# Rodos Porting Guide

Aus I8-Wiki

## Inhaltsverzeichnis

## Create Enviroment

Each Rodos Hardwareport has a separate folder in /src/bare-metal/. The folder shall have a name corresponding to the platform the port is made for.

## Functions to implement

A Rodos Port needs some special functions, which will be used by the Rodos-Kernel. Some of them are essential for Rodos, other can be left as a stub and are not necessary on all platforms or are only used for special features.

- Red: Important, hardware specific function.
- Green: Function may be a stub, depending on hardware.
- Gray: Function is most likely a stub on baremetal or returns only a static value.

| File | Function | Description |
|---|---|---|
| hostinfos.cpp | bool getIsHostBigEndian(); | Returns whether the target Host is BigEndian. |
| | long getSpeedKiloLoopsPerSecond(); | Returns how many kiloloops per second the target Platform can perform. |
| | const char* getHostCpuArch(); | Returns the CPU Architecture. |
| | const char* getHostBasisOS(); | Returns the Host Operating System Rodos is running on. |
| hw_specific.cpp | void hwInit(); | Responsible for Hardware Initialization |
| | long* hwInitContext(long* stack, void* object); | create context on stack and return a pointer to it |
| | void hwResetAndReboot(); | reset and reboot |
| | void sp_partition_yield(); | special for use with Host OS |
| | void startIdleThread(); | Called to start Idle Thread |
| not specified | void hwInitTime(); | Initializes the timer-functions |
| | long long unsigned int hwGetNanoseconds(); | Returns nanoseconds since startup. |
| | void Timer::init(); | Initializes the hardware timer. |
| | void Timer::stop(); | Disables the timer interrupt. |
| | void Timer::start(); | Re-enables the timer interrupt. |
| | void Timer::setInterval(const long long interval); | Sets the timer interval. |
| | void __asmSwitchToContext(long* context); | Switches to the passed context. |
| | void __asmSaveContextAndCallScheduler(); | Saves the current context and calls the scheduler. |

## System Configuration

### params.h

There is a number of RODOS-related configuration parameters that define hardware limits and behavior of the operating system. The name of this header is expected to be params.h and must contain a list of preprocessor directives.

```
#define OSVERSION "RODOS-linux-8"
```

```
/*************** System Configuration *********/

/** Memory for allocation (xmalloc) eg for all stacks ***/
#define XMALLOC_SIZE            2000000
/** default stack size (in bytes) for threads */
#define DEFAULT_STACKSIZE       32000
/** stack size (in bytes) for scheduler thread */
#define SCHEDULER_STACKSIZE DEFAULT_STACKSIZE

/** time interval between timer interrupts in microseconds */
#define PARAM_TIMER_INTERVAL   100000

/*** time for time slice to swtich between threads with same priority ***/
#define  TIME_SLICE_FOR_SAME_PRIORITY (100*MILLISECONDS)

/** default priority for newly created threads */
#define DEFAULT_THREAD_PRIORITY                    100

/** user threads shall not have a priority higher than this */
#define MAX_THREAD_PRIORITY                    1000

/** high priority levels for priority ceiling  */
#define NETWORKREADER_PRIORITY          (MAX_THREAD_PRIORITY + 2)
/** high priority levels for priority ceiling  */
#define CEILING_PRIORITY                        (NETWORKREADER_PRIORITY + 1)



/** using a network, the maximal number of nodes attached */
#define MAX_NUMBER_OF_NODES                         10
/** if using network it may have a limit for pakets, eg udp 1500 */
#define MAX_NETWORK_MESSAGE_LENGTH              1300


/*** If using a network: Maximal number of subscribers per node */
#define MAX_SUBSCRIBERS                                                 60
```

The setting for *XMALLOC_SIZE* defines how much memory is going to be available all together. One has to understand that the whole amount of memory RODOS will consume is static and known at compile time. The reserved amount is used by the xmalloc function that also allocates every thread's stack defined by *DEFAULT_STACKSIZE*.

Usually the setting *PARAM_TIMER_INTERVAL* defines the interval between two consecutive timer interrupts. As for real-time operating systems, the timer Interrupt Service Routine (ISR) is also the place from which the scheduler is called. This is later explained as preemptive context switching. In case of a priority conflict, scheduling should be done with the round-robin algorithm. For this, the conflicting threads are assigned equal time slices that can be defined with *TIME_SLICE_FOR_SAME_PRIORITY*. When executing such threads, the variable *timeToTryAgainToSchedule* is set to the current time, plus this time slice definition. The ISR is then implemented in a way that it calls the scheduler only if the time at that moment has passed *timeToTryAgainToSchedule*. For the time these threads are active, the scheduler is called less frequently and only in the interval specified by *TIME_SLICE_FOR_SAME_PRIORITY*.

## hostinfos.cpp

Contains Functions, which return information about the environment Rodos is running on.

**bool getIsHostBigEndian();**

- Returns whether the target Host is BigEndian.
- e.g.:

```
bool  getIsHostBigEndian()          { return true; }
```

**long getSpeedKiloLoopsPerSecond();**

- Returns how many kiloloops per second the target Platform can perform. This value can be determined with the performace test /tutorials/advanced_and_complex/development-tests/cpuspeedtest.cpp. If there is no reliable value for this, the function shall return -1.
- e.g.:

```
long  getSpeedKiloLoopsPerSecond()  { return 350000; }
```

**const char* getHostCpuArch();**

- Returns the CPU Architecture.
- e.g.:

```
const char* getHostCpuArch()        { return "x86";       }
```

**const char* getHostBasisOS();**

- Returns the Host Operating System Rodos is running on. If direct on Hardware Host OS is "baremetal".
- e.g.:

```
const char* getHostBasisOS()        { return "baremetal"; }
```

## Startup and Control Routines

### hw_specific.cpp

**void hwInit();**

- may be a stub
- Responsible for Hardware Initialization
- e.g.:

```
void hwInit() {  }
```

**long* hwInitContext(long* stack, void* object);**

- create context on stack and return a pointer to it
- heavily dependant on the used processor

For every thread hwInitContext() is called to set up the context with certain parameters. The first argument to this function is a pointer to the start of the thread’s stack and provides the basis to compute an appropriate stack pointer. Stacks are generally very architecturespecific and may require special alignments. The second argument is the thread's object pointer ( this ) that is passed to the callback function. It is stored in the register that makes up the first argument on a function call. This initial entry point is the callback function threadStartupWrapper() . Its address has to be stored within a context structure so it will be loaded to the program counter when switching to the thread. The startup wrapper begins the thread execution by jumping to its run() routine. Upon exit, hwInitContext() returns a pointer to the newly created context. This is usually the address of a C struct containing space for a number of registers which is located within the initially allocated stack frame.

- e.g.: (avr32)

```
long* hwInitContext(long* stack, void* object) {
        /* create first thread context as it would be created by a context switch interrupt. */
        stack -= 4;                                                 // R8-R11
        *stack = (long) object;                             // R12
        stack--;
        *stack = 0;                                                     // LR
        stack--;
        *stack = (long)(threadStartupWrapper);  // PC
        stack--;
        *stack = 0x00600000;                                // start value of processor status re
                                                                            // mode bits
                                                                            // exception
                                                                            // global int
        stack -= 8;                                                     // R1-R7

        return stack;
}
```

**void hwResetAndReboot();**

- reset and reboot - may be a stub
- e.g.:

```
void hwResetAndReboot() {  }
```

**void sp_partition_yield();**

- stub on bare-metal
- e.g.:

```
void sp_partition_yield() {  }
```

**void startIdleThread();**

- stub on most bare-metal targets
- e.g.:

```
void startIdleThread() { }
```

## Timing Interfaces

**void hwInitTime();**

- Initializes the timer-functions e.g. starts a hardware timer to count nanoseconds.
- may be a stub
- e.g.:

```
void hwInitTime() {  }
```

**long long unsigned int hwGetNanoseconds();**

- Returns nanoseconds since startup. In most cases a hardware timer will be used.
- e.g.: (arm_cortex)

```
unsigned long long hwGetNanoseconds()
{
        return nanoTime;
}
```

**void Timer::init();**

Initializes the hardware timer. Serves the same purpose as hwInitTime(), so one of them may be a stub.

**void Timer::stop();**

Used to disable the timer interrupt. This is called during a voluntary conext switch, to prevent it from being interrupted. Special caution has to be taken, if the system time is updated in the timer interrupt, as it may miss an interrupt thus the time may be wrong.

**void Timer::start();**

Re-enables the timer interrupt.

**void Timer::setInterval(const long long interval);**

Sets the timer interval.

## Context Switching

**void __asmSwitchToContext(long* context);**

**void __asmSaveContextAndCallScheduler();**