DataTransferKit Documentation

Release 3.0.0

Stuart Slattery, Damien Lebrun-Grandie

July 13, 2018

Contents

1 Getting started with DTK **3**
- 1.1 Overview . . . 3
- 1.2 DataTransferKit Development Team . . . 3
- 1.3 DataTransferKit Packages . . . 3
- 1.4 Questions, Bug Reporting, and Issue Tracking . . . 4
- 1.5 Publications . . . 4

2 Installation **5**
- 2.1 Install third-party libraries . . . 5
- 2.2 DTKData repository . . . 5
- 2.3 Building DTK . . . 6
- 2.4 Build this documentation . . . 6
- 2.5 Generate Doxygen documentation . . . 6

3 DataTransferKit API **9**
- 3.1 C++ API . . . 9
- 3.2 C API . . . 12

4 Developer Tools **21**
- 4.1 Run DTK development environment in a Docker container . . . 21
- 4.2 Make the container GPU-aware . . . 21
- 4.3 Code completion for Vim . . . 22

5 Coding Style Guidelines **23**
- 5.1 ClangFormat . . . 23
- 5.2 Style Guide . . . 23

6 Indices and tables **25**

Warning: This is the documentation for the development version of DTK. There may be significant differences from the latest stable release. Please follow this link if you are looking for DTK 2.0

Contents:

Warning: This is the documentation for the development version of DTK. There may be significant differences from the latest stable release. Please follow this link if you are looking for DTK 2.0

CHAPTER 1

Getting started with DTK

Overview

DataTransferKit is an open-source software library of parallel solution transfer services for multiphysics simulations. DTK uses a general operator design to provide scalable algorithms for solution transfer between shared volumes and surfaces.

DTK was originally developed at the University of Wisconsin - Madison as part of the Computational Nuclear Engineering Group (CNERG) and is now actively developed at the Oak Ridge National Laboratory as part of the Computational Engineering and Energy Sciences (CEES) group.

DTK is supported and used by the following projects and programs:

- Oak Ridge National Laboratory (ORNL) Laboratory Directed Research and Development (LDRD)
- Consortium for Advanced Simulation of Light Water Reactors (CASL)
- Nuclear Energy Advanced Modeling and Simulation (NEAMS)
- National Highway Traffic Safety Administration (NHTSA)

DataTransferKit Development Team

DTK is developed and maintained by:

- Stuart Slattery
- Damien Lebrun-Grandie
- Bruno Turcksin
- Andrey Prokopenko
- Roger Pawlowski
- Alex McCaskey

DataTransferKit Packages

DTK has the following packages:

Utils General utilities for software development including exception handling, and other functional programming tools

Interface Interfaces with user applications and operators

Search Search algorithms leveraged by all operators

Meshfree Point cloud based operators (e.g., nearest neighbor, moving least squares, spline interpolation)

Discretization Mesh based operators (e.g., interpolation, L2 projection)

Questions, Bug Reporting, and Issue Tracking

Questions, bug reporting and issue tracking are provided by GitHub. Please report all bugs by creating a new issue. You can ask questions by creating a new issue with the question tag.

Publications

Publications to date related to DataTransferKit:

- S. Slattery, *"Mesh-Free Data Transfer Algorithms for Partitioned Multiphysics Problems: Conservation, Accuracy, and Parallelism"*, Journal of Computational Physics, vol. 307, pp. 164-188, 2016.
- S. Slattery, S. Hamilton, T. Evans, *"A Modified Moving Least Square Algorithm for Solution Transfer on a Spacer Grid Surface"*, ANS MC2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method, Nashville, Tennessee · April 19–23, 2015, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2015).
- R. Schmidt, K. Belcourt, R. Hooper, R. Pawlowski, K. Clarno, S. Simunovic, S. Slattery, J. Turner, S. Palmtag, *"An Approach for Coupled-Code Multiphysics Core Simulations from a Common *Input"*, Annals of Nuclear Energy, Volume 84, pp. 140-152, 2014.
- S. Slattery, P.P.H. Wilson, R. Pawlowski, *"The Data Transfer Kit: A Geometric Rendezvous-Based Tool for Multiphysics Data Transfer"*, International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013), American Nuclear Society, Sun Valley, ID, May 5-9, 2013.

Warning: This is the documentation for the development version of DTK. There may be significant differences from the latest stable release. Please follow this link if you are looking for DTK 2.0

CHAPTER 2

Installation

This section provide guidelines for installing DataTransferKit and its TPLs.

Install third-party libraries

The following third party libraries (TPLs) are used by DTK:

Packages	Dependency	Version
Boost	Optional	1.61.0
BLAS/LAPACK	Required	N/A
MPI	Optional	N/A
Trilinos	Required	12.X

Note: DTK is built as an external package of Trilinos. Thus, DTK needs the source of the Trilinos library rather than an installed version.

The dependencies of DataTransferKit may be built using Spack package manager. You need to install the following packages:

```
$ spack install openblas
$ spack install boost
$ spack install mpi
```

Once installed, the module files for the packages must be loaded into the environment by doing

```
$ spack load openblas
$ spack load boost
$ spack load openmpi
```

DTKData repository

The DTKData repository contains mesh files used in DTK examples. To build the examples and include the mesh files include the git submodule with your cloned repository:

```
$ git submodule init
$ git submodule update
```

Another way to achieve this is to pass the `--recursive` option to the `git clone` command which will automatically initialize and update DTKData in the DataTransferKit repository.

Building DTK

DTK is configured and built using TriBITS. DTK builds within Trilinos effectively as an extension package. First, link DTK into the Trilinos main directory:

```
$ cd $TRILINOS_DIR
$ ln -s $DTK_DIR
```

Create a `do-configure` script such as:

```
EXTRA_ARGS=$@

cmake \
    -D CMAKE_BUILD_TYPE=Release \
    -D TPL_ENABLE_MPI=ON \
    -D TPL_ENABLE_BLAS=ON \
    -D TPL_ENABLE_LAPACK=ON \
    -D TPL_ENABLE_Boost=ON \
    -D Trilinos_ENABLE_EXPLICIT_INSTANTIATION=ON \
    -D Tpetra_INST_INT_UNSIGNED_LONG=ON \
    -D Tpetra_INST_INT_LONG_LONG=OFF \
    -D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES=OFF \
    -D Trilinos_EXTRA_REPOSITORIES="DataTransferKit" \
    -D Trilinos_ENABLE_DataTransferKit=ON \
    -D DataTransferKit_ENABLE_DBC=ON \
    -D DataTransferKit_ENABLE_TESTS=ON \
    -D DataTransferKit_ENABLE_EXAMPLES=ON \
    $EXTRA_ARGS \
    $TRILINOS_DIR
```

and run it from your build directory:

```
$ mkdir build && cd build
$ ../do-configure
```

More install scripts can be found in `scripts/`.

Note: The above `do-configure` script may get outdated. You can always refer to `scripts/docker_cmake` which is used in the Jenkins CI builds and therefore is required to be always up-to-date.

Build this documentation

Building documentation requires sphinx. (Re)configure with `-D DataTransferKit_ENABLE_ReadTheDocs=ON` and run:

```
$ make docs
```

Open the `index.html` in the directory `DataTransferKit/docs/html`.

Generate Doxygen documentation

Configure with `-D DataTransferKit_ENABLE_Doxygen=ON` and run:

```
$ make doxygen
```

Checkout `DataTransferKit/docs/doxygen/html/index.html`.

Warning: This is the documentation for the development version of DTK. There may be significant differences from the latest stable release. Please follow this link if you are looking for DTK 2.0

CHAPTER 3

DataTransferKit API

C++ API

User Function Registry

template <**class** Scalar>
class `DataTransferKit::UserFunctionRegistry`

Registry for user functions.

The registry is the mechanism by which user functions and the data to be called with those user functions are registered with DTK.

Template Parameters

- `Scalar`: Scalar type of the fields this user function implementation will provide.

Set Geometry

void **setNodeListSizeFunction**(NodeListSizeFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Node list size function.

void **setNodeListDataFunction**(NodeListDataFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Node list data function.

void **setBoundingVolumeListSizeFunction**(BoundingVolumeListSizeFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Bounding volume list size function.

void **setBoundingVolumeListDataFunction**(BoundingVolumeListDataFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Bounding volume list data function.

void **setPolyhedronListSizeFunction**(PolyhedronListSizeFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Polyhedron list size function.

void **setPolyhedronListDataFunction**(PolyhedronListDataFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Polyhedron list data function.

void **setCellListSizeFunction**(CellListSizeFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Cell list size function.

void **setCellListDataFunction**(CellListDataFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Cell list data function.

void **setBoundarySizeFunction**(BoundarySizeFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Boundary data function.

void **setBoundaryDataFunction**(BoundaryDataFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Boundary data function.

void **setAdjacencyListSizeFunction**(AdjacencyListSizeFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Adjacency list size function.

void **setAdjacencyListDataFunction**(AdjacencyListDataFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Adjacency list data function.

Set Degree-of-freedom Maps

void **setDOFMapSizeFunction**(DOFMapSizeFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Single dofs per object dof map size.

void **setDOFMapDataFunction**(DOFMapDataFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Single dofs per object dof map data.

void **setMixedTopologyDOFMapSizeFunction**(MixedTopologyDOFMapSizeFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Multiple dofs per object dof map size.

void **setMixedTopologyDOFMapDataFunction**(MixedTopologyDOFMapDataFunction &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Multiple dofs per object dof map data.

Set Fields

void **setFieldSizeFunction**(FieldSizeFunction<Scalar> &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Field size.

void **setPullFieldDataFunction**(PullFieldDataFunction<Scalar> &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Pull field.

void **setPushFieldDataFunction**(PushFieldDataFunction<Scalar> &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Push field.

void **setEvaluateFieldFunction**(EvaluateFieldFunction<Scalar> &&*func*, std::shared_ptr<void> *user_data* = nullptr)

Evaluate field.

Public Types

template<>
using **`UserImpl`** = std::pair<CallableObject, std::shared_ptr<void>>
User implementation.

User Application

template <**class** Scalar, **class** ParallelModel>
class `DataTransferKit::UserApplication`
High-level interface to user applications.

The user application provides a high-level interface to compose DTK input data structures and push and pull field data to and from the application through sequences of user function calls.

Template Parameters

- `Scalar`: Scalar type of the fields this user application will provide.
- `ParallelModel`: The DTK parallel model indicating the on-node parallelism of the user application. Indicates where data will be allocated.

Type Aliases

template<>
using **`ExecutionSpace`** = **typename** ParallelTraits::ExecutionSpace

Public Functions

`UserApplication`(**const** std::shared_ptr<UserFunctionRegistry<Scalar>> &*user_functions*)
Constructor.

auto **`getNodeList`**()
Get a node list from the application.

auto **`getBoundingVolumeList`**()
Get a bounding volume list from the application.

auto **`getPolyhedronList`**()
Get a polyhedron list from the application.

auto **`getCellList`**()
Get a cell list from the application.

template <**class** ListType>
void **`getBoundary`**(ListType &*list*)
Get a boundary from the application and put it in the given list.

template <**class** ListType>
void **`getAdjacencyList`**(ListType &*list*)
Get adjacencies from the application and put it in the given list.

auto **`getDOFMap`**(std::string &*discretization_type*)
Get a dof id map from the application.

auto **`getField`**(**const** std::string &*field_name*)
Get a field with a given name from the application.

void **pullField**(**const** std::string &*field_name*, Field<Scalar, Kokkos::LayoutLeft, ExecutionSpace> *field*)
Pull a field with a given name to the application.

void **pushField**(**const** std::string &*field_name*, **const** Field<Scalar, Kokkos::LayoutLeft, ExecutionSpace> *field*)
Push a field with a given name to the application.

void **evaluateField**(**const** std::string &*field_name*, **const** EvaluationSet<Kokkos::LayoutLeft, ExecutionSpace> *eval_set*, Field<Scalar, Kokkos::LayoutLeft, ExecutionSpace> *field*)
Ask the application to evaluate a field with a given name.

C API

C interface header.

Typedefs

typedef struct _DTK_UserApplicationHandle ***DTK_UserApplicationHandle**
DTK user application handle.

Must be created using *DTK_create()* to be a valid handle.

The handle essentially hides C++ implementation details from the user.

typedef void(* DTK_NodeListSizeFunction) (void *user_data, unsigned *space_dim, size_t *loca
Prototype function to get the size parameters for building a node list.

Register with a user application using *DTK_set_function()* by passing DTK_NODE_LIST_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `space_dim`: Spatial dimension.
- `local_num_nodes`: Number of nodes DTK will allocate memory for.

typedef void(* DTK_NodeListDataFunction) (void *user_data, Coordinate *coordinates)
Prototype function to get the data for a node list.

Register with a user application using *DTK_set_function()* by passing DTK_NODE_LIST_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `coordinates`: Node coordinates.

typedef void(* DTK_BoundingVolumeListSizeFunction) (void *user_data, unsigned *space_dim, si
Prototype function to get the size parameters for building a bounding volume list.

Register with a user application using *DTK_set_function()* by passing DTK_BOUNDING_VOLUME_LIST_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `space_dim`: Spatial dimension.
- `local_num_volumes`: Number of volumes DTK will allocate memory for.

typedef void(* DTK_BoundingVolumeListDataFunction) (void *user_data, Coordinate *bounding_vo
Prototype function to get the data for a bounding volume list.

Register with a user application using *DTK_set_function()* by passing DTK_BOUNDING_VOLUME_LIST_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `bounding_volumes`: Bounding volumes.

typedef void(* DTK_PolyhedronListSizeFunction) (void *user_data, unsigned *space_dim, size_t
Prototype function to get the size parameters for building a polyhedron list.

Register with a user application using *DTK_set_function()* by passing DTK_POLYHEDRON_LIST_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `space_dim`: Spatial dimension.
- `local_num_nodes`: Number of nodes DTK will allocate memory for.
- `local_num_faces`: Number of faces DTK will allocate memory for.
- `total_face_nodes`: Total number of nodes for all faces.
- `local_num_cells`: Number of cells DTK will allocate memory for.
- `total_cell_faces`: Total number of faces for all cells.

typedef void(* DTK_PolyhedronListDataFunction) (void *user_data, Coordinate *coordinates, Lo
Prototype function to get the data for a polyhedron list.

Register with a user application using *DTK_set_function()* by passing DTK_POLYHEDRON_LIST_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `coordinates`: Node coordinates.
- `faces`: Connectivity list of faces.
- `nodes_per_face`: Number of nodes per face.
- `cells`: Connectivity list of polyhedrons.
- `faces_per_cell`: Number of faces per cell.
- `face_orientation`: Orientation of the faces.

typedef void(* DTK_CellListSizeFunction) (void *user_data, unsigned *space_dim, size_t *loca
Prototype function to get the size parameters for building a cell list.

Register with a user application using *DTK_set_function()* by passing DTK_CELL_LIST_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `space_dim`: Spatial dimension.
- `local_num_nodes`: Number of nodes DTK will allocate memory for.
- `local_num_cells`: Number of cells DTK will allocate memory for.
- `total_cell_nodes`: Total number of nodes for all cells.

typedef void(* DTK_CellListDataFunction) (void *user_data, Coordinate *coordinates, LocalOrd

Prototype function to get the data for a mixed topology cell list.

Register with a user application using *DTK_set_function()* by passing DTK_CELL_LIST_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `coordinates`: Node coordinates.
- `cells`: List of cells.
- `cell_topologies`: Topologies of the cells.

typedef void(* DTK_BoundarySizeFunction) (void *user_data, size_t *local_num_faces)

Prototype function to get the size parameters for a boundary.

Register with a user application using *DTK_set_function()* by passing DTK_BOUNDARY_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `local_num_faces`: Number of faces owned by this process.

typedef void(* DTK_BoundaryDataFunction) (void *user_data, LocalOrdinal *boundary_cells, uns

Prototype function to get the data for a boundary.

Register with a user application using *DTK_set_function()* by passing DTK_BOUNDARY_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `boundary_cells`: Indices of the cells on the boundary.
- `cell_faces_on_boundary`: Indices of the faces within a given cell that is on the boundary.

typedef void(* DTK_AdjacencyListSizeFunction) (void *user_data, size_t *total_adjacencies)

Prototype function to get the size parameters for building an adjacency list.

Register with a user application using *DTK_set_function()* by passing DTK_ADJACENCY_LIST_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `total_adjacencies`: Total number of adjacencies in the list.

typedef void(* DTK_AdjacencyListDataFunction) (void *user_data, GlobalOrdinal *global_cell_i

Prototype function to get the data for an adjacency list.

Register with a user application using *DTK_set_function()* by passing DTK_ADJACENCY_LIST_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `global_cell_ids`: The global ids of the local cells in the list.
- `adjacent_global_cell_ids`: The global ids of the cells adjacent to the local cells in the list. These may live on another process.
- `adjacencies_per_cell`: The number of adjacencies each local cell has. These serve as offsets into the adjacent_global_cell_ids array.

typedef void(* DTK_DOFMapSizeFunction) (void *user_data, size_t *local_num_dofs, size_t *loc

Prototype function to get the size parameters for a degree-of-freedom id map with a single number of dofs per object.

Register with a user application using *DTK_set_function()* by passing DTK_DOF_MAP_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `local_num_dofs`: Number of degrees of freedom owned by this process.
- `local_num_objects`: Number of objects on this process.
- `dofs_per_objects`: Degrees of freedom per object.

typedef void(* DTK_DOFMapDataFunction) (void *user_data, GlobalOrdinal *global_dof_ids, Loca

Prototype function to get the size data for a degree-of-freedom id map with a single number of dofs per object.

Register with a user application using *DTK_set_function()* by passing DTK_DOF_MAP_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `global_dof_ids`: Globally unique ids for DOFs on this process.
- `object_dof_ids`: For every object of the given type in the object list give the local dof ids for that object. The local dof ids correspond to the index of the entry in the global dof id view.
- `discretization_type`: Type of discretization.

typedef void(* DTK_MixedTopologyDofMapSizeFunction) (void *user_data, size_t *local_num_dofs

Prototype function to get the size parameters for a degree-of-freedom id map with each object having a potentially different number of dofs (e.g. mixed topology cell lists or polyhedron lists).

Register with a user application using *DTK_set_function()* by passing DTK_MIXED_TOPOLOGY_DOF_MAP_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `local_num_dofs`: Number of degrees of freedom owned by this process.

- `local_num_objects`: Number of objects on this process.
- `total_dofs_per_objects`: Total degrees of freedom per objects.

typedef void(* DTK_MixedTopologyDofMapDataFunction) (void *user_data, GlobalOrdinal *global_

Prototype function to get the data for a multiple object degree-of-freedom id map (e.g. mixed topology cell lists or polyhedron lists).

Register with a user application using *DTK_set_function()* by passing DTK_MIXED_TOPOLOGY_DOF_MAP_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `global_dof_ids`: Globally unique ids for DOFs on this process.
- `object_dof_ids`: Local object IDs.
- `dofs_per_object`: Degrees of freedom per object.
- `discretization_type`: Type of discretization.

typedef void(* DTK_FieldSizeFunction) (void *user_data, const char *field_name, unsigned *fi

Prototype function to get the size parameters for a field.

Register with a user application using *DTK_set_function()* by passing DTK_FIELD_SIZE_FUNCTION as the `type` argument.

Field must be of size local_num_dofs in the associated dof_id_map.

Parameters

- `user_data`: Custom user data.
- `field_name`: Name of the field.
- `field_dimension`: Dimension of the field (i.e. 1 for the pressure, or 3 for the velocity in 3-D)
- `local_num_dofs`: Number of degrees of freedom owned by this process.

typedef void(* DTK_PullFieldDataFunction) (void *user_data, const char *field_name, double *

Prototype function to pull data from the application into a field.

Register with a user application using *DTK_set_function()* by passing DTK_PULL_FIELD_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Custom user data.
- `field_name`: Name of the field to pull.
- `field_dofs`: Degrees of freedom for that field.

typedef void(* DTK_PushFieldDataFunction) (void *user_data, const char *field_name, const do

Prototype function to push data from a field into the application.

Register with a user application using *DTK_set_function()* by passing DTK_PUSH_FIELD_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Custom user data.
- `field_name`: Name of the field to push.

- `field_dofs`: Degrees of freedom for that field.

typedef void(* DTK_EvaluateFieldFunction) (void *user_data, const char *field_name, const Co

Prototype function to evaluate a field at a given set of points in a given set of objects.

Register with a user application using *DTK_set_function()* by passing DTK_EVALUATE_FIELD_FUNCTION as the `type` argument.

Parameters

- `user_data`: Custom user data.
- `field_name`: Name of the field to evaluate.
- `evaluate_points`: Coordinates of the points at which to evaluate the field.
- `objects_ids`: ID of the cell/face with repect of which the coordinates are expressed.
- `values`: Field values.

Enums

enum DTK_ExecutionSpace

Execution space (where functions execute)

Values:

DTK_SERIAL

DTK_OPENMP

DTK_CUDA

enum DTK_FunctionType

Passed as the `type` argument to *DTK_set_function()* in order to indicate what callback function is being registered with the user application.

Note Callback functions are passed as pointers to functions that take no arguments and return nothing (`void(*)()`) so the value of the DTK_FunctionType enum is necessary to indicate what is being registered with the user application and how to cast the function pointer back to the appropriate signature.

Values:

DTK_NODE_LIST_SIZE_FUNCTION
See *DTK_NodeListSizeFunction()*

DTK_NODE_LIST_DATA_FUNCTION
See *DTK_NodeListDataFunction()*

DTK_BOUNDING_VOLUME_LIST_SIZE_FUNCTION
See *DTK_BoundingVolumeListSizeFunction()*

DTK_BOUNDING_VOLUME_LIST_DATA_FUNCTION
See *DTK_BoundingVolumeListDataFunction()*

DTK_POLYHEDRON_LIST_SIZE_FUNCTION
See *DTK_PolyhedronListSizeFunction()*

DTK_POLYHEDRON_LIST_DATA_FUNCTION
See *DTK_PolyhedronListDataFunction()*

DTK_CELL_LIST_SIZE_FUNCTION
See *DTK_CellListSizeFunction()*

DTK_CELL_LIST_DATA_FUNCTION
See *DTK_CellListDataFunction()*

DTK_BOUNDARY_SIZE_FUNCTION
See *DTK_BoundarySizeFunction()*

DTK_BOUNDARY_DATA_FUNCTION
See *DTK_BoundaryDataFunction()*

DTK_ADJACENCY_LIST_SIZE_FUNCTION
See *DTK_AdjacencyListSizeFunction()*

DTK_ADJACENCY_LIST_DATA_FUNCTION
See *DTK_AdjacencyListDataFunction()*

DTK_DOF_MAP_SIZE_FUNCTION
See *DTK_DOFMapSizeFunction()*

DTK_DOF_MAP_DATA_FUNCTION
See *DTK_DOFMapDataFunction()*

DTK_MIXED_TOPOLOGY_DOF_MAP_SIZE_FUNCTION
See *DTK_MixedTopologyDofMapSizeFunction()*

DTK_MIXED_TOPOLOGY_DOF_MAP_DATA_FUNCTION
See *DTK_MixedTopologyDofMapDataFunction()*

DTK_FIELD_SIZE_FUNCTION
See *DTK_FieldSizeFunction()*

DTK_PULL_FIELD_DATA_FUNCTION
See *DTK_PullFieldDataFunction()*

DTK_PUSH_FIELD_DATA_FUNCTION
See *DTK_PushFieldDataFunction()*

DTK_EVALUATE_FIELD_FUNCTION
See *DTK_EvaluateFieldFunction()*

Functions

const char ***DTK_version**()
Get the current version of DTK.

Return Returns a string containing the version number for DTK.

const char ***DTK_git_commit_hash**()
Get the current repository hash.

Note If the source code is not under revision control (e.g. downloaded as a tarball), this functions returns an error string indicating that it is not a GIT repository.

Return Returns a string containing the revision number.

DTK_UserApplicationHandle **DTK_create**(*DTK_ExecutionSpace space*)
Create a DTK handle.

Return DTK_create returns a handle for the user application.

Parameters

- `space`: Execution space for the callback functions that are to be registered using *DTK_set_function()*.

bool **DTK_is_valid**(*DTK_UserApplicationHandle handle*)
Indicates whether a DTK handle is valid.

A handle is valid if it was created by *DTK_create()* and has not yet been deleted by *DTK_destroy()*.

Return true if the given user application handle is valid; false otherwise.

Parameters

- `handle`: The DTK user application handle to check.

void **DTK_destroy**(*DTK_UserApplicationHandle handle*)
Destroy a DTK handle.

Parameters

- `handle`: User application handle.

void **DTK_initialize**()
Initializes the DTK execution environment.

This initializes Kokkos if it has not already been initialized.

void **DTK_initialize_cmd**(int **argc*, char ****argv*)
Initialize DTK.

This initializes Kokkos if it has not already been initialized.

This version of initialize() effectively calls `Kokkos::initialize( *argc, *argv )`. Pointers to `argc` and `argv` arguments are passed in order to match MPI_Init's interface. This function name was suffixed with _cmd because, unlike C++, C does not allow to overload functions.

Parameters

- `argc`: Pointer to the number of argument.
- `argv`: Pointer to the argument vector.

bool **DTK_is_initialized**()
Indicates whether DTK has been initialized.

This function may be used to determine whether DTK has been initialized.

void **DTK_finalize**()
Finalizes DTK.

This function terminates the DTK execution environment. If DTK initialized Kokkos, finalize Kokkos. However, if Kokkos was initialized before DTK, then this function does NOT finalize Kokkos.

const char ***DTK_error**(int *err*)
Get DTK error message.

All DTK functions set `errno` error code upon completion. If DTK function fails, the code is nonzero. This function provides a way to get the the error message associated with the error code. If the error code is unknown, DTK returns a string stating that.

Return Returns corresponding error string. If the error code is 0 (success), return empty string.

Parameters

- `err`: Error number (typically, errno)

void **`DTK_set_function`**(*DTK_UserApplicationHandle handle*, *DTK_FunctionType type*, void (**f*))
void **user_data* Register a function as a callback.

This registers a custom function as a callback for DTK to communicate with the user application.

Parameters

- `handle`: User application handle.
- `type`: Type of callback function.
- `f`: Pointer to user defined callback function.
- `user_data`: Pointer to the user data that will be passed to the callback function when executing it.

Warning: This is the documentation for the development version of DTK. There may be significant differences from the latest stable release. Please follow this link if you are looking for DTK 2.0

CHAPTER 4

Developer Tools

Run DTK development environment in a Docker container

To start a container from the DTK pre-built Docker image that is used in the automated build on Jenkins, run:

```
[host]$ cd docker
[host]$ echo COMPOSE_PROJECT_NAME=$USER > .env # [optional] specify a project name
[host]$ docker-compose pull # pull the most up-to-date version of the DTK base image
[host]$ docker-compose up -d # start the container
```

This will mount the local DTK source directory into the container at `${TRILINOS_DIR}/DataTransferKit`. The environment variable `TRILINOS_DIR` is already defined and contains the path to a release version of Trilinos that has been downloaded into the DTK base image. We recommend you use a `.env` file to specify an alternate project name (the default being the directory name, i.e. `docker`). This will let you run multiple isolated environments on a single host. Here the service name will be prefixed by your username which will prevent interferences with other developers on the same system.

Then to launch an interactive Bash session inside that container, do:

```
[host]$ docker-compose exec dtk_dev bash
```

Configure, build, and test as you would usually do:

```
[container]$ cd $TRILINOS_DIR/DataTransferKit
[container]$ mkdir build && cd build
[container]$ ../scripts/docker_cmake
[container]$ make -j<N>
[container]$ ctest -j<N>
```

Do not forget to cleanup after yourself:

```
[container]$ exit
[host]$ docker-compose down # stop and remove the container
```

Make the container GPU-aware

Follow these instructions to launch containers leveraging the NVIDIA GPUs on the host machine:

```
[host]$ cd docker
[host]$ nvidia/setup_nvidia_docker_compose.py # extend the regular Compose file to leverage GPUs
```

```
[host]$ docker-compose build # add the CUDA development tools to the DTK base image
[host]$ docker-compose up -d # as previously described
```

Do not forget to set the environment for CUDA before you configure:

```
[container]$ # assuming you are in $TRILINOS_DIR/DataTransferKit/build
[container]$ source ../scripts/set_kokkos_env.sh # set environment for CUDA
[container]$ ../scripts/docker_cuda_cmake # configure
[container]$ # now you may build and test
```

Note: If you haven't run *nvidia-docker* before, you may get the following error when attempting to create and start the container

```
ERROR: Volume nvidia_driver_396.26 declared as external, but could not be found. Please create the vo
```

Then run the command below and try again.

```
nvidia-docker run --rm nvidia/cuda nvidia-smi
```

Code completion for Vim

Configure with `-D DataTransferKit_ENABLE_YouCompleteMe=ON` to generate a `.ycm_extra_conf.py` file at the root of your DTK source directory tree for use with YouCompleteMe.

Warning: This is the documentation for the development version of DTK. There may be significant differences from the latest stable release. Please follow this link if you are looking for DTK 2.0

CHAPTER 5

Coding Style Guidelines

DataTransferKit developers follow a set of style guidelines and use the clang format tool to create a consistent appearance to all source code committed to the repository.

ClangFormat

ClangFormat (version 6.0) is used to check the C++ code formatting style in DTK. A pull request that does not comply will be rejected. Configure with `-D DataTransferKit_ENABLE_ClangFormat=ON` and do `make format-cpp` to apply the formatting style before your commit. Alternatively, run `ctest -V -R check_format_cpp` display the diff without applying the changes.

Style Guide

The following conventions are used in the code.

Names of classes, structs, and enumerations are camel case and capitalized:

```
class ExampleClassName {};
```

Function names are camel case and not capitalized:

```
void exampleFunctionName() const;
```

Variable names are lower case and have underscores to separate words:

```
double example_double_var;
std::vector<int> example_vec_var;
```

If a variable is class data prefixed with a _:

```
class ExampleClass
{
  public:
    int _a_public_var;
  private:
    double _class_double_var;
    std::vector<int> _example_vec_var;
  protected:
    std::string _a_protected_string;
};
```

Previously, the convention for class data was to prefix the variable name with `d_` so this will be seen throughout the code. We will be transitioning to the _ prefix convention in future work and slowly transition existing code.

When a function as both input and output arguments, the inputs should come first:

```
void myFunction(int const input_1, int const input_2, int &output)
```

The `clang-format` tool described above enforces spacing, line breaks, and other general file formatting requirements. Header files are suffixed with `.hpp` and non-templated implementation files are suffixed with `.cpp`. Header guards are needed for all header files following the convention of `DTK_CLASSNAME_HPP`. For example:

```
#ifndef DTK_EXAMPLECLASS_HPP
#define DTK_EXAMPLECLASS_HPP

class ExampleClass
{
    // Class definition...
};

#endif
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

D

DataTransferKit::UserApplication (C++ class), 11
DataTransferKit::UserApplication::evaluateField (C++ function), 12
DataTransferKit::UserApplication::getAdjacencyList (C++ function), 11
DataTransferKit::UserApplication::getBoundary (C++ function), 11
DataTransferKit::UserApplication::getBoundingVolumeList (C++ function), 11
DataTransferKit::UserApplication::getCellList (C++ function), 11
DataTransferKit::UserApplication::getDOFMap (C++ function), 11
DataTransferKit::UserApplication::getField (C++ function), 11
DataTransferKit::UserApplication::getNodeList (C++ function), 11
DataTransferKit::UserApplication::getPolyhedronList (C++ function), 11
DataTransferKit::UserApplication::pullField (C++ function), 11
DataTransferKit::UserApplication::pushField (C++ function), 12
DataTransferKit::UserApplication::UserApplication (C++ function), 11
DataTransferKit::UserApplication<Scalar, ParallelModel>::ExecutionSpace (C++ type), 11
DataTransferKit::UserFunctionRegistry (C++ class), 9
DataTransferKit::UserFunctionRegistry::setAdjacencyListDataFunction (C++ function), 10
DataTransferKit::UserFunctionRegistry::setAdjacencyListSizeFunction (C++ function), 10
DataTransferKit::UserFunctionRegistry::setBoundaryDataFunction (C++ function), 10
DataTransferKit::UserFunctionRegistry::setBoundarySizeFunction (C++ function), 10
DataTransferKit::UserFunctionRegistry::setBoundingVolumeListDataFunction (C++ function), 9
DataTransferKit::UserFunctionRegistry::setBoundingVolumeListSizeFunct (C++ function), 9
DataTransferKit::UserFunctionRegistry::setCellListDataFunction (C++ function), 10
DataTransferKit::UserFunctionRegistry::setCellListSizeFunction (C++ function), 9
DataTransferKit::UserFunctionRegistry::setDOFMapDataFunction (C++ function), 10
DataTransferKit::UserFunctionRegistry::setDOFMapSizeFunction (C++ function), 10
DataTransferKit::UserFunctionRegistry::setEvaluateFieldFunction (C++ function), 10
DataTransferKit::UserFunctionRegistry::setFieldSizeFunction (C++ function), 10
DataTransferKit::UserFunctionRegistry::setMixedTopologyDOFMapDataF (C++ function), 10
DataTransferKit::UserFunctionRegistry::setMixedTopologyDOFMapSizeFu (C++ function), 10
DataTransferKit::UserFunctionRegistry::setNodeListDataFunction (C++ function), 9
DataTransferKit::UserFunctionRegistry::setNodeListSizeFunction (C++ function), 9
DataTransferKit::UserFunctionRegistry::setPolyhedronListDataFunction (C++ function), 9
DataTransferKit::UserFunctionRegistry::setPolyhedronListSizeFunction (C++ function), 9
DataTransferKit::UserFunctionRegistry::setPullFieldDataFunction (C++ function), 10
DataTransferKit::UserFunctionRegistry::setPushFieldDataFunction (C++ function), 10
DataTransferKit::UserFunctionRegistry<Scalar>::UserImpl (C++ type), 11
DTK_ADJACENCY_LIST_DATA_FUNCTION (C++ class), 18
DTK_ADJACENCY_LIST_SIZE_FUNCTION (C++ class), 18
DTK_BOUNDARY_DATA_FUNCTION (C++ class), 18
DTK_BOUNDARY_SIZE_FUNCTION (C++ class), 18
DTK_BOUNDING_VOLUME_LIST_DATA_FUNCTION (C++ class), 17
DTK_BOUNDING_VOLUME_LIST_SIZE_FUNCTION

(C++ class), 17
DTK_CELL_LIST_DATA_FUNCTION (C++ class), 18
DTK_CELL_LIST_SIZE_FUNCTION (C++ class), 17
DTK_create (C++ function), 18
DTK_CUDA (C++ class), 17
DTK_destroy (C++ function), 19
DTK_DOF_MAP_DATA_FUNCTION (C++ class), 18
DTK_DOF_MAP_SIZE_FUNCTION (C++ class), 18
DTK_error (C++ function), 19
DTK_EVALUATE_FIELD_FUNCTION (C++ class), 18
DTK_ExecutionSpace (C++ type), 17
DTK_FIELD_SIZE_FUNCTION (C++ class), 18
DTK_finalize (C++ function), 19
DTK_FunctionType (C++ type), 17
DTK_git_commit_hash (C++ function), 18
DTK_initialize (C++ function), 19
DTK_initialize_cmd (C++ function), 19
DTK_is_initialized (C++ function), 19
DTK_is_valid (C++ function), 19
DTK_MIXED_TOPOLOGY_DOF_MAP_DATA_FUNCTION
(C++ class), 18
DTK_MIXED_TOPOLOGY_DOF_MAP_SIZE_FUNCTION
(C++ class), 18
DTK_NODE_LIST_DATA_FUNCTION (C++ class), 17
DTK_NODE_LIST_SIZE_FUNCTION (C++ class), 17
DTK_OPENMP (C++ class), 17
DTK_POLYHEDRON_LIST_DATA_FUNCTION (C++
class), 17
DTK_POLYHEDRON_LIST_SIZE_FUNCTION (C++
class), 17
DTK_PULL_FIELD_DATA_FUNCTION (C++ class),
18
DTK_PUSH_FIELD_DATA_FUNCTION (C++ class),
18
DTK_SERIAL (C++ class), 17
DTK_set_function (C++ function), 20
DTK_UserApplicationHandle (C++ type), 12
DTK_version (C++ function), 18