ClearSpeed

CSX600 Runtime Software User Guide

User Guide

06-UG-1345 1.20.2.11

17 April 2007

ClearSpeed Technology, Inc.
3031 Tisch Way, Suite 200
San Jose, CA 95128

Tel: 408-557-2067
Fax: 408-557-9054

Email: info@clearspeed.com
Web: www.clearspeed.com

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

Conventions

Convention	Description
`commands`	This typeface means that the command must be entered exactly as shown in the text and the `[Return]` or `[Enter]` key pressed.
`Screen displays`	This typeface represents information as it appears on the screen.
`[Key]` names	Key names appear in the text written with brackets. For example `[Return]` or `[F7]`. If it is necessary to press more than one simultaneously, the key names are linked with a plus (+) sign: Press `[Ctrl]+[Alt]+[Del]`
Bold-face text	Signal names, instructions and register names are displayed in bold. Selections made via the menu hierarchy of a software application.
Words in *italicized* type	Italics emphasize a point, concept or denote new terms.
	This symbol indicates important information or instructions.

1. Information and data contained in this document, together with the information contained in any and all associated ClearSpeed documents including without limitation, data sheets, application notes and the like ('Information') is provided in connection with ClearSpeed products and is provided for information only. Quoted figures in the Information, which may be performance, size, cost, power and the like are estimates based upon analysis and simulations of current designs and are liable to change.
2. Such Information does not constitute an offer of, or an invitation by or on behalf of ClearSpeed, or any ClearSpeed affiliate to supply any product or provide any service to any party having access to this Information. Except as provided in ClearSpeed Terms and Conditions of Sale for ClearSpeed products, ClearSpeed assumes no liability whatsoever.
3. ClearSpeed products are not intended for use, whether directly or indirectly, in any medical, life saving and/ or life sustaining systems or applications.
4. The worldwide intellectual property rights in the Information and data contained therein is owned by ClearSpeed. No license whether express or implied either by estoppel or otherwise to any intellectual property rights is granted by this document or otherwise. You may not download, copy, adapt or distribute this Information except with the consent in writing of ClearSpeed.
5. The system vendor remains solely responsible for any and all design, functionality and terms of sale of any product which incorporates a ClearSpeed product including without limitation, product liability, intellectual property infringement, warranty including conformance to specification and or performance.
6. Any condition, warranty or other term which might but for this paragraph have effect between ClearSpeed and you or which would otherwise be implied into or incorporated into the Information (including without limitation, the implied terms of satisfactory quality, merchantability or fitness for purpose), whether by statute, common law or otherwise are hereby excluded.
7. ClearSpeed reserves the right to make changes to the Information or the data contained therein at any time without notice.

Contents

1 Introduction

This document describes the components that make up the CSX600 runtime package. These are:

- Stand-alone host tools to reset the Advance board and to load and run programs on the boards. See chapter 2, *Running code*, on page 8.
- A standard source code debugger, `csgdb`, is provided to allow debugging of applications running on the Advance board. See chapter 3, *Debugger Reference*, on page 13.
- 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 5, *Diagnostic software reference*, on page 83.
- The host application programming interface and libraries used by a user application on the host to control and communicate with the CSX600 processor. See chapter 4, *Host interface library*, on page 37.
- A kernel level driver is installed at boot time on the host operating system and provides a very low level interface to the CSX600 processor. See chapter 6, *Kernel level driver*, on page 85.

2 Running code

To run compiled code on the CSX600 processor or a simulator, a program needs to be run on the host computer to load the code and start it running. As part of the runtime software, a simple host program called `csrun` is provided. This will boot the CSX processor(s), load and run the specified CSX executable and then provide host services such as I/O.

Before code is run on the CSX600 processor, it needs to be reset. The command `csreset` (see page 10) can be used to reset one or all of the CSX600 processors in a system.

Note: You should ensure that applications are properly terminated because a background process connected to the board will prevent other applications from connecting to it. If a program using the Advance board is terminated abnormally, it is possible that it may continue to run in the background. Further attempts to use the board will fail to connect, giving the process ID of the process still using the board. This process must be terminated before the Advance board can be used by another program.

2.1 csrun

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

2.1.1 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 paths specified in the `CSPATH` environment variable to find the executable.

Note: To run the program `fred.csx` in the *current directory,* the directory *must* be specified (as `.' or an absolute path). For example:

```
csrun ./fred.csx   (on Linux)
csrun .\fred.csx   (on Windows)
```

2.1.2 Command line options

The `csrun` command line options are summarized in Table 2.1**.**

Long name	Short name	Valid values	Description
`--chip`	`-c`	*integer*	Select which chip to run on (0 or 1). Default: 0.
`--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 board or a simulator instance (0,1,2,...). Default: any available.

Table 2.1 csrun command line options summary

Long name	Short name	Valid values	Description
`--sim`	`-s`		Connect to a simulator rather than hardware.
`--verbose`	`-v`		Switch on verbose output.
`--version`	`-V`		Display version information.

Table 2.1 csrun command line options summary

`-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 board. The default if not specified is 0, the first chip.

`-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 board or instance of the simulator to connect to. Instance numbers start from 0. The default, if not specified, is 0 which connects to the next available board or simulator.

`-s`
`--sim`

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

`-v`
`--verbose`

Displays more information from `csrun`.

`-V`
`--version`

Displays version information. This includes the overall software release version (the "distribution" version) and the specific build of ***csrun***.

Example

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

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

In order to load an executable that is not on the current `CSPATH` environment path, a full path name can be specified, for example:

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

2.2 csreset

`csreset` provides a means of resetting the CSX600 processor. At a system level `csreset` configures the bus and gets the CCBRs so you can do memory transfers to both MTAPs. `csreset` also initializes the DDR memory. It configures the appropriate 96 PEs for each MTAP, 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.

2.2.1 When to use csreset

You need to run `csreset` on each board 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 board, run the script `recover_board` as described in 2.2.4, *Recovering the board*, on page 12. 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 boards installed in the system are reset. `csreset` also provides a finer level of control over which boards, or chips on a board, are to be reset.

2.2.2 Invoking csreset

The command line for `csreset` is:

```
csreset [option]*
```

2.2.3 Command line options

The `csreset` command line options are summarized in Table 2.2.

Long name	Short name	Valid values	Description
`--all`	`-A`		Reset all boards.
`--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 board or a simulator instance.
`--no-reset`			Connect but do not reset processors.
`--sim`	`-s`		Connect to a simulator rather than hardware.
`--verbose`	`-v`		Prints out detailed information on the screen about the board and chips.
`--version`	`-V`		Display version information.

Table 2.2 csreset command line options summary

-A
--all

Resets all the boards 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 board. If the `-c` option is not used, all chips on the specified boards 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 board or instance of the simulator to connect to. Instance numbers start from 0. The default, if not specified, is 0 which connects to the next available board or simulator.

--no-reset

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

-s
--sim

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

--verbose

Gives detailed information about the FPGA version, the temperature (°C) of the boards, the board's serial number, the final board test date, the installed memory type and the MTAP fuses.

Examples

Resetting all boards:

```
csreset -A
```

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

```
cserest -i 1 -A
```

Resetting all simulated boards:

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

2.2.4 Recovering the board

This release includes a script for resetting the Advance board 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 board, set the environment variable `LLDINST` to the instance number of the board to be recovered.

For example, to reset the first board 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 2.2.2, *Invoking csreset*, on page 10.

This procedure can be repeated with different values of `LLDINST` to reset each board 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=
```

3 Debugger Reference

This chapter describes the changes made to the standard GDB command set to allow debugging of the CSX architecture.

`csgdb` is a port of the GNU open source debugger GDB to support the CSX family of microprocessors. It has been extended to provide support for the **C*n*** language and the data-parallel architecture of the CSX processors.

The aim in porting GNU GDB to support the CSX architecture is to utilize as much of the standard functionality provided and change only what is required to allow access to novel features of CSX processors. The standard GDB reference manual *Debugging with GDB* [2] is provided with the SDK. This is a comprehensive document that explains how to use all of the standard features of the debugger.

3.1 New commands and features

The following new commands have been added to the debugger:

- `connect` command (see *Connect command and options*, on page 14).
- *linked* instructions (see *Disassemble command*, on page 18).
- `peregs` command (see *Reading poly registers – peregs command*, on page 19).
- `pex` command (see *Reading poly memory – pex command*, on page 20).
- `whatis` command displays information about poly values (see *Symbolic debug*, on page 22).
- `sysreg` command (see *System register viewer*, on page 25).
- `semaphores` command (see *TSC semaphore viewer*, on page 30).

In addition to these new commands, it is now also possible to display the enable state (see *Viewing the enable state*, on page 21).

3.2 Invoking the debugger

When debugging a "stand-alone" application that runs entirely on the CSX600, use the following command to start the debugger from the command line:

```
csgdb [executable-file]
```

The name of the executable file to be debugged can be passed on the command line.

The debugger initializes, prints a copyright notice and then displays a command prompt: `(gdb)`.

3.2.1 Using the debugger with a host application

When the code running on the CSX600 is loaded and used by an application running on the host, use the following method to allow both the host application and the debugger to connect to the CSX code:

1. Set the environment variables `CS_CSAPI_DEBUGGER=1` and `CS_CSAPI_DEBUGGER_ATTACH=1`.

   Setting `CS_CSAPI_DEBUGGER` initializes the debug interface inside the host application.

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

2. Start the host application and note the *port number* displayed by the host application. The full path of the file loaded is also displayed by the host application.
3. In another window, start the debugger with a command line of the form:

   ```
   csgdb csx_file_name port_number
   ```

   or:

   ```
   ddd --debugger csgdb csx_file_name port_number
   ```

   where *port_number* is the value that was displayed by the host application.
4. Set a breakpoint in the CSX code where you want to stop.
5. Press **[Run]** in `ddd`, or use the `r` command in `csgdb`.
6. Press a key in the host application window to allow it to continue.

3.3 Commands

The following sections provide an overview of the debugger commands with a detailed description of the new features for the CSX architecture. For clarity, irrelevant output from `csgdb` has been omitted from the following examples. Where necessary, these omissions are marked with ellipses (. . .).

3.3.1 Connect command and options

`csgdb` is a cross-debugger, that is, it runs on a host machine and debugs a program running on another system (the CSX processor). The `connect` command has been added to allow you to specify and initialize the connection between `csgdb` and the remote device. This connection must be made before any code can be executed within the debugger environment.

The connect command is executed from the `csgdb` prompt:

```
(gdb) connect
0x80000000 in ?? ()
(gdb)
```

When connected, `csgdb` shows the current program counter (PC) value and the symbolic information associated with that location.

There are a number of options to the connect command. These are the same as those used with `csrun`.

Long name	Short name	Valid values	Description
`--chip`	`-c`	integer	Select a given chip.
`--host`		name or address	Connect to a specified host.
`--instance`	`-i`	integer	Select a board or a simulator instance.
`--mode`	`-m`	`direct` or `attach`	Select the mode of operation.
`--sim`	`-s`		Connect to a simulator rather than hardware.

Table 3.1 Connect options summary

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

Selects the chip to connect to. Note that chip numbering starts with 1, as opposed to instancing, which is zero based. The default is 1.

--host *name* | *address*

If the daemon is running on a different host (that is, not the `localhost`), use this option to inform `csgdb` where the connection should be made.

-i *number*
--instance *number*

Specifies which instance of the board or the simulator to connect to. Instancing is 0 based. The default is 0.

-m direct | attach
--mode attach | attach

Specifies which mode `csgdb` should use to access the device. The default mode is `direct`. By specifying `attach` the device can be accessed via the debug interface built into CSAPI host applications.

-s
--sim

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

3.3.2 Loading code

The executable code to be debugged needs to be loaded into the target device. This can be done in two ways:

- If the executable file name has been passed to the debugger on the command line, it can be loaded by using the `load` command with no arguments:

```
csgdb cfitest.csx
...
(gdb) connect
0x80000000 in __FRAME_BEGIN_MONO__ ()
(gdb) load
(gdb)
```

- The executable file name can also be provided as an argument to the `load` command:

```
csgdb
...
(gdb) connect
0x80000000 in ?? ()
(gdb) load /csxtests/cfitest.csx
(gdb)
```

3.3.3 Executing code

Once code is loaded you can start the code executing. The continue command, `c`, starts the code from the entry point and runs until the program terminates:

```
csgdb
...
(gdb) connect
0x80000000 in ?? ()
(gdb) load /csxtests/cfitest.csx
(gdb) c
Continuing.

Mtap 0 has terminated.
Program exited normally.
(gdb)
```

The code can also be single stepped at the instruction or source code statement level by using the `stepi` or `step` commands. Subroutine branches can be stepped over at the instruction or source code level by using the `nexti` or `next` command.

The command `run` performs the equivalent of a `load` followed by `continue` and is very useful for starting or restarting program execution.

3.3.4 Mono debugging

`csgdb` supports the debugging of mono code at both the instruction and source code level. The standard GDB commands documented in *Debugging with GDB* [2] are all available for use when debugging mono code.

Reading mono registers – `regs` command

The `regs` command is used for reading mono registers and takes a *size* parameter to allow the registers to be viewed as 2, 4 or 8-byte values. The register size is optional. If not supplied, it defaults to 2 bytes.

For example, to display mono registers as 4-byte values:

```
(gdb) regs 4
pc      0x8001380c
ret     0x80013898
pred    0x0035
0m4     0x80000000
4m4     0x0
8m4     0x0
12m4    0x0
...
56m4    0x0
60m4    0x80013898
(gdb)
```

The three other registers (`pc`, `ret` and `pred)` are provided whenever the mono registers are displayed. These correspond to the program counter, the function return address and the predicates register.

Individual registers can be specified within the GDB command language by using register names displayed by the `regs` or `peregs` command. The register names must be preceded by a $. The register names correspond to the names in the assembler syntax for registers, without the colon (:). That is, `0:m4` becomes `$0m4` in the debugger command language syntax.

For a detailed description of the assembly mapping, refer to *Registers*, on page 35.

For example:

```
(gdb) p/x $16m4
$1 = 0x8001380c
(gdb) p/x $ret+8
$2 = 0x800138a0
(gdb)
```

For further information on the GDB command language variables, see *Debugging with GDB* [2].

Writing mono registers – `regs` command

The debugger supports the setting of values into mono registers via the command language. This is done using the following syntax:

```
(gdb) set $16m4=0x11223344
(gdb) p/x $16m4
$1 = 0x11223344
(gdb) set $16m2=0x8899
(gdb) p/x $16m4
$2 = 0x11228899
```

The values are written to the device on a restart such as a step or continue. Currently only mono registers can be written.

Reading mono memory – `x` command

The standard GDB `x` command is used to read mono memory and can be used to display a variety of formats.

Examples of using the `x` command

Displaying 3 words from address `0x80000000`:

```
(gdb) x/3w 0x80000000
0x80000000 <__FRAME_BEGIN_MONO__>:      0x00000000      0x80013898      0x00000000
(gdb)
```

Displaying 10 bytes from the current PC:

```
(gdb) x/10b $pc
0x8001380c <main+20>:   0x00    0x00    0x80    0x30    0x00    0x00    0x80    0x30
0x80013814 <main+28>:   0x0a    0x00
(gdb)
```

The symbolic register names can be passed to this command and also symbolic information from the code.

Displaying 10 bytes from `main()`:

```
(gdb) x/10b main
0x800137f8 <main>:      0x00    0x00    0xa0    0x11    0x80    0x79    0x78    0x88
0x80013800 <main+8>:    0x80    0x59
(gdb)
```

The `x` command can also be used to list instructions in the following way:

```
(gdb) x/11i main
0x800137f8 <main>:      st              0:m4            16:m4
0x800137fc <main+4>:    mono.result.get 60:m2           Return_Lo
0x80013800 <main+8>:    mono.result.get 62:m2           Return_Hi
0x80013804 <main+12>:   st              0:m4            60:m4           0x4
0x80013808 <main+16>:   st              4:m2            8:p4
0x8001380c <main+20>:   nop.poly
0x80013810 <main+24>:   nop.poly
0x80013814 <main+28>:   mono.immed      0xa                                             \
0x80013818 <main+32>:   mov             0:p2            mono_immediate                  /
0x8001381c <main+36>:   mono.immed      0x0                                             \
0x80013820 <main+40>:   mov             2:p2            mono_immediate                  /
(gdb)
```

Disassemble command

The `x` command can be used to display a number of instructions but the `disassemble` command can be used to display the assembler instructions for a whole function.

For example, to disassemble the whole of the function `main()`:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x800137f8 <main+0>:    st              0:m4            16:m4
0x800137fc <main+4>:    mono.result.get 60:m2           Return_Lo
0x80013800 <main+8>:    mono.result.get 62:m2           Return_Hi
0x80013804 <main+12>:   st              0:m4            60:m4           0x4
0x80013808 <main+16>:   st              4:m2            8:p4
...
0x80013858 <main+96>:   ld              16:m4           0:m4
0x8001385c <main+100>:  ld              60:m4           0:m4            0x4
0x80013860 <main+104>:  j.lo            60:m2                                           \
0x80013864 <main+108>:  j.hi            62:m2                                           /
End of assembler dump.
(gdb)
```

The > marks at the end of a pair of lines denote *linked* instructions. When you set a breakpoint on the second half of the linked pair, the debugger automatically moves it back one instruction. When you attempt to single step over the first part of a linked instruction, the debugger does not return until all parts of the sequence have been executed.

Breakpoints

Breakpoints are used to stop execution of code at a particular point of interest. GDB has good support for various different kinds of breakpoints and these are well documented in *Debugging with GDB* [2].

As you have full symbolic debug you can set a breakpoint on a function call. For example:

```
(gdb) b main
Breakpoint 1 at 0x8001380c: file cfitest.cn, line 27.
(gdb) c
Continuing.
Breakpoint 1, main () at cfitest.cn:27
27        poly int value = 10;
(gdb)
```

Breakpoints can also be set on a specific address in the code, as shown below:

```
Breakpoint 1, main () at cfitest.cn:27
27        poly int value = 10;
(gdb) x/4i $pc
0x8001380c <main+20>:   nop.poly
0x80013810 <main+24>:   nop.poly
0x80013814 <main+28>:   mono.immed      0xa                                     \
0x80013818 <main+32>:   mov             0:p2            mono_immediate          /
(gdb) b *0x80013814
Breakpoint 2 at 0x80013814: file cfitest.cn, line 27.
(gdb) c
Continuing.
Breakpoint 2, 0x80013814 in main () at cfitest.cn:27
27        poly int value = 10;
(gdb)
```

Symbolic debug

Full symbolic debug of functions and variables is available for mono types. Objects can be viewed using the standard commands described in *Debugging with GDB* [2].

The most common way of viewing symbolic data is with the `print` (or `p`) command.

For example, printing a mono integer variable:

```
Breakpoint 1, main () at csgdb_example.cn:22
22        mono_int++;
(gdb) print mono_int
$1 = 1
(gdb)
```

Printing a mono array and an array element:

```
Breakpoint 1, main () at csgdb_example.cn:22
22        mono_int++;
(gdb) p mono_int_array
$1 = {1000, 1001, 1002, 1003}
(gdb) p mono_int_array[3]
$2 = 1003
(gdb)
```

3.3.5 Poly debugging

`csgdb` has been extended to provide support for the poly multiplicity specifier in the **C**^{n} language and to allow visibility of the processing elements (PEs) in the CSX architecture. As with mono data, *Debugging with GDB* [2] provides the majority of the command descriptions for debugging poly code but there are some additional commands which are documented here.

Reading poly registers – `peregs` command

This command is the poly equivalent of the `regs` command.

The `peregs` command is used for reading poly registers and takes a *size* parameter to allow the registers to be viewed as 1, 2, 4 or 8-byte values. The register size is optional. If not supplied, it defaults to 1 byte.

Also, the poly registers are special as they encapsulate the value from all of the PEs in a single register value. If the value is the same across the whole array, the value is only printed once but the debugger informs you how many times the value repeats.

For example, displaying poly registers as 4-bytes values:

```
(gdb) peregs 4
0p4     {0x34 <repeats 96 times>}
4p4     {0x34 <repeats 96 times>}
8p4     {0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xa, 0xb, 0xc, 0xd, 0xe,
0xf, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c,
0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2a,
0x2b, 0x2c, 0x2d, 0x2e, 0x2f, 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38,
0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f, 0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46,
0x47, 0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f, 0x50, 0x51, 0x52, 0x53, 0x54,
0x55, 0x56, 0x57, 0x58, 0x59, 0x5a, 0x5b, 0x5c, 0x5d, 0x5e, 0x5f, 0x60, 0x61, 0x62,
0x63, 0x64, 0x65, 0x66, 0x67}
...
116p4   {0xdeaddddd <repeats 96 times>}
120p4   {0xdeadeeee <repeats 96 times>}
124p4   {0xdeadffff <repeats 96 times>}
(gdb)
```

As with the `regs` command the register names are available in the command language. The poly registers are accessed from the command line with the name preceded by a `$`, as shown below:

```
(gdb) print/d $8p4
$3 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95}
(gdb)
```

The poly registers can also be indexed to view the value on an individual PE. In this example, the values of register 8 on PEs 10 and 84 are displayed:

```
(gdb) print/d $8p4[10]
$4 = 10
(gdb) print/d $8p4[84]
$5 = 84
(gdb)
```

Reading poly memory – `pex` command

This command is the poly equivalent of the standard `x` command.

The `pex` command is used to display memory across a range of PEs using the same formats available with the `x` command. Symbolic names can also be passed to the `pex` command.

Examples of the `pex` command

Printing a word in decimal from the address `0x28` on PEs 0 to 20:

```
(gdb) pex/dw 0x28 0..20
(PE 0)    0x28 <__FRAME_BEGIN_POLY__+40>:        1000
(PE 1)    0x28 <__FRAME_BEGIN_POLY__+40>:        1001
(PE 2)    0x28 <__FRAME_BEGIN_POLY__+40>:        1002
...
(PE 18)   0x28 <__FRAME_BEGIN_POLY__+40>:        1018
(PE 19)   0x28 <__FRAME_BEGIN_POLY__+40>:        1019
(PE 20)   0x28 <__FRAME_BEGIN_POLY__+40>:        1020
(gdb)
```

Printing 4 bytes from address `&poly_int_array` on PEs 50 to 75:

```
(gdb) pex/4b &poly_int_array 50..75
(PE 50)  0x28 <__FRAME_BEGIN_POLY__+40>:        0x1a    0x04    0x00    0x00
(PE 51)  0x28 <__FRAME_BEGIN_POLY__+40>:        0x1b    0x04    0x00    0x00
(PE 52)  0x28 <__FRAME_BEGIN_POLY__+40>:        0x1c    0x04    0x00    0x00
...
(PE 73)  0x28 <__FRAME_BEGIN_POLY__+40>:        0x31    0x04    0x00    0x00
(PE 74)  0x28 <__FRAME_BEGIN_POLY__+40>:        0x32    0x04    0x00    0x00
(PE 75)  0x28 <__FRAME_BEGIN_POLY__+40>:        0x33    0x04    0x00    0x00
(gdb)
```

Viewing the enable state

When debugging applications that use poly conditional code, it is useful to see the enable state of the PEs. In `csgdb`, the enable state is mapped into a register name `$enable`. This register can be viewed using the standard `print` command.

Displaying the PE enable state info

Printing the enable state at `main`, where all PEs are enabled:

```
Breakpoint 1, main () at csgdb_example.cn:4
4          mono int mono_int = 0;
(gdb) p/x $enable
$1 = {0xff <repeats 96 times>}
(gdb)
```

Printing the enable state inside a poly conditional:

```
Breakpoint 2, main () at csgdb_example.cn:23
23           mono_int++;
(gdb) p/x $enable
$2 = {0xfe <repeats 48 times>, 0xff <repeats 48 times>}
(gdb)
```

Printing the enable state as binary to see all eight levels within each PE:

```
(gdb) p/t $enable
$3 = {11111110 <repeats 48 times>, 11111111 <repeats 48 times>}
(gdb)
```

Printing a simplified view of the enable state can be done as follows with + denoting enabled and – denoting disabled:

```
Breakpoint 4, main () at simple.cn:8
8            X = 10;
(gdb) p $enabled
$1 = '-' <repeats 49 times>, '+' <repeats 47 times>
(gdb)
```

The number of enabled PEs can be displayed by doing the following:

```
(gdb) p $numenb
$3 = 47
```

In a similar fashion, the total number of PEs can be retrieved:

```
(gdb) p $numpes
$4 = 96
```

The number of disabled PEs can be worked out using the command language:

```
(gdb) p $numpes-$numenb
$5 = 49
```

Displaying the PE mac status info

This is done by viewing the following command language variable:

```
(gdb) p/x $status
$6 = {0xc9 <repeats 48 times>, 0xd2, 0xc2 <repeats 47 times>}
(gdb) p/t $status
$7 = {11001001 <repeats 48 times>, 11010010, 11000010 <repeats 47 times>}
(gdb)
```

Displaying the PE fpadd status

The `fpadd` status is displayed by viewing the following command language variable:

```
(gdb) p/x $fpadd
$8 = {0xc0 <repeats 96 times>}
(gdb) p/t $fpadd
$9 = {11000000 <repeats 96 times>}
```

Displaying the PE fpmul status

To see the current `fpmul` status, issue the following command:

```
(gdb) p/x $fpmul
$10 = {0xc2 <repeats 96 times>}
(gdb) p/t $fpmul
$11 = {11000010 <repeats 96 times>}
```

Symbolic debug

Full symbolic debug of variables is available for poly types. Objects can be viewed using the standard commands described in the GDB reference value. Poly variables are special inside the debugger as they are expanded to display the value on every PE.

For example, displaying the value of a poly integer produces the following result:

```
(gdb) p poly_int
$4 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95}
(gdb)
```

The value of the variable `poly_int` from each PE is displayed.

A poly array produces similar results with the array values displayed from each PE. For example, to display a four element poly integer array.

```
(gdb) p poly_int_array
$2 = {{1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010, 1011, 1012,
1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026,
1027, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040,
1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051, 1052, 1053, 1054,
1055, 1056, 1057, 1058, 1059, 1060, 1061, 1062, 1063, 1064, 1065, 1066, 1067, 1068,
```

```
1069, 1070, 1071, 1072, 1073,  1074, 1075, 1076, 1077, 1078, 1079, 1080, 1081, 1082,
1083, 1084, 1085, 1086, 1087, 1088, 1089, 1090, 1091, 1092, 1093, 1094, 1095}, {2000,
2002, 2004, 2006, 2008, 2010, 2012, 2014, 2016, 2018, 2020, 2022, 2024, 2026, 2028,
2030, 2032, 2034, 2036, 2038, 2040, 2042, 2044, 2046, 2048, 2050, 2052, 2054, 2056,
2058, 2060, 2062, 2064, 2066, 2068, 2070, 2072, 2074, 2076, 2078, 2080, 2082, 2084,
2086, 2088, 2090, 2092, 2094, 2096, 2098, 2100, 2102, 2104, 2106, 2108, 2110, 2112,
2114, 2116, 2118, 2120, 2122, 2124, 2126, 2128, 2130, 2132, 2134, 2136, 2138, 2140,
2142, 2144, 2146, 2148, 2150, 2152, 2154, 2156, 2158, 2160, 2162, 2164, 2166, 2168,
2170, 2172, 2174, 2176, 2178, 2180, 2182, 2184, 2186, 2188, 2190}, {3000, 3003, 3006,
3009, 3012, 3015, 3018, 3021, 3024, 3027, 3030, 3033, 3036, 3039, 3042, 3045, 3048,
3051, 3054, 3057, 3060, 3063, 3066, 3069, 3072, 3075, 3078, 3081, 3084, 3087, 3090,
3093, 3096, 3099, 3102, 3105, 3108, 3111, 3114, 3117, 3120, 3123, 3126, 3129, 3132,
3135, 3138, 3141, 3144, 3147, 3150, 3153, 3156, 3159, 3162, 3165, 3168, 3171, 3174,
3177, 3180, 3183, 3186, 3189, 3192, 3195, 3198, 3201, 3204, 3207, 3210, 3213, 3216,
3219, 3222, 3225, 3228, 3231, 3234, 3237, 3240, 3243, 3246, 3249, 3252, 3255, 3258,
3261, 3264, 3267, 3270, 3273, 3276, 3279, 3282, 3285}, {4000, 4004, 4008, 4012, 4016,
4020, 4024, 4028, 4032, 4036, 4040, 4044, 4048, 4052, 4056, 4060, 4064, 4068, 4072,
4076, 4080, 4084, 4088, 4092, 4096, 4100, 4104, 4108, 4112, 4116, 4120, 4124, 4128,
4132, 4136, 4140, 4144, 4148, 4152, 4156, 4160, 4164, 4168, 4172, 4176, 4180, 4184,
4188, 4192, 4196, 4200, 4204, 4208, 4212, 4216, 4220, 4224, 4228, 4232, 4236, 4240,
4244, 4248, 4252, 4256, 4260, 4264, 4268, 4272, 4276, 4280, 4284, 4288, 4292, 4296,
4300, 4304, 4308, 4312, 4316, 4320, 4324, 4328, 4332, 4336, 4340, 4344, 4348, 4352,
4356, 4360, 4364, 4368, 4372, 4376, 4380}}
(gdb)
```

An individual array element can be viewed as follows:

```
(gdb) p poly_int_array[2]
$4 = {3000, 3003, 3006, 3009, 3012, 3015, 3018, 3021, 3024, 3027, 3030, 3033, 3036,
3039, 3042, 3045, 3048, 3051, 3054, 3057, 3060, 3063, 3066, 3069,3072, 3075, 3078,
3081, 3084, 3087, 3090, 3093, 3096, 3099, 3102, 3105, 3108, 3111, 3114, 3117, 3120,
3123, 3126, 3129, 3132, 3135, 3138, 3141, 3144, 3147, 3150, 3153, 3156, 3159, 3162,
3165, 3168, 3171, 3174, 3177, 3180, 3183, 3186, 3189, 3192, 3195, 3198, 3201, 3204,
3207, 3210, 3213, 3216, 3219, 3222, 3225, 3228, 3231, 3234, 3237, 3240, 3243, 3246,
3249, 3252, 3255, 3258, 3261, 3264, 3267, 3270, 3273, 3276, 3279, 3282, 3285}
(gdb)
```

The symbolic names of poly variables can also be passed to the `pex` command. It uses the address of the variable and displays the data accordingly.

The standard GDB `whatis` command displays the type and multiplicity of a variable:

```
(gdb) whatis poly_int_array
type = poly int [4]<96 PEs>
(gdb)
```

The information `<96 PEs>` describes how many elements the variable is visible over and the type displays the poly keyword.

3.3.6 Hardware threads

`csgdb` has extended the threading support in GDB to allow it to debug the threads supported by the hardware. After loading an application with multiple threads the debugger can see the state of all threads in the system.

The state of the threads can be seen by using the `info threads` command:

```
(gdb) info threads
* 8 Software Thread 7  0x02003b18 in _start7 ()
  7 Software Thread 6  _start6 () at thread_test.is:68
  6 Software Thread 5  _start5 () at thread_test.is:87
  5 Software Thread 4  _start4 () at thread_test.is:102
  4 Software Thread 3  _start3 () at thread_test.is:117
  3 Software Thread 2  _start2 () at thread_test.is:132
  2 Software Thread 1  _start1 () at thread_test.is:147
  1 Software Thread 0  _start () at thread_test.is:162
0x02003b18 in _start7 ()
(gdb)
```

The currently executing thread is marked with `*`.

Each thread has an identifier assigned to it by `csgdb` and these are used to select which thread to view. The software thread number is displayed after the GDB identifier. In the example above, the currently executing thread has the GDB identifier 8 and is software thread 7.

The `thread` command is used to select a thread.

```
(gdb) (gdb) thread 2
[Switching to thread 2 (Software Thread 1)]#0  _start1 () at thread_test.is:147
147    sem.wait SEM_SIG6
(gdb) list
142    sem.wait SEM_THREAD_FINISH
143
144_start1::
145.global _start1
146
147    sem.wait SEM_SIG6
148    mov  32:m2, 0
149_testloop6::
150.global _testloop6
151    j.ifn 32:m2, _testloop6
(gdb)
```

All following commands are applied to the currently selected thread unless otherwise specified. For example, commands can be executed on all threads in the following way:

```
(gdb) thread apply all x/i $pc

Thread 8 (Software Thread 7):
0x2003b18 <_start7>:sem.put         77              4

Thread 7 (Software Thread 6):
0x200312c <_start6>:mov             16:m2           0x20

Thread 6 (Software Thread 5):
0x200317c <_start5>:sem.select      36

Thread 5 (Software Thread 4):
0x20031b0 <_start4>:sem.select      37
```

```
Thread 4 (Software Thread 3):
0x20031e4 <_start3>:sem.select      38

Thread 3 (Software Thread 2):
0x2003218 <_start2>:sem.select      39

Thread 2 (Software Thread 1):
0x200324c <_start1>:sem.select      3A

Thread 1 (Software Thread 0):
0x2003280 <_start>:sem.select      3B
#0  0x02003b18 in _start7 ()
```

You can change the view of whether csgdb displays the software or hardware mapping of the threads.To view the hardware mapping, use the command:

```
set print cs_hardware_thread_view on
```

When any of the thread commands are used with this option set, the debugger displays the hardware thread identifiers.

```
(gdb) info threads
* 8 Hardware Thread 0  0x02003b18 in _start7 ()
  7 Hardware Thread 1  _start6 () at thread_test.is:68
  6 Hardware Thread 2  _start5 () at thread_test.is:87
  5 Hardware Thread 3  _start4 () at thread_test.is:102
  4 Hardware Thread 4  _start3 () at thread_test.is:117
  3 Hardware Thread 5  _start2 () at thread_test.is:132
  2 Hardware Thread 6  _start1 () at thread_test.is:147
  1 Hardware Thread 7  _start () at thread_test.is:162
0x02003b18 in _start7 ()
```

To disable this mode and to view the software mapping, use the command:

```
set print cs_hardware_thread_view off
```

3.3.7 System register viewer

The debugger lets you view the system register present in the CSX600 device. The values and fields contained within them are presented in a form that can be clearly understood. To display this information, use the command `sysreg` which has been added to csgdb.

Getting help in csgdb

The `sysreg` command has limited online help which can be accessed with the `help` argument to `sysreg`.

```
(gdb) sysreg help

***** MTAP system register viewer help *****

Command  Arguments                          Description
-------   ---------                         -----------
list     {reg group}     | {reg full name}   List system register information

display  {reg group}     | {reg full name}   Display system register values
                         | {address}

set      {reg full name} | {address} {value} Write to system registers
         {reg full name} | {bitfield}{value}

return   {reg full name} | {address} {var}   Return register value to csgdb variable

(gdb)
```

Note: Chip select bits are not applied when an immediate address is specified.

The help option lists the other available options to the command and the arguments that go with them.

Listing system register information

The `list` option lets you view the information about the register groups, registers contained within the groups and the bit field information for individual registers. For example, to get a list of all registers groups use the `list` option with no arguments.

```
(gdb) sysreg list

System register group list :

Group Name
----------
```

```
CCBRS
CCBRS_P0
CCBRS_P1
ISU
ISU_GIU
ISU_GSU
ISU_IG
LMI
LMI_DMA
LMI_DMA_AEU
LMI_LMICOM
LMI_LMIRIF
LMI_LMISRV
MTAP
MTAP_AC
MTAP_AC_DB
MTAP_AC_DB_CM
MTAP_AC_IT
MTAP_AC_MS
MTAP_GPIOC0
MTAP_GPIOC0_GPIOC
MTAP_GPIOE0
MTAP_GPIOE0_GPIOE
MTAP_TSC
MTAP_TSC_ICACHE
MTAP_TSC_ICACHE_IBUFFER
MTAP_TSC_LSU
MTAP_TSC_SCHED
MTAP_TSC_SEM
MTAP_TSC_TP
MTAP_TSC_TP_DP
MTAP_TSC_TP_TPREG
SYS

(gdb)
```

This command lists all the available register groups that are visible to the debugger.

Viewing the information about a register group

You can view the register groups themselves by passing the name of the group along with the `list` argument. The register group names are displayed when you use the `list` argument on its own.

For example, to display the contents of the `MTAP_TSC_SCHED` group, enter the following command:

```
(gdb) sysreg list MTAP_TSC_SCHED

System register list for group MTAP_TSC_SCHED :

Register Name
-------------
CONTROL
STATUS
SWITCH_THREAD

(gdb)
```

This lists the register names that are contained within the group.

Listing the information about a specific register

The `list` option also lets you display information about a specific register and by passing the full register name (in the format `group_name`) along with the `list` argument.

For example, to view the information for the `STATUS` register of the `MTAP_TSC_SCHED` group:

```
(gdb) sysreg list MTAP_TSC_SCHED STATUS

System register definition
--------------------------
Name :          STATUS
Group :         MTAP_TSC_SCHED
Description :   Status of each thread
Reset Value :   255
Address :       0x102

Bit Fields
----------
Name :  READY
First : 0
Last :  7
Width : 8

Name :  THREAD
First : 16
Last :  18
Width : 3

Name :  YIELD
First : 8
Last :  15
Width : 8

(gdb)
```

This command lists the definition and bit field information for the requested register.

Displaying system register values

As well as displaying the information regarding the make up of each of the groups and registers, it is possible to display the values contained within them. The values can be displayed at a group level or also at an individual register level to allow the bit-field values to be displayed. This is done by using the `display` argument to the `sysreg` command.

Displaying the values of a register group

The values of all registers contained within a register group can be displayed using the `display` argument with a valid register group name. For example:

```
(gdb) sysreg display MTAP_TSC_SCHED

System register display for group MTAP_TSC_SCHED :

Register Name                   Value
-------------                   -----
CONTROL                         0x8
STATUS                          0x70000
SWITCH_THREAD                   0x7

(gdb)
```

Displaying the value of an individual register

The value of a register can be displayed by passing a valid register name or hex address with the `display` argument. For example:

```
(gdb) sysreg display MTAP_TSC_SCHED STATUS

System register display for MTAP_TSC_SCHED_STATUS :

Value : 0x70000 0b00000000000001110000000000000000

Bit Fields                      Value
----------                      -----
READY                           0x0
THREAD                          0x7
YIELD                           0x0

(gdb)
```

Returning a register value to a GDB variable

The `return` option lets you store the value of a register in a GDB variable for use in a script.

The following example prints the value of register `MTAP_TSC_TPREG_REGISTER_R1` through the GDB variable `$regval`:

```
(gdb) sysreg return MTAP_TSC_TPREG_REGISTER_R1 regval
(gdb) print $regval
```

Writing to registers

The `set` option lets you write values to system registers.

Full register names or hex addresses can be specified. As well as writing to whole registers, valid bit field names can be specified when only part of a register requires writing to.

```
(gdb) set MTAP_TSC_TPREG_REGISTER_R1 123
(gdb) set 06700A84 123
```

3.3.8 TSC semaphore viewer

When synchronizing code between the TSC and the host processor of separate threads within the TSC, semaphores are used. The debugger lets you list the information about the semaphores of the TSC it is connected to. The command, `semaphores`, has been added to csgdb to let you view the semaphore information. The command can also be shortened to `sem` and the command language takes care of extending the name.

Getting help in csgdb

The `semaphores` command has limited online help which can be accessed with the `help` argument to `semaphores`.

```
(gdb) sem help

***** TSC semaphore viewer help *****

Use help {command} for more detailed command specific instruction.

Command    Arguments                       Description
-------    ---------                       -----------
display    {semaphore} | all | allval      Display all semaphore information
value      {semaphore} | all | allval      Display the current semaphore value(s)
nonzero    {semaphore} | all | allnon      Display the current nonzero status(s)
interrupt  {semaphore} | all | allint      Display the current interrupt enable
status(s)
overflow   {semaphore} | all | allovr      Display the current overflow status
thread     {thread} | all                 Display the current thread / semaphore use

(gdb)
```

Displaying semaphore information

Using the `semaphores` command with the display option lists all information about a particular set of semaphores. The arguments to this command can be an individual semaphore, all semaphores or all semaphores which currently have a value. Use the `all` argument to list the information. For example, to display information about all TSC semaphores:

```
(gdb) sem display all

Semaphore    Value    NonZero    Interrupt    Overflow
---------    -----    -------    ---------    --------
0            0        0          0            0
1            0        0          0            0
2            0        0          0            0
3            0        0          0            0
4            0        0          0            0
...
...
...
123          0        0          0            0
124          0        0          0            0
125          0        0          0            0
126          0        0          0            0
127          0        0          1            0
```

There are 128 semaphores and this command lists all the information about all of them.

Listing information for all semaphores with a current value

The `semaphores` command with the `display` argument can be used to list information about only those semaphores with a value. This is done by passing the `allval` argument to the command.

```
(gdb) sem display allval

Semaphore     Value     NonZero     Interrupt     Overflow
---------     -----     -------     ---------     --------
30            3         1           0             0
34            2         1           0             0
36            1         1           0             0
39            4         1           0             0
45            1         1           0             0
55            2         1           0             0
56            1         1           0             0
68            1         1           0             0
119           4         1           0             0
```

Listing semaphore information for an individual semaphore

To limit the information displayed, the debugger can list information for an individual semaphore. This is done by passing the semaphore number to the `semaphores display` command.

```
(gdb) sem display 34

Semaphore     Value     NonZero     Interrupt     Overflow
---------     -----     -------     ---------     --------
34            2         1           0             0
```

Displaying only the values of semaphores

The following examples show the use of the `value` argument to the `semaphores` command. Use the options as follows:

- `all` displays the current value of all semaphores.
- `allval` displays the values for the semaphores which have a value greater than 0.
- A semaphore number displays the information for just that numbered semaphore.

```
(gdb) sem value all

Semaphore           Value
---------           -----
0                   0
1                   0
2                   0
3                   0
4                   0
...
...
...
123                 0
124                 0
125                 0
126                 0
127                 0
```

```
(gdb) sem value allval

Semaphore           Value
---------           -----
30                  3
34                  2
36                  1
39                  4
45                  1
55                  2
56                  1
68                  1
119                 4

(gdb) sem value 30

Semaphore           Value
---------           -----
30                  3
```

Displaying only the nonzero status of the semaphores

The debugger can list the nonzero status of each of the semaphores. This is done by passing the `nonzero` option to the `semaphores` command. The arguments that work with the `nonzero` option are:

- `all` to list the nonzero fields for all semaphores.
- `allnon` to list just those with values in the nonzero field.
- An individual semaphore number.

```
(gdb) sem nonzero all

Semaphore           NonZero
---------           -------
0                   1
1                   1
2                   1
3                   1
4                   1
...
...
...
123                 1
124                 1
125                 1
126                 1
127                 0

(gdb) sem nonzero allnon
```

```
Semaphore           NonZero
---------           -------
0                   1
1                   1
2                   1
3                   1
4                   1
...
...
...
122                 1
123                 1
124                 1
125                 1
126                 1

(gdb) sem nonzero 125

Semaphore           NonZero
---------           -------
125                 1
```

Displaying the interrupt enable status of the semaphores

It is possible to list the interrupt enable status of each of the semaphores. This is done by passing the `interrupt` option to the `semaphores` command. The arguments that work with the `nonzero` option are:

- `all` to list the interrupt fields for all semaphores.
- `allint` to list just those with values in the interrupt field.
- An individual semaphore number.

```
(gdb) sem interrupt all

Semaphore           Interrupt
---------           ---------
0                   0
1                   0
2                   0
3                   0
4                   0
...
...
...
123                 0
124                 0
125                 0
126                 0
127                 1

(gdb) sem interrupt allint

Semaphore           Interrupt
---------           ---------
127                 1

(gdb) sem interrupt 120

Semaphore           Interrupt
---------           ---------
120                 0
```

Displaying the overflow status of the semaphores

The debugger can display the overflow status of each of the semaphores. This is done by passing the `overflow` option to the `semaphores` command. The arguments that work with the `overflow` option are:

- `all` to list the overflow fields for all semaphores.
- `allovr` to list just those with values in the overflow field.
- An individual semaphore number.

```
(gdb) sem overflow all

Semaphore           Overflow
---------           --------
0                   0
1                   0
2                   0
3                   0
4                   0
...
...
...
123                 0
124                 0
125                 0
126                 0
127                 0

(gdb) sem overflow allovr

Semaphore           Overflow
---------           --------
10                  1
18                  1
123                 1

(gdb) sem overflow 20

Semaphore           Overflow
---------           --------
20                  0
```

Displaying the current thread / semaphore usage

It is also possible to view the semaphore usage for the TSC threads so that you can see which semaphore number is currently selected. This is done using the `thread` argument to the `semaphores` command. The `thread` argument takes sub options of either `all` or a semaphore number to specify what to display.

```
(gdb) sem thread all

Thread          Semaphore
------          ---------
0               10
1               23
2               43
3               101
4               102
5               32
6               22
7               25

(gdb) sem thread 1

Thread          Semaphore
------          ---------
1               23

(gdb) sem thread 2

Thread          Semaphore
------          ---------
2               43
```

3.4 Registers

Table 3.2 shows the register mapping for the debugger and assembly language.

Debugger register name	Assembler register name	Description
$pc	n/a	Program counter
$pred	n/a	Predicates register
$pestat	n/a	Full PE status register
$ret	n/a	Return register
$status	n/a	Nonfloating point PE status
$enable	n/a	Enable state (all levels)
$fpadd	n/a	PE floating point add status
$fpmul	n/a	PE floating point mul status
$enabled	n/a	PE enabled register
$numenb	n/a	Number of enabled PE's
$numpes	n/a	Number of PE's
$0m2 ... $62m2	0:m2 ... 62:m2	2 byte mono registers

Table 3.2 Debugger to Assembly Language Register Mapping

Debugger register name	Assembler register name	Description
$0m4 ... $60m4	0:m4 ... 60:m4	4 byte mono registers
$0m8 ... $56m2	0:m8 ... 56:m8	8 byte mono registers
$0p1 ... $127p1	0:p1 ... 127:p1	1 byte poly registers
$0p2 ... $126p2	0:p2 ... 126:p2	2 byte poly registers
$0p4 ... $124p4	0:p4 ... 124:p4	4 byte poly registers
$0p8 ... $120p8	0:p8 ... 120:p8	8 byte poly registers
$0p16 ... $112p16	0:p16 ... 112:p16	4 byte poly vector registers
$0p32 ... $96p32	0:p32 ... 96:p32	8 byte poly vector registers

Table 3.2 Debugger to Assembly Language Register Mapping

3.5 Using DDD

As `csgdb` is a port of the GDB debugger for the CSX architecture, it is possible to use the standard Linux DDD graphical user interface (GUI). This is not provided as part of the SDK but can be used if installed. The DDD GUI comes as standard on most Linux systems. For Windows, you need to have the cygwin tools installed.

You can start DDD by using `csgdb` as follows:

```
> ddd -debugger csgdb [executable-file]
```

Once it has been started you must connect `csgdb` to the target as described in the section *Connect command and options*, on page 14.

After this you can set a breakpoint and select **[Run]** in DDD. You can then debug the application using the GUI front end.

4 Host interface library

This chapter describes the CSX600 driver library and the ClearSpeed Application Programming Interface (CSAPI) functions.

To load code onto a CSX600 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 board*s* via the CSAPI. The user-level libraries make use of a kernel-level driver to provide a complete driver for the Advance boards.

4.1 CSX600 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 and in some instances the individual processor on the board, most of the CSAPI functions take a state pointer which describes the board and, if necessary, the processor. To use the CSAPI interface, a `CSAPI_new` call must be made to build and return this state variable.

The library is thread safe. It is safe for concurrent threads to access the library but this is only guaranteed if the same `CSAPIState` instance is used. The CSAPI library uses host semaphores and other concurrency objects to provide concurrent but safe access to the Advance boards.

4.2 Linking host applications with CSAPI

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

- In Linux: `libcleard_stud_lib.a`
- In Windows XP: `cleard_stub_lib.lib`

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

4.2.1 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 simple CSAPI program as follows:

```
gcc -I $CSHOME/include/cs_api -L $CSHOME/lib simple.c -lcleard_stub_lib -ldl
```

The `-ldl` option links in the dynamic loader library (`libdl.a`) which allows the `stublib` 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. No other libraries need to be loaded for CSAPI functionality.

4.2.2 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 setup the environment using the `setup_env.bat` script and then invoke Visual Studio with the option `/useenv`. This will use the appropriate environment setup.

Note: Currently you need to add the `CSAPI INCLUDE` directory to the INCLUDE path.

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. Setup the environment using the ClearSpeed Script as follows:
   a. Execute setup_env within the ClearSpeed install directory:
      `setup_en`
   b. Add the `INCLUDE` directory as follows:
      `set INCLUDE=%CSHOME%\include;%INCLUDE`
3. Invoke VC++ with this environment:
   `msdev /useenv`
4. Setup the development project as described in Table 4.1.

Task	Steps
Select a new project	Select Win32 Console Application as an example.
Set Project settings: static library location	1. Specify the location of the static component of the runtime library, that is, the file `cleard_stub_lib.lib`. 2. Select `Project->Settings` and select the Link tab. 3. Add `%CSHOME%\lib` to the "Additional library path". 4. Add the file `cleard_stub_lib.lib` to the list of library modules.
Set Project type	1. Select `Project->Settings` and select the C/C++ tab and the category Code Generation. 2. Select the MultiThreaded option. This is needed as the runtime is built as a multithreaded application

Table 4.1 Setting up the development project

3. Add the source file:
   Use `File->Add` to add a C/C++ file to the project

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

4.3 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 CSX600 processors on one or more Advance boards (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.

4.3.1 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.

4.3.2 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. Therefore, you must call `CSAPI_new` before using the rest of the CSAPI functions.

The one exception to this is `CSAPI_version`. You can call this 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` defined above. 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.

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, `libcleard_stub_lib.a` or `cleard_stub_lib.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.

After calling `CSAPI_new`, you can call `CSAPI_num_cards` passing the `CSAPIState` object, to obtain the number of Advance boards installed in the system. You can then make further calls to `CSAPI_new`, depending on the number of boards or simulators that you want to use. One `CSAPIState` object must be created for each board or simulator used by the host application.

You can then connect each `CSAPIState` object to a board or simulator using the `CSAPI_connect` function. You *must* check the return code to ensure that the connection was successful. If the return code is not equal to `DRVErrno_success`, you can pass it to the `CSAPI_get_error_string` function, which will fill a provided char array with an error string corresponding to the return code. If the `CSAPI_connect` function failed because the board or simulator was already in use, the error string will contain the user name and process ID currently connected to the board or simulator.

Access control

Access to boards 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 Windows systems, it is stored in the installation directory. 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 board or simulator is still running.

Initialization

Once connected you can call `CSAPI_reset` on each of the processors (see *Obtaining information* 4.3.3 for information on how to determine the number of processors). This avoids the need to run `csreset` before the application is run, but a call to `CSAPI_reset` will add a small 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 board and driver. This function is provided for debugging purposes and is not needed in normal use. Some of the parameters require the board to be reset after they have been configured, so you may need to call `CSAPI_reset` again after reconfiguration.

4.3.3 Obtaining information

Once a `CSAPIState` object has been created and connected to a board, you can call various CSAPI functions to obtain information about the processors on the board:

- `CSAPI_version` can be called to obtain the processor, firmware and kernel driver versions.
- `CSAPI_num_processors` can be called to determine the number of processors on the board.
- `CSAPI_num_pes` can be called to determine the number of processing elements on each processor.
- `CSAPI_num_semaphores` can be called to determine the total number of TSC semaphores on each processor.
- `CSAPI_num_threads` can be called to determine the total number of threads on each processor.
- `CSAPI_endianness` can be called to determine the endianness of each processor.

4.3.4 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 board. 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 available address on the board.

You can load multiple dynamically linked CSX programs at the same time. After each program is loaded, call `CSAPI_get_last_loaded_handle` to obtain a `DRVProcess` handle for the program. You can then use this handle with the functions `CSAPI_run_process` and `CSAPI_get_symbol_value_loaded`.

If the program was linked statically, it is assumed that there is only one program loaded on the board. In this case, you can call the `CSAPI_run` and `CSAPI_get_symbol_value` functions without needing a process handle.

It is not possible to load a statically linked CSX program if the memory allocation functions have been used (see 4.3.9, *Memory allocation*, on page 43). The `CSAPI_load` function will return an error indicating that a dynamic allocation of memory has already been made.

Running the CSX program

Once a CSX program has been loaded, you can run it by calling either `CSAPI_run` or `CSAPI_run_process`, depending on whether the program was statically or dynamically linked. 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 can call `CSAPI_run` and `CSAPI_run_process` multiple times after a CSX program has been loaded. However, it should be noted that static variables in the CSX program will only be initialized when the program is loaded. The static variables will retain their values between runs, and should be explicitly reinitialized in the CSX code where necessary. Local variables and dynamically allocated memory are not preserved between runs so their values will not be retained (see the Memory allocation section below).

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`.

4.3.5 Unloading a program and disconnecting

If the CSX program was linked dynamically, you can unload it by calling `CSAPI_unload` with the `DRVProcess` 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 board, you can delete the `CSAPIState` for the board by calling `CSAPI_delete`. This will disconnect from the board and destroy the state object. All programs loaded on the board and all memory allocations on the board will be discarded. When the last `CSAPIState` object has been deleted, `CSAPI_delete` will unload the `cleard` library.

Always call `CSAPI_delete` before exiting your host application so that the board 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 board 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.

Example application

The following code will:

1. Create a CSAPI state object.
2. Connect to a board.

3. Reset processor zero.
4. Load and run a CSX program.
5. Wait for the CSX program to finish and obtain the exit code.

```
void main()
{
    DRVErrno return_code;
    struct CSAPIState *s;
    int csx_exit_code;
    unsigned int proc_inx = 0;

    s = CSAPI_new( CM_Direct );

    return_code = CSAPI_connect( s , NULL , CSAPI_INSTANCE_ANY );

    return_code = CSAPI_reset( s , proc_inx , 0 );

    return_code = CSAPI_load( s , proc_inx , CSX_FILE_NAME );

    return_code = CSAPI_run( s , proc_inx );

    return_code = CSAPI_wait_on_terminate( s , proc_inx );

    return_code = CSAPI_get_return_value( s , proc_inx , &csx_exit_code );

    CSAPI_delete( s );
}
```

The return code from each CSAPI call is not checked in this simple example. For a more complete example, look at the Mandelbrot source code provided as part of the runtime installation.

4.3.6 Events

If you want the host application to respond to an event, you need to register a callback for the event. You also need to call the `CSAPI_register_application` function before registering any callbacks.

The current 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 identified by numbers, which are defined in the `csapi.h` header file. The following events are defined:

1. Break - Triggered when the CSX program hits a break point.
2. Terminate - Triggered when the CSX program terminates.
3. Print - Triggered by a call to `printf` from the board.
4. Stack overflow - Triggered if the call stack in the CSX program overflows.
5. Semaphore nonzero - Triggered when a TSC semaphore is signalled.
6. Semaphore overflow - Triggered when a TSC semaphore overflows because it was signalled too many times.
7. Malloc - Triggered by a call to `malloc` in the CSX program.

Registering callbacks is not necessary for normal use, so you do not usually need to call `CSAPI_register_application`, `CSAPI_get_callback` or `CSAPI_register_callback`.

4.3.7 Semaphores

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

You must register semaphores that the host will wait on before using them, by calling the `CSAPI_register_semaphore` function. The host can only wait on semaphores that have been registered. Calling `CSAPI_semaphore_signal` on a semaphore that has been registered will return the error code `DRVErrno_semaphore_registered`. Calling `CSAPI_semaphore_wait` on a semaphore that has not been registered will return the error code `DRVErrno_semaphore_not_registered`. The CSX program must not wait on a semaphore that has been registered for use by the host.

4.3.8 Symbols

Symbols are typically global variables or entry points in the CSX program loaded on the board. The `CSAPI_get_symbol_value` and `CSAPI_get_symbol_value_loaded` functions return the address of the symbol on the board (for statically and dynamically linked CSX code, respectively). This address could be the start of a fixed-size 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 board.

If you loaded a statically linked CSX program, you should call the `CSAPI_get_symbol_value` function with the name of the appropriate `.csx` file. The `.csx` file will contain the fixed address at which the symbol is located.

If you loaded a dynamically linked CSX program, you should call the `CSAPI_get_symbol_value_loaded` function with the `DRVProcess` handle for the program. This will allow the relocated address of the symbol to be determined.

If the symbol is a pointer to memory that has not yet been allocated, you can give it a value by calling the CSAPI memory allocation functions described in 4.3.9, *Memory allocation*, on page 43. This avoids the need to allocate memory and write the result to a symbol using several CSAPI calls.

4.3.9 Memory allocation

The CSAPI memory allocation functions currently only allocate memory from the DRAM attached to each processor. The embedded SRAM is not managed by the driver. Dynamically linked CSX programs will only be loaded to the DRAM memory.

Both CSX processors on the Advance board can access all of the memory on the board. Each processor can access its own DRAM and the DRAM attached to the other processor. However, accessing DRAM attached to another processor adds latency and will increase bus contention if both processors frequently access each other's DRAM. 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.

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

You can use the `CSAPI_get_free_mem` function to obtain the total available DRAM 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 DRAM 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.

The CSAPI allocation functions cannot be used once a statically linked program has been loaded on to the board. The allocation functions will return an error indicating that a statically linked program has already been loaded.

Memory allocation

Memory can be allocated in three ways:

- CSX programs containing static arrays can be loaded on to the board. These static arrays will be contained within the memory allocated for the loaded program.
- 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 board.
  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.
- The **C**″ 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 stack is placed on top of the allocated memory blocks, where it can grow upwards (see Figure 4.1 on page 45). 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 4.1 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.

Figure 4.1 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 most recently loaded CSX program. 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`).

The CSAPI allocation functions cannot be used once a statically linked CSX program has been loaded. They will return an error indicating that a statically linked program has already been loaded.

Memory allocated with the CSAPI allocation functions can be released by calling `CSAPI_free`. 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 board. 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`. 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 terminates any memory allocated by `malloc` will be released.

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.

4.3.10 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 board (*mono* memory). The address on the board is usually obtained from a CSAPI memory allocation or a call to `CSAPI_get_symbol_value`. You would use the latter when the symbol is a static array in the CSX program or a pointer containing an address returned by `malloc`.

The `CSAPI_read_mono_memory` and `CSAPI_write_mono_memory` functions will flush the data cache to ensure cache coherency and halt the processor while the data is being transferred. Halting the processor avoids any performance impact due to the processor accessing DRAM at the same time as the data is transferred to or from the host. Halting the processor will cause a delay in the CSX program while the data is being transferred. You can avoid this by using `CSAPI_read_mono_memory_raw` and `CSAPI_write_mono_memory_raw`. These functions do not flush the data cache or halt the processor. These should be used when you know that the CSX processor will not be accessing the DRAM while the data is transferred.

Asynchronous transfers

The above four mono memory functions will block until the data transfer has completed. Blocking the main host thread can be avoided by calling the memory transfer functions in separate host threads, allowing the main host application to continue during the transfer. CSAPI also 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 four functions to acknowledge when an asynchronous transfer has completed. The first two, `CSAPI_read_mono_memory_async_wait` and `CSAPI_write_mono_memory_async_wait`, will block until the corresponding read or write operation has completed. When these functions return, another call to `CSAPI_read_mono_memory_async` or `CSAPI_write_mono_memory_async` can be made without blocking.

Two further functions are provided for polling the status of the asynchronous transfer. These are `CSAPI_read_mono_memory_async_poll` and `CSAPI_write_mono_memory_async_poll`. These functions return immediately and return a value to indicate the status of the transfer. The first time these functions are called after the transfer has completed, they will indicate that the transfer has completed. Calling the functions again after this will indicate that the transfer has *not* completed until another transfer has started and completed.

Register access

The following functions are used to access control registers in the 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.

4.4 Example of host and CSX code cooperation

A simple example of both the host and the CSX600 processor assembly code is presented below. It shows how to designate an area of mono memory and synchronize host and board processing.

Note: this example does not include as much error checking as a real program would require to make it robust. After every statement of the form `status = CSAPI...` the returned value should be checked to ensure it is equal to `DRVErrno_success`, or to report an error if it is not.

```
.section .mono.bss

_SHARED_MEMORY_::
.fill 16,1,0                  // reserve 16 bytes as the data transfer buffer
.global _SHARED_MEMORY_       // make it visible globally

.section .text

.LL_loop_start::
   sem.wait _SEM_DATA_IN_READY_
   ....
   load data from _SHARED_MEMORY_ and process
   ....
   sem.sig _SEM_PROCESSING_COMPLETE_
j .LL_loop_start        // go back and wait for the next batch
```

The corresponding user application code could look like this:

```
// payload, that corresponds in size to the _SHARED_MEMORY_ above
struct MyPayload
{
   int a;
   int b;
   int c;
   int d;
};

#define PROC1 0

main()
{
   DRVErrno status;
   struct MyPayload pl;
   unsigned int shared_memory;     // this is the address on the CSX600 processor

   // create a state object
   struct CSAPIState * state = CSAPI_new(CM_Direct);

   // connect to the hardware instance 0
   status = CSAPI_connect( state, 0, 0 );

   if ( status == DRVErrno_success )
   {
        // NOTE: for simplicity the status checking will be omitted
        // from this example

        // obtain the shared memory address using the helper function
        status = CSAPI_get_symbol_value( state ,
                                         "/home/mh/simple.csx",
                                         "_SHARED_MEMORY_",
                                         &shared_memory );
```

```
        status = CSAPI_register_application( state , PROC1 );
        status = CSAPI_load( state, PROC1, "/home/mh/simple.csx" );
        status = CSAPI_register_semaphore ( state, PROC1, _SEM_PROCESSING_COMPLETE_ );
        status = CSAPI_run( state, PROC1 );

        while ( ... there is data to process ... )
        {
             // prepare the data
             pl.a = 1; pl.b = 2; ...

             // copy the data onto the card side
             status = CSAPI_write_mono_memory(
                                  state,
                                  PROC1,
                                  shared_memory,
                                  sizeof( MyPayload ),
                                  (void*)(&pl) );

             // signal the start semaphore
             status = CSAPI_semaphore_signal( state, PROC1, _SEM_DATA_IN_READY_ );

             // wait for results
             // in case of multi-threaded applications
             // something useful could be done during the wait
             // for single threaded applications, the explicit polling/do
             // something useful loop could be implemented

             status = CSAPI_semaphore_wait( state, PROC1, _SEM_PROCESSING_COMPLETE_ );

             // consume the data after processing, assuming here, that
             // the shared memory is used both for receiving and sending
             // data

             status = CSAPI_read_mono_memory(
                                  state,
                                  PROC1,
                                  shared_memory,
                                  sizeof( MyPayload ),
                                  (void*)(&pl) );
        }
    }
    return 0;
}
```

Note: For simplicity, the code above does not use double-buffering which is the recommended method for efficiently interleaving compute and memory transfer operations.

4.5 ClearSpeed host application programming interface (CSAPI)

This section describes the CSAPI.

The state of the API is held in a structure called `CSAPIState`. Before it can be used it has to be correctly initialized. After use it has to be disposed off. Most of the functions return a `DRVErrno` code to indicate success or failure. The return values are described in 4.5.2, *Error codes*, on page 51. All the structures, constants and function prototypes are declared in the `csapi_errno.h` and `csapi.h` header files.

4.5.1 Common parameters

Many of the functions have a set of common parameters, which are described in this section. The remaining parameters are described for each function.

Parameter	`struct CSAPIState *s`
Description	A pointer to an API state structure. A new instance of this structure is created and initialized by a call to the `CSAPI_new()` function.

Table 4.2 CSAPI state structure

Parameter	`unsigned int proc_inx`
Description	A processor index.

Table 4.3 Processor index

Figure 4.2 shows the timeline for driver interactions.

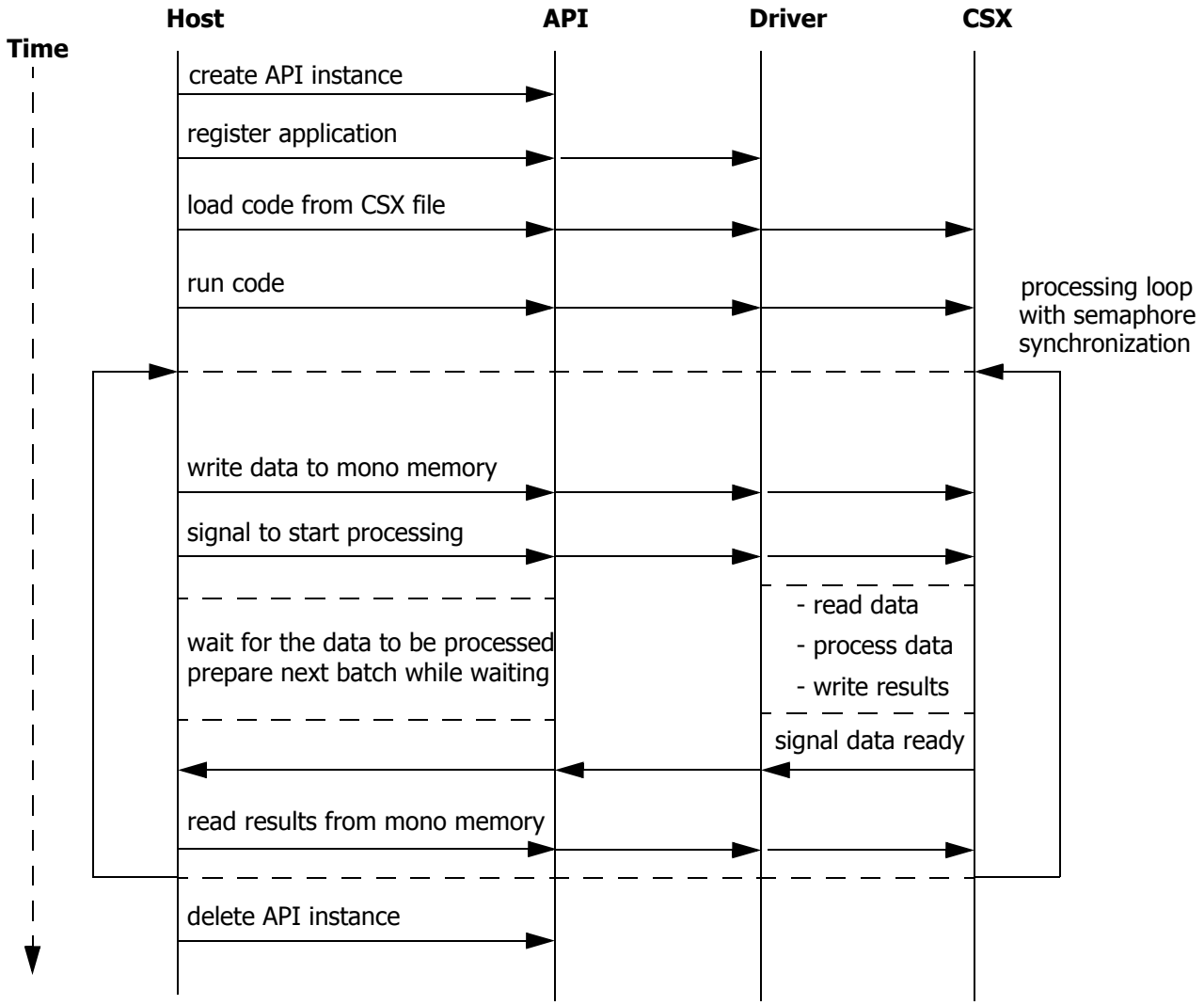

Figure 4.2 Timeline for driver interactions

4.5.2 Error codes

Error codes, listed in Table 4.4, are defined in `csapi_errno.h`. An error string can be obtained by using the `CSAPI_get_error_string` function.

DRVErrno	Description
`success (= 0)`	Operation succeeded.
`error (= 1)`	Returned on a generic error condition. Usually the API calls return a more specific error code as defined below.
`address_in_use`	Failed to connect to given host.
`bad_csapi_arg`	An invalid argument was passed to a CSAPI function.
`bad_csapi_state`	`CSAPIState *s` parameter is invalid.

Table 4.4 Error codes

DRVErrno	Description
bad_inet_address	An invalid inet address was given.
dynamic_allocation	Failed to load statically linked program. Already loaded dynamically.
cannot_allocate_event	Cannot allocate event.
cannot_connect	Cannot connect.
cannot_open_file	The specified file cannot be opened. There may be various reasons for this: the volume is not mounted, network connection failure, and so on.
connection_broken	The connection to the driver has been broken.
failed_to_create_thread	Internal error with thread management.
failed_to_find_memory_block	Trying to delete with an invalid pointer or with a pointer that has already been deleted.
failed_to_load_function	Internal error when loading function.
failed_to_lock_mutex	Failed to lock mutex.
failed_to_unlock_mutex	Failed to unlock mutex.
failed_to_signal_semaphore	Internal error with thread management.
failed_while_waiting_for_semaphore	Internal error with thread management.
FPGA_upgrade_required	FPGA upgrade required.
invalid_proc_inx	proc_inx parameter is invalid.
kernel_max_app_count_exceeded	The maximum number of connecting user applications has already been reached. No more connections can be accepted.
kernel_que	Kernel que.
loaded_statically	Failed to allocated dynamically. Already loaded statically linked program.
loaded_dynamically	CSX program is dynamically linked. Use dynamic version of this function.
no_symbol	The requested symbol was not found in the CSX executable.
not_connected	The application is not connected to a board. Either CSAPI_connect has not been called or it was not successful.
not_enough_blocks	Unable to allocate memory block as internal limit has been reached.
not_enough_memory	Unable to allocate memory on the board as there is not a large enough gap for the requested size.
not_loaded	The program is not loaded
permission_denied	The operation could not be performed on the specified processor as an application has not been loaded or registered on that processor.
program_running	Cannot allocate memory while program is running.
semaphore_not_registered	Semaphore has not been registered for host to work on.
semaphore_number_out_of_bounds	The specified semaphore number is out of range.
semaphore_registered	Semaphore has been registered for host to work on.
socket	Socket error.

Table 4.4 Error codes

4.5.3 Initialization and maintenance functions

The CSAPI functions used for initialization and maintenance are described in Table 4.5 to Table 4.8.

Function	`struct CSAPIState* CSAPI_new(` `  enum CSAPIMode mode );`
Description	Loads the dynamic library for the selected mode and creates a new instance of the API state. A pointer to the API state structure is passed as a parameter to all of the CSAPI functions.
Parameters	mode: Use `CM_Direct` unless attaching the debugger or using tracing.
Returns	A pointer to a newly-created and initialized instance of the API state structure.

Table 4.5 struct CSAPIState CSAPI_new*

Function	`void CSAPI_delete(` `  struct CSAPIState* const s );`
Description	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.

Table 4.6 void CSAPI_delete

Function	`DRVErrno CSAPI_connect(`
	`  struct CSAPIState* const s,`
	`  const char* host_or_addr,`
	`  unsigned int instance );`

Description	Connects to the driver either locally or on the specified host. Connect should be called once for each `CSAPIState` created with `CSAPI_new`. Disconnect by deleting the state with `CSAPI_delete`.
Parameters	s: State created by `CSAPI_new` `host_or_addr`: Host on which the simulator is running or NULL for local hardware. For the host name, use either localhost, DNS name or IP address N.N.N.N. `instance`: Instance number for hardware or simulator, starting from zero. Use `CSAPI_INSTANCE_ANY` to connect to any hardware available

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_failed_to_create_thread`
	`DRVErrno_lldclient_error,`
	`DRVErrno_error`

Table 4.7 DRVErrno CSAPI_connect

Function	`DRVErrno CSAPI_reset(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int flags );`

Description	Resets the processor 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.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to be reset: 0 or 1 flags: Use `CSAPIFlags_FULL_SYSTEM_RESET` to reset CCBR, CCIs, DDRs and bus monitor

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.8 DRVErrno CSAPI_reset

4.5.4 Program setup

The CSAPI functions used for program setup are described in Table 4.9 to Table 4.12.

Function	`DRVErrno CSAPI_register_application(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx );`

Description	Each connecting application needs to register itself with the driver. This tells the driver that the processor is active and tells the processor where to send its events.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to register application for: 0 or 1

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_not_connected,`
	`DRVErrno_kernel_max_app_count_exceeded`

Table 4.9 DRVErrno CSAPI_register_application

Function	`DRVErrno CSAPI_load(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  const char* prog_name );`

Description	Loads the executable code from a .csx file onto a specified processor.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to load application to: 0 or 1. `prog_name`: Specifies the path to the CSX executable, which is relative to the CSPATH environment variable. If the program is in the local directory, ./ must be used or be in CSPATH

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg,`
	`DRVErrno_not_connected`
	`DRVErrno_cannot_open_file`
	`DRVErrno_error`

Table 4.10 DRVErrno CSAPI_load

Function	`DRVErrno CSAPI_get_last_loaded_handle(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  struct DRVProcess** process );`

Description	Sets process to contain the pointer to the previously loaded executable.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to get last process pointer for: 0 or 1. `process`: Pointer to value to be given a pointer to the last loaded process on the processor

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg,`
	`DRVErrno_not_connected`
	`DRVErrno_connection_broken`

Table 4.11 DRVErrno CSAPI_get_last_loaded_handle

Function	`DRVErrno CSAPI_unload(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  struct DRVProcess* process );`

Description	Unloads the specified process from the specified processor.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to unload application from: 0 or 1. `process`: Pointer to the process to be unloaded

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_error`

Table 4.12 DRVErrno CSAPI_unload

4.5.5 Processor control

The CSAPI functions used for controlling the processor are described in Table 4.13 to Table 4.18.

Function	`DRVErrno CSAPI_run(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx );`

Description	Executes a program loaded onto the given processor.
Parameters	s: State created by `CSAPI_new` proc_inx: Index of processor to start (continue) running: 0 or 1.

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.13 DRVErrno CSAPI_run

Function	`DRVErrno CSAPI_run_process(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  struct DRVProcess* process );`

Description	Executes the specified process on the processor it has been loaded on
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to start (continue) running: 0 or 1. `process`: Pointer to the process to execute

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg,`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.14 DRVErrno CSAPI_run_process

Function	`DRVErrno 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: 0 or 1.

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_not_connected,`
	`DRVErrno_permission_denied`
	`DRVErrno_error`

Table 4.15 DRVErrno CSAPI_halt

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

Description	Starts the execution of a halted program.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to start (restart) running: 0 or 1.

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.16 DRVErrno CSAPI_start

Function	`DRVErrno CSAPI_wait_on_terminate(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx );`

Description	Awaits the termination signal from the program running on the given processor.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to wait on for termination: 0 or 1.

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`

Table 4.17 DRVErrno CSAPI_wait_on_terminate

Function	`DRVErrno CSAPI_get_return_value(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  int* const return_value );`

Description	Obtains the return status of the CSX application which has just terminated. This should be called after `CSAPI_wait_on_terminate`. If the function is not successful, return_value is undefined.
Parameters	s: State created by `CSAPI_new` proc_inx: Index of processor to get return value from: 0 or 1. `return_value`: Pointer to value to be given the return value from the terminated CSX program

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg`

Table 4.18 DRVErrno CSAPI_get_return_value

4.5.6 Accessing registers

The CSAPI functions used for accessing the registers are described in Table 4.19 to Table 4.22.

Function	`DRVErrno CSAPI_write_control_register(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int reg_addr,`
	`  unsigned int 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.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to write control register on: 0 or 1. `reg_addr`: Address of control register to write to. `value`: New value to be written to the control register.

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.19 DRVErrno CSAPI_write_control_register

Function	`DRVErrno CSAPI_write_control_register_raw(`
	`  struct CSAPIState* const s,`
	`  unsigned int reg_addr,`
	`  unsigned int 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.
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	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.20 DRVErrno CSAPI_write_control_register_raw

Function	`DRVErrno CSAPI_read_control_register(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int reg_addr,`
	`  unsigned int* 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.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to read control register on: 0 or 1. `reg_addr`: Address of control register to read from `value`: Pointer to memory to be given the value read from the control register

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg,`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.21 DRVErrno CSAPI_read_control_register

Function	`DRVErrno CSAPI_read_control_register_raw(`
	`  struct CSAPIState* const s,`
	`  unsigned int reg_addr,`
	`  unsigned int* 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.
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	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.22 DRVErrno CSAPI_read_control_register_raw

4.5.7 Accessing mono memory and registers

The CSAPI functions used for accessing mono memory and registers are described in Table 4.23 to Table 4.32. Use of the register access functions requires detailed knowledge of the architecture and the function of the control registers.

Function	`DRVErrno CSAPI_write_mono_memory(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int address,`
	`  unsigned int 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) on the board given by address. The size is specified in bytes. The processor specified by proc_inx is halted and the data cache is flushed during the transfer. The appropriate processor for the mono memory address should be used.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to halt while writing to DRAM memory: 0 or 1. `address`: Address of mono memory (DRAM on the board) to start writing to `size`: Number of bytes to copy from data_in (on host) to address (on the board) `data_in`: Address on host to start copying data from

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg,`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.23 DRVErrno CSAPI_write_mono_memory

Function	`DRVErrno CSAPI_write_mono_memory_raw(`
	`  struct CSAPIState* const s,`
	`  unsigned int address,`
	`  unsigned int size,`
	`  const void* data_in );`

Description	Same as `CSAPI_write_mono_memory`, except that the processor is not halted and the data cache is not flushed. This may cause the transfer to take longer, but allows the processor to continue during the transfer.
Parameters	s: State created by `CSAPI_new` `address`: Address of mono memory (DRAM on the board) to start reading from `size`: Number of bytes to copy from address (on the board) to `data_out` (on host) `data_out`: Pre-allocated address on host to start copying data to

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.24 DRVErrno CSAPI_write_mono_memory_raw

Function	`DRVErrno CSAPI_read_mono_memory(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int address,`
	`  unsigned int size,`
	`  void* const data_out );`

Description	Copies data from a location in mono memory (DRAM) on the board given by address, to a pre-allocated buffer on the host pointed to by data_out. The size is specified in bytes. The processor specified by proc_inx is halted and the data cache is flushed during the transfer. The appropriate processor for the mono memory address should be used. The data_out buffer must be allocated on the host before calling this function.
Parameters	s: State created by `CSAPI_new` proc_inx: Index of processor to halt while reading from DRAM memory: 0 or 1. `address`: Address of mono memory (DRAM on the board) to start reading from `size`: Number of bytes to copy from address (on the board) to `data_out` (on host) `data_out`: Pre-allocated address on host to start copying data to

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg,`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.25 DRVErrno CSAPI_read_mono_memory

Function	`DRVErrno CSAPI_read_mono_memory_raw(`
	`  struct CSAPIState* const s,`
	`  unsigned int address,`
	`  unsigned int size,`
	`  void* const data_out );`

Description	Same as `CSAPI_read_mono_memory`, except that the processor is not halted and the data cache is not flushed. This may cause the transfer to take longer, but allows the processor to continue during the transfer.
Parameters	s: State created by `CSAPI_new` `address`: Address of mono memory (DRAM on the board) to start reading from `size`: Number of bytes to copy from address (on the board) to `data_out` (on host) `data_out`: Pre-allocated address on host to start copying data to

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.26 DRVErrno CSAPI_read_mono_memory_raw

Function	`DRVErrno CSAPI_write_mono_memory_async(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int address,`
	`  unsigned int size,`
	`  const void* data_in );`

Description	Starts the `CSAPI_write_mono_memory` function in a separate thread and returns immediately. The host is then free to continue in the current thread. The host can check when the transfer has completed by using the `CSAPI_write_mono_memory_async_poll` function, or it can wait for the transfer to complete by using the blocking `CSAPI_write_mono_memory_async_wait` function.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to halt while writing to DRAM memory: 0 or 1. `address`: Address of mono memory (DRAM on the board) to start writing to `size`: Number of bytes to copy from `data_in` (on host) to address (on the board) `data_in`: Address on host to start copying data from

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg,`
	`DRVErrno_failed_to_signal_semaphore`
	`DRVErrno_failed_while_waiting_for_semaphore`

Table 4.27 DRVErrno CSAPI_write_mono_memory_async

Function	`DRVErrno CSAPI_write_mono_memory_async_wait(` `  struct CSAPIState* const s );`
Description	Allows the host to wait for the completion of an asynchronous transfer started by `CSAPI_write_mono_memory_async`. The wait will only return once the transfer has completed or if there is an error. The return status should be checked to ensure that the transfer completed successfully.
Parameters	s: State created by `CSAPI_new`

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_failed_while_waiting_for_semaphore,`
	`DRVErrno_failed_to_signal_semaphore`

Table 4.28 DRVErrno CSAPI_write_mono_memory_async_wait

Function	`DRVErrno CSAPI_write_mono_memory_async_poll(`
	`  struct CSAPIState* const s,`
	`  unsigned int* const completed );`

Description	Allows the host to check the status of an asynchronous transfer started by `CSAPI_write_mono_memory_async`. The status is passed back using the completed parameter. The return status should be checked to ensure that a valid status was obtained.
Parameters	s: State created by `CSAPI_new` completed: Pointer to value to be given the status of the transfer in progress

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_failed_while_waiting_for_semaphore`

Table 4.29 DRVErrno CSAPI_write_mono_memory_async_poll

Function	`DRVErrno CSAPI_read_mono_memory_async(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int address,`
	`  unsigned int size,`
	`  void* const data_out );`

Description	Starts the `CSAPI_read_mono_memory` function in a separate thread and returns immediately. The host is then free to continue in the current thread. The host can check when the transfer has completed by using the `CSAPI_read_mono_memory_async_poll` function, or it can wait for the transfer to complete by using the blocking `CSAPI_read_mono_memory_async_wait` function.
Parameters	s: State created by `CSAPI_new` proc_inx: Index of processor to halt while reading from DRAM memory: 0 or 1. `address`: Address of mono memory (DRAM on the board) to start reading from `size`: Number of bytes to copy from address (on the board) to `data_out` (on host) `data_out`: Pre-allocated address on host to start copying data to

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg,`
	`DRVErrno_failed_to_signal_semaphore`
	`DRVErrno_failed_while_waiting_for_semaphore`

Table 4.30 DRVErrno CSAPI_read_mono_memory_async

Function	`DRVErrno CSAPI_read_mono_memory_async_wait(` `  struct CSAPIState* const s );`
Description	Allows the host to wait for the completion of an asynchronous transfer started by `CSAPI_read_mono_memory_async`. The wait will only return once the transfer has completed or if there is an error. The return status should be checked to ensure that the transfer completed successfully.
Parameters	s: State created by `CSAPI_new`

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_failed_while_waiting_for_semaphore,`
	`DRVErrno_failed_to_signal_semaphore`

Table 4.31 DRVErrno CSAPI_read_mono_memory_async_wait

Function	`DRVErrno CSAPI_read_mono_memory_async_poll(`
	`struct CSAPIState* const s,`
	`unsigned int* const completed );`

Description	Allows the host to check the status of an asynchronous transfer started by `CSAPI_read_mono_memory_async`. The status is passed back using the completed parameter. The return status should be checked to ensure that a valid status was obtained.
Parameters	s: State created by `CSAPI_new` `completed`: Pointer to value to be given the status of the transfer in progress

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_failed_while_waiting_for_semaphore`

Table 4.32 DRVErrno CSAPI_read_mono_memory_async_poll

4.5.8 Endian functions

The CSAPI endian functions are described in Table 4.33 to Table 4.34.

Function	`DRVErrno CSAPI_buffer_to_native_endian(`
	`struct CSAPIState* const s,`
	`unsigned int proc_inx,`
	`void* out,`
	`const void* in,`
	`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: 0 or 1. `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

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg,`
	`DRVErrno_not_connected`

Table 4.33 DRVErrno CSAPI_buffer_to_native_endian

Function	`DRVErrno CSAPI_endianness(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int* const endian );`

Description	Gets the current endianness of the specified processor on the current board.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor get endianness of: 0 or 1. `endian`: Pointer to value to be given the current endianness of the specified processor on the current board. Value will be set to either `CS_LITTLE_ENDIAN` or `CS_BIG_ENDIAN`

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg`

Table 4.34 DRVErrno CSAPI_endianness

4.5.9 Thread functions

The thread functions are described in Table 4.35 to Table 4.36.

Function	`DRVErrno 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` 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: 0 or 1. `thread_id`: ID of thread to switch to on processor `old_thread_id`: Pointer to value to be given the ID of thread that was running on processor

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg,`
	`DRVErrno_not_connected`
	`DRVErrno_permission_denied`
	`DRVErrno_error`

Table 4.35 DRVErrno CSAPI_set_thread

Function	`DRVErrno 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 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: 0 or 1. `num_of_threads`: Pointer to value to be given the number of threads supported by the processor

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg`

Table 4.36 DRVErrno CSAPI_num_threads

4.5.10 Semaphore handling

The CSAPI functions used for handling semaphore are described in Table 4.37 to Table 4.40.

Function	`DRVErrno CSAPI_register_semaphore(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int sem_number );`

Description	Registers that the host is intending to wait on the specified semaphore number on the specified processor. Registration must be performed only on semaphores that the host will be waiting on. A race condition will be created if the host signals, or the board processor waits on, a registered semaphore.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to register semaphore on: 0 or 1. `sem_number`: Semphore number to register on processor

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_not_connected`
	`DRVErrno_semaphore_number_out_of_bounds`
	`DRVErrno_error`

Table 4.37 DRVErrno CSAPI_register_semaphore

Function	`DRVErrno CSAPI_semaphore_wait(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int sem_number );`

Description	Allows the host to wait on the specified semaphore on the specified processor. The semaphore must be registered before it can be waited upon.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor wait for semaphore on: 0 or 1. `sem_number`: Semaphore number to wait on

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_failed_while_waiting_for_semaphore,`
	`DRVErrno_semaphore_number_out_of_bounds`

Table 4.38 DRVErrno CSAPI_semaphore_wait

Function	`DRVErrno CSAPI_semaphore_signal(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int sem_number );`

Description	Signals the specified semaphore on the specified processor. The semaphore must be NOT be registered if it is going to be signalled by the host.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to signal semaphore on: 0 or 1. `sem_number`: Semaphore number to signal

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_not_connected`
	`DRVErrno_error`

Table 4.39 DRVErrno CSAPI_semaphore_signal

Function	`DRVErrno CSAPI_num_semaphores(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int* const num_of_semaphores );`

Description	Provides the total number of semaphores supported by the specified processor. Note this will include system semaphores, and semaphores used by the debugger and asynchronous memory read / write functions.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to count semaphore on: 0 or 1. `num_of_semaphores`: Pointer to value to be given the number of semaphores supported

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg`

Table 4.40 DRVErrno CSAPI_num_semaphores

4.5.11 Callback functions

The CSAPI functions for callback are described in Table 4.41 to Table 4.42.

Function	`DRVErrno CSAPI_get_callback(`
	`  struct CSAPIState* const s,`
	`  unsigned int event,`
	`  CSAPI_EventFnPtr* 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`: Event to trigger the call back function `event_cb_out`: Pointer to value to be given the call back Function pointer

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_error`

Table 4.41 DRVErrno CSAPI_get_callback

Function	`DRVErrno CSAPI_register_callback(`
	`  struct CSAPIState* const s,`
	`  unsigned int event,`
	`  CSAPI_EventFnPtr event_cb,`
	`  void* user_data );`

Description	Registers a user call back function for an event signalled by the board. When the event occurs the event call back function will be called with the state, event data and a pointer to specified user data.
Parameters	s: State created by `CSAPI_new` `event`: Event to trigger the call back function `event_cb`: Function pointer to an event 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

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_error`

Table 4.42 DRVErrno CSAPI_register_callback

4.5.12 Memory allocation using CSAPI functions

The CSAPI functions for allocating mono memory (DRAM) are described in Table 4.43 to Table 4.48. The main purpose of these functions is to allocate space for the heap on the CSX600 processor. Currently, the allocation functions only allocate from the DRAM subset of mono memory attached to the processors on the board.

Function	`DRVErrno CSAPI_get_free_mem(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int mem_inx,`
	`  unsigned int* const size_out );`

Description	Returns the total amount of mono memory (DRAM) free for allocation and local to the specified processor. 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 get free memory on: 0 or 1. `mem_inx`: Index of memory. Currently unused as free space can only be used for mono memory `size_out`: Pointer to variable used to return the amount of free memory local to the processor

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg,`
	`DRVErrno_program_running`

Table 4.43 DRVErrno CSAPI_get_free_mem

Function	`DRVErrno CSAPI_allocate_shared_memory(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int mem_inx,`
	`  unsigned int size,`
	`  unsigned int alignment,`
	`  const char* symbol_name,`
	`  unsigned int* const mem_ptr_out );`

Description	Allocates mono memory (DRAM) local to the specified processor for use by both the CSX program and the host. An alignment can be specified for the allocation. The allocated address will be assigned to the variable specified by `symbol_name` in the current CSX program. The same address will be returned in the `mem_ptr_out` parameter. 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: 0 or 1. `mem_inx`: Index of memory. Currently unused as only mono memory can be allocated `size`: Number of contiguous bytes to attempt to allocate in the free space `alignment`: Bytes to align allocated memory to on the board `symbol_name`: Name of static global variable in the CSX program to be given the allocated address. If this is `NULL` then the address is not passed to the CSX program. `mem_ptr_out`: Pointer to value to be given the address of the allocated memory on the processor

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_program_running,`
	`DRVErrno_no_symbol`
	`DRVErrno_cannot_open_file`
	`DRVErrno_not_enough_blocks`
	`DRVErrno_not_enough_memory`
	`DRVErrno_error`

Table 4.44 DRVErrno CSAPI_allocate_shared_memory

Function	`DRVErrno CSAPI_allocate_static_shared_memory(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int mem_inx,`
	`  unsigned int mem_ptr_in,`
	`  unsigned int size,`
	`  const char* symbol_name );`

Description	Allocates mono memory (DRAM) local to the specified processor at the address specified by `mem_ptr_in`. This is the same as `CSAPI_allocate_shared_memory`, except that it attempts to allocate a block of memory of the requested size at the specified address. It returns an error if a large enough area of contiguous memory is not available at that address. 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: 0 or 1. `mem_inx`: Index of memory. Currently unused as only mono memory can be allocated `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 `symbol_name`: Name of static global variable in the CSX program to be given the allocated address. If this is `NULL` then the address is not passed to the CSX program.

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_program_running,`
	`DRVErrno_no_symbol`
	`DRVErrno_cannot_open_file`
	`DRVErrno_not_enough_blocks`
	`DRVErrno_address_in_use`
	`DRVErrno_error`

Table 4.45 DRVErrno CSAPI_allocate_static_shared_memory

Function	`DRVErrno CSAPI_free(`
	`  struct CSAPIState* const s,`
	`  unsigned int proc_inx,`
	`  unsigned int mem_ptr );`

Description	Releases previously allocated memory local to the specified processor at the address given by `mem_ptr`. This memory on the board is then available to be allocated by `CSAPI_allocate_shared_memory`. Only memory allocated by `CSAPI_allocate_shared_memory` can be freed. 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 release allocated memory on: 0 or 1. `mem_ptr`: Address of the allocated memory that is to be released.

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_program_running,`
	`DRVErrno_failed_to_find_memory_block`

Table 4.46 DRVErrno CSAPI_free

Function	`DRVErrno CSAPI_get_symbol_value(`
	`  struct CSAPIState* const s,`
	`  const char* prog_name,`
	`  const char* symbol_name,`
	`  unsigned int* const symbol_value_out );`

Description	Returns the value of the symbol symbol_name in the given CSX program. 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 board pointed to by the variable.
Parameters	s: State created by `CSAPI_new` `prog_name`: Name of the CSX program that declares the 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	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_no_symbol`
	`DRVErrno_cannot_open_file,`
	`DRVErrno_error`

Table 4.47 DRVErrno CSAPI_get_symbol_value

Function	`DRVErrno CSAPI_get_symbol_value_loaded(`
	`  struct CSAPIState* const s,`
	`  struct DRVProcess* process,`
	`  const char* symbol_name,`
	`  unsigned int* const symbol_value_out );`

Description	Returns the symbol value as loaded in the given process. This will include any relocations that have been applied
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	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_no_symbol`

Table 4.48 DRVErrno CSAPI_get_symbol_value_loaded

4.5.13 Utility functions

The CSAPI utility functions are described in Table 4.49 to Table 4.53.

Function	`DRVErrno CSAPI_set_system_param(`
	`  struct CSAPIState* const s,`
	`  enum CSAPISystemParameters what,`
	`  unsigned int vi,`
	`  const char* vs );`

Description	Sets the specified system parameter (using the CSAPISystemParameters enumeration) to a value given to either the vi or vs parameter as appropriate. See the CSAPISystemParameters enumeration to see which parameters use which input variable. The user documentation should be referred to for valid combinations of system parameters.
Parameters	s: State created by `CSAPI_new` `what`: Name of parameter to be set, defined by an enumeration at the top of this file `vi`: Integer value to set the parameter with (if applicable) `vs`: String value to set the parameter with (if applicable)

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_error`

Table 4.49 DRVErrno CSAPI_set_system_param

Function	`DRVErrno CSAPI_num_cards(`
	`  struct CSAPIState* const s,`
	`  unsigned int* const num_of_cards );`

Description	Provides the number of boards in the current system. Note that this does not include simulators. It is still necessary to call `CSAPI_new` before this function to load the library of functions. It is valid to create a CSAPIState with `CSAPI_new`, call this function and then delete the state with `CSAPI_delete`.
Parameters	s: State created by `CSAPI_new`. Required to ensure the dynamic CSAPI library is loaded `num_of_cards`: Pointer to value to be given the number of boards in the current system

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_arg`
	`DRVErrno_lldclient_error`

Table 4.50 DRVErrno CSAPI_num_cards

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

Description	Returns the number of processors on the board we are connected to.
Parameters	s: State created by `CSAPI_new` `num_of_processors`: Pointer to value to be given the number of processors
Returns	`DRVErrno_bad_csapi_state` `DRVErrno_bad_csapi_arg`

Table 4.51 DRVErrno CSAPI_num_processors

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

Description	Returns the number of processing elements on the specified processor on the board we are connected to.
Parameters	s: State created by `CSAPI_new` `proc_inx`: Index of processor to get count of processing elements on: 0 or 1. `num_of_pes`: Pointer to value to be given the number of processing elements

Returns	`DRVErrno_success`
	`DRVErrno_bad_csapi_state`
	`DRVErrno_invalid_proc_inx`
	`DRVErrno_bad_csapi_arg`

Table 4.52 DRVErrno CSAPI_num_pes

Function	`DRVErrno CSAPI_get_error_string(`
	`  struct CSAPIState* const s,`
	`  DRVErrno error_code,`
	`  char* const error_string,`
	`  const 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.
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 DRVErrno `error_string`: Pointer to pre-allocated buffer to receive error string `max_string_length`: Length of pre-allocated buffer to receive error string
Returns	`DRVErrno_success` `DRVErrno_bad_csapi_arg`

Table 4.53 DRVErrno CSAPI_get_error_string

4.6 Calling CSAPI routines

Some CSAPI functions require a connection to the board before they can be called and others do not. If the board is in use by another user, you will not have a connection to the board. 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 "`not connected`" error, they can be called before connecting to a board.

4.6.1 Functions that can called before connecting to the board

The following functions can be called before a connection to a board is established:

```
CSAPI_new
CSAPI_delete
CSAPI_connect
```

The following functions are not currently related to board connection and can be called at any time:

```
CSAPI_num_boards
CSAPI_get_error_string
```

In short, the five functions outlined in this section are safe to be called when the board is in use by another user (`new`, `delete`, `connect`, `num_boards` and `get_error_string`).

Functions that do not communicate with the board

The following functions do not actually communicate with a board, so they can be called without being connected:

```
CSAPI_get_free_semaphores
CSAPI_allocate_shared_semaphore
CSAPI_allocate_static_shared_semaphore
CSAPI_free_semaphore
CSAPI_get_free_memory
[CSAPI_allocate_shared_memory]
[CSAPI_allocate_static_shared_memory]
CSAPI_free_memory
CSAPI_num_processors
CSAPI_num_pes
CSAPI_unload
CSAPI_endianness
CSAPI_set_system_param
CSAPI_num_threads
CSAPI_get_callback
CSAPI_register_callback
```

Note: The functions `CSAPI_allocate_shared_memory` and `CSAPI_allocate_static_shared_memory` are only in this category if the CSX program symbol name parameter is `NULL`, in which case the address of the allocated memory is not passed to the CSX program.

Note: When you have multiple board types, they should *not* be called when not connected.

4.6.2 Functions that should not be called when not connected

The following functions do not actually communicate with the board, but they will not return until a connection has been established. They should *not* be called when not connected:

```
CSAPI_wait_on_terminate
CSAPI_get_return_value
```

Lastly, the current design allows these functions to be called when not connected, but when programs are loaded dynamically they should *not* be called when no longer connected:

```
CSAPI_get_symbol_value
CSAPI_get_symbol_value_loaded
```

4.7 Access control

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 is most easily disabled by removing the write privilege from the lock file parent folder.

4.7.1 The lock file

The lock file, which must be writable, is created in `/var/lock/clearspeed/` on Linux. Under Windows the lock file is created in the installation directory (typically `C:\Program Files\clearspeed\csx600_m512_le`).

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.
```

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.
```

If the first line is 0, the file is not currently being modified. When the file is accessed the 0 is changed to a random number then the file is read. When the file is updated this number is first checked to ensure nothing has changed since the file was read, and after the changes have been written it is changed back to 0. If the line stays as the same non-zero value for too long, it is considered stale and is ignored.

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 board (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 2006
```

All text from the time entry on line 5 to the next asterisk is ignored. So, if a board 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 is used to check that the process that locked the board is still running and allows stale entries in the lock file to be identified.

4.8 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 intermedi-

ate 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.

If the transfer does not satisfy these constraints, a slower data transfer mechanism will be used. The slower data transfer mechanism is typically one hundredth the speed of DMA, so should not normally be used.

DMA on some chip sets can cause problems and it is possible to disable DMA. If the environment variable `CS_DISABLE_DMA` is set, DMA will not be used. This may be useful in diagnosing problems. The variable is set as follows:

```
export CS_DISABLE_DMA=1   (on Linux/bash)
set CS_DISABLE_DMA=1      (on Windows)
```

5 Diagnostic software reference

This chapter describes the commands relating to the board diagnostic operations. The board 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 board 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.

5.1 Diagnostic tests using Perl

The diagnostic tests, which are run using Perl, enable you to check that all the parts of the board are working properly. For example, the tests check the CSX600 processors, the memory, and the performance of the DMA transfers.

You should always run these tests:

- After you have installed the boards.
- If you experience a specific problem when using the board.

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

5.1.1 Full diagnostic tests for Windows XP

To run the diagnostic tests:

1. Double-click the desktop shortcut `csx600_m512_le` to open the command window.
2. 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]`.

5.1.2 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]`.

5.1.3 What to do if the tests fail

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

If the tests fail to detect the board 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 board and drivers are installed correctly but some of the tests still fail, refer to the troubleshooting guide in the *Advance Accelerator Board User Guide* to help you solve the problem.

5.2 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 board or boards 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).

5.2.1 How to run the Mandelbrot demonstration in Windows XP

To run the Mandelbrot demonstration:

3. Double-click the desktop shortcut `csx600_m512_le` created by the installation. This opens a command window.
4. Reset all the boards, by entering the following command in the new window:

   ```
   csreset -Av
   ```
5. 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.
6. Press [Esc] to exit from the Mandelbrot program.

5.2.2 How to run the Mandelbrot demonstration in Linux

To run the Mandelbrot program:

1. Reset the board:

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

   ```
   ./app_mandelbrot
   ```

6 Kernel level driver

This chapter is of interest to anyone who wants to understand what the kernel driver does. It is a particularly good source of information for people who want to modify the kernel driver so that it can be embedded in the kernel source code.

This chapter describes the various parts of the kernel driver by looking at the main stages the driver goes through:

1. Module loading and unloading
2. Device opening, closing and mmap
3. Interrupt handling
4. DMA ioctls

The driver source code is referred to so it is useful to have this source available.

Many of the terms used in this documentation can be found in *Linux Device Drivers, Third Edition* [3].

6.1 Overview

The ClearSpeed Linux driver is a Linux kernel module which provides the lowest level functionality to connect to an Advance board via the PCI. The driver is provided in source form under the terms of the GNU General Public License (GPL) in the following three files:

- csx_driver.h

  A header file that contains the definitions common to the user side of the driver and the kernel side. Specifically it defines the ioctls the kernel driver provides.
- csx_driver.c

  The source code of the driver.
- Makefile

  The makefile for the driver. The driver uses the kBuild mechanism for module building.

The kernel driver acts like a Linux char device, providing the standard functionality that these devices provide:

- open
- close
- mmap and so on.

The driver is also a PCI driver and uses the standard PCI subsystem API to register callbacks and so on. In addition, the driver provides a set of ioctls to implement the extra functionality the Advance board needs specifically for scatter-gather DMA operations.

The driver uses the `get_user_pages`() functionality available only in the 2.6 based kernels to implement the DMA operations hence the driver cannot be used in 2.4 based kernels where a different mechanism is used.

Note: The `get_user_pages` functionality became stable after the 2.6.7 release of the kernel and again the driver cannot be used with kernels before this release. `get_user_pages` is heavily used by InfiniBand drivers and these drivers first encountered the `get_user_pages` problems.

The ClearSpeed driver consists of two parts:

- A user-space driver which is provided in a shared library.
- The kernel driver which is provided as a kernel module.

An unusual feature of the ClearSpeed driver is that the kernel module contains few dependencies on the architecture of the ClearSpeed system on the PCI board. Most of the functionality that a standard driver might provide such as handling interrupts is provided by the user-space driver rather than by the kernel side driver.

For each Advance board on a system, the driver dynamically generates two `/dev` (and `/sys/class`) entries. For example, if two PCI Advance boards are discovered, four `/dev/` entries will be created by the driver. The "control device" (`csx_ctl`) (`/dev/csxnc`) maps in the control and configuration spaces of the device and the main asynchronous PCI interrupt signal. The "memory device" (`csx_mem`) (`/dev/csxnm`) maps the memory space on the device and handles the DMA interrupt. This two device paradigm will be encountered throughout the driver where operations are performed on the control/memory device pair.

In general, the style of the driver is to provide a teardown function which is the exact opposite of the creation function where resources are released in the exact reverse order of their allocation. This document describes the module operating as an "out-of-tree" module, that is, not compiled into the kernel. Once the driver is accepted into the Linux driver tree we hope to provide an in-tree driver.

6.2 Module loading and unloading

The driver provides module load and unload functions via `csx_init` and `csx_exit`. `csx_init` obtains a block of device numbers for the attached PCI boards and initialize the `/sys/class` hierarchy for `udev` and user use. `csx_init` also registers the device with the PCI subsystem via a call to `pci_register_driver()`. See *Linux Device Drivers, Third Edition* [3] for details on the PCI subsystem. The driver registers its callback function (`csx_probe`) to be called when a PCI device with the specified <Vendor id,Device id> pair is found.

At this point, the driver has not created any /`dev` entries or `/sys/class` entries, this only happens when the PCI subsystem detects a device with the specified <Vendor id,Device id> pair, not at module load time. However, without PCI hotplugging, device detection follows shortly after the kernel is booted and this step is described next.

The PCI callback function csx_probe is called for each ClearSpeed device found and the steps taken for each device are summarized in Table 6.1.

Steps	Substeps
1. The device is enabled as a PCI device and as a bus master.	
2. The control device (/dev/csxnc) is created.	Setup of the control device is carried out by the function csx_setup_ctl and consists of: a. Establishing a kernel map for the PCI control memory space (BAR0). This is used by the kernel driver to mask interrupts, read various control registers and other control operations. The mapping is built via ioremap_nocache(). b. Initializing the interrupt queue. c. Creating the /dev device entry.
3. The memory device (/dev/csxnm) is created.	The memory device is similar to the control device setup with the following differences: • No kernel mapping is built for the memory aperture (BAR2). This is not needed by the kernel and would be wasteful of kernel resources. This behavior can be changed via the CS_NOMAPMEM macro. • Coherent DMA space is allocated for the DMA descriptors via dma_alloc_coherent(). This space is used for DMA descriptors while DMA is in flight. The driver specifies the devices PCI address features via a call to dma_set_mask to indicate that the device is 64 bit DMA capable. Note that the default is to have only 32 bit capable PCI devices and without this call the kernel will build bounce buffers for DMA transfers. At this stage, coherent DMA mappings can be allocated for contiguous DMA buffers used as bounce buffers. This can be used as an alternative to scatter-gather and is enabled via the CSX_BOUNCE macro.
4. The interrupt handler is registered.	

Table 6.1 Steps taken by csx_probe

A side effect of Step 3 is that the udev daemon is called with an ADD device request. Similarly, if the remove function (csx_remove) is called, udev is called with a REMOVE request. Since the driver presents two devices, a pair of ADD requests is issued to udev (matched by a pair of REMOVE requests on release). udev will scan the /sys/class structure for changes and carry out operations such as setting permissions and setting device symbolic links. The current driver system does not make full use of udev. This is due to differences in the udev setup on various systems and due to bugs in udev. Instead, the installation script performs the device permission settings. One bug in particular caused udev to momentarily add a device and then remove it.

However, udev could be used for permission settings and the relevant rules could be added to the udev rule set on a specific machine. A rule like the following would correctly set the device permissions:

```
KERNEL=="csx",MODE="0666"
```

Once this step is complete, the device is ready for use.

6.3 Device opening, closing and mmap

The device pair for a physical PCI device present a standard set of char device operations defined in `csx_ctl_flops` for the control device and `csx_mem_fops` for the memory device. The operations provided are:

- open/release

  Standard device open via file descriptor. Note that unusually perhaps, the driver does not impose any access control on open. This functionality is provided in the user-land driver. Any per-connection data structures are allocated (freed).

- poll

  Poll is used to wait for an interrupt from the device. The poll on the control device is the asynchronous PCI interrupt wait. The poll on the memory device is the synchronous DMA complete interrupt.The functionality is implemented by adding the caller to an interrupt queue which the actual interrupt handler will release. This model allows the user-land device driver to wait simultaneously on DMA interrupts and PCI interrupts in separate threads.

- read/write

  Read and write provide control register reads and writes for the Advance board via the kernel's mapping of the control aperture space. These are not in fact used by the current user-land driver, instead a mapped address space is used. However, these operations can be used if a strict sequentiality is needed by the driver or if aperture mapping is not available.

  They are also useful in driver bring up.

  The memory device has read and write operations as well as llseek. This provides a nonmemory-mapped interface to the memory on the Advance board. Seek moves to a position within the aperture and read or write is then used to access a number of bytes at this offset. These operations are not currently used and the memory mapped interface is used instead.

- mmap

  Both the control and memory devices provide mmap operations. They are used to provide user-space mappings of the control and data apertures respectively. The data aperture mapping is in fact much smaller than the full data address range available and the aperture acts more like an address window.

  This windowing functionality is not provided by the kernel driver but rather by the user-land driver. The mapping is built by `io_remap_page_range` in earlier 2.6 kernels and `remap_pfn_range` in later kernels.

  The driver marks the mapping as being for IO with no caching.

  The mmap operation is used by the user-land driver via the `mmap`() system call on the appropriate open file descriptor.

6.4 Interrupt handling

All interrupts from the device are routed to `csx_intr_handler`. This uses the kernel control mapping to examine the device to determine if the interrupt is PCI or DMA. A thread waiting on these interrupt queues will be woken up. The driver must also acknowledge the interrupt to avoid it being refired once the kernel is left. This acknowledgement is handled differently for the two interrupt sources.

For PCI interrupts, the interrupt mask is closed but the interrupt itself is not cleared. Interrupt clearing is handled in the user-land driver. PCI interrupt clearing is too complex to be handled in the kernel.

For DMA interrupts, the interrupt is acknowledged by the kernel by the usual technique of writing back the interrupt source bits. The interrupt mask is not changed by the kernel driver.

Note: The kernel driver uses kernel memory write flushing macros to ensure that these interrupt operations are flushed before leaving the kernel.

6.5 DMA ioctls

The DMA engine on the PCI device is programmed by the user-land driver to perform scatter-gather DMA to and from the Advance board. The approach taken is to do as much as possible in the user-land driver and obtain the necessary kernel functionality via ioctls on the memory device.

The model called "zero-copy user-space access" DMA allows DMA directly on user data avoiding any bounce buffer copying. For further information, see *get_user_pages usage* [4]. Using a bounce buffer can loose 30% of DMA performance.

DMA data buffers use so-called scatter-gather mappings which are a subset of "streaming DMA" mappings which are efficient but have a stringent set of rules concerning their coherency. The DMA descriptors use a "coherent DMA" mapping. The kernel driver uses the scatter-gather coherency functions to mark the buffers as being coherent for the device or coherent for the host. For a general description of DMA mapping, see *KernelSourceTree*/Documentation/DMA-mapping.txt (replacing *KernelSourceTree* by the location of downloaded kernel sources).

In outline, a DMA operation to transfer a single (virtual address) contiguous buffer from host memory to or from the ClearSpeed device is as follows:

```
Obtain physical page address mapping for user buffer: Lock_data_buffer
Construct Descriptor set and program DMA engine
Fire DMA: Fire_DMA
Poll on memory device
Unlock_data_buffer
```

The kernel driver provides a pair of resources for DMA to allow for concurrency between buffer mapping for transfer #N and the actual DMA transfer for transfer #(N-1). This allows for the overlap between DMA preparation functions and DMA traffic as shown in Figure 6.1.

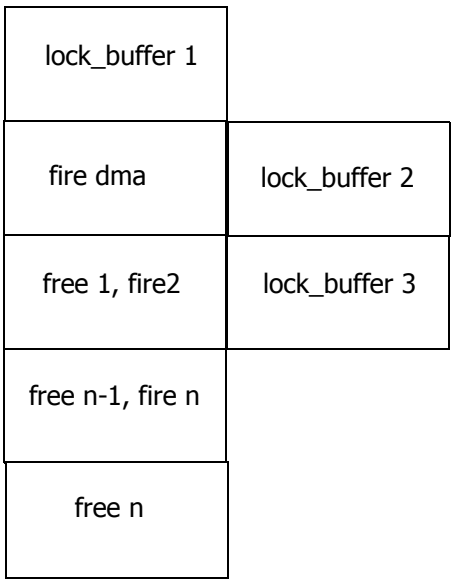

Figure 6.1 DMA concurrency

Thus, only the initial setup and final teardown operations are not overlapped with DMA.

Kernel ioctls require unique numbers and a set of macros are provided by the kernel to help build the necessary unique numbers and to help characterize the ioctl use. For the ClearSpeed ioctl set, use the prefix 0xC5 as the "IOCTL MAGIC" number.

The user-land driver interfaces to the kernel ioctls via a library that provides a function for each ioctl. These wrapper functions collect the ioctl parameters into a struct specific to the ioctl and pass this struct to the kernel trap. A full specification of ioctl functionality, parameter types, return codes and so on is given in this header file. See csx_driver.h for more information.

Each ioctl has a checking step in the kernel where all the parameters are validated before the ioctl is executed. This is not described in this document as the intent should be clear from the source code.

The ioctls provided for DMA support are the following:

csx_lock_buffer_for_dma

This ioctl takes the user data buffer and locks the physical page set for the buffer into memory in preparation for DMA. `get_user_pages`() is used to pin the pages to physical addresses and to obtain the physical page addresses. If the virtual buffer has not been accessed and hence does not have any physical addresses, this step can require the kernel to generate the physical addresses. This can be a slow operation but this latency can be alleviated by overlapping the buffer lock and unlock with a DMA transfer.

The second step is to generate the "scatter-gather" list which provides the PCI bus addresses as needed for DMA. This is carried out by the call to `dma_map_sg`(). This allows the kernel to perform any architecture specific operations necessary for memory DMA. Note that `get_user_pages`() returns the number of full pages mapped (or pinned). This will include any partial pages mapped due to nonpage-aligned user buffers. Mapping to the scatter-gather set can result in a different page count such as:

```
pages_pinned by get_user_pages>= pages_mapped by dma_map_sg
```

The kernel driver has to retain this distinction in order to properly free the resources in the matching `unlock_dma` ioctl.

Note: Physical memory addresses are not suitable for DMA and the addresses returned by the `dma_map_sg` function must be used. For example, some 32 bit machines remap the PCI address space beyond the 4 GB boundary and on these machines, PCI addresses can be 33 bits in length.

The set of PCI bus addresses for the transfer are returned to the user-land driver where they are used to program the card DMA engine.

`csx_unlock_data_buffer`

This reverses the lock buffer operation undoing the DMA scatter-gather list mapping and undoing `get_user_pages`. As many others have noted, it is strange that the kernel does not provide a `put_user_pages` function to undo the effect of `get_user_pages`. Instead, `page_cache_release` is called on each pinned page. Note again the distinction between the pages mapped for the `dma_unmap_sg` call and the pinned set as returned by `get_user_pages`.

These two locking functions also use the DMA synch calls to mark the physical pages for coherency. On a write to device operation, `csx_lock_buffer_for_dma` calls `dma_sync_sg_for_device`. On a read from device operation, `csx_unlock_data_buffer` calls `dma_sync_for_cpu`.

`copy_to_desc_buffer`

The kernel driver reserves DMA coherent space for the DMA descriptors used by the DMA engine. The user-land driver constructs the descriptor set based on the mapped page information from the buffer lock functions and this ioctl copies the user descriptor set into this DMA coherent space. This is generally safer than using user locked space for DMA descriptors.

`csx_fire_dma`

This is an ioctl to finally fire the DMA by setting the valid bit in the DMA engine. An ioctl ensures that all the necessary memory and control flush operations have taken place.

`csx_copy_and_fire`

This ioctl combines the descriptor copy and DMA fire into a single ioctl call for efficiency.

`csx_get_page_size`

This is used to provide the user space with the running kernel's physical page size. This is needed on systems like Itanium where the page size can vary.

6.6 Miscellaneous

6.6.1 Class interface

The device driver provides a pair of class device interfaces for the `/sys/class` hierarchy. The principal reason for this is to provide the structure needed by `udev` but the system allows for arbitrary information to be provided. The function `csx_show_status` gives an example of what can be provided. This displays the version number of the FPGA and driver build time for instance. This structure can be extended to allow the driver to provide extra information to the user space.

6.6.2 /proc interface

Later versions of the kernel driver also provide a `/proc` interface for displaying performance information via `/proc/driver/csxn`. At the moment this is only experimental.

6.6.3 Moving functionality into kernel driver

The split between user land operations and kernel side operations means that the kernel driver ends up knowing little about the device and for performance reasons alone it is worth considering moving some functionality into the kernel. For example, some of the interrupt handling could be moved into the kernel driver. However, there is a feature of the method to access the devices control space which makes this very difficult. Most control information on the csx device is available via the PVCI bus.

To access this bus, the following operations are needed:

1. A write to an address register
2. A read or write to obtain the result.

This makes PVCI accesses nonatomic thus making it impossible for the kernel to access the PCVI space safely.

6.6.4 Resources.

The resources are described in Table 6.2

Resource	Description
csx.ko	Kernel module driver.
/dev/csxNc, /dev/csxNm	Control and Memory devices for the N'th CSX device on the PCI bus. Note that N is NOT the slot number but an arbitrary discovery order.
/sys/class/csxctl/csxNc	devclass entry for N'th card control device. This directory contains the status file for presenting user information.
/sys/class/csxmem/csxNm	devclass entry for Nth card memory device.

Table 6.2

7 Bibliography

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

[2] *Debugging with GDB*
Richard Stallman, Roland Pesch, Stan Shebs, et al.
ISBN 1-882114-77-9
Free Software Foundation, Inc.

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

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

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

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