Miosix kernel 2.0 features

Note: this document is a draft, since the current Miosix version is 1.54. When all planned features will be included, the kernel version will be bumped to 2.0. However, **all** features described here are already implemented and tested.

Multiplatform kernel

Miosix is currently available for the ARM7 based LPC2000 family of microcontrollers, and for the ARM Cortex M3 based STM32 microcontrollers.

For each platform it provides:

- A suitable linker script, startup code and low-level routines to allow preemption and context switching.
- A driver to access a serial port.
- A driver to allow the filesystem module read/write access on an SD or MicroSD card.
- A platform independent abstraction over GPIOs, the Gpio tempate class.
- A platform independent set of delay routines, to obtain delays down to microseconds.

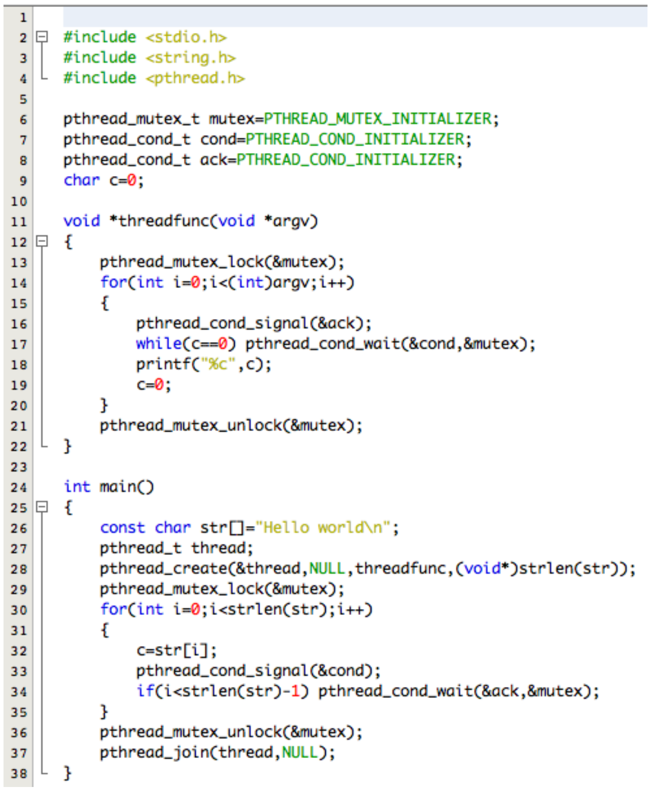
In addition, Miosix has been refactored for easy porting to other architectures, by separating architecture independent code from low level code.

Multithreading support

The Miosix kernel is a single process, multithread kernel. This is because most microcontrollers lack an MMU and therefore are incapable of the memory separation required to fully implement processes.

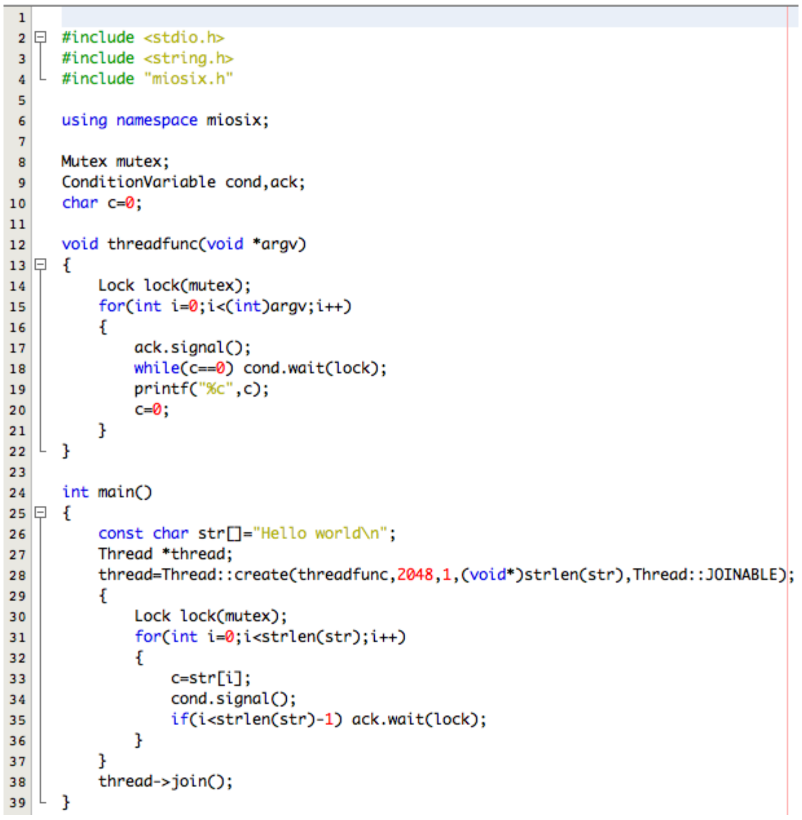
While multithreading is a key feature of the kernel, available since its first releases, starting from Miosix 1.54 a preliminary implementation of the POSIX thread API is available, with support for threads, mutexes and condition variables.

This is an example of a pthread program (a classic producer-consumer model) that can be run on Miosix. Please note that the same program can be run on a desktop operating systems unchanged:



```c

#include <stdio.h>
#include <string.h>
#include <pthread.h>

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
pthread_cond_t ack=PTHREAD_COND_INITIALIZER;
char c=0;

void *threadfunc(void *argv)
{
    pthread_mutex_lock(&mutex);
    for(int i=0;i<(int)argv;i++)
    {
        pthread_cond_signal(&ack);
        while(c==0) pthread_cond_wait(&cond,&mutex);
        printf("%c",c);
        c=0;
    }
    pthread_mutex_unlock(&mutex);
}

int main()
{
    const char str[]="Hello world\n";
    pthread_t thread;
    pthread_create(&thread,NULL,threadfunc,(void*)strlen(str));
    pthread_mutex_lock(&mutex);
    for(int i=0;i<strlen(str);i++)
    {
        c=str[i];
        pthread_cond_signal(&cond);
        if(i<strlen(str)-1) pthread_cond_wait(&ack,&mutex);
    }
    pthread_mutex_unlock(&mutex);
    pthread_join(thread,NULL);
}
```

While this is the same program coded with the native Miosix API:



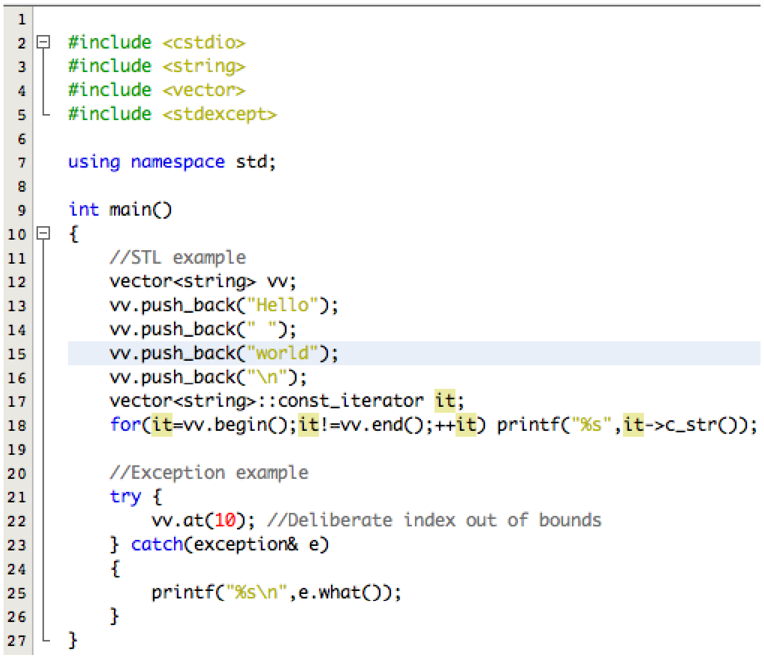
```

#include <stdio.h>
#include <string.h>
#include "miosix.h"

using namespace miosix;

Mutex mutex;
ConditionVariable cond,ack;
char c=0;

void threadfunc(void *argv)
{
    Lock lock(mutex);
    for(int i=0;i<(int)argv;i++)
    {
        ack.signal();
        while(c==0) cond.wait(lock);
        printf("%c",c);
        c=0;
    }
}

int main()
{
    const char str[]="Hello world\n";
    Thread *thread;
    thread=Thread::create(threadfunc,2048,1,(void*)strlen(str),Thread::JOINABLE);
    {
        Lock lock(mutex);
        for(int i=0;i<strlen(str);i++)
        {
            c=str[i];
            cond.signal();
            if(i<strlen(str)-1) ack.wait(lock);
        }
    }
    thread->join();
}
```

C and C++ standard library

Miosix integrates with the Newlib C standard library, which is designed for embedded systems, and with libstdc++, which is the C++ standard library provided with GCC.

The C/C++ library's console I/O facilities (printf, cout) are by default redirected to a serial port, and the filesystem functions (fopen, fstream) are integrated with the filesystem module. The C++'s STL library and exception handling are provided. Dynamic memory allocation is of course available.

This is a simple example that shows C++'s STL and exception support in Miosix. This code would compile without modifications even on a desktop computer:



```

#include <cstdio>
#include <string>
#include <vector>
#include <stdexcept>

using namespace std;

int main()
{
    //STL example
    vector<string> vv;
    vv.push_back("Hello");
    vv.push_back(" ");
    vv.push_back("world");
    vv.push_back("\n");
    vector<string>::const_iterator it;
    for(it=vv.begin();it!=vv.end();++it) printf("%s",it->c_str());

    //Exception example
    try {
        vv.at(10); //Deliberate index out of bounds
    } catch(exception& e)
    {
        printf("%s\n",e.what());
    }
}
```

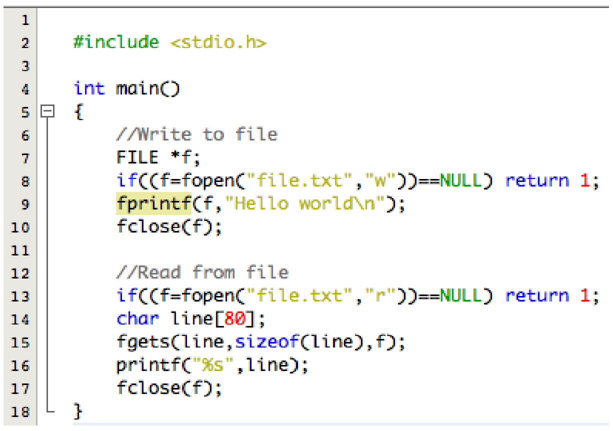
Filesystem support

Miosix includes read/write support on FAT16 and FAT32 filesystems. Code to create directories and list their content is also provided.

For write support, the option SYNC_AFTER_WRITE allows to choose the fastest write speed, or a safe mode where data is written to disk as soon as possible, to minimize the risk of data loss in case when power is removed unexpectedly.

Filesystem access is integrated with the C/C++ standard library, so that it is not necessary to learn a custom API to write or read files.

This example shows file read/write using the C library on Miosix. Again, this code would compile without modifications even on a desktop computer:



```c

#include <stdio.h>

int main()
{
    //Write to file
    FILE *f;
    if((f=fopen("file.txt","w"))==NULL) return 1;
    fprintf(f,"Hello world\n");
    fclose(f);

    //Read from file
    if((f=fopen("file.txt","r"))==NULL) return 1;
    char line[80];
    fgets(line,sizeof(line),f);
    printf("%s",line);
    fclose(f);
}
```

Modular architecture

Miosix provides many high level features, but recognizes that the code size associated with them might not be acceptable in all embedded applications.

First of all, part of the code size optimization can be done automatically by the linker. Since Miosix lacks a loader to dynamically load executable code, the list of functions and classes that are used by an application are known at compile time. This means that the linker can strip the parts of the C and C++ libraries that are not used.

In addition, Miosix has modular architecture where the developer can choose only the features required:

- If code does not use C++ exceptions, it is possible to save code size by disabling its support.
- Certain parts of the kernel, such as the filesystem module, can be removed if not used.
- Boot logs and error logs can be removed in release builds.

Lastly it is possible to choose compiler optimization, in miosix/config/Makefile.inc. The default is -O2 (moderate speed optimization), but -Os is available to optimize for minimum size.

As a proof of concept, Miosix 1.54 for stm32 was compiled with the following minimal configuration:

- No C++ exception support (but the rest of the C++ language fatures still available)
- Release mode (code running from FLASH, no JTAG_DISABLE_SLEEP)
- GCC Compiler optimization -Os
- No filesystem
- Options: EXIT_ON_HEAP_FULL, WITH_STDIN_BUFFER, No boot logs, No error logs

The resulting code size was just 6228 Bytes.

However, it is suggested to use a microcontroller with at least 64KB of FLASH and 16KB of RAM.

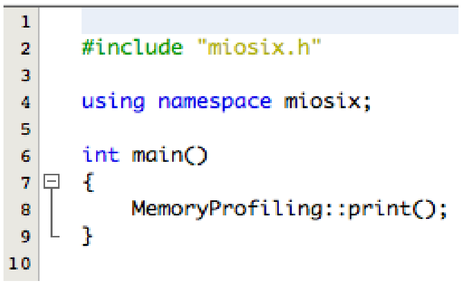
Memory statistics gathering

Microcontrollers often have little RAM memory available, and this limitation is even more evident when using multithreading. Since each thread needs to have its own stack, it is important to select the right size for it, not too large to avoid excessie RAM usage, and not too low to cause stack overflows.

Miosix includes code designed to gather statistics about memory about both stack and heap memory usage. These statistics can be used to take decisions at runtime, or simply printed during the development phase, and used as a guide for optimizing code.

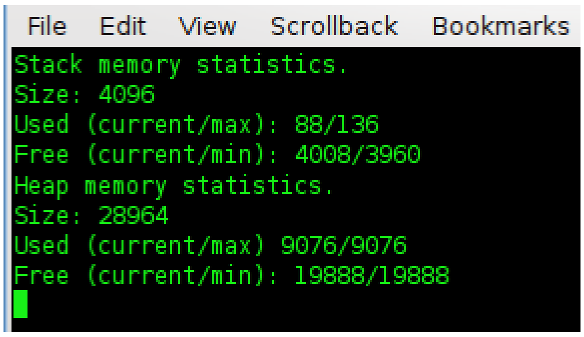
The statistics code is in miosix/util.h and is contained in the MemoryProfiling class. To check what this class provides, see the doxygen documentation.

Here is a simple example code:



```

#include "miosix.h"

using namespace miosix;

int main()
{
    MemoryProfiling::print();
}

```

And the output on the console:



```
Stack memory statistics.
Size: 4096
Used (current/max): 88/136
Free (current/min): 4008/3960
Heap memory statistics.
Size: 28964
Used (current/max) 9076/9076
Free (current/min): 19888/19888
```

The print() member fuction of the MemoryProfiling class is designed to print a short summary of the information that can be gathered through this class:

- It prints the stack size of the thread from which it was called. As can be seen, main() has a 4KBytes stack.
- The currently used stack, which is 88Bytes and the maximum stack used since the thread was spawned, 136Bytes.
- The current and minimum free stack, which is simply the stack size minus the used stack.
- The heap size.
- The current and maximum used heap.
- The current and minimum free heap.

While the heap is global in Miosix, which means that all threads share the same heap, each thread has its own stack. This implies that calling MemoryProfiling::print() from different threads will return different data about stack sizes and usage, to reflect the situation of the threads, but the same heap data.

When spawning threads, their stack sized can of course be customized. To see how, look at the doxygen documentation of Thread::create().

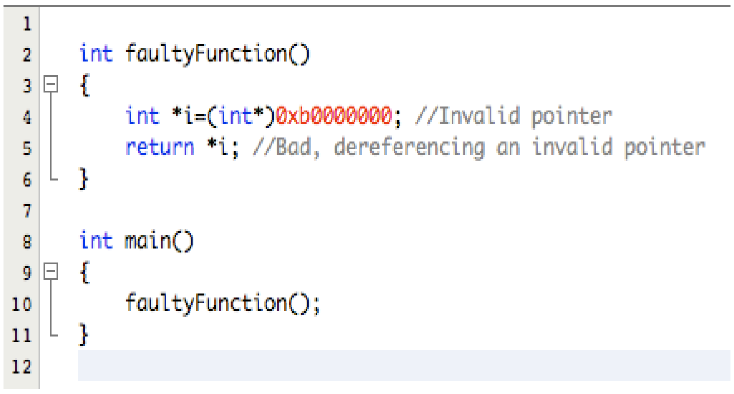
The size of the main() thread can be customized too, even if it defaults to 4KB.

Detailed debug information

When coding is not uncommon to introduce bugs. Miosix can make use of the undelying hardware to catch bugs as soon as possible and make this debug data available to the developer to speed up bug fixing.

As usual the modular architecture of the kernel allows to remove this debug support inside the kernel to minimize code size of release builds. The option is named WITH_ERRLOG and is in miosix/config/miosix_settings.h

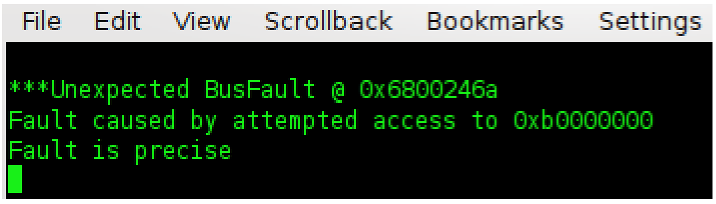
This is an example of a pointer error to see how does Miosix react to it.



```
 1
 2  int faultyFunction()
 3  {
 4      int *i=(int*)0xb0000000; //Invalid pointer
 5      return *i; //Bad, dereferencing an invalid pointer
 6  }
 7
 8  int main()
 9  {
10      faultyFunction();
11  }
12
```

The location 0xb0000000 in the STM32 microcontroller is an invalid address, so dereferencing the pointer is an error.

This is what Miosix prints on the serial port when this code is run.



```
***Unexpected BusFault @ 0x6800246a
Fault caused by attempted access to 0xb0000000
Fault is precise
```

The Cortex M3 microcontroller catches the violation and issues a BusFault exception. Miosix kicks in and makes the debug information available. Debug information starts with three asterisks to distinguish it from data that the developer prints through printf.

The debug data gives many precious informations:

- It tells the kind of fault that occurred (BusFault)
- It tells the address of the instruction that caused the fault (0x6800246a). By itself the address might not seem very useful, but using two programs provided with GCC, addr2line and c++filt it is possible to make use of it

  ```
  arm-eabi-addr2line 0x6800246a -e main.elf -f | arm-eabi-c++filt
  faultyFunction()
  demo.cpp:5
  ```

  As can be seen it is possible to get the name of the function, the file and the source code line (5) where the fault happened. For this information to be reliable, code needs to be compiled with optimizations disabled (using -O0 as OPT_OPTIMIZATION) since optimizations in the compiler, such as function inlining can confuse this.
- It tells the memory location whose attempted access caused the fault (0xb0000000).

Miosix can detect stack overflows using two different checks that are done at every context switch:

- Check that the stack pointer of the preempted thread is within bounds.
- Check that a watermark area placed at the end of each stack has not been overwritten. The size of this area is configurable to optimize the tradeoff between safety and fast context switches. The option is called WATERMARK_LEN.

Of course these are not the only errors that can be detected, other include deadlock caused by a thread locking a non recursive mutex twice, errors in parameters to system calls, attempted use of filesystem calls in builds where the filesystem is not enabled, etc.

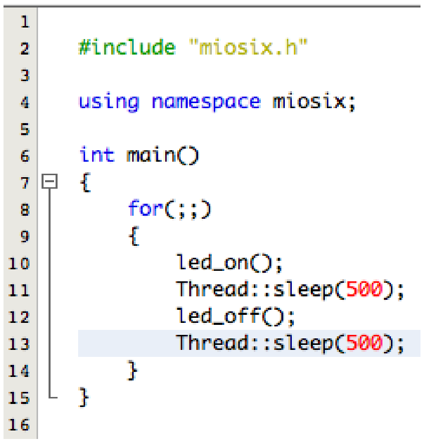
The kernel is also friendly towards JTAG in-circuit debuggers, providing an option named JTAG_DISABLE_SLEEP to prevent the idle thread from putting the processor in low power mode, since this confuses in-circuit debuggers.

Suitable for low power applications

Most modern microcontrollers provide an idle mode where the CPU is stopped waiting for interrupts while peripherals are running, which can provide a consderable power consumption reduction.

Miosix takes advantage of this, in a way that is transparent to the developer. Its kernel, when no thread is ready to run, runs the idle thread which puts the CPU in idle mode.

This simple exampe code blinks a led at an 1Hz rate.



```cpp

#include "miosix.h"

using namespace miosix;

int main()
{
    for(;;)
    {
        led_on();
        Thread::sleep(500);
        led_off();
        Thread::sleep(500);
    }
}

```

In this application the CPU is idle for most of the time, because when main() calls sleep(), no thread is ready to run, and the idle thread puts the CPU in idle state.