ClearSpeed™

# CSX600 Runtime Software

# User Guide

Version 3.0

**Document No.** 06-UG-1345 **Revision:** 3.D

January 2008

# Table of contents

# 1 Introduction

This document describes the components that make up the CSX processor runtime package. These are:

- Stand-alone host tools to reset the Advance card and to load and run programs on the cards. See *Chapter 2: Running code on page 9*.
- The host application programming interface and libraries used by a user application on the host to control and communicate with the CSX processor. See *Chapter 3: Host interface library on page 15*.
- A set of diagnostic tools are provided with the runtime and driver software. These can be used to verify the correct installation of hardware and drivers and also to generate diagnostic information if problems are found. See chapter *Chapter 4: Diagnostic software reference on page 83*.

# 2 Running code

To run compiled code on the CSX processor or a simulator, a program needs to be run on the host computer to load the code and start it running. A simple host program called `csrun` is provided as part of the runtime software. This will connect to a card, load and run the CSX executable and then wait for the program to terminate.

Before code is run on the CSX processor, the processor needs to be reset. The command `csreset` (see *Section 2.1 on page 9*) can be used to reset one or all of the CSX processors in a system.

*Note:* *You should ensure that applications are properly terminated because a background process connected to the card will prevent other applications from connecting to it. If a program using the Advance card is terminated abnormally, it is possible that it may continue to run in the background. Further attempts to use the card will fail to connect, giving the process ID of the process still using the card. This process must be terminated before the Advance card can be used by another program. See Section 3.8: Access control and the lock file on page 81.*

## 2.1 csreset

`csreset` provides a means of resetting the CSX processor. At a system level `csreset` configures the bus and sets the CCBRs so you can do memory transfers to both CSX processors. `csreset` also initializes the DDR memory. It configures the appropriate 96 PEs for each CSX processor, taking redundancy into account. It loads the microcode and sets up the PE number in PE memory. It also clears the DDR memory, which is necessary to avoid spurious ECC errors.

### 2.1.1 When to use csreset

You need to run `csreset` on each card after a hardware reset, specifically after a system boot. After this initial boot, `csreset` should only be run if a program goes wrong and hangs or if the setup is destroyed. For example, if the code trashes the PE number.

*Note:* *If `csreset` is unable to reset the card, run the script `recover_board` as described in Section 2.1.4: Recovering the card on page 11. It is important that you then rerun `csreset` after this.*

Under normal circumstances, resetting should only be needed once, at system or simulator startup, when all chips on the cards installed in the system are reset. `csreset` also provides a finer level of control over which cards, or chips on a card, are to be reset.

### 2.1.2 Invoking csreset

The command line for `csreset` is:

```
csreset [option]*
```

### 2.1.3 Command-line options

The `csreset` command-line options are summarized in *Table 1*.

| Long name | Short name | Valid values | Description |
|---|---|---|---|
| `--all` | `-A` | | Reset all cards. |
| `--chip` | `-c` | integer | Select a given chip. |
| `--help` | `-h` | | Displays information on command-line usage. |
| `--host` | | *name* or *address* | Connect to a simulator on a remote host. |
| `--instance` | `-i` | integer | Select a card or a simulator instance. |
| `--no-reset` | | | Connect but do not reset processors. |
| `--sim` | `-s` | | Connect to a simulator rather than hardware. |
| `--timeout` | `-t` | | Timeout for reset, in seconds. |
| `--verbose` | `-v` | | Prints out detailed information on the screen about the card and chips. |
| `--version` | `-V` | | Display version information. |

**Table 1. csreset command-line options summary**

**`-A`**
**`--all`**

Resets all the cards in the system. The `-A` option has no effect if the `-i` option is used.

**`-c`** ***`chip`***
**`--chip`** ***`chip`***

Selects the chip to reset. Chip numbering starts with 0 for the first chip on a card. If the `-c` option is not used, all chips on the specified cards will be reset.

**`-h`**
**`--help`**

Displays information on command line usage.

**`--host`** ***`name`*** | ***`address`***

If the simulator is running on a remote host, this option must be used to inform `csreset` where the connection should be made.

**`-i`** ***`number`***
**`--instance`** ***`number`***

Specifies which card or instance of the simulator to connect to. Instance numbers start from 0. If there are multiple cards installed, either the `-A` or `-i` option must be used.

**`--no-reset`**

Causes `csreset` to connect to the specified card or simulator but not perform a reset. This can be useful with the `--verbose` option to get information about a card without resetting it.

**`-s`**
**`--sim`**

Specifies that `csreset` should connect to a simulator rather than search for hardware.

**`-t`**
**`--timeout`**

Specifies the timeout for reset in seconds. The default is no timeout.

**`--verbose`**

Gives detailed information about the FPGA version, the temperature (°C) of the cards, the card’s serial number, the final card test date, the installed memory type and the CSX processor fuses.

#### Examples

Resetting all cards:

```
csreset -A
```

Resetting all chips on the second card (instance 1):

```
csreset -i 1
```

Resetting all simulated cards:

```
csreset --sim -A
```

### 2.1.4 Recovering the card

This release includes a script for resetting the Advance card when `csreset` fails to do so. This does a ‘hard’ reset of the processors. This functionality will be incorporated into `csreset` in a future release.

Before using the reset script (`recover_board`), gather any diagnostic or debugging information as all state information will be lost by the hard reset. For example, make a note of the output from `csreset -v`.

Before running the script, make sure you have set up your environment. In Linux, source the `bashrc` file (usually present in `/opt/clearspeed/csx600_m512_le/bin`). In Windows, start a command prompt using the shortcut from the ClearSpeed start menu item.

If you have more than one card, set the environment variable `LLDINST` to the instance number of the card to be recovered.

For example, to reset the first card in Linux, enter:

```
export LLDINST=0
```

The same variable is set in Windows by using the command:

```
set LLDINST=0
```

To run the script:

1. Type the command: `recover_board`

   The message `Board recovery utility` is displayed on the screen.

2. Press either [`Return`] to continue or [`Ctrl`]+[`C`] to exit.

   If you press [`Return`], the following will appear on the screen:

   ```
   Starting...
   25%
   50%
   ```

```
75%
DONE
Board recovery attempted.
```

3. Rerun `csreset` as described in *Section 2.1.2: Invoking csreset on page 9*.

This procedure can be repeated with different values of `LLDINST` to reset each card in the system. Remember to ‘unset’ the environment variable after this.

To unset the variable in Linux, enter:

```
unset LLDINST
```

To unset the variable in Windows, use the command:

```
set LLDINST=
```

## 2.2 csrun

`csrun` is a simple system loader that enables executables to be run without the need to create a host application.

### 2.2.1 Invoking csrun

The command line for `csrun` is:

```
csrun [option]* filename
```

Where the *filename* parameter is the executable `.csx` file name. `csrun` will search the current directory and the paths specified in the `CSPATH` environment variable to find the executable.

### 2.2.2 Command-line options

The `csrun` command-line options are summarized in *Table 2*.

| Long name | Short name | Valid values | Description |
|---|---|---|---|
| `--chip` | `-c` | integer | Select which chip to run on (0 or 1). Default: 0. |
| `--csx-args` | `-a` | string | Arguments for argc and argv in the CSX program. This must be the last option before the CSX file name. |
| `--help` | `-h` | | Displays information on command line usage. |
| `--host` | | *name* or *address* | Connect to a simulator on a remote host. Default: localhost. |
| `--instance` | `-i` | integer | Select a card or a simulator instance (0,1,2,...). Default: any available. |
| `--print-csx` | `-p` | | Print the CSX return value to stdout., if the CSX file ran successfully. |
| `--reset` | `-r` | | Perform a reset before loading and running the CSX file. |
| `--return-csx` | `-e` | | Return the CSX return value as the exit code, if the CSX file ran successfully. |

**Table 2. csrun command-line options summary**

| Long name | Short name | Valid values | Description |
|---|---|---|---|
| `--sim` | `-s` | | Connect to a simulator rather than hardware. |
| `--timeout` | `-t` | integer | Timeout for connect and execute, in seconds. This causes csrun to wait for an available card. |
| `--verbose` | `-v` | | Switch on verbose output. |
| `--version` | `-V` | | Display version information. |

**Table 2. csrun command-line options summary (continued)**

**`-a string`**
**`--csx-args string`**

Arguments for `argc` and `argv` in the CSX program. This must be the last option before the CSX file name.

**`-c`** ***`chip-id`***
**`--chip`** ***`chip-id`***

Selects the chip on which to run the code. Chip numbering starts with 0 for the first chip on a card. The default if not specified is 0, the first chip.

**`-e`**
**`--return-csx`**

Specfies that the CSX return value is returned as the exit code if the CSX file ran successfully.

**`-h`**
**`--help`**

Displays information on command line usage.

**`--host`** ***`name`*** | ***`address`***

If the simulator is running on a different host, this option must be used to inform `csrun` where the connection should be made.

**`-i`** ***`number`***
**`--instance`** ***`number`***

Specifies which card or instance of the simulator to connect to. Instance numbers start from 0. The default, if not specified, will connect to the next available card or simulator.

**`-p`**
**`--print-csx`**

Prints the CSX return value to stdout, if the CSX file ran successfully.

**`-r`**
**`--reset`**

Performs a reset before loading and running the CSX file.

**`-s`**
**`--sim`**

Specifies that `csrun` should connect to a simulator rather than search for hardware.

**`-t integer`**
**`--reset integer`**

Timeout for connect and execute, in seconds. This causes csrun to wait for an available card.

**`-v`**
**`--verbose`**

Displays more information from `csrun`.

**`-V`**
**`--version`**

Displays version information. This includes the overall software release version and the specific build of `csrun`.

**Example**

To run an executable on the second chip on the third card, use a command of the form:

```
csrun -i 2 -c 1 executable_name.csx
```

To load an executable that is not in the current `CSPATH` search path, a full path name can be specified, for example:

```
csrun /home/fred/csx/fred.csx
```

# 3 Host interface library

This chapter describes the CSX processor driver library and the ClearSpeed Application Programming Interface (CSAPI) functions. To load code onto a CSX processor and communicate with it, a host program must use the CSAPI.

A set of host driver libraries is provided to allow host applications to communicate with and control the installed Advance boards via the CSAPI. The user-level libraries make use of a kernel-level driver to provide a complete driver for the Advance boards.

The word ‘card’ is used in this chapter to refer to an installed Advance card or an Advance card simulator. The host interface is used in the same way on simulators and hardware, except where indicated in the text.

## 3.1 CSX processor driver library

The driver library provides an API known as CSAPI which is available for C and C++ programs using the header file `csapi.h`. The library consists of a set of dynamic shared libraries. The driver libraries are provided as `.dll` files on Windows or `.so` files on Linux.

To identify the Advance boards the CSAPI functions take a state object which describes the card. To use the CSAPI interface, a `CSAPI_new` call must be made to build and return this state object.

The CSAPI library is thread safe. The CSAPI library uses host semaphores and other concurrency objects to provide concurrent but safe access to the Advance boards.

## 3.2 Linking host applications with CSAPI

The runtime requires the host application to link against one library. The other libraries are loaded dynamically. The only library that needs to be linked statically for the runtime is the CSAPI stub library:

- In Linux: `libcsapi.a`
- In Windows XP: `csapi.lib`

The following instructions explain how to setup the environment so you can compile and run a simple CSAPI program on a Linux or Microsoft Windows XP operating system.

### 3.2.1 Linux

The following describes how to link the host application with `csapi` on a Linux operating system.

Before you start to build programs using CSAPI, it is vital to set up the environment by sourcing `bashrc`. When you have done this, compile and link a CSAPI program as follows:

```
gcc -I $CSHOSTINC -L $CSHOSTLIB mandlebrot1.c -lcsapi -ldl
```

The `-I` option specifies the include directory for the CSAPI header files, which are included by the host application. The `-L` option specifies the library directory for the CSAPI stub library, which is statically linked to by the host application.

The `-ldl` option links in the dynamic loader library (`libdl.a`) which allows the CSAPI stub library to load the relevant functional libraries. The correct runtime library is selected, based on a number of dynamic environmental factors, such as whether the debugger is used.

The dynamically loaded libraries are located via the environment variable `LD_LIBRARY_PATH` which is set by the `bashrc` script.

### 3.2.2 Microsoft Windows

The following describes how to use Microsoft Visual Studio to build host-side applications.

To build CSAPI programs on Windows using Visual Studio, you need to setup a Visual Studio Project, specifying the library path and runtime library under the Linker options.

The ClearSpeed runtime supports Visual Studio 2005. If you use an unsupported version of Visual Studio, you may get an error message about corrupt debug information when linking the application.

The recommended way to use Visual Studio is to run it from the ClearSpeed command prompt with the option `/useenv`. This will use the appropriate environment setup.

It is also possible to set the Project environment from within Visual Studio but this is generally not as convenient as using the environment set by the ClearSpeed script.

To use Visual Studio to build `csapi` programs, do the following in a DOS shell:

1. Type one of the following:
   – For the full edition of Visual Studio, type: `devenv.exe /useenv`
   – For the Express edition, type: `vcexpress /useenv`
2. Setup the development project as described in *Table 3*.

| Task | Steps |
|---|---|
| Select a new project | a) Select `File->New Project`.<br>b) Select `Win32->Win32 Console Application` as an example.<br>c) Enter the name of the project, then select `Next->Finish`. |
| Set Project properties | a) Select `View->Solution Explorer`.<br>b) Select `Project Name` then right-click `Properties`.<br>c) Expand `Configuration Properties -> C/C++ -> General`.<br>d) Add `%CSHOSTINC%` under "Additional Include Directories".<br>e) Add the following under “Linker” complete with quotes:<br>`"C: \ProgramFiles\ClearSpeed\csx600_m512_le\lib\csapi.lib"` |

**Table 3. Setting up the development project**

3. You can now either edit `<projectname>.c`, or you can add your own file in the Solution Explorer and remove `<projectname>.c`.
4. To build your application, select `Build->Build <projectname>`.

You should now be able to compile and run a simple program using `CSAPI`.

## 3.3 Using CSAPI

An accelerated application consists of two parts: the code running on the host processor (the *host program*) and the code running on the CSX processors on one or more Advance cards (the *CSX program*). The CSX code may be a library that accelerates standard functions, such as CSXL, or custom code that accelerates the main functions in your application.

## 3.4 Building programs

It is recommended that your host application checks it is using a compatible version of the CSAPI. It can do this by checking the value of `CSAPI_HEADER_VERSION_MAJOR` defined in the `csapi.h` header file. This will confirm that the CSAPI functions have the expected parameters. It should also check the value of `CSAPI_HEADER_VERSION_MINOR` to confirm that the CSAPI functions behave as expected.

If the major version is not the expected value, the program will not compile.

If the major version matches but the minor version is different, the application will build but you will need to do extra testing to ensure that any changes in the CSAPI behavior do not affect your program.

### 3.4.1 Connection and initialization

Most CSAPI functions require a `CSAPIState` object to be passed as a parameter. This is created by calling the `CSAPI_new` function, which must be called once for each card used by the host application.

One exception to this is the `CSAPI_num_cards` function, which obtains the number of Advance cards installed in the system. This function can be called before creating the `CSAPIState` objects. The `CSAPI_num_cards` function does not include simulators.

It is also possible to call the `CSAPI_version` function without a `CSAPIState` when requesting the interface, runtime package or build versions. The interface version is the same as the `CSAPI_HEADER_VERSION_MAJOR/MINOR` definitions from the `csapi.h` header file. The runtime package version will be the version of the distribution against which the user application has been linked, and the build version gives the build time of this distribution.

You must call `CSAPI_new` before using the rest of the CSAPI functions. The `CSAPI_new` function loads the CSAPI library and provides a `CSAPIState` object that is used by the other CSAPI functions. You must statically link your code against the CSAPI stub library, `libcsapi.a` or `csapi.lib`, which will then load the dynamically linked CSAPI library at runtime. This library contains the CSAPI functions called by the host. If the ClearSpeed debugger, `csgdb`, is used, a different dynamically linked library will automatically be loaded to provide function tracing to the debug and trace tools.

You can then connect each `CSAPIState` object to a card or simulator using the `CSAPI_connect` function. This function allows the connection type to be specified using one of the types given in *CSAPIConnectType on page 74*. Use `CSC_Direct` for hardware and `CSC_Socket` for a simulator.

You *must* check the return code to ensure that the connection was successful. If the return code is not equal to `CSAPIErrno_success`, you can pass it to the

`CSAPI_get_error_string` function, which will fill the provided char array with an error string corresponding to the return code. If the `CSAPI_connect` function failed because the card was already in use, the error string will contain the user name and process ID currently connected to the card.

#### Initialization

Once connected you can call `CSAPI_reset` on each of the processors (see *Section 3.4.2: Obtaining information on page 18* for information on how to determine the number of processors). This avoids the need to run `csreset` before the application is run; a call to `CSAPI_reset` will add a short delay to the initialization of the application.

You can call the `CSAPI_set_system_param` function after the connection and modify the configuration of the card and driver. This function is provided for debugging purposes and is not needed in normal use. Some of the parameters require the card to be reset after they have been configured, so you may need to call `CSAPI_reset` again after reconfiguration.

### 3.4.2 Obtaining information

Once a `CSAPIState` object has been created and connected to a card, you can call various CSAPI functions (listed in *Table 4*) to obtain information about the processors on the card.

| Function | Description |
|---|---|
| `CSAPI_version` | Obtains the processor, firmware and kernel driver versions. It can also be called to obtain the card type (for example, e620). |
| `CSAPI_endianness` | Determines the endianness of each processor. |
| `CSAPI_get_system_param` | Determines the value of various system parameters (see *Section 3.4.11: System parameters*). |
| `CSAPI_num_threads` | Determines the total number of threads on each processor. |
| `CSAPI_num_semaphores` | Determines the number of available semaphores on each processor (see *Section 3.4.6: Semaphores*). |
| `CSAPI_get_free_memory` | Determines the amount of available memory on each processor (see *Section 3.4.8: Memory allocation*). |
| `CSAPI_num_processors` | Determines the number of processors on the card. |
| `CSAPI_num_pes` | Determines the number of processing elements on each processor. |
| `CSAPI_connection_info` | Determines the connection type (card or simulator) and the instance number of the current connection. |
| `CSAPI_feature` | Checks for the existence of features on the card (see *Section 3.4.12: Features*). |
| `CSAPI_status` | Reads the current status of card (see *Section 3.4.13: Status*). |

**Table 4. CSAPI information functions**

### 3.4.3 Loading and running a program

This section describes how to load and run a CSX program.

#### Loading CSX programs

The `CSAPI_load` function loads a CSX program from a `.csx` file on to the card. If both processors are used, you can choose to load either two statically linked `.csx` files (one per processor) or a single dynamically linked `.csx` (which is relocated for each processor by the loader).

Before each program is loaded, `CSAPI_load` will initialize the processor so that it is ready to run. This involves halting the processor, clearing the caches and semaphores, and running the bootstrap to reinitialize the PEs.

If the CSX program was linked statically, the `.csx` file will contain the address at which the program will be loaded. If the program was linked dynamically, it will be loaded to an appropriate address on the card.

You can load multiple CSX programs at the same time. Each call to `CSAPI_load` will provide a `CSAPIProcess` handle for the program. You can then use this handle with the other CSAPI functions.

Statically linked CSX programs will fail to load if the address they will be loaded to is already in use. The `CSAPI_load` function will return the error `CSAPIErrno_address_in_use`. This could be caused by another CSX program that has already been loaded, or because the address has already been allocated by calling the `CSAPI_allocate_shared_memory` function. This can be avoided by loading statically linked CSX programs before loading dynamically linked CSX programs and allocating memory (see *Section 3.4.8: Memory allocation on page 21*).

### Running the CSX program

Once a CSX program has been loaded, you can run it by calling `CSAPI_run`. The CSX program will then run until either `CSAPI_halt` is called or it terminates. The host program can wait for the CSX program to terminate by calling `CSAPI_wait_on_terminate`. When this function returns, check the return code to ensure the CSX program terminated successfully. You can get the exit code for the CSX program by calling `CSAPI_get_return_value`.

You must call `CSAPI_load` before calling `CSAPI_run` again. This will reinitialize the processor and the loaded memory sections. The load will return the same handle as before if the file name is the same. The CSX program will only be read from the file on the first call. Subsequent calls will load the CSX program from host memory.

Variables in the CSX program will be initialized by calling `CSAPI_load`. Their values can be modified before calling `CSAPI_run`. These modifications must be repeated after each call to `CSAPI_load`. The following functions will modify variables in the CSX program

- `CSAPI_get_symbol_value` followed by a write to the symbol address with the `CSAPI_write_mono_memory` function (see *Section 3.4.7: Symbols*).
- `CSAPI_allocate_shared_memory` when the `process` and `symbol_name` parameters are not `NULL` (see *Section 3.4.8: Memory allocation*).
- `CSAPI_allocate_shared_semaphore` when the `process` and `symbol_name` parameters are not `NULL` (see *Section 3.4.6: Semaphores on page 21*).

The `CSAPI_start` and `CSAPI_halt` functions are provided for debugging and are not needed in normal use. It should be noted that calling these functions is reference counted, so the processor will only be started when `CSAPI_start` has been called the same number of times as `CSAPI_halt`.

### 3.4.4 Unloading a program and disconnecting

You can unload a CSX program by calling `CSAPI_unload` with the `CSAPIProcess` handle that was obtained when it was loaded. This will release the resources being used by the CSX program so that another program can be loaded in its place.

When you no longer require a card, you can delete the `CSAPIState` for the card by calling `CSAPI_delete`. This will disconnect from the card and destroy the state object. All programs loaded on the card and all memory allocations on the card will be discarded. When the last `CSAPIState` object has been deleted, `CSAPI_delete` will unload the CSAPI library.

Always call `CSAPI_delete` before exiting your host application so that the card can be put in a low-power state and the next application can connect cleanly. If you terminate the host application without calling `CSAPI_delete`, the processors on the card may continue running. This means that the next application will need to call `CSAPI_reset` so that the processors stop and reinitialize. Furthermore, the next application will take longer to connect while the driver processes the stale entry left in the lock file and checks that the previous application is no longer running. (See *Section 3.8: Access control and the lock file*)

### 3.4.5 Events

The card will generate an interrupt when it requires interaction with the host. These interrupts are usually handled by the driver, but some interrupts will generate an event, which you can intercept.

The event must be handled by a callback function. The default callback function can be obtained by calling `CSAPI_get_callback`. Your callback function can be registered by calling `CSAPI_register_callback`. Your callback function should call the original callback function before starting or when it has finished. This allows multiple callback functions to be chained together on a particular event.

Events are enumerated in the `csapi_events.h` header file. The following events are defined in *Table 5*.

| Event | Description |
|---|---|
| `CSE_TemperatureAlert` | One of the devices on the card has exceeded its temperature alert threshold. |
| `CSE_BusMonitor` | The bus monitor has been triggered, typically by an access to an invalid address. |
| `CSE_DdrEcc` | An ECC event has been generated by the local memory controller. |
| `CSE_SystemEcc` | An ECC event has been generated by the CCBR, TSC, PIO, Array Controller or a PE. |
| `CSE_Break` | The program running on the processor has hit a break point, used by the debugger. |
| `CSE_Terminate` | The program running on the processor has terminated. |
| `CSE_Print` | The program running on the processor is printing to the terminal on the host. |
| `CSE_StackOverflow` | The stack for the program currently running on the processor has overflowed. |
| `CSE_SemaphoreNonZero` | A semaphore on the processor has been signalled. |
| `CSE_SemaphoreOverflow` | A semaphore on the processor has been signalled too many times and overflowed. |

**Table 5. Events in csapi_events.h**

| Event | Description |
|---|---|
| `CSE_PapiOverflow` | One of the PAPI counters on the processor has overflowed. |
| `CSE_Malloc` | The malloc function on the processor requires another block allocation from the host. |
| `CSE_StackCheck` | The dynamic stack on the processor needs to be checked by the host. |

**Table 5. Events in csapi_events.h (continued)**

### 3.4.6 Semaphores

The CSAPI library includes semaphore functions to signal and wait on semaphores. These semaphores are typically used to synchronize between the host program and the CSX program. The `CSAPI_semaphore_wait` function blocks the host application until the specified semaphore is signalled. The `CSAPI_semaphore_signal` function signals the specified semaphore, which the CSX program can wait on.

There are two types of semaphore on the CSX600 processor:

- TSC (Thread Sequence Controller) semaphores are used to synchronize threads within a processor and between the host and a processor.
- GSU (Global Semaphore Unit) semaphores are used to synchronize processors on the same card, the host DMA engine and the host application. GSU semaphores can be signalled with a data value, which is passed to the processor or host application when it waits on the semaphore.

| Event | TSC | GSU |
|---|---|---|
| Total number on each processor | 128 | 16 |
| Signal overflow limit | 255 | 65536 (32 with data) |
| Can carry data | No | Yes |
| Can be operated on by other processors | No | Yes |

**Table 6. Properties of CSX600 semaphores**

### 3.4.7 Symbols

Symbols are typically global variables or entry points in the CSX program loaded on the card. The `CSAPI_get_symbol_value` function returns the address of the symbol on the card. This address could be the start of an array or a single variable, for example. The host program can read from or write to this address, and this provides a basic method for the host to transfer data to or from the card.

If the symbol is a pointer to memory that has not yet been allocated, you can allocate memory and write the result to a symbol with a single call to one of the CSAPI memory allocation functions described in *Section 3.4.8: Memory allocation*. This avoids the need to allocate memory and write the result to a symbol using separate CSAPI calls.

### 3.4.8 Memory allocation

The CSAPI memory allocation functions allocate memory from either the DRAM or SRAM attached to each processor.

Both CSX processors on the Advance card can access all of the memory on the card. Each processor can access its own memory and the memory attached to the other processor. However, accessing memory attached to another processor adds latency and will increase bus contention if both processors frequently access each other's memory. Therefore, when using the CSAPI allocation functions, you should allocate memory in the address space of the processor that will access the memory most frequently.

A single block of memory can only be allocated from the DRAM or SRAM attached to a single processor. If there is not enough memory available, the allocation functions will return an error rather than try to use the memory attached to the other processor.

You can use the `CSAPI_get_free_memory` function to obtain the total available memory on the specified processor. This will take account of any dynamically linked CSX programs that have been loaded and any memory allocations that have been made. The total available memory is a summation of all the available memory blocks. This is not necessarily the same as the largest amount of memory that can successfully be allocated.

Memory can be allocated in three ways:

- Static allocation: CSX programs containing static arrays can be loaded on to the card. These static arrays will be contained within the memory allocated for the loaded program.
- Host side dynamic allocation: The CSAPI memory allocation functions can be called before you run the CSX program. These allocations are termed *shared memory* as they can be used by both the host and the card.

*Note:* *This memory is not truly shared memory; it is not directly accessible by both the host and the CSX processors. However, the address of the allocated memory is made available to both the host program and the CSX program.*

- CSX program dynamic allocation: The **C*n*** memory allocation function `malloc` can be called by the CSX program. These allocations are termed *runtime memory* as they will automatically be released when the CSX program terminates. By default, these memory allocations are only visible to the CSX program. However, the address of the allocation can be obtained by the host program if necessary.

## Static allocation

CSX program allocations and CSAPI memory allocations are made from the lowest available address. When the CSX program starts running, the dynamic stack (*[6]: SDK Reference Manual*) is placed on top of the allocated memory blocks, where it can grow upwards (see *Figure 1*). Memory allocated in the CSX program using `malloc`, is made at the highest available address. The size of the stack is checked when the allocation is made to ensure the allocation does not collide with the stack.

*Figure 1* shows the memory map for one processor. The relative locations of CSX programs, 'shared' memory allocations, runtime memory allocations and the program call stack are also shown.

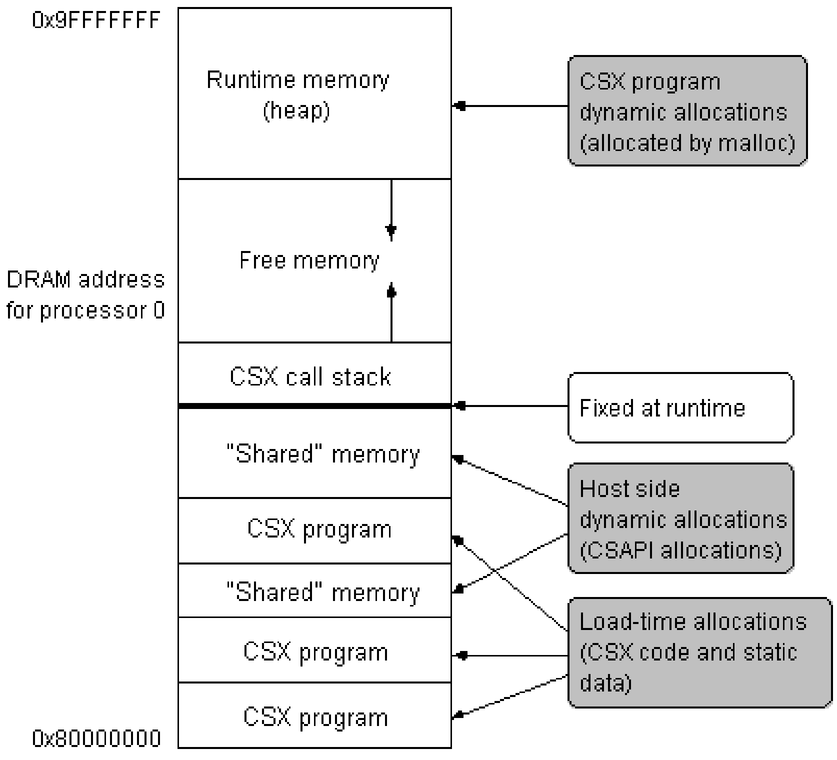
0x9FFFFFFF
Runtime memory (heap)
CSX program dynamic allocations (allocated by malloc)
Free memory
DRAM address for processor 0
CSX call stack
Fixed at runtime
"Shared" memory
Host side dynamic allocations (CSAPI allocations)
CSX program
"Shared" memory
Load-time allocations (CSX code and static data)
CSX program
CSX program
0x80000000

**Figure 1. Memory map**

## Host side dynamic allocation

Host side allocations of 'shared' memory are made with the functions `CSAPI_allocate_shared_memory` and `CSAPI_allocate_static_shared_memory`. The first of these will find the lowest available address with the required available space. The second function will allocate memory at the specified address if possible. This can be useful for debug and testing, but you should normally use the `CSAPI_allocate_shared_memory` function.

The CSAPI allocation functions also pass the address of the allocated memory to a symbol (global variable) in the loaded CSX program, identified by its process handle. This avoids

the need to call `CSAPI_get_symbol_value` and `CSAPI_write_mono_memory` to pass the address of the allocated memory to the CSX program. This is the preferred method for memory allocation where the size of the memory required can be determined before the CSX program starts running. The allocation will persist after the CSX program has terminated and will be available to the host program until it is explicitly released using `CSAPI_free_memory`. You must call this when the memory region is no longer required. The only other way to release the memory is to destroy the state by calling `CSAPI_delete`.

#### CSX program dynamic allocation (malloc)

When the size of the memory block required can only be determined by the CSX program, it can be allocated by calling `malloc` on the card. You can obtain the address of the allocated memory in the host program by storing it in a global variable in the CSX program and then calling `CSAPI_get_symbol_value` on the host. When the CSX program terminates any memory allocated by `malloc` will be released. If the memory is used to return results to the host, the host program should read the data before the CSX program terminates. This could be controlled by using a pair of semaphores: the CSX program would signal one to tell the host that the data is ready to be read from the memory. Then the host would signal the other when it has finished, allowing the CSX program to terminate.

When the CSX program calls `malloc`, an event is sent to the host program where the memory allocations are managed. The host returns an available address of the required size or larger and this address is used by `malloc`. Calling `free` in the CSX program will allow `malloc` to reuse the memory without needing to return to the host— the host does not need to be informed of the memory release because memory can only be allocated by calling `malloc` when the CSX program is running.

This means that there will be a small delay when `malloc` needs to go to the host for more memory. You can minimize the effect of this by keeping the number of calls to `malloc` to a minimum and ensuring that each call requests the total memory that will be required, rather than making multiple calls for small amounts of memory. Calling `malloc` and then `free` in a loop will not have a significant performance impact as only the first call to `malloc` will need to go to the host for memory. Subsequent calls will simply reallocate the memory that has just been released.

### 3.4.9 Memory and register access

#### Memory access

`CSAPI_read_mono_memory` and `CSAPI_write_mono_memory` are the basic functions for transferring data between a host program and memory on the card (*mono* memory). The address on the card is usually obtained from a CSAPI memory allocation or a call to `CSAPI_get_symbol_value`. You would typically use the latter when the symbol is a global variable in the CSX program or a pointer containing an address returned by `malloc`.

The memory transfer functions take a structure of transfer parameters. The parameters used are shown in *Table 7*.

| Parameters | Description |
|---|---|
| `address_check` | Validates or checks the allocation of the card side address. Options are:<br>● No check.<br>● Check the address is valid.<br>● Check the address has been allocated. |
| `alignment_fix` | Breaks up or shifts badly aligned data to maximize use of DMA. Options are:<br>● No check (badly aligned data will be transferred slowly).<br>● Return an error if the data is badly aligned,<br>● Break up badly aligned data into sections with good alignment that can use DMA,<br>● Shift data to another address so that DMA can be used. |
| `halt_processor` | Flushes the cache on the affected processor before the transfer. |
| `flush_cache` | Reads environment variables to determine the connection type (see *Chapter 5 on page 85*). |
| `start_sem` | GSU semaphore to semaphore to wait on before the transfer is started. |
| `complete_sem` | GSU semaphore to semaphore to signal when the transfer completes. |

**Table 7. Memory transfer parameters**

Two predefined structures are available with the following settings:

- `CSAPI_TRANSFER_PARAMS_SAFE` - Checks the address is valid and will shift the data on the host to use DMA. Halts the processor and flushes the cache to avoid any memory contention or cache coherency issues.
- `CSAPI_TRANSFER_PARAMS_FAST` - Does not check the address is valid and will only break up data to use DMA. An error will be returned if breaking up the data does not solve the alignment issue. The processor will not be halted and the cache will not be flushed. You must ensure the processor is not accessing mono memory when this type of transfer is performed from the host, and the processor must flush its cache before the transfer (see the *[6]: SDK Reference Manual*).

## Asynchronous transfers

CSAPI provides two asynchronous memory transfer functions that allow one read operation and one write operation to be carried out in the background. The asynchronous transfers are started by calling `CSAPI_read_mono_memory_async` or `CSAPI_write_mono_memory_async`. These functions will return immediately, and a second call to the same function will block until one of the associated acknowledgement functions has been called. Therefore, a single-threaded application must acknowledge each asynchronous transfer before starting another in the same direction, and a multithreaded application must ensure that each asynchronous transfer will be acknowledged as soon as possible to avoid delaying the next asynchronous transfer.

There are two functions to acknowledge when an asynchronous transfer has completed. These are `CSAPI_read_mono_memory_async_wait` and `CSAPI_write_mono_memory_async_wait`. These functions will block until the corresponding read or write operation has completed, or until the given timeout period has elapsed. When these functions return successfully, another call to `CSAPI_read_mono_memory_async` or `CSAPI_write_mono_memory_async` can be made without blocking.

The defined value `CSAPI_NO_TIMEOUT` can used for the `timeout_ms` parameter to wait indefinitely. You can use a value of 0 for the `timeout_ms` parameter to poll the current status of the transfer. In this case the `CSAPI_read_mono_memory_async_wait` or `CSAPI_write_mono_memory_async_wait` function will return immediately. If the return code is `CSAPIErrno_success` then the transfer completed successfully. If the return code is `CSAPIErrno_timeout` then the transfer is still in progress. Any other return code indicates that the transfer failed. Note that the functions will return `CSAPIErrno_success` only once for each transfer. Subsequent calls will begin waiting on the next transfer, even if the transfer has not started yet.

#### Semaphore controlled transfers

The transfer parameters for the CSAPI memory transfer functions allow the DMA engine to be linked to one or two GSU semaphores. (See *Section 3.4.6: Semaphores*). The semaphore handles must be stored in the `start_sem` and `complete_sem` members of the transfer parameters structure. The transfer will start when the `start_sem` semaphore is signalled, and when the transfer completes the DMA engine will signal the `complete_sem` semaphore.

Linking the DMA engine to GSU semaphores can be used to remove the latency between sending data to the card and starting the compute cycle on the card. It can also remove the delay between completing the compute cycle and transferring the results back to the host.

### 3.4.10 Register access

The following functions are used to access control registers in the CSX processors:

- `CSAPI_read_control_register`
- `CSAPI_write_control_register`
- `CSAPI_read_control_register_raw`
- `CSAPI_write_control_register_raw`

These functions are provided for advanced debugging and are not needed in normal use. The 'raw' functions use an absolute register address. The other functions calculate the register address for the specified processor.

### 3.4.11 System parameters

The `CSAPI_get_system_param` and `CSAPI_set_system_param` functions provide access to a number of system parameters that can be modified at runtime. Most of these parameters can also be modified by setting an environment variable (see *Chapter 5: Runtime environment variables on page 85*). The parameter types are listed in *Section 3.7.14: Enumerated types on page 73*.

The following parameter types have an immediate effect:

- CSP_Verbosity
- CSP_LogLevel
- CSP_DmaReadThreshold
- CSP_DmaWriteThreshold
- CSP_OptimumDmaRead
- CSP_OptimumDmaWrite

The `CSP_OptimumDmaRead` and `CSP_OptimumDmaWrite` types will cause the given value to be written to a file and used for all future connections (see *Section 3.9: DMA issues on page 83*). The `CSP_DmaReadThreshold` and `CSP_DmaWriteThreshold` types should be used to affect the current DMA threshold.

The `CSP_ZeroBss` parameter will have an effect the next time `CSAPI_load` is called.

*Note: The compiler expects the BSS sections to be cleared to zero, so Cn programs may not run correctly if this parameter is set to 0 (off).*

The other parameter types will not have an effect until `CSAPI_reset` is called. During reset these parameters are applied to the card. The default values for these parameters are optimized for the connected card. Changing these values is not recommended and may result in undefined behavior.

### 3.4.12 Features

The `CSAPI_feature` function provides information about the features of the card or processor. The function takes the address of an integer, which will be set to a value corresponding to the type of feature requested. The feature types are listed in *Section 3.7.14: Enumerated types on page 73*. The values for the feature types are determined by the card type, which is known after `CSAPI_connect` has been called. The result for processor 0 can be used for other processors on the card if they are known to be of the same type.

The `CSAPI_feature` function should be used to check for the availability of features on different Advance cards. For example, the DMA engine on an e620 card can be linked to GSU semaphores, while the DMA engine on an X620 card cannot. If the host program is written to support cards with and without this feature, it should use the `CSAPI_feature` function instead of using the card type. This will make the program forwards compatible with other cards that may or may not support this feature.

The `CSF_NumDmaChannels` type will return the number of DMA channels that can be used at the same time. The PCI Express bus can support full data rates in both read and write directions at the same time. The host program must either perform these transfers from separate threads or use the asynchronous memory transfer functions (see *Memory and register access on page 24*). Two transfers in the same direction will share the available bandwidth, so each transfer will take longer. However, this can still be useful when performing small transfers while a large transfer is in progress.

The `CSF_DmaReadThreshold` and `CSF_DmaWriteThreshold` types have been replaced by the `CSP_DmaReadThreshold` and `CSP_DmaWriteThreshold` types used with the `CSAPI_get_system_param` function. The `CSF_DmaReadThreshold` and `CSF_DmaWriteThreshold` types must not be used.

### 3.4.13 Status

The `CSAPI_status` function provides information about the status of the card or processor. The function takes the address of a `CSAPIStatus` structure, and populates the string member and either the integer or float arrays, depending on the type of status requested. The status types are listed in *Section 3.7.14: Enumerated types on page 73*. The `CSU_ProcCoreFrequency` and `CSU_ProcTemperature` status types apply to a processor, identified by the `proc_inx` parameter. The other status types apply to the card, the FPGA or they return a value for each processor.

The returned values are the result of a query or measurement that is performed when the `CSAPI_status` function is called. The `CSU_ProcCoreFrequency`, `CSU_FindOptimumDmaRead` and `CSU_FindOptimumDmaWrite` status types result in a measurement, which can take a few hundred milliseconds. The other status types perform a query to the card or PCI configuration, which is relatively quick.

## 3.5 Examples

### 3.5.1 Example of a host program

A simple example of a host program is presented below. It shows how to create a CSAPI state object and connect it to a card. It will then reset processor zero, load and run a program and wait for the program to terminate. Finally it will obtain the exit code.

```
void main()
{
   CSAPIErrno ret;
   struct CSAPIState *s;
   struct CSAPIProcess *process;
   int csx_exit_code;

   s = CSAPI_new();

   ret = CSAPI_connect( s, CSH_Private, CSC_Direct, “localhost”,
CSAPI_INSTANCE_ANY, 0 );

   ret = CSAPI_reset( s, 0, CSR_FullReset, CSAPI_NO_TIMEOUT );

   ret = CSAPI_load( s, 0, CSX_FILE_NAME, NULL, &process,
CSAPI_NO_TIMEOUT );

   ret = CSAPI_run( s, process, NULL );

   ret = CSAPI_wait_on_terminate( s, process, CSAPI_NO_TIMEOUT );

   ret = CSAPI_get_return_value( s, process, &csx_exit_code );

   CSAPI_delete( s );
}
```

The return code from each CSAPI call is not checked in this simple example. For more complete examples refer to the source code provided as part of the SDK installation.

### 3.5.2 Example of a host and CSX code cooperation

A simple example of both the host and the CSX600 processor **C<sup>n</sup>** code is presented below. It shows how to designate an area of mono memory and synchronize host and card processing.

#### Host code:

```
#include <stdlib.h>
#include "csapi.h"

int main( int argc, char *argv[])
{
CSAPIErrno ret;
struct CSAPIState *s;
struct CSAPIProcess *process;
unsigned int proc_inx = 0;

struct CSAPISemaphore *sem_start_processing, *sem_processing_done;
CSAPIMemoryAddress problem_buffer, result_buffer, address;
unsigned int buffer_length = 1024;
char problem[1024], result[1024];

s = CSAPI_new();

ret = CSAPI_connect( s, CSH_Private, CSC_Direct, "localhost",
CSAPI_INSTANCE_ANY, 0 );

ret = CSAPI_load( s, proc_inx, "interaction.csx", NULL, &process,
CSAPI_NO_TIMEOUT );

ret = CSAPI_allocate_shared_semaphore( s, proc_inx, CST_Tsc, process,
"sem_start_processing", &sem_start_processing );
ret = CSAPI_allocate_shared_semaphore( s, proc_inx, CST_Tsc, process,
"sem_processing_done", &sem_processing_done );

ret = CSAPI_allocate_shared_memory( s, proc_inx, CSM_Dram, buffer_length,
8, process, "problem_buffer", &problem_buffer );
ret = CSAPI_allocate_shared_memory( s, proc_inx, CSM_Dram, buffer_length,
8, process, "result_buffer", &result_buffer );

// Write the buffer length to the variable in the csx program
ret = CSAPI_get_symbol_value( s, process, "buffer_length", &address );
ret = CSAPI_write_mono_memory( s, CSAPI_TRANSFER_PARAMS_SAFE, address,
sizeof(buffer_length), &buffer_length );

ret = CSAPI_run( s, process, NULL );

// Write the problem buffer to the card and signal the semaphore
ret = CSAPI_write_mono_memory( s, CSAPI_TRANSFER_PARAMS_SAFE,
problem_buffer, buffer_length, &problem );

ret = CSAPI_semaphore_signal( s, sem_start_processing, 0 );

// Wait for the card to signal the semaphore and read the result
ret = CSAPI_semaphore_wait( s, sem_processing_done, NULL, CSAPI_NO_TIMEOUT
);

ret = CSAPI_read_mono_memory( s, CSAPI_TRANSFER_PARAMS_SAFE, result_buffer,
buffer_length, &result );

ret = CSAPI_wait_on_terminate( s, process, CSAPI_NO_TIMEOUT );

ret = CSAPI_free_memory( s, problem_buffer );
ret = CSAPI_free_memory( s, result_buffer );

ret = CSAPI_free_semaphore( s, sem_start_processing );
ret = CSAPI_free_semaphore( s, sem_processing_done );

ret = CSAPI_unload( s, process );

CSAPI_delete( s );
}
```

### Card code:

```
#include <stdio.h>
#include <lib_ext.h>

// These are used for signalling between host and card - MUST be global
short sem_start_processing;
short sem_processing_done;

unsigned int buffer_length;
char *problem_buffer;
char *result_buffer;

int main( int argc, char *argv[] )
{
int i;

// Wait for a signal from host program
printf("CARD: waiting for signal from host program before proceeding -
sem_start_processing: %d\n", sem_start_processing);
sem_wait( sem_start_processing );
printf("CARD: got signal from host program\n");

// Compute result buffer from problem buffer
// ...

// Send a signal to the host program
printf("CARD: sending signal to host program - sem_processing_done: %d\n",
sem_processing_done);
sem_sig( sem_processing_done );
printf("CARD: sent signal to host program\n");

return 0;
}
```

*Figure 2* shows the timeline for driver interactions.

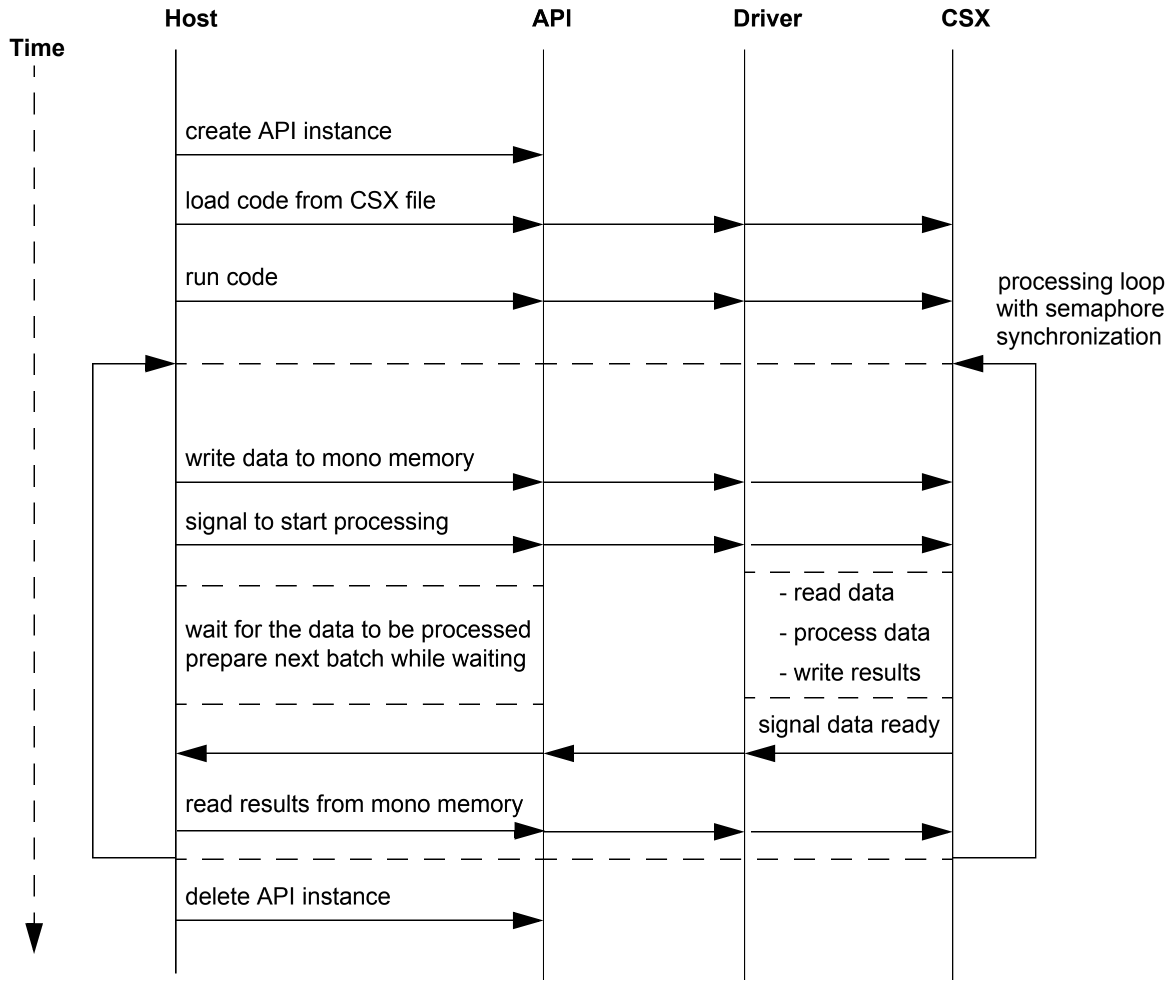
Host
API
Driver
CSX
Time
create API instance
load code from CSX file
run code
processing loop with semaphore synchronization
write data to mono memory
signal to start processing
- read data
- process data
- write results
wait for the data to be processed prepare next batch while waiting
signal data ready
read results from mono memory
delete API instance

**Figure 2. Timeline for driver interactions**

## 3.6 Calling CSAPI functions

Some CSAPI functions require a connection to the card before they can be called and others do not. If the card is in use by another user, you will not have a connection. You will only be able to call functions that do not require a connection. These can be identified by looking at the return codes - if they do not return the `CSAPIErrno_not_connected` error, they can be called before connecting to a card.

### 3.6.1 Functions that can be called before connecting to the card

The following functions can be called before a connection to a card is established and when the card is in use by another user:

`CSAPI_new`
`CSAPI_delete`
`CSAPI_num_cards`
`CSAPI_version` (for CSAPI header, Runtime package and Runtime build version types)
`CSAPI_connect`
`CSAPI_free_semaphore`
`CSAPI_get_callback`
`CSAPI_register_callback`
`CSAPI_free_memory`
`CSAPI_get_error_string`
`CSAPI_get_error_string_length`

### 3.6.2 Functions that do not communicate with the card

The following functions do not actually communicate with a card, but they relate to the connected card, so they must be called after connection:
`CSAPI_get_free_semaphores`
`CSAPI_allocate_shared_semaphore` (if `symbol_name` is `NULL`)
`CSAPI_allocate_static_shared_semaphore` (if `symbol_name` is `NULL`)
`CSAPI_get_free_memory`
`CSAPI_allocate_shared_memory` (if `symbol_name` is `NULL`)
`CSAPI_allocate_static_shared_memory` (if `symbol_name` is `NULL`)
`CSAPI_num_processors`
`CSAPI_num_pes`
`CSAPI_endianness`
`CSAPI_set_system_param`
`CSAPI_num_threads`

## 3.7 ClearSpeed host application programming interface (CSAPI)

This section describes the CSAPI. The CSAPI functions and structures are declared in the `csapi.h` header file, which includes the other CSAPI header files (except `csapi_events.h`). You only need to include `csapi.h` in your program, and if you are decoding the data passed from an event you will also need to include `csapi_events.h`.

All the CSAPI functions (except `CSAPI_delete`) return a `CSAPIErrno` code to indicate success or failure. The return codes are declared in the `csapi_errno.h` and are described in *Section 3.7.1: Error codes on page 33*.

Enumerated types are described in `csapi_types.h`, which is included by `csapi.h` (see also *Section 3.7.14: Enumerated types on page 73*).

The state of the API is held in a structure called `CSAPIState`. This structure is not defined, and you only need to declare and use a pointer to the state. The state is created by the `CSAPI_new` function and deleted by the `CSAPI_delete` function. One `CSAPIState` structure is needed for each card.

## 3.7.1 Error codes

Error codes, listed in *Table 8*, are defined in `csapi_errno.h`. An error string can be obtained by using the `CSAPI_get_error_string` function.

| DRVErrno | Description |
|---|---|
| `CSAPIErrno_success (= 0)` | No error |
| `CSAPIErrno_error (= 1)` | Generic error |
| `CSAPIErrno_failed_to_load_library` | Failed to load dynamically linked library |
| `CSAPIErrno_not_implemented` | Not implemented |
| `CSAPIErrno_not_connected` | Not connected |
| `CSAPIErrno_already_connected` | Already connected |
| `CSAPIErrno_critical_error` | Critical error |
| `CSAPIErrno_check_failed` | Check failed, reset required |
| `CSAPIErrno_fpga_upgrade_required` | FPGA is not supported. FPGA upgrade required |
| `CSAPIErrno_failed_to_read_csx_version` | Failed to read version from CSX file |
| `CSAPIErrno_incompatible_csx_version` | CSX file and driver package are incompatible |
| `CSAPIErrno_not_loaded` | No program loaded |
| `CSAPIErrno_semaphore_number_invalid` | Semaphore number invalid |
| `CSAPIErrno_semaphore_not_allocated` | Semaphore not allocated |
| `CSAPIErrno_semaphore_already_allocated` | Semaphore already allocated |
| `CSAPIErrno_no_semaphores_of_type_available` | No semaphores of type available |
| `CSAPIErrno_no_symbol` | Symbol not found |
| `CSAPIErrno_no_start_symbol` | _start symbol not found |
| `CSAPIErrno_invalid_proc_inx` | Invalid processor index |
| `CSAPIErrno_failed_to_create_thread` | Failed to create thread |
| `CSAPIErrno_failed_while_waiting_for_semaphore` | Failed while waiting for semaphore |
| `CSAPIErrno_failed_to_signal_semaphore` | Failed to signal semaphore |
| `CSAPIErrno_bad_csapi_state` | Bad CSAPIState |
| `CSAPIErrno_bad_csapi_arg` | Bad CSAPI argument |
| `CSAPIErrno_program_running` | Not available while program running |
| `CSAPIErrno_program_not_running` | Program is not running or has terminated |

**Table 8. Error codes**

| DRVErrno | Description |
|---|---|
| `CSAPIErrno_not_enough_memory` | Not enough memory |
| `CSAPIErrno_not_enough_blocks` | Reached limit on number of allocated blocks |
| `CSAPIErrno_address_in_use` | Memory address already allocated |
| `CSAPIErrno_address_not_aligned` | Memory address not aligned for DMA |
| `CSAPIErrno_invalid_address` | Memory address invalid for mem_type |
| `CSAPIErrno_invalid_memory_type` | Invalid mem_type for processor |
| `CSAPIErrno_address_not_allocated` | Address not allocated |
| `CSAPIErrno_invalid_stack_size` | Check stack declaration |
| `CSAPIErrno_string_truncated` | The returned string has been truncated |
| `CSAPIErrno_timeout` | Operation timed out |
| `CSAPIErrno_requested_resource_is_in_use` | Resource is in use by another user |
| `CSAPIErrno_no_resources_available` | All resources are in use, specify an instance to see the user |

**Table 8. Error codes (continued)**

## 3.7.2 Initialization and maintenance functions

The following CSAPI functions used for initialization and maintenance are described in this section:

- CSAPI_new *on page 34*.
- CSAPI_delete *on page 35*.
- CSAPI_num_cards *on page 35*.
- CSAPI_version *on page 36*.
- CSAPI_connect *on page 36*.
- CSAPI_disconnect *on page 37*.
- CSAPI_reset *on page 38*.

### CSAPI_new

#### Function

```
struct CSAPIState* CSAPI_new();
```

#### Description

Loads the dynamic library for the runtime. This will normally be the DIRECT library, but if the appropriate environment variable is set then the TRACE or DEBUG version of the library will be loaded. It also creates a new CSAPIState instance and returns a pointer, which is passed as a parameter to all of the CSAPI functions. This CSAPIState instance is for use with a single connection, so CSAPI_new must be called once for each card or simulator being used. This function will serialize calls internally if called simultaneously from two threads. This serialization has a performance impact, which can be avoided by making all calls to `CSAPI_new` from a single thread, or by serializing calls externally.

### Parameters

None.

### Returns

A pointer to a newly created and initialized instance of the CSAPIState structure. This will be NULL if there was a problem loading the library, so it must be checked on return.

## CSAPI_delete

### Function

```
void CSAPI_delete(
    struct CSAPIState* const s );
```

### Description

Disconnects from any connected resource, unloads the dynamic library and destroys the instance of the API state. It is not possible to call any CSAPI function with this state once it has been passed to `CSAPI_delete`.

### Parameters

`s`: State created by `CSAPI_new`. Will be destroyed by this function.

### Returns

Nothing.

## CSAPI_num_cards

### Function

```
CSAPIErrno CSAPI_num_cards(
    unsigned int* const num_of_cards );
```

### Description

Provides the number of cards in the current system. Note that this does not include simulators. This function can be called before `CSAPI_new`.

### Parameters

`num_of_cards`: Pointer to integer to be given the number of cards in the current system

### Returns

`CSAPIErrno_success, CSAPIErrno_bad_csapi_arg, CSAPIErrno_failed_to_load_library, CSAPIErrno_lldclient_error`

## CSAPI_version

### Function

```
CSAPIErrno CSAPI_version(
    struct CSAPIState* const s,
    CSAPIVersionType version_type,
    struct CSAPIVersion* version );
```

### Description

Returns the version, time stamp or serial number of the object specified by the `version_type` parameter (see *CSAPIVersionType on page 79*). The result will use the relevant members of the `CSAPIVersion` structure, which includes major, minor and string. The string member will always be given a value. This function can be called before `CSAPI_new` with a `NULL CSAPIState`, although some version types require a state and connection.

### Parameters

`s`: State created by `CSAPI_new`. May be NULL for CsapiHeader, RuntimePackage, RuntimeBuild.

`version_type`: Object to return the version for, as defined in *CSAPIVersionType on page 79*

`version`: Pointer to structure to be given the version. The string member will always be set

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_failed_to_load_library, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_error
```

## CSAPI_connect

### Function

```
CSAPIErrno CSAPI_connect(
    struct CSAPIState* const s,
    CSAPISharingType sharing_type,
    CSAPIConnectType connect_type,
     const char* host_name,
    unsigned int instance,
    unsigned int timeout_ms );
```

### Description

Connects to hardware or a simulator, depending on the `connection_type`. If the connection type is `CSC_Socket`, the target can either be local or on a host specified by `host_name`. If `CSC_EnvironmentVariable` is used then the resulting `connection_type`, `host_name` and `instance` will be determined from environment variables. The connection can either be private or shared. If a shared connection is used, all processes must use the same `sharing_type`. Connect must be called once for each `CSAPIState` created with `CSAPI_new`. If the hardware or simulator is already in use, the connection will be retried for the period given by `timeout_ms`. On Windows systems a nonzero value for the `timeout_ms` parameter will cause the current thread to be switched to the first processor. This provides a consistent time reference for retrying the connection. If the error `CSAPIErrno_check_failed` is returned, the connection will be valid, but the card must be reset. `CSAPI_check` can be called to identify the problem. Disconnect by calling `CSAPI_disconnect` or by deleting the state with `CSAPI_delete`.

### Parameters

`s`: State created by CSAPI_new

`sharing_type`: Sharing mode, either CSH_Private, CSH_SharedUser or CSH_SharedPublic, as described in *CSAPISharingType on page 77*.

`connect_type`: Connection type, either `CSC_Direct`, `CSC_Socket`, `CSC_Usb` or `CSC_EnvironmentVariable` as described in *CSAPIConnectType on page 74*.

`host_name`: Host to connect to when `connection_type` is `CSC_Socket`. Use "localhost" for the local machine. An IP address can also be given as N.N.N.N.

`instance`: Instance number for hardware or simulator, starting from zero. Use `CSAPI_INSTANCE_ANY` to connect to any hardware available

`timeout_ms`: Period of time to retry connection when already in use. Use 0 for no retries

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_already_connected,
CSAPIErrno_failed_to_create_thread, CSAPIErrno_lldclient_error,
CSAPIErrno_requested_resource_is_in_use,
CSAPIErrno_no_resources_available, CSAPIErrno_check_failed,
CSAPIErrno_fpga_upgrade_required, CSAPIErrno_error
```

## CSAPI_disconnect

### Function

```
CSAPIErrno CSAPI_disconnect(
    struct CSAPIState* const s );
```

### Description

Disconnects from hardware or simulator without unloading the dynamic library. Once disconnected, the resource will be available for other processes to connect to. Calling `CSAPI_delete` will automatically call `CSAPI_disconnect`.

### Parameters

`s`: State created by CSAPI_new

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_not_connected, CSAPIErrno_error
```

### CSAPI_reset

#### Function

```
CSAPIErrno CSAPI_reset(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    CSAPIResetType reset_type,
    unsigned int timeout_ms );
```

#### Description

Resets the processor (MTAP core) specified by `proc_inx`. This includes resetting the DMA, GSU and PIO engine, stopping and setting up the TSC and setting up the system endianness and instruction cache. The hardware semaphores and interrupts are then initialized and the bootstrap code is run to set up the microcode. Note that some types of reset will disconnect and reconnect before completing. The function will block until the reset has completed or the period given by the `timeout_ms` parameter has elapsed. This will be indicated by returning `CSAPIErrno_timeout`. The return status must be checked to ensure that the reset completed successfully. The `CSAPI_NO_TIMEOUT` value can be used with the `timeout_ms` parameter for an infinite wait.

#### Parameters

`s`: State created by CSAPI_new

`proc_inx`: Index of processor (MTAP core) to be reset. Index from 0 upwards

`reset_type`: Use `CSR_FullSystem` to reset CCBR, CCIs, DDRs and Bus monitor without disconnecting, as described in *CSAPIResetType on page 76*.

`timeout_ms`: Timeout period in milliseconds, after which the function will return `CSAPIErrno_timeout`

#### Returns

`CSAPIErrno_success, CSAPIErrno_bad_csapi_state, CSAPIErrno_invalid_proc_inx, CSAPIErrno_not_connected, CSAPIErrno_critical_error, CSAPIErrno_timeout, CSAPIErrno_error`

## 3.7.3 Program setup

The following CSAPI functions used for program setup are described in this section:

- CSAPI_load *on page 38*.
- CSAPI_unload *on page 40*.

### CSAPI_load

#### Function

```
CSAPIErrno CSAPI_load(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    const char* prog_name,
    const char* section_names,
    struct CSAPIProcess** process,
    unsigned int timeout_ms );
```

### Description

Loads the executable code from a `.csx` file onto a specified processor. Two searches are made for `prog_name`. The first search is relative to the current location. The second search prefixes the file name with each entry in the `CSPATH` environment variable. This is not done if the file name starts with '.', '/' or a Microsoft Windows drive specification such as 'C:\'. The `.csx` extension is not required as part of `prog_name` - the search will be repeated with `.csx` added to the end of the given string. The `section_names` parameter can be used to specify sections to be loaded to ESRAM. All other sections will be loaded to DRAM. If the `section_names` parameter is NULL then only the `.text` section will be loaded to ESRAM. If the string is "ALL" then all sections will be loaded to ESRAM and an error will be returned if this is not possible. If the string is "ANY" then as many sections as possible will be loaded to ESRAM. Any other string will be interpreted as a comma separated list of section names to be loaded to ESRAM. Any sections not in the list will be loaded to DRAM. The `section_names` parameter must be NULL if the csx file is statically linked. The `process` parameter will be given a handle to the loaded process. This will be needed for other CSAPI function calls. If the same csx file has already been loaded, the process handle for that instance will be returned, and the program sections will be reloaded from host memory.The function will block until the load has completed or the period given by the `timeout_ms` parameter has elapsed. This will be indicated by returning `CSAPIErrno_timeout`. The return status must be checked to ensure that the load completed successfully. The `CSAPI_NO_TIMEOUT` value can be used with the `timeout_ms` parameter for an infinite wait.

### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor to load application to. Index from 0 upwards

`prog_name`: Specifies the path to the CSX executable, which can be relative, absolute or dependent on the CSPATH environment variable. See the function description for details

`section_names`: Specifies which section names you want to be loaded into the ESRAM

`process`: Pointer to variable to be given a pointer to the loaded process

`timeout_ms`: Timeout period in milliseconds, after which the function will return `CSAPIErrno_timeout`

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_bad_csapi_arg,
CSAPIErrno_not_connected, CSAPIErrno_critical_error,
CSAPIErrno_program_running, CSAPIErrno_locatefile_error,
CSAPIErrno_timeout, CSAPIErrno_check_failed,
CSAPIErrno_failed_to_read_csx_version,
CSAPIErrno_incompatible_csx_version, CSAPIErrno_error
CSAPIErrno_invalid_stack_size
```

### CSAPI_unload

#### Function

```
CSAPIErrno CSAPI_unload(
    struct CSAPIState* const s,
    struct CSAPIProcess* process );
```

#### Description

Unloads the specified process and releases the memory allocated for the program.

#### Parameters

`s`: State created by `CSAPI_new`

`process`: Pointer to the process to be unloaded

#### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_not_loaded, CSAPIErrno_program_running,
CSAPIErrno_address_not_allocated, CSAPIErrno_error,
```

## 3.7.4 Processor control

The following CSAPI functions which control the processor are described in this section:

- CSAPI_run *on page 40*.
- CSAPI_halt *on page 41*.
- CSAPI_start *on page 41*.
- CSAPI_wait_on_terminate *on page 42*.
- CSAPI_get_return_value *on page 42*.

### CSAPI_run

#### Function

```
CSAPIErrno CSAPI_run(
    struct CSAPIState* const s,
    struct CSAPIProcess* process,
    const char* csx_args );
```

#### Description

Executes a loaded process. The process will be executed on the processor that it was loaded on. If the `csx_args` parameter is not NULL, it will be split where there are spaces and the resulting list will be passed to the main function in the csx file, with the csx filename as the first argument. If the `csx_args` parameter is NULL, no parameters will be passed to the main function in the csx file (`argc` will be zero).

### Parameters

`s`: State created by `CSAPI_new`

`process`: Pointer to the process to execute

`csx_args`: White space separated list of options for `argc/argv` parameters of `main()` in csx file

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_not_connected, CSAPIErrno_critical_error,
CSAPIErrno_not_loaded, CSAPIErrno_no_start_symbol, CSAPIErrno_error
```

## CSAPI_halt

### Function

```
CSAPIErrno CSAPI_halt(
    struct CSAPIState* const s,
    unsigned int proc_inx );
```

### Description

Halts the execution of a program running on the specified processor.

### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor to stop running. Index from 0 upwards

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_error
```

## CSAPI_start

### Function

```
CSAPIErrno CSAPI_start(
    struct CSAPIState* const s,
    unsigned int proc_inx );
```

### Description

Restarts the execution of a halted program.

### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor to start (restart) running. Index from 0 upwards

### Returns

`CSAPIErrno_success`, `CSAPIErrno_bad_csapi_state`, `CSAPIErrno_invalid_proc_inx`, `CSAPIErrno_not_connected`, `CSAPIErrno_critical_error`, `CSAPIErrno_program_not_running`, `CSAPIErrno_error`

## CSAPI_wait_on_terminate

### Function

```
CSAPIErrno CSAPI_wait_on_terminate(
    struct CSAPIState* const s,
    struct CSAPIProcess* process,
    unsigned int timeout_ms );
```

### Description

Awaits the termination signal from the processor on which the given process has been loaded. The function will block until the program has terminated or the period given by the `timeout_ms` parameter has elapsed. This will be indicated by returning `CSAPIErrno_timeout`. The return status must be checked to ensure that the program terminated successfully. The `CSAPI_NO_TIMEOUT` value can be used with the `timeout_ms` parameter for an infinite wait.

### Parameters

`s`: State created by `CSAPI_new`

`process`: Pointer to the process that defines the processor to to wait on for termination

`timeout_ms`: Timeout period in milliseconds, after which the function will return `CSAPIErrno_timeout`

### Returns

`CSAPIErrno_success`, `CSAPIErrno_bad_csapi_state`, `CSAPIErrno_bad_csapi_arg`, `CSAPIErrno_not_connected`, `CSAPIErrno_critical_error`, `CSAPIErrno_not_loaded`,

`CSAPIErrno_failed_while_waiting_for_semaphore`, `CSAPIErrno_timeout`, `CSAPIErrno_error`

## CSAPI_get_return_value

### Function

```
CSAPIErrno CSAPI_get_return_value(
    struct CSAPIState* const s,
    struct CSAPIProcess* process,
    int* const return_value );
```

### Description

Obtains the return status from the processor on which the given process has been loaded. This can be called after `CSAPI_wait_on_terminate`. If the function is not successful, `return_value` is undefined.

### Parameters

`s`: State created by `CSAPI_new`

`process`: Pointer to the process that defines the processor to get the return value from

`return_value`: Pointer to integer to be given the return value from the terminated CSX program

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_bad_csapi_arg,
CSAPIErrno_not_connected, CSAPIErrno_critical_error,
CSAPIErrno_not_loaded
```

## 3.7.5 Accessing registers

The following CSAPI functions for accessing registers are described in this section:

- CSAPI_write_control_register *on page 43*.
- CSAPI_write_control_register_raw *on page 44*.
- CSAPI_read_control_register *on page 44*.
- CSAPI_read_control_register_raw *on page 45*.

### CSAPI_write_control_register

#### Function

```
CSAPIErrno CSAPI_write_control_register(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    CSAPIRegisterAddress reg_addr,
    CSAPIRegisterValue value );
```

#### Description

Writes the given value to a control register at address `reg_addr` in the processor specified by `proc_inx`. Use of this function requires detailed knowledge of the architecture and the function of the control registers. See the architecture manual for details.

#### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor to write control register on. Index from 0 upwards

`reg_addr`: Address of control register to write to `value`: New value to be written to the control register

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_error
```

## CSAPI_write_control_register_raw

### Function

```
CSAPIErrno CSAPI_write_control_register_raw(
    struct CSAPIState* const s,
    CSAPIRegisterAddress reg_addr,
    CSAPIRegisterValue value );
```

### Description

Same as `CSAPI_write_control_register`, except that the `reg_addr` must be shifted for the appropriate processor index before calling this function. This allows for a more efficient implementation. See the architecture manual for details.

### Parameters

`s`: State created by `CSAPI_new`

`reg_addr`: Address of control register to write to

`value`: New value to be written to the control register

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_not_connected, CSAPIErrno_critical_error,
CSAPIErrno_erro
```

## CSAPI_read_control_register

### Function

```
CSAPIErrno CSAPI_read_control_register(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    CSAPIRegisterAddress reg_addr,
    CSAPIRegisterValue* const value );
```

### Description

Reads the control register at address `reg_addr` from the processor specified by `proc_inx` into the memory location pointed to by value. If the function is not successful, the value is undefined. Use of this function requires detailed knowledge of the architecture and the function of the control registers. See the architecture manual for details.

#### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor to read control register on. Index from 0 upwards

`reg_addr`: Address of control register to read from

`value`: Pointer to memory to be given the value read from the control register

#### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_bad_csapi_arg,
CSAPIErrno_not_connected, CSAPIErrno_critical_error,
CSAPIErrno_error
```

### CSAPI_read_control_register_raw

#### Function

```
CSAPIErrno CSAPI_read_control_register_raw(
    struct CSAPIState* const s,
    CSAPIRegisterAddress reg_addr,
    CSAPIRegisterValue* const value );
```

#### Description

Same as `CSAPI_read_control_register`, except that the `reg_addr` must be shifted for the appropriate processor index before calling this function. This allows for a more efficient implementation. See the architecture manual for details.

#### Parameters

`s`: State created by `CSAPI_new`

`reg_addr`: Address of control register to read from

`value`: Pointer to memory to be given the value read from the control register

#### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_error
```

## 3.7.6 Accessing mono memory

The following CSAPI functions for accessing mono memory are described in this section:

- CSAPI_write_mono_memory *on page 46*.
- CSAPI_read_mono_memory *on page 47*.
- CSAPI_write_mono_memory_async *on page 48*.
- CSAPI_write_mono_memory_async_wait *on page 48*.
- CSAPI_read_mono_memory_async *on page 50*.
- CSAPI_read_mono_memory_async_wait *on page 50*.

## CSAPI_write_mono_memory

### Function

```
CSAPIErrno CSAPI_write_mono_memory(
    struct CSAPIState* const s,
    struct CSAPITransferParam* tp,
    CSAPIMemoryAddress address,
    CSAPIMemorySize size,
    const void* data_in );
```

### Description

Copies data from a buffer on the host pointed to by `data_in`, to a location in mono memory (DRAM or ESRAM) given by `address`. The `size` is specified in bytes. The transfer parameters control the following (the affected processor is identified by the `address` parameter):

- Halting the processor during the transfer. This can avoid the slow down caused by memory page conflicts.
- Flushing and invalidating the cache on the processor before the transfer. - Triggering the transfer from a GSU semaphore and signalling a GSU semaphore on transfer completion.
- Checking the validity of the start address and the alignment of the host buffer and address. If the buffers are not aligned and `CSAPI_TRANSFER_PARAMS_FAST` is used then an error will be returned. If `CSAPI_TRANSFER_PARAMS_SAFE` is used then the data will be bounced via an aligned buffer to maximize transfer speed. If the transfer size is zero then the address check, alignment check, halt and flush operations will still be performed (if set by the transfer parameters). The function will block until the transfer has completed.

### Parameters

`s`: State created by `CSAPI_new`

`tp`: Transfer parameters. For example `CSAPI_TRANSFER_PARAMS_SAFE` or `CSAPI_TRANSFER_PARAMS_FAST`

`address`: Address of mono memory (DRAM or ESRAM) to start writing to

`size`: Number of bytes to copy from `data_in` (on host) to address (on card)

`data_in`: Address on host to start copying data from (used to identify processor)

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_address_not_aligned,
CSAPIErrno_invalid_address, CSAPIErrno_address_not_allocated,
CSAPIErrno_error
```

## CSAPI_read_mono_memory

### Function

```
CSAPIErrno CSAPI_read_mono_memory(
    struct CSAPIState* const s,
    struct CSAPITransferParam* tp,
    CSAPIMemoryAddress address,
    CSAPIMemorySize size,
    void* const data_out );
```

### Description

Copies data from a location in mono memory (DRAM or ESRAM) given by `address`, to a buffer on the host pointed to by `data_out`, which must be allocated on the host before calling this function. The `size` is specified in bytes. The transfer parameters control the following (the affected processor is identified by the `address` parameter):

- Halting the processor during the transfer. This can avoid the slow down caused by memory page conflicts.
- Flushing and invalidating the cache on the processor before the transfer. - Triggering the transfer from a GSU semaphore and signalling a GSU semaphore on transfer completion.
- Checking the validity of the start address and the alignment of the host buffer and address. If the buffers are not aligned and `CSAPI_TRANSFER_PARAMS_FAST` is used then an error will be returned. If `CSAPI_TRANSFER_PARAMS_SAFE` is used then the data will be bounced via an aligned buffer to maximize transfer speed. If the transfer size is zero then the address check, alignment check, halt and flush operations will still be performed (if set by the transfer parameters). The function will block until the transfer has completed.

### Parameters

`s`: State created by CSAPI_new

`tp`: Transfer parameters. For example, `CSAPI_TRANSFER_PARAMS_SAFE` or `CSAPI_TRANSFER_PARAMS_FAST`

`address`: Address of mono memory (DRAM or ESRAM) to start reading from

`size`: Number of bytes to copy from address (on card) to data_out (on host)

`data_out`: Pre-allocated address on host to start copying data to (used to identify processor)

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_address_not_aligned,
CSAPIErrno_invalid_address, CSAPIErrno_address_not_allocated,
CSAPIErrno_error
```

## CSAPI_write_mono_memory_async

### Function

```
CSAPIErrno CSAPI_write_mono_memory_async(
    struct CSAPIState* const s,
    struct CSAPITransferParam* tp,
    CSAPIMemoryAddress address,
    CSAPIMemorySize size,
    const void* data_in );
```

### Description

Calls the `CSAPI_write_mono_memory` function in a separate thread and returns once the transfer is in progress. The host is then free to continue in the current thread. The host can wait for the transfer to complete by using the `CSAPI_write_mono_memory_async_wait` function.  A second call to the `CSAPI_write_mono_memory_async` function will block until the previous asynchronous transfer has completed and the wait function has been called. There is a small overhead in waiting for asynchronous transfers to complete, so for small transfers it is more efficient to call `CSAPI_write_mono_memory` instead of `CSAPI_write_mono_memory_async` and `CSAPI_write_mono_memory_async_wait`.

### Parameters

`s`: State created by `CSAPI_new`

`tp`: Transfer parameters. For example, `CSAPI_TRANSFER_PARAMS_SAFE` or `CSAPI_TRANSFER_PARAMS_FAST`

`address`: Address of mono memory (DRAM or ESRAM) to start writing to

`size`: Number of bytes to copy from data_in (on host) to address (on card)

`data_in`: Address on host to start copying data from (used to identify processor)

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error,
CSAPIErrno_failed_while_waiting_for_semaphore,
CSAPIErrno_failed_to_signal_semaphore,
CSAPIErrno_address_not_aligned, CSAPIErrno_invalid_address,
CSAPIErrno_address_not_allocated, CSAPIErrno_error
```

## CSAPI_write_mono_memory_async_wait

### Function

```
CSAPIErrno CSAPI_write_mono_memory_async_wait(
    struct CSAPIState* const s,
    unsigned int timeout_ms );
```

### Description

Allows the host to wait for the completion of an asynchronous transfer started by `CSAPI_write_mono_memory_async`. The function will block until the transfer has completed or the period given by the timeout_ms parameter has elapsed. This will be indicated by returning `CSAPIErrno_timeout`. The return status must be checked to ensure that the transfer completed successfully. The `CSAPI_NO_TIMEOUT` value can be used with the `timeout_ms` parameter for an infinite wait and a value of zero can be used to poll the status of the transfer, where the return status will be `CSAPIErrno_timeout` if the transfer is still in progress. There is a small overhead in waiting for asynchronous transfers to complete, so for small transfers it is more efficient to call `CSAPI_write_mono_memory` instead of `CSAPI_write_mono_memory_async` followed by `CSAPI_write_mono_memory_async_wait`.

### Parameters

`s`: State created by `CSAPI_new`

`timeout_ms`: Timeout period in milliseconds, after which the function will return `CSAPIErrno_timeout`

### Returns

`CSAPIErrno_success`, `CSAPIErrno_bad_csapi_state`, `CSAPIErrno_invalid_proc_inx`, `CSAPIErrno_bad_csapi_arg`, `CSAPIErrno_not_connected`, `CSAPIErrno_critical_error`, `CSAPIErrno_failed_while_waiting_for_semaphore`, `CSAPIErrno_failed_to_signal_semaphore`, `CSAPIErrno_invalid_address`, `CSAPIErrno_address_not_allocated`, `CSAPIErrno_timeout`, `CSAPIErrno_error`

## CSAPI_read_mono_memory_async

### Function

```
CSAPIErrno CSAPI_read_mono_memory_async(
    struct CSAPIState* const s,
    struct CSAPITransferParam* tp,
    CSAPIMemoryAddress address,
    CSAPIMemorySize size,
    void* const data_out );
```

### Description

Calls the `CSAPI_read_mono_memory` function in a separate thread and returns once the transfer is in progress. The host is then free to continue in the current thread. The host can wait for the transfer to complete by using the `CSAPI_read_mono_memory_async_wait` function. A second call to the `CSAPI_read_mono_memory_async` function will block until the previous asynchronous transfer has completed and the wait function has been called. There is a small overhead in waiting for asynchronous transfers to complete, so for small transfers it is more efficient to call `CSAPI_read_mono_memory` instead of `CSAPI_read_mono_memory_async` followed by `CSAPI_read_mono_memory_async_wait`.

### Parameters

`s`: State created by `CSAPI_new`.

`tp`: Transfer parameters. For example, `CSAPI_TRANSFER_PARAMS_SAFE` or `CSAPI_TRANSFER_PARAMS_FAST`.

`address`: Address of mono memory (DRAM or ESRAM) to start reading from.

`size`: Number of bytes to copy from address (on card) to `data_out` (on host).

`data_out`: Pre-allocated address on host to start copying data to (used to identify processor).

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error,
CSAPIErrno_failed_while_waiting_for_semaphore,
CSAPIErrno_failed_to_signal_semaphore,
CSAPIErrno_address_not_aligned, CSAPIErrno_invalid_address,
CSAPIErrno_address_not_allocated, CSAPIErrno_error
```

## CSAPI_read_mono_memory_async_wait

### Function

```
CSAPIErrno CSAPI_read_mono_memory_async_wait(
    struct CSAPIState* const s,
    unsigned int timeout_ms );
```

### Description

Allows the host to wait for the completion of an asynchronous transfer started by `CSAPI_read_mono_memory_async`. The function will block until the transfer has completed or the period given by the timeout_ms parameter has elapsed. This will be indicated by returning `CSAPIErrno_timeout`. The return status must be checked to ensure that the transfer completed successfully. The `CSAPI_NO_TIMEOUT` value can be used with the `timeout_ms` parameter for an infinite wait and a value of zero can be used to poll the status of the transfer, where the return status will be `CSAPIErrno_timeout` if the transfer is still in progress. There is a small overhead in waiting for asynchronous transfers to complete, so for small transfers it is more efficient to call `CSAPI_read_mono_memory` instead of `CSAPI_read_mono_memory_async` followed by `CSAPI_read_mono_memory_async_wait`.

### Parameters

`s`: State created by `CSAPI_new`.

`timeout_ms`: Timeout period in milliseconds, after which the function will return `CSAPIErrno_timeout`.

### Returns

`CSAPIErrno_success, CSAPIErrno_bad_csapi_state, CSAPIErrno_invalid_proc_inx, CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected, CSAPIErrno_critical_error, CSAPIErrno_failed_while_waiting_for_semaphore, CSAPIErrno_failed_to_signal_semaphore, CSAPIErrno_invalid_address, CSAPIErrno_address_not_allocated, CSAPIErrno_timeout, CSAPIErrno_error`

## 3.7.7 Endian functions

The following CSAPI endian functions are described in this section:

- CSAPI_buffer_to_native_endian *on page 51*.
- CSAPI_endianness *on page 52*.

### CSAPI_buffer_to_native_endian

### Function

```
CSAPIErrno CSAPI_buffer_to_native_endian(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    void* out,
    const void* in,
    unsigned int size );
```

### Description

Copies and converts data from the endianness used by the processor `proc_inx` in the current connection to the native host endianness. The number of bytes to copy and convert from in to out is set by the size parameter.

#### Parameters

`s`: State created by `CSAPI_new`. Required to ensure the dynamic CSAPI library is loaded.

`proc_inx`: Index of processor to convert endianness from. Index from 0 upwards.

`out`: Pointer to memory to write out going data to, in native host endianness.

`in`: Pointer to memory to read incoming data from, in the selected processor endianness.

`size`: Number of bytes to copy and convert.

### CSAPI_endianness

#### Function

```
CSAPIErrno CSAPI_endianness(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    CSAPIEndianType* const endian );
```

#### Description

Gets the current endianness of the specified processor on the current card.

#### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor get endianness of. Index from 0 upwards

`endian`: Pointer to variable to be given the current endianness of the specified processor on the current card. Value will be set to either `CSN_LittleEndian` or `CSN_BigEndian`, as described in *CSAPIEndianType on page 75*.

#### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_bad_csapi_arg,
CSAPIErrno_not_connected, CSAPIErrno_critical_error
```

## 3.7.8 System parameter control

The following CSAPI functions for controlling system parameters are described in this section:

- CSAPI_set_system_param *on page 53*.
- CSAPI_get_system_param *on page 53*.

## CSAPI_set_system_param

### Function

```
CSAPIErrno CSAPI_set_system_param(
    struct CSAPIState* const s,
    CSAPISystemParam what,
    unsigned int vi,
    const char* vs );
```

### Description

Sets the specified system parameter (using the `CSAPISystemParam` enumeration) to a value given to either the `vi` or `vs` parameter as appropriate. Some of the parameters are only applied during reset. See the `CSAPISystemParam` enumeration to see which parameters use which input variable, and which parameters are applied during reset. See *Section 3.4.11: System parameters on page 26* for valid combinations of system parameters.

### Parameters

`s`: State created by `CSAPI_new`

`what`: Name of parameter to be set, defined by an enumeration

`vi`: Integer value to set the parameter with (if applicable)

`vs`: String value to set the parameter with (if applicable)

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_error
```

## CSAPI_get_system_param

### Function

```
CSAPIErrno CSAPI_get_system_param(
    struct CSAPIState* const s,
    CSAPISystemParam what,
    unsigned int* vi );
```

### Description

Gets the value of the specified system parameter using the `CSAPISystemParam` enumeration. Some of the parameters are only applied during reset. See the `CSAPISystemParam` enumeration to see which parameters return the value currently in use and which parameters return the value that will be applied during the next call to `CSAPI_reset`. See *Section 3.4.11: System parameters on page 26*.

### Parameters

`s`: State created by `CSAPI_new`

`what`: Name of parameter to get, defined by an enumeration `vi`: Pointer to integer to be given the current value

#### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_error
```

### 3.7.9 Thread functions

The following CSAPI thread functions are described in this section:

- CSAPI_set_thread *on page 54*.
- CSAPI_num_threads *on page 54*.

### CSAPI_set_thread

#### Function

```
CSAPIErrno CSAPI_set_thread(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    unsigned int thread_id,
    unsigned int* const old_thread_id );
```

#### Description

Switches to the specified thread on specified processor, identified by thread ID. The previous thread ID is returned in `old_thread_id`, if it is not NULL, so that we can switch back to it if necessary.

#### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor to set thread on. Index from 0 upwards

`thread_id`: ID of thread to switch to on processor

`old_thread_id`: Pointer to integer to be given the ID of thread that was running on processor  Use NULL if this is not required

#### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_error
```

### CSAPI_num_threads

#### Function

```
CSAPIErrno CSAPI_num_threads(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    unsigned int* const num_of_threads );
```

#### Description

Provides the total number of threads supported by the specified processor. Note that this will include threads used by the debugger or other system applications.

#### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor to count threads on. Index from 0 upwards

`num_of_threads`: Pointer to integer to be given the number of threads supported by the processor

#### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_bad_csapi_arg,
CSAPIErrno_not_connected, CSAPIErrno_critical_error
```

## 3.7.10 Semaphore handling

The following CSAPI functions for handling semaphores are described in this section:

- CSAPI_get_free_semaphores *on page 55*.
- CSAPI_allocate_shared_semaphore *on page 56*.
- CSAPI_allocate_static_shared_semaphore *on page 57*.
- CSAPI_allocate_duplicate_shared_semaphore *on page 58*.
- CSAPI_semaphore_wait *on page 59*.
- CSAPI_semaphore_signal *on page 59*.
- CSAPI_semaphore_get *on page 60*.
- CSAPI_semaphore_put *on page 61*.
- CSAPI_free_semaphore *on page 61*.

### CSAPI_get_free_semaphores

#### Function

```
CSAPIErrno CSAPI_get_free_semaphores(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    CSAPISemaphoreType sem_type,
    unsigned int* num_of_semaphores );
```

#### Description

Returns the number of available semaphores of the specified type on the specified processor. Note that the number of available GSU semaphores will be the same for both the `CST_GsuWithoutData` and `CST_GsuWithData` types.

#### Parameters

`s`: State created by CSAPI_new

`proc_inx`: Index of processor to count available semaphores on. Index from 0 upwards sem_type: Semaphore type to count. CST_Tsc, CST_GsuWithoutData, CST_GsuWithData as described in *CSAPISemaphoreType on page 76* .

num_of_semaphores: Pointer to integer to be given number of available semaphores of type on processor

### Returns

CSAPIErrno_success, CSAPIErrno_bad_csapi_state, CSAPIErrno_invalid_proc_inx, CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected, CSAPIErrno_critical_error

## CSAPI_allocate_shared_semaphore

### Function

```
CSAPIErrno CSAPI_allocate_shared_semaphore(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    CSAPISemaphoreType sem_type,
    struct CSAPIProcess* process,
    const char* symbol_name,
    struct CSAPISemaphore** semaphore );
```

### Description

Allocates a semaphore of the specified type on the specified processor for use by both the CSX program and the host. The semaphore type can be either TSC or GSU. The TSC semaphores are typically used within a processor or between the host and processor. There are 128 TSC semaphores per processor and they can be signalled up to 255 times. The GSU semaphores are typically used between multiple processors, the host, the PIO unit and the DMA unit. There are 16 GSU semaphores per processor and they can be signalled up to 65536 times without data or 32 times with data. The allocated semaphore will be given to the variable specified by symbol_name, if it is not NULL, in the CSX program specified by the process parameter. The same semaphore will be returned in the semaphore parameter. If symbol_name is NULL, the process parameter must also be NULL. The semaphore signalled state will be reset to zero (that is, not signalled) when it is allocated.

### Parameters

s: State created by CSAPI_new

proc_inx: Index of processor allocate semaphore on. Index from 0 upwards

sem_type: Semaphore type to allocate. CST_Tsc, CST_GsuWithoutData, CST_GsuWithData as described in *CSAPISemaphoreType on page 76* .

process: Process handle for loaded CSX program. Use NULL if symbol_name is NULL (that is, unused)

symbol_name: Name of static global variable in the CSX program to be given the allocated semaphore

semaphore: Pointer to semaphore handle to be given the allocated semaphore

### Returns

`CSAPIErrno_success`, `CSAPIErrno_bad_csapi_state`, `CSAPIErrno_invalid_proc_inx`, `CSAPIErrno_bad_csapi_arg`, `CSAPIErrno_not_connected`, `CSAPIErrno_critical_error`, `CSAPI_not_loaded`, `CSAPIErrno_no_symbol`, `CSAPIErrno_no_semaphores_of_type_available`, `CSAPIErrno_not_enough_memory`, `CSAPIErrno_error`

## CSAPI_allocate_static_shared_semaphore

### Function

```
CSAPIErrno CSAPI_allocate_static_shared_semaphore(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    CSAPISemaphoreType sem_type,
    unsigned int sem_number,
    struct CSAPIProcess* process,
    const char* symbol_name,
    struct CSAPISemaphore** semaphore );
```

### Description

Attempts to allocate a semaphore of the specified type on the specified processor with a semaphore number given by `sem_number`. This is the same as `CSAPI_allocate_shared_semaphore`, except that a specific semaphore is returned instead of allocating any available semaphore of the specified type. The returned error code must be checked to ensure that the requested semaphore is available. This can be useful for supporting programs where the semaphore number is hard coded in the CSX program. The allocated semaphore will be given to the variable specified by `symbol_name`, if it is not `NULL`, in the CSX program specified by the process parameter. The same semaphore will be returned in the semaphore parameter. If `symbol_name` is `NULL`, the `process` parameter must also be `NULL`. The semaphore signalled state will be reset to zero (that is, not signalled) when it is allocated.

### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor to allocate semaphore on. Index from 0 upwards

`sem_type`: Semaphore type to allocate. `CST_Tsc`, `CST_GsuWithoutData`, `CST_GsuWithData` as described in *CSAPISemaphoreType on page 76* .

`sem_number`: Semaphore number to be allocated, if possible

`process`: Process handle for loaded CSX program. Use NULL if `symbol_name` is NULL (that is, unused)

`symbol_name`: Name of static global variable in the CSX program to be given the allocated semaphore

`semaphore`: Pointer to semaphore handle to be given the allocated semaphore

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_bad_csapi_arg,
CSAPIErrno_not_connected, CSAPIErrno_critical_error,
CSAPI_not_loaded, CSAPIErrno_no_symbol,
CSAPIErrno_semaphore_number_invalid,
CSAPIErrno_semaphore_already_allocated,
CSAPIErrno_not_enough_memory, CSAPIErrno_error
```

## CSAPI_allocate_duplicate_shared_semaphore

### Function

```
CSAPIErrno CSAPI_allocate_duplicate_shared_semaphore(
    struct CSAPIState* const s,
    struct CSAPISemaphore* semaphore,
    struct CSAPIProcess* process,
    const char* symbol_name );
```

### Description

Attempts to allocate a duplicate semaphore from an existing semaphore handle. The semaphore handle must be for a semaphore allocated on another card. If the same semaphore is available on the card specified by the parameter then it will be allocated on this card too. The returned error code must be checked to ensure that the requested semaphore is available. This can be useful for programs that use multiple cards, where a separate semaphore handle is required for the same semaphore on each card. The same semaphore handle can be used on both cards if this function has been called successfully. The allocated semaphore will be given to the variable specified by `symbol_name`, if it is not `NULL`, in the CSX program specified by the process parameter. The same semaphore will be returned in the semaphore parameter. If `symbol_name` is `NULL`, the process parameter must also be `NULL`.The semaphore signalled state will be reset to zero (that is, not signalled) when it is allocated.

### Parameters

`s`: State created by `CSAPI_new` for the card to allocate the duplicate semaphore on

`Semaphore`: Pointer to existing semaphore handle to be duplicated, allocated on another card

`Process`: Process handle for loaded CSX program. Use NULL if `symbol_name` is NULL (that is, unused)

`symbol_name`: Name of static global variable in the CSX program to be given the allocated semaphore

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_bad_csapi_arg,
CSAPIErrno_not_connected, CSAPIErrno_critical_error,
CSAPI_not_loaded, CSAPIErrno_no_symbol,
CSAPIErrno_semaphore_number_invalid,
CSAPIErrno_semaphore_already_allocated,
CSAPIErrno_not_enough_memory, CSAPIErrno_error
```

## CSAPI_semaphore_wait

### Function

```
CSAPIErrno CSAPI_semaphore_wait(
    struct CSAPIState* const s,
    struct CSAPISemaphore* const semaphore,
    CSAPISemaphoreData* const data,
    unsigned int timeout_ms );
```

### Description

Allows the host to wait on the specified semaphore. The function will block until the semaphore is signalled or the period given by the `timeout_ms` parameter has elapsed. This will be indicated by returning `CSAPIErrno_timeout`. The return status must be checked to ensure that the semaphore was signalled. The `CSAPI_NO_TIMEOUT` value can be used with the `timeout_ms` parameter for an infinite wait. A value of zero can be used with the `timeout_ms` parameter to try waiting on the semaphore. If the semaphore was already signalled, it will be decremented and function will return `CSAPIErrno_success`. If the semaphore type is `CST_GsuWithData`, then a pointer can be used to receive the data that the semaphore was signalled with. If the semaphore is a different type or the data is not required, this pointer must be `NULL`.

### Parameters

`s`: State created by `CSAPI_new`

`semaphore`: Semaphore allocated by `CSAPI_allocate_shared_semaphore`

data: Pointer to variable to be given the data that CST_GsuWithData semaphore was signalled with. This must be NULL for other semaphore types. Top bit of data is always zero

`timeout_ms`: Timeout period in milliseconds, after which the function will return `CSAPIErrno_timeout`

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_semaphore_number_invalid,
CSAPIErrno_semaphore_not_allocated,
CSAPIErrno_failed_while_waiting_for_semaphore, CSAPIErrno_timeout,
CSAPIErrno_error
```

## CSAPI_semaphore_signal

### Function

```
CSAPIErrno CSAPI_semaphore_signal(
    struct CSAPIState* const s,
    struct CSAPISemaphore* const semaphore,
    CSAPISemaphoreData data );
```

### Description

Signals the specified semaphore. If the semaphore type is `CST_GsuWithData` then the semaphore can be signalled with a data value, given by the data parameter. This data can then be received when the semaphore is waited on (either on the host or on the processor). If the semaphore is a different type then the data parameter will be ignored.

### Parameters

`s`: State created by `CSAPI_new`

`semaphore`: Semaphore allocated by `CSAPI_allocate_shared_semaphore`

`data`: Data to signal the `CST_GsuWithData` semaphore with. Top bit of data must be zero

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_semaphore_number_invalid,
CSAPIErrno_semaphore_not_allocated
```

## CSAPI_semaphore_get

### Function

```
CSAPIErrno CSAPI_semaphore_get(
struct CSAPIState* const s,
    struct CSAPISemaphore* const semaphore,
    unsigned int* const proc_inx,
    CSAPISemaphoreType* const sem_type,
    unsigned int* const sem_number,
    CSAPISemaphoreValue* const sem_value );
```

### Description

Returns the processor index, semaphore type, semaphore number and current semaphore value (signalled count) for the specified semaphore. This will be the same processor index and semaphore type that was specified when the semaphore was allocated. The semaphore value must only be used for debugging as the result is inherently not thread safe. The `proc_inx`, `sem_type`, `sem_number` and `sem_value` parameters can each be given a `NULL` pointer if they are not required. The returned error code must be checked to ensure the values returned are valid.

### Parameters

`s`: State created by `CSAPI_new`.

`semaphore`: Semaphore allocated by `CSAPI_allocate_shared_semaphore`.

`proc_inx`: Pointer to integer to be given the processor index that the semaphore was allocated on.

`sem_type`: Pointer to variable to be given the semaphore type of the allocated semaphore, as described in *CSAPISemaphoreType on page 76* .

`sem_number`: Semaphore number of allocated semaphore. This should not be used directly.

`sem_value`: Pointer to variable to be given the current value (signalled count) of semaphore.

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_semaphore_number_invalid,
CSAPIErrno_semaphore_not_allocated, CSAPIErrno_error
```

## CSAPI_semaphore_put

### Function

```
CSAPIErrno CSAPI_semaphore_put(
    struct CSAPIState* const s,
    struct CSAPISemaphore* const semaphore,
    CSAPISemaphoreValue sem_value );
```

### Description

Sets the current semaphore value (signalled count) for the specified semaphore. The maximum value depends on the semaphore type as follows: CST_Tsc=255, CST_GsuWithoutData=65535, CST_GsuWithData=32. This function must only be used for debugging as setting the value is inherently not thread safe, unlike signalling and waiting.

### Parameters

`s`: State created by `CSAPI_new`

`semaphore`: Semaphore allocated by `CSAPI_allocate_shared_semaphore`

`sem_value`: Value (signalled count) to set semaphore to. Maximum value depends on semaphore type

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_semaphore_number_invalid,
CSAPIErrno_semaphore_not_allocated, CSAPIErrno_error
```

## CSAPI_free_semaphore

### Function

```
CSAPIErrno CSAPI_free_semaphore(
    struct CSAPIState* const s,
    struct CSAPISemaphore* const semaphore );
```

#### Description

Releases previously allocated semaphore specified by the semaphore parameter. The semaphore is then available to be allocated by `CSAPI_allocate_shared_semaphore`.

#### Parameters

`s`: State created by `CSAPI_new`

`semaphore`: Semaphore allocated by `CSAPI_allocate_shared_semaphore`

#### Returns

`CSAPIErrno_success, CSAPIErrno_bad_csapi_state, CSAPIErrno_bad_csapi_arg, CSAPIErrno_semaphore_number_invalid, CSAPIErrno_semaphore_not_allocated`

## 3.7.11 Callback functions

The following CSAPI callback functions are described in this section:

- CSAPI_get_callback *on page 62*.
- CSAPI_register_callback *on page 63*.

### CSAPI_get_callback

#### Function

```
CSAPIErrno CSAPI_get_callback(
    struct CSAPIState* const s,
    CSAPIEventType event_type,
    CSAPIEventFnPtr* const event_cb_out );
```

#### Description

Gets the function call back registered with `CSAPI_register_callback` for the specified event. Typically used to check see if a call back is registered, or to chain user call back functions.

#### Parameters

`s`: State created by `CSAPI_new.`

`event_type`: Event to trigger the call back function, as described in *CSAPIEventType on page 75*.

`event_cb_out`: Pointer to variable to be given the call back Function pointer

#### Returns

`CSAPIErrno_success, CSAPIErrno_bad_csapi_state, CSAPIErrno_bad_csapi_arg, CSAPIErrno_error`

### CSAPI_register_callback

#### Function

```
CSAPIErrno CSAPI_register_callback(
    struct CSAPIState* const s,
    CSAPIEventType event_type,
    CSAPIEventFnPtr event_cb,
    void* user_data );
```

#### Description

Registers a user call back function for an event signalled by the card. When the event occurs the event call back function will be called with the state, event data and a pointer to specified user data. The user data can be a NULL pointer if not required.

#### Parameters

`s`: State created by `CSAPI_new`.

`event_type`: Event to trigger the call back function, as described in *CSAPIEventType on page 75*.

`event_cb`: Function pointer to an even handler on the host. See definition at top of file.

`user_data`: Pointer to user data to be passed to the event handler on the host. Can be NULL.

#### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_error
```

## 3.7.12 Memory allocation using CSAPI functions

The following CSAPI functions for allocating memory are described in this section:

- CSAPI_get_free_memory *on page 63*.
- CSAPI_allocate_shared_memory *on page 64*.
- CSAPI_allocate_static_shared_memory *on page 65*.
- CSAPI_free_memory *on page 66*.

### CSAPI_get_free_memory

#### Function

```
CSAPIErrno CSAPI_get_free_memory(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    CSAPIMemoryType mem_type,
    CSAPIMemorySize* const size_out );
```

#### Description

Returns the available mono memory (DRAM or ESRAM) local to the specified processor. Processors can see mono memory on other processors but will not access this as efficiently as local memory.

#### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor to get free memory on. Index from 0 upwards

`mem_type`: Memory region to get the available memory for. `CSM_Dram` or `CSM_Esram`, as described in *CSAPIMemoryType on page 76*.

`size_out`: Pointer to variable to be given the current amount of free memory local to processor

#### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_bad_csapi_arg,
CSAPIErrno_not_connected, CSAPIErrno_critical_error,
CSAPIErrno_program_running, CSAPIErrno_invalid_memory_type
```

### CSAPI_allocate_shared_memory

#### Function

```
CSAPIErrno CSAPI_allocate_shared_memory(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    CSAPIMemoryType mem_type,
    CSAPIMemorySize size,
    unsigned int alignment,
    struct CSAPIProcess* process,
    const char* symbol_name,
    CSAPIMemoryAddress* const mem_ptr_out );
```

#### Description

Allocates mono memory (DRAM or ESRAM) local to the specified processor for use by both the CSX program and the host. The memory allocation can use an alignment and the allocated address will be given to the variable specified by `symbol_name`, if not NULL, in the CSX program specified by the process parameter. The same address will be returned in the `mem_ptr_out` parameter. If symbol_name is NULL, the process parameter must also be NULL. Processors can see mono memory on other processors but will not access this as efficiently as local memory. Memory allocation and releasing can only be done when the processor is not running.

#### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor to allocate shared memory on. Index from 0 upwards `mem_type`: Memory region to allocate shared memory from. `CSM_Dram` or `CSM_Esram`, as described in *CSAPIMemoryType on page 76*.

`size`: Number of contiguous bytes to attempt to allocate in the free space

`alignment`: Bytes to align allocated memory to on card

`process`: Process handle for loaded CSX program. Use NULL if symbol_name is NULL (that is, unused)

`symbol_name`: Name of static global variable in the CSX program to be given the allocated address

`mem_ptr_out`: Pointer to variable to be given address of the allocated memory on processor

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_bad_csapi_arg,
CSAPIErrno_not_connected, CSAPIErrno_critical_error,
CSAPIErrno_not_loaded, CSAPIErrno_program_running,
CSAPIErrno_no_symbol, CSAPIErrno_not_enough_blocks,
CSAPIErrno_not_enough_memory, CSAPIErrno_invalid_memory_type,
CSAPIErrno_error
```

## CSAPI_allocate_static_shared_memory

### Function

```
CSAPIErrno CSAPI_allocate_static_shared_memory(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    CSAPIMemoryType mem_type,
    CSAPIMemoryAddress mem_ptr_in,
    CSAPIMemorySize size,
    struct CSAPIProcess* process,
    const char* symbol_name );
```

### Description

Attempts to allocate mono memory (DRAM or ESRAM) local to the specified processor at the address given by `mem_ptr_in`. This is the same as `CSAPI_allocate_shared_memory`, except that instead of returning an address for the allocated memory, it simply returns success or fail for allocating at the address given by `mem_ptr_in`. This can be useful for supporting programs where the addresses of the data arrays are hard coded in the CSX program. Processors can see mono memory on other processors but will not access this as efficiently as local memory. Memory allocation and releasing can only be done when the processor is not running.

### Parameters

`s`: State created by `CSAPI_new.`

`proc_inx`: Index of processor to allocate static shared memory on. Index from 0 upwards.

`mem_type`: Memory region to allocate static shared memory from. `CSM_Dram` or `CSM_Esram`, as described in *CSAPIMemoryType on page 76*.

`mem_ptr_in`: Desired address of the allocated memory on the processor.

`size`: Number of bytes to attempt to allocate at the mem_ptr_in address.

`process`: Process handle for loaded CSX program. Use NULL if `symbol_name` is NULL (that is, unused).

`symbol_name`: Name of static global variable in the CSX program to be given the allocated address.

#### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_bad_csapi_arg,
CSAPIErrno_not_connected, CSAPIErrno_critical_error,
CSAPIErrno_not_loaded, CSAPIErrno_program_running,
CSAPIErrno_no_symbol, CSAPIErrno_not_enough_blocks,
CSAPIErrno_not_enough_memory, CSAPIErrno_address_in_use,
CSAPIErrno_invalid_address, CSAPIErrno_invalid_memory_type,
CSAPIErrno_error
```

### CSAPI_free_memory

#### Function

```
CSAPIErrno CSAPI_free_memory(
    struct CSAPIState* const s,
    CSAPIMemoryAddress mem_ptr );
```

#### Description

Releases previously allocated memory at the address given by `mem_ptr`. This memory on the card is then available to be allocated by `CSAPI_allocate_shared_memory`. Memory allocation and releasing can only be done when the processor is not running.

#### Parameters

`s`: State created by `CSAPI_new`

`mem_ptr`: Address of the allocated memory on the processor, which is to be released

#### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_program_running, CSAPIErrno_address_not_allocated
```

## 3.7.13 Status and information

The following CSAPI functions for getting status information are described in this section:

- CSAPI_get_symbol_value *on page 67*.
- CSAPI_num_processors *on page 68*.
- CSAPI_num_pes *on page 68*.
- CSAPI_process_info *on page 69*.
- CSAPI_connection_info *on page 69*.
- CSAPI_feature *on page 70*.
- CSAPI_check *on page 71*.
- CSAPI_status *on page 71*.
- CSAPI_get_error_string *on page 72*.
- CSAPI_get_error_string_length *on page 73*.

### CSAPI_get_symbol_value

#### Function

```
CSAPIErrno CSAPI_get_symbol_value(
    struct CSAPIState* const s,
    struct CSAPIProcess* process,
    const char* symbol_name,
    CSAPIMemoryAddress* const symbol_value_out );
```

#### Description

Returns the value of the symbol symbol_name as loaded in the given process. This is typically the address of the corresponding static variable in the program. The address can then be used with the `CSAPI_read_mono_memory` and `CSAPI_write_mono_memory` functions to access the contents of the memory on the card pointed to by the variable.

#### Parameters

`s`: State created by `CSAPI_new`

`process`: Pointer to the process containing the given symbol

`symbol_name`: Name of the symbol to obtain the value for (typically a static variable)

`symbol_value_out`: On success this is set to the value associated with symbol_name. (typically the address of a static variable given by symbol_name)

#### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_not_loaded,
CSAPIErrno_no_symbol
```

## CSAPI_num_processors

### Function

```
CSAPIErrno CSAPI_num_processors(
    struct CSAPIState* const s,
    unsigned int* const num_of_processors );
```

### Description

Provides the number of processors (MTAP cores) on the card we are connected to.

### Parameters

`s`: State created by `CSAPI_new`

`num_of_processors`: Pointer to integer to be given the number of processors (MTAP cores)

### Returns

```
CSAPIErrno_success,  CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error
```

## CSAPI_num_pes

### Function

```
CSAPIErrno CSAPI_num_pes(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    unsigned int* const num_of_pes );
```

### Description

Provides the number of processing elements on the specified processor on the card we are connected to.

### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor to get count of processing elements on. Index from 0 upwards

`num_of_pes`: Pointer to integer to be given the number of processing elements

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_invalid_proc_inx, CSAPIErrno_bad_csapi_arg,
CSAPIErrno_not_connected, CSAPIErrno_critical_error
```

## CSAPI_process_info

### Function

```
CSAPIErrno CSAPI_process_info(
    struct CSAPIState* const s,
    struct CSAPIProcess* const process,
    char** const csx_file_name,
    unsigned int* const proc_inx,
    unsigned int* const dynamic );
```

### Description

Returns the full csx filename (including path), processor index and csx linkage type for the specified process. The linkage type is either static or dynamic, depending on how the csx file was linked and whether it can be relocated when loaded. The processor index will be the same one that was specified when the process was loaded, regardless of where a statically linked process has actually been loaded. The `csx_file_name`, `proc_inx` and dynamic parameters can each be given a NULL pointer if they are not required. The returned error code must be checked to ensure the values returned are valid.

### Parameters

`s`: State created by `CSAPI_new`

`process`: Process returned by `CSAPI_load`

`csx_file_name`: Pointer to variable to be given the file name of the loaded csx file, including the full path. The returned pointer will be valid while the process is loaded.

`proc_inx`: Pointer to integer to be given the processor index that the process was loaded on

`dynamic`: Pointer to integer to be given the linkage type of the process (0=static, 1=dynamic)

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_not_loaded, CSAPIErrno_error
```

## CSAPI_connection_info

### Function

```
CSAPIErrno CSAPI_connection_info(
    struct CSAPIState* const s,
    CSAPISharingType* const sharing_type,
    CSAPIConnectType* const connect_type,
    char** const host_name,
    unsigned int* const instance );
```

### Description

Returns the sharing type, connection type, host name and instance number in use for the current connection. The values will be the actual values for the current connection and may differ from the original values passed to `CSAPI_connect`. The instance number will be the actual instance number in use, even if `CSAPI_INSTANCE_ANY` was passed to `CSAPI_connect`. The `sharing_type`, `connect_type`, `host_name` and instance parameters can each be given a NULL pointer if they are not required. The returned error code must be checked to ensure the values returned are valid.

### Parameters

`s`: State created by `CSAPI_new`

`sharing_type`: Pointer to variable to be given the sharing mode, for example, `CSH_Private`, as described in *CSAPISharingType on page 77*.

`connect_type`: Pointer to variable to be given the connection type, for example, `CSC_Direct`, `CSC_Socket`, as described in *CSAPIConnectType on page 74*.

`host_name`: Pointer to variable to be given the host name for the current connection. The returned pointer will be valid while connected.

`instance`: Pointer to integer to be given actual instance number in use (hardware or simulator)

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_error
```

## CSAPI_feature

### Function

```
CSAPIErrno CSAPI_feature(
    struct CSAPIState* const s,
    unsigned int proc_inx,
    CSAPIFeatureType feature_type,
    unsigned int* const value );
```

### Description

Returns the availability or value of the specified feature for the card or simulator on the current connection. The returned error code must be checked to ensure the value returned is valid.

### Parameters

`s`: State created by `CSAPI_new`

`proc_inx`: Index of processor to get the feature for (where applicable). Index from 0 upwards

`feature_type`: Feature to return the value for, as defined in *CSAPIFeatureType on page 76*.

`value`: Pointer to integer to be given the availability or value of the specified feature

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_error
```

## CSAPI_check

### Function

```
CSAPIErrno CSAPI_check(
    struct CSAPIState* const s,
    CSAPICheckType check_type,
    unsigned int* const error,
    struct CSAPIStatus* status );
```

### Description

Returns the result of the specified check for the card or simulator on the current connection. If the check fails the integer pointed to by the error parameter will be set to a non-zero value. If the status parameter is not NULL then the string field will be given a description of the check that failed. The returned error code must be checked to ensure the values returned are valid.

### Parameters

`s`: State created by `CSAPI_new`

`check_type`: Check to perform, as defined in *CSAPICheckType on page 74*.

`error`: Pointer to integer to be given the result of the check, where 0 indicates PASS

`status`: Pointer to structure to be given a description of the failure. Use NULL if not required

### Returns

```
CSAPIErrno_success, CSAPIErrno_bad_csapi_state,
CSAPIErrno_bad_csapi_arg, CSAPIErrno_not_connected,
CSAPIErrno_critical_error, CSAPIErrno_error
```

## CSAPI_status

### Function

```
CSAPIErrno CSAPI_status(
struct CSAPIState* const s,
    unsigned int proc_inx,
    CSAPIStatusType status_type,
    struct CSAPIStatus* status );
```

### Description

Returns the status of the object specified by the `status_type` parameter for the processor specified by the `proc_inx` parameter (where applicable). The result will use the relevant members of the `CSAPIStatus` structure (see the `CSAPIStatusType` enumeration), which includes a string and a list of numbers. The string member will always be given a value. The returned error code must be checked to ensure the status structure is valid.

On Windows systems, the `CSU_ProcCoreFrequency`, `CSU_FindOptimumDmaRead` and `CSU_FindOptimumDmaWrite` status types will cause the current thread to be switched to the first processor. This provides a consistent time reference for the measurements.

### Parameters

`s`: State created by `CSAPI_new.`

`proc_inx`: Index of processor to get the status for (where applicable). Index from 0 upwards.

`status_type`: Object to return the status for, as defined in *CSAPIStatusType on page 77*.

`status`: Pointer to structure to be given the status. The string member will always be populated.

### Returns

`CSAPIErrno_success, CSAPIErrno_bad_csapi_state, CSAPIErrno_not_connected, CSAPIErrno_critical_error, CSAPIErrno_error`

## CSAPI_get_error_string

### Function

```
CSAPIErrno CSAPI_get_error_string(
    struct CSAPIState* const s,
    CSAPIErrno error_code,
    char* const error_string,
    unsigned int max_string_length );
```

### Description

This function decodes an `error_code` and copies a short description of the error to the provided char buffer. The string will be truncated at `max_string_length` if necessary to avoid over-running the provided buffer. If this occurs then the function will return `CSAPIErrno_string_truncated`, although the error string will still be valid.

### Parameters

`s`: State created by `CSAPI_new`. Required to ensure the dynamic CSAPI library is loaded

`error_code`: Error code to get the string for. Must be of type `CSAPIErrno`

`error_string`: Pointer to pre-allocated buffer to receive error string

`max_string_length`: Length of pre-allocated buffer to receive error string

#### Returns

`CSAPIErrno_success, CSAPIErrno_bad_csapi_arg, CSAPIErrno_string_truncated`

### CSAPI_get_error_string_length

#### Function

```
CSAPIErrno CSAPI_get_error_string_length(
    struct CSAPIState* const s,
    CSAPIErrno error_code,
    unsigned int* const length );
```

#### Description

Returns the length of the short description string for the given `error_code`. This allows a buffer to be allocated that will hold the same error string from `CSAPI_get_error_string` without truncation.

#### Parameters

`s`: State created by `CSAPI_new`. Required to ensure the dynamic CSAPI library is loaded

`error_code`: Error code to get the string length for. Must be of type `CSAPIErrno`

`length`: Pointer to integer to be given the length of the error string

#### Returns

`CSAPIErrno_success, CSAPIErrno_bad_csapi_arg`

## 3.7.14 Enumerated types

### CSAPIAddressCheckType

#### Description

Address check type definitions for use with the `address_check` member of the `CSAPITransferParam` structure.

#### Possible values

`CSA_NoCheck:` No check is made on the card side address being used

`CSA_Valid`: A basic check is made to ensure the card side address is a valid address.

`CSA_Allocated`: A longer check is made to ensure the card side address is within an allocated block.

## CSAPIAlignmentFixType

### Description

Alignment fix type definitions for use with the `alignment_fix` member of the `CSAPITransferParam` structure.

### Possible values

`CSL_NoCheck`: No check is made, so badly aligned buffers will be transfered without DMA (that is, slowly).

`CSL_ReturnError`: The alignment is checked and an error will be returned if the buffers are badly aligned.

`CSL_BreakUp`: If the source and destination have the same mis-alignment, then most of the data will be transferred using DMA. An error will be returned if the mis-alignment is different.

`CSL_Shift`: The host buffer will be copied to have the same alignment as the card address.

## CSAPICheckType

### Description

Check type definitions for use with the `CSAPI_check` function .

### Possible Values

`CSK_Register`:  Check register accesses.

`CSK_ProcRegister`:  Check register accesses on the processor.

`CSK_Status`: Check for critical error conditions in the status registers.

`CSK_Memory`: Check small memory transfers that do not use DMA engine.

`CSK_Dma`: Check large memory transfers that use the DMA engine.

`CSK_Full`: Run all the checks one after the other.

## CSAPIConnectType

### Description

Connection type definitions for use with the `CSAPI_connect` function.

### Possible values

`CSC_Direct`: A direct connection to a card fitted in the local machine.

`CSC_Socket`: A TCP socket connection to a simulator on the local machine or another machine.

`CSC_Usb`: A connection via USB to the debug port on the card. Only some functions are available.

`CSC_EnvironmentVariable`: Read environment variables to determine the connection type. Default is `CSC_Direct`. See *Chapter 5: Runtime environment variables on page 85*.

## CSAPIEndianType

### Description

Endianness definitions for use with the `CSAPI_endianness` function.

### Possible values

`CSN_LittleEndian`: Increasing numeric significance with increasing memory address, for example, x86 systems.

`CSN_BigEndian`: Decreasing numeric significance with increasing memory address, for example, PowerPC, SPARC, DEC Alpha.

## CSAPIEventType

### Description

Event type definitions for use with the event handling functions.

### Possible values

`CSE_TemperatureAlert`: One of the devices on the card has exceeded its temperature alert threshold.

`CSE_BusMonitor`: The bus monitor has been triggered, typically by an access to an invalid address.

`CSE_DdrEcc`: An ECC event has been generated by the local memory controller.

`CSE_SystemEcc`: An ECC event has been generated by the CCBR, TSC, PIO, Array Controller or a PE.

`CSE_Break`: The program running on the processor has hit a break point, used by the debugger.

`CSE_Terminate`: The program running on the processor has terminated.

`CSE_Print`: The program running on the processor is printing to the terminal on the host.

`CSE_StackOverflow`: The stack for the program currently running on the processor has overflowed.

`CSE_SemaphoreNonZero`: A semaphore on the processor has been signalled.

`CSE_SemaphoreOverflow`: A semaphore on the processor has been signalled too many times and overflowed.

`CSE_PapiOverflow`: One of the PAPI counters on the processor has overflowed.

`CSE_Malloc`: The `malloc` function on the processor requires another block allocation from the host.

`CSE_StackCheck`: The dynamic stack on the processor needs to be checked by the host.

### CSAPIFeatureType

#### Description

Feature type definitions for use with the CSAPI_feature function

#### Possible values

`CSF_DmaReadThreshold`: Depricated. Use CSP_DmaReadThreshold with CSAPI_get_system_param instead.

`CSF_DmaWriteThreshold`: Depricated. Use CSP_DmaWriteThreshold with CSAPI_get_system_param instead.

`CSF_DmaAlignment`: The alignment required by the DMA engine for the card address and host buffer (bytes).

`CSF_DmaGsuLinkage`: Availability of linking GSU semaphores to DMA transfers (0 or 1).

`CSF_NumDmaChannels`: The number of DMA channels available.

### CSAPIMemoryType

#### Description

Memory type definitions for use with memory functions.

#### Possible values

`CSM_Dram`: The external DRAM attached to each processor.

`CSM_Esram`: The internal ESRAM on each processor.

`CSM_Edram`: The internal EDRAM on each processor (none on present systems).

### CSAPIResetType

#### Description

Reset type definitions for use with the `CSAPI_reset` function.

#### Possible values

`CSR_Processor`: Processor (MTAP core) reset.

`CSR_FullSystem`: CCBR, CCIs, DDRs and Bus monitor, plus usual Processor reset.

`CSR_PciLink`: Reset the PCI link, reconnect and then reset the full system.

### CSAPISemaphoreType

#### Description

Semaphore type definitions for use with semaphore functions.

### Possible values

`CST_Tsc`: Thread Sequence Controller semaphore. Use within a processor and between host and processor.

`CST_GsuWithoutData`: Global Semaphore Unit semaphore. Use between multiple processors, host, PIO and DMA units.

`CST_GsuWithData`: Global Semaphore Unit semaphore that includes data with each semaphore signal.

## CSAPISharingType

### Description

Sharing type definitions for use with the CSAPI_connect function

### Possible values

`CSH_Private`: Normal connection ensuring the current process has exclusive access to the card.

`CSH_SharedUser`: Shared connection restricting access to processes being run by the same user or root.

`CSH_SharedPublic`: Shared connection allowing any user or process access to the card.

## CSAPIStatusType

### Description

Status type definitions for use with the CSAPI_status function.

### Possible values

`CSU_CardLocation`: (3 integers) Card location on PCI bus: Bus number, Device number and Function number.

`CSU_PcieLinkWidth`: (1 integer) The current PCI express link width (number of lanes).

`CSU_PciConfigSpaceStatus`: (1 integer) The current PCI config space status value.

`CSU_FpgaCcbrFrequency`: (1 integer) The speed of the FPGA CCBR link in MHz.

`CSU_ProcCoreFrequency`: (1 float) The core frequency of the processor in MHz, to an accuracy of 0.1MHz.

`CSU_ProcTemperature`: (2 integers) Temperature values for the processor in degrees celcius: On Die and on Card.

`CSU_FpgaTemperature`: (2 integers) Temperature values for the FPGA in degrees celcius: On Die and on Card.

`CSU_TemperatureAlert`: (3 integers) Temperature alert status (1 = over temperature): FPGA and each processor.

`CSU_PowerSupplyFail`: (3 integers) Power Supply alert status (1 = out of range): FPGA, 1.8v IO and Processor core.

`CSU_CriticalFailure`: (2 integers) Previous critical failure, cleared once read: PSU Fail, Temperature critical.

`CSU_HdpError`: (3 integers) Host Debug Port error, cleared once read: FPGA and each processor.

`CSU_TransactionError`: (1 integer) Transaction error type from HIF DMA or HIF initiator.

`CSU_FanSpeed`: (2 floats) Fan speed values in rpm for the 2 fans, if fitted.

`CSU_FindOptimumDmaRead`: (1 integer) Performs memory reads to find the optimum threshold for DMA read transfers.

`CSU_FindOptimumDmaWrite`: (1 integer) Performs memory writes to find the optimum threshold for DMA write transfers.

## CSAPISystemParam

### Description

System Parameter definitions for use with the `CSAPI_set_system_param` and `CSAPI_get_system_param` functions.

### Possible values

`CSP_Verbosity = 0`: Use integer value (<0> or 1).

`CSP_LogLevel`: Use integer value.

`CSP_DmaReadThreshold`: Use integer value (bytes).

`CSP_DmaWriteThreshold`: Use integer value (bytes).

`CSP_OptimumDmaRead`: Use integer value (bytes).

`CSP_OptimumDmaWrite`: Use integer value (bytes).

`CSP_ForceBusBigEndian`: Use integer value (<0> or 1 only) (Applied during reset).

`CSP_ForceBusLittleEndian`: Use integer value (<0> or 1 only) (Applied during reset).

`CSP_ZeroBss`: Use integer value (<0> or 1) (Applied during reset).

`CSP_LoadFuseMask`: Use string value (filename) (Set only, Applied during reset).

`CSP_SaveFuseMask`: Use string value (filename) (Set only, Applied during reset).

`CSP_FuseStrictCheck`: Use integer value (<0> or 1) (Set only, Applied during reset).

`CSP_CcbrClockSlow`: Use integer value (75, 80, <100>, 250, 300) (Applied during reset).

`CSP_CcbrClockFast`: Use integer value (100, <200>, 250, 300, 375, 400) (Applied during reset).

`CSP_CcbrCrc`: Use integet value (0 or <1>) (Applied during reset).

`CSP_DdrClock`: Use integer value (133, 166, <200>, 233, 266) (Applied during reset).

`CSP_IcacheBypassActive`: Use integer value (<0> or 1) (Applied during reset).

`CSP_DcacheBypassActive`: Use integer value (<0> or 1) (Applied during reset).

`CSP_DdrEccOn`: Use integer value (<0> or 1) (Applied during reset).

`CSP_TraceMode`: Used internally by debug / trace layer (Set only).

### CSAPIVersionType

#### Description

Version type definitions for use with the `CSAPI_version` function. If a connection is not required, then NULL can be passed to `CSAPI_version` for the `CSAPIState` parameter. All other types require a connected state.

#### Possible values

`CSV_CsapiHeader`: (major + minor) CSAPI header versions defined in csapi.h (connection not required).

`CSV_RuntimePackage`: (major + minor) Version of the installed runtime package (connection not required).

`CSV_RuntimeBuild`: (string only)  Internal build version of the runtime (connection not required).

`CSV_CsxKernel`:  (major) Version of the csx kernel interface.

`CSV_FpgaVersion`: (major) Version of the FPGA on the connected card.

`CSV_FpgaTimeStamp`: (major) Timestamp for the FPGA image on the connected card.

`CSV_CardType`: (string only) Product name of the connected card.

`CSV_CardSerialNumber`: (string only) Serial Number of the connected card.

`CSV_CardFinalTestDate`: (major) Timestamp for the final test date of the connected card.

`CSV_MtapVersion`: (major) Version of the first MTAP on the connected card.

## 3.7.15 CSAPI structures

The following structures are used by some of the CSAPI functions. In each case, the structure must be declared in your code, and the address of the structure must be passed to the CSAPI function.

### CSAPIVersion

#### Description

The `CSAPIVersion` structure is used by the `CSAPI_version` function. If the call to `CSAPI_version` is successful, the members of the `CSAPIVersion` structure will be given values. Some version types will not use the major or minor version number members (see *CSAPIVersionType on page 79*). All version types will use the string member.

#### Members

`major`: This is the major part of the version number, defined as an integer.

`minor`: This is the minor part of the version number, defined as an integer.

`string`: This is a NULL terminated string showing the major and minor version numbers in the form `major.minor`, for example, `2.1.`

## CSAPIStatus

### Description

The `CSAPIStatus` structure is used by the `CSAPI_status` and `CSAPI_check` functions. If the call to the CSAPI function is successful, some of the members of the `CSAPIStatus` structure will be given values. The values that are set depend on the status or check type requested (see *CSAPIStatusType* and *CSAPICheckType on page 74*). All status types and check types will use the string member.

### Members

`integers`: An array of integers. The number of integers given values depends on the status or check type. `integers[0]` is always the first integer used.

`floats`: An array of floats. The number of floats given values depends on the status or check type. `floats[0]` is always the first float used.

`string`: This is a NULL terminated string showing result of the status query or check.

## CSAPITransferParam

### Description

The `CSAPITransferParam` structure is used by the `CSAPI_read_mono_memory` and `CSAPI_write_mono_memory` functions, and the asynchronous variants. The members of the structure form extra parameters for controlling the transfer (see *Table 7 on page 25*). Two predefined structures are available, which are described in *Section 3.4.9: Memory and register access on page 24*. You must initialize a locally declared `CSAPITransferParam` structure by copying in values from one of the predefined structures. Local modifications can then be made. This provides forward compatibility when extra parameters are added to the structure.

### Members

`address_check`: Validates or checks the allocation of the card side address.

`alignment_fix`: Breaks up or shifts badly aligned data to maximise use of DMA.

`halt_processor`: Halts affected processor to avoid memory page conflicts during transfer.

`flush_cache`: Flushes the cache on the affected processor before the transfer.

`start_sem`: Semaphore to wait on before the transfer is started. Set to NULL if not used.

`complete_sem`: Semaphore to signal when the transfer completes. Set to NULL if not used.

See *Table 7 on page 25* for more information on the members of the `CSAPITransferParam` structure.

## 3.8 Access control and the lock file

Access to cards and simulators is controlled using entries in a lock file called `cs_lock_file.txt`. On Linux systems, this file is stored in the `/var/lock/clearspeed` directory. On Microsoft Windows systems, it is stored in the installation directory (typically `C:\Program Files\clearspeed\csx600`). Entries are added during connection and removed during disconnection. Entries include the host application's process ID, which is used by other applications to confirm that the process connected to the card or simulator is still running.

Access control prevents concurrent access to Advance boards. The access control is not intended to be 100% secure - it is intentionally made open, with a plain text lock file that can be edited and deleted. The locking mechanism can be disabled by removing the write privilege from the lock file parent folder. Doing this will allow concurrent access to the Advance cards resulting in undefined behavior.

The lock file, `cs_lock_file.txt`, is created if it does not already exist. If it cannot be created, the following warning will be printed:

```
Warning: Not using lock file. Check rw permissions for
/var/lock/clearspeed/cs_lock_file.txt.
```

Another file, `cs_lock_file.lock`, is created when the lock file is accessed. This file is used to prevent concurrent access to the `cs_lock_file.txt` file. Each process creates its own file, which is then symbolically linked to the `cs_lock_file.lock` file. Only one process can be symbolically linked at one time, and so only one process can access the `cs_lock_file.txt` file at one time. If the existing link exists for too long, it is considered stale and is ignored.

The lock file contains the following header block:

```
Lock file for the ClearSpeed driver. Each entry starts with an
asterisk. White space is ignored. Entries are Type, Instance,
UserID, PID, Lock Time. All entries present in this file are
considered locked.
```

The header block of information text is fixed and is regenerated each time the file is written. The lock-file entries follow this. Each entry consists of five lines following an asterisk. The following is an example for the entry for a card (2), instance 0, locked by user ‘tims’ running process ID 18830 at time 1134472924. The last line shows the time in text, but this is not read by the software.

```
Resource 1 *
2
0
tims
18830
1134472924
Locked by tims on Tue Dec 13 11:22:04 2007
```

All text from the time entry on line 5 to the next asterisk is ignored. If a card or simulator is locked, it has an entry in the lock file. If it is not locked, it does not have an entry in the lock file. The process ID can be used to check that the process that locked the card is still running and allows stale entries in the lock file to be identified.

## 3.9 DMA issues

The Advance board uses Direct Memory Access (DMA) to perform data transfer with the host and by default DMA is used to transfer all but the smallest pieces of data. Data is normally transferred directly to and from the user buffers passed to the `CSAPI_read_mono_memory` and `CSAPI_write_mono_memory` functions, no intermediate buffering should be used if at all possible. This implies that, depending on host architecture characteristics, it is safer to align the host data buffers on a system page boundary. For x86 and x86-64 the page size is 4 KB. The reason for this is that the entire system page needs to be operated on while the DMA transfer is proceeding and reads and writes into a page that includes either the start or end of the DMA buffer may cause obscure and difficult to track problems due to race conditions between DMA operations and host memory traffic.

Also, for PCI-X, the DMA engine can only transfer data that is at least 8-byte aligned on both the host machine and the Advance board. For PCIe, it is 4-byte aligned. The alignment value can be obtained from the `CSAPI_feature` function using the type `CSF_DmaAlignment`. If the data is not aligned then the behavior depends on the `alignment_check` member of the transfer parameters structure (see *Section 3.4.9: Memory and register access*).

The threshold for determining if a transfer will use the DMA engine can be read and modified using the `CSAPI_get_system_param` and `CSAPI_set_system_param` functions (see *Section 3.4.11: System parameters on page 26*). There are separate thresholds for reading from and writing to the card. An optimum threshold value is stored in a file called `cs_dma_stats_file_card_00.txt`, which is created in the same directory as the `cs_lock_file.txt` file (see *Section 3.8: Access control and the lock file on page 82*). The contents of this file are read during connection, and can also be read and modified using the `CSAPI_get_system_param` and `CSAPI_set_system_param` functions. The csreset utility (see *Section 2.1: csreset on page 9*) will find the optimum DMA threshold using the `CSAPI_status function` and update the file using the `CSAPI_set_system_param` function after resetting the card.

# 4 Diagnostic software reference

This chapter is under review--the software development is not finished yet.

This chapter describes the commands relating to the card diagnostic operations. The card diagnostic software is a package installed on the host operating system and includes two diagnostic tools:

- Diagnostic tests using Perl
- A Mandelbrot demonstration

The Mandelbrot demonstration is a quick way of checking that the card has been installed correctly. However, to run a full set of tests, you must use the Perl-based diagnostic tests.

The following sections give the instructions for running these tests on both Windows XP and Linux, as appropriate.

## 4.1 Diagnostic tests using Perl

The diagnostic tests, which are run using Perl, enable you to check that all the parts of the card are working properly. For example, the tests check the CSX processors, the memory, and the performance of the DMA transfers.

You should always run these tests:

- After you have installed the cards.
- If you experience a specific problem when using the card.

To run the full tests, you must have the Perl interpreter installed on your machine. Linux machines usually have Perl already installed. For a ready-to-install distribution of Perl for Windows, refer to ActivePerl at:

http://www.activestate.com

### 4.1.1 Full diagnostic tests for Windows XP

To run the diagnostic tests:

5. Double-click the desktop shortcut `csx600_m512_le` to open the command window.
6. In the new window, run the following commands:

```
perl -S run_tests.pl
```

*Note: Some of the tests may take several minutes to run to completion. If you need to interrupt this process, press `[Ctrl]+[C]`.*

### 4.1.2 Full diagnostic tests for Linux

To run the diagnostic tests:

1. Go to a Linux directory where you have writing privileges, for example:

```
cd /tmp
source /opt/clearspeed/csx600_m512_le/bin/bashrc
```

2. Run the program:

```
perl run_tests.pl
```

*Note: Some of these tests may take several minutes to run to completion. If you need to interrupt this process, press* `[Ctrl]+[C]`*.*

### 4.1.3 What to do if the tests fail

If any of the tests fail, it is possible that the card is not plugged in correctly. We recommend that you reinstall the card and run the tests again.

If the tests fail to detect the card and all the tests fail, you may not have installed the driver correctly. We recommend you reinstall the drivers as described in the installation instructions on: http://support.clearspeed.com/.

If the card and drivers are installed correctly but some of the tests still fail, refer to the troubleshooting guide in the *[4]: Advance X620 Accelerator Card User Guide* or the *[5]: Advance e620 Accelerator Card User Guide* to help you solve the problem.

## 4.2 Mandelbrot demonstration

The Mandelbrot set is a type of infinitely complex mathematical object known as a fractal. It can be run to check that the card or cards have been installed correctly. This test is successful when it displays a Mandelbrot set. This program is based in part on the work of the FLTK project (www.fltk.org).

### 4.2.1 How to run the Mandelbrot demonstration in Windows XP

To run the Mandelbrot demonstration:

1. Double-click the desktop shortcut `csx600_m512_le` created by the installation. This opens a command window.
2. Reset all the cards, by entering the following command in the new window:
   ```
   csreset -Av
   ```
3. Run the Mandelbrot demo:
   ```
   app_mandelbrot
   ```
   Running `app_mandelbrot` causes a window to appear which zooms in and out of a Mandelbrot set. If this does not happen, please run the full diagnostic tests described below.
4. To terminate the Mandelbrot program, do one of the following:
   – Press [Esc] to exit from the Mandelbrot GUI window.
   – Press [CTRL]+[C] in the console window where the application was invoked.

### 4.2.2 How to run the Mandelbrot demonstration in Linux

To run the Mandelbrot program:

1. Reset the card:
   ```
   source /opt/clearspeed/csx600_m512_le/bin/bashrc
   csreset -Av
   ```
2. Run the Mandelbrot demo:
   ```
   ./app_mandelbrot
   ```

# 5 Runtime environment variables

This chapter contains a description of the runtime environment variables used by the runtime software.

## 5.1 CSAPI_CONNECT_SIM

If the `CSAPI_connect` function is called with the `CSC_EnvironmentVariable` connection type, the driver will look for this environment variable. If it is found, the `CSAPI_connect` function will continue with a connection type of `CSC_Socket`, otherwise it will continue with a connection type of `CSC_Direct`. The value of the environment variable is not used.

## 5.2 CSAPI_INSTANCE

If the `CSAPI_connect` function is called with the `CSC_EnvironmentVariable` connection type, the driver will look for this environment variable. If it is found, the `CSAPI_connect` function will use the value of the environment variable for the `instance` parameter. Otherwise, it will continue with the value that was passed to the function. The environment variable must be set with an integer value.

## 5.3 CSAPI_HOST

If the `CSAPI_connect` function is called with the `CSC_EnvironmentVariable` connection type, the driver will look for this environment variable. If it is found, the `CSAPI_connect` function will use the value of the environment variable for the `host_name` parameter, otherwise it will continue with the value that was passed to the function. The environment variable must be set with a string suitable for the `host_name` parameter. This can be "localhost", a remote host name or an IP address.

## 5.4 CS_CLEARD_ARGS

The driver library makes use of an environment variable `CS_CLEARD_ARGS` to allow options to be passed to the library components. In general, it is not necessary to use these options but, for example, it is possible to specify a verbose option to the library by setting this environment variable:

- `export CS_CLEARD_ARGS="--verbose"` (Linux)
- `set CS_CLEARD_ARGS="--verbose"` (Windows)

Some of these options are also available as command line arguments to the tools. The environment variable `CS_CLEARD_ARGS` is essentially a command line to the CSAPI driver libraries.

The options which can be specified in `CS_CLEARD_ARGS` are listed in *Table 9*.

| Long name | Short Name | Valid Values | Description |
|---|---|---|---|
| `--bus-big-endian` | `-B` | | Force the ClearConnect bus to be configured as big endian, even for a little endian CSX processor configuration. |
| `--bus-little-endian` | `-b` | | Force the ClearConnect bus to be configured as little endian, even for a big endian CSX processor configuration. |
| `--ccbr-clock-fast` | | *integer* | Clock speed, in MHz, for interface between CSX processors. |
| `--ccbr-clock-slow` | | *integer* | Clock speed, in MHz, for interface to FPGA. |
| `--ccbr-crc` | | `on`\|`off` | CCBR CRC checking. |
| `--cstrace` | | *filename* | Set filename for trace output and enabled trace. |
| `--dma-read-threshold` | | *integer* | Read threshold in bytes for using DMA engine. |
| `--dma-write-threshold` | | *integer* | Write threshold in bytes for using DMA engine. |
| `--dcache-bypass` | | `on`\|`off` | Data cache bypass mode. The default is off. |
| `--ddr-clock` | | *integer* | Set speed, in MHz, of DDR2 interface. |
| `--ecc` | | `on`\|`off` | Turn error checking on or off. |
| `--fuse-check` | | `on`\|`off` | Strict PE fuse check (`on` may be faster). The default is `off`. |
| `--help` | `-h` | | Outputs the information in this table. |
| `--icache-bypass` | | `on`\|`off` | Instruction cache bypass mode. The default is `off`. |
| `--load-fuse-mask` | | *filename* | Load PE configuration mask from file. |
| `--logall` | | | Set logging to the highest level. |
| `--logmask` | `-l` | *integer* (hex or decimal) | Controls the log. |
| `--save-fuse-mask` | | *filename* | Save PE configuration mask to file. |
| `--verbose` | `-v` | | Switch on verbose display. |
| `--version` | `-V` | | Display version string. |
| `--zero-bss` | | `on`\|`off` | Initialize BSS sections to zero. The default is `on`. |

**Table 9. Driver command-line options**

## 5.5 CS_DISABLE_DMA

Setting `CS_DISABLE_DMA` to any value will disable the use of the DMA engine for transferring data to and from the card (see *Section 3.9: DMA issues on page 82*).

## 5.6 CS_CSAPI_DEBUGGER

Setting `CS_CSAPI_DEBUGGER` to a nonzero value initializes the debug interface inside the host application.

## 5.7 CS_CSAPI_DEBUGGER_ATTACH

Setting `CS_CSAPI_DEBUGGER_ATTACH` to a nonzero value allows the host application to attach to the device before the host application executes any code, and set a breakpoint.

## 5.8 LLDINST

Sets the card or simulator instance to use, if the application does not define it. Either a number or the word 'any'. If not defined or the value is ‘any’, the first available card will be used.

## 5.9 LLDHOST

Sets the host name for a simulator. If not defined, the simulator is assumed to be running on the local host.

# 6 Bibliography

[1] *GNU Manuals Online*
http://www.gnu.org/manual/

[2] *Linux Device Drivers, Third Edition*
Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman
ISBN 0-596-00590-3
O'Reilly Media, Inc.

[3] *get_user_pages usage*
http://lwn.net/Articles/28548/
Note that lwn.net is a useful resource for kernel information.

[4] *Advance X620 Accelerator Card User Guide*
Document Number: 06-UG-1302
ClearSpeed Technology

[5] *Advance e620 Accelerator Card User Guide*
Document Number: 06-UG-1443
ClearSpeed Technology

[6] *SDK Reference Manual*
Document Number: 06-RM-1436
ClearSpeed Technology

**ClearSpeed Technology, Inc.**
3031 Tisch Way, Suite 200
San Jose, CA 95128
United States of America

Tel: +1 408 557 2067
Fax: +1 408 557 9054

**ClearSpeed Technology plc**
3110 Great Western Court
Hunts Ground Road
Bristol BS34 8HP
United Kingdom

Tel: +44 (0)117 317 2000
Fax: +44 (0)117 317 2002

Email: info@clearspeed.com

Web: http://www.clearspeed.com

Support: http://support.clearspeed.com