



SciEngines
massively parallel computing

RIVYERA API

Host-API (Java)

Version 1.91.01 B2

SciEngines GmbH
Fraunhoferstrasse 13
24118 Kiel
Germany

www.sciengines.com

Revision: 921 1.91.01 B2 February 22, 2013

The Information in this document is provided for use with SciEngines GmbH ('SciEngines') products. No license, express or implied, to any intellectual property associated with this document or such products is granted by this document.

This document and other materials distributed with SciEngines products and marked as confidential ("Confidential Information") shall be treated with care to prevent unauthorized disclosure, but in no event less than reasonable care.

All products described in this document whose name is prefaced by 'COPACOBANA', 'RIVYERA', 'SciEngines' or 'SciEngines enhanced' ('SciEngines products') are owned by SciEngines GmbH (or those companies that have licensed technology to SciEngines) and are protected by patents, trade secrets, copyrights or other industrial property rights. The SciEngines products described in this document may still be in development. The final form of each product and release date thereof is at the sole and absolute discretion of SciEngines. Your purchase, license and/or use of SciEngines products shall be subject to SciEngines's then current sales terms and conditions.

Trademarks

The following are trademarks of SciEngines GmbH in the United States and other countries:

- SciEngines GmbH,
- SciEngines Massively Parallel Computing,
- SciEngines Logo,
- COPACOBANA, COPACOBANA RIVYERA, RIVYERA, IPANEMA

Trademarks of other companies

- Intel is a registered trademark of Intel Corporation.
- Linux is a registered trademark of Linus Torvalds.
- Windows is a registered trademark of Microsoft Corporation.
- Oracle, Oracle Enterprise Linux are a registered trademark of the Oracle Corporation.
- RedHat, RedHat Enterprise Linux are a registered trademark of the RedHat Corporation.
- Xilinx, Virtex and ISE are registered trademarks of Xilinx in the United States and other countries.
- ChipScope, CORE Generator and PlanAhead are trademarks of Xilinx, Inc.

Thank you for choosing an original SciEngines product.

Imprint

Responsible for content:

Firm SciEngines GmbH

Street Fraunhoferstr. 13

ZIP D-24118

City Kiel

Country Germany

Phone +49 431 5302 482

Email info@sciengines.com

WWW <http://www.sciengines.com>

CEO Gerd Pfeiffer

Commercial Register Amtsgericht Kiel

Commercial Register No. HR B 9565 KI

VAT- Identification Number DE 814955925

Disclaimer: Any information contained in this document is confidential, and only intended for reception and use by the specified person who bought the SciEngines product. Drawings, pictures, illustration and estimations are non binding and for illustration purposes only. If you are not the intended recipient, please return the document to the sender and delete any copies afterwards. In this case any copying, forwarding, printing, disclosure and use is strictly prohibited.

Table of Contents

1 Basic Information	11
1.1 General ideas of parallel programming	11
1.2 Concept of using SciEngines RIVYERA	12
1.3 API version information	14
1.4 RIVYERA API Addressing Scheme	16
1.4.1 Physical Address Components	16
1.4.2 Address Wildcards	16
1.4.3 Virtual Address Components	16
1.4.4 Target Addresses	17
1.4.5 Source Addresses	17
2 RIVYERA API Structure	18
2.1 RIVYERA API Register Paradigm	18
2.2 RIVYERA API Routing Strategies	19
2.2.1 Smart Routing	19
3 Java API Introduction	20
3.1 Machine addressing	20
3.2 Autonomous FPGA writes	21
3.3 Cross-language example code	21
4 SciEngines_API Class Reference	24
4.1 Detailed Description	24
4.2 Member Function Documentation	27
4.2.1 se_getMachineCount	27
4.2.2 se_allocMachine	27
4.2.3 se_freeMachine	27
4.2.4 se_read	27
4.2.5 se_write	27
4.2.6 se_program	27
4.2.7 se_deprogram	27
4.2.8 se_waitForData	27
4.2.9 se_getSlotCount	27
4.2.10 se_getSlotInfo	27
4.2.11 se_getFPGACount	27

4.2.12	se_getFPGAInfo	27
4.2.13	se_getControllerCount	27
4.2.14	se_getControllerInfo	27
4.2.15	se_getTemperature	27
4.2.16	se_getMaxTemperature	27
4.2.17	se_flush	27
4.2.18	se_program_slave	27
4.2.19	se_deprogram_slave	27
4.2.20	se_comment	27
5	SciEngines_API_Const Class Reference	28
5.1	Member Data Documentation	28
5.1.1	SE_API_VERSION_MAJOR	28
5.1.2	SE_API_VERSION_MINOR	29
5.1.3	SE_API_VERSION_SP	29
5.1.4	SE_API_VERSION_REVISION	29
5.1.5	SE_TIMEOUT_INFINITE	29
5.1.6	SE_ADDR_FPGA_ALL	29
5.1.7	SE_ADDR_SLOT_ALL	29
5.1.8	SE_ADDR_CONTR_ALL	29
5.1.9	SE_ADDR_FPGA_HOST	30
5.1.10	SE_ADDR_REG_EOT	30
5.1.11	SE_LENGTH_ADDR_SLOT	30
5.1.12	SE_LENGTH_ADDR_FPGA	30
5.1.13	SE_LENGTH_ADDR_REG	30
5.1.14	SE_LENGTH_CMD	30
5.1.15	SE_READ_ACTIVE	30
5.1.16	SE_READ_PASSIVE	30
6	SeAddress Class Reference	31
6.1	Detailed Description	31
6.2	Constructor & Destructor Documentation	31
6.2.1	SeAddress	31
6.3	Member Function Documentation	31
6.3.1	toString	31
6.4	Member Data Documentation	31

6.4.1	fpga	31
6.4.2	reg	32
6.4.3	slot	32
6.4.4	contr	32
7	SeApiException Class Reference	33
7.1	Detailed Description	33
7.2	Constructor & Destructor Documentation	33
7.2.1	SeApiException	33
7.3	Member Function Documentation	34
7.3.1	getErrorCode	34
8	SeApiFailedException Class Reference	35
8.1	Constructor & Destructor Documentation	35
8.1.1	SeApiFailedException	35
8.1.2	SeApiFailedException	35
8.2	Member Function Documentation	35
8.2.1	getErrorCode	35
9	SeApiFileErrorException Class Reference	36
9.1	Constructor & Destructor Documentation	36
9.1.1	SeApiFileErrorException	36
9.1.2	SeApiFileErrorException	36
9.2	Member Function Documentation	36
9.2.1	getErrorCode	36
10	SeApiInvalidAddressException Class Reference	37
10.1	Constructor & Destructor Documentation	37
10.1.1	SeApiInvalidAddressException	37
10.1.2	SeApiInvalidAddressException	37
10.2	Member Function Documentation	37
10.2.1	getErrorCode	37
11	SeApiInvalidMachineException Class Reference	38
11.1	Constructor & Destructor Documentation	38
11.1.1	SeApiInvalidMachineException	38
11.1.2	SeApiInvalidMachineException	38
11.2	Member Function Documentation	38

11.2.1	getErrorCode	38
12	SeApiMachineInUseException Class Reference	39
12.1	Constructor & Destructor Documentation	39
12.1.1	SeApiMachineInUseException	39
12.1.2	SeApiMachineInUseException	39
12.2	Member Function Documentation	39
12.2.1	getErrorCode	39
13	SeApiMachineNotAvailableException Class Reference	40
13.1	Constructor & Destructor Documentation	40
13.1.1	SeApiMachineNotAvailableException	40
13.1.2	SeApiMachineNotAvailableException	40
13.2	Member Function Documentation	40
13.2.1	getErrorCode	40
14	SeApiReadTimeoutException Class Reference	41
14.1	Constructor & Destructor Documentation	41
14.1.1	SeApiReadTimeoutException	41
14.1.2	SeApiReadTimeoutException	41
14.2	Member Function Documentation	41
14.2.1	getErrorCode	41
15	SeApiWriteTimeoutException Class Reference	42
15.1	Constructor & Destructor Documentation	42
15.1.1	SeApiWriteTimeoutException	42
15.1.2	SeApiWriteTimeoutException	42
15.2	Member Function Documentation	42
15.2.1	getErrorCode	42
16	SeControllerInfo Class Reference	43
16.1	Detailed Description	43
16.2	Member Function Documentation	43
16.2.1	getDriverName	43
16.2.2	getMachineSlot	43
16.2.3	getSerial	43
16.2.4	toString	43

17 SeFPGAInfo Class Reference	44
17.1 Detailed Description	44
17.2 Member Function Documentation	44
17.2.1 getType	44
17.2.2 isProgrammed	44
17.2.3 getFirmwareVersion	44
17.2.4 getFirmwareBuild	44
17.2.5 toString	44
18 SeOptions Class Reference	45
18.1 Member Enumeration Documentation	45
18.1.1 SeWriteBehavior	45
18.1.2 SeRoutingMethod	45
18.2 Constructor & Destructor Documentation	45
18.2.1 SeOptions	45
18.3 Member Function Documentation	45
18.3.1 getWriteBehavior	45
18.3.2 setWriteBehavior	45
18.3.3 getRoutingMethod	45
18.3.4 setRoutingMethod	45
19 SeSlotInfo Class Reference	46
19.1 Detailed Description	46
19.2 Member Function Documentation	46
19.2.1 isController	46
19.2.2 getControllerIndex	46
19.2.3 getFpgaCount	46
19.2.4 getSerial	46
19.2.5 getPrevContr	47
19.2.6 getNextContr	47
19.2.7 getFirmwareVersion	47
19.2.8 getFirmwareBuild	47
19.2.9 toString	47

1 Basic Information

The main purpose of the RIVYERA API is to interface single and multiple FPGAs in a massively parallel architecture as simple and easy as possible. We intended to provide an infrastructure for your FPGA designs which is powerful enough to transport the benefits of a massively parallel architecture without raising the complexity of your design.

Therefore, we provide a simple interface which makes the idiosyncratic implementation of the physical layers disappear and provide a high-level view into our machines in which details do not get in the way of your work.

This introduction offers a brief overview of the SciEngines RIVYERA machine. It describes the physical and structural machine features from the programmers' point of view.

1.1 General ideas of parallel programming

Traditionally, software has been written for serial computation. There are two naive reasons for serial computation concepts: one is that thinking in a **serial**, causal way is easy for most humans, the other is that computers started mechanically. Still during the early 1980s, the most common input way for data or programs had been the punched tape or tape recorder. Most of today's computers are **von-Neumann-architectures**. Named after the Hungarian mathematician John von Neumann who first stated the general requirements for an electronic computer in his 1945 papers. Since then, virtually all computers have followed this basic design, which differed from earlier computers programmed through '*hard wiring*'. Standard CPUs are designed to provide a good instruction mixture for almost all commonly used algorithms. Therefore, for a class of target algorithms they cannot be as effective as possible in terms of design freedom. Most software is intended to be run on such general purpose computers having one single central processing unit (*CPU*). A problem is splitted into a discrete series of instructions using these computers. Each instruction is executed one after the other and only a single instruction may be executed at any moment in time.

All SciEngines machines, especially RIVYERA, follow a totally parallel architectural concept. It provides a large number of field programmable gate arrays (*FPGAs*), which are able to implement a huge number of individual processing elements. In the most simple sense, **FPGA parallel computing** is the simultaneous use of multiple computational resources like processing elements to solve large computational problems. The RIVYERA API allows a simple and assisted way of generating hundreds of processing elements per FPGA. To solve a complex task, it is split into discrete parts that can be solved concurrently. Each part is computed in its own processing element. Unlike a classical CPU, the discrete parts are further split to a series of instructions which are executed in highly problem optimized dedicated hardware. This hardware task is coded in the hardware description language VHDL. The instructions from each part are executed simultaneously

on different processing elements and FPGAs.

General computational problems usually demonstrate characteristics such as the ability to be split into discrete pieces of work that can be solved simultaneously and execute multiple program instructions at any moment in time. Therefore, problems are solved in less time with SciEngines RIVYERA than with a single computational resource like a CPU.

1.2 Concept of using SciEngines RIVYERA

To efficiently use SciEngines RIVYERA, the computational problem or algorithm is split in two general parts (see figure 1). One part is the strict software or frontend part which remains on the integrated host PC inside the RIVYERA machine. The other part is the core algorithm which is accelerated by using the FPGAs on a single RIVYERA machine or even on multiple RIVYERA machines. The FPGAs programmable by the user are further referenced to as *UserFPGAs*.

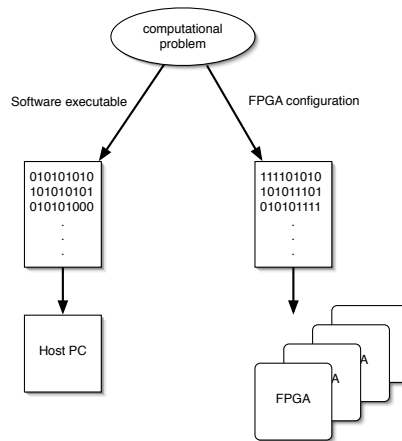


Figure 1: Partitioning of a problem into host- and machine-parts

In general, the software part could be seen as a frontend for the user or as a data interface to provide the resources for the FPGA accelerated parts. Also, simple pre- or postcomputations are ideal for this part. The RIVYERA Host-API offers rich interface functions which are easily adaptable into existing code.

CAUTION

In a massively parallel architecture the **flow control** should always be a point to think about. To achieve the best speedup, the flow control should be done **within the Machine-API**, e.g. by designing a special FPGA entity. Compared to FPGA architectures, PC architectures react much slower, because incoming events always have to be analyzed by schedulers, memory managers and other OS components. Therefore, the programmer always adds an artificial delay when allowing the FPGAs to wait for a PC reaction. Flow control in your PC software using the Host-API is still fast and quick to implement but might not result in the speedup your design is capable of.

The second part implements the acceleration, flow control and multiple processing elements to solve the computational problem. The RIVYERA Machine-API offers useful adequate functions which easily allow you to implement the key parts of the algorithm. To free your mind, it allows you an implementation without taking care of low level communication and multiplication of your processing elements.

To create the host part and the machine part of your application, different software tools are useful. On the host side, high level languages such as C or C++ and even Java are addressed by the RIVYERA Host-API. In order to design efficient processing elements, VHDL or Verilog is recommended. Implementations using cross-language compilers like SystemC are possible, but will most likely not result in the expected speedups.

In order to move any suitable computational problem to the RIVYERA machine, the computational problem should be partitioned into the two mentioned parts (see figure 2). For the integrated frontend on the host PC, the usage of any suitable compiler and development environment will create adequate results. As an IDE, we would like to suggest Eclipse. We would also like to recommend the usage of the Gnu C Compiler (*gcc*) or any comparable Unix based compiler in order to create executable code on the integrated RIVYERA Host PC¹. Machines shipped with Unix based operating systems, like Linux, usually provide a preinstalled *gcc* or equivalent compiler. All available RIVYERA machines provide templates for several programming languages like C/C++ or Java.

On the FPGA, we recommend the usage of the XILINX[®] ISE[®] development environment. Most third party compilers and IDEs might work as there are no other templates included except the ones provided for ISE[®]. Using the RIVYERA Machine-API allows simple interfacing of your VHDL-implemented processing elements.

¹RIVYERA API has been tested with Linux/*gcc*. Other compilers may work but are not officially supported.

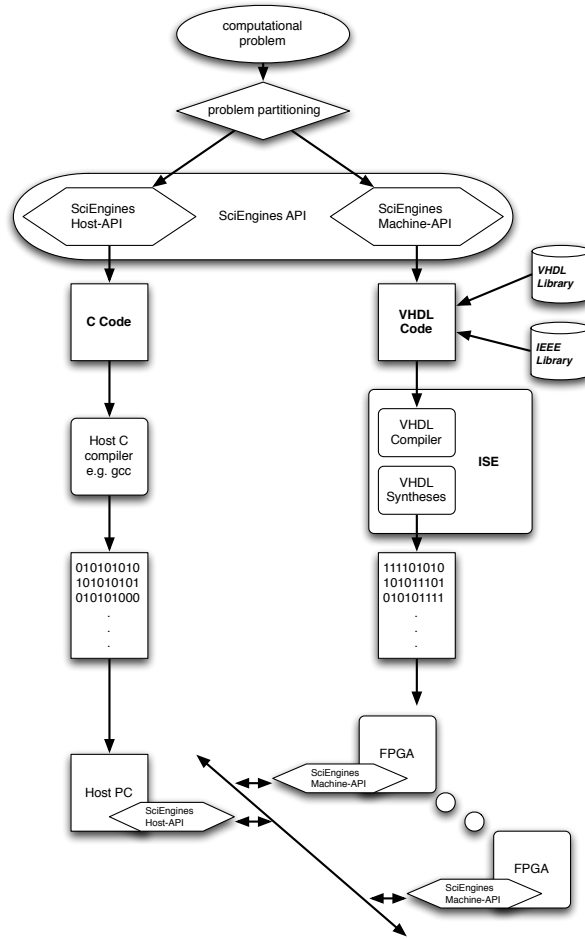


Figure 2: Design flow for multicomponent software systems

1.3 API version information

The SciEngines API follows a simple versioning scheme. All API versions are denoted aa.bb.cc s with the symbols as follows.

- aa: **Major API version**
Major API version changes indicate that the complete code structure will have to be changed if migrating. A changing Major version often indicate complete restructurings of the APIs code and therefore have a very long interval.
- bb: **Minor API version**
A change in the API minor version will be performed if new features will be supported.
- cc: **API Service Pack** (sometimes abbreviated with *SP*)
The API Service Pack will increase if there have been bug fixes.
- s: **API revision string**
The revision string can be an arbitrary string to explain the version. For example, "RC1" as a revision string may indicate that this is the *first release candidate* of a new API version.

Within this scheme, there is one specific characteristic. All versions with $b \geq 90$ are pre-release versions of a higher major version. For example, API 1.90.00 is the first alpha version of API 2.00.00.

1.4 RIVYERA API Addressing Scheme

The addressing scheme in the RIVYERA API is straightforward. Every single data word travels through the machine containing two addresses. One (the so called *target*) of these contains information where it should be sent to, the other one (so called *source*) tells the receiver where this word originated. Each address is built from multiple components which will be explained below.

1.4.1 Physical Address Components

To gain highest possible flexibility, every FPGA in the whole RIVYERA is uniquely identifiable and can therefore be addressed individually. The addressing scheme contains two physical fields: *Slot* and *FPGA address*. These fields are derived from the physical machine structure. Every RIVYERA machine physically consists of one or more FPGA Cards, each of which is plugged into a backplane slot. All plugged cards are numbered from index 0 to index `CARD_COUNT-1`, retaining their physical order. The index of each card is called its slot index. Multiple FPGAs may reside on each card. Similar to the cards in one system, the FPGAs are numbered in order, starting at index 0 as well. However, all FPGAs on one card share the same slot index. Using both the slot and FPGA index, every FPGA may be addressed uniquely throughout a whole RIVYERA machine.

1.4.2 Address Wildcards

Physical Address Components may be replaced by wildcards, such as `ADDR_SLOT_ALL` or `ADDR_FPGA_ALL`. Using these wildcards, it is possible to create broadcast- or very simple multicast-addresses. For example `slot=ADDR_SLOT_ALL, fpga=0` refers to the first FPGA on all cards, whereas `slot=0, fpga=ADDR_FPGA_ALL` selects all FPGAs on slot 0. `slot=ADDR_SLOT_ALL, fpga=ADDR_FPGA_ALL` of course selects every FPGA on every slot.

1.4.3 Virtual Address Components

The addressing scheme is completed by two more fields: *command* and *register*. Both fields do not have any physical means but are only useful for communication. The *command* field may be one of *read* or *write*. Whereas *write* commands do not imply a dedicated behaviour on the FPGA side, *read* commands assume a proper answer. Please see section 2.5.1 (Responding to Read Requests) in the VHDL-Documentation for more information. The *register* address field **MAY** be used to create multiple data streams. It can be considered as a stream identifier. As either sent and received words always contain information about their source and target register the user is given a very powerful possibility to create and design his very own dataflows. A very common way to use the *register* field is to use different types of streams for each *register*. For example, consider an FPGA design which has two

calculation cores which have to be fed with independent data. In this example, it would make sense to use register 0 for core 1 and register 1 for core 2. Please note that using multiple registers does not affect communication bandwidth.

1.4.4 Target Addresses

A target address specifies where a given data word is to be delivered to and how the target shall interpret the incoming word. For example, incoming words with `api_i_tgt_cmd_out = CMD_WR` tells the target FPGA that the sender does not expect an answer. Whenever `api_i_tgt_cmd_out = CMD_RD` your user logic is expected to send a number of words specified in `api_i_data_out` back to the sender.

Please note that as a receiver, you will not see the target slot and FPGA fields of an incoming word, because these are given implicitly by data receipt.

1.4.5 Source Addresses

Source addresses contain information about the source of an incoming data word. While a source's slot and FPGA information is straightforward, the *command* and *register* fields are more complex to understand. In general, both *source command* and *source register* do not have to be taken into account. Whenever the user FPGA receives data from the host interface, the *source command* will be `CMD_WR` and the *source register* will be set to `0x0`. However, you are free to implement designs that effectively use these fields within inter-FPGA-communication, for example to tell the receiver to send responds to a defined target address.

2 RIVYERA API Structure

The RIVYERA is designed as a linear systolic array of FPGAs. This means that every FPGA is only connected to its predecessor and its successor. Hence, all data uses the same transport channel and in order to maintain the correctness of order, data frames are not allowed to overtake each other. These specific features have to be kept in mind when designing your code for RIVYERA.

2.1 RIVYERA API Register Paradigm

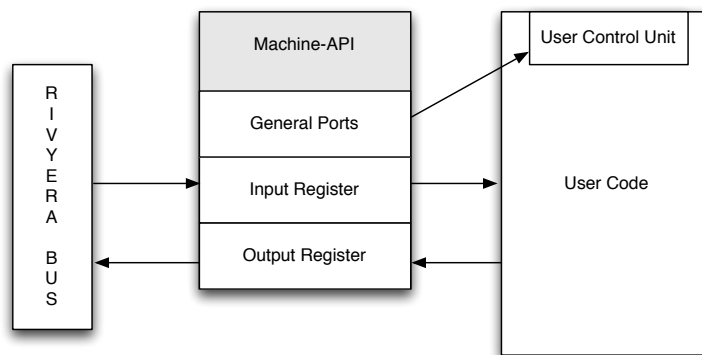


Figure 3: VHDL-API taking care of user design's I/O

Figure 3 shows the block diagram of one example of an FPGA design. The host interface provided by the Machine-API is instantiated once and connects to an addressed FPGA.

This design paradigm will be modelled by the Machine-API and, accordingly, by the Host-API.

■ Input Register

The SciEngines RIVYERA API enables the user to send and receive streamed data to and from an FPGA. Using this mechanism it is possible to send data from host to one or multiple FPGAs as well as transfer data between FPGAs and send data from FPGA to the host. A streams consists of individual 64 bit data words which are transferred in order. This means: Words written earlier to an FPGA arrive earlier than words which are written later.

■ Output Register

The SciEngines RIVYERA API provides a single register which can be used to send data. Whenever the user wants to send data to either the host PC or any other (possibly multiple) FPGA(s), he may provide data to this output register.

Both Input- and Output Register are realized as BlockRAM FIFOs.

2.2 RIVYERA API Routing Strategies

SciEngines API will support multiple routing-schemes, so the RIVYERA can be adapted according to each user's needs. Currently, the only supported routing scheme is Smart Routing.

2.2.1 Smart Routing

The Smart Routing strategy, which is enabled by default, will determine the shortest route through the RIVYERA for every sent word. It will make full usage of the machine's architecture with its Card-to-Card shortcuts.

Broadcasted transfers will automatically be spreaded in both communication directions to reduce the worst-case latency. The following illustrations sketch one FPGA Card with 8 FPGAs. The sender of a word is always coloured in bright green, whereas the links that are used to pass a word are highlighted red. Please note that exactly the same routing method applies to FPGA Cards with different numbers of FPGAs.

Figure 4 depicts the route of a word written to all FPGAs by the Host-Application. The host-connected Service FPGA duplicates the word and sends it to its User FPGAs using both ring directions. All FPGAs but three and four do both: forwarding the incoming word to their successors and forwarding it to the internal user User Logic. The FPGAs three and four forward the word to their User Logic, but do not forward it to the next FPGA. Therefore, no FPGA gets the word twice.

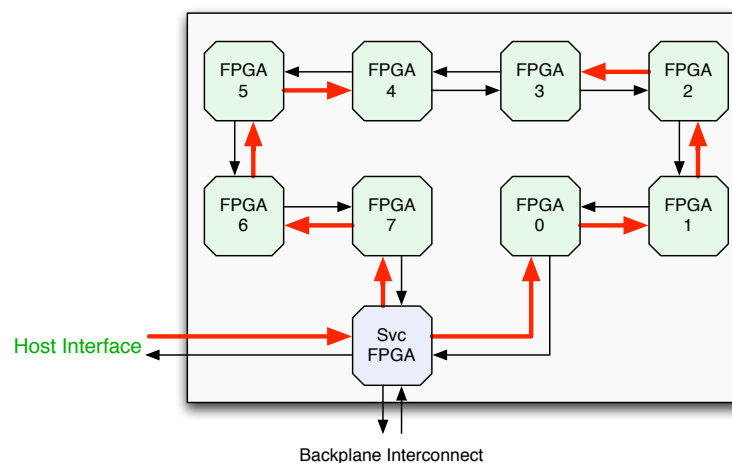


Figure 4: Routing of a host-initiated write

The same principle of routing applies for FPGA ↔ FPGA Transfers as shown in Figure 5. If an FPGA issues a broadcast, then it is broadcasted to both directions and it is assured by the API that no FPGA gets the same word twice.

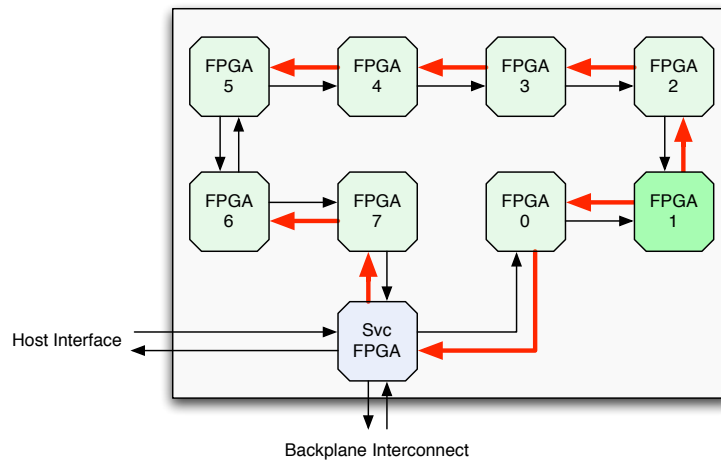


Figure 5: Routing of an fpga-initiated write

3 Java API Introduction

The RIVYERA Host-API forms one endpoint of host-machine communication. It models the Input/Output/Control register paradigm as introduced in section 2.1. Input registers of a FPGA can be filled using `se_write()`, control registers are accessed via `se_regRead()` and `se_regWrite()` and the FPGA output register is read using `se_read()`. Note that reading an output register does not physically read a register, but tells the FPGA to send data to the controller (see Machine-API documentation). Additionally, reading an output register has to be distinguished between *active* and *passive* reading. When issuing an active read request, the user's FPGA design will be actively asked to send some data, whereas passive reads only seek through words that are already written to the host. The Host-API contains a second write method called `se_message()`. This method behaves exactly like `se_write()` but only it does send its messages with a special *message command*. This command may be interpreted by VHDL-Code in any way so the programmer is given a tool to distinguish two different write types.

The programming of FPGAs is done by `se_program()`, which takes a bitfile to download it to the selected FPGAs.

The SciEngines RIVYERA API is completed with management functions such as `se_getSlotCount()` or `se_getFPGAInfo()` which makes it possible to figure out the whole machine's setup without having physical access to it.

3.1 Machine addressing

The addressing of machine components in general is done straightforward using the class `SeAddress`. The user needs to specify an element by its index, so `addr.fpga = 0` means to address the first FPGA. The only complex feature is Multi-/Broadcasting mode. Whenever you specify a component of `SeAddress` as `SE_ADDR_ALL`, you tell the API to address *all* of these components (so `addr.fpga = SE_ADDR_ALL` would address

all FPGAs). This way you can create Multicast addresses (e.g. `addr.slot = SE_ADDR_ALL`, `addr.fpga = 0` for the first FPGA on all cards), or true Broadcast addresses (`addr.slot = SE_ADDR_ALL`, `addr.fpga = SE_ADDR_ALL`).

3.2 Autonomous FPGA writes

There might be some cases in which the FPGAs need to communicate with the host software without being requested to. For convenience, these FPGA write actions will be called *autonomous writes*. Whenever your design needs to make use of this communication method, the Host-API method `se_waitForData()` comes in handy. When invoked, this method listens for write interrupts. It does return if it recognizes that data is sent to the specified controller. Once the method has returned, it provides the user with information of the write source, so the user can invoke `se_read()` in order to read the incoming data.

3.3 Cross-language example code

In this section, a simple straight forward example of the usage of the RIVYERA Host-API and the RIVYERA Machine-API is presented.

With the given example, the main aspects of host-machine intercommunication are given. It contains normal intercommunication as well as *autonomous fpga writes* which were introduced in section 3.2. The design uses one input register and one control register.

The VHDL part of this example design performs the following steps

1. *Reading the incoming data*
The incoming data is read and the value of the control register is added to it. After this it gets stored. Additionally, the FPGA starts a counter.
2. *Waiting for host to read data*
Once an incoming read request is recognized, the FPGA reacts by writing the manipulated data to the read request's source.
3. *Writing the counter value to host*
After the FPGA sent its manipulated data, it writes the value of the counter to the host without being requested.

The host part of the design consists of the following steps

1. *Initialization*
During initialization, the machine is allocated and memory for transfer is allocated.
2. *Register write*
The value to be added to the incoming data is written to the control register.

3. *Data write*

The data is written to machine.

4. *Data read*

The manipulated data is read from machine.

5. *Wait for FPGA to send data*

The VHDL part will send data on its own, so the host part needs to be ready for that.

6. *Read the waiting data*

Once `se_waitForData()` returns, the data is ready to be read.

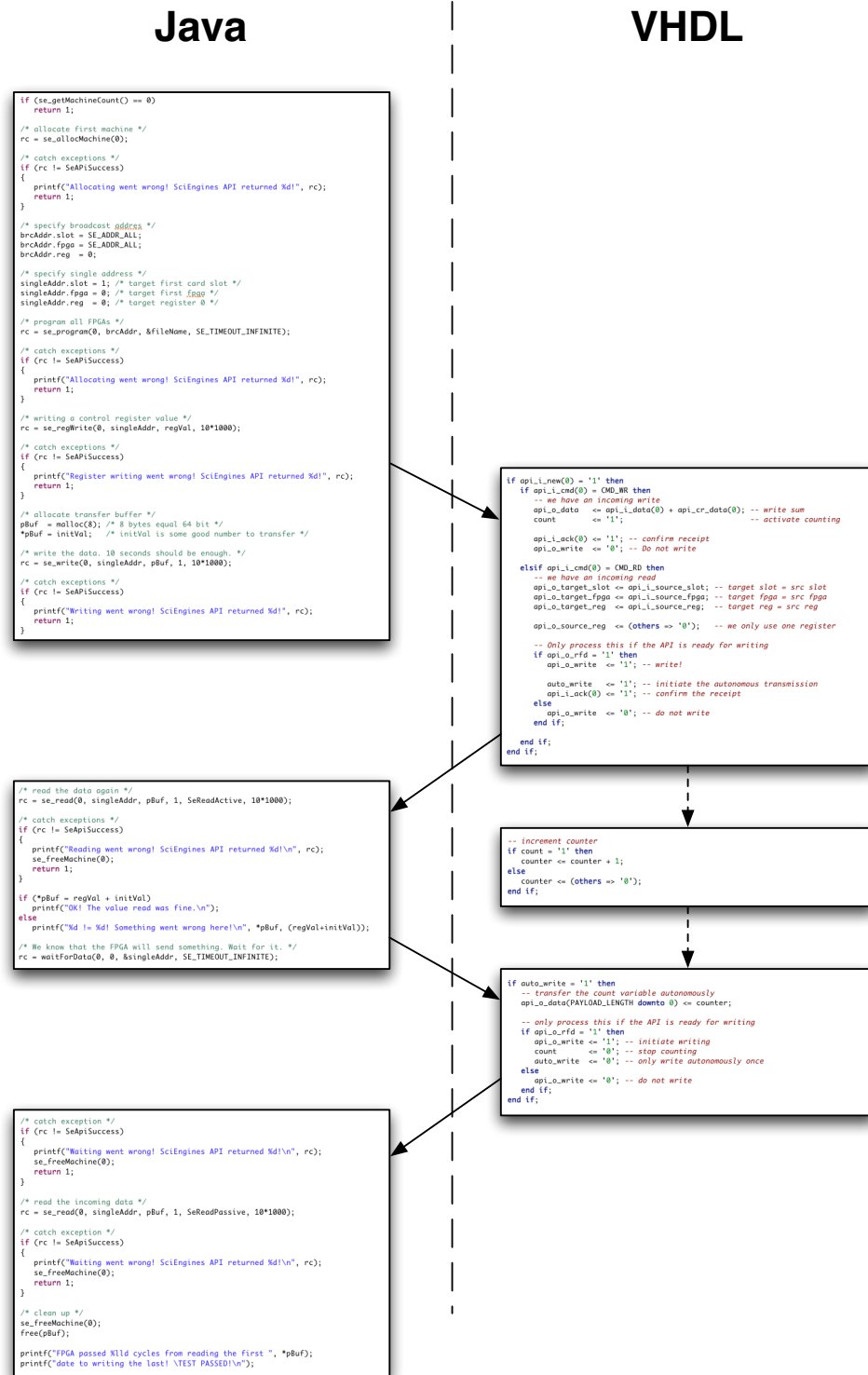


Figure 6: Java ↔ VHDL Co-Programming

Class Documentation

4 SciEngines_API Class Reference

Static Public Member Functions

- static int [se_getMachineCount](#) ()
- static void [se_allocMachine](#) (int machine, [SeOptions](#) options) throws [SeApiException](#)
- static void [se_freeMachine](#) (int machine) throws [SeApiException](#)
- static long [se_read](#) (int machine, [SeAddress](#) addr, ByteBuffer payload, long size, int mode, long timeout) throws [SeApiException](#)
- static long [se_write](#) (int machine, [SeAddress](#) addr, ByteBuffer payload, long size, long timeout) throws [SeApiException](#)
- static void [se_program](#) (int machine, [SeAddress](#) addr, String filename, long timeout) throws [SeApiException](#)
- static void [se_deprogram](#) (int machine, [SeAddress](#) addr) throws [SeApiException](#)
- static [SeAddress](#) [se_waitForData](#) (int machine, int controller, long timeout) throws [SeApiException](#)
- static int [se_getSlotCount](#) (int machine) throws [SeApiException](#)
- static [SeSlotInfo](#) [se_getSlotInfo](#) (int machine, int slot) throws [SeApiException](#)
- static int [se_getFPGACount](#) (int machine, int slot) throws [SeApiException](#)
- static [SeFPGAInfo](#) [se_getFPGAInfo](#) (int machine, [SeAddress](#) addr) throws [SeApiException](#)
- static int [se_getControllerCount](#) (int machine) throws [SeApiException](#)
- static [SeControllerInfo](#) [se_getControllerInfo](#) (int machine, int controller) throws [SeApiException](#)
- static double [se_getTemperature](#) (int machine, int slot) throws [SeApiException](#)
- static double [se_getMaxTemperature](#) (int machine, int slot) throws [SeApiException](#)
- static void [se_flush](#) (int machine, int controller, long timeout) throws [SeApiException](#)
- static void [se_program_slave](#) (int machine, [SeAddress](#) addr, String filename, long timeout) throws [SeApiException](#)
- static void [se_deprogram_slave](#) (int machine, [SeAddress](#) addr) throws [SeApiException](#)
- static void [se_comment](#) (String str)

4.1 Detailed Description

This is the central class of the SciEngines API. It contains all methods used to communicate with a SciEngines device.

Author:

Jost Bissel

Daniel Siebert

4.2 Member Function Documentation

- 4.2.1 `static int se_getMachineCount () [static]`
- 4.2.2 `static void se_allocMachine (int machine, SeOptions options)
throws SeApiException [static]`
- 4.2.3 `static void se_freeMachine (int machine) throws SeApiException
[static]`
- 4.2.4 `static long se_read (int machine, SeAddress addr, ByteBuffer
payload, long size, int mode, long timeout) throws
SeApiException [static]`
- 4.2.5 `static long se_write (int machine, SeAddress addr, ByteBuffer
payload, long size, long timeout) throws SeApiException
[static]`
- 4.2.6 `static void se_program (int machine, SeAddress addr, String
filename, long timeout) throws SeApiException [static]`
- 4.2.7 `static void se_deprogram (int machine, SeAddress addr) throws
SeApiException [static]`
- 4.2.8 `static SeAddress se_waitForData (int machine, int controller,
long timeout) throws SeApiException [static]`
- 4.2.9 `static int se_getSlotCount (int machine) throws SeApiException
[static]`
- 4.2.10 `static SeSlotInfo se_getSlotInfo (int machine, int slot) throws
SeApiException [static]`
- 4.2.11 `static int se_getFPGACount (int machine, int slot) throws
SeApiException [static]`
- 4.2.12 `static SeFPGAInfo se_getFPGAInfo (int machine, SeAddress
addr) throws SeApiException [static]`
- 4.2.13 `static int se_getControllerCount (int machine) throws
SeApiException [static]`
- 4.2.14 `static SeControllerInfo se_getControllerInfo (int machine, int
controller) throws SeApiException [static]`
- 4.2.15 `static double se_getTemperature (int machine, int slot) throws
SeApiException [static]`
- 4.2.16 `static double se_getMaxTemperature (int machine, int slot)
throws SeApiException [static]`
- 4.2.17 `static void se_flush (int machine, int controller, long timeout)
throws SeApiException [static]`
- 4.2.18 `static void se_program_slave (int machine, SeAddress
addr, String filename, long timeout) throws SeApiException`

5 SciEngines_API_Const Class Reference

Static Public Attributes

- static final int [SE_API_VERSION_MAJOR](#) = SciEngines_API_Const_JNI.SE_API_VERSION_MAJOR
- static final int [SE_API_VERSION_MINOR](#) = SciEngines_API_Const_JNI.SE_API_VERSION_MINOR
- static final int [SE_API_VERSION_SP](#) = SciEngines_API_Const_JNI.SE_API_VERSION_SP
- static final String [SE_API_VERSION_REVISION](#) = SciEngines_API_Const_JNI.SE_API_VERSION_REVISION
- static final int [SE_TIMEOUT_INFINITE](#) = SciEngines_API_Const_JNI.SE_TIMEOUT_INFINITE
- static int [SE_ADDR_FPGA_ALL](#) = SciEngines_API_Const_JNI.SE_ADDR_FPGA_ALL
- static int [SE_ADDR_SLOT_ALL](#) = SciEngines_API_Const_JNI.SE_ADDR_SLOT_ALL
- static int [SE_ADDR_CONTR_ALL](#) = SciEngines_API_Const_JNI.SE_ADDR_CONTR_ALL
- static int [SE_ADDR_FPGA_HOST](#) = SciEngines_API_Const_JNI.SE_ADDR_FPGA_HOST
- static int [SE_ADDR_REG_EOT](#) = SciEngines_API_Const_JNI.SE_ADDR_REG_EOT
- static int [SE_LENGTH_ADDR_SLOT](#) = SciEngines_API_Const_JNI.SE_LENGTH_ADDR_SLOT
- static int [SE_LENGTH_ADDR_FPGA](#) = SciEngines_API_Const_JNI.SE_LENGTH_ADDR_FPGA
- static int [SE_LENGTH_ADDR_REG](#) = SciEngines_API_Const_JNI.SE_LENGTH_ADDR_REG
- static int [SE_LENGTH_CMD](#) = SciEngines_API_Const_JNI.SE_LENGTH_CMD
- static int [SE_READ_ACTIVE](#) = SciEngines_API_Const_JNI.SE_READ_ACTIVE
- static int [SE_READ_PASSIVE](#) = SciEngines_API_Const_JNI.SE_READ_PASSIVE

5.1 Member Data Documentation

5.1.1 final int SE_API_VERSION_MAJOR = SciEngines_API_Const_JNI.SE_API_VERSION_MAJOR [static]

Major API version.

**5.1.2 final int SE_API_VERSION_MINOR = SciEngines_ -
API_Const_JNI.SE_API_VERSION_MINOR
[static]**

Minor API version.

**5.1.3 final int SE_API_VERSION_SP = SciEngines_API_Const_JNI.SE_ -
API_VERSION_SP [static]**

API Service Pack.

**5.1.4 final String SE_API_VERSION_REVISION =
SciEngines_API_Const_JNI.SE_API_VERSION_REVISION
[static]**

API Revision.

**5.1.5 final int SE_TIMEOUT_INFINITE = SciEngines_API_Const_ -
JNI.SE_TIMEOUT_INFINITE [static]**

Constant used whenever a method shall wait infinitely.

**5.1.6 int SE_ADDR_FPGA_ALL = SciEngines_API_Const_JNI.SE_ -
ADDR_FPGA_ALL [static]**

Constant used as wildcard for FPGA index. This constant may be used for writing to multiple FPGAs or programming multiple FPGAs at once. E.g. slot = 1, fpga = ADDR_FPGA_ALL specifies a Multicast to every FPGA in slot 1.

**5.1.7 int SE_ADDR_SLOT_ALL = SciEngines_API_Const_JNI.SE_ -
ADDR_SLOT_ALL [static]**

Constant used as wildcard for slot index. This constant may be used for writing to multiple slots or programming multiple slots at once. E.g. slot = SE_SLOT_ALL, fpga = 3 specifies a Multicast to each FPGA 3 in every slot.

**5.1.8 int SE_ADDR_CONTR_ALL = SciEngines_API_Const_JNI.SE_ -
ADDR_CONTR_ALL [static]**

Constant used as wildcard for controller index. This constant may be used for se_waitForData to wait on all controllers for incoming data.

5.1.9 int SE_ADDR_FPGA_HOST = SciEngines_API_Const_JNI.SE_ADDR_FPGA_HOST [static]

Constant used whenever you need to communicate to the host. E.g. slot = 0, fpga = SE_ADDR_FPGA_HOST initiates a transfer to the host interface at slot 0.

5.1.10 int SE_ADDR_REG_EOT = SciEngines_API_Const_JNI.SE_ADDR_REG_EOT [static]

Constant used for ending a transfer. This can only be used from within user FPGA.

5.1.11 int SE_LENGTH_ADDR_SLOT = SciEngines_API_Const_JNI.SE_LENGTH_ADDR_SLOT [static]

Length of the slot address field in bits.

5.1.12 int SE_LENGTH_ADDR_FPGA = SciEngines_API_Const_JNI.SE_LENGTH_ADDR_FPGA [static]

Length of the fpga address field in bits.

5.1.13 int SE_LENGTH_ADDR_REG = SciEngines_API_Const_JNI.SE_LENGTH_ADDR_REG [static]

Length of the register address field in bits.

5.1.14 int SE_LENGTH_CMD = SciEngines_API_Const_JNI.SE_LENGTH_CMD [static]

Length of the command field in bits.

5.1.15 int SE_READ_ACTIVE = SciEngines_API_Const_JNI.SE_READ_ACTIVE [static]

Constant used to invoke active read mode.

5.1.16 int SE_READ_PASSIVE = SciEngines_API_Const_JNI.SE_READ_PASSIVE [static]

Constant used to invoke passive read mode.

6 SeAddress Class Reference

Public Member Functions

- [SeAddress](#) (int *contr*, int *slot*, int *fpga*, int *reg*)
- String [toString](#) ()

Public Attributes

- int *fpga* = 0
- int *reg* = 0
- int *slot* = 0
- int *contr* = 0

6.1 Detailed Description

A structure containing all necessary information to address a machine element. In order to create a Multi-/Broadcast address, use [SciEngines_API_Const#SE_ADDR_CONTR_ALL](#), [SciEngines_API_Const#SE_ADDR_SLOT_ALL](#), [SciEngines_API_Const#SE_ADDR_FPGA_ALL](#) on any of the components.

Author:

Jost Bissel
Daniel Siebert

6.2 Constructor & Destructor Documentation

6.2.1 [SeAddress](#) (int *contr*, int *slot*, int *fpga*, int *reg*)

Creates an [SeAddress](#) instance to address an FPGA

Parameters:

contr Controller index
slot Slot index
fpga FPGA index
reg FPGA's register index

6.3 Member Function Documentation

6.3.1 [String toString](#) ()

6.4 Member Data Documentation

6.4.1 int *fpga* = 0

The index of the target FPGA.

6.4.2 int reg = 0

The index of the target register.

6.4.3 int slot = 0

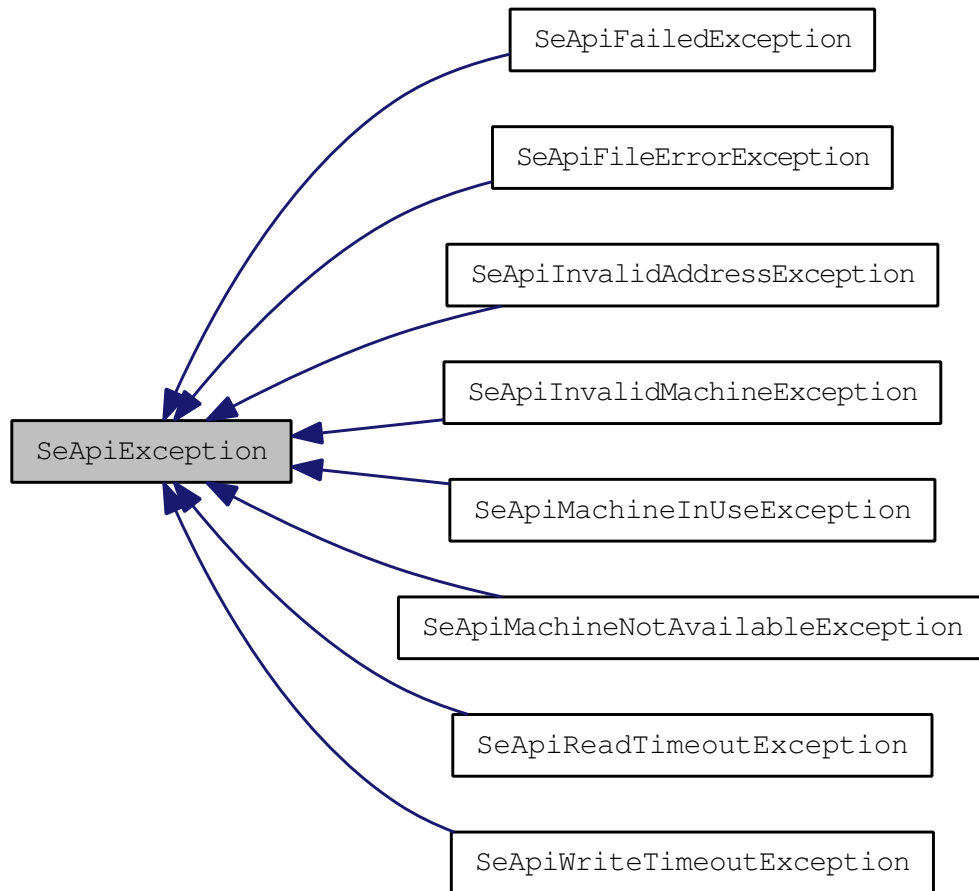
The index of the target slot.

6.4.4 int contr = 0

The index of the target controller.

7 SeApiException Class Reference

Inheritance diagram for SeApiException:



Public Member Functions

- [SeApiException](#) (int errorCode, String message)
- int [getErrorCode](#) ()

7.1 Detailed Description

Class representing exceptions that might occur while running SciEngines API. This is the superclass for all SeApiExceptions. To catch all SeApiExceptions, you may simply catch this superclass.

Author:

Jost Bissel

7.2 Constructor & Destructor Documentation

7.2.1 SeApiException (int *errorCode*, String *message*)

Creates a new instance using the given error code and message.

Parameters:

errorCode Integer specifying the error.

message String specifying the error.

7.3 Member Function Documentation

7.3.1 int getErrorCode ()

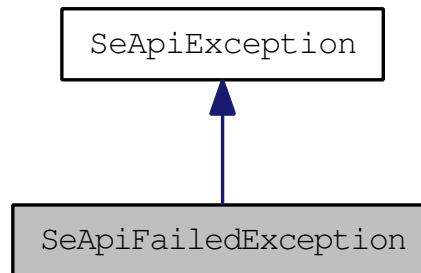
Returns the error code of this exception.

Returns:

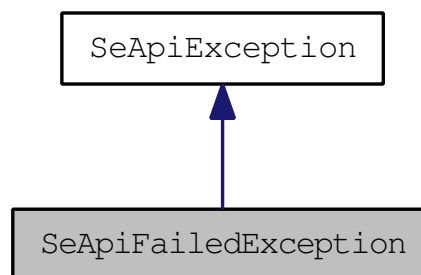
Exception's error code.

8 SeApiFailedException Class Reference

Inheritance diagram for SeApiFailedException:



Collaboration diagram for SeApiFailedException:



Public Member Functions

- [SeApiFailedException \(\)](#)
- [SeApiFailedException \(String message\)](#)
- `int getErrorCode \(\)`

8.1 Constructor & Destructor Documentation

8.1.1 [SeApiFailedException \(\)](#)

8.1.2 [SeApiFailedException \(String *message*\)](#)

8.2 Member Function Documentation

8.2.1 `int getErrorCode \(\)` [inherited]

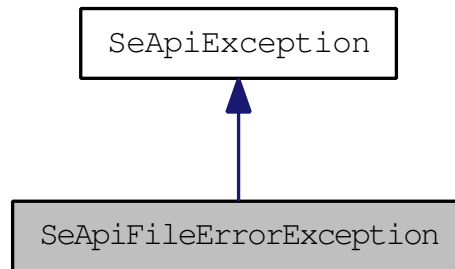
Returns the error code of this exception.

Returns:

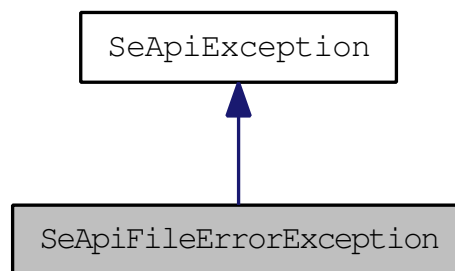
Exception's error code.

9 SeApiFileErrorException Class Reference

Inheritance diagram for SeApiFileErrorException:



Collaboration diagram for SeApiFileErrorException:



Public Member Functions

- [SeApiFileErrorException \(\)](#)
- [SeApiFileErrorException \(String message\)](#)
- [int getErrorCode \(\)](#)

9.1 Constructor & Destructor Documentation

9.1.1 SeApiFileErrorException ()

9.1.2 SeApiFileErrorException (String *message*)

9.2 Member Function Documentation

9.2.1 int getErrorCode () [inherited]

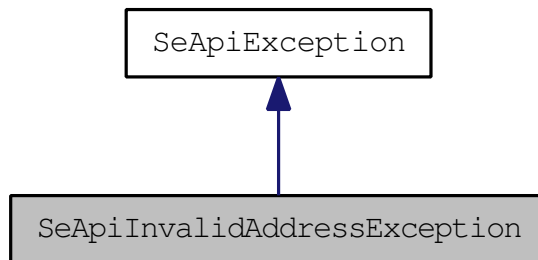
Returns the error code of this exception.

Returns:

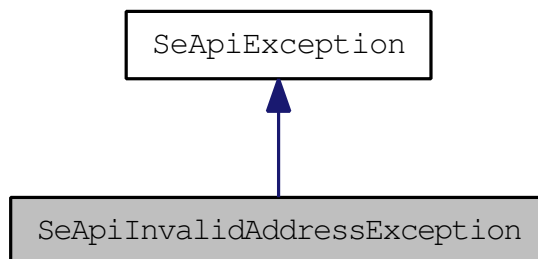
Exception's error code.

10 SeApiInvalidAddressException Class Reference

Inheritance diagram for SeApiInvalidAddressException:



Collaboration diagram for SeApiInvalidAddressException:



Public Member Functions

- [SeApiInvalidAddressException \(\)](#)
- [SeApiInvalidAddressException \(String message\)](#)
- [int getErrorCode \(\)](#)

10.1 Constructor & Destructor Documentation

10.1.1 [SeApiInvalidAddressException \(\)](#)

10.1.2 [SeApiInvalidAddressException \(String message\)](#)

10.2 Member Function Documentation

10.2.1 [int getErrorCode \(\)](#) `[inherited]`

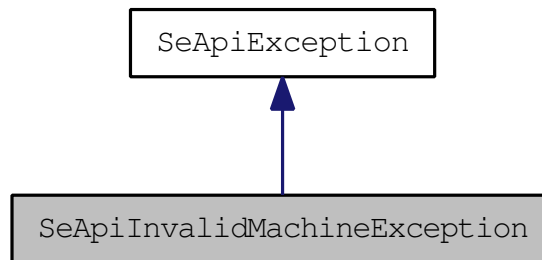
Returns the error code of this exception.

Returns:

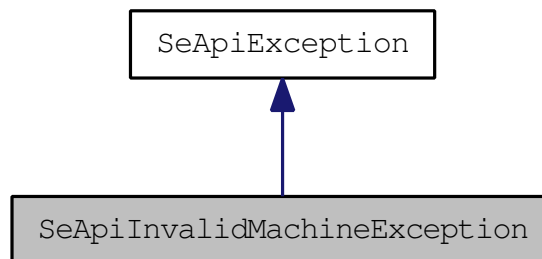
Exception's error code.

11 SeApiInvalidMachineException Class Reference

Inheritance diagram for SeApiInvalidMachineException:



Collaboration diagram for SeApiInvalidMachineException:



Public Member Functions

- [SeApiInvalidMachineException \(\)](#)
- [SeApiInvalidMachineException \(String message\)](#)
- [int getErrorCode \(\)](#)

11.1 Constructor & Destructor Documentation

11.1.1 SeApiInvalidMachineException ()

11.1.2 SeApiInvalidMachineException (String *message*)

11.2 Member Function Documentation

11.2.1 int getErrorCode () `[inherited]`

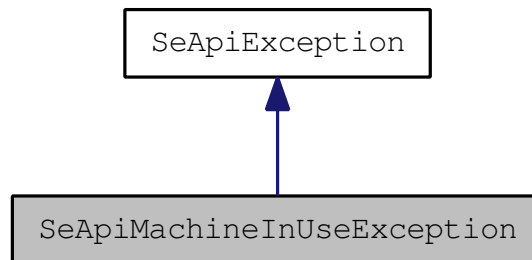
Returns the error code of this exception.

Returns:

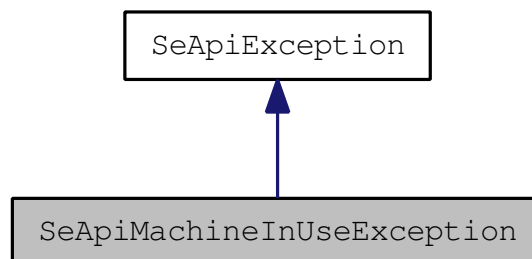
Exception's error code.

12 SeApiMachineInUseException Class Reference

Inheritance diagram for SeApiMachineInUseException:



Collaboration diagram for SeApiMachineInUseException:



Public Member Functions

- [SeApiMachineInUseException \(\)](#)
- [SeApiMachineInUseException \(String message\)](#)
- `int getErrorCode \(\)`

12.1 Constructor & Destructor Documentation

12.1.1 [SeApiMachineInUseException \(\)](#)

12.1.2 [SeApiMachineInUseException \(String *message*\)](#)

12.2 Member Function Documentation

12.2.1 `int getErrorCode \(\) [inherited]`

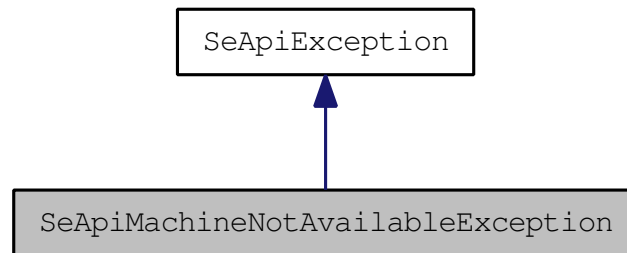
Returns the error code of this exception.

Returns:

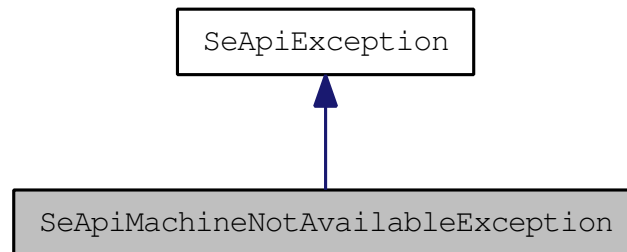
Exception's error code.

13 SeApiMachineNotAvailableException Class Reference

Inheritance diagram for SeApiMachineNotAvailableException:



Collaboration diagram for SeApiMachineNotAvailableException:



Public Member Functions

- [SeApiMachineNotAvailableException \(\)](#)
- [SeApiMachineNotAvailableException \(String message\)](#)
- [int getErrorCode \(\)](#)

13.1 Constructor & Destructor Documentation

13.1.1 [SeApiMachineNotAvailableException \(\)](#)

13.1.2 [SeApiMachineNotAvailableException \(String message\)](#)

13.2 Member Function Documentation

13.2.1 [int getErrorCode \(\)](#) [inherited]

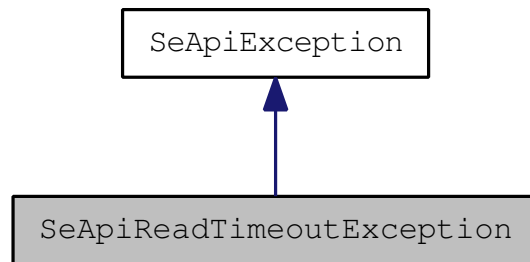
Returns the error code of this exception.

Returns:

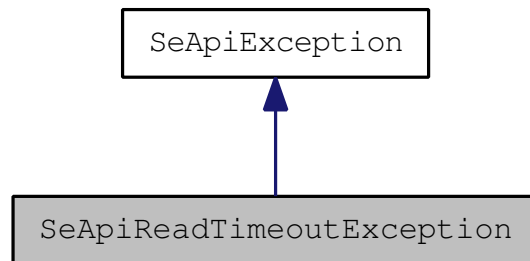
Exception's error code.

14 SeApiReadTimeoutException Class Reference

Inheritance diagram for SeApiReadTimeoutException:



Collaboration diagram for SeApiReadTimeoutException:



Public Member Functions

- [SeApiReadTimeoutException \(\)](#)
- [SeApiReadTimeoutException \(String message\)](#)
- `int getErrorCode \(\)`

14.1 Constructor & Destructor Documentation

14.1.1 [SeApiReadTimeoutException \(\)](#)

14.1.2 [SeApiReadTimeoutException \(String *message*\)](#)

14.2 Member Function Documentation

14.2.1 `int getErrorCode \(\) [inherited]`

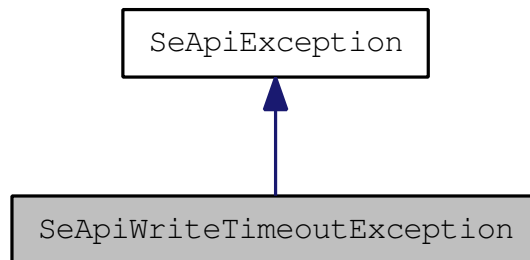
Returns the error code of this exception.

Returns:

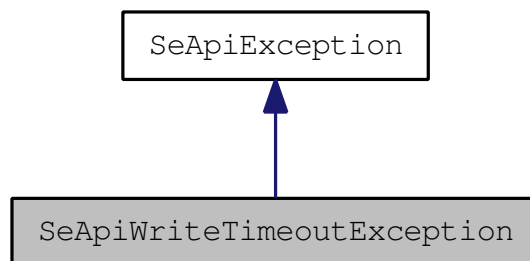
Exception's error code.

15 SeApiWriteTimeoutException Class Reference

Inheritance diagram for SeApiWriteTimeoutException:



Collaboration diagram for SeApiWriteTimeoutException:



Public Member Functions

- [SeApiWriteTimeoutException \(\)](#)
- [SeApiWriteTimeoutException \(String message\)](#)
- `int getErrorCode \(\)`

15.1 Constructor & Destructor Documentation

15.1.1 [SeApiWriteTimeoutException \(\)](#)

15.1.2 [SeApiWriteTimeoutException \(String *message*\)](#)

15.2 Member Function Documentation

15.2.1 `int getErrorCode \(\) [inherited]`

Returns the error code of this exception.

Returns:

Exception's error code.

16 SeControllerInfo Class Reference

Public Member Functions

- String [getDriverName](#) ()
- int [getMachineSlot](#) ()
- int [getSerial](#) ()
- String [toString](#) ()

16.1 Detailed Description

A class containing useful information about a controller.

Author:

Jost Bissel
Daniel Siebert

16.2 Member Function Documentation

16.2.1 String [getDriverName](#) ()

Returns:

The driver used to access this controller.

16.2.2 int [getMachineSlot](#) ()

Returns:

The machineSlot

16.2.3 int [getSerial](#) ()

Returns:

The serial

16.2.4 String [toString](#) ()

17 SeFPGAInfo Class Reference

Public Member Functions

- SeFPGAInfo [getType](#) ()
- boolean [isProgrammed](#) ()
- int [getFirmwareVersion](#) ()
- int [getFirmwareBuild](#) ()
- String [toString](#) ()

17.1 Detailed Description

A class containing useful information about an FPGA.

Author:

Jost Bissel
Daniel Siebert

17.2 Member Function Documentation

17.2.1 SeFPGAInfo [getType](#) ()

Returns:

The type

17.2.2 boolean [isProgrammed](#) ()

Indicates whether this fpga is programmed or not.

Returns:

Whether this fpga is programmed or not

17.2.3 int [getFirmwareVersion](#) ()

Returns:

The FPGA's firmware version.

17.2.4 int [getFirmwareBuild](#) ()

Returns:

The FPGA's firmware build.

17.2.5 String [toString](#) ()

18 SeOptions Class Reference

Public Types

- enum `SeWriteBehavior` { `se_write_async`, `se_write_sync` }
- enum `SeRoutingMethod` { `se_routing_normal`, `se_routing_systolic` }

Public Member Functions

- `SeOptions` (`SeWriteBehavior` writeBehavior, `SeRoutingMethod` routingMethod)
- `SeWriteBehavior` `getWriteBehavior` ()
- void `setWriteBehavior` (`SeWriteBehavior` writeBehavior)
- `SeRoutingMethod` `getRoutingMethod` ()
- void `setRoutingMethod` (`SeRoutingMethod` routingMethod)

18.1 Member Enumeration Documentation

18.1.1 enum `SeWriteBehavior`

Enumerator:

`se_write_async`
`se_write_sync`

18.1.2 enum `SeRoutingMethod`

Enumerator:

`se_routing_normal`
`se_routing_systolic`

18.2 Constructor & Destructor Documentation

18.2.1 `SeOptions` (`SeWriteBehavior` writeBehavior, `SeRoutingMethod` routingMethod)

18.3 Member Function Documentation

18.3.1 `SeWriteBehavior` `getWriteBehavior` ()

18.3.2 void `setWriteBehavior` (`SeWriteBehavior` writeBehavior)

18.3.3 `SeRoutingMethod` `getRoutingMethod` ()

18.3.4 void `setRoutingMethod` (`SeRoutingMethod` routingMethod)

19 SeSlotInfo Class Reference

Public Member Functions

- boolean [isController](#) ()
- int [getControllerIndex](#) ()
- int [getFpgaCount](#) ()
- int [getSerial](#) ()
- int [getPrevContr](#) ()
- int [getNextContr](#) ()
- int [getFirmwareVersion](#) ()
- int [getFirmwareBuild](#) ()
- String [toString](#) ()

19.1 Detailed Description

A class containing useful information about a slot.

Author:

Jost Bissel

19.2 Member Function Documentation

19.2.1 boolean [isController](#) ()

Returns:

True, if controller else false.

19.2.2 int [getControllerIndex](#) ()

Returns:

The index of the controller, if [isController\(\)](#) returns true

19.2.3 int [getFpgaCount](#) ()

Returns:

The fpgaCount

19.2.4 int [getSerial](#) ()

Returns:

The serial

19.2.5 int getPrevContr ()**Returns:**

This card's previous controller index.

19.2.6 int getNextContr ()**Returns:**

This card's next controller index.

19.2.7 int getFirmwareVersion ()**Returns:**

The FPGA's firmware version.

19.2.8 int getFirmwareBuild ()**Returns:**

The FPGA's firmware build.

19.2.9 String toString ()