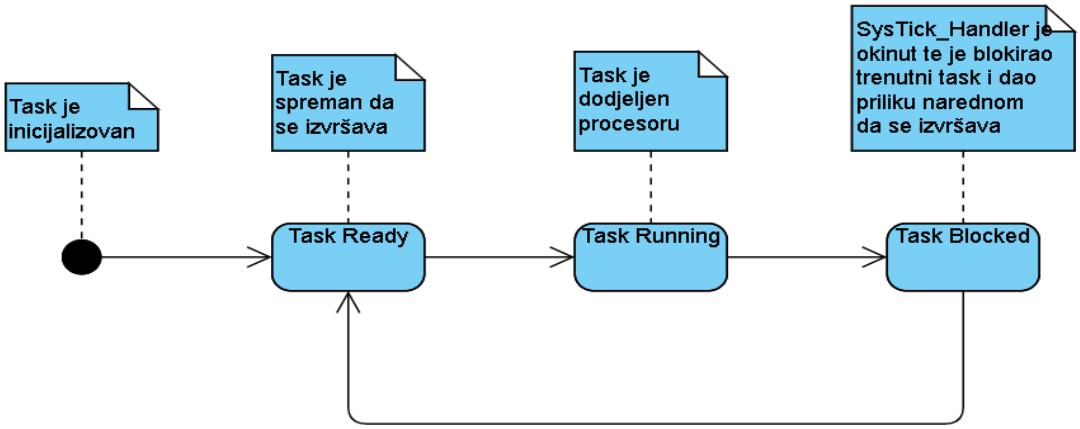
PendSV handler se poziva kad se okine sa linijama koda ispod.

```
uint32_t volatile *pICSR = (uint32_t *)0xE000ED04;
*pICSR |= (1 << 28);
```

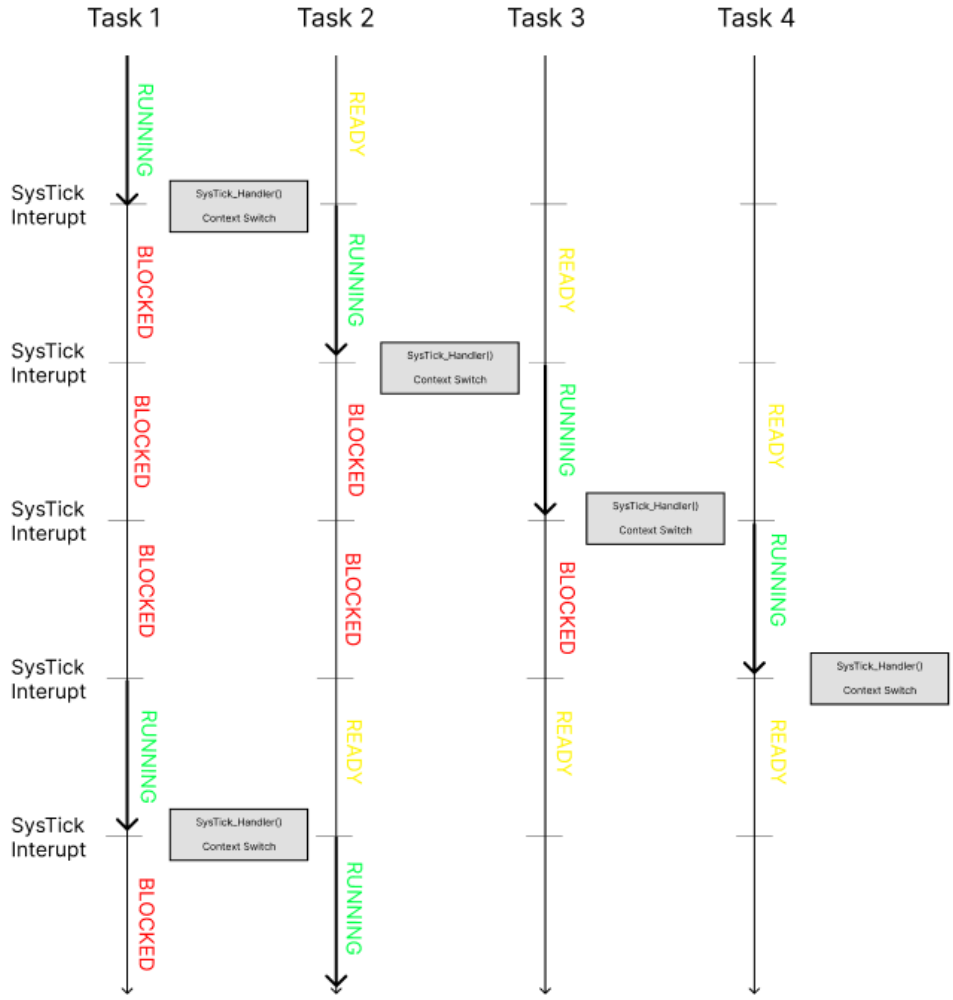
Primjer upotrebe SysTick Round-Robin CPU Task Schedulera

```
__attribute__((noreturn))
void task1(void){
  //this is task 1
  while(1);
}
__attribute__((noreturn))
void task2(void){
  //this is task 2
  while(1);
}
__attribute__((noreturn))
void task3(void){
  //this is task 3
  while(1);
}
__attribute__((noreturn))
void task4(void){
  //this is task 4
  while(1);
}

int main(void)
{
  Scheduler_Init(task1,task2,task3,task4);

  //this never gets executed
  for(;;);
}
```

3.3 Dijagrami



Slika 4. Task State Machine dijagram

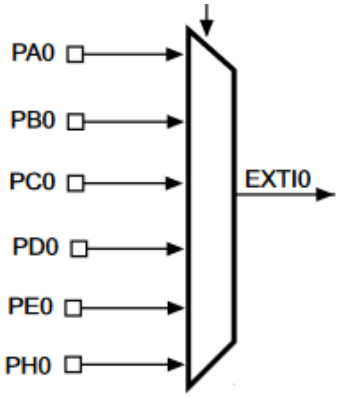


Slika 5. Sekvenca izvršavanja sa stanjima kroz vrijeme (Round-Robin sekvenca)

4. Hendliranje eksternih interupta

4.1 Šta su EXTI i kako možemo reagovati na signale generisana na GPIO linijama

Eksterni interupti se spajaju na NVIC kroz eksterni interupt kontroler (EXTI) koji nam daje mogućnost mapiranja više GPIO linija na NVIC. Svaka linija se može konfigurisati da okidaju različite događaje, kao npr. EXTI0 linija se podesi da prisluškuje ulazne signale sa linije 0 (nije bitna kombinacija porta).



Svaka linija se može konfigurisati da okidaju interupt na sljedeće događaje:

- Rastuća ivica (Rising edge)
- Opadajuća ivica (Falling edge)
- Kombinacija oba

Eksterni interupt/event kontroler se sastoji od 24 detekotra ivica (za različite linije) koji se koriste za generisanje interupta/eventa. Podržava čak do 81 GPIO da se spoji na glavnih 16 eksterni interupt linija.

Linije od 0-4 imaju zasebni eksterni interupt handler, dok linije 5-9 dijele jedan kao i 10-15.

```
/******  STM32 specific Interrupt Numbers ***************************************
  EXTI0_IRQn                  = 6,      /*!< EXTI Line0 Interrupt
  EXTI1_IRQn                  = 7,      /*!< EXTI Line1 Interrupt
  EXTI2_IRQn                  = 8,      /*!< EXTI Line2 Interrupt
  EXTI3_IRQn                  = 9,      /*!< EXTI Line3 Interrupt
  EXTI4_IRQn                  = 10,     /*!< EXTI Line4 Interrupt

  EXTI9_5_IRQn                = 23,     /*!< External Line[9:5] Interrupts

  EXTI15_10_IRQn              = 40,     /*!< External Line[15:10] Interrupts
```

Soruce code iz **stm32f401xe.h**

4.2 Implementacija EXTI handelinga

Pri implementaciji EXTI handelinga moramo konfigurisati sljedeće :

- Definiranje prioriteta IRQ-a
- Postavljanje enable bit-a u registar željene linije

```
void Init_EXTI(int line, int priority, void (*event)()){
  func = event;
  switch(line){
    case 0:
    {
      NVIC_SetPriority(EXTI0_IRQn, priority);
      NVIC_EnableIRQ(EXTI0_IRQn);
      break;
    }
    case 1:
    {
      NVIC_SetPriority(EXTI1_IRQn, priority);
      NVIC_EnableIRQ(EXTI1_IRQn);
      break;
    }
    //...
```

Unutar Init_EXTI dinamički proslijedimo liniju koju prisluškujemo za event i dinamički postavljamo funkciju koju želimo izvršiti u trenutku vršenja IRQ za datu liniju.

Dalje u istoj funkciji za raspon 5-9 biramo EXTI9_5_IRQn, a za 10-15 biramo EXTI15_10_IRQn.

```
//...
case 5:
case 6:
case 7:
case 8:
case 9:
{
  NVIC_SetPriority(EXTI9_5_IRQn, priority);
  NVIC_EnableIRQ(EXTI9_5_IRQn);
  break;
}
case 10:
case 11:
case 12:
case 13:
case 14:
{
  NVIC_SetPriority(EXTI15_10_IRQn, priority);
  NVIC_EnableIRQ(EXTI15_10_IRQn);
  break;
}
```

Pri handliranju samog eventa/interupta trebamo uraditi sljedeće :

- Provjeriti pending registar EXTI->PR datog EXTI enable-an
- Izvršiti željenu radnju
- Disable-at pending registar EXTI->PR zapisivanjem 1 u njegov nulti bit

```
void EXTI0_IRQHandler(void){
  if (EXTI->PR & (1<<0)) {
    func();
    EXTI->PR |= (1<<0);
  }
}
```

Nakon toga jednostavno EXTIn možemo inicijalizirati prosljeđivanjem linije, prioriteta i funkcije.

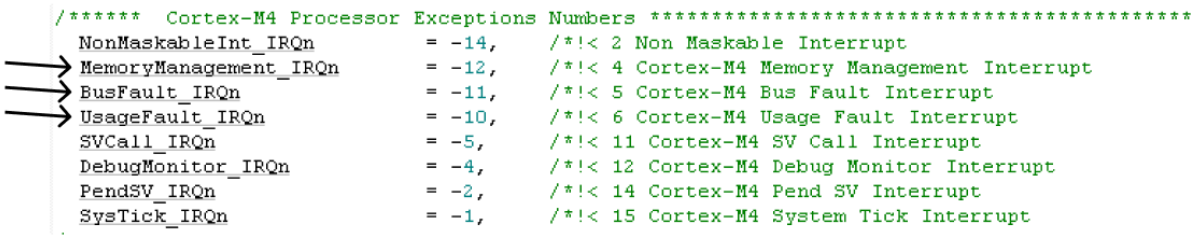
```
#include "EXTI.h"
#include <stdio.h>

void some_func(void);

void some_func(void){
  printf("EXTI interupt handling funkcija me pozvala");
}

int main(void){

  int prioritet = 0;
  int broj_linija = 15;

  int i;

  for(i=0; i < broj_linija;i++){
    Init_EXTI(i,prioritet,some_func);
  }

}
```

5. Fault Handleranje

5.1 Šta je fault i kada se javlja

Fault u programerskom žargonu je exception (izuzetak) koji se dogodi pri nepravilnom odnosu prema hardveru ili je sam sistem zaštitio neki resurs i generisao softverski jedan od spomenutih izuzetaka.

Slika ispod pokazuje interupte izazvane izuzetcima (za sve Cortex-M4 procesore) koje mi možemo konfigurisati.



```
/******  Cortex-M4 Processor Exceptions Numbers ***********************************************
  NonMaskableInt_IRQn         = -14,    /*!< 2 Non Maskable Interrupt
  MemoryManagement_IRQn       = -12,    /*!< 4 Cortex-M4 Memory Management Interrupt
  BusFault_IRQn               = -11,    /*!< 5 Cortex-M4 Bus Fault Interrupt
  UsageFault_IRQn             = -10,    /*!< 6 Cortex-M4 Usage Fault Interrupt
  SVCall_IRQn                 = -5,     /*!< 11 Cortex-M4 SV Call Interrupt
  DebugMonitor_IRQn           = -4,     /*!< 12 Cortex-M4 Debug Monitor Interrupt
  PendSV_IRQn                 = -2,     /*!< 14 Cortex-M4 Pend SV Interrupt
  SysTick_IRQn                = -1,     /*!< 15 Cortex-M4 System Tick Interrupt
```

Bus Fault je izuzetak koji se dogodi kada imamo fault vezan za memoriju kojoj instrukcija pristupa ili neki drugi vid memorijske transakcije (error detektovan na busu unutar memorijskog sistema).

Memory management fault se dogodi kada imamo problem oko zaštićene memorije. Kada MPU (Memory Protection Unit) definiše neki ograničenje na memorijski region a mi pokušamo pristupiti dobijemo MemMange fault (npr. Regioni prozvani kao XN (execute never) uvijek izazovu ovaj fault).

Usage fault se dešava kada imamo fault vezan za izvršavanje instrukcije, pa imamo:

- Nepostojeća instrukcija
- Error pri povratku sa izuzetka
- Dijeljenje nulom
- Ilegalan pristup unaligned adresi (npr. Word ili halfword nije pravilno align-an)

Hard fault se poziva kada nemamo ni jedan drugi exception mehanizam da ga pravilno hendlira.

5.2 Implementacija fault handelinga

Proces omogućavanja ovih interupta je poprilično jednostavan. Sliku ispod pokazuje kako se postavljaju enable bitovi (1) na 18. , 17. i 16. bit unutar SHCSR registra. Hard fault je uvijek enable-an od početka.

```
void enable_processor_faults(void)
{
  //  uint32_t volatile *SHCSR = (uint32_t *)0xE000ED24;
  //  *SHCSR = (1 << 18); // Enabling the Usage Fault - setting 18th bit to 1
  //  *SHCSR = (1 << 17); // Enabling the Bus Fault - setting 17th bit to 1
  //  *SHCSR = (1 << 16); // Enabling the MemManage Fault - setting 16th bit to 1

  // OR

  SCB->SHCSR |=(1 << 18);
  SCB->SHCSR |=(1 << 17);
  SCB->SHCSR |=(1 << 16);
}
```

Slika ispod prikazuje hendliranje ova 4 interupta (Oni se automatski pozivaju kada se adekvatan fault dogodi).

```
__attribute__((noreturn)) void MemManage_Handler(void)
{
  printf("Exception: MemManage Fault\n");
  while(1);
}

//Gets triggered when user program trys to access memory in a unappropriate way
__attribute__((noreturn)) void BusFault_Handler(void)
{
  printf("Exception: BusFault\n");
  while(1);
}

//Gets triggered when user program causes unappropriate memmory management
__attribute__((noreturn)) void UsageFault_Handler(void)
{
  printf("Exception: Usage Fault\n");
  while(1);
}

// Gets triggered when there is no other appropriate fault handler
__attribute__((noreturn)) void HardFault_Handler(void)
{
  printf("Exception: HardFault\n");
  while(1);
}
```

6. Reference

[1] STM32F4xx Data Sheet

[2] STM32F4xx Reference Manuel

[3] Embedded systems with ARM Cortex-M3 Microcontrollers in Assembly Language and C

[4] developer.arm.com/documentation