Embedded Parallel Operating System

Software/Hardware Integration Lab

EPOS 2 User Guide

Table of contents

1. Introduction

This document is a reference guide to the EPOS API. It is designed focusing application development on top of EPOS.

1.1. EPOS Overview

The **Embedded Parallel Operating System (EPOS)** aims at automating the development of dedicated computing systems, so that developers can concentrate on what really matters: their applications. EPOS relies on the **Application-Driven Embedded System Design Method (ADESD)** proposed by Antônio Augusto Fröhlich to design and implement both software and hardware components that can be automatically adapted to fulfill the requirements of particular applications. Additionally, EPOS features a set of tools to select, adapt and plug components into an application-specific framework, thus enabling the automatic generation of an application-oriented system instance. Such an instance consists of a hardware platform implemented in terms of programmable logic, and the corresponding run-time support system implemented in terms of abstractions, hardware mediators, scenario adapters and aspect programs.

The deployment of ADESD in EPOS is helping to produce components that are highly reusable, adaptable and maintainable. Low overhead and high performance are achieved by a careful implementation that makes use of *generative programming* techniques, including *static metaprogramming*. Furthermore, the fact that EPOS components are exported to users by means of coherent interfaces defined in the context of the application domain largely improves usability. All these technological advantages are directly reflected in the development process, reducing NRE costs and the time-to-market of software/hardware integrated projects.

OpenEPOS is a streamlined version of EPOS in which more complex, less stable research components have been removed to produce a system that can be easily used for industrial or university applications.

1.2. OpenEPOS License

OpenEPOS 2.x is licensed under the The GNU Lesser General Public Licence 2.1. In this site, **EPOS** and **OpenEPOS** are used interchangeably to designate the specific set of components publicly released in this site under the LGPL license. Other components, not listed in this documentation and not released through this site, are usually subject to more restrictive licences. For additional information, please contact lisha@lisha.ufsc.br.

Older versions of OpenEPOS are licensed under EPOS Software License v1.0.

1.3. Main Features

An overview of the features currently implemented in each version as well as a list of supported architectures and machines (i.e. platforms) is show below. You can download the releases from here.

Feature		**OpenEPOS Release**			
		1.0	**1.1**	**1.2**	**2.0**
Architectures	AVR8	√	√	√	–
	ARM7 (ARMv4)	–	√	√	–
	ARMv7-M	–	–	–	√
	Ix86 (IA-32)	√	√	√	√
	Ix86_64	–	–	√	–
	PowerPC	≈	√	√	–
	MIPS	≈	√	√	–
Machines	EPOSMote I	√	√	√	–
	EPOSMote I	√	–	–	
	EPOSMote II	–	√	√	–
	EPOSMote III	–	–	–	√
	TI Stellaris LM3S9B96 (QEMU)	–	–	–	√
	Atmega16	√	√	√	–
	Atmega128	√	√	√	–
	Atmega1281	√	√	√	–
	At90can128	√	√	√	–
	ML310	≈	√	√	–

	Plasma	≈	√	√	–
	PC	√	√	√	√
Devices	UART	√	√	√	√
	USART	√	√	√	–
	Ethernet	√	√	√	√
	Radio (IEEE 802.15.4)	√	√	√	√
	EEPROM	√	√	√	√
	Flash	√	√	√	√
	Timer	√	√	√	√
	SPI	√	√	√	√
Process	Multithreading	√	√	√	√
	Real-time Scheduling	√	√	√	√
	Multicore (SMP)	√	√	√	√
	Synchronization	√	√	√	√
	Multitasking	–	–	–	√
Memory	Dynamic Memory Allocation	√	√	√	√
	Scratch-pad Memory	–	≈	√	√
	Flash	√	√	√	√
Timing	Timed Events	√	√	√	√
	Chronometer	√	√	√	√
	Real-time Clock	√	√	√	√
	Watch-dog Timer	–	–	√	√
Communication	C-MAC	≈	√	√	–
	TSTP	–	–	–	√
	IEEE 802.15.4	–	–	–	√
	ELP	≈	√	√	–

	ADHOP	≈	√	√	−
	TCP/IP	≈	√	√	√
	SIP	−	≈	√	−
	RTP	−	≈	√	−
	IEEE 1415	−	≈	√	−
Power	Power Management API	√	√	√	≈
	Energy-aware Scheduling	√	√	√	≈
	Energy-aware, Real-time Scheduling	√	√	√	≈
Development Tools	GCC 4.0.x	√	√	√	−
	GCC 4.4.x	√	√	√	√
	QEMU	√	√	√	√
	GDB on QEMU	−	√	√	√

2. Setting up EPOS

2.1. Downloading EPOS

You can download OpenEPOS releases from the download page and development versions from EPOS SVN Server.

2.2. Downloading the toolchain

2.2.1. GCC

Recent versions of EPOS can go with any (recent) GCC version. However, since EPOS is itself the operating system, the compiler cannot rely on a libc compiled for another OS (such as LINUX). A cross-compiler is needed even if your source and target machines are x86-based PCs. You can download a precompiled GCC for EPOS from the download page or compile a newlib-based toolchain yourself following these instructions.

2.2.2. as86

If you don't have the "as86" command installed, you need to install the dev86 package (or bin86 in Ubuntu).

2.2.3. 32-bit libs

If you use a 64-bit operating system, you will need to install a set of 32-bit libs. In **Ubuntu** 64, you need these packages:
ia32-libs lib32stdc++6 libc6-i386 libc6-dev-i386

As noted in the comments: "If package ia32-libs is not available, the following packages replace it: lib32z1 lib32ncurses5 lib32bz2-1.0"

In **Fedora**, install these packages:
glibc-devel.i686 libstdc++.i686 libstdc++-devel zlib.i686

2.3. Installing

Simply open a release tarball or check out a version from the SVN server at the place you want EPOS to be installed. You don't need to bother about the chosen path nor set any environment variable. EPOS is fully self contained.

If you also downloaded a toolchain tarball, open it at /usr/local/<architecture> whenever possible. For instance, if you downloaded the ia32 toolchain, you should extract it at **/usr/local/ia32/gcc-4.4.4**.
If you downloaded the arm toolchain for EPOSMote III, you should extract it at /usr/local/arm/gcc-4.4.4

If you do not have access to that path, you'll have to adjust the makedefs file in EPOS' main directory accordingly.

3. Running EPOS

3.1. Compiling

At the directory where you installed EPOS' source code, just type:

```
$ make
```

The system will be configured and compiled (i.e. generated) successive times for each application found in the `app` directory. Both software and hardware components will be generated according with each application needs and stored in the `img` directory.

If you have multiple applications or multiple deployment scenarios, but want to operate on a single one, you can specify it using the APPLICATION parameter like this:

```
$ make APPLICATION=philosophers_dinner
```

If everything goes right, you should end with something like this:

```
EPOS bootable image tool

  EPOS mode: library
  Machine: cortex_m
  Model: emote3
  Processor: armv7 (32 bits, little-endian)
  Memory: 31 KBytes
  Boot Length: 0 - 0 (min - max) KBytes
  Node id: will get from the network

  Creating EPOS bootable image in "hello.img":
    Adding application "hello": done.

  Adding specific boot features of "cortex_m": done.

  Image successfully generated (52686 bytes)!
```

3.2. Running

First of all, you'll need to install a platform specific back-end for EPOS to run on. During development, this is usually a QEMU virtual machine for your target architecture (e.g. qemu-system-i386, qemu-system-arm). Then, simply type

```
$ make [APPLICATION=<application>] run
```

Note: for the EPOSMote III platform, please refer to the EPOSMote III Programming Tutorial.

3.3. Configuring

Trait classes are EPOS main configuration mechanism. Whenever an application-specific instance of EPOS is produced (that is, whenever EPOS is built), the builder looks for a file named `$APPLICATION_traits.h` in the `app` directory. For instance, if the application's main file is `app/producer_consumer.cc`, then the builder will look for a file named `app/producer_consumer_traits.h` to configure EPOS accordingly. If such file does not exist, then the building will take a default configuration from `include/system/traits.h` (mostly for quick starters or testing, since a valina configuration makes little sense for an application-specific framework).

Detailed information about the Traits of each component in EPOS is given in section 4, but a typical traits file usually looks like this:

```
#ifndef __traits_h
#define __traits_h

#include <system/config.h>

__BEGIN_SYS

// Global Configuration
template<typename T>
struct Traits
{
    static const bool enabled = true;
    static const bool debugged = false;
    static const bool hysterically_debugged = false;
    typedef TLIST<> ASPECTS;
};

template<> struct Traits<Build>
{
    enum {LIBRARY, BUILTIN, KERNEL};
    static const unsigned int MODE = LIBRARY;

    enum {IA32, ARMv7};
    static const unsigned int ARCHITECTURE = IA32;

    enum {PC, Cortex_M, Cortex_A};
    static const unsigned int MACHINE = PC;

    enum {Legacy, eMote3, LM3S811};
    static const unsigned int MODEL = Legacy;

    static const unsigned int CPUS = 8;
    static const unsigned int NODES = 1; // > 1 => NETWORKING
};

// Utilities
template<> struct Traits<Debug>
{
    static const bool error   = true;
    static const bool warning = true;
    static const bool info    = false;
    static const bool trace   = false;
};
...

__END_SYS

#endif
(END)
```

In order to enable debugging, a programmer would make:

```
Traits<T>::debugged = true
```

In order to collect execution traces, he would make:

```
Traits<Debug>::trace = true
```

To build for EPOSMote II, he would adjust `Traits<Build>` like this:

```
template<> struct Traits<Build>
{
    enum {LIBRARY, BUILTIN, KERNEL};
    static const unsigned int MODE = LIBRARY;

    enum {IA32, ARMv7};
    static const unsigned int ARCHITECTURE = ARMv7;

    enum {PC, Cortex_M, Cortex_A};
    static const unsigned int MACHINE = Cortex_M;

    enum {Legacy, eMote3, LM3S811};
    static const unsigned int MODEL = eMote3;

    static const unsigned int CPUS = 1;
    static const unsigned int NODES = 1;
};
```

Note: from EPOS 2.0, even makefile customizations are no longer needed. Makefiles now parse the application's traits to adjust themselves and produce a proper instance of EPOS.

Note: after changing the traits of any application, issue a `make veryclean` to clean up internal configuration info.

4. EPOS API

EPOS programming API is defined around a set of reusable components that are modeled and implemented following the ADESD methodology. Whenever possible, componentes implement constructs that are well-established in the OS community, so you can usually refer to the classic systems literature to understand EPOS components. Software components, also called abstractions, are platform-independent and are encapsulated as C++ classes. Platform-specific elements are encapsulated as *Hardware Mediators*, which are functionally equivalent to device drivers in Unix, but do not build a traditional HAL. Instead, they sustain the interface contract between abstractions and hardware components by means of static metaprogramming techniques. Mediators gets dissolved or embedded into abstractions at compile-time. EPOS also offers common data structures, such as lists, vectors, and hash tables, through a set o class utilities.

4.1. Memory Management

Most embedded applications won't require programmers to directly manage memory. When EPOS is in *LIBRARY* mode, which implies in disabling multitasking support (multithreading and multicore are still allowed), it automatically arranges for an address space for the (single) application with code and data segments. The data segment is adjusted to incorporate all the memory available in the system and a *heap* is created to export that memory to programmers. In this way, programmers can simply allocate and release memory using the corresponding C++ operators.

Nonetheless, EPOS provides a comprehensive set of memory abstractions, including **Address Spaces** (for multitasking environments, in which each Task has its own address space), adjustable memory **Segments**, **DMA Buffers**, and support to dedicated memory devices, such as **Scratchpad** and **Flash**.

4.1.1. Dynamic Memory (Heap)

Dynamic memory allocation is supported in EPOS through the ordinary C++ operators **new** and **delete**. The default algorithm implemented by EPOS is the Buddy Allocator. Some examples of memory allocation and release in EPOS are depicted bellow. All valid C++ heap operations are also valid EPOS memory allocation operations.

```
Thread * thread = new Thread(&function);
Mutex * mutex = new Mutex;

int ** matrix = new int*[ROWS];
for(int i = 0; i < ROWS; ++i)
    matrix[i] = new int[COLUMNS];

for(int i = 0; i < ROWS; ++i)
    delete [] matrix[i];
delete [] matrix;

delete mutex;
delete thread;
```

As mentioned before, the Heap in non-multitasking configurations contains all the memory available to applications in the machine. For multitasking or explicit multiheap configurations, the default size of a Heap is defined by a machine trait: `Traits<Application>::HEAP_SIZE` .

Note: Most application traits reuse architecture and machine traits simply by including default values from system files, so the trait mentioned here might be a forward to another trait, in this case `Traits<Machine>::HEAP_SIZE` in `include/machine/MACHINE_NAME/MODEL_NAME_traits.h` .

Note: Differently from UNIX, EPOS does not automatically extends the Heap in multitasking configurations when it is depleted. This an unusual situation in an embedded system. However, programmers can explicitly resize the data segment and feed the Heap with additional memory by invoking `Heap::free()` .

4.1.2. Stacks

Each **Thread** in EPOS has its own stack, which is allocated from the **Heap** during instantiation. The default size for such stacks is carefully defined for each combination of architecture and machine through the `Traits<Application>::STACK_SIZE` trait. A Thread can also have the size of its Stack defined at creation time.

Note: Most application traits reuse architecture and machine traits simply by including default values from system files, so the trait mentioned here might be a forward to another trait, in this case `Traits<Machine>::STACK_SIZE` in `include/machine/MACHINE_NAME/MODEL_NAME_traits.h`.
The figure below shows an example that exposes the relationship mentioned above.

4.1.3. Memory Segments

Memory segments are chunks of allocated memory ready to be used by applications. In order to be actually used by applications, a Segment must be attached to an Address Space. Per-se, it is only an allocation unit.

This abstraction is of little use for single-task configurations using the *LIBRARY* mode, since all the memory available in the machine is injected into the Heap at initialization time. However, for other configurations, or for Segments designating I/O regions, it delivers a high-level interface for both main memory and I/O devices. A Segment can be mapped to any (large enough) slot in an Address Space. It can also be dynamically grown or shrinked. Resize operations, however, will fail if the Segment is created with the CT flag (contiguous) and there is no adjacent slots to fulfill the request.
Header
`include/segment.h`
Interface

```
class Segment
{
public:
    typedef MMU::Flags Flags;
    typedef CPU::Phy_Addr Phy_Addr;

public:
    Segment(unsigned int bytes, Flags flags = Flags::APP);
    Segment(Phy_Addr phy_addr, unsigned int bytes, Flags flags);
    ~Segment();

    unsigned int size() const;
    Phy_Addr phy_address() const;
    int resize(int amount);
};
```

Methods

- `Segment(unsigned int bytes, Flags flags = Flags::APP)`
  Creates a memory segment of `bytes` bytes. Meaningful `flags` are:
  - `RW` : read-write (read-only if absent)
  - `CWT` : cache write-through (write-back if absent)
  - `CD` : cache disable (cached if absent)
  - `CT` : contiguous (scattered if absent)
  - `APP` : application default flags (to be always ORed)

  Note: this method can cause the fatal error `Out of Memory` in case the allocation goes beyond the system's capability.

- `Segment(Phy_Addr phy_addr, unsigned int bytes, Flags flags)`
  Encapsulates a memory region of `bytes` bytes starting at `phy_addr` as a Segment that can be attached to an Address Space. This constructor does not allocate memory. It simply maps a preexisting memory region as a Segment. In addition to the flags described above, this constructor can take:
  - `IO` : memory mapped I/O (main memory if absent)
- `~Segment()`
  Destroys a segment, releasing the associated memory (unless the segment was created with `IO` , case in which only the corresponding page tables are released).
- `unsigned int size() const`
  Returns the current size of a Segment.
- `Phy_Addr phy_address() const`
  Returns the physical address of contiguous Segment (i.e. a Segment created with `CT`). Requesting the physical address of a scattered segment is invalid and returns `Phy_Addr(false)` .
- `int resize(int amount)`
  Grows or shrinks a Segment by `amount` bytes. A contiguous Segment (i.e. a Segment created with `CT`) can only be expanded using adjacent memory blocks.

  Note: this method can cause the fatal error `Out of Memory` in case the allocation goes beyond the system's capability.

4.1.4. Address Spaces

An Address Space abstracts the range of CPU addresses valid for a given Process. Segments must be attached to an Address Space in order to be accessed. Each Task has its own Address Space.

For single-task configurations using the LIBRARY mode, a virtual Task is created during system initialization. Explicitly accessing this Address Space is rather unconventional, but it can be accessed like this:

```
Address_Space * as = new Address_Space(MMU::current());
```

For multitasking configuration, the current Address Space can be obtained with:

```
Address_Space * as = Task::self()->address_space();
```

Header
`include/address_space.h`
Interface

```
class Address_Space
{
public:
    Address_Space();
    Address_Space(MMU::Page_Directory * pd);
    ~Address_Space();

    using MMU::Directory::pd;

    Log_Addr attach(Segment * seg);
    Log_Addr attach(Segment * seg, const Log_Addr & addr);
    void detach(Segment * seg);
    void detach(Segment * seg, const Log_Addr & addr);

    Phy_Addr physical(const Log_Addr & address);
};
```

Methods

- `Address_Space()`
  Creates an Address Space, usually for a new Task.
- `Address_Space(MMU::Page_Directory * pd)`
  Returns a reference to an Address Space using `pd` as the primary page table.
- `~Address_Space()`
  Destroys an Address Space
- `~MMU::Page_Directory * pd()`
  Returns the current primary page table (called page directory by Intel).
- `Log_Addr attach(Segment * seg)`
  Attaches the Segment designated by `seg` at the first available address. If the target Address Space does not feature any slot large enough to contain the Segment, then `Log_Addr(false)` is returned.
- `Log_Addr attach(Segment * seg, const Log_Addr & addr)`
  Attaches the Segment designated by `seg` at address `addr` . If the target address is already mapped, the `Log_Addr(false)` is returned.
- `void detach(Segment * seg)`
  Detaches the Segment designated by `seg` from the Address Space. Detaching a Segment that has not been previously attached might be a harmful operation in some architectures.
- `void detach(Segment * seg, const Log_Addr & addr)`
  Detaches the Segment designated by `seg` from `addr` at the Address Space. Detaching a Segment that has not been previously attached or detaching it from a different address might be a harmful operation in some architectures.
- `Phy_Addr physical(const Log_Addr & addr)`
  Returns the physical address currently bound to `addr` in the Address Space.

4.2. Process Management

In EPOS, process management is accomplished by three components: **Task**, **Thread**, and **Scheduler**. They were designed and implemented to match the corresponding concepts described in the classic systems literature. The isolation of scheduling policies from the implementation of process defines an elegant framework for future developments. This design is discussed in depth in this paper.

4.2.1. Task

If a process is a program in execution, then a Task is the static portion of that process, encompassing its code and data segments, while a Thread abstracts its dynamic aspects, featuring a private context and stack. A Thread is thus said to run on a Task. Each Task has its own Address Space. Segments can be attached to any Address Space and thus can be shared among Tasks. Threads are handled independently of belonging to the same Task or to different ones.

Note: for single-task configurations using the LIBRARY mode, a virtual Task is created during system initialization. Explicitly accessing this Task is rather unconventional, but it can be done using the `Task::self()` method.

Header

`include/task.h`

Interface

```
class Task
{
    template<typename ... Tn>
    Task(Segment * cs, Segment * ds, int (* entry)(Tn ...), Tn ... an);
    template<typename ... Tn>
    Task(const Thread::Configuration & conf, Segment * cs, Segment * ds,
         int (* entry)(Tn ...), Tn ... an);
    ~Task();

    Address_Space * address_space();

    Segment * code_segment();
    Segment * data_segment();

    Log_Addr code();
    Log_Addr data();

    Thread * main();

    static Task * volatile self();
}
```

Methods

- `template<typename ... Tn>`
  `Task(Segment * cs, Segment * ds, int (* entry)(Tn ...), Tn ... an)`
  Creates a Process by implicitly creating a Task and a Thread. The Task is created with the code Segment given by `cs` and the data Segment given by `ds`. Thread is created to run the function given by `entry` on the associated Task. The C++ parameter pack is consistently passed to the Thread following the architecture's call convention (stack, register set, window, etc).

Note: the code Segment is mapped to the new Task's Address Space in accordance with the memory model in place (defined the the application's Traits), so it is usually possible to assume `entry` is a valid address within the code Segment. Nevertheless, this is an assumption for the method and programmers are to ensure it for any exotic scenario.

- `template<typename ... Tn> Task(const Thread::Configuration & conf, Segment * cs, Segment * ds, int (* entry)(Tn ...), Tn ... an)`
  This constructor is similar the the previous, but takes an addition Configuration pack (see Thread for details).
- `~Task()`
  Destroys a Task and consequently deletes (i.e. kills) all its Threads.
- `Address_Space * address_space()`
  Returns the Task's Address Space.
- `Segment * code_segment()`
  Returns the Task's code Segment.
- `Segment * data_segment()`
  Returns the Task's data Segment.
- `Log_Addr code()`
  Returns the address the Task's code Segment is mapped to in its Address Space.
- `Log_Addr data()`
  Returns the address the Task's data Segment is mapped to in its Address Space.
- `Thread * main()`
  Returns the address of the function used to create the Task's first Thread (usually the function `main()`).
- `static Task * volatile self()`
  Returns a reference to the running Task.

4.2.2. Thread

If a process is a program in execution, then a Thread encompasses its dynamic aspects. A Thread has a private context and a private stack. It runs a Task's code and manipulate its data.
Header
`include/thread.h`
Interface

```
class Thread
{
public:
    enum State {
        RUNNING,
        READY,
        SUSPENDED,
        WAITING,
        FINISHING
    };

    typedef Scheduling_Criteria::Priority Priority;
    typedef Traits<Thread>::Criterion Criterion;
    enum {
        HIGH    = Criterion::HIGH,
        NORMAL  = Criterion::NORMAL,
        LOW     = Criterion::LOW,
        MAIN    = Criterion::MAIN,
        IDLE    = Criterion::IDLE
    };

    struct Configuration {
        State state;
        Criterion criterion;
        Task * task;
        unsigned int stack_size;
    };

    template<typename ... Tn>
    Thread(int (* entry)(Tn ...), Tn ... an);
    template<typename ... Tn>
    Thread(const Configuration & conf, int (* entry)(Tn ...), Tn ... an);
    ~Thread();

    const volatile State & state();

    const volatile Priority & priority();
    void priority(const Priority & p);

    Task * task();

    int join();
    void pass();
    void suspend();
    void resume();

    static Thread * volatile self();
    static void yield();
    static void exit(int status = 0);
}
```

Types

- `State`
  Defines the states a Thread can assume.
  - `RUNNING` : the Thread is running on a CPU.
  - `READY` : the Thread is ready to be executed, but there are no available CPUs at the moment.
  - `SUSPENDED` : the Thread is suspended and therefore it is not eligible to be scheduled.
  - `WAITING` : the Thread is blocked waiting for a resource (e.g. Semaphore, Communicator, File).
  - `FINISHING` : the Thread called exit and its state is being held for an eventual `join()` .

- `Priority`
  Defines an integer representation for the priorities a Thread can assume.

- `Criterion`
  Defines the priorities a Thread can assume. It is usually an import of a type defined in the `Scheduling_Criteria` namespace. It is used by EPOS as the ordering criterion for all scheduling decisions. Many priorities can have symbolic representations. The following are the most typical ones:
  - `HIGH` : the highest priority an user-level Thread can have.
  - `LOW` : the lowest priority an user-level Thread can have.
  - `NORMAL` : the priority assigned to Threads by default.
  - `MAIN` : the priority assigned to the first Thread of a Task (usually running the `main()` function). It is often an alias for `NORMAL` .
  - `IDLE` : the Idle Thread priority (usually `LOW` - 1).

- `Configuration`
  This type is used to define a configuration pack for Threads. The following parameters can be adjusted:
  - `state` : designates the Thread's initial state. `READY` is the default. `SUSPENDED` can be used to prevent scheduling after creation. A Thread created as `SUSPENDED` must be explicitly activated with `resume()` .
  - `criterion` : designates the Thread's initial priority. `NORMAL` is the default. Any value between `LOW` and `MAX` , or any Criterion mapping to that interval is valid.
  - `task` : can be (rarely) used to create a Thread over another Task. Default is to create a Thread on the currently running Task.
  - `stack_size` : designates the size in bytes of Thread's Stack. The default is `Traits<Application>::STACK_SIZE` .

Methods

- `template<typename ... Tn>`
  `Thread(int (* entry)(Tn ...), Tn ... an)`
  Creates a Thread on the running Task to run the function given by `entry` . The C++ parameter pack is consistently passed to the Thread following the architecture's call convention (stack, register set, window, etc).

- `template<typename ... Tn>`
  `Thread(const Configuration & conf, int (* entry)(Tn ...), Tn ... an)`
  Creates a Thread on the running Task to run the function given by `entry` . The Thread's

creation is controlled by `conf` (see the `Configuration` type declaration above). The remainder of the C++ parameter pack is consistently passed to the Thread following the architecture's call convention (stack, register set, window, etc).

- `~Thread()`
  Destroys a Thread (respecting the corresponding C++ object's semantics; e.g. the destructor does not delete the object).
- `const volatile State & state()`
  Returns the Thread's current state.
- `const volatile Priority & priority()`
  Returns the Thread's current priority (i.e. an integer representing the ordering imposed by the Criterion in place).
- `void priority(const Priority & p)`
  Adjusts the Thread's priority according to the Criterion in place.
- `Task * task()`
  Returns the Task this Thread is running on.
- `int join()`
  Waits for this Thread to finish and returns the value passed over at `return` (or `exit()`).

  Note: the `int` return type is defined by the C++ standard as the only one valid for the `main()` function and therefore requires EPOS to follow it. POSIX further limits the interpretation of that integer to 8 bits. EPOS would prefer to abolish it if there were a `void main()` valid signature. Programmers can define their own semantics for the integer in EPOS.
- `void pass()`
  Hands the CPU over to this Thread. This function can be used to implement user-level schedulers. A Thread can be created with a higher priority to act as the scheduler. EPOS scheduler will always elect it, but it can in turn `pass()` the CPU to another Thread. Accounting is done for the Thread receiving the CPU, but timed scheduling criteria are not reset. In this way, the calling Thread is charged only for the time it took to hand the CPU over to another Thread, which inherits the CPU without further intervention from EPOS' scheduler.
- `void suspend()`
  Suspends the execution of this Thread. The Thread's state is set to `SUSPENDED` and it will not be eligible for scheduling until `resume()` is called. If called for the running Thread, this method triggers a rescheduling.
- `void resume()`
  Resumes the execution of this Thread by setting its state to `READY` and notifying the Scheduler. Whether or not this notification will trigger a reschedule is Criterion dependent. Resuming a Thread that is not suspended is an invalid operation, even if for most policies this would have no effects.
- `static Thread * volatile self()`
  Returns a reference to the running Thread (actually, a volatile referente to a pointer designating it).

- `static void yield()`
  Yields the CPU by triggering a reschedule operation the excludes the running Thread from election. If the scheduler can find another Thread to take over the CPU, then the calling Thread's state is set to `READY` and the that Thread is put to run. Since the Thread yielding the CPU is in `READY` state it can be rescheduled at any subsequent time.

- `static void exit(int status = 0)`
  Causes the termination of the calling Thread, which has its state set to `FINISHING`. The Thread's context is preserved for an eventual `join()` operation (util the corresponding C++ object is deleted). If there is already a pending `join()` at the time `exit()` is called, then the waiting Thread is reactivated (i.e. its state is set to `READY` and the scheduler is notified). This method always triggers a reschedule.

4.2.3. RT_Thread

The Real-time Thread abstraction is an specialization of Thread designed to handle a variety of scenarios in the realm or Periodic Real-time Scheduling.
Header
`include/periodic_thread.h`
Interface

```
class RT_Thread
{
public:
    typedef RTC::Microsecond Microsecond;

    enum { INFINITE = RTC::INFINITE };

    enum {
        SAME    = Scheduling_Criteria::RT_Common::SAME,
        NOW     = Scheduling_Criteria::RT_Common::NOW,
        UNKNOWN = Scheduling_Criteria::RT_Common::UNKNOWN,
        ANY     = Scheduling_Criteria::RT_Common::ANY
    };

public:
    RT_Thread(void (* function)(),
              const Microsecond & deadline,
              const Microsecond & period = SAME,
              const Microsecond & capacity = UNKNOWN,
              const Microsecond & activation = NOW,
              int times = INFINITE,
              int cpu = ANY,
              unsigned int stack_size = STACK_SIZE);
}
```

Methods

- `RT_Thread(void (* function)(), const Microsecond & deadline, const Microsecond & period = SAME, const Microsecond & capacity = UNKNOWN, const Microsecond & activation = NOW, int times = INFINITE, int cpu = ANY, unsigned int stack_size = STACK_SIZE)`
  Creates a Periodic Thread on the running Task to run the function given by `function`. These are the constructor's parameters:

- `deadline` : the Periodic Thread's deadline in μs.
- `period` : the Periodic Thread's period in μs (a new *job* is released at every `period` μs). `SAME` makes it equal to the Thread's deadline.
- `capacity` : designates the time each *job* takes to finish. This is only meaningful for a few real-time algorithms (i.e. Scheduling Criteria) and is usually given as a Worst Case Execution Time estimate for the Thread's *jobs*. Leave it as `UNKNOWN` for Criteria that does not use it.
- `activation` : a time to wait before releasing the Thread's first *job* (i.e. before activating it).
- `times` : periodic threads usually run forever and have this parameter passed as `INFINITE` . You can restrict the number of job releases with this parameter.
- `cpu` : some multicore Scheduling Criteria allows programmers to specify the first CPU the Thread will run on. Some of them, the partitioned ones, will even restrict the execution of subsequent Thread's *jobs* to that CPU.
- `stack_size` : designates the size in bytes of Thread's Stack. The default is Traits<Application>::STACK_SIZE.

4.2.4. Scheduler

EPOS provides a family of schedulers that covers a large variety of scenarios, from ordinary time-sharing algorithms to sophisticated real-time, energy-aware multicore ones. EPOS Scheduler can be instantiated multiple times to schedule different classes of resources, such as disks and networks, but each resource class has a single scheduler. In order to select the Thread Scheduler (or CPU Scheduler, depending on your perspective) simply pick one of them from the `Scheduling_Criteria` namespace and edit your application's Traits file to designate it as `Traits<Thread>::Criterion` .

There are four basic Traits for a Thread Scheduling Criterion: preemptive, timed, dynamic, and energy-aware:

- Preemptive: a Preemptive Criterion requires a reevaluation, and eventually a rescheduling, whenever a Thread enters the `READY` state, independently of the previous state (e.g. a newly created Thread, a Thread released from a Mutex, a Thread that was waiting fro I/O). A non-preemptive Criterion will only be reevaluated when the `RUNNING` Thread explicitly causes it state to change (e.g. by blocking on a Synchronizer or by invoking I/O operations). All timed Criteria are preemptive. Most priority-based Criteria are also preemptive. *Shortest Job First* is a non-Preemptive Criterion.
- Timed: a Timed Criterion requires a `QUANTUM` to be specified (in μs). This constant defines the maximum time a Thread can run before the Scheduler rechecks the Criterion in place (eventually scheduling another Thread). The valeu of `Traits<Thread>::QUANTUM` must be carefully chosen: a value of a few μs will cause the system to reevaluate the Criterion too often and will result in (very) large overhead, eventually bringing the system to thrash; a value of hundreds of ms will enable CPU-bound threads to monopolize the CPU, eventually degrading the system responsiveness. Values between 100 μs and 100 ms are common. All timed Criteria are preemptive. *Round-robin* is a Timed Criterion.
- Dynamic: a Dynamic Criterion is recalculated at run-time to constantly reflect the police in force. There are two moments at which a Dynamic Criterion can be recalculated: at dispatch and at release. For Aperiodic Threads, for which no period is defined, it is done when the Thread leaves the CPU (i.e. another Thread is dispatched). For Periodic Threads, recalculating at dispatch would not be adequate, since jobs of other Threads will still be released before the

next activation and they may influence on the calculations. Therefore, Periodic Threads subjected to Dynamic Criteria are reevaluated before the release of each *job*. Earliest Deadline First is Dynamic Criterion.
- Energy-aware: Criteria with this trait will cause low priority Threads to be suspended whenever their execution could cause a critical Thread to fail due to the lack of power. In order to enforce such a regimen, Energy-aware Criteria require `Traits<System>::LIFE_SPAN` to be defined. An energy monitoring mechanism is also enabled in the platforms supporting it.

The following are EPOS standard Scheduling Criteria. Many others exist and implementing yours is not difficult.

- `FCFS` : First-come, First Served (FIFO)
- `Priority` (Static and Dynamic)
- `RR` : Round-Robin
- `GRR` : Multicore Round-Robin
- `CPU Affinity` (multicore)
- `[G|P|C]RM` : Rate Monotonic (single-core and global, partitioned or clustered multicore)
- `[G|P|C]DM` : Deadline Monotonic (single-core and global, partitioned or clustered multicore)
- `[G|P|C]EDF` : Earliest Deadline First (single-core and global, partitioned or clustered multicore)
- `[G|P|C]LLF` : Least Laxity First (single-core and global, partitioned or clustered multicore)

4.3. Process Coordination (Synchronizers)

Process coordination in EPOS is realized by the Synchronizer and the Communicator families of abstractions. The former is described here and the latter in the next section.

Synchronizers are used to coordinate process execution so concurrent (or parallel) Threads can share resources without corrupting them. . avoid race conditions during the execution of parallel programs. A race condition occurs when a thread accesses a piece of data that is being modified by another thread, obtaining an intermediate value and potentially corrupting that piece of data.

4.3.1. Semaphore

The Semaphore member of the Synchronizer family realizes a semaphore variable as invented by Dijkstra. A semaphore variable is an integer variable whose value can only be manipulated indirectly through the atomic operations `p()` and -=v()+-.

Note: besides being useful to synchronize critical sections, Semaphores can be also used as atomic resource counters as in the Producer-consumer problem.

Header

`include/semaphore.h`

Interface

```
class Semaphore
{
public:
    Semaphore(int v = 1);
    ~Semaphore();

    void p();
    void v();
}
```

Methods

- `Semaphore(v : int = 1)`
  Creates a Semaphore, which, by default, is initialized with 1.
- `~Semaphore()`
  Destroys a Semaphore, releasing eventual blocked Threads.
- `p()`
  Atomically decrements the value of a semaphore. Invoking `p()` on a semaphore whose value is less than or equal to zero causes the Thread to wait until the value becomes positive again.
- `v()`
  Atomically increments the value of a Semaphore, eventually unblocking a waiting Thread if the value becomes positive (i.e. making its state `READY` and notifying the Scheduler).

4.3.2. Mutex

The Mutex member of the Synchronizer family implements a Binary Semaphore.
Header
`include/mutex.h`
Interface

```
class Mutex
{
public:
    Mutex();
    ~Mutex();

    void lock();
    void unlock();
}
```

Methods

- `Mutex()`
  Creates a Mutex.
- `~Mutex()`
  Destroys a Mutex, releasing eventual blocked Threads.
- `lock()`
  Locks a Mutex. Subsequent invocations cause the calling Threads to block.

- `unlock()`
  Unlocks a Mutex. When a Thread invokes `unlock()` on a Mutex for which there are blocked Threads, the first Thread put to wait is unblocked (by making its state `READY` and notifying the Scheduler) and the Mutex is immediately locked (atomically). If no threads are waiting, the unlock operation has no effect.

4.3.3. Condition

The Condition member of the Synchronizer family realizes a system abstraction inspired on the condition variable language concept, which allows a Thread to wait for a predicate on shared data to become true. It is often used by programming languages to implement Monitors.
Header
`include/condition.h`
Interface

```
class Condition
{
public:
    Condition();
    ~Condition();

    void wait();
    void signal();
    void broadcast();
}
```

Methods

- `Condition()`
  Creates a condition variable.

- `~Condition()`
  Destroys a condition variable, releasing eventual blocked Threads.

- `wait()`
  Implicitly unlocks the shared data and puts the calling Thread to wait for the assertion of a predicate. Several threads can be waiting on the same condition. The assertion of a predicate can be announced either to the first blocked Thread or to all blocked Threads. When a thread returns from the wait operation, it implicitly regains control over the critical section.

- `signal()`
  Announces the assertion of a predicate to the first waiting Thread, releasing it for execution (i.e. making its state `READY` and notifying the Scheduler).

- `broadcast()`
  Announces the assertion of a predicate to all waiting Threads, making their state `READY` and notifying the Scheduler.

4.4. Timing

Time management in EPOS encompasses abstractions to measure time intervals, to keep track of the current time, and also to trigger timed events.

4.4.1. Clock

- Clock abstracts a *Real-time Clock* (RTC) in platforms that feature one. It can be used to get and set the current time and date.

Header

`include/clock.h`

Interface

```
class Clock
{
public:
    typedef RTC::Microsecond Microsecond;
    typedef RTC::Second Second;

    class Date {
    public:
        Date() {}
        Date(unsigned int Y, unsigned int M, unsigned int D,
             unsigned int h, unsigned int m, unsigned int s);
        Date(const Second & seconds, unsigned long epoch_days = 0);

        operator Second();
        Second to_offset(unsigned long epoch_days = 0);

        unsigned int year();
        unsigned int month();
        unsigned int day();
        unsigned int hour();
        unsigned int minute();
        unsigned int second();

        void adjust_year(int y);
    }

public:
    Clock();
    ~Clock();

    Microsecond resolution();

    Second now();

    Date date();
    void date(const Date & d);
}
```

Types

- `Microsecond` : an unsigned integer representing μs. Whenever possible, its resolution (number of bits) is adjusted according to `Traits<System>::LIFE_SPAN` .
- `Second` : an unsigned integer representing seconds. Whenever possible, its resolution (number of bits) is adjusted according to `Traits<System>::LIFE_SPAN` .

- `Date` : data structure to store the components of a date: year, month, day, hour, minute, and second; as unsigned integers. It features methods to convert this representation of data to and from an offset in seconds from a given epoch

Methods

- `Clock()`
  Constructs a Clock.
- `~Clock()`
  Destroys a Clock.
- `Microsecond resolution()`
  Returns the Clock resolution in µs.
- `Second now()`
  Returns the current time in seconds.
- `Date date()`
  Returns the current date.
- `void date(Date & d)`
  Sets the current date.

4.4.2. Chronometer

Chronometer abstracts a timepiece able to measure time intervals. Its precision and resolution depends on the timing devices available in the platform (e.g. real-time clocks, CPU clock counters, high-performance timers).
Header
`include/chronometer.h`
Interface

```
class Chronometer
{
public:
    typedef TSC::Hertz Hertz;
    typedef RTC::Microsecond Microsecond;

public:
    Chronometer();
    ~Chronometer();

    Hertz frequency();

    void reset();
    void start();
    void lap();
    void stop();

   Microsecond read();
}
```

Methods

- `Chronometer()`
  Constructs a Chronometer.
- `~Chronometer()`
  Destroys a Chronometer.
- `Hertz frequency()`
  Returns the Chronometer frequency in Hertz.
- `void reset()`
  Resets the Chronometer.
- `void start()`
  Starts counting time. Can be used only once for each counting procedure. Subsequent invocations are ignored (use `reset()` before using `start()` again).
- `void lap()`
  Takes a snapshot of the current time counting. A `read()` will return the interval accumulated for all laps since `start()` . Time counting continues normally.
- `void stop()`
  Stops counting time. A `read()` will return the interval elapsed since `start()` .
- `Microsecond read()`
  Return the measured time in µs. Before `start()` the method returns 0. After `start()` it returns the time measured between `start()` and the last `lap()` or `stop()` .

4.4.3. Alarm

Alarm abstracts timed events in EPOS. An Alarm uses a hardware timer to trigger high-level timed events. These events are abstracted by the `Handler` utility, which declares an interface for polymorphic objects that implement the call operator `void operator()()` - (see Handler. An event Handler can be a function or any other object implementing its interface. For example, a Thread Handler holds a reference to a Thread and binds the call operator to `resume()` . A Semaphore Handler holds a reference to a Semaphore and binds the call operator to `v()` . In this case, the Handler itself is Thread synchronized on that Semaphore.

Note: A Sempahore Handler is particularly interesting for it has memory: if an event cannot be handled in time, it will be stored handled lately (as soon as the the first occurrence gets handled and the scheduler allows). Other Handlers might lose late events.
Header
`include/alarm.h`
Interface

```
class Alarm
{
public:
    typedef RTC::Microsecond Microsecond;

    enum { INFINITE = RTC::INFINITE };

public:
    Alarm(const Microsecond & time, Handler * handler, int times = 1);
    ~Alarm();

    static Hertz frequency();

    static void delay(const Microsecond & time);
}
```

Methods

- `Alarm(const Microsecond & time, Handler *handler, int times = 1)`
  Creates an Alarm to trigger `handler` after `time` μs. The event will occur `times` times or forever if `INFINITE` is given. Handler must be polymorphic and it must implement the call operator (-+void operator()()+--) for the trigger. See the Handler utility for details.

- `~Alarm()`
  Destroys an Alarm;

- `Hertz frequency()`
  Returns the Alarm's frequency in Hertz.

- `void delay(const Microsecond & time)`
  Delays a Thread execution by `time` μs.

4.4.4. Delay

Delay is used to delay the execution of Threads by a given, usually small, amount of time.
Header
`include/alarm.h`
Interface

```
class Delay
{
public:
    Delay(const Microsecond & time);
}
```

Methods

- `Delay(const Microsecond & time)`
  Creates a Delay object to delay the execution of the calling Thread by `time` μs. The object is implicitly destroyed afterwards and there are no methods to act on it meanwhile.

4.5. Communication

Communication in EPOS is delegated to the families of abstractions shown in the Figure bellow. Application processes communicate with each other using a **Communicator**, which acts as an end-point to a communication **Channel** implemented on a **Network** interfaced by a **Network Interface Card (NIC)**. For example, a TCP connection in EPOS is abstracted as a Communicator for a TCP Channel over an IP Network. Ethernet would be a good candidate for the NIC in this example. Several other protocols have been designed for EPOS, most of them avoiding the issues imposed by TCP/IP on embedded systems, specially critical ones and those for IoT. In order to improve usability, EPOS exports rather different protocols under this same interface, ranging from simple serial ports to TCP/IP and TSTP.

4.5.1. Link

Link is a point-to-point Communicator. Links are used in EPOS to abstract serial communication for a variety of hardware devices, including serial ports such as UART, USART, SPI, and USB. It is also used to create virtual connections on packet switching networks.
Interface

```
template<typename Protocol>
class Link
{
public:
    typedef typename Protocol::Address Address;
    typedef typename Protocol::Address::Local Local_Address;

public:
    Link(const Local_Address & local, const Address & peer = Address::NULL);
    ~Link();

    int read(void * data, unsigned int size);
    int write(const void * data, unsigned int size);

    const Address & peer();
}
```

Methods

- `Link(const Local_Address & local, const Address & peer = Address::NULL)`
  Creates a Link between `local` and `peer` . The calling Thread is blocked until a connection with Peer is established. The local Communicator address is relative to the local host and is given as a `Local_Address` , while the Peer's address must be given as fully qualified `Address` . For some protocols, it is valid to leave `peer` undefined (-+Address::NULL+-) thus indicating that the connection can be established with any Peer. After the connection is established, that address can be retrieved with `peer()` (unless the Network itself does not define addresses, such as for a serial line).

- `~Link()`
  Destroys a Link, properly finishing an eventual connection.

- `int read(void * data, unsigned int size)`
  Reads `size` bytes of data from the Link and stores it at `data` . The calling Thread is blocked until `size` bytes are received.

- `int write(const void * data, unsigned int size)`
  Writes `size` bytes of data starting at `data` into the Link.

- `const Address & peer()`
  Returns the Peer's address. Calling the method before a connection has been established returns `Address::NULL` .

4.5.2. Port

Port is a multi-point Communicator for connection-oriented networks. A Thread can listen on a Port for connection requests from other Threads. Upon connection a Link is returned and both Threads can exchange data. It is widely used with the *Client-Server* architecture, with Servers listening on a Port for Clients' requests.
Header
`include/communicator.h`
Interface

```
template<typename Protocol>
class Port
{
public:
    typedef typename Protocol::Address Address;
    typedef typename Protocol::Address::Local Local_Address;

public:
    Port(const Local_Address & local);
    ~Port();

    Link<Channel> * listen();
    Link<Channel> * connect(const Address & to);
}
```

Methods

- `Port(const Local_Address & local)`
  Creates a Port with address `local` to listen on for connection requests. Creating a Port on a previously assigned Address is invalid for most Protocols.

- `~Port()`
  Destroys the Port, releasing the local address and closing eventually open connections (i.e. Links).

- `Link<Channel> * listen();`
  Listens for a connection request. The calling Thread is blocked until a connection can be established. A Link to the Peer Communicator is returned upon connect.

- `Link<Channel> * connect(const Address & to)`
  Connects to a Port at address `to` . The calling Thread is blocked until a connection can be established. A Link to the peer is returned upon connect. If a connection cannot be established, including because there was already a connection to that address and the underlying Protocol does not support multiple connections, 0 is returned.

4.5.3. Mailbox

A Mailbox is a multi-point Communicator for connectionless Protocols. A Thread can receive messages from a Mailbox and it can also send messages through it to any other Mailbox.

Header

`include/communicator.h`

Interface

```
template<typename Protocol>
class Mailbox
{
public:
    typedef typename Protocol::Address Address;
    typedef typename Protocol::Address::Local Local_Address;

public:
    Mailbox(const Local_Address & local);
    ~Mailbox();

    int send(const Address & to, const void * data, unsigned int size);
    int receive(Address * from, void * data, unsigned int size);
}
```

Methods

- `Mailbox(const Local_Address & local)`
  Creates a Mailbox with address `local` . The Mailbox can be used both to sent and to receive messages. Creating a Mailbox on a previously assigned Address is invalid for most Protocols.
- `~Mailbox()`
  Destroys the Mailbox, releasing the local address.
- `int send(const Address & to, const void * data, unsigned int size)`
  Sends a Message to `to` containing `size` bytes of data stored at `data` . The method returns the number of bytes effectively sent.
- `int receive(Address * from, void * data, unsigned int size)`
  Receives a Message and copies up to `size` bytes of its data to `data` . The calling Thread is blocked until the packet is received. `from` is updated with the address of the sender. The number of bytes effectively received (and copied) is returned.

4.5.4. Channel

Channels in EPOS are used to model communication protocols classified at level four (transport) according to the OSI model. TCP, UDP, ELP, TSTP are Channels. Implementing a new protocol in EPOS is easier than in ordinary Unix-like systems, but nevertheless requires programming knowledge beyond that what could be covered in a User's Guide. Please, refer to our publications for additional information.

4.5.5. Network

Networks in EPOS are used to model communication protocols classified at level three (network) according to the OSI model. IP, ELP, and TSTP are Networks. Implementing a new protocol in EPOS is easier than in ordinary Unix-like systems, but nevertheless requires programming knowledge beyond that what could be covered in a User's Guide. Please, refer to our publications for additional information.

4.5.6. IPC

TODO

4.5.7. TCP/IP

TCP/IP is the standard stack of protocols for communication on the Internet. EPOS implements the protocol stack as specified in the RFCs. Some embedded optimizations described elsewhere are not included in OpenEPOS and therefore this version is fully interoperable with other systems.

4.5.7.1. ARP

The Address Resolution Protocol (ARP) is implemented in EPOS following RFC 826 for Ethernet and likewise for other network technologies. It is implicitly enabled for each NIC for which IP is enabled and there is no user-visible configuration.

4.5.7.2. DHCP

The Dynamic Host Configuration Protocol (DHCP) is implemented in EPOS following RFC 2131. Since it depends on UDP, IP must be initialized first for any NIC using DHCP. See Configuring Networing for details.

4.5.7.3. IP

The Internet Protocol version 4 (IPv4) is implemented in EPOS following RFC 791. An IP Communicator is not defined in EPOS, so applications should not directly use it. For testing and new protocol development, direct IP access can be gained through ICMP. In order to enable IP for a given NIC, list it as the chosen protocol in the applicaiton's Traits. See Configuring Networing for details.

4.5.7.4. ICMP

The Internet Control Message Protocol (ICMP) is implemented in EPOS following RFC 792. An ICMP Communicator is defined in EPOS as Mailbox<ICMP>. See Mailbox above for the general interface. Messages for this Mailbox are ICMP packets specified in the RFC and described bellow.
Header

```
include/icmp.h
```

Interface

```
class ICMP
{
public:
    // ICMP Packet Types
    typedef unsigned char Type;
    enum {
        ECHO_REPLY              = 0,
        UNREACHABLE             = 3,
        SOURCE_QUENCH           = 4,
        REDIRECT                = 5,
        ALTERNATE_ADDRESS       = 6,
        ECHO                    = 8,
        ROUTER_ADVERT           = 9,
        ROUTER_SOLIC            = 10,
        TIME_EXCEEDED           = 11,
        PARAMETER_PROBLEM       = 12,
        TIMESTAMP               = 13,
        TIMESTAMP_REPLY         = 14,
        INFO_REQUEST            = 15,
        INFO_REPLY              = 16,
        ADDRESS_MASK_REQ        = 17,
        ADDRESS_MASK_REP        = 18,
        TRACEROUTE              = 30,
        DGRAM_ERROR             = 31,
        MOBILE_HOST_REDIR       = 32,
        IPv6_WHERE_ARE_YOU      = 33,
        IPv6_I_AM_HERE          = 34,
        MOBILE_REG_REQ          = 35,
        MOBILE_REG_REP          = 36,
        DOMAIN_NAME_REQ         = 37,
        DOMAIN_NAME_REP         = 38,
        SKIP                    = 39
    };

    // ICMP Packet Codes
    typedef unsigned char Code;
    enum {
        NETWORK_UNREACHABLE     = 0,
        HOST_UNREACHABLE        = 1,
        PROTOCOL_UNREACHABLE    = 2,
        PORT_UNREACHABLE        = 3,
        FRAGMENTATION_NEEDED    = 4,
        ROUTE_FAILED            = 5,
        NETWORK_UNKNOWN         = 6,
        HOST_UNKNOWN            = 7,
        HOST_ISOLATED           = 8,
        NETWORK_PROHIBITED      = 9,
        HOST_PROHIBITED         = 10,
        NETWORK_TOS_UNREACH     = 11,
        HOST_TOS_UNREACH        = 12,
        ADMIN_PROHIBITED        = 13,
        PRECEDENCE_VIOLATION    = 14,
        PRECEDENCE_CUTOFF       = 15
    };

    class Address: public IP::Address;

    struct Header
    {
        unsigned char  _type;
        unsigned char  _code;
        unsigned short _checksum;
        unsigned short _id;
        unsigned short _sequence;
    } __attribute__((packed));

    // ICMP Packet
    static const unsigned int MTU = 56; // to make a traditional 64-byte packet
    static const unsigned int HEADERS_SIZE = sizeof(IP::Header) + sizeof(Header);

    typedef unsigned char Data[MTU];

    class Packet: public Header
    {
    public:
        Packet();
        Packet(const Type & type, const Code & code);
        Packet(const Type & type, const Code & code, unsigned short id, unsigned short
seq);

        Header * header();

        template<typename T>
        T * data();

        void sum();
        bool check();

    private:
        Data _data;
    } __attribute__((packed));

    typedef Packet PDU;
}
```

4.5.7.5. UDP

The User Datagram Protocol (UDP) is implemented in EPOS following RFC 768. An UDP Communicator is defined in EPOS as Mailbox<UDP>. See Mailbox above for the general interface. Messages for this Mailbox are UDP datagrams specified in the RFC and fully abstracted by EPOS. The interface is therefore that of any Mailbox.

4.5.7.6. TCP

The Transmission Control Protocol (TCP) is implemented in EPOS following RFC 793. A TCP Communicator is defined in EPOS as Port<TCP>, which upon each connection yields a Link<TPC>. See Communicator above for the general interfaces. Packet for this Mailbox are TCP segments specified in the RFC and fully abstracted by EPOS. The interfaces are therefore those of Port and Link.

4.5.8. Configuring Networking

Enabling networking in EPOS is easy. You just have to set `Traits<Build>::NODES` to a value larger than one. The following configuration fragments configure a PC with 3 NICs, 2 x PCNet32 and 1 x E100, enables IP for all of them and defines DHCP for the first and the third, while the second is statically configured.
Configuration

```
template<> struct Traits<PC_Ethernet>: public Traits<PC_Common>
{
    typedef LIST<PCNet32, PCNet32, E100> NICS;
};

template<> struct Traits<Build>
{
    static const unsigned int NODES = 100;
};

template<> struct Traits<Network>: public Traits<void>
{
    // This list is positional, with one network for each NIC in Traits<NIC>::NICS
    typedef LIST<IP, IP, IP> NETWORKS;
};

template<> struct Traits<IP>: public Traits<Network>
{
    struct Default_Config {
        static const unsigned int  TYPE    = DHCP;
        static const unsigned long ADDRESS = 0;
        static const unsigned long NETMASK = 0;
        static const unsigned long GATEWAY = 0;
    };

    template<unsigned int UNIT>
    struct Config: public Default_Config {};
};

template<> struct Traits<IP>::Config<0> //: public Traits<IP>::Default_Config
{
    static const unsigned int  TYPE      = MAC;
    static const unsigned long ADDRESS   = 0x0a000100;  // 10.0.1.x x=MAC[5]
    static const unsigned long NETMASK   = 0xffffff00;  // 255.255.255.0
    static const unsigned long GATEWAY   = 0;           // 10.0.1.1
};
```

4.6. Utilities

4.6.1. Queue

The EPOS has 4 types of queues. They are:

1. *Queue*;
2. *Ordered_Queue*;
3. *Relative_Queue*;
4. *Scheduling_Queue*.

These queues inherit from one of two classes. They are:

1. *Queue_Wrapper (Non-Atomic)*;
2. *Queue_Wrapper (Atomic)*.

4.6.1.1. Queue

Queue is a traditional queue, with insertions at the tail and removals either from the head or from specific objects.

4.6.1.2. Ordered_Queue

Ordered_Queue is an ordered queue, i.e. objects are inserted in-order based on the integral value of "element.rank". Note that "rank" implies an order, but does not necessarily need to be "the absolute order" in the queue; it could, for instance, be a priority information or a time-out specification. Insertions must first tag "element" with "rank". Removals are just like in the traditional queue. Elements of both Queues may be exchanged. The figure below shows an example of ordered queue.

4.6.1.3. Relative_Queue

Relative_Queue is an ordered queue, i.e. objects are inserted in-order based on the integral value of "element.rank" just like above. But differently from that, a Relative Queue handles "rank" as relative offsets. This is very useful for alarm queues. Elements of Relative Queue cannot be exchanged with elements of the other queues described earlier. The figure below shows an example of relative queue.

4.6.1.4. Scheduling_Queue

Scheduling_Queue is an ordered queue whose ordering criterion is externally definable and for which selecting methods are defined (e.g. choose). This utility is most useful for schedulers, such as CPU or I/O. There are two scheduling queues in EPOS, but one of them inherits from Scheduling_List.

Methods

- `Scheduling_Queue()`
  Creates a scheduling queue.

- `unsigned int size()`
  Returns the number of elements of the scheduling queue.

- `Element * volatile & chosen()`

- `void insert(Element * e)`
  Adds the element "e" in the scheduling queue.

- `Element * remove(Element * e)`
  Removes and returns the element "e" in the scheduling queue.
- `Element * choose()`
- `Element * choose_another()`
- `Element * choose(Element * e)`

Methods inherited from Scheduling_List

- `Element * remove(Element * e)`
  Removes and returns the element "e" in the scheduling queue.
- `Element * choose()`
- `Element * choose(Element * e)`

4.6.1.5. Queue_Wrapper

Queue_Wrapper's are the base classes of queues. The difference between atomic and non-atomic is that in the first, all methods and functions initialize calling lock() and end with a call to unlock().

Methods

- `void lock()`
  Acquires the spinlock.
- `void unlock()`
  Releases the spinlock.
- `bool empty()`
  Returns true if the queue is empty, otherwise, false.
- `unsigned int size()`
  Returns the number of elements of the queue.
- `Element * head()`
  Returns the first element of the queue.
- `Element * tail()`
  Returns the last element of the queue.
- `void insert(Element * e)`
  Adds the element "e" at the end of the queue.
- `Element * remove()`
  Removes and returns the first element of queue. If the queue is empty, returns 0.
- `Element * remove(Element * e)`
  Removes and returns the element "e" in the queue.
- `Element * remove(const Object_Type * obj)`
  Removes the element with the content "obj" and returns this element. It returns 0 if the content "obj" is not in the queue.
- `Element * search(const Object_Type * obj)`
  Returns the element with the content "obj". It returns 0 if the content "obj" is not in the queue.

- `Element * volatile & chosen()`
- `Element * choose()`
- `Element * choose_another()`
- `Element * choose(Element * e)`
- `Element * choose(const Object_Type * obj)`

4.6.1.6. Example

$EPOS/src/utility/queue_test.cc

```
// EPOS Queue Utility Test Program

#include <utility/ostream.h>
#include <utility/queue.h>

using namespace EPOS;

struct Integer1 {
    Integer1(int _i) : i(_i), e(this) {}

    int i;
    Queue<Integer1>::Element e;
};

struct Integer2 {
    Integer2(int _i, int _r) : i(_i), e(this, _r) {}

    int i;
    Ordered_Queue<Integer2>::Element e;
};

struct Integer3 {
    Integer3(int _i, int _r) : i(_i), e(this, _r) {}

    int i;
    Relative_Queue<Integer3>::Element e;
};

int main()
{
    OStream cout;

    cout << "Queue Utility Test" << endl;

    cout << "\nThis is an integer queue:" << endl;
    Integer1 i1(1), i2(2), i3(3), i4(4);
    Queue<Integer1> q1;
    cout << "Inserting the integer " << i1.i << "" << endl;
    q1.insert(&i1.e);
    cout << "Inserting the integer " << i2.i << "" << endl;
    q1.insert(&i2.e);
    cout << "Inserting the integer " << i3.i << "" << endl;
    q1.insert(&i3.e);
    cout << "Inserting the integer " << i4.i << "" << endl;
    q1.insert(&i4.e);
    cout << "The queue has now " << q1.size() << " elements." << endl;
    cout << "Removing the element whose value is " << i2.i << " => "
         << q1.remove(&i2)->object()->i << "" << endl;
    cout << "Removing the queue's head => " << q1.remove()->object()->i
         << "" << endl;
    cout << "Removing the element whose value is " << i4.i << " => "
         << q1.remove(&i4)->object()->i << "" << endl;
    cout << "Removing the queue's head =>" << q1.remove()->object()->i
```

```
        << "" << endl;
    cout << "The queue has now " << q1.size() << " elements." << endl;


    cout << "\nThis is an ordered integer queue:" << endl;
    Integer2 j1(1, 2), j2(2, 3), j3(3, 4), j4(4, 1);
    Ordered_Queue<Integer2> q2;
    cout << "Inserting the integer " << j1.i
         << " with rank " << j1.e.rank() << "." << endl;
    q2.insert(&j1.e);
    cout << "Inserting the integer " << j2.i
         << " with rank " << j2.e.rank() << "." << endl;
    q2.insert(&j2.e);
    cout << "Inserting the integer " << j3.i
         << " with rank " << j3.e.rank() << "." << endl;
    q2.insert(&j3.e);
    cout << "Inserting the integer " << j4.i
         << " with rank " << j4.e.rank() << "." << endl;
    q2.insert(&j4.e);
    cout << "The queue has now " << q2.size() << " elements." << endl;
    cout << "Removing the element whose value is " << j2.i << " => "
         << q2.remove(&j2)->object()->i << "" << endl;
    cout << "Removing the queue's head => " << q2.remove()->object()->i
         << "" << endl;
    cout << "Removing the queue's head => " << q2.remove()->object()->i
         << "" << endl;
    cout << "Removing the queue's head => " << q2.remove()->object()->i
         << "" << endl;
    cout << "The queue has now " << q2.size() << " elements." << endl;


    cout << "\nThis is an integer queue with relative ordering:" << endl;
    Integer3 k1(1, 2), k2(2, 3), k3(3, 4), k4(4, 1);
    Relative_Queue<Integer3> q3;
    cout << "Inserting the integer " << k1.i
         << " with relative order " << k1.e.rank() << "." << endl;
    q3.insert(&k1.e);
    cout << "Inserting the integer " << k2.i
         << " with relative order " << k2.e.rank() << "." << endl;
    q3.insert(&k2.e);
    cout << "Inserting the integer " << k3.i
         << " with relative order " << k3.e.rank() << "." << endl;
    q3.insert(&k3.e);
    cout << "Inserting the integer " << k4.i
         << " with relative order " << k4.e.rank() << "." << endl;
    q3.insert(&k4.e);
    cout << "The queue has now " << q3.size() << " elements." << endl;
    cout << "Removing the element whose value is " << j2.i << " => "
         << q3.remove(&k2)->object()->i << "" << endl;
    cout << "Removing the queue's head => " << q3.remove()->object()->i
         << "" << endl;
    cout << "Removing the queue's head => " << q3.remove()->object()->i
         << "" << endl;
    cout << "Removing the queue's head => " << q3.remove()->object()->i
```

```
        << "" << endl;
    cout << "The queue has now " << q3.size() << " elements." << endl;

    return 0;
}
```

4.6.2. List

The EPOS has 9 types of list. They are:

- *Simple_List*;
- *Simple_Ordered_List*;
- *Simple_Relative_List*;
- *Simple_Grouping_List*;
- *List*;
- *Ordered_List*;
- *Relative_List*;
- *Scheduling_List*;
- *Grouping_List*.

4.6.2.1. Simple_List

Singly-Linked List.

Methods

- `Simple_List()`
  Creates and initializes a simple list.
- `bool empty() cons`
  Returns true if the simple list is empty, otherwise, false.
- `unsigned int size() const`
  Returns the number of elements of the simple list.
- `Element * head()`
  Returns the first element of the simple list.
- `Element * tail()`
  Returns the last element of the simple list.
- `Iterator begin()`
  Returns an iterator to the first element of simple list.
- `Iterator end()`
  Returns an iterator to the last element of simple list.
- `void insert(Element * e)`
  Adds the element "e" at the end of the simple list.
- `void insert_head(Element * e)`
  Adds the element "e" at the beggining of the simple list.

- `void insert_tail(Element * e)`
  Adds the element "e" at the end of the simple list.
- `Element * remove()`
  Removes and returns the first element of simple list. If the simple list is empty, returns 0.
- `Element * remove(Element * e)`
  Removes and returns the element "e" in the simple list.
- `Element * remove_head()`
  Removes and returns the first element of simple list. If the simple list is empty, returns 0.
- `Element * remove_tail()`
  Removes and returns the last element of simple list. If the simple list is empty, returns 0.
- `Element * remove(const Object_Type * obj)`
  Removes the element with the content "obj" and returns this element. It returns 0 if the content "obj" is not in the simple list.
- `Element * search(const Object_Type * obj)`
  Returns the element with the content "obj". It returns 0 if the content "obj" is not in the simple list.

4.6.2.2. Simple_Ordered_List

Singly-Linked, Ordered List.

Methods

- `void insert(Element * e)`
  Adds the element "e" in the correct position according to the rank of the element.
- `Element * remove()`
  Removes and returns the first element of simple ordered list. If the simple ordered list is empty, returns 0.
- `Element * remove(Element * e)`
  Removes and returns the element "e" in the simple ordered list. If the simple ordered list is empty, returns 0.
- `Element * remove(const Object_Type * obj)`
  Removes the element with the content "obj" and returns this element. It returns 0 if the content "obj" is not in the simple ordered list.
- `Element * search_rank(int rank)`
  Returns the element with rank equal to the value of the parameter "rank". It returns 0 if the "rank" value is not in the simple ordered list.
- `Element * remove_rank(int rank)`
  Removes and returns the element with rank equal to the value of the parameter "rank". It returns 0 if the "rank" value is not in the simple ordered list.

4.6.2.3. Simple_Relative_List

Singly-Linked, Relative Ordered List.

4.6.2.4. Simple_Grouping_List

Singly-Linked, Grouping List.

Methods

- `Simple_Grouping_List()`
- `unsigned int grouped_size()`
- `Element * search_size(unsigned int s)`
- `Element * search_left(const Object_Type * obj)`
- `void insert_merging(Element * e, Element ** m1, Element ** m2 )`
- `Element * search_decrementing(unsigned int s)`

4.6.2.5. List

Doubly-Linked List.

Methods

- `List()`
  Creates and initializes a list.
- `bool empty() const`
  Returns true if the list is empty, otherwise, false.
- `unsigned int size() const`
  Returns the number of elements of the list.
- `Element * head()`
  Returns the first element of the list.
- `Element * tail()`
  Returns the last element of the list.
- `Iterator begin()`
  Returns an iterator to the first element of list.
- `Iterator end()`
  Returns an iterator to the last element of list.
- `void insert(Element * e)`
  Adds the element "e" at the end of the list.
- `void insert_head(Element * e)`
  Adds the element "e" at the beggining of the list.
- `void insert_tail(Element * e)`
  Adds the element "e" at the end of the list.
- `Element * remove()`
  Removes and returns the first element of list. If the list is empty, returns 0.
- `Element * remove(Element * e)`
  Removes and returns the element "e" in the list.
- `Element * remove_head()`
  Removes and returns the first element of list. If the list is empty, returns 0.

- `Element * remove_tail()`
  Removes and returns the last element of list. If the list is empty, returns 0.
- `Element * remove(const Object_Type * obj)`
  Removes the element with the content "obj" and returns this element. It returns 0 if the content "obj" is not in the list.
- `Element * search(const Object_Type * obj)`
  Returns the element with the content "obj". It returns 0 if the content "obj" is not in the list.

4.6.2.6. Ordered_List

Doubly-Linked, Ordered List.

Methods

- `void insert(Element * e)`
  Adds the element "e" in the correct position according to the rank of the element.
- `Element * remove()`
  Removes and returns the first element of ordered list. If the ordered list is empty, returns 0.
- `Element * remove(Element * e)`
  Removes and returns the element "e" in the ordered list. If the ordered list is empty, returns 0.
- `Element * remove(const Object_Type * obj)`
  Removes the element with the content "obj" and returns this element. It returns 0 if the content "obj" is not in the ordered list.
- `Element * search_rank(int rank)`
  Returns the element with rank equal to the value of the parameter "rank". It returns 0 if the "rank" value is not in the ordered list.
- `Element * remove_rank(int rank)`
  Removes and returns the element with rank equal to the value of the parameter "rank". It returns 0 if the "rank" value is not in the ordered list.

4.6.2.7. Relative_List

Doubly-Linked, Relative Ordered List.
Generalization of `Ordered List`

4.6.2.8. Scheduling_List

Doubly-Linked, Scheduling List.

Methods

- `Scheduling_List()`
- `bool empty() const`
- `unsigned int size() const`
- `Element * head()`
- `Element * tail()`
- `Iterator begin()`
- `Iterator end()`

- `Element * volatile & chosen()`
- `void insert(Element * e)`
- `Element * remove(Element * e)`
- `Element * remove(Object_Type * obj)`
- `Element * choose()`
- `Element * choose_another()`
- `Element * choose(Element * e)`
- `Element * choose(const Object_Type * obj)`

4.6.2.9. Grouping_List

Doubly-Linked, Grouping List.
Generalization of `List`

Methods

- `Grouping_List`
- `unsigned int grouped_size()`
- `Element * search_size(unsigned int s)`
- `Element * search_left(const Object_type * obj)`
- `void insert_merging(Element * e, Element ** m1, Element ** m2)`
- `Element * search_decrementing(unsigned int s)`

4.6.2.10. Example

src/utility/list_test.cc

```
// EPOS List Utility Test Program

#include <utility/ostream.h>
#include <utility/malloc.h>
#include <utility/list.h>

using namespace EPOS;

const int N = 10;

void test_simple_list();
void test_list();
void test_ordered_list();
void test_relative_list();
void test_grouping_list();
void test_simple_grouping_list();
void test_scheduling_list();

OStream cout;

int main()
{
    cout << "List Utility Test" << endl;

    test_simple_list();
    test_simple_grouping_list();
    test_list();
    test_ordered_list();
    test_relative_list();
    test_grouping_list();
    test_scheduling_list();

    cout << "\nDone!" << endl;

    return 0;
}

void test_simple_list ()
{
    cout << "\nThis is a singly-linked list of integers:" << endl;
    Simple_List<int> l;
    int o[N];
    Simple_List<int>::Element * e[N];
    cout << "Inserting the following integers into the list ";
    for(int i = 0; i < N; i++) {
        o[i] = i;
        e[i] = new Simple_List<int>::Element(&o[i]);
        l.insert(e[i]);
        cout << i;
        if(i != N - 1)
            cout << ", ";
    }
    cout << endl;
    cout << "The list has now " << l.size() << " elements" << endl;
    cout << "They are: ";
    for(Simple_List<int>::Iterator i = l.begin(); i != l.end(); i++) {
        cout << *i->object();
        if(Simple_List<int>::Iterator(i->next()) != l.end())
            cout << ", ";
    }
    cout << endl;
    cout << "Removing the element whose value is " << o[N/2] << " => "
         << *l.remove(&o[N/2])->object() << endl;
    cout << "Removing the list's head => " << *l.remove_head()->object()
         << endl;
    cout << "Removing the element whose value is " << o[N/4] << " => "
         << *l.remove(&o[N/4])->object() << endl;
    cout << "Removing the list's tail => " << *l.remove_tail()->object()
         << endl;
    cout << "Trying to remove an element that is not on the list => "
         << l.remove(&o[N+1]) << endl;
    cout << "Removing all remaining elements => ";
    while(l.size() > 0) {
        cout << *l.remove()->object();
        if(l.size() > 0)
            cout << ", ";
    }
    cout << endl;
    cout << "The list has now " << l.size() << " elements" << endl;
    for(int i = 0; i < N; i++)
        delete e[i];
}

void test_simple_grouping_list()
{
    cout << "\nThis is a simple grouping list of integers:" << endl;
    Simple_Grouping_List<int> l;
    int o[N * 2];
    Simple_Grouping_List<int>::Element * e[N * 2];
    Simple_Grouping_List<int>::Element * d1 = 0, * d2 = 0;
    cout << "Inserting the following group of integers into the list ";
    for(int i = 0; i < N * 2; i += 4) {
        o[i] = i;
        o[i + 1] = i + 1;
        e[i] = new Simple_Grouping_List<int>::Element(&o[i], 2);
        l.insert_merging(e[i], &d1, &d2);
        cout << i << "(2), ";
        if(d1) {
            cout << "[nm]"; // next merged
            delete d1;
        }
        if(d2) {
            cout << "[tm]"; // this merged
            delete d2;
        }
    }
    for(int i = 2; i < N * 2; i += 4) {
        o[i] = i;
        o[i + 1] = i + 1;
        e[i] = new Simple_Grouping_List<int>::Element(&o[i], 2);
        l.insert_merging(e[i], &d1, &d2);
        cout << i << "(2)";
        if(d1) {
            cout << "[nm]"; // next merged
            delete d1;
        }
        if(d2) {
            cout << "[tm]"; // this merged
            delete d2;
        }
        if(i < (N - 1) * 2)
            cout << ", ";
    }
    cout << endl;
    cout << "The list has now " << l.size() << " elements that group "
         << l.grouped_size() << " elements in total" << endl;
    cout << "They are: ";
    for(Simple_Grouping_List<int>::Iterator i = l.begin(); i != l.end(); i++) {
        cout << *i->object();
        if(Simple_Grouping_List<int>::Iterator(i->next()) != l.end())
            cout << ", ";
    }
    cout << endl;
    cout << "Allocating 1 element from the list => ";
    d1 = l.search_decrementing(1);
    if(d1) {
        cout << *(d1->object() + d1->size()) << endl;
        if(!d1->size()) {
            cout << "[rm]"; // removed
            delete d1;
        }
    } else
        cout << "failed!" << endl;
    cout << "Allocating 6 more elements from the list => ";
    d1 = l.search_decrementing(6);
    if(d1) {
        cout << *(d1->object() + d1->size());
        if(!d1->size()) {
            cout << "[rm]"; // removed
            delete d1;
        }
        cout << endl;
    } else
        cout << "failed!" << endl;
    cout << "Allocating " << N * 2 << " more elements from the list => ";
    d1 = l.search_decrementing(N * 2);
    if(d1) {
        cout << *(d1->object() + d1->size());
        if(!d1->size()) {
            cout << "[rm]"; // removed
            delete d1;
        }
        cout << endl;
    } else
        cout << "failed!" << endl;
    cout << "Allocating " << (N * 2)-7 << " more elements from the list => ";
    d1 = l.search_decrementing((N * 2) - 7);
    if(d1) {
        cout << *(d1->object() + d1->size());
        if(!d1->size()) {
            cout << "[r]"; // removed
            delete d1;
        }
        cout << endl;
    } else
        cout << "failed!" << endl;
    cout << "The list has now " << l.size() << " elements that group "
         << l.grouped_size() << " elements in total" << endl;
}

void test_list ()
{
    cout << "\nThis is a doubly-linked list of integers:" << endl;
    List<int> l;
    int o[N];
    List<int>::Element * e[N];
    cout << "Inserting the following integers into the list ";
    for(int i = 0; i < N; i++) {
        o[i] = i;
        e[i] = new List<int>::Element(&o[i]);
        l.insert(e[i]);
        cout << i;
        if(i != N - 1)
            cout << ", ";
    }
    cout << endl;
    cout << "The list has now " << l.size() << " elements" << endl;
    cout << "They are: ";
    for(List<int>::Iterator i = l.begin(); i != l.end(); i++) {
        cout << *i->object();
        if(List<int>::Iterator(i->next()) != l.end())
            cout << ", ";
    }
    cout << endl;
    cout << "Removing the element whose value is " << o[N/2] << " => "
         << *l.remove(&o[N/2])->object() << endl;
    cout << "Removing the list's head => " << *l.remove_head()->object()
         << endl;
    cout << "Removing the element whose value is " << o[N/4] << " => "
         << *l.remove(&o[N/4])->object() << endl;
    cout << "Removing the list's tail => " << *l.remove_tail()->object()
         << endl;
    cout << "Trying to remove an element that is not on the list => "
         << l.remove(&o[N+1]) << endl;
    cout << "Removing all remaining elements => ";
    while(l.size() > 0) {
        cout << *l.remove()->object();
        if(l.size() > 0)
            cout << ", ";
    }
    cout << endl;
    cout << "The list has now " << l.size() << " elements" << endl;
    for(int i = 0; i < N; i++)
        delete e[i];
}

void test_ordered_list ()
{
    cout << "\nThis is an ordered, linked list of integers:" << endl;
    Ordered_List<int> l;
    int o[N];
    Ordered_List<int>::Element * e[N];
    cout << "Inserting the following integers into the list ";
    for(int i = 0; i < N; i++) {
        o[i] = i;
        e[i] = new Ordered_List<int>::Element(&o[i], N - i - 1);
        l.insert(e[i]);
        cout << i << "(" << N - i - 1 << ")";
        if(i != N - 1)
            cout << ", ";
    }
    cout << endl;
    cout << "The list has now " << l.size() << " elements" << endl;
    cout << "They are: ";
    for(Ordered_List<int>::Iterator i = l.begin(); i != l.end(); i++) {
        cout << *i->object();
        if(Ordered_List<int>::Iterator(i->next()) != l.end())
            cout << ", ";
    }
    cout << endl;
    cout << "Removing the element whose value is " << o[N/2] << " => "
         << *l.remove(&o[N/2])->object() << endl;
    cout << "Removing the list's head => " << *l.remove_head()->object()
         << endl;
    cout << "Removing the element whose value is " << o[N/4] << " => "
         << *l.remove(&o[N/4])->object() << endl;
    cout << "Removing the list's tail => " << *l.remove_tail()->object()
         << endl;
    cout << "Trying to remove an element that is not on the list => "
         << l.remove(&o[N+1]) << endl;
    cout << "Removing all remaining elements => ";
    while(l.size() > 0) {
        cout << *l.remove()->object();
        if(l.size() > 0)
            cout << ", ";
    }
    cout << endl;
    cout << "The list has now " << l.size() << " elements" << endl;
    for(int i = 0; i < N; i++)
```

```
        delete e[i];
}

void test_relative_list ()
{
    cout << "\nThis is a realtive ordered, linked list of integers:" << endl;
    Relative_List<int> l;
    int o[N];
    Relative_List<int>::Element * e[N];
    cout << "Inserting the following integers into the list ";
    for(int i = 0; i < N; i++) {
        o[i] = i;
        e[i] = new Relative_List<int>::Element(&o[i], N - i - 1);
        l.insert(e[i]);
        cout << i << "(" << N - i - 1 << ")";
        if(i != N - 1)
            cout << ", ";
    }
    cout << endl;
    cout << "The list has now " << l.size() << " elements" << endl;
    cout << "They are: ";
    for(Relative_List<int>::Iterator i = l.begin(); i != l.end(); i++) {
        cout << *i->object();
        if(Relative_List<int>::Iterator(i->next()) != l.end())
            cout << ", ";
    }
    cout << endl;
    cout << "Removing the element whose value is " << o[N/2] << " => "
         << *l.remove(&o[N/2])->object() << endl;
    cout << "Removing the list's head => " << *l.remove_head()->object()
         << endl;
    cout << "Removing the element whose value is " << o[N/4] << " => "
         << *l.remove(&o[N/4])->object() << endl;
    cout << "Removing the list's tail => " << *l.remove_tail()->object()
         << endl;
    cout << "Trying to remove an element that is not on the list => "
         << l.remove(&o[N+1]) << endl;
    cout << "Removing all remaining elements => ";
    while(l.size() > 0) {
        cout << *l.remove()->object();
        if(l.size() > 0)
            cout << ", ";
    }
    cout << endl;
    cout << "The list has now " << l.size() << " elements" << endl;
    for(int i = 0; i < N; i++)
        delete e[i];
}

void test_grouping_list()
{
    cout << "\nThis is a grouping list of 32-byte buffers:" << endl;
    Grouping_List<char> l;
    char o[N][32];
    Grouping_List<char>::Element * e[N];
    Grouping_List<char>::Element * d1 = 0, * d2 = 0;
    cout << "Inserting the following buffers into the list ";
    for(int i = 0; i < N; i += 2) {
        o[i][0] = 48 + i;
        o[i][1] = '\0';
        e[i] = new Grouping_List<char>::Element(&o[i][0], sizeof(char[32]));
        l.insert_merging(e[i], &d1, &d2);
        cout << &o[i] << " (" << sizeof(char[32]) << "), ";
        if(d1) {
            cout << "[nm]"; // next merged
            delete d1;
        }
        if(d2) {
            cout << "[tm]"; // this merged
            delete d2;
        }
    }
    for(int i = 1; i < N; i += 2) {
        o[i][0] = 48 + i;
        o[i][1] = '\0';
        e[i] = new Grouping_List<char>::Element(&o[i][0], sizeof(char[32]));
        l.insert_merging(e[i], &d1, &d2);
        cout << &o[i] << " (" << sizeof(char[32]) << "), ";
        if(d1) {
            cout << "[nm]"; // next merged
            delete d1;
        }
        if(d2) {
            cout << "[tm]"; // this merged
            delete d2;
        }
    }
    cout << endl;
    cout << "The list has now " << l.size() << " elements that group "
         << l.grouped_size() << " bytes in total" << endl;
    cout << "Allocating 1 byte from the list => ";
    d1 = l.search_decrementing(1);
    if(d1) {
        cout << (void *)(d1->object() + d1->size()) << endl;
        if(!d1->size()) {
            cout << "[rm]"; // removed
            delete d1;
        }
    } else
        cout << "failed!" << endl;
    cout << "List size=" << l.size() << endl;
    cout << "Allocating 6 more bytes from the list => ";
    d1 = l.search_decrementing(6);
    if(d1) {
        cout << (void *)(d1->object() + d1->size()) << endl;
        if(!d1->size()) {
            cout << "[rm]"; // removed
            delete d1;
        }
        cout << endl;
    } else
        cout << "failed!" << endl;
    cout << "Allocating " << N * 2 << " more bytes from the list => ";
    d1 = l.search_decrementing(N * 2);
    if(d1) {
        cout << (void *)(d1->object() + d1->size()) << endl;
        if(!d1->size()) {
            cout << "[rm]"; // removed
            delete d1;
        }
        cout << endl;
    } else
        cout << "failed!" << endl;
    cout << "Allocating " << (N * 2)-7 << " more bytes from the list => ";
    d1 = l.search_decrementing((N * 2) - 7);
    if(d1) {
        cout << (void *)(d1->object() + d1->size()) << endl;
        if(!d1->size()) {
            cout << "[r]"; // removed
            delete d1;
        }
        cout << endl;
    } else
        cout << "failed!" << endl;
    cout << "The list has now " << l.size() << " elements that group "
         << l.grouped_size() << " elements in total" << endl;
}

#include<scheduler.h>
void test_scheduling_list ()
{
    cout << "\nThis is priority scheduling list of integers:" << endl;
    Scheduling_List<int, Scheduling_Criteria::Priority> l;
    int o[N];
    Scheduling_List<int, Scheduling_Criteria::Priority>::Element * e[N];
    cout << "Inserting the following integers into the list ";
    for(int i = 0; i < N; i++) {
        o[i] = i;
        e[i] = new Scheduling_List<int, Scheduling_Criteria::Priority>::Element(&o[i],
N - i - 1);
        l.insert(e[i]);
        cout << i << "(" << N - i - 1 << ")";
        if(i != N - 1)
            cout << ", ";
    }
    cout << endl;
    cout << "The list has now " << l.size() << " elements" << endl;
    cout << "They are: ";
    for(Scheduling_List<int, Scheduling_Criteria::Priority>::Iterator i = l.begin(); i
!= l.end(); i++) {
        cout << *i->object();
        if(Scheduling_List<int, Scheduling_Criteria::Priority>::Iterator(i->next()) !=
l.end())
            cout << ", ";
    }
    cout << endl;
    cout << "Scheduling the list => " << *l.choose()->object() << endl;
    cout << "They are: ";
    for(Scheduling_List<int, Scheduling_Criteria::Priority>::Iterator i = l.begin(); i
!= l.end(); i++) {
        cout << *i->object();
        if(Scheduling_List<int, Scheduling_Criteria::Priority>::Iterator(i->next()) !=
l.end())
            cout << ", ";
    }
    cout << endl;
    cout << "Forcing scheduling of antorher element => " <<
        *l.choose_another()->object() << endl;
    cout << "They are: ";
    for(Scheduling_List<int, Scheduling_Criteria::Priority>::Iterator i = l.begin(); i
!= l.end(); i++) {
        cout << *i->object();
        if(Scheduling_List<int, Scheduling_Criteria::Priority>::Iterator(i->next()) !=
l.end())
            cout << ", ";
    }
    cout << endl;
    cout << "Forcing scheduling of element whose value is " << o[N/2] << " => "
         << *l.choose(e[N/2])->object() << endl;
    cout << "They are: ";
    for(Scheduling_List<int, Scheduling_Criteria::Priority>::Iterator i = l.begin(); i
!= l.end(); i++) {
        cout << *i->object();
        if(Scheduling_List<int, Scheduling_Criteria::Priority>::Iterator(i->next()) !=
l.end())
            cout << ", ";
    }
    cout << endl;
    cout << "Removing the list's head => " << *l.remove(l.choose())->object()
         << endl;
    cout << "Removing the element whose value is " << o[N/4] << " => "
         << *l.remove(e[N/4])->object() << endl;
    cout << "Removing all remaining elements => ";
    while(l.size() > 0) {
        cout << *l.remove(l.choose())->object();
        if(l.size() > 0)
            cout << ", ";
    }
    cout << endl;
    cout << "The list has now " << l.size() << " elements" << endl;
    for(int i = 0; i < N; i++)
        delete e[i];
}
```

4.6.3. Hash

The EPOS has 2 types of Hash. They are:

- *Simple_Hash*;
- *Hash*;

4.6.3.1. Simple_Hash

Hash Table with a single Synonym List in order to change the hash function, simply redefine the operator % for objects of type T and Key.

Methods

- `Simple_Hash()`
  Creates a simple hash table.
- `bool empty() const`
  Returns true if the simple hash table is empty, otherwise, false.
- `unsigned int size() const`
  Returns the number of elements of the simple hash table.
- `void insert(Element * e)`
  Adds the element "e" in the simple hash table.
- `Element * remove(Element * e)`
  Removes and returns the element "e" in the simple hash table.
- `Element * remove(const Object_Type * obj)`
  Removes the element with the content "obj" and returns this element. It returns 0 if the content "obj" is not in the simple hash table.
- `Element * search(const Object_Type * obj)`
  Returns the element with the content "obj". It returns 0 if the content "obj" is not in the simple hash table.
- `Element * search_key(const Key & key)`
  Returns the element with the key "key". It returns 0 if the key "key" is not in the simple hash table.
- `Element * remove_key(int key)`
  Removes the element with the key "key" and returns this element. It returns 0 if the key "key" is not in the simple hash table.

4.6.3.2. Hash

Hash Table with a Synonym List for each Key.

Methods

- `Hash()`
  Creates a hash table.
- `bool empty() const`
  Returns true if the hash table is empty, otherwise, false.
- `unsigned int size() const`
  Returns the number of elements of the hash table.

- `void insert(Element * e)`
  Adds the element "e" in the hash table.
- `Element * remove(Element * e)`
  Removes and returns the element "e" in the hash table.
- `Element * remove(const Object_Type * obj)`
  Removes the element with the content "obj" and returns this element. It returns 0 if the content "obj" is not in the hash table.
- `Element * search(const Object_Type * obj)`
  Returns the element with the content "obj". It returns 0 if the content "obj" is not in the hash table.
- `Element * search_key(const Key & key)`
  Returns the element with the key "key". It returns 0 if the key "key" is not in the hash table.
- `Element * remove_key(int key)`
  Removes the element with the key "key" and returns this element. It returns 0 if the key "key" is not in the hash table.

4.6.3.3. Example

src/utility/hash_test.cc

```
// EPOS Hash Utility Test Program

#include <utility/ostream.h>
#include <utility/malloc.h>
#include <utility/hash.h>

using namespace EPOS;

const int N = 10;

void test_few_synonyms_hash();
void test_many_synonyms_hash();

OStream cout;

int main()
{
    cout << "Hash Utility Test" << endl;

    test_few_synonyms_hash();
    test_many_synonyms_hash();

    cout << "\nDone!" << endl;

    return 0;
}

void test_few_synonyms_hash()
{
    cout << "\nThis is a hash table of integeres with few synonyms:" << endl;

    Simple_Hash<int, N> h;
    int o[N * 2];
    Simple_Hash<int, N>::Element * e[N * 2];

    cout << "Inserting the following integers into the hash table ";
    for(int i = 0; i < N * 2; i++) {
        o[i] = i;
        e[i] = new Simple_Hash<int, N>::Element(&o[i], i);
        h.insert(e[i]);
        cout << i;
        if(i != N * 2 - 1)
            cout << ", ";
    }
    cout << "" << endl;

    cout << "The hash table has now " << h.size() << " elements:" << endl;
    for(int i = 0; i < N * 2; i++) {
        cout << "[" << i << "]={o=" << *h.search_key(i)->object()
             << ",k=" << h.search_key(i)->key() << "}";
        if(i != N * 2 - 1)
            cout << ", ";
    }
    cout << "" << endl;

    cout << "Removing the element whose value is " << o[N/2] << " => "
         << *h.remove(&o[N/2])->object() << "" << endl;
    cout << "Removing the element whose key is " << 1 << " => "
         << *h.remove_key(1)->object() << "" << endl;
    cout << "Removing the element whose key is " << 11 << " => "
         << *h.remove_key(11)->object() << "" << endl;
    cout << "Removing the element whose value is " << o[N/4] << " => "
         << *h.remove(&o[N/4])->object() << "" << endl;
    cout << "Removing the element whose key is " << N-1 << " => "
         << *h.remove_key(N-1)->object() << "" << endl;
    cout << "Trying to remove an element that is not on the hash => "
         << h.remove(&o[N/2]) << "" << endl;

    cout << "The hash table has now " << h.size() << " elements:" << endl;
    for(int i = 0; i < N * 2; i++) {
        cout << "[" << i << "]={o=" << *h.search_key(i)->object()
             << ",k=" << h.search_key(i)->key() << "}";
        if(i != N * 2 - 1)
            cout << ", ";
    }
    cout << "" << endl;

    cout << "Removing all remaining elements => ";
    for(int i = 0; i < N * 2; i++) {
        cout << *h.remove_key(i)->object();
        if(i != N * 2 - 1)
            cout << ", ";
    }
    cout << "" << endl;

    for(int i = 0; i < N * 2; i++)
        delete e[i];
}

void test_many_synonyms_hash()
{
    cout << "\nThis is a hash table of integeres with many synonyms:" << endl;

    Hash<int, N> h;
    int o[N * N];
    Hash<int, N>::Element * e[N * N];

    cout << "Inserting the following integers into the hash table ";
    for(int i = 0; i < N * N; i++) {
        o[i] = i;
        e[i] = new Hash<int, N>::Element(&o[i], i);
        h.insert(e[i]);
        cout << i;
        if(i != N * N - 1)
            cout << ", ";
    }
    cout << "The hash table has now " << h.size() << " elements:" << endl;
    for(int i = 0; i < N * N; i++) {
        cout << "[" << i << "]={o=" << *h.search_key(i)->object()
             << ",k=" << h.search_key(i)->key() << "}";
        if(i != N * N - 1)
            cout << ", ";
    }
    cout << "" << endl;

    cout << "Removing the element whose value is " << o[N/2] << " => "
         << *h.remove(&o[N/2])->object() << "" << endl;
    cout << "Removing the element whose key is " << 1 << " => "
         << *h.remove_key(1)->object() << "" << endl;
    cout << "Removing the element whose key is " << 11 << " => "
         << *h.remove_key(11)->object() << "" << endl;
    cout << "Removing the element whose value is " << o[N/4] << " => "
         << *h.remove(&o[N/4])->object() << "" << endl;
    cout << "Removing the element whose key is " << N-1 << " => "
         << *h.remove_key(N-1)->object() << "" << endl;
    cout << "Trying to remove an element that is not on the hash => "
         << h.remove(&o[N/2]) << "" << endl;

    cout << "The hash table has now " << h.size() << " elements:" << endl;
    for(int i = 0; i < N * N; i++) {
        cout << "[" << i << "]={o=" << *h.search_key(i)->object()
             << ",k=" << h.search_key(i)->key() << "}";
        if(i != N * N - 1)
            cout << ", ";
    }
    cout << "" << endl;

    cout << "Removing all remaining elements => ";
    for(int i = 0; i < N * N; i++) {
        cout << *h.remove_key(i)->object();
        if(i != N * N - 1)
            cout << ", ";
    }
    cout << "" << endl;

    for(int i = 0; i < N * N; i++)
        delete e[i];
}
```

4.6.4. Vector

The Vector data structure API is presented below:

Methods

- `Vector()`
  Creates and initializes a vector.
- `bool empty() const`
  Returns true if the vector is empty, otherwise, false.

- `unsigned int size()`
  Returns the number of elements of the vector.
- `Element * get(int i)`
  Returns the element at position "i".
- `bool insert(Element * e, unsigned int i)`
  Adds the element "e" in position "i". If added correctly, returns true, otherwise false.
- `Element * remove(unsigned int i)`
  Removes the element at position "i" and returns this element. It returns 0 if the position "i" is invalid.
- `Element * remove(Element * e)`
  Removes the element "e" and returns this element. It returns 0 if the element "e" is not in the vector.
- `Element * remove(const Object_Type * obj)`
  Removes the element with the content "obj" and returns this element. It returns 0 if the content "obj" is not in the vector.
- `Element * search(const Object_Type * obj)`
  Returns the element with the content "obj". It returns 0 if the content "obj" is not in the vector.

4.6.4.1. Example

src/utility/vector_test.cc

```
// EPOS Vector Utility Test Program

#include <utility/ostream.h>
#include <utility/malloc.h>
#include <utility/vector.h>

using namespace EPOS;

const int N = 10;

OStream cout;

int main()
{

    cout << "Vector Utility Test" << endl;

    cout << "\nThis is a vector of integers:" << endl;
    Vector<int, N> v;
    int o[N];
    Vector<int, N>::Element * e[N];
    cout << "Inserting the following integers into the vector ";
    for(int i = 0; i < N; i++) {
        o[i] = i;
        e[i] = new Vector<int, N>::Element(&o[i]);
        v.insert(e[i], i);
        cout << "[" << i << "]=" << i;
        if(i != N - 1)
            cout << ", ";
    }
    cout << "" << endl;

    cout << "The vector has now " << v.size() << " elements:" << endl;
    for(int i = 0; i < N; i++) {
        cout << "[" << i << "]=" << *v[i]->object();
        if(i != N - 1)
            cout << ", ";
    }
    cout << "" << endl;

    for(int i = 0; i < N; i++)
        (*v[i]->object())++;
    cout << "The vector's elements were incremented and are now:" << endl;
    for(int i = 0; i < N; i++) {
        cout << "[" << i << "]=" << *v[i]->object();
        if(i != N - 1)
            cout << ", ";
    }
    cout << "" << endl;

    cout << "Removing the element whose value is " << o[N/2] << " => "
         << *v.remove(&o[N/2])->object() << "" << endl;
    cout << "Removing the second element => " << *v.remove(1)->object()
```

```
         << "" << endl;
    cout << "Removing the element whose value is " << o[N/4] << " => "
         << *v.remove(&o[N/4])->object() << "" << endl;
    cout << "Removing the last element => " << *v.remove(N - 1)->object()
         << "" << endl;
    cout << "Trying to remove an element that is not on the vector => "
         << v.remove(&o[N/2]) << "" << endl;
    cout << "Removing all remaining elements => ";
    for(int i = 0; i < N; i++) {
        cout << *v.remove(i)->object();
        if(i != N - 1)
            cout << ", ";
    }
    cout << "" << endl;
    cout << "The vector has now " << v.size() << " elements" << endl;
    for(int i = 0; i < N; i++)
        delete e[i];

    cout << "\nDone!" << endl;

    return 0;
}
```

4.6.5. Handler

EPOS allows application processes to handle events at user-level through the **Handler** family of abstractions depicted in Figure below.

4.6.5.1. Handler

Handler is the base class of **Thread_Handler**, **Function_Handler** and **Semaphore_Handler**.

Methods

- `Handler()`
  Creates a handler.
- `virtual ~Handler()`
  Destroys the handler.
- `virtual void operator()() = 0`
  This method overrides the () operator. It is a pure virtual method, then, it is required to be implemented by a derived class that is not abstract.
- `void operator delete(void * object)`
  This method overrides the delete operator.

4.6.5.2. Thread_Handler

The **Thread_Handler** member assigns a thread to handle an interrupt. Such a thread must have been previously created by the application in the suspended state.

Methods

- `Thread_Handler(Thread * h)`
  Creates a handler for the thread "h".
- `~Thread_Handler()`
  Destroys the thread handler.
- `void operator()()`
  Overrides the () operator. This operator calls the method "resume" of the thread handled by the handler.

4.6.5.3. Function_Handler

The **Function_Handler** member assigns an ordinary function supplied by the application to handle an event. It is then resumed at every occurrence of the corresponding event.

Methods

- `Function_Handler(Function * h)`
  Creates a handler for the function "h".
- `~Function_Handler()`
  Destroys the function handler.
- `void operator()()`
  Overrides the () operator. This operator calls the function handled by the handler.

4.6.5.4. Semaphore_Handler

The **Semaphore_Handler** assigns a semaphore, previously created by the application and initialized with zero, to an event. The OS invokes operation **v** on this semaphore at every event occurrence, while the handling thread invokes operation **p** to wait for an event.

Methods

- `Semaphore_Handler(Semaphore * h)`
  Creates a handler for the semaphore "h".
- `~Semaphore_Handler()`
  Destroys the semaphore handler.
- `void operator()()`
  Overrides the () operator. This operator calls the method "v" of the semaphore handled by the handler.

4.6.5.5. Example

src/utility/handler_test.cc

```
// EPOS Event Handler Utility Test Program

#include <utility/ostream.h>
#include <utility/handler.h>
#include <alarm.h>
#include <thread.h>
#include <semaphore.h>
#include <chronometer.h>

using namespace EPOS;

const int iterations = 100;
const int period_a = 100;
const int period_b = 200;
const int period_c = 300;

void func_a(void);
int func_b(void);
int func_c(void);
int max(int a, int b, int c) { return ((a >= b) && (a >= c)) ? a : ((b >= a) && (b >=
c) ? b : c); }

OStream cout;

Semaphore sem_c(0);
Thread * thread_b;
Thread * thread_c;

int main()
{
    cout << "Event Handler Utility Test" << endl;

    cout << "\nThis test consists in creating three event handlers as follows:" << end
l;
    cout << "  Handler 1 triggers a \"function\" that prints an \"a\" every " << period
_a << " ms;" << endl;
    cout << "  Handler 2 triggers a \"thread\" that prints a \"b\" every " << period_b
<< " ms;" << endl;
    cout << "  Handler 3 triggers a \"v\" on a \"semaphore\" that controls another thre
ad that prints a \"c\" every " << period_c << "ms." << endl;

    thread_b = new Thread(&func_b);
    thread_c = new Thread(&func_c);

    cout << "Threads B and C have been created!" << endl;

    Function_Handler handler_a(&func_a);
    Thread_Handler handler_b(thread_b);
    Semaphore_Handler handler_c(&sem_c);

    cout << "Now the alarms will be created, along with a chronometer to keep track of
the total execution time. I'll then wait for the threads to finish...\n" << endl;
```

```
    Chronometer chrono;
    chrono.start();

    Alarm alarm_a(period_a * 1000, &handler_a, iterations);
    Alarm alarm_b(period_b * 1000, &handler_b, iterations);
    Alarm alarm_c(period_c * 1000, &handler_c, iterations);

    int status_b = thread_b->join();
    int status_c = thread_c->join();

    chrono.stop();

    cout << "\n\nThread B exited with status " << status_b
         << " and thread C exited with status " << status_c << endl;

    cout << "\nThe estimated time to run the test was " << max(period_a, period_b, peri
od_c) * iterations << " ms. The measured time was " << chrono.read() / 1000 <<" ms!" <<
endl;

    delete thread_b;
    delete thread_c;

    cout << "I'm also done, bye!" << endl;

    return 0;
}

void func_a()
{
    cout << "a";
    return;
}

int func_b(void)
{
    cout << "B";
    for(int i = 0; i < iterations; i++) {
        thread_b->suspend();
        cout << "b";
    }
    cout << "B";
    return 'B';
}

int func_c(void)
{
    cout << "C";
    for(int i = 0; i < iterations; i++) {
        sem_c.p();
        cout << "c";
    }
    cout << "C";
    return 'C';
}
```

4.6.6. Observer

Through its interface, the *Observed* object can *attach* a new observer, *detach* an existing observer, and *notify* all the observers by calling their *update* method. There is also a *Conditionally_Observed* class which behaves just like *Observed*, but its *notify* method only notifies the observers that fulfill a given condition. All their methods, including the destructors, are virtual methods, so they can be overridden. The observers abstraction API is exemplified below.

Observed Methods

- `Observed()`

Constructs an Observed object.

- `~Observed()`
  Destructs an Observed object.
- `void attach(Observer * o)`
  Attaches a new Observer object to this Observed object.
- `void detach(Observer * o)`
  Detaches an existing Observer object from this Observed objest.
- `void notify()`
  Notifies all the attached Observers, calling their update method.

Observer Methods

- `Observer()`
  Constructs an Observer object.
- `~Observer()`
  Destructs an Observer object.
- `void update(Observed * o)`
  Updates the Observer state regarding a given Observed object.

Conditionally_Observed Methods

- `Conditionally_Observed()`
  Constructs a Conditionally Observed object.
- `~Conditionally_Observed()`
  Destructs a Conditionally Observed object.
- `void attach(Conditional_Observer * o, int c)`
  Attaches a new Conditional Observer object to this Conditionally Observed object.
- `void detach(Conditional_Observer * o, int c)`
  Detaches an existing Conditional Observer object from this Conditionally Observed objest.
- `void notify(int c)`
  Notifies all the attached Conditional Observers that satisfy the condition.

Conditional_Observer Methods

- `Conditional_Observer()`
  Constructs a Conditional Observer object.
- `~Conditional_Observer()`
  Destructs a Conditional Observer object.
- `void update(Conditionally_Observed * o)`
  Updates the Conditional Observer state regarding a given Conditionally Observed object.

4.6.6.1. Example

Please note that the *update* method is a pure virtual function and its abstract class doesn't implement it. That's why it's implemented here in the application code. The constructors and destructors were overridden for debugging reasons.

```
#include <utility/observer.h>
#include <utility/ostream.h>
#include <utility/debug.h>

__USING_SYS

OStream cout;

class Test_Observed;

class Test_Observer : public Conditional_Observer {
    public:
            Test_Observer(){
                db<Test_Observer>(TRC) << "Test_Observer:: " << this << " is saying
hi\n";
            };
            ~Test_Observer() {
                db<Test_Observer>(TRC) << "Test_Observer:: " << this << " is waving
goodbye\n";
            }
            void update(Conditionally_Observed * o){
                cout << "Notify received.\t";
                db<Test_Observer>(TRC) << "Test_Observer::update(o=" << o << ")\n
\n";
            }
};

class Test_Observed : public Conditionally_Observed {
    public:
            Test_Observed(){
                db<Test_Observed>(TRC) << "Test_Observed:: " << this << " is saying
hi\n";
            };
            ~Test_Observed() {
                db<Test_Observed>(TRC) << "Test_Observed:: " << this << " is waving
goodbye\n";
            }
};

int main() {
        cout << "\nConstructing objects.\n";
        Test_Observed * root = new Test_Observed();
        Test_Observer * observer1 = new Test_Observer();
        Test_Observer * observer2 = new Test_Observer();
        Test_Observer * observer3 = new Test_Observer();
        Test_Observer * observer4 = new Test_Observer();

        cout << "\nAttaching observers.\n";
        root->attach(observer1,1);
        root->attach(observer2,1);
        root->attach(observer3,3);
        root->attach(observer4,4);
        cout << "\nNotifying just the first two observers.\n";
        root->notify(1);
        cout << "\nNow trying to detach one of them.\n";
        root->detach(observer2,1);
        cout << "\nTrying to notify them again.\n"
                 << "Only one of them will update itself.\n";
        root->notify(1);
        cout << "\nNotifying the next-to-last observer.\n";
        root->notify(3);
        cout << "\nNotifying the last observer.\n";
        root->notify(4);

        cout << "Detaching and destructing objects.\n";
        root->detach(observer1,1);
        root->detach(observer3,3);
        root->detach(observer4,4);
        delete observer1;
        delete observer2;
        delete observer3;
        delete observer4;
        delete root;
    return 0;
}
```

4.6.7. CRC

The *CRC* class defines the Cyclic Redundancy Check (CRC) function used by EPOS. It consists of just one static method called *crc16*, responsible for calculating the CRC code.

In a real data transmission application, the data transmitted might be affected by noise in the communication channels. To detect an accidental change to the data, we firstly calculate a CRC code that is unique for the block of data we are going to send. We then send both the data and the code to the receiver. The receiver recalculates the CRC code. If the new CRC code does not match the one calculated earlier, then our block of data was undesiraly changed.

This class' simple abstraction API is described below.

Methods

- `unsigned short crc16(char * ptr, int size)`
  Calculates a short, fixed-length CRC code for a given block of data.

4.6.7.1. Example

Here is a simple example of how the *CRC* class could be used. The application calculates the CRC code and then make some changes to the data, so we can see the CRC code being used to detect accidental changes to data.

```
#include <utility/crc.h>
#include <utility/ostream.h>

__USING_SYS

OStream cout;

struct message
{
        char * block;
        int size;
        int crc;
};

typedef struct message message_t;

int main() {
        cout << "\nCreating a block of data. It's just a string.\n";
        message_t m;
        m.block = "My block of data.";
        m.size = strlen(m.block);
        cout << "Let's print our block of data: \"" << m.block << "\".\n";
        cout << "Let's calculate our CRC code.\n";
        m.crc = CRC::crc16(m.block,m.size);
        cout << "We now have our CRC code. It's " << m.crc << " .\n\n";

        message_t received_m;
        int i;
        for (i = 0; i < m.size; i++) received_m.block[i] = m.block[i];
        received_m.size = strlen(received_m.block);
        cout << "Now we are making an undesired change on the data,\n"
                 << "simulating a noise in the transmission channel.\n\n";
        received_m.block[0] = 'B';
        received_m.block[15] = 'e';

        cout << "The old block of data is \"" << m.block << "\".\n"
                << "And the changed block is \"" << received_m.block << "\".\n\n";

        cout << "Let's calculate the CRC code again.\n";
        received_m.crc = CRC::crc16(received_m.block,received_m.size);
        cout << "The new CRC code is " << received_m.crc << " .\n\n";

        if (m.crc != received_m.crc)
                cout << "The CRC codes don't match. The message was changed.\n\n";
        else
                cout << "The CRC codes are equal. The message has no error.\n\n";
    return 0;
}
```

4.6.8. OStream

The *OStream* class is the EPOS Output Stream implementation. Applications can print any formatted data on the standard output stream using the insertion operator **<<**. This abstration API is described below.

Methods

- `OStream()`
  Constructs an OStream object.
- `OStream & operator<<(const Endl & endl)`
  Prints a newline (**\n**) character on the output stream.
- `OStream & operator<<(const Hex & hex)`
  Defines the base for numeral streams as hexadecimal.
- `OStream & operator<<(const Dec & dec)`
  Defines the base for numeral streams as decimal.
- `OStream & operator<<(const Oct & oct)`
  Defines the base for numeral streams as octal.
- `OStream & operator<<(const Bin & bin)`
  Defines the base for numeral streams as binary.
- `OStream & operator<<(char c)`
  Prints a character on the output stream.
- `OStream & operator<<(unsigned char c)`
  Statically casts an *unsigned char* to a *char* and prints it on the output stream.
- `OStream & operator<<(int i)`
  Prints an integer on the output stream.
- `OStream & operator<<(short s)`
  Statically casts a *short* to an *int* and prints it on the output stream.
- `OStream & operator<<(long l)`
  Statically casts a *long* to an *int* and prints it on the output stream.
- `OStream & operator<<(unsigned int u)`
  Prints an *unsigned int* on the output stream.
- `OStream & operator<<(unsigned short s)`
  Statically casts an *unsigned short* to an *unsigned int* and prints it on the output stream.
- `OStream & operator<<(unsigned long l)`
  Statically casts an *unsigned long* to an *unsigned int* and prints it on the output stream.
- `OStream & operator<<(long long int u)`
  Prints an *long long int* on the output stream.
- `OStream & operator<<(unsigned long long int u)`
  Prints an *unsigned long long int* on the output stream.
- `OStream & operator<<(const void * p)`
  Prints a pointer on the output stream.

- `OStream & operator<<(const char * s)`
  Prints a string on the output stream.

4.6.8.1. Example

$EPOS/src/utility/ostream.h

```
// EPOS OStream Utility Test Program

#include <utility/ostream.h>

using namespace EPOS;

int main()
{
    OStream cout;

    cout << "OStream test" << endl;
    cout << "This is a char:\t\t\t" << 'A' << endl;
    cout << "This is a negative char:\t" << '\377' << endl;
    cout << "This is an unsigned char:\t" << 'A' << endl;
    cout << "This is an int:\t\t\t" << (1 << sizeof(int) * 8 - 2) << endl
         << "\t\t\t\t" << hex << (1 << sizeof(int) * 8 - 2) << "(hex)" << endl
         << "\t\t\t\t" << dec << (1 << sizeof(int) * 8 - 2) << "(dec)" << endl
         << "\t\t\t\t" << oct << (1 << sizeof(int) * 8 - 2) << "(oct)" << endl
         << "\t\t\t\t" << bin << (1 << sizeof(int) * 8 - 2) << "(bin) "
         << endl;
    cout << "This is a negative int:\t\t" << (1 << sizeof(int) * 8 - 1) << endl
         << "\t\t\t\t" << hex << (1 << sizeof(int) * 8 - 1) << "(hex)" << endl
         << "\t\t\t\t" << dec << (1 << sizeof(int) * 8 - 1) << "(dec)" << endl
         << "\t\t\t\t" << oct << (1 << sizeof(int) * 8 - 1) << "(oct)" << endl
         << "\t\t\t\t" << bin << (1 << sizeof(int) * 8 - 1) << "(bin) " << endl;
    cout << "This is a string:\t\t" << "string" << endl;
    cout << "This is a pointer:\t\t" << &cout << endl;

    return 0;
}
```

4.6.9. Spin Lock

The *Spin* class defines a Spin Lock utility, which is a busy waiting lock. When a thread acquires a lock, it enters a loop and keeps checking repeatedly until the lock becomes available. The Spin Lock abstration API is described below.

Methods

- `Spin()`
  Constructs an Spin object.

- `void acquire()`
  Acquires the spin lock, entering a loop.

- `void release()`
  Releases the spin lock.

4.6.9.1. Example

The Spin Lock is vastly applied on EPOS atomic operations. For example, the *Heap* class could use a Spin Lock to prevent its *free* method from being called before the *alloc* method finishes its allocation.

The code below shows the wrapper for atomic heap operations.

$EPOS/include/utility/heap.h

```
template<typename T>
class Heap_Wrapper<T, true>: public T
{
public:
    Heap_Wrapper() {}
    Heap_Wrapper(void * addr, unsigned int bytes): T(addr, bytes) {}

    bool empty() {
        enter();
        bool tmp = T::empty();
        leave();
        return tmp;
    }

    unsigned int size() {
        enter();
        unsigned int tmp = T::size();
        leave();
        return tmp;
    }

    void * alloc(unsigned int bytes) {
        enter();
        void * tmp = T::alloc(bytes);
        leave();
        return tmp;
    }

    void free(void * ptr) {
        enter();
        T::free(ptr);
        leave();
    }

    void free(void * ptr, unsigned int bytes) {
        enter();
        T::free(ptr, bytes);
        leave();
    }

private:
    void enter() {
        _lock.acquire();
        CPU::int_disable();
    }

    void leave() {
        _lock.release();
        CPU::int_enable();
    }
private:
    Spin _lock;
};
```

4.6.10. Random

The EPOS has a random number generator called *Random*. It consists of just a static *random* method. The Random abstraction API is described below.

Methods

- `Random()`
  Constructs a Random object.

4.6.10.1. Example

```
#include <utility/random.h>
#include <utility/ostream.h>
#include <thread.h>
#include <clock.h>

__USING_SYS

OStream cout;
Clock clock;

int main()
{
        unsigned long int ini = clock.now();
        cout << "Unix time right now is " << ini << ". Let it be our initial seed.\n";
        unsigned long int seed = Pseudo_Random::random(ini);
    cout << "Therefore, our first random number is also " << seed << ".\n\n";

    cout << "Now we're gonna try to generate 10 new random numbers.\n"
         << "Our n will be the Unix time above plus the amount of already\n"
         << "generated numbers. Hence, our first n will be the Unix time itself.\n";
        int i = 0;
        for (;i < 10;i++){
                seed = Pseudo_Random::random(ini+i);
                cout << seed << "\n";
        }
        int d = 6;
        cout << "\nAnd those were our numbers. Now let's keep generating new\n"
                 << "numbers this way until we have at least 10 numbers with\n"
                 << d << " digits. \n"
                 << "To accomplish that, the next n will be the Unix time plus the\n"
                 << "amount of trials. So our first n will be the Unix time again.\n
\n";
        int j = 0;
        i = 0;
        unsigned long int k = 0;
        while (i < 10){
                seed = Pseudo_Random::random(ini+k);
                if ((seed > 99999)&&(seed < 1000000)){
                        cout << seed << "\n";
                        i++;
                }
                j++;
                k++;
                if (k == 4294967295){
                        cout << "Stop! Patience Overflow! Sorry, something went wrong.
Bye!\n";
                        Thread::self()->exit();
                }
        }
        cout << "\nDone. We had to generate " << j << " numbers until\n"
                 << "at least 10 of them were numbers with " << d << " digits.\n\n";
    return 0;
}
```

4.7. Hardware Mediators

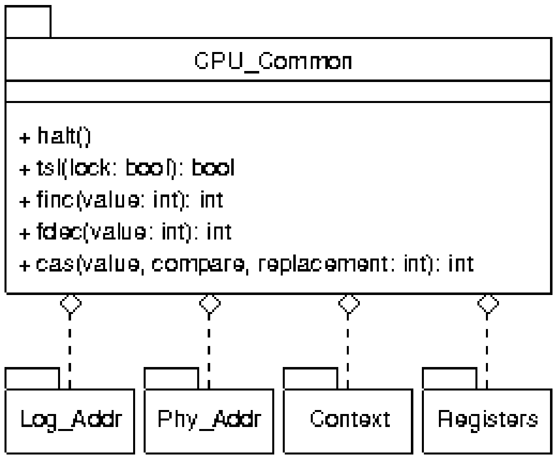
4.7.1. CPU

The CPU mediator is responsible for abstracting types and behaviour of CPU components.

Generic implementations of CPU interface are provided by CPU_Common. Architecture-specifc implementations are provided by each architecure's CPU mediator (e.g., IA32_CPU, AVR8_CPU, etc).

The CPU mediator also defines two important types (Log_Addr and Phy_Addr) to abstract, respectively, logical and physical addresses. Such types, being classes, also implements a set of constructors and operators to enable proper handling of such abstractions.

Bellow is a class diagram for this interface.



Methods

- `static void halt()`
  This function is reimplemented in the CPU mediators of archtectures providing better ways to halt a CPU. A basic implementation in CPU_Common halts the processor by entering a perpetual loop (*for(;;);*).

Note: this default implementation is a "no return" point. Specific implementations should rely in hardware resources such as sleep modes to allow the system to came back from a halt.

- `static bool tsl(volatile bool & lock)`
  This function is reimplemented in the CPU mediators of archtectures providing better ways to garantee an atomic register value change. A basic implementation in CPU_Common uses C code to change a boolean value, which is not garanteed to be atomic.

- `static int finc(volatile int & number)`
  This function is reimplemented in the CPU mediators of archtectures providing better ways to garantee an atomic register value increment. A basic implementation in CPU_Common uses C code to increment an integer value, which is not garanteed to be atomic.

- `static int fdec(volatile int & number)`
  This function is reimplemented in the CPU mediators of archtectures providing better ways to garantee an atomic register value decrement. A basic implementation in CPU_Common uses C code to decrement an integer value, which is not garanteed to be atomic.

4.7.2. MMU

The MMU is a hardware mediator responsible for abstracting the memory management and memory protection from the hardware. It's generally abstract the Memory Management Unit (MMU) of the target architecture, or provide a software implementation for this functions. The class diagram bellow shows the hierarchy of the low level memory abstractions.

More information can be found in EPOS Developer's Guide.

4.7.3. TSC

The Time Stamp Counter (TSC) is responsible for counting CPU ticks. If a given platform does not feature a hardware TSC, its functionality may be emulated by an ordinary periodic timer. Basically, the TSC API is formed by the **Hertz frequency()** and **Time_Stamp time_stamp()** methods. The first returns the TSC or timer frequency. The second, returns the current number of ticks.

Methods

- `Hertz frequency()`
- `Time_Stamp time_stamp()`

Types

- `typedef unsigned long Hertz`
- `typedef unsigned long long Time_Stamp`

4.7.4. Machine

The Machine mediator is responsible for abstracting target platform. It also provides a set of class methods that implement machine related functions (e.g.: panic, reboot, power off, etc).

Generic implementations of Machine interface are provided by Machine_Common. Machine-specifc implementations are provided by each machine's Machine mediator (e.g., PC, ATMega128, Plasma, etc).

The Machine mediator also defines the io map (Machine::IO), a structure responsible for abstracting each platform I/O address space.

Methods

- `static void delay(const RTC::Mircrossecond & time)`
- `static void panic()`
  This function should be called by the operating system when it "doesn't know" how to revert an error state. When called, it stops all system activities in order to avoid a greater damage.

Note: calling panic() is a "no return" point, i.e., there's no way to recover from a panic state but rebooting the system.

- `static void reboot()`
  When called, this function causes the system to be shutdown and rebooted.
- `static void poweroff()`
  When called, this function causes the system to be shutdown.
- `static unsigned int n_cpus()`
  This function returns the number of CPUs present in the current platform (to be used in SMP configurations). Returns 1 when no SMP configuration is available.
- `static unsigned int cpu_id()`
  This function returns the ID of the CPU in which the code is currently running (to be used in SMP configurations). Returns 1 when no SMP configuration is available.
- `static void smp_init(unsigned int n_cpus)`
  This functions initializes a SMP configuration (when available).
- `static void smp_barrier(int n_cpus)`
  This functions implements a barrier to enforce synchronization of all CPUs.
- `static void init()`
  This function is called at system startup and is responsible to configure the platform and get the system ready to start other components initialization.

4.7.5. IC

The IC mediator is responsible for abstracting target platform's scheme/hardware for handling interrupts/exceptions (referred to only as "interrupts" in the remaining of the text). It also provides a set of methods enable/disable interrupts and to assign interrupt handlers.

Bellow are the signatures for the component's interface methods. **Interrupt_Id** is a enumeration of the available interrupt request queues (IRQs), and is defined for each implementation of the IC mediator. **Interrupt_Handler** is the following function typedef:

```
typedef void (* Interrupt_Handler)();
```

What means that a interrupt handler should be a method with the following signature:

```
void handler();
```

Methods

- `static void int_vector(Interrupt_Id irq, Interrupt_Handler handler)`
  This method maps handler to a given IRQ.

- `static void enable(Interrupt_Id irq)`
  Enables interrupts for a given IRQ.

- `static void disable()`
  Disables all interrupts.

- `static void disable(Interrupt_Id irq)`
  Disables interrupts for a given IRQ.

4.7.6. RTC

The RTC family of mediators is responsible for keeping track of current time. It defines two types, as shown below, **Microsecond** and **Second**.
RTC Types

- `typedef unsigned long Microsecond`
- `typedef unsigned long Second`

The RTC API is depicted in the Figure below. It has a inner class `Date` that defines a date structure composed by the year (_Y), month (_M), day (_D), hours (_h), minutes (_m), and seconds (_s), representing a Date.

Date Types

- `unsigned int _Y`
- `unsigned int _M`
- `unsigned int _D`
- `unsigned int _h`
- `unsigned int _m`
- `unsigned int _s`

RTC Methods

- `RTC()`
  Constructs a RTC object.

- `Date date()`
  Returns a Date object that contains the current date.

- `void date(const Date & d)`
  Sets a date received by argument.

- `Second seconds_since_epoch()`
  Returns the number of seconds since an EPOCH. The EPOCH is defined in the Machine Traits. For instance, Traits<PC_RTC>::EPOCH_DAYS.

Date Methods

- `Date()`
- `Date(unsigned int _Y, unsigned int _M, unsigned int _D, unsigned int _h, unsigned int _m, unsigned int _s)`
- `unsigned int year()`

- `unsigned int month()`
- `unsigned int day()`
- `unsigned int hour()`
- `unsigned int minute()`
- `unsigned int second()`
- `void operator <<`

4.7.7. Timers

The Timer family of mediators is responsible for counting time. Based on a given and configurable frequency, the timer will increment or decrement a counter until it reaches zero or a pre-defined value. When this happens, an interrupt is generated and the event is handled by the specific timer interrupt handler. Each machine timer can be configured (its frequency) in its Traits class. The EPOS Timer family of mediators defines three types as shown below:

Types

- `typedef TSC::Hertz Hertz`
- `typedef TSC::Hertz Tick`
- `typedef Handler::Function Handler`

There are some differences between the timers of each architecture, but the common API is presented below.

Methods

- `void enable()`
  Enables the timer by turning on the timer interrupt.
- `void disable()`
  Disables the timer by turning off the timer interrupt.
- `Hertz frequency()`
  Returns the current timer frequency.
- `void frequency(Hertz & f)`
  Sets the timer frequency to **f**.
- `void reset()`
  Resets the timer counter.
- `Tick read()`
  Reads the current timer counter value.
- `int init()`
  Initializes the timer. This method must only be called by the system during the system bootstraping.

PC Timer API

The PC machine has only one Timer, named **Timer**. The Scheduler_Timer, Alarm_Timer, and user-defined Timers are multiplexed transparently by Timer.

Timer(const Hertz & frequency, const Handler * handler, const Channel & channel, bool retrigger)
Creates a Timer with **frequency**, associates its handler to **handler**, defines if it will be **retrigger** or not, and sets its **channel**. The channel can be SCHEDULER or ALARM.

4.7.8. UART

UART (Universal Assrynchronous Receiver/Transmitter) is used for serial communication over a peripheral device serial port. The UART API in EPOS is presented below.

Methods

- `UART(unsigned int unit = 0)`
  Creates an UART object. The unit defines which hardware device is being used. By default, the first device is choosen.
- `UART(unsigned int baud, unsigned int data_bits, unsigned int parity, unsigned int stop_bits, unsigned int unit = 0)`
  Creates an UART object with the baud rate (*baud*), data bits number (*data_bits*), parity bits numere (*parity*), stop bis number (*stop_bits*), and unit (by default 0).
- `void config(unsigned int baud, unsigned int data_bits, unsigned int parity, unsigned int stop_bits)`
  Configure an UART with the baud rate (*baud*), data bits number (*data_bits*), parity bits number (*parit*), and stop bits number (''stop_bits').
- `void config(unsigned int * baud, unsigned int * data_bits, unsigned int * parity, unsigned int * stop_bits)`
  Configure an UART with the baud rate (**baud*), data bits number (**data_bits*), parity bits number (**parity*), and stop bits number (**stop_bits*).
- `char get()`
  Gets a byte from an UART device. The method will wait until the data is ready.
- `void put(char c)`
  Sends a byte (*c*) to an UART device. The method will wait until the data is transfered.

4.7.8.1. Example

```
// EPOS PC UART Mediator Test Program

#include <utility/ostream.h>
#include <uart.h>

using namespace EPOS;

int main()
{
    OStream cout;

    cout << "PC_UART test\n" << endl;

    PC_UART uart(115200, 8, 0, 1);

    cout << "Loopback transmission test (conf = 115200 8N1):";
    uart.loopback(true);

    for(int i = 0; i < 256; i++) {
        uart.put(i);
        int c = uart.get();
        if(c != i)
            cout << " failed (" << c << ", should be " << i << ")!" << endl;
    }
    cout << " passed!" << endl;

    cout << "Link transmission test (conf = 9200 8N1):";
    uart.config(9600, 8, 0, 1);
    uart.loopback(false);

    for(int i = 0; i < 256; i++) {
        uart.put(i);
        for(int j = 0; j < 0xffffff; j++);
        int c = uart.get();
        if(c != i)
            cout << " failed (" << c << ", should be " << i << ")!" << endl;
    }
    cout << " passed!" << endl;

    return 0;
}
```

4.7.9. NIC

The Network Interface Card (NIC) family of hardware mediators provides access to network interface cards. All NIC devices implement the minimal interface specified bellow:

- `NIC(unsigned int unit=0)`
  Specifies the **unit** to be instantiated based on the order defined in System: :Traits: :‹Machine_NIC›::NICS.

- `~NIC()`
  Destructs a NIC previously created. It deallocates all memory used by the NIC.

- `int send(const Address, const Protocol &prot, const void *data, unsigned int size)`
  Sends `size` bytes of data to `dst` with protocol `prot` .
- `int receive(Address *src, Protocol *prot, void *data, unsigned int size)`
  Receives `size` bytes of data, `src` and `prot` are set by the method accordingly.
- `void reset()`
  Resets the NIC device.
- `unsigned int mtu()`
  Returns the device mtu (Maximum Transmission Unit).
- `const Address address()`
  Returns the device address.
- `const Statistics statistics()`
  Returns the NIC Statistics (which provides transmission and reception statistics).

4.7.10. Radio

The Low Power Radio family describes a set of methods and structures common for MAC (Medium Access Control) protocols for low-power radios. This includes packet format, the addressing word size, a structure for storing transmission statistics, and methods for sending and receiving data frames.

4.7.11. EEPROM

EEPROMs (Electrically-Erasable Programmable Read-Only Memory) are non-volatile storage device. An EEPROM have have a high read/write latency and are not area-efficient, so it's commonly used to store small configuration data. EEPROMs also have a limited life - that is, the number of times it can be reprogrammed is limited to tens or hundreds of thousands of times. Bellow is shown the public interface for the EEPROM mediator.

Methods

- `unsigned char read(const Address & a)`
  Reads and returns the byte stored at address *a*
- `void write(const Address & a, unsigned char d)`
  Reprograms the EEPROM. Writes byte *d* at address *a*
- `int size()`
  Returns the EEPROM size

THIS MUST BE RELOCATED

$EPOS/include/machine/$MACH/memory_map.h

```
template<>
struct Memory_Map<PC>
{
    // Physical Memory
    enum {
        MEM_BASE =      Traits<PC>::MEM_BASE,
        MEM_TOP =       Traits<PC>::MEM_TOP
    };

    // Logical Address Space
    enum {
        APP_LOW =       Traits<PC>::APP_LOW,
        APP_CODE =      Traits<PC>::APP_CODE,
        APP_DATA =      Traits<PC>::APP_DATA,
        APP_HIGH =      Traits<PC>::APP_HIGH,

        PHY_MEM =       Traits<PC>::PHY_MEM,
        IO =            Traits<PC>::IO_BASE,
        APIC =          IO,
        VGA =           IO +  4 * 1024,
        PCI =           IO + Traits<PC_Display>::FRAME_BUFFER_SIZE,

        SYS =           Traits<PC>::SYS,
        IDT =           SYS + 0x00000000,
        GDT =           SYS + 0x00001000,
        SYS_PT =        SYS + 0x00002000,
        SYS_PD =        SYS + 0x00003000,
        SYS_INFO =      SYS + 0x00004000,
        TSS0 =          SYS + 0x00005000,
        SYS_CODE =      SYS + 0x00300000,
        SYS_DATA =      SYS + 0x00340000,
        SYS_STACK =     SYS + 0x003c0000,
        SYS_HEAP =      SYS + 0x00400000
    };
};
```

For an detailed explanation about the meaning of the above constants, please refer to the EPOS Developer's guide.

When **tasks** are being used, the **Address_Space** abstraction is used to abstracts the memory segments that belongs to the address space of a task. Its public interface is described bellow. For more information see the Task and MMU abstraction.

Review Log

Ver	Date	Authors	Main Changes

1.0	Apr 4, 2016	Rodrigo Meurer	Import and cleanup from EPOS 1 documentation; Substitution of JPEG UML images by textual class interfaces
1.1	Apr 10, 2016	Guto Fröhlich	Major review

Menu

Home

Overview

Documentation

EPOS Software

EPOS Hardware

EPOS Makers

Publications

Login

Register