RIVYERA API

Host-API (C/C++)

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

Table of Contents

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*.

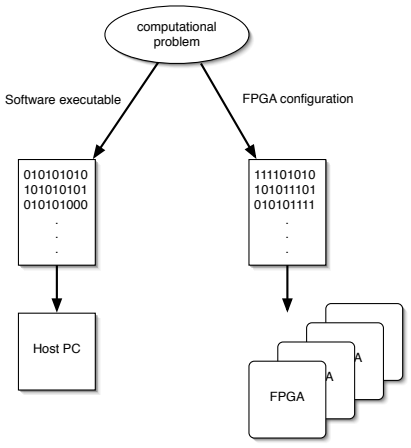

Figure 1: Partitioning of a problem into host- and machine-parts

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 PC[1]. 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.

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

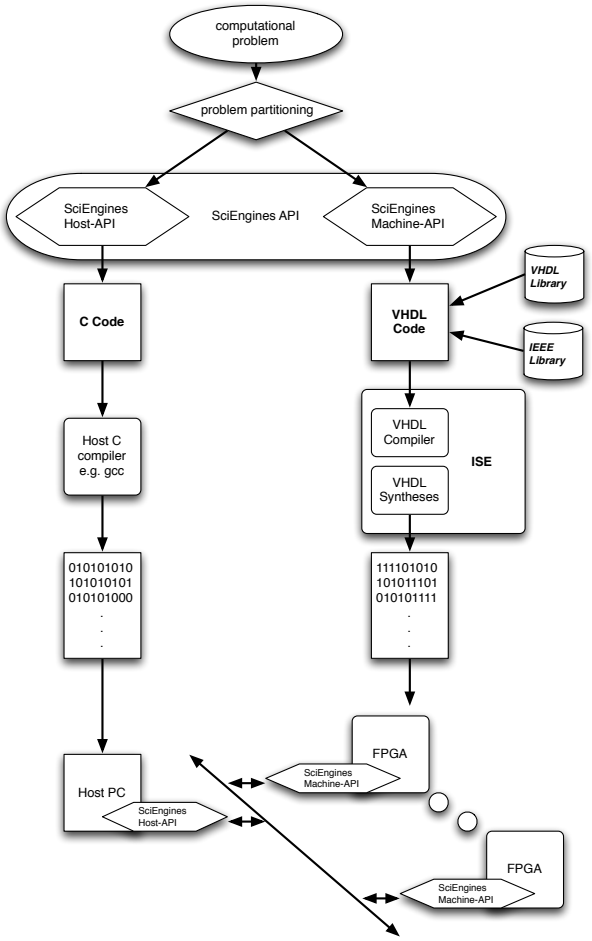

Figure 2: Design flow for multicomponent software systems

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 suppported.
- `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 it's 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 bandwith.

1.4.4 Target Adresses

A target address specifies where a given data word is to be delivered to and how the target shall interprete 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 Adresses

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

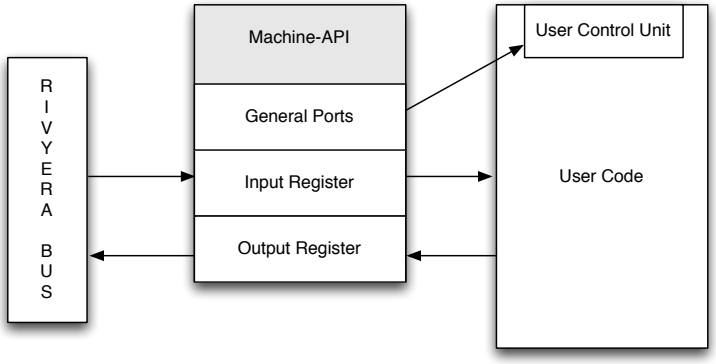

Figure 3: VHDL-API taking care of user design's I/O

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.

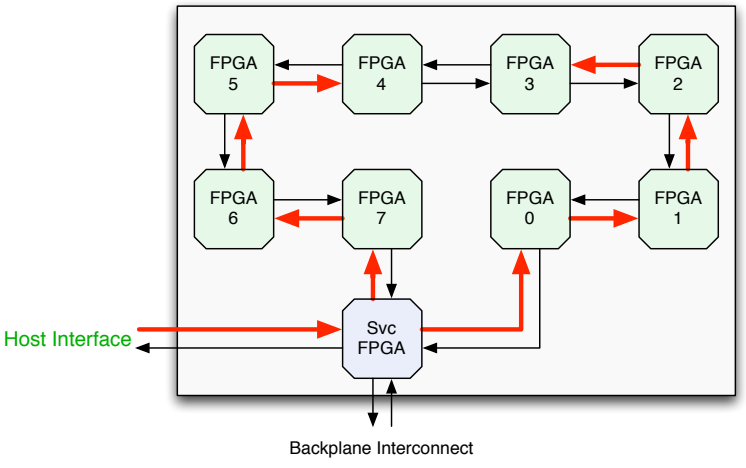

Figure 4: Routing of a host-initiated write

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 assurred by the API that no FPGA gets the same word twice.

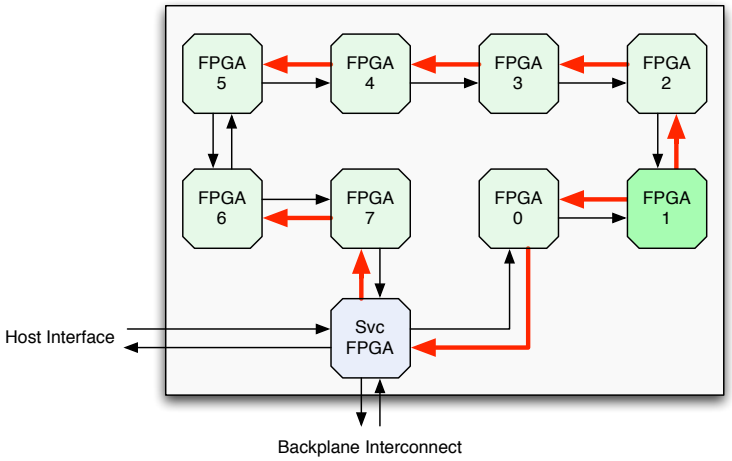

Figure 5: Routing of an fpga-initiated write

3 C/C++ API Introduction

The RIVYERA Host-API forms one endpoint of host-machine communication. It models the Input/Output register paradigm as introduced in section 2.1. Input registers of a FPGA can be filled using se_write , 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 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 an .rbt or .bit file to download it to the selected FPGAs.

The SciEngines RIVYERA API is completed with management functions such as se_getSlotCount() or se_getFPGAInfo() which make 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 structure SE_ADDR. 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 specifiy a component of SE_ADDR as SE_ADDR_SLOT_ALL or SE_ADDR_FPGA_ALL 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`).

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 may invoke se_read in order to read the incoming data.

Data Structure Documentation

4 SE_ADDR Struct Reference

Data Fields

- se_contr_t contr
- se_slot_t slot
- se_fpga_t fpga
- se_reg_t reg

4.1 Detailed Description

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

4.2 Field Documentation

4.2.1 se_contr_t contr

The index of the target slot.

4.2.2 se_slot_t slot

The index of the target slot.

4.2.3 se_fpga_t fpga

The index of the target FPGA.

4.2.4 se_reg_t reg

The index of the target register.

5 SE_CONTROLLERINFO Struct Reference

Data Fields

- const char * driver_name
- se_slot_t machineSlot
- unsigned int serial

5.1 Detailed Description

A structure containing useful information about a controller.

5.2 Field Documentation

5.2.1 const char* driver_name

The driver used to access this controller

5.2.2 se_slot_t machineSlot

The (machine-side) slot number of the controller.

5.2.3 unsigned int serial

The serial number of the controller.

6 SE_FPGAINFO Struct Reference

Data Fields

- SE_FPGA_TYPE type
- se_flag_t isProgrammed
- unsigned int firmwareVersion
- unsigned int firmwareBuild

6.1 Detailed Description

A structure containing useful information about an FPGA.

6.2 Field Documentation

6.2.1 SE_FPGA_TYPE type

Enum value of the FPGAs chip type

6.2.2 se_flag_t isProgrammed

Flag indicating if this FPGA is programmed and reacting.

6.2.3 unsigned int firmwareVersion

The FPGA's firmware version. bits 31..24: Major Version bits 23..16: Minor Version bits 15.. 8: Version revision bits 7.. 0: reserved (valid only, if programmed)

6.2.4 unsigned int firmwareBuild

The FPGA's firmware build. (valid only, if programmed)

7 SE_OPTIONS_T Struct Reference

Data Fields

- SE_WRITE_BEHAVIOR_T write_behavior
- SE_ROUTING_METHOD_T routing_method

7.1 Field Documentation

7.1.1 SE_WRITE_BEHAVIOR_T write_behavior

7.1.2 SE_ROUTING_METHOD_T routing_method

8 SE_SLOTINFO Struct Reference

Data Fields

- unsigned int serial
- se_fpga_t fpgaCount
- se_flag_t isContr
- se_contr_t contrIndex
- se_contr_t prevContr
- se_contr_t nextContr
- unsigned int firmwareVersion
- unsigned int firmwareBuild

8.1 Detailed Description

A structure containing useful information about a slot.

8.2 Field Documentation

8.2.1 unsigned int serial

The serial number of the slot's card.

8.2.2 se_fpga_t fpgaCount

Number of FPGAs on this slot's card.

8.2.3 se_flag_t isContr

Flag indicating, if this slot is connected to host

8.2.4 se_contr_t contrIndex

The index of the controller, if isContr is not 0

8.2.5 se_contr_t prevContr

Contains the previous controller index for given machine and slot index. If the machine has only one controller, the value will always be 0. If there is for a slot no previous controller, the own controller or -if given slot is not a controller- the next controller index is contained.

Example: Assume a machine having 16 slots with indices 0..15 where slot 1, slot 7 and slot 14 are directly connected to host via individual controllers 0..2. Then for slots 0..7 controller 0 (which is connected to slot 1) is contained, for slots 8..14 controller 1 (which is connected to slot 7) is contained and for remaining slot 15 controller 2 (connected to slot 14) is contained.

8.2.6 se_contr_t nextContr

8.2.7 unsigned int firmwareVersion

The card's firmware version. bits 31..24: Major Version bits 23..16: Minor Version bits 15.. 8: Version revision bits 7.. 0: Hardware revision

8.2.8 unsigned int firmwareBuild

The card's firmware build.

File Documentation

9 SeHostAPI.h File Reference

Functions

- se_machine_t se_getMachineCount ()

  Returns the number of machines connected to a host.

- SE_STATUS se_getSlotCount (se_machine_t machine, se_slot_t *pSlotCount)

  Returns the number of slots of a machine.

- SE_STATUS se_getFPGACount (se_machine_t machine, se_slot_t slot, se_fpga_t *pFPGACount)

  Returns the number of FPGAs for given slot and machine index.

- SE_STATUS se_getControllerCount (se_machine_t machine, se_contr_t *pCount)

  Returns the number of controllers.

- SE_STATUS se_allocMachine (se_machine_t machine, const SE_OPTIONS_T *pOptions)

  Allocates a machine.

- SE_STATUS se_freeMachine (se_machine_t machine)

  Deallocates a machine.

- SE_STATUS se_read (se_machine_t machine, const SE_ADDR *pAddr, __uint64_t *pPayload, size_t size, SE_READMODE mode, size_t *pCount, se_time_t timeout)

  Reads from given FPGA register to a local user defined buffer.

- SE_STATUS se_write (se_machine_t machine, const SE_ADDR *pAddr, const __uint64_t *pPayload, size_t size, size_t *pCount, se_time_t timeout)

  Writes given payload to the specified target input register.

- SE_STATUS se_flush (se_machine_t machine, se_contr_t controller, se_time_t timeout)
- SE_STATUS se_program (se_machine_t machine, const SE_ADDR *pAddr, const char *pFilename, se_time_t timeout)

  Downloads a given .rbt or .bit file to a machine element.

- SE_STATUS se_deprogram (se_machine_t machine, const SE_ADDR *pAddr)

  Unconfigures a given machine element.

- SE_STATUS se_program_slave (se_machine_t machine, const SE_ADDR *pAddr, const char *pFilename, se_time_t timeout)

  Downloads a given .rbt or .bit file to a machine element.

- SE_STATUS se_deprogram_slave (se_machine_t machine, const SE_ADDR *pAddr)

  Unconfigures a given machine element.

- SE_STATUS se_waitForData (se_machine_t machine, se_contr_t controller, SE_ADDR *pAddr, size_t *pCount, se_time_t timeout)

  Waits for incoming data at a given controller.

- SE_STATUS se_getTemperature (se_machine_t machine, se_slot_t slot, double *pCurrentTemp, double *pMaxTemp)

  Returns the current and maximum temperature measured at an FPGA.

- SE_STATUS se_getSlotInfo (se_machine_t machine, se_slot_t slot, SE_SLOTINFO *pInfo)

  Reports slot information.

- SE_STATUS se_getFPGAInfo (se_machine_t machine, const SE_ADDR *pAddr, SE_FPGAINFO *pInfo)

  Reports FPGA information.

- SE_STATUS se_getControllerInfo (se_machine_t machine, se_contr_t controller, SE_CONTROLLERINFO *pInfo)

  Reports controller specific information.

- void se_comment (const char *fmt,...)

  For debugging purposes this function enables to write a comment to a macro file, when macro file writing is enabled. This comment is also written to log with detailed level.

9.1 Function Documentation

9.1.1 se_machine_t se_getMachineCount ()

This function returns the number of machines that are connected to a host PC. In most RIVYERA setups the typical number of machines connected to the host is one due to the integrated PC.

Returns:

The number of possibly accessible machines.

```
/*--- BEGIN EXAMPLE ---*/
se_machine_t machines;

machines = se_getMachineCount();

printf("Found %d RIVYERA machines.", machines);
/*--- END EXAMPLE ---*/
```

9.1.2 SE_STATUS se_getSlotCount (se_machine_t *machine*, se_slot_t * *pSlotCount*)

This function returns the slot count of a selected machine.

Parameters:

machine Machine to get slot count from.

pSlotCount Pointer to an integer to store the result in.

Return values:

SeApiSuccess Returned if the function succeeded.

SeApiMachineNotAvailable Returned if the machine was not allocated before.

```
/*--- BEGIN EXAMPLE ---*/
SE_STATUS rc;             /* API return code */
se_slot_t slotCount ;     /* Number of slots of a machine */
se_machine_t machine = 0; /* Machine to get slot count of */

/* Get number of slots of given machine */
rc = se_getSlotCount(machine, &slotCount);

if (rc == SeApiSuccess) {
   /* Everything went fine */
   printf("Machine %d has %d slots.", machine+1, slotCount);
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.3 SE_STATUS se_getFPGACount (se_machine_t *machine*, se_slot_t *slot*, se_fpga_t * *pFPGACount*)

This function returns the FPGA count of a selected machine and slot index.

Parameters:

machine Machine to get FPGA count from.

slot Slot whose FPGA count is requested

pFPGACount Pointer to an integer to store the result in.

Return values:

SeApiSuccess Returned if the function succeeded.

SeApiMachineNotAvailable Returned if the machine was not allocated before.

SeApiInvalidAddress Returned if the given slot index is invalid.

```
/*--- BEGIN EXAMPLE ---*/
SE_STATUS    rc;             /* API return code */
se_fpga_t    fpgaCount ;     /* Number of FPGAs of a slot */
se_machine_t machine = 0;    /* Machine to get FPGA count of */
se_slot_t    slot = 3;       /* Slot to get the FPGA count of */

/* Get number of FPGAs of given slot and machine */
rc = se_getFPGACount(machine, slot, &fpgaCount);

if (rc == SeApiSuccess) {
   /* Everything went fine */
   printf("Card %d in machine %d has %d slots.", slot + 1, machine+1, fpgaCount);

} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.4 SE_STATUS se_getControllerCount (se_machine_t *machine*, se_contr_t * *pCount*)

This function returns the controller count of a machine. The number of controllers depends on the machine configuration. Each machine has at least one controller.

Parameters:

machine Machine index to allocate for usage.

pCount Pointer to an unsigned integer, where result should be stored in.

Return values:

SeApiSuccess Returned if the function succeeded.

SeApiMachineNotAvailable Returned if the machine was not allocated before.

```
/*--- BEGIN EXAMPLE ---*/
SE_STATUS rc;              /* API return code */
se_contr_t contrCount;     /* Number of controllers belonging to a machine */
se_machine_t machine = 0;  /* Machine index to get controller count of */

/* Get number of controllers of given machine */
rc = se_getControllerCount(machine, &contrCount);

if (rc == SeApiSuccess) {
```

```
   /* Everything went fine */
   printf("Machine %d has %d controllers.", machine+1, contrCount);
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.5 SE_STATUS se_allocMachine (se_machine_t *machine*, const SE_OPTIONS_T * *pOptions*)

This function allows the user to allocate a machine before usage. It is *NEEDED* to be called before a machine is usable. Otherwise any API call (except se_getMachineCount) will fail with SeApiMachineNotAvailable. The invocation of se_allocMachine() will set up all the variables needed for usage and lock the machine from being used by other processes. This function returns SeApiInvalidMachine if the given machine index is invalid (e.g. if only one machine is present and you want to allocate machine 3) or SeApiMachineInUse if the desired machine is already in use.

When a machine gets allocated, information like the number of slots is requested from each connected controller belonging to the machine. If a timeout occurs while requesting these information either SeApiWriteTimeout or SeApiReadTimeout is returned. This usually happens if a controller is flooded by a previous allocation. Using the shell command "se_machine restart" resets all controllers so that the machine is able to be allocated again.

Parameters:

machine Machine index to allocate for usage, starting at index 0 for first machine.

pOptions Pointer to a structure defining some options that remain active permanently until se_freeMachine is called.

Return values:

SeApiSuccess Returned if the machine was successfully allocated.

SeApiInvalidMachine Returned if the given machine does not exist.

SeApiMachineInUse Returned if the given machine was allocated by a different process.

SeApiWriteTimeout Returned, if a timeout occurred writing read requests to the machine.

SeApiReadTimeout Returned, if a timeout occurred reading information from the machine.

```
/*--- BEGIN EXAMPLE ---*/
SE_STATUS   rc; /* Return code */

if (se_getMachineCount() <= 0){
```

```
   /* calling se_allocMachine() would return SeApiMachineNotAvailable */
   printf("No machine available.");
   return;
}

/* allocate first machine using default options (set to NULL) */
rc = se_allocMachine(0, NULL);

if (rc == SeApiSuccess) {
   /* Everything went fine */
   printf("Successfully allocated machine for usage.\n");
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.6 SE_STATUS se_freeMachine (se_machine_t *machine*)

This function allows the user to release the usage lock of a machine. After invoking se_freeMachine() the given machine is free for usage by any other process. This function returns SeApiMachineNotAvailable if the desired machine is not allocated.

Parameters:

machine Machine index that is to be released.

Return values:

SeApiSuccess Returned if the machine was successfully released.

SeApiMachineNotAvailable Returned if the machine was not allocated before.

SeApiWriteTimeout Returned, if a timeout occurred writing reset command to the machine.

```
/*--- BEGIN EXAMPLE ---*/
SE_STATUS rc; /* Return code */

rc = se_freeMachine(0); /* free first machine */

if (rc == SeApiSuccess) {
   /* Everything went fine */
   printf("Successfully deallocated previously allocated machine.\n");
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.7 SE_STATUS se_read (se_machine_t *machine*, const SE_ADDR * *pAddr*, __uint64_t * *pPayload*, size_t *size*, SE_READMODE *mode*, size_t * *pCount*, se_time_t *timeout*)

This function reads a stream of 64 bit words into a user defined buffer. There are different read modes available. The most commonly used read mode is the passive read (SeReadPassive) which reads data that has been written to the host without a special priorly written read request. When using the active read (SeReadActive) a special read request is sent to the target advising it to write size number of words to the request's origin (in this case the host). From the FPGA's point of view, a read request uses the same mechanism as a normal received word except that the command (api_i_cmd_in) is set to CMD_RD rather than CMD_WR. The number of requested words is then taken from api_i_data. After sending the read request, se_read behaves like in passive read mode and waits for the data being written. When using se_read with mode SeReadRequest then only the request is sent to specified target without reading any data. In this mode it is safe to set a NULL pointer for pPayload and in contrast to the other modes, setting a broadcast address for slot (SE_ADDR_SLOT_ALL) and/or for fpga (SE_ADDR_FPGA_ALL) is allowed. The requested data may be read later using a regular passive read.

Warning:

If mode is not set to SeReadRequest, you may not use SE_ADDR_SLOT_ALL or SE_ADDR_FPGA_ALL in the given address. Doing so will result in returning SeApiInvalidAddress.

Parameters:

machine Machine to read from.

pAddr Address specifying read target element (*NOTE:* Using SE_ADDR_SLOT_ALL for slot or SE_ADDR_FPGA_ALL for fpga within address will result in returning SeApiInvalidAddress if mode is not set to SeReadRequest).

pPayload Pointer to user buffer.

size Number of 64 Bit words to read from the machine.

mode Read mode to use for the read process. The most common mode is SeReadPassive, which reads data from FPGA without a prior read request. Using mode SeReadRequest does not read at all but writes a special read request to the target FPGA. For this mode, broadcasting the request by setting

pCount Pointer to __uint64_t to store the number of words, that actually was transferred (may be NULL)

timeout Time in ms that execution may take at most. Use SE_TIMEOUT_INFINITE to deactivate the function's timeout.

Return values:

SeApiSuccess Returned if the transfer completed successfully.

SeApiMachineNotAvailable Returned if the given machine was not allocated before.

SeApiInvalidAddress Returned if the given address is invalid.

SeApiReadTimeout Returned if a timeout occurred before the transfer was completed.

SeApiWriteTimeout Returned if a timeout occurred during a read request. This may occur for mode equal to SeReadRequest or SeReadActive, only.

```
/*--- BEGIN EXAMPLE ---*/
__uint64_t  *pBuf;            /* The transmission buffer */
size_t      transfer = 52;    /* The number of 64-Bit words that we
                                 want to read */
size_t      transferCount;    /* The number of 64-Bit words actually
                                 transferred*/
SE_STATUS   rc;               /* Return code */
SE_ADDR     addr = {0,        /* use first controller */
                    3,        /* set fourth slot as source */
                    0,        /* set first FPGA as source */
                    1};       /* set register 1 as source */

/* Allocate transfer buffer */
pBuf = (__uint64_t *)malloc(transfer * sizeof(__uint64_t));

/* Read actively from first machine, wait infinitely for completion */
rc = se_read(0, &addr, pBuf, transfer, SeReadActive, &transferCount,
      SE_TIMEOUT_INFINITE);

if (rc == SeApiSuccess) {
   /* Everything went fine */
   printf("Successfully read %ld words.\n", transferCount);
} else if (rc == SeApiWriteTimeout) {
   /* Unable to write the read request (we did an active read!) */
   printf("Timeout occurred writing the read request.\n");
} else if (rc == SeApiReadTimeout) {
   /* Unable to send all data */
   printf("Timeout occurred reading data. Only got %ld of %ld words.\n",
         transferCount, transfer);
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}

free(pBuf);
/*--- END EXAMPLE ---*/
```

9.1.8 SE_STATUS se_write (se_machine_t *machine*, const SE_ADDR * *pAddr*, const __uint64_t * *pPayload*, size_t *size*, size_t * *pCount*, se_time_t *timeout*)

This function enables the user to write a given payload to the specified target element. This function blocks until either the transfer completed successfully or until `timeout` milliseconds have passed without returning.

Parameters:

machine Machine to write to.

pAddr Pointer to SE_ADDR that addresses one or all FPGAs at one or all slots. For broadcast addressing SE_ADDR_SLOT_ALL and/or SE_ADDR_FPGA_ALL may be used for slot field and/or fpga field.

pPayload Pointer to user buffer.

size Number of 64 Bit words to write to machine.

pCount Pointer to size_t to store the number of words, that actually was transferred (may be NULL)

timeout Amount of time (in milliseconds) that execution may take at most. Use SE_TIMEOUT_INFINITE to deactivate the function's timeout.

Return values:

SeApiSuccess Returned if the transfer completed successfully.

SeApiInvalidMachine Returned if the given machine index is not valid.

SeApiMachineNotAvailable Returned if the given machine was not allocated before.

SeApiInvalidAddress Returned if the given address is invalid.

SeApiWriteTimeout Returned if the timeout exceeded before the transfer was completed.

```
/*--- BEGIN EXAMPLE ---*/
SE_STATUS  rc;                        /* Return code */
__uint64_t *pBuf;                     /* The transmission buffer */
size_t     transfer = 30;             /* The number of 64-Bit words that we
                                         want to read */
size_t     transferCount;             /* The number of 64-Bit words actually
                                         transferred*/
SE_ADDR    addr = {0,                 /* use first controller */
                   6,                 /* set seventh slot as target */
                   SE_ADDR_FPGA_ALL,  /* set all FPGAs as target */
                   0};                /* set register 0 as target */

/* Allocate transfer buffer */
pBuf = (__uint64_t *)malloc(transfer * sizeof(__uint64_t));

/* Write to first machine, wait infinitely for completion */
rc = se_write(0, &addr, pBuf, transfer, &transferCount, SE_TIMEOUT_INFINITE);

if (rc == SeApiSuccess) {
   /* Everything went fine */
   printf("Successfully wrote %ld words.\n", transferCount);
} else if (rc == SeApiWriteTimeout) {
   /* Unable to write all data */
   printf("Timeout occurred writing data. Only %ld of %ld words transferred.\n",
          transferCount, transfer);
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
```

```
}

free(pBuf);
/*--- END EXAMPLE ---*/
```

9.1.9 SE_STATUS se_flush (se_machine_t *machine*, se_contr_t *controller*, se_time_t *timeout*)

Flushes the write cache for given machine and controller. This function blocks until either the write cache is completely written to given controller or if this function has run `timeout` milliseconds without returning.

Parameters:

machine Machine whose cache should be flushed

controller Machine's controller whose cache should be flushed

timeout integer specifying the time in ms which this function may block at most.

Return values:

SeApiSuccess Returned if the flush completed successfully.

SeApiInvalidMachine Returned if the given machine index is not valid.

SeApiMachineNotAvailable Returned if the given machine was not allocated before.

SeApiInvalidAddress Returned if the given controller index is invalid.

SeApiWaitTimeout Returned if the timeout exceeded before the flush was completed.

```
/*--- BEGIN EXAMPLE ---*/
SE_STATUS    rc;                        /* Return code */
se_machine_t machine = 0;               /* use first machine to work on*/
__uint64_t   word = 0x1234abcd;         /* write word 0x1234abcd */
SE_ADDR      addr = {0,                 /* set the controller later... */
                     6,                 /* set seventh slot as target */
                     SE_ADDR_FPGA_ALL,  /* set all FPGAs as target */
                     0};                /* set register 0 as target */

/* write a word asynchronously */
rc = se_write(machine, &addr, &word, 1, NULL, 0);
if (rc != SeApiSuccess) {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}

/* wait at most one second for all enqueued
 * write jobs to be written to the controller */
rc = se_flush(machine, addr.contr, 1000);
if (rc == SeApiSuccess) {
   /* Everything went fine */
```

```
   printf("Successfully flushed all data to controller %d!\n", addr.contr);
} else {
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.10 SE_STATUS se_program (se_machine_t *machine*, const SE_ADDR * *pAddr*, const char * *pFilename*, se_time_t *timeout*)

This function parses a bit file (.bit), an ASCII encoded bit file or a simulation file (for a simulated machine, please refer to Simulation API documentation) and writes the bitstream to one or more addressed FPGAs.

Since they are slightly faster to parse it is recommended to use .bit files rather than .rbt files. Nevertheless, the parsed bitstream for both filetypes is identical when created from same netlist. For creating .bit files using ISE, the "Create Bit File" option has to be activated in "Generate Programming File" properties (which is default). For creating .rbt files, the "Create ASCII configuration file" otions has to be activated.

To program multiple FPGAs at the same time it is possible to use SE_ADDR_SLOT_ALL and/or SE_ADDR_FPGA_ALL for the address struct's slot and fpga fields.

This function blocks until either the programming completed successfully or until `timeout` milliseconds have passed without returning.

Warning:

Since the FPGAs on a card form a ring, all FPGAs need to be programmed. If there are one or more FPGAs not being programmed, then the physical communication ring is not closed and se_program will return SeApiFailed. Nevertheless it is possible to program individual FPGAs on a card. In that case it is necessary to firstly program the whole card using SE_ADD_FPGA_ALL for the address struct's fpga field to enable communication and afterwards program individual FPGAs with a different FPGA design.

Parameters:

machine Machine index to write to.

pAddr Pointer to SE_ADDR that addresses one or all FPGAs at one or all slots. For broadcast addressing SE_ADDR_SLOT_ALL and/or SE_ADDR_FPGA_ALL may be used for slot field and/or fpga field.

pFilename String holding a path to an .rbt, .bit or .sim file.

timeout Amount of time (in milliseconds) that execution may take at most. Use SE_TIMEOUT_INFINITE to disable the function's timeout.

Return values:

SeApiSuccess Returned if the programming completed successfully.

SeApiMachineNotAvailable Returned if the given machine was not allocated before.

SeApiInvalidAddress Returned if the given address is invalid.

SeApiWaitTimeout Returned if the timeout exceeded before the transfer was completed.

SeApiFileError Returned if the given file either does not exist or the user has no read permissions.

```
/*--- BEGIN EXAMPLE ---*/
SE_STATUS   rc;                     /* Return code */
SE_ADDR     addr = {0,              /* use first controller */
                    1,              /* set second slot as target */
                    SE_ADDR_FPGA_ALL, /* set all FPGAs as target */
                    0};             /* register is ignored */

char *pFilename = "example.bit";          /* Pointer to filename */

/* Program first machine, wait infinitely for completion */
rc = se_program(0, &addr, pFilename, SE_TIMEOUT_INFINITE);

if (rc == SeApiSuccess) {
   /* Everything went fine */
   printf("Successfully programmed FPGAs.\n");
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.11 SE_STATUS se_deprogram (se_machine_t *machine*, const SE_ADDR * *pAddr*)

This function enables the user to delete any previously downloaded configuration. After invoking this function, the target will lose all programmed logic. This function is useful if you have non-exclusive access to a machine and don't want other users to use your code.

Parameters:

machine Machine index to deprogram.

pAddr Pointer to SE_ADDR that addresses one or all FPGAs at one or all slots. For broadcast addressing SE_ADDR_SLOT_ALL and/or SE_ADDR_FPGA_ALL may be used for slot field and/or fpga field.

Return values:

SeApiSuccess Returned if the deprogramming completed successfully.

SeApiMachineNotAvailable Returned if the given machine was not allocated before.

SeApiInvalidAddress Returned if the given address is invalid.

```
/*--- BEGIN EXAMPLE ---*/
SE_STATUS   rc;                      /* Return code */
SE_ADDR     addr = {0,               /* use first controller */
                    SE_ADDR_SLOT_ALL, /* set all slots as target */
                    SE_ADDR_FPGA_ALL, /* set all FPGAs as target */
                    0};              /* register is ignored */

/* Deprogram first machine */
rc = se_deprogram(0, &addr);

if (rc == SeApiSuccess) {
   /* Everything went fine */
   printf("Successfully deprogrammed FPGAs.\n");
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.12 SE_STATUS se_program_slave (se_machine_t *machine*, const SE_ADDR * *pAddr*, const char * *pFilename*, se_time_t *timeout*)

This function parses a bit file (.bit) or an ASCII encoded bit file.

To program multiple FPGAs at the same time it is possible to use SE_ADDR_SLOT_ALL and/or SE_ADDR_FPGA_ALL for the address struct's slot and fpga fields.

This function blocks until either the programming completed successfully or until `timeout` milliseconds have passed without returning.

Information: Slave FPGAs are are a special setup for some Spartan 6 machines (those having 16 FPGAs per card which are connected via a so called expansion module).

Attention: After programming the slave FPGA, the master FPGA has to be programmed again using se_program since it is partially programmed with an internal bitfile that enables the slave programming. Like with se_program it is necessary to programm all FPGAs for a slot at once. Please also read the description to se_program before using se_program_slave!

Parameters:

machine Machine index to write to.

pAddr Pointer to SE_ADDR that addresses one or all FPGAs at one or all slots. For broadcast addressing SE_ADDR_SLOT_ALL and/or SE_ADDR_FPGA_ALL may be used for slot field and/or fpga field.

pFilename String holding a path to an .rbt or .bit.

timeout Amount of time (in milliseconds) that execution may take at most. Use SE_TIMEOUT_INFINITE to disable the function's timeout.

Return values:

SeApiSuccess Returned if the programming completed successfully.

SeApiMachineNotAvailable Returned if the given machine was not allocated before.

SeApiInvalidAddress Returned if the given address is invalid.

SeApiWaitTimeout Returned if the timeout exceeded before the transfer was completed.

SeApiFileError Returned if the given file either does not exist or the user has no read permissions.

9.1.13 SE_STATUS se_deprogram_slave (se_machine_t *machine*, const SE_ADDR * *pAddr*)

This function enables the user to delete any previously downloaded configuration for slave FPGAs. After invoking this function, the target will lose all programmed logic. This function is useful if you have non-exclusive access to a machine and don't want other users to use your code.

Parameters:

machine Machine index to deprogram.

pAddr Pointer to SE_ADDR that addresses one or all FPGAs at one or all slots. For broadcast addressing SE_ADDR_SLOT_ALL and/or SE_ADDR_FPGA_ALL may be used for slot field and/or fpga field.

Return values:

SeApiSuccess Returned if the deprogramming completed successfully.

SeApiMachineNotAvailable Returned if the given machine was not allocated before.

SeApiInvalidAddress Returned if the given address is invalid.

```
/*--- BEGIN EXAMPLE ---*/
SE_STATUS   rc;                      /* Return code */
SE_ADDR     addr = {0,               /* use first controller */
                    SE_ADDR_SLOT_ALL, /* set all slots as target */
                    SE_ADDR_FPGA_ALL, /* set all FPGAs as target */
                    0};              /* register is ignored */

/* Deprogram first machine */
rc = se_deprogram(0, &addr);

if (rc == SeApiSuccess) {
   /* Everything went fine */
   printf("Successfully deprogrammed FPGAs.\n");
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.14 SE_STATUS se_waitForData (se_machine_t *machine*, se_contr_t *controller*, SE_ADDR * *pAddr*, size_t * *pCount*, se_time_t *timeout*)

This function waits for incoming data at a given controller. After returning, `pAddr` will contain the address from which data can be read. This function is very helpful whenever your VHDL design initiates transfer by itself. Whenever an FPGA initiates a transfer to host (by specifying slot 0 as its target), this particular controller enables se_waitForData() to return.

If controller is set to SE_ADDR_CONTR_ALL this function will wait at all controllers and returns as soon as for one controller is data present.

Parameters:

machine Machine index to wait for data at.

controller Index of the controller to wait for data at. If controller is set to SE_ADDR_CONTR_ALL se_waitForData will wait at all controllers.

pAddr Pointer to SE_ADDR to store the data source in.

pCount Pointer to size_t to store the number of words that are ready to be read at address pAddr. May be NULL if the number of words is not needed.

timeout Amount of time (in ms) that execution may take at most. Use SE_TIMEOUT_INFINITE to disable the function's timeout.

Return values:

SeApiSuccess Returned if data occurred within time limits.

SeApiMachineNotAvailable Returned if the given machine was not allocated before.

SeApiInvalidAddress Returned if the given controller index is invalid.

SeApiWaitTimeout Returned if the timeout exceeded before data occurred.

```
/*--- BEGIN EXAMPLE ---*/
int         machine, controller; /* Machine and controller indices */
SE_ADDR     addr;                /* Address to store the data source in */
size_t      dataCount;           /* Size of the incoming data */
SE_STATUS   rc;                  /* Return code */

machine     = 0;                 /* Listen for data on first machine */
controller  = SE_ADDR_CONTR_ALL; /* Listen for data on all controllers */

/* Listen for incoming data at first machine, all controllers for 5 seconds */
rc = se_waitForData(machine, controller, &addr, &dataCount, 5 * 1000);

if (rc == SeApiSuccess) {
   /* Everything went fine */
   printf("%ld words read to be read passively from controller %d, slot %d, FPGA
      %d, reg %d.\n",
         dataCount, addr.contr, addr.slot, addr.fpga, addr.reg);
} else if (rc == SeApiReadTimeout) {
```

```
   /* No data available to be read. */
   printf("There is nothing to be read.\n");
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.15 SE_STATUS se_getTemperature (se_machine_t *machine*, se_slot_t *slot*, double * *pCurrentTemp*, double * *pMaxTemp*)

Reads the temperature sensor for a given machine and slot index. The temperature in Celsius is stored to pTemp which is a double.

Parameters:

machine Machine index to get the temperature from.

slot Slot index to get the temperature from.

pCurrentTemp Pointer to a double value that will contain the current temperature after the function returned successfully.

pMaxTemp Pointer to a double value that will contain the maximum temperature after the function returned successfully.

Return values:

SeApiSuccess Returned if temperature was read successfully.

SeApiFailed Returned if temperature was not read successfully.

9.1.16 SE_STATUS se_getSlotInfo (se_machine_t *machine*, se_slot_t *slot*, SE_SLOTINFO * *pInfo*)

This function allows the user to get information about the slot's equipment. After invocation, `pInfo` will contain the desired information.

Parameters:

machine Machine index to get information from.

slot Slot index to retrieve information about.

pInfo Pointer to SE_SLOTINFO to store result in.

Return values:

SeApiSuccess Returned if the information was retrieved successfully.

SeApiMachineNotAvailable Returned if the given machine was not allocated before (see se_allocMachine).

SeApiInvalidAddress Returned if the given slot index does not exist.

```
/*--- BEGIN EXAMPLE ---*/
int         machine = 0; /* Machine to get slot info of */
int         slot    = 1; /* Slot to retrieve information about */
SE_STATUS   rc;          /* Return code */
SE_SLOTINFO info;        /* Slot information structure */

/* Get number of slots of given machine */
rc = se_getSlotInfo(machine, slot, &info);

printf("Slot %d information:\n", (slot+1));

if (rc == SeApiSuccess) {
   if (info.equipment == SeController)
      printf("-- Equipment  : Controller\n");
   else
      printf("-- Equipment  : Card\n");
   printf("-- Serial     : %X\n", info.serial);
   printf("-- FPGA Count : %d\n", info.fpgaCount);
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.17 SE_STATUS se_getFPGAInfo (se_machine_t *machine*, const SE_ADDR * *pAddr*, SE_FPGAINFO * *pInfo*)

This function returns information about a specified FPGA chip type at a selected address. It offers the user a brief overview about FPGA type and features like attached DRAM. After invocation, `pInfo` will contain the desired information.

Parameters:

machine Machine index to get information from.

pAddr Pointer to SE_ADDR that addresses an FPGA.

pInfo Pointer to SE_FPGAINFO to store result in.

Return values:

SeApiSuccess Returned if the information was retrieved successfully.

SeApiInvalidMachine Returned if the given machine index is not valid.

SeApiMachineNotAvailable Returned if the given machine was not allocated before.

SeApiInvalidAddress Returned if the given address is invalid.

```
/*--- BEGIN EXAMPLE ---*/
SE_STATUS    rc;            /* Return code */
SE_FPGAINFO  info;          /* FPGA information structure */
se_machine_t machine = 0;   /* Machine to get slot info of */
SE_ADDR      addr = {1,     /* use second controller */
                     3,     /* set fourth slot as target */
```

```
                    4,     /* set fifth FPGA as target */
                    0};    /* register is ignored */

/* Get fpga infos */
rc = se_getFPGAInfo(machine, &addr, &info);

if (rc == SeApiSuccess) {
   /* Everything went fine */
   printf("FPGA %d on slot %d information:\n", (addr.fpga+1), (addr.slot+1));
   printf("-- Slot            : %d\n", info.slot);
   printf("-- Index           : %d\n", info.index);
   printf("-- Type            : %s\n", se_type2str(info.type));
   printf("-- Programmed      : %s\n", info.programmed ? "true" : "false");
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.18 SE_STATUS se_getControllerInfo (se_machine_t *machine*, se_contr_t *controller*, SE_CONTROLLERINFO * *pInfo*)

This function returns information about a specified controller. This information typically consists of a serial number, host-side interface information (Vendor ID, etc.) and machine-side information (slot number, etc.). After invocation, `pInfo` will contain the desired information.

Parameters:

machine Machine index to get information from.

controller Controller index to retrieve information about.

pInfo Pointer to SE_CONTROLLERINFO to store the result in.

Return values:

SeApiSuccess Returned if the information was retrieved successfully.

SeApiMachineNotAvailable Returned if the given machine was not allocated before.

SeApiInvalidAddress Returned if the given controller index is invalid.

```
/*--- BEGIN EXAMPLE ---*/
se_machine_t      machine; /* Machine to get slot info of */
se_contr_t        ctrl;    /* Slot to retrieve information about */
SE_STATUS         rc;      /* Return code */
SE_CONTROLLERINFO info;    /* Controller information structure */

machine    = 0;            /* Get slot info of first machine */
ctrl       = 0;            /* Get information about first controller */

/* Get number of slots of given machine */
rc = se_getControllerInfo(machine, ctrl, &info);

if (rc == SeApiSuccess) {
```

```
   /* Everything went fine */
   printf("Controller %d information:\n", (ctrl+1));
   printf("-- Driver name  : 0%s\n",  info.driver_name);
   printf("-- Machine Slot : %d\n",   info.machineSlot);
   printf("-- Serial       : %#X\n",  info.serial);
} else {
   /* An error occurred. Print out the error code
    * as number and as human readable string */
   printf("ERROR! SciEngines API returned %d (= %s)!\n", rc, se_status2str(rc));
}
/*--- END EXAMPLE ---*/
```

9.1.19 void se_comment (const char * *fmt*, ...)

Parameters:

fmt string defining the format printing optional arguments

10 SeHostAPITypes.h File Reference

Data Structures

- struct SE_OPTIONS_T
- struct SE_ADDR
- struct SE_CONTROLLERINFO
- struct SE_SLOTINFO
- struct SE_FPGAINFO

Typedefs

- typedef unsigned int se_slot_t
- typedef unsigned int se_fpga_t
- typedef unsigned int se_reg_t
- typedef unsigned int se_cmd_t
- typedef unsigned int se_contr_t
- typedef unsigned int se_machine_t
- typedef unsigned char se_flag_t
- typedef unsigned long int se_time_t

Enumerations

- enum SE_STATUS {
  SeApiSuccess, SeApiFailed, SeApiInvalidMachine, SeApiMachineNotAvailable,
  SeApiMachineInUse, SeApiInvalidAddress, SeApiWriteTimeout, SeApiReadTimeout,
  SeApiFileError }
- enum SE_READMODE { SeReadActive, SeReadPassive, SeReadRequest }
- enum SE_FPGA_TYPE {
  none, xc3s1000_4ft256, xc3s1500_4fg676, xc3s5000_4fg676,
  xc6slx75_3fg484, xc6slx150_3fg676, xc4vsx35_10ff668 }
- enum SE_ROUTING_METHOD_T { se_routing_normal, se_routing_systolic }
- enum SE_WRITE_BEHAVIOR_T { se_write_async, se_write_sync }

Functions

- const char * se_status2str (SE_STATUS status)

  Translates a status code to a readable string.

- const char * se_type2str (SE_FPGA_TYPE type)

  Translates an FPGA type to a readable string.

Variables

- const unsigned short SE_API_VERSION_MAJOR
- const unsigned short SE_API_VERSION_MINOR
- const unsigned short SE_API_VERSION_SP
- const char * SE_API_VERSION_REVISION
- const unsigned int SE_API_BUILD
- const se_time_t SE_TIMEOUT_INFINITE
- const se_contr_t SE_ADDR_CONTR_ALL
- const se_slot_t SE_ADDR_SLOT_ALL
- const se_fpga_t SE_ADDR_FPGA_ALL
- const se_fpga_t SE_ADDR_FPGA_HOST
- const se_reg_t SE_ADDR_REG_EOT
- const unsigned int SE_LENGTH_ADDR_SLOT
- const unsigned int SE_LENGTH_ADDR_FPGA
- const unsigned int SE_LENGTH_ADDR_REG
- const unsigned int SE_LENGTH_CMD

10.1 Typedef Documentation

10.1.1 typedef unsigned int se_slot_t

10.1.2 typedef unsigned int se_fpga_t

10.1.3 typedef unsigned int se_reg_t

10.1.4 typedef unsigned int se_cmd_t

10.1.5 typedef unsigned int se_contr_t

10.1.6 typedef unsigned int se_machine_t

10.1.7 typedef unsigned char se_flag_t

10.1.8 typedef unsigned long int se_time_t

10.2 Enumeration Type Documentation

10.2.1 enum SE_STATUS

The SciEngines RIVYERA Host API return values.

Enumerator:

SeApiSuccess Returned whenever something worked well.

SeApiFailed Returned as a general error code.

SeApiInvalidMachine Returned if the specified machine does not exist.

SeApiMachineNotAvailable Returned if the specified machine is not yet allocated.

SeApiMachineInUse Returned if the specified machine is allocated by another user.

SeApiInvalidAddress Returned if the specified address was not valid.

SeApiWriteTimeout Returned if the method returned due to a timeout during write.

SeApiReadTimeout Returned if the method returned due to a timeout during read.

SeApiFileError Returned when a specified file was not found or user has insufficient privileges.

10.2.2 enum SE_READMODE

Enumeration containing all necessary values for the SciEngines API read modes. Active read mode means that the controller initiates a read request to the target element, so there will be data with api_i_cmd = CMD_RD incoming at its input register. In passive read mode, the controller only seeks already received data. In read request mode, reading will return immediately after initiating the read request. It will not wait for the FPGA to respond. After a read request, it is recommended to use a passive read to wait for the FPGAs respond. In general case, when your FPGA code makes use of the autonomous write capability, it is recommended to use passive read mode. When the FPGA code does not use autonomous writes and only reacts to incoming read requests, it is not needed to think about passive reads.

Enumerator:

SeReadActive Active read mode.

SeReadPassive Passive read mode.

SeReadRequest Requests a read, only

10.2.3 enum SE_FPGA_TYPE

Enum containing all chips supported by SciEngines API.

Enumerator:

none

xc3s1000_4ft256

xc3s1500_4fg676

xc3s5000_4fg676

xc6slx75_3fg484

xc6slx150_3fg676

xc4vsx35_10ff668

10.2.4 enum SE_ROUTING_METHOD_T

Enumerator:

se_routing_normal

se_routing_systolic

10.2.5 enum SE_WRITE_BEHAVIOR_T

Enumerator:

se_write_async

se_write_sync

10.3 Function Documentation

10.3.1 const char* se_status2str (SE_STATUS *status*)

This method translates status codes to char arrays, so they can easily be printed.

Parameters:

status Status code to be converted to a string.

Returns:

Constant resulting string.

10.3.2 const char* se_type2str (SE_FPGA_TYPE *type*)

This method translates FPGA Types to readable strings that can easily be printed.

Parameters:

type FPGA Type to be converted to a string.

Returns:

Constant resulting string.

10.4 Variable Documentation

10.4.1 const unsigned short SE_API_VERSION_MAJOR

Major API version.

10.4.2 const unsigned short SE_API_VERSION_MINOR

Minor API version.

10.4.3 const unsigned short SE_API_VERSION_SP

API Service Pack.

10.4.4 const char* SE_API_VERSION_REVISION

API Revision (Human readable).

10.4.5 const unsigned int SE_API_BUILD

API Build number.

10.4.6 const se_time_t SE_TIMEOUT_INFINITE

Constant used whenever a method shall wait infinitely.

10.4.7 const se_contr_t SE_ADDR_CONTR_ALL

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

10.4.8 const se_slot_t SE_ADDR_SLOT_ALL

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.

10.4.9 const se_fpga_t SE_ADDR_FPGA_ALL

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.

10.4.10 const se_fpga_t SE_ADDR_FPGA_HOST

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

10.4.11 const se_reg_t SE_ADDR_REG_EOT

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

10.4.12 const unsigned int SE_LENGTH_ADDR_SLOT

Length of the slot address field in bits.

10.4.13 const unsigned int SE_LENGTH_ADDR_FPGA

Length of the FPGA address field in bits.

10.4.14 const unsigned int SE_LENGTH_ADDR_REG

Length of the register address field in bits.

10.4.15 const unsigned int SE_LENGTH_CMD

Length of the command field in bits.