MIKE
Powered by DHI

SDK User Guide

DFS file system, PFS file system

DHI A/S headquarters
Agern Allé 5
DK-2970 Hørsholm
Denmark

+45 4516 9200 Telephone
+45 4516 9333 Support
+45 4516 9292 Telefax

mike@dhigroup.com
www.mikepoweredbydhi.com

sdk_userguide.docx/JGR/2017-02-09 - © DHI A/S

CONTENTS

SDK User Guide
DFS file system, PFS file system

1 Introduction 1
1.1 Contents of User Guide 2
1.2 Related Content 2

2 DFS File Contents 3
2.1 Header 3
2.2 DFS Items 4
2.2.1 Item unit conversion 5
2.2.2 Static items 6
2.2.3 Dynamic items 6
2.3 Temporal Axes 7
2.4 Spatial Axes 8
2.4.1 Equidistant axes 9
2.4.2 Non-equidistant axes 9
2.4.3 Curve-linear axes 10
2.5 Custom Blocks 10
2.6 Geographic Map Projection and Coordinate Systems 11
2.7 Compression 12
2.8 Statistics Stored in the DFS File 13

3 The DFS .NET API 15
3.1 Prerequisites for Using the .NET API 16
3.2 Opening a DFS file – DfsFileFactory 16
3.3 Generic DFS File – IDfsFile 17
3.4 Accessing Static Items – IDfsFileStaticIO 17
3.5 Accessing Dynamic Item Data – IDfsFileIO 18
3.6 Header Information – IDfsFileInfo 19
3.6.1 DFS Items 19
3.6.2 Temporal Axis – IDfsTemporalAxis 20
3.6.3 Spatial Axis – IDfsSpatialAxis 21
3.7 DFS Parameters – IDfsParameters 21
3.7.1 Data converters – IDfsDataConverter 21
3.8 Creating New Generic DFS Files – DfsBuilder 22

4 DFS File Formats and .NET API 25
4.1 DFS0 File 25
4.2 DFS1 File 25
4.3 DFS2 File 26
4.4 DFS3 File 26
4.5 Creating DFS1, DFS2, DFS3 Files 27
4.6 DFSU File 28

5 PFS File Content 31
5.1 The PFS .NET API 32
5.1.1 PFSFile 33
5.1.2 PFSBuilder 33
5.1.3 PFSTokenReader 33

6 Using the .NET API 35
6.1 Creating a Project in Visual Studio or SharpDevelop 35
6.2 Using the Command Prompt 36

APPENDICES

A Modify Times 39

B Storing Unicode Strings as Items 41

C DFS and COM 43

D DFS in Matlab 45

E DFS in Matlab through COM 47

F Description of the Item Value Types 49

G DFS Data Type Tags Currently Used 55

INDEX

Index 59

1 Introduction

The DFS (Data File System) and PFS (Program File System) are two file formats that are used extensively within the MIKE Powered by DHI software.

The DFS (Data File System) is a binary data file format. It provides a general file format for handling spatially distributed and time dependent data, ranging from measurements of temperature at a single point, Figure 1.1, to water levels in the North Sea in a 2D grid generated by DHI's flow models (MIKE 21), Figure 1.2.

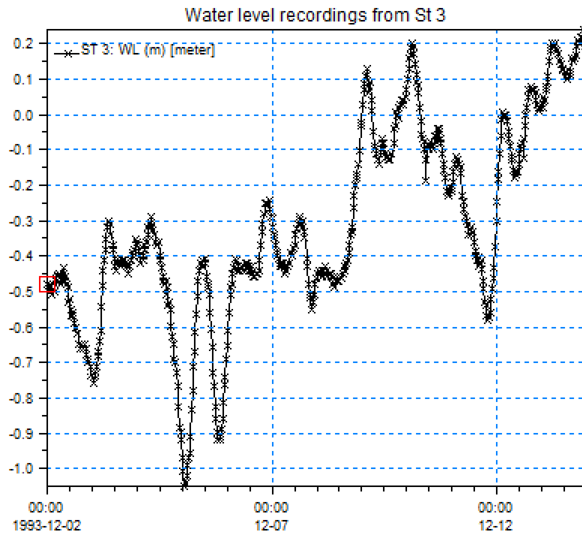


Figure 1.1 Example from dfs0 file, water level measurements from Station 3

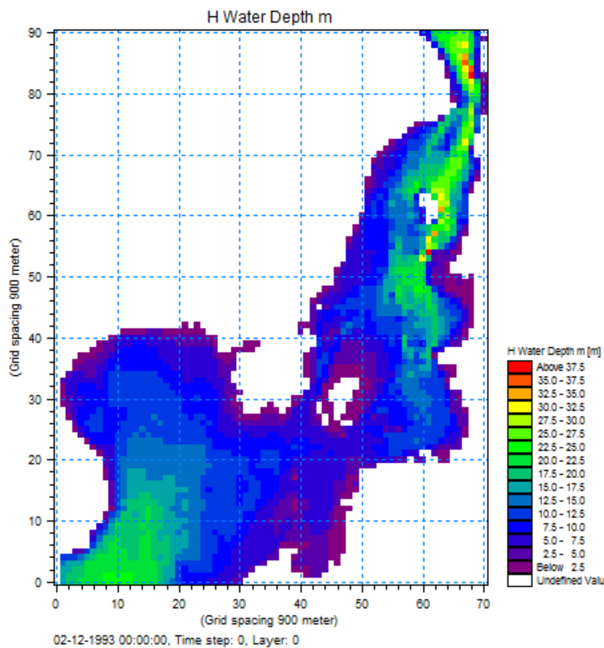


Figure 1.2 Example from dfs2 file, first time step of a 2D water level item

The PFS (Program File System) is a text file format. It is used as inputs to various tools and engines, i.e. most of the setup files for tools and engines are using the format of PFS. Examples are (not a complete list, there are many more than these):

MIKE 11: .sim11, .hd11, .nwk11, .rr11, .ad11
MIKE 21: .m21, .m21c, .m21fm, .sw, .pms
MIKE SHE: .she, .wbl
Plot composer: .plc
MIKE Zero toolbox: .mzt

This document serves several purposes:

1. It is an introduction to the DFS and PFS, and the terms and definitions used when talking about DFS and PFS files.
2. It functions as a guide on how to read and write DFS and PFS files.

1.1 Contents of User Guide

This first section is an introduction to the document and its contents.

Section 2 describes the contents stored in a DFS file, and introduces the terms and definitions used in conjunction with DFS.

Section 3 describes the available .NET Application Programme interface (API) for working with DFS files.

Section 4 describes each of the DFS file formats and the specialised .NET API for working with the files.

Section 5 describes the content stored in a PFS file and the .NET API for working with PFS files.

Section 6 describes how to use the .NET API from e.g. Visual Studio.

The appendix includes further details and examples.

1.2 Related Content

All documents related to MIKE SDK are available as PDF files (Acrobat Reader) or online help from the MIKE SDK Documentation Index. This index includes links to the following documentation:

- This user guide
- Documentation of the DFSU file format, Technical Documentation
- Class library documentation for the DFS .NET API
- Class library documentation for the PFS .NET API
- Class library documentation of the DHI.Projections library

A number of examples using the DFS.NET API in C#, Iron Python and CsScript can be downloaded from GitHub at https://github.com/DHI/MIKECore-Examples.
For Matlab users a number of examples can be found in the DHI Matlab Toolbox. The toolbox can be downloaded from GitHub at https://github.com/DHI/DHI-MATLAB-Toolbox/releases.

 DFS file system, PFS file system - © DHI A/S

2 DFS File Contents

A DFS file is a binary file that contains data for a number of quantities at a number of times.

A DFS file is conceptually split into:

- A header section, containing general information for the file, as start time, geographic map projection, etc.
- A section with static data, containing data for a number of items. Static data does not have any notion of time, and is thus independent of time.
- A section with dynamic data, containing data for a number of time steps and items.

Header

Static data
Item1, item2, ...

Dynamic data
Timestep 1: item1, item2, ...
Timestep 2: item1, item2, ...

Figure 2.1 DFS file contents

The Dynamic data section is usually by far the section using most disc space.

2.1 Header

The header contains metadata describing the file, its contents and especially the contents of the two data sections.

- File title; user defined title of the file.
- Application title; title of the application that created the file.
- Application version number; version of the application that created the file.
- Data type; used to tag the file as a special DFS file type, see (1) below.
- Type of DFS file storing format, see (2) below.
- Type of statistics; the level of statistics stored for dynamic items.
- Delete values, see (3) below.
- Geographic map projection information.
- Time axis information.
- Custom blocks; a number of (small) arrays of a certain type, identified by its name.
- Compression encoding; when the file is compressed, defines where compressed data point belongs to.

Furthermore, the header contains descriptions of each of the dynamic items. The header does not contain any information on the static items; neither does it contain the number of static items in the file. Static items must be read one by one until there is no more. A read operation will provide as well static item data as information on the static item.

The type of statistics, geographic map projection information, time axis information, custom blocks and compression encoding is described in the next sections.

1. The data type tag is a user specified integer. The data type tag is used as an identification tag for the type of DFS file at hand. The user should tag bathymetries, result files, input files etc. matching those tags required for the DFS file type at hand. A wrong data type tag in some contexts is an error. Not all DFS files and tools handling DFS use the tag.

   In Appendix G there is a list of data type tags currently used within the DHI model complex. A programmer writing a new type of DFS file should choose the tag carefully, such that it does not interfere with existing DFS file types.

2. The storing format specifies whether all items are stored in all time steps, and if a time-varying spatial axes is used. Currently the DFS file system only supports files with all items in all time steps and not any of the time-varying spatial axes. Thus this is not used, and is not presently planned to be explored.

3. A delete value represents a not-defined value. If a value is not available, or if it does not make sense to set a value, setting the delete value indicates that there is no value. There is a delete value for data of type float, double, integer (32 bit), unsigned integer and byte (8 bit).

2.2 DFS Items

An item is the smallest data unit that is read and written to a DFS file. It can either be static, in which case it only has one set of values, or it can be dynamic, in which case it has a set of values for each time step in the file. Both types of items are described by:

- Name; user defined description of the item
- EUM quantity, i.e., EUM type and EUM unit
- Spatial axis
- Type of data, being float, double, integer, etc.
- Reference coordinates and orientation

EUM quantity

An item defines the data being stored by use of the DHI EUM system. EUM is short of Engineering Unit Management. The EUM system specifies a combination of a type and a unit. The EUM type could be ‘water level’, and the unit ‘meters’. The EUM system assures that the type and unit matches, i.e. it is not possible to specify a unit of square meters for a water level.

Spatial axis

The size and dimension of an item is defined by its spatial axis. An item can store data of the form of:

- Scalars – dfs0 data
- Vectors – dfs1 data
- Matrices – dfs2 data
- Cubes/3D matrices – dfs3 data

The items of a DFS file need not all have the same spatial axis. However, many of the specialised DFS file formats requires that all items have the same spatial axis. See Section 2.4 for details on the different spatial axes.

Type of data

Types of data that can be stored in a DFS item (the `DfsSimpleType`) are:

- float
- double
- byte (8 bit integer, char in C++)
- short (16 bit integer)
- unsigned short (16 bit integer)
- integer (32 bit integer)
- unsigned integer (32 bit integer)

Reference coordinates and orientation

An item also holds a set of reference coordinates and orientation, which can be used to translate and rotate the spatial axis of the item compared to the user coordinate system in the file. However, these are not used by most DFS file types: Only in some versions of the dfs0 file is the reference coordinates set. See Section 2.6 for details on coordinate systems and geographic map projections.

2.2.1 Item unit conversion

The EUM quantity defines the item type and item unit that is used when storing item data in the file. As an example, the type could be water level and the unit meters. It is possible to specify a conversion unit, in case another unit than the one stored in the file is preferable. Setting a conversion unit of feet for an item means that whenever data for that item is read from or written to the file, it is in feet. The data is still stored in the file in meters, and converted on the fly.

Two types of unit conversion are available:

1. UBG (Unit Base Group) conversion
2. Free conversion

The UBG conversion will convert data to the unit specified in the Unit Base Group settings. The UBG system is used to set the default unit for various item types, and is dependent on your configuration. For example can a user choose to use Imperial units, in which e.g. length are in feet or mile.

Using the free conversion, the user needs to specify the unit that the data is to be converted to.

Unit conversion can be specified not only for the item data but also for the spatial axis of the item, thus converting the data of a spatial axis. Example; for a 1D equidistant axis the starting point x0 and the axis interval dx values will be converted.

Unit conversions are not stored in the file, but must be set every time the file is opened.

Note when using unit conversions: EUM types and units reported from the file are not changed, only the data is changed. Example, having a file with spatial axis in meters, when setting unit conversion for the spatial axis to feet and then requesting the spatial axis, the spatial axis will report the unit meters, but the axis data will be in feet.

2.2.2 Static items

A static item stores one set of values for the item. A static item has no notion of time.

A DFS file can have any number of static items. To access the static items, they must be read one by one, i.e. it is not known in advance how many static items a file has. Static item data and information on its unit, data type, etc are stored together.

On top of what describes the generic DFS item as described previously, a static item includes the actual data of the static item.

2.2.3 Dynamic items

A dynamic item varies in time. Information of the dynamic item like unit, data type, etc is stored in the header, while the data is stored separately on a time-step and item basis.

On top of what describes the generic DFS item described previously, it also includes

- Value type, being instantaneous, forward step, etc.
- A list of associated static item numbers
- Statistics of the item data

Each of these are described in the following.

Value type

The *Value type* in time specifies how each value is to be interpreted between two time step values.

- Instantaneous; the value is defined at the time specified.
- Accumulated; the value is an accumulated value from the start time of the file to the time specified.
- Step-accumulated; the value is accumulated between last time step to current time step.
- Mean-step-backward; mean value from previous time step time to current time step time. This is also sometimes called ‘mean-step-accumulated’.
- Mean-step-forward; mean value from current time step time to next time step time. This is also sometimes called ‘reverse-mean-step-accumulated’.

It is currently only the dfs0 file format that utilises the different value types. The remainder of the DHI DFS files uses the instantaneous type. See Appendix F for examples of the different value types.

Associated static item numbers

A dynamic item can have a list of static item numbers that in some way is associated with the dynamic item. These are not very often used, and there is not predefined definition of the properties of the association – it depends on the type of DFS file.

Statistics

Together with each dynamic item, some statistics of its data can be stored, e.g., the max and min value for all data values and time steps. See section 2.8 for details.

2.3 Temporal Axes

The time in the DFS file can be specified as a relative time, starting from zero, or as an absolute time, starting from a specified date and time. The former is called a time axis, the latter a calendar axis.

Each of the two exists in an equidistant and a non-equidistant version.

For the equidistant type temporal axes you can specify a start time offset. It defines the time of the first time step relative to the start time.

For the non-equidistant type time axes the actual times for each time step is stored together with the dynamic item data. The times are therefore not available in the temporal axis definition, but are retrieved when reading data for an item-time step. The times stored with each time step is the time in a given time unit relative to the start of the file.

The *equidistant time* axis is defined by:

- Time unit
- Time step size
- Start time offset
- Number of time steps

The *non-equidistant time* axis is defined by:

- Time unit
- Time span – difference between first and last time step.

The *equidistant calendar* axis is defined by:

- Start date and time
- Time unit
- Time step size
- Start time offset
- Number of time steps

The *non-equidistant calendar* axis is defined by:

- Start date and time
- Time unit
- Time span – difference between first and last time step

The two non-equidistant temporal axes also provide a start time offset. This start time offset cannot be set, but has the time value of the first time step stored in the file, and is for information only.

See also the Appendix A for details on how the temporal axis parameters are handled

2.4 Spatial Axes

The spatial axis defines the dimension of the data, and the size of the data in each dimension for an item.

The axes that are available currently belong into three categories:

- The equidistant axes
- The non-equidistant axes
- The curve-linear axis

All axes coordinates are specified in the user defined coordinate system, see Section 2.6 for details.

The data in an item can either be '*node based*' or '*element based*'. The difference can be seen in Figure 2.2 and Figure 2.3, which both define an item with 9 values ordered in a 3 by 3 grid. Figure 2.2 defines the values on the nodes and Figure 2.3 defines the values in the centre of each element.

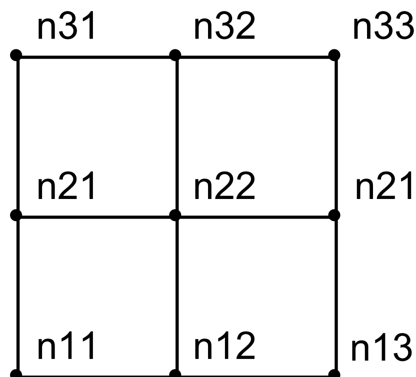


Figure 2.2 Node based 3 by 3 values

e31	e32	e33
e21	e22	e21
e11	e12	e13

Figure 2.3 Element based 3 by 3 values

There is also a set of time-varying axes, which are currently not supported by DFS.

2.4.1 Equidistant axes

The equidistant axes define a structured orthogonal grid of a certain dimension and size. The axes specify for each dimension:

- The start coordinate offset
- The grid spacing
- The number of data points in that dimension

There are equidistant axes ranging in dimensions from 1D to 3D.

The equidistant axes do not specify whether the values are 'node based' or 'element based'. See Figure 2.2 and Figure 2.3 for the difference.

Currently all files in DHI software with an equidistant axis are element based.

2.4.2 Non-equidistant axes

The 1D non-equidistant axis defines a line in 2D/3D space and specifies a number of (x,y,z) coordinates where the data values belong. The number of coordinates matches the number of data values, the values are defined on the coordinates, and i.e. they are 'node values'. The 1D non-equidistant axis is conceptually more alike a 1D curve-linear axis, but historically not named as such.

The 2D and 3D non-equidistant axes define an orthogonal grid with non-equidistant grid spacing. For each dimension is specified the coordinates in that dimension. The number of coordinates is one longer than the number of data values, i.e. the values are 'element values'.

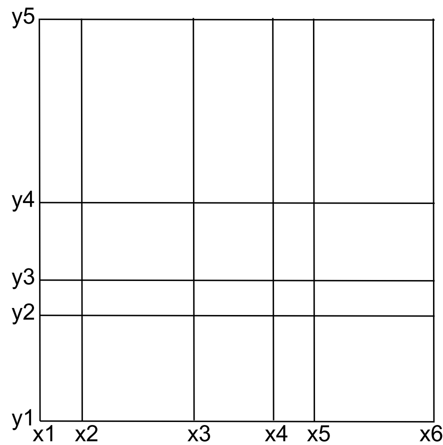


Figure 2.4 Non-equidistant 2D grid axis

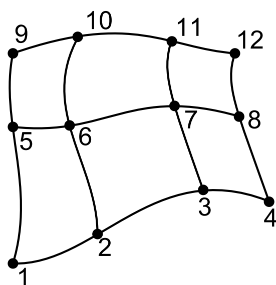
Figure 2.4 shows a 2D non-equidistant axis with 6 ‘x’ coordinates and 5 ‘y’ coordinates. The number of data values in the item is (6-1) x (5-1) = 20.

2.4.3 Curve-linear axes

There is a 1D curve linear axis, but it is called the 1D non-equidistant axis and described in previous section.

The 2D and 3D curve linear axis describe a grid that can bend, i.e. it is no longer Cartesian.

The grids are specified by a number of node coordinates. The number of coordinates in each dimension is one larger than the number of data values in each dimension, i.e. the values are ‘element values’, cf. Figure 2.3. Nodes are numbered as shown in the figure below, in x-direction first, then y, and for 3D then z.



2.5 Custom Blocks

A custom block is a (small) vector containing data of a certain type. It is identified by its name. The vector is stored in the header section, as opposed to the static items.

A custom block contains a name and the vector data.

The vector type can be any of the `DfsSimpleType` types, see Section 2.2.

2.6 Geographic Map Projection and Coordinate Systems

The projection information stored in a DFS file consists of:

- A projection string
- A reference longitude and latitude coordinate
- A reference orientation

The DFS file works with three coordinate systems:

1. *Geographical coordinate system*, containing longitude and latitude coordinates
2. *Projected coordinate system*, containing easing and northing coordinates
3. *User defined coordinate system*, containing x and y coordinates

All coordinates in a DFS file are stored in the user defined coordinate system.

Note that every item defines the unit used within the user defined coordinate system, being meters, feet etc, overriding the unit in the projection definition. Each axis can specify its own unit to use in the user defined coordinate system, however many tools assume that all axes of one file use the same unit.

The projection string defines the conversion from geographical coordinates to projected coordinates and back.

The user defined coordinate system is a translated and rotated version of the projected coordinate system. The reference longitude and latitude coordinates defines the origin of the user defined coordinate system. And the orientation defines the rotation clock-wise from true geographical north to the user defined coordinate system y-axis, the compass heading of the y-axis. Note that the orientation is defined based on north of the geographical coordinate system, not the projected coordinate system.

If setting the reference longitude and latitude matching the origin of the projected coordinate system, and setting the orientation to match north in the projection (zero for most map projections), then the projected coordinate system equals the user defined coordinate system. Note that the origin of a projected coordinate system usually is influenced by its false-easting and false-northing parameters, hence the projected coordinate system and the user coordinate system often differs by exactly the false-easting and false-northing values.

Older dfs files may not contain any projection information at all. New dfs files cannot be created without projection information.

2.7 Compression

Files from certain application areas tend to have many values that are delete values. A result file from MIKE 3 contains data in a 3D matrix, and it is not uncommon that 80-90% of the values are delete values.

It is possible to specify which data values are to be stored, and which are not necessary to store, thereby eliminating values that are always delete values from the DFS file and minimising the DFS file size. It is assumed that those data values not being stored are delete values.

When enabling compression, a set of *encoding keys* is specified, that defines which indices in the data set are to be stored. The encoding keys are 3 integer arrays specifying the x, y, and z indices to be stored. The indices stored in the encoding key arrays are zero based.

Example: Assume that data is a 3D matrix, and that a file contains data that are not delete values (10), positioned at z-layer 0 as in this figure.

	0	1	2
5	10	10	10
4	-10.41361	-12.52072	10
3	-12.93544	10	-17.74857
2	10	-17.67839	10
1	-16.32145	10	10
0	-17.45152	10	10

The encoding keys for this file will be

```
xkey = [0,0,1,0,2,0,1];
ykey = [0,1,2,3,3,4,4];
zkey = [0,0,0,0,0,0,0];
```

The encoding key array lengths match the number of data values being stored in the file.

If the data is only 2D, then the z key array should contain all zeros. If the data is only 1D, then the y and z key arrays should contain all zeros.

When writing and reading dynamic data items of a compressed file, compression and decompression is handled automatically on the fly, i.e. the user must provide the full 3D array when writing, and is given the full 3D array when reading.

Compression currently works under the following conditions:

- All the dynamic items in the file must have the same spatial axis.
- All the dynamic items must store its data as floats.

Compression and decompression of static items is not supported on DFS file level. Some file types still store only the compressed data, but then defines a dummy 1D spatial axis having the size of the compressed data. Manual compression and decompression by the user is required when writing and reading such static item data.

2.8 Statistics Stored in the DFS File

When writing item data to a DFS file, the DFS keeps track of some statistical properties for each item. There are two level of statistics for each item: Global and local.

The global level is the default level. It stores for each item the minimum, the maximum and the number of delete values over all data values, and all time steps.

The local level stores for each data point/grid point/element of an item

- Minimum value.
- Maximum value.
- Mean value.
- Standard deviation.
- Auto correlation
- Number of non-delete values
- Number of delete values
- Number of non-delete value pairs, being the number of times two consecutive time steps both contain non-delete values.

The DFS .NET API does not currently support retrieving local level statistics.

3 The DFS .NET API

The DFS .NET API is a set of interfaces and classes for reading, modifying and creating DFS files.

The API is available through the namespace:

```
DHI.Generic.MikeZero.DFS
```

The assembly is registered in the GAC (Global assembly cache). It is installed with MIKE Zero, MIKE URBAN or the MIKE SDK installer.

When developing applications/tools that uses the DFS .NET API, please do install the MIKE SDK. The MIKE SDK contains documentation and binaries for developing such applications/tools.

For running the application/tool, MIKE Zero, MIKE URBAN or MIKE SDK will be sufficient, i.e. if MIKE Zero or MIKE URBAN is already installed, the MIKE SDK is not required.

Class Library Documentation

These chapters provide an overview of the functionality provided by the interfaces and classes. It does not in detail document every interface and class. Detailed documentation can be found in the class library documentation, which exists in the form of

- Online documentation
- Assembly documentation xml-file: DHI.Generic.MikeZero.DFS.xml

The online class library documentation for DFS can be accessed via the MIKE SDK documentation index.

The assembly documentation file is located together with the assembly in the MIKE SDK bin folder. The assembly documentation xml file is used by IDE's like Visual Studio or SharpDevelop, making the documentation available within the IDE. The documentation is also available in the Visual Studio Object Browser, when loading the

DHI.Generic.MikeZero.DFS.dll.

It requires that the assembly documentation xml file is available together with the assembly file.

Examples

For examples, look in the examples that can be downloaded from GitHub. This includes examples in C#, CS-script (C# scripting engine) and IronPython. Examples for Matlab are provided with the DHI Matlab Toolbox.

DFS API structure

The API is divided into two levels, each giving different access to the DFS functionality: The generic API and the specialised API's.

The generic API allows for reading/modifying/creating DFS files with all the freedoms and restrictions that DFS impose.

The specialised API's works with one type of DFS file and limits the use of DFS to what the type of DFS file utilises. For example, when using the API for a dfs2 file, that API assumes that all items are using the same 2D spatial axis, and does not allow access to the spatial axis on item level, only on file level.

The user should use the specialised API's whenever possible. The generic API should only be used when the file at hand cannot be handled by one of the specialised APIs.

The following sections will give a short overview of the general API, while Section 4 will describe each specialised file type and their API.

3.1 Prerequisites for Using the .NET API

In order to use the .NET API, one of the DHI Software products including the `DHI.Generic.MikeZero.DFS` must be installed (e.g. any MIKE ZERO product, MIKE URBAN or the MIKE SDK).

In order to develop against the .NET API, the MIKE SDK must be installed.

The .NET API only supports 64 bit mode.

3.2 Opening a DFS file – DfsFileFactory

The `DfsFileFactory` provides methods for opening existing DFS files. It currently supports the following file types:

- dfs1
- dfs2
- dfs3
- dfsu (not all types)
- generic dfs

If your DFS file type is not listed as one of the supported specialised types in the list above, you can always open it using the generic DFS functionality.

Section 4 will describe each specialised file type and their API.

Each file can be opened in read mode, edit mode or append mode.

In read mode the data in the file cannot be updated. The file is opened for reading, and the file pointer is positioned at the first item-time step in the file.

In edit mode, the data in the file can be updated. The file pointer is positioned at the first item-time step, as in read mode.

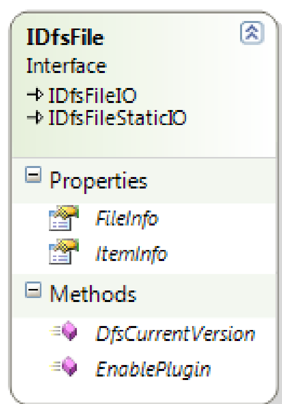
In append mode the file is opened for editing, and the file pointer is positioned after the last item-time step in the file: If writing data to the file, the data will be appended.

The `DfsFileFactory` also provides support for reading and writing mesh files.

3.3 Generic DFS File – IDfsFile

The `IDfsFile` is the main entrance to a generic DFS file.

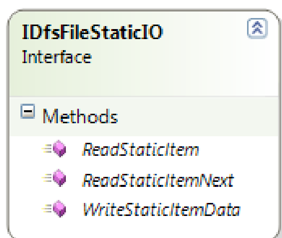
A file can be opened for reading, editing or appending by using one of the static `DfsFileFactory` methods `DfsGenericOpen`, `DfsGenericOpenEdit` or `DfsGenericOpenAppend`.



The `IDfsFile` provides functionality for reading and writing dynamic and static items by extending the `IDfsFileIO` and `IDfsFileStaticIO` interfaces. Furthermore it provides access to the header (`FileInfo`) and a list of item info defining each of the dynamic items.

3.4 Accessing Static Items – IDfsFileStaticIO

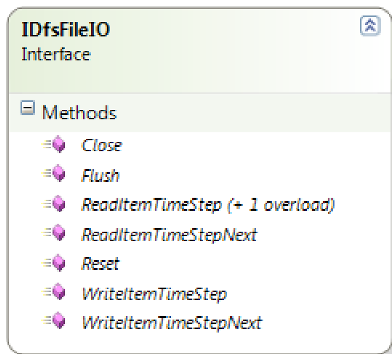
The interface `IDfsFileStaticIO` provides functionality for reading and writing static item and its data.



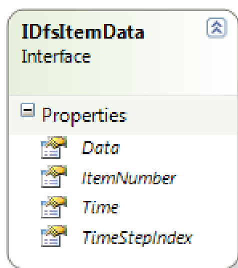
The read functions return an `IDfsStaticItem` which includes the specification of the static item, and the static item data values. Use the `ReadStaticItemNext` to iterate through all the static items. When no more static items are present, null is returned.

3.5 Accessing Dynamic Item Data – IDfsFileIO

The interface `IDfsFileIO` provides functionality for reading and writing dynamic item data. To get information on the dynamic items use the `IDfsFile.FileInfo` list.



The read functions return an `IDfsItemData` interface, which contains the item number, the time of the time step and the data values.



Depending on the `DfsSimpleType` of the dynamic item being read, the `IDfsItemData` can be cast to its generic type: If the dynamic item stores float values, it can be cast to `IDfsItemData<float>`. Alternatively can the data from `IDfsItemData.Data` be cast to the native array, example `float[]`.

Regarding the ‘next’ version of the read and write functions: It reads/writes data for the next dynamic item-time step. If called as the very first read/write function, it returns the first time step of the first item. It then cycles through each time step, and each item in the time step.

For a file with 3 items, it returns (item-number, time step index) in the following order:

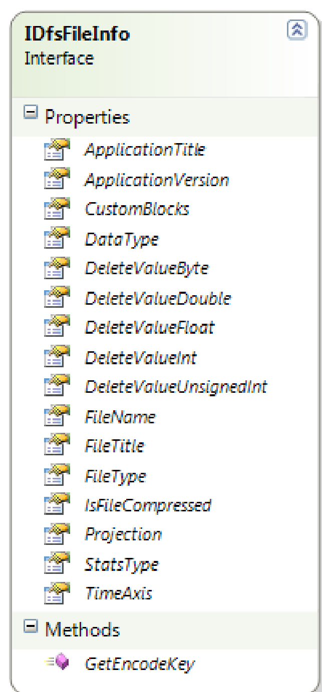
(1,0), (2,0), (3,0), (1,1), (2,1), (3,1), (1,2),...

If data is explicitly read for item-time step (2,4), next item-time step will be (3,4).

If one of the methods reading/writing static item data is called, the iterator is reset, and the next call to this method again returns the first item-time step. Similar for the static items, whenever a dynamic item is being read/written, the iterator for the static items is also reset.

3.6 Header Information – IDfsFileInfo

The `IDfsFileInfo` interface provides access to data in the header, not including those specifying the dynamic data. It is available from the `IDfsFile` interface (the `FileInfo` property) and also from some of the specialised DFS file interfaces, e.g. `IDfs123File`.



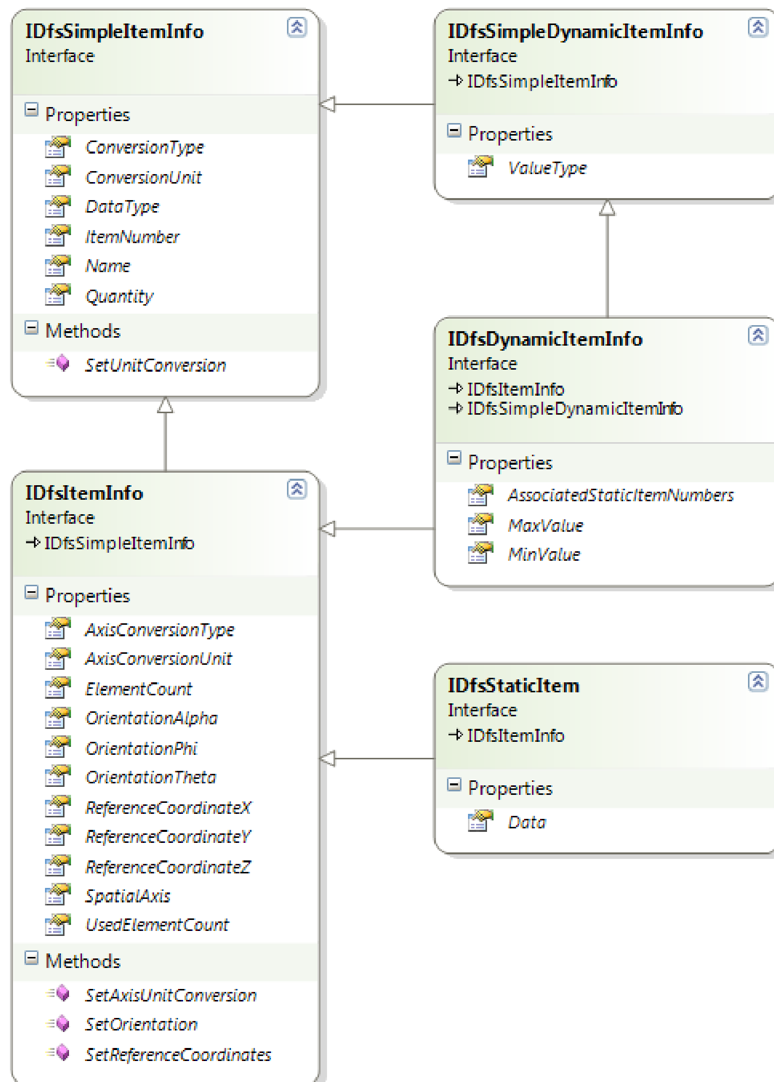
3.6.1 DFS Items

The item information exists in two versions: A full version and a simple version. The simple version hides the spatial information of the axis, i.e. spatial axis and reference coordinates and orientation is hidden.

Most of the DFS files do not allow individual items to have different spatial axis. Such files will provide the axis definition on the level of the file class, not on item level. Hence the item information for such files is limited to the simple version, without any spatial information.

For the dynamic item the versions are the full `IDfsDynamicItemInfo` interface and the simple `IDfsSimpleDynamicItemInfo` interface.

For the static items, currently only the full version of the interface exists, `IDfsStaticItem`.



Each DFS file type class includes a list of its dynamic items, either as a list of `IDfsDynamicItemInfo` or as a list of `IDfsSimpleDynamicItemInfo`.

3.6.2 Temporal Axis – IDfsTemporalAxis

The `IDfsTemporalAxis` interface is the base interface for all the specialised temporal axes: All the specialised temporal axes extend the `IDfsTemporalAxis` interface.

The `IDfsTemporalAxis` interface has a `TimeAxisType` property that defines which of the specialised time axis it really is. There is no temporal axis implementing only the `IDfsTemporalAxis` interface.

The specialised temporal axes are:

- `IDfsEqCalendarAxis`
- `IDfsEqTimeAxis`
- `IDfsNonEqCalendarAxis`
- `IDfsNonEqTimeAxis`

Based on the `TimeAxisType` the temporal axis object can be cast to its matching specialised temporal axis.

3.6.3 Spatial Axis – IDfsSpatialAxis

The `IDfsSpatialAxis` interface is the base interface for all specialised spatial axes: All the specialised temporal axes extend the `IDfsSpatialAxis` interface.

The `IDfsSpatialAxis` interface has an `AxisType` property that defines which of the specialised spatial axis it really is. There is no spatial axis implementing only the `IDfsSpatialAxis` interface.

The specialised spatial axes are:
- `IDfsAxisEqD0`
- `IDfsAxisEqD1`
- `IDfsAxisEqD2`
- `IDfsAxisEqD3`
- `IDfsAxisEqD4`
- `IDfsAxisNeqD1`
- `IDfsAxisNeqD2`
- `IDfsAxisNeqD3`
- `IDfsAxisCurveLinearD2`
- `IDfsAxisCurveLinearD3`

Based on the `AxisType` the spatial axis object can be cast to its matching specialised spatial axis.

3.7 DFS Parameters – IDfsParameters

The `IDfsParameters` contains a number of parameters that can be set when reading/editing DFS files. It is used as argument in the `DfsFileFactory` methods.

- `UbgConversion`: When set, it will set the unit conversion to UBG for all items and item axis. See section 2.2.1 for details. Default is false.
- `ModifyTimes`: When set, the times will be modified as described in Appendix A. Default value is false.
- `EnablePlugin`: When set, it will enable the DFS plugin, which for some DFS file types will produce derived dynamic items. For example can a file with the bathymetry in the static data, and water level in the dynamic data produce a surface elevation as an additional derived dynamic item.

Furthermore a set of data converters can be specified, which are described in the following section.

3.7.1 Data converters – IDfsDataConverter

An `IDfsDataConverter` can do conversion of an item and its data. A data converter also has the ability to change the item properties, as the EUM quantity, the data type etc. The following converters are provided with the .NET API:

- `DfsConvertFloatToDouble`: This will convert item data from float to double precision numbers. It also updates the data type returned by the item, to double.
- `DfsConvertToUbg`: This will convert item data and item spatial axis to UBG units. See section 2.2.1 for details. Compared to setting the unit conversion to UBG units, using this data converter will also change the item EUM type and unit.
- `DfsConvertToBaseUnit`: This will convert item data and item spatial axis to EUM base units (almost always SI). It also updates the item and spatial axis EUM type and unit to base units.

The converters are designed to return item data that are still consistent, i.e. when converting data to another unit, also the item quantity is updated to that unit.

Example: Assume you are developing a tool that does some analysis of DFS data, basing the analysis on SI units and double precision data. Adding the two converters, `DfsConvertToBaseUnit` and `DfsConvertFloatToDouble`, to the DFS parameters, the tool need not bother which unit or type the data is stored in, conversions are performed automatically when required.

Users can implement their own data converters, by implementing the `IDfsDataConverter` interface. The converters described above are provided as part of the example codes.

3.8 Creating New Generic DFS Files – DfsBuilder

The `DfsBuilder` class are used when creating new DFS files. The builder assures that the DFS file is build and defined correctly, assuring that all data that is required to be set, is set.

Only use the `DfsBuilder` class if one of the specialised DFS file builder classes does not provide the desired functionality.

The `DfsBuilder` class works together with the `DfsFactory` class: The `DfsFactory` class creates objects that can be used as arguments for the `DfsBuilder` methods, as e.g. when specifying the temporal axis.

The builder works in two stages: The first stage all header information and information of the dynamic items is provided. In the second stage static items are added.

In the first stage the following header information must be provided:

- Geographic map projection
- Data type tag
- Temporal axis

Furthermore, all the dynamic items must be added and specified correctly. For that a `DfsDynamicItemBuilder` is used. For each dynamic item you must specify:

- Name.
- Quantity
- Spatial axis
- Type of data
- Value type

The remainder of the functionality in the builders are optional:

- File title
- Application title and version number
- Delete values
- Custom blocks

And for the dynamic items the following optional parameters:

- Value type; default is Instantaneous
- Associates static items
- Reference coordinates and orientation

During the first stage, you can at any time call the Validate function which will return a list of strings with errors. If the list is empty, you are ready to enter stage 2.

When the header and the dynamic items are fully defined, i.e. no validation errors left, the CreateFile function can be called, and the builder will enter stage 2. In the second stage any number of static items can be added. You can use the `IDfsStaticItemBuilder` to build a generic static item. With the static item builder, the following information is required:

- Name.
- Quantity
- Spatial axis
- Data

There are also a number of convenience functions for adding ‘simpler’ static items, by providing a name and a dataset only, and then dummy axis and quantity values will be used.

If/when all static items have been added, the GetFile method is called which returns an `IDfsFile`. From that point the builder is no longer required (and is actually invalidated). Dynamic item data is added by using the methods in the `IDfsFileIO` interface.

4 DFS File Formats and .NET API

This chapter describes each of the specialised file formats that are used within the MIKE Powered by DHI software package.

To open a file in one of the file formats, use the `DfsFileFactory`.

4.1 DFS0 File

A dfs0 file is also called a time series file. A dfs0 file only contains items with the 0D equidistant spatial axis.

There are two ways of geo-referencing the individual time series in the file:

1. By setting the reference lon-lat coordinates in the projection
2. By setting the reference (x,y,z) coordinates in the dynamic item

Using the former case, all dynamic items must be situated at the same location. The latter case allows for having different locations for each dynamic item. However, many of the DHI tools assume the former. There are also many dfs0 files that are not geo-referenced at all.

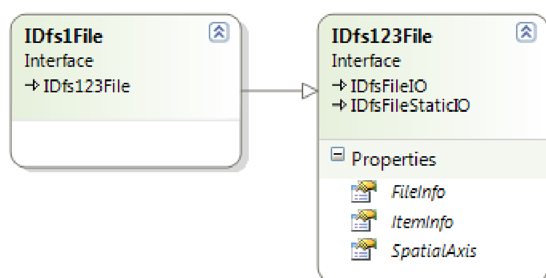
A dfs0 file generally does not have any static items or custom blocks. If any are present, most tools ignore them.

To open/write a dfs0 file, the generic dfs API must be used. There is currently no dedicated API for handling dfs0 files, so it is the responsibility of the user to utilise the generic dfs API in a way that fulfils the requirements for a dfs0 file.

4.2 DFS1 File

A dfs1 file is also called a line series file. A dfs1 file only contains items with one of the 1D spatial axes. All values are instantaneous.

Use one of the static `DfsFileFactory` methods `Dfs1FileOpen`, `Dfs1FileOpenEdit` or `Dfs1FileOpenAppend` to open a dfs1 file. It will return an `IDfs1File`.



The IDfs1File interface extends the IDfs123File, IDfsFileIO and IDfsFileStaticIO. It provides information on the header, the items and the spatial axis.

To create a new dfs1 file, use the Dfs1Builder class, see Section 4.5.

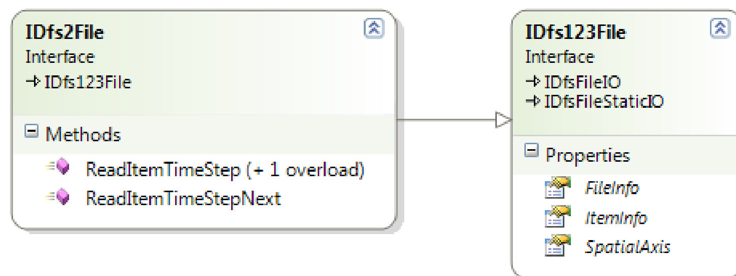
4.3 DFS2 File

A dfs2 file is also called a grid series file. A dfs2 file only contains static and dynamic items with one of the 2D spatial axes. All values are instantaneous.

Values in a dfs2 file are 'element based', i.e. values are defined in the centre of each grid cell.

The definition of the origin, which is set by the origin coordinates in the projection, depends (for historical reasons) on the projection being used. For the 'NON-UTM' projection, the origin is the lower left grid boundary (the lower left node). For all other projections the origin is the centre of the lower left element. The two definitions differ by half a grid cell.

Use one of the static DfsFileFactory methods Dfs2FileOpen, Dfs2FileOpenEdit or Dfs2FileOpenAppend to open a dfs2 file. It will return an IDfs2File.



The IDfs2File interface extends the IDfs123File, IDfsFileIO and IDfsFileStaticIO. It provides information on the header, the items and the spatial axis. It overrides the read methods in IDfsFileIO to return IDfsItemData2D instead of IDfsItemData. The IDfsItemData2D provides 2D indexing into the data.

To create a new dfs2 file, use the Dfs2Builder class, see section 4.5.

4.4 DFS3 File

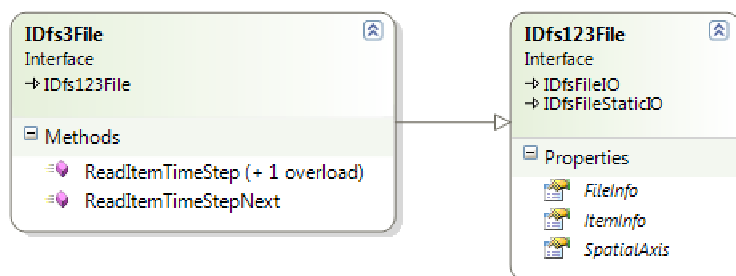
A dfs3 file is also called a grid series file. A dfs3 file only contains static and dynamic items with one of the 3D spatial axes. All values are instantaneous.

Values in a dfs3 file are 'element based', i.e. values are defined in the centre of each grid cell.

If the file is compressed, data in static items can choose to store only the compressed part of the data, and decompression is required. In that case it will have a dummy 1D axis

of the compressed data size. You can use the `DfsExtensions.dfsDecode()` method and `DfsExtensions.dfsEncode` to decompress and compress data.

Use one of the static `DfsFileFactory` methods `Dfs3FileOpen`, `Dfs3FileOpenEdit` or `Dfs3FileOpenAppend` to open a dfs3 file. It will return an `IDfs3File`.



The `IDfs3File` interface extends the `IDfs123File`, `IDfsFileIO` and `IDfsFileStaticIO`. It provides information on the header, the items and the spatial axis. It overrides the read methods in `IDfsFileIO` to return `IDfsItemData3D` instead of `IDfsItemData`. The `IDfsItemData3D` provides 3D indexing into the data.

To create a new dfs3 file, use the `Dfs3Builder` class, see section 4.5.

4.5 Creating DFS1, DFS2, DFS3 Files

To create a new file of type dfs1, dfs2 or dfs3, use one of the `Dfs1Builder`, `Dfs2Builder` or `Dfs3Builder` classes.

The difference between the builders is which spatial axis type that it accepts, and the type of file that it returns in the end.

The builders work together with the `DfsFactory` class which can create the objects to use as arguments for the builder methods

The following information must be provided

- Map projection
- Data type tag
- Temporal axis
- Spatial axis

The type of the spatial axis accepted by the builder depends on the builder. Example: The `Dfs2Builder` only accepts 2D axes valid for a dfs2 file, etc.

All the dynamic items must be added and specified correctly. For each dynamic item you must specify:

- Name
- Quantity
- Type of data
- Value type

When the header and the dynamic items are fully defined, i.e. no validation errors left, the `CreateFile` function can be called, and the builder will enter stage 2. In the second stage any number of static items can be added. For each static item the following information is required:

- Name
- Quantity
- Data

If/when all static items have been added, the `GetFile` method is called which returns an `IDfs1File`, `IDfs2File` or `IDfs3File` depending on the builder. From that point the builder is no longer required (and is actually invalidated).

Dynamic item data is added by using the methods in the `IDfsFileIO` interface.

4.6 DFSU File

The DFSU file and its geometry is in detail described in the DHI Flexible File Formats specification (FM_FileSpecification.pdf).

The geometrical definition of a dfsu file is stored in one custom block and a number of static items. A dfsu file cannot have any other custom blocks or static items than those defining the geometry. These should not be hand edited, but edited through the `IDfsuFile` interface.

The temporal axis is always an equidistant calendar axis, with time unit in seconds.

The dfsu file stores all coordinates in the projected coordinate system and not the user defined coordinate system, as other dfs file types. Hence the reference longitude and latitude and orientation are not relevant for a dfsu file.

The mesh in a dfsu file consists of a number of nodes and a number of elements.

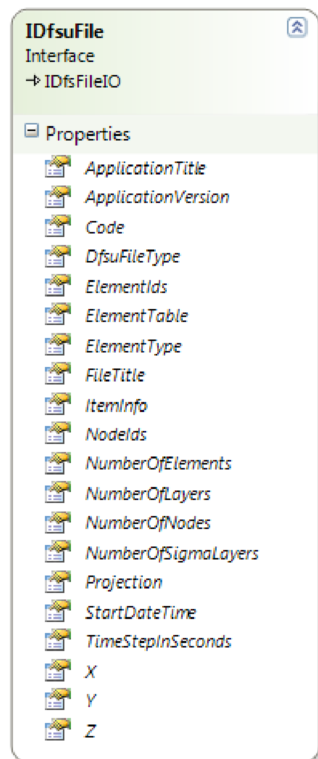
Each node has:

- Node id
- X,Y,Z coordinate
- Code for the boundary

Each element has:

- Element id
- Element type; triangular, quadrilateral, prism etc.
- Element table; specifies for each element the nodes that defines the element. The element table references a node by specifying the number in the list of nodes (not the index!). 2D elements specify their node in counter clockwise order. For details, see the FM_FileSpecification.pdf.

Use one of the static `DfsFileFactory` methods DfsuFileOpen, `DfsuFileOpenEdit` or `DfsuFileOpenAppend` to open a dfsu file. It will return an `IDfsuFile`.



The `IDfsuFile` extends the `IDfsFileIO`. The geometry definition, header and item information is available directly from the `IDfsuFile` interface.

To create a new dfsu file, use the `DfsuBuilder` class. The following header information must be provided:

- Projection information
- Temporal information.
- Node coordinates and code
- Element table

Furthermore at least one dynamic item must be added. For each dynamic item the name and EUM quantity is required.

When the header and the dynamic items are fully defined, i.e. no validation errors left, the `CreateFile` function can be called, which will return an `IDfsuFile`. From that point the builder is no longer required (and is actually invalidated).

Dynamic item data is added by using the methods in the `IDfsFileIO` interface.

5 PFS File Content

A PFS file is a text file that contains parameters and settings for tools and engines.

A PFS file contains

- Targets (out-most section)
- Sections
- Sub-sections
- Keywords
- Parameters, in the form of double, integer, bool, filename, string, CLOB or undefined
- CLOB (Character Large OBject) can itself contain parameters in the form of double, integer, bool and strings.

An example of a small PFS file is included here.

```
[Run11]
   key1 = 2, true, , 12
   key2 = 3.3, 4, 'someText'
   key2 = '<CLOB:2,-2.75,7.07,4,7.07>'
   [Results]
      [Result]
         outid = 'default out'
         file = |.\output.res11|
      EndSect  // Result

   EndSect  // Results

EndSect  // Run11
```

A PFS file is build up hierarchically, starting with targets and ending with parameters.

A PFS file can contain any number of targets. A target is also a section. It has a name.

A section has a name. It can contain any number of sub-sections and any number of keywords, in a certain order.

A keyword has a name. It can contain any number of parameters.

A parameter contains a value being either

- Integer; a number without a decimal separator
- Double; floating point number with a decimal separator
- Boolean; true or false
- Text string; delimited by single pings: `'text'`.
- File name; delimited by a bar: `|filename|`.
- CLOB; special form of text string: `'<CLOB:2,-2.75,7.07,4,7.07>'`.

Names of targets, sections and keywords need not be unique: Having a list of data is often stored as a set of sections or keywords with the same name.

A file name is a special type of string. It is assumed that the path of the string is relative to the location of the PFS file, and the absolute path is returned when requesting its value.

5.1 The PFS .NET API

The PFS .NET API is a set of interfaces and classes for reading, modifying and creating DFS files.

The API is available through the namespace:

```
DHI.PFS
```

The assembly is registered in the GAC (Global assembly cache). It is installed with MIKE Zero, MIKE URBAN or the MIKE SDK installer.

When developing applications/tools that uses the PFS .NET API, please do install the MIKE SDK. The MIKE SDK contains documentation and binaries for developing such applications/tools.

For running the application/tool, MIKE Zero, MIKE URBAN or MIKE SDK will be sufficient, i.e. if MIKE Zero or MIKE URBAN is already installed, the MIKE SDK is not required.

Class Library Documentation

This chapter provide an overview of the functionality provided by the interfaces and classes. It does not in detail document every interface and class. Detailed documentation can be found in the class library documentation, which exists in the form of

- Online documentation
- Assembly documentation xml-file: DHI.PFS.xml

The online class library documentation for PFS can be accessed via the MIKE SDK documentation index.

The assembly documentation file is located together with the assembly in the MIKE SDK bin folder. The assembly documentation xml file is used by IDE's like Visual Studio or SharpDevelop, making the documentation available within the IDE. The documentation is also available in the Visual Studio Object Browser, when loading the

`DHI.PFS.dll`.

It requires that the assembly documentation xml file is available together with the assembly file.

Examples

For examples, look in the examples which can be downloaded from GitHub. A number of C# examples are included in C# in the file:

Examples\CSharp\ExamplesPFS.cs

5.1.1 PFSFile

The `PFSFile` is used for reading and navigating the data in the PFS file. From a `PFSFile` object it is possible to get targets, sub-sections, keywords and parameters. It also enables all kinds of modifications to a PFS file.

- Requesting and searching for sections and keywords by name and number.
- Extracting values from parameters.
- Modifying section and keyword names
- Modifying parameter values
- Inserting sections, keywords and parameters
- Deleting sections, keywords and parameters.

5.1.2 PFSBuilder

The `PFSBuilder` is used for creating a new PFS file in a line-by-line manner.

5.1.3 PFSTokenReader

The `PFSTokenReader` is a class for reading only of PFS files. It provides a low level reader for fast, forward-only access to PFS data, with minimal memory consumption while reading.

It is useful when handling large PFS files where a huge number of data needs to be extracted.

6 Using the .NET API

6.1 Creating a Project in Visual Studio or SharpDevelop

This section describes how to make a small console program in Visual Studio or #develop (SharpDevelop) that loads a dfs file. The procedure for the two developer environments are very similar

Prerequisites:

- A MIKE Powered by DHI product that includes the `DHI.Generic.MikeZero.DFS`
- .NET 4.0 or later.
- Visual Studio, the free Express version should suffice, or #develop.

To create and setup a project in Visual Studio:

1. Open Visual Studio

2. a) Visual Studio: Select 'File' – 'New' – 'Project' and select a 'Visual C#', 'Windows', 'Console Application'. Find an appropriate name and location for your project.
   b) SharpDevelop: Select 'File' – 'New' – 'Solution'. Select 'C#', 'Windows Applications', 'Console Application'. Find an appropriate name and location for your project.

3. Right click on the 'References' and select 'Add Reference'. Select the 'Browse' tab. Go to the MIKE SDK bin folder, which is usually located in the default DHI installation folder under Program Files (x86)

   Find and locate the file `DHI.Generic.MikeZero.DFS` …

   Select it and press ok.

4. Do the same for

   ```
   DHI.Generic.MikeZero.EUM
   ```

Now the project is ready. It has created a 'Program.cs' file containing a Program class with a Main function. When adding the appropriate using directives, the DFS API is readily available.

Example, modify the 'Program.cs', to contain:

```
using System;
using DHI.Generic.MikeZero.DFS;
using DHI.Generic.MikeZero.DFS.dfs123;

namespace ConsoleApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      Dfs2File dfs2File = DfsFileFactory.Dfs2FileOpen(@'c:\Work\test\MyFile.dfs2');

      IDfsAxisEqD2 axis = (IDfsAxisEqD2)dfs2File.SpatialAxis;
      Console.Out.WriteLine('Size of grid: {0} x {1}', axis.XCount, axis.YCount);
    }
  }
}
```

This will print out the size of a 2D equidistant axis.

6.2 Using the Command Prompt

It is possible to compile the above Program.cs without Visual Studio. In a command prompt, locate the folder with the Program.cs file and issue:

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe
  /r:"C:\Program Files (x86)\DHI\MIKE SDK\2021\bin\DHI.Generic.MikeZero.DFS.dll"
  /r:"C:\Program Files (x86)\DHI\MIKE SDK\2021\bin\DHI.Generic.MikeZero.EUM.dll"
  /platform:x86
  Program.cs
```

Paths may need to be updated. That will build a Program.exe that can be run. The platform argument is only required if you want to build an application that always runs in 32 bit mode.

Such a command can be issued on any Windows computer with .NET installed.

APPENDICES

A Modify Times

To ease the use of the temporal axes, when the temporal axes properties are returned by DFS, these can be modified as follows:

- Equidistant time: No modifications take place.
- Non-equidistant time: The start time offset is stored in the file as the time of the first time-step. When reading the file again, the start time offset provides the time of the first time step, while the time of all time steps are subtracted the start time offset such that the first time step returns time 0.
- Equidistant calendar: Start time offset and start date-time is stored in the file as specified. When reading the file again, the start time offset is returned as 0, its value being added to the start date time. If the start time offset contains fractions of seconds, these are lost.
- Non-equidistant calendar: The start time offset is stored in the file as the time of the first time-step. When reading the file again, the start time offset is returned as zero, its value being added to the start date time, and all time step times are subtracted the start time offset such that the first time step returns time 0. If the start time offset (the time of the first time step) contains fractions of seconds, these are lost.

If using the .NET API, use the `IDfsParameters` to specify whether times should be modified, see section 3.7. The default in the .NET API is false.

B Storing Unicode Strings as Items

String data in the form of Unicode strings can as such not be stored as data in a dynamic or static item. A Unicode string as is used within .NET consists of 16 bit characters, and needs to be converted into a series of bytes (8 bit).

You can store strings in ANSI format (which is the default MIKE ZERO string type), by converting them to/from a byte string using the default ANSI encoder. The following two lines show how to encode and decode ANSI strings into bytes:

```
byte[] ansiBytes = System.Text.Encoding.Default.GetBytes('MyString');
string mystring  = System.Text.Encoding.Default.GetString(ansiBytes);
```

The ANSI format depends on the computer at hand; you cannot transfer all characters from a Danish computer to a French computer. That is a general problem for all strings stored in the DFS files, as also for e.g., the item names and file title. Hence special characters as the Danish æ, ø and å may not be handled correctly.

When storing a string as the data of a static (or dynamic) item, you can also convert the string to UTF-8, which can hold all type of characters that can be stored in a C# string as bytes. That will make sure that you get the very same string back, that you actually wrote, independent of the computer that read/writes the data.

```
byte[] unibytes = System.Text.Encoding.UTF8.GetBytes('MyString');
string mystring = System.Text.Encoding.UTF8.GetString(unibytes);
```

C DFS and COM

The DHI.Generic.MikeZero.DFS assembly is registered as a COM interop assembly, and is accessible through COM. All the interfaces and the major classes are COM enabled.

There are two issues when using a .NET component through COM:

1. COM does not support generic types.
2. COM from .NET does not support interface inheritance.

There are a number of generic lists in the interfaces that cannot be accessed using COM. An example is the list of dynamic items in the `IDfsFile`. In .NET it is declared as:

```
IList<IDfsDynamicItemInfo> ItemInfo { get; }
```

All these generic lists are made available using the `DfsComHelper` class. For every generic list, the `DfsComHelper` has a method that makes the list available as a non-generic list. Using the example above:

```
public IList GetItemInfoDfs(IDfsFile dfsfile)
```

See the documentation on the `DfsComHelper` for which methods are made available.

The second issue is more an inconvenience. The lack of support for interface inheritance means that you need to query and cast to a base interface in order to use functionality from the base interface.

Example: The `IDfsFile` interface extends the `IDfsFileIO` and `IDfsFileStaticIO`. In order to get the `IDfsFileIO` functionality from the `IDfsFile` interface, you need to query `IDfsFile` for the `IDfsFileIO` interface and have another variable containing a reference to that interface as well, and similar for the `IDfsFileStaticIO` interface. Hence you would need 3 references of different type to the same object to get full functionality.

D DFS in Matlab

Since Matlab R2011a, Matlab on Windows has support for .NET version 4.0. You can load and access the DFS classes and interfaces exactly as defined in the .NET APIs.

There are no limitations with regards to lists or generic types. Matlab converts an object returned by any method or property to the class of the object (even if the method returns an interface). This can imply a performance hit. If it does so, make sure to make as few calls to the .NET components as possible.

One important difference between Matlab and .NET: Matlab uses 1-based indexing while .NET uses 0 based indexing: The first element in a Matlab array is at index 1, while the first element in a .NET array is at index 0.

To load the DFS assembly and make the DFS classes available:

```
NET.addAssembly('DHI.Generic.MikeZero.DFS');
import DHI.Generic.MikeZero.DFS.*;
```

The import statement is not required, but eliminates the need to write fully qualified names for all classes and interfaces in the `DHI.Generic.MikeZero.DFS` namespace. However, if you experience problems finding the DFS classes and methods, try writing the fully qualified name everywhere (i.e. prepend DFS class names with `DHI.Generic.MikeZero.DFS`). This problem has been experienced for some GUI functions and when using the Matlab compiler.

To convert data arrays from .NET types to Matlab, you can use the `single` or `double` methods from Matlab. Example; if data is a 1D `System.Single[]`, or a 2D `System.Single[,]` then

```
a = single(data);
b = double(data);
```

will give a 1D or 2D matrix in Matlab of type single or double.

To convert arrays from the Matlab matrix types to .NET types:

```
aNet = NET.convertArray(a);
aNet = NET.convertArray(a, 'System.Double', 4);
```

The `convertArray` method will make .NET type arrays that can be used in .NET methods with array arguments. The latter specifies the type and the size, while the former infers the type and the size from the Matlab matrix type and size.

To convert strings between .NET string and Matlab strings, use:

```
strNet = System.String('a .NET System.String');
str = char(strNet);
```

Matlab supports extension methods, though only as static methods.

Example; invoking an extension method in .NET

```
DateTime[] times = dfs2File.FileInfo.TimeAxis.GetDateTimes();
```

In Matlab the similar method is called as a static method.

```
times = DfsExtensions.GetDateTimes(dfs2.FileInfo.TimeAxis);
```

Below is an example that loads a dfs2 file and modifies data for the first item and time step.

```
% Load assembly and import namespace
NET.addAssembly('DHI.Generic.MikeZero.DFS');
import DHI.Generic.MikeZero.DFS.*;

% Open dfs2 file for editing (Dfs2File)
dfs2 = DfsFileFactory.Dfs2FileOpenEdit(copy_OresundHD.dfs2');

% Read first item and time step (DfsItemData2D<System*Single>)
data2D = dfs2.ReadItemTimeStepNext;

% Convert to a 2D matlab array and modify the data
b = double(data2D.To2DArray);
b = b+10;

% Write data back to the file and close it.
dfs2.WriteItemTimeStep(1,0,0,NET.convertArray(single(b(:))))
dfs2.Close
```

Note that the DFS system stores matrix data in row-major order (values in x-direction first), while Matlab stores matrix data in column-major order (values in y-direction first). To get 2D data printed correctly in plots, it is required to transpose the data matrix. For 3D data, the x-y dimension must also be swapped.

E DFS in Matlab through COM

For versions of Matlab not having support for .NET, the DFS library can be accessed through COM. There are, however, a number of limitations in the Matlab COM support that makes life somewhat complicated.

1. Matlab only support COM interface querying on objects, not on interfaces. This means that if your object implements several interfaces, you cannot cast between them directly in Matlab.

   Example, every DFS file object implements the interfaces `IDfsFile`, `IDfsFileIO` and `IDfsFileStaticIO`. If you in Matlab get hold of the DFS file object as an `IDfsFile`, then you cannot cast it to `IDfsFileIO` or `IDfsFileStaticIO`.

   To accommodate this, a converter class has been created, called `DfsTypeConverter`. This contains a number of methods that translates one interface to another interface, on the form.

   ```
   public IDfsFileIO DfsFileIO(IDfsFile dfsFile)
   ```

   See documentation on the `DfsTypeConverter` for a list of what is provided.

2. When Matlab passes array data to a COM method, it does it by default as a 2D array. To pass it as a 1D vector, it is required to set the feature:

   ```
   COM_SafeArraySingleDim
   ```

   This can be set and reset by the commands:

   ```
   feature('COM_SafeArraySingleDim', 1)
   feature('COM_SafeArraySingleDim', 0)
   ```

3. COM DateTime objects are not well supported in Matlab; the object is accessible only as a string, which must be parsed. The format of the string depends on the locale on the computer, and will differ from country to country (and differs from the locale used in the datevec method, which cannot be used without a user specified formatting string). The format also depends on the actual date and time in the string.

   The `DfsTypeConverter` has a method returning an array of year, month, day, hours, minutes and seconds from a date time object:

   ```
   double[] GetDateTimeVector(System.DateTime time);
   ```

If that does not work, you can try something like (format strings depending on your local settings):

```
if (length(timestr) <= 8)
  datetime = datevec(timestr,'HH:MM:SS');
elseif (length(timestr) <= 10)
  datetime = datevec(timestr,'dd-mm-yyyy');
elseif (length(timestr) <= 12)
  datetime = datevec(timestr,'HH:MM:SS.FFF');
elseif(length(timestr) <= 19)
  datetime = datevec(timestr,'dd-mm-yyyy HH:MM:SS');
else
  datetime = datevec(timestr,'dd-mm-yyyy HH:MM:SS.FFF');
end
```

To create a COM date time in Matlab, you can use;

```
COM.Date(2000,12,31,16,30,00)
```

If the `DfsTypeConverter` lacks methods for certain conversions, it is possible to create and build your own assembly providing the required conversion. Remember to COM enable the assembly.

Below is an example of how to load a dfs2 file, update data of one time step and closing the file again.

```
% Create a file factory and a type converter
filefactory = actxserver('DHI.Generic.MikeZero.DFS.DfsFileFactory');
converter = actxserver('DHI.Generic.MikeZero.DFS.DfsTypeConverter');

% Open a dfs2 file (IDfs2File)
dfs2 = filefactory.Dfs2FileOpenEdit('test_OresundHD.dfs2')

% convert the IDfs2File interface to a IDfsFileIO
dfs2io = converter.Dfs2FileIO(dfs2)

% Read 2D data (IDfsItemData2D)
data2d = dfs2.ReadItemTimeStepNext();

% Convert it to a float[,] and update data
d = converter.GetFloatMatrix(data2d);
d = d - 10;

% Write data back to the file and close it
feature('COM_SafeArraySingleDim', 1)
converter.WriteItemTimeStepFloat(dfs2io, 1, 0, 0, d(:));
feature('COM_SafeArraySingleDim', 0)
dfs2io.Close()
```

F Description of the Item Value Types

The value type specifies specify how each data value of an item is to be interpreted in time.

- **Instantaneous**: The values represent the item at the exact times specified. For example, data from an instant measuring devices, as e.g. the wind velocity in [m/s]. Linear variation between the time steps is usually assumed, i.e. to get values in between time steps, linear interpolation can be used.
- **Accumulated**: The values represent successive accumulation of the item over time, from the start time of the file to the current time. For example, the rainfall accumulated over a year in [mm], and recorded every day. Linear variation between the time steps is usually assumed, i.e. to get values in between time steps, linear interpolation can be used.
- **Step Accumulated**: The values represent accumulation of the item over one time step, from the previous time step time to the current time step. For example, rainfall measured for a year, recording for every day the amount of rain in [mm] falling that day.
- **Mean Step Backward**: The values represent the item from the previous time step to the current time step. For example, the average rain rate between the last and the current time step. Also called “mean step accumulated”.
- **Mean Step Forward**: The values represent the item from the current time step to the next time step. Also called “reverse mean step accumulated”.

The step accumulated is often used in the following context. Say that we start measuring rainfall at 10:00:00. At 11:00:00 we take the measuring device, register the amount of rain it has collected, say it has the value of 10, and empties the device. At 12:00:00 the same process takes place, say with the value of 15, and so on every hour. In a time series we will have the value 10 at time step 11:00:00 and the value 15 at time step 12:00:00 and so on. Values represent the accumulation of the item over a time span covering from the previous time step to the current time step.

The accumulated, step accumulated, and mean step forward and backward value types can represent the exact same data. The Instantaneous value type cannot be directly converted forth and back between the other value types.

Table F. 1 shows how to represent the same rain time series with the 4 different equivalent value types. Note that they do not have the same EUM type and unit.
You convert data from step accumulated values to mean step backward values by dividing the step accumulated value with the time for the time step. Example, from 18:10:00 to 18:12:00 there was 0.36 mm = 360 μm of rain within 120 seconds, giving 3 $\mu m/s$ in that period.

The forward and backward step values differ by being switched one place in the table.

Table F. 1 Values for a rain measurement time series for different value types, representing the exact same rain

Time	Accumulated	Step accumulated	Mean step backward	Mean step forward
	[mm]	[mm]	[µm/s]	[µm/s]
18:00:00	0	0	0	0
18:10:00	0	0	0	3
18:12:00	0.36	0.36	3	7
18:15:00	1.62	1.26	7	12
18:16:00	2.34	0.72	12	14
18:20:00	5.70	3.36	14	42
18:22:00	10.74	5.04	42	10
18:24:00	11.94	1.20	10	20
18:27:00	15.54	3.60	20	18
18:28:00	16.62	1.08	18	14
18:29:00	17.46	0.84	14	10
18:32:00	19.26	1.80	10	4
18:34:00	19.74	0.48	4	5
18:40:00	21.54	1.80	5	1
18:47:00	21.96	0.42	1	0
19:00:00	21.96	0	0	0

A graphical interpretation of the different value types is shown in Figure F. 1 - Figure F. 5.

The instantaneous version has the same values as the forward step from Table F. 1.

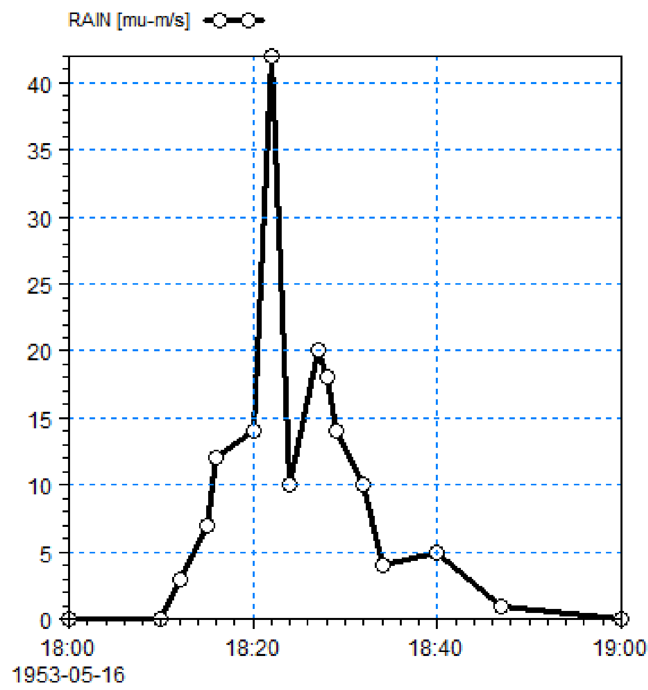


Figure F. 1 Instantaneous value type time series: Points are connected by lines.

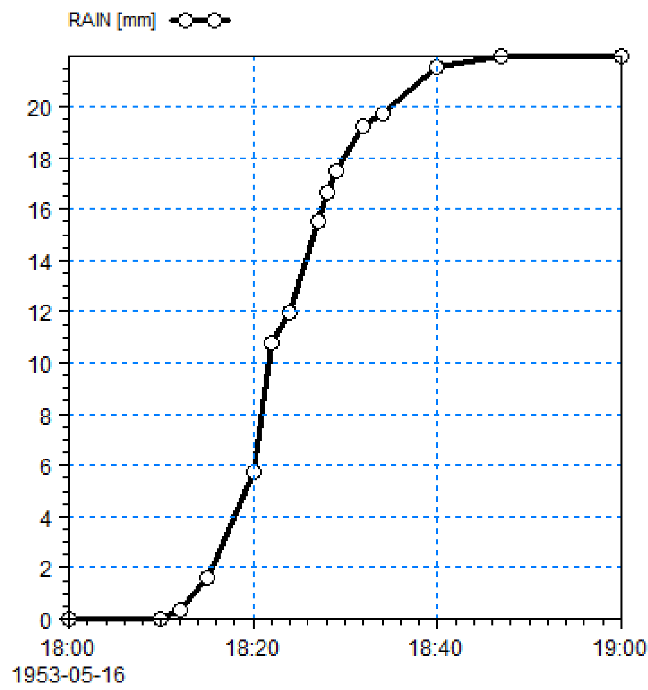


Figure F. 2 Accumulated value type: An Accumulated time series shall always be increasing

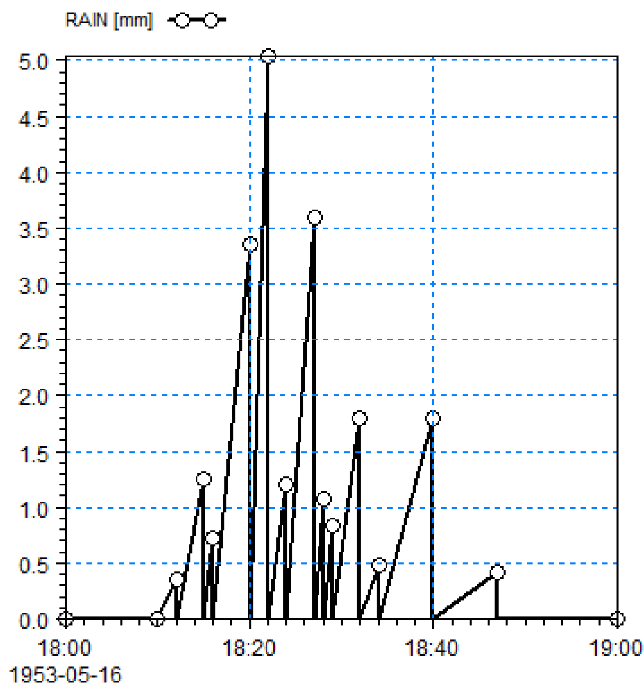


Figure F. 3 Step accumulated value type in a time series: A line is drawn from the x-axis at the previous time step to the data point

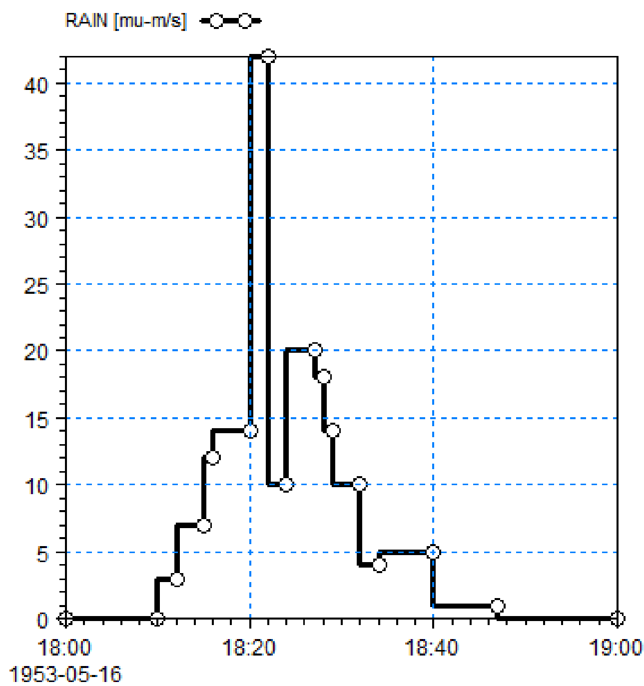


Figure F. 4 Mean step backward value type for a time series: A line is drawn from the previous time step to the current time step with the value at current time step

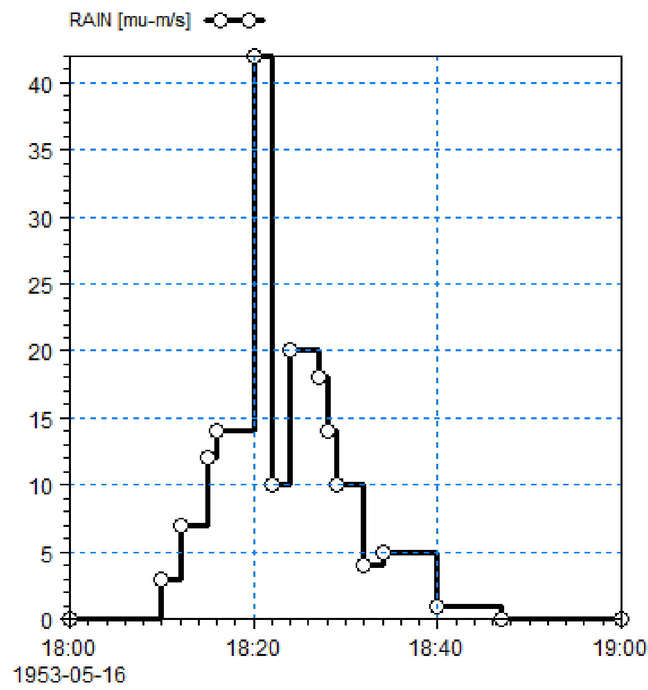


Figure F. 5 Mean step forward value type for a time series: A line is drawn from the current time step to the next time step with the value at the current time step

G DFS Data Type Tags Currently Used

Data type tag	Data description	DFS file type	Product suite name
0	Generic data for general use	dfs0+1+2	MIKE Zero
-1	MIKE 21 HD output (H only)	dfs2	MIKE 21
1	M21TRN output	dfs1+2	MIKE 21
1	MIKE 21 HD simulation output (H,P,Q)	dfs1+2	MIKE 21
2	M21TRA output	dfs1+2	MIKE 21
2	MIKE 21 AD,EU,ME,MT,WQ hot start data	dfs1+2	MIKE 21
2	MIKE 21 AD,EU,ME,MT,WQ simulation output	dfs1+2	MIKE 21
3	MIKE 3 HD, AD, EU, ME, MT, WQ simulation output	dfs0+1+2+3	MIKE 3
4	MIKE 21 ST output	dfs2	MIKE 21
11	MIKE 21 HD hot start data	dfs2	MIKE 21
31	MIKE 3 ACM hot start data	dfs3	MIKE 3
32	MIKE 3 ACM input data (HD) for a AD standalone simulation	dfs3	MIKE 3
33	MIKE 3 HD, AD, EU, ME, MT, WQ simulation output (vertical plane)	dfs2	MIKE 3
34	MIKE 3 HD structure output	dfs0	MIKE 3
35	MIKE 3 HS hot start data	dfs3	MIKE 3
36	MIKE 3 HS input data (HD) for a AD standalone simulation	dfs3	MIKE 3
51	M21NPA output hot data	dfs0	MIKE 21
51	M21PA output hot data	dfs0	MIKE 21
61	M21SA output hot data	dfs0	MIKE 21
71	M21RMS output	dfs2	MIKE 21
101	Cross-shore profile	dfs1	LITPACK
102	Timeseries wave climate	dfs0	LITPACK
103	Event duration wave climate	dfs2	LITPACK
104	Time series for LITSTP	dfs0	LITPACK
105	Coastline alignment	dfs1	LITPACK
106	Cross-section of trench	dfs1	LITPACK
107	Wave events for LITPROF	dfs0	LITPACK
108	Source file for LITLINE	dfs0	LITPACK
109	Wave events for LITTREN	dfs0	LITPACK
111	LITDRIFT output (Littoral current)	dfs1	LITPACK
112	LITDRIFT output (Littoral drift)	dfs1	LITPACK
113	LITDRIFT output (Littoral budget)	dfs0+1	LITPACK
114	LITDRIFT output (Budget rose))	dfs0	LITPACK
116	LITSTP output (single event)	dfs0	LITPACK
117	LITSTP output (single event, extended output)	dfs0+1	LITPACK

Data type tag	Data description	DFS file type	Product suite name
118	LITSTP output (time series events)	dfs0+1	LITPACK
120	LITPROF output	dfs1	LITPACK
121	LITLINE output	dfs1	LITPACK
122	LITTREN output	dfs1	LITPACK
217	CONTROL SIGNAL FILE	dfs1	Wave Syntheziser
217	WS Wave generated control signals (WS_WG_DATA)	dfs1	Wave Syntheziser
218	WS_DAQ_DATA	unknown	Wave Syntheziser
219	WS data acquired by WS controller component (WS_DAQ_DATA_RECALIBRATED)	unknown	Wave Syntheziser
220	WS_AWACS_DATA	unknown	Wave Syntheziser
221	WS_AWACS_REFLECTION_DATA	unknown	Wave Syntheziser
222	WS_SPECTRAL_DATA	unknown	Wave Syntheziser
223	WS_CROSSING_DATA	unknown	Wave Syntheziser
224	WS_REFLECTION_DATA	unknown	Wave Syntheziser
225	WS_FILTERING_DATA	unknown	Wave Syntheziser
230	WS_DIRECTIONAL_DATA	unknown	Wave Syntheziser
240	WS_MATLAB_DATA	unknown	Wave Syntheziser
250	WS_NORTEK_ADV_DATA	unknown	Wave Syntheziser
800	Pier data for MIKE 21 HD	dfs1	MIKE 21
901	ADCP current data for ADCPLT	dfs1	MIKE Zero
930	Radiation stresses	dfs2	MIKE 21
2000	Flow Model FM, Finite Element data file + mesh data	dfsu	MIKE FM
2001	Flow Model FM, Finite Volume data file	dfsu	MIKE FM
2002	Spectral Wave Model FM, Finite Element data file	dfsu	MIKE FM
2003	Spectra Wave Model FM, Finite Volume data file	dfsu	MIKE FM
3000	Layer data from GEO Model	dfs2	MIKE GEOModel
10001	Output from Grd2Mike tool	dfs2	MIKE Zero
10001	MIKE SHE general 2D result file (all model cells)	dfs2	MIKE SHE
10002	MIKE SHE 2D UZ result file (UZ cells), static items containing mapping from UZ cell to model cell	dfs2	MIKE SHE
10003	MIKE SHE 3D UZ result file (UZ nodes, UZ cells), static items containing mapping from UZ cell to model cell	dfs3	MIKE SHE
10004	MIKE SHE 3D SZ Flow result file (all model cells X SZ layers), static items containing hydro-geologic properties	dfs3	MIKE SHE
10005	MIKE SHE 3D SZ "non-Flow" result file (all model cells X SZ layers), static items containing hydro-geologic properties	dfs3	MIKE SHE
10006	MIKE SHE 3D SZ "others" result file (all model cells X SZ layers), without hydro-properties as static items	dfs3	MIKE SHE

INDEX

Index

.NET namespace 17, 34
32/64 bit 18
Accumulated 51
Application title 5
Application version number 5
Associated static item 9
Class library documentation 17, 34
Compression 14
Coordinate Systems 13
Curve-linear axes 12
Custom Blocks 12
Data converters 23
Data type tag 5, 57
Delete value 5
DFS Items 6, 21
DFS Parameters 23
Dfs0 27
Dfs1 27
Dfs2 28
Dfs3 28
DfsSimpleType 7
Dfsu 30
Dynamic items 8, 21
Element based 10
Enable Plugin 23
Encoding keys 14
Equidistant axes 11
Equidistant calendar 9
Equidistant time 9
EUM 6
EUM quantity 6
Examples 17, 34
File title 5
Free conversion 7
Geographical coordinate system 13
Header 5, 21
Instantaneous 51
Item axis unit conversion 7
Item statistics 15
Item unit conversion 7
Map Projection 13
Matlab 47
Mean Step Backward 51
Mean Step Forward 51
Modify Times 23, 41
Node based 10
Non-equidistant axes 11
Non-equidistant calendar 10
Non-equidistant time 9
Projected coordinate system 13
Reference coordinates 7
Reference orientation 7
Spatial Axes 10, 23

Static items 8, 19, 21
Statistics 15
Step Accumulated 51
Temporal Axes 9, 22
Type of data 7
UBG 7
UBG conversion 7, 23
User defined coordinate system 13
Value type 8, 51
Visual Studio 37