EPOS 2 User Guide

Software/Hardware Integration Lab at UFSC

EPOS 2.2 User Guide

Table of contents

1. Introduction

This document is a reference guide to the EPOS API. It is designed focusing on application development with EPOS 2.2 (for other guides visit EPOS Documentation).

1.1. EPOS Overview

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

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

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

1.2. OpenEPOS License

OpenEPOS 2.2 is licensed under the The GNU General Public Licence 2.0. In this site, **EPOS** and **OpenEPOS** are used interchangeably to designate the specific set of components publicly released in this site under the GPL license. Other components, not listed in this documentation and not released through this site, are usually subject to more restrictive licenses. For additional information, please contact epos@lisha.ufsc.br.

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

1.3. Main Features

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

Feature	Release					
	1.0	1.1	1.2	2.0	2.1	2.2

Architectures	AVR8	√	√	√	–	–	–
	ARMv3 (ARM7)	–	√	√	–	–	–
	ARMv7-M	–	–	–	√	√	√
	x86 (IA-32)	√	√	√	√	√	√
	x86_64	–	–	√	–	–	–
	PowerPC	≈	√	√	–	–	–
	MIPS	≈	√	√	–	–	–
Machines	EPOSMote I (AVR8)	√	–	–	–	–	–
	EPOSMote II (ARM7TDI)	–	√	√	–	–	–
	EPOSMote III (ARM Cortex-M3)	–	–	–	√	√	√
	PC	√	√	√	√	√	√
	Atmega16 (AVR8)	√	√	√	–	–	–
	Atmega128 (AVR8)	√	√	√	–	–	–
	Atmega1281 (AVR8)	√	√	√	–	–	–
	At90can128 (AVR8)	√	√	√	–	–	–
	ML310 (PPC32)	≈	√	√	–	–	–
	LEON3	≈	√	√	–	–	–
	Plasma (MIPS)	≈	√	√	–	–	–
	LM3S9B96 (ARM Cortex-M3@QEMU)	–	–	–	√	√	√
	Realview PBX (ARM Cortex-A9@QEMU)	–	–	–	–	√	√
	Raspberry PI3 (ARM Cortex-A53)	–	–	–	–	–	√
	Xilinx Zynq-7000 (Cortex-A9MP)	–	–	–	√	√	√
Devices	UART	√	√	√	√	√	√
	USART	√	√	√	–	–	√
	Ethernet	√	√	√	√	√	√
	Radio (IEEE 802.15.4)	√	√	√	√	√	√
	EEPROM	√	√	√	√	√	√
	Flash	√	√	√	√	√	√
	Timer	√	√	√	√	√	√
	SPI	√	√	√	√	√	√
	PMU	–	–	–	–	√	√
Process	Multithreading	√	√	√	√	√	√
	Real-time Scheduling	√	√	√	√	√	√
	Multicore (SMP)	√	√	√	√	√	√
	Synchronization	√	√	√	√	√	√
	Multitasking	–	–	–	√	√	√

	Non-Intrusive Monitoring	–	–	–	–	–	√
Memory	Dynamic Memory Allocation	√	√	√	√	√	√
	Scratch-pad Memory	–	≈	√	√	√	√
	Flash	√	√	√	√	√	√
Timing	Timed Events	√	√	√	√	√	√
	Chronometer	√	√	√	√	√	√
	Real-time Clock	√	√	√	√	√	√
	Watch-dog Timer	–	–	√	√	√	√
Communication	C-MAC	≈	√	√	–	–	–
	TSTP	–	–	–	√	√	√
	IEEE 802.15.4	–	–	–	√	√	√
	ELP	≈	√	√	√	√	√
	ADHOP	≈	√	√	–	–	–
	TCP/IP	≈	√	√	√	√	√
	SIP	–	≈	√	–	–	–
	RTP	–	≈	√	–	–	–
	PTP	–	≈	√	–	√	√
	HeCoPS	–	√	–	√	√	√
Power	Power Management API	√	√	√	≈	≈	≈
	Energy-aware Scheduling	√	√	√	≈	√	√
	Energy-aware, Real-time Scheduling	√	√	√	≈	√	√
	DVFS	√	–	–	–	√	√
SmartData	Sensing	–	–	–	–	√	√
	Actuating	–	–	–	–	√	√
	Clerk	–	–	–	–	–	√
	Machine Learning	–	–	–	–	–	√
Development Tools	GCC 4.0.x	√	√	√	–	–	–
	GCC 4.4.x	√	√	√	√	√	√
	GCC 7.2.x	–	–	–	–	√	√
	GCC 8.3.1	–	–	–	–	–	√
	QEMU	√	√	√	√	√	√
	GDB on QEMU	–	√	√	√	√	√

2. Setting up EPOS

2.1. Downloading EPOS

You can download OpenEPOS releases from the download page and development versions from LISHA's GitLab.

2.2. Downloading the toolchain

2.2.1. GCC

Recent versions of EPOS can go with any (recent) GCC version. However, since EPOS is itself the operating system, the compiler cannot rely on a libc compiled for another OS (such as LINUX). A cross-compiler is needed even if your source and target machines are x86-based PCs. You can use your distro's cross-compilers (version 2.2 onwards), download a precompiled GCC for EPOS from the downloads page (version 2.1 or older), or compile a newlib-based toolchain yourself following these instructions. In case you want to compile EPOS for RISC-V (version 2.1 onwards) and your OS does not have a native cross-compiler package, download a precompiled GCC for EPOS on the downloads page (available for Fedora 32 onwards and Ubuntu 18.04 onwards) or compile a 32 bits linux-based toolchain following the instructions on the official toolchain repository.

Distribution	**Target Architecture**	**Packages**
Fedora	x86/x86_64	`binutils-x86_64-linux-gnu` `gcc-c++-x86_64-linux-gnu`
Fedora	32 bits ARM	`arm-none-eabi-binutils-cs` `arm-none-eabi-gcc-cs-c++` `arm-none-eabi-newlib`
Fedora	32 bits RISC-V	`autoconf` `automake` `python3` `libmpc-devel` `mpfr-devel` `gmp-devel` `gawk` `bison` `flex` `texinfo` `patchutils` `gcc` `gcc-c++` `zlib-devel` `expat-devel`
Ubuntu 18.04 onwards	x86/x86_64	`binutils-x86-64-linux-gnu` `bin86`
Ubuntu 18.04 onwards	32 bits ARM	`binutils-arm-none-eabi` `gcc-arm-none-eabi`
Ubuntu 18.04 onwards	32 bits RISC-V	`pkg-config` `libglib2.0-dev` `libpixman-1-dev` `autoconf` `automake` `autotools-dev` `curl` `python3` `libmpc-dev` `libmpfr-dev` `libgmp-dev` `gawk` `build-essential` `bison` `flex` `texinfo` `gperf` `libtool` `patchutils` `bc` `zlib1g-dev` `libexpat-dev`

2.2.2. as86/ld86

If you don't have the "as86" command installed, you need to install the `bin86` (Ubuntu) or `dev86` (Fedora) package. It is used to compile the PC's bootstrap code (which must be Intel 8086).

2.2.3. 32-bit libs

If your host is a 64-bit operating system, you will need to install a set of 32-bit libraries. The table below shows the packages.

Distribution	**Packages**
Ubuntu all versions	`ia32-libs lib32stdc++6 libc6-i386 libc6-dev-i386`
Ubuntu 16.04 onwards	Ubuntu all versions packages substituting `ia32-libs` for `lib32z1 lib32ncurses5 libbz2-1.0:i386`
Ubuntu 17.04	Ubuntu all versions packages and `gcc-multilib g++-multilib`
Ubuntu 18.04	Ubuntu all versions packages and `gcc-multilib g++-multilib`

Fedora	`glibc-devel.i686 libstdc++.i686 libstdc++-devel zlib.i686`

2.3. Installing

Simply open a release tarball or clone a branch from the GitLab at the place you want EPOS to be installed. You don't need to bother about the chosen path nor set any environment variable. EPOS is fully self-contained.

If you also downloaded a toolchain tarball, open it at /usr/local/<architecture> whenever possible. If you do not have access to that path, you'll have to adjust the makedefs file in EPOS' main directory accordingly.

For instance, if you downloaded the **ia32 toolchain**, you should extract it at **/usr/local/ia32/gcc-7.2.0**. If you downloaded the **arm toolchain** for EPOSMote III, you should extract it at **/usr/local/arm/gcc-7.2.0**

3. Running EPOS

3.1. Compiling

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

```
$ make
```

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

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

```
$ make APPLICATION=hello
```

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

```
EPOS bootable image tool EPOS mode: library Machine: pc Model: legacy_pc Processor: ia32 (32 bits, little-endian) Memory: 262144 KBytes Boot Length: 512 - 512 (min - max) KBytes UUID: a5a205927f92887e Creating EPOS bootable image in "hello.img": Adding bootstrap "/home/guto/epos/merge/img/boot_legacy_pc": done. Adding setup "/home/guto/epos/merge/img/setup_legacy_pc": done. Adding application "hello": done. Adding system info: done. Adding specific boot features of "legacy_pc": done. Image successfully generated (69784 bytes)!
```

3.2. Running

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

```
```

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

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

3.2.1. Running on Bare Metal

In principle, there is nothing to be done to run EPOS on a real machine (i.e. without QEMU). Note, however, that there are many flavors of x86 and ARM CPUs and although EPOS tries not to make use of non-standard CPU features, it may happen that your real hardware has peculiarities that are not handled by EPOS. Furthermore, there are lots of buggy devices out there and commercial operating systems are full of workarounds to avoid igniting (often unrecoverable) problems. This is not true for EPOS!

3.2.2. Running on Virtualized Host

You can run EPOS on a `qemu-kvm` to get access to platform features not emulated by QEMU. Intel x86 PMU, for instance, is only available with KVM. However, many other aspects of QEMU differ in this mode. Check KVM FAQ for details.

3.3. Configuring

Trait classes are EPOS main configuration mechanism. Whenever an application-specific instance of EPOS is produced (that is, whenever EPOS is built), the builder looks for a file named `$APPLICATION/$APPLICATION_traits.h` in the `app` directory. For instance, if the application's main file is `app/producer_consumer/producer_consumer.cc`, then the builder will look for a file named `app/producer_consumer/producer_consumer_traits.h` to configure EPOS accordingly.

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

```
#ifndef __traits_h #define __traits_h #include <system/config.h> __BEGIN_SYS template<> struct Traits<Build>: public Traits<void> { static const unsigned int MODE = LIBRARY; static const unsigned int ARCHITECTURE = IA32; static const unsigned int MACHINE = PC; static const unsigned int MODEL = Legacy_PC; static const unsigned int CPUS = 1; static const unsigned int NODES = 1; // (> 1 => NETWORKING) static const unsigned int EXPECTED_SIMULATION_TIME = 60; // s (0 => not simulated) }; // Utilities template<> struct Traits<Debug>: public Traits<void> { static const bool error = true; static const bool warning = true; static const bool info = false; static const bool trace = false; }; ... __END_SYS #endif (END)
```

A set of configuration tokens and default values is kept at `include/system/traits.h`:

```
emplate<typename T> struct Traits { // EPOS software architecture (aka mode) enum {LIBRARY, BUILTIN, KERNEL}; // CPU hardware architectures enum {AVR8, H8, ARMv4, ARMv7, ARMv8, IA32, X86_64, SPARCv8, PPC32}; // Machines enum {eMote1, eMote2, STK500, RCX, Cortex, PC, Leon, Virtex}; // Machine models enum {Unique, Legacy_PC, eMote3, LM3S811, Zynq, Realview_PBX, Raspberry_Pi3}; // Serial display engines enum {UART, USB}; // Life span multipliers enum {FOREVER = 0, SECOND = 1, MINUTE = 60, HOUR = 3600, DAY = 86400, WEEK = 604800, MONTH = 2592000, YEAR = 31536000}; // IP configuration strategies enum {STATIC, MAC, INFO, RARP, DHCP}; // SmartData predictors enum :unsigned char {NONE, LVP,
```

DBP}; // Default traits static const bool enabled = true; static const bool debugged = true; static const bool monitored = false; static const bool hysterically_debugged = false; typedef LIST<> DEVICES; typedef TLIST<> ASPECTS; };

Debugging is therefore enabled by default for critical errors. In order to collect execution traces, a programmer would make:

Traits<Debug>::trace = true
To build for EPOSMote III, they would adjust `Traits<Build>` like this:

template<> struct Traits<Build>: public Traits<void> { static const unsigned int MODE = LIBRARY; static const unsigned int ARCHITECTURE = ARMv7; static const unsigned int MACHINE = Cortex; static const unsigned int MODEL = eMote3; static const unsigned int CPUS = 1; static const unsigned int NODES = 1; // (> 1 => NETWORKING) static const unsigned int EXPECTED_SIMULATION_TIME = 0; // s (0 => not simulated) };

Note: From EPOS 2.0, makefile customizations are no longer needed. Makefiles now parse the application's traits to adjust themselves and produce a proper instance of EPOS.
Note: EPOS makefiles care for cleaning the configuration between any two builds, but if you change an application's traits, this will not be perceived as a different build. In this case, issue a `make veryclean` to clean up internal configuration info.

4. EPOS API

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

4.1. Memory Management

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

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

4.1.1. Dynamic Memory (Heap)

Dynamic memory allocation is supported in EPOS through the ordinary C++ operators **new** and

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

Examples

```
Thread * thread = new Thread(&function); Mutex * mutex = new Mutex; int ** matrix = new int*[ROWS]; for(int i = 0; i < ROWS; ++i) matrix[i] = new int[COLUMNS]; for(int i = 0; i < ROWS; ++i) delete [] matrix[i]; delete [] matrix; delete mutex; delete thread;
```

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

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

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

4.1.2. Stacks

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

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

4.1.3. Memory Segments

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

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

Header
`include/memory.h`

Interface

```
```

class Segment { public: typedef MMU::Flags Flags; typedef CPU::Phy_Addr Phy_Addr; public: Segment(unsigned int bytes, Flags flags = Flags::APP); Segment(Phy_Addr phy_addr, unsigned int bytes, Flags flags); ~Segment(); unsigned int size() const; Phy_Addr phy_address() const; int resize(int amount); };

Methods

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

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

- `Segment(Phy_Addr phy_addr, unsigned int bytes, Flags flags)`
  Encapsulates a memory region of `bytes` bytes starting at `phy_addr` as a Segment that can be attached to an Address Space. This constructor does not allocate memory. It simply maps a preexisting memory region as a Segment. In addition to the flags described above, this constructor can take:
  - `IO`: memory-mapped I/O (main memory if absent)

- `~Segment()`
  Destroys a segment, releasing the associated memory (unless the segment was created with `IO`, case in which only the corresponding page tables are released).

- `unsigned int size() const`
  Returns the current size of a Segment.

- `Phy_Addr phy_address() const`
  Returns the physical address of the contiguous Segment (i.e. a Segment created with `CT`). Requesting the physical address of a scattered segment is invalid and returns `Phy_Addr(false)`.

- `int resize(int amount)`
  Grows or shrinks a Segment by `amount` bytes. A contiguous Segment (i.e. a Segment created with `CT`) can only be expanded using adjacent memory blocks.

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

Examples

```
```

// With MULTITASKING ENABLED // Creates a Segment of 400K (that can be shared with other

processes): Segment shared * = new Segment(400*1024); Task::self()->address_space()->attach(shared); // Resizes the Data Segment by 1M: Task::self()->data_segment()->resize(1024*1024); // Independently of MULTITASKING being enabled or not // Maps a PCI device's memory region so I can also be accessed from de CPU PCI::Locator loc = PCI::scan(VENDOR_ID, DEVICE_ID, UNIT); PCI::Header hdr; PCI::header(loc, &hdr); Segment * io_mem; if(hdr) io_mem = new Segment(hdr.region[MEM].phy_addr, hdr.region[MEM].size, Flags::CD); else io_mem = 0; // Creates a 16 K Segment that can be shared with and I/O device for DMA operations Segment * dma_mem = new Segment(16*1024, Flags::CT); Phy_Addr dma_addr = dma_mem->phy_address();

4.1.4. Address Spaces

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

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

Address_Space * as = new Address_Space(MMU::current());
For multitasking configuration, the current Address Space can be obtained with:

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

Header

`include/memory.h`

Interface

class Address_Space { public: Address_Space(); Address_Space(MMU::Page_Directory * pd); ~Address_Space(); using MMU::Directory::pd; Log_Addr attach(Segment * seg); Log_Addr attach(Segment * seg, const Log_Addr & addr); void detach(Segment * seg); void detach(Segment * seg, const Log_Addr & addr); Phy_Addr physical(const Log_Addr & address); };

Methods

- `Address_Space()`
  Creates an Address Space, usually for a new Task.
- `Address_Space(MMU::Page_Directory * pd)`
  Returns a reference to an Address Space using `pd` as the primary page table.
- `~Address_Space()`
  Destroys an Address Space
- `~MMU::Page_Directory * pd()`
  Returns the current primary page table (called page directory by Intel).
- `Log_Addr attach(Segment * seg)`
  Attaches the Segment designated by `seg` at the first available address. If the target Address Space does not feature any slot large enough to contain the Segment, then `Log_Addr(false)` is returned.
- `Log_Addr attach(Segment * seg, const Log_Addr & addr)`
  Attaches the Segment designated by `seg` at address `addr`. If the target address is already

mapped, the `Log_Addr(false)` is returned.

- `void detach(Segment * seg)`
  Detaches the Segment designated by `seg` from the Address Space. Detaching a Segment that has not been previously attached might be a harmful operation in some architectures.

- `void detach(Segment * seg, const Log_Addr & addr)`
  Detaches the Segment designated by `seg` from `addr` at the Address Space. Detaching a Segment that has not been previously attached or detaching it from a different address might be a harmful operation in some architectures.

- `Phy_Addr physical(const Log_Addr & addr)`
  Returns the physical address currently bound to `addr` in the Address Space.

Examples

```
// Creates a Segment of 400K and attaches it to the Address Space: Segment shared * = new Segment(400*1024); // With MULTITASKING ENABLED Task::self()->address_space()->attach(shared); // With MULTITASKING DISABLED Address_Space(MMU::current()).attach(shared);
```

4.2. Process Management

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

4.2.1. Task

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

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

Header
`include/process.h`

Interface

```
class Task { template<typename ... Tn> Task(Segment * cs, Segment * ds, int (* entry)(Tn ...), Tn ... an); template<typename ... Tn> Task(const Thread::Configuration & conf, Segment * cs, Segment * ds, int (* entry)(Tn ...), Tn ... an); ~Task(); Address_Space * address_space(); Segment * code_segment(); Segment * data_segment(); Log_Addr code(); Log_Addr data(); Thread * main(); static Task * volatile self(); }
```

Methods

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

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

- `template<typename ... Tn>`
  `Task(const Thread::Configuration & conf, Segment * cs, Segment * ds, int (* entry)(Tn ...), Tn ... an)`
  This constructor is similar the the previous, but takes an addition Configuration pack (see Thread for details).

- `~Task()`
  Destroys a Task and consequently deletes (i.e. kills) all its Threads.

- `Address_Space * address_space()`
  Returns the Task's Address Space.

- `Segment * code_segment()`
  Returns the Task's code Segment.

- `Segment * data_segment()`
  Returns the Task's data Segment.

- `Log_Addr code()`
  Returns the address the Task's code Segment is mapped to in its Address Space.

- `Log_Addr data()`
  Returns the address the Task's data Segment is mapped to in its Address Space.

- `Thread * main()`
  Returns the address of the function used to create the Task's first Thread (usually the function `main()`).

- `static Task * volatile self()`
  Returns a reference to the running Task.

Examples

```
// Create a new Task and its initial Thread from an ELF object ELF * elf = ... ; Address_Space * as = Task::self()->address_space(); // Create and load the CODE segment Segment * cs = new Segment(elf->segment_size(0)); CPU::Log_Addr code = as->attach(cs); if(elf->load_segment(0, code) < 0) { cerr << "Application code segment is corrupted!" << endl; return; } as->detach(cs); // Create and load the DATA segment with room for a Heap Segment * ds = new
```

Segment(elf->segment_size(1) + S::Traits<Application>::HEAP_SIZE); CPU::Log_Addr data = as->attach(ds); if(elf->load_segment(1, data) < 0) { cerr << "Application data segment is corrupted!" << endl; return; } as->detach(ds); // Create the Task int (* entry)() = CPU::Log_Addr(elf->entry()); Task * task = new Task(cs, ds, entry); // Wait for it to finish task->main()->join();

4.2.2. Thread

If a process is a program in execution, then a Thread encompasses its dynamic aspects. A Thread has a private context and a private stack. It runs a Task's code and manipulates its data.

Header

`include/process.h`

Interface

class Thread { public: enum State { RUNNING, READY, SUSPENDED, WAITING, FINISHING }; typedef Scheduling_Criteria::Priority Priority; typedef Traits<Thread>::Criterion Criterion; enum { HIGH = Criterion::HIGH, NORMAL = Criterion::NORMAL, LOW = Criterion::LOW, MAIN = Criterion::MAIN, IDLE = Criterion::IDLE }; struct Configuration { State state; Criterion criterion; Task * task; unsigned int stack_size; }; template<typename ... Tn> Thread(int (* entry)(Tn ...), Tn ... an); template<typename ... Tn> Thread(const Configuration & conf, int (* entry)(Tn ...), Tn ... an); ~Thread(); const volatile State & state(); const volatile Priority & priority(); void priority(const Priority & p); Task * task(); int join(); void pass(); void suspend(); void resume(); static Thread * volatile self(); static void yield(); static void exit(int status = 0); }

Types

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

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

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

- `Configuration`

This type is used to define a configuration pack for Threads. The following parameters can be adjusted:

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

Methods

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

- `template<typename ... Tn>`
  `Thread(const Configuration & conf, int (* entry)(Tn ...), Tn ... an)`
  Creates a Thread on the running Task to run the function given by `entry`. The Thread's creation is controlled by `conf` (see the `Configuration` type declaration above). The remainder of the C++ parameter pack is consistently passed to the Thread following the architecture's call convention (stack, register set, window, etc).

- `~Thread()`
  Destroys a Thread (respecting the corresponding C++ object's semantics; e.g. the destructor does not delete the object).

- `const volatile State & state()`
  Returns the Thread's current state.

- `const volatile Priority & priority()`
  Returns the Thread's current priority (i.e. an integer representing the ordering imposed by the Criterion in place).

- `void priority(const Priority & p)`
  Adjusts the Thread's priority according to the Criterion in place.

- `Task * task()`
  Returns the Task this Thread is running on.

- `int join()`
  Waits for this Thread to finish and returns the value passed over at `return` (or `exit()`).

  Note: the `int` return type is defined by the C++ standard as the only one valid for the `main()` function and therefore requires EPOS to follow it. POSIX further limits the interpretation of that integer to 8 bits. EPOS would prefer to abolish it if there were a `void main()` valid signature. Programmers can define their own semantics for the integer in EPOS.

- `void pass()`
  Hands the CPU over to this Thread. This function can be used to implement user-level schedulers. A Thread can be created with a higher priority to act as the scheduler. EPOS scheduler will always elect it, but it can in turn `pass()` the CPU to another Thread. Accounting is done for the Thread receiving the CPU, but timed scheduling criteria are not reset. In this way, the calling Thread is charged only for the time it took to hand the CPU over to another Thread, which inherits the CPU without further intervention from EPOS' scheduler.

- `void suspend()`
  Suspends the execution of this Thread. The Thread's state is set to `SUSPENDED` and it will not be eligible for scheduling until `resume()` is called. If called for the running Thread, this method triggers a rescheduling.

- `void resume()`
  Resumes the execution of this Thread by setting its state to `READY` and notifying the Scheduler. Whether or not this notification will trigger a reschedule is Criterion dependent. Resuming a Thread that is not suspended is an invalid operation, even if for most policies this would have no effects.

- `static Thread * volatile self()`
  Returns a reference to the running Thread (actually, a volatile reference to a pointer designating it).

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

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

Examples

Complete me!

4.2.3. RT_Thread

The Real-time Thread abstraction is an specialization of Thread designed to handle a variety of scenarios in the realm or Periodic Real-time Scheduling.

Header

`include/real-time.h`

Interface

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

Methods

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

Examples

□□□□□□□□

Complete me!

4.2.4. Scheduler

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

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

- Preemptive: a Preemptive Criterion requires a reevaluation, and eventually a rescheduling, whenever a Thread enters the `READY` state, independently of the previous state (e.g. a newly created Thread, a Thread released from a Mutex, a Thread that was waiting fro I/O). A non-preemptive Criterion will only be reevaluated when the `RUNNING` Thread explicitly causes it state to change (e.g. by blocking on a Synchronizer or by invoking I/O operations). All timed Criteria

are preemptive. Most priority-based Criteria are also preemptive. *Shortest Job First* is a non-Preemptive Criterion.
- Timed: a Timed Criterion requires a `QUANTUM` to be specified (in µs). This constant defines the maximum time a Thread can run before the Scheduler rechecks the Criterion in place (eventually scheduling another Thread). The value of `Traits<Thread>::QUANTUM` must be carefully chosen: a value of a few µs will cause the system to reevaluate the Criterion too often and will result in (very) large overhead, eventually bringing the system to thrash; a value of hundreds of ms will enable CPU-bound threads to monopolize the CPU, eventually degrading the system responsiveness. Values between 100 µs and 100 ms are common. All timed Criteria are preemptive. *Round-robin* is a Timed Criterion.
- Dynamic: a Dynamic Criterion is recalculated at run-time to constantly reflect the police in force. There are two moments at which a Dynamic Criterion can be recalculated: at dispatch and at release. For Aperiodic Threads, for which no period is defined, it is done when the Thread leaves the CPU (i.e. another Thread is dispatched). For Periodic Threads, recalculating at dispatch would not be adequate, since jobs of other Threads will still be released before the next activation and they may influence on the calculations. Therefore, Periodic Threads subjected to Dynamic Criteria are reevaluated before the release of each *job*. Earliest Deadline First is Dynamic Criterion.
- Energy-aware: Criteria with this trait will cause low priority Threads to be suspended whenever their execution could cause a critical Thread to fail due to the lack of power. In order to enforce such a regimen, Energy-aware Criteria require `Traits<System>::LIFE_SPAN` to be defined. An energy monitoring mechanism is also enabled in the platforms supporting it.

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

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

4.3. Process Coordination (Synchronizers)

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

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

4.3.1. Semaphore

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

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

Header
`include/synchronizer.h`

Interface

```
```

class Semaphore { public: Semaphore(int v = 1); ~Semaphore(); void p(); void v(); }

Methods

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

Examples

```
```

Complete me!

4.3.2. Mutex

The Mutex member of the Synchronizer family implements a Binary Semaphore.

Header
`include/synchronizer.h`

Interface

```
```

class Mutex { public: Mutex(); ~Mutex(); void lock(); void unlock(); }

Methods

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

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

Examples

Complete me!

4.3.3. Condition

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

Header

`include/synchronizer.h`

Interface

class Condition { public: Condition(); ~Condition(); void wait(); void signal(); void broadcast(); }

Methods

- `Condition()`
  Creates a condition variable.
- `~Condition()`
  Destroys a condition variable, releasing eventual blocked Threads.
- `wait()`
  Implicitly unlocks the shared data and puts the calling Thread to wait for the assertion of a predicate. Several threads can be waiting on the same condition. The assertion of a predicate can be announced either to the first blocked Thread or to all blocked Threads. When a thread returns from the wait operation, it implicitly regains control over the critical section.
- `signal()`
  Announces the assertion of a predicate to the first waiting Thread, releasing it for execution (i.e. making its state `READY` and notifying the Scheduler).
- `broadcast()`
  Announces the assertion of a predicate to all waiting Threads, making their state `READY` and notifying the Scheduler.

Examples

4.4. Timing

Time management in EPOS encompasses abstractions to measure time intervals, to keep track of the

current time, and also to trigger timed events.

4.4.1. Clock

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

Header

`include/time.h`

Interface

```
```

class Clock { public: class Date { public: Date() {} Date(unsigned int Y, unsigned int M, unsigned int D, unsigned int h, unsigned int m, unsigned int s); Date(const Second & seconds, unsigned long epoch_days = 0); operator Second(); Second to_offset(unsigned long epoch_days = 0); unsigned int year(); unsigned int month(); unsigned int day(); unsigned int hour(); unsigned int minute(); unsigned int second(); void adjust_year(int y); } public: Clock(); ~Clock(); Microsecond resolution(); Second now(); Date date(); void date(const Date & d); }

Types

- `Microsecond`
  An unsigned integer representing µs. Its resolution is adjusted according to `Traits<System>::LIFE_SPAN` either to 32 or 64 bits.
- `Second`
  An unsigned integer representing seconds. Its resolution is adjusted according to `Traits<System>::LIFE_SPAN` either to 32 or 64 bits.
- `Date`
  Data structure to store the components of a date: year, month, day, hour, minute, and second; as unsigned integers. It features methods to convert this representation of data to and from an offset in seconds from a given epoch

Methods

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

Examples

Complete me!

4.4.2. Chronometer

Chronometer abstracts a timepiece able to measure time intervals. Its precision and resolution depend on the timing devices available in the platform (e.g. real-time clocks, CPU clock counters, high-performance timers).

Header

`include/time.h`

Interface

class Chronometer { public: Chronometer(); ~Chronometer(); Hertz frequency(); void reset(); void start(); void lap(); void stop(); Microsecond read(); }

Methods

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

Examples

Complete me!

4.4.3. Alarm

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

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

Header

`include/time.h`

Interface

class Alarm { public: Alarm(const Microsecond & time, Handler * handler, int times = 1); ~Alarm(); static Hertz frequency(); static void delay(const Microsecond & time); }

Methods

- `Alarm(const Microsecond & time, Handler *handler, int times = 1)`
  Creates an Alarm to trigger `handler` after `time` µs. The event will occur `times` times or forever if `INFINITE` is given. Handler must be polymorphic and it must implement the call operator (-+void operator()()+--) for the trigger. See the Handler utility for details.
- `~Alarm()`
  Destroys an Alarm;
- `Hertz frequency()`
  Returns the Alarm's frequency in Hertz.
- `void delay(const Microsecond & time)`
  Delays a Thread execution by `time` µs.

Examples

Complete me!

4.4.4. Delay

Delay is used to delay the execution of Threads by a given, usually small, amount of time.

Header

`include/time.h`

Interface

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

Methods

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

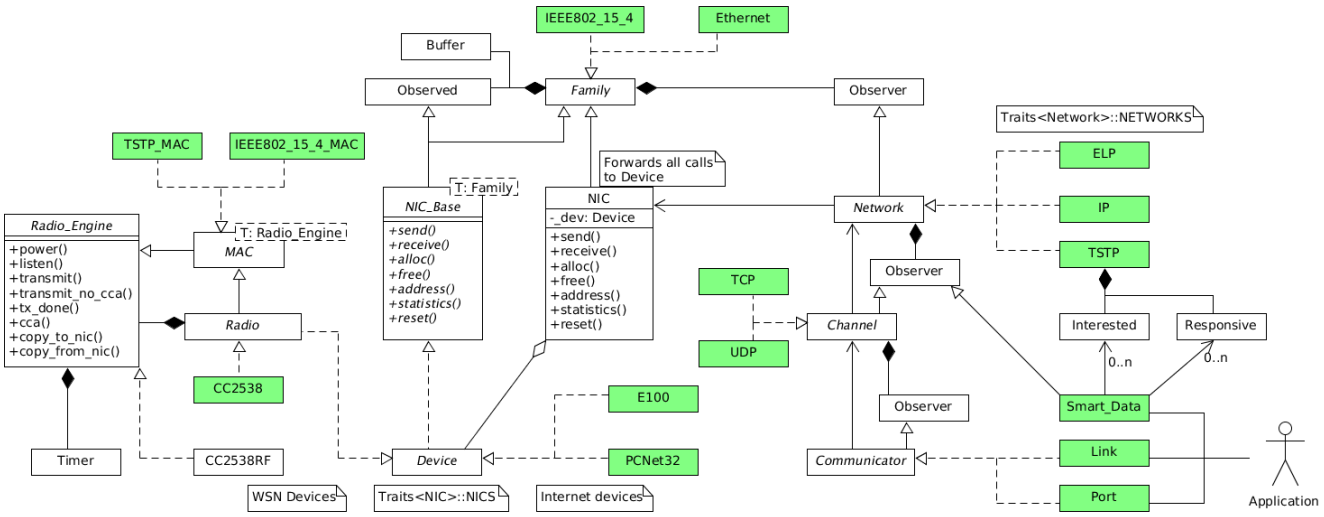
Examples

Complete me!

4.5. Communication

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

An overview of EPOS' communication structure is shown below.



4.5.1. Link

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

Header

`include/communicator.h`

Interface

template<typename Protocol> class Link { public: typedef typename Protocol::Address Address; typedef typename Protocol::Address::Local Local_Address; public: Link(const Local_Address & local, const Address & peer = Address::NULL); ~Link(); int read(void * data, unsigned int size); int write(const void * data, unsigned int size); const Address & peer(); }

Methods

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

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

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

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

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

Examples

Complete me!

4.5.2. Port

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

Header

`include/communicator.h`

Interface

template<typename Protocol> class Port { public: typedef typename Protocol::Address Address; typedef typename Protocol::Address::Local Local_Address; public: Port(const Local_Address & local); ~Port(); Link<Channel> * listen(); Link<Channel> * connect(const Address & to); }

Methods

- `Port(const Local_Address & local)`

Creates a Port with address `local` to listen on for connection requests. Creating a Port on a previously assigned Address is invalid for most Protocols.

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

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

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

Examples

Complete me!

4.5.3. Mailbox

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

Header

`include/communicator.h`

Interface

template<typename Protocol> class Mailbox { public: typedef typename Protocol::Address Address; typedef typename Protocol::Address::Local Local_Address; public: Mailbox(const Local_Address & local); ~Mailbox(); int send(const Address & to, const void * data, unsigned int size); int receive(Address * from, void * data, unsigned int size); }

Methods

- `Mailbox(const Local_Address & local)`
  Creates a Mailbox with address `local`. The Mailbox can be used both to sent and to receive messages. Creating a Mailbox on a previously assigned Address is invalid for most Protocols.

- `~Mailbox()`
  Destroys the Mailbox, releasing the local address.

- `int send(const Address & to, const void * data, unsigned int size)`
  Sends a Message to `to` containing `size` bytes of data stored at `data`. The method returns the number of bytes effectively sent.

- `int receive(Address * from, void * data, unsigned int size)`
  Receives a Message and copies up to `size` bytes of its data to `data`. The calling Thread is blocked until the packet is received. `from` is updated with the address of the sender. The

number of bytes effectively received (and copied) is returned.

Examples

Complete me!

4.5.4. Channel

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

4.5.5. Network

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

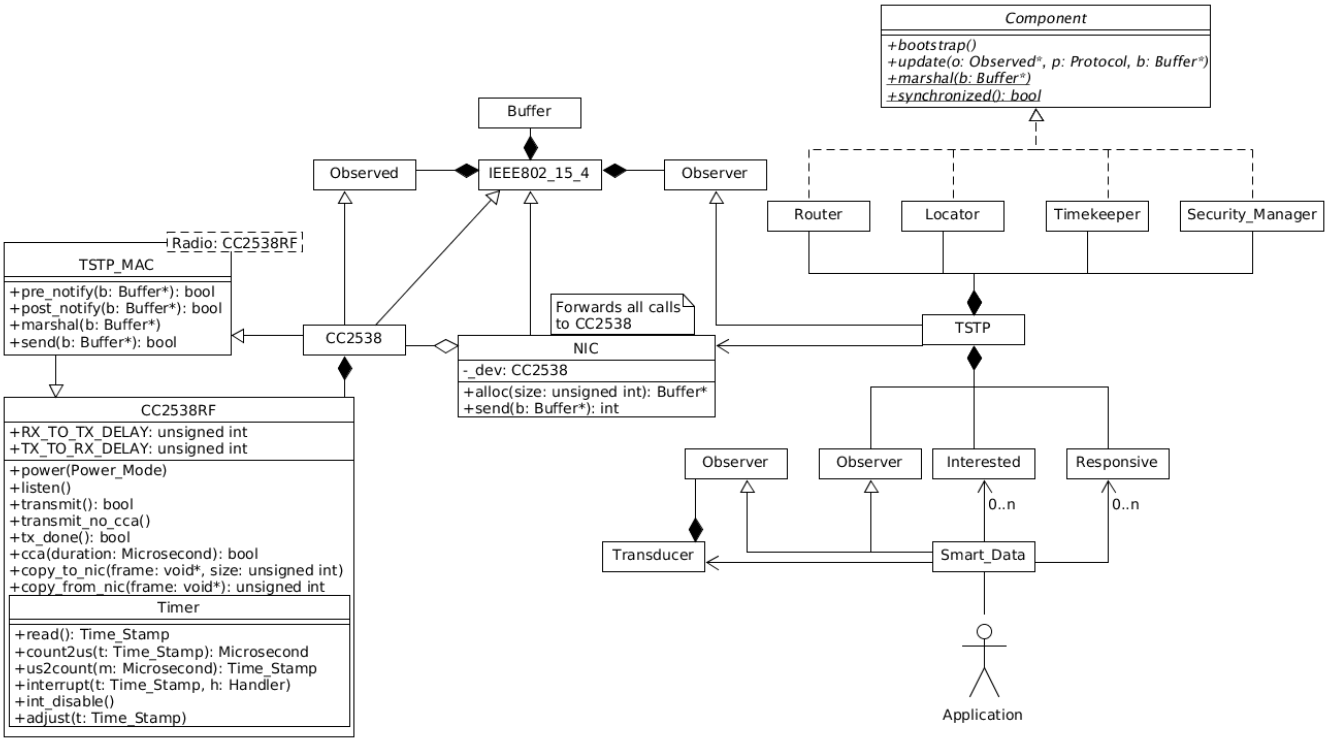
4.5.6. IPC

TODO

4.5.7. TSTP

The **Trusful Space-Time Protocol (TSTP)** is EPOS' response to Wireless Sensor Network in terms of protocols. It consolidates over a decade of research on the theme in a cross-layer protocol that features Semantic Data, Authentication, Encryption, Timing, Location, Convergecast Routing, and an Energy-Efficient MAC. It is best exposed to users through SmartData.

TSTP comprises the whole network stack, from the application layer to the Medium Access Control (MAC) layer (the current implementation of TSTP MAC assumes an IEEE 802.15.4 2450MHz DSSS PHY layer). The figure below shows a configuration of EPOS' network stack for TSTP@EPOSMoteIII with a single NIC.



TSTP is a **geographic** protocol, and every node in the network is synchronized in space and time, which means that every node has **spatial coordinates** and a **synchronized clock**. A TSTP network consists of *one* **sink** node and any number of **sensor** nodes. The sink is the reference for clock synchronization and spatial localization.

The main TSTP subcomponents are **Router**, **Locator**, **Timekeeper**, **Security Manager**, and **MAC**. Each one is responsible for different aspects of TSTP's functionality, and the design of each one can be intimately connected to another component.

4.5.7.1. Configuration

To enable TSTP, you need to set TSTP as a network for a NIC, as in the example below (see also Configuring Networking).

> template<> struct Traits<Network>: public Traits<Build> { typedef LIST<TSTP> NETWORKS; static const unsigned int RETRIES = 3; static const unsigned int TIMEOUT = 10; // s static const bool enabled = (Traits<Build>::NODES > 1) && (NETWORKS::Length > 0); };

TSTP allows for pre-definition of coordinates for static nodes. You can define the coordinates of the node in `src/component/tstp_init.cc`, at the `Locator::bootstrap()` method, by setting the `_here` variable: `_here = Coordinates(10,10,10);`.

To deploy a sink node, set its coordinates to (0,0,0) inside `Locator::bootstap();`. Any node which is not at (0,0,0) will act as a **sensor** node located in the given coordinates.

4.5.7.2. Bootstrap

Before calling your application's `main()`, TSTP components need to **bootstrap**. This is done during TSTP's initialization (`TSTP::init()` at `src/component/tstp_init.cc`), by calling the `bootstrap()` method of each component. The Locator sets the initial coordinates for the node; the Timekeeper

synchronizes the clock with the network; The Security Manager establishes a shared cryptographic key with the sink.

Note: the `main()` method of a sensor node's application will not be executed until bootstrap is complete. You need a running TSTP sink to be able to run an application on a TSTP sensor!

4.5.7.3. Interaction between components

The present tightly-coupled cross-layered design does not imply in a monolithic software implementation. To make component interaction efficient and uncoupled, three strategies are used: **Zero-copy Buffer Management**, **Metadata Gathering**, and **Event Propagation**.

4.5.7.3.1. Zero-copy Buffer Management

Zero-copy is a mechanism that implements efficient message passing and it consists on transferring the ownership of pointers to data buffers from one component to another.
As only pointers are transferred, it avoids copying data between components (such as layers of a layered protocol stack), thus achieving better efficiency.

TSTP uses EPOS' Zero-Copy Buffer utility to share messages between components, NIC, and application.

4.5.7.3.2. Metadata Gathering

To allow network components to share information that goes beyond the message's header (e.g. precise time of arrival), we enrich each buffer with metadata, which is visible to every component but not transmitted through the network. Upon reception of a message, each component must populate and adjust specific portions of the metadata. Because the order of buffer processing and which component is responsible for which piece of metadata are well-defined, each component knows what metadata it can use for its own purposes. For example, the MAC component inserts the values of RSSI and precise SFD time stamp read from the radio hardware. The buffer (holding the message and the metadata) is passed to the Locator, which uses the RSSI to update its position estimation. The buffer is then passed to the Timekeeper, which potentially uses the SFD time stamp to synchronize the node's clock with the sender of that message. This way, the Timekeeper and Locator implementations are hardware-independent (reading the radio's registers is the role of the MAC), and indeed independent on the implementation of any other component, as long as the necessary metadata is populated in each network buffer at the right time.

The metadata available at TSTP buffers is defined in `include/nic.h`:

```
// Buffer Metadata added to frames by higher-level protocols struct Metadata { int rssi; // Received Signal Strength Indicator unsigned long long sfd_time_stamp; // Start-of-frame reception time stamp unsigned int id; // Message identifier unsigned long long offset; // MAC contention offset bool destined_to_me; // Whether this node is the final destination for this message bool downlink; // Message direction (downlink == from sink to sensor) unsigned long long deadline; // Time until when this message must arrive at the final destination unsigned int my_distance; // This node's distance to the message's final destination unsigned int sender_distance; // Last hop's distance to the message's final destination bool is_new; // Whether this message was just created by this node bool is_microframe; // Whether this message is a Microframe bool relevant; // Whether any component is interested in this message bool trusted; // If true, this message was successfully verified by the Security Manager bool freed; // If true, the
```

MAC will not free this buffer unsigned int attempts; // Number times the MAC tried to transmit this buffer unsigned int microframe_count; // Number of Microframes left until data };

4.5.7.3.3. Event Propagation

Buffers are propagated to components using the publisher/subscriber design pattern, in which the upper layers are observers (subscribers) of the lower layers (publishers).
Components observe the NIC by calling the `attach()` method. When a message is received by the NIC, it notifies any observers attached to it *in the order they were attached*, by calling the respective -+update()-+ method, passing a reference to the buffer containing the message (including all the headers) and metadata. The last component to be notified is the one responsible for the API, which delivers the message to the application (usually a SmartData object) if necessary.

4.5.7.4. Coordinates

On TSTP, the coordinate is relative to the Sink. The sink node is capable to convert this coordinate to another coordinate system. For example, if you want to convert the coordinate to the system relative to Earth's mass center (ECEF), you need to convert the Sink's position from traditional coordinate system to ECEF. We recommend you to follow this steps:

1. Access this site. Put a reference name of your location and adjust the position of the marker. Copy the lat/lng/alt of the Sink's location for the next step.

2. Now, you need access this site. Paste the lat/lng/alt in the respective fields. Click on the button "LLH to ECEF". Done, you will get the position relative to the Earth's mass center. Note that for being used with the TSTP you must convert from km to cm.

4.5.8. TCP/IP

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

4.5.8.1. ARP

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

4.5.8.2. DHCP

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

4.5.8.3. IP

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

4.5.8.4. ICMP

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

Header
`include/icmp.h`

Interface

```
class ICMP { public: // ICMP Packet Types typedef unsigned char Type; enum { ECHO_REPLY = 0, UNREACHABLE = 3, SOURCE_QUENCH = 4, REDIRECT = 5, ALTERNATE_ADDRESS = 6, ECHO = 8, ROUTER_ADVERT = 9, ROUTER_SOLIC = 10, TIME_EXCEEDED = 11, PARAMETER_PROBLEM = 12, TIMESTAMP = 13, TIMESTAMP_REPLY = 14, INFO_REQUEST = 15, INFO_REPLY = 16, ADDRESS_MASK_REQ = 17, ADDRESS_MASK_REP = 18, TRACEROUTE = 30, DGRAM_ERROR = 31, MOBILE_HOST_REDIR = 32, IPv6_WHERE_ARE_YOU = 33, IPv6_I_AM_HERE = 34, MOBILE_REG_REQ = 35, MOBILE_REG_REP = 36, DOMAIN_NAME_REQ = 37, DOMAIN_NAME_REP = 38, SKIP = 39 }; // ICMP Packet Codes typedef unsigned char Code; enum { NETWORK_UNREACHABLE = 0, HOST_UNREACHABLE = 1, PROTOCOL_UNREACHABLE = 2, PORT_UNREACHABLE = 3, FRAGMENTATION_NEEDED = 4, ROUTE_FAILED = 5, NETWORK_UNKNOWN = 6, HOST_UNKNOWN = 7, HOST_ISOLATED = 8, NETWORK_PROHIBITED = 9, HOST_PROHIBITED = 10, NETWORK_TOS_UNREACH = 11, HOST_TOS_UNREACH = 12, ADMIN_PROHIBITED = 13, PRECEDENCE_VIOLATION = 14, PRECEDENCE_CUTOFF = 15 }; class Address: public IP::Address; struct Header { unsigned char _type; unsigned char _code; unsigned short _checksum; unsigned short _id; unsigned short _sequence; } __attribute__((packed)); // ICMP Packet static const unsigned int MTU = 56; static const unsigned int HEADERS_SIZE = sizeof(IP::Header) + sizeof(Header); typedef unsigned char Data[MTU]; class Packet: public Header { public: Packet(); Packet(const Type & type, const Code & code); Packet(const Type & type, const Code & code, unsigned short id, unsigned short seq); Header * header(); template<typename T> T * data(); void sum(); bool check(); private: Data _data; } __attribute__((packed)); typedef Packet PDU; }
```

Examples

4.5.8.5. UDP

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

Examples

4.5.8.6. TCP

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

Examples

4.5.9. Networking Configuration

Enabling networking in EPOS is easy. You just have to set `Traits<Build>::NODES` to a value larger than one. The following configuration snippet sets up a PC with 3 NICs, 2 x PCNet32 and 1 x RTL8139, enables IP for all of them and defines a MAC-based IP selection for the first, STATIC for the second, and DCHP for the third.
Configuration

```
template<> struct Traits<Ethernet>: public Traits<Machine_Common> { typedef LIST<PCNet32, PCNet32, RTL8139> DEVICES; static const unsigned int UNITS = DEVICES::Length; static const bool enabled = (Traits<Build>::NODES > 1) && (UNITS > 0); }; template<> struct Traits<Build> { static const unsigned int NODES = 100; }; template<> struct Traits<Network>: public Traits<Build> { typedef LIST<IP, IP, IP> NETWORKS; static const unsigned int RETRIES = 3; static const unsigned int TIMEOUT = 10; // s static const bool enabled = (Traits<Build>::NODES > 1) && (NETWORKS::Length > 0); }; template<> struct Traits<IP>: public Traits<Network> { typedef Ethernet NIC_Family; static constexpr unsigned int NICS[] = {0, 1, 2}; static const unsigned int UNITS = COUNTOF(NICS); struct Default_Config { static const unsigned int TYPE = DHCP; static const unsigned long ADDRESS = 0; static const unsigned long NETMASK = 0; static const unsigned long GATEWAY = 0; }; template<unsigned int UNIT> struct Config: public Default_Config {}; static const unsigned int TTL = 0x40; // Time-to-live static const bool enabled = Traits<Network>::enabled && (NETWORKS::Count<IP>::Result > 0); }; template<> struct Traits<IP>::Config<0> { static const unsigned int TYPE = MAC; static const unsigned long ADDRESS = 0x0a000100; // 10.0.1.x x=MAC[5] static const unsigned long NETMASK = 0xffffff00; // 255.255.255.0 static const unsigned long GATEWAY = 0; // 10.0.1.1 }; template<> struct Traits<IP>::Config<1> { static const unsigned int TYPE = STATIC; static const unsigned long ADDRESS = 0x0a000110; // 10.0.1.16 static const unsigned long NETMASK = 0xffffff00; // 255.255.255.0 static const unsigned long GATEWAY = 0; // 10.0.1.1 }; template<> struct Traits<IP>::Config<2> { static const unsigned int TYPE = DHCP; static const unsigned long ADDRESS = 0; static const unsigned long NETMASK = 0; static const unsigned long GATEWAY = 0; };
```

4.6. Sensing and Actuation (Wireless Sensor Network)

Wireless Sensor Network (WSN) has been a very hot topic of research for EPOS. This family of abstractions consolidates over a decade of research behind a simple, easy-to-use, application-oriented API. **Sensors** and **Actuators** are both abstracted to users through a novel, data-centric construct we named **SmartData**.

4.6.1. SmartData

SmartData encapsulates a large set of recurring design patterns in the realm of WSN behind a powerful, application-oriented interface. A SmartData "observes" a TSTP network and a Transducer (a sensor or an actuator) and interfaces them in a data-centric way. Programmers interact with it just like they would with an ordinary piece of Data coming either from the network or from a (potentially remote) transducer. A SmartData can be of two kinds: a Physical Quantity identified through the corresponding SI (derived) Unit, or a piece of Digital data. The first is used for most sensors (and related actuators) that we call a "meter": accelerometer, magnetometer, thermometer, voltmeter, amperimeter, etc. The latter is used for transducers whose final purpose is to produce digital data, such as switches, buttons, cameras, and audio capture devices.

SmartData honors **TSTP** option for a Convergecast routing strategy that does not use addresses in favor of Space-Time coordinates. Consequently, whenever a SmartData is advertised, it is advertised

to the *Sink*, which can declare *Interests* for a given kind of (smart) data. Sensors behind a SmartData can *Respond* to such interests, while actuators can receive *Commands* from the *Sink*.

Header

`include/smartdata.h`

Interface

template<typename Transducer> class SmartData: private TSTP::Observer, private Transducer::Observer { public: static const unsigned int UNIT = Transducer::UNIT; static const unsigned int NUM = Transducer::NUM; static const unsigned int ERROR = Transducer::ERROR; typedef TSTP::Unit Unit; typedef typename TSTP::Unit::Get<NUM>::Type Value; typedef TSTP::Error Error; typedef TSTP::Coordinates Coordinates; typedef TSTP::Time Time; enum Mode { PRIVATE = 0, ADVERTISED = 1, COMMANDED = 3 }; public: Smart_Data(unsigned int dev, const Microsecond & expiry, const Mode & mode = PRIVATE); Smart_Data(const Region & region, const Microsecond & expiry, const Microsecond & period = 0); ~Smart_Data(); operator Value(); const Coordinates & location() const; const Time & time() const; };

Types

- `Unit`
  Represents the type of the encapsulated piece of data, either an SI Quantity or Digital data. For SI Quantities, `Unit` encodes the associated SI Unit. For Digital data, it encodes a type Id. See SI Quantities for additional information.

- `Value`
  Represents the encapsulated piece of data (i.e. the SmartData content). For Digital data, it defines a string of bytes (`unsigned char[]`). For SI Quantities, `Value` is an alias to the native C++ type associated with the `NUM` field encoded in `Unit`. See SI Quantities for additional information.

- `Error`
  For SI Quantities, `Error` represents the scale of the measurement error as an order of magnitude (i.e. 10^{ERROR}).

- `Coordinates`
  Represents the location where the data was produced. It designates a 3D-point in a Cartesian Coordinate System. This type does not assume a fixed center for the Coordinate System. It can be used to represent positions relative to the Sink, to Earth's center or to any other arbitrary point. The (x, y, z) triple is stored as 8, 16, or 32-bit scaled signed integers depending on the configured network size. See TSTP Coordinates for additional information.

- `Time`
  Represents the time in which the data was produced as an offset in µs from `Traits<RTC>::EPOCH` (usually January 1st, 1970). It is stored as an `unsigned long long int` (64 bits).

- `Mode`
  Defines the operation mode for a local transducer (sensor or actuator). Has no meaning for remote transducers.
  - `PRIVATE`: the local SmartData is private to the process that created it. It does not get advertised to the network and therefore cannot be monitored nor controlled remotely.

    - `ADVERTISED`: the local SmartData is advertised to the network and therefore can be remotely monitored (by the sink).
    - `COMMANDED`: the local SmartData is advertised to the network to be remotely controlled (by the sink). Declaring a local SmartData `COMMANDED` does not implicitly means it is `ADVERTISED`. That must be explicitly declared using (`ADVERTISED` | `COMMANDED`).

Constants

- `UNIT`
  Defines the type of the data produced by the transducer associated with an SmartData instance. It is a numerical representation of a `Unit`. See SI Quantities for additional information.

- `NUM`
  It is only defined for SmartData that encapsulate SI Quantities, case in which it designates how that quantity is encoded. It corresponds to the `NUM` field in `Unit`. See SI Quantities for additional information.

- `ERROR`
  It is only defined for SmartData that encapsulate SI Quantities, case in which it designates the associated transducer's measurement error scale as a magnitude order.

Methods

- `Smart_Data(unsigned int dev, const Microsecond & expiry, const Mode & mode = PRIVATE)`
  Creates a SmartData to abstract the unit `dev` of (local) `Transducer` (a template parameter designating either a Smart Sensor or a Smart Actuator). The data sampled from the transducer is considered valid for `expiry` µs. Accessing the data through `operator Value()` after this time in invalid. The default mode of operation is `PRIVATE`.
  Note: during the application development phase, SmartData can be configured to log access to expired data or even to produce fatal errors when it happens. However, this should never happen in a production system and ensuring this is a design matter.

- `Smart_Data(const Region & region, const Microsecond & expiry, const Microsecond & period = 0)`
  Creates a SmartData to abstract a remote Transducer capable of handling `UNIT` in a region designated by `region` (an sphere of radius *r* centered at (*x*, *y*, *z*) from *t0* until *t1*; see TSTP Coordinates for additional information). The locally stored data received from the transducer is considered valid for `expiry` µs. Accessing the data through `operator Value()` after this time in invalid. If `period` is specified, then transducers matching the selection criteria (`UNIT`, `region`) are instructed to send new data every `period` µs.

- `const Coordinates & location() const`
  Returns the location where the data was produced. Either `TSTP::here()` for local transducers or the coordinates of the remote transducer that produced the data.
  Note: TSTP uses a Cartesian Coordinate System centered at the Sink. This method converts such relative coordinates to an absolute representation centered at the Earth's center whenever the Sink absolute location is known.

- `const Time & time() const`
  Returns the time in which data was produced as an offset in µs from `Traits<RTC>::EPOCH` (usually January 1st, 1970).

4.6.2. Unit

EPOS type **Unit** designates the kind of data encapsulated in a SmartData object, either an SI Quantity or plain Digital Data. Its most significant bit (i.e. bit 31) determines the case: encoded SI Units have it set (i.e. field `SI` = 1), while Digital Data have it clear (i.e. field `SI` = 0)

A Physical Quantity can be identified through the corresponding SI (derived) Unit. Whenever a SmartData encapsulates a physical quantity, such information is encoded in a manner inspired by IEEE 1451 TEDs. Conversely, Digital Data is simply tagged with an application-specific `type` and a `length`. SmartData types are allocated by LISHA on demand and their lengths are type-dependent. The highest significant bit of the `type` field, `multi`, defines if a Digital Unit represents a MultiSmartData, a collection of SmartData that share common characteristics and are merged into a single structure to avoid the replication of metadata. For instance, merging data with the same origin (Multi-Unit SmartData) or same Unit (Multi-Value or Multi-Device SmartData). Most types will be associated with a length expressed in bytes, limiting the SmartData size to 64 KB, but some types will define coarser granularities (currently limited by the IoT database to 2GB per SmartData instance). The `length` field of a digital unit is network protocol dependent, which with TSTP over IEEE 802.15.4 is limited to 81 bytes (this size may vary with network size and lifespan due to the scale of time and space).

Header

`include/smartdata.h`

Digital Data Format

Bit	31	29	16	0
	0	multi	type	length

SI Unit Format

Bit	31	29	27	24	21	18	15	12	9	6	3	0
	1	NUM	MOD	sr+4	rad+4	m+4	kg+4	s+4	A+4	K+4	mol+4	cd+4

Interface

class Unit { public: // Valid values for field SI enum : unsigned long { DIGITAL = 0U << 31, // The Unit is plain digital data. Subsequent 15 bits designate the data type. Lower 16 bits are application-specific, usually a device selector. SI = 1U << 31, // The Unit is SI. Remaining bits are interpreted as specified here. SID = SI }; // Valid values for field NUM enum : unsigned long { I32 = 0 << 29, // Value is an integral int stored in the 32 last significant bits of a 32-bit big-endian integer. I64 = 1 << 29, // Value is an integral int stored in the 64 last significant bits of a 64-bit big-endian integer. F32 = 2 << 29, // Value is a real int stored as an IEEE 754 binary32 big-endian floating point. D64 = 3 << 29, // Value is a real int stored as an IEEE 754 binary64 big-endian double precision floating point. NUM = D64 // AND mask to select NUM bits }; // Valid values for field MOD enum : unsigned long { DIR = 0 << 27, // Unit is described by the product of SI base units raised to the powers recorded in the remaining fields. DIV = 1 << 27, // Unit is U/U, where U is described by the product SI base units raised to the powers recorded in the

remaining fields. LOG = 2 << 27, // Unit is log_e(U), where U is described by the product of SI base units raised to the powers recorded in the remaining fields. LOG_DIV = 3 << 27, // Unit is log_e(U/U), where U is described by the product of SI base units raised to the powers recorded in the remaining fields. MOD = LOG_DIV // AND mask to select MOD bits }; // Masks to select the SI units enum : unsigned long { SR = 7 << 24, RAD = 7 << 21, M = 7 << 18, KG = 7 << 15, S = 7 << 12, A = 7 << 9, K = 7 << 6, MOL = 7 << 3, CD = 7 << 0 }; // Mask to select field LEN of digital data enum : unsigned long { LEN = (1 << 16) - 1 }; // Helper to create digital units template<unsigned int _TYPE, bool _MULTI , unsigned int _SUBTYPE, unsigned int _LEN> class Digital_Unit { public: // DIGITAL | multi | type | length enum : unsigned long { UNIT = DIGITAL | _MULTI << 29 | _TYPE << 24 | _SUBTYPE << 16 | _LEN << 0 }; // LEN field can be an index into a dictionary of accepted lengths for the specific unit private: // Compile-time verifications static const typename IF<(_MULTI & (~1)) , void, bool>::Result Invalid_TYPE = false; static const typename IF<(_TYPE & (~((1 << 6) - 1))), void, bool>::Result Invalid_TYPE = false; static const typename IF<(_SUBTYPE & (~((1 << 8) - 1))), void, bool>::Result Invalid_SUBTYPE = false; static const typename IF<(_LEN & (~LEN)) , void, bool>::Result Invalid_LEN = false; }; // Helper to create SI units template<int _MOD, int _SR, int _RAD, int _M, int _KG, int _S, int _A, int _K, int _MOL, int _CD> class SI_Unit { public: // SI | MOD | sr | rad | m | kg | s | A | K | mol | cd enum : unsigned long { UNIT = SI | _MOD | (4 + _SR) << 24 | (4 + _RAD) << 21 | (4 + _M) << 18 | (4 + _KG) << 15 | (4 + _S) << 12 | (4 + _A) << 9 | (4 + _K) << 6 | (4 + _MOL) << 3 | (4 + _CD) }; private: // Compile-time verifications static const typename IF<(_MOD & (~MOD)), void, bool>::Result Invalid_MOD = false; static const typename IF<((_SR + 4) & (~7u)), void, bool>::Result Invalid_SR = false; static const typename IF<((_RAD + 4) & (~7u)), void, bool>::Result Invalid_RAD = false; static const typename IF<((_M + 4) & (~7u)), void, bool>::Result Invalid_M = false; static const typename IF<((_KG + 4) & (~7u)), void, bool>::Result Invalid_KG = false; static const typename IF<((_S + 4) & (~7u)), void, bool>::Result Invalid_S = false; static const typename IF<((_A + 4) & (~7u)), void, bool>::Result Invalid_A = false; static const typename IF<((_K + 4) & (~7u)), void, bool>::Result Invalid_K = false; static const typename IF<((_MOL + 4) & (~7u)), void, bool>::Result Invalid_MOL = false; static const typename IF<((_CD + 4) & (~7u)), void, bool>::Result Invalid_CD = false; }; // Typical SI Quantities enum Quantity : unsigned long { // mod, sr, rad, m, kg, s, A, K, mol, cd unit | HEX Acceleration = SI_Unit<DIR, +0, +0, +1, +0, -2, +0, +0, +0, +0>::UNIT, // m/s2 | Angle = SI_Unit<DIR, +0, +1, +0, +0, +0, +0, +0, +0, +0>::UNIT, // rad Amount_of_Substance = SI_Unit<DIR, +0, +0, +0, +0, +0, +0, +0, +1, +0>::UNIT, // mol Angular_Velocity = SI_Unit<DIR, +0, +1, +0, +0, -1, +0, +0, +0, +0>::UNIT, // rad/s Area = SI_Unit<DIR, +0, +0, +2, +0, +0, +0, +0, +0, +0>::UNIT, // m2 Current = SI_Unit<DIR, +0, +0, +0, +0, +0, +1, +0, +0, +0>::UNIT, // Ampere Electric_Current = Current, Force = SI_Unit<DIR, +0, +0, +1, +1, -2, +0, +0, +0, +0>::UNIT, // Newton Humidity = SI_Unit<DIR, +0, +0, -3, +1, +0, +0, +0, +0, +0>::UNIT, // kg/m3 Length = SI_Unit<DIR, +0, +0, +1, +0, +0, +0, +0, +0, +0>::UNIT, // m Luminous_Intensity = SI_Unit<DIR, +0, +0, +0, +0, +0, +0, +0, +0, +1>::UNIT, // cd Mass = SI_Unit<DIR, +0, +0, +0, +1, +0, +0, +0, +0, +0>::UNIT, // kg Power = SI_Unit<DIR, +0, +0, +2, +1, -3, +0, +0, +0, +0>::UNIT, // Watt Pressure = SI_Unit<DIR, +0, +0, -1, +1, -2, +0, +0, +0, +0>::UNIT, // Pascal Velocity = SI_Unit<DIR, +0, +0, +1, +0, -1, +0, +0, +0, +0>::UNIT, // m/s Sound_Intensity = SI_Unit<DIR, +0, +0, +0, +1, -3, +0, +0, +0, +0>::UNIT, // W/m2 Temperature = SI_Unit<DIR, +0, +0, +0, +0, +0, +0, +1, +0, +0>::UNIT, // Kelvin Time = SI_Unit<DIR, +0, +0, +0, +0, +1, +0, +0, +0, +0>::UNIT, // s Speed = Velocity, Volume = SI_Unit<DIR, +0, +0, +3, +0, +0, +0, +0, +0, +0>::UNIT, // m3 Voltage = SI_Unit<DIR, +0, +0, +2, +1, -3, -1, +0, +0, +0>::UNIT, // Volt Water_Flow = SI_Unit<DIR, +0, +0, +3, +0, -1, +0, +0, +0, +0>::UNIT, // m3/s Ratio = SI_Unit<LOG_DIV, -4, -4, -4, -4, -4, -4, -4, -4, -4>::UNIT, // not an SI unit Percent = SI_Unit<LOG_DIV, -4, -4, -4, -4, -4, -4, -4, -4, -3>::UNIT, // not an SI unit, a ratio < 1 PPM = SI_Unit<LOG_DIV, -4, -4, -4, -4, -4, -4, -4, -4, -2>::UNIT, // not an SI unit, a ratio in parts per million PPB = SI_Unit<LOG_DIV, -4, -4, -4, -4, -4, -4, -4, -4, -1>::UNIT, // not an SI

unit, a ratio in parts per billion Relative_Humidity = SI_Unit<LOG_DIV, -4, -4, -4, -4, -4, -4, -4, -4, +0>::UNIT, // not an SI unit, a percentage representing the partial pressure of water vapor in the mixture to the equilibrium vapor pressure of water over a flat surface of pure water at a given temperature Power_Factor = SI_Unit<LOG_DIV, -4, -4, -4, -4, -4, -4, -4, -4, +1>::UNIT, // not an SI unit, a ratio of the real power absorbed by the load to the apparent power flowing in the circuit; a dimensionless number in [-1,1] Counter = SI_Unit<LOG_DIV, -4, -4, -4, -4, -4, -4, -4, -4, +2>::UNIT, // not an SI unit, the current value of an external counter Antigravity = SI_Unit<LOG_DIV, +3, +3, +3, +3, +3, +3, +3, +3, +3>::UNIT // for Dummy_Transducer :-) }; // Digital data types enum Digital_Data: unsigned long { // multi, type, subtype, length // Switches Direction = Digital_Unit<0, 0, 1, 1>::UNIT, Switch = Digital_Unit<0, 0, 0, 1>::UNIT, On_Off = Switch, // RFIDs and SmartCartds RFID32 = Digital_Unit<0, 1, 0, 4>::UNIT, // Audio and Video (from RTP) A/V Hz Ch Ref PCMU = Digital_Unit<0, 2, 0, 0>::UNIT, // A 8000 1 [RFC3551] GSM = Digital_Unit<0, 2, 3, 0>::UNIT, // A 8000 1 [RFC3551] G723 = Digital_Unit<0, 2, 4, 0>::UNIT, // A 8000 1 [Vineet_Kumar][RFC3551] DVI4_8 = Digital_Unit<0, 2, 5, 0>::UNIT, // A 8000 1 [RFC3551] DVI4_16 = Digital_Unit<0, 2, 6, 0>::UNIT, // A 16000 1 [RFC3551] LPC = Digital_Unit<0, 2, 7, 0>::UNIT, // A 8000 1 [RFC3551] PCMA = Digital_Unit<0, 2, 8, 0>::UNIT, // A 8000 1 [RFC3551] G722 = Digital_Unit<0, 2, 9, 0>::UNIT, // A 8000 1 [RFC3551] L16_2 = Digital_Unit<0, 2, 10, 0>::UNIT, // A 44100 2 [RFC3551] L16_1 = Digital_Unit<0, 2, 11, 0>::UNIT, // A 44100 1 [RFC3551] QCELP = Digital_Unit<0, 2, 12, 0>::UNIT, // A 8000 1 [RFC3551] CN = Digital_Unit<0, 2, 13, 0>::UNIT, // A 8000 1 [RFC3389] MPA = Digital_Unit<0, 2, 14, 0>::UNIT, // A 90000 [RFC3551][RFC2250] G728 = Digital_Unit<0, 2, 15, 0>::UNIT, // A 8000 1 [RFC3551] DVI4_11 = Digital_Unit<0, 2, 16, 0>::UNIT, // A 11025 1 [Joseph_Di_Pol] DVI4_22 = Digital_Unit<0, 2, 17, 0>::UNIT, // A 22050 1 [Joseph_Di_Pol] G729 = Digital_Unit<0, 2, 18, 0>::UNIT, // A 8000 1 [RFC3551] CelB = Digital_Unit<0, 2, 25, 0>::UNIT, // V 90000 [RFC2029] JPEG = Digital_Unit<0, 2, 26, 0>::UNIT, // V 90000 [RFC2435] nv = Digital_Unit<0, 2, 28, 0>::UNIT, // V 90000 [RFC3551] H261 = Digital_Unit<0, 2, 31, 0>::UNIT, // V 90000 [RFC4587] MPV = Digital_Unit<0, 2, 32, 0>::UNIT, // V 90000 [RFC2250] MP2T = Digital_Unit<0, 2, 33, 0>::UNIT, // AV 90000 [RFC2250] H263 = Digital_Unit<0, 2, 34, 0>::UNIT, // V 90000 [Chunrong_Zhu] WAV_FLAC = Digital_Unit<0, 2, 35, 0>::UNIT, // A 20000 WAV - FLAC compression // MultiUnit SmartData Motion_Vector_Global = Digital_Unit<1, 0, 0, 1>::UNIT, // subtype is object class, LEN > 1 is a list type with LEN elements Motion_Vector_Local = Digital_Unit<1, 1, 0, 1>::UNIT // subtype is object class, LEN > 1 is a list type with LEN elements }; // SI Factors typedef char Factor; enum { // Name Code Symbol Factor ATTO = (8 - 8), // a 0.000000000000000001 FEMTO = (8 - 7), // f 0.000000000000001 PICO = (8 - 6), // p 0.000000000001 NANO = (8 - 5), // n 0.000000001 MICRO = (8 - 4), // µ 0.000001 MILI = (8 - 3), // m 0.001 CENTI = (8 - 2), // c 0.01 DECI = (8 - 1), // d 0.1 NONE = (8), // - 1 DECA = (8 + 1), // da 10 HECTO = (8 + 2), // h 100 KILO = (8 + 3), // k 1000 MEGA = (8 + 4), // M 1000000 GIGA = (8 + 5), // G 1000000000 TERA = (8 + 6), // T 1000000000000 PETA = (8 + 7) // P 1000000000000000 }; template<unsigned long UNIT> struct Get { typedef typename IF<((unsigned long)(UNIT & SID) == SI) && ((unsigned long)(UNIT & NUM) == I32), long int, typename IF<((unsigned long)(UNIT & SID) == SI) && ((unsigned long)(UNIT & NUM) == I64), long long int, typename IF<((unsigned long)(UNIT & SID) == SI) && ((unsigned long)(UNIT & NUM) == F32), float, typename IF<((unsigned long)(UNIT & SID) == SI) && ((unsigned long)(UNIT & NUM) == D64), double, typename IF<((unsigned long)(UNIT & SID) == DIGITAL) && ((unsigned long)(UNIT & LEN) == 1), unsigned char, typename IF<((unsigned long)(UNIT & SID) == DIGITAL) && ((unsigned long)(UNIT & LEN) == 2), unsigned short, typename IF<((unsigned long)(UNIT & SID) == DIGITAL) && ((unsigned long)(UNIT & LEN) == 4), unsigned long, typename IF<((unsigned long)(UNIT & SID) == DIGITAL) && ((unsigned long)(UNIT & LEN) > 4), unsigned char*, //[UNIT & LEN] void>::Result>::Result>::Result>::Result>::Result>::Result>::Result Type; }; template<typename T> struct GET; template<unsigned long U> struct Wrap { enum : unsigned

long { UNIT = U }; }; public: Unit(): _unit(0) {} Unit(unsigned long u) { _unit = u; } operator unsigned long() const { return _unit; } unsigned int value_size() const { return (_unit & SI) && ((_unit & NUM) == I32) ? sizeof(long int) : (_unit & SI) && ((_unit & NUM) == I64) ? sizeof(long long int) : (_unit & SI) && ((_unit & NUM) == F32) ? sizeof(float) : (_unit & SI) && ((_unit & NUM) == D64) ? sizeof(double) : !(_unit & SI) ? _unit & LEN : 0; } static unsigned int value_size(unsigned long unit) { return (unit & SI) && ((unit & NUM) == I32) ? sizeof(long int) : (unit & SI) && ((unit & NUM) == I64) ? sizeof(long long int) : (unit & SI) && ((unit & NUM) == F32) ? sizeof(float) : (unit & SI) && ((unit & NUM) == D64) ? sizeof(double) : !(unit & SI) ? unit & LEN : 0; } int sr() const { return ((_unit & SR) >> 24) - 4 ; } int rad() const { return ((_unit & RAD) >> 21) - 4 ; } int m() const { return ((_unit & M) >> 18) - 4 ; } int kg() const { return ((_unit & KG) >> 15) - 4 ; } int s() const { return ((_unit & S) >> 12) - 4 ; } int a() const { return ((_unit & A) >> 9) - 4 ; } int k() const { return ((_unit & K) >> 6) - 4 ; } int mol() const { return ((_unit & MOL) >> 3) - 4 ; } int cd() const { return ((_unit & CD) >> 0) - 4 ; } } __attribute__((packed));

Constants

- `NUM` field
  - `I32`: quantity is encoded as a 32-bit, little-endian, integral number.
  - `I64`: quantity is encoded as a 64-bit, little-endian, integral number.
  - `F32`: quantity is encoded as a 32-bit, little-endian, IEEE 754 binary32, floating point number.
  - `D64`: quantity is encoded as a 64-bit, little-endian, IEEE 754 binary64, floating point number.

- `MOD` field
  - `DIR`: unit is directly described by the product of SI base units raised to the powers recorded in the remaining fields.
  - `DIV`: unit is U/U, where U is described by the product SI base units raised to the powers recorded in the remaining fields.
  - `LOG`: unit is $\log_e(U)$, where U is described by the product of SI base units raised to the powers recorded in the remaining fields.
  - `LOG_DIV`: unit is $\log_e(U/U)$, where U is described by the product of SI base units raised to the powers recorded in the remaining fields.

- Basic Units fields (encoded as 4 + exponent, with exponent ranging from -4 to +3)
  - `SR`: exponent of the Steradian component of the SI derived unit.
  - `RAD`: exponent of the Radian component of the SI derived unit.
  - `M`: exponent of the Meter component of the SI derived unit.
  - `KG`: exponent of the Kilogram component of the SI derived unit.
  - `S`: exponent of the Second component of the SI derived unit.
  - `A`: exponent of the Ampere component of the SI derived unit.
  - `K`: exponent of the Kelvin component of the SI derived unit.
  - `MOL`: exponent of the Mole component of the SI derived unit.
  - `CD`: exponent of the Candela component of the SI derived unit.

- Typical SI Derived Units as a function of Basic Units fields
  A set of constants designating typical derivations from SI Basic Units is provided here.

- Counts (Not SI UNITS)
  Counts, even parts per million and percentages, do not fit the (type, subtype, length) idea of Digital UNITS. Therefore, they are modeled as SI UNITS. They are:
  - Ratio;

    - Percent, a ratio < 1;
    - Parts per Milion (PPM), a ratio in parts per million, and Parts per Billion (PPB), a ratio in parts per billion;
    - Relative_Humidity, a percentage representing the partial pressure of water vapor in the mixture to the equilibrium vapor pressure of water over a flat surface of pure water at a given temperature;
    - Power_Factor, a ratio of the real power absorbed by the load to the apparent power flowing in the circuit; a dimensionless number in the range [-1,1];
    - Counter, the current value of an external counter;

- MultiUnits
  AV MultiUnits are defined at the LISHA's AV SmartData Model.

- SI Unit Prefixes
  A set of constants designating the SI Unit Prefixes is provided.

Metaprograms

- `Get<int N>::Type`
  Returns the C++ native type `Value` is aliased to:
    - `I32`: `signed long int`;
    - `I64`: `signed long long int`;
    - `F32`: `float`;
    - `D64`: `double`.

- `GET<typename T>::NUM`
  Returns the `NUM` field associated with `T`:
    - `double`: `D64`.
    - `float`: `F32`.
    - `long long int`: `I64`.
    - otherwise: `I32`.

Methods

- Interoperability with `unsigned long`
  A constructor and a conversion operator are provided so that `Unit` can be used as if it were an ordinary `unsigned long`.

- Basic SI Unit exponent extraction
  Methods are provided to get the exponent for each of the SI Basic Units in `Unit`.

Examples

4.6.3. Persistent Storage

Whenever a piece of **SmartData** is stored in a database, file system, or any sort of persistent memory that can be externally accessed, a canonical format is used.

Header

`include/smartdata.h`

Interface

- **STATIC**

```
template<typename Transducer> class Smart_Data: private TSTP::Observer, private Transducer::Observer { public: struct DB_Record { unsigned char type; unsigned long unit; double value; unsigned char error; unsigned char confidence; long x; long y; long z; long device; unsigned long long t; }; struct DB_Series { unsigned char type; unsigned long unit; long x; long y; long z; long device; unsigned long r; unsigned long long t0; unsigned long long t1; }; };
```

- **MOBILE**

```
template<typename Transducer> class Smart_Data: private TSTP::Observer, private Transducer::Observer { public: struct DB_Record { unsigned char type; unsigned long unit; Value value; unsigned char error; unsigned char confidence; unsigned char[8] ID; float longitude; float latitude; float altitude; long device; unsigned long long t; }; struct DB_Series { unsigned char type; unsigned long unit; unsigned char[8] ID; long device; unsigned long long t0; unsigned long long t1; }; };
```

  - `Value`: The type `Value` is dependent on the unit specification as previously described.
  - `ID`: a 64-bit cryptographic identifier resultant of a hash of the Public certificate of a mobile device producing SmartData. During execution, the mobile version of SmartData comprises an origin based on ID and time instead of Space and Time. For Persistent storage, a source of position is assumed to be available at the mobile device to compose the DB_Record. However, a DB_Series will use the origin information and base coordinates on the DB_Records for the specific ID and time range (t0 until t1) and the radio range of the object.
  - `longitude`, `latitude`, `altitude`: For local management of SmartData in a mobile device, such as an Autonomous Vehicles (AV), several objects identified via vision perception are represented with coordinates in relation to the AV current position, with the absence of the Z coordinate (altitude). Therefore, to avoid errors in local processing due to converting latitude and longitude to ECEF coordinates, position is handled locally in latitude and longitude represented in radians SI Quantity. Nevertheless, when data is communicated in V2X or forwarded to persistent storage, a transformation is applied to the coordinates based on the vehicle coordinates (which comprise the altitude) to obtain a complete perception of longitude, latitude, and altitude.

Types

- `DB_Record`
  Defines an interoperable format for the content of a SmartData representing Digital Data or an SI Quantity, according to `unit` value. The format is used for both, storing and transmission using non-native protocols. See IoT+with+EPOS for additional information.

- `DB_Series`
  Defines an interoperable format to designate time-series of Smart Data stored in a database or streamed using non-native protocols. See IoT+with+EPOS for additional information.

Methods

- `DB_Record db_record() const`
  Returns an `DB_Record` representing the content of this SmartData in an interoperable format.

- `DB_Series db_series() const`
  Returns a `Series` associated with this SmartData in an interoperable format.

4.6.4. Transducers

A Transducer class interfaces a hardware mediator for a transducer (a sensor and/or actuator) with a SmartData instance. Some Transducers may require the user to call the constructor for hardware initialization and binding. Besides that, the application should only use SmartData objects, and **not** the Transducers directly. Each specific sensor has a transducer class, and we show a simple example below. Consult the header file for all available transducers and their particular implementations.

Header

`include/machine/<machine>/transducer.h`

Interface

class Transducer: public Transducer_Hardware_Mediator { public: static const unsigned int UNIT; static const unsigned int NUM; static const int ERROR; static const bool INTERRUPT; static const bool POLLING; typedef Transducer_Hardware_Mediator::Observer Observer; typedef Transducer_Hardware_Mediator::Observed Observed; public: Transducer(); static void sense(unsigned int dev, Smart_Data<Transducer> * data); static void actuate(unsigned int dev, Smart_Data<Transducer> * data, const Smart_Data<Transducer>::Value & command); }; typedef Smart_Data<Transducer> Smart_Transducer;

Types

- `Observer`
  A redefinition of the mediator's Observer type. Only present if the mediator is observable in an event-driven scheme (i.e. when `INTERRUPT = true`).

- `Observed`
  A redefinition of the mediator's Observed type. Only present if the mediator is observable in an event-driven scheme (i.e. when `INTERRUPT = true`).

- `Smart_Transducer`
  At the end of the transducer header file, there are definitions for all available SmartData for the corresponding machine, with appropriate Transducer type bindings. These are the classes that the application should use.

Constants

- `UNIT`
  Defines the type of data produced by the sensor associated with this Transducer. It is a numerical representation of a `Unit`. See SI Quantities for additional information.

- `NUM`
  It is only defined for Transducers that encapsulate SI Quantities, case in which it designates

how that quantity is encoded. It corresponds to the `NUM` field in `Unit`. See SI Quantities for additional information.

- `ERROR`
  It is only defined for Transducers that encapsulate SI Quantities, case in which it designates the associated transducer's measurement error scale as a magnitude order.

- `INTERRUPT`
  Whether this transducer is observable in an *event-driven* way. If true, the SmartData interfacing with this transducer will call its `attach` method during construction, so that it can be notified whenever a new value is available from the sensor and call its `sense` method to get that value (See Observer for details on observers).

- `POLLING`
  Whether this transducer is observable in a *time-triggered* way. If true, the SmartData interfacing with this transducer may call its `sense` method whenever it needs a new sensor reading.

Methods

- `Transducer()`
  Some transducers require the application to call their constructors in order to initialize the corresponding hardware mediator and bind it to a `dev` number known to a SmartData instance. Consult the actual implementation you are using for details.

- `static void sense(unsigned int dev, Smart_Data<Transducer> * data)`
  This method is called by a SmartData instance when it needs to get a new reading from the sensor when this transducer is capable of sensing. This method implements the actual hardware reading, usually by forwarding it to the base mediator class, and assigns the results to `*data`. SmartData encapsulates all the protocol interactions and decisions regarding to when this method should be called. The `dev` parameter is used to distinguish between multiple sensors of the same kind, and it is defined by the user and passed to the SmartData constructor. Some transducers require the same `dev` number to be passed at the constructor for correct binding. Consult the actual implementation you are using for details.

- `static void actuate(unsigned int dev, Smart_Data<Transducer> * data, const Smart_Data<Transducer>::Value & command)`
  Similar to `sense`. The value `command` should be written to the hardware mediator, when this Transducer is capable of actuating.

Examples

```
// SmartData Declarations typedef Smart_Data<Accelerometer> Acceleration; typedef Smart_Data<Voltmeter> Voltage; typedef Smart_Data<Thermometer> Temperature; typedef Smart_Data<Photometer> Illuminace; // SmartData Usage // Local acceleration data from accelerometer "0" // with expiration time of "expiry" µs. Acceleration a(0, expiry); cout << "The acceleration here is" << a << "m/s^2." << endl; // Remote temperature in Kelvin from a region centered at (x, y, z), with radius "r", // from time "t0" until time "t1", updated every "period" µs // with expiration time of "expiry" µs. Temperature k(Region(Coordinates(x, y, z), r, t0, t1), expiry, period); for(Time t = TSTP::now(); t < t1; t = TSTP::now()) { cout << "The temperature there is "
```

<< k << "K." << endl; Delay(period); }

4.7. Utilities

EPOS provides a set of **Utility Classes** that can be used for both application and system development. Although of far more limited scope, programmers can take them as EPOS counterpart to *libc* and *libstdc++*.

4.7.1. Containers

Operating systems spend most of their CPU time managing lists. Processes, resources, buffers, and virtually any other object in the system are kept in and moved across lists. Therefore, EPOS Lists have been carefully designed for efficiency. Although similar to the C++ Standard Library Lists, they have a key difference: objects subject to list insertion and removal must contain a linkage data structure (viz. `Element`) within themselves. In this way, EPOS Lists do not waste time with memory allocation and deallocation of such operations. Objects must be aware of how many lists can contain them at the same time and declared the necessary number of linkage data structures.

EPOS provides the following containers: Vector, List, Hash Table, Queue, Bitmap, and Zero-Copy Buffers. They are build atop 4 basic types of Lists, each implemented both as a single-linked and as double-linked. Single-linked Lists are prefixed with `Simple_`. Singly-linked lists require less memory, but depend on sequential search operations. Doubly-linked ones require more memory, but support removal (and other operations) from arbitrary positions. The four basic types are: ordinary, ordered, relatively ordered, and grouping.

Ordered containers are kept ordered by a `Rank`. Types acting as Rank must either declare `operator int()` or declare the full set of logic operators. Relatively ordered containers are also kept ordered by `Rank`, but ranks are interpreted as offsets from/to neighboring elements. Operations ensure that such relative ranks are properly adjusted whenever an element is inserted into or removed from a relatively ordered container. EPOS also provides a Grouping container that implements the Buddy algorithm. It is mostly used to implement memory allocators.

Besides basic containers, EPOS also provides a powerful Scheduling List framework.

4.7.1.1. Linkage Elements and Ranks

Header
`include/utility/list.h`

Interface

class List_Element_Rank { public: List_Element_Rank(int r = 0); operator int(); }; namespace List_Elements { typedef List_Element_Rank Rank; // Vector Element template<typename T> class Pointer { public: typedef T Object_Type; typedef Pointer Element; public: Pointer(const T * o); T * object(); }; // Hash Table Element template<typename T, typename R = Rank> class Ranked { public: typedef T Object_Type; typedef R Rank_Type; typedef Ranked Element; public: Ranked(const T * o, const R & r = 0); T * object(); const R & rank(); const R & key(); void rank(const R & r); int promote(const R & n = 1) ; int demote(const R & n = 1); }; // Simple List Element template<typename T> class Singly_Linked { public: typedef T Object_Type; typedef Singly_Linked Element; public: Singly_Linked(const T * o); T * object(); Element * next(); void next(Element * e); }; // Simple Ordered List Element // Hash Table's Synonym List Element template<typename T, typename R = Rank> class Singly_Linked_Ordered { public: typedef T Object_Type; typedef Rank Rank_Type; typedef Singly_Linked_Ordered Element; public:

Singly_Linked_Ordered(const T * o, const R & r = 0); T * object() const ; Element * next(); void next(Element * e); const R & rank(); const R & key(); void rank(const R & r); int promote(const R & n = 1); int demote(const R & n = 1); }; // Simple Grouping List Element template<typename T> class Singly_Linked_Grouping { public: typedef T Object_Type; typedef Singly_Linked_Grouping Element; public: Singly_Linked_Grouping(const T * o, int s); T * object(); Element * next(); void next(Element * e); unsigned int size(); void size(unsigned int l); void shrink(unsigned int n); void expand(unsigned int n); }; // List Element template<typename T> class Doubly_Linked { public: typedef T Object_Type; typedef Doubly_Linked Element; public: Doubly_Linked(const T * o); T * object(); Element * prev(); Element * next(); void prev(Element * e); void next(Element * e); }; // Ordered List Element template<typename T, typename R = Rank> class Doubly_Linked_Ordered { public: typedef T Object_Type; typedef Rank Rank_Type; typedef Doubly_Linked_Ordered Element; public: Doubly_Linked_Ordered(const T * o, const R & r = 0); T * object(); Element * prev(); Element * next(); void prev(Element * e); void next(Element * e); const R & rank(); void rank(const R & r); int promote(const R & n = 1); int demote(const R & n = 1); }; // Scheduling List Element template<typename T, typename R = Rank> class Doubly_Linked_Scheduling { public: typedef T Object_Type; typedef Rank Rank_Type; typedef Doubly_Linked_Scheduling Element; public: Doubly_Linked_Scheduling(const T * o, const R & r = 0); T * object(); Element * prev(); Element * next(); void prev(Element * e); void next(Element * e); const R & rank(); void rank(const R & r); int promote(const R & n = 1); int demote(const R & n = 1); }; // Grouping List Element template<typename T> class Doubly_Linked_Grouping { public: typedef T Object_Type; typedef Doubly_Linked_Grouping Element; public: Doubly_Linked_Grouping(const T * o, int s); T * object(); Element * prev(); Element * next(); void prev(Element * e); void next(Element * e); unsigned int size(); void size(unsigned int l); void shrink(unsigned int n); void expand(unsigned int n); }; };

Types

- `List_Element_Rank`
  The basic Rank type for ordered containers. It declares a constructor and `operator int()` to become interoperable with the native C++ type. Customized rank types can either follow this approach or must define the full set of logic operators for sorting operations.

- Linkage Elements
  - `template<typename T>`
    `class Pointer`
    Linkage Element for `Vector`.

  - `template<typename T, typename R = Rank>`
    `class Ranked`
    Linkage Element for `Hash`.

  - `template<typename T, typename R = Rank>`
    `class Singly_Linked_Ordered`
    Linkage Element for `Simple_Ordered_List`. It is also used as element in Hash Table's synonyms list.

  - `template<typename T>`
    `class Singly_Linked_Grouping`
    Linkage Element for `Simple_Grouping_List`.

  - `template<typename T>`
    `class Doubly_Linked`
    Linkage Element for `List`.

- `template<typename T, typename R = Rank>`
  `class Doubly_Linked_Ordered`
  Linkage Element for `Ordered_List`.

- `template<typename T>`
  `class Doubly_Linked_Grouping`
  Linkage Element for `Grouping_List`.

- `template<typename T, typename R = Rank>`
  `class Doubly_Linked_Scheduling`
  Linkage Element for `Scheduling_List`.
- Common Type Exports
  - `Object_Type`
    An alias for the type of the object associated with the Element.
  - `Rank_Type`
    An alias for the type of the Rank of the object associated with the Element. It is only defined for Ordered containers.
  - `Element`
    An alias for the type of the Element.

Methods

- `T * object()`
  Returns a pointer to the object associated with the Element.

- `Element * prev()`
  Returns a pointer to the previous Element linked with the Element or 0 if it is the Head. It is only defined for doubly-linked containers.

- `void prev(Element * e)`
  Sets the previous link in the Element to `e`. It is only defined for doubly-linked containers.

- `Element * next()`
  Returns a pointer to the next Element linked with the Element or 0 if it is the Tail.

- `void next(Element * e)`
  Sets the next link in the Element to `e`.

- `const R & rank()`
  For ordered containers, returns the Element's Rank.

- `void rank(const R & r)`
  For ordered containers, sets the Element's Rank. It does not reorder the container, though. This method is meant to be called by the sorting algorithms during reordering.

- `int promote(const R & n = 1)`
  For ordered containers, increments the Element's Rank by `n`. It does not reorder the container, though. This method is meant to be called by the sorting algorithms during reordering.

- `int demote(const R & n = 1)`
  For ordered containers, decrements the Element's Rank by `n`. It does not reorder the container, though. This method is meant to be called by the sorting algorithms during reordering.

- `unsigned int size()`

Only defined for Grouping Lists, returns the size of the resource set associated with the Element.

- `void size(unsigned int l)`
  Only defined for Grouping Lists, sets the size of the resource set associated with the Element.
- `void shrink(unsigned int n)`
  Only defined for Grouping Lists, decrements the size of the resource set associated with the Element by `n`.
- `void expand(unsigned int n)`
  Only defined for Grouping Lists, increments the size of the resource set associated with the Element by `n`.

4.7.1.2. Iterators

The following Iterators are common to all EPOS containers. They can be used mostly like those in the C++ Standard Library.

Header

`include/utility/list.h`

Interface

```
namespace List_Iterators { // Forward Iterator (for singly linked lists) template<typename El> class Forward { public: typedef El Element; public: Forward(); Forward(Element * e); operator Element *(); Element & operator*(); Element * operator->(); Iterator & operator++(); Iterator operator++(int); bool operator==(const Iterator & i); bool operator!=(const Iterator & i); }; // Bidirectional Iterator (for doubly linked lists) template<typename El> class Bidirecional { public: typedef El Element; public: Bidirecional(); Bidirecional(Element * e); operator Element *(); Element & operator*(); Element * operator->(); Iterator & operator++(); Iterator operator++(int); Iterator & operator--(); Iterator operator--(int) bool operator==(const Iterator & i); bool operator!=(const Iterator & i); }; }
```

Types

- `Forward`
  An Iterator for singly-linked containers.
- `Forward`
  An Iterator for doubly-linked containers.

4.7.1.3. Vector

EPOS provides a **Vector** container similar to that in the C++ Standard Library.

Header

`include/utility/vector.h`

Interface

```
template<typename T, unsigned int SIZE, typename El = List_Elements::Pointer<T> > class
```

Vector { public: typedef T Object_Type; typedef El Element; public: Vector(); bool empty(); unsigned int size(); Element * operator[](unsigned int i); bool insert(Element * e, unsigned int i); Element * remove(unsigned int i); Element * remove(Element * e); Element * remove(const Object_Type * obj); Element * search(const Object_Type * obj) ; };

Methods

- `Vector()`
  Creates a vector.
- `bool empty()`
  Returns true if the vector is empty and false otherwise.
- `unsigned int size()`
  Returns the number of elements in the vector.
- `Element * get(int i)`
  Returns a pointer to the element stored at position `i` in the vector.
- `bool insert(Element * e, unsigned int i)`
  Inserts element `e` in the vector at position `i`. If the position was already occupied, returns false. Otherwise, returns true.
- `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 element `e` from the vector and returns a pointer to it, or returns 0 if the element is not in the vector.
- `Element * remove(const Object_Type * obj)`
  Searches the vector for an element containing the object pointed by `obj`. If found, removes that element from the vector and returns a pointer to it, otherwise it returns 0.
- `Element * search(const Object_Type * obj)`
  Searches the vector for an element containing the object pointed by `obj`. If found, returns a pointer to it, otherwise returns 0.

Examples

□□□□□□□□

4.7.1.4. Lists

EPOS provides 9 implementations of list: ordinary, ordered, relatively ordered, grouping, and scheduling. The first 4 are provided both as singly-linked and as doubly-linked. The scheduling list is only provided as doubly-linked. Singly-linked lists are prefixed with `Simple_`, define a `Forward` Iterator and by default use a `Singly_Linked`linkage element. Ordered lists are kept ordered by a Rank. Relatively ordered list elements have their ranks interpreted as offsets from/to neighboring elements. Operations ensure that such relative ranks are properly adjusted whenever an element is inserted into or removed from a relatively ordered container. Grouping lists implement the Buddy algorithm and are mostly used to implement memory allocators.

Header

`include/utility/list.h`

Interface

template<typename T, typename El = List_Elements::Doubly_Linked<T> > class List { public: typedef T Object_Type; typedef El Element; typedef List_Iterators::Bidirecional<El> Iterator; public: List(); bool empty(); unsigned int size(); Element * head(); Element * tail(); Iterator begin(); Iterator end(); void insert(Element * e) ; void insert_head(Element * e); void insert_tail(Element * e); Element * remove(); Element * remove(Element * e); Element * remove_head(); Element * remove_tail(); Element * remove(const Object_Type * obj); Element * search(const Object_Type * obj); }; template<typename T, typename R = List_Element_Rank, typename El = List_Elements::Doubly_Linked_Ordered<T, R>, bool relative = false> class Ordered_List: public List<T, El> { public: typedef T Object_Type; typedef R Rank_Type; typedef El Element; typedef List_Iterators::Bidirecional<El> Iterator; public: Ordered_List(); using Base::empty; using Base::size; using Base::head; using Base::tail; using Base::begin; using Base::end; void insert(Element * e); Element * remove(); Element * remove(Element * e); using Base::remove_head; using Base::remove_tail; Element * remove(const Object_Type * obj); Element * remove_rank(const Rank_Type & rank) ; using Base::search; Element * search_rank(const Rank_Type & rank); }; template<typename T, typename R = List_Element_Rank, typename El = List_Elements::Doubly_Linked_Ordered<T, R> > class Simple_Relative_List: public Ordered_List<T, R, El, true> {}; template<typename T, typename El = List_Elements::Doubly_Linked_Ordered<T> > class Grouping_List: public List<T, El> { public: typedef T Object_Type; typedef El Element; typedef List_Iterators::Bidirecional<El> Iterator; public: Grouping_List(); using Base::empty; using Base::size; using Base::head; using Base::tail; using Base::begin; using Base::end; unsigned int grouped_size(); void insert_merging(Element * e, Element ** m1, Element ** m2); Element * search_size(unsigned int s); Element * search_left(const Object_Type * obj) ; Element * search_decrementing(unsigned int s) ; }; template<typename T, typename R = typename T::Criterion, typename El = List_Elements::Doubly_Linked_Scheduling<T, R> > class Scheduling_List: private Ordered_List<T, R, El> { public: typedef T Object_Type; typedef R Rank_Type; typedef El Element; typedef typename Base::Iterator Iterator; public: Scheduling_List(); using Base::empty; using Base::size; using Base::head; using Base::tail; using Base::begin; using Base::end; Element * volatile & chosen(); void insert(Element * e) ; Element * remove(Element * e); Element * choose() ; Element * choose_another(); Element * choose(Element * e) ; };

Types

- `template<typename T, typename El>`
  `List`
  A doubly-linked List of objects to type `T`, which are linked using `El`.

- `template<typename T, typename R, typename El>`
  `Ordered_List`
  A doubly-linked Ordered List of objects of type `T`, which are ranked by `R` and linked using `El`.

- `template<typename T, typename R, typename El>`
  `Relative_List`
  A doubly-linked Relatively Ordered List of objects of type `T`, which are ranked by `R` and linked using `El`.

- `template<typename T, typename El>`
- `Grouping_List`
  A doubly-linked Grouping (Buddy) List of objects to type `T`, which are linked using `El`.

- `template<typename T, typename R, typename El>`
  `Scheduling_List`
  A doubly-linked Scheduling List of objects of type `T`, which are ranked by `R` and linked using `El`. Objects subject to scheduling must export a type +-Criterion+- compatible with those described in section Scheduler.

- `template<typename T, typename El>`
  `Simple_List`
  A singly-linked List of objects to type `T`, which are linked using `El`.

- `template<typename T, typename R, typename El>`
  `Simple_Ordered_List`
  A singly-linked Ordered List of objects of type `T`, which are ranked by `R` and linked using `El`.

- `template<typename T, typename R, typename El>`
  `Simple_Relative_List`
  A singly-linked Relatively Ordered List of objects of type `T`, which are ranked by `R` and linked using `El`.

- `template<typename T, typename El>`
- `Simple_Grouping_List`
  A singly-linked Grouping (Buddy) List of objects to type `T`, which are linked using `El`.

- Common Type Exports
  - `Object_Type`
    An alias for the type of the objects stored in the container.
  - `Rank_Type`
    An alias for the type of the Rank of the objects stored in the container. It is only defined for Ordered containers.
  - `Element`
    An alias for the container's Element.
  - `Iterator`
    An alias for the container's Iterator.

Methods

- `List()`
  `Ordered_List()`
  `Relative_List()`
  `Grouping_List()`
  `Scheduling_List()`
  `Simple_List()`
  `Simple_Ordered_List()`
  `Simple_Relative_List()`
  `Simple_Grouping_List()`
  Creates a list.

- `bool empty()`
  Returns true if the list is empty and false otherwise.

- `unsigned int size()`
  Returns the number of elements in 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 the list.

- `Iterator end()`
  Returns an iterator to the last element of the list.

- `void insert(Element * e)`
  Inserts element `e` in the list. For unordered lists, insertion is performed at the tail. For ordered lists, the position is determined by `e->rank()`. The method is not defined for grouping lists.

- `void insert_head(Element * e)`
  Inserts element `e` in the list's head. It is not defined for ordered and grouping lists.

- `void insert_tail(Element * e)`
  Inserts element `e` in the list's tail. It is not defined for ordered and grouping lists.

- `Element * remove()`
  Removes the element at the list's head and returns a pointer to it. If the list is empty, returns 0. It is not defined for grouping lists.

- `Element * remove(Element * e)`
  Removes element `e` from the list and returns a pointer to it. For relatively ordered lists, the ranks of neighbor elements are adjusted accordingly.
  Note: removing an Element that is not in the list with this method will probably corrupt the last container it was on. This is a fast method to be used inside the OS. Applications will more likely use `Element * remove(const Object_Type * obj)`.

- `Element * remove_head()`
  Removes the element at the list's head and returns a pointer to it. If the list is empty, returns 0. It is not defined for grouping lists.

- `Element * remove_tail()`
  Removes the element at the list's tail and returns a pointer to it. If the list is empty, returns 0. It is not defined for grouping lists.

- `Element * remove(const Object_Type * obj)`
  Searches the list for an element containing the object pointed by `obj`. If found, removes that element from the list and returns a pointer to it, otherwise returns 0. For relatively ordered lists, the ranks of neighbor elements are adjusted accordingly.
  Note: trying to remove an object that is not in the list with this method is harmless; 0 is returned in this case.

- `Element * remove_rank(int rank)`
  Searches the list for the first element whose rank is `rank`. If found, removes that element from the list and returns a pointer to it, otherwise returns 0. For relatively ordered lists, the ranks of

neighbor elements are adjusted accordingly.
Note: trying to remove an object that is not in the list with this method is harmless; 0 is returned in this case.

- `Element * search(const Object_Type * obj)`
  Searches the list for an element containing the object pointed by `obj`. If found, returns a pointer to it, otherwise returns 0.

- `Element * search_rank(int rank)`
  Returns a pointer to the first element in the list whose rank is `rank` or 0 if there is no element in the list with that rank. This method is only defined for ordered lists.

- `unsigned int grouped_size()`
  For grouping lists, returns the sum of all the resource sets stored in the list, that is, the sum of the return value of method `size()` applied to each element in the grouping list. This method is not defined for other kinds of list.

- `Element * search_size(unsigned int s)`
  For grouping lists, searches for the first element in the list whose size is equal to or larger than `s`. If found, returns a pointer to it, otherwise returns 0. This method is not defined for other kinds of list.

- `void insert_merging(Element * e, Element ** m1, Element ** m2 )`
  Inserts element `e` in the grouping list. If the insertion does not cause mergers, then output parameters `m1` and `m2` are set to 0. Conversely, if the insertion causes a merger with an adjacent element, that element is removed from the list and its *size* is incorporated by `e`. On the adjacency with a preceding element (i.e. an element whose object pointer is less than `e->object()) , -+m1` is updated with a pointer to that element and the object pointer in the element being inserted is adjusted accordingly (-+e->object(m1->object())+-). On the adjacency with a following element, `m2` is updated with a pointer to that element. This method is not defined for other kinds of list.
  Note: if `m1` and `m2` were dynamically allocated somewhere else, deleting them is up to who allocated them.

- `Element * search_decrementing(unsigned int s)`
  For grouping lists, searches for the first element in the list whose size is equal to or larger than `s`. If found, returns a pointer to it and decrements its size by `s`, otherwise returns 0. This method is not defined for other kinds of list.
  Note: for performance reasons, this method uses *first-fit*, while the traditional Buddy Allocator uses *best-fit*.

- `Element * volatile & chosen()`
  For scheduling lists, returns a reference to a volatile pointer to the element currently chosen. This method is not defined for other kinds of list.

- `Element * choose()`
  For scheduling lists, applies the `Criterion` in force (see Scheduler) to select an element that will figure as the new *chosen* and returns a pointer to that element. This method is not defined for other kinds of list.

- `Element * choose_another()`
  For scheduling lists, applies the `Criterion` in force (see Scheduler) to select an element that will figure as the new *chosen* and returns a pointer to that element. The element currently

chosen is excluded from the selection, so even if the criteria would elect it, another element will be returned. This method is not defined for other kinds of list.

- `Element * choose(Element * e)`
  For scheduling lists, ignores the `Criterion` in force (see Scheduler) and select `e` as the new *chosen*. The method returns a pointer to the new *chosen* element (most likely `e`). This method is not defined for other kinds of list.

4.7.1.5. Queue

A **Queue** is just a wrapper to a **List** that is able to make the operation on that List atomic through the use of a Spin Lock. Its interface is that of a `Scheduling_List`.

Examples

4.7.1.6. Hash

EPOS provides two implementations of Hast Tables. The first, **Simple Hash**, handles *collisions* by putting all *synonyms* in the same singly-linked ordered list. That is, it is implemented with a Vector plus a List of synonyms. The second, named just **Hash**, handles *collisions* by putting *synonyms* on separate lists, one for each Key. It is implemented as a Vector of Lists. The type used as **Key** is required to implement `operator%()`.

Header

`include/hash.h`

Interface

template<typename T, unsigned int SIZE, typename Key = int> class Simple_Hash { public: typedef T Object_Type; typedef Key Rank_Type; typedef typename List_Elements::Singly_Linked_Ordered<T, Key> Element; class Forward; typedef Forward Iterator; public: Simple_Hash(); Iterator begin(); Iterator end(); bool empty(); unsigned int size(); void insert(Element * e) ; Element * remove(Element * e); Element * remove(const Object_Type * obj); Element * remove_key(const Key & key); Element * search(const Object_Type * obj) ; Element * search_key(const Key & key); }; template<typename T, unsigned int SIZE, typename Key = int, typename El = List_Elements::Singly_Linked_Ordered<T, Key>, typename L = Simple_Ordered_List<T, Key, El> > class Hash { public: typedef T Object_Type; typedef El Element; typedef L List; public: Hash(); Iterator begin(); Iterator end(); bool empty(); unsigned int size(); void insert(Element * e) ; Element * remove(Element * e) ; Element * remove(const Object_Type * obj); Element * remove_key(const Key & key); Element * search(const Object_Type * obj); Element * search_key(const Key & key); List * operator[](const Key & key) ; };

Methods

- `Hash()`
  `Simple_Hash()`
  Creates a hash table.

- `Iterator begin()`
  Returns an iterator to the first element in the hash table.

- `Iterator end()`

Returns an iterator to the last element in the hash table.

- `bool empty()`
  Returns true if the hash table is empty and false otherwise.

- `unsigned int size()`
  Returns the number of elements in the hash table.

- `void insert(Element * e)`
  Inserts element `e` in the hash table.

- `Element * remove(Element * e)`
  Removes element `e` from the hash table and returns a pointer to it.
  Note: removing an Element that is not in the table with this method will probably corrupt the last container it was on. This is a fast method to be used inside the OS. Applications will more likely use `Element * remove(const Object_Type * obj)`.

- `Element * remove(const Object_Type * obj)`
  Searches the hash table for an element containing the object pointed by `obj`. If found, removes that element from the table and returns a pointer to it, otherwise returns 0.
  Note: trying to remove an object that is not in the list with this method is harmless; 0 is returned in this case.

- `Element * remove_key(const Key & key)`
  Searches the hash table for the first element whose key is `key`. If found, removes that element from the table and returns a pointer to it, otherwise returns 0.
  Note: trying to remove an object that is not in the table with this method is harmless; 0 is returned in this case.

- `Element * search(const Object_Type * obj)`
  Searches the hash table for an element containing the object pointed by `obj`. If found, returns a pointer to it, otherwise returns 0.

- `Element * search_key(const Key & key)`
  Returns a pointer to the first element in the hash table whose key is `key` or 0 if there is no element in the table by that key.

Examples

4.7.2. OStream

EPOS provides an Output Stream similar to that in the C++ Standard Library. Applications can print formatted data on the standard output stream using `operator<<()`.

Header

`include/utility/ostream.h`

Interface

```
class OStream { public: struct Begl {}; struct Endl {}; struct Hex {}; struct Dec {}; struct Oct
```

{}; struct Bin {}; struct Err {}; public: OStream(); OStream & operator<<(const Begl & begl); OStream & operator<<(const Endl & endl); OStream & operator<<(const Hex & hex); OStream & operator<<(const Dec & dec); OStream & operator<<(const Oct & oct); OStream & operator<<(const Bin & bin); OStream & operator<<(const Err & err); OStream & operator<<(char c); OStream & operator<<(unsigned char c); OStream & operator<<(int i); OStream & operator<<(short s); OStream & operator<<(long l); OStream & operator<<(unsigned int u); OStream & operator<<(unsigned short s); OStream & operator<<(unsigned long l); OStream & operator<<(long long int u); OStream & operator<<(unsigned long long int u); OStream & operator<<(const void * p); OStream & operator<<(const char * s); OStream & operator<<(float f); }; extern OStream::Begl begl; extern OStream::Endl endl; extern OStream::Hex hex; extern OStream::Dec dec; extern OStream::Oct oct; extern OStream::Bin bin;

Types

- `Begl`
  Marks the beginning of a segment of the stream (ended by `Endl`) that is to be atomically output on multicore configurations.
- `Endl`
  Encapsulates a `\n` delimiter besides marking the end of a segment of the stream (started by `Begl`) that is to be atomically output on multicore configurations.
- `Hex`
  Selects hexadecimal mode for the output of integer numbers.
- `Dec`
  Selects decimal mode for the output of integer numbers.
- `Oct`
  Selects octal mode for the output of integer numbers.
- `Bin`
  Selects binary mode for the output of integer numbers.
- `Err`
  Signalizes an error to the operating system. Besides producing a log message, usually causes a Thread abort.

Methods

- `OStream()`
  Creates an OStream object.
- `OStream & operator<<(...)`
  Converts the argument to a string and pushes it into the stream.

Examples

□□□□□□□

4.7.3. Random

EPOS provides a **Pseudorandom Number Generator** based on the linear congruential generator. Whenever the machine features devices that can be used to produce enough entropy, such as ADC

converters and RF transceivers, the algorithm is fed with a really random *seed* and therefore becomes a true Random Number Generator.

Header
`include/utility/random.h`

Interface

```
class Random { public: static int random(); };
```

Methods

- `int random()`
  Returns a random (or pseudo-random) integral number.

Examples

4.7.4. CRC

EPOS provides Cyclic Redundancy Check (CRC) functions to calculate check of 8, 16, 32, and 64 bits.

Header
`include/utility/crc.h`

Interface

```
class CRC { public: static unsigned char crc8(char * ptr, int size); static unsigned short crc16(char * ptr, int size); static unsigned long crc32(char * ptr, int size); static unsigned long long crc64(char * ptr, int size); };
```

Methods

- `unsigned short crc8(char * ptr, int size)`
  Calculates the CRC8 of the data given by (`ptr+, -+size`).
- `unsigned short crc16(char * ptr, int size)`
  Calculates the CRC16 of the data given by (`ptr+, -+size`).
- `unsigned short crc32(char * ptr, int size)`
  Calculates the CRC32 of the data given by (`ptr+, -+size`).
- `unsigned short crc64(char * ptr, int size)`
  Calculates the CRC64 of the data given by (`ptr+, -+size`).

Examples

4.7.5. Spinlock

EPOS provides Spinlocks for busy waiting synchronization. This utility is meant to be used inside the system. Applications are more likely to use Synchronization abstractions.

Header

`include/utility/spin.h`

Interface

```
class Spin { public: Spin(); void acquire(); void release(); };
```

Methods

- `Spin()`
  Creates a Spinlock.
- `void acquire()`
  Spins in a busy waiting loop until the Spinlock gets available, atomically acquiring it.
- `void release()`
  Releases the Spinlock.

Examples

4.7.6. Observer

EPOS provides a set of implementations for the Observer design pattern. *Observers* can be attached to *Observed* objects to get notification about changes in its state through invocations of `update()`.

4.7.6.1. Observer/Observed

This is the traditional design pattern.

Header

`include/utility/observer.h`

Interface

```
class Observer; class Observed { public: Observed(); ~Observed(); virtual void attach(Observer * o); virtual void detach(Observer * o); virtual bool notify(); }; class Observer { protected: Observer(); public: ~Observer(); virtual void update(Observed * o) = 0; };
```

Methods

- `Observed()`
  Creates an Observed object.
- `~Observed()`
  Destroys an Observed object, detaching all Observers.
- `void attach(Observer * o)`

Attaches Observer `o` to get notifications about this Observed object.

- `void detach(Observer * o)`
  Detaches Observer `o` from this Observed object, so it won't get notified anymore.

- `void notify()`
  Notifies all attached Observers, calling their `update()` method.

- `Observer()`
  Creates an Observer object.

- `~Observer()`
  Destroys an Observer object.

- `void update(Observed * o)`
  This pure virtual method must be implemented by the Observer to get notifications about changes to an Observed object `o`.

4.7.6.2. Conditional Observer x Conditionally Observed

This utility is similar to the traditional design pattern, but *Conditional Observers* are only notified about *Conditionally Observed* objects matching a given condition.

Header

`include/utility/observer.h`

Interface

template<typename T = int> class Conditional_Observer; template<typename T = int> class Conditionally_Observed { public: typedef T Observing_Condition; public: Conditionally_Observed(); ~Conditionally_Observed(); virtual void attach(Conditional_Observer<T> * o, T c); virtual void detach(Conditional_Observer<T> * o, T c); virtual bool notify(T c); }; template<typename T> class Conditional_Observer { public: typedef T Observing_Condition; protected: Conditional_Observer(); public: ~Conditional_Observer(); virtual void update(Conditionally_Observed<T> * o, T c) = 0; };

Methods

- `Conditionally_Observed()`
  Creates a Conditionally Observed object.

- `~Conditionally_Observed()`
  Destroys a Conditionally Observed object.

- `void attach(Conditional_Observer<T> * o, T c)`
  Attaches Observer `o` to get notifications about this Observed object whenever the condition `c` matches.

- `void detach(Conditional_Observer<T> * o, T c)`
  Detaches Observer `o` from this Observed object on condition `c`.

- `void notify(T c)`
  Notifies all Observers attached on condition `c`, calling their `update()` method.

- `Conditional_Observer()`
  Creates a Conditional Observer object.

- `~Conditional_Observer()`
  Destroys a Conditional Observer object.

- `void update(Conditionally_Observed<T> * o, T c)`
  This pure virtual method must be implemented by the Observer to get notifications about changes to an Observed object `o` whenever `c` matches.

4.7.6.3. Unconditional Observer x Unconditionally Observed with Data

This utility is similar to the traditional design pattern, but *Observers* get a pointer to data from ''Observed' objects at each notification.

Header

`include/utility/observer.h`

Interface

template<typename T1> class Data_Observed<T1, void> { public: typedef T1 Observed_Data; public: Data_Observed(); ~Data_Observed(); virtual void attach(Data_Observer<T1, void> * o); virtual void detach(Data_Observer<T1, void> * o); virtual bool notify(T1 * d) ; }; template<typename T1> class Data_Observer<T1, void> { public: typedef T1 Observed_Data; protected: Data_Observer(); public: ~Data_Observer(); virtual void update(Data_Observed<T1, void> * o, T1 * d) = 0; };

Methods

- `Data_Observed()`
  Creates an Observed object.

- `~Data_Observed()`
  Destroys an Observed object, detaching all Observers.

- `void attach(Data_Observer<T1, void> * o)`
  Attaches Observer `o` to get notifications about this Observed object.

- `void detach(Data_Observer<T1, void> * o)`
  Detaches Observer `o` from this Observed object, so it won't get notified anymore.

- `void notify(T1 * d)`
  Notifies all attached Observers, calling their `update(T1 * d)` method passing a pointer to the piece of data within the Observed object.

- `Data_Observer()`
  Creates an Observer object.

- `~Data_Observer()`
  Destroys an Observer object.

- `void update(Data_Observed<T1, void> * o, T1 * d)`
  This pure virtual method must be implemented by the Observer to get notifications about changes to an Observed object `o`. A pointer to the piece of data within the Observed object is

passed through `d`.

4.7.6.4. Conditional Observer x Conditionally Observed with Data

This utility combines the Conditional and Data Observer patterns. *Observers* get a pointer to data from ''Observed' objects whenever a condition is matched.

Header

`include/utility/observer.h`

Interface

template<typename T1, typename T2 = void> class Data_Observer; template<typename T1, typename T2 = void> class Data_Observed { public: typedef T1 Observed_Data; typedef T2 Observing_Condition; public: Data_Observed(); ~Data_Observed(); virtual void attach(Data_Observer<T1, T2> * o, T2 c); virtual void detach(Data_Observer<T1, T2> * o, T2 c); virtual bool notify(T2 c, T1 * d); }; template<typename T1, typename T2> class Data_Observer { public: typedef T1 Observed_Data; typedef T2 Observing_Condition; protected: Data_Observer(); public: ~Data_Observer(); virtual void update(Data_Observed<T1, T2> * o, T2 c, T1 * d) = 0; };

Methods

- `Data_Observed()`
  Creates an Observed object.
- `~Data_Observed()`
  Destroys an Observed object, detaching all Observers.
- `void attach(Data_Observer<T1, T2> * o, T2 c)`
  Attaches Observer `o` to get notifications about this Observed object whenever `c` matches.
- `void detach(Data_Observer<T1, T2> * o, T2 c)`
  Detaches Observer `o` from this Observed object on condition `c`.
- `void notify(T2 c, T1 * d)`
  Notifies all Observers attached on condition `c`, calling their `update(Data_Observed<T1, T2> * o, T2 c, T1 * d)` method passing a pointer to the piece of data within the Observed object through `d`.
- `Data_Observer()`
  Creates an Observer object.
- `~Data_Observer()`
  Destroys an Observer object.
- -+void update(Data_Observed<T1, T2> * o, T2 c, T1 * d)
  This pure virtual method must be implemented by the Observer to get notifications about changes to an Observed object `o` whenever `c` matches. A pointer to the piece of data within the Observed object is passed through `d`.

Examples

4.7.7. Handler

EPOS allows application processes to handle events at user-level through the **Handler** family of abstractions. Handlers can be time-triggered by Alarm or event-driven by interrupts. It is important to notice that *Semaphore* is the only handler with memory, so delayed events are not lost. Therefore, it is the right option for most of the events handled at user-level.

Header

`include/utility/handler.h`

Interface

```
class Handler { public: typedef void (Function)(); public: Handler(); virtual ~Handler(); virtual void operator()() = 0; }; class Function_Handler: public Handler { public: Function_Handler(Function * h); ~Function_Handler(); void operator()(); // h(); }; template<typename T> class Functor_Handler: public Handler { public: typedef void (Functor)(T *); Functor_Handler(Functor * h, T * o); ~Functor_Handler(); void operator()(); // h(o); }; class Thread_Handler : public Handler { public: Thread_Handler(Thread * h); ~Thread_Handler(); void operator()(); // h->resume(); }; class Semaphore_Handler: public Handler { public: Semaphore_Handler(Semaphore * h); ~Semaphore_Handler(); void operator()(); // h->v(); }; class Mutex_Handler: public Handler { public: Mutex_Handler(Mutex * h); ~Mutex_Handler(); void operator()(); // h->unlock(); }; class Condition_Handler: public Handler { public: Condition_Handler(Condition * h); ~Condition_Handler(); void operator()(); // h->signal(); };
```

Methods

- `Function_Handler(Function * h)`
  `Functor_Handler(Functor * h, T * o)`
  `Thread_Handler(Thread * h)`
  `Semaphore_Handler(Semaphore * h)`
  `Mutex_Handler(Mutex * h)`
  `Condition_Handler(Condition * h)`
  Creates a handler on object `h`.

- `~Function_Handler()`
  `~Functor_Handler()`
  `~Thread_Handler()`
  `~Semaphore_Handler()`
  `~Mutex_Handler()`
  `~Condition_Handler()`
  Destroys the handler.

- `void operator()()`
  The *call operator* is used to invoke the Handler. As a pure virtual method in the base class, it must be defined for each kind of Handler.
  - For `Function_Handler`, it calls the function given by `h`.
  - For `Functor_Handler`, it calls the functor `h` on object `o`.
  - For `Thread_Handler`, it calls `resume()` on the Thread given by `h`.
  - For `Semaphore_Handler`, it calls `v()` on the Semaphore given by `h`.
  - For `Mutex_Handler`, it calls `unlock()` on the Mutex given by `h`.
  - For `Condition_Handler`, it calls `signal()` on the Condition Variable given by `h`.

Examples

4.7.8. Buffer (Zero-Copy)

Header

`include/utility/buffer.h`

Interface

```
template<typename Owner, typename Data, typename Shadow = void, typename Metadata = Dummy> class Buffer: private Data, public Metadata { public: typedef Simple_List<Buffer<Owner, Data, Shadow, Metadata> > List; typedef typename List::Element Element; public: Buffer(Shadow * s); Buffer(Owner * o, unsigned int s); template<typename ... Tn> Buffer(Owner * o, unsigned int s, Tn ... an); Data * data(); Data * frame(); Data * message(); bool lock(); void unlock(); Owner * owner() const; Owner * nic() const; void owner(Owner * o); void nic(Owner * o); Shadow * shadow() const; Shadow * back() const; unsigned int size() const; void size(unsigned int s); Element * link1(); Element * link(); Element * lint(); Element * link2(); Element * lext(); friend Debug & operator<<(Debug & db, const Buffer & b); };
```

Methods

-

Examples

4.8. Hardware Mediators

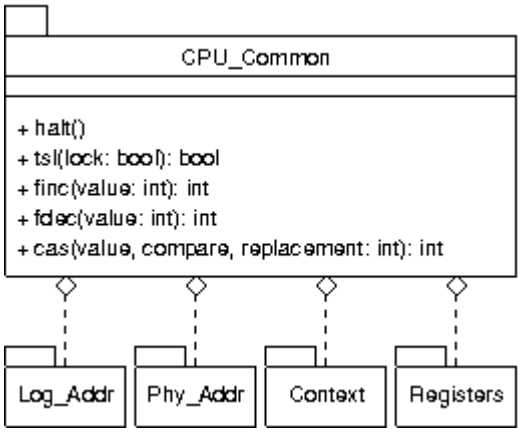
4.8.1. CPU

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

Generic implementations of CPU interface are provided by CPU_Common. Architecture-specifc implementations are provided by each architecture'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.

Below is a class diagram for this interface.



Methods

- `static void halt()`
  This function is reimplemented in the CPU mediators of architectures 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 come back from a halt.

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

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

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

4.8.2. MMU

The MMU is a hardware mediator responsible for abstracting 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 below shows the hierarchy of the low level memory abstractions.

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

4.8.3. TSC

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

first returns the TSC or timer frequency. The second, returns the current number of ticks.

Methods

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

Types

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

4.8.4. 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-specific implementations are provided by each machine's Machine mediator (e.g., PC, ATMega128, Plasma, etc).

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

Methods

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

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

- `static void reboot()`
  When called, this function causes the system to be shut down 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.

4.8.5. IC

The IC mediator is responsible for abstracting the 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.

Below are the signatures for the component's interface methods. **Interrupt_Id** is an 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)();
That means that an interrupt handler should be a method with the following signature:

void handler();

Methods

- `static void int_vector(Interrupt_Id irq, Interrupt_Handler handler)`
  This method maps handler to a given IRQ.
- `static void enable(Interrupt_Id irq)`
  Enables interrupts for a given IRQ.
- `static void disable()`
  Disables all interrupts.
- `static void disable(Interrupt_Id irq)`
  Disables interrupts for a given IRQ.

4.8.6. RTC

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

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

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

Date Types

- `unsigned int _Y`
- `unsigned int _M`

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

RTC Methods

- `RTC()`
  Constructs an RTC object.

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

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

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

Date Methods

- `Date()`
- `Date(unsigned int _Y, unsigned int _M, unsigned int _D, unsigned int _h, unsigned int _m, unsigned int _s)`
- `unsigned int year()`
- `unsigned int month()`
- `unsigned int day()`
- `unsigned int hour()`
- `unsigned int minute()`
- `unsigned int second()`
- `void operator <<`

4.8.7. Timers

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

Types

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

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

Methods

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

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.

4.8.8. UART

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

Methods

- `UART(unsigned int unit = 0)`
  Creates a UART object. The unit defines which hardware device is being used. By default, the first device is chosen.
- `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 a UART device. The method will wait until the data is ready.

- `void put(char c)`
  Sends a byte (*c*) to a UART device. The method will wait until the data is transferred.

4.8.8.1. Example

// EPOS PC UART Mediator Test Program #include <utility/ostream.h> #include <uart.h> using namespace EPOS; int main() { OStream cout; cout << "PC_UART test\n" << endl; PC_UART uart(115200, 8, 0, 1); cout << "Loopback transmission test (conf = 115200 8N1):"; uart.loopback(true); for(int i = 0; i < 256; i++) { uart.put(i); int c = uart.get(); if(c != i) cout << " failed (" << c << ", should be " << i << ")!" << endl; } cout << " passed!" << endl; cout << "Link transmission test (conf = 9200 8N1):"; uart.config(9600, 8, 0, 1); uart.loopback(false); for(int i = 0; i < 256; i++) { uart.put(i); for(int j = 0; j < 0xffffff; j++); int c = uart.get(); if(c != i) cout << " failed (" << c << ", should be " << i << ")!" << endl; } cout << " passed!" << endl; return 0; }

4.8.9. NIC

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

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

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

- `int send(const Address, const Protocol &prot, const void *data, unsigned int size)`
  Sends `size` bytes of data to `dst` with protocol `prot`.

- `int receive(Address *src, Protocol *prot, void *data, unsigned int size)`
  Receives `size` bytes of data, `src` and `prot` are set by the method accordingly.

- `void reset()`
  Resets the NIC device.

- `unsigned int mtu()`
  Returns the device mtu (Maximum Transmission Unit).

- `const Address address()`
  Returns the device address.

- `const Statistics statistics()`

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

4.8.10. Radio

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

4.8.11. EEPROM

EEPROMs (Electrically-Erasable Programmable Read-Only Memory) are non-volatile storage device. An EEPROM has a high read/write latency and is 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. Below is shown the public interface for the EEPROM mediator.

Methods

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

THIS MUST BE RELOCATED

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

template<> struct Memory_Map<PC> { // Physical Memory enum { MEM_BASE = Traits<PC>::MEM_BASE, MEM_TOP = Traits<PC>::MEM_TOP }; // Logical Address Space enum { APP_LOW = Traits<PC>::APP_LOW, APP_CODE = Traits<PC>::APP_CODE, APP_DATA = Traits<PC>::APP_DATA, APP_HIGH = Traits<PC>::APP_HIGH, PHY_MEM = Traits<PC>::PHY_MEM, IO = Traits<PC>::IO_BASE, APIC = IO, VGA = IO + 4 * 1024, PCI = IO + Traits<PC_Display>::FRAME_BUFFER_SIZE, SYS = Traits<PC>::SYS, IDT = SYS + 0x00000000, GDT = SYS + 0x00001000, SYS_PT = SYS + 0x00002000, SYS_PD = SYS + 0x00003000, SYS_INFO = SYS + 0x00004000, TSS0 = SYS + 0x00005000, SYS_CODE = SYS + 0x00300000, SYS_DATA = SYS + 0x00340000, SYS_STACK = SYS + 0x003c0000, SYS_HEAP =

```
    SYS + 0x00400000 }; };
```

For a 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 belong to the address space of a task. Its public interface is described below. For more information see the Task and MMU abstraction.

Review Log

Ver	Date	Authors	Main Changes
1.0	Apr 4, 2016	Rodrigo Meurer	Import and cleanup from EPOS 1 documentation; Substitution of JPEG UML images by textual class interfaces
1.1	Apr 10, 2016	Guto Fröhlich	Major review
1.2	Mai 12, 2016	Guto Fröhlich	Added section on SmartData
1.3	Mai 14, 2016	Guto Fröhlich	Utilities classes rewritten
1.4	Feb 24, 2020	Guto Fröhlich	Major 2.2 review
1.5	Aug 2x, 2023	José Hoffmann	WiP: Updating Digital Units and adding MultiSmartData