ESR Consortium
SP-2.0

Shielded Plug
Profile Specification

ESR0014

Reference: ESR-SPE-0014-SP
Version: 2.0
Rev: A

DEFINITIONS

"ESR" means the Specification, including any modifications and upgrades, where these terms have been stated or referred to, and made available to You by ESR Consortium, including without limitation, texts, drawing, codes and examples.

"ESR Consortium" means the non-profit entity, registered in France in accordance with the French law of 1901.

"You" means the legal entity or entities represented by the individual executing this Agreement.

READ RIGHTS

Subject to the terms and conditions contained herein, ESR Consortium grants to You a non-exclusive, non-transferable, worldwide, and royalty-free license to view and read the ESR solely for purposes of Your internal evaluation.

GENERAL TERMS

THIS DOCUMENTATION IS PROVIDED "AS IS", WITHOUT WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED.

THE READING OF THE ESR AND ALL CONSEQUENCES ARISING THEREOF IS YOUR SOLE RESPONSIBILITY. ESR CONSORTIUM SHALL NOT BE LIABLE TO YOU FOR ANY LOSS OR DAMAGE CAUSED BY, ARISING FROM, DIRECTLY OR INDIRECTLY, OR IN CONNECTION WITH THE ESR.

COPYRIGHT

ESR Consortium does claim any right in this ESR. You are free to use this ESR to make any clean room implementations or derivative work as long as You don't claim that Your work is compliant with the ESR. Compliance tests are available from the ESR Consortium.

MISCELLANEOUS

This Agreement shall be governed by, and interpreted in accordance with French Law. In no event shall this Agreement be construed against the drafter.

This Agreement contains the entire understanding between the parties concerning its subject matter and supersedes any other agreement or understanding, whether written or oral, which may exist or have existed between the parties on the subject matter hereof.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION.

ESR CONSORTIUM MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN ANY ESR PUBLICATION AT ANY TIME.

Trademarks

Java™ is Sun Microsystems' trademark for a technology for developing application software and deploying it in cross-platform, networked environments. When it is used in this documentation without adding the ™ symbol, it includes implementations of the technology by companies other than Sun.

Java™,all Java-based marks and all related logos are trademarks or registered trademarks of Sun Microsystems Inc, in the United States and other Countries.

Contents

Tables

Illustrations

1 PREFACE TO SP 2.0 PROFILE, ESR0014

This document defines the, *SP 2.0* profile , targeting Embedded Platforms.

1.1 Who Should Use this Specification

This specification targets the following audiences:

- ESR Consortium Members who want to build implementation that complies to the SP profile specification.
- Application developers designing application using the SP.

1.2 How This Specification is Organized

This specification is organized as follow:

- **Introduction** is a short chapter explaining what is SP, why it has been designed and what are its main assets.
- **Specification** describes concepts and semantics.
- **SP API Documentation** lists the SP APIs in as javadoc.

1.3 Comments

Your comments about SP are welcome. Please send them by electronic mail to the following address: `comments @e-s-r.net` , with SP in your subject line.

1.4 Glossary

- *ESR*: Embedded Specification Request
- *baremetal*: a Java virtual machine is said to be *baremetal* when it does not require an OS/RTOS to run. A baremetal Java virtual machine is in fact an OS/RTOS that also embeds a Java engine. The device boots directly in Java.

1.5 Related Literature

1.6 Document Conventions

In this document, references to methods of a Java class are written as `ClassName.methodName(args)`. This applies to both static and instance methods. Where the method is static this will be made clear in the accompanying text.

1.7 Implementation Notes

The SP specification does not include any implementation details. SP implementors are free to use whatever techniques they deem appropriate to implement the specification, with (or without) collaboration of any Java virtual machine provider. SP experts have taken great care not to mention any special Java virtual machines, nor any of their special features, in order to encourage fair competing implementations.

2 INTRODUCTION

2.1 General description

Lots of highly secure applications have software architectures based on processes which run independently with no interactions except some data exchanges. Data are published in a shared space between producers who « *Publish* » and users who « *Subscribe* » to the data.

This kind of architecture is common in industrial control, automatic system supervision, telecoms, and all applications which need to propagate data asynchronously.

This specification SP *ShieldedPlug* offers a well-defined segregation between producers and consumers of data. Processes which publish data have a minimal semantic relation to data subscribers. Also thanks to the same mechanism the processes using the data don't need to be aware of the producers.

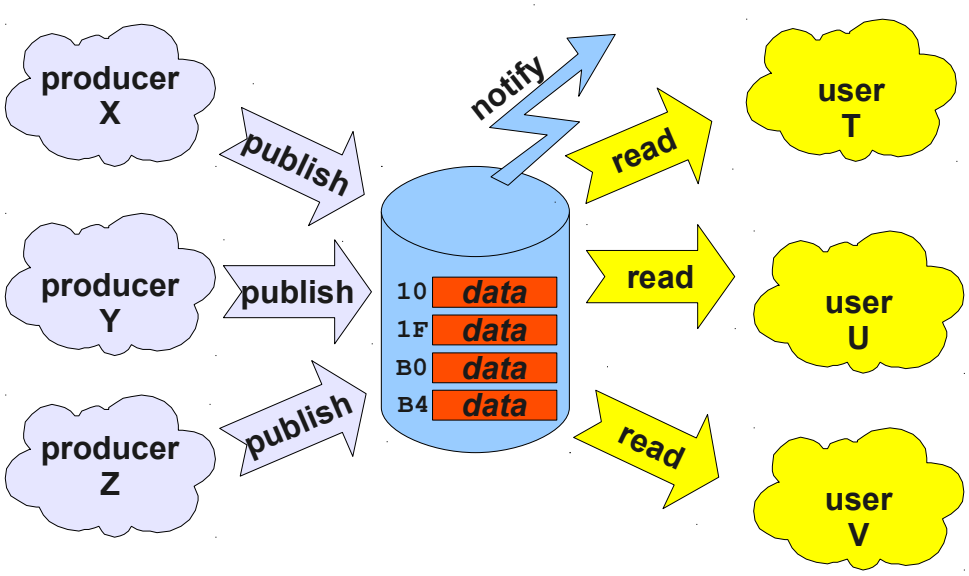


Figure 2-1: The Publish/Subscribe Concept

Data published are copies of the original data, providing complete isolation between producers and consumers.

2.2 Genesis

SP is driven by three factors: most software[1] is written in Java and in C; soon lots of software will have to be certified; and most micro-controllers are used for devices which have limited resources in terms of calculation capability and memory capacity.

Most critical software is certified by following a Security Insurance approach (by analogy to Quality Insurance). The level of trust needed by the software is obtained thanks to the strict application of engineering rules. Those rules have been established after years of experience.. Due

1 The word software refers to all programs executing on a program unit, which is likely to be a micro-controller.

to the complex nature of software programming – an intellectual activity – the main concepts of software certification is: "*software failures have only one origin: the software engineering*". Therefore, the concepts used to write programs are chosen to minimize the probability of introducing a software error and also to minimize the impact of potential errors by using isolation. However, typically those rules are not scientifically proven by any mathematical approach.

A software architecture that minimizes the effects of programming errors (defensive programming), associated with a suitable development process, allows segregation of the functional parts into different layers of trust. The safer parts are much more expensive to produce. The SP specification is born from the desire to provide a framework for safe sharing of data between different processes (either in C or Java) while keeping in mind that the software will be run on devices where costs matter a lot.

2.3 Main functionalities

SP provides segregation of the processes, which can be written either in C or in Java. It allows the certification of each individual part separately.

The data sharing between processes uses the concept of shared memory blocks, with introspection on those blocks. Facilities provided include: notification when the content changes, re-initialization of the block, testing the presence of data in the data block, and a mechanism for serialization and de-serialization.

SP allows the creation of several data stores. These can be defined entirely statically, or increase in number during the execution of a program.

Reading and writing in the shared memory are operations with predictable performance characteristics.

3 SPECIFICATION

The Java API chapter at the end of the document is part of the specification.

3.1 Databases

SP uses the notion of *databases*. Several databases can exist on the same system. In Java each database is an instance of the class `ShieldedPlug`. In C each database is an instance of the structure `ShieldedPlug`. A database is made up of *blocks* that cannot be divided. Each block is a memory space with contiguous addresses, and has a unique identifier (called an *index*) defined by an `int`. The size of a block is defined at construction time and cannot be modified.

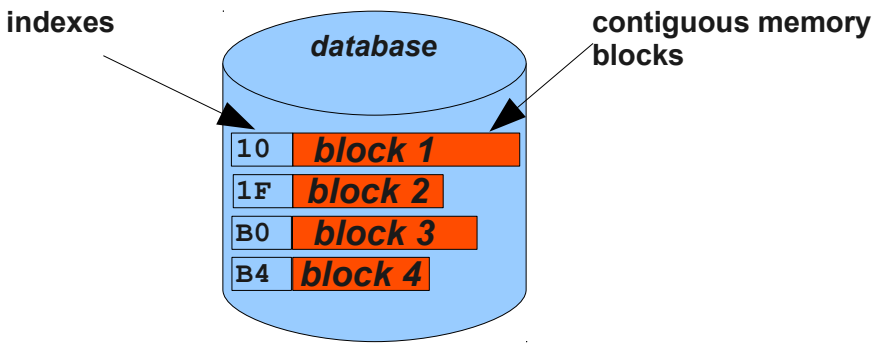


Figure 3-1: Example of a database having four blocks of different sizes

A database can optionally be defined with a fixed number of indexes and memory blocks. If that is the case it is defined as immutable, and `ShieldedPlug.isImmutable()` returns true. If not, a memory block can be destroyed by using `ShieldedPlug.delete(int)`, and created by specifying an ID, a size, and the number of tasks that can wait for this block, using `ShieldedPlug.create(int,int,int)`.

The number of memory blocks used by a database can be obtained using `ShieldedPlug.getSize()`. The list of the IDs of all memory blocks in the database can be obtained using `ShieldedPlug.getIDs()`. Finally, the length of a block with a particular ID is obtained using `ShieldedPlug.getLength(int)`.

A database has its own unique ID (using an `int` value), which identifies it. The static method `ShieldedPlug.getDatabase(int)` returns the database with the provided ID.

If the system allows the creation of new databases at runtime, the static method `ShieldedPlug.createDatabase(int)` returns a new database (or `null` if creations are forbidden).

```
//Main APIs in Java
int getSize();
int[] getIDs();
int getLength(int blockID);

static ShieldedPlug getDatabase(int ID);
```

```
//Main APIs in C
int32_t SP_getSize(ShieldedPlug sp);
int32_t SP_getIDs(ShieldedPlug sp, int32_t* IDs, int32_t* length);
int32_t SP_getLength(ShieldedPlug sp, int32_t blockID);

ShieldedPlug SP_getDatabase(int32_t ID);
```

The database display method `ShieldedPlug.toString()` produces an XML description of the database structure. This description can be used by third-party software as an input if the software uses the same specification.

Tag name	Description	Attribute(s)
`shieldedPlug`	Root element.	
`database`	Defines a database	`version` : string, context specific. `name` : string, name used to generate the database in the C header and in the Java interface. `id` : `int`, unique id for a database in the program. `immutable` : `true` or `false`. If not mentioned, defaults to `true`.
`block`	Defines each block	`id` : `int`, unique id for a block in a database. `name` : string, name used to generate constants in the C header and in the Java interface. `length` : `int`, the number of bytes in the block. `maxTasks` : `int`, indicates the maximum number of tasks that can wait for this block. If not mentioned, unlimited. This field may be mandatory on certain targets.

Table 3-1: XML description of databases

Figure 3-2 shows an example of a produced description file.

```
<shieldedPlug>
    <database name="MyBase" id="0" immutable="true" version="2.1.2">
        <block id="0x10" name="TEMP" length="100" maxTasks="1"/>
        <block id="0x1F" name="V0" length="50" maxTasks="2"/>
        <block id="0xB0" name="V1" length="75" maxTasks="4"/>
        <block id="0xB4" name="I0" length="25" maxTasks="1"/>
    </database>
</shieldedPlug>
```

Figure 3-2: Example of database description file

3.2 Correspondence between Java and C types

Depending on the language used to access to a database, types have different names.

Java	Specification	C
`void`	void	`void`
`boolean`	8 bits,only two values	`uint8_t`
`byte`	8 bits, signed	`int8_t`
`char`	16 bits, unsigned	`uint16_t`
`short`	16 bits,signed	`int16_t`
`int`	32 bits, signed	`int32_t`
`long`	64 bits,signed	`int64_t`
`float`	IEEE 754 on 32 bits	`float`
`double`	IEEE 754 on 64 bits	`double`

Table 3-2: Correspondence between Java and C types

3.3 Atomicity and execution time

All access to a database is serialized by the implementation: there will be only one access (either read or write) at a time. Each access is atomic whatever the number of bytes. All bytes of a block are processed as one operation, it also means the byte array size for a read or a write operation should exactly match the block size. This avoids inconsistency.

A database does not use a separate thread to execute requests; each request executes in the context of the calling thread.

Database access is forbidden in an interrupt context.

Read/Write access time of a block depends only on the size of the block, and is independent of the size and complexity of the database.

3.4 Reading data

A read is done on a specific memory block identified by its ID. The general reading method `ShieldedPlug.read(int, byte[])` fills the byte array with all the data held in the block (identified by the first parameter).

Additional methods are provided to read the base types directly, such as `readInt`, `readLong`, `readFloat`, `readDouble`. Repeated calls to these methods will return the same value, assuming there have been no interleaving writes to the block.

Errors use two different mechanisms: in C a negative return code is used, in Java the exception mechanism is used. The following errors can occur: invalid memory block ID, the block length is different from the size of the provided byte array, data is not available from this memory block.

It is possible to de-serialize a memory block to an object by associating a memory block with a reader that implements the `SPReader` interface. The method `ShieldedPlug.readObject(int)` returns an object by invoking the specified reader with the method `SPReader.readObject(ShieldedPlug sp, int id)`.

The association of a reader with a memory block is made with the method `ShieldedPlug.setReader(SPReader, int)`.

```
//Main Java APIs
void   read      (int blockID, byte[] data) throws EmptyBlockException;

int    readInt   (int blockID) throws EmptyBlockException;
float  readFloat (int blockID) throws EmptyBlockException;
long   readLong  (int blockID) throws EmptyBlockException;
double readDouble(int blockID) throws EmptyBlockException;

Object readObject(int blockID) throws EmptyBlockException;
void   setReader (int blockID, SPReader r);
```

In the C language, the first parameter of the call is what would in Java be the method receiver: a reference to the database which we are working on.

```
//Main C APIs
int32_t SP_read (ShieldedPlug sp, int32_t blockID, void* buff, int32_t
size);
```

3.5 Writing data

A write is done on a specific memory block identified by its ID. The general writing method `ShieldedPlug.write (int, byte[])` writes the provided byte array into the block (identified by the first parameter).

Additional methods are provided to write the base types directly, such as `writeInt`, `writeLong`, `writeFloat`, `writeDouble`. When using these methods a block is assumed to hold only a single value, which might not occupy the whole block. Repeated calls to these methods will overwrite the previous value.

Errors use two different mechanisms: in C a negative return code is used, in Java the exception mechanism is used. The following errors can occur: invalid memory block ID, the block length is different from the size of the provided byte array.

It is possible to serialize a memory block to an object by associating a memory block with a specific writer implementing the `SPWriter` interface. The method

`ShieldedPlug.writeObject(int,Object)` invokes the specified writer with a call to the method `SPWriter.writeObject(ShieldedPlug sp, int id, Object o)`.

The association of a writer with a memory block is made with the method `ShieldedPlug.setWriter(SPWriter, int)`.

```
//Main Java APIs
void write      (int blockID, byte[] data);

void writeInt   (int blockID, int data);
void writeFloat (int blockID, float data);
void writeLong  (int blockID, long data);
void writeDouble(int blockID, double data);

void writeObject(int blockID, Object o);
void setWriter  (int blockID, SPWriter w);
```

In the C language, the first parameter of the call is what would in Java be the method receiver: a reference to the database which we are working on.

```
//Main C APIs
int32_t SP_write(ShieldedPlug sp, int blockID, void* buff);
```

3.6 Notification of modification

Each memory block has a flag that indicates that an update has occurred since the last read. It is possible to test this state : `ShieldedPlug.isPending(int)`. This flag is set to `false` when reading, and to `true` when writing.

A task can wait for the modification of a memory block by using `ShieldedPlug.waitFor(int)`. This method suspends the current task if and only if the method `pending` returns `false` on the specified memory block. A task can also wait on several memory blocks, the task is released when one of the blocks is modified (`ShieldedPlug.waitFor(int[])`).

A memory block can have a limit to the number of tasks potentially waiting for it (cf 3.1). `ShieldedPlug.getMaxTasks(id)` returns the maximum number of tasks, or -1 if this number is infinite.

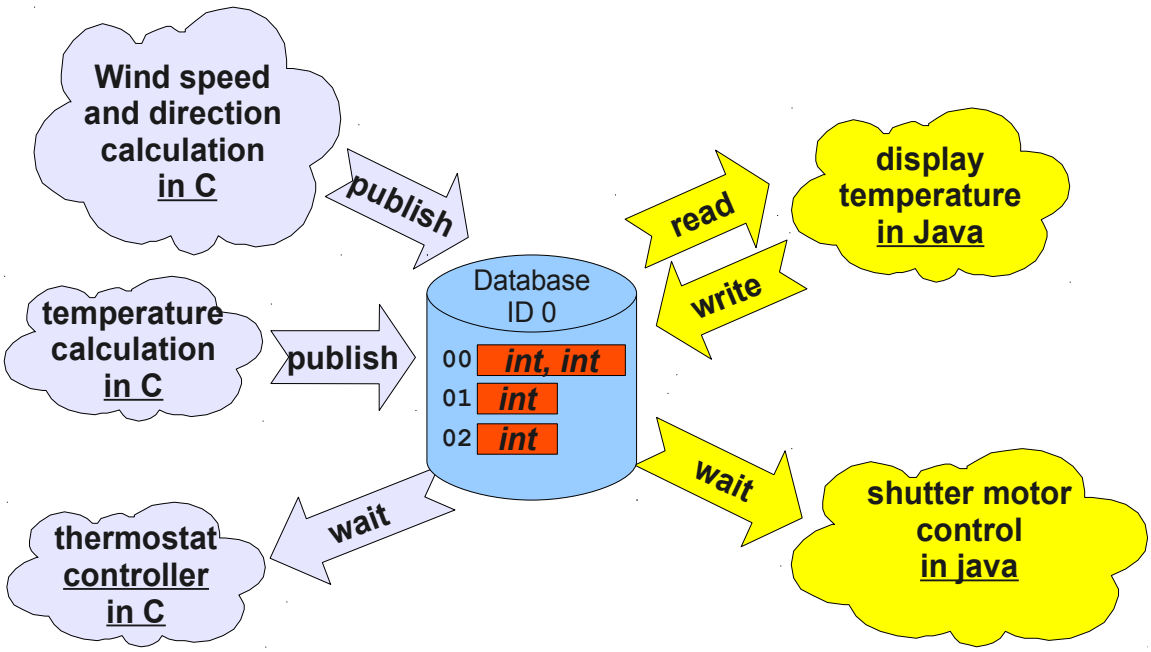
Also, a memory block has a flag indicating if its data are available or not. This flag is initially false and is set to `true` when writing data. It can be set to `false` using the method `ShieldedPlug.reset(int)`. It is possible to test this flag using `ShieldedPlug.isDataAvailable(int)`.

```
//Main Java APIs
boolean isPending(int blockID);
boolean isDataAvailable(int blockID);
boolean reset(int blockID);
void    waitFor(int blockID) throws InterruptedException;
int[]   waitFor(int[] blockIDs) throws InterruptedException;

//Main C APIs
int32_t SP_isPending(ShieldedPlug sp, int32_t blockID);
int32_t SP_isDataAvailable(ShieldedPlug sp, int32_t blockID);
int32_t SP_reset(ShieldedPlug sp, int32_t_t blockID);
int32_t SP_waitFor(ShieldedPlug sp, int32_t blockID);
int32_t SP_waitFor(ShieldedPlug sp, int32_t* blockIDs, int32_t*
modifiedIDs, int32_t* length);
```

4 USE CASE

Below is an example of using a database SP. The code that publishes the data is written in C, and the code that receives the data is written in Java. The data is transferred using two memory blocks. One is a scalar value, the other is a more complex object representing a two dimensional vector.



The database is described as follows:

```
<shieldedPlug>
    <database name="Forecast" id="0" immutable="true" version="1.0.0">
        <block id="0" name="WIND" length="8" maxTasks="1"/>
        <block id="1" name="TEMP" length="4" maxTasks="1"/>
        <block id="2" name="THERMOSTAT" length="4" maxTasks="1"/>
    </database>
</shieldedPlug>
```

4.1 Java Code

From the database description we can create a Java interface.

```
public interface Forecast {
      public static final int ID         = 0;
      public static final int WIND       = 0;
      public static final int TEMP       = 1;
      public static final int THERMOSTAT = 2;
}
```

Here are the implementations of the Wind class and its reader, which de-serializes it: first `int` is the speed and second is the direction.

```
public class Wind {
      public int speed;   //in ms [0..]
      public int direction; //in degree [0..360]
}
```

```
import ej.bon.ByteArray;

public class WindReader implements SPReader {
      private static final int SPEED = 0;
      private static final int DIRECTION = 4;
      public Object readObject(ShieldedPlug database, int blockID)
                                              throws EmptyBlockException {
            Wind w = new Wind();
            byte[] data = new byte[database.getLength(blockID)];
            database.read(blockID, data);
            w.speed   = ByteArray.readInt(data, SPEED);
            w.direction = ByteArray.readInt(data, DIRECTION);
            return w;
      }
}
```

Below is the task that reads the published wind data.

```
static {
      ShieldedPlug.getDatabase(Forecast.ID).setReader(Forecast.WIND,
                                                      new WindReader());
}

public void run(){
      ShieldedPlug database = ShieldedPlug.getDatabase(Forecast.ID);
      try{
            while (isRunning){
                  ///reading the wind when changing
                  database.waitFor(Forecast.WIND);
                  Wind w = (Wind) database.readObject(Forecast.WIND);
                  execute(calculation(w));
            }
      }
      catch( EmptyBlockException e){
            print("Error");
      }
      catch(InterruptedException e){
            //the current task has been interrupted
      }
}
```

Below is the task that reads the published temperature and controls the thermostat.

```
public void run(){
      ShieldedPlug database = ShieldedPlug.getDatabase(Forecast.ID);
      while (isRunning){
            //reading the temperature every 30 seconds
            //and update thermostat status
            try {
                  int temp = database.readInt(Forecast.TEMP);
                  print(temp);

                  //update the thermostat status
                  database.writeInt(Forecast.THERMOSTAT,
                                    temp>tempLimit ? 0 : 1);
            }
            catch(EmptyBlockException e){
                  print("Temperature not available");
            }
            sleep(30000);
      }
}
```

4.2 C Code

C header that declares the constants defined in the XML description of the database.

```
#define Forecast_ID 0
#define Forecast_WIND 0
#define Forecast_TEMP 1
#define Forecast_THERMOSTAT 2
```

Publication of wind and temperature is performed by two functions.

```
#include <sp.h>

struct Wind {
      int32_t speed;
      int32_t direction;
};

void windPublication(){
      struct Wind w;
      ShieldedPlug database = SP_getDatabase(Forecast_ID);
      w.speed = speed();
      w.direction = direction();
      SP_write(database, Forecast_WIND, &w);
}

void temperaturePublication(){
      ShieldedPlug database = SP_getDatabase(Forecast_ID);
      int32_t temp = temperature();
      SP_write(database, Forecast_TEMP, &temp);
}
```

Thermostat controller task waits for data from the ShieldedPlug.

```
#include <sp.h>

void thermostatTask(){
      int32_t thermostatOrder;
      ShieldedPlug database = SP_getDatabase(Forecast_ID);

      while(1){
            SP_waitFor(database, Forecast_THERMOSTAT);

            SP_read(database, Forecast_THERMOSTAT, &thermostatOrder);
            if(thermostatOrder == 0) {
                  thermostatOFF();
            }
            else {
                  thermostatON();
            }
      }
}
```

5 API

5.1 C Header File: sp.h

```
/*
 * Copyright ESR consortium. All rights reserved.
 * Modification and distribution is permitted under certain conditions.
 * PROPRIETARY : use is subject to license terms.
 */

/*
 * Header file for Shielded Plug (SP), version 1.1
 */
#ifndef SP_H
#define SP_H

#include <stdint.h>

#define SP_SUCCESS                              0 //function succeeded
#define SP_ERR_INVALID_BLOCK_ID                -1 //invalid block ID
#define SP_ERR_EMPTY_BLOCK                     -2 //no data available in the block
#define SP_ERR_INTERRUPTED                     -3 //current thread has been interrupted
#define SP_ERR_TOO_MANY_WAITING_THREADS        -4 //the limit on the number of threads waiting
                                                  //on a block has been reached

typedef void* ShieldedPlug;

/*
 * Returns the database identified by the given ID, or 0 if ID is undefined.
 */
ShieldedPlug SP_getDatabase(int32_t ID);

/*
 * Returns the number of blocks in the given database.
 */
int32_t SP_getSize(ShieldedPlug sp);

/*
 * Fills the given array with the IDs of the blocks available in this database.
 * If length is lower than the number of blocks in the database, only length IDs are
 * copied.
 * If length is greater than the number of blocks in the database, the array is only
 * partially filled.
 *
 * Returns the number of blocks in the given database.
 */
int32_t SP_getIDs(ShieldedPlug sp, int32_t* blocksIDs, int32_t length);

/*
 * Returns the length in bytes of the block with the given ID.
 * Returns <code>SP_ERR_INVALID_BLOCK_ID</code> if no block is defined with the given ID.
 */
int32_t SP_getLength(ShieldedPlug sp, int32_t blockID);

/*
 * Returns the maximum number of tasks that can wait at the same time on the block defined
 * with the given ID.
 *
 * Returns <code>SP_ERR_INVALID_BLOCK_ID</code> if no block is defined with the given ID.
 */
int32_t SP_getMaxTasks(ShieldedPlug sp, int32_t blockID);

/*
 * Fills the given buffer with data from the block with the given ID.
 * The number of bytes read is equal to the block size.
 *
 * Returns <code>SP_SUCCESS</code> on success, otherwise returns one of the following
 * errors:
 * - <code>SP_ERR_INVALID_BLOCK_ID</code>    if no block is defined with the given ID.
 * - <code>SP_ERR_EMPTY_BLOCK</code>         if no data available in the block.
 */
int32_t SP_read (ShieldedPlug sp, int32_t blockID, void* buff);
```

```
/*
 * Writes bytes from the given buffer to the block with the given ID.
 * The number of written bytes is equal to the block size. If any tasks are waiting for
 * data to be written to this block they are all unblocked.
 *
 * Returns <code>SP_SUCCESS</code> on success, otherwise returns one of the following
 * errors:
 * - <code>SP_ERR_INVALID_BLOCK_ID</code>   if no block is defined with the given ID.
 */
int32_t SP_write(ShieldedPlug sp, int32_t blockID, void* buff);

/*
 * Causes current thread to wait until another thread writes data into the block with the
 * given ID.
 * If data has been written in the block since the last read, this method returns
 * immediately.
 *
 * Returns <code>SP_SUCCESS</code> on success, otherwise returns one of the following
 * errors:
 * - <code>SP_ERR_INVALID_BLOCK_ID</code> if no block is defined with the given ID.
 * - <code>SP_ERR_TOO_MANY_WAITING_THREADS</code> if the limit on the number of threads
 * waiting on the block has been reached.
 * - <code>SP_ERR_INTERRUPTED</code> if another thread has interrupted the current thread.
 */
int32_t SP_waitFor(ShieldedPlug sp, int32_t blockID);

/*
 * Causes current thread to wait until another thread writes data into one of the specified
 * blocks.
 * If data has been written in one of the specified blocks since the last read from it,
 * this method returns immediately.
 *
 * Parameters:
 * - blocksIDs:              list of block IDs.
 * - modifiedIDs:     filled with the list of IDs of the blocks that have been written to.
 * - length:          before the call: the number of IDs in blocksIDs; after the call: the
 * number of IDs in modifiedsIDs.
 *
 * Returns <code>SP_SUCCESS</code> on success, otherwise returns one of the following
 * error:
 * - <code>SP_ERR_INVALID_BLOCK_ID</code>   if one of the ID does not correspond to an
 * existing block.
 * - <code>SP_ERR_TOO_MANY_WAITING_THREADS</code>   if the limit on the number of threads
 * waiting on a block has been reached.
 * - <code>SP_ERR_INTERRUPTED</code>        if another thread has interrupted the current
 * thread.
 */
int32_t SP_waitForSeveral(ShieldedPlug sp, int32_t* blockIDs, int32_t* modifiedIDs,
int32_t* length);

/*
 * Returns 1 if data has been written into the block since last read, 0 otherwise.
 * Returns <code>SP_ERR_INVALID_BLOCK_ID</code> if no block is defined with the given ID.
 */
int32_t SP_isPending(ShieldedPlug sp, int32_t blockID);

/*
 * Indicates whether or not data are available in the block with the given ID.
 * Initially no data are available in a block. When data are written in a block, they
 * remain available until method SP_reset(ShieldedPlug, int32_t) is called.
 *
 * Returns 1 if data are available in the block, 0 otherwise.
 * Returns <code>SP_ERR_INVALID_BLOCK_ID</code> if no block is defined with the given ID.
 */
int32_t SP_isDataAvailable(ShieldedPlug sp, int32_t blockID);

/*
 * Resets (clears) data of the block with the given ID.
 * After execution, SP_isDataAvailable method would return
 * 0 (unless data were written after calling SP_reset and before
 * calling SP_isDataAvailable).
 *
 * Returns <code>SP_SUCCESS</code> on success, otherwise returns
 * <code>SP_ERR_INVALID_BLOCK_ID</code> if no block is defined with the given ID.
 */
int32_t SP_reset(ShieldedPlug sp, int32_t blockID);

#endif /* SP_H */
```

Package Summary		Pa
ej.sp	Contains Shielded Plug classes (ESR014).	1

Package ej.sp

Contains Shielded Plug classes (`ESR014`).

See:

[Description]

Interface Summary		Page
SPReader	The `SPReader` interface provides a method for reconstructing objects from a block.	*31*
SPWriter	The `SPWriter` interface provides a method for serializing objects into a block.	*32*

Class Summary		Page
ShieldedPlug	A shielded plug is a database that contains several memory blocks.	*18*

Exception Summary		Page
EmptyBlockException	Thrown by methods in ShieldedPlug class to indicate that no data is available in a block.	*17*
TooManyWaitingThreadsException	Signals that too many threads are waiting for a block.	*33*

Package ej.sp Description

Contains Shielded Plug classes (`ESR014`).

Class EmptyBlockException

ej.sp

```
java.lang.Object
  └
    java.lang.Throwable
      └
        java.lang.Exception
          └
            ej.sp.EmptyBlockException
```

All Implemented Interfaces:

Serializable

```
public class EmptyBlockException
extends Exception
```

Thrown by methods in ShieldedPlug class to indicate that no data is available in a block.

Constructor Summary	*Page*
EmptyBlockException() Builds a new EmptyBlockException with null as its error message string.	*17*

Constructor Detail

EmptyBlockException

```
public EmptyBlockException()
```

Builds a new EmptyBlockException with null as its error message string.

Class ShieldedPlug

ej.sp

```
java.lang.Object
  └
    ej.sp.ShieldedPlug
```

```
public class ShieldedPlug
extends Object
```

A shielded plug is a database that contains several memory blocks.

A shielded plug can be created at runtime using createDatabase(int) or be created at startup and retrieved by getDatabase(int).

The list of memory blocks IDs can be retrieve using getIDs().
There are two sorts of shielded plugs:

1. The immutable ones (isImmutable()) that cannot be modified.
2. The mutable ones (!isImmutable()) can be modified by adding or removing blocks using create(int, int) or create(int, int, int) or delete(int).

Each block has fixed length (getLength(int) passing the block ID).
All access to a database is serialized by the implementation: there will be only one access (either read or write) at a time. Each access to a block is atomic, this avoids inconsistency:

- It can be read using one of the read methods that match its length.
- It can be written using one the write methods that match its length.

Each memory block has a flag that indicates that an update has occurred since the last read. It is possible to test this state: isPending(int). This flag is set to false when reading, and to true when writing.

A task can wait for the modification of a memory block by using waitFor(int). This method suspends the current task if and only if the method pending returns false on the specified memory block. A task can also wait on several memory blocks, the task is released when one of the blocks is modified waitFor(int[]).

Each memory block has a flag indicating if its data are available or not. It is possible to test this flag using isDataAvailable(int). This flag is initially `false` and is set to `true` when writing data. It can be set to `false` using the method reset(int).

Method Summary		Page
void	**create**(int blockID, int length) Creates a block with the given ID.	21
void	**create**(int blockID, int length, int maxTasks) Creates a block with the given ID.	21
static ShieldedPlug	**createDatabase**(int ID) Creates a new empty database with the given ID.	20
void	**delete**(int blockID) Deletes the block with the given ID.	21
static ShieldedPlug	**getDatabase**(int ID) Returns the database defined at the given ID.	20
int	**getID**() Gets the ID of this database.	22
int[]	**getIDs**() Gets the list of IDs of the blocks available in this database.	22

`int`	**getLength**`(int blockID)` Returns the length of the block with the given ID.	*22*
`int`	**getMaxTasks**`(int blockID)` Gets the maximum number of tasks that can wait at the same time on the block defined with the given ID.	*22*
SPReader	**getReader**`(int blockID)` Gets the SPReader used to de-serialize objects from the block with the given ID. If no SPReader is defined for the block, `null` is returned.	*26*
`int`	**getSize**`()` Gets the number of blocks of this database.	*22*
SPWriter	**getWriter**`(int blockID)` Gets the SPWriter used to serialize objects into the block with the given ID. If no SPWriter is defined for the block, `null` is returned.	*28*
`boolean`	**isDataAvailable**`(int blockID)` Determines whether data in the block with the given ID are available or not. By default no data is available in a block.	*30*
`boolean`	**isImmutable**`()` Gets if this database is immutable or not.	*21*
`boolean`	**isPending**`(int blockID)` Gets if data has been written into the block since last read.	*29*
`void`	**read**`(int blockID, byte[] data)` Fills the given array with data from the block with the given ID.	*23*
`void`	**read**`(int blockID, byte[] data, int destOffset)` Fills the given array with `block.length` bytes from the block with the given ID.	*23*
`double`	**readDouble**`(int blockID)` Reads eight input bytes from the block with the given ID and returns a `double` value. The way the `double` is built from the eight bytes is platform dependent.	*25*
`float`	**readFloat**`(int blockID)` Reads four input bytes from the block with the given ID and returns a `float` value. The way the `float` is built from the four bytes is platform dependent.	*24*
`int`	**readInt**`(int blockID)` Reads four input bytes from the block with the given ID and returns an `int` value. The way the `int` is built from the four bytes is platform dependent.	*24*
`long`	**readLong**`(int blockID)` Reads eight input bytes from the block with the given ID and returns a `long` value. The way the `long` is built from the eight bytes is platform dependent.	*24*
`Object`	**readObject**`(int blockID)` Invokes the `readObject` method of the SPReader registered for the block with the given ID.	*25*
`void`	**reset**`(int blockID)` Resets data of the block with the given ID. After execution of this method, isDataAvailable(int) method would return `false` (unless data were written after calling `reset(int)` and before calling isDataAvailable(int)).	*30*
`void`	**setReader**`(int blockID, SPReader reader)` Registers the given SPReader to de-serialize objects from the block with the given ID. If an SPReader is already defined for the block, it is replaced by the given SPReader.	*25*
`void`	**setWriter**`(int blockID, SPWriter writer)` Registers the given SPWriter to serialize objects into the block with the given ID. If an SPWriter is already defined for the block, it is replaced by the given SPWriter.	*28*

void	**waitFor**(int blockID) Causes current thread to wait until another thread write data into the block with the given ID. If data has been written in the block since last read, this method returns immediately.	*29*
int[]	**waitFor**(int[] blockIDs) Causes current thread to wait until another thread write data into at least one block from the blocks with the given IDs. If data has been written in one block since last read from it, this method returns immediately.	*29*
void	**write**(int blockID, byte[] data) Writes block length bytes from the specified byte array to the block with the given ID. The `write(blockID, data)` method has the same effect as:	*26*
void	**write**(int blockID, byte[] data, int srcOffset) Writes block length bytes from the specified byte array to the block with the given ID. Element `data[destOffset]` is the first byte written to the block.	*26*
void	**writeDouble**(int blockID, double value) Writes a `double` value, which is comprised of eight bytes, to the block with the given ID. The way the `double` is written from the eight bytes is platform dependent.	*28*
void	**writeFloat**(int blockID, float value) Writes a `float` value, which is comprised of four bytes, to the block with the given ID. The way the `float` is written from the four bytes is platform dependent.	*27*
void	**writeInt**(int blockID, int value) Writes an `int` value, which is comprised of four bytes, to the block with the given ID. The way the `int` is written from the four bytes is platform dependent.	*27*
void	**writeLong**(int blockID, long value) Writes a `long` value, which is comprised of eight bytes, to the block with the given ID. The way the `long` is written from the eight bytes is platform dependent.	*27*
void	**writeObject**(int blockID, Object o) Invokes the `writeObject` method of the SPWriter registered for the block with the given ID.	*28*

Method Detail

getDatabase

```
public static ShieldedPlug getDatabase(int ID)
```

Returns the database defined at the given ID.

Parameters:
`ID` - the identification number of the requested database
Returns:
the database with the given ID
Throws:
`IllegalArgumentException` - if no database is defined with the given ID

---

createDatabase

```
public static ShieldedPlug createDatabase(int ID)
```

Creates a new empty database with the given ID.

Parameters:
`ID` - the identification number of the created database
Returns:
the created database
Throws:
`IllegalArgumentException` - if a database with the given ID already exists
`SecurityException` - if the platform cannot create dynamically databases

---

isImmutable

```
public boolean isImmutable()
```

Gets if this database is immutable or not.

Returns:
`true` if no block can be added or remove to this database, `false` otherwise

---

delete

```
public void delete(int blockID)
```

Deletes the block with the given ID.

Parameters:
`blockID` - the ID of the block to delete
Throws:
`IllegalArgumentException` - if no block is defined with the given ID
`SecurityException` - if this database is immutable

---

create

```
public void create(int blockID,
                   int length,
                   int maxTasks)
```

Creates a block with the given ID.

Parameters:
`blockID` - the ID of the block to create
`length` - the length in bytes of the block to create
`maxTasks` - maximum number of tasks that can wait at the same time for a modification of the block
Throws:
`IllegalArgumentException` - if a block is already defined with the given ID
`SecurityException` - if this database is immutable

---

create

```
public void create(int blockID,
                   int length)
```

Creates a block with the given ID. An unlimited number of tasks will be able to wait at the same time for a modification of the block.

Parameters:
`blockID` - the ID of the block to create
`length` - the length in bytes of the block to create
Throws:
`IllegalArgumentException` - if a block is already defined with the given ID
`SecurityException` - if this database is immutable

---

getID

```
public int getID()
```

Gets the ID of this database.

Returns:
the ID of this database

---

getSize

```
public int getSize()
```

Gets the number of blocks of this database.

Returns:
the number of blocks in this database

---

getIDs

```
public int[] getIDs()
```

Gets the list of IDs of the blocks available in this database.

Returns:
the list of the IDs of the blocks available in this database

---

getLength

```
public int getLength(int blockID)
```

Returns the length of the block with the given ID.

Parameters:
`blockID` - the ID of the block
Returns:
the length in bytes
Throws:
`IllegalArgumentException` - if no block is defined with the given ID

---

getMaxTasks

```
public int getMaxTasks(int blockID)
```

Gets the maximum number of tasks that can wait at the same time on the block defined with the given ID.

Parameters:
`blockID` - the ID of the block
Returns:
the maximum number of tasks that can wait at the same time on the block defined with the given ID, or -1 if infinite
Throws:
`IllegalArgumentException` - if no block is defined with the given ID

---

read

```
public void read(int blockID,
                 byte[] data)
          throws EmptyBlockException
```

Fills the given array with data from the block with the given ID. The number of bytes read is equal to the length of the block.
The `read(blockID, data)` method has the same effect as:

```
read(blockID, data, 0)
```

Parameters:
`blockID` - the ID of the block
`data` - the buffer into which the data is read
Throws:
EmptyBlockException - if no data is available in the block
`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if `data.length` is lower than block length

---

read

```
public void read(int blockID,
                 byte[] data,
                 int destOffset)
          throws EmptyBlockException
```

Fills the given array with `block.length` bytes from the block with the given ID. The first byte read is stored into element `data[destOffset]`.

If `destOffset` is negative or `destOffset` + block length is greater than the length of the array `data`, then an `IndexOutOfBoundsException` is thrown.

Parameters:
`blockID` - the ID of the block
`data` - the buffer into which the data is read
`destOffset` - the start offset in array `data` at which the data is written
Throws:
EmptyBlockException - if no data is available in the block
`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if `destOffset` is negative or if `data.length` is lower than `destOffset` + block length

---

readInt

```
public int readInt(int blockID)
            throws EmptyBlockException
```

Reads four input bytes from the block with the given ID and returns an `int` value.
The way the `int` is built from the four bytes is platform dependent.

This method is suitable for reading bytes written by the `writeInt` method.

Parameters:
`blockID` - the ID of the block
Returns:
the `int` value read
Throws:
EmptyBlockException - if no data is available in the block
`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if block length is not four bytes

---

readLong

```
public long readLong(int blockID)
             throws EmptyBlockException
```

Reads eight input bytes from the block with the given ID and returns a `long` value.
The way the `long` is built from the eight bytes is platform dependent.

This method is suitable for reading bytes written by the `writeLong` method.

Parameters:
`blockID` - the ID of the block
Returns:
the `long` value read
Throws:
EmptyBlockException - if no data is available in the block
`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if block length is not height bytes

---

readFloat

```
public float readFloat(int blockID)
               throws EmptyBlockException
```

Reads four input bytes from the block with the given ID and returns a `float` value.
The way the `float` is built from the four bytes is platform dependent.

This method is suitable for reading bytes written by the `writeFloat` method.

Parameters:
`blockID` - the ID of the block
Returns:
the `float` value read

Throws:
EmptyBlockException - if no data is available in the block
`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if block length is not four bytes

---

readDouble

```
public double readDouble(int blockID)
                  throws EmptyBlockException
```

Reads eight input bytes from the block with the given ID and returns a `double` value.
The way the `double` is built from the eight bytes is platform dependent.

This method is suitable for reading bytes written by the `writeDouble` method.

Parameters:
`blockID` - the ID of the block
Returns:
the `double` value read
Throws:
EmptyBlockException - if no data is available in the block
`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if block length is not height bytes

---

readObject

```
public Object readObject(int blockID)
                  throws EmptyBlockException
```

Invokes the `readObject` method of the SPReader registered for the block with the given ID.
The SPReader is responsible for the de-serialization of the object from the block.

Parameters:
`blockID` - the ID of the block
Returns:
the object read from the block
Throws:
EmptyBlockException - if no data is available in the block
`IllegalArgumentException` - if no block is defined with the given ID
`NullPointerException` - if no SPReader has been registered for the block
`IndexOutOfBoundsException` - if block length is lower than the size needed for object de-serialization

---

setReader

```
public void setReader(int blockID,
                      SPReader reader)
```

Registers the given SPReader to de-serialize objects from the block with the given ID.
If an SPReader is already defined for the block, it is replaced by the given SPReader.

Parameters:
`blockID` - the ID of the block
`reader` - the SPReader

Throws:
`IllegalArgumentException` - if no block is defined with the given ID

---

getReader

```
public SPReader getReader(int blockID)
```

Gets the SPReader used to de-serialize objects from the block with the given ID.
If no SPReader is defined for the block, `null` is returned.

Parameters:
`blockID` - the ID of the block
Returns:
the SPReader set or null if none
Throws:
`IllegalArgumentException` - if no block is defined with the given ID

---

write

```
public void write(int blockID,
                  byte[] data)
```

Writes block length bytes from the specified byte array to the block with the given ID.
The `write(blockID, data)` method has the same effect as:

```
write(blockID, data, 0)
```

Parameters:
`blockID` - the ID of the block
`data` - the data to write
Throws:
`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if `data.length` value is lower than block length

---

write

```
public void write(int blockID,
                  byte[] data,
                  int srcOffset)
```

Writes block length bytes from the specified byte array to the block with the given ID.
Element `data[destOffset]` is the first byte written to the block.

If `destOffset` is negative, or `destOffset` + block length is greater than the length of the array `data`, then an `IndexOutOfBoundsException` is thrown.

Parameters:
`blockID` - the ID of the block
`data` - the data to write
`srcOffset` - the start offset in the data
Throws:
`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if `destOffset` is negative or if `data.length` is lower than `offset` + block length

---

writeInt

```
public void writeInt(int blockID,
                     int value)
```

Writes an `int` value, which is comprised of four bytes, to the block with the given ID.
The way the `int` is written from the four bytes is platform dependent.

The bytes written by this method may be read by the `readInt` method, which will then return an `int` equal to `value`.

Parameters:

`blockID` - the ID of the block
`value` - the `int` value to be written

Throws:

`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if block length is not four bytes

---

writeLong

```
public void writeLong(int blockID,
                      long value)
```

Writes a `long` value, which is comprised of eight bytes, to the block with the given ID.
The way the `long` is written from the eight bytes is platform dependent.

The bytes written by this method may be read by the `readLong` method, which will then return a `long` equal to `value`.

Parameters:

`blockID` - the ID of the block
`value` - the `long` value to be written

Throws:

`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if block length is not eight bytes

---

writeFloat

```
public void writeFloat(int blockID,
                       float value)
```

Writes a `float` value, which is comprised of four bytes, to the block with the given ID.
The way the `float` is written from the four bytes is platform dependent.

The bytes written by this method may be read by the `readFloat` method, which will then return a `float` equal to `value`.

Parameters:

`blockID` - the ID of the block
`value` - the `float` value to be written

Throws:

`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if block length is not four bytes

---

writeDouble

```
public void writeDouble(int blockID,
                        double value)
```

Writes a `double` value, which is comprised of eight bytes, to the block with the given ID.
The way the `double` is written from the eight bytes is platform dependent.

The bytes written by this method may be read by the `readDouble` method, which will then return a `double` equal to `value`.

Parameters:

`blockID` - ID of the block
`value` - the `double` value to be written

Throws:

`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if block length is not eight bytes

---

writeObject

```
public void writeObject(int blockID,
                        Object o)
```

Invokes the `writeObject` method of the SPWriter registered for the block with the given ID.
The SPWriter is responsible for the serialization of the object into the block.

Parameters:

`blockID` - the ID of the block
`o` - the object to be written

Throws:

`IllegalArgumentException` - if no block is defined with the given ID
`NullPointerException` - if no SPWriter has been registered for the block
`IndexOutOfBoundsException` - if block length is lower than the size needed for object serialization

---

setWriter

```
public void setWriter(int blockID,
                      SPWriter writer)
```

Registers the given SPWriter to serialize objects into the block with the given ID.
If an SPWriter is already defined for the block, it is replaced by the given SPWriter.

Parameters:

`blockID` - the ID of the block
`writer` - the SPWriter

Throws:

`IllegalArgumentException` - if no block is defined with the given ID

---

getWriter

```
public SPWriter getWriter(int blockID)
```

Gets the SPWriter used to serialize objects into the block with the given ID.
If no SPWriter is defined for the block, `null` is returned.

Parameters:
`blockID` - the ID of the block
Returns:
the SPWriter set or null if none
Throws:
`IllegalArgumentException` - if no block is defined with the given ID

---

waitFor

```
public void waitFor(int blockID)
            throws InterruptedException
```

Causes current thread to wait until another thread write data into the block with the given ID.
If data has been written in the block since last read, this method returns immediately.

Parameters:
`blockID` - the ID of the block
Throws:
`InterruptedException` - if another thread has interrupted the current thread The interrupted status of the current thread is cleared when this exception is thrown
`IllegalArgumentException` - if no block is defined with the given ID
TooManyWaitingThreadsException - if too many threads are waiting for new data

---

waitFor

```
public int[] waitFor(int[] blockIDs)
             throws InterruptedException
```

Causes current thread to wait until another thread write data into at least one block from the blocks with the given IDs.
If data has been written in one block since last read from it, this method returns immediately.

Parameters:
`blockIDs` - the list of block IDs
Returns:
the list of IDs of the blocks that has been written
Throws:
`InterruptedException` - if another thread has interrupted the current thread The interrupted status of the current thread is cleared when this exception is thrown
`IllegalArgumentException` - if one of the ID does not correspond to an existing block
TooManyWaitingThreadsException - if too many threads are waiting for new data

---

isPending

```
public boolean isPending(int blockID)
```

Gets if data has been written into the block since last read.

Parameters:
`blockID` - the ID of the block
Returns:
`true` if data has been written into the block since last read, `false` otherwise

Throws:
`IllegalArgumentException` - if no block is defined with the given ID

---

isDataAvailable

```
public boolean isDataAvailable(int blockID)
```

Determines whether data in the block with the given ID are available or not.
By default no data is available in a block. When data are written in a block, they remain available until method `reset(int)` is called.

Parameters:
`blockID` - the ID of the block
Returns:
`true` if data is available in the block `false` otherwise
Throws:
`IllegalArgumentException` - if no block is defined with the given ID

---

reset

```
public void reset(int blockID)
```

Resets data of the block with the given ID.
After execution of this method, isDataAvailable(int) method would return `false` (unless data were written after calling `reset(int)` and before calling isDataAvailable(int)).

Parameters:
`blockID` - the ID of the block
Throws:
`IllegalArgumentException` - if no block is defined with the given ID

Interface SPReader

ej.sp

`public interface` **`SPReader`**

The `SPReader` interface provides a method for reconstructing objects from a block.

Method Summary		Page
`Object`	**readObject**(ShieldedPlug `sp, int blockID`) Reads and returns an object from a block of the given ShieldedPlug.	*31*

Method Detail

readObject

```
Object readObject(ShieldedPlug sp,
                  int blockID)
           throws EmptyBlockException
```

Reads and returns an object from a block of the given ShieldedPlug. The class implementing this interface defines how the object is "read".

Parameters:

`sp` - the ShieldedPlug from which data is read
`blockID` - the ID of the block

Returns:

the object read from the ShieldedPlug

Throws:

EmptyBlockException - if no data is available in the block
`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if the block length is lower than the size needed for object de-serialization

Interface SPWriter

ej.sp

`public interface` **`SPWriter`**

The `SPWriter` interface provides a method for serializing objects into a block.

Method Summary		Page
`void`	**writeObject** (ShieldedPlug `sp, int blockID, Object o)` Writes an object into a block of the given ShieldedPlug.	*32*

Method Detail

writeObject

```
void writeObject(ShieldedPlug sp,
                 int blockID,
                 Object o)
```

Writes an object into a block of the given ShieldedPlug. The class implementing this interface defines how the object is written.

Parameters:

`sp` - the ShieldedPlug into which data is written
`blockID` - ID of the block
`o` - the object to be written

Throws:

`IllegalArgumentException` - if no block is defined with the given ID
`IndexOutOfBoundsException` - if the block length is lower than the size needed for object serialization

Class TooManyWaitingThreadsException

ej.sp

```
java.lang.Object
  └
    java.lang.Throwable
      └
        java.lang.Exception
          └
            java.lang.RuntimeException
              └
                ej.sp.TooManyWaitingThreadsException
```

All Implemented Interfaces:

Serializable

```
public class TooManyWaitingThreadsException
extends RuntimeException
```

Signals that too many threads are waiting for a block.

Constructor Summary	*Page*
TooManyWaitingThreadsException() Builds a TooManyWaitingThreadsException with no detail message.	*33*

Constructor Detail

TooManyWaitingThreadsException

```
public TooManyWaitingThreadsException()
```

Builds a TooManyWaitingThreadsException with no detail message.