Smart sensing daemon for Miosix
Design of an energy efficient sensor daemon for Miosix

Rizzi Alessandro Maria
Matr. 783504, (alessandromaria.rizzi@mail.polimi.it)

Report for the master course of Real Time Operating Systems
Reviser: PhD. Patrick Bellasi (bellasi@elet.polimi.it)

Received: March, 05 2013

Abstract

The goal of the project is to design and implement a daemon for Miosix embedded OS performing the "smart-sensing", which is a new power-efficient way of performing a series of reads from a sensor spaced out by a fixed amount of time. The difference between this method and a normal data acquisition from the sensor lies in the fact that the board remains in suspension most of the time, waking up in a low-consumption mode only to read sensor data.

1 Introduction

In context of wireless sensor network great attention is given to energy-efficiency issues.
The reason for this is the fact that every node of the network must operate continuously for a long time period of time without any physical maintenance, which would be impractical due to the large number of nodes and their barely reachable location.
Besides, the main purpose af a WSN is the monitoring of some quantities acquired by some sensors.
From these two general properties of WSN it's clear that data acquisition from sensors is a key activity for such a network and should be optimized as much as possible on the energy-efficiency aspect.
We can imagine that on each node of the WSN will be executed different task. The major part of the task would be devoted to data acquisition and elaboration which means that will be busy for a certain amount of time perfoming some reads from a sensor and the spending some time in elaborating the data.
It would be reasonable suppose that the node would spend most of the time in waiting the data from the sensors. In the most simple implementation the board would remain always active, maybe just reducing the microcontroller consumption.
Otherwise it's possible to improve the energy consumption of the node first of all by making the board be suspended if there are no active task. The suspension mode is a special mode which is present in most of microcontroller platforms which has very little energy consumption: both the microcontroller and the memory are not powered.
Then it's possible to improve further the energy-efficiency by performing the requested reads in a special low-consumption mode.
The project aims to the development of the complete infrastracture for this activity, starting from an operating system which is able to run multiple task and to suspend/resume them.
The rest of the section is divided as follows: in subsection 1.1 is presented the Miosix Embedded OS; in subsection 1.2 the board used.

1.1 Miosix Embedded OS

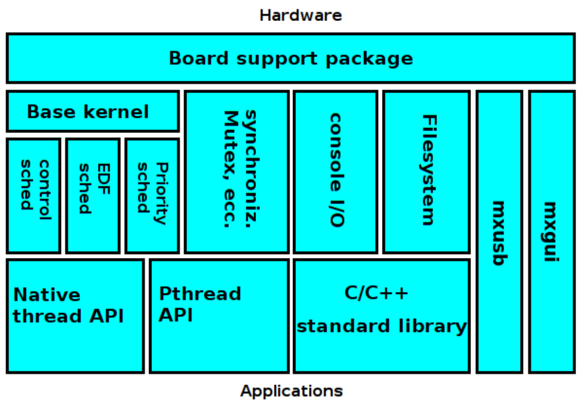


Figure 1: Miosix kernel architecture.

Miosix is a kernel for microcontrollers. Its goals are to provide an environment as much "standard compliant" as possible (in which developing application for an embedded system isn't much different than developing a standard desktop application) and avoiding performance or code-footprint penalties for the unused features. It provides:

- Multithreading with a pthread-like API
- C and C++ support including standard C and STL libraries.

- Device drivers for most complex devices
- Filesystem support with POSIX-like API
- Different scheduler to be chosen

It supports only 32bit microcontrollers and has an experimental support for multitasking which lacks some features (like dynamic-loading of executables from the filesystem). Its basic architecture is presented in **Figure 1**.
For this project has been particularly important some recent additions to it, which are multitasking with the possibility of suspending the system when there are no active tasks.

1.2 The stm3220g-eval board

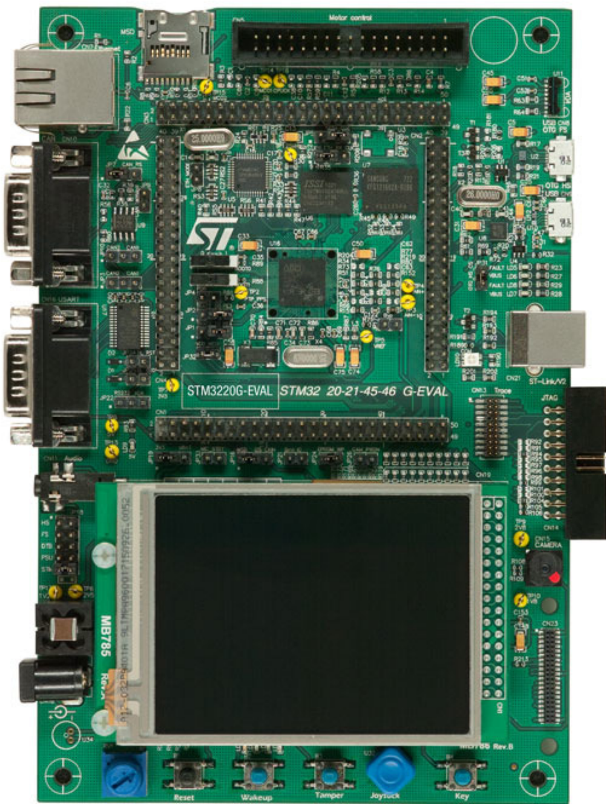


Figure 2: Picture of the STM stm3220g-eval board.

The board used in the project is the ST Microelectronics stm3220g-eval (**Figure 2**).
It has a 120MHz Cortex-M3 microcontroller (STM32F207IG), which has 1MiB of FLASH, 128KiB of RAM and 2MiB of external SRAM.
The board is pretty rich in terms of hardware features, including an USB port, an RS-232 port, an audio DAC, a 320x240 3,2" LCD display, a MicroSD card slot, 4 LEDs, a 4-direction joystick, an RTC with backup battery and a potentiometer (which has been used as source of acquisition data).
In addition were added 128KiB of MRAM in order to save the processes during suspension.

In the development of the project the board has been flashed through the USB port, which has been used also as a serial console. Due to the lack of a JTAG in-circuit debugger, this feature has not been used in the development phase.

2 ADC driver

The first step in the development of the project has been the implementation of a module for Miosix capable of perform reads from a sensor.
This operation has been implemented as performing analog reads from a specific GPIO pin.
In particular the module has to be able to perform the following operations:

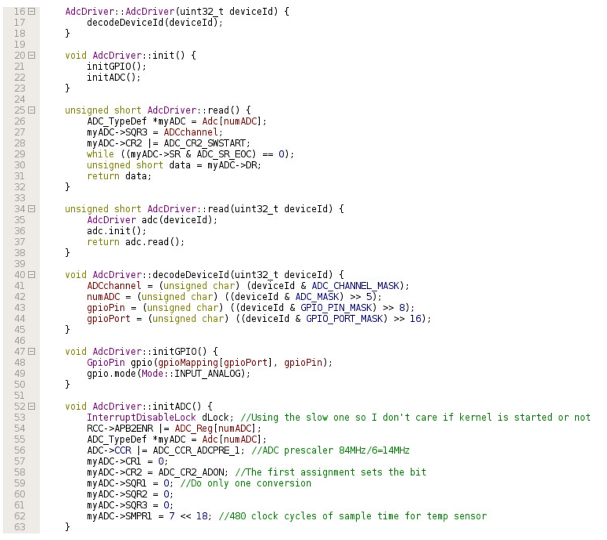
- Initialize a specific GPIO pin on a specific GPIO port to perform analog reads
- Initialize a specific ADC to perform an analog read
- Perform a single read on a given ADC channel

For the specific application context (WSN) characterized by a very low frequency of reads, high performances are not required.
So it's no need of using DMA. In fact also the initialization phase is repeated at every read.
One problem arised regarding the way of determinating the following informations of a given input: GPIO pin, GPIO port, ADC to be used, ADC channel connected to the GPIO port. This informations are not easily derivable from the other ones (it's necessary to look at the reference manual to obtain the correct mapping informations). Besides, while the ADC mechanism is quite general in all the ST Microelectronics' microcontrollers family (it has been succesfully tested in STM32F4 Discovery board), the mapping between GPIO pin, ADC and ADC channel are subject to change on different boards.
So the solution adopted has been to pack all the required informations into a 32bit integer as following:

- bits 0-4: ADC channel (0 the first, 1 the second, etc...)
- bits 5-7: ADC (0 the first, 1 the second, etc...)
- bits 8-15: GPIO pin
- bits 16-19: GPIO port (A=0, B=1, C=2, etc...)
- bits 20-31: Not used

In this way an input is identified by a single 32 bit integer.
The input used in the project test has been the potentiometer: this because it has many possible values and it's easy to modify it and check the correctness.
The relevant part of implementation is shown in **Figure 3**. In particular the init method initialize both the GPIO and ADC as shown. There are two version of read the

first, which accepts no parameters, is meanto for performing a read from an already initialized object. The second one, which requires the deviceId, is a static method which perform all the require operations (initilization, read) in a single step and is the only used outside the class.



```
    AdcDriver::AdcDriver(uint32_t deviceId) {
        decodeDeviceId(deviceId);
    }

    void AdcDriver::init() {
        initGPIO();
        initADC();
    }

    unsigned short AdcDriver::read() {
        ADC_TypeDef *myADC = Adc[numADC];
        myADC->SQR3 = ADCchannel;
        myADC->CR2 |= ADC_CR2_SWSTART;
        while ((myADC->SR & ADC_SR_EOC) == 0);
        unsigned short data = myADC->DR;
        return data;
    }

    unsigned short AdcDriver::read(uint32_t deviceId) {
        AdcDriver adc(deviceId);
        adc.init();
        return adc.read();
    }

    void AdcDriver::decodeDeviceId(uint32_t deviceId) {
        ADCchannel = (unsigned char) (deviceId & ADC_CHANNEL_MASK);
        numADC = (unsigned char) ((deviceId & ADC_MASK) >> 5);
        gpioPin = (unsigned char) ((deviceId & GPIO_PIN_MASK) >> 8);
        gpioPort = (unsigned char) ((deviceId & GPIO_PORT_MASK) >> 16);
    }

    void AdcDriver::initGPIO() {
        GpioPin gpio(gpioMapping[gpioPort], gpioPin);
        gpio.mode(Mode::INPUT_ANALOG);
    }

    void AdcDriver::initADC() {
        InterruptDisableLock dLock; //Using the slow one so I don't care if kernel is started or not
        RCC->APB2ENR |= ADC_Reg[numADC];
        ADC_TypeDef *myADC = Adc[numADC];
        ADC->CCR |= ADC_CCR_ADCPRE_1; //ADC prescaler 84MHz/6=14MHz
        myADC->CR1 = 0;
        myADC->CR2 = ADC_CR2_ADON; //The first assignment sets the bit
        myADC->SQR1 = 0; //Do only one conversion
        myADC->SQR2 = 0;
        myADC->SQR3 = 0;
        myADC->SMPR1 = 7 << 18; //480 clock cycles of sample time for temp sensor
    }
```

Figure 3: Extract from ADC driver implementation

3 SmartSensing module

This part is the core of the project.
It has the task of keep track of the data requests and schedule them. It's main idea is quite simple: it keeps track of the request of an acquisition job and schedule the correct time to perform the required reads either by scheduling a new wake up or by the kernel daemon if they occour when the system is already running.
The module can be divided in the following conceptual part:

- Data structures
- Reqest registration and query
- Kernel daemon

In the rest part of this section we analyze these different conceptual parts plus the integration with the rest of the kernel from the module point of view.

3.1 Data structures

In this subsection are described the data structures utilized and the methods which operate upon them.
The basic idea it to subdivide the data requested in two sets: one with the data which are in common with all the acquisiton jobs, and one with the group of data of each job.
Since the data must survive the suspension process during which all the RAM content is lost, it is allocated at the end of the backup SRAM, which is maintained during the whole process, as shown in **Figure 4**. The reason for this allocation is to minimize the compatibility issues with Miosix.

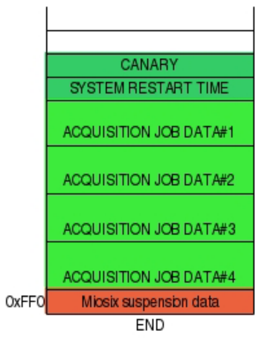
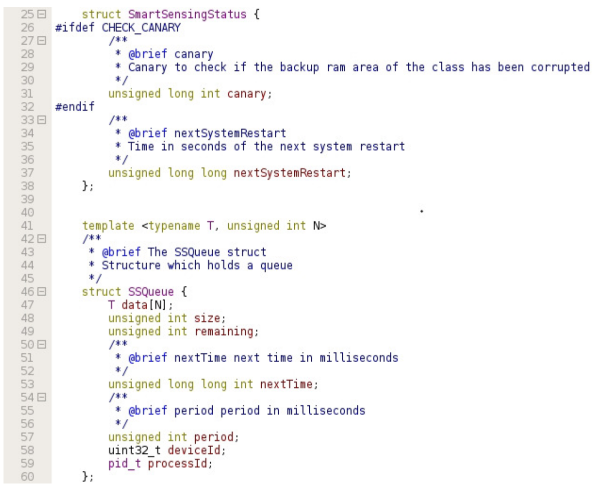


Figure 4: Structure of the end of backup SRAM

The data is disposed as follows: first the general data, which is formed by a canary, used to check if the other data structures haven't been overwritten by other data stored in the backup SRAM, and an integer which store the next system full boot time in milliseconds. Then there is a fixed-size array of the data of each job. This choice has been done due to the extremely low number of allowed jobs which are a consequece of the low number of tasks allowed (just 4).
More precisely the job acquisition data is composed of a fixed-size array which holds the data acquired, the number of total samples requested at the job, the number of remaining acquisition, the time in milliseconds when the next acquisition has to be taken, the space in milliseconds between two acquisitions, the device identifier and the id of the process which has requested the job. Each array element is referred from now on in this report and in the code as "queue" because it behaves like a queue which mantains the data coming from the sensor.



```
    struct SmartSensingStatus {
#ifdef CHECK_CANARY
        /**
         * @brief canary
         * Canary to check if the backup ram area of the class has been corrupted
         */
        unsigned long int canary;
#endif
        /**
         * @brief nextSystemRestart
         * Time in seconds of the next system restart
         */
        unsigned long long nextSystemRestart;
    };


    template <typename T, unsigned int N>
    /**
     * @brief The SSQueue struct
     * Structure which holds a queue
     */
    struct SSQueue {
        T data[N];
        unsigned int size;
        unsigned int remaining;
        /**
         * @brief nextTime next time in milliseconds
         */
        unsigned long long int nextTime;
        /**
         * @brief period period in milliseconds
         */
        unsigned int period;
        uint32_t deviceId;
        pid_t processId;
    };
```

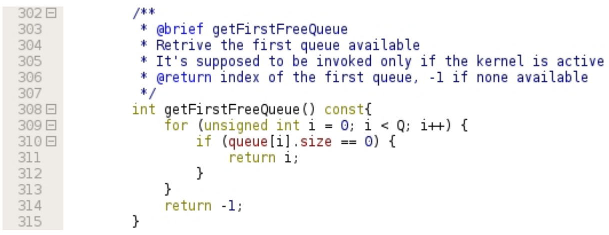
Figure 5: Data structures

Due to the extremely low size of the number of jobs, the

algorith implemented for searching values into the array is just an exaustive scan.
The methods for access and manipulate the queues can be subdivided in the following categories:

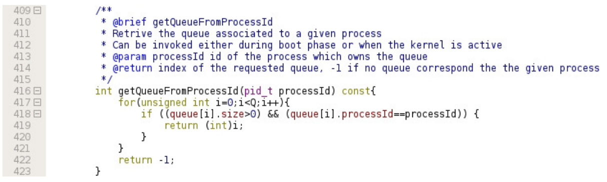
- Methods for retrive a single queue which match a given parameter/condition

  In this category there are getFirstFreeQueue (**Figure 6**) which retrive the first unused queue and getQueueFromProcessId (**Figure 7**) which retrive the index of the first queue which belongs to a process with the given id.



```
        /**
         * @brief getFirstFreeQueue
         * Retrive the first queue available
         * It's supposed to be invoked only if the kernel is active
         * @return index of the first queue, -1 if none available
         */
        int getFirstFreeQueue() const{
            for (unsigned int i = 0; i < Q; i++) {
                if (queue[i].size == 0) {
                    return i;
                }
            }
            return -1;
        }
```

Figure 6: getFirstFreeQueue method

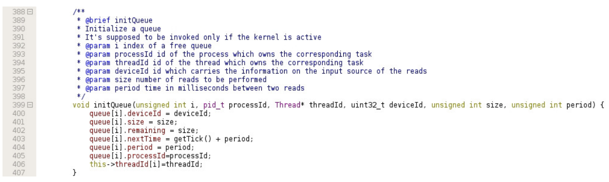


```
        /**
         * @brief getQueueFromProcessId
         * Retrive the queue associated to a given process
         * Can be invoked either during boot phase or when the kernel is active
         * @param processId id of the process which owns the queue
         * @return index of the requested queue, -1 if no queue correspond the the given process
         */
        int getQueueFromProcessId(pid_t processId) const{
            for(unsigned int i=0;i<Q;i++){
                if ((queue[i].size>0) && (queue[i].processId==processId)) {
                    return (int)i;
                }
            }
            return -1;
        }
```

Figure 7: getQueueFromProcessId method

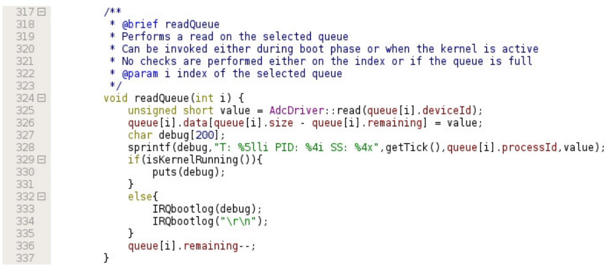
- Methods performing operations on a single queue

  Here there are initQueue (**Figure 8**), readQueue (**Figure 9**) and resetQueue (**Figure 10**), which respectively initalize a queue with the given data, add to a queue a new read from its associated input; and set a queue not utilized.



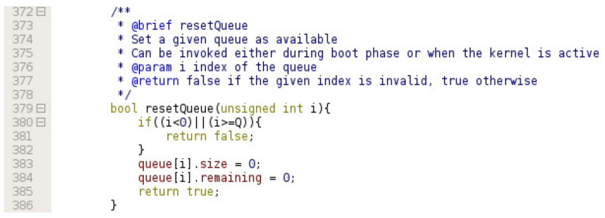
```
        /**
         * @brief initQueue
         * Initialize a queue
         * It's supposed to be invoked only if the kernel is active
         * @param i index of a free queue
         * @param processId id of the process which owns the corresponding task
         * @param threadId id of the thread which owns the corresponding task
         * @param deviceId id which carries the information on the input source of the reads
         * @param size number of reads to be performed
         * @param period time in milliseconds between two reads
         */
        void initQueue(unsigned int i, pid_t processId, Thread* threadId, uint32_t deviceId, unsigned int size, unsigned int period) {
            queue[i].deviceId = deviceId;
            queue[i].size = size;
            queue[i].remaining = size;
            queue[i].nextTime = getTick() + period;
            queue[i].period = period;
            queue[i].processId=processId;
            this->threadId[i]=threadId;
        }
```

Figure 8: initQueue method



```
        /**
         * @brief readQueue
         * Performs a read on the selected queue
         * Can be invoked either during boot phase or when the kernel is active
         * No checks are performed either on the index or if the queue is full
         * @param i index of the selected queue
         */
        void readQueue(int i) {
            unsigned short value = AdcDriver::read(queue[i].deviceId);
            queue[i].data[queue[i].size - queue[i].remaining] = value;
            char debug[200];
            sprintf(debug,"T: %5lli PID: %4i SS: %4x",getTick(),queue[i].processId,value);
            if(isKernelRunning()){
                puts(debug);
            }
            else{
                IRQbootlog(debug);
                IRQbootlog("\r\n");
            }
            queue[i].remaining--;
        }
```

Figure 9: readQueue method

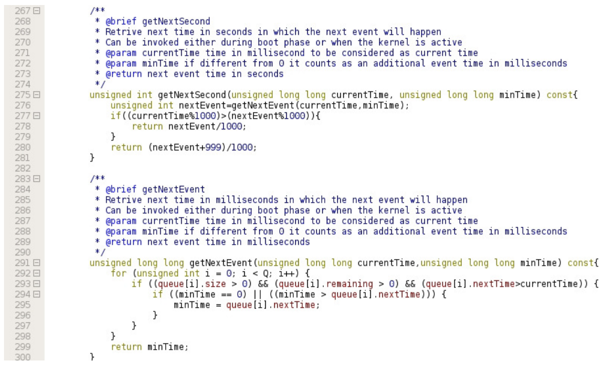


```
        /**
         * @brief resetQueue
         * Set a given queue as available
         * Can be invoked either during boot phase or when the kernel is active
         * @param i index of the queue
         * @return false if the given index is invalid, true otherwise
         */
        bool resetQueue(unsigned int i){
            if((i<0)||(i>=Q)){
                return false;
            }
            queue[i].size = 0;
            queue[i].remaining = 0;
            return true;
        }
```

Figure 10: resetQueue method

- Methods for extract timing information from the whole set of queues

  We consider the methods getNextEvent and getNextSecond (**Figure 11**) which given the current time and optionally an event time return the first occurrency in the future which is either a queue read event or the event given, respectively in millisecond and second. In particular the latter has been designed to cope with the different granularity of time, which is potentially problematic.



```
        /**
         * @brief getNextSecond
         * Retrive next time in seconds in which the next event will happen
         * Can be invoked either during boot phase or when the kernel is active
         * @param currentTime time in millisecond to be considered as current time
         * @param minTime if different from 0 it counts as an additional event time in milliseconds
         * @return next event time in seconds
         */
        unsigned int getNextSecond(unsigned long long currentTime, unsigned long long minTime) const{
            unsigned int nextEvent=getNextEvent(currentTime,minTime);
            if((currentTime%1000)>(nextEvent%1000)){
                return nextEvent/1000;
            }
            return (nextEvent+999)/1000;
        }

        /**
         * @brief getNextEvent
         * Retrive next time in milliseconds in which the next event will happen
         * Can be invoked either during boot phase or when the kernel is active
         * @param currentTime time in millisecond to be considered as current time
         * @param minTime if different from 0 it counts as an additional event time in milliseconds
         * @return next event time in milliseconds
         */
        unsigned long long getNextEvent(unsigned long long currentTime,unsigned long long minTime) const{
            for (unsigned int i = 0; i < Q; i++) {
                if ((queue[i].size > 0) && (queue[i].remaining > 0) && (queue[i].nextTime>currentTime)) {
                    if ((minTime == 0) || ((minTime > queue[i].nextTime))) {
                        minTime = queue[i].nextTime;
                    }
                }
            }
            return minTime;
        }
```

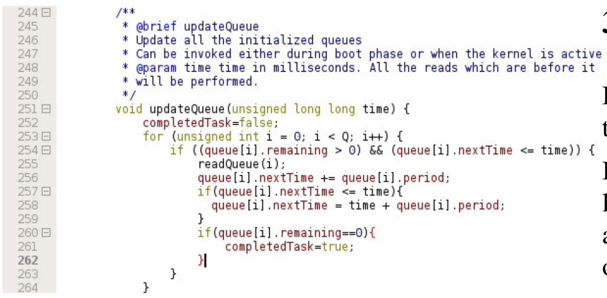
Figure 11: getNextEvent and getNextSecond methods

- Methods performing operations on the entire set of queues

  Here there are methods which scans all the queue checking if they match a condition and, if so, perform an action on the selected queue.

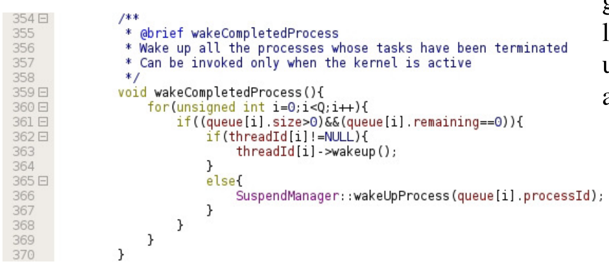
  The method updateQueue (**Figure 12**), given the current time in milliseconds, apply the readQueue method to all those queues which require a new read.

  The method wakeCompletedProcess (**Figure 13**) wakes up all the processes whose queue requires no further reads.



```
/**
 * @brief updateQueue
 * Update all the initialized queues
 * Can be invoked either during boot phase or when the kernel is active
 * @param time time in milliseconds. All the reads which are before it
 * will be performed.
 */
void updateQueue(unsigned long long time) {
    completedTask=false;
    for (unsigned int i = 0; i < Q; i++) {
        if ((queue[i].remaining > 0) && (queue[i].nextTime <= time)) {
            readQueue(i);
            queue[i].nextTime += queue[i].period;
            if(queue[i].nextTime <= time){
              queue[i].nextTime = time + queue[i].period;
            }
            if(queue[i].remaining==0){
                completedTask=true;
            }
        }
    }
```

Figure 12: updateQueue method



```
/**
 * @brief wakeCompletedProcess
 * Wake up all the processes whose tasks have been terminated
 * Can be invoked only when the kernel is active
 */
void wakeCompletedProcess(){
    for(unsigned int i=0;i<Q;i++){
        if((queue[i].size>0)&&(queue[i].remaining==0)){
            if(threadId[i]!=NULL){
                threadId[i]->wakeup();
            }
            else{
                SuspendManager::wakeUpProcess(queue[i].processId);
            }
        }
    }
}
```

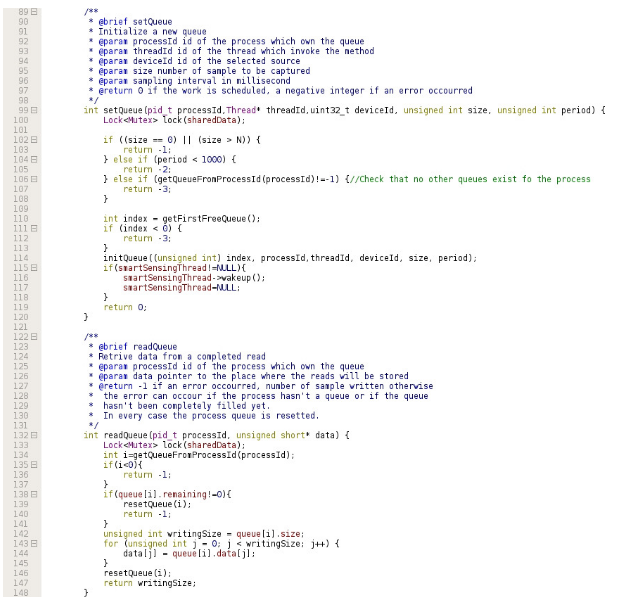
Figure 13: wakeCompletedProcess method

3.2 Request registration and query

In this subsection we discuss about the methods used by the rest of the operating system to interact with the module to allocate a new acquisition job or retrive the data of a completed one (**Figure 14**).

The former (setQueue) just initializes a new jobs after checking that the provided data are correct.

The latter (readQueue) retrives the reads from a completed job and frees the place in the array occupied by it.



```
/**
 * @brief setQueue
 * Initialize a new queue
 * @param processId id of the process which own the queue
 * @param threadId id of the thread which invoke the method
 * @param deviceId id of the selected source
 * @param size number of sample to be captured
 * @param sampling interval in millisecond
 * @return 0 if the work is scheduled, a negative integer if an error occourred
 */
int setQueue(pid_t processId,Thread* threadId,uint32_t deviceId, unsigned int size, unsigned int period) {
    Lock<Mutex> lock(sharedData);

    if ((size == 0) || (size > N)) {
        return -1;
    } else if (period < 1000) {
        return -2;
    } else if (getQueueFromProcessId(processId)!=-1) {//Check that no other queues exist fo the process
        return -3;
    }

    int index = getFirstFreeQueue();
    if (index < 0) {
        return -3;
    }
    initQueue((unsigned int) index, processId,threadId, deviceId, size, period);
    if(smartSensingThread!=NULL){
        smartSensingThread->wakeup();
        smartSensingThread=NULL;
    }
    return 0;
}

/**
 * @brief readQueue
 * Retrive data from a completed read
 * @param processId id of the process which own the queue
 * @param data pointer to the place where the reads will be stored
 * @return -1 if an error occourred, number of sample written otherwise
 *  the error can occour if the process hasn't a queue or if the queue
 *  hasn't been completely filled yet.
 *  In every case the process queue is resetted.
 */
int readQueue(pid_t processId, unsigned short* data) {
    Lock<Mutex> lock(sharedData);
    int i=getQueueFromProcessId(processId);
    if(i<0){
        return -1;
    }
    if(queue[i].remaining!=0){
        resetQueue(i);
        return -1;
    }
    unsigned int writingSize = queue[i].size;
    for (unsigned int j = 0; j < writingSize; j++) {
        data[j] = queue[i].data[j];
    }
    resetQueue(i);
    return writingSize;
}
```

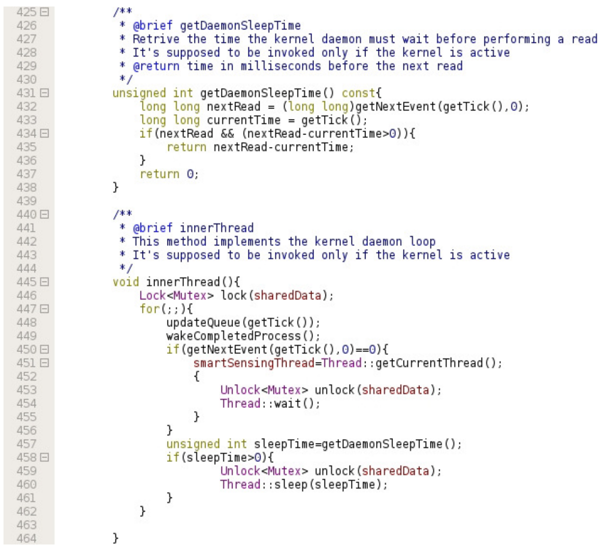
Figure 14: setQueue and readQueue methods

3.3 Kernel daemon

It has the task of performing reads which can occour when the kernel is active.

It is simply composed of a thread (**Figure 15**) which delays itself on the next acquisition time, performs the read and so on. If there are no active jobs it just wait until a new one is created.

It's interesting to note the peculiar usage of C++ RAII design pattern in the lock and unlock mechanisms: a mutex lock (or unlock) is associated to an object. When it goes out of scope, the mutex is automatically unlocked (or locked); otherwise i.e. the lack of a mutex lock after the unlock at line 453 appears wrong. This usage is encouraged by the Miosix design.



```
425     /**
426      * @brief getDaemonSleepTime
427      * Retrive the time the kernel daemon must wait before performing a read
428      * It's supposed to be invoked only if the kernel is active
429      * @return time in milliseconds before the next read
430      */
431     unsigned int getDaemonSleepTime() const{
432         long long nextRead = (long long)getNextEvent(getTick(),0);
433         long long currentTime = getTick();
434         if(nextRead && (nextRead-currentTime>0)){
435             return nextRead-currentTime;
436         }
437         return 0;
438     }
439
440     /**
441      * @brief innerThread
442      * This method implements the kernel daemon loop
443      * It's supposed to be invoked only if the kernel is active
444      */
445     void innerThread(){
446         Lock<Mutex> lock(sharedData);
447         for(;;){
448             updateQueue(getTick());
449             wakeCompletedProcess();
450             if(getNextEvent(getTick(),0)==0){
451                 smartSensingThread=Thread::getCurrentThread();
452                 {
453                     Unlock<Mutex> unlock(sharedData);
454                     Thread::wait();
455                 }
456             }
457             unsigned int sleepTime=getDaemonSleepTime();
458             if(sleepTime>0){
459                     Unlock<Mutex> unlock(sharedData);
460                     Thread::sleep(sleepTime);
461             }
462         }
463
464     }
```

Figure 15: kernel daemon thread

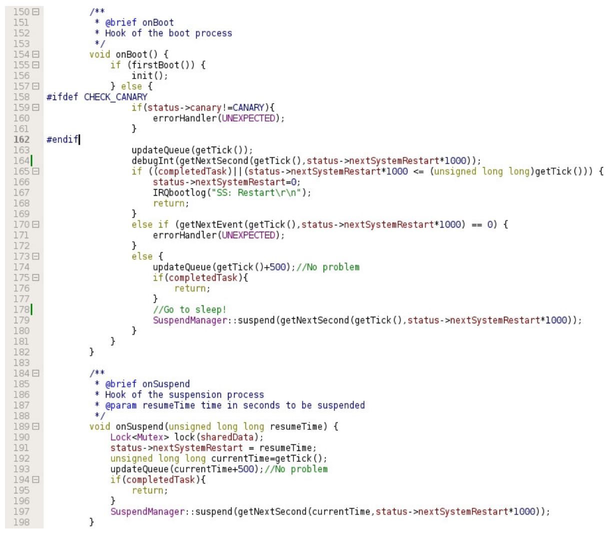
3.4 System integration

In this subsection we discuss the system integration from the smart sensing module point of view. We defer to the next section the description of how the whole system interact with the module.

Basically it's composed by the methods onBoot and onSuspend (**Figure 16**).

The onBoot method is executed before that the system is fully started. It has the task of perform the required reads and either continue the boot process if a full start is required (or is the first boot of the board) or reschedule the next wakeup and suspend the board.

The onSuspend method instead deals with the definition of the next wake up time that must include the ones required for the scheduled reads.



```
        /**
         * @brief onBoot
         * Hook of the boot process
         */
        void onBoot() {
            if (firstBoot()) {
                init();
            } else {
#ifdef CHECK_CANARY
                if(status->canary!=CANARY){
                    errorHandler(UNEXPECTED);
                }
#endif
                updateQueue(getTick());
                debugInt(getNextSecond(getTick(),status->nextSystemRestart*1000));
                if ((completedTask)||(status->nextSystemRestart*1000 <= (unsigned long long)getTick())) {
                    status->nextSystemRestart=0;
                    IRQbootlog("SS: Restart\r\n");
                    return;
                }
                else if (getNextEvent(getTick(),status->nextSystemRestart*1000) == 0) {
                    errorHandler(UNEXPECTED);
                }
                else {
                    updateQueue(getTick()+500);//No problem
                    if(completedTask){
                        return;
                    }
                    //Go to sleep!
                    SuspendManager::suspend(getNextSecond(getTick(),status->nextSystemRestart*1000));
                }
            }
        }

        /**
         * @brief onSuspend
         * Hook of the suspension process
         * @param resumeTime time in seconds to be suspended
         */
        void onSuspend(unsigned long long resumeTime) {
            Lock<Mutex> lock(sharedData);
            status->nextSystemRestart = resumeTime;
            unsigned long long currentTime=getTick();
            updateQueue(currentTime+500);//No problem
            if(completedTask){
                return;
            }
            SuspendManager::suspend(getNextSecond(currentTime,status->nextSystemRestart*1000));
        }
```

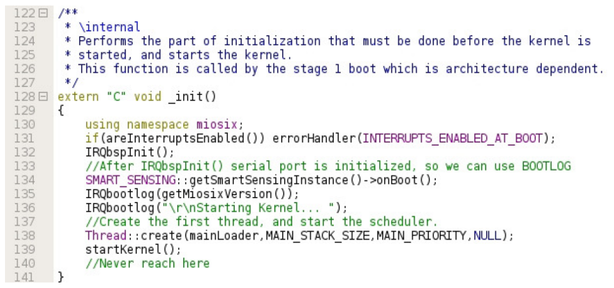
Figure 16: onBoot and onSuspend methods

4 System integration

In this section is described how the new module is integrated into Miosix, in particular in three phase: the boot, the suspension and the process syscall.

4.1 Boot integration

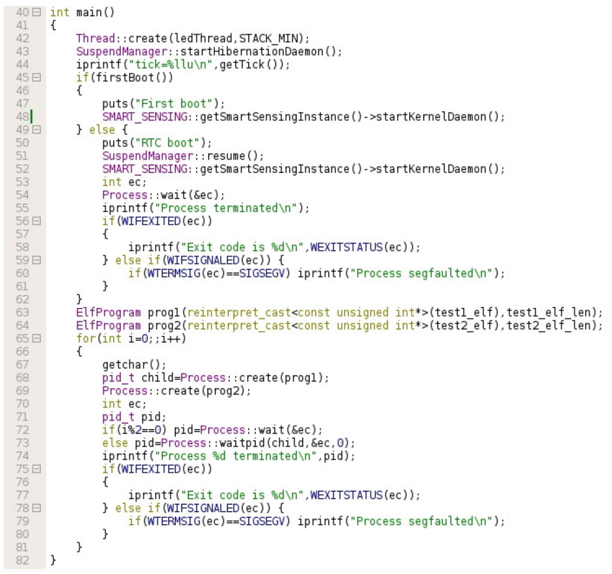
The boot integration is performed by placing an hook in the boot process of Miosix (**Figure 17**) just before that the kernel is started with the call of onBoot, which has been previously presented, at line 134.



```
122 /**
123  * \internal
124  * Performs the part of initialization that must be done before the kernel is
125  * started, and starts the kernel.
126  * This function is called by the stage 1 boot which is architecture dependent.
127  */
128 extern "C" void _init()
129 {
130     using namespace miosix;
131     if(areInterruptsEnabled()) errorHandler(INTERRUPTS_ENABLED_AT_BOOT);
132     IRQbspInit();
133     //After IRQbspInit() serial port is initialized, so we can use BOOTLOG
134     SMART_SENSING::getSmartSensingInstance()->onBoot();
135     IRQbootlog(getMiosixVersion());
136     IRQbootlog("\r\nStarting Kernel... ");
137     //Create the first thread, and start the scheduler.
138     Thread::create(mainLoader,MAIN_STACK_SIZE,MAIN_PRIORITY,NULL);
139     startKernel();
140     //Never reach here
141 }
```

Figure 17: _init procedure

Moreover has been modified the main procedure (**Figure 18**), called after the completion of the kernel boot process. This has been done in order to start the daemon which performs the reads occourring when the system is active. The modifications are the addition of call to startKernelDaemon at lines 48 and 52.



```
40 int main()
41 {
42     Thread::create(ledThread,STACK_MIN);
43     SuspendManager::startHibernationDaemon();
44     iprintf("tick=%llu\n",getTick());
45     if(firstBoot())
46     {
47         puts("First boot");
48         SMART_SENSING::getSmartSensingInstance()->startKernelDaemon();
49     } else {
50         puts("RTC boot");
51         SuspendManager::resume();
52         SMART_SENSING::getSmartSensingInstance()->startKernelDaemon();
53         int ec;
54         Process::wait(&ec);
55         iprintf("Process terminated\n");
56         if(WIFEXITED(ec))
57         {
58             iprintf("Exit code is %d\n",WEXITSTATUS(ec));
59         } else if(WIFSIGNALED(ec)) {
60             if(WTERMSIG(ec)==SIGSEGV) iprintf("Process segfaulted\n");
61         }
62     }
63     ElfProgram prog1(reinterpret_cast<const unsigned int*>(test1_elf),test1_elf_len);
64     ElfProgram prog2(reinterpret_cast<const unsigned int*>(test2_elf),test2_elf_len);
65     for(int i=0;;i++)
66     {
67         getchar();
68         pid_t child=Process::create(prog1);
69         Process::create(prog2);
70         int ec;
71         pid_t pid;
72         if(i%2==0) pid=Process::wait(&ec);
73         else pid=Process::waitpid(child,&ec,0);
74         iprintf("Process %d terminated\n",pid);
75         if(WIFEXITED(ec))
76         {
77             iprintf("Exit code is %d\n",WEXITSTATUS(ec));
78         } else if(WIFSIGNALED(ec)) {
79             if(WTERMSIG(ec)==SIGSEGV) iprintf("Process segfaulted\n");
80         }
81     }
82 }
```

Figure 18: main procedure

4.2 Suspension integration

This part is particularly important because it coordinates the suspension manager of Miosix and our smart sensing module.

The suspension manager has the task of keeping track of which processes are active and if there are no active task it can decide to put the system into suspesion.

It's important to specify that at the moment all the processes which are "suspension-ready" (the ones who are in a state where can be useful to suspend the system) have an associated resume time.

The only syscall that allows the system suspension is the sleep one, mainly because it's the only syscall during which the system can really do nothing; while in all other cases the process is waiting for something carried out by another process/the operating system itself, which requres the operating system being active (as we see later this project have caused further modifications reagarding this aspect).

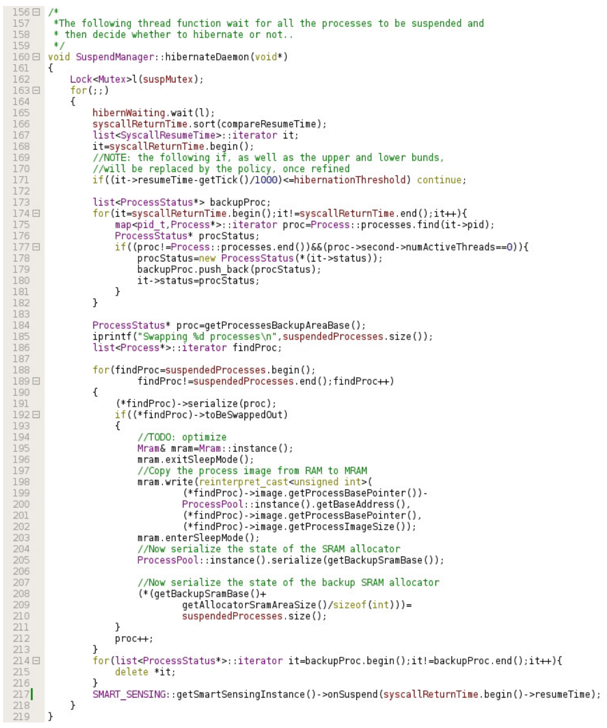
So if there are no working processes and the suspension manager decides to put the system into suspension (this happens in order to avoid the suspension process overhead when the time to be spent in suspension is too little) it saves the status of all the processes into the MRAM/backup SRAM and then the system is put into suspension.

This task is performed, in particular, by the hibernateDaemon procedure (**Figure 19**) which is run in a separate thread.

More precisely the last part of it (which sets the suspension time and actually performs the suspension) has been divided in the suspend method (**Figure 20**). In its stead has been put a call to the onSuspend method (line 217), which has been previously discussed.

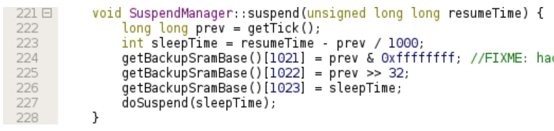
This has been done in order to allow the smart sensiong module to change the suspension time.

In addition this was necessary because the suspend method is called also by the onBoot method in smart sensing module, which has the necessity of performing a new suspension after that some reads were carried out. In this special case there is no need of saving the state (it hasn't been already changed) while it's useful to re-suspend the system.



```
/*
 *The following thread function wait for all the processes to be suspended and
 * then decide whether to hibernate or not..
 */
void SuspendManager::hibernateDaemon(void*)
{
    Lock<Mutex>l(suspMutex);
    for(;;)
    {
        hibernWaiting.wait(l);
        syscallReturnTime.sort(compareResumeTime);
        list<SyscallResumeTime>::iterator it;
        it=syscallReturnTime.begin();
        //NOTE: the following if, as well as the upper and lower bunds,
        //will be replaced by the policy, once refined
        if((it->resumeTime-getTick()/1000)<=hibernationThreshold) continue;

        list<ProcessStatus*> backupProc;
        for(it=syscallReturnTime.begin();it!=syscallReturnTime.end();it++){
            map<pid_t,Process*>::iterator proc=Process::processes.find(it->pid);
            ProcessStatus* procStatus;
            if((proc!=Process::processes.end())&&(proc->second->numActiveThreads==0)){
                procStatus=new ProcessStatus(*(it->status));
                backupProc.push_back(procStatus);
                it->status=procStatus;
            }
        }

        ProcessStatus* proc=getProcessesBackupAreaBase();
        iprintf("Swapping %d processes\n",suspendedProcesses.size());
        list<Process*>::iterator findProc;

        for(findProc=suspendedProcesses.begin();
                findProc!=suspendedProcesses.end();findProc++)
        {
            (*findProc)->serialize(proc);
            if((*findProc)->toBeSwappedOut)
            {
                //TODO: optimize
                Mram& mram=Mram::instance();
                mram.exitSleepMode();
                //Copy the process image from RAM to MRAM
                mram.write(reinterpret_cast<unsigned int>(
                        (*findProc)->image.getProcessBasePointer())-
                        ProcessPool::instance().getBaseAddress(),
                        (*findProc)->image.getProcessBasePointer(),
                        (*findProc)->image.getProcessImageSize());
                mram.enterSleepMode();
                //Now serialize the state of the SRAM allocator
                ProcessPool::instance().serialize(getBackupSramBase());

                //Now serialize the state of the backup SRAM allocator
                (*(getBackupSramBase()+
                        getAllocatorSramAreaSize()/sizeof(int)))=
                        suspendedProcesses.size();
            }
            proc++;
        }
        for(list<ProcessStatus*>::iterator it=backupProc.begin();it!=backupProc.end();it++){
            delete *it;
        }
        SMART_SENSING::getSmartSensingInstance()->onSuspend(syscallReturnTime.begin()->resumeTime);
    }
}
```

Figure 19: hibernateDaemon procedure



```
    void SuspendManager::suspend(unsigned long long resumeTime) {
        long long prev = getTick();
        int sleepTime = resumeTime - prev / 1000;
        getBackupSramBase()[1021] = prev & 0xffffffff; //FIXME: hack
        getBackupSramBase()[1022] = prev >> 32;
        getBackupSramBase()[1023] = sleepTime;
        doSuspend(sleepTime);
    }
```

Figure 20: suspend method

4.3 Syscall integration part 1

The project has the purpose of give an useful tool to the processess. However in order to be utilized by them there must be provided a way of communication between the processes and the smart sensing module. This has been first of all achieved by the usage of non-blocking syscalls: one for schedule the reads, and one for retriving the data.

In order to make them work they must be spaced by a sleep of the requried time.

This method has the advantage of having a simple implementation (the behaviour of the suspension manager has not to be modified) and has been chosen to test the entire module.

A more advanced implementation is presented in the next subsection.

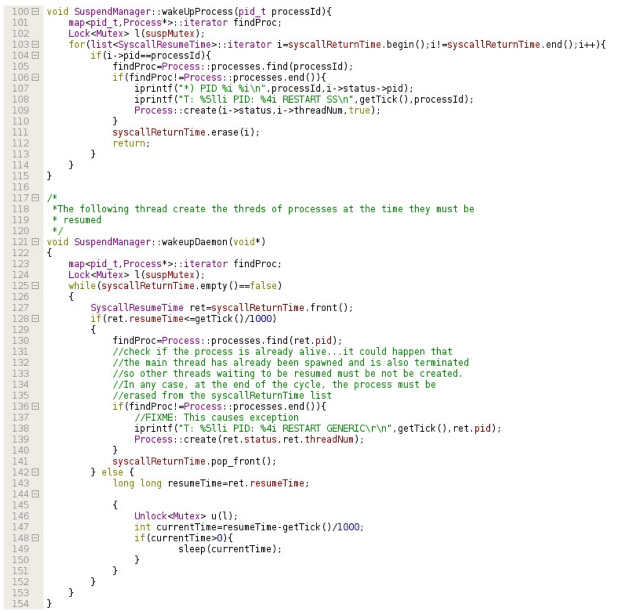
4.4 Syscall integration part 2

In this implementation all the work is performed by a single syscall which is a blocking read.

This has required an additional effort because first of all the smart sensing module has to wake up a process whose required job has been completed. Then also the suspension manager logic has to be modified because there is a new suspension-prone syscall which hasn't a defined resume time. In addition also the process manager has to be modified because the new particular syscall has to be resumed INSIDE the same syscall, while previously was resumed at the first instruction of the process AFTER the syscall. This last problem is caused by the peculiar way in which the suspension is performed: during a suspension ALL the kernel data is lost and only the processes ones data (i.e. stack and registers) are saved.

To cope with this several modification has been made.

First of all has been added in the SuspendManager class a new method, wakeUpProcess (**Figure 21**), which has the task of wake up a process whose smart sensing operation has finished.

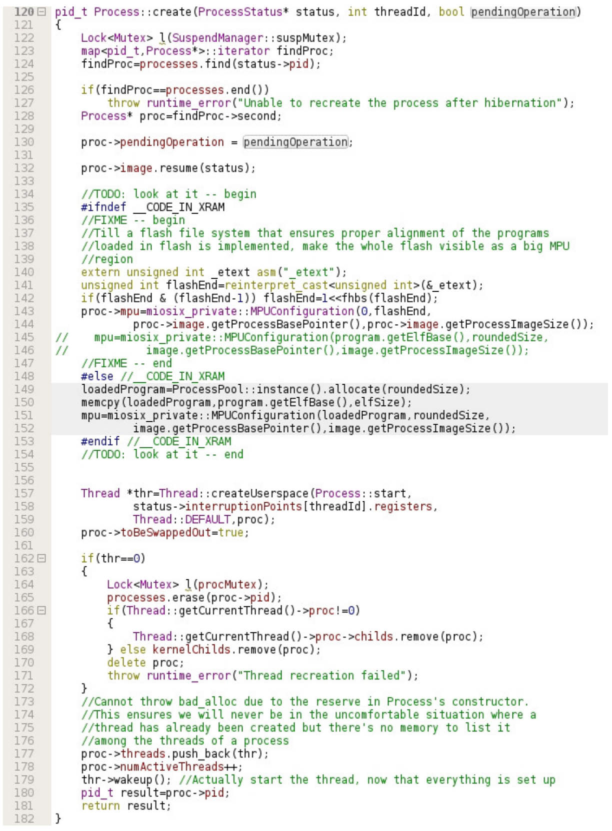


```
void SuspendManager::wakeUpProcess(pid_t processId){
    map<pid_t,Process*>::iterator findProc;
    Lock<Mutex> l(suspMutex);
    for(list<SyscallResumeTime>::iterator i=syscallReturnTime.begin();i!=syscallReturnTime.end();i++){
        if(i->pid==processId){
            findProc=Process::processes.find(processId);
            if(findProc!=Process::processes.end()){
                iprintf("*) PID %i %i\n",processId,i->status->pid);
                iprintf("T: %5lli PID: %4i RESTART SS\n",getTick(),processId);
                Process::create(i->status,i->threadNum,true);
            }
            syscallReturnTime.erase(i);
            return;
        }
    }
}

/*
 *The following thread create the threds of processes at the time they must be
 * resumed
 */
void SuspendManager::wakeupDaemon(void*)
{
    map<pid_t,Process*>::iterator findProc;
    Lock<Mutex> l(suspMutex);
    while(syscallReturnTime.empty()==false)
    {
        SyscallResumeTime ret=syscallReturnTime.front();
        if(ret.resumeTime<=getTick()/1000)
        {
            findProc=Process::processes.find(ret.pid);
            //check if the process is already alive...it could happen that
            //the main thread has already been spawned and is also terminated
            //so other threads waiting to be resumed must be not be created.
            //In any case, at the end of the cycle, the process must be
            //erased from the syscallReturnTime list
            if(findProc!=Process::processes.end()){
                //FIXME: This causes exception
                iprintf("T: %5lli PID: %4i RESTART GENERIC\r\n",getTick(),ret.pid);
                Process::create(ret.status,ret.threadNum);
            }
            syscallReturnTime.pop_front();
        } else {
            long long resumeTime=ret.resumeTime;

            {
                Unlock<Mutex> u(l);
                int currentTime=resumeTime-getTick()/1000;
                if(currentTime>0){
                        sleep(currentTime);
                }
            }
        }
    }
}
```

Figure 21: wakeUpProcess and wakeupDaemon methods

Then the process manager of Miosix has been modified in order to perform a two part syscall.

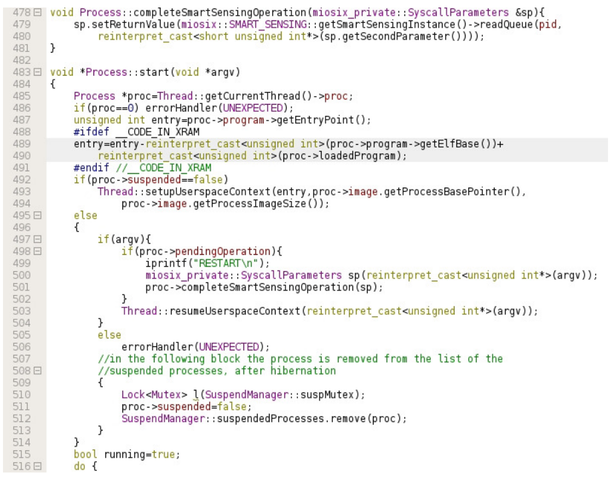
First has been added a variable to the Process class which keep track of a pending syscall related to the process. The method create in the class Process (**Figure 22**) has been modified by adding the variable pendingOperation, which if it's true, it means that a smart sensing read syscall is pending.



```cpp
pid_t Process::create(ProcessStatus* status, int threadId, bool pendingOperation)
{
    Lock<Mutex> l(SuspendManager::suspMutex);
    map<pid_t,Process*>::iterator findProc;
    findProc=processes.find(status->pid);

    if(findProc==processes.end())
        throw runtime_error("Unable to recreate the process after hibernation");
    Process* proc=findProc->second;

    proc->pendingOperation = pendingOperation;

    proc->image.resume(status);

    //TODO: look at it -- begin
    #ifndef __CODE_IN_XRAM
    //FIXME -- begin
    //Till a flash file system that ensures proper alignment of the programs
    //loaded in flash is implemented, make the whole flash visible as a big MPU
    //region
    extern unsigned int _etext asm("_etext");
    unsigned int flashEnd=reinterpret_cast<unsigned int>(&_etext);
    if(flashEnd & (flashEnd-1)) flashEnd=1<<fhbs(flashEnd);
    proc->mpu=miosix_private::MPUConfiguration(0,flashEnd,
            proc->image.getProcessBasePointer(),proc->image.getProcessImageSize());
//    mpu=miosix_private::MPUConfiguration(program.getElfBase(),roundedSize,
//            image.getProcessBasePointer(),image.getProcessImageSize());
    //FIXME -- end
    #else //__CODE_IN_XRAM
    loadedProgram=ProcessPool::instance().allocate(roundedSize);
    memcpy(loadedProgram,program.getElfBase(),elfSize);
    mpu=miosix_private::MPUConfiguration(loadedProgram,roundedSize,
            image.getProcessBasePointer(),image.getProcessImageSize());
    #endif //__CODE_IN_XRAM
    //TODO: look at it -- end


    Thread *thr=Thread::createUserspace(Process::start,
            status->interruptionPoints[threadId].registers,
            Thread::DEFAULT,proc);
    proc->toBeSwappedOut=true;

    if(thr==0)
    {
        Lock<Mutex> l(procMutex);
        processes.erase(proc->pid);
        if(Thread::getCurrentThread()->proc!=0)
        {
            Thread::getCurrentThread()->proc->childs.remove(proc);
        } else kernelChilds.remove(proc);
        delete proc;
        throw runtime_error("Thread recreation failed");
    }
    //Cannot throw bad_alloc due to the reserve in Process's constructor.
    //This ensures we will never be in the uncomfortable situation where a
    //thread has already been created but there's no memory to list it
    //among the threads of a process
    proc->threads.push_back(thr);
    proc->numActiveThreads++;
    thr->wakeup(); //Actually start the thread, now that everything is set up
    pid_t result=proc->pid;
    return result;
}
```

Figure 22: create method

This variable is used in the start method of Process class (**Figure 23**), which is used to complete the syscall before continuing the execution of the process, with the method completeSmartSensingOperation that simply copy the reads into the memory of the process and set as return value the number of bytes read (as the standard POSIX read requires).



```cpp
void Process::completeSmartSensingOperation(miosix_private::SyscallParameters &sp){
    sp.setReturnValue(miosix::SMART_SENSING::getSmartSensingInstance()->readQueue(pid,
        reinterpret_cast<short unsigned int*>(sp.getSecondParameter())));
}

void *Process::start(void *argv)
{
    Process *proc=Thread::getCurrentThread()->proc;
    if(proc==0) errorHandler(UNEXPECTED);
    unsigned int entry=proc->program->getEntryPoint();
    #ifdef __CODE_IN_XRAM
    entry=entry-reinterpret_cast<unsigned int>(proc->program->getElfBase())+
        reinterpret_cast<unsigned int>(proc->loadedProgram);
    #endif //__CODE_IN_XRAM
    if(proc->suspended==false)
        Thread::setupUserspaceContext(entry,proc->image.getProcessBasePointer(),
            proc->image.getProcessImageSize());
    else
    {
        if(argv){
            if(proc->pendingOperation){
                iprintf("RESTART\n");
                miosix_private::SyscallParameters sp(reinterpret_cast<unsigned int*>(argv));
                proc->completeSmartSensingOperation(sp);
            }
            Thread::resumeUserspaceContext(reinterpret_cast<unsigned int*>(argv));
        }
        else
            errorHandler(UNEXPECTED);
        //in the following block the process is removed from the list of the
        //suspended processes, after hibernation
        {
            Lock<Mutex> l(SuspendManager::suspMutex);
            proc->suspended=false;
            SuspendManager::suspendedProcesses.remove(proc);
        }
    }
    bool running=true;
    do {
```

Figure 23: completeSmartSensingOperation and part of start methods

The core of the syscall is presented at **Figure 24**. Here we can notice a swith over the possible syscall, where the interesting part is the read one (with id equal to 4).

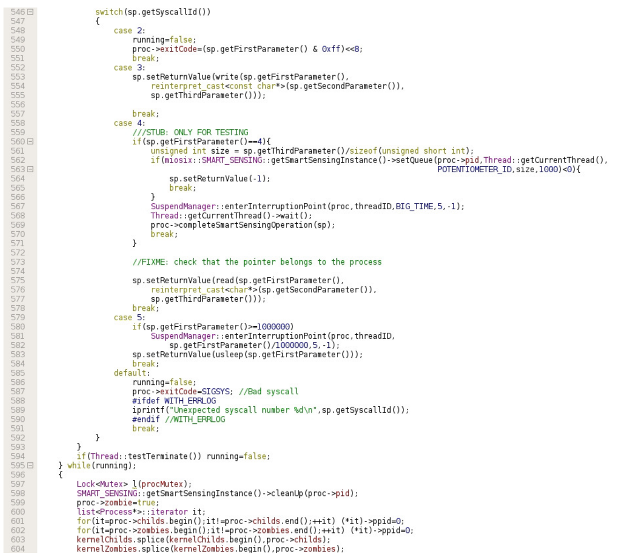
The code first of all check the value of the file description. If it's equal to 4 it's interpreted as 5 reads from the potentiometer spaced by 1 second each.

This trick has been used because the filesystem managent is beyond the scope of this project and this has been a rapid way for testing the smart sensing mechanism. We can imagine that the information of device id, number of reads and period can be memorized in some data structures by a modified open syscall and then utilized here.

So the smart sensing operation is set and at line 567 is signaled to the suspension manager that the process can enter to suspension. Here is used another trick to mantain as much as possible the compatibility with the suspension manager: the smart sensing read is treated like a sleep of a very long interval.

The rest of the syscall (line 568-569) wait and complete the syscall in the case that all the smart sensing operation is performed without putting the system in suspension. In this case the thread of the process is woken up by the smart sensing class when the job is finished. The whole mechanism works because in case of suspension all the kernel data is lost and so both the process thread status and the thread id inside the SmartSensing class.

The last addition is in line 598, which calls the method cleanUp in the smart sensing module when the process is being destroyed. This is necessary to eventually free the queues of the process inside the SmartSensing class.



```cpp
546            switch(sp.getSyscallId())
547            {
548                case 2:
549                    running=false;
550                    proc->exitCode=(sp.getFirstParameter() & 0xff)<<8;
551                    break;
552                case 3:
553                    sp.setReturnValue(write(sp.getFirstParameter(),
554                        reinterpret_cast<const char*>(sp.getSecondParameter()),
555                        sp.getThirdParameter()));
556
557                    break;
558                case 4:
559                    ///STUB: ONLY FOR TESTING
560                    if(sp.getFirstParameter()==4){
561                        unsigned int size = sp.getThirdParameter()/sizeof(unsigned short int);
562                        if(miosix::SMART_SENSING::getSmartSensingInstance()->setQueue(proc->pid,Thread::getCurrentThread(),
563                                                                      POTENTIOMETER_ID,size,1000)<0){
564                            sp.setReturnValue(-1);
565                            break;
566                        }
567                        SuspendManager::enterInterruptionPoint(proc,threadID,BIG_TIME,5,-1);
568                        Thread::getCurrentThread()->wait();
569                        proc->completeSmartSensingOperation(sp);
570                        break;
571                    }
572
573                    //FIXME: check that the pointer belongs to the process
574
575                    sp.setReturnValue(read(sp.getFirstParameter(),
576                        reinterpret_cast<char*>(sp.getSecondParameter()),
577                        sp.getThirdParameter()));
578                    break;
579                case 5:
580                    if(sp.getFirstParameter()>=1000000)
581                        SuspendManager::enterInterruptionPoint(proc,threadID,
582                            sp.getFirstParameter()/1000000,5,-1);
583                    sp.setReturnValue(usleep(sp.getFirstParameter()));
584                    break;
585                default:
586                    running=false;
587                    proc->exitCode=SIGSYS; //Bad syscall
588                    #ifdef WITH_ERRLOG
589                    iprintf("Unexpected syscall number %d\n",sp.getSyscallId());
590                    #endif //WITH_ERRLOG
591                    break;
592            }
593        }
594        if(Thread::testTerminate()) running=false;
595    } while(running);
596    {
597        Lock<Mutex> l(procMutex);
598        SMART_SENSING::getSmartSensingInstance()->cleanUp(proc->pid);
599        proc->zombie=true;
600        list<Process*>::iterator it;
601        for(it=proc->childs.begin();it!=proc->childs.end();++it) (*it)->ppid=0;
602        for(it=proc->zombies.begin();it!=proc->zombies.end();++it) (*it)->ppid=0;
603        kernelChilds.splice(kernelChilds.begin(),proc->childs);
604        kernelZombies.splice(kernelZombies.begin(),proc->zombies);
```

Figure 24: extract from start method

In summary we recap the whole syscall mechanism which is quite complex due to the various suspension call. This can be summed up with the following steps:

1. The process invoke a smart sensing read syscall.

2. The syscall manager inside the kernel sets up the smart sensing operation, signals to the suspension module the possibility of a suspension and waits.

3. If the operation is completed without any suspension the process wakes up and the syscall returns normally, otherwise a suspension is performed.

4. The system can be started and suspended several times. The last one the smart sensing module calls the wakeUpProcess in the SuspendManager class directly if the read occours when the kernel is already started or when the smart sensing daemon is started if the last read happens before the boot process.

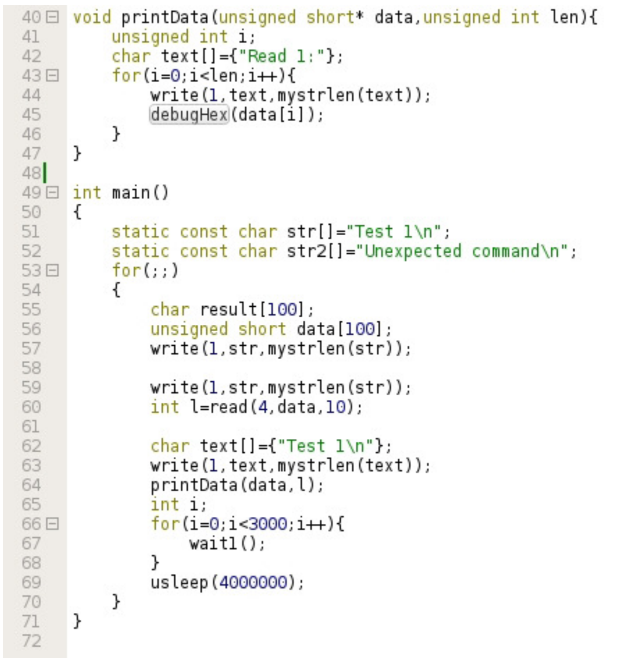
5. The wakeUpProcess wake up the corresponding process with the flag of the pending syscall set.

6. The start method in the Process class detects the pending syscall, completes it and return the control to the process.

7. The process continues its execution as nothing were happened between.

5 Testing

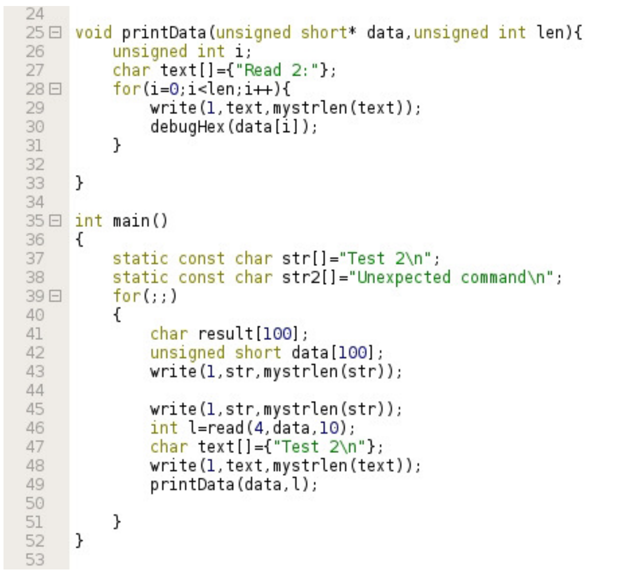
For the testing of the project were used two programs.
The first (**Figure 25**) performs a series of a smart sensing acquisition composed of 5 reads spaced by 1 second and 3 seconds of active wait (it's like a sleep but it doesn't allow the system going into suspension).



```c
void printData(unsigned short* data,unsigned int len){
    unsigned int i;
    char text[]={"Read 1:"};
    for(i=0;i<len;i++){
        write(1,text,mystrlen(text));
        debugHex(data[i]);
    }
}

int main()
{
    static const char str[]="Test 1\n";
    static const char str2[]="Unexpected command\n";
    for(;;)
    {
        char result[100];
        unsigned short data[100];
        write(1,str,mystrlen(str));

        write(1,str,mystrlen(str));
        int l=read(4,data,10);

        char text[]={"Test 1\n"};
        write(1,text,mystrlen(text));
        printData(data,l);
        int i;
        for(i=0;i<3000;i++){
            wait1();
        }
        usleep(4000000);
    }
}
```

Figure 25: code of the first testing program

The second one (**Figure 26**) instead performs just a series of smart sensing acquisition again of 5 reads spaced by 1 second.



```c
void printData(unsigned short* data,unsigned int len){
    unsigned int i;
    char text[]={"Read 2:"};
    for(i=0;i<len;i++){
        write(1,text,mystrlen(text));
        debugHex(data[i]);
    }

}

int main()
{
    static const char str[]="Test 2\n";
    static const char str2[]="Unexpected command\n";
    for(;;)
    {
        char result[100];
        unsigned short data[100];
        write(1,str,mystrlen(str));

        write(1,str,mystrlen(str));
        int l=read(4,data,10);
        char text[]={"Test 2\n"};
        write(1,text,mystrlen(text));
        printData(data,l);

    }
}
```

Figure 26: code of the second testing program

The resulting log correctly outlines the alternation of smart sensing reads and system activations; also the kernel daemon part behaves correctly and the values read from the potentiometer were correct.

References

[1] Wolfgang Wieser. *Programming STM32 F2, F4 ARMs under Linux: A Tutorial from Scratch*, 2012.

[2] Trevor Martin. *The Insider's Guide To The STM32 ARM Based Microcontroller - 2nd Edition*. 2009.

[3] ST Microelectronics. *STM3220G-EVAL evaluation board user manual (UM1057)*, 2012.

[4] ST Microelectronics. *STM32F2xx reference manual (RM0033)*, 2011.

[5] ST Microelectronics. *STM32F2xx programming manual (PM0056)*, 2011.

[6] Federico Terraneo. *Control based design of OS components*, 2011.