List of functions : FreeRTOS - WIZ820io

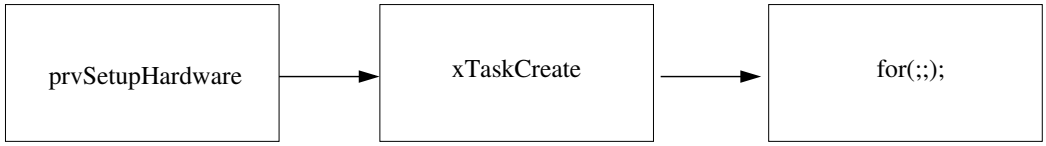
Ljubljana, 10. Julij 2014

1 Uvod

FreeRTOS aplikacije se poganjajo kot vse ostale, dokler ne pokličemo ukaza vTaskStartScheduler(), ki ga ponavadi kličemo iz "main()". Preden uporabljamo funkcije RTOS-a se je dobro prepričati, da delujejo vse funkcije za izbrano tarčo?. FreeRTOS je zgrajen okoli treh glavnih C datotek

- tasks.c - kjer so rutine za ustvarjanje nalog, etc.
- queue.c - kjer so rutine za manipulacijo z nalogami (blokiranje, pošilanje v ozadje, ...)
- list.c - ?
- portable/[compiler]/[architecture]/port.c

Glavne mape, ki jih moramo vključiti pri prevajanju sta **Source/include** in **Source/portable/[compiler]/[architecture]**. Vsak projekt mora vsebovati tudi konfiguracijsko datoteko "FreeRTOSConfig.h", kjer so definirane rutine in spremenljivke, ki temeljijo na namembnosti našega projekta. Najlažji način za ustvarjanje novih aplikacij je, da iz strani *www.freertos.org* povlečemo source in v mapi **Demo** poiščemo ustrezno mapo za našo tarčo. Nato najprej prevedemo projekt in ga preizkusimo na naši tarči. Ko smo se prepričali, da deluje, lahko nadaljujemo z nadgrajevanjem demo primera na željeni projekt.
Struktura funkcije **main()** bi naj izgledala takole



2 Interrupts - interupti

Interrupti so lahko bodisi strojni - zunanji bodisi programerski. Definiramo jih s pomočjo NVIC strukture. Pri tem moramo vključiti ustrezne header datoteke. Nastaviti moramo kateri kanalo opazujemo, kakšne prioritete ima naš interrupt, ter ga seveda omogčiti.

```
NVIC_InitStruct.NVIC_IRQChannel = EXTI0_IRQn;
NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 0x1;
NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0x0;
NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
```

V kolikor želimo uporabiti zunanji interrupt moramo prej definirati zunanj interupt na EXTI vodilu.

```
EXTI_InitStruct.EXTI_Line = EXTI_Line0;
EXTI_InitStruct.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStruct.EXTI_Trigger = EXTI_Trigger_Falling;
EXTI_InitStruct.EXTI_LineCmd = ENABLE;
```

Ob dogodku na vodilu se potem pokliče funkcija **EXTIn_IRQHandler(void)** v katero moramo definirati in v njej počisti interrupt flag in nato naredimo, kar želimo ob določenem interruptu. Naloge so v FreeRTOS osrednji del strukture programa. Nalogo ustvarimo s klicem **vTaskCreate**, kjer določimo, nalogo, njeno ime, prioriteto, parametre ki jih želimo podati in TaskHandle. Naloge - Tasks so lahko v štirih stanjih

- running : kadar se naloga izvršuje
- ready : so tiste naloge, ki čakajo na izvršitev, ker se trenutno izvršuje naloga z enako oziroma višjo prioriteto
- blocked : kadar naloga čaka na nek zunanji dogodek ali podatek oziroma sprostitev semaforja oz. čakalne vrste.
- suspended : v tem stanju je naloga samo kadar kličemo **vTaskSuspend()**. Iz tega stanja je mogoče priti samo z **vTaskResume()**

3 Queues - čakalne vrste

Komunikacija med nalogami poteka s pomočjo čakalnih vrst. Med nalogami lahko pošiljamo in sprejemamo vrednosti lokalnih spremenljivk s pomočjo **xQueueSend** in **xQueueRecive**.

4 Semaphores - semaforji

Semaforji so nadgradnja čakalnih vrst (Queue) in se uporabljajo za sinhronizacijo med nalogami, kot tudi znotraj nalog. Za sinhronizacicijo mend nalogami ustvarimo **xSemaphoreCreateBinary()**. Lahko bi uporabili tudi **xSemaphoreCreateMutex()**, vendar v dokumentaciji pri uporabi ISR slednjega ne priporočajo. Dostop do določenega dela programa blokiramo s funkcijo

```
xSemaphoreTake(xSemaphoreHandle, TickType_t);
```

ki ga blokira dokler ga ne sprostimo s funkcijo

```
xSemaphoreGive(xSemaphoreHandle);
```

Spet se je pri uporabi ISR bolje poslužiti **xSemaphoreTakeFromISR** in **xSemaphoreGiveFromISR**. [1]

5 Echo TCP server

Preprost Echo TCP server ustvari en socket in čaka na klienta, da se poveže. Ob vzpostavitvi povezave ne naredi ničesar. Ob prejemu podatkov pa vrne iste podatke. Če so podatki enaki 'quit' zapre socket in pošlje nalogo v spanje. TCP server je nadgradnja demo primera iz git repozitorija *git://github.com/vagabondtt1503/STM32F4-Discovery-FreeRTOS-GCC.git*. Dodamo mu dve novi datoteki s funkcijami za upravljanje z Wiznet modulom in TCP protokolom (spi.c, W5200.c) in prilagodimo **main.c**.

5.1 main.c

Najprej vključimo header datoteke, ki jih potrebujemo za naš server in definiramo prototip naloge.

```
#include "spi.h"
#include "W5200.c"
void set_macTask(void *  pvParameters)
```

Funkciji **main()** dodamo nov task

```
xTaskCreate(set_macTask,
                    "SETMAC",
                    configMINIMAL_STACK_SIZE,
                    NULL, mainFLASH_TASK_PRIORITY,
                    &set_macTaskHandle);
```

V funkcijo **prvSetupHardware** dodamo našo funkcijo, ki nastavi strojno opremo za prenos podatkov preko SPI vodila in interrupte.

```
static void prvSetupHardware(void)
{
        .....
```

[1] Primer Echo TCP serverja, ki komunicira preko SPI s pomočjo DMA ne deluje, saj je iz neznanega razloga prioriteta veliko nižja kot prioriteta trenutne naloge in zaradi tega obstane v zanki znotraj funkcije **xSemaphoreGiveFromISR**.

```
        init_SPI1();

        ....
}
```

Nazadnje še določimo nalogo. Najprej je potrebno inicializirati wiznet, mu nastaviti mac naslov, ip naslov, gateway in masko.

```
uint8_t address[6] = {0xdd,0xaa,0xbb,0xcc,0x11,0x22};
const  uint8_t ip[4] = {0xc0, 0xa8, 0x0, 0x08};
const uint8_t subnet[4] = {0xff,0xff,0xff,0x0};
const uint8_t gw[4] = {0xc0, 0xa8, 0x0, 0x01};

wiznet_initialize();
wiznet_configure_network(ip, subnet, gw);
wiznet_set_hardware_address(address);
```

Sedaj ustvarimo socket na portu 80, nastavimo wiznet v način poslušanja.

```
/*create socket and send byte */
uint8_t s;
s = socket(W5200_Sn_MR_TCP, 80, 0);
uint8_t buf [100];
listen(s);
```

Naloga tukaj vstopi v neskončno zanko, kjer pošlje sama sebe v spanje, dokler se ne zgodi dogodek na zunanjem pinu, ki smo ga določili v "spi.c". Ob dogodku, določi kakšen dokodek se je zgodil in v primeru prejema podatkov prejme podatke in jih pošlje nazaj.

```
int len;
for( ;; )
{
  /*we are now listening
  suspend task set_macTask()*/
  vTaskSuspend(set_macTaskHandle);
  len = recv(s, buf, 100, 0);

  if ( !strncmp(buf, "quit", 4))
  {
    uint8_t * r = "Bye\n";
    send(s, r, 4,0);
    closesocket(s);
  }
```

```
  send(s, buf, len, 0);
}
vTaskDelete( NULL );
```

5.2 W5200.c

Dogodek na zunanjem pinu (interrupt wizneta) je v "spi.c" definiran tako, da požene funkcijo **void EXTI0_IRQHandler(void)**. Kjer najprej preverimo, če se je res zgodil dogodek, nato iz wiznet registra preberemo na katerem socketu in kateri interrupt se je zgodil, ter sporočimo, da smo obdelali interrupt.

```
void EXTI0_IRQHandler(void) //EXTI0 ISR
{
  // cehck if EXTI line is asserted
  if(EXTI_GetITStatus(EXTI_Line0) != RESET)
  {
  //clear interrupt
  EXTI_ClearFlag(EXTI_Line0);
  locate_interrupt();
  }
}

void locate_interrupt()
{
  uint8_t code = 0x4;
  spi_read(W5200_IMR2, 1, &code);
  spi_read(W5200_Sn_IR(code), 1, &code);

  spi_dma_sendByte(W5200_Sn_IR(0x0), 0xff);
  switch (code )
  {
    case 0x1:
        // connection established
        // do nothing wait for input.
        break;
    case 0x2:
        break;
    case 0x4:
        // recived data resume task
        vTaskResume(set_macTaskHandle);
```

```
            break;
        default:
            break;
    }
}
```

5.3 spi.c

V tej datoteki je inicializacijska funkcija SPI, kjer definiramo vse pine, ki jih SPI uporablja. Definirani so interrupti in nekatere funkcije, ki se sprožijo ob interruptih. Poleg tega so v tej datoteki funkcije za branje registrov preko SPI vodila.