EPOS User Guide

Table of contents

1. Introduction

This document is a reference to the EPOS API. It is designed for those who want to get start with EPOS use and development.

1.1. EPOS Overview

EPOS (*Embedded Parallel Operating System*) 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 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.

1.2. Supported Architectures

EPOS supports a wide range variety of architectures, varying from 32 to 8 bits. The current supported architectures are:

- IA32 (single and multi-core)
- AVR8 (atmega16, atmega128, atmega1281, and at90can128 microcontrollers)
- PPC32 (including ml310)
- MIPS (including plasma soft-core)
- ARM7

1.3. EPOS License

Question and Answers

1.4. Documentation Roadmap

The EPOS User Guider is organized in three parts:

- Installing EPOS

This part describes how to download and install EPOS. Moreover, it presents what are the system requirements in order to run EPOS.

- Running EPOS

This part explains the basic features for running EPOS. It presents the configuration modes, teaches how to set up a specific architecture and machine, common installation problems, EPOS configurability features and tools used during the system generation, and how to compile and run an EPOS application in several architectures.

- API Reference

The last part is the EPOS API reference. It presents the system components, system utilities, and hardware mediators.

2. Downloading and Installing

2.1. System Requirements

EPOS is a cross-compiled system under GNU/Linux for several host or target platforms, using GNU Compiler Collection (GCC).
Binary distributions of these compilers are available in the EPOS website:

- IA32
  - available on-line: *ia32-gcc-4.4.4*
  - extract the file into */usr/local/ia32*
- ARM
  - available on-line: *arm-gcc-4.4.4*
  - extract the file into */usr/local/arm*
- AVR8
  - available on-line: *avr-gcc-4.0.2*
  - extract the file into */usr/local/avr*
- PPC32
  - available on-line: *ppc32-gcc-4.0.2*
  - extract the file into */usr/local/ppc32*
- MIPS32
  - available on-line: *mips-gcc-4.0.2*
  - extract the file into */usr/local/mips*

GCC manual documents how to use the GNU compilers and the internals of the GNU compilers, including how to port them to new targets, as well as their features and incompatibilities. It corresponds to GCC version 4.0.2.
You can also see some information here.
The C preprocessor manual, you can see here.

2.2. Installing

- EPOS is available on-line through: EPOS Download Page
- In order to work with EPOS, you need to extract this file and set some environment variables:
  - export EPOS=/path/to/epos
  - export PATH=$PATH:$EPOS/bin

3. Running EPOS

3.1. Configuring

- EPOS Configuration is done through the definition of variables in $EPOS/makedefs as follows:
  - **MODE** : Configure EPOS System Architecture. Possible values are:
    - **Library**: System is linked with application.
    - **Builtin**: System and Application are on the same AddressSpace
    - **Kernel**: System and Application are on different AddressSpace with a SystemCall layer between them.
  - **ARCH** : Configure the Architecture used for generation of the system. Possible values are:
    - **ARCH_IA32**: Intel x86 32-bits Architecture
    - **ARCH_AVR8**: Atmel AVR 8-bits Architecture
    - **ARCH_PPC32**: IBM PowerPC 32-bits Architecture
    - **ARCH_MIPS32**: MIPS 32-bits Architecture
  - **MACH** : Configure the Machine used for generation of the system. Possible values are:
    - **MACH_PC**: Personal Computer Machines
    - **MACH_ATMEGA16**: Atmel ATMega16 Microprocessor Family
    - **MACH_ATMEGA128**: Atmel ATMega128 Microprocessor Family
    - **MACH_ATMEGA1281**: Atmel ATMega1281 Microprocessor Family
    - **MACH_AT90CAN128**: Atmel AT90CAN128 Microprocessor Family
    - **MACH_ML310**: Xilinx ML310 and ML403 Evaluation Boards
    - **MACH_PLASMA**: Plasma Microprocessor

- In the *$EPOS/makedefs*, set the machine (*MACH*) and architecture (*ARCH*) of the application in "*System configuration*" as follows:

$EPOS/makedefs

```
# Supported software architectures MODE_KERNEL := kernel MODE_BUILTIN := builtin MODE_LIBRARY := library # Supported hardware architectures ARCH_IA32 := ia32 ARCH_AVR8 := avr8 ARCH_PPC32 := ppc32 ARCH_MIPS32 := mips32 # Supported machines MACH_PC := pc MACH_ATMEGA16 := atmega16 MACH_ATMEGA128 := atmega128 MACH_ATMEGA1281 := atmega1281 MACH_AT90CAN128 := at90can128 MACH_ML310 := ml310 MACH_PLASMA := plasma # System configuration MODE := $(MODE_LIBRARY) ARCH := $(ARCH_IA32) MACH := $(MACH_PC)
```

3.2. Compiling

With the EPOS configured, you are ready to build the system. In *$EPOS*, just type:

```
$ make all
```

3.2.1. Common Installation Problems

Common problems on Ubuntu and Ubuntu x64.

- **Ubuntu 32 bits**

EPOS uses some specific commands and tools that are not in Ubuntu by default. The list of common

installation errors that you might get are describe below.

command as16 not found: if you are compiling EPOS for IA32 architecture, the assembler 16 bits must be installed in your machine. To do so, you must install bin86 package

sudo apt-get install bin86

make: tcsh: command not found: install tcsh

sudo apt-get install tcsh

pci_ids.h file not found: link the correct file

ln -s /usr/src/kernel version/include/linux/pci_ids.h /usr/include/linux/pci_ids.h

command source not found: make sh point to bash

sudo rm /bin/sh
and
sudo ln -s /bin/bash /bin/sh

- **Ubuntu 64 bits**

First step is to do the same procedure described for **Ubuntu 32 bits** (above).

After that, some other packages must be installed using apt-get. Command is bellow.

sudo apt-get install ia32-libs lib32stdc++6 libc6-i386 libc6-dev-i386
In **Ubuntu 14.04**, use the following packages:

sudo apt-get install lib32stdc++6 libc6-i386 libc6-dev-i386 lib32z1 lib32ncurses5 lib32bz2-1.0

With this, the Linux system is configured. Now you have to follow the guides in the section **Cross-compiling EPOS for IA32 from a x86_64 system** (bellow).

- **Cross-compiling EPOS for IA32 from a x86_64 system**

If you are trying to generate an EPOS image to IA32 from a x86_64 Linux system, you're likely to see assembly errors such as:

/tmp/ccLFqhYC.s: Assembler messages:
/tmp/ccLFqhYC.s:8: Error: suffix or operands invalid for `push'
/tmp/ccLFqhYC.s:33: Error: suffix or operands invalid for `pop'

/tmp/ccLFqhYC.s:70: Error: suffix or operands invalid for `pop'
This is because you need to tell the system that you want to build a 32-bits image rather than a 64-bits image (system default). To solve it, edit the file *makedefs* in the EPOS tree root, replacing the following line:

$(MACH_PC)_CC_FLAGS :=
with this line:

$(MACH_PC)_CC_FLAGS := -Xassembler --32
This can be found around line 130.

3.3. EPOS Tools

- **eposcc**: this tool is responsible for compiling your application to EPOS.
  - Firstly we need compile your application using the eposcc tool, as follow:
    - $ eposcc -D __ia32 -D __pc -c -ansi -O2 app/your_app.cc -o app/your_app.o
  - Now, you must link your application with the system, using one of the EPOS architecture mode (Library, Builtin and Kernel). If you decided use Library mode, use the following command:
    - $ eposcc --library -o app/your_app app/your_app.o
- **eposmkbi**: this tool builds the system boot image.
  - You must create an EPOS image for load in your machine, to do this use the command eposmkbi, as follows:
    - # eposmkbi img/your_app.img app/your_app
  - You can also create an EPOS image using a multi-file source code, as follows:
    - # eposmkbi img/your_app.img app/your_app1 app/your_app2 app/your_app3 app/your_app4 ...

3.4. Building

Once you have compiled EPOS, you are now ready to build your application. For this, you must use the eposcc tool, that is installed on system when you build EPOS. This tool is a shell for the GCC compiler, and accepts most options of GCC. This tools also are a shell for GCC Linker. We must build an unique system image to boot EPOS at the target platform. To do it, we will use the eposmkbi tool. For now we assume that your application is only implemented in a single source file, but the process presented here can be extended to a multi-file source code.

3.4.1. Creating an EPOS Application

- Create a new file in *$EPOS/app*:

$EPOS/app/helloworld.cc

```
#include <utility/ostream.h> __USING_SYS OStream cout; int main() { cout << "\n Hello World! \n"; return 0; }
```

3.4.2. Building an EPOS Application

With the EPOS configured and built, you are ready to build the application. You can also use the makefile provided above. In *$EPOS*, just type the following command:

```
$ make APPLICATION=helloworld
```

3.5. Running

Now you can download EPOS image on your target machine, this step is different for each platform you are using.
You can run EPOS application using a VM (Qemu, VMware, VirtualBox), AVR (atmega128, atmega16, atmega1281, at90can128), PPC or Mips.

3.5.1. IA32

3.5.1.1. Virtual Machine QEMU

In *$EPOS*, just type:

```
$ qemu -fda img/helloworld.img -serial stdio
```

3.5.1.2. Virtual Machine VMware Player

In the configuration file of VMware Player virtual machine, add or replace the following parameters and values:

/path/to/virtual/machine/my_mach.cfg

Floppy floppy0.present = "TRUE" floppy0.fileName = "$EPOS/img/helloworld.img" floppy0.startConnected = "TRUE" floppy0.fileType = "file" # Serial serial0.present = "TRUE" serial0.fileName = "$EPOS/helloworld.out" serial0.startConnected = "TRUE" serial0.fileType = "file"

In *$EPOS*, just type:

$ vmplayer /path/to/virtual/machine/my_mach.cfg
Sample file: my_mach.cfg

3.5.1.3. Virtual Machine VirtualBox

Just type:

$ VirtualBox

- To create a new virtual machine, click *New* and *Next*.
- Enter the name, **Epos**, for the Virtual machine and select the OS type. Here we can choose *Other* and *Other/Unknown* as the type and version of OS, click *Next*.
- Set the memory size for the virtual machine and click *Next*.
- In the next window, click *New* to create or add an existing Hard Disk, and click *Next*.
- Click *Next* and *Finish* to complete adding the virtual machine.
- To change the settings of the VM, click *Settings* and select *Floppy* or *Storage/Floppy* category. The EPOS system is presented to the virtual machine as an image file. For example:

$EPOS/img/helloworld.img

- In *Serial Ports* settings screen, a virtual serial port 1 is connected to *Port Number* **COM1**, and the *Port Mode* **Host Device**. Set the path to the host serial device. For example:

$EPOS/helloworld.out
Start the virtual machine **Epos**.

3.5.2. AVR8

In *$EPOS*, just type:

$ avr-objcopy -O ihex img/helloworld.img epos.hex

3.5.2.1. MACH_ATMEGA16

Using STK500:

avrdude -P /dev/ttyUSB0 -c stk500v2 -p atmega16 -U flash:w:epos.hex -C /usr/local/avr/tools/etc/avrdude.conf -F

3.5.2.2. MACH_ATMEGA128

If you are using MICA2 Mote, you can type:

$ uisp -dprog=mib510 -dpart=atmega128 -dserial=/dev/ttyUSB0 --erase --upload --verify if=epos.hex

3.5.2.3. MACH_ATMEGA1281

If you can use JTAG ICE, just type:

avrdude -p m1281 -c jtagmkII -P usb:00A0000025CB -U flash:w:epos.hex -v

3.5.2.4. MACH_AT90CAN128

Using STK500:

avrdude -P /dev/ttyUSB0 -c stk500v2 -p at90can128 -U flash:w:epos.hex -C /usr/local/avr/tools/etc/avrdude.conf -F

3.5.3. PPC32

3.5.4. MIPS32

The EPOS port to MIPS32 focus on the basic 32-bits MIPS architecture. It should work with any implementation of MIPS32. The following sections show how to get EPOS running on the MIPS implementations supported by EPOS.

3.5.4.1. Plasma

The Plasma CPU is a small synthesizable 32-bit RISC microprocessor. It executes all MIPS I user mode instructions except unaligned load and store operations. More information at opencores.org.

Although we recommend the use of the most recent version of plasma, we've made available here a version of the processor and other software tools that were tested and are know to work properly with EPOS:

- Plasma version xxx from yyy
- ELF loader for EPOS

The versions of the Plasma core and the ELF loader above were tested with a Xilinx Spartan-3 FPGA. A version of the Plasma processor running on the Spartan-3 FPGA is available at the EPOS Hardware download section. We recommend using Xilinx ISE 9.1 or newer for synthesizing it.

3.5.5. ARM7

3.5.5.1. EPOSMoteII (MC13224V)

There are two ways to program the ARM7 version of the EPOSMoteII: through its serial port or using a JTAG adapter. Both approaches are described bellow.

Programming EPOSMoteII through the serial port

We assume that you are using an EPOSMote-Startup board and have it connected to a Linux PC through an USB port. This is what you need before programming the EPOSMoteII through its serial port:

- GNU's binutils for ARM (objcopy). That is bundled together with GCCs available for download at EPOS' site;
- Python2, for running the programming scripts. That may be easily installed in your Linux system through its software update system;
- "red-bsl.py" and "ssl.bin" files, bundled together with OpenEPOS since version 1.1, under the "tools/emote" folder.

Copyright notice: "red-bsl.py" was developed by Andrew Pullin and is freely available under de terms of GNU's GPLv3 (see script's header for details). Thanks Andrew!

Having the tools installed, you'll need to:

1. Open a terminal and go to OpenEPOS' root directory;
2. Convert OpenEPOS' generated image from its default format (elf32-little) to a raw binary file by issuing the command bellow (we assume you are using OpenEPOS' default application, but you may adjust it to your application/image).
   arm-objcopy -I elf32-little -O binary img/mc13224v_app.img img/mc13224v_app.bin
3. Upload the image to EPOSMoteII RAM memory for testing. It is done by the command bellow. The "-t" option defines the USB port to which the EPOSMoteII is connected. You may need to adjust it to the

USB port Linux has assumed the EPOSMoteII is attached (it may be easily obtained by running a "dmesg" command). The "-f" command defines the binary that will be uploaded to the device. In this case, it is the output of the command from the previous step. During the upload procedure you may be asked to reset the board. If so, do it by pressing EPOSMoteII's reset button. After the upload, you'll be asked to press <enter> to start your application. This gives you time to open your serial terminal application to see the EPOSMoteII's outputs.
python red-bsl.py -t /dev/ttyUSB0 -f img/mc13224v_app.bin

4. If you wish, you may also upload the image to EPOSMoteII flash memory, but, be aware that:

IF YOU ARE USING A BETA OR ALFA VERSION OF EPOSMoteII YOU MAY NEED A JTAG ADAPTER IN ORDER TO ERASE THE FLASH MEMORY. OpenEPOS INCLUDES A FLASH ERASING UTILITY THAT MAY WORK (DETAILS BELLOW), BUT IF YOU UPLOAD A CORRUPTED IMAGE OF OpenEPOS TO EPOSMoteII FLASH, THIS FLASH WON'T BE ERASED UNLESS YOU HAVE A JTAG ADAPTER. TO IDENTIFY IF IT IS OK WITH YOUR EPOSMote, JUST CHECK THE BOARD: ALFA AND BETA VERSIONS OF EPOSMote DO *NOT* HAVE THE FLASH ERASING JUMPERS, LABELED J4 AND J5 ON THE BOARD. IF YOUR BOARD HAS THESE JUMPERS, JUST IGNORE THIS WARING.

That checked, follow these steps:

1. If you are using Alfa or Beta versions of EPOSMoteII:
    1. Be sure that the image you want to write to EPOSMoteII has the Traits<MC13224V>::flash_erase_checking boolean set to true (found in include/mach/mc13224v/traits.h). True is the default value. If it wasn't set to true, change it, run a "make clean" command, and rebuild OpenEPOS ("make all");
    2. Be sure that the image is operating correctly before writing it to the device's flash by testing it on the device's RAM before;
2. Add the "-S" option to the command-line that uploaded the image to the RAM. It should look like this:
   python red-bsl.py -t /dev/ttyUSB0 -f img/mc13224v_app.bin -S

Programming EPOSMoteII using a JTAG adapter

In order to do that, you'll need to:

1. Have a JTAG adapter for ARM processors;
2. Install OpenOCD and GNU GDB for ARM as described here;
3. Have a ".gdbinit" file at the root directory of your OpenEPOS tree with the following contents:

```
target remote localhost:3333
monitor gdb_breakpoint_override hard
define connect
    target remote localhost:3333
end
define reset
    monitor soft_reset_halt
    set *0x80020010 = 0
    set *0x80003050 = 0x87651234
    monitor reg pc 0x00400000

end
```

That provided, follow these steps to upload the image to the board:

1. Connect the JTAG adapter to the EPOSMoteII's JTAG interface;
2. Power-up the EPOSMoteII (either through a battery or through the USB interface at the EPOSMote-Startup board);
3. Start OpenOCD as described here;
4. Go to OpenEPOS root directory and start "arm-gdb" (so it would run the .gdbinit script);
5. Run the commands bellow under GDB. The "load" command will upload OpenEPOS' elf32-little image to the device's RAM. The "continue" command will start program execution.

reset
load img/mc13224v_app.img

continue

3.5.5.2. Integrator/CP

Integrator/CP is a reference ARM platform. We ported EPOS to it to be able to run the ARM7 version of EPOS in Qemu, making system development and port easier.

To run EPOS on Qemu Integrator/CP emulator just run under the EPOS root:

make run

It is actualy a short-cut to:

qemu-system-arm -no-reboot -nographic -m 128 -kernel integratorcp_app.img

3.5.6. Possible Causes of Running Problems

- AVR Machines
  - **Timer does not behave as expected**: make sure to set the appropriate timer frequency in the machine's traits file (include/mach/MACHINE/traits.h). Verify if the input clock source is correct (in case of using a STK500 platform for example).
  - **UART does not behave as expected**: make sure to set the appropriate machine clock in the machine's traits file (include/mach/MACHINE/traits.h - consult the microcontroller's datasheet to verify its clock frequency). Verify if the microcontroller's fuse bits are correct.

3.6. Configurability

Traits is a class used in place of template parameters. A *traits class* provides a way of associating information with a compile-time entity (a type, integral constant, or address). Its goal is to increase reuse and maintenance in Software Engineering by defining new programming language constructs.
In EPOS, the *traits* is used to set system stack size, system heap size, clock frequency, as well as define and/or enable features in system parts, mediators, utilities and abstractions.
In EPOS, you can enable debugging by setting a Boolean constant (debugged = true) in the following structure:

$EPOS/include/traits.h

```
template <class Imp> struct Traits { static const bool enabled = true; static const bool debugged = false; };
```

You can define which information the debug returns.

- ERR (error) informs when an error occurs.

- WRN (warning) shows warnings of possible errors and situations that can lead to errors.
- INF (info) shows debug information about events that occur during execution.
- TRC (trace) demonstrates the data flow and call flow of methods/functions.

$EPOS/include/traits.h

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

You can also enable debugging only in a specific class/object.

$EPOS/include/traits.h

```
template <> struct Traits<Lists>: public Traits<void> { static const bool debugged = false; };
```

Example, trace debugging:

$EPOS/include/utility/list.h

```
... void insert_head(Element * e) { db<Lists>(TRC) << "List::insert_head(e=" << e << ") => {p=" <<
(e ? e->prev() : (void *) -1) << ",o=" << (e ? e->object() : (void *) -1) << ",n=" << (e ? e->next() : (void
*) -1) << "}\n"; ...
```

After the changes made to the *traits*, in *$EPOS*, just type:

```
$ make veryclean all
```

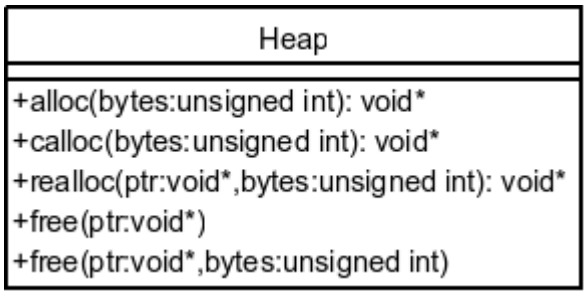
4. EPOS Programming

EPOS programming API is composed by two types of software structures: components or abstractions and hardware mediators. Components are C++ classes with a well-defined API and behavior. They are platform-independent. Platform-specific support is implemented through 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, thus dissolving mediator code into abstractions at compile-time. EPOS also offers common data structures, known as utilities, such as lists, vectors, and hash tables.

4.1. Components

4.1.1. Memory Management

The **Heap** abstraction is the higher level abstraction responsible for memory management on EPOS. It keeps a list of free memory blocks and handles allocation and deallocation requests. Its interface is described in the diagram bellow.



Two Heap instances are created on EPOS during the system initialization. One is used to implement the **malloc** and **new operator**, that are both used to handle memory allocation request from the application.

The other is used to implement the **kmalloc**, that is used to handle OS memory allocation requests(e.g memory for the thread's stack). The implementation of this functions are described bellow.

utility/malloc.h

```
... inline void * malloc(unsigned int bytes) { return __SYS(Application)::heap()->alloc(bytes); } inline void * calloc(unsigned int n, unsigned int bytes) { return __SYS(Application)::heap()->calloc(n * bytes); } inline void free(void * ptr) { __SYS(Application)::heap()->free(ptr); } inline void * operator new(unsigned int bytes) { return malloc(bytes); } inline void * operator new[](unsigned int bytes) { return malloc(bytes); } inline void operator delete(void * object) { free(object); } inline void operator delete[](void * object) { free(object); } ...
```

system/kmalloc.h

```
... inline void * kmalloc(unsigned int bytes) { return System::heap()->alloc(bytes); } inline void * kcalloc(unsigned int n, unsigned int bytes) { return System::heap()->calloc(n * bytes); } inline void kfree(void * ptr) { System::heap()->free(ptr); } ...
```

The size of the memory blocks that will be managed by each heap is machine dependent and is defined in the **Traits**:

traits.h

```
... template <> struct Traits<Machine>: public Traits<Machine_Common> { ... static const unsigned int APPLICATION_STACK_SIZE = 16 * 1024; static const unsigned int APPLICATION_HEAP_SIZE = 16 * 1024 * 1024; static const unsigned int SYSTEM_STACK_SIZE = 4096; static const unsigned int SYSTEM_HEAP_SIZE = 16 * APPLICATION_STACK_SIZE; ... }; ...
```

APPLICATION_HEAP_SIZE and **SYSTEM_HEAP_SIZE** are the size of the application heap and the system heap in bytes. **APPLICATION_STACK_SIZE** is the size of the stack of application threads, like the **main thread**, created by the system to run the application, and threads created by the application. **SYSTEM_STACK_SIZE** is the size of the threads created by the system to execute system tasks, for example, the **idle thread** that is scheduled when there is no other threads ready to run.

This parameters must be set according to the applications requirements and the memory available on the target platform. For example, all the threads stacks are allocated using the system heap, so **APPLICATION_STACK_SIZE**, **SYSTEM_STACK_SIZE**, and **SYSTEM_HEAP_SIZE** must be sized so the system heap can allocate memory for all the stacks. Note that **APPLICATION_HEAP_SIZE + SYSTEM_HEAP_SIZE** must be equal or less than the amount of memory available for data minus the memory that the compiler used for global variables allocation.

The figure below shows an example that exposes the relationship mentioned above.

Everything allocate using **kmalloc**, the **main** and **idle** thread structures, and all the threads stacks, are allocated on the system heap. Everything allocated with **malloc** and **new**, and other application variables, are allocated on the application heap.

4.1.1.1. Memory Segments

Memory segments are used to abstract a logical memory segment allocated by the MMU. For more details see the MMU abstraction.

4.1.1.2. Address Spaces

A different **Memory_Map** abstraction exists for each machine and it defines the memory map for that machine (e.g the memory mapping of IO devices). Besides defining the machine specific memory map, this abstractions must define at least the following constants:

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

```
template <> struct Memory_Map<Machine> { enum { MEM_BASE = 0, MEM_SIZE = 4096, }; enum { BASE = 0x000000, TOP = 0x001000, APP_LO = 0x000000, APP_CODE = 0x000000, APP_DATA = 0x800150, APP_HI = 0x00ffff, PHY_MEM = 0x800100, IO_MEM = 0x800020, SYS = 0x000000, INT_VEC = 0x000000, SYS_INFO = 0x000100, SYS_CODE = 0x000000, SYS_DATA = 0x800150, SYS_STACK = 0x8010ff }; };
```

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.

4.1.2. Process Management

In EPOS, process management is accomplished by three components: Task, Thread and Scheduler. These components follow the classic definition of operating systems components. The implementation of such concepts in EPOS, however, was done in a innovative way. Innovative in the sense that these concepts are, actually, implemented as a family of components (classes), with each one of these components being responsible for implementing a specific version of the concept. For example, the Thread component may be a simple thread, implemented by the Thread class, or a periodic thread, implemented by the Periodic_Thread class, and so on. In a similar way, different schedulers are also implemented as different classes. In order to avoid an explosion in the number of classes in the system, the EPOS implementation for these components relies heavily in static meta-programming techniques (i.e., C++ templates). The following sections will make a better description of both concepts and EPOS implementation for these components.

4.1.2.1. Task

A task (as well as a thread) is a program in execution. In EPOS, the Task component abstracts the classic concept of Process. A process is not only composed by its code (*text section*). It also embraces other entities such as its context (CPU registers, state, etc), data memory (*data section*) and a stack memory (for temporary data). An important characteristic of a process is the notion of address space abstraction. Conceptually, each process, when running, assumes it is alone in the system (CPU), and has access to the whole system's address space. In EPOS, the address space concept (which is related to virtual memory systems) is abstracted by the system's memory manager. The Task implementation uses the EPOS' Address_Space abstraction to map physical memory segments (the Segment abstraction) into its virtual Address_Space. This is usually useful in systems featuring a MMU enabling address space protection. In embedded systems, however, such mechanisms are seldom available. In such systems, EPOS creates, during the system setup, a single process (Task), which maps the whole physical memory to an unique Task, allowing multi-tasking to be accomplished through the Thread component.

Summarizing, if the system you are planning to use will feature a MMU and/or has some address space protection or address translation mechanism, and you are willing to use such mechanisms, go on in this

section and use the Task component. If not, if your system is a simple microcontroler, or other processor without a MMU or any other memory protection mechanism, jump this section and use the Thread component.

TODO: Threads will be more important and more useful, so this section will be finished later.

4.1.2.2. Thread

A thread is, perhaps, a simplification of a task. "Perhaps" because, although simplifying data exchange and speeding up context switching, it brings rise to new issues which must be dealt in order the control the concurrent execution. Informally, a thread is a process without an exclusive address space. Although each thread has its own context (CPU registers, state, etc) and stack memory, threads, as opposed to tasks, share the same data memory (heap). EPOS provides a feature-full implementation for the Thread component. The figure bellow shows the Thread component interface (hiding implementation details) and its relation to other system components.

These are the C++ signatures for the Periodic_Thread interface and the description of each method:

```
Thread(int (* entry)(),
const State & state = READY, const Criterion & criterion = NORMAL, unsigned int stack_size
= STACK_SIZE)
```

Creates a thread with the following parameters:

- `entry`: entry point for the thread (defines the thread behavior). `entry` should be a C++ function with signature `int func()`.
- `state`: defines the state of the thread upon its creation. Default value is READY, i.e., it is able to run the next time the defined period is reached.
- `criterion`: defines the criterion to be used for this thread. The criterion is based on the Criterion defined by the Scheduler. It is better explained in the Scheduler section of this guide.
- `stack_size`: defines the size of the thread's stack. By default it takes the value set by the system's Traits. If a larger (or smaller) stack is desired, this parameter will allow you to do so.

```
template<typename T1>
Thread(int (* entry)(T1 a1), T1 a1,
const State & state = READY, const Criterion & criterion = NORMAL, unsigned int stack_size
= STACK_SIZE)
```

Creates a thread. The difference from the first constructor version is that with this constructor you are able to pass 1 parameter to the entry point function through the thread's constructor. The constructor parameters are:

- `entry`: parametrized entry point for the thread (defines the thread behavior). `entry` should be a C++ function with signature `int func(T1 a1)`, where `T1` defines the type for the argument `a1` and can be of any type.
- `a1`: argument 1 to the entry point function, of type `T1`.
- `state`: defines the state of the thread upon its creation. Default value is READY, i.e., it is able to run the next time the defined period is reached.
- `criterion`: defines the criterion to be used for this thread. The criterion is based on the Criterion defined by the Scheduler. It is better explained in the Scheduler section of this guide.

- `stack_size`: defines the size of the thread's stack. By default it takes the value set by the system's Traits. If a larger (or smaller) stack is desired, this parameter will allow you to do so.

```
template<typename T1, typename T2>
Thread(int (* entry)(T1 a1, T2 a2), T1 a1, T2 a2,
const State & state = READY, const Criterion & criterion = NORMAL, unsigned int stack_size
= STACK_SIZE)
```

Creates a thread. The difference from the first constructor version is that with this constructor you are able to pass 2 parameter to the entry point function through the thread's constructor. The constructor parameters are:

- `entry`: parametrized entry point for the thread (defines the thread behavior). `entry` should be a C++ function with signature `int func(T1 a1, T2 a2)`, where `T1` and `T2` define the type for the arguments `a1` and `a2` and can be of any type.
- `a1`: argument 1 to the entry point function, of type `T1`.
- `a2`: argument 2 to the entry point function, of type `T2`.
- `state`: defines the state of the thread upon its creation. Default value is READY, i.e., it is able to run the next time the defined period is reached.
- `criterion`: defines the criterion to be used for this thread. The criterion is based on the Criterion defined by the Scheduler. It is better explained in the Scheduler section of this guide.
- `stack_size`: defines the size of the thread's stack. By default it takes the value set by the system's Traits. If a larger (or smaller) stack is desired, this parameter will allow you to do so.

```
template<typename T1, typename T2, typename T3>
Thread(int (* entry)(T1 a1, T2 a2, T3 a3), T1 a1, T2 a2, T3 a3,
const State & state = READY, const Criterion & criterion = NORMAL, unsigned int stack_size
= STACK_SIZE)
```

Creates a thread. The difference from the first constructor version is that with this constructor you are able to pass 3 parameter to the entry point function through the thread's constructor. The constructor parameters are:

- `entry`: parametrized entry point for the thread (defines the thread behavior). `entry` should be a C++ function with signature `int func(T1 a1, T2 a2, T3 a3)`, where `T1`, `T2`, and `T3` define the type for the arguments `a1`, `a2`, and `a3`, and can be of any type.
- `a1`: argument 1 to the entry point function, of type `T1`.
- `a2`: argument 2 to the entry point function, of type `T2`.
- `a3`: argument 3 to the entry point function, of type `T3`.
- `state`: defines the state of the thread upon its creation. Default value is READY, i.e., it is able to run the next time the defined period is reached.
- `criterion`: defines the criterion to be used for this thread. The criterion is based on the Criterion defined by the Scheduler. It is better explained in the Scheduler section of this guide.
- `stack_size`: defines the size of the thread's stack. By default it takes the value set by the system's Traits. If a larger (or smaller) stack is desired, this parameter will allow you to do so.

```
~Thread()
```

The thread destructor stops the thread execution, call the `exit(int)` function (if not explicitly called before), and deletes the Thread object.

`const volatile State & state()`

Returns the thread's state as a `State` type. `State` is an enumeration defined as follows:

periodic_thread_test.cc

```
enum State { BEGINNING, READY, RUNNING, SUSPENDED, WAITING, FINISHING };
```

`const volatile Criterion & criterion() const`

Returns the value of the scheduling criterion assigned to a thread. The type `Criterion` is defined as a member of the `Scheduling_Criteria` namespace. It is better explained in the Scheduler section of this guide.

`Priority priority() const`

Returns the priority of a thread as defined by the adopted `Criterion`. It is better explained in the Scheduler section of this guide.

`void priority(const Priority & p)`

Resets the thread's priority to the value of `p`. `Priority` is defined by the selected `Criterion`. It is better explained in the Scheduler section of this guide.

`int join()`

The `join()` method suspends the execution of the calling thread (i.e., the thread that is running) until the called thread finishes its execution.

`void pass()`

The `pass()` method forces the scheduler to schedule the called thread. The calling thread (i.e., the thread that is running) is put back into the scheduling queue, as `READY`. This procedure is only accomplished if the called thread is able to run, i.e., it is in `READY` state. If this condition does not hold, the `pass()` method will return and the calling thread continues its normal execution.

`void suspend()`

This method puts the thread in the `SUSPENDED` state by removing it from the scheduling queue. By calling this method, another thread is scheduled based on the defined scheduling criterion.

`void resume()`

This method puts the thread in the `READY` state by inserting it into the scheduling queue. Note that this method does not schedule the thread right away. It just puts the thread in the `READY` state and reinserts it into the scheduling queue. The thread execution is dependent on the selected scheduling criterion.

`static Thread * self()`

This static method returns a pointer to the current thread, i.e., returns a pointer to a Thread object representing the thread executing in the moment the method is called.

`static void yield()`

This static method forces a reschedule, i.e., it replaces the currently running thread by another thread based on the selected scheduling criterion. The thread that was removed from the `RUNNING` state is put in `READY` state, is reinserted into the scheduling queue, and, thus, is subject to be rescheduled anytime, based on the selected scheduling criterion.

`static void sleep(Queue * q)`

The `sleep(Queue)` method (as well as `wakeup(Queue)` `wakeup_all(Queue)`) are used by EPOS synchronizers, specially the `Condition` component (condition variable), to implement the process synchronization procedures. By calling the `sleep(Queue)` method a thread is put in `WAITING` state, and it is prevented from executing until `wakeup(Queue)` is called to the same thread object.

`static void wakeup(Queue * q)`

The `wakeup(Queue)` method (as well as `sleep(Queue)` `wakeup_all(Queue)`) are used by EPOS synchronizers, specially the `Condition` component (condition variable), to implement the process synchronization procedures. By calling the `wakeup(Queue)` method a thread is put back in `READY` state and reinserted into the scheduling queue.

`static void wakeup_all(Queue * q)`

The `wakeup_all(Queue)` method (as well as `sleep(Queue)` `wakeup(Queue)`) are used by EPOS synchronizers, specially the `Condition` component (condition variable), to implement the process synchronization procedures. By calling the `wakeup_all(Queue)` all threads in `q` are put back in `READY` state and reinserted into the scheduling queue.

`static void exit(int status = 0)`

By calling this method, the currently running thread is stopped and put in `FINISHING` state. If there are "joining threads" for the running thread (i.e., threads that called `join()` for the running thread), these threads have its state set back to `READY` and are reinserted into the scheduling queue.

`static void init()`

This method initializes the Thread component, by creating the first thread (-+main()-+), activating the scheduler, and starting the application. There is no need to call this method from the application. The system calls it automatically after the boot.

4.1.2.3. Periodic_Thread

There is also a specialization of the Thread component called Periodic_Thread. A periodic thread is a special case of thread that runs once every time a given period is reached. This component interface is depicted in the figure bellow.

These are the C++ signatures for the Periodic_Thread interface and the description of each method:

`Periodic_Thread(int (* entry)(), const Microsecond & period, int times = Alarm::INFINITE, const State & state = READY, unsigned int stack_size = STACK_SIZE)`

Creates a periodic thread with the following parameters:

- `entry`: entry point for the thread (defines the thread behavior). `entry` should be a C++ function with signature `int func()`.
- `period`: defines the interval between the executions of this thread.
- `times`: defines how many times the periodic thread should run. Default value is infinite, i.e., it will execute forever.
- `state`: defines the state of the thread upon its creation. Default value is READY, i.e., it is able to run the next time the defined period is reached.
- `stack_size`: defines the size of the thread's stack. By default it takes the value set by the system's Traits. If a larger (or smaller) stack is desired, this parameter will allow you to do so.

For more information see the code example bellow (lines 30, 31, 32).

```
template<class T1>
Periodic_Thread(int (* entry)(T1 a1), T1 a1,
const Microsecond & period, int times = Alarm::INFINITE,
const State & state = READY, unsigned int stack_size = STACK_SIZE)
```

Creates a periodic thread. The difference from the first constructor version is that with this constructor you are able to pass 1 parameter to the entry point function through the thread's constructor. The constructor parameters are:

- `entry`: parametrized entry point for the thread (defines the thread behavior). `entry` should be a C++ function with signature `int func(T1 a1)`, where `T1` defines the type for the argument `a1` and can be of any type.
- `a1`: argument 1 to the entry point function, of type `T1`.
- `period`: defines the interval between the executions of this thread.
- `times`: defines how many times the periodic thread should run. Default value is infinite, i.e., it will execute forever.
- `state`: defines the state of the thread upon its creation. Default value is READY, i.e., it is able to run the next time the defined period is reached.
- `stack_size`: defines the size of the thread's stack. By default it takes the value set by the system's Traits. If a larger (or smaller) stack is desired, this parameter will allow you to do so.

For more information take a look at the semaphore_test.cc example at the EPOS tree. It uses parameters to the `int philosopher(int n, int l, int c)` entry function.

```
template<class T1, class T2>
Periodic_Thread(int (* entry)(T1 a1, T2 a2), T1 a1, T2 a2,
const Microsecond & period, int times = Alarm::INFINITE,
const State & state = READY, unsigned int stack_size = STACK_SIZE)
```

Creates a periodic thread. The difference from the first constructor version is that with this constructor you are able to pass 2 parameters to the entry point function through the thread's constructor. The constructor parameters are:

- `entry`: parametrized entry point for the thread (defines the thread behavior). `entry` should be a C++ function with signature `int func(T1 a1, T2 a2)`, where `T1` and `T2` define the type for the arguments `a1` and `a2` and can be of any type.
- `a1`: argument 1 to the entry point function, of type `T1`.
- `a2`: argument 2 to the entry point function, of type `T2`.
- `period`: defines the interval between the executions of this thread.
- `times`: defines how many times the periodic thread should run. Default value is infinite, i.e., it will

execute forever.
- `state`: defines the state of the thread upon its creation. Default value is READY, i.e., it is able to run the next time the defined period is reached.
- `stack_size`: defines the size of the thread's stack. By default it takes the value set by the system's Traits. If a larger (or smaller) stack is desired, this parameter will allow you to do so.

For more information take a look at the semaphore_test.cc example at the EPOS tree. It uses parameters to the `int philosopher(int n, int l, int c)` entry function.

```
template<class T1, class T2, class T3>
Periodic_Thread(int (* entry)(T1 a1, T2 a2, T3 a3), T1 a1, T2 a2, T3 a3,
const Microsecond & period, int times = Alarm::INFINITE,
const State & state = READY, unsigned int stack_size = STACK_SIZE)
```

Creates a periodic thread. The difference from the first constructor version is that with this constructor you are able to pass 3 parameters to the entry point function through the thread's constructor. The constructor parameters are:

- `entry`: parametrized entry point for the thread (defines the thread behavior). `entry` should be a C++ function with signature `int func(T1 a1, T2 a2, T3 a3)`, where `T1`, `T2`, and `T3` define the type for the arguments `a1`, `a2`, and `a3`, and can be of any type.
- `a1`: argument 1 to the entry point function, of type `T1`.
- `a2`: argument 2 to the entry point function, of type `T2`.
- `a3`: argument 3 to the entry point function, of type `T3`.
- `period`: defines the interval between the executions of this thread.
- `times`: defines how many times the periodic thread should run. Default value is infinite, i.e., it will execute forever.
- `state`: defines the state of the thread upon its creation. Default value is READY, i.e., it is able to run the next time the defined period is reached.
- `stack_size`: defines the size of the thread's stack. By default it takes the value set by the system's Traits. If a larger (or smaller) stack is desired, this parameter will allow you to do so.

For more information take a look at the semaphore_test.cc example at the EPOS tree. It uses parameters to the `int philosopher(int n, int l, int c)` entry function.

`static void wait_next()`
This method should be used within the entry point function to indicate that the thread execution for this cycle is done and now it will wait for the next period to be reached. Usually this function is used to block a loop that implements the thread actions. For more information see the code example bellow (lines 60, 71, 82).

periodic_thread_test.cc

// EPOS-- Periodic Thread Abstraction Test Program #include <utility/ostream.h> #include <periodic_thread.h> #include <chronometer.h> __USING_SYS const int iterations = 100; const int period_a = 100; // ms const int period_b = 200; // ms const int period_c = 400; // ms int 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; int main() { cout << "Periodic Thread Abstraction Test\n"; cout << "\nThis test consists in creating three periodic threads as follows:\n"; cout << " Thread 1 prints \"a\" every " << period_a << " ms;\n"; cout << " Thread 2 prints \"b\" every " << period_b << " ms;\n"; cout << " Thread 3 prints \"c\" every " << period_c << "ms.\n"; Periodic_Thread thread_a(&func_a, period_a * 1000, iterations); Periodic_Thread thread_b(&func_b, period_b * 1000, iterations); Periodic_Thread thread_c(&func_c, period_c * 1000, iterations); cout << "Threads have been created. I'll wait for them to finish...\n\n"; Chronometer chrono; chrono.start(); int status_a = thread_a.join(); int

status_b = thread_b.join(); int status_c = thread_c.join(); chrono.stop(); cout << "\n\nThread A exited with status " << status_a << ", thread B exited with status " << status_b << " and thread C exited with status " << status_c << ".\n"; cout << "\nThe estimated time to run the test was " << max(period_a, period_b, period_c) * iterations << " ms. The measured time was " << chrono.read() / 1000 <<" ms!\n"; cout << "I'm also done, bye!\n"; return 0; } int func_a() { cout << "A"; for(int i = 0; i < iterations; i++) { Periodic_Thread::wait_next(); cout << "a"; } cout << "A"; return 'A'; } int func_b(void) { cout << "B"; for(int i = 0; i < iterations; i++) { Periodic_Thread::wait_next(); cout << "b"; } cout << "B"; return 'B'; } int func_c(void) { cout << "C"; for(int i = 0; i < iterations; i++) { Periodic_Thread::wait_next(); cout << "c"; } cout << "C"; return 'C'; }

4.1.2.4. Scheduler

EPOS provides a family of schedulers. This family includes traditional schedulers (round-robin, priority, etc) and real-time schedulers (EDF, RM, etc). EPOS' schedulers are configurable, and its parameters can be set at the Traits structure. The implementation of the EPOS' scheduler is a quite complex piece of code, thus, this "user guide" section will focus on the scheduler configuration rather than on its internals.

The system only has one scheduler. This component operates accordingly to a Scheduling Criterion defined at configuration time. This criterion is set by the application programmer at configuration time. Once Thread is the only component which depends on the system scheduler, the scheduling configuration is done through the Thread's traits. The code bellow depicts the default scheduler configuration:

traits.h

template <> struct Traits<Thread>: public Traits<void> { typedef Scheduling_Criteria::Priority Criterion; static const bool smp = false; static const unsigned int QUANTUM = 10000; // us };

Bellow is a short description for each of these configurable features:

`Criterion`: defines the scheduling policy to be employed by the system scheduler. These policies are available through the `Scheduling_Criteria` namespace. The available criteria are Round-Robin, First-Come First-Served (FCFS), Earliest Deadline First (EDF), Rate Monotonic (RM), and CPU Affinity. Only one criterion can exist at the same time;

`smp`: enables the system scheduler to use extra CPU cores when available;

`QUANTUM`: defines the scheduler's time quantum, i.e., the time-slice to be used by the scheduler. The scheduler will reschedule a thread in the period given by QUANTUM.

4.1.3. Process Coordination

Process coordination in EPOS is realized by the Synchronizer and Communicator families of abstractions. Process coordination using Synchronizer family is explained in this section. Process coordination through Communicator family can be performed as a side-effect of inter-process communication. For inter-process communication see section Communication of this user guide.

4.1.3.1. Synchronizers

Synchronizers are used to 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.

The Synchronizer_Common class is the common package for Synchronizer Abstractions. Synchronizer_Common is not used directly but through its subclasses, the abstractions: Semaphore,

Mutex, and Condition.

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.
The Semaphore class implements the Semaphore member of the Synchronizer family. This class is located in the include/semaphore.h file.

- **Semaphore(v : int = 1)**

Creates a Semaphore instance. By default, a semaphore is by initialized with "1" in EPOS, but it can be initialized with any other value.

- **~Semaphore()**

Destroys a Semaphore instance.

- **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.

The Mutex member of the Synchronizer family implements a simple mutual exclusion device that supplies two atomic operations: lock and unlock.

- **Mutex()**

Creates a Mutex instance.

- **~Mutex()**

Destroys a Mutex instance.

- **lock()**

Locks a mutex. Subsequent invocations cause the calling threads to wait.

- **unlock()**

Unlocks a mutex. When a thread invokes the operation unlock on a mutex and there are threads waiting on it, the first thread put to wait is allowed to continue execution, immediately locking the mutex. If no threads are waiting, the unlock operation has no effect.

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.

- **Condition()**

Creates a condition variable.

- **~Condition()**

Destroys a condition variable.

- **wait()**

Implicitly unlocks the shared data and puts the 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 in two ways: operation signal announces it to the first waiting thread, and operation broadcast announces it to all waiting 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.

- **broadcast()**

Announces the assertion of a predicate to all waiting threads.

4.1.3.2. Examples

The application implemented in src/abstraction/semaphore_test.cc demonstrates the semaphore usage in order to solve the The dining philosophers problem.

semaphore_test.cc

```
// EPOS-- Semaphore Test Program #include <utility/ostream.h> #include <thread.h> #include <semaphore.h> #include <alarm.h> #include <display.h> __USING_SYS const int iterations = 10; Semaphore sem_display; Thread * phil[5]; Semaphore * chopstick[5]; OStream cout; int philosopher(int n, int l, int c) { int first = (n < 4)? n : 0; int second = (n < 4)? n + 1 : 4; for(int i = iterations; i > 0; i--) { sem_display.p(); Display::position(l, c); cout << "thinking"; sem_display.v(); Delay thinking(100000); chopstick[first]->p(); // get first chopstick chopstick[second]->p(); // get second chopstick sem_display.p(); Display::position(l, c); cout << " eating "; sem_display.v(); Delay eating(500000); chopstick[first]->v(); // release first chopstick chopstick[second]->v(); // release second chopstick } sem_display.p(); Display::position(l, c); cout << " done "; sem_display.v(); return(iterations); } int main() { sem_display.p(); Display::clear(); cout << "The Philosopher's Dinner:\n"; for(int i = 0; i < 5; i++) chopstick[i] = new Semaphore; phil[0] = new Thread(&philosopher, 0, 5, 32); phil[1] = new Thread(&philosopher, 1, 10, 44); phil[2] = new Thread(&philosopher, 2, 16, 39); phil[3] = new Thread(&philosopher, 3, 16, 24); phil[4] = new Thread(&philosopher, 4, 10, 20); cout << "Philosophers are alife and hungry!\n"; cout << "The dinner is served ...\n"; Display::position(7, 44); cout << '/'; Display::position(13, 44); cout << '\\'; Display::position(16, 35); cout << '|'; Display::position(13, 27); cout << '/'; Display::position(7, 27); cout << '\\'; sem_display.v(); for(int i = 0; i < 5; i++) { int ret = phil[i]->join(); sem_display.p(); Display::position(20 + i, 0); cout << "Philosopher " << i << " ate " << ret << " times \n"; sem_display.v(); } for(int i = 0; i < 5; i++) delete chopstick[i]; for(int i = 0; i < 5; i++) delete phil[i]; cout << "The end!\n"; return 0; }
```

As one can see, there are 5 threads representing philosophers and 5 semaphores representing chopsticks. The p operation means a philosopher taking a chopstick and the v operation, a philosopher releasing that chopstick. The semaphores are created without any parameter for the constructor so, they are initialized with "1".
There is also a semaphore to control the display access, preventing out-of-order cout messages.

The same dining philosophers problem is solved using the Mutex and Condition abstractions in the files src/abstraction/semaphore_test.cc and abstraction/condition_test.cc. The application using Condition creates a barrier where all philosophers wait before trying to eat for the first time. The solution using Mutex is practically the same as using Semaphore.

4.1.4. Time Management

Time is managed by the families of components shown in Figure below. The **Clock** abstraction is responsible for keeping track of the current time, and is only available on systems that feature a real-time clock device, which is in turn abstracted by a member of the **RTC** family of mediators. The **Chronometer** abstraction is used to measure time intervals, through the use of a timestamp counter (**TSC**) mediator. If a given platform does not feature a hardware TSC, its functionality may be emulated by an ordinary periodic timer. The **Alarm** abstraction can be used to generate timed events, and also to put a thread to *sleep* for a certain time. For this purpose, an application instantiates a handler and registers it with an Alarm specifying a time period and the number of times the handler object is to be invoked. For more information please consult the Handler utility and Timer mediator.

4.1.4.1. Clock

The Clock component API is depicted in the Figure below. It uses a Real-Time Clock (RTC) mediator in order to get the current time and date. The types Microsecond, Second, and Date are also defined by the RTC mediator.

- **Clock()**

Construct a Clock component object. Allocates all memory needed by the object.

- **Microsecond resolution()**

This method returns the Clock resolution.

- **Second now()**

This method returns the current time in seconds.

- **Date date()**

This method returns the current date.

- **void date(Date & d)**

It sets the current date to the argument value d.

4.1.4.2. Alarm

The Alarm component API is presented in the Figure below. The Alarm uses a dedicated hardware timer when the architecture has multiple timers or shares a timer with the scheduler. A Handler utility is used to handle the event when the Alarm triggers.

- **Alarm(const Microsecond & time, Handler *handler, int times = 1)**

Constructs an Alarm that will be trigger after **time** microseconds and will be handled by the **handler**. This event will occur **times** number of times. The default number of times is 1.

- **~Alarm()**

Destructs an Alarm previously created. It deallocates all memory used by the Alarm.

- **Hertz resolution()**

Returns the Alarm resolution in Hertz.

- **void delay(const Microsecond & time)**

Delay a Thread execution by **time** microseconds.

- **int init()**

Initializes the Alarm component. This method is called during the system bootstrapping and must not be used in the application.

4.1.4.3. Chronometer

The Chronometer abstraction API is exemplified in the UML diagram below. The Chronometer uses a Real-Time Clock (RTC) mediator to count time. It also has two private attributes _start and _stop that are used in its methods.

- **Chronometer()**

Constructs a Chronometer object.

- **Hertz frequency()**

Return the Chronometer frequency. It is a call to RTC frequency method.

- **void start()**

Start counting time.

- **void stop()**

Stop counting if the Chronometer is active.

- **void reset()**

Resets the Chronometer.

- **int lap()**

If the Chronometer was already started, the lap method will read the current time to _stop attribute. After that, the difference between _start and _stop can be read by the read() method.

- **Microsecond read()**

Return the time in Microsecond since the Chronometer starts counting.

- **Time_Stamp ticks()**

Returns how many RTC ticks have passed.

4.1.4.4. Example

Example of time service utilization

```
#include <chronometer.h> #include <utility/ostream.h> #include <utility/handler.h> #include <clock.h> #include <alarm.h> static int iterations = 100; static Alarm::Microsecond time = 100000; int main() { OStream cout; Clock clock; Chronometer chron; // Read current system time cout << "Current Time: " << clock.now() << endl; // Create a handler function and associate it // to a periodic time event. Handler_Function handler(&func); Alarm alarm(time, &handler, iterations); // Start a chronometer and put this thread to sleep // Afterwards, stop and read the chronometer chron.start();
```

Alarm::delay(time * (iterations + 1)); chron.stop(); cout << "Elapsed time: " << chron.read() << endl; return 0; }

4.1.5. 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 interface to a communication **Channel** implemented over a **Network**.

4.1.5.1. Communicator

Members of the **Communicator** family are end-points for a communication channel, including interfaces to Asynchronous Remote Memory Segment (that supports asynchronous access to a memory segment in a remote node) and Active Message Handler (in which messages, besides transporting data, also carries a reference to a handler that is invoked, in the context of the receiving process, to handle the message upon arrival).

4.1.5.2. Channel

Members of the **Channel** family implement communication protocols classified at level four (transport) according to the OSI model, including members as Stream and Datagram.

- **Channel()**

Construct a Channel object. Allocates all memory needed by it.

- **Channel(Address &a)**

Construct a Channel object, and defines the address (**a**) to be used. Allocates all memory needed by it.

- **~Channel()**

Destructs a Channel previously created. It deallocates all memory used by the Channel.

- **int send(const Address &dst,const void *ptr, unsigned int size)**

Sends **size** bytes of data to **dst**.

- **int receive(Address &src, void *ptr, int size)**

Receives **size** bytes of data, **src** is set by the method accordingly.

4.1.5.3. Network

Network family members provide the physical means to build logical channels. Members of this family abstract the particularities of each network technology, so that all networks are equivalent from the standpoint of the channels.

- **Network()**

Construct a Network object with the default underlying device. Allocates all memory needed by it.

- **Network(unsigned int unit)**

Construct a Network object specifying the underlying device to be used. Allocates all memory needed by it.

- **~Network()**

Destructs a Network object previously created. It deallocates all memory used by the object.

- **int send(const Address & to, const void * data, unsigned int size)**

Sends **size** bytes of data to **to**.

- **int receive(Address * from, void * data, unsigned int size)**

Receives **size** bytes of data, **from** is set by the method accordingly.

- **MAC_Address arp(const Address & addr)**

Send an arp request.

- **Address rarp(const MAC_Address & addr)**

Send a rarp request.

- **void update(NIC_Common::Observed * o, int p)**

Used only by the ARP protocol, updates sha and spa.

- **void reset()**

Resets the underlying device.

- **const Address & address()**

Returns the underlying device address.

- **const Statistics & statistics()**

Returns the underlying device Statistics (which provides transmission and reception statistics).

4.1.5.4. 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).

4.1.5.5. Example

Bellow is an example showing how to utilize the NIC mediator, which should work for all EPOS architectures. For the example to work the application should be built twice, once as it is, and another time swapping the commented line in the main function. Then uploading the generated images to two machines should do the trick.

Example of communication service utilization

```
#include <alarm.h> #include <machine.h> #include <nic.h> #include <utility/ostream.h> __USING_SYS const unsigned char SINK_ID = 0x01; struct Msg { int id; int x; int y; }; void sender(unsigned char id) { NIC nic; unsigned char src, prot; unsigned int size; Msg msg; int i; while (true) { for (i = 5; i < 8; i++) { CPU::out8(Machine::IO::PORTB, (1 << i)); msg.id = id; msg.x = 10; msg.y = 20; nic.send(SINK_ID, 0, &msg, sizeof(msg)); Alarm::delay(100000); } } } int receiver() { NIC nic; Msg msg; OStream cout; unsigned char src, prot; int i; cout << "Sink\n"; while (true) { while (!(nic.receive(&src, &prot, &msg, sizeof(msg)) > 0)); cout << "######################\n"; cout << "# Sender id = " << msg.id << "\n"; cout << "# x = " << msg.x << "\n"; cout << "# y = " << msg.y << "\n"; } } int main() { // sender(1); receiver(); }
```

4.1.5.6. Configuring network cards

Network cards are statically configured in the Traits<> class of each platform.
If you want to have two PCNet32 cards configured on the PC platform you must edit /include/mach/pc/traits.h to have:

pc/traits.h

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

IP configuration is platform agnostic and is present in /include/traits.h, currently only a single IP is supported and no dynamic configuration like DHCP is available.

traits.h

```
template <> struct Traits<IP>: public Traits<void>{ static const unsigned int ADDRESS = 0xc0a80a01; // 192.168.10.1 static const unsigned int NETMASK = 0xffffff00; // 255.255.255.0 static const unsigned int BROADCAST = 0; // 0= Default Broadcast Address };
```

4.1.5.7. TCP/IP Networking

TCP/IP is the standard stack of protocols for communication on the Internet.
Currently we support TCP/IP over Ethernet on the PC machine and our first tests with ARM and ZigBee are showing good results even without an adaptation layer.
Instructions on how to develop applications using TCP/IP can be found here.

4.2. Utilities

4.2.1. 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)*.

4.2.1.1. Queue

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

4.2.1.2. 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.

4.2.1.3. 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.

4.2.1.4. 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.

- **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)**

- **Element * remove(Element * e)**

Removes and returns the element "e" in the scheduling queue.

- **Element * choose()**
- **Element * choose(Element * e)**

4.2.1.5. 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().

- **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)**

4.2.1.6. Example

```
#include <utility/ostream.h> #include <utility/queue.h> __USING_SYS; 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\n"; cout << "\nThis is an integer queue:\n"; Integer1 i1(1), i2(2), i3(3), i4(4); Queue<Integer1> q1; cout << "Inserting the integer " << i1.i << "\n"; q1.insert(&i1.e); cout << "Inserting the integer " << i2.i << "\n"; q1.insert(&i2.e); cout << "Inserting the integer " << i3.i << "\n"; q1.insert(&i3.e); cout << "Inserting the integer " << i4.i << "\n"; q1.insert(&i4.e); cout << "The queue has now " << q1.size() << " elements.\n"; cout << "Removing the element whose value is " << i2.i << " => " << q1.remove(&i2)->object()->i << "\n"; cout << "Removing the queue's head => " << q1.remove()->object()->i << "\n"; cout << "Removing the element whose value is " << i4.i << " => " << q1.remove(&i4)->object()->i << "\n"; cout << "Removing the queue's head =>" << q1.remove()->object()->i << "\n"; cout << "The queue has now " << q1.size() << " elements.\n"; cout << "\nThis is an ordered integer queue:\n"; 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() << ".\n"; q2.insert(&j1.e); cout << "Inserting the integer " << j2.i << " with rank " << j2.e.rank() << ".\n"; q2.insert(&j2.e); cout << "Inserting the integer " << j3.i << " with rank " << j3.e.rank() << ".\n"; q2.insert(&j3.e); cout << "Inserting the integer " << j4.i << " with rank " << j4.e.rank() << ".\n"; q2.insert(&j4.e); cout << "The queue has now " << q2.size() << " elements.\n"; cout << "Removing the element whose value is " << j2.i << " => " << q2.remove(&j2)->object()->i << "\n"; cout << "Removing the queue's head => " << q2.remove()->object()->i << "\n"; cout << "Removing the queue's head => " << q2.remove()->object()->i << "\n"; cout << "Removing the queue's head => " << q2.remove()->object()->i << "\n"; cout << "The queue has now " << q2.size() << " elements.\n"; cout << "\nThis is an integer queue with relative ordering:\n"; 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() << ".\n"; q3.insert(&k1.e); cout << "Inserting the integer " << k2.i << " with relative order " << k2.e.rank() << ".\n"; q3.insert(&k2.e); cout << "Inserting the integer " << k3.i << " with relative order " << k3.e.rank() << ".\n"; q3.insert(&k3.e); cout << "Inserting the integer " << k4.i << " with relative order " << k4.e.rank() << ".\n"; q3.insert(&k4.e); cout << "The queue has now " << q3.size() << " elements.\n"; cout << "Removing the element whose value is " << j2.i << " => " << q3.remove(&k2)->object()->i << "\n"; cout << "Removing the queue's head => " <<
```

```
q3.remove()->object()->i << "\n"; cout << "Removing the queue's head => " <<
q3.remove()->object()->i << "\n"; cout << "Removing the queue's head => " <<
q3.remove()->object()->i << "\n"; cout << "The queue has now " << q3.size() << " elements.\n";
return 0; }
```

4.2.2. 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*.

4.2.2.1. Simple_List

Singly-Linked List.

- **Simple_List()**

Creates and initializes a simple list.

- **bool empty() const**

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.

4.2.2.2. Simple_Ordered_List

Singly-Linked, Ordered List.

- **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.

4.2.2.3. Simple_Relative_List

Singly-Linked, Relative Ordered List.

4.2.2.4. Simple_Grouping_List

Singly-Linked, Grouping List.

4.2.2.5. List

Doubly-Linked List.

- **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.

4.2.2.6. Ordered_List

Doubly-Linked, Ordered List.

- **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.

4.2.2.7. Relative_List

Doubly-Linked, Relative Ordered List.

4.2.2.8. Scheduling_List

Doubly-Linked, Scheduling List.

4.2.2.9. Grouping_List

Doubly-Linked, Grouping List.

4.2.2.10. Example

□□□□□□□

```
#include <utility/ostream.h> #include <utility/malloc.h> #include <utility/list.h> __USING_SYS; const int N = 10; void test_simple_list(); void test_list(); void test_ordered_list(); void test_relative_list(); void test_scheduling_list(); void test_grouping_list(); void test_simple_grouping_list(); OStream cout; int main() { cout << "List Utility Test\n"; test_simple_list(); test_simple_grouping_list(); test_list(); test_ordered_list(); test_relative_list(); test_scheduling_list(); test_grouping_list(); cout << "\nDone!\n"; return 0; } void test_simple_list () { cout << "\nThis is a singly-linked list of integers:\n"; 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 << "\n"; cout << "The list has now " << l.size() << " elements\n"; 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 << "\n"; cout << "Removing the element whose value is " << o[N/2] << " => " << *l.remove(&o[N/2])->object() << "\n"; cout << "Removing the list's head => " << *l.remove_head()->object() << "\n"; cout << "Removing the element whose value is " << o[N/4] << " => " << *l.remove(&o[N/4])->object() << "\n"; cout << "Removing the list's tail => " << *l.remove_tail()->object() << "\n"; cout << "Trying to remove an element that is not on the list => " << l.remove(&o[N+1]) << "\n"; cout << "Removing all remaining elements => "; while(l.size() > 0) { cout << *l.remove()->object(); if(l.size() > 0) cout << ", "; } cout << "\n"; cout << "The list has now " << l.size() << " elements\n"; for(int i = 0; i < N; i++) delete e[i]; } void test_simple_grouping_list() { cout << "\nThis is a simple grouping list of integers:\n"; 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 << "\n"; cout << "The list has now " << l.size() << " elements that group " << l.grouped_size() << " elements in total\n"; 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 << "\n"; cout << "Allocating 1 element from the list => "; d1 = l.search_decrementing(1); if(d1) { cout << *(d1->object() + d1->size()) << "\n"; if(!d1->size()) { cout << "[rm]"; // removed delete d1; } } else cout << "failed!\n"; 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 << "\n"; } else cout << "failed!\n"; 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 << "\n"; } else cout << "failed!\n"; 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 << "\n"; } else cout << "failed!\n"; cout << "The list has now " << l.size() << " elements that group " << l.grouped_size() << " elements in total\n"; } void test_list () { cout << "\nThis is a doubly-linked list of integers:\n"; 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 << "\n"; cout << "The list has now " << l.size() << " elements\n"; 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 << "\n"; cout << "Removing the element whose value is " << o[N/2] << " => " << *l.remove(&o[N/2])->object() << "\n"; cout << "Removing the list's head => " << *l.remove_head()->object() << "\n"; cout << "Removing the element whose value is " << o[N/4] << " => " << *l.remove(&o[N/4])->object() << "\n"; cout << "Removing the list's tail => " << *l.remove_tail()->object() << "\n"; cout << "Trying to remove an element that is not on the list => " << l.remove(&o[N+1]) << "\n"; cout << "Removing all remaining
```

```
elements => "; while(l.size() > 0) { cout << *l.remove()->object(); if(l.size() > 0) cout << ", "; } cout << "\n"; cout << "The list has now " << l.size() << " elements\n"; for(int i = 0; i < N; i++) delete e[i]; } void test_ordered_list () { cout << "\nThis is an ordered, linked list of integers:\n"; 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 << "\n"; cout << "The list has now " << l.size() << " elements\n"; 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 << "\n"; cout << "Removing the element whose value is " << o[N/2] << " => " << *l.remove(&o[N/2])->object() << "\n"; cout << "Removing the list's head => " << *l.remove_head()->object() << "\n"; cout << "Removing the element whose value is " << o[N/4] << " => " << *l.remove(&o[N/4])->object() << "\n"; cout << "Removing the list's tail => " << *l.remove_tail()->object() << "\n"; cout << "Trying to remove an element that is not on the list => " << l.remove(&o[N+1]) << "\n"; cout << "Removing all remaining elements => "; while(l.size() > 0) { cout << *l.remove()->object(); if(l.size() > 0) cout << ", "; } cout << "\n"; cout << "The list has now " << l.size() << " elements\n"; for(int i = 0; i < N; i++) delete e[i]; } void test_relative_list () { cout << "\nThis is a realtive ordered, linked list of integers:\n"; 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 << "\n"; cout << "The list has now " << l.size() << " elements\n"; 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 << "\n"; cout << "Removing the element whose value is " << o[N/2] << " => " << *l.remove(&o[N/2])->object() << "\n"; cout << "Removing the list's head => " << *l.remove_head()->object() << "\n"; cout << "Removing the element whose value is " << o[N/4] << " => " << *l.remove(&o[N/4])->object() << "\n"; cout << "Removing the list's tail => " << *l.remove_tail()->object() << "\n"; cout << "Trying to remove an element that is not on the list => " << l.remove(&o[N+1]) << "\n"; cout << "Removing all remaining elements => "; while(l.size() > 0) { cout << *l.remove()->object(); if(l.size() > 0) cout << ", "; } cout << "\n"; cout << "The list has now " << l.size() << " elements\n"; for(int i = 0; i < N; i++) delete e[i]; } void test_scheduling_list () { cout << "\nThis is scheduling list of integers:\n"; Scheduling_List<int> l; int o[N]; Scheduling_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 Scheduling_List<int>::Element(&o[i], N - i - 1); l.insert(e[i]); cout << i << "(" << N - i - 1 << ")"; if(i != N - 1) cout << ", "; } cout << "\n"; cout << "The list has now " << l.size() << " elements\n"; cout << "They are: "; for(Scheduling_List<int>::Iterator i = l.begin(); i != l.end(); i++) { cout << *i->object(); if(Scheduling_List<int>::Iterator(i->next()) != l.end()) cout << ", "; } cout << "\n"; cout << "Scheduling the list => " << *l.choose()->object() << "\n"; cout << "They are: "; for(Scheduling_List<int>::Iterator i = l.begin(); i != l.end(); i++) { cout << *i->object(); if(Scheduling_List<int>::Iterator(i->next()) != l.end()) cout << ", "; } cout << "\n"; cout << "Forcing scheduling of antorher element => " << *l.choose_another()->object() << "\n"; cout << "They are: "; for(Scheduling_List<int>::Iterator i = l.begin(); i != l.end(); i++) { cout << *i->object(); if(Scheduling_List<int>::Iterator(i->next()) != l.end()) cout << ", "; } cout << "\n"; cout << "Forcing scheduling of element whose value is " << o[N/2] << " => " << *l.choose(e[N/2])->object() << "\n"; cout << "They are: "; for(Scheduling_List<int>::Iterator i = l.begin(); i != l.end(); i++) { cout << *i->object(); if(Scheduling_List<int>::Iterator(i->next()) != l.end()) cout << ", "; } cout << "\n"; cout << "Removing the list's head => " << *l.remove(l.choose())->object() << "\n"; cout << "Removing the element whose value is " << o[N/4] << " => " << *l.remove(e[N/4])->object() << "\n"; cout << "Removing all remaining elements => "; while(l.size() > 0) { cout << *l.remove(l.choose())->object(); if(l.size() > 0) cout << ", "; } cout << "\n"; cout << "The list has now " << l.size() << " elements\n"; for(int i = 0; i < N; i++) delete e[i]; } void test_grouping_list() { cout << "\nThis is a grouping list of integers:\n"; Grouping_List<int> l; int o[N * 2]; Grouping_List<int>::Element * e[N * 2]; 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 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 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 << "\n"; cout << "The list has now " << l.size() << " elements that group " << l.grouped_size() << " elements in total\n"; cout << "They are: "; for(Grouping_List<int>::Iterator i = l.begin(); i != l.end(); i++) { cout << *i->object(); if(Grouping_List<int>::Iterator(i->next()) != l.end()) cout << ", "; } cout << "\n"; cout << "Allocating 1 element from the list => "; d1 = l.search_decrementing(1); if(d1) { cout << *(d1->object() + d1->size()) << "\n"; if(!d1->size()) { cout << "[rm]"; // removed delete d1; } } else cout << "failed!\n"; 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 << "\n"; } else cout << "failed!\n"; 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 << "\n"; } else cout << "failed!\n"; 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 << "\n"; } else cout << "failed!\n"; cout << "The list has now " << l.size() << " elements that group " << l.grouped_size() << " elements in total\n"; }
```

4.2.3. Hash

The EPOS has 2 types of Hash. They are:

- *Simple_Hash*;
- *Hash*;

4.2.3.1. 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.

- **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.

4.2.3.2. Hash

Hash Table with a Synonym List for each Key.

- **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.

4.2.3.3. Example

```
#include <utility/ostream.h> #include <utility/malloc.h> #include <utility/hash.h> __USING_SYS; const int N = 10; void test_few_synonyms_hash(); void test_many_synonyms_hash(); OStream cout; int
```

```
main() { cout << "Hash Utility Test\n"; test_few_synonyms_hash(); test_many_synonyms_hash(); cout << "\nDone!\n"; return 0; } void test_few_synonyms_hash() { cout << "\nThis is a hash table of integeres with few synonyms:\n"; 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 << "\n"; cout << "The hash table has now " << h.size() << " elements:\n"; 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 << "\n"; cout << "Removing the element whose value is " << o[N/2] << " => " << *h.remove(&o[N/2])->object() << "\n"; cout << "Removing the element whose key is " << 1 << " => " << *h.remove_key(1)->object() << "\n"; cout << "Removing the element whose key is " << 11 << " => " << *h.remove_key(11)->object() << "\n"; cout << "Removing the element whose value is " << o[N/4] << " => " << *h.remove(&o[N/4])->object() << "\n"; cout << "Removing the element whose key is " << N-1 << " => " << *h.remove_key(N-1)->object() << "\n"; cout << "Trying to remove an element that is not on the hash => " << h.remove(&o[N/2]) << "\n"; cout << "The hash table has now " << h.size() << " elements:\n"; 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 << "\n"; 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 << "\n"; 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:\n"; 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:\n"; 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 << "\n"; cout << "Removing the element whose value is " << o[N/2] << " => " << *h.remove(&o[N/2])->object() << "\n"; cout << "Removing the element whose key is " << 1 << " => " << *h.remove_key(1)->object() << "\n"; cout << "Removing the element whose key is " << 11 << " => " << *h.remove_key(11)->object() << "\n"; cout << "Removing the element whose value is " << o[N/4] << " => " << *h.remove(&o[N/4])->object() << "\n"; cout << "Removing the element whose key is " << N-1 << " => " << *h.remove_key(N-1)->object() << "\n"; cout << "Trying to remove an element that is not on the hash => " << h.remove(&o[N/2]) << "\n"; cout << "The hash table has now " << h.size() << " elements:\n"; 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 << "\n"; 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 << "\n"; for(int i = 0; i < N * N; i++) delete e[i]; }
```

4.2.4. Vector

The Vector data structure API is presented in the Figure below

- **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.

4.2.4.1. Example

```
#include <utility/ostream.h> #include <utility/malloc.h> #include <utility/vector.h> __USING_SYS; const int N = 10; OStream cout; int main() { cout << "Vector Utility Test\n"; cout << "\nThis is a vector of integers:\n"; 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 << "\n"; cout << "The vector has now " << v.size() << " elements:\n"; for(int i = 0; i < N; i++) { cout << "[" << i << "]=" << *v.get(i)->object(); if(i != N - 1) cout << ", "; } cout << "\n"; for(int i = 0; i < N; i++) (*v.get(i)->object())++; cout << "The vector's elements were incremented and are now:\n"; for(int i = 0; i < N; i++) { cout << "[" << i << "]=" << *v.get(i)->object(); if(i != N - 1) cout << ", "; } cout << "\n"; cout << "Removing the element whose value is " << o[N/2] << " => " << *v.remove(&o[N/2])->object() << "\n"; cout << "Removing the second element => " << *v.remove(1)->object() << "\n"; cout << "Removing the element whose value is " << o[N/4] << " => " << *v.remove(&o[N/4])->object() << "\n"; cout << "Removing the last element => " << *v.remove(N - 1)->object() << "\n"; cout << "Trying to remove an element that is not on the vector => " << v.remove(&o[N/2]) << "\n"; cout << "Removing all remaining elements => "; for(int i = 0; i < N; i++) { cout << *v.remove(i)->object(); if(i != N - 1) cout << ", "; } cout << "\n"; cout << "The vector has now " << v.size() << " elements\n"; for(int i = 0; i < N; i++) delete e[i]; cout << "\nDone!\n"; return 0; }
```

4.2.5. Handler

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

4.2.5.1. Handler

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

- **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.

4.2.5.2. 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.

- **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.

4.2.5.3. 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.

- **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.

4.2.5.4. 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.

- **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.

4.2.6. 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 in the UML diagrams below.

- **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()**

Constructs an Observer object.

- **~Observer()**

Destructs an Observer object.

- **void update(Observed * o)**

Updates the Observer state regarding a given Observed object.

- **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()**

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.

4.2.6.1. 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; }
```

4.2.7. 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 in the UML diagram below.

- **unsigned short crc16(char * ptr, int size)**

Calculates a short, fixed-length CRC code for a given block of data.

4.2.7.1. 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; }
```

4.2.8. 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 in the UML diagram below.

- **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.

4.2.8.1. Example

$EPOS/src/utility/ostream_test.cc

```
// EPOS-- OStream Utility Test Program #include <utility/ostream.h> __USING_SYS; int main() { OStream cout; cout << "OStream test\n"; cout << "This is a char:\t\t\t" << 'A' << "\n"; cout << "This is a negative char:\t" << '\377' << "\n"; cout << "This is an unsigned char:\t" << 'A' << "\n"; cout << "This is an int:\t\t\t" << (1 << sizeof(int) * 8 - 1) - 1 << "\n" << "\t\t\t\t" << hex << (1 << sizeof(int) * 8 - 1) - 1 << "(hex)\n" << "\t\t\t\t" << dec << (1 << sizeof(int) * 8 - 1) - 1 << "(dec)\n" << "\t\t\t\t" << oct << (1 << sizeof(int) * 8 - 1) - 1 << "(oct)\n" << "\t\t\t\t" << bin << (1 << sizeof(int) * 8 - 1) - 1 << "(bin) " << endl; cout << "This is a negative int:\t\t" << (1 << sizeof(int) * 8 - 1) << "\n" << "\t\t\t\t" << hex << (1 << sizeof(int) * 8 - 1) << "(hex)\n" << "\t\t\t\t" << dec << (1 << sizeof(int) * 8 - 1) << "(dec)\n" << "\t\t\t\t" << oct << (1 << sizeof(int) * 8 - 1) << "(oct)\n" << "\t\t\t\t" << bin << (1 << sizeof(int) * 8 - 1) << "(bin) " << endl; cout << "This is a string:\t\t" << "string" << "\n"; cout << "This is a pointer:\t\t" << &cout << "\n"; return 0; }
```

4.2.9. 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 in the UML diagram below.

- **Spin()**

Constructs an Spin object.

- **void acquire()**

Acquires the spin lock, entering a loop.

- **void release()**

Releases the spin lock.

4.2.9.1. 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<> class Heap_Wrapper<true>: public Heap_Common { public: Heap_Wrapper() {} Heap_Wrapper(void * addr, unsigned int bytes): Heap_Common(addr, bytes) { free(addr, bytes); } void * alloc(unsigned int bytes) { _lock.acquire(); void * tmp = Heap_Common::alloc(bytes); _lock.release(); return tmp; } void * calloc(unsigned int bytes) { _lock.acquire(); void * tmp = Heap_Common::calloc(bytes); _lock.release(); return tmp; } void free(void * ptr) { _lock.acquire(); Heap_Common::free(ptr); _lock.release(); } void free(void * ptr, unsigned int bytes) { _lock.acquire(); Heap_Common::free(ptr, bytes); _lock.release(); } private: Spin _lock; };
```

4.2.10. Random

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

- **Pseudo_Random()**

Constructs an Pseudo_Random object.

- **unsigned long int random(unsigned long int)**

Implements the Linear Congruential Generator.

4.2.10.1. 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; }
```

4.3. Hardware Mediators

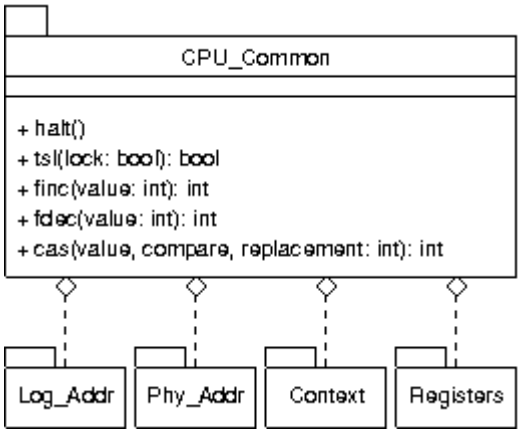
4.3.1. CPU

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.



- **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.

4.3.2. 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.

4.3.3. 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.

The mediator also defines two types:

typedef unsigned long Hertz; typedef unsigned long long Time_Stamp;

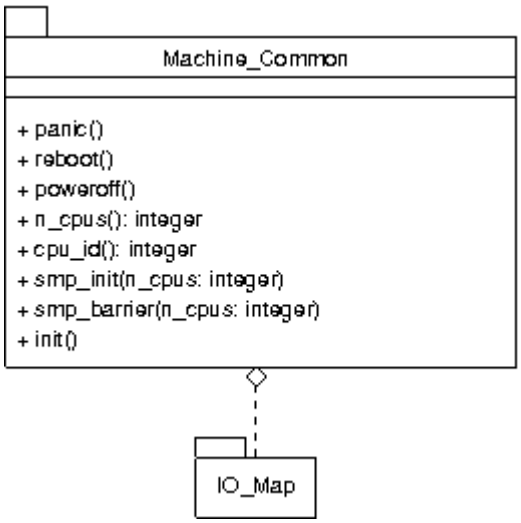
4.3.4. Machine

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.

Bellow is a class diagram for this interface.



- **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.

4.3.5. 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();

- **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.

4.3.6. RTC

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

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.

- **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.

4.3.7. 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:

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.

- **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.

AVR Timer API

The AVR architecture usually has more than one Timer. For example, in the ATMega128 microcontroller, there are three Timers implemented in EPOS, Timer_1, Timer_2, and Timer_3. Timer_1 is dedicated to Scheduler, and it is named Scheduler_Timer. Timer_3 is dedicated to Alarm, and it is named Alarm_Timer.

- **Timer()**

Creates a Timer object.

- **Timer(const Hertz & f)**

Creates a Timer object and sets its frequency to **f**.

- **Timer(const Microsecond & quantum, const Handler * handler)**

Creates a Timer object, sets its frequency to **1000000 / quantum**, and associates its interrupt handler to **handler**.

- **Timer(const Handler * handler)**

Creates a Timer object and associates its interrupt handler to **handler**.

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.

4.3.8. ADC

An ADC (Analog-to-Digital Converter) is a hardware device responsible for converting an analog signal to a digital signal (discrete number). Usually, it converts an input analog voltage to a digital representation of this input, proportional to the signal amplitude. The EPOS base ADC class defines a list of possible channels, a reference for the ADC clock, and the operation mode (a single conversion or free running mode), as shown below.

```
enum Channel {
SINGLE_ENDED_ADC0 = 0,
SINGLE_ENDED_ADC1 = 1,
SINGLE_ENDED_ADC2 = 2,
SINGLE_ENDED_ADC3 = 3,
SINGLE_ENDED_ADC4 = 4,
SINGLE_ENDED_ADC5 = 5,
SINGLE_ENDED_ADC6 = 6,
SINGLE_ENDED_ADC7 = 7
};

enum Reference {
SYSTEM_REF = 0,
EXTERNAL_REF = 1,
INTERNAL_REF = 3
};

enum Trigger {
SINGLE_CONVERSION_MODE = 0,
FREE_RUNNING_MODE = 1,

};
```

The ADC API is depicted below:

- **ADC()**

Constructs an ADC object. By default, the constructor uses the SINGLE_ENDED_ADC0 channel, SYSTEM_REF as reference, the SINGLE_CONVERSION_MODE as running mode, and the machine CLOCK >> 7 as the ADC frequency,

- **ADC(unsigned char channel, Hertz frequency)**

Constructs an ADC object using the *channel* and *frequency* received as arguments. By default, the ADC reference is SYSTEM_REF and the running mode is SINGLE_CONVERSION_MODE.

- **ADC(unsigned char channel, unsigned char reference, unsigned char trigger, Hertz frequency)**

Constructs an ADC object using the *channel*, *frequency*, *trigger* mode, and *frequency* received as arguments.

- **void config(unsigned char channel, unsigned char reference, unsigned char trigger, Hertz frequency)**

Configure the ADC *channel* with a *reference*, a *trigger* mode, and a *frequency*.

- **void config(unsigned char * channel, unsigned char * reference, unsigned char * trigger, Hertz * frequency)**

Configure the ADC **channel* with a **reference*, a **trigger* mode, and a **frequency*.

- **int sample()**

Reads the conversion carried out by an ADC channel as configured.

- **int get()**

Returns the last read data.

- **bool finished()**

Returns true if an ADC conversion has finished, false otherwise.

- **bool enable()**

Enables the ADC device.

- **void disable()**

Disables the ADC device.

- **void reset()**

Resets the ADC device.

4.3.9. Sensor

EPOS supports the following sensors:

- **MTS300 Sensor Board**: it has temperature and photo sensors. Available in the ATMega128 and ATMega1281 machines.
- **SHT11 Sensor Chip**: it has temperature and humidity sensors. Available in the ATMega1281 machine.
- **ADXL202 Accelerometer Sensor**: available in the ATMega128 and ATMega1281 machines.

Temperature, Photo, and Humidity sensors API:

- **Temperature_Sensor()**

Constructs a Temperature sensor object.

- **Photo_Sensor()**

Constructs a Photo sensor object.

- **Humidity_Sensor()**

Constructs a Humidity sensor object.

- **bool enable()**

Enables the sensor to start sensing.

- **void disable()**

Disables the sensor to stop sensing.

- **int sample()**

Reads a Temperature, Photo, or Humidity value.

- **int get()**

Gets the last read value.

- **bool data_ready()**

Returns true if the data is read to be read, false otherwise.

Accelerometer Sensor API:

- **Accelerometer()**

Constructs an Accelerometer sensor object.

- **int sample_x()**

Reads the x value of the accelerometer.

- **int sample_y()**

Reads the y value of the accelerometer.

- **int get_x()**

Gets the last x read value.

- **int get_y()**

Gets the last y read value.

- **void disable_x()**

Disables the x axis.

- **void disable_y()**

Disables the y axis.

- **bool enable_x()**

Enables the x axis.

- **bool enable_y()**

Enables the y axis.

- **bool data_ready_x()**

Returns true if the x data is ready to be read.

- **bool data_ready_y()**

Returns true if the y data is ready to be read.

4.3.9.1. Example

```
/** Light and Temperature sensor demo */ #include <display.h> #include <machine.h> #include <nic.h> __USING_SYS struct Message { unsigned int src; unsigned int dst; unsigned int accel_x; unsigned int accel_y; unsigned int temperature; unsigned int light; }; static unsigned int FIRST_ID = 0x01; static unsigned int LAST_ID = 0x02; static unsigned int MY_ID = 0x01; static unsigned int SINK_ID = 0x33; void sensor() { CPU::out8(Machine::IO::DDRA, 0x07); CPU::out8(Machine::IO::PORTA, ~0); unsigned char count; Accelerometer accel; Temperature_Sensor temperature; Photo_Sensor photo; NIC nic; unsigned char src, prot; unsigned int size; Message msg; CPU::out8(Machine::IO::PORTA, 0x5); while(1) { if(nic.receive(&src, &prot, &msg, sizeof(msg)) && (msg.dst == MY_ID)) { CPU::out8(Machine::IO::PORTA, ~++count); msg.src = MY_ID; msg.dst = SINK_ID; msg.accel_x = accel.sample_x(); msg.accel_y = accel.sample_y(); msg.temperature = temperature.sample(); msg.light = photo.sample(); nic.send(0, 0, &msg, sizeof(msg)); } memset(&msg,sizeof(msg),0); } } int main() { sensor(); }
```

4.3.10. 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.

- **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.

4.3.10.1. Example

```
#include <utility/ostream.h> #include <uart.h> __USING_SYS int main() { OStream cout; cout << "UART test\n\n"; 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 << ")!\n"; } cout << " passed!\n"; 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 << ")!\n"; } cout << " passed!\n"; return 0; }
```

4.3.11. 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.

4.3.12. SPI

Serial Peripheral Interface (SPI) is a synchronous serial data link that operates in full duplex. Devices communicates using a master/slave relationship, in which the master initiates the data frame. When the master generates a clock and selects a slave device, data may be transferred in both directions simultaneously.

SPI specifies four signals: Serial Clock (SCLK); Master Output, Slave Input (MOSI); Master Input, Slave Output (MISO); and Slave Select (SS). SCLK is generated by the master and input to all slaves. MOSI carries data from master to slave. MISO carries data from slave back to master. A slave device is selected when the master asserts its SS signal. If multiple slave devices exist, the master generates a separate slave select signal for each slave.

- **void configure()**

Enable SPI.

- **bool complete()**

Returns true if the end of transmission flag is set.

- **void int_enable()**

Causes the SPI interrupt to be executed.

- **void int_disable()**

Deactivates the SPI interrupt.

- **char get()**

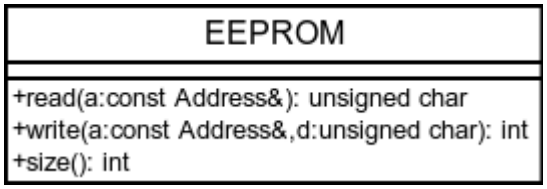
Get a byte.

- **void put(char c)**

Sends a byte.

4.3.13. 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. The class digram bellow shows the public interface for the EEPROM mediator.



- **unsigned char read(const Address & a)**

Reads and returns the byte stored at address *a*

- **void write(const Address & a, unsigned char d)**

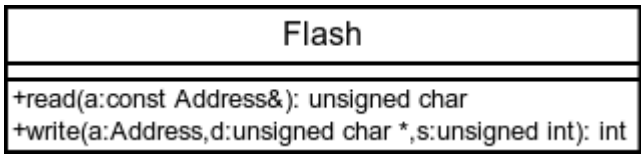
Reprograms the EEPROM. Writes byte *d* at address *a*

- **int size()**

Returns the EEPROM size

4.3.14. Flash

Flash memories are non-volatile storage devices which improves EEPROMs in terms of area, latency and life-time. The class digram bellow shows the public interface for the Flash mediator.



- **unsigned char read(const Address & a)**

Reads and returns the byte stored at address *a*

- **int write(Address a, unsigned char * d, unsigned int s)**

Writes up to *s* bytes pointed by *d* at address *a*