

ClearSpeed[™]

ClearSpeed DFT Library

User Guide

Document No. 06-UG-1337 Revision: 4.B

28 September 2010

Contents

1	Introduction	4
1.1	Requirements	4
1.1.1	Package contents	4
1.2	Patterns of use	5
1.3	Performance guidelines	5
1.4	Environment variables	6
1.5	User examples	6
2	Basic usage	8
2.1	Supported features	8
2.2	Data formats	9
2.3	Memory descriptors	9
2.4	Plans	11
2.5	Flags	12
3	ClearSpeed DFT Host Library	14
4	ClearSpeed DFT Card Library	29
	Index	35

1 Introduction

The ClearSpeed Discrete Fourier Transform (CSDFT) library consists of two principle parts; a shared host-side library with a set of default Advance accelerator CSX600 executables which the host-side library requires to function and a number of static libraries for the CSX600. The host-side library is linked and called by an application running on the host computer in order to offload DFT operations to the Advance accelerator card(s). The static CSX600 card-side libraries are designed to be linked into a user created CSX600 executable running on the CSX600 itself. Creating their own CSX600 executable, allows users to extend the base functionality with their own routines and the host-side library can access them by using the user created CSX600 executable instead of the default executables. The two versions of the library have a common API and a number of functions in common. This manual has a section for each version of the library.

1.1 Requirements

In addition to the ClearSpeed base package (for any of our supported platforms), the ClearSpeed SDK should also be installed in order to build the user examples included with the CSDFT library.

1.1.1 Package contents

The CSDFT installation includes the following files

```
include
    csdft.h                <CSX600 processor include file>
include/csdft
    csdft.h                <host include files>
    csdft_support.h
    cs_complex.h
lib
    libcsdft.so            <shared host library - deprecated name>
    libcsxl_csdft.so      <shared host library>
csa
    libfft_cs_codelet.csa  <CSX600 processor static libraries>
    libfft_cs_processor.csa
    libfft_cs_twiddle.csa
    libfft_framework_gen.csa
csx
    fft_cs_processor_0.csx <CSX600 processor executables>
    fft_cs_processor_1.csx
examples/csdft
```

<Miscellaneous examples>

doc/csdft

<Documentation and license terms>

1.2 Patterns of use

The ClearSpeed DFT Library consists of two similar APIs, one of which runs on the host and one on the processors on a ClearSpeed Advance accelerator.

The CSDFT host-side API, while providing a similar interface to that of the CSDFT Advance accelerator card-side API, has additional heuristics which try and efficiently utilize both CSX600 processors on a card. To this end when a plan representing more than one DFT is executed the host-side library will automatically try and load-balance the work between the CSX600 processors. The host API is mostly intended to be the target for developers working to port existing host-side based applications, which may already use one of the many third party DFT libraries available (such as FFTW, MKL and ACML), to use the ClearSpeed Advance accelerator.

The CSDFT card-side API is very similar to the host-side API but the mode of execution is different. Calls made to the card-side API from within a Cn or assembly language program are run on the single CSX600 which made the call.

The CSDFT card-side API can also be thought of as an embedded version of the API. This means that any Cn or assembly language program can call CSDFT routines. This also means that you can construct a `.csx` file which uses the card-side CSDFT library but where the data operated on comes from the host, via the usual CSAPI calls, without having to use the host-side CSDFT library. This means it is more suitable for users working to port full applications to run more or less entirely on the Advance accelerator rather than simply trying to offload the DFT portions of their application to the Advance accelerator.

1.3 Performance guidelines

The current build of the CSDFT library running on a PCI-X Advance accelerator has particular performance characteristics that should be understood before using the library.

For an Advance X620 card the performance observed on the host, up to a certain point, is typically limited by PCI-X transfer speeds. This is due to the $N \log_2 N : N^2$ ratio of compute to IO with 1D transforms, in particular. Above a certain size, which can be dependent on host CPU, chipset and interface (PCI-X or PCIe) and how many transforms are performed at a time, card-side and host-side performance do start to converge. For straightforward transforms, peak performance per CSX600 of about 7 GFLOPS for single precision and 3.6 GFLOPS for double precision are possible.

If data is sent to a device and then multiple operations performed (such as forward and backward DFT pairs), before sending the data back to the host, then this is likely to give a much better overall performance as it can optimize memory bandwidth more effectively. For example when a convolution is required, the provided convolution interface will give much better performance than using the standard forward and backward interfaces. This is also true for user operations performed between DFTs, especially if they follow a similar pattern to a traditional forward DFT -> user operation -> backward DFT that the library currently supports.

1.4 Environment variables

There are three environment variables that can be set by a user which affect the behavior of the CSDFT host-side library.

The first is `CS_CSAPI_DEBUGGER`, which will need to be enabled when debugging with `csgdb` (you will also need to specify `CS_CSAPI_DEBUGGER_ATTACH` as well, please see the SDK documentation for further information about `csgdb` environment variables). This environment variable affects which version of the card-side executable that the host-side library will search for and load (release vs. debug). In this case it will be the debug versions of either the default CSDFT library CSX files or any user built CSX files. If they are not present then the library will signal an error. Please note that the CSDFT Library package only contains release versions of the CSX600 executables and static libraries.

Another environment variable that may be set is `CS_FFT_CSX` which is used when the user needs to override the default card executable with one which contains their user routines. The environment variable needs a full path to the relevant csx file and must include the root of the csx filename, but without any appended `_0`, `_1` or `_debug`. Be careful on Windows as the default installation location for ClearSpeed software is under the Program Files directory which has a space in it. This can cause Windows to add "" to any command line completed by using the [Tab] key and this will cause the host-side library to fail to locate the csx file. Please note the `CS_CSAPI_DEBUGGER` and `CS_CSAPI_DEBUGGER_ATTACH` should only be used in conjunction with a user CSX600 executable being defined by `CS_FFT_CSX` because of the lack of debug versions of the default CSX600 executables.

The last environment variable that you can set is `CS_USE_SIM`. When set this variable signals the host-side library that rather than connecting to an Advance accelerator, the library is actually connecting to one of ClearSpeed's software simulators. The CSX executable can take some time to load (possibly minutes).

1.5 User examples

The CSDFT Library package includes a set of examples which demonstrate the basic principles of using the CSDFT Library. They work both as an example of use and also a simple test that the package has been installed correctly and the library is working. If you have not installed the SDK from the ClearSpeed Developers archive then the examples which require user functions will not be built.

To build and test the examples (assuming all packages have been installed in the default location):

```
source /opt/clearspeed/csx600_m512_le/bin/bashrc
cd /opt/clearspeed/csx600_m512_le/examples/csdft/
make -s all test
```

If the ClearSpeed SDK is not installed then type:

```
make -s test INSTALLED_SDK=false
```

You should see messages of the form:

```
**** 1D forward reverse single precision complex test passed
```

The tests are split into separate directories and there is also a single directory which builds a user CSX600 executable file which some of the tests use (in the directory `user_csx`).

ClearSpeed DFT Library

The tests include straightforward 1D, 2D and 3D, single and double precision plans as well as some 'user function' plans.

2 Basic usage

This section covers the basic features, in terms of transform sizes, types and data formats supported by the CSDFT Library. It goes on to describe the basic operational model of the CSDFT Library particularly the concepts behind *memory descriptors* and *plans*.

2.1 Supported features

The supported sizes of transforms are limited to powers of two from 128 to 8192 for 1D, square and non-square powers of two from 128 to 2048 for 2D transforms, and 128 cubed for 3D transforms. The library provides support for forward and backward transforms, in single and double precision, real (specifically real to complex and complex to real) and complex floating point representation, with the data stored in either in interleaved (the default) or split array storage formats (a single array is the default for real data).

The supported interfaces for 1D DFTs are:

- Power of two sizes from 128 to 8192
- Single and double precision
- Complex types
 - Complex to complex (forward and backward)
- Interleaved
- Natural and optimal order (only optimal order supported by CSDFT card-side API see [Section 2.4: Plans on page 11](#))

The supported interfaces for 2D transforms are:

- Power of two square and non-square sizes from 128 to 2048
- Single and double precision
- Real and complex types
- Complex to complex (forward and backward)
 - Real to complex (forward)
 - Complex to real (backward)
- Interleaved and split array inputs
- Natural and optimal order

The supported interfaces for 3D transforms are:

- Size 128 cubed only
- Single and double precision
- Complex types
 - Complex to complex (forward and backward)
- Interleaved inputs
- Natural order only

2.2 Data formats

The two sample library data formats assumed are *interleaved* and *split*. In interleaved mode, real and imaginary data values are next to each other in one contiguous piece of memory as shown in [Figure 1](#).

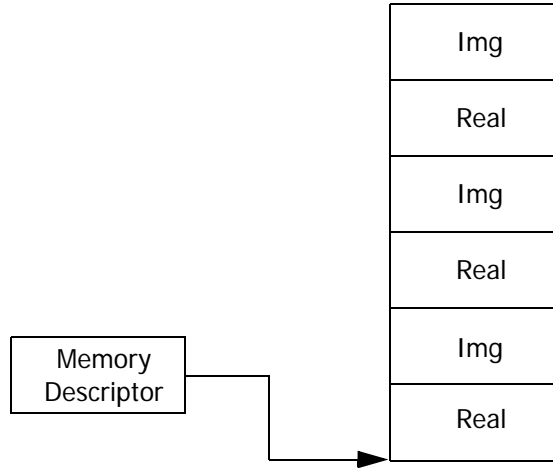


Figure 1. Interleaved mode

In split mode, the memory descriptor is used to point to two separate memory blocks, one for real, the other for imaginary data as shown in [Figure 2](#).

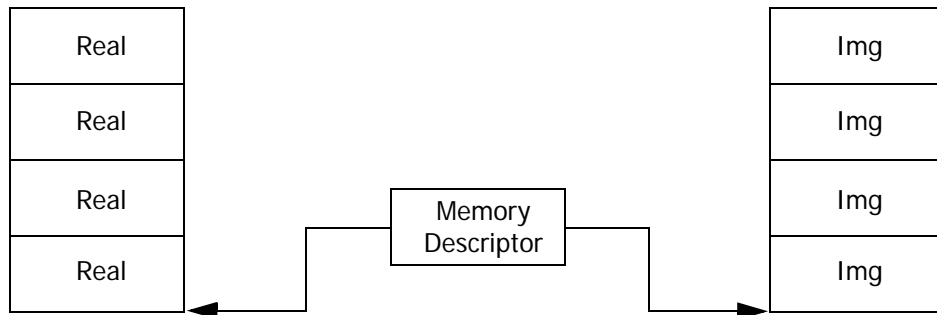


Figure 2. Split mode

The CSDFT library provides platform-independent single and double-precision structures, `FloatComplex` and `DoubleComplex`, which implement the interleaved memory layout. Developers may also use their own structures to implement interleaved or split mode data layout. Alternatively, if using a C99-compatible compiler, developers may wish to use the complex data types built into these languages (Fortran has a similar data type).

Data is also arranged in a row-major order (also known as 'C' format), which means that as you move through adjoining memory locations, the last dimension's index varies most quickly, and the first dimension's index varies most slowly.

2.3 Memory descriptors

The memory descriptor structure, `CSDftMemoryDescriptor`, provides an abstract datatype which can describe memory on the host or card to the CSDFT API.

The memory descriptor is essentially an intelligent pointer. As well as maintaining a pointer to the actual data, the memory descriptor also contains the following information:

- How the memory was allocated (allocated as part of the memory descriptor or referenced);
- The location on the host or card;
- The size of the block(s) of memory described;
- Whether there is one contiguous block of memory or it is split into two (necessary for split array inputs).

The memory descriptor may be used to describe input and outputs residing in memory for all DFT functions.

Through the use of these memory descriptors, developers do not need to concentrate on where data is being stored (be it on the host or on the card) when executing plans with data described by these descriptors. All the logic required to move the source data to the card to be executed on and where the destination is to be stored is handled by the CSDFT Library.

The memory descriptor can be used to maximize library efficiency by ensuring that data is moved from the host to the card, multiple operations performed on it and then returned to the host. Instead of moving the data from the host to the card and back for each operation.

An example of the `CSDftMemoryDescriptor` is shown in the code fragment below:

```
// Declare memory descriptors to describe input and output data
CSDftMemoryDescriptor data_in;
CSDftMemoryDescriptor data_out;

// Declare other variables
int n=128, num_ffts=3;

// Create input and output data memory descriptors
// referencing memory on the host
data_in = CSDFT_system_to_memory_descriptor((void *)
source_data_pointer,
                                           sizeof(DoubleComplex) * n *
num_ffts);
data_out = CSDFT_system_to_memory_descriptor((void *)
output_data_pointer,
                                           sizeof(DoubleComplex) * n *
num_ffts);
// Create the plan
plan = CSDFT_create_plan_1d(...);

// Execute plan
status = CSDFT_execute_dft(plan, data_in, data_out);
```

2.4 Plans

The CSDFT library uses the concept of a *plan* to define a DFT before it is executed. A plan contains information about:

- the direction
- the precision
- the data format
 - whether data is complex or real
 - whether data is natural or optimal
- the number of samples
- the scale factor to use

All but the last two in the list above are set by using flags. The specified flags are ORed together and passed into the “create plan” functions as an unsigned integer – a list of the flags with an explanation of their meaning is provided in [Section 2.5: Flags on page 12](#).

Data manipulated on the card and returned to the host can be in one of two formats, natural order (using the `CSDFT_NATURAL_ORDER` flag) or a more optimal order (`CSDFT_OPTIMAL_ORDER`) used by the CSDFT library. The default is natural order, though specifying this will incur a performance penalty on the card (This mode is not supported directly for 1D on the card but uses the host to perform the transform from optimal to natural order). Note that it is possible to work only in optimal order and to this end the library provides some support routines to move data between these two data formats.

To achieve no apparent scaling in a forward DFT, the scale factor should be 1.0. In a backwards DFT however, the scale factor should be $1/(n^{DIMENSION})$. For example, a 1D DFT of size 128 would have a scale factor of 1/128, while a 2D DFT of size 128x128 would have a scale factor of 1/(128*128)

For example:

```
unsigned int flags;
CSDftPlan forward;
CSDftStatus status;

// Create the flags for a forward, optimal, double and complex DFT
flags = CSDFT_FORWARD | CSDFT_OPTIMAL_ORDER | CSDFT_DOUBLE |
CSDFT_COMPLEX;

// Create the 1D plan - 5 ffts each of size 256
forward = CSDFT_create_plan_1d(flags, 256, 5, 1.0);

// Execute the plan
status = CSDFT_execute_dft(forward, ...);
```

2.5 Flags

A number of functions use flag parameters to provide information about the operation to be performed. These are described in [Table 1](#).

Flag	Description	Applicable
CSDFT_FORWARD	Perform a forward DFT	All
CSDFT_BACKWARD	Perform a backward DFT	All except convolution
CSDFT_NATURAL_ORDER	For forward DFTs, set order of output to natural. For backward DFTs, assume order to input is natural.	All except 1D card-side function
CSDFT_OPTIMAL_ORDER	For forward DFTs, set order of output to optimal. For backward DFTs, assume order to input is optimal.	All
CSDFT_SRC_SINGLE	Source is a single precision floating point format	All
CSDFT_SRC_DOUBLE	Source is in double precision floating point format	All
CSDFT_SRC_REAL	Source is of real type (Only valid for a forward Real to Complex forward transform)	Real to Complex forward transform
CSDFT_SRC_COMPLEX	Source is of complex type. This is the default	All
CSDFT_SRC_INTERLEAVED	Source is an interleaved array of data. This is the default.	All
CSDFT_SRC_SPLIT_ARRAY	Source is two arrays of data, one representing the real component, the other imaginary. Only valid for complex data. Not valid for optimal format data (for instance with a backward DFT).	All except optimal format data and 1D DFT.
CSDFT_DST_SINGLE	Destination is a single precision floating point format	All
CSDFT_DST_DOUBLE	Destination is a double precision floating point format	All
CSDFT_DST_REAL	Destination is of real type (Only valid for a Complex to Real backward transform)	Complex to real backward transform
CSDFT_DST_COMPLEX	Destination is of complex type. This is the default	All
CSDFT_DST_INTERLEAVED	Destination is an interleaved array of data. This is the default	All
CSDFT_DST_SPLIT_ARRAY	Destination is two arrays of data, one representing the real component, the other imaginary. Only valid for complex data. Not valid for optimal format data (for instance with a backward DFT).	All except optimal format data

Table 1. Flag parameters

ClearSpeed DFT Library

CSDFT_SINGLE	Source and destination are in single precision floating point format	All
CSDFT_DOUBLE	Source and destination are in double precision floating point format	All
CSDFT_REAL	Source and destination are of real type	All
CSDFT_COMPLEX	Source and destination are of complex type. This is the default	All
CSDFT_TRANSFORM_KERNEL	The kernel will undergo a DFT before being multiplied with the transformed source	Convolution

Table 1. Flag parameters

3 ClearSpeed DFT Host Library

This section documents the ClearSpeed DFT host-side interface.

CSDFT_create_plan_1d, CSDFT_create_plan_2d, CSDFT_create_plan_3d — Create a 1,2 or 3D plan for a DFT

```
#include "csdft/csdft.h"
CSDftPlan CSDFT_create_plan_1d( CSDftFlags flags, unsigned int size,
unsigned int num_ffts, double scale );
CSDftPlan CSDFT_create_plan_2d( CSDftFlags flags, unsigned int
size_x, unsigned int size_y, unsigned int num_ffts, double scale );
CSDftPlan CSDFT_create_plan_3d( CSDftFlags flags, unsigned int
size_x, unsigned int size_y, unsigned int size_z, unsigned int
num_ffts, double scale );
```

Description

These functions create plans for 1, 2 and 3 dimensional DFTs, which can then be executed using the CSDFT_execute_dft routine.

This library supports:

1D DFT

- Power of two sizes from 128 to 8192
- Single and Double precision
- Complex to Complex (forward and backward)
- Interleaved
- Natural and Optimal order else
- Optimal

2D DFT

- Square and non square Power of two sizes from 128 to 2048
- Single and Double precision
- Real and Complex types:
 - Complex to Complex (forward and backward)
 - Real to Complex (forward)
 - Complex to Real (backward)
- Interleaved and Split Array inputs
- Natural and Optimal order

3D DFT

- 128 cubed size only
- Single and Double precision
- Complex to Complex (forward and backward)
- Interleaved

- Natural order only

The complex data generated from a real to complex forward 2D DFT is a partial representation of the full complex output. This is due to the symmetry of the resulting data. Only the first $(n/2)+1$ columns are represented; in natural mode this means each row has only $(n/2)+1$ samples, with no padding at the end of the row. In optimal mode, only the first $(n/2)+1$ columns are output. When doing a backward complex to real DFT, this half-complex format is assumed.

It is possible to create multiple plans and relate them to multiple DFT computations.

The ClearSpeed DFT library performs the following forward 1D FFT:

$$Y[k]=\text{Sum}\{j=0,n-1\}(X[j]*\exp(-2*\pi*j*k*\text{sqrt}(-1)/n))$$

The ClearSpeed DFT library performs the following backward 1D DFT:

$$Y[k]=\text{Sum}\{j=0,n-1\}(X[j]*\exp(2*\pi*j*k*\text{sqrt}(-1)/n))$$

All the various configuration parameters related to a DFT are contained in a plan object in the CSDFT API. A descriptor is created by the following API calls for 1d, 2d and 3D versions of `CSDFT_create_plan_<1D|2D|3D>`.

Returns

This function returns a plan if there are no errors found in the supplied flags or parameters, otherwise it returns NULL. If there is an error, then the global status will be set to indicate an error, which can be retrieved by using `CSDFT_get_status()`.

CSDFT_execute_dft — Synchronous Execution of a DFT

```
#include "csdft/csdft.h"
CSDftStatus CSDFT_execute_dft( const CSDftPlan plan,
CSDftMemoryDescriptor source, CSDftMemoryDescriptor destination );
```

Description

This function performs the DFT as described by the plan parameter, using data specified by the source and destination memory descriptors. This function is a synchronous interface and will only return when the DFT has completed.

Returns

Returns a status code of `CSDFT_NO_ERROR` if the function is successful, `CSDFT_INVALID_PLAN` if the plan passed as a parameter is not valid (such as the handle being NULL), `CSDFT_INVALID_MEMORY_DESCRIPTOR` if the handles for the memory descriptors are invalid or `CSDFT_BOARD_ERROR` if the library cannot connect to an Advance Card.

CSDFT_free_plan — Free a DFT plan

```
#include "csdft/csdft.h"
CSDftStatus CSDFT_free_plan( CSDftPlan plan );
```

Description

This function frees up the previously created plan.

Returns

This function frees all associated resources for a plan and returns a CSDftStatus result of CSDFT_NO_ERROR or CSDFT_INVALID_MEMORY_DESCRIPTOR if *plan* is a null pointer.

CSDFT_create_user_function — Create user function

```
#include "csdft/csdft.h"
CSDftUserFunction CSDFT_create_user_function(const char *func,
unsigned int common_size, unsigned int block_size_src, unsigned int
block_size_dst, unsigned int num_blocks);
```

Description

This function creates a handle to a 'user function', which is in the ClearSpeed executable (CSX) on the ClearSpeed Advance board. This function needs to be built into 2 CSX files (one for each chip) which the user has compiled. These CSX files need to have been linked with the board CSDFT library. They need to be of the form name_0.csx and name_1.csx. If debugging is necessary, then `_debug` should be added after the digit e.g. `myscx_0_debug.csx`. In order to load this CSX file the environment variable `CS_FFT_CSX` must be set to be the name part of the CSX without the suffixes e.g. `/home/csx/myscx`

The user function must be of the following format:

```
void funcname(CSDftMemoryDescriptor common, CSDftMemoryDescriptor in,
CSDftMemoryDescriptor out);
```

This first parameter in `CSDFT_create_user_function()`, *func*, is the name of the function to be called in the CSX executable on the board. *block_size_src* and *block_size_dst* are the size in bytes of the source and the destination respectively, and *num_blocks* is the number of blocks in the source and destination.

The number of blocks indicates how many times the user function will be called. Each time, an amount of data equal to *block_size_src* will be given to the user function, and an amount of data equal to *block_size_dst* will be written to the destination. The common data parameter is broadcast once to each CSX600 running the user function.

To actually execute the board function, `CSDFT_execute_user_function()` must be called.

Examples

```
/*
To create a handle to a user function called myfunc
*/
CSDftUserFunction func;
/*
Send 1Kb of data to myfunc() at a time; repeat 10 times.
*/
func = CSDFT_create_user_function("myfunc", 0, 1024, 1024, 10);
```

Returns

This function returns a CSDftUserFunction if there are no errors, other it returns NULL. If there is an error, the global status will be set to indicate an error and can be retrieved by using `CSDDFT_get_status()`.

CSDFT_execute_user_function — Synchronous Execution of a user function

```
#include "csdft/csdft.h"
CSDftStatus CSDFT_execute_user_function(CSDftUserFunction
func,CSDftMemoryDescriptor common, CSDftMemoryDescriptor in,
CSDftMemoryDescriptor out, CSDftProcessorID processor );
CSDftStatus CSDFT_execute_user_function(CSDftUserFunction
func,CSDftMemoryDescriptor common, CSDftMemoryDescriptor in,
CSDftMemoryDescriptor out );
```

Description

This function executes a user function residing in a CSX executable on a ClearSpeed Advance Board. The first parameter, *func*, is the handle to the user function, and is created using *CSDFT_create_user_function()*. *in* and *out* are memory descriptors describing the source and destination data locations respectively. *common* is a memory descriptor pointing to data to be sent to all relevant chips on the Advance boards (i.e. will act like a broadcast).

processor describes to which board and processor this data should be sent to. To choose a specific processor on a specific board, *CSDFT_BOARD_N* should be ORed with *CSDFT_PROCESSOR_M*, where N is the board number (starting at 0 and up to 6) and M is the processor number (starting at 0 and up to 3). Alternatively, developers can choose to send data to any free processor in sequence - to do this, *CSDFT_BOARD_SEQUENCE* should be ORed with *CSDFT_PROCESSOR_SEQUENCE*. If a specific processor is chosen, currently only one block of data may be sent (i.e the *num_blocks* parameter of *CSDFT_create_user_function()* must be 1).

The input blocks of data will be read contiguously from the input descriptor and written to the output descriptor.

The input blocks of data will be taken contiguously from the input descriptor and written out to the output descriptor.

```
/*
To create a handle to a user function called myfunc
*/
CSDftUserFunction func;
/*
Send 10Kb of data to myfunc(), 1Kb at a time.
*/
func = CSDFT_create_user_function("myfunc", 0, 1024, 1024, 10);
/*
Execute the function, sending data to whichever processor is free
*/
CSDFT_execute_user_function(func, md_null, md_in, md_out,
CSDFT_BOARD_SEQUENCE | CSDFT_PROCESSOR_SEQUENCE)
To create a handle to a user function called myfunc
*/
CSDftUserFunction func;
//Send 1Kb of data to myfunc() at a time; repeat 10 times.
func = CSDFT_create_user_function("myfunc", 0, 1024, 1024, 10);
```

Returns

This function returns a *CSDftUserFunction* if there are no errors, other it returns *NULL*. If there is an error, the status will be set to indicate an error.

CSDFT_free_user_func — Frees a previously created user function.

```
#include "csdft/csdft.h"
CSDftStatus CSDFT_free_user_func( CSDftUserFunc handle ) ;
```

Description

This function frees a previously created user function. If this function cannot free the user function, it returns CSDFT_INVALID_USER_FUNCTION.

Returns

This function returns CSDFT_NO_ERROR or CSDFT_INVALID_USER_FUNCTION as required.

CSDFT_create_convolution_plan_2d — Create a 2D plan for a convolution operation

```
#include "csdft/csdft.h"
CSDftPlan CSDFT_create_convolution_plan_2d(CSDftFlags flags, unsigned
int size_x, unsigned int size_y, unsigned int num_ffts, double scale);
```

Description

Performing a convolution is similar to any other DFT - a plan should be created, specifying the correct flags and then executed. As usual, flags are ORed together - however one flag of specific note is CSDFT_TRANSFORM_KERNEL. If set, the kernel will undergo a DFT before being convolved with the transformed source. Otherwise, the kernel will be convolved "as is" with the transformed source. By default, the kernel is assumed to not need to be pre-transformed, and, in this implementation, the kernel is assumed to be the result of a single DFT. Therefore, the same kernel will be reused if *num_ffts* is greater than 1.

To perform a convolution with no apparent scaling, the scale parameter should be $1/(size*size)$. For example, scale would be $1/(512*512)$ when performing a 512x512 DFT.

The complex data generated from a real to complex forward 2D DFT is a partial representation of the full complex output. This is due to the symmetry of the resulting data. Only the first $(n/2)+1$ columns are represented; in natural mode this means each row has only $(n/2)+1$ samples, with no padding at the end of the row. In optimal mode, only the first $n/2+1$ columns are output. When performing a backward complex to real DFT, this half-complex format is assumed. The kernel MUST be in optimal order when performing convolutions.

Returns

This function returns a plan if there are no errors found in the supplied flags or parameters, otherwise it returns NULL. If there is an error, then the global status will be set to indicate an error, which can be retrieved by using *CSDFT_get_status()*.

CSDFT_execute_convolution — Synchronous Execution of a convolution

```
#include "csdft/csdft.h"
CSDftStatus CSDFT_execute_convolution( const CSDftPlan plan,
CSDftMemoryDescriptor kernel, CSDftMemoryDescriptor source,
CSDftMemoryDescriptor destination ) ;
```

Description

This function uses memory descriptors to specify the memory locations of the kernel, source and destination data. A memory descriptor is an opaque data structure which represents data, both in host and board space. More information on this structure is available in the Introduction section.

If the convolution is performed correctly and no errors occur, the solution will be in the memory space referenced by the destination memory descriptors.

Returns

Returns a status code of CSDFT_NO_ERROR if the function is successful, CSDFT_INVALID_PLAN if the plan passed as a parameter is not valid (such as the handle being NULL), CSDFT_INVALID_MEMORY_DESCRIPTOR if the handles for the memory descriptors are invalid or CSDFT_BOARD_ERROR if the library cannot connect to an Advance Board.

CSDFT_get_status — Returns the current status for the CSDFT library

```
#include "csdft/csdft.h"
CSDftStatus CSDFT_get_status( void );
```

Description

This function returns the current status code for the CSDFT library. Specifically it returns the status code of the last executed CSDft library function. All Memory Descriptor, Plan and Execute functions will either set the global status code or return their status code as a return value as their prototype dictates.

The following are the valid status codes which will be returned or set by CSDFT library functions.

- CSDFT_NO_ERROR No current error detected.
- CSDFT_INVALID_MEMORY_DESCRIPTOR When set by one of the Execute functions or one of the memory manipulation functions this code indicates that one of the pointer parameters to the function were NULL.
- CSDFT_MEMORY_ALLOCATION_ERROR When set by one of the Memory Descriptor functions that allocate memory this code indicates that a memory allocation error has been encountered either on the host or on the Advance board.
- CSDFT_BOARD_ERROR This code can be set as a result of the CSDFT library not being able to find an Advance board to connect to. If there is definitely a board present in the computer, typing "csreset -v" into your command prompt should identify or fix the problem. Alternatively, this error is set due to an internal, unrecoverable error on the board or in the driver.
- CSDFT_INVALID_BOARD This code is returned when an operation is adjudged to be sent to a board that does not exist.
- CSDFT_INVALID_PROCESSOR This code is returned when an operation is adjudged to be sent to a processor that does not exist on a board that does exist.
- CSDFT_INVALID_PLAN When set by one of the planner functions this code indicates that one or more of the flags of a requested DFT was invalid. When returned

by one of the execute functions it indicates that the plan was somehow invalid or not defined (NULL).

- **CSDFT_INVALID_SYMBOL** Set as a result of one of the User Planner functions not finding the symbol in the loaded CSX file on the Advance board. Usually this is the result of the wrong path being set in the CS_FFT_CSX environment variable.
- **CSDFT_INVALID_SIZE** Set as a result of an unsupported size being specified as part of the plan. This is most likely the result of a size being set that is not currently supported by the CSDFT library (currently a power of 2).
- **CSDFT_INVALID_FLAG** Set as a result of incompatible options being chosen when creating a plan. An example of this is choosing different source and destination formats.
- **CSDFT_INVALID_PARAMETER** Set as a result of incompatible parameters being specified when calling a function.
- **CSDFT_INVALID_USER_FUNCTION** Set as a result of an invalid user function handle being specified when calling a function. An example of this is passing a null handle to *CSDFT_execute_user_function()*.

This function can be used to test for reasons behind unexpected behaviour and when used in conjunction with the *CSDFT_return_error_message()* can provide textual messages for the user at runtime.

Returns

This function returns a status of CSDFT_NO_ERROR if has been no error in the last performed function, otherwise it returns another of the listed status codes.

CSDFT_return_error_message — Returns a string describing a status value of the CSDFT library

```
#include "csdft/csdft.h"
const char* CSDFT_return_error_message( CSDftStatus status );
```

Description

This function returns a string representing a textual description of the the current status code for the CSDFT library.

The following are the status strings returned by the routine.

- "No error" Corresponds to the CSDFT_NO_ERROR status code.
- "Error: Allocating memory" Corresponds to the CSDFT_MEMORY_ALLOCATION status code.
- "Error: Invalid plan" Corresponds to the CSDFT_INVALID_PLAN status code.
- "Error: Invalid symbol" Corresponds to the CSDFT_INVALID_SYMBOL status code.
- "Error: Board not found or unrecoverable" Corresponds to the CSDFT_BOARD_ERROR status code.
- "Error: Invalid board" Corresponds to the CSDFT_INVALID_BOARD status code.
- "Error: Invalid flag" Corresponds to the CSDFT_INVALID_FLAG status code.
- "Error: Invalid size" Corresponds to the CSDFT_INVALID_SIZE status code.

- "Error: Invalid processor" Corresponds to the CSDFT_INVALID_PROCESSOR status code.
- "Error: Invalid memory descriptor" Corresponds to the CSDFT_INVALID_MEMORY_DESCRIPTOR status code.

Returns

This function returns a const char * string which corresponds to the status code used as an argument.

CSDFT_get_null_descriptor — Return a special null memory descriptor

```
#include "csdft/csdft.h"  
CSDftMemoryDescriptor CSDFT_get_null_descriptor( void ) ;
```

Description

This function returns a null memory descriptor which is a special type of memory descriptor that indicates to a routine to which it is passed that the parameter is deliberately set to null. This helps with parameter error checking for the library.

For example, a null memory descriptor may be used when executing a user function, when one of the parameter sizes is zero.

Returns

This function returns a handle to a CSDftMemoryDescriptor on success and a NULL handle on failure as well as setting the global status code to CSDFT_NO_ERROR or CSDFT_MEMORY_ALLOCATION_ERROR as required.

CSDFT_malloc_host — Allocates a memory descriptor encapsulating a data buffer allocated on the host

```
#include "csdft/csdft.h"  
CSDftMemoryDescriptor CSDFT_malloc_host( unsigned int size_in_bytes )  
;
```

Description

This function returns a CSDftMemoryDescriptor, where the data buffer is allocated on the host by the routine. The parameter is the size in bytes of the buffer required. If the function cannot allocate memory for the buffer for any reason it returns a NULL handle and sets the global status code to be CSDFT_MEMORY_ALLOCATION_ERROR. In order to safely access the buffer in the case where the *CSDFT_malloc_host()* function has created the buffer, you should use the *CSDFT_memory_descriptor_to_system()* call to provide a pointer to the buffer.

Returns

This function returns a handle to a CSDftMemoryDescriptor on success and a NULL handle on failure as well as setting the global status code to CSDFT_NO_ERROR or CSDFT_MEMORY_ALLOCATION_ERROR as required.

CSDFT_memory_descriptor_to_system — Given a host based memory descriptor returns a pointer to a buffer in system memory

```
#include "csdft/csdft.h"
void * CSDFT_memory_descriptor_to_system( CSDftMemoryDescriptor handle
) ;
```

Description

This function returns a void * pointer to the data buffer that is encapsulated by the CSDftMemoryDescriptor. The parameter is a handle to a CSDftMemoryDescriptor. If the function cannot return a valid pointer to system allocated data, for example if a board based memory descriptor was passed in, then it returns a NULL pointer and sets the global status code to be CSDFT_MEMORY_ERROR.

Returns

This function returns a pointer to system allocated data on success and a NULL pointer on failure as well as setting the global status code to CSDFT_NO_ERROR or CSDFT_INVALID_MEMORY_DESCRIPTOR as required.

CSDFT_system_to_memory_descriptor — Returns a host based memory descriptor encapsulating user provided source data.

```
#include "csdft/csdft.h"
CSDftMemoryDescriptor CSDFT_system_to_memory_descriptor( void *
source_data, unsigned int size_in_bytes ) ;
```

Description

This function returns a CSDftMemoryDescriptor where the data buffer was pre-allocated on the host. The parameters are: a pointer to the array of data already allocated on the host and the size in bytes of the buffer. If the function cannot for some reason create a Memory Descriptor it returns a NULL handle and sets the global status code to be CSDFT_MEMORY_ERROR.

Returns

This function returns a handle to a CSDftMemoryDescriptor on success and a NULL handle on failure as well as setting the global status code to CSDFT_NO_ERROR or CSDFT_MEMORY_ALLOCATION_ERROR as required.

CSDFT_system_split_array_to_memory_descriptor — Creates a host based memory descriptor encapsulating two user provided source data arrays

```
#include "csdft/csdft.h"
CSDftMemoryDescriptor CSDFT_system_split_array_to_memory_descriptor(
void * source_data_real, void * source_data_imag, unsigned int
size_in_bytes )
```

Description

This function returns a CSDftMemoryDescriptor encapsulating the separate real and imaginary data buffers which are pre-allocated on the host. The parameters are: two pointers to arrays of data already allocated on the host (representing real and imaginary parts) and the size in bytes of the data. If the function cannot for some reason create a

Memory Descriptor it returns a NULL handle and sets the global status code to be CSDFT_MEMORY_ALLOCATION_ERROR.

Returns

This function returns a handle to a CSDftMemoryDescriptor on success and a NULL handle on failure as well as setting the global status code to CSDFT_NO_ERROR or CSDFT_MEMORY_ALLOCATION_ERROR as required.

CSDFT_processor_to_memory_descriptor — Returns a board based memory descriptor encapsulating user provided source data.

```
#include "csdft.h"
CSDftMemoryDescriptor CSDFT_processor_to_memory_descriptor( unsigned
int source_data, unsigned int size_in_bytes, unsigned int
board_instance, unsigned int processor_instance ) ;
```

Description

This function returns a CSDftMemoryDescriptor where the data buffer was pre-allocated on the board. The parameters are: a pointer to the array of data already allocated on the board, the size in bytes of the buffer, the board number (if only one board is in use, this will be 0) and the processor the memory should be allocated on. If the function cannot for some reason create a Memory Descriptor it returns a NULL handle and sets the global status code to be CSDFT_MEMORY_ALLOCATION_ERROR.

Returns

This function returns a handle to a CSDftMemoryDescriptor on success and a NULL handle on failure as well as setting the global status code to CSDFT_NO_ERROR or CSDFT_MEMORY_ALLOCATION_ERROR as required.

CSDFT_memory_descriptor_copy — Copies data encapsulated in a memory descriptor

```
#include "csdft/csdft.h"
CSDFTStatus CSDFT_memory_descriptor_copy( CSDftMemoryDescriptor
source, CSDftMemoryDescriptor destination, unsigned int size_in_bytes
);
```

Description

This function copies the *size_in_bytes* amount of data from the memory descriptor *source* to the memory descriptor *destination*. If *destination* is not the same or bigger size as *size_in_bytes*, a memory descriptor error will be returned. This function is especially useful when copying to and from the board, as the function will ensure previous operations have completed.

Returns

This function returns a CSDFTStatus code.

CSDFT_free — Deallocates a memory descriptor created by the CSDFT library

```
#include "csdft/csdft.h"
CSDftStatus CSDFT_free( CSDftMemoryDescriptor handle ) ;
```

Description

This function deallocates a memory descriptor created by one of a number of routines provided by the CSDFT library. The function is passed a handle to a CSDftMemoryDescriptor and returns a CSDftStatus code indicating success or failure. In the case of the Memory Descriptor object being deallocated without any problems this will be CSDFT_NO_ERROR and in the case where there was a problem, such as trying to deallocate an already freed object, the status code will be CSDFT_MEMORY_ERROR.

Returns

This function returns a status of CSDFT_NO_ERROR or CSDFT_MEMORY_ERROR.

CSDFT_bitreverse_1D_c — Returns a bitreversed version of the single precision input data

```
#include "csdft/csdft_support.h"
CSDftStatus CSDFT_bitreverse_1D_c      ( void          *data,
                                         unsigned int  x_size,
                                         unsigned int
                                         number_of_arrays ) ;
```

Description

This function performs bit reversal and conversion to and from natural and optimal board format for the input data. Currently supports conversion only for 1D and 2D complex interleaved arrays.

For any DFT there exists an optimal input or output.

- 1D forward output and 1D backward input: values are in bit reversed order
- 2D forward output and 2D backward input: values are in natural column major order, with the contents of each column in bit-reversed order. (i.e. column 0, column 1, column 2)

For the 1D bit-reverse routines the *x_size* parameter represents the number of elements within each array, and the *number_of_arrays* parameter describes the number of individual arrays of data to be operated upon.

For the 2D routines, the *x_size* parameter represents the number of columns and the *y_size* parameter represents the number of rows.

Returns

This function returns a status of CSDFT_NO_ERROR if has been no error in the last performed function, otherwise it returns another of the library status codes.

CSDFT_bitreverse_1D_z — Returns a bitreversed version of the double precision input data

```
#include "csdft/csdft_support.h"
CSDftStatus CSDFT_bitreverse_1D_z      ( void          *data,
                                         unsigned int  x_size,
                                         unsigned int
                                         number_of_arrays ) ;
```

Description

This function performs bit reversal and conversion to and from natural and optimal board format for the input data. Currently supports conversion only for 1D and 2D complex interleaved arrays.

For any DFT there exists an optimal input or output.

- 1D forward output and 1D backward input: values are in bit reversed order
- 2D forward output and 2D backward input: values are in natural column major order, with the contents of each column in bit-reversed order. (i.e. column 0, column 1, column 2)

For the 1D bit-reverse routines the *x_size* parameter represents the number of elements within each array, and the *number_of_arrays* parameter describes the number of individual arrays of data to be operated upon.

For the 2D routines, the *x_size* parameter represents the number of columns and the *y_size* parameter represents the number of rows.

Returns

This function returns a status of `CSDFT_NO_ERROR` if has been no error in the last performed function, otherwise it returns another of the library status codes.

CSDFT_optimal_to_natural_2D_c — Transforms the input from board optimal order to natural order for single precision data

```
#include "csdft/csdft_support.h"
CSDftStatus CSDFT_optimal_to_natural_2D_c( void          *data,
                                           unsigned int  x_size,
                                           unsigned int  y_size,
                                           unsigned int
number_of_arrays ) ;
```

Description

This function performs bit reversal and conversion to and from natural and optimal board format for the input data. Currently supports conversion only for 1D and 2D complex interleaved arrays.

For any DFT there exists an optimal input or output.

- 1D forward output and 1D backward input: values are in bit reversed order
- 2D forward output and 2D backward input: values are in natural column major order, with the contents of each column in bit-reversed order. (i.e. column 0, column 1, column 2)

For the 1D bit-reverse routines the *x_size* parameter represents the number of elements within each array, and the *number_of_arrays* parameter describes the number of individual arrays of data to be operated upon.

For the 2D routines, the *x_size* parameter represents the number of columns and the *y_size* parameter represents the number of rows.

Returns

This function returns a status of `CSDFT_NO_ERROR` if has been no error in the last performed function, otherwise it returns another of the library status codes.

CSDFT_optimal_to_natural_2D_z — Transforms the input from board optimal order to natural order for double precision data

```
#include "csdft/csdft_support.h"
CSDftStatus CSDFT_optimal_to_natural_2D_z( void      *data,
                                           unsigned int x_size,
                                           unsigned int y_size,
                                           unsigned int
number_of_arrays ) ;
```

Description

This function performs bit reversal and conversion to and from natural and optimal board format for the input data. Currently supports conversion only for 1D and 2D complex interleaved arrays.

For any DFT there exists an optimal input or output.

- 1D forward output and 1D backward input: values are in bit reversed order
- 2D forward output and 2D backward input: values are in natural column major order, with the contents of each column in bit-reversed order. (i.e. column 0, column 1, column 2)

For the 1D bit-reverse routines the *x_size* parameter represents the number of elements within each array, and the *number_of_arrays* parameter describes the number of individual arrays of data to be operated upon.

For the 2D routines, the *x_size* parameter represents the number of columns and the *y_size* parameter represents the number of rows.

Returns

This function returns a status of CSDFT_NO_ERROR if has been no error in the last performed function, otherwise it returns another of the library status codes.

CSDFT_natural_to_optimal_2D_c — Transforms the input from natural order to board optimal order for single precision data

```
#include "csdft/csdft_support.h"
CSDftStatus CSDFT_natural_to_optimal_2D_c( void      *data,
                                           unsigned int x_size,
                                           unsigned int y_size,
                                           unsigned int
number_of_arrays ) ;
```

Description

This function performs bit reversal and conversion to and from natural and optimal board format for the input data. Currently supports conversion only for 1D and 2D complex interleaved arrays.

For any DFT there exists an optimal input or output.

- 1D forward output and 1D backward input: values are in bit reversed order
- 2D forward output and 2D backward input: values are in natural column major order, with the contents of each column in bit-reversed order. (i.e. column 0, column 1, column 2)

For the 1D bit-reverse routines the *x_size* parameter represents the number of elements within each array, and the *number_of_arrays* parameter describes the number of individual arrays of data to be operated upon.

For the 2D routines, the *x_size* parameter represents the number of columns and the *y_size* parameter represents the number of rows.

Returns

This function returns a status of `CSDFT_NO_ERROR` if has been no error in the last performed function, otherwise it returns another of the library status codes.

CSDFT_natural_to_optimal_2D_z — Transforms the input from natural order to board optimal order for double precision data

```
#include "csdft/csdft_support.h"
CSDftStatus CSDFT_natural_to_optimal_2D_z( void      *data,
                                           unsigned int x_size,
                                           unsigned int y_size,
                                           unsigned int
number_of_arrays ) ;
```

Description

This function performs bit reversal and conversion to and from natural and optimal board format for the input data. Currently supports conversion only for 1D and 2D complex interleaved arrays.

For any DFT there exists an optimal input or output.

- 1D forward output and 1D backward input: values are in bit reversed order
- 2D forward output and 2D backward input: values are in natural column major order, with the contents of each column in bit-reversed order. (i.e. column 0, column 1, column 2)

For the 1D bit-reverse routines the *x_size* parameter represents the number of elements within each array, and the *number_of_arrays* parameter describes the number of individual arrays of data to be operated upon.

For the 2D routines, the *x_size* parameter represents the number of columns and the *y_size* parameter represents the number of rows.

Returns

This function returns a status of `CSDFT_NO_ERROR` if has been no error in the last performed function, otherwise it returns another of the library status codes.

CSDFT_ilog2 — Returns the integer log to the base two value of the input

```
#include "csdft/csdft_support.h"
unsigned int CSDFT_ilog2 ( unsigned int n ) ;
```

Description

Calculates the base 2 log of *n*. This can be used to determine the number of significant bits in the value. This function is used by the bit reverse functions.

Returns

The base 2 logarithm of the parameter.

CSDFT_get_symbol_value — Obtains the Board memory address of a specified symbol (on the first device of an Advance Board)

```
#include "csdft/csdft.h"
CSDftStatus CSDFT_get_symbol_value( const char *symbol, unsigned int
*symbol_address, unsigned int board_instance, unsigned int
processor_instance );
```

Description

This function is used to obtain the address of a symbol in a CSX file. This address is then used when creating a memory descriptor on the board, using *CSDFT_processor_to_memory_descriptor()*. The parameters are: *symbol* is the name of the symbol within the CSX executable, *symbol_address* is a pointer to an integer which will be populated with the symbol address, *board_instance* is the board number (starting at 0), and *processor_instance* is the processor number (again, starting at 0).

Examples

In the CSX executable on the ClearSpeed Advance board:

```
double blank_fft[1024][1024][2];
```

In the host executable:

```
unsigned int address = 0;
CSDFTMemoryDescriptor mem_desc;
/* Get the address of the symbol */
CSDFT_get_symbol_value("blank_fft", &int, 0, 0);
/* Allocate memory on board 0, processor 0 */
mem_desc = CSDFT_processor_to_memory_descriptor(int,
sizeof(Double)*1024*1024*2, 0, 0);
```

Returns

This function returns a status of CSDFT_NO_ERROR if has been no error in the last performed function, otherwise it returns another of the library status codes.

CSDFT_get_csapi_handle — Obtains the CSAPI handle for the first instance of an Advance Board in a system used by the CSDFT library

```
#include "csdft/csdft_support.h"
CSDftStatus CSDFT_get_csapi_handle ( struct CSAPIState **handle
) ;
CSDftStatus CSDFT_get_csapi_handle_board( struct CSAPIState
**handle,
unsigned int board_instance
) ;
```

Description

These functions allow the user to obtain a valid CSAPI handle to a particular Advance Board. These functions should be used if the developer wishes to access the low level CSAPI (which is not recommended), and otherwise should be avoided.

Returns

This function returns a status of CSDFT_NO_ERROR if has been no error in the last performed function, otherwise it returns CSDFT_INVALID_BOARD.

4 ClearSpeed DFT Card Library

This section documents the ClearSpeed DFT card-side interface.

CSDFT_create_plan_1d, CSDFT_create_plan_2d, CSDFT_create_plan_3d — Create a 1,2 or 3D plan for a DFT

```
#include "csdft.h"
CSDftPlan CSDFT_create_plan_1d( CSDftFlags flags, unsigned int size,
unsigned int num_ffts, double scale );
CSDftPlan CSDFT_create_plan_2d( CSDftFlags flags, unsigned int
size_x, unsigned int size_y, unsigned int num_ffts, double scale );
CSDftPlan CSDFT_create_plan_3d( CSDftFlags flags, unsigned int
size_x, unsigned int size_y, unsigned int size_z, unsigned int
num_ffts, double scale );
```

Description

These functions create plans for 1, 2 and 3 dimensional DFTs, which can then be executed using the CSDFT_execute_dft routine.

This library supports:

1D DFT

- Power of two sizes from 128 to 8192
- Single and Double precision
- Complex to Complex (forward and backward)
- Interleaved

2D DFT

- Square and non square Power of two sizes from 128 to 2048
- Single and Double precision
- Real and Complex types:
 - Complex to Complex (forward and backward)
 - Real to Complex (forward)
 - Complex to Real (backward)
- Interleaved and Split Array inputs
- Natural and Optimal order

3D DFT

- 128 cubed size only
- Single and Double precision
- Complex to Complex (forward and backward)
- Interleaved
- Natural order only

The complex data generated from a real to complex forward 2D DFT is a partial representation of the full complex output. This is due to the symmetry of the resulting data. Only the first $(n/2)+1$ columns are represented; in natural mode this means each row has only $(n/2)+1$ samples, with no padding at the end of the row. In optimal mode, only the first

$(n/2)+1$ columns are output. When doing a backward complex to real DFT, this half-complex format is assumed.

It is possible to create multiple plans and relate them to multiple DFT computations.

The ClearSpeed DFT library performs the following forward 1D FFT:

$$Y[k]=\text{Sum}\{j=0,n-1\}(X[j]*\exp(-2*\pi*j*k*\text{sqrt}(-1)/n))$$

The ClearSpeed DFT library performs the following backward 1D DFT:

$$Y[k]=\text{Sum}\{j=0,n-1\}(X[j]*\exp(2*\pi*j*k*\text{sqrt}(-1)/n))$$

All the various configuration parameters related to a DFT are contained in a plan object in the CSDFT API. A descriptor is created by the following API calls for 1d, 2d and 3D versions of CSDFT_create_plan_<1D|2D|3D>.

Returns

This function returns a plan if there are no errors found in the supplied flags or parameters, otherwise it returns NULL. If there is an error, then the global status will be set to indicate an error, which can be retrieved by using `CSDFT_get_status()`.

CSDFT_execute_dft — Synchronous Execution of a DFT

```
#include "csdft.h"
CSDftStatus CSDFT_execute_dft( const CSDftPlan plan,
CSDftMemoryDescriptor source, CSDftMemoryDescriptor destination );
```

Description

This function performs the DFT as described by the plan parameter, using data specified by the source and destination data pointers. This function is a synchronous interface and will only return when the DFT has completed.

DFTs within the source and destination data are expected to be stored contiguously.

Returns

Returns a status code of CSDFT_NO_ERROR if the function is successful, CSDFT_INVALID_PLAN if the plan passed as a parameter is not valid (such as the handle being NULL), CSDFT_INVALID_MEMORY_DESCRIPTOR if the handles for the memory descriptors are invalid or CSDFT_BOARD_ERROR if the library cannot connect to an Advance Card.

CSDFT_create_convolution_plan_1d, CSDFT_create_convolution_plan_2d — Create a 1D or 2D plan for a convolution operation

```
#include "csdft.h"
CSDftPlan CSDFT_create_convolution_plan_1d(CSDftFlags flags, unsigned
int size, unsigned int num_ffts, double scale);
CSDftPlan CSDFT_create_convolution_plan_2d(CSDftFlags flags, unsigned
int size_x, unsigned int size_y, unsigned int num_ffts, double scale);
```

Description

Performing a convolution is similar to any other DFT - a plan should be created, specifying the correct flags and then executed. As usual, flags are ORed together - however one flag of specific note is CSDFT_TRANSFORM_KERNEL. If set, the kernel will undergo a DFT before being convolved with the transformed source. Otherwise, the kernel will be

convolved "as is" with the transformed source. By default, the kernel is assumed to not need to be pre-transformed, and, in this implementation, the kernel is assumed to be the result of a single DFT. Therefore, the same kernel will be reused if *num_ffts* is greater than 1. The kernel must be pretransformed when the convolution is on the board.

To perform a convolution with no apparent scaling, the scale parameter should be $1/n$ for both 1D and 2D, where 'n' is the size of the DFT to the power of the dimension. For example, scale would be $1/256$ when performing a 1D dft with 256 samples and $1/(256*256)$ when performing a 256x256 2D.

The complex data generated from a real to complex forward 2D DFT is a partial representation of the full complex output. This is due to the symmetry of the resulting data. Only the first $(n/2)+1$ columns are represented; in natural mode this means each row has only $(n/2)+1$ samples, with no padding at the end of the row. In optimal mode, only the first $n/2+1$ columns are output. When performing a backward complex to real DFT, this half-complex format is assumed. The kernel MUST be in optimal order when performing convolutions.

Returns

This function returns a plan if there are no errors found in the supplied flags or parameters, otherwise it returns NULL. If there is an error, then the global status will be set to indicate an error, which can be retrieved by using *CSDFT_get_status()*.

CSDFT_execute_convolution — Synchronous Execution of a convolution

```
#include "csdft.h"
CSDftStatus CSDFT_execute_convolution( const CSDftPlan plan,
CSDftMemoryDescriptor kernel, CSDftMemoryDescriptor source,
CSDftMemoryDescriptor destination ) ;
```

Description

This function uses memory descriptors to specify the memory locations of the kernel, source and destination data. A memory descriptor is an opaque data structure which represents data, both in host and board space. More information on this structure is available in the Introduction section.

If the convolution is performed correctly and no errors occur, the solution will be in the memory space referenced by the destination memory descriptors.

Returns

Returns a status code of CSDFT_NO_ERROR if the function is successful, CSDFT_INVALID_PLAN if the plan passed as a parameter is not valid (such as the handle being NULL), CSDFT_INVALID_MEMORY_DESCRIPTOR if the handles for the memory descriptors are invalid or CSDFT_BOARD_ERROR if the library cannot connect to an Advance Board.

CSDFT_free_plan — Free a DFT plan

```
#include "csdft.h"
CSDftStatus CSDFT_free_plan( CSDftPlan plan ) ;
```

Description

This function frees up the previously created plan.

Returns

This function frees all associated resources for a plan and returns a CSDftStatus result of CSDFT_NO_ERROR or CSDFT_INVALID_MEMORY_DESCRIPTOR if *plan* is a null pointer.

CSDFT_get_status — Returns the current status for the CSDFT library

```
#include "csdft.h"
CSDftStatus CSDFT_get_status( void );
```

Description

This function returns the current status code for the CSDFT library. Specifically it returns the status code of the last executed CSDft library function. All Plan and Execute functions will either set the global status code or return their status code as a return value as their prototype dictates.

The following are the valid status codes which will be returned or set by CSDFT library functions.

- CSDFT_NO_ERROR No current error detected.
- CSDFT_INVALID_MEMORY_DESCRIPTOR When set by one of the Execute functions or one of the memory manipulation functions this code indicates that one of the pointer parameters to the function were NULL.
- CSDFT_MEMORY_ALLOCATION_ERROR When set by one of the Memory Descriptor functions that allocate memory this code indicates that a memory allocation error has been encountered either on the host or on the Advance board.
- CSDFT_BOARD_ERROR This code can be set as a result of the CSDFT library not being able to find an Advance board to connect to. If there is definitely a board present in the computer, typing "csreset -v" into your command prompt should identify or fix the problem. Alternatively, this error is set due to an internal, unrecoverable error on the board or in the driver.
- CSDFT_INVALID_BOARD This code is returned when an operation is adjudged to be sent to a board that does not exist.
- CSDFT_INVALID_PROCESSOR This code is returned when an operation is adjudged to be sent to a processor that does not exist on a board that does exist.
- CSDFT_INVALID_PLAN When set by one of the planner functions this code indicates that one or more of the flags of a requested DFT was invalid. When returned by one of the execute functions it indicates that the plan was somehow invalid or not defined (NULL).
- CSDFT_INVALID_SIZE Set as a result of an unsupported size being specified as part of the plan. This is most likely the result of a size being set that is not currently supported by the CSDFT library (currently a power of 2).
- CSDFT_INVALID_FLAG Set as a result of incompatible options being chosen when creating a plan. An example of this is choosing different source and destination formats.
- CSDFT_INVALID_PARAMETER Set as a result of incompatible parameters being specified when calling a function.
- CSDFT_INVALID_USER_FUNCTION Set as a result of an invalid user function handle being specified when calling a function. An example of this is passing a null handle to *CSDFT_execute_user_function()*.

This function can be used to test for reasons behind unexpected behaviour and when used in conjunction with the `CSDFT_return_error_message()` can provide textual messages for the user at runtime.

Returns

This function returns a status of `CSDFT_NO_ERROR` if has been no error in the last performed function, otherwise it returns another of the listed status codes.

CSDFT_board_to_memory_descriptor — Returns a board based memory descriptor encapsulating user provided source data.

```
#include "csdft.h"
CSDftMemoryDescriptor CSDFT_board_to_memory_descriptor( void
*source_data, unsigned int size_in_bytes) ;
```

Description

This function returns a `CSDftMemoryDescriptor` where the data buffer was pre-allocated on the board. The parameters are: a pointer to the array of data already allocated on the host and the size in bytes of the buffer. If the function cannot for some reason create a Memory Descriptor it returns a NULL handle and sets the global status code to be `CSDFT_MEMORY_ERROR`.

Returns

This function returns a handle to a `CSDftMemoryDescriptor` on success and a NULL handle on failure as well as setting the global status code to `CSDFT_NO_ERROR` or `CSDFT_MEMORY_ERROR` as required.

CSDFT_memory_descriptor_to_board — Given a board based memory descriptor returns a pointer to a buffer in board memory

```
#include "csdft.h"
void * CSDFT_memory_descriptor_to_board( CSDftMemoryDescriptor handle
) ;
```

Description

This function returns a `void *` pointer to the data buffer that is encapsulated by the `CSDftMemoryDescriptor`. The parameter is a handle to a `CSDftMemoryDescriptor`. If the function cannot for some reason return a valid pointer to system allocated data then it returns a NULL pointer and sets the global status code to be `CSDFT_MEMORY_ERROR`.

Returns

This function returns a pointer to system allocated data on success and a NULL pointer on failure as well as setting the global status code to `CSDFT_NO_ERROR` or `CSDFT_MEMORY_ERROR` as required.

CSDFT_free — Deallocates a memory descriptor created by the CSDFT library

```
#include "csdft.h"
CSDftStatus CSDFT_free( CSDftMemoryDescriptor handle ) ;
```

Description

This function deallocates a memory descriptor created by one of a number of routines provided by the CSDFT library. The function is passed a handle to a CSDftMemoryDescriptor and returns a CSDftStatus code indicating success or failure. In the case of the Memory Descriptor object being deallocated without any problems this will be CSDFT_NO_ERROR and in the case where there was a problem, such as trying to deallocate an already freed object, the status code will be CSDFT_MEMORY_ERROR.

Returns

This function returns a status of CSDFT_NO_ERROR or CSDFT_MEMORY_ERROR.

Index

C

CSDFT_bitreverse_1D_c		Host	24
CSDFT_bitreverse_1D_z		Host	24
CSDFT_board_to_memory_descriptor		Board	33
CSDFT_create_convolution_plan_1d			30
CSDFT_create_convolution_plan_2d		Board	30
		Host	18
CSDFT_create_plan_1d		Board	29
		Host	14
CSDFT_create_plan_2d		Board	29
		Host	14
CSDFT_create_plan_3d		Board	29
		Host	14
CSDFT_create_user_function		Host	16
CSDFT_execute_convolution		Board	31
		Host	18
CSDFT_execute_dft		Board	30
		Host	15
CSDFT_execute_user_function		Host	17
CSDFT_free		Board	33
		Host	23
CSDFT_free_plan		Board	31
		Host	15
CSDFT_free_user_func		Host	18
CSDFT_get_csapi_handle		Host	28
CSDFT_get_csapi_handle_board		Host	28
CSDFT_get_null_descriptor		Host	21
CSDFT_get_status		Board	32
		Host	19
CSDFT_get_symbol_value		Host	28
CSDFT_ilog2		Host	27
CSDFT_malloc_host		Host	21
CSDFT_memory_descriptor_copy		Host	23
CSDFT_memory_descriptor_to_board		Board	33
CSDFT_memory_descriptor_to_system		Host	22
CSDFT_natural_to_optimal_2D_c		Host	26
CSDFT_natural_to_optimal_2D_z		Host	27
CSDFT_optimal_to_natural_2D_c		Host	25
CSDFT_optimal_to_natural_2D_z		Host	26
CSDFT_processor_to_memory_descriptor		Board	23
CSDFT_return_error_message		Host	20
CSDFT_system_split_array_to_memory_descriptor		Host	22
CSDFT_system_to_memory_descriptor		Host	22

ClearSpeed Technology Ltd
130 Aztec West
Park Avenue
Bristol BS32 4UB
United Kingdom

Tel: +44 (0)1454 629 623
Fax: +44 (0)1454 629 624

Email: info@clearspeed.com

Web: <http://www.clearspeed.com>

Support: <http://support.clearspeed.com>

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.

© Copyright ClearSpeed Technology Ltd 2010. All rights reserved.

Advance is a registered trademark of ClearSpeed Technology Ltd

ClearSpeed, ClearConnect, Advance and the ClearSpeed logo are trade marks or registered trade marks of ClearSpeed Technology Ltd. All other brands and names are the property of their respective owners.