Terraneo Federico fede.tft@hotmail.it

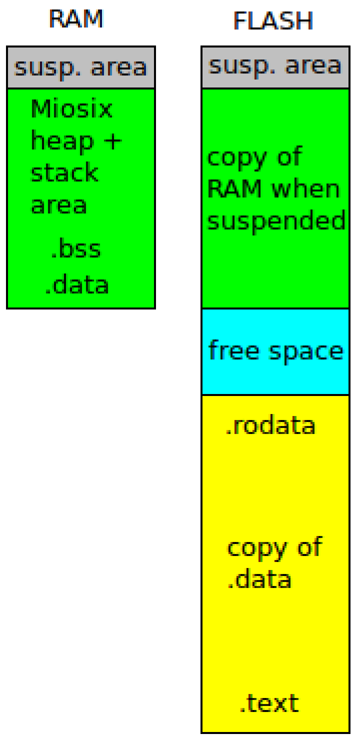
Suspend to FLASH support in Miosix

Revision 1.0

High level overview

- Upon request from the application code, the content of the RAM memory is copied to FLASH memory, and the microcontroller is placed in a deep sleep state. Power supply can also be removed.
- When a proper event is received (RESET pin, RTC interrupt) or, if power was removed, when power is applied back the device copies the saved state in RAM an restarts execution.

Memory layout



This is the typical memory layout of a Miosix application, except for two changes:

– a small portion at the top of the RAM memory, whose size is still to be determined, but is probably around 64Bytes is reserved. The reason for this will be explained below. This portion is called RAM suspend area.

– Starting from the top of the flash memory an area of the same size of the RAM memory (eventually rounded to an integer number of FLASH pages) is reserved. At the top of this area there is a portion called FLASH suspend area, of the same size of its RAM counterpart.

Problems

There are four problems in implementing suspend to FLASH:

1) At boot, Miosix must have a way to find if the last time the application was shut down or suspended. If the application was shut down, it should continue normal boot, while if the application was suspended it should resume the previous state.
2) During resume a function must copy the saved state from FLASH to RAM. However since all the RAM memory is overwritten in this step, also the activation record of the function that performs the copy would be overwritten, resulting in a crash.
3) The code to perform the suspend/resume should write to the FLASH as little as possible, to avoid wearing it out.
4) The state of an application not only includes the RAM, but also the state of the microcontroller's peripherals. A way to save/restore such state is needed.

Solution to problem 1: Select what to do at boot

The solution to this problem uses the FLASH suspend area. Since this RAM area is not used by the application there is no need to save it in FLASH. Instead it can contain the following data:

```
struct SuspendArea
{
    unsigned int flag;
    unsigned int saved_sp;
    unsigned int saved_lr;
    unsigned int saved_registers[REGISTER_NUM];
    char padding[PADDING_SIZE];//To reach size of the suspend area
};
```

The flag field could be used to understand if the last time the application was shut down it was due to a suspend or not. For example the special value 0x55aa55aa could be used to indicate that the application was suspended. If the boot code finds this value it should copy the saved state from FLASH to RAM, clear the last flash page so that it no longer contains the 0x55aa55aa flag and resume execution.

The additional saved_sp, saved_lr and saved_registers fields contains the saved stack pointer, link register and the value of registers so that the application can resume from the point where it was suspended. Since the suspend/resume code mimics a function call/return, there is no need to save registers r0-r3 which in the ARM ABI are used to pass parameters to a function. It is however necessary to save the CPSR register which contain the processor's status.

Solution to problem 2: Avoid trashing the activation record of the resume function

This solution uses the RAM suspend area.

The resume code can start checking the flag value. If it indicates that the application was suspended it should set the stack pointer to the top of the RAM suspend area, and should call the function to copy the FLASH back in RAM. Since this function does not write into the RAM suspend area, it will not overwrite its own activation record. At the end this function should set the stack pointer, link register and registers to the values saved in the struct SuspendArea. Therefore, by returning it will jump back to the place where the suspend function was called, resuming execution.

Solution to problem 3: Minimizing FLASH writes

All FLASH pages devoted to hold the copy of RAM but the one that contains the suspend area will obviously be written only once per suspend operation. This is the minimum possible to achieve suspend functionality.

The page that contains the suspend area however requires two writes, one at suspend to store the value 0x55aa55aa and the values of the link register and stack pointer, and one during resume to trash the value of the flag so that subsequent boot will not cause a resume. However, if the FLASH controller supports separate operation for FLASH erase and write, it is possible to perform the FLASH write during suspend, and the FLASH erase at reboot. The erase operation leaves the FLASH content to a value, usually 0xff but surely not 0x55aa55aa effectively trashing the flag value.

Solution to problem 4: Saving peripheral state

This still has to be addressed.

Draft implementation

The proposed implementation uses two functions for the suspend phase and two functions for the resume phase. They are called suspendStage1(), suspendStage2(), resumeStage1() and resumeStage2().

Their implementation is now outlined.

suspendStage1():

This function starts by calling eventual callbacks of code interested in suspend events, pausing the kernel and syncing the filesystem content.

It then disables interrupts to make sure the following piece of code is atomic.

Next, it calls suspendStage2()

During the call to suspendStage2() the system is suspended. When the system reboots it copies back the RAM value and resumes execution by returning from the suspendStage2() function.

Next it erases the FLASH page that contains the FLASH suspend area, to trash the value of the flag variable.

Code continues by enabling interrupts, restarting the kernel and calling eventual callbacks of code interested in resume events, and returning from the suspendStage1() function.

suspendStage2():

This function contains assembler code to save the content of registers, the stack pointer and the link register in the suspend area mapped to RAM. It also write the value 0x55aa55aa in its appropriate RAM location.

The stack pointer must be saved before it is decremented by the compiler to make room for the function's local variables.

To do so it is possible to use the C++'s behavior of scopes, like this:

```
void suspendStage2()
{
    //Put assembler code to save stack pointer here
    {
        //Declare variables here, in this inner scope
    }
}
```

In architectures that have a write buffer, such as the Cortex M3 a proper memory barrier operation must be inserted at this point to make sure the RAM content is consistent.

It then saves the RAM content to FLASH. The FLASH sector that contains the suspend area is written last, so that if power is removed during the suspend operation, at the next reboot the processor will not resume to an inconsistent state but rather it won't find the 0x55aa55aa value and will perform a clean boot.

The last part of the function contains code to put the microcontroller in low power mode.

This function shall never return.

resumeStage1():

This function is called early during the Miosix boot, so that interrupts are not yet enabled.

It checks if the suspend area mapped to FLASH contains the valid flag 0x55aa55aa.

If the flag is invalid, it returns so that normal boot can continue.

If the flag is found, some assembler code to set the stack pointer to the top of the RAM suspend area is executed, and then the resumeStage2() function is called. By doing that the resmeStage2() function will have its activation record in the RAM suspend area.

The call to resumeStage2() never returns back into the resumeStage1() function. Instead it returns into the suspended code.

resumeStage2():

This function copies the FLASH memory part reserved as copy of RAM back into RAM. However, it does not overwrite the RAM suspend area.

It then restores the value of registers copying them from the FLASH suspend area.

Care must be taken to restore the stack pointer after the compiler decrements to deallocate local variables.

To do so it is possible to use the C++'s behavior of scopes, like this:

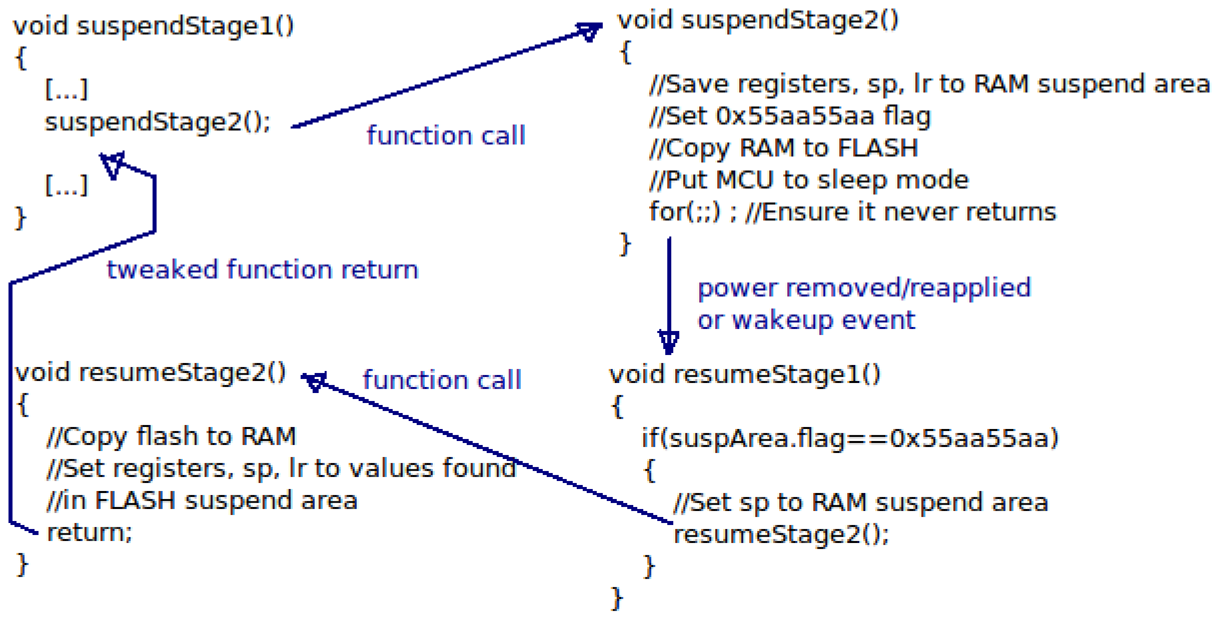
```
void suspendStage2()
{
    {
        //Declare variables here, in this inner scope
    }
    //Put assembler code restore stack pointer here
}
```

In architectures that have a write buffer a proper memory barrier operation must be inserted at this point to make sure the RAM content is consistent.

Now that the link register and stack pointer are restored to the saved values, the function returns into the suspended code.

Code flow

Since the code flow is unusual due to link register trickery, this image tries to explain the expected code flow.



```
void suspendStage1()
{
   [...]
   suspendStage2();

   [...]
}
```

```
void suspendStage2()
{
   //Save registers, sp, lr to RAM suspend area
   //Set 0x55aa55aa flag
   //Copy RAM to FLASH
   //Put MCU to sleep mode
   for(;;) ; //Ensure it never returns
}
```

```
void resumeStage2()
{
   //Copy flash to RAM
   //Set registers, sp, lr to values found
   //in FLASH suspend area
   return;
}
```

```
void resumeStage1()
{
   if(suspArea.flag==0x55aa55aa)
   {
      //Set sp to RAM suspend area
      resumeStage2();
   }
}
```