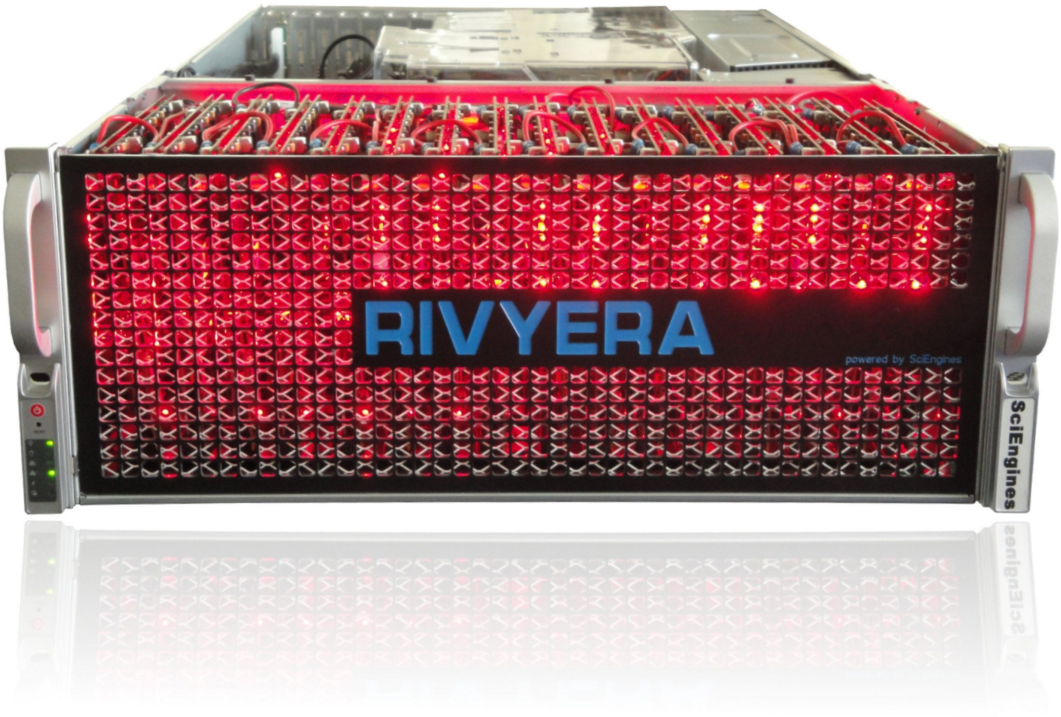
SciEngines
massively parallel computing

RIVYERA API

SciEngines RIVYERA Host-API Documentation

Development User Guide, Host-API (C/C++)

Version 1.95.01



SciEngines GmbH

Revision: 1363 1.95.01 March 9, 2022

Released version

SciEngines RIVYERA Host-API Documentation

Development User Guide, Host-API (C/C++)
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*.

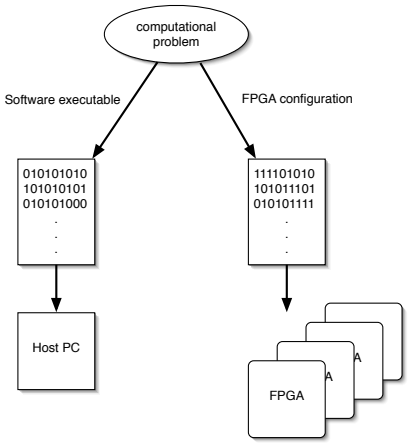


Figure 1. Partitioning of a problem into host- and machine-parts

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.

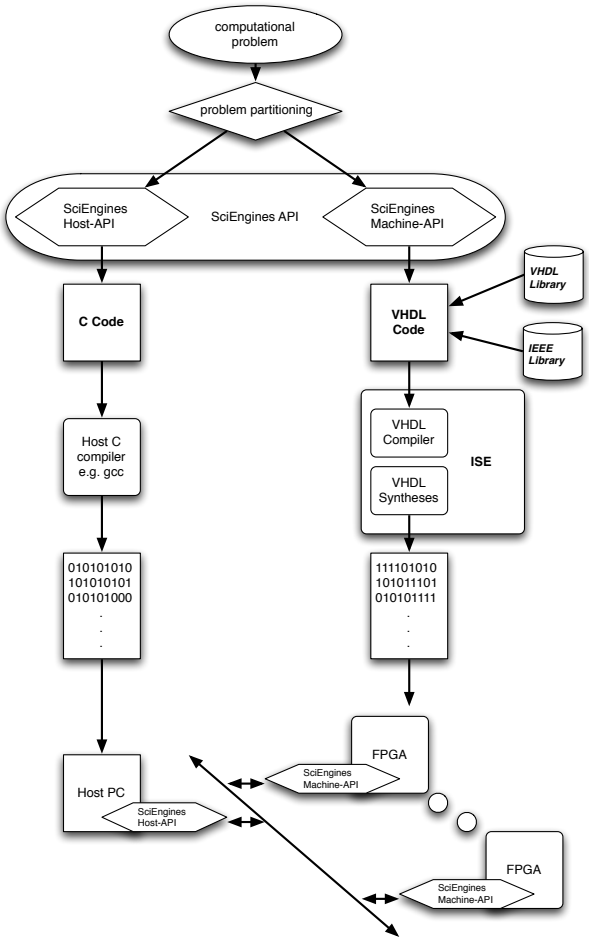


Figure 2. Design flow for multi-component software systems

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

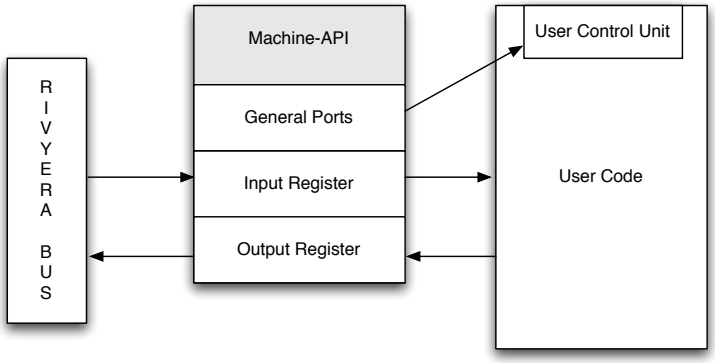


Figure 3. VHDL-API taking care of user design's I/O

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.

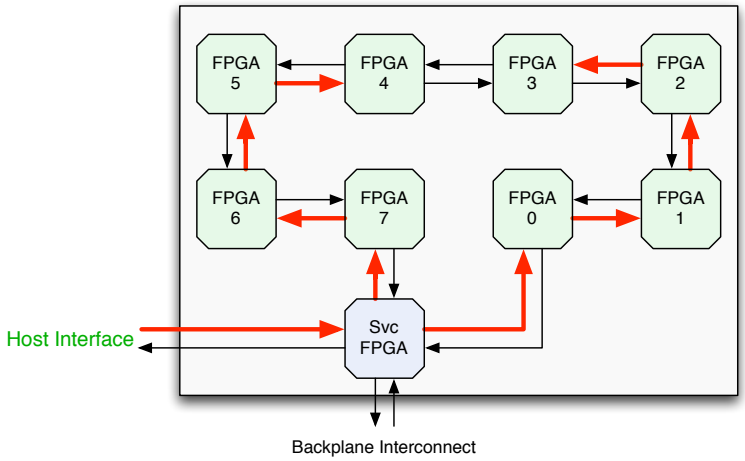


Figure 4. Routing of a host-initiated write

The same principle of routing applies for FPGA $\leftrightarrow$ 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.

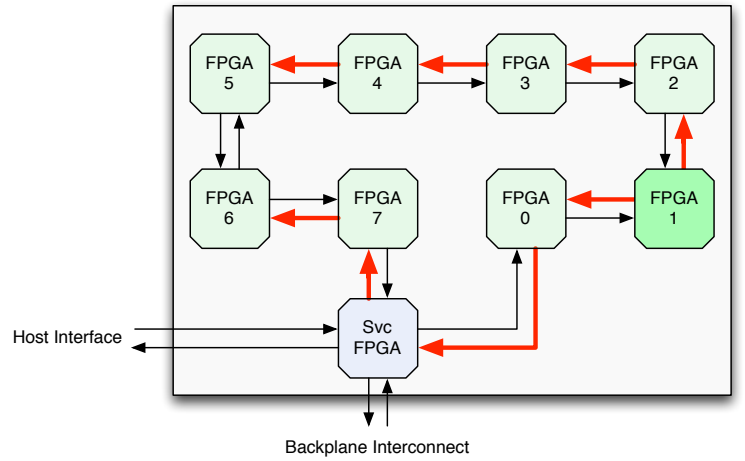


Figure 5. Routing of an FPGA-initiated write

1.3 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 1.2.1. 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.

1.3.1 Machine addressing

The addressing of machine components in general is 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 specify 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`).

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. These FPGA write actions are 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.

2 Data Structure Documentation

2.1 SE_ADDR Struct Reference

Data Fields

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

2.1.1 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.

2.1.2 Field Documentation

se_contr_t contr

The target controller index.

se_slot_t slot

The target slot address.

se_fpga_t fpga

The target FPGA address.

se_reg_t reg

The target register address.

2.2 SE_CONTROLLERINFO Struct Reference

Data Fields

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

2.2.1 Detailed Description

A structure containing useful information about a controller.

2.2.2 Field Documentation

const char* driver_name

The driver used to access this controller

se_slot_t machineSlot

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

unsigned int serial

The serial number of the controller.

2.3 SE_FPGAINFO Struct Reference

Data Fields

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

2.3.1 Detailed Description

A structure containing useful information about an FPGA.

2.3.2 Field Documentation

SE_FPGA_TYPE type

Enum value of the FPGAs chip type.

se_flag_t isProgrammed

Flag indicating if this FPGA is programmed and reacting.

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)

unsigned int firmwareBuild

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

2.4 SE_OPTIONS_T Struct Reference

Data Fields

- SE_WRITE_BEHAVIOR_T write_behavior
- SE_ROUTING_METHOD_T routing_method

2.4.1 Field Documentation

SE_WRITE_BEHAVIOR_T write_behavior

SE_ROUTING_METHOD_T routing_method

2.5 SE_PROGINFO Struct Reference

Data Fields

- se_flag_t is_programmed
- se_flag_t lic_present
- __int32_t lic_lifetime

2.5.1 Detailed Description

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

Since

1.93.10#1195

2.5.2 Field Documentation

se_flag_t is_programmed

Is set to 1 if the last call to se_program() was successful or se_deprogram() has been called unsuccessfully. Otherwise it is set to 0.

se_flag_t lic_present

If a license is present (no matter whether it has lapsed or not) then the value is set to 1, otherwise it is set to 0.

__int32_t lic_lifetime

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 INT32_MAX. If no license is present then the value is set to 0.

2.6 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

2.6.1 Detailed Description

A structure containing useful information about a slot.

2.6.2 Field Documentation

unsigned int serial

The serial number of the slot's card.

se_fpga_t fpgaCount

Number of FPGAs on this slot's card.

se_flag_t isContr

Flag indicating, if this slot is connected to host.

se_contr_t contrIndex

The index of the controller, if isContr is not 0.

se_contr_t prevContr

Contains the previous controller index for the given machine index and slot address. 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 the given slot is not a controller- the next controller index is contained.

Example: assume a machine having 16 slots with addresses 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 the remaining slot 15 controller 2 (connected to slot 14) is contained.

se_contr_t nextContr

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

Example: Assume a machine having 16 slots with addresses 0..15 where slot 1, slot 7 and slot 14 are directly connected to host via individual controllers 0..2. Then for slot 0 controller 0 (which is connected to slot 1) is contained, for slots 1..6 controller 1 (which is connected to slot 7) is contained and for remaining slots 7..15 controller 2 (connected to slot 14) is contained.

unsigned int firmwareVersion

The card's firmware version. bits 31..24: Major Version bits 23..16: Minor Version bits 15.. 8: Version revision bits 3.. 0: Hardware major revision bits 7.. 4: Hardware minor revision

unsigned int firmwareBuild

The card's firmware build.

3 File Documentation

3.1 SeHostAPI.h File Reference

SciEngines RIVYERA API types.

Include dependency graph for SeHostAPI.h:

Functions

- se_machine_t se_getMachineCount (void)

  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 machine index and slot address.
- 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, SE_OPTIONS_T const ∗pOptions)

  Allocates a machine.
- SE_STATUS se_freeMachine (se_machine_t machine)

  Deallocates a machine.
- SE_STATUS se_read (se_machine_t machine, SE_ADDR const ∗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, SE_ADDR const ∗pAddr, __uint64_t const ∗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, SE_ADDR const ∗pAddr, char const ∗pFilename, se_time_t timeout)

  Downloads a given .rbt or .bit file to a machine element.
- SE_STATUS se_deprogram (se_machine_t machine, SE_ADDR const ∗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_getProgInfo (se_machine_t machine, se_slot_t slot, SE_PROGINFO ∗pInfo)

  Reports most recent program information.
- SE_STATUS se_getFPGAInfo (se_machine_t machine, SE_ADDR const ∗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 (char const ∗pFmt,...)

  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.

3.1.1 Detailed Description

This file declares the SciEngines RIVYERA Host API functions.

Author

Jost Bissel
Daniel Siebert

Copyright

3.1.2 Function Documentation

se_machine_t se_getMachineCount (void)

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.

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 has not been allocated before.

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 index and slot address.

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 has not been allocated before.
SeApiInvalidAddress	Returned if the given slot address is invalid.

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.
SeApiMachineNot-Available	Returned if the machine has not been allocated before.

SE_STATUS se_allocMachine (se_machine_t *machine,* SE_OPTIONS_T const * *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. (may be NULL)

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.

SE_STATUS se_freeMachine (se_machine_t *machine*)

This function allows the user to release the usage lock of a machine. After invoking se_free-Machine() 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 has not been allocated before.
SeApiReadTimeout	Returned if a timeout occurred reading the reset acknowledgment.
SeApiWriteTimeout	Returned, if a timeout occurred writing reset command to the machine.

SE_STATUS se_read (se_machine_t *machine,* SE_ADDR const * *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 an explicit previous 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 the 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. (may be NULL, in conjunction with mode SeReadRequest)

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 the slot address to SE_ADDR_SLOT_ALL or the FPGA address to SE_ADDR_FPGA_ALL is allowed. Upon using the mode SeReadActive, a read request is sent and afterwards the reply is read passively. The active read combines the modes SeReadRequest and SeReadPassive.
pCount	Pointer to a size_t variable to store the number of words, that were actually 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 has not been 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.

SE_STATUS se_write (se_machine_t *machine,* SE_ADDR const * *pAddr,* __uint64_t const * *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 has 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 a size_t variable to store the number of words, that were actually 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.
SeApiMachineNot-Available	Returned if the given machine has not been allocated before.
SeApiInvalidAddress	Returned if the given address is invalid.
SeApiWriteTimeout	Returned if the timeout was exceeded before the transfer was completed.

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 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.
SeApiMachineNot-Available	Returned if the given machine was not allocated before.
SeApiInvalidAddress	Returned if the given controller index is invalid.
SeApiWriteTimeout	Returned if the timeout was exceeded before the flush was completed.

SE_STATUS se_program (se_machine_t *machine,* SE_ADDR const * *pAddr,* char const * *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 the 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" options 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 first program the whole card using SE_ADDR_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.
SeApiWriteTimeout	Returned if the timeout was exceeded before the transfer was completed.
SeApiReadTimeout	Returned if the timeout was exceeded before there was an answer from programmed FPGAs.
SeApiFileError	Returned if the given file either does not exist or the user has no read permissions.
SeApiLicenseError	Returned if the there is no license present for given file

SE_STATUS se_deprogram (se_machine_t *machine,* SE_ADDR const * *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.

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 transfers by itself. Whenever an FPGA initiates a transfer to host (see VHDL documentation for getting information of how to send data to host), 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 there is data present at any controller.

Background information: When receiving data from the controller, this data is stored in so called buckets - one bucket for each source address. A bucket contains several streams of data. Such a stream is growing when receiving more data until a special EOT (End Of Transfer) word is received. When receiving an EOT, the currently active stream is closed for writing and a new stream within the same bucket is created. A stream remains active until its data is completely read. Upon stream creation this stream is enqueued to a linked list. When a stream is completely read, it is removed from this linked list. Once a stream is closed for writing due to a received EOT word, this stream is enqueued to a second linked list that holds these EOT-finished streams.

This function returns address and size information for the first entry from the linked list containing EOT-finished streams. If this linked list is empty, the oldest entry from the ordinary linked list is used. Streams containing an EOT are therefore prioritized. Prioritizing for EOT words between controllers is not done when using SE_ADDR_CONTR_ALL. The size information stored in pCount is the number of words for one stream plus 1, if there is also an EOT received. If there is no EOT word received, the stored number is only the number of words.

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 has not been allocated before.
SeApiInvalidAddress	Returned if the given controller index is invalid.
SeApiReadTimeout	Returned if the timeout was exceeded before data occurred.

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

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

Parameters

machine	Machine index to get the temperature from.
slot	Slot address 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.

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 address 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 address does not exist.

SE_STATUS se_getProgInfo (se_machine_t *machine,* se_slot_t *slot,* SE_PROGINFO * *pInfo*)

This function reports information about the programming status for a specific slot. The information stored within the provided structure refers to the most recent call to either se_program() or se_deprogram() - depending on which of these two functions has been called most recently for this specific slot.

Since

1.93.10#1195

Parameters

machine	Machine index to get information from.
slot	Slot address to retrieve information about.
pInfo	Pointer to SE_PROGINFO 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 address does not exist.

SE_STATUS se_getFPGAInfo (se_machine_t *machine,* SE_ADDR const * *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.
SeApiMachineNot-Available	Returned if the given machine has not been allocated before.
SeApiInvalidAddress	Returned if the given address is invalid.

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.
SeApiMachineNot-Available	Returned if the given machine was not allocated before.
SeApiInvalidAddress	Returned if the given controller index is invalid.

void se_comment (char const * *pFmt,* ...)

Parameters

pFmt	string defining the format printing optional arguments

3.2 SeHostAPITypes.h File Reference

SciEngines RIVYERA API types.

Include dependency graph for SeHostAPITypes.h: This graph shows which files directly or indirectly include this file:

Data Structures

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

Macros

- #define none fpga_none
- #define xc3s1000_4ft256 fpga_xc3s1000_4ft256
- #define xc3s1500_4fg676 fpga_xc3s1500_4fg676
- #define xc3s5000_4fg676 fpga_xc3s5000_4fg676
- #define xc6slx75_3fg484 fpga_xc6slx75_3fg484
- #define xc6slx150_3fg676 fpga_xc6slx150_3fgg676
- #define xc4vsx35_10ff668 fpga_xc4vsx35_10ff668
- #define arria10gx115h4f34e3sg fpga_10ax115h4f34e3sg
- #define arria10gx032h4f34e3sg fpga_10ax032h4f34e3sg
- #define fpga_xc6slx75_3fg484 fpga_xc6slx75_3fgg484
- #define fpga_xc6slx150_3fg676 fpga_xc6slx150_3fgg676

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, SeApiLicenseError }
- enum SE_READMODE { SeReadActive, SeReadPassive, SeReadRequest }
- enum SE_FPGA_TYPE {
  fpga_none, fpga_xc3s1000_4ft256, fpga_xc3s1500_4fg676, fpga_xc3s5000_4fg676,
  fpga_xc6slx75_3fgg484, fpga_xc6slx150_3fgg676, fpga_xc4vsx35_10ff668, fpga_10ax115h4f34e3sg,
  fpga_10ax032h4f34e3sg }
- enum SE_ROUTING_METHOD_T { se_routing_normal }
- enum SE_WRITE_BEHAVIOR_T { se_write_async, se_write_sync }

Functions

- char const * 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

3.2.1 Detailed Description

This file contains all types and constants which are needed for using the SciEngines RIVYERA Host API.

Author

Jost Bissel
Daniel Siebert

Copyright

3.2.2 Macro Definition Documentation

#define none fpga_none

#define xc3s1000_4ft256 fpga_xc3s1000_4ft256

#define xc3s1500_4fg676 fpga_xc3s1500_4fg676

#define xc3s5000_4fg676 fpga_xc3s5000_4fg676

#define xc6slx75_3fg484 fpga_xc6slx75_3fg484

#define xc6slx150_3fg676 fpga_xc6slx150_3fgg676

#define xc4vsx35_10ff668 fpga_xc4vsx35_10ff668

#define arria10gx115h4f34e3sg fpga_10ax115h4f34e3sg

#define arria10gx032h4f34e3sg fpga_10ax032h4f34e3sg

#define fpga_xc6slx75_3fg484 fpga_xc6slx75_3fgg484

#define fpga_xc6slx150_3fg676 fpga_xc6slx150_3fgg676

3.2.3 Typedef Documentation

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

3.2.4 Enumeration Type Documentation

enum SE_STATUS

The SciEngines RIVYERA Host API return values.

Enumerator

SeApiSuccess Returned upon success.

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.

SeApiLicenseError Returned when a specific functionality is locked lacks a valid license.

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 FPGA's response. In general, 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 consider passive reads.

Enumerator

SeReadActive Active read mode.

SeReadPassive Passive read mode.

SeReadRequest Requests a read, only

enum SE_FPGA_TYPE

Enum containing all chips supported by SciEngines API.

Enumerator

fpga_none

fpga_xc3s1000_4ft256

fpga_xc3s1500_4fg676

fpga_xc3s5000_4fg676

fpga_xc6slx75_3fgg484

fpga_xc6slx150_3fgg676

fpga_xc4vsx35_10ff668

fpga_10ax115h4f34e3sg

fpga_10ax032h4f34e3sg

enum SE_ROUTING_METHOD_T

Enumerator

se_routing_normal

enum SE_WRITE_BEHAVIOR_T

Enumerator

se_write_async

se_write_sync

3.2.5 Function Documentation

char const∗ se_status2str (SE_STATUS *status*)

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

Parameters

status	Status code to be converted to a string.

Returns

Constant resulting string.

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.

3.2.6 Variable Documentation

const unsigned short SE_API_VERSION_MAJOR

Major API version.

const unsigned short SE_API_VERSION_MINOR

Minor API version.

const unsigned short SE_API_VERSION_SP

API Service Pack.

const char∗ SE_API_VERSION_REVISION

API Revision (Human readable).

const unsigned int SE_API_BUILD

API Build number.

const se_time_t SE_TIMEOUT_INFINITE

Constant used whenever a method shall wait infinitely.

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.

const se_slot_t SE_ADDR_SLOT_ALL

Constant used as wildcard for slot address. 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.

const se_fpga_t SE_ADDR_FPGA_ALL

Constant used as wildcard for FPGA address. 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.

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.

const se_reg_t SE_ADDR_REG_EOT

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

const unsigned int SE_LENGTH_ADDR_SLOT

Length of the slot address field in bits.

const unsigned int SE_LENGTH_ADDR_FPGA

Length of the FPGA address field in bits.

const unsigned int SE_LENGTH_ADDR_REG

Length of the register address field in bits.

const unsigned int SE_LENGTH_CMD

Length of the command field in bits.

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