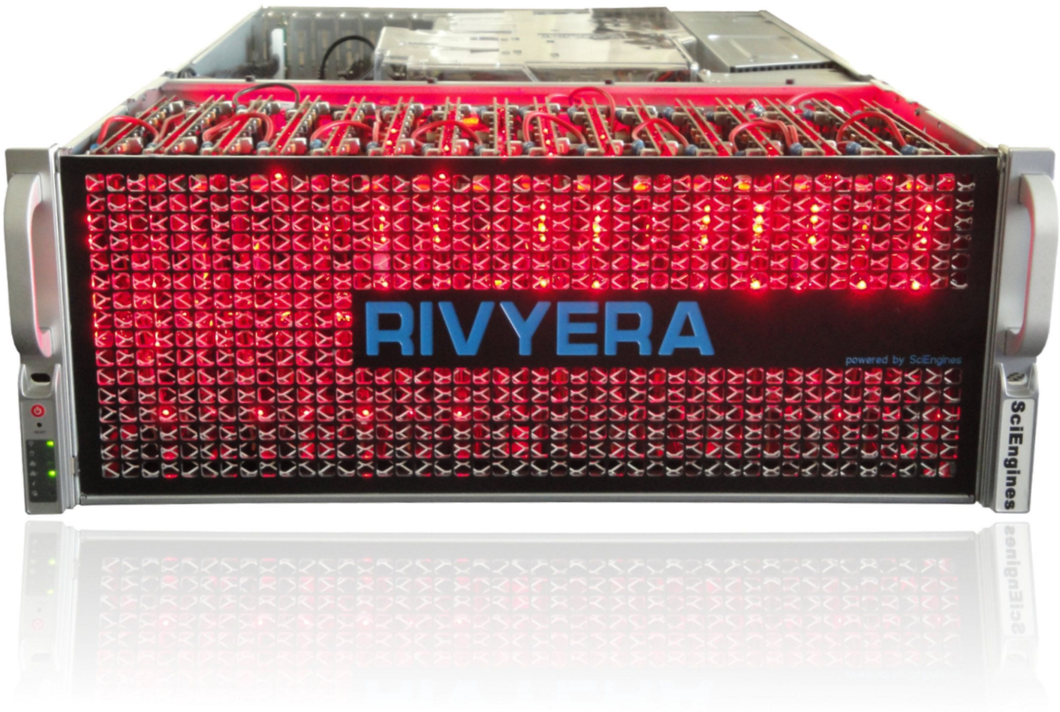
RIVYERA/API

RIVYERA API

SciEngines RIVYERA Host-API Documentation

Development User Guide, Host-API (Java)

Version 1.95.01



SciEngines GmbH

Revision: 1363 1.95.01 March 9, 2022

Released version

SciEngines Software Division

SciEngines RIVYERA Host-API Documentation

Development User Guide, Host-API (Java)
Version 1.95.01

SciEngines GmbH

SciEngines GmbH
Am-Kiel-Kanal 2
24106 Kiel
Germany

Public

Released version

Prepared for SciEngines GmbH, Am-Kiel-Kanal 2, 24106 Kiel, Germany
Under No Contract Number

Abstract: This introduction offers a brief overview of the SciEngines RIVYERA computer. It describes the physical and structural details from the programmers' point of view.

The main purpose of the RIVYERA API is to interface with single and multiple FPGAs in a massively parallel architecture as simply and easily as possible. We intend to provide an infrastructure for your FPGA designs which allows you to leverage the benefits of a massively parallel architecture without raising the complexity of your design.
Therefore, we provide a simple interface hiding the idiosyncratic implementation details of the physical layers while permitting a high-level view of your RIVYERA computer.

Contents

Figures and Tables

Figures

Tables

1 General

1.1 Basic Information

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

The main purpose of the RIVYERA API is to interface with single and multiple FPGAs in a massively parallel architecture as simply and easily as possible. We intend to provide an infrastructure for your FPGA designs which allows you to leverage the benefits of a massively parallel architecture without raising the complexity of your design.
Therefore, we provide a simple interface hiding the idiosyncratic implementation details of the physical layers while permitting a high-level view of your RIVYERA computer.

1.1.1 General ideas of parallel programming

Traditionally, software has been written for serial computation. There are two historic 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 way to input data or programs was via punched tape or magnetic tape drives. 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 split into a discrete series of instructions, each instruction is executed one after the other and only a single instruction may be executed at a time.

The SciEngines approach follows a massively parallelized 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 simplest case, **FPGA parallel computing** is the simultaneous use of multiple resources like processing elements to solve large computational problems. The RIVYERA API allows to interface hundreds of such 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.

1.1.2 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 computer. The other part is the core algorithm which is accelerated by using the FPGAs on a single RIVYERA computer or even on multiple RIVYERA computers. The FPGAs programmable by the user are referred to as *UserFPGAs*.

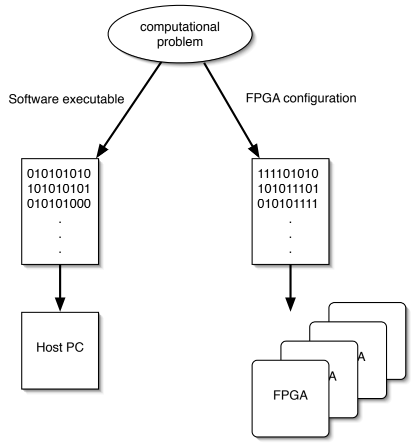


Figure 1. Partitioning of a problem into host- and machine-parts

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 post-computations are ideal for this part. The RIVYERA Host-API offers a rich set of interface functions which can be easily used by 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 functions which easily allows you to implement the key parts of the algorithm.

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 computer, 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. The recommended tools are Eclipse for the IDE and the Gnu C Compiler (*gcc*) or any comparable Unix based compiler in order to create executable code on the integrated RIVYERA Host PC [1]. Machines shipped with Unix based operating systems, like Linux, usually provide a pre-installed gcc or equivalent compiler. All available RIVYERA computers provide templates for several programming languages like C/C++ or Java.

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

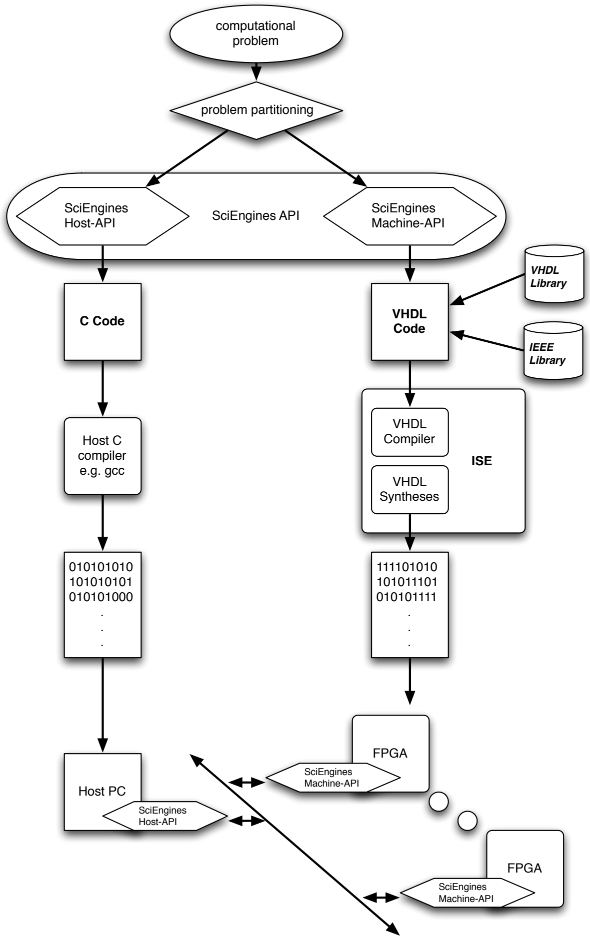


Figure 2. Design flow for multi-component software systems

For FPGA design and implementation, the recommended development environments for the differing RIVYERA architectures are:

- `RIVYERA S6-LX150`: XILINX® ISE® development environment.
- `RIVYERA X-32G1`: INTEL® QUARTUS® development environment.

Most third party compilers and IDEs might work as there are no other templates included except the ones provided for ISE® and QUARTUS®. Using the RIVYERA Machine-API allows simple interfacing of your VHDL-implemented processing elements.

1.1.3 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 triggered by new features.
- `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 annotating 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 caveat: All versions with `bb` $\geq$ `90` are pre-release versions of a higher major version. For example, API 1.90.00 was the first alpha version of API 2.00.00.

1.1.4 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 of these (the so called *target*) 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.

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 computer 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 computer.

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.

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 contain one of *read* or *write*. *Write* commands do not imply a dedicated behavior on the FPGA side, whereas *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 both sent and received words always contain information about their source and target register the user can leverage a very powerful feature to create and design his very own data-flows. A very common way to use the *register* field is to employ 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.

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.

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.

1.2 RIVYERA API Structure

In the RIVYERA architecture 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.

1.2.1 RIVYERA API Register Paradigm

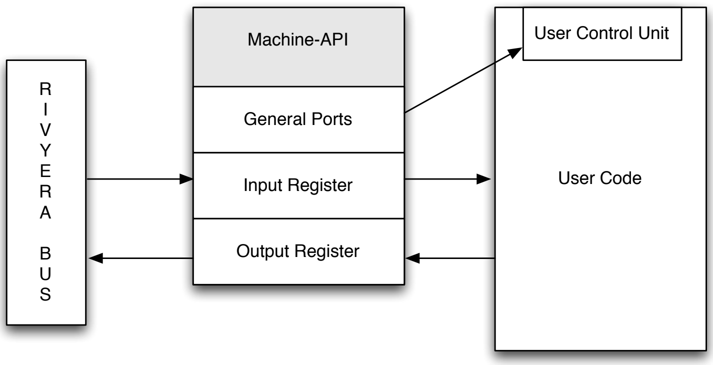


Figure 3. VHDL-API taking care of user design's I/O

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 modeled 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 FPGAs to the host. A stream 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.

1.2.2 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. All routing strategies are strictly deterministic. Therefore, every sent word takes exactly the same path through the RIVYERA, depending on its physical source and target address. SciEngines API does not avoid links with high traffic.

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 spread in both communication directions to reduce the worst-case latency. The following illustrations show one FPGA card with 8 FPGAs. The sender of a word is always colored 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 numbers 3 and 4 do both: forwarding the incoming word to their successors and forwarding it to the internal user User Logic. The FPGAs 3 and 4 forward the word to their own user logic, but do not forward it to the next FPGA. Therefore, no FPGA gets the word twice.

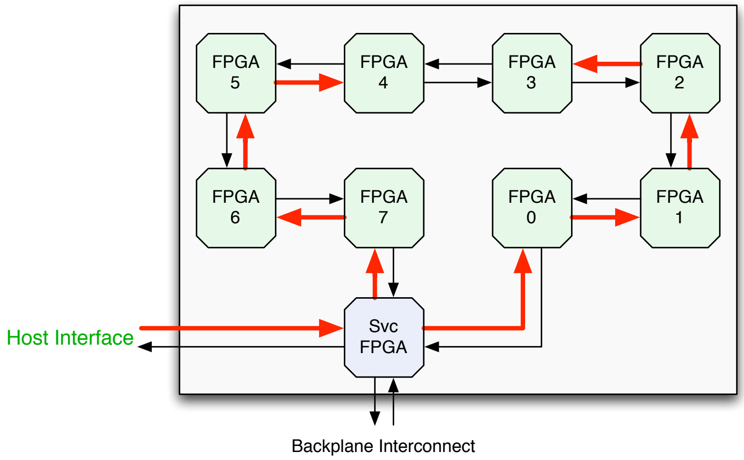


Figure 4. Routing of a host-initiated write

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 in both directions and it is assured by the API that no FPGA gets the same word twice.

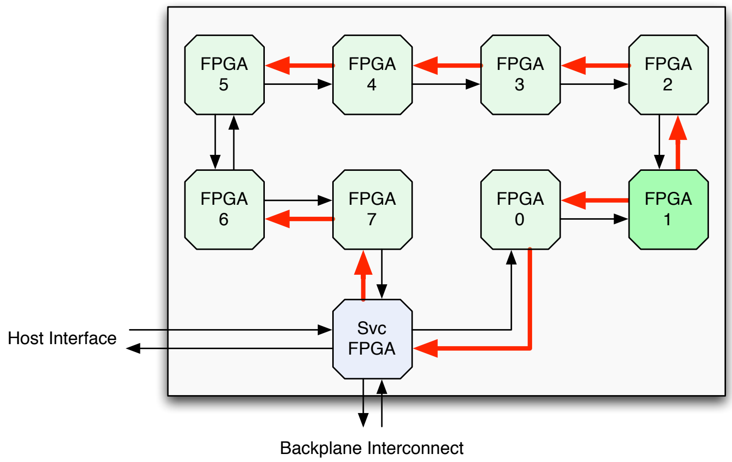


Figure 5. Routing of an FPGA-initiated write

1.3 Java API Introduction

The RIVYERA Host-API forms one endpoint of host-machine communication. It models the Input/Output register paradigm as introduced in section 1.2.1. Input registers of a FPGA can be filled using se_write(), and the FPGA output register is read using se_read(). Reading an output register has to be distinguish 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 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(), se_getSlotInfo() and se_getFPGAInfo() which makes it possible to figure out the whole machine's setup without having physical access to it.

1.3.1 Machine addressing

The addressing of machine components in general is 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 the slot or fpga fields of SeAddress as SE_ADDR_SLOT_ALL or or SE_ADDR_-FPGA_ALL respectively you tell the API to address *all* of these components (so `addr.fpga = SE_ADDR_FPGA_ALL` would address all FPGAs). This way you can create Multicast addresses (e.g. `addr.slot = SE_ADDR_SLOT_ALL`, `addr.fpga = 0` for the first FPGA on all cards), or true Broadcast addresses (`addr.slot = SE_ADDR_SLOT_ALL`, `addr.fpga = SE_ADDR_FPGA_ALL`).

1.3.2 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 received from the specified controller. Use SE_ADDR_CONTR_ALL as the controller index when waiting for data from any controller is desired. Once the method has returned, it provides the user with information of the write source, so the user can invoke se_read() with *passive* operation mode in order to read the incoming data.

2 Namespace Documentation

2.1 Package com

Packages

- package sciengines

2.2 Package com.sciengines

Packages

- package rivyera

2.3 Package com.sciengines.rivyera

Packages

- package api

2.4 Package com.sciengines.rivyera.api

Packages

- package types

Classes

- class SciEngines_API
- class SciEngines_API_Const

2.5 Package com.sciengines.rivyera.api.types

Packages

- package exceptions

Classes

- class SeAddress
- class SeControllerInfo
- class SeFPGAInfo
- enum SeFPGAType
- class SeOptions
- class SeProgInfo
- class SeSlotInfo

2.6 Package com.sciengines.rivyera.api.types.exceptions

Classes

- class SeApiException
- class SeApiFailedException
- class SeApiFileErrorException
- class SeApiInvalidAddressException
- class SeApiInvalidMachineException
- class SeApiLicenseErrorException
- class SeApiMachineInUseException
- class SeApiMachineNotAvailableException
- class SeApiReadTimeoutException
- class SeApiWriteTimeoutException

3 Class Documentation

3.1 SciEngines_API Class Reference

Static Public Member Functions

- static int se_getMachineCount ()
- static void se_allocMachine (int machine) throws SeApiException
- 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 SeProgInfo se_getProgInfo (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_comment (String str)
- static String se_type2str (SeFPGAType type)

3.1.1 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

3.1.2 Member Function Documentation

static int se_getMachineCount ( ) `[static]`

static void se_allocMachine (int *machine*) throws SeApiException `[static]`

static void se_allocMachine (int *machine,* SeOptions *options*) throws SeApiException `[static]`

static void se_freeMachine (int *machine*) throws SeApiException `[static]`

static long se_read (int *machine,* SeAddress *addr,* ByteBuffer *payload,* long *size,* int *mode,* long *timeout*) throws SeApiException `[static]`

static long se_write (int *machine,* SeAddress *addr,* ByteBuffer *payload,* long *size,* long *timeout*) throws SeApiException `[static]`

static void se_program (int *machine,* SeAddress *addr,* String *filename,* long *timeout*) throws SeApiException `[static]`

static void se_deprogram (int *machine,* SeAddress *addr*) throws SeApiException `[static]`

static SeAddress se_waitForData (int *machine,* int *controller,* long *timeout*) throws SeApiException `[static]`

static int se_getSlotCount (int *machine*) throws SeApiException `[static]`

static SeSlotInfo se_getSlotInfo (int *machine,* int *slot*) throws SeApiException `[static]`

static SeProgInfo se_getProgInfo (int *machine,* int *slot*) throws SeApiException `[static]`

static int se_getFPGACount (int *machine,* int *slot*) throws SeApiException `[static]`

static SeFPGAInfo se_getFPGAInfo (int *machine,* SeAddress *addr*) throws SeApiException `[static]`

static int se_getControllerCount (int *machine*) throws SeApiException `[static]`

static SeControllerInfo se_getControllerInfo (int *machine,* int *controller*) throws SeApiException `[static]`

static double se_getTemperature (int *machine,* int *slot*) throws SeApiException `[static]`

static double se_getMaxTemperature (int *machine,* int *slot*) throws SeApiException `[static]`

static void se_flush (int *machine,* int *controller,* long *timeout*) throws SeApiException `[static]`

static void se_comment (String *str*) `[static]`

- 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
- static int SE_READ_REQUEST = SciEngines_API_Const_JNI.SE_READ_REQUEST

3.2.1 Member Data Documentation

final int SE_API_VERSION_MAJOR = SciEngines_API_Const_JNI.SE_API_VERSION_MAJOR `[static]`

Major API version.

final int SE_API_VERSION_MINOR = SciEngines_API_Const_JNI.SE_API_VERSION_MINOR `[static]`

Minor API version.

final int SE_API_VERSION_SP = SciEngines_API_Const_JNI.SE_API_VERSION_SP `[static]`

API Service Pack.

final String SE_API_VERSION_REVISION = SciEngines_API_Const_JNI.SE_API_VERSION_REVISION `[static]`

API Revision.

final int SE_TIMEOUT_INFINITE = SciEngines_API_Const_JNI.SE_TIMEOUT_INFINITE `[static]`

Constant used whenever a method shall wait infinitely.

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.

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.

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.

int SE_ADDR_FPGA_HOST = SciEngines_API_Const_JNI.SE_ADDR_FPGA_HOST `[static]`

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

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.

int SE_LENGTH_ADDR_SLOT = SciEngines_API_Const_JNI.SE_LENGTH_ADDR_SLOT `[static]`

Length of the slot address field in bits.

int SE_LENGTH_ADDR_FPGA = SciEngines_API_Const_JNI.SE_LENGTH_ADDR_FPGA `[static]`

Length of the fpga address field in bits.

int SE_LENGTH_ADDR_REG = SciEngines_API_Const_JNI.SE_LENGTH_ADDR_REG `[static]`

Length of the register address field in bits.

int SE_LENGTH_CMD = SciEngines_API_Const_JNI.SE_LENGTH_CMD `[static]`

Length of the command field in bits.

int SE_READ_ACTIVE = SciEngines_API_Const_JNI.SE_READ_ACTIVE `[static]`

Constant used to invoke active read mode.

int SE_READ_PASSIVE = SciEngines_API_Const_JNI.SE_READ_PASSIVE `[static]`

Constant used to invoke passive read mode.

int SE_READ_REQUEST = SciEngines_API_Const_JNI.SE_READ_REQUEST `[static]`

Constant used to invoke a read request.

3.3 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

3.3.1 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_FPG-A_ALL on any of the components.

Author

Jost Bissel
Daniel Siebert

3.3.2 Constructor & Destructor Documentation

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

3.3.3 Member Function Documentation

String toString ( )

3.3.4 Member Data Documentation

int fpga = 0

The index of the target FPGA.

int reg = 0

The index of the target register.

int slot = 0

The index of the target slot.

int contr = 0

The index of the target controller.

3.4 SeApiException Class Reference

Inheritance diagram for SeApiException:

Public Member Functions

- SeApiException (int errorCode, String message)
- int getErrorCode ()

3.4.1 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

3.4.2 Constructor & Destructor Documentation

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.

3.4.3 Member Function Documentation

int getErrorCode ( )

Returns the error code of this exception.

Returns

Exception's error code.

3.5 SeApiFailedException Class Reference

Inheritance diagram for SeApiFailedException:

Collaboration diagram for SeApiFailedException:

Public Member Functions

- SeApiFailedException ()
- SeApiFailedException (String message)
- int getErrorCode ()

3.5.1 Constructor & Destructor Documentation

SeApiFailedException ( )

SeApiFailedException (String *message*)

3.5.2 Member Function Documentation

int getErrorCode ( ) `[inherited]`

Returns the error code of this exception.

Returns

Exception's error code.

3.6 SeApiFileErrorException Class Reference

Inheritance diagram for SeApiFileErrorException:

Collaboration diagram for SeApiFileErrorException:

Public Member Functions

- SeApiFileErrorException ()
- SeApiFileErrorException (String message)
- int getErrorCode ()

3.6.1 Constructor & Destructor Documentation

SeApiFileErrorException ( )

SeApiFileErrorException (String *message*)

3.6.2 Member Function Documentation

int getErrorCode ( ) `[inherited]`

Returns the error code of this exception.

Returns
    Exception's error code.

3.7 SeApiInvalidAddressException Class Reference

Inheritance diagram for SeApiInvalidAddressException:

Collaboration diagram for SeApiInvalidAddressException:

Public Member Functions

- SeApiInvalidAddressException ()
- SeApiInvalidAddressException (String message)
- int getErrorCode ()

3.7.1 Constructor & Destructor Documentation

SeApiInvalidAddressException ( )

SeApiInvalidAddressException (String *message*)

3.7.2 Member Function Documentation

int getErrorCode ( ) `[inherited]`

Returns the error code of this exception.

Returns

Exception's error code.

3.8 SeApiInvalidMachineException Class Reference

Inheritance diagram for SeApiInvalidMachineException:

Collaboration diagram for SeApiInvalidMachineException:

Public Member Functions

- SeApiInvalidMachineException ()
- SeApiInvalidMachineException (String message)
- int getErrorCode ()

3.8.1 Constructor & Destructor Documentation

SeApiInvalidMachineException ( )

SeApiInvalidMachineException (String *message*)

3.8.2 Member Function Documentation

int getErrorCode ( ) `[inherited]`

Returns the error code of this exception.

Returns

Exception's error code.

3.9 SeApiLicenseErrorException Class Reference

Inheritance diagram for SeApiLicenseErrorException:

Collaboration diagram for SeApiLicenseErrorException:

Public Member Functions

- SeApiLicenseErrorException ()
- SeApiLicenseErrorException (String message)
- int getErrorCode ()

3.9.1 Constructor & Destructor Documentation

SeApiLicenseErrorException ( )

SeApiLicenseErrorException (String *message*)

3.9.2 Member Function Documentation

int getErrorCode ( ) `[inherited]`

Returns the error code of this exception.

Returns

Exception's error code.

3.10 SeApiMachineInUseException Class Reference

Inheritance diagram for SeApiMachineInUseException:

Collaboration diagram for SeApiMachineInUseException:

Public Member Functions

- SeApiMachineInUseException ()
- SeApiMachineInUseException (String message)
- int getErrorCode ()

3.10.1 Constructor & Destructor Documentation

SeApiMachineInUseException ( )

SeApiMachineInUseException (String *message*)

3.10.2 Member Function Documentation

int getErrorCode ( ) `[inherited]`

Returns the error code of this exception.

Returns

Exception's error code.

3.11 SeApiMachineNotAvailableException Class Reference

Inheritance diagram for SeApiMachineNotAvailableException:

Collaboration diagram for SeApiMachineNotAvailableException:

Public Member Functions

- SeApiMachineNotAvailableException ()
- SeApiMachineNotAvailableException (String message)
- int getErrorCode ()

3.11.1 Constructor & Destructor Documentation

SeApiMachineNotAvailableException ( )

SeApiMachineNotAvailableException (String *message*)

3.11.2 Member Function Documentation

int getErrorCode ( ) `[inherited]`

Returns the error code of this exception.

Returns

Exception's error code.

3.12 SeApiReadTimeoutException Class Reference

Inheritance diagram for SeApiReadTimeoutException:

Collaboration diagram for SeApiReadTimeoutException:

Public Member Functions

- SeApiReadTimeoutException ()
- SeApiReadTimeoutException (String message)
- int getErrorCode ()

3.12.1 Constructor & Destructor Documentation

SeApiReadTimeoutException ( )

SeApiReadTimeoutException (String *message*)

3.12.2 Member Function Documentation

int getErrorCode ( ) `[inherited]`

Returns the error code of this exception.

Returns

Exception's error code.

3.13 SeApiWriteTimeoutException Class Reference

Inheritance diagram for SeApiWriteTimeoutException:

Collaboration diagram for SeApiWriteTimeoutException:

Public Member Functions

- SeApiWriteTimeoutException ()
- SeApiWriteTimeoutException (String message)
- int getErrorCode ()

3.13.1 Constructor & Destructor Documentation

SeApiWriteTimeoutException ( )

SeApiWriteTimeoutException (String *message*)

3.13.2 Member Function Documentation

int getErrorCode ( ) `[inherited]`

Returns the error code of this exception.

Returns

Exception's error code.

3.14 SeControllerInfo Class Reference

Public Member Functions

- String getDriverName ()
- int getMachineSlot ()
- int getSerial ()
- String toString ()

3.14.1 Detailed Description

A class containing useful information about a controller.

Author

Jost Bissel
Daniel Siebert

3.14.2 Member Function Documentation

String getDriverName ( )

Returns

The driver used to access this controller.

int getMachineSlot ( )

Returns

The machineSlot

int getSerial ( )

Returns

The serial

String toString ( )

3.15 SeFPGAInfo Class Reference

Public Member Functions

- SeFPGAType getType ()
- boolean isProgrammed ()
- int getFirmwareVersion ()
- int getFirmwareBuild ()
- String toString ()

3.15.1 Detailed Description

A class containing useful information about an FPGA.

Author

Jost Bissel
Daniel Siebert

3.15.2 Member Function Documentation

SeFPGAType getType ( )

Returns

The type

boolean isProgrammed ( )

Indicates whether this FPGA is programmed or not.

Returns

Whether this FPGA is programmed or not

int getFirmwareVersion ( )

Returns

The FPGA's firmware version.

int getFirmwareBuild ( )

Returns

The FPGA's firmware build.

String toString ( )

3.16 SeFPGAType Enum Reference

Public Member Functions

- String toString ()

Public Attributes

- fpga_none
- fpga_xc3s1000_4ft256
- fpga_xc3s1500_4fg676
- fpga_xc3s5000_4fg676
- fpga_xc6slx75_3fgg484
- fpga_xc6slx150_3fgg676
- fpga_xc4vsx35_10ff668
- fpga_10ax115h4f34e3sg

3.16.1 Detailed Description

Enum containing all chips supported by SciEngines API.

3.16.2 Member Function Documentation

String toString ( )

3.16.3 Member Data Documentation

fpga_none

fpga_xc3s1000_4ft256

fpga_xc3s1500_4fg676

fpga_xc3s5000_4fg676

fpga_xc6slx75_3fgg484

fpga_xc6slx150_3fgg676

fpga_xc4vsx35_10ff668

fpga_10ax115h4f34e3sg

3.17 SeOptions Class Reference

Classes

- enum SeRoutingMethod
- enum SeWriteBehavior

Public Member Functions

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

3.17.1 Constructor & Destructor Documentation

SeOptions (SeWriteBehavior *writeBehavior,* SeRoutingMethod *routingMethod*)

3.17.2 Member Function Documentation

SeWriteBehavior getWriteBehavior ( )

void setWriteBehavior (SeWriteBehavior *writeBehavior*)

SeRoutingMethod getRoutingMethod ( )

void setRoutingMethod (SeRoutingMethod *routingMethod*)

3.18 SeProgInfo Class Reference

Public Member Functions

- boolean isProgrammed ()
- boolean isLicPresent ()
- int getLicLifetime ()
- String toString ()

3.18.1 Detailed Description

A class containing the program information for a specific slot, saved during the last call to either se_program() or se_deprogram().

Author

Daniel Siebert

3.18.2 Member Function Documentation

boolean isProgrammed ( )

Returns

true if the last call to SciEngines_API#se_program(int, SeAddress, String, long) was successful or SciEngines_API#se_deprogram(int, SeAddress) has been called unsuccessfully, otherwise false.

boolean isLicPresent ( )

Returns

true if a license is present (no matter whether it has lapsed or not), otherwise false.

int getLicLifetime ( )

Returns

The license's remaining lifetime in minutes. This value is negative in case the license has lapsed. If the license's lifetime is infinite then the value is set to Integer#MAX_VALUE . If no license is present then the value is set to 0

String toString ( )

3.19 SeOptions.SeRoutingMethod Enum Reference

Public Attributes

- se_routing_normal

3.19.1 Member Data Documentation

se_routing_normal

3.20 SeSlotInfo Class Reference

Public Member Functions

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

3.20.1 Detailed Description

A class containing useful information about a slot.

Author

Jost Bissel

3.20.2 Member Function Documentation

boolean isController ( )

Returns

True, if controller else false.

int getControllerIndex ( )

Returns

The index of the controller, if isController() returns true

int getFpgaCount ( )

Returns

The fpgaCount

int getSerial ( )

Returns

The serial

int getPrevContr ( )

Returns

This card's previous controller index.

int getNextContr ( )

Returns

This card's next controller index.

int getFirmwareVersion ( )

Returns

The FPGA's firmware version.

int getFirmwareBuild ( )

Returns

The FPGA's firmware build.

String toString ( )

3.21 SeOptions.SeWriteBehavior Enum Reference

Public Attributes

- se_write_async
- se_write_sync

3.21.1 Member Data Documentation

se_write_async

se_write_sync

Imprint:

SciEngines GmbH
Am Kiel-Kanal 2
D-24106 Kiel Germany

Phone:	+49(0)431-9086-2000
Fax:	+49(0)431-9086-2009
E-Mail:	info@SciEngines.com
Internet:	www.SciEngines.com

CEO:	Gerd Pfeiffer
Commercial Register:	Amtsgericht Kiel
Commercial Register No.:	HR B 9565 KI
VAT-Identification Number:	DE 814955925