System on Chip Architecture
Lab 1 Report

Daniele Castro S253244
System-on-chip architecture
Politecnico of Turin

CONTENTS

LIST OF FIGURES

System on Chip Architecture
Lab 1 Report

***Abstract*—In this lab I have implemented a simple PSM (Power State Machine) using the MCU STM32F051R8 on the discovery board. The project,the way is made, is almost useless as it only gives us the possibility to develop simple functions to switch in various low power modes and reuse them in more useful projects. Ihave also made some measurements for each low power mode so that I can estimate the lowest power consumption for each state with every peripheral switched off since I have not switched on any of them.**

I. Introduction

I am going to explain in sequence
1) The PSM states
2) How the transition happens
3) The code I have written

II. Background

As described in the Reference Manual:
”The device features three low-power modes:

- Sleep mode (CPU clock off, all peripherals including Cortex®-M0 core peripherals like NVIC, SysTick, etc. are kept running)
- Stop mode (all clocks are stopped)
- Standby mode (1.8V domain powered-off)”

As described in the Assignment:

- ”SLEEP (WFI) mode is waken-up by any interrupt/event. In this case we use the user button to exit.
- STOP mode requires RTC by setting EXTI line 17 that is connected to the RTC. We use RTC as we do not have EXTI source connected.
- STANBY mode uses WKUP pin 1, that is PA.0 connected to the USER button. No RTC is used. GPIO configuration is present in SLEEP and STOP,it is not present in STANDBY mode because we have to reset after this state and we want a kind of stay off state. We need power just to handle any reset event. Wakeup source from Sleep and Stop mode is an EXTI Line configured in event mode. The wakeup source from Standby mode is the WKUP1 pin (PA0).”

III. Proposed Solution

Since it is not possible to switch between low power states without passing in ACTIVE mode, I have implemented the following PSM to switch between states. At power on the board waits for a user button pressure so that it can enter in the first low-power mode. Then, once entered in STOP mode, it waits for another user button pressure to exit from the current state. When exited, the board goes into sleep mode and exits from it after the RTC overflows. So we can finally go into standby mode. Here, the user button pressure will cause a reset of the board. Also note the switching order:

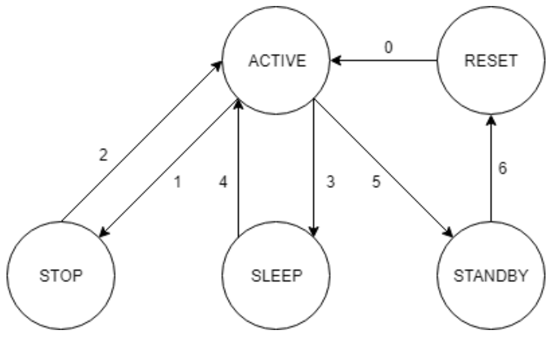


Fig. 1. Power State Machine

IV. Results and Discussion

Here I show some figures of measurements I have done during the lab sessions that display current consumption of each low power state and active state

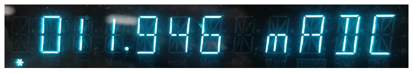


Fig. 2. Active mode power consumption

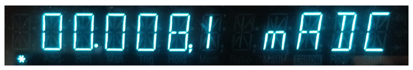


Fig. 3. Stop mode power consumption

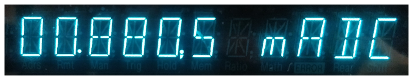


Fig. 4. Sleep mode power consumption

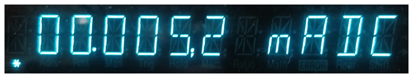


Fig. 5. Standby mode power consumption

As I did not switch on any peripheral, these results are good because they match values reported in the datasheet. Concerning the code implementation I have only commented out some `while();` loops at the end of the functions

```
void SleepMode_Measure(void);
void StopMode_Measure(void);
```

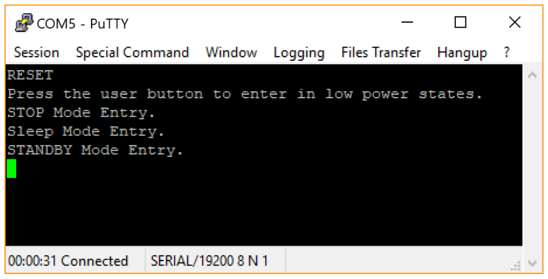
so that, after wake up, I can move into active state and I can move to the next low power state. I have also added:

```
void led_modes(uint8_t mode, uint8_t times);
void UART2_init(void);
```

so that, every time I switch between states, I can both flash green and blue LEDs on the board in four different patterns and I can also print, on the serial terminal, the specific name of the current state for debugging purposes. To do that I had to write down the low level implementations and declarations of

```
#ifdef __GNUC__
  /* With GCC/RAISONANCE, small printf (option LD
      Linker->Libraries->Small printf
     set to 'Yes') calls __io_putchar() */
  #define PUTCHAR_PROTOTYPE int __io_putchar(int
      ch)
#else
  #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE
      *f)
#endif /* __GNUC__ */

#ifdef __GNUC__
  /* With GCC/RAISONANCE, small printf (option LD
      Linker->Libraries->Small printf
     set to 'Yes') calls __io_putchar() */
  #define PUTCHAR_PROTOTYPE int __io_getchar(int
      ch)
#else
  #define GETCHAR_PROTOTYPE int fgetc(FILE *f)
#endif /* __GNUC__ */

...

PUTCHAR_PROTOTYPE
{
  /* fputc implementation: */
  /* write a character to the USART */
  USART_SendData(USART2, (uint8_t) ch);
  /* Loop until transmit data register is empty */
  while (USART_GetFlagStatus(USART2,
      USART_FLAG_TXE) == RESET)
  {}
  return ch;
}

GETCHAR_PROTOTYPE
{
        FILE *fd;
        char ch;
  /* fgetc implementation: */
  /* Loop until RXNE = 1 */
  while (USART_GetFlagStatus(USART2,
      USART_FLAG_RXNE) == RESET)
  {}
  /* read a character to the USART */
        ch = USART_ReceiveData(USART2);
        if(ch == '\r' || ch == '\n')    printf("\n
            \r");
        else fputc(ch, fd);
  return ch;
}
```

as a result I obtained on the serial console a full listing of what was currently happening on the board:


COM5 - PuTTY
Session Special Command Window Logging Files Transfer Hangup ?

```
RESET
Press the user button to enter in low power states.
STOP Mode Entry.
Sleep Mode Entry.
STANDBY Mode Entry.
```

00:00:31 Connected
SERIAL/19200 8 N 1

Fig. 6. Serial terminal window

Finally, here is the piece of code I have implemented to switch between all the states:

```
printf("STOP Mode Entry.\n\r");
led_modes(3, 4);
...
StopMode_Measure();

UART2_init();
printf("Sleep Mode Entry.\n\r");
led_modes(2, 4);
...
SleepMode_Measure();

UART2_init();
printf("STANDBY Mode Entry.\n\r");
led_modes(0, 4);
...
StandbyMode_Measure(); //here it blocks and resets
     if user button is pressed.

/*following code will never reached:*/
UART2_init();
printf("Exiting all of the modalities.\n\r");
led_modes(1, 4);
while(1);
```

```
void StopMode_Measure(void);
```

- configures RTC
- configures GPIO as analog to reduce current consumption
- Disable GPIOs clock
- configures EXTI
- configures NVIC
- enables RTC
- enters in STOP mode

Note that RTC (Real Time Clock) is clocked by a low power/frequency clock source called LSI that is an internal RC oscillator. It has also some registers called RCC that can control the clock source of this and other peripherals and save the current status when there is no power.

```
void SleepMode_Measure(void);
```

- simply disables GPIO clock
- configures user button in EXTI mode
- enters in SLEEP mode

```c
void StandbyMode_Measure(void);
```

- sets PIN 2 as wake up PIN
- enters in STANDBY mode

It is also important to notice that, has the name suggests, inside these functions

```c
void led_modes(uint8_t mode, uint8_t times);
void UART2_init(void);
```

I have initialized again the peripherals each time, expecially the GPIO as it is, all the times, disabled by the low-power switching functions I have mentioned above.
As shown in the comments, I have also put an unreachable piece of code, that explains the fact that after exiting from the last power mode (standby) we can only reset.

V. Conclusions

In this discussion I have accomplished the assignment requested and, as an extra, I have also properly configured the serial port to print debug messages and, eventually, get strings from the input even if I have not used it in this specific lab. I would like to reuse these functions in future labs.

References

[1] STM32F051R8 Reference Manual (RM0091), [pdf] Available at: Link.
Accessed on: Feb. 10, 2019.

[2] STM32F0DISCOVERY User Manual (UM1525), [pdf] Available at: Link.
Accessed on: Feb. 10, 2019.