wasmachine

0.1

Generated by Doxygen 1.8.11

Contents

Chapter 1

LibSerial class

This class is based on a library implemented by Philippe Lucidarme (University of Angers) serialib@googlegroups.com
The class LibSerial offers simple access to the serial port devices for Linux.
It can be used for any serial device (Built-in serial port, USB to RS232 converter, Arduino board, Raspberry Pi or any hardware using or emulating a serial port). class allows basic operations like :

- opening and closing connection
- reading data (characters, strings or arrays of bytes)
- writing data (characters, strings or arrays of bytes)
- peeking data and flushing the receive buffer

Chapter 2

Compile-time configuration

The RTOS can be configured by changing some #define's in pRTOS.h

global_logging

All RTOS debug logging can be switched off by defining **global_logging** as 0 It is advised to make all application debug logging likewise dependent on this macro. Check pRTOS.cpp for examples

RTOS_STATISTICS_ENABLED

By default **RTOS_STATISTICS_ENABLED** is defined as 1, which enables printing statistics for all objects. It can be defined as 0 to reduce code and data size. NOTE: doing so will disable deadlock detection!

When statistics is enabled, hitting Ctrl-C terminates the RTOS schedular and shows the statistics.

RTOS_DEFAULT_STACK_SIZE

The default stack size is 4K. You can choose another value by changing the initialization of **RTOS_DEFAULT_S↩ TACK_SIZE**.

RTOS_MIN_PRIORITY

The priority you can assign to a task ranges from 0 to 1000, 0 being the highest priority and 1000 the lowest. Tasks should all have a different priority.

Chapter 3

Debug logging support

The RTOS uses std::cout output stream for logging support.

The RTOS defines the trace macro, which can be used like cout, but prefixes each output with the current source file name and the current source line number. Hence (after the appropriate preparations) the statement

```
trace << "n=" << n << "\n";
```

can create the output line

```
main.c:20 n=15
```

This provides an easy way to check if and when a certain line of code is executed, and optionally print some debugging information.

Note that using the logging mechanism influences the execition of the task, maybe resulting in missing their deadlines. The suggested initialization does not implement buffering, so using cout or trace can change the timing of a task that does printing considerably.

All objects (RTOS, task, event, all waitables, mutex, pool, mailbox, channel) can be printed to an ostream using the $<<$ operator. Printing the RTOS will print all RTOS objects.

Chapter 4

Unit macro's

These macro's make it easier to specify (interval) times

```
#define  S * ( 1000 * 1000 )
#define MS * 1000
#define US * 1
```

Time in the RTOS is expressed in microseconds. Using these marco's avoids knowledge of this detail.

Chapter 5

Non-preemptive task switching

The RTOS uses non-preemptive task switching. This means that the CPU can be switched to another task only when the currenly executing task (directly or indirectly) calls an RTOS function. Three groups of RTOS function calls can cause such a task switch:

1. functions that cause the current task to become non-runnable, like task::wait(), and task::suspend()
2. functions that make a higher-priority task runnable, like flag::set(), and task::resume()
3. the function task::release(), which only purpose is to give up the CPU to a higher priority task.

A task can be made runnable either by an excplicit action from another task, like an event_flag::set() or task↩ ::resume() call, or implicitly by the expiration of a timer. But even for the latter case (timer expiration) the switching to another task can occur only when an RTOS function is called.

The diagram below shows the state-event diagram for a task. The transitions from ready to running and back are governed by the RTOS always selecting the highest-priority runnable task. The events that cause the transitions between runnable and blocked and vice versa, and between blocked-and-suspended and suspended are the same, they are shown in the enlarged box. A task can only get blocked by doing something (wait, read a mailbox, etc), hence there is no transition from suspended to blocked-and-suspended.

Chapter 6

Latency

When a task is activated by a timeout (either by a timer or clock, or because it is a periodic task) at a certain moment in time it will in general not be run at that time exactly, but at some later time. This delay is called the latency. Two things contribute to the latency:

1. Higher priority taks will be run first, until no higher priority tasks are runnable.
2. When a lower priority task is running when the timer fires the RTOS will notice this only when one of the RTOS functions is called that does a rescheduling: task::wait(), flag::set(), task::suspend(), task::resume(), task::release().

The first contribution is a consequence of the design of the application. When you feel that it is inappropriate that a particular higher-priority task is run first and hence contributes to the latency of the task that is activated by the timer, you have set the task priorities wrong.

The second contribution is specific for a non-preemptive RTOS. (A preemptive RTOS would immediately stop (preempt) the running lower-priority task and switch to the higher-priority task.) When you have lower priority tasks in your system that use a lot of CPU time between the RTOS calls that do rescheduling you can insert task↩ ::release() calls. This call checks whether any timers that have timed out made a higher-priority task runnable, and if so, switches to that task.

When a task is made runnable by an explicit action of another task, for instance a task:resume() call or a flag::set() call, only the first source of delay (higher priority tasks that are runnable) is applicable, because inside such calls the RTOS will immediatley switch to the highest priority runnable task.

Chapter 7

Hierarchical Index

7.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 8

Class Index

8.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 9

File Index

9.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 10

Class Documentation

10.1 Broadcaster Class Reference

Public Member Functions

- void **add** (WebSocket *ws)
- void **remove** (WebSocket *ws)
- void **broadcast** (const string &message)

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/wsmulticaster.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/wsmulticaster.cpp

10.2 RTOS::channel< T, SIZE > Class Template Reference

waitable data queue

`#include <pRTOS.h>`

Inheritance diagram for RTOS::channel< T, SIZE >:

Public Member Functions

- channel (task *t, const char *name="")
  constructor, specify stored type, number of entries, and name
- void write (T item)
  write an item to the queue
- T read (void)
  read an item from the queue
- void clear (void)
  clear the waitable

Additional Inherited Members

10.2.1 Detailed Description

template<class T, const int SIZE>
class RTOS::channel< T, SIZE >

waitable data queue

The (communication) channel is a template class that stores a queue of values. Values can be written at the tail of the queue, up to the number of entries for which the channel was created. It is an error to write to a channel that is full. Writes are not blocking. Any task can write to a channel.

A channel is created for a particular task. Only this owner task can read from the channel. A read will block until an entry is available. Reads are from the head of the queue.

A channel is a waitable, so the task that owns the channel can wait for the channel to be non-empty, after which a read from a channel will be non-blocking (because the channel is not empty). After a wait() that returns the channel's event, the channel will set itself again (because the wait did not cause it to become empty). Only a read that results in an empty queue will clear the channel.

The example below shows how writing to cout can be buffered by first writing to a 2kB channel, and reading from that channel at a maximum of one character per 2 MS. The UART hardware in the LPC2148 chip buffers one character, which at default baudrate (38k4) takes $\sim$ 1 MS to write. So by writing at a maximum rate of one character per 2 MS no blocking will occur.

```
class output_class : public task {
public:
   channel< char, 2048 > buffer( this, "buffer" );
   timer hartbeat( this, "hartbeat" );
   void main( void ){
      for( ; ; ) {
         wait( buffer );
         cout << buffer.get();
         timer.set( 2 MS );
         wait( timer );
     }
   }
}

output_class output;

void print( char * s ){
   while( *s != '\0' ) { output.buffer.write( *s++ ); }
}
```

10.2.2 Constructor & Destructor Documentation

10.2.2.1 template<class T, const int SIZE> RTOS::channel< T, SIZE >::channel (task * *t*, const char * *name* = " ") [inline]

constructor, specify stored type, number of entries, and name

The template argument T must be a class that has a non-arguments constructor and supports assignment.

10.2.3 Member Function Documentation

10.2.3.1 template<class T, const int SIZE> void RTOS::channel< T, SIZE >::clear (void) [inline], [virtual]

clear the waitable

This is automatically doen when the waitable causes a task::wait() call to return it.

Reimplemented from RTOS::waitable.

The documentation for this class was generated from the following file:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h

10.3 RTOS::channel_base Class Reference

RTOS private implementation class.

`#include <pRTOS.h>`

Inheritance diagram for RTOS::channel_base:

Public Member Functions

- void **print** (std::ostream &s, bool header=true) const

Protected Member Functions

- **channel_base** (task *t, const char *name)

Protected Attributes

- const char * **channel_name**
- channel_base * **next_channel**
- int **writes**
- int **ignores**
- int **qSize**
- int **head**
- int **tail**

Friends

- class **RTOS**

10.3.1 Detailed Description

RTOS private implementation class.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.cpp

10.4 RTOS::clock Class Reference

free-running clock, ticks at a fixed frequency

`#include <pRTOS.h>`

Inheritance diagram for RTOS::clock:

Public Member Functions

- clock (task *t, unsigned long int _period, const char *name="")
  create a clock for task t, specify interval and name
- void clear (void)
  clear the waitable within the clock
- unsigned long int interval (void)
  the interval of the clock
- void print (std::ostream &s, bool header=true) const
  print the clock (for debugging)

Friends

- class **RTOS**

Additional Inherited Members

10.4.1 Detailed Description

free-running clock, ticks at a fixed frequency

A clock is a waitable which is automatically sets itself at fixed intervals. The interval between these moments is specified when the clock is created. A clock is always running, even when the task to which it belongs is suspended.

10.4.2 Constructor & Destructor Documentation

10.4.2.1 RTOS::clock::clock (task * *t,* unsigned long int *_period,* const char * *name* = " ")

create a clock for task t, specify interval and name

The name is used for debugging and statistics.

10.4.3 Member Function Documentation

10.4.3.1 void RTOS::clock::clear (void) [inline], [virtual]

clear the waitable within the clock

Note that this does not stop the clock.

Reimplemented from RTOS::waitable.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.cpp

10.5 CommunicatingSocket Class Reference

`#include <PracticalSocket.h>`

Inheritance diagram for CommunicatingSocket:

Public Member Functions

- void send (const void *buffer, int bufferLen) throw (SocketException)
- size_t recv (void *buffer, int bufferLen) throw (SocketException)
- size_t recvFully (void *buffer, int bufferLen) throw (SocketException)
- SocketAddress getForeignAddress () throw (SocketException)

Additional Inherited Members

10.5.1 Detailed Description

Abstract base class representing a socket that, once connected, has a foreign address and can communicate with the socket at that foreign address.

10.5.2 Member Function Documentation

10.5.2.1 SocketAddress CommunicatingSocket::getForeignAddress ( ) throw SocketException)

Get the address of the peer to which this socket is connected. The socket must be connected before this method can be called.

Returns

foreign address

Exceptions

SocketException	thrown if unable to fetch foreign address

10.5.2.2 size_t CommunicatingSocket::recv (void * *buffer*, int *bufferLen*) throw SocketException)

Read into the given buffer up to bufferLen bytes data from this socket. The socket must be connected before recv can be called.

Parameters

buffer	buffer to receive the data
bufferLen	maximum number of bytes to read into buffer

Returns

number of bytes read, 0 for EOF.

Exceptions

SocketException	thrown if unable to receive data

10.5.2.3 size_t CommunicatingSocket::recvFully (void * *buffer*, int *bufferLen*) throw SocketException)

Block until bufferLen bytes are read into the given buffer, until the socket is closed or an error is encoutered. The socket must be connected before recvFully can be called.

Parameters

buffer	buffer to receive the data
bufferLen	maximum number of bytes to read into buffer

Returns

number of bytes read, 0 for EOF, and -1 for error

Exceptions

SocketException	thrown if unable to receive data

10.5.2.4 void CommunicatingSocket::send (const void * *buffer*, int *bufferLen*) throw SocketException)

Write bufferLen bytes from the given buffer to this socket. The socket must be connected before send() can be called.

Parameters

buffer	buffer to be written
bufferLen	number of bytes from buffer to be written

Exceptions

SocketException	thrown if unable to send data

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.cpp

10.6 Door Class Reference

Public Member Functions

- Door ()
- char * getLockCommand ()

  Returns the bytes to set the lock on.
- char * getUnlockCommand ()

  Returns the bytes to set the lock off.
- char * getStatusCommand ()

  Returns the bytes to get the status of the lock.

10.6.1 Detailed Description

Used as an interface to get the appropiate bytes for the uart

10.6.2 Constructor & Destructor Documentation

10.6.2.1 Door::Door ( )

file Heater.cpp version 0.1 author Remco Nijkamp / Jordan Ranirez / Kevin Damen / Jeroen Kok date 19-01-2016

10.6.3 Member Function Documentation

10.6.3.1 char * Door::getLockCommand ( )

Returns the bytes to set the lock on.

Returns

a char pointer to a 2 char array

10.6.3.2 char * Door::getStatusCommand ( )

Returns the bytes to get the status of the lock.

Returns

a char pointer to a 2 char array

10.6.3.3 char * Door::getUnlockCommand ( )

Returns the bytes to set the lock off.

Returns

a char pointer to a 2 char array

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/Door.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/Door.cpp

10.7 RTOS::event Class Reference

set of things that can happen, or a thing that has happened

`#include <pRTOS.h>`

Inheritance diagram for RTOS::event:

Public Member Functions

- void print (std::ostream &s) const
  prints an event, for debugging only
- bool operator== (const event &rhs) const
  report wether two events are the same
- bool operator== (const waitable &rhs) const
  report whether an event corresponds to a waitable
- bool operator!= (const event &rhs) const
  report wether two events are not the same
- bool operator!= (const waitable &rhs) const
  report whether an event does not correspond to a waitable
- event operator+ (const event &rhs) const
  add two waitables, result can be used in a wait() call

Protected Member Functions

- event (task *t, unsigned int mask)

  constructor, used by concrete events

Protected Attributes

- task * t

  the owner task
- unsigned int mask

  the mask of this event, one bit is set

Friends

- class **waitable_set**

10.7.1 Detailed Description

set of things that can happen, or a thing that has happened

An event

- is the result of adding waitables
- is accepted as argument to wait()
- is returned by wait()
- can be compared to a waitable

The task::wait() calls return an event. Such an event can be compared to a waitable. The result is true if and only if the waitable caused the event.

Events are the only RTOS objects that can be destroyed (without causing an error).

10.7.2 Member Function Documentation

10.7.2.1 RTOS::event RTOS::event::operator+ (const event & *rhs*) const

add two waitables, result can be used in a wait() call

Waitables can be added (operator+) to construct a 'set of waitables' as argument to a task::wait() call.

10.7.3 Member Data Documentation

10.7.3.1 unsigned int RTOS::event::mask [protected]

the mask of this event, one bit is set

The bit that is set is unique among the events owned by a task.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.cpp

10.8 fiber_t Class Reference

Provides a coroutine-based tasking service using the POSIX ucontext.

`#include <libfiber.h>`

Public Member Functions

- fiber_t (void(*func)(void), int sz)
- ~fiber_t (void)
- void resume (void)
- void * stackBase (void)
- int stackSize (void)
- int stackUsed (void)

Static Public Attributes

- static fiber_t main_fiber
- static fiber_t * running_fiber = &main_fiber

10.8.1 Detailed Description

Provides a coroutine-based tasking service using the POSIX ucontext.

10.8.2 Constructor & Destructor Documentation

10.8.2.1 fiber_t::fiber_t (void(*)(void) *func,* int *sz*) [inline]

Construct a fiber object

Parameters

func	the function body for the fiber
sz	the size of the stack

10.8.2.2 fiber_t::~fiber_t (void) [inline]

Destroy a fiber object

10.8.3 Member Function Documentation

10.8.3.1 void fiber_t::resume (void)

Resume this fiber

10.8.3.2 void* fiber_t::stackBase (void) [inline]

Return the base address off the stack

Returns

the base address off the stack

10.8.3.3 int fiber_t::stackSize (void) [inline]

Return the size off the stack

Returns

the size off the stack

10.8.3.4 int fiber_t::stackUsed (void)

Return the # stack bytes used

Returns

bytes used

10.8.4 Member Data Documentation

10.8.4.1 fiber_t fiber_t::main_fiber [static]

The fiber object for the main thread.

10.8.4.2 fiber_t * fiber_t::running_fiber = &main_fiber [static]

Pointer to the object for the running fiber.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/libfiber.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/libfiber.cpp

10.9 RTOS::flag Class Reference

basic synchronisation mechanism.

`#include <pRTOS.h>`

Inheritance diagram for RTOS::flag:

Public Member Functions

- flag (task *t, const char *name="")
  constructor, specify onwer and name
- void set (void)
  sets the flag
- void print (std::ostream &s, bool header=true) const
  prints flag infomation (for debugging)

Friends

- class **RTOS**

Additional Inherited Members

10.9.1 Detailed Description

basic synchronisation mechanism.

The basic synchronization mechanism is the (event) flag. Like all waitables, a flag is created for a particular task. A flag is set by a flag::set() call (or the task::set(flag) call, which has the same effect). Like all waitables, when a task is waiting for a flag (using a task::wait call) and that flag becomes set, the wait call will clear the flag, and return an event that compares equal to the flag. Note that a flag does not count: setting a flag that is already set has no effect on the flag.

A flag must be created for a specific task. The normal place to do this is in the task's creator. An flag is initially cleared.

The example below shows a led_task that responds to two event flags. The shift flag will cause it to shift the pattern on the LEDs one position to the left, while the invert flag will cause it to invert the pattern. Two addional tasks do notghing but set these flags at fixed intervals. The result is a sort of one-direction Kitt display, which will occasionally flip polarity. Note that in this example the wait call excplicitly mentions the flags it waits for.

10.9.2 Constructor & Destructor Documentation

10.9.2.1 RTOS::flag::flag (task * t, const char * name = " ")

constructor, specify onwer and name

This call creates a timer for task t. The name is used for debugging and statistics.

10.9.3 Member Function Documentation

10.9.3.1 void RTOS::flag::set (void)

sets the flag

Setting a flag causes the task that waits for this flag to be awakened.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.cpp

10.10 Heater Class Reference

Public Member Functions

- Heater ()
- char * getOnCommand ()

  Returns the bytes to set the heater on.
- char * getOffCommand ()

  Returns the bytes to set the heater off.
- char * getStatusCommand ()

  Returns the bytes to get the status of the heater.

10.10.1 Detailed Description

Used as an interface to get the appropiate bytes for the uart

10.10.2 Constructor & Destructor Documentation

10.10.2.1 Heater::Heater ( )

file Heater.cpp version 0.1 author Remco Nijkamp / Jordan Ranirez / Kevin Damen / Jeroen Kok date 19-01-2016

10.10.3 Member Function Documentation

10.10.3.1 char * Heater::getOffCommand ( )

Returns the bytes to set the heater off.

Returns

a char pointer to a 2 char array

10.10.3.2 char * Heater::getOnCommand ( )

Returns the bytes to set the heater on.

Returns

a char pointer to a 2 char array

10.10.3.3 char * Heater::getStatusCommand ( )

Returns the bytes to get the status of the heater.

Returns

a char pointer to a 2 char array

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/Heater.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/Heater.cpp

10.11 LibSerial Class Reference

This class can manage a serial port. The class allows basic operations (opening the connection, reading, writing data and closing the connection).

```
#include <libserial.h>
```

Public Member Functions

- LibSerial (void)

  Constructor of class LibSerial.
- ~LibSerial (void)

  Destructor of class LibSerial. It closes the connection.
- int open (const char *device, unsigned int bauds)

  Open the serial port.
- void close (void)

  Close the connection with the current device.
- int writeChar (char)

  Write a char on the current serial port.
- int readChar (char *pByte)

  Wait for a char from the serial device and return the data read.
- int writeString (const char *string)

  Write a string on the current serial port.
- int readString (char *string, char finalChar, unsigned int maxBytes)

  Read a string from the serial device.
- int write (const void *buffer, unsigned int nbBytes)

  Write an array of data on the current serial port.
- int read (void *buffer, unsigned int maxBytes)

  Read an array of bytes from the serial device.
- void flush (void)

  Empty send and receive buffers.
- int peek (void)

  Return the number of bytes in the receive buffer.

10.11.1 Detailed Description

This class can manage a serial port. The class allows basic operations (opening the connection, reading, writing data and closing the connection).

10.11.2 Member Function Documentation

10.11.2.1 int LibSerial::open (const char * *device*, unsigned int *bauds*)

Open the serial port.

Parameters

device	: Port name /dev/ttyS0, /dev/ttyAMA0, /dev/ttyUSB0 ...
bauds	: Baud rate of the serial port.

Supported baud rate for Linux :

- 110

- 300
- 600
- 1200
- 2400
- 4800
- 9600
- 19200
- 38400
- 57600
- 115200

The device is configured 8N1: 8 bits, no parity, 1 stop bit

Returns

1 success
-1 error while opening the device
-2 unable to set non-blocking mode
-3 speed (bauds) not recognized
-4 unable to set baud rate
-5 unable to set flags

10.11.2.2 int LibSerial::peek (void)

Return the number of bytes in the receive buffer.

Returns

The number of bytes in the receive buffer

10.11.2.3 int LibSerial::read (void * *buffer,* unsigned int *maxBytes*)

Read an array of bytes from the serial device.

Parameters

buffer	: array of bytes read from the serial device
maxBytes	: maximum allowed number of bytes to read

Returns

1 success, return the number of bytes read
-1 error while reading the bytes

10.11.2.4 int LibSerial::readChar (char * *pChar*)

Wait for a char from the serial device and return the data read.

Parameters

pChar	: char read on the serial device

Returns

1 success
-1 error while reading the char

10.11.2.5 int LibSerial::readString (char * *string,* char *finalChar,* unsigned int *maxChars*)

Read a string from the serial device.

Parameters

string	: string read on the serial device
finalChar	: final char of the string
maxChars	: maximum allowed number of chars read

Returns

>0 success, return the number of chars read
-1 error while reading the char
-2 maxChars is reached

10.11.2.6 int LibSerial::write (const void * *buffer,* unsigned int *nbBytes*)

Write an array of data on the current serial port.

Parameters

buffer	: array of bytes to send on the port
nbBytes	: number of bytes to send

Returns

1 success
-1 error while writting data

10.11.2.7 int LibSerial::writeChar (char *Char*)

Write a char on the current serial port.

Parameters

Char	: char to send on the port

Returns

1 success
-1 error while writting data

10.11.2.8 int LibSerial::writeString (const char * *string*)

Write a string on the current serial port.

Parameters

string	: string to send on the port (must be terminated by '\0')

Returns

1 success
-1 error while writting data

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/libserial.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/libserial.cpp

10.12 RTOS::mailbox< T > Class Template Reference

Synchronously handing over of a data item.

`#include <pRTOS.h>`

Inheritance diagram for RTOS::mailbox< T >:

Public Member Functions

- mailbox (const char *name="")
  constructor, specify mailbox name
- void write (const T item)
- T read (void)
  read a value from the mailbox

10.12.1 Detailed Description

template<class T>
class RTOS::mailbox< T >

Synchronously handing over of a data item.

A mailbox is a template class synchronization mechanism. A single value can be written to the mailbox. Another task can read the value from the mailbox. The read and write calls wait on each other before they are allowed to proceed.

A mailbox is not created for a particular task, and it is not a waitable.

Initially a mailbox is empty. The write() operation writes to the mailbox, and, if present, unblocks a reading task and returns, otherwise the writing task is blocked. The read() operation blocks the calling task until there is a value in the mailbox. Then it reads the value, unblocks the task that wrote to the mailbox, and returns.

10.12.2 Constructor & Destructor Documentation

10.12.2.1 template<class T > RTOS::mailbox< T >::mailbox (const char * *name* = " ") [inline]

constructor, specify mailbox name

Create a mailbox. The mailbox is initially empty. The template argument T must be a class that has a non-arguments constructor and supports assignment.

10.12.3 Member Function Documentation

10.12.3.1 template<class T > T RTOS::mailbox< T >::read (void) [inline]

read a value from the mailbox

If a writing tasks is waiting for the mailbox it is unblocked and the reader gets the data. Otherwise the current task is blocked until it is released by a writer.

10.12.3.2 template<class T > void RTOS::mailbox< T >::write (const T *item*) [inline]

write an item into the mailbox

The current (writing) task stores an item in the mailbox. If a client (reader) is waiting, it is unblocked. Otherwise the task waits (is blocked) until a reading task has read the item.

The documentation for this class was generated from the following file:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h

10.13 RTOS::mailbox_base Class Reference

RTOS private implementation class.

`#include <pRTOS.h>`

Inheritance diagram for RTOS::mailbox_base:

Public Member Functions

- **mailbox_base** (const char *name)
- void **print** (std::ostream &s, bool header=true) const

Public Attributes

- task * **writer**
- task * **reader**
- const char * **mailbox_name**
- unsigned int **writes**
- unsigned int **reads**
- mailbox_base * **next_mailbox**

10.13.1 Detailed Description

RTOS private implementation class.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.cpp

10.14 Motor Class Reference

Public Member Functions

- Motor ()
- char * getMotorSpeedCommand ()

  Returns the bytes to get the speed of the motor.
- char * turn (bool dir, int speed)

  Sets the right speed and the right direction (or left direction) of the motor.

10.14.1 Detailed Description

Used as an interface to get the appropiate bytes for the uart

10.14.2 Constructor & Destructor Documentation

10.14.2.1 Motor::Motor ( )

file Heater.cpp version 0.1 author Remco Nijkamp / Jordan Ranirez / Kevin Damen / Jeroen Kok date 19-01-2016

10.14.3 Member Function Documentation

10.14.3.1 char * Motor::getMotorSpeedCommand ( )

Returns the bytes to get the speed of the motor.

Returns

a char pointer to a 2 char array

10.14.3.2 char * Motor::turn (bool *dir,* int *speed*)

Sets the right speed and the right direction (or left direction) of the motor.

Parameters

dir	the direction of the motor. false = right, true = left
speed	the speed of the rpm. 0 <= x <= 1600

Returns

the bytes used to set the RPM in a direction.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/Motor.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/Motor.cpp

10.15 MotorController Class Reference

Inheritance diagram for MotorController:

Public Member Functions

- **MotorController** (WashingMachineController *wascontroller)
- void setUartPointer (UART *u)

  Used to (re)set the pointer to the uart.
- int getMotorSpeed ()

  returns the current speed of the motor
- void setMotorJob (int job, int time)

  Sets which job should be started and for how long.
- void setResponseFlag ()

  sets the response flag
- void writeResponse (char *response)

  writes a response in the response pool
- void main ()

  task body, must be provided by a derived class

Additional Inherited Members

10.15.1 Member Function Documentation

10.15.1.1 int MotorController::getMotorSpeed ( )

returns the current speed of the motor

Returns

the speed int RPM

10.15.1.2 void MotorController::main ( ) [virtual]

task body, must be provided by a derived class

A task is created by inheriting from task and providing a main() function. Initialisation of the task, including creating its waitables, should be done in the constructor. Don't forget to call the constructor of the task class!

The main() is the body of the task. It should never terminate.

Each task has a unique priority (an unsigned integer). A lower value indicates a higher priority. The RTOS scheduler will always run the task with the higest-priority runnable (neither blocked nor suspended) task. A task runs until it changes this 'situation' by using an RTOS call that changes its own state to not runnable, or the state of a higher priority task to runnable.

Timers are served only when the RTOS is activated by calling any of its state-changing interfaces. Hence the longest run time between such calls determines the granularity (time wise responsiveness) of the application. Within a time consuming computation a task can call release() to have the RTOS serve the timers.

Implements RTOS::task.

10.15.1.3 void MotorController::setMotorJob (int *job*, int *time*)

Sets which job should be started and for how long.

Parameters

int	which job should be used
int	the time in seconds

Returns

void

10.15.1.4 void MotorController::setResponseFlag ( )

sets the response flag

Returns

void

10.15.1.5 void MotorController::setUartPointer (UART * *u*)

Used to (re)set the pointer to the uart.

Parameters

*u**	pointer to the UART object this controller should use.

10.15.1.6 void MotorController::writeResponse (char * *response*)

writes a response in the response pool

Parameters

response	a char array with two positions

Returns

void

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/MotorController.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/MotorController.cpp

10.16 RTOS::mutex Class Reference

mutual execlusion semaphore

`#include <pRTOS.h>`

Public Member Functions

- mutex (const char *name="")

  constructor, specify the name
- ~mutex (void)

  generates an error
- void print (std::ostream &stream, bool header=true) const

  prints a mutex, for debugging only.
- void wait (void)
- void signal (void)

  release the mutex

Friends

- class **RTOS**

10.16.1 Detailed Description

mutual execlusion semaphore

A mutex (mutual exclusion semaphore) is a synchronization mechanism that is used to give a task exclusive access to some resource: the task can execute a sequence of statements, being sure that no other task is accessing the same resource.

A typical use is to protect a resource (for instance global data) that should be used by only one task at a time, so it can update it and leave it in a consistent state.

A mutex is not created for a particular task, and it is not a waitable.

Initially a mutex is free. The mutex::wait() operation blocks the task until the mutex is free, and then claims the mutex for the executing task. The mutex::signal() operation frees the mutex again. It is an error to call mutex::signal on a mutex that is not currently owned by the executing task.

10.16.2 Constructor & Destructor Documentation

10.16.2.1 RTOS::mutex::mutex (const char * *name* = " ")

constructor, specify the name

The name is used for debugging only.

10.16.2.2 RTOS::mutex::~mutex (void)

generates an error

A mutex should never be destroyed

10.16.3 Member Function Documentation

10.16.3.1 void RTOS::mutex::signal (void)

release the mutex

If one or more tasks are waiting for the mutex the fires one is released, and it now owns the mutex. Otherwise, if the mutex is cleared it is now set.

It is an error for a task to call signal() on a mutex that it does not own (that it did not call wait() on). After the signal the task no longer owns the mutex.

10.16.3.2 void RTOS::mutex::wait (void)

claim the mutex

If the mutex was set it it is now cleared, and the calling task owns the mutex.

Otherwise the current task waits (is halted) until the owning task calls signal() on the same mutex. The signal() calls will release the tasks in the order of their wait() calls.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.cpp

10.17 RTOS::pool< T > Class Template Reference

place to store and rectrieve data, no built-in synchronisation

```
#include <pRTOS.h>
```

Inheritance diagram for RTOS::pool< T >:

Public Member Functions

- pool (const char *name="")
  construct a pool, specify its name (for debgging only)
- void write (T item)
  atomic write operation on a pool
- T read (void)
  atomic read operation on a pool

Additional Inherited Members

10.17.1 Detailed Description

template<class T>
class RTOS::pool< T >

place to store and rectrieve data, no built-in synchronisation

A (communication) pool is a template class that stores a single value. It supports the read and write operations, which are guaranteed to be atomic. (On a non-preemptive RTOS every assignment is atomic, but the pool template is still usefull to make it explicit that data is transferred between tasks.) A pool is just a variable.

The example below demonstrates the use of a pool to maintain a seconds-since-startup counter. Note that the call RTOS::runtime() returns the time elapsed since startup, so there is no need to maintain a seconds-since-startup this way yourself.

```
pool< unsigned int > seconds;

void show_time( void ){
   unsigned int n = seconds.read();
   std::cout << ( seconds / 60 ) % 60 << ":" << seconds % 60;
}

class seconds_counter_class : public periodic_task {
   seconds_counter( void ){
      periodic_task::periodic_task( "sec-counter", 10, 1000 MS );
      seconds.write( 0 );
   }
   void main( void ){
      for( ; ; ) {
         (void) wait(); // only one thing to wait for
         seconds.write( seconds.read() + 1 );
      }
   }
}

seconds_counter_class seconds_counter;
```

10.17.2 Constructor & Destructor Documentation

10.17.2.1 template<class T> RTOS::pool< T >::pool (const char * *name* = " ") [inline]

construct a pool, specify its name (for debgging only)

Use it to make (global) variables use for communication between tasks explicit.

The template argument T must be a class that has a non-arguments constructor and supports assignment.

10.17.3 Member Function Documentation

10.17.3.1 template<class T> T RTOS::pool< T >::read (void) [inline]

atomic read operation on a pool

A read opeartion returns the most recently written data.

In the context of co-operative multitasking a read of write operation on anything is always atomic, unless the implementation of that operating somehow invokes the RTOS. But for clearness it is a good idea to implement such task-global data as pools.

10.17.3.2 template<class T> void RTOS::pool< T >::write (T *item*) [inline]

atomic write operation on a pool

A read opeartion returns the most recently written data.

In the context of co-operative multitasking a read of write operation on anything is always atomic, unless the implementation of that operating somehow invokes the RTOS. But for clearness it is a good idea to implement such task-global data as pools.

The documentation for this class was generated from the following file:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h

10.18 RTOS::pool_base Class Reference

RTOS private implementation class.

`#include <pRTOS.h>`

Inheritance diagram for RTOS::pool_base:

Public Member Functions

- **pool_base** (const char *name)
- void **print** (std::ostream &s, bool header=true) const

Public Attributes

- unsigned int **reads**
- unsigned int **writes**
- pool_base * **next_pool**
- const char * **pool_name**

Friends

- class **RTOS**

10.18.1 Detailed Description

RTOS private implementation class.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.cpp

10.19 Pump Class Reference

Public Member Functions

- Pump ()
- char * getOnCommand ()

  Returns the bytes to set the pump on.
- char * getOffCommand ()

  Returns the bytes to set the pump off.
- char * getStatusCommand ()

  Returns the bytes to get the status of the pump.

10.19.1 Detailed Description

Used as an interface to get the appropiate bytes for the uart

10.19.2 Constructor & Destructor Documentation

10.19.2.1 Pump::Pump ( )

file Heater.cpp version 0.1 author Remco Nijkamp / Jordan Ranirez / Kevin Damen / Jeroen Kok date 19-01-2016

10.19.3 Member Function Documentation

10.19.3.1 char * Pump::getOffCommand ( )

Returns the bytes to set the pump off.

Returns

a char pointer to a 2 char array

10.19.3.2 char * Pump::getOnCommand ( )

Returns the bytes to set the pump on.

Returns

a char pointer to a 2 char array

10.19.3.3 char * Pump::getStatusCommand ( )

Returns the bytes to get the status of the pump.

Returns

a char pointer to a 2 char array

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/Pump.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/Pump.cpp

10.20 RTOS Class Reference

static class, namespace-like container for RTOS declarations

`#include <pRTOS.h>`

Classes

- class channel
  waitable data queue
- class channel_base
  RTOS private implementation class.
- class clock
  free-running clock, ticks at a fixed frequency
- class event
  set of things that can happen, or a thing that has happened
- class flag
  basic synchronisation mechanism.
- class mailbox
  Synchronously handing over of a data item.
- class mailbox_base
  RTOS private implementation class.
- class mutex
  mutual execlusion semaphore
- class pool
  place to store and rectrieve data, no built-in synchronisation
- class pool_base
  RTOS private implementation class.
- class task
  an independent thread of execution
- class timer
  one-short timer
- class waitable
  abstract thing that a task can wait for

Static Public Member Functions

- static void run (void)

  runs the scheduler
- static task * current_task (void)

  returns (a pointer to) the currently executing task
- static unsigned long long int run_time (void)

  get elapsed time in micro seconds since OS startup
- static void print (std::ostream &stream)

  prints statistics about the RTOS to the stream.
- static void statistics_clear (void)

  clears the statistics.
- static void display_statistics (void)

  print the statistics collect for the used RTOS objects

10.20.1 Detailed Description

static class, namespace-like container for RTOS declarations

The RTOS is a static class, instantiation is not needed. After creating the tasks, call RTOS::run() to start the scheduling of the tasks. RTOS::run() will never return.

10.20.2 Member Function Documentation

10.20.2.1 static void RTOS::statistics_clear (void) [inline], [static]

clears the statistics.

The actual clearing will be done later, inside run(), when the current task has given up the processor.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.cpp

10.21 SoapDispenser Class Reference

Public Member Functions

- SoapDispenser ()
- char * getOpenCommand ()

  Returns the bytes used to open the soap dispenser.
- char * getCloseCommand ()

  Returns the bytes used to close the soap dispenser.
- char * getStatusCommand ()

  Returns the bytes used to get the status of the soap dispenser.

10.21.1 Detailed Description

Used as an interface to get the appropiate bytes for the uart

10.21.2 Constructor & Destructor Documentation

10.21.2.1 SoapDispenser::SoapDispenser ( )

file Heater.cpp version 0.1 author Remco Nijkamp / Jordan Ranirez / Kevin Damen / Jeroen Kok date 19-01-2016

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/SoapDispenser.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/SoapDispenser.cpp

10.22 Socket Class Reference

Inheritance diagram for Socket:

Public Member Functions

- SocketAddress getLocalAddress () throw (SocketException)
- void close ()

Static Public Member Functions

- static void cleanUp () throw (SocketException)

Protected Member Functions

- Socket ()
- void **createSocket** (const SocketAddress &address, int type, int protocol) throw (SocketException)

Protected Attributes

- int sockDesc

10.22.1 Constructor & Destructor Documentation

10.22.1.1 Socket::Socket ( ) [protected]

You can only construct this object via a derived class.

10.22.2 Member Function Documentation

10.22.2.1 void Socket::cleanUp ( ) throw SocketException) [static]

If WinSock, unload the WinSock DLLs; otherwise do nothing. We ignore this in our sample client code but include it in the library for completeness. If you are running on Windows and you are concerned about DLL resource consumption, call this after you are done with all Socket instances. If you execute this on Windows while some instance of Socket exists, you are toast. For portability of client code, this is an empty function on non-Windows platforms so you can always include it.

Parameters

buffer	buffer to receive the data
bufferLen	maximum number of bytes to read into buffer

Returns

number of bytes read, 0 for EOF, and -1 for error

Exceptions

SocketException	thrown WinSock clean up fails

10.22.2.2 void Socket::close (void)

Close this socket.

10.22.2.3 SocketAddress Socket::getLocalAddress ( ) throw SocketException)

Get the local address

Returns

local address of socket

Exceptions

SocketException	thrown if fetch fails

10.22.3 Member Data Documentation

10.22.3.1 int Socket::sockDesc [protected]

Socket descriptor, protected so derived classes can read it easily (may want to change this)

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.cpp

10.23 SocketAddress Class Reference

```
#include <PracticalSocket.h>
```

Public Types

- enum AddressType { **TCP_SOCKET**, **TCP_SERVER**, **UDP_SOCKET** }

Public Member Functions

- SocketAddress (const char *host, const char *service, AddressType atype=TCP_SOCKET) throw (Socket↩ Exception)
- SocketAddress (const char *host, in_port_t port, AddressType atype=TCP_SOCKET) throw (Socket↩ Exception)
- SocketAddress (sockaddr *addrVal=NULL, socklen_t addrLenVal=0)
- std::string getAddress () const throw (SocketException)
- in_port_t getPort () const throw (SocketException)
- sockaddr * getSockaddr () const
- socklen_t getSockaddrLen () const

Static Public Member Functions

- static std::vector< SocketAddress > lookupAddresses (const char *host, const char *service, AddressType atype=TCP_SOCKET) throw (SocketException)
- static std::vector< SocketAddress > lookupAddresses (const char *host, in_port_t port, AddressType atype=TCP_SOCKET) throw (SocketException)

10.23.1 Detailed Description

Container aggregating an address and a port for a socket. SocketAddress offers value semantics.

10.23.2 Member Enumeration Documentation

10.23.2.1 enum SocketAddress::AddressType

Type of address being requested.

10.23.3 Constructor & Destructor Documentation

10.23.3.1 SocketAddress::SocketAddress (const char * *host*, const char * *service*, AddressType *atype* = TCP_SOCKET) throw SocketException)

Make a SocketAddress for the given host and service.

10.23.3.2 SocketAddress::SocketAddress (const char * *host*, in_port_t *port*, AddressType *atype* = TCP_SOCKET) throw SocketException)

Make a SocketAddress for the given host and port number.

10.23.3.3 SocketAddress::SocketAddress (sockaddr * *addrVal* = NULL, socklen_t *addrLenVal* = 0)

Make a SocketAddress that wraps a copy of the given sockaddr structure of the given addreLenVal legth in bytes. If used as a default constructur, the SocketAddress is created in an uninitialized state, and none of its get methods should be used until it is initialized.

10.23.4 Member Function Documentation

10.23.4.1 string SocketAddress::getAddress ( ) const throw SocketException)

Return a string representation of the address portion of this object.

10.23.4.2 in_port_t SocketAddress::getPort ( ) const throw SocketException)

Return a numeric value for the port portion of this object.

10.23.4.3 sockaddr* SocketAddress::getSockaddr ( ) const [inline]

Return a pointer to the sockaddr structure wrapped by this object.

10.23.4.4 socklen_t SocketAddress::getSockaddrLen ( ) const [inline]

Return the length of the sockaddr structure wrapped by this object.

10.23.4.5 vector< SocketAddress > SocketAddress::lookupAddresses (const char * *host*, const char * *service*, AddressType *atype* = TCP_SOCKET) throw SocketException) [static]

Return a list of all matching addresses for the given host and service. Either, but not both of host and service can be null. The returned list of addresses may be empty.

10.23.4.6 vector< SocketAddress > SocketAddress::lookupAddresses (const char * *host*, in_port_t *port*, AddressType *atype* = TCP_SOCKET) throw SocketException) [static]

Return a list of all matching addresses for the given host and port. Either, but not both of host and service can be null (or zero). The returned list of addresses may be empty.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.cpp

10.24 SocketException Class Reference

`#include <PracticalSocket.h>`

Inheritance diagram for SocketException:

Public Member Functions

- SocketException (const std::string &message) throw ()
- SocketException (const std::string &message, const std::string &detail) throw ()

10.24.1 Detailed Description

Signals a problem with the execution of a socket call.

10.24.2 Constructor & Destructor Documentation

10.24.2.1 SocketException::SocketException (const std::string & *message*) throw)

Construct a SocketException with a user message followed by a system detail message.

Parameters

message	explanatory message

10.24.2.2 SocketException::SocketException (const std::string & *message*, const std::string & *detail*) throw)

Construct a SocketException with a explanatory message.

Parameters

message	explanatory message
detail	detail message

The documentation for this class was generated from the following file:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.h

10.25 SocketStreamBuffer< CharT, Traits > Class Template Reference

Inheritance diagram for SocketStreamBuffer< CharT, Traits >:

Public Types

- typedef Traits::int_type **int_type**

Public Member Functions

- **SocketStreamBuffer** (TCPSocket ∗sock)

Protected Member Functions

- int_type **overflow** (int_type c=Traits::eof())
- int **sync** ()
- int_type **underflow** ()

10.25.1 Detailed Description

template<class CharT, class Traits = std::char_traits<CharT>>
class SocketStreamBuffer< CharT, Traits >

Subclass of basic_streambuf for reading and writing to instances of TCPSocket.

The documentation for this class was generated from the following file:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.cpp

10.26 SocketTimedOutException Class Reference

`#include <PracticalSocket.h>`

Inheritance diagram for SocketTimedOutException:

Public Member Functions

- SocketTimedOutException (const std::string &message) throw ()

10.26.1 Detailed Description

Signals a time out .

10.26.2 Constructor & Destructor Documentation

10.26.2.1 SocketTimedOutException::SocketTimedOutException (const std::string & *message*) throw)

Construct a SocketTimedOutException with a user message followed by a system detail message.

Parameters

message	explanatory message

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.cpp

10.27 RTOS::task Class Reference

an independent thread of execution

`#include <pRTOS.h>`

Inheritance diagram for RTOS::task:

Public Member Functions

- task (unsigned int priority=RTOS_MIN_PRIORITY, const char *tname="", unsigned int stacksize=RTOS_↩ DEFAULT_STACK_SIZE)

  constructor, specify priority, name and stack size
- ~task (void)

  throws an error, beacuse tasks should never be destroyed
- virtual void suspend (void)

  suspend a task (prevent execution until a resume)
- virtual void resume (void)

  continue a suspended task
- void release (void)

  release the CPU to the scheduler
- void sleep (unsigned int time)

  wait for some time
- unsigned int priority (void) const

  report the task priority
- const char * name (void) const

  report the task name
- bool is_suspended (void) const

  report whether the task is currently suspended
- bool is_blocked (void) const

  report whether the task is currently blocked
- bool is_ready (void) const

  report whether the task is currently ready for execution
- void print (std::ostream &stream, bool header=true) const

  print task statistics
- event wait (void)

  wait for all waitables created for this task

- event wait (const waitable &w)

  wait for a single waitable
- event wait (const event &set)

  wait for a set of waitables
- void set (flag &f)

  set a flag
- void ignore_activation_time (void)

  ignore this activation for the statistics

Protected Member Functions

- virtual void main (void)=0

  task body, must be provided by a derived class

Friends

- class **periodic_task**
- class **waitable_set**
- class **flag**
- class **RTOS**
- void **task_trampoline** (void)

10.27.1 Detailed Description

an independent thread of execution

A task is an independent thread of execution, using its own stack. Tasks share the single CPU, so only one task can be running at any time. The RTOS determines which task is running. A task has two bi-value states that determine whether the task is runnable: the suspended/resumed flag and the waiting/non-waiting flag. A task is runnable only when it is both resumed, and non-waiting. When a task is created it is resumed and non-waiting. All tasks (and the RTOS code) run in the same memory space, without protection from each other. So a 'wild pointer' in one task can destroy data in another task, or even in the RTOS.

Each task is created with a fixed priority, which can be any unsigned integer value below RTOS_MIN_PRIORITY (= 98). After creation the priority can not be changed. The value 0 indicates the highest task priority, a higher number indicates a lower priority. Each task must have a unqiue priority, it is an error to create a task with same priority as an existing task. You can omit the priority, in which case the RTOS will select an unused priority starting at RTOS_MIN_PRIORITY (in other words, it will choose a low priority for your task).

Each task has its own stack. You can specify the size of the stack at task creation. If you omit the stack size, RTOS_DEFAULT_STACK_SIZE will be used (default: 4 Kb). This will be enough for most tasks, if you take care not to allocate big things on the stack, and avoid very deep nesting (watch out for recursion!).

A task is created by instatiating a class that derives from RTOS::task and supplies a main(). This main() should never return. The fragment below shows how you can do this. The task name is used for statistics and debugging. As shown for the name, it might be wise to get the task parameters as arguments to the constructor of your task.

```
class my_task_class : public RTOS::task {
public:
   my_task_class( const char * name ):
      task(
         name,  // name of the task
         10,    // task priority
         16384  // task stack size
      ){}
private:
   void main( void ){
      // put the code of your task here
   }
};
my_task_class my_task( "my first task" );
my_task_class my_task( "my second task" );
```

The example below is a complete program that shows the standard part (initialization, and a main that calls RTOS::run()), a function for writing to an individual LED, a task class that blinks a LED, and two instatiations of this class. Note that the sleep() call is used instead of mkt_wait_ms or mkt_wait_us. Sleep() causes other tasks to run while this task is waiting, whereas the mkt_wait_* calls would use monopolize the CPU to do a busy wait.

Subsequent examples will not show the standard initialization (the part up to the comment line).

```
#include "pRTOS.h"

int main( void ) {
   RTOS::run();
   return 0;
}

// end of standard part

class blinker : public RTOS::task {
public:
   blinker( int LED, int period ):
      LED( LED ), period( period ){}
private:
   int LED, period;
   void main( void ){
      for( ; ; ) {
         led_set( LED, 1 );
         sleep( period / 2 );
         led_set( LED, 0 );
         sleep( period / 2 );
      }
   }
};

blinker led0( 0, 1000 MS );
blinker led1( 1,  300 MS );
```

A task can be suspended and resumed by the task::suspend() and task::resume() calls. The suspend/resume state does not count the number of suspends and resumes: a suspend call on an already suspended task (or a resume on an already resumed task) has no effect. Suspend and resume are intended for use by the application code: the RTOS will never suspend or resume a task. (The RTOS uses the waiting/non-waiting state, which can not be changed directly by the application.)

The example below shows one task that beeps the speaker at 1 kHz, while the other taks suspends and resumes the first task to make it beep 5 times, after which it suspends itself, which ends all activity. (This will trigger the RTOS deadlock detection, because a normal embedded application should never terminate.)

```
class beeper : public RTOS::task {
public:
   unsigned int speaker;
   beeper( unsigned int speaker ): speaker( speaker ){}
   void main( void ){
      mkt_pin_configure( speaker, mkt_output);
      for( ; ; ) {
         mkt_pin_write( speaker, 1  );
         sleep( 500 US );
         mkt_pin_write( speaker, 0  );
         sleep( 500 US );
      }
```

```
   }
};
beeper speaker( 10 );

class suspender : public RTOS::task {
   void main( void ){
      for( int i = 0; i < 5 ; i++ ) {
         speaker.resume();
         sleep( 500 MS );
         speaker.suspend();
         sleep( 1 S );
      }
      suspend();
   }
};
suspender task2;
```

10.27.2 Constructor & Destructor Documentation

10.27.2.1 RTOS::task::task (unsigned int *priority* = RTOS_MIN_PRIORITY, const char * *tname* = " ", unsigned int *stacksize* = RTOS_DEFAULT_STACK_SIZE)

constructor, specify priority, name and stack size

Priorities are reasonably-valued (below RTOS_MIN_PRIORITY) unsigned integers. 0 is te highest priority. Priorities must be unqiue. The default causes the constructor to choose a free priority starting at RTOS_MIN_PRIORITY (default: 1000).

The name is used for debugging and statistics.

A stack of stack_size bytes is allocated for the task. The default is 4 kB.

10.27.3 Member Function Documentation

10.27.3.1 void RTOS::task::ignore_activation_time (void) [inline]

ignore this activation for the statistics

Calling this function makes the RTOS statistics ignore the current task activation as far as statistics is concerned. You can use this to avoid pullution of your task statistics with the timing effects of debug logging. But make sure you don't use it in the 'normal' execution paths, because that would make the statistics lie to you.

10.27.3.2 virtual void RTOS::task::main (void) [protected], [pure virtual]

task body, must be provided by a derived class

A task is created by inheriting from task and providing a main() function. Initialisation of the task, including creating its waitables, should be done in the constructor. Don't forget to call the constructor of the task class!

The main() is the body of the task. It should never terminate.

Each task has a unique priority (an unsigned integer). A lower value indicates a higher priority. The RTOS scheduler will always run the task with the higest-priority runnable (neither blocked nor suspended) task. A task runs until it changes this 'situation' by using an RTOS call that changes its own state to not runnable, or the state of a higher priority task to runnable.

Timers are served only when the RTOS is activated by calling any of its state-changing interfaces. Hence the longest run time between such calls determines the granularity (time wise responsiveness) of the application. Within a time consuming computation a task can call release() to have the RTOS serve the timers.

Implemented in WashingMachineController, MotorController, WaterController, TempController, and UART.

10.27.3.3 void RTOS::task::release (void)

release the CPU to the scheduler

Sevices timers and releases the CPU to a higher priority task if is ready.

10.27.3.4 void RTOS::task::resume (void) [virtual]

continue a suspended task

Has no effect when the task is not suspended.

Can be extended by an application task to suit its needs.

10.27.3.5 void RTOS::task::sleep (unsigned int *time*)

wait for some time

Sleeps the task (prevents execution) for the indicated time.

10.27.3.6 void RTOS::task::suspend (void) [virtual]

suspend a task (prevent execution until a resume)

Suspends the task (prevents execution). Has no effect when the task is already suspended. Can be extended by an application task.

A concrete task can extend this operation to suit its needs.

10.27.3.7 event RTOS::task::wait (void) [inline]

wait for all waitables created for this task

Wait (prevent execution) until at least one of the waitables is set. Return and clear that waitable. Three variants for the parameter:

- The default (no parameter) waits for all waitables defined for the task.
- One waitable as argument waits for that specific waitable.
- The addition (operator+) of waitables: wait for any one of those waitables.

It is an error to wait for waitables that have not been created for this task.

10.27.3.8 event RTOS::task::wait (const waitable & *w*) [inline]

wait for a single waitable

Wait (prevent execution) until at least one of the waitables is set. Return and clear that waitable. Three variants for the parameter:

- The default (no parameter) waits for all waitables defined for the task.
- One waitable as argument waits for that specific waitable.
- The addition (operator+) of waitables: wait for any one of those waitables.

It is an error to wait for waitables that have not been created for this task.

10.27.3.9 event RTOS::task::wait (const event & *set*) [inline]

wait for a set of waitables

Wait (prevent execution) until at least one of the waitables is set. Return and clear that waitable. Three variants for the parameter:

- The default (no parameter) waits for all waitables defined for the task.
- One waitable as argument waits for that specific waitable.
- The addition (operator+) of waitables: wait for any one of those waitables.

It is an error to wait for waitables that have not been created for this task.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.cpp

10.28 TCPServer Class Reference

Public Member Functions

- void **runTCPServer** (WasmachineApp *wasapp, int port)

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/TCPServer.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/TCPServer.cpp

10.29 TCPServerSocket Class Reference

`#include <PracticalSocket.h>`

Inheritance diagram for TCPServerSocket:

Public Member Functions

- TCPServerSocket ()
- TCPServerSocket (in_port_t localPort, int queueLen=5) throw (SocketException)
- void bind (const SocketAddress &localAddress) throw (SocketException)
- TCPSocket * accept () throw (SocketException)

Additional Inherited Members

10.29.1 Detailed Description

TCP socket class for servers

10.29.2 Constructor & Destructor Documentation

10.29.2.1 TCPServerSocket::TCPServerSocket ( )

Make an unbound socket.

10.29.2.2 TCPServerSocket::TCPServerSocket (in_port_t *localPort,* int *queueLen* = 5) throw SocketException)

Construct a TCP socket for use with a server, accepting connections on the specified port on any interface

Parameters

localPort	local port of server socket, a value of zero will give a system-assigned unused port
queueLen	maximum queue length for outstanding connection requests (default 5)

Exceptions

SocketException	thrown if unable to create TCP server socket

10.29.3 Member Function Documentation

10.29.3.1 TCPSocket * TCPServerSocket::accept ( ) throw SocketException)

Blocks until a new connection is established on this socket or error

Returns

new connection socket

Exceptions

SocketException	thrown if attempt to accept a new connection fails

10.29.3.2 void TCPServerSocket::bind (const SocketAddress & *localAddress*) throw SocketException)

Bind this socket to the given local address.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.cpp

10.30 TCPSocket Class Reference

`#include <PracticalSocket.h>`

Inheritance diagram for TCPSocket:

Public Member Functions

- TCPSocket ()
- TCPSocket (const char *foreignAddress, in_port_t foreignPort) throw (SocketException)
- void bind (const SocketAddress &localAddress) throw (SocketException)
- void connect (const SocketAddress &foreignAddress) throw (SocketException)
- std::iostream & getStream () throw (SocketException)

Friends

- class **TCPServerSocket**

Additional Inherited Members

10.30.1 Detailed Description

TCP socket for communication with other TCP sockets

10.30.2 Constructor & Destructor Documentation

10.30.2.1 TCPSocket::TCPSocket ( )

Make a socket that is neither bound nor connected.

10.30.2.2 TCPSocket::TCPSocket (const char * *foreignAddress*, in_port_t *foreignPort*) throw SocketException)

Construct a TCP socket with a connection to the given foreign address and port. This is interface is provided as a convience for typical applications that don't need to worry about the local address and port.

Parameters

foreignAddress	foreign address (IP address or name)
foreignPort	foreign port

Exceptions

SocketException	thrown if unable to create TCP socket

10.30.3 Member Function Documentation

10.30.3.1 void TCPSocket::bind (const SocketAddress & *localAddress*) throw SocketException)

Bind this socket to the given local address.

10.30.3.2 void TCPSocket::connect (const SocketAddress & *foreignAddress*) throw SocketException)

Connect this socket to the given foreign address.

10.30.3.3 iostream & TCPSocket::getStream ( ) throw SocketException)

Return a reference to an I/O stream wrapper around this CommunicatingSocket. The caller can use this object to send and receive text-encoded messages over the socket. The returned stream is owned by the socket and is created on the first call to getStream.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.cpp

10.31 TempController Class Reference

Inheritance diagram for TempController:

Public Member Functions

- **TempController** (WashingMachineController *wascontroller)
- void setUartPointer (UART *u)

  Used to (re)set the pointer to the uart.
- void setTemp (int temp)

  sets the temprature of the emulator
- void setResponseFlag ()

  sets the response flag
- void writeResponse (char *response)

  writes a response in the response pool
- void main ()

  task body, must be provided by a derived class

Additional Inherited Members

10.31.1 Member Function Documentation

10.31.1.1 void TempController::main ( ) [virtual]

task body, must be provided by a derived class

A task is created by inheriting from task and providing a main() function. Initialisation of the task, including creating its waitables, should be done in the constructor. Don't forget to call the constructor of the task class!

The main() is the body of the task. It should never terminate.

Each task has a unique priority (an unsigned integer). A lower value indicates a higher priority. The RTOS scheduler will always run the task with the higest-priority runnable (neither blocked nor suspended) task. A task runs until it changes this 'situation' by using an RTOS call that changes its own state to not runnable, or the state of a higher priority task to runnable.

Timers are served only when the RTOS is activated by calling any of its state-changing interfaces. Hence the longest run time between such calls determines the granularity (time wise responsiveness) of the application. Within a time consuming computation a task can call release() to have the RTOS serve the timers.

Implements RTOS::task.

10.31.1.2 void TempController::setResponseFlag ( )

sets the response flag

Returns

void

10.31.1.3 void TempController::setTemp (int *temp*)

sets the temprature of the emulator

Parameters

temp	in celcius degrees

Returns

void

10.31.1.4 void TempController::setUartPointer (UART * *u*)

Used to (re)set the pointer to the uart.

Parameters

*u**	pointer to the UART object this controller should use.

10.31.1.5 void TempController::writeResponse (char * *response*)

writes a response in the response pool

Parameters

response	a char array with two positions

Returns

void

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/TempController.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/TempController.cpp

10.32 TempSensor Class Reference

Public Member Functions

- TempSensor ()
- char * getTempCommand ()

  Returns the bytes used to get the temprature from the temprature sensor.

10.32.1 Detailed Description

Used as an interface to get the appropiate bytes for the uart

10.32.2 Constructor & Destructor Documentation

10.32.2.1 TempSensor::TempSensor ( )

file Heater.cpp version 0.1 author Remco Nijkamp / Jordan Ranirez / Kevin Damen / Jeroen Kok date 19-01-2016

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/TempSensor.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/TempSensor.cpp

10.33 The Class Reference

10.33.1 Detailed Description

class

Makes sure the motor is rotating at the right RPM in the right direction (or in the left direction)

class

Makes sure that the temprature is right.

class

class

Makes sure the water is at the right level

The documentation for this class was generated from the following file:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/MotorController.h

10.34 RTOS::timer Class Reference

one-short timer

```
#include <pRTOS.h>
```

Inheritance diagram for RTOS::timer:

Public Member Functions

- timer (task *t, const char *name="")
  create a timer for task t, specify its name
- void set (unsigned long int time)
- void cancel (void)
  stop and clear the timer
- void print (std::ostream &s, bool header=true) const
  print the timer (for debugging)

Friends

- class **RTOS**

Additional Inherited Members

10.34.1 Detailed Description

one-short timer

A (one-shot) timer is a special type of flag, which can be instructed to set itself after a fixed amount of time. The amount of time is supplied with the timer::set() call. This call starts the timer. A timer that is running (waiting for its timeout to expire) can be canceled by the timer::cancel() call. When a timer that is already running is set again the previous timeout is overwritten by the new one. The suspend/resume state of its owner taks has no effect on a timer: even when the task is suspended the timer will run to its timeout and set isetlf. But of course the task, being suspended, will not be able to react.

10.34.2 Member Function Documentation

10.34.2.1 void RTOS::timer::cancel (void)

stop and clear the timer

Stop the timer (when it was running), and clears its (when it was set).

10.34.2.2 void RTOS::timer::set (unsigned long int *time*)

Start the timer: it will set itself after the indicated timeout, starting from now. When the timer was already running the previous timout is overwritten.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.cpp

10.35 UART Class Reference

Inheritance diagram for UART:

Public Member Functions

- **UART** (const char *device, unsigned int baudrate, MotorController *motorctrl, TempController *tempctrl, WaterController *waterctrl, WashingMachineController *wasctrl)
- void **executeCommand** (char *s)
- void **writeChannel** (char *request)
- void main ()

  task body, must be provided by a derived class

Additional Inherited Members

10.35.1 Member Function Documentation

10.35.1.1 void UART::main ( ) [virtual]

task body, must be provided by a derived class

A task is created by inheriting from task and providing a main() function. Initialisation of the task, including creating its waitables, should be done in the constructor. Don't forget to call the constructor of the task class!

The main() is the body of the task. It should never terminate.

Each task has a unique priority (an unsigned integer). A lower value indicates a higher priority. The RTOS scheduler will always run the task with the higest-priority runnable (neither blocked nor suspended) task. A task runs until it changes this 'situation' by using an RTOS call that changes its own state to not runnable, or the state of a higher priority task to runnable.

Timers are served only when the RTOS is activated by calling any of its state-changing interfaces. Hence the longest run time between such calls determines the granularity (time wise responsiveness) of the application. Within a time consuming computation a task can call release() to have the RTOS serve the timers.

Implements RTOS::task.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/UART.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/UART.cpp

10.36 uart_error Class Reference

Inheritance diagram for uart_error:

Public Member Functions

- **uart_error** (const std::string &error)
- const char * **what** () const override

10.36.1 Detailed Description

file Heater.cpp version 0.1 author Remco Nijkamp / Jordan Ranirez / Kevin Damen / Jeroen Kok date 19-01-2016

The documentation for this class was generated from the following file:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/UART.cpp

10.37 UDPSocket Class Reference

`#include <PracticalSocket.h>`

Inheritance diagram for UDPSocket:

Public Member Functions

- UDPSocket () throw (SocketException)
- void **bind** (const SocketAddress &localAddress) throw (SocketException)
- void **connect** (const SocketAddress &foreignAddress) throw (SocketException)
- void disconnect () throw (SocketException)
- void sendTo (const void *buffer, int bufferLen, const SocketAddress &foreignAddress) throw (Socket↩ Exception)
- int recvFrom (void *buffer, int bufferLen, SocketAddress &sourceAddress) throw (SocketException)
- void setMulticastTTL (unsigned char multicastTTL) throw (SocketException)
- void joinGroup (const std::string &multicastGroup) throw (SocketException)
- void leaveGroup (const std::string &multicastGroup) throw (SocketException)
- void setBroadcast () throw (SocketException)
- void setMulticastLoop (bool loop) throw (SocketException)
- void setTimeOut (int sec) throw (SocketException)

Additional Inherited Members

10.37.1 Detailed Description

UDP socket class

10.37.2 Constructor & Destructor Documentation

10.37.2.1 UDPSocket::UDPSocket ( ) throw SocketException)

Construct a UDP socket

Exceptions

SocketException	thrown if unable to create UDP socket

10.37.3 Member Function Documentation

10.37.3.1 void UDPSocket::disconnect ( ) throw SocketException)

Unset foreign address and port

Returns

true if disassociation is successful

Exceptions

SocketException	thrown if unable to disconnect UDP socket

10.37.3.2 void UDPSocket::joinGroup (const std::string & *multicastGroup*) throw SocketException)

Join the specified multicast group

Parameters

multicastGroup	multicast group address to join

Exceptions

SocketException	thrown if unable to join group

10.37.3.3 void UDPSocket::leaveGroup (const std::string & *multicastGroup*) throw SocketException)

Leave the specified multicast group

Parameters

multicastGroup	multicast group address to leave

Exceptions

SocketException	thrown if unable to leave group

10.37.3.4 int UDPSocket::recvFrom (void * *buffer,* int *bufferLen,* SocketAddress & *sourceAddress*) throw SocketException)

Read read up to bufferLen bytes data from this socket. The given buffer is where the data will be placed

Parameters

buffer	buffer to receive data
bufferLen	maximum number of bytes to receive
sourceAddress	address of datagram source
sourcePort	port of data source

Returns

number of bytes received and -1 for error

Exceptions

SocketException	thrown if unable to receive datagram
SocketTimedOutException	thrown after time out period has elapsed

10.37.3.5 void UDPSocket::sendTo (const void * *buffer*, int *bufferLen*, const SocketAddress & *foreignAddress*) throw SocketException)

Send the given buffer as a UDP datagram to the specified address/port

Parameters

buffer	buffer to be written
bufferLen	number of bytes to write
foreignAddress	address to send to

Exceptions

SocketException	thrown if unable to send datagram

10.37.3.6 void UDPSocket::setBroadcast ( ) throw SocketException)

Allow the socket to send broadcast

Exceptions

SocketException	thrown if unable to set broadcast allowance

10.37.3.7 void UDPSocket::setMulticastLoop (bool *loop*) throw SocketException)

Enables or disables the socket to recieve the multicast packets it sends.

Parameters

loop	true means enabling, false disabling

Exceptions

SocketException	thrown if unable to set broadcast allowance

10.37.3.8 void UDPSocket::setMulticastTTL (unsigned char *multicastTTL*) throw SocketException)

Set the multicast TTL

Parameters

multicastTTL	multicast TTL

Exceptions

SocketException	thrown if unable to set TTL

10.37.3.9 void UDPSocket::setTimeOut (int *sec*) throw SocketException)

Set time out period, i.e. the maximum amount of time method recvFrom wil wait.

Parameters

time	recvFrom wil wait in sec.

Exceptions

SocketException	thrown if unable to set time out.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/PracticalSocket.cpp

10.38 Valve Class Reference

Public Member Functions

- Valve ()
- char * getOnCommand ()

  Returns the bytes to set the valve open.
- char * getOffCommand ()

  Returns the bytes to set the valve close.
- char * getStatusCommand ()

  Returns the bytes to get the status of the valve.

10.38.1 Detailed Description

Used as an interface to get the appropiate bytes for the uart

10.38.2 Constructor & Destructor Documentation

10.38.2.1 Valve::Valve ( )

file Heater.cpp version 0.1 author Remco Nijkamp / Jordan Ranirez / Kevin Damen / Jeroen Kok date 19-01-2016

10.38.3 Member Function Documentation

10.38.3.1 char * Valve::getOffCommand ( )

Returns the bytes to set the valve close.

Returns

a char pointer to a 2 char array

10.38.3.2 char * Valve::getOnCommand ( )

Returns the bytes to set the valve open.

Returns

a char pointer to a 2 char array

10.38.3.3 char * Valve::getStatusCommand ( )

Returns the bytes to get the status of the valve.

Returns

a char pointer to a 2 char array

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/Valve.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/Valve.cpp

10.39 RTOS::waitable Class Reference

abstract thing that a task can wait for

```
#include <pRTOS.h>
```

Inheritance diagram for RTOS::waitable:

Public Member Functions

- virtual void clear (void)

  clear the waitable

Protected Member Functions

- waitable (task *task, const char *name)

  constructor, specify owner and name
- void set (void)

  set the waitable

Friends

- class **waitable_set**

Additional Inherited Members

10.39.1 Detailed Description

abstract thing that a task can wait for

The operation clear() is provided (virtual, the default only clears the waitable) set() is provided but private (not all waitables can be set by the user).

Waitable is an abstract class (there are no objects that are just a waitable). flag, timer, clock and channel are concrete classes that inherit from waitable. A waitable is always created for a particular task. A maximum of 31 waitables can be created for each task. (Actually the maximum is 32, but one waitable created internally to implement the sleep() call.) A waitable can be in two states: set or cleared. A waitable is initially cleared.

A task can wait for one, a subset, or all waitables created for it. The default is to wait for all waitables created for the task, the other variants are specified by supplying to the task:wait() call either a single waitable, or the sum (operator+) of the waitables you want to wait for. When one of the waitables that is waited for becomes set the wait() call clears that waitable and returns an event that compares equal to the waitable. (Note that some waitables, for instance the channel, can immediately set itself again.) The calling task can compare that event to the waitables to see which event happened. When more than one of the waited-for waitables is set the wait() call makes an arbitrary choice from these waitables.

10.39.2 Constructor & Destructor Documentation

10.39.2.1 RTOS::waitable::waitable (task * *task*, const char * *name*) [protected]

constructor, specify owner and name

The name is used for debugging only.

10.39.3 Member Function Documentation

10.39.3.1 virtual void RTOS::waitable::clear (void) [inline], [virtual]

clear the waitable

This is automatically doen when the waitable causes a task::wait() call to return it.

Reimplemented in RTOS::channel< T, SIZE >, RTOS::channel< char *, 100 >, and RTOS::clock.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/pRTOS.cpp

10.40 WashingMachineController Class Reference

Inheritance diagram for WashingMachineController:

Public Member Functions

- WashingMachineController (Wasprogramma &was)

  Creates the controller and assignes a washingschedule.
- void setTempReached ()

  Sets the temprature as reached.
- void setWaterLevelReached ()

  Sets the waterlevel as reached.
- void setMotorDone ()

  Sets the motor as done with the job.
- void startWasprogramma ()

  Starts the washing schedule.
- void stopWasprogramma ()

  Stops the washing schedule.
- void setResponseFlag ()

  sets the response flag
- void writeResponse (char *response)

  writes a response in the response pool
- void main ()

  task body, must be provided by a derived class

Additional Inherited Members

10.40.1 Constructor & Destructor Documentation

10.40.1.1 WashingMachineController::WashingMachineController (Wasprogramma & *was*)

Creates the controller and assignes a washingschedule.

file Heater.cpp version 0.1 author Remco Nijkamp / Jordan Ranirez / Kevin Damen / Jeroen Kok date 19-01-2016

Returns

void

10.40.2 Member Function Documentation

10.40.2.1 void WashingMachineController::main ( ) [virtual]

task body, must be provided by a derived class

A task is created by inheriting from task and providing a main() function. Initialisation of the task, including creating its waitables, should be done in the constructor. Don't forget to call the constructor of the task class!

The main() is the body of the task. It should never terminate.

Each task has a unique priority (an unsigned integer). A lower value indicates a higher priority. The RTOS scheduler will always run the task with the higest-priority runnable (neither blocked nor suspended) task. A task runs until it changes this 'situation' by using an RTOS call that changes its own state to not runnable, or the state of a higher priority task to runnable.

Timers are served only when the RTOS is activated by calling any of its state-changing interfaces. Hence the longest run time between such calls determines the granularity (time wise responsiveness) of the application. Within a time consuming computation a task can call release() to have the RTOS serve the timers.

Implements RTOS::task.

10.40.2.2 void WashingMachineController::setMotorDone ( )

Sets the motor as done with the job.

Returns

void

10.40.2.3 void WashingMachineController::setResponseFlag ( )

sets the response flag

Returns

void

10.40.2.4 void WashingMachineController::setTempReached ( )

Sets the temprature as reached.

Returns

void

10.40.2.5 void WashingMachineController::setWaterLevelReached ( )

Sets the waterlevel as reached.

Returns

void

10.40.2.6 void WashingMachineController::startWasprogramma ( )

Starts the washing schedule.

Returns

void

10.40.2.7 void WashingMachineController::stopWasprogramma ( )

Stops the washing schedule.

Returns

void

10.40.2.8 void WashingMachineController::writeResponse (char * *response*)

writes a response in the response pool

Parameters

response	a char array with two positions

Returns

void

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/WashingMachine↩ Controller.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/WashingMachine↩ Controller.cpp

10.41 WasmachineApp Class Reference

Inheritance diagram for WasmachineApp:

Public Member Functions

- **WasmachineApp** (Broadcaster ∗broadcaster)
- void onTextMessage (const string &msg, WebSocket ∗ws) override
- void **sendTextMessage** (const string &msg, WebSocket ∗ws)
- void onClose (WebSocket ∗ws) override
- void **broadcastMessage** (const string &msg)
- Broadcaster ∗ **getBroadcaster** ()

10.41.1 Member Function Documentation

10.41.1.1 void WasmachineApp::onClose (WebSocket ∗ *ws*) [override], [virtual]

Will be called when the websocket has been close.

Parameters

ws	the websocket that has been closed

Implements WebSocketListener.

10.41.1.2 void WasmachineApp::onTextMessage (const string & *message*, WebSocket ∗ *ws*) [override], [virtual]

Will be called when a text message from the other side has been recieved

Parameters

message	the text message recieved
ws	the websocket that recieved the message

Implements WebSocketListener.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/WasmachineApp.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/WasmachineApp.cpp

10.42 Wasprogramma Class Reference

Public Member Functions

- Wasprogramma (int temp, int waterlevel, int time, int job)
- int **getLevel** ()
- int **getTemp** ()
- int **getTime** ()
- int **getJob** ()

10.42.1 Constructor & Destructor Documentation

10.42.1.1 Wasprogramma::Wasprogramma (int *temp*, int *waterlevel*, int *time*, int *job*)

file Heater.cpp version 0.1 author Remco Nijkamp / Jordan Ranirez / Kevin Damen / Jeroen Kok date 19-01-2016

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/WasProgramma.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/WasProgramma.cpp

10.43 WaterController Class Reference

Inheritance diagram for WaterController:

Public Member Functions

- **WaterController** (WashingMachineController ∗wascontroller)
- void setUartPointer (UART ∗u)

  Used to (re)set the pointer to the uart.
- void setWaterLevel (int level)

  sets the waterlevel
- void setResponseFlag ()

  sets the response flag
- void writeResponse (char ∗response)

  writes a response in the response pool
- void main ()

  task body, must be provided by a derived class

Additional Inherited Members

10.43.1 Member Function Documentation

10.43.1.1 void WaterController::main ( ) [virtual]

task body, must be provided by a derived class

A task is created by inheriting from task and providing a main() function. Initialisation of the task, including creating its waitables, should be done in the constructor. Don't forget to call the constructor of the task class!

The main() is the body of the task. It should never terminate.

Each task has a unique priority (an unsigned integer). A lower value indicates a higher priority. The RTOS scheduler will always run the task with the higest-priority runnable (neither blocked nor suspended) task. A task runs until it changes this 'situation' by using an RTOS call that changes its own state to not runnable, or the state of a higher priority task to runnable.

Timers are served only when the RTOS is activated by calling any of its state-changing interfaces. Hence the longest run time between such calls determines the granularity (time wise responsiveness) of the application. Within a time consuming computation a task can call release() to have the RTOS serve the timers.

Implements RTOS::task.

10.43.1.2 void WaterController::setResponseFlag ( )

sets the response flag

Returns

void

10.43.1.3 void WaterController::setUartPointer (UART * *u*)

Used to (re)set the pointer to the uart.

Parameters

*u**	pointer to the UART object this controller should use.

10.43.1.4 void WaterController::setWaterLevel (int *level*)

sets the waterlevel

Parameters

level	the waterlevel in precent

Returns

void

10.43.1.5 void WaterController::writeResponse (char * *response*)

writes a response in the response pool

Parameters

response	a char array with two positions

Returns

void

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/WaterController.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/WaterController.cpp

10.44 WaterSensor Class Reference

Public Member Functions

- WaterSensor ()
- char * getWaterLevelCommand ()

  Returns the bytes to get the waterlevel.

10.44.1 Detailed Description

Used as an interface to get the appropiate bytes for the uart

10.44.2 Constructor & Destructor Documentation

10.44.2.1 WaterSensor::WaterSensor ( )

file Heater.cpp version 0.1 author Remco Nijkamp / Jordan Ranirez / Kevin Damen / Jeroen Kok date 19-01-2016

10.44.3 Member Function Documentation

10.44.3.1 char * WaterSensor::getWaterLevelCommand ( )

Returns the bytes to get the waterlevel.

Returns

a char pointer to a 2 char array

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/WaterSensor.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/WaterSensor.cpp

10.45 WebSocket Class Reference

`#include <websocket.h>`

Public Member Functions

- WebSocket (TCPSocket *sock)
- void close ()
- void setListener (WebSocketListener *)
- void sendTextMessage (const string &message) throw (WebSocketException, SocketException)
- string getForeignAddress ()

10.45.1 Detailed Description

Websocket that is able to communicate with an another websocket over the underlying TCPSocket. It can send a message to the other side and listens to incomming messages.

See also

RFC6455

Author

jan.zuurbier@hu.nl

10.45.2 Constructor & Destructor Documentation

10.45.2.1 WebSocket::WebSocket (TCPSocket * *sock*)

Construct a web socket

Parameters

sock	underlying TCP socket

10.45.3 Member Function Documentation

10.45.3.1 void WebSocket::close (void)

Closes the websocket connection

10.45.3.2 string WebSocket::getForeignAddress ( ) [inline]

Get the IP address of the remote web socket

10.45.3.3 void WebSocket::sendTextMessage (const string & *message*) throw WebSocketException, SocketException)

Send a text message to the other side

Parameters

message	the message to be sent

Exceptions

WebSocketException	if the message could not be sent.
SocketException	if there is a problem with the underlying tcp socket

10.45.3.4 void WebSocket::setListener (WebSocketListener * /)

Sets the listener for this websocket that handles incomming messages.

The documentation for this class was generated from the following files:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/websocket.h
- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/websocket.cpp

10.46 WebSocketException Class Reference

Inheritance diagram for WebSocketException:

Public Member Functions

- **WebSocketException** (const std::string &message) throw ()
- **WebSocketException** (const std::string &message, const std::string &detail) throw ()

The documentation for this class was generated from the following file:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/websocket.h

10.47 WebSocketListener Class Reference

Inheritance diagram for WebSocketListener:

Public Member Functions

- virtual void onTextMessage (const string &message, WebSocket *ws)=0
- virtual void onClose (WebSocket *ws)=0

10.47.1 Member Function Documentation

10.47.1.1 virtual void WebSocketListener::onClose (WebSocket * *ws*) [pure virtual]

Will be called when the websocket has been close.

Parameters

ws	the websocket that has been closed

Implemented in WasmachineApp.

10.47.1.2 virtual void WebSocketListener::onTextMessage (const string & *message*, WebSocket * *ws*) [pure virtual]

Will be called when a text message from the other side has been recieved

Parameters

message	the text message recieved
ws	the websocket that recieved the message

Implemented in WasmachineApp.

The documentation for this class was generated from the following file:

- C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/websocket.h

Chapter 11

File Documentation

11.1 C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/↩ Heater.cpp File Reference

```
#include "Heater.h"
```

11.1.1 Detailed Description

Version

0.1

Author

Remco Nijkamp / Jordan Ranirez / Kevin Damen / Jeroen Kok

Date

19-01-2016

11.2 C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/libserial.cpp File Reference

Class to manage the serial port on Linux systems.

```
#include "libserial.h"
```

11.2.1 Detailed Description

Class to manage the serial port on Linux systems.

Version

1.0

Date

26 november 2014

11.3 C:/Users/jeroen/Documents/GitHub/Thema6-team10/visual studio/Project1/Project1/libserial.h File Reference

Serial library to communicate through a serial port on Linux systems.

```
#include <sys/types.h>
#include <termios.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
```

Classes

- class LibSerial

  This class can manage a serial port. The class allows basic operations (opening the connection, reading, writing data and closing the connection).

11.3.1 Detailed Description

Serial library to communicate through a serial port on Linux systems.

Author

Marten Wensink

Version

1.0

Date

26 november 2014

This class is based on a library implemented by Philippe Lucidarme (University of Angers) serialib@googlegroups.com It can be used to communicate through a serial port on Linux systems.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE X CONSORTIUM BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

This is a licence-free software, it can be used by anyone who try to build a better world.