

ARC Data Library libarcdata

Generated by Doxygen 1.6.1

Wed Oct 24 16:26:17 2012



# Contents

<b>1</b>	<b>Summary of libarcdata</b>	<b>1</b>
<b>2</b>	<b>Data Structure Index</b>	<b>3</b>
2.1	Class Hierarchy . . . . .	3
<b>3</b>	<b>Data Structure Index</b>	<b>5</b>
3.1	Data Structures . . . . .	5
<b>4</b>	<b>Data Structure Documentation</b>	<b>7</b>
4.1	Arc::CacheParameters Struct Reference . . . . .	7
4.1.1	Detailed Description . . . . .	7
4.2	Arc::DataBuffer Class Reference . . . . .	8
4.2.1	Detailed Description . . . . .	9
4.2.2	Constructor & Destructor Documentation . . . . .	9
4.2.2.1	DataBuffer . . . . .	9
4.2.2.2	DataBuffer . . . . .	9
4.2.3	Member Function Documentation . . . . .	9
4.2.3.1	add . . . . .	9
4.2.3.2	buffer_size . . . . .	10
4.2.3.3	checksum_object . . . . .	10
4.2.3.4	checksum_valid . . . . .	10
4.2.3.5	eof_read . . . . .	10
4.2.3.6	eof_read . . . . .	10
4.2.3.7	eof_write . . . . .	10
4.2.3.8	eof_write . . . . .	10
4.2.3.9	error . . . . .	10
4.2.3.10	error_read . . . . .	10
4.2.3.11	error_write . . . . .	11
4.2.3.12	for_read . . . . .	11

4.2.3.13	for_read	11
4.2.3.14	for_write	11
4.2.3.15	for_write	11
4.2.3.16	is_notwritten	12
4.2.3.17	is_notwritten	12
4.2.3.18	is_read	12
4.2.3.19	is_read	12
4.2.3.20	is_written	12
4.2.3.21	is_written	13
4.2.3.22	set	13
4.2.3.23	wait_any	13
4.3	Arc::DataCallback Class Reference	14
4.3.1	Detailed Description	14
4.4	Arc::DataHandle Class Reference	15
4.4.1	Detailed Description	15
4.4.2	Member Function Documentation	17
4.4.2.1	GetPoint	17
4.5	Arc::DataMover Class Reference	18
4.5.1	Detailed Description	18
4.5.2	Member Function Documentation	18
4.5.2.1	checks	18
4.5.2.2	checks	19
4.5.2.3	force_to_meta	19
4.5.2.4	secure	19
4.5.2.5	set_default_max_inactivity_time	19
4.5.2.6	set_default_min_average_speed	19
4.5.2.7	set_default_min_speed	19
4.5.2.8	Transfer	19
4.5.2.9	Transfer	20
4.5.2.10	verbose	20
4.6	Arc::DataPoint Class Reference	21
4.6.1	Detailed Description	23
4.6.2	Member Enumeration Documentation	24
4.6.2.1	DataPointAccessLatency	24
4.6.2.2	DataPointInfoType	24
4.6.3	Constructor & Destructor Documentation	25

4.6.3.1	DataPoint	25
4.6.4	Member Function Documentation	25
4.6.4.1	AddChecksumObject	25
4.6.4.2	AddLocation	25
4.6.4.3	AddURLOptions	25
4.6.4.4	Check	25
4.6.4.5	CompareLocationMetadata	26
4.6.4.6	CompareMeta	26
4.6.4.7	CreateDirectory	26
4.6.4.8	CurrentLocationMetadata	26
4.6.4.9	FinishReading	26
4.6.4.10	FinishWriting	26
4.6.4.11	GetFailureReason	27
4.6.4.12	List	27
4.6.4.13	NextLocation	27
4.6.4.14	Passive	27
4.6.4.15	PostRegister	27
4.6.4.16	PrepareReading	28
4.6.4.17	PrepareWriting	28
4.6.4.18	PreRegister	28
4.6.4.19	PreUnregister	29
4.6.4.20	ProvidesMeta	29
4.6.4.21	Range	29
4.6.4.22	ReadOutOfOrder	29
4.6.4.23	Registered	29
4.6.4.24	Resolve	29
4.6.4.25	Resolve	30
4.6.4.26	SetAdditionalChecks	30
4.6.4.27	SetMeta	30
4.6.4.28	SetSecure	30
4.6.4.29	SetURL	30
4.6.4.30	SortLocations	31
4.6.4.31	StartReading	31
4.6.4.32	StartWriting	31
4.6.4.33	Stat	31
4.6.4.34	Stat	32

4.6.4.35	StopReading	32
4.6.4.36	StopWriting	32
4.6.4.37	TransferLocations	32
4.6.4.38	Unregister	32
4.6.4.39	WriteOutOfOrder	33
4.6.5	Field Documentation	33
4.6.5.1	valid_url_options	33
4.7	Arc::DataPointDirect Class Reference	34
4.7.1	Detailed Description	35
4.7.2	Member Function Documentation	35
4.7.2.1	AddChecksumObject	35
4.7.2.2	AddLocation	35
4.7.2.3	CompareLocationMetadata	35
4.7.2.4	CurrentLocationMetadata	35
4.7.2.5	NextLocation	36
4.7.2.6	Passive	36
4.7.2.7	PostRegister	36
4.7.2.8	PreRegister	36
4.7.2.9	PreUnregister	36
4.7.2.10	ProvidesMeta	37
4.7.2.11	Range	37
4.7.2.12	ReadOutOfOrder	37
4.7.2.13	Registered	37
4.7.2.14	Resolve	37
4.7.2.15	SetAdditionalChecks	37
4.7.2.16	SetSecure	38
4.7.2.17	SortLocations	38
4.7.2.18	Unregister	38
4.7.2.19	WriteOutOfOrder	38
4.8	Arc::DataPointIndex Class Reference	39
4.8.1	Detailed Description	40
4.8.2	Member Function Documentation	40
4.8.2.1	AddChecksumObject	40
4.8.2.2	AddLocation	40
4.8.2.3	Check	40
4.8.2.4	CompareLocationMetadata	41

4.8.2.5	CurrentLocationMetadata	41
4.8.2.6	FinishReading	41
4.8.2.7	FinishWriting	41
4.8.2.8	NextLocation	41
4.8.2.9	Passive	41
4.8.2.10	PrepareReading	42
4.8.2.11	PrepareWriting	42
4.8.2.12	ProvidesMeta	42
4.8.2.13	Range	42
4.8.2.14	ReadOutOfOrder	43
4.8.2.15	Registered	43
4.8.2.16	SetAdditionalChecks	43
4.8.2.17	SetSecure	43
4.8.2.18	SortLocations	43
4.8.2.19	StartReading	43
4.8.2.20	StartWriting	44
4.8.2.21	StopReading	44
4.8.2.22	StopWriting	44
4.8.2.23	TransferLocations	44
4.8.2.24	WriteOutOfOrder	44
4.9	Arc::DataPointLoader Class Reference	46
4.9.1	Detailed Description	46
4.10	Arc::DataPointPluginArgument Class Reference	47
4.10.1	Detailed Description	47
4.11	Arc::DataSpeed Class Reference	48
4.11.1	Detailed Description	48
4.11.2	Constructor & Destructor Documentation	48
4.11.2.1	DataSpeed	48
4.11.2.2	DataSpeed	49
4.11.3	Member Function Documentation	49
4.11.3.1	hold	49
4.11.3.2	set_base	49
4.11.3.3	set_max_data	49
4.11.3.4	set_max_inactivity_time	49
4.11.3.5	set_min_average_speed	50
4.11.3.6	set_min_speed	50

4.11.3.7	set_progress_indicator . . . . .	50
4.11.3.8	transfer . . . . .	50
4.11.3.9	verbose . . . . .	50
4.11.3.10	verbose . . . . .	50
4.12	Arc::DataStatus Class Reference . . . . .	51
4.12.1	Detailed Description . . . . .	51
4.12.2	Member Enumeration Documentation . . . . .	51
4.12.2.1	DataStatusType . . . . .	51
4.13	Arc::FileCache Class Reference . . . . .	53
4.13.1	Detailed Description . . . . .	53
4.13.2	Constructor & Destructor Documentation . . . . .	54
4.13.2.1	FileCache . . . . .	54
4.13.2.2	FileCache . . . . .	54
4.13.2.3	FileCache . . . . .	54
4.13.3	Member Function Documentation . . . . .	55
4.13.3.1	AddDN . . . . .	55
4.13.3.2	CheckCreated . . . . .	55
4.13.3.3	CheckDN . . . . .	55
4.13.3.4	CheckValid . . . . .	55
4.13.3.5	File . . . . .	55
4.13.3.6	GetCreated . . . . .	56
4.13.3.7	GetValid . . . . .	56
4.13.3.8	Link . . . . .	56
4.13.3.9	Release . . . . .	56
4.13.3.10	SetValid . . . . .	57
4.13.3.11	Start . . . . .	57
4.13.3.12	Stop . . . . .	57
4.13.3.13	StopAndDelete . . . . .	57
4.14	Arc::FileCacheHash Class Reference . . . . .	58
4.14.1	Detailed Description . . . . .	58
4.15	Arc::FileInfo Class Reference . . . . .	59
4.15.1	Detailed Description . . . . .	59
4.16	Arc::URLMap Class Reference . . . . .	60



# Chapter 1

## Summary of libarcdata

libarcdata is a library for data access. It provides a uniform interface to several types of grid storage and catalogs using various protocols. See the DataPoint inheritance diagram for a list of currently supported protocols. The interface can be used to read, write, list, transfer and delete data to and from storage systems and catalogs.

The library uses ARC's dynamic plugin mechanism to load plugins for specific protocols only when required at runtime. These plugins are called Data Manager Components (DMCs). The DataHandle class should be used to automatically load the required DMC at runtime. To create a new DMC for a protocol which is not yet supported see the instruction and examples in the DataPoint class documentation. This documentation also gives a complete overview of the interface.

The following protocols are currently supported in standard distributions of ARC (except XRootd, which is not yet distributed).

ARC (`arc://`) - Protocol to access the Chelonia storage system developed by ARC.

File (`file://`) - Regular local file system.

GridFTP (`gsiftp://`) - GridFTP is essentially the FTP protocol with GSI security. Regular FTP can also be used.

HTTP(S/G) (`http://`) - Hypertext Transfer Protocol. HTTP over SSL (HTTPS) and HTTP over GSI (HTTPG) are also supported.

LDAP (`ldap://`) - Lightweight Directory Access Protocol. LDAP is used in grids mainly to store information about grid services or resources rather than to store data itself.

LFC (`lfc://`) - The LCG File Catalog (LFC) is a replica catalog developed by CERN. It consists of a hierarchical namespace of grid files and each filename can be associated with one or more physical locations.

RLS (`rls://`) - The Replica Location Service (RLS) is a replica catalog developed by Globus. It maps filenames in a flat namespace to one or more physical locations, and can also store meta-information on each file.

SRM (`srm://`) - The Storage Resource Manager (SRM) protocol allows access to data distributed across physical storage through a unified namespace and management interface.

XRootd (`root://`) - Protocol for data access across large scale storage clusters. More information can be found at <http://xrootd.slac.stanford.edu/>



## Chapter 2

# Data Structure Index

### 2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Arc::CacheParameters . . . . .	7
Arc::DataBuffer . . . . .	8
Arc::DataCallback . . . . .	14
Arc::DataHandle . . . . .	15
Arc::DataMover . . . . .	18
Arc::DataPoint . . . . .	21
Arc::DataPointDirect . . . . .	34
Arc::DataPointIndex . . . . .	39
Arc::DataPointLoader . . . . .	46
Arc::DataPointPluginArgument . . . . .	47
Arc::DataSpeed . . . . .	48
Arc::DataStatus . . . . .	51
Arc::FileCache . . . . .	53
Arc::FileCacheHash . . . . .	58
Arc::FileInfo . . . . .	59
Arc::URLMap . . . . .	60



## Chapter 3

# Data Structure Index

### 3.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">Arc::CacheParameters</a> (Contains data on the parameters of a cache ) . . . . .	7
<a href="#">Arc::DataBuffer</a> (Represents set of buffers ) . . . . .	8
<a href="#">Arc::DataCallback</a> (This class is used by <a href="#">DataHandle</a> to report missing space on local filesystem ) . . . . .	14
<a href="#">Arc::DataHandle</a> (This class is a wrapper around the <a href="#">DataPoint</a> class ) . . . . .	15
<a href="#">Arc::DataMover</a> ( <a href="#">DataMover</a> provides an interface to transfer data between two DataPoints ) . . . . .	18
<a href="#">Arc::DataPoint</a> (A <a href="#">DataPoint</a> represents a data resource and is an abstraction of a URL ) . . . . .	21
<a href="#">Arc::DataPointDirect</a> (This is a kind of generalized file handle ) . . . . .	34
<a href="#">Arc::DataPointIndex</a> (Complements <a href="#">DataPoint</a> with attributes common for Indexing Service URLs ) . . . . .	39
<a href="#">Arc::DataPointLoader</a> (Class used by <a href="#">DataHandle</a> to load the required DMC ) . . . . .	46
<a href="#">Arc::DataPointPluginArgument</a> (Class representing the arguments passed to DMC plugins ) . . . . .	47
<a href="#">Arc::DataSpeed</a> (Keeps track of average and instantaneous transfer speed ) . . . . .	48
<a href="#">Arc::DataStatus</a> (Status code returned by many <a href="#">DataPoint</a> methods ) . . . . .	51
<a href="#">Arc::FileCache</a> ( <a href="#">FileCache</a> provides an interface to all cache operations ) . . . . .	53
<a href="#">Arc::FileCacheHash</a> ( <a href="#">FileCacheHash</a> provides methods to make hashes from strings ) . . . . .	58
<a href="#">Arc::FileInfo</a> ( <a href="#">FileInfo</a> stores information about files (metadata) ) . . . . .	59
<a href="#">Arc::URLMap</a> . . . . .	60



## Chapter 4

# Data Structure Documentation

### 4.1 Arc::CacheParameters Struct Reference

Contains data on the parameters of a cache.

```
#include <FileCache.h>
```

#### 4.1.1 Detailed Description

Contains data on the parameters of a cache.

The documentation for this struct was generated from the following file:

- FileCache.h

## 4.2 Arc::DataBuffer Class Reference

Represents set of buffers.

```
#include <DataBuffer.h>
```

### Data Structures

- struct **buf\_desc**
- class **checksum\_desc**

### Public Member Functions

- [operator bool](#) () const
- [DataBuffer](#) (unsigned int size=65536, int blocks=3)
- [DataBuffer](#) (Checksum \*cksum, unsigned int size=65536, int blocks=3)
- [~DataBuffer](#) ()
- [bool set](#) (Checksum \*cksum=NULL, unsigned int size=65536, int blocks=3)
- [int add](#) (Checksum \*cksum)
- [char \\* operator\[\]](#) (int n)
- [bool for\\_read](#) (int &handle, unsigned int &length, bool wait)
- [bool for\\_read](#) ()
- [bool is\\_read](#) (int handle, unsigned int length, unsigned long long int offset)
- [bool is\\_read](#) (char \*buf, unsigned int length, unsigned long long int offset)
- [bool for\\_write](#) (int &handle, unsigned int &length, unsigned long long int &offset, bool wait)
- [bool for\\_write](#) ()
- [bool is\\_written](#) (int handle)
- [bool is\\_written](#) (char \*buf)
- [bool is\\_notwritten](#) (int handle)
- [bool is\\_notwritten](#) (char \*buf)
- [void eof\\_read](#) (bool v)
- [void eof\\_write](#) (bool v)
- [void error\\_read](#) (bool v)
- [void error\\_write](#) (bool v)
- [bool eof\\_read](#) ()
- [bool eof\\_write](#) ()
- [bool error\\_read](#) ()
- [bool error\\_write](#) ()
- [bool error\\_transfer](#) ()
- [bool error](#) ()
- [bool wait\\_any](#) ()
- [bool wait\\_used](#) ()
- [bool wait\\_for\\_read](#) ()
- [bool wait\\_for\\_write](#) ()
- [bool checksum\\_valid](#) () const
- [const CheckSum \\* checksum\\_object](#) () const
- [bool wait\\_eof\\_read](#) ()
- [bool wait\\_read](#) ()
- [bool wait\\_eof\\_write](#) ()
- [bool wait\\_write](#) ()



- bool [wait\\_eof](#) ()
- unsigned long long int [eof\\_position](#) () const
- unsigned int [buffer\\_size](#) () const

## Data Fields

- [DataSpeed](#) speed

### 4.2.1 Detailed Description

Represents set of buffers. This class is used during data transfer using [DataPoint](#) classes.

### 4.2.2 Constructor & Destructor Documentation

#### 4.2.2.1 Arc::DataBuffer::DataBuffer (unsigned int *size* = 65536, int *blocks* = 3)

Constructor

##### Parameters:

*size* size of every buffer in bytes.

*blocks* number of buffers.

#### 4.2.2.2 Arc::DataBuffer::DataBuffer (Checksum \* *cksum*, unsigned int *size* = 65536, int *blocks* = 3)

Constructor

##### Parameters:

*size* size of every buffer in bytes.

*blocks* number of buffers.

*cksum* object which will compute checksum. Should not be destroyed till [DataBuffer](#) itself.

### 4.2.3 Member Function Documentation

#### 4.2.3.1 int Arc::DataBuffer::add (Checksum \* *cksum*)

Add a checksum object which will compute checksum of buffer.

##### Parameters:

*cksum* object which will compute checksum. Should not be destroyed till [DataBuffer](#) itself.

##### Returns:

integer position in the list of checksum objects.

**4.2.3.2 unsigned int Arc::DataBuffer::buffer\_size () const**

Returns size of buffer in object. If not initialized then this number represents size of default buffer.

**4.2.3.3 const CheckSum\* Arc::DataBuffer::checksum\_object () const**

Returns CheckSum object specified in constructor, returns NULL if index is not in list.

**Parameters:**

*index* of the checksum in question.

**4.2.3.4 bool Arc::DataBuffer::checksum\_valid () const**

Returns true if checksum was successfully computed, returns false if index is not in list.

**Parameters:**

*index* of the checksum in question.

**4.2.3.5 bool Arc::DataBuffer::eof\_read ()**

Returns true if object was informed about end of transfer on 'read' side.

**4.2.3.6 void Arc::DataBuffer::eof\_read (bool v)**

Informs object if there will be no more request for 'read' buffers. v true if no more requests.

**4.2.3.7 bool Arc::DataBuffer::eof\_write ()**

Returns true if object was informed about end of transfer on 'write' side.

**4.2.3.8 void Arc::DataBuffer::eof\_write (bool v)**

Informs object if there will be no more request for 'write' buffers. v true if no more requests.

**4.2.3.9 bool Arc::DataBuffer::error ()**

Returns true if object was informed about error or internal error occurred.

**4.2.3.10 void Arc::DataBuffer::error\_read (bool v)**

Informs object if error occurred on 'read' side.

**Parameters:**

*v* true if error.

#### 4.2.3.11 void Arc::DataBuffer::error\_write (bool *v*)

Informs object if error accured on 'write' side.

**Parameters:**

*v* true if error.

#### 4.2.3.12 bool Arc::DataBuffer::for\_read ()

Check if there are buffers which can be taken by [for\\_read\(\)](#). This function checks only for buffers and does not take eof and error conditions into account.

#### 4.2.3.13 bool Arc::DataBuffer::for\_read (int & *handle*, unsigned int & *length*, bool *wait*)

Request buffer for READING INTO it.

**Parameters:**

*handle* returns buffer's number.

*length* returns size of buffer

*wait* if true and there are no free buffers, method will wait for one.

**Returns:**

true on success For python bindings pattern of this method is (bool, handle, length) for\_read(wait). Here buffer for reading to be provided by external code and provided to [DataBuffer](#) object through [is\\_read\(\)](#) method. Content of buffer must not exceed provided length.

#### 4.2.3.14 bool Arc::DataBuffer::for\_write ()

Check if there are buffers which can be taken by [for\\_write\(\)](#). This function checks only for buffers and does not take eof and error conditions into account.

#### 4.2.3.15 bool Arc::DataBuffer::for\_write (int & *handle*, unsigned int & *length*, unsigned long int & *offset*, bool *wait*)

Request buffer for WRITING FROM it.

**Parameters:**

*handle* returns buffer's number.

*length* returns size of buffer

*wait* if true and there are no free buffers, method will wait for one. For python bindings pattern of this method is (bool, handle, length, offset, buffer) for\_write(wait). Here buffer is string with content of buffer provided by [DataBuffer](#) object;

**4.2.3.16 bool Arc::DataBuffer::is\_notwritten (char \* *buf*)**

Informs object that data was NOT written from buffer (and releases buffer).

**Parameters:**

*buf* - address of buffer

**4.2.3.17 bool Arc::DataBuffer::is\_notwritten (int *handle*)**

Informs object that data was NOT written from buffer (and releases buffer).

**Parameters:**

*handle* buffer's number.

**4.2.3.18 bool Arc::DataBuffer::is\_read (char \* *buf*, unsigned int *length*, unsigned long long int *offset*)**

Informs object that data was read into buffer.

**Parameters:**

*buf* - address of buffer

*length* amount of data.

*offset* offset in stream, file, etc.

**4.2.3.19 bool Arc::DataBuffer::is\_read (int *handle*, unsigned int *length*, unsigned long long int *offset*)**

Informs object that data was read into buffer.

**Parameters:**

*handle* buffer's number.

*length* amount of data.

*offset* offset in stream, file, etc. For python bindings pattern of that method is bool is\_read(handle,buffer,offset). Here buffer is string containing content of buffer to be passed to [DataBuffer](#) object.

**4.2.3.20 bool Arc::DataBuffer::is\_written (char \* *buf*)**

Informs object that data was written from buffer.

**Parameters:**

*buf* - address of buffer

**4.2.3.21 bool Arc::DataBuffer::is\_written (int *handle*)**

Informs object that data was written from buffer.

**Parameters:**

*handle* buffer's number.

**4.2.3.22 bool Arc::DataBuffer::set (Checksum \* *cksum* = NULL, unsigned int *size* = 65536, int *blocks* = 3)**

Reinitialize buffers with different parameters.

**Parameters:**

*size* size of every buffer in bytes.

*blocks* number of buffers.

*cksum* object which will compute checksum. Should not be destroyed till [DataBuffer](#) itself.

**4.2.3.23 bool Arc::DataBuffer::wait\_any ()**

Wait (max 60 sec.) till any action happens in object. Returns true if action is eof on any side.

The documentation for this class was generated from the following file:

- DataBuffer.h

## 4.3 Arc::DataCallback Class Reference

This class is used by [DataHandle](#) to report missing space on local filesystem.

```
#include <DataCallback.h>
```

### 4.3.1 Detailed Description

This class is used by [DataHandle](#) to report missing space on local filesystem. One of 'cb' functions here will be called if operation initiated by `DataHandle::StartReading` runs out of disk space.

The documentation for this class was generated from the following file:

- `DataCallback.h`

## 4.4 Arc::DataHandle Class Reference

This class is a wrapper around the [DataPoint](#) class.

```
#include <DataHandle.h>
```

### Public Member Functions

- [DataHandle](#) (const URL &url, const UserConfig &usercfg)
- [~DataHandle](#) ()
- [DataPoint](#) \* [operator->](#) ()
- const [DataPoint](#) \* [operator->](#) () const
- [DataPoint](#) & [operator\\*](#) ()
- const [DataPoint](#) & [operator\\*](#) () const
- bool [operator!](#) () const
- [operator bool](#) () const

### Static Public Member Functions

- static [DataPoint](#) \* [GetPoint](#) (const URL &url, const UserConfig &usercfg)

#### 4.4.1 Detailed Description

This class is a wrapper around the [DataPoint](#) class. It simplifies the construction, use and destruction of [DataPoint](#) objects and should be used instead of [DataPoint](#) classes directly. The appropriate [DataPoint](#) subclass is created automatically and stored internally in [DataHandle](#). A [DataHandle](#) instance can be thought of as a pointer to the [DataPoint](#) instance and the [DataPoint](#) can be accessed through the usual dereference operators. A [DataHandle](#) cannot be copied.

This class is main way to access remote data items and obtain information about them. Below is an example of accessing last 512 bytes of files stored at GridFTP server.

```
#include <iostream>
#include <arc/data/DataPoint.h>
#include <arc/data/DataHandle.h>
#include <arc/data/DataBuffer.h>

using namespace Arc;

int main(void) {
    #define DESIRED_SIZE 512
    Arc::UserConfig usercfg;
    URL url("gsiftp://localhost/files/file_test_21");
    DataPoint* handle = DataHandle::GetPoint(url,usercfg);
    if(!handle) {
        std::cerr<<"Unsupported URL protocol or malformed URL"<<std::endl;
        return -1;
    };
    FileInfo info;
    if(!handle->Stat(info)) {
        std::cerr<<"Failed Stat"<<std::endl;
        return -1;
    };
    unsigned long long int fsize = handle->GetSize();
    if(fsize == (unsigned long long int)-1) {
        std::cerr<<"file size is not available"<<std::endl;
    }
}
```

```

        return -1;
    };
    if(fsize == 0) {
        std::cerr<<"file is empty"<<std::endl;
        return -1;
    };
    unsigned long long int foffset;
    if(fsize > DESIRED_SIZE) {
        handle->Range(fsize-DESIRED_SIZE,fsize-1);
    };
    unsigned int wto;
    DataBuffer buffer;
    if(!handle->PrepareReading(10,wto)) {
        std::cerr<<"Failed PrepareReading"<<std::endl;
        return -1;
    };
    if(!handle->StartReading(buffer)) {
        std::cerr<<"Failed StopReading"<<std::endl;
        return -1;
    };
    for(;;) {
        int n;
        unsigned int length;
        unsigned long long int offset;
        if(!buffer.for_write(n,length,offset,true)) {
            break;
        };
        std::cout<<"BUFFER: "<<offset<<": "<<length<<" : "<<std::string((const char*)
            (buffer[n]),length)<<std::endl;
        buffer.is_written(n);
    };
    if(buffer.error()) {
        std::cerr<<"Transfer failed"<<std::endl;
    };
    handle->StopReading();
    handle->FinishReading();
    return 0;
}

```

### And the same example in python

```

import arc

desired_size = 512
usercfg = arc.UserConfig()
url = arc.URL("gsiftp://localhost/files/file_test_21")
handle = arc.DataHandle.GetPoint(url,usercfg)
info = arc.FileInfo("")
handle.Stat(info)
print "Name: ", info.GetName()
fsize = info.GetSize()
if fsize > desired_size:
    handle.Range(fsize-desired_size,fsize-1)
buffer = arc.DataBuffer()
res, wto = handle.PrepareReading(10)
handle.StartReading(buffer)
while True:
    n = 0
    length = 0
    offset = 0
    ( r, n, length, offset, buf) = buffer.for_write(True)
    if not r: break
    print "BUFFER: ", offset, ": ", length, " :", buf
    buffer.is_written(n);

```



## 4.4.2 Member Function Documentation

### 4.4.2.1 static `DataPoint*` `Arc::DataHandle::GetPoint` (`const URL & url`, `const UserConfig & usercfg`) [`inline`, `static`]

Returns a pointer to new [DataPoint](#) object corresponding to URL. This static method is mostly for bindings to other languages and if availability scope of obtained [DataPoint](#) is undefined.

The documentation for this class was generated from the following file:

- `DataHandle.h`

## 4.5 Arc::DataMover Class Reference

[DataMover](#) provides an interface to transfer data between two DataPoints.

```
#include <DataMover.h>
```

### Public Member Functions

- [DataMover](#) ()
- [~DataMover](#) ()
- [DataStatus Transfer](#) ([DataPoint](#) &source, [DataPoint](#) &destination, [FileCache](#) &cache, const [URLMap](#) &map, callback cb=NULL, void \*arg=NULL, const char \*prefix=NULL)
- [DataStatus Transfer](#) ([DataPoint](#) &source, [DataPoint](#) &destination, [FileCache](#) &cache, const [URLMap](#) &map, unsigned long long int min\_speed, time\_t min\_speed\_time, unsigned long long int min\_average\_speed, time\_t max\_inactivity\_time, callback cb=NULL, void \*arg=NULL, const char \*prefix=NULL)
- [DataStatus Delete](#) ([DataPoint](#) &url, bool errcont=false)
- bool [verbose](#) ()
- void [verbose](#) (bool)
- void [verbose](#) (const std::string &prefix)
- bool [retry](#) ()
- void [retry](#) (bool)
- void [secure](#) (bool)
- void [passive](#) (bool)
- void [force\\_to\\_meta](#) (bool)
- bool [checks](#) ()
- void [checks](#) (bool v)
- void [set\\_default\\_min\\_speed](#) (unsigned long long int min\_speed, time\_t min\_speed\_time)
- void [set\\_default\\_min\\_average\\_speed](#) (unsigned long long int min\_average\_speed)
- void [set\\_default\\_max\\_inactivity\\_time](#) (time\_t max\_inactivity\_time)
- void [set\\_progress\\_indicator](#) ([DataSpeed::show\\_progress\\_t](#) func=NULL)
- void [set\\_preferred\\_pattern](#) (const std::string &pattern)

### 4.5.1 Detailed Description

[DataMover](#) provides an interface to transfer data between two DataPoints. Its main action is represented by Transfer methods

### 4.5.2 Member Function Documentation

#### 4.5.2.1 void Arc::DataMover::checks (bool v)

Set if to make check for existence of remote file (and probably other checks too) before initiating 'reading' and 'writing' operations.

#### Parameters:

*v* true if allowed (default is true).

**4.5.2.2 bool Arc::DataMover::checks ()**

Check if check for existence of remote file is done before initiating 'reading' and 'writing' operations.

**4.5.2.3 void Arc::DataMover::force\_to\_meta (bool)**

Set if file should be transferred and registered even if such LFN is already registered and source is not one of registered locations.

**4.5.2.4 void Arc::DataMover::secure (bool)**

Set if high level of security (encryption) will be used during transfer if available.

**4.5.2.5 void Arc::DataMover::set\_default\_max\_inactivity\_time (time\_t *max\_inactivity\_time*) [inline]**

Set maximal allowed time for waiting for any data. For more information see description of [DataSpeed](#) class.

**4.5.2.6 void Arc::DataMover::set\_default\_min\_average\_speed (unsigned long long int *min\_average\_speed*) [inline]**

Set minimal allowed average transfer speed (default is 0 averaged over whole time of transfer. For more information see description of [DataSpeed](#) class.

**4.5.2.7 void Arc::DataMover::set\_default\_min\_speed (unsigned long long int *min\_speed*, time\_t *min\_speed\_time*) [inline]**

Set minimal allowed transfer speed (default is 0) to 'min\_speed'. If speed drops below for time longer than 'min\_speed\_time' error is raised. For more information see description of [DataSpeed](#) class.

**4.5.2.8 DataStatus Arc::DataMover::Transfer (DataPoint & *source*, DataPoint & *destination*, FileCache & *cache*, const URLMap & *map*, unsigned long long int *min\_speed*, time\_t *min\_speed\_time*, unsigned long long int *min\_average\_speed*, time\_t *max\_inactivity\_time*, callback *cb* = NULL, void \* *arg* = NULL, const char \* *prefix* = NULL)**

Initiates transfer from 'source' to 'destination'.

**Parameters:**

*min\_speed* minimal allowed current speed.

*min\_speed\_time* time for which speed should be less than 'min\_speed' before transfer fails.

*min\_average\_speed* minimal allowed average speed.

*max\_inactivity\_time* time for which should be no activity before transfer fails.

#### 4.5.2.9 **DataStatus Arc::DataMover::Transfer (DataPoint & *source*, DataPoint & *destination*, FileCache & *cache*, const URLMap & *map*, callback *cb* = NULL, void \* *arg* = NULL, const char \* *prefix* = NULL)**

Initiates transfer from 'source' to 'destination'.

##### **Parameters:**

*source* source URL.

*destination* destination URL.

*cache* controls caching of downloaded files (if destination url is "file:///"). If caching is not needed default constructor FileCache() can be used.

*map* URL mapping/conversion table (for 'source' URL).

*cb* if not NULL, transfer is done in separate thread and 'cb' is called after transfer completes/fails.

*arg* passed to 'cb'.

*prefix* if 'verbose' is activated this information will be printed before each line representing current transfer status.

#### 4.5.2.10 **void Arc::DataMover::verbose (const std::string & *prefix*)**

Activate printing information about transfer status.

##### **Parameters:**

*prefix* use this string if 'prefix' in [DataMover::Transfer](#) is NULL.

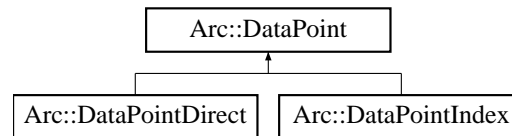
The documentation for this class was generated from the following file:

- DataMover.h

## 4.6 Arc::DataPoint Class Reference

A [DataPoint](#) represents a data resource and is an abstraction of a URL.

#include <DataPoint.h> Inheritance diagram for Arc::DataPoint::



### Public Types

- enum [DataPointAccessLatency](#) { [ACCESS\\_LATENCY\\_ZERO](#), [ACCESS\\_LATENCY\\_SMALL](#), [ACCESS\\_LATENCY\\_LARGE](#) }
- enum [DataPointInfoType](#) {  
[INFO\\_TYPE\\_MINIMAL](#) = 0, [INFO\\_TYPE\\_NAME](#) = 1, [INFO\\_TYPE\\_TYPE](#) = 2, [INFO\\_TYPE\\_TIMES](#) = 4,  
[INFO\\_TYPE\\_CONTENT](#) = 8, [INFO\\_TYPE\\_ACCESS](#) = 16, [INFO\\_TYPE\\_STRUCT](#) = 32, [INFO\\_TYPE\\_REST](#) = 64,  
[INFO\\_TYPE\\_ALL](#) = 127 }

### Public Member Functions

- virtual [~DataPoint](#) ()
- virtual const URL & [GetURL](#) () const
- virtual const UserConfig & [GetUserConfig](#) () const
- virtual bool [SetURL](#) (const URL &url)
- virtual std::string [str](#) () const
- virtual [operator bool](#) () const
- virtual bool [operator!](#) () const
- virtual [DataStatus PrepareReading](#) (unsigned int timeout, unsigned int &wait\_time)
- virtual [DataStatus PrepareWriting](#) (unsigned int timeout, unsigned int &wait\_time)
- virtual [DataStatus StartReading](#) ([DataBuffer](#) &buffer)=0
- virtual [DataStatus StartWriting](#) ([DataBuffer](#) &buffer, [DataCallback](#) \*space\_cb=NULL)=0
- virtual [DataStatus StopReading](#) ()=0
- virtual [DataStatus StopWriting](#) ()=0
- virtual [DataStatus FinishReading](#) (bool error=false)
- virtual [DataStatus FinishWriting](#) (bool error=false)
- virtual [DataStatus Check](#) ()=0
- virtual [DataStatus Remove](#) ()=0
- virtual [DataStatus Stat](#) ([FileInfo](#) &file, [DataPointInfoType](#) verb=INFO\_TYPE\_ALL)=0
- virtual [DataStatus Stat](#) (std::list< [FileInfo](#) > &files, const std::list< [DataPoint](#) \* > &urls, [DataPointInfoType](#) verb=INFO\_TYPE\_ALL)=0
- virtual [DataStatus List](#) (std::list< [FileInfo](#) > &files, [DataPointInfoType](#) verb=INFO\_TYPE\_ALL)=0
- virtual [DataStatus CreateDirectory](#) (bool with\_parents=false)=0
- virtual void [ReadOutOfOrder](#) (bool v)=0
- virtual bool [WriteOutOfOrder](#) ()=0

- virtual void [SetAdditionalChecks](#) (bool v)=0
- virtual bool [GetAdditionalChecks](#) () const =0
- virtual void [SetSecure](#) (bool v)=0
- virtual bool [GetSecure](#) () const =0
- virtual void [Passive](#) (bool v)=0
- virtual [DataStatus](#) [GetFailureReason](#) (void) const
- virtual void [Range](#) (unsigned long long int start=0, unsigned long long int end=0)=0
- virtual [DataStatus](#) [Resolve](#) (bool source)=0
- virtual [DataStatus](#) [Resolve](#) (bool source, const std::list< [DataPoint](#) \* > &urls)=0
- virtual bool [Registered](#) () const =0
- virtual [DataStatus](#) [PreRegister](#) (bool replication, bool force=false)=0
- virtual [DataStatus](#) [PostRegister](#) (bool replication)=0
- virtual [DataStatus](#) [PreUnregister](#) (bool replication)=0
- virtual [DataStatus](#) [Unregister](#) (bool all)=0
- virtual bool [CheckSize](#) () const
- virtual void [SetSize](#) (const unsigned long long int val)
- virtual unsigned long long int [GetSize](#) () const
- virtual bool [CheckChecksum](#) () const
- virtual void [SetChecksum](#) (const std::string &val)
- virtual const std::string & [GetChecksum](#) () const
- virtual const std::string [DefaultChecksum](#) () const
- virtual bool [CheckCreated](#) () const
- virtual void [SetCreated](#) (const Time &val)
- virtual const Time & [GetCreated](#) () const
- virtual bool [CheckValid](#) () const
- virtual void [SetValid](#) (const Time &val)
- virtual const Time & [GetValid](#) () const
- virtual void [SetAccessLatency](#) (const [DataPointAccessLatency](#) &latency)
- virtual [DataPointAccessLatency](#) [GetAccessLatency](#) () const
- virtual long long int [BufSize](#) () const =0
- virtual int [BufNum](#) () const =0
- virtual bool [Cache](#) () const
- virtual bool [Local](#) () const =0
- virtual int [GetTries](#) () const
- virtual void [SetTries](#) (const int n)
- virtual void [NextTry](#) (void)
- virtual bool [IsIndex](#) () const =0
- virtual bool [IsStageable](#) () const
- virtual bool [AcceptsMeta](#) () const =0
- virtual bool [ProvidesMeta](#) () const =0
- virtual void [SetMeta](#) (const [DataPoint](#) &p)
- virtual bool [CompareMeta](#) (const [DataPoint](#) &p) const
- virtual std::vector< URL > [TransferLocations](#) () const
- virtual const URL & [CurrentLocation](#) () const =0
- virtual const std::string & [CurrentLocationMetadata](#) () const =0
- virtual [DataPoint](#) \* [CurrentLocationHandle](#) () const =0
- virtual [DataStatus](#) [CompareLocationMetadata](#) () const =0
- virtual bool [NextLocation](#) ()=0
- virtual bool [LocationValid](#) () const =0
- virtual bool [LastLocation](#) ()=0

- virtual bool [HaveLocations](#) () const =0
- virtual [DataStatus AddLocation](#) (const URL &url, const std::string &meta)=0
- virtual [DataStatus RemoveLocation](#) ()=0
- virtual [DataStatus RemoveLocations](#) (const [DataPoint](#) &p)=0
- virtual [DataStatus ClearLocations](#) ()=0
- virtual int [AddChecksumObject](#) (Checksum \*cksum)=0
- virtual const Checksum \* [GetChecksumObject](#) (int index) const =0
- virtual void [SortLocations](#) (const std::string &pattern, const [URLMap](#) &url\_map)=0
- virtual void [AddURLOptions](#) (const std::map< std::string, std::string > &options)

## Protected Member Functions

- [DataPoint](#) (const URL &url, const UserConfig &usercfg, PluginArgument \*parg)

## Protected Attributes

- std::set< std::string > [valid\\_url\\_options](#)

### 4.6.1 Detailed Description

A [DataPoint](#) represents a data resource and is an abstraction of a URL. [DataPoint](#) uses ARC's Plugin mechanism to dynamically load the required Data Manager Component (DMC) when necessary. A DMC typically defines a subclass of [DataPoint](#) (e.g. [DataPointHTTP](#)) and is responsible for a specific protocol (e.g. http). DataPoints should not be used directly, instead the [DataHandle](#) wrapper class should be used, which automatically loads the correct DMC.

[DataPoint](#) defines methods for access to the data resource. To transfer data between two DataPoints, [DataMover::Transfer\(\)](#) can be used.

There are two subclasses of [DataPoint](#), [DataPointDirect](#) and [DataPointIndex](#). None of these three classes can be instantiated directly. [DataPointDirect](#) and its subclasses handle "physical" resources through protocols such as file, http and gsiftp. These classes implement methods such as [StartReading\(\)](#) and [StartWriting\(\)](#). [DataPointIndex](#) and its subclasses handle resources such as indexes and catalogs and implement methods like [Resolve\(\)](#) and [PreRegister\(\)](#).

When creating a new DMC, a subclass of either [DataPointDirect](#) or [DataPointIndex](#) should be created, and the appropriate methods implemented. [DataPoint](#) itself has no direct external dependencies, but plugins may rely on third-party components. The new DMC must also add itself to the list of available plugins and provide an Instance() method which returns a new instance of itself, if the supplied arguments are valid for the protocol. Here is an example implementation of a new DMC for protocol MyProtocol which represents a physical resource accessible through protocol my://

```
#include <arc/data/DataPointDirect.h>

namespace Arc {

class DataPointMyProtocol : public DataPointDirect {
public:
    DataPointMyProtocol(const URL& url, const UserConfig& usercfg);
    static Plugin* Instance(PluginArgument *arg);
    virtual DataStatus StartReading(DataBuffer& buffer);
    ...
};

DataPointMyProtocol::DataPointMyProtocol(const URL& url, const UserConfig& userc
```

```

        fg) {
        ...
    }

    DataPointMyProtocol::StartReading(DataBuffer& buffer) { ... }

    ...

    Plugin* DataPointMyProtocol::Instance(PluginArgument *arg) {
        DataPointPluginArgument *dmcarg = dynamic_cast<DataPointPluginArgument*>(arg);

        if (!dmcarg)
            return NULL;
        if (((const URL &)(*dmcarg)).Protocol() != "my")
            return NULL;
        return new DataPointMyProtocol(*dmcarg, *dmcarg);
    }

    } // namespace Arc

    Arc::PluginDescriptor PLUGINS_TABLE_NAME[] = {
        { "my", "HED:DMC", 0, &Arc::DataPointMyProtocol::Instance },
        { NULL, NULL, 0, NULL }
    };
};

```

## 4.6.2 Member Enumeration Documentation

### 4.6.2.1 enum Arc::DataPoint::DataPointAccessLatency

Describes the latency to access this URL. For now this value is one of a small set specified by the enumeration. In the future with more sophisticated protocols or information it could be replaced by a more fine-grained list of possibilities such as an int value.

#### Enumerator:

**ACCESS\_LATENCY\_ZERO** URL can be accessed instantly.

**ACCESS\_LATENCY\_SMALL** URL has low (but non-zero) access latency, for example staged from disk.

**ACCESS\_LATENCY\_LARGE** URL has a large access latency, for example staged from tape.

### 4.6.2.2 enum Arc::DataPoint::DataPointInfoType

Describes type of information about URL to request.

#### Enumerator:

**INFO\_TYPE\_MINIMAL** Whatever protocol can get with no additional effort.

**INFO\_TYPE\_NAME** Only name of object (relative).

**INFO\_TYPE\_TYPE** Type of object - currently file or dir.

**INFO\_TYPE\_TIMES** Timestamps associated with object.

**INFO\_TYPE\_CONTENT** Metadata describing content, like size, checksum, etc.

**INFO\_TYPE\_ACCESS** Access control - ownership, permission, etc.

**INFO\_TYPE\_STRUCT** Fine structure - replicas, transfer locations, redirections.

**INFO\_TYPE\_REST** All the other parameters.

**INFO\_TYPE\_ALL** All the parameters.



### 4.6.3 Constructor & Destructor Documentation

#### 4.6.3.1 Arc::DataPoint::DataPoint (const URL & *url*, const UserConfig & *usercfg*, PluginArgument \* *parg*) [protected]

Constructor. Constructor is protected because DataPoints should not be created directly. Subclasses should however call this in their constructors to set various common attributes.

**Parameters:**

*url* The URL representing the [DataPoint](#)  
*usercfg* User configuration object

### 4.6.4 Member Function Documentation

#### 4.6.4.1 virtual int Arc::DataPoint::AddChecksumObject (Checksum \* *cksum*) [pure virtual]

Add a checksum object which will compute checksum during transmission.

**Parameters:**

*cksum* object which will compute checksum. Should not be destroyed till DataPointer itself.

**Returns:**

integer position in the list of checksum objects.

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

#### 4.6.4.2 virtual DataStatus Arc::DataPoint::AddLocation (const URL & *url*, const std::string & *meta*) [pure virtual]

Add URL to list.

**Parameters:**

*url* Location URL to add.  
*meta* Location meta information.

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

#### 4.6.4.3 virtual void Arc::DataPoint::AddURLOptions (const std::map< std::string, std::string > & *options*) [virtual]

Add URL options to this DataPoint's URL object. Invalid options for the [DataPoint](#) instance will not be added.

#### 4.6.4.4 virtual DataStatus Arc::DataPoint::Check () [pure virtual]

Query the [DataPoint](#) to check if object is accessible. If possible this method will also try to provide meta information about the object. It returns positive response if object's content can be retrieved.

Implemented in [Arc::DataPointIndex](#).

#### 4.6.4.5 **virtual DataStatus Arc::DataPoint::CompareLocationMetadata () const [pure virtual]**

Compare metadata of [DataPoint](#) and current location. Returns inconsistency error or error encountered during operation, or success

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

#### 4.6.4.6 **virtual bool Arc::DataPoint::CompareMeta (const DataPoint & p) const [virtual]**

Compare meta information from another object. Undefined values are not used for comparison.

##### Parameters:

*p* object to which to compare.

#### 4.6.4.7 **virtual DataStatus Arc::DataPoint::CreateDirectory (bool *with\_parents* = false) [pure virtual]**

Create a directory. If the protocol supports it, this method creates the last directory in the path to the URL. It assumes the last component of the path is a file-like object and not a directory itself, unless the path ends in a directory separator. If *with\_parents* is true then all missing parent directories in the path will also be created.

##### Parameters:

*with\_parents* If true then all missing directories in the path are created

#### 4.6.4.8 **virtual const std::string& Arc::DataPoint::CurrentLocationMetadata () const [pure virtual]**

Returns meta information used to create current URL. Usage differs between different indexing services.

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

#### 4.6.4.9 **virtual DataStatus Arc::DataPoint::FinishReading (bool *error* = false) [virtual]**

Finish reading from the URL. Must be called after transfer of physical file has completed and if [PrepareReading\(\)](#) was called, to free resources, release requests that were made during preparation etc.

##### Parameters:

*error* If true then action is taken depending on the error.

Reimplemented in [Arc::DataPointIndex](#).

#### 4.6.4.10 **virtual DataStatus Arc::DataPoint::FinishWriting (bool *error* = false) [virtual]**

Finish writing to the URL. Must be called after transfer of physical file has completed and if [PrepareWriting\(\)](#) was called, to free resources, release requests that were made during preparation etc.

**Parameters:**

*error* If true then action is taken depending on the error.

Reimplemented in [Arc::DataPointIndex](#).

**4.6.4.11 virtual DataStatus Arc::DataPoint::GetFailureReason (void) const [virtual]**

Returns reason of transfer failure, as reported by callbacks. This could be different from the failure returned by the methods themselves.

**4.6.4.12 virtual DataStatus Arc::DataPoint::List (std::list< FileInfo > &files, DataPointInfoType verb = INFO\_TYPE\_ALL) [pure virtual]**

List hierarchical content of this object. If the [DataPoint](#) represents a directory or something similar its contents will be listed.

**Parameters:**

*files* will contain list of file names and requested attributes. There may be more attributes than requested. There may be less if object can't provide particular information.

*verb* defines attribute types which method must try to retrieve. It is not a failure if some attributes could not be retrieved due to limitation of protocol or access control.

**4.6.4.13 virtual bool Arc::DataPoint::NextLocation () [pure virtual]**

Switch to next location in list of URLs. At last location switch to first if number of allowed retries is not exceeded. Returns false if no retries left.

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

**4.6.4.14 virtual void Arc::DataPoint::Passive (bool v) [pure virtual]**

Request passive transfers for FTP-like protocols.

**Parameters:**

*true* to request.

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

**4.6.4.15 virtual DataStatus Arc::DataPoint::PostRegister (bool replication) [pure virtual]**

Index Service postregistration. Used for same purpose as PreRegister. Should be called after actual transfer of file successfully finished.

**Parameters:**

*replication* if true, the file is being replicated between two locations registered in Indexing Service under same name.

Implemented in [Arc::DataPointDirect](#).

#### 4.6.4.16 **virtual DataStatus Arc::DataPoint::PrepareReading (unsigned int *timeout*, unsigned int & *wait\_time*) [virtual]**

Prepare [DataPoint](#) for reading. This method should be implemented by protocols which require preparation or staging of physical files for reading. It can act synchronously or asynchronously (if protocol supports it). In the first case the method will block until the file is prepared or the specified timeout has passed. In the second case the method can return with a ReadPrepareWait status before the file is prepared. The caller should then wait some time (a hint from the remote service may be given in *wait\_time*) and call [PrepareReading\(\)](#) again to poll for the preparation status, until the file is prepared. In this case it is also up to the caller to decide when the request has taken too long and if so cancel it by calling [FinishReading\(\)](#). When file preparation has finished, the physical file(s) to read from can be found from [TransferLocations\(\)](#).

##### Parameters:

***timeout*** If non-zero, this method will block until either the file has been prepared successfully or the timeout has passed. A zero value means that the caller would like to call and poll for status.

***wait\_time*** If timeout is zero (caller would like asynchronous operation) and ReadPrepareWait is returned, a hint for how long to wait before a subsequent call may be given in *wait\_time*.

Reimplemented in [Arc::DataPointIndex](#).

#### 4.6.4.17 **virtual DataStatus Arc::DataPoint::PrepareWriting (unsigned int *timeout*, unsigned int & *wait\_time*) [virtual]**

Prepare [DataPoint](#) for writing. This method should be implemented by protocols which require preparation of physical files for writing. It can act synchronously or asynchronously (if protocol supports it). In the first case the method will block until the file is prepared or the specified timeout has passed. In the second case the method can return with a WritePrepareWait status before the file is prepared. The caller should then wait some time (a hint from the remote service may be given in *wait\_time*) and call [PrepareWriting\(\)](#) again to poll for the preparation status, until the file is prepared. In this case it is also up to the caller to decide when the request has taken too long and if so cancel or abort it by calling [FinishWriting\(true\)](#). When file preparation has finished, the physical file(s) to write to can be found from [TransferLocations\(\)](#).

##### Parameters:

***timeout*** If non-zero, this method will block until either the file has been prepared successfully or the timeout has passed. A zero value means that the caller would like to call and poll for status.

***wait\_time*** If timeout is zero (caller would like asynchronous operation) and WritePrepareWait is returned, a hint for how long to wait before a subsequent call may be given in *wait\_time*.

Reimplemented in [Arc::DataPointIndex](#).

#### 4.6.4.18 **virtual DataStatus Arc::DataPoint::PreRegister (bool *replication*, bool *force* = false) [pure virtual]**

Index service preregistration. This function registers the physical location of a file into an indexing service. It should be called \*before\* the actual transfer to that location happens.

##### Parameters:

***replication*** if true, the file is being replicated between two locations registered in the indexing service under same name.

*force* if true, perform registration of a new file even if it already exists. Should be used to fix failures in Indexing Service.

Implemented in [Arc::DataPointDirect](#).

#### 4.6.4.19 virtual DataStatus Arc::DataPoint::PreUnregister (bool *replication*) [pure virtual]

Index Service preunregistration. Should be called if file transfer failed. It removes changes made by PreRegister.

##### Parameters:

*replication* if true, the file is being replicated between two locations registered in Indexing Service under same name.

Implemented in [Arc::DataPointDirect](#).

#### 4.6.4.20 virtual bool Arc::DataPoint::ProvidesMeta () const [pure virtual]

If endpoint can provide at least some meta information directly.

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

#### 4.6.4.21 virtual void Arc::DataPoint::Range (unsigned long long int *start* = 0, unsigned long long int *end* = 0) [pure virtual]

Set range of bytes to retrieve. Default values correspond to whole file.

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

#### 4.6.4.22 virtual void Arc::DataPoint::ReadOutOfOrder (bool *v*) [pure virtual]

Allow/disallow [DataPoint](#) to produce scattered data during reading\* operation.

##### Parameters:

*v* true if allowed (default is false).

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

#### 4.6.4.23 virtual bool Arc::DataPoint::Registered () const [pure virtual]

Check if file is registered in Indexing Service. Proper value is obtainable only after Resolve.

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

#### 4.6.4.24 virtual DataStatus Arc::DataPoint::Resolve (bool *source*, const std::list< DataPoint \* > & *urls*) [pure virtual]

Resolves several index service URLs. Can use bulk calls if protocol allows. The protocols and hosts of all the DataPoints in urls must be the same and the same as this DataPoint's protocol and host. This method can be called on any of the urls, for example `urls.front()->Resolve(true, urls);`

**Parameters:**

- source* true if [DataPoint](#) objects represent source of information
- urls* List of DataPoints to resolve. Protocols and hosts must match and match this DataPoint's protocol and host.

**4.6.4.25 virtual DataStatus Arc::DataPoint::Resolve (bool *source*) [pure virtual]**

Resolves index service URL into list of ordinary URLs. Also obtains meta information about the file.

**Parameters:**

- source* true if [DataPoint](#) object represents source of information.

Implemented in [Arc::DataPointDirect](#).

**4.6.4.26 virtual void Arc::DataPoint::SetAdditionalChecks (bool *v*) [pure virtual]**

Allow/disallow additional checks. Check for existence of remote file (and probably other checks too) before initiating reading and writing operations.

**Parameters:**

- v* true if allowed (default is true).

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

**4.6.4.27 virtual void Arc::DataPoint::SetMeta (const DataPoint & *p*) [virtual]**

Copy meta information from another object. Already defined values are not overwritten.

**Parameters:**

- p* object from which information is taken.

**4.6.4.28 virtual void Arc::DataPoint::SetSecure (bool *v*) [pure virtual]**

Allow/disallow heavy security during data transfer.

**Parameters:**

- v* true if allowed (default depends on protocol).

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

**4.6.4.29 virtual bool Arc::DataPoint::SetURL (const URL & *url*) [virtual]**

Assigns new URL. Main purpose of this method is to reuse existing connection for accessing different object at same server. Implementation does not have to implement this method. If supplied URL is not suitable or method is not implemented false is returned.

#### 4.6.4.30 virtual void Arc::DataPoint::SortLocations (const std::string & *pattern*, const URLMap & *url\_map*) [pure virtual]

Sort locations according to the specified pattern.

##### Parameters:

*pattern* a set of strings, separated by |, to match against.

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

#### 4.6.4.31 virtual DataStatus Arc::DataPoint::StartReading (DataBuffer & *buffer*) [pure virtual]

Start reading data from URL. Separate thread to transfer data will be created. No other operation can be performed while reading is in progress.

##### Parameters:

*buffer* operation will use this buffer to put information into. Should not be destroyed before [StopReading\(\)](#) was called and returned.

Implemented in [Arc::DataPointIndex](#).

#### 4.6.4.32 virtual DataStatus Arc::DataPoint::StartWriting (DataBuffer & *buffer*, DataCallback \* *space\_cb* = NULL) [pure virtual]

Start writing data to URL. Separate thread to transfer data will be created. No other operation can be performed while writing is in progress.

##### Parameters:

*buffer* operation will use this buffer to get information from. Should not be destroyed before `stop_writing` was called and returned.

*space\_cb* callback which is called if there is not enough space to store data. May not implemented for all protocols.

Implemented in [Arc::DataPointIndex](#).

#### 4.6.4.33 virtual DataStatus Arc::DataPoint::Stat (std::list< FileInfo > & *files*, const std::list< DataPoint \* > & *urls*, DataPointInfoType *verb* = INFO\_TYPE\_ALL) [pure virtual]

Retrieve information about several DataPoints. If a [DataPoint](#) represents a directory or something similar, information about the object itself and not its contents will be obtained. This method can use bulk operations if the protocol supports it. The protocols and hosts of all the DataPoints in *urls* must be the same and the same as this DataPoint's protocol and host. This method can be called on any of the *urls*, for example `urls.front()->Stat(files, urls)`; Calling this method with an empty list of *urls* returns success if the protocol supports bulk Stat, and an error if it does not.

##### Parameters:

*files* will contain objects' names and requested attributes. There may be more attributes than requested. There may be less if objects can't provide particular information. The order of this vector matches the order of *urls*. If a stat of any url fails then the corresponding [FileInfo](#) in this list will evaluate to false.

**urls** list of DataPoints to stat. Protocols and hosts must match and match this DataPoint's protocol and host.

**verb** defines attribute types which method must try to retrieve. It is not a failure if some attributes could not be retrieved due to limitation of protocol or access control.

#### 4.6.4.34 **virtual DataStatus Arc::DataPoint::Stat (FileInfo &file, DataPointInfoType verb = INFO\_TYPE\_ALL) [pure virtual]**

Retrieve information about this object. If the [DataPoint](#) represents a directory or something similar, information about the object itself and not its contents will be obtained.

##### Parameters:

**file** will contain object name and requested attributes. There may be more attributes than requested. There may be less if object can't provide particular information.

**verb** defines attribute types which method must try to retrieve. It is not a failure if some attributes could not be retrieved due to limitation of protocol or access control.

#### 4.6.4.35 **virtual DataStatus Arc::DataPoint::StopReading () [pure virtual]**

Stop reading. Must be called after corresponding start\_reading method, either after all data is transferred or to cancel transfer. Use buffer object to find out when data is transferred. Must return failure if any happened during transfer.

Implemented in [Arc::DataPointIndex](#).

#### 4.6.4.36 **virtual DataStatus Arc::DataPoint::StopWriting () [pure virtual]**

Stop writing. Must be called after corresponding start\_writing method, either after all data is transferred or to cancel transfer. Use buffer object to find out when data is transferred. Must return failure if any happened during transfer.

Implemented in [Arc::DataPointIndex](#).

#### 4.6.4.37 **virtual std::vector<URL> Arc::DataPoint::TransferLocations () const [virtual]**

Returns physical file(s) to read/write, if different from [CurrentLocation\(\)](#). To be used with protocols which re-direct to different URLs such as Transport URLs (TURLs). The list is initially filled by PrepareReading and PrepareWriting. If this list is non-empty then real transfer should use a URL from this list. It is up to the caller to choose the best URL and instantiate new [DataPoint](#) for handling it. For consistency protocols which do not require redirections return original URL. For protocols which need redirection calling StartReading and StartWriting will use first URL in the list.

Reimplemented in [Arc::DataPointIndex](#).

#### 4.6.4.38 **virtual DataStatus Arc::DataPoint::Unregister (bool all) [pure virtual]**

Index Service unregistration. Remove information about file registered in Indexing Service.



**Parameters:**

*all* if true, information about file itself is (LFN) is removed. Otherwise only particular physical instance is unregistered.

Implemented in [Arc::DataPointDirect](#).

**4.6.4.39 virtual bool Arc::DataPoint::WriteOutOfOrder () [pure virtual]**

Returns true if URL can accept scattered data for \*writing\* operation.

Implemented in [Arc::DataPointDirect](#), and [Arc::DataPointIndex](#).

**4.6.5 Field Documentation****4.6.5.1 std::set<std::string> Arc::DataPoint::valid\_url\_options [protected]**

Subclasses should add their own specific options to this list

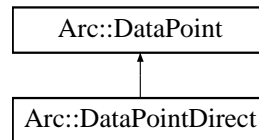
The documentation for this class was generated from the following file:

- DataPoint.h

## 4.7 Arc::DataPointDirect Class Reference

This is a kind of generalized file handle.

#include <DataPointDirect.h> Inheritance diagram for Arc::DataPointDirect:



### Public Member Functions

- virtual bool [IsIndex](#) () const
- virtual bool [IsStageable](#) () const
- virtual long long int [BufSize](#) () const
- virtual int [BufNum](#) () const
- virtual bool [Local](#) () const
- virtual void [ReadOutOfOrder](#) (bool v)
- virtual bool [WriteOutOfOrder](#) ()
- virtual void [SetAdditionalChecks](#) (bool v)
- virtual bool [GetAdditionalChecks](#) () const
- virtual void [SetSecure](#) (bool v)
- virtual bool [GetSecure](#) () const
- virtual void [Passive](#) (bool v)
- virtual void [Range](#) (unsigned long long int start=0, unsigned long long int end=0)
- virtual int [AddChecksumObject](#) (Checksum \*cksum)
- virtual const Checksum \* [GetChecksumObject](#) (int index) const
- virtual [DataStatus](#) [Resolve](#) (bool source)
- virtual bool [Registered](#) () const
- virtual [DataStatus](#) [PreRegister](#) (bool replication, bool force=false)
- virtual [DataStatus](#) [PostRegister](#) (bool replication)
- virtual [DataStatus](#) [PreUnregister](#) (bool replication)
- virtual [DataStatus](#) [Unregister](#) (bool all)
- virtual bool [AcceptsMeta](#) () const
- virtual bool [ProvidesMeta](#) () const
- virtual const URL & [CurrentLocation](#) () const
- virtual [DataPoint](#) \* [CurrentLocationHandle](#) () const
- virtual const std::string & [CurrentLocationMetadata](#) () const
- virtual [DataStatus](#) [CompareLocationMetadata](#) () const
- virtual bool [NextLocation](#) ()
- virtual bool [LocationValid](#) () const
- virtual bool [HaveLocations](#) () const
- virtual bool [LastLocation](#) ()
- virtual [DataStatus](#) [AddLocation](#) (const URL &url, const std::string &meta)
- virtual [DataStatus](#) [RemoveLocation](#) ()
- virtual [DataStatus](#) [ClearLocations](#) ()
- virtual void [SortLocations](#) (const std::string &, const [URLMap](#) &)

### 4.7.1 Detailed Description

This is a kind of generalized file handle. Differently from file handle it does not support operations `read()` and `write()`. Instead it initiates operation and uses object of class [DataBuffer](#) to pass actual data. It also provides other operations like querying parameters of remote object. It is used by higher-level classes `DataMove` and `DataMovePar` to provide data transfer service for application.

### 4.7.2 Member Function Documentation

#### 4.7.2.1 `virtual int Arc::DataPointDirect::AddChecksumObject (Checksum * cksum) [virtual]`

Add a checksum object which will compute checksum during transmission.

**Parameters:**

*cksum* object which will compute checksum. Should not be destroyed till `DataPointer` itself.

**Returns:**

integer position in the list of checksum objects.

Implements [Arc::DataPoint](#).

#### 4.7.2.2 `virtual DataStatus Arc::DataPointDirect::AddLocation (const URL & url, const std::string & meta) [virtual]`

Add URL to list.

**Parameters:**

*url* Location URL to add.

*meta* Location meta information.

Implements [Arc::DataPoint](#).

#### 4.7.2.3 `virtual DataStatus Arc::DataPointDirect::CompareLocationMetadata () const [virtual]`

Compare metadata of [DataPoint](#) and current location. Returns inconsistency error or error encountered during operation, or success

Implements [Arc::DataPoint](#).

#### 4.7.2.4 `virtual const std::string& Arc::DataPointDirect::CurrentLocationMetadata () const [virtual]`

Returns meta information used to create current URL. Usage differs between different indexing services.

Implements [Arc::DataPoint](#).

#### 4.7.2.5 virtual bool Arc::DataPointDirect::NextLocation () [virtual]

Switch to next location in list of URLs. At last location switch to first if number of allowed retries is not exceeded. Returns false if no retries left.

Implements [Arc::DataPoint](#).

#### 4.7.2.6 virtual void Arc::DataPointDirect::Passive (bool v) [virtual]

Request passive transfers for FTP-like protocols.

##### Parameters:

*true* to request.

Implements [Arc::DataPoint](#).

#### 4.7.2.7 virtual DataStatus Arc::DataPointDirect::PostRegister (bool replication) [virtual]

Index Service postregistration. Used for same purpose as PreRegister. Should be called after actual transfer of file successfully finished.

##### Parameters:

*replication* if true, the file is being replicated between two locations registered in Indexing Service under same name.

Implements [Arc::DataPoint](#).

#### 4.7.2.8 virtual DataStatus Arc::DataPointDirect::PreRegister (bool replication, bool force = false) [virtual]

Index service preregistration. This function registers the physical location of a file into an indexing service. It should be called \*before\* the actual transfer to that location happens.

##### Parameters:

*replication* if true, the file is being replicated between two locations registered in the indexing service under same name.

*force* if true, perform registration of a new file even if it already exists. Should be used to fix failures in Indexing Service.

Implements [Arc::DataPoint](#).

#### 4.7.2.9 virtual DataStatus Arc::DataPointDirect::PreUnregister (bool replication) [virtual]

Index Service preunregistration. Should be called if file transfer failed. It removes changes made by PreRegister.

##### Parameters:

*replication* if true, the file is being replicated between two locations registered in Indexing Service under same name.

Implements [Arc::DataPoint](#).

**4.7.2.10 virtual bool Arc::DataPointDirect::ProvidesMeta () const [virtual]**

If endpoint can provide at least some meta information directly.

Implements [Arc::DataPoint](#).

**4.7.2.11 virtual void Arc::DataPointDirect::Range (unsigned long long int *start* = 0, unsigned long long int *end* = 0) [virtual]**

Set range of bytes to retrieve. Default values correspond to whole file.

Implements [Arc::DataPoint](#).

**4.7.2.12 virtual void Arc::DataPointDirect::ReadOutOfOrder (bool *v*) [virtual]**

Allow/disallow [DataPoint](#) to produce scattered data during reading\* operation.

**Parameters:**

*v* true if allowed (default is false).

Implements [Arc::DataPoint](#).

**4.7.2.13 virtual bool Arc::DataPointDirect::Registered () const [virtual]**

Check if file is registered in Indexing Service. Proper value is obtainable only after Resolve.

Implements [Arc::DataPoint](#).

**4.7.2.14 virtual DataStatus Arc::DataPointDirect::Resolve (bool *source*) [virtual]**

Resolves index service URL into list of ordinary URLs. Also obtains meta information about the file.

**Parameters:**

*source* true if [DataPoint](#) object represents source of information.

Implements [Arc::DataPoint](#).

**4.7.2.15 virtual void Arc::DataPointDirect::SetAdditionalChecks (bool *v*) [virtual]**

Allow/disallow additional checks. Check for existence of remote file (and probably other checks too) before initiating reading and writing operations.

**Parameters:**

*v* true if allowed (default is true).

Implements [Arc::DataPoint](#).

**4.7.2.16 virtual void Arc::DataPointDirect::SetSecure (bool *v*) [virtual]**

Allow/disallow heavy security during data transfer.

**Parameters:**

*v* true if allowed (default depends on protocol).

Implements [Arc::DataPoint](#).

**4.7.2.17 virtual void Arc::DataPointDirect::SortLocations (const std::string & *pattern*, const URLMap & *url\_map*) [inline, virtual]**

Sort locations according to the specified pattern.

**Parameters:**

*pattern* a set of strings, separated by |, to match against.

Implements [Arc::DataPoint](#).

**4.7.2.18 virtual DataStatus Arc::DataPointDirect::Unregister (bool *all*) [virtual]**

Index Service unregistration. Remove information about file registered in Indexing Service.

**Parameters:**

*all* if true, information about file itself is (LFN) is removed. Otherwise only particular physical instance is unregistered.

Implements [Arc::DataPoint](#).

**4.7.2.19 virtual bool Arc::DataPointDirect::WriteOutOfOrder () [virtual]**

Returns true if URL can accept scattered data for \*writing\* operation.

Implements [Arc::DataPoint](#).

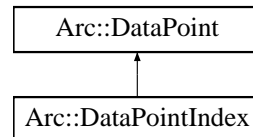
The documentation for this class was generated from the following file:

- DataPointDirect.h

## 4.8 Arc::DataPointIndex Class Reference

Complements [DataPoint](#) with attributes common for Indexing Service URLs.

#include <DataPointIndex.h> Inheritance diagram for Arc::DataPointIndex::



### Public Member Functions

- virtual const URL & [CurrentLocation](#) () const
- virtual const std::string & [CurrentLocationMetadata](#) () const
- virtual [DataPoint](#) \* [CurrentLocationHandle](#) () const
- virtual [DataStatus](#) [CompareLocationMetadata](#) () const
- virtual bool [NextLocation](#) ()
- virtual bool [LocationValid](#) () const
- virtual bool [HaveLocations](#) () const
- virtual bool [LastLocation](#) ()
- virtual [DataStatus](#) [RemoveLocation](#) ()
- virtual [DataStatus](#) [ClearLocations](#) ()
- virtual [DataStatus](#) [AddLocation](#) (const URL &url, const std::string &meta)
- virtual void [SortLocations](#) (const std::string &pattern, const [URLMap](#) &url\_map)
- virtual bool [IsIndex](#) () const
- virtual bool [IsStageable](#) () const
- virtual bool [AcceptsMeta](#) () const
- virtual bool [ProvidesMeta](#) () const
- virtual void [SetChecksum](#) (const std::string &val)
- virtual void [SetSize](#) (const unsigned long long int val)
- virtual bool [Registered](#) () const
- virtual void [SetTries](#) (const int n)
- virtual long long int [BufSize](#) () const
- virtual int [BufNum](#) () const
- virtual bool [Local](#) () const
- virtual [DataStatus](#) [PrepareReading](#) (unsigned int timeout, unsigned int &wait\_time)
- virtual [DataStatus](#) [PrepareWriting](#) (unsigned int timeout, unsigned int &wait\_time)
- virtual [DataStatus](#) [StartReading](#) ([DataBuffer](#) &buffer)
- virtual [DataStatus](#) [StartWriting](#) ([DataBuffer](#) &buffer, [DataCallback](#) \*space\_cb=NULL)
- virtual [DataStatus](#) [StopReading](#) ()
- virtual [DataStatus](#) [StopWriting](#) ()
- virtual [DataStatus](#) [FinishReading](#) (bool error=false)
- virtual [DataStatus](#) [FinishWriting](#) (bool error=false)
- virtual std::vector< URL > [TransferLocations](#) () const
- virtual [DataStatus](#) [Check](#) ()
- virtual [DataStatus](#) [Remove](#) ()
- virtual void [ReadOutOfOrder](#) (bool v)
- virtual bool [WriteOutOfOrder](#) ()

- virtual void [SetAdditionalChecks](#) (bool v)
- virtual bool [GetAdditionalChecks](#) () const
- virtual void [SetSecure](#) (bool v)
- virtual bool [GetSecure](#) () const
- virtual [DataPointAccessLatency](#) [GetAccessLatency](#) () const
- virtual void [Passive](#) (bool v)
- virtual void [Range](#) (unsigned long long int start=0, unsigned long long int end=0)
- virtual int [AddChecksumObject](#) (Checksum \*cksum)
- virtual const Checksum \* [GetChecksumObject](#) (int index) const

### 4.8.1 Detailed Description

Complements [DataPoint](#) with attributes common for Indexing Service URLs. It should never be used directly. Instead inherit from it to provide a class for specific a Indexing Service.

### 4.8.2 Member Function Documentation

#### 4.8.2.1 virtual int Arc::DataPointIndex::AddChecksumObject (Checksum \* *cksum*) [virtual]

Add a checksum object which will compute checksum during transmission.

##### Parameters:

*cksum* object which will compute checksum. Should not be destroyed till DataPointer itself.

##### Returns:

integer position in the list of checksum objects.

Implements [Arc::DataPoint](#).

#### 4.8.2.2 virtual DataStatus Arc::DataPointIndex::AddLocation (const URL & *url*, const std::string & *meta*) [virtual]

Add URL to list.

##### Parameters:

*url* Location URL to add.

*meta* Location meta information.

Implements [Arc::DataPoint](#).

#### 4.8.2.3 virtual DataStatus Arc::DataPointIndex::Check () [virtual]

Query the [DataPoint](#) to check if object is accessible. If possible this method will also try to provide meta information about the object. It returns positive response if object's content can be retrieved.

Implements [Arc::DataPoint](#).



#### 4.8.2.4 virtual DataStatus Arc::DataPointIndex::CompareLocationMetadata () const [virtual]

Compare metadata of [DataPoint](#) and current location. Returns inconsistency error or error encountered during operation, or success

Implements [Arc::DataPoint](#).

#### 4.8.2.5 virtual const std::string& Arc::DataPointIndex::CurrentLocationMetadata () const [virtual]

Returns meta information used to create current URL. Usage differs between different indexing services.

Implements [Arc::DataPoint](#).

#### 4.8.2.6 virtual DataStatus Arc::DataPointIndex::FinishReading (bool *error* = false) [virtual]

Finish reading from the URL. Must be called after transfer of physical file has completed and if [PrepareReading\(\)](#) was called, to free resources, release requests that were made during preparation etc.

##### Parameters:

*error* If true then action is taken depending on the error.

Reimplemented from [Arc::DataPoint](#).

#### 4.8.2.7 virtual DataStatus Arc::DataPointIndex::FinishWriting (bool *error* = false) [virtual]

Finish writing to the URL. Must be called after transfer of physical file has completed and if [PrepareWriting\(\)](#) was called, to free resources, release requests that were made during preparation etc.

##### Parameters:

*error* If true then action is taken depending on the error.

Reimplemented from [Arc::DataPoint](#).

#### 4.8.2.8 virtual bool Arc::DataPointIndex::NextLocation () [virtual]

Switch to next location in list of URLs. At last location switch to first if number of allowed retries is not exceeded. Returns false if no retries left.

Implements [Arc::DataPoint](#).

#### 4.8.2.9 virtual void Arc::DataPointIndex::Passive (bool *v*) [virtual]

Request passive transfers for FTP-like protocols.

##### Parameters:

*true* to request.

Implements [Arc::DataPoint](#).

#### 4.8.2.10 **virtual DataStatus Arc::DataPointIndex::PrepareReading (unsigned int *timeout*, unsigned int & *wait\_time*) [virtual]**

Prepare [DataPoint](#) for reading. This method should be implemented by protocols which require preparation or staging of physical files for reading. It can act synchronously or asynchronously (if protocol supports it). In the first case the method will block until the file is prepared or the specified timeout has passed. In the second case the method can return with a ReadPrepareWait status before the file is prepared. The caller should then wait some time (a hint from the remote service may be given in *wait\_time*) and call [PrepareReading\(\)](#) again to poll for the preparation status, until the file is prepared. In this case it is also up to the caller to decide when the request has taken too long and if so cancel it by calling [FinishReading\(\)](#). When file preparation has finished, the physical file(s) to read from can be found from [TransferLocations\(\)](#).

##### Parameters:

***timeout*** If non-zero, this method will block until either the file has been prepared successfully or the timeout has passed. A zero value means that the caller would like to call and poll for status.

***wait\_time*** If timeout is zero (caller would like asynchronous operation) and ReadPrepareWait is returned, a hint for how long to wait before a subsequent call may be given in *wait\_time*.

Reimplemented from [Arc::DataPoint](#).

#### 4.8.2.11 **virtual DataStatus Arc::DataPointIndex::PrepareWriting (unsigned int *timeout*, unsigned int & *wait\_time*) [virtual]**

Prepare [DataPoint](#) for writing. This method should be implemented by protocols which require preparation of physical files for writing. It can act synchronously or asynchronously (if protocol supports it). In the first case the method will block until the file is prepared or the specified timeout has passed. In the second case the method can return with a WritePrepareWait status before the file is prepared. The caller should then wait some time (a hint from the remote service may be given in *wait\_time*) and call [PrepareWriting\(\)](#) again to poll for the preparation status, until the file is prepared. In this case it is also up to the caller to decide when the request has taken too long and if so cancel or abort it by calling [FinishWriting\(true\)](#). When file preparation has finished, the physical file(s) to write to can be found from [TransferLocations\(\)](#).

##### Parameters:

***timeout*** If non-zero, this method will block until either the file has been prepared successfully or the timeout has passed. A zero value means that the caller would like to call and poll for status.

***wait\_time*** If timeout is zero (caller would like asynchronous operation) and WritePrepareWait is returned, a hint for how long to wait before a subsequent call may be given in *wait\_time*.

Reimplemented from [Arc::DataPoint](#).

#### 4.8.2.12 **virtual bool Arc::DataPointIndex::ProvidesMeta () const [virtual]**

If endpoint can provide at least some meta information directly.

Implements [Arc::DataPoint](#).

#### 4.8.2.13 **virtual void Arc::DataPointIndex::Range (unsigned long long int *start* = 0, unsigned long long int *end* = 0) [virtual]**

Set range of bytes to retrieve. Default values correspond to whole file.

Implements [Arc::DataPoint](#).

**4.8.2.14 virtual void Arc::DataPointIndex::ReadOutOfOrder (bool *v*) [virtual]**

Allow/disallow [DataPoint](#) to produce scattered data during reading\* operation.

**Parameters:**

*v* true if allowed (default is false).

Implements [Arc::DataPoint](#).

**4.8.2.15 virtual bool Arc::DataPointIndex::Registered () const [virtual]**

Check if file is registered in Indexing Service. Proper value is obtainable only after Resolve.

Implements [Arc::DataPoint](#).

**4.8.2.16 virtual void Arc::DataPointIndex::SetAdditionalChecks (bool *v*) [virtual]**

Allow/disallow additional checks. Check for existence of remote file (and probably other checks too) before initiating reading and writing operations.

**Parameters:**

*v* true if allowed (default is true).

Implements [Arc::DataPoint](#).

**4.8.2.17 virtual void Arc::DataPointIndex::SetSecure (bool *v*) [virtual]**

Allow/disallow heavy security during data transfer.

**Parameters:**

*v* true if allowed (default depends on protocol).

Implements [Arc::DataPoint](#).

**4.8.2.18 virtual void Arc::DataPointIndex::SortLocations (const std::string & *pattern*, const URLMap & *url\_map*) [virtual]**

Sort locations according to the specified pattern.

**Parameters:**

*pattern* a set of strings, separated by |, to match against.

Implements [Arc::DataPoint](#).

**4.8.2.19 virtual DataStatus Arc::DataPointIndex::StartReading (DataBuffer & *buffer*) [virtual]**

Start reading data from URL. Separate thread to transfer data will be created. No other operation can be performed while reading is in progress.

**Parameters:**

*buffer* operation will use this buffer to put information into. Should not be destroyed before [StopReading\(\)](#) was called and returned.

Implements [Arc::DataPoint](#).

#### 4.8.2.20 **virtual DataStatus Arc::DataPointIndex::StartWriting (DataBuffer & *buffer*, DataCallback \* *space\_cb* = NULL) [virtual]**

Start writing data to URL. Separate thread to transfer data will be created. No other operation can be performed while writing is in progress.

**Parameters:**

*buffer* operation will use this buffer to get information from. Should not be destroyed before *stop\_*-writing was called and returned.

*space\_cb* callback which is called if there is not enough space to store data. May not implemented for all protocols.

Implements [Arc::DataPoint](#).

#### 4.8.2.21 **virtual DataStatus Arc::DataPointIndex::StopReading () [virtual]**

Stop reading. Must be called after corresponding *start\_reading* method, either after all data is transferred or to cancel transfer. Use buffer object to find out when data is transferred. Must return failure if any happened during transfer.

Implements [Arc::DataPoint](#).

#### 4.8.2.22 **virtual DataStatus Arc::DataPointIndex::StopWriting () [virtual]**

Stop writing. Must be called after corresponding *start\_writing* method, either after all data is transferred or to cancel transfer. Use buffer object to find out when data is transferred. Must return failure if any happened during transfer.

Implements [Arc::DataPoint](#).

#### 4.8.2.23 **virtual std::vector<URL> Arc::DataPointIndex::TransferLocations () const [virtual]**

Returns physical file(s) to read/write, if different from [CurrentLocation\(\)](#). To be used with protocols which re-direct to different URLs such as Transport URLs (TURLs). The list is initially filled by *PrepareReading* and *PrepareWriting*. If this list is non-empty then real transfer should use a URL from this list. It is up to the caller to choose the best URL and instantiate new [DataPoint](#) for handling it. For consistency protocols which do not require redirections return original URL. For protocols which need redirection calling *StartReading* and *StartWriting* will use first URL in the list.

Reimplemented from [Arc::DataPoint](#).

#### 4.8.2.24 **virtual bool Arc::DataPointIndex::WriteOutOfOrder () [virtual]**

Returns true if URL can accept scattered data for *\*writing\** operation.

Implements [Arc::DataPoint](#).

The documentation for this class was generated from the following file:

- DataPointIndex.h

## 4.9 Arc::DataPointLoader Class Reference

Class used by [DataHandle](#) to load the required DMC.

```
#include <DataPoint.h>
```

### 4.9.1 Detailed Description

Class used by [DataHandle](#) to load the required DMC.

The documentation for this class was generated from the following file:

- DataPoint.h

## 4.10 Arc::DataPointPluginArgument Class Reference

Class representing the arguments passed to DMC plugins.

```
#include <DataPoint.h>
```

### 4.10.1 Detailed Description

Class representing the arguments passed to DMC plugins.

The documentation for this class was generated from the following file:

- DataPoint.h

## 4.11 Arc::DataSpeed Class Reference

Keeps track of average and instantaneous transfer speed.

```
#include <DataSpeed.h>
```

### Public Member Functions

- [DataSpeed](#) (time\_t base=DATASPEED\_AVERAGING\_PERIOD)
- [DataSpeed](#) (unsigned long long int min\_speed, time\_t min\_speed\_time, unsigned long long int min\_average\_speed, time\_t max\_inactivity\_time, time\_t base=DATASPEED\_AVERAGING\_PERIOD)
- [~DataSpeed](#) (void)
- void [verbose](#) (bool val)
- void [verbose](#) (const std::string &prefix)
- bool [verbose](#) (void)
- void [set\\_min\\_speed](#) (unsigned long long int min\_speed, time\_t min\_speed\_time)
- void [set\\_min\\_average\\_speed](#) (unsigned long long int min\_average\_speed)
- void [set\\_max\\_inactivity\\_time](#) (time\_t max\_inactivity\_time)
- time\_t [get\\_max\\_inactivity\\_time](#) ()
- void [set\\_base](#) (time\_t base\_=DATASPEED\_AVERAGING\_PERIOD)
- void [set\\_max\\_data](#) (unsigned long long int max=0)
- void [set\\_progress\\_indicator](#) (show\_progress\_t func=NULL)
- void [reset](#) (void)
- bool [transfer](#) (unsigned long long int n=0)
- void [hold](#) (bool disable)
- bool [min\\_speed\\_failure](#) ()
- bool [min\\_average\\_speed\\_failure](#) ()
- bool [max\\_inactivity\\_time\\_failure](#) ()
- unsigned long long int [transferred\\_size](#) (void)

### 4.11.1 Detailed Description

Keeps track of average and instantaneous transfer speed. Also detects data transfer inactivity and other transfer timeouts.

### 4.11.2 Constructor & Destructor Documentation

#### 4.11.2.1 Arc::DataSpeed::DataSpeed (time\_t *base* = DATASPEED\_AVERAGING\_PERIOD)

Constructor

**Parameters:**

*base* time period used to average values (default 1 minute).



#### 4.11.2.2 Arc::DataSpeed::DataSpeed (unsigned long long int *min\_speed*, time\_t *min\_speed\_time*, unsigned long long int *min\_average\_speed*, time\_t *max\_inactivity\_time*, time\_t *base* = DATASPEED\_AVERAGING\_PERIOD)

Constructor

##### Parameters:

*base* time period used to average values (default 1 minute).

*min\_speed* minimal allowed speed (Butes per second). If speed drops and holds below threshold for *min\_speed\_time*\_seconds error is triggered.

*min\_speed\_time*

*min\_average\_speed* minimal average speed (Bytes per second) to trigger error. Averaged over whole current transfer time.

*max\_inactivity\_time* - if no data is passing for specified amount of time (seconds), error is triggered.

### 4.11.3 Member Function Documentation

#### 4.11.3.1 void Arc::DataSpeed::hold (bool *disable*)

Turn off speed control.

##### Parameters:

*disable* true to turn off.

#### 4.11.3.2 void Arc::DataSpeed::set\_base (time\_t *base* = DATASPEED\_AVERAGING\_PERIOD)

Set averaging time period.

##### Parameters:

*base* time period used to average values (default 1 minute).

#### 4.11.3.3 void Arc::DataSpeed::set\_max\_data (unsigned long long int *max* = 0)

Set amount of data to be transferred. Used in verbose messages.

##### Parameters:

*max* amount of data in bytes.

#### 4.11.3.4 void Arc::DataSpeed::set\_max\_inactivity\_time (time\_t *max\_inactivity\_time*)

Set inactivity tiemout.

##### Parameters:

*max\_inactivity\_time* - if no data is passing for specified amount of time (seconds), error is triggered.

#### 4.11.3.5 void Arc::DataSpeed::set\_min\_average\_speed (unsigned long long int *min\_average\_speed*)

Set minimal average speed.

##### Parameters:

*min\_average\_speed* minimal average speed (Bytes per second) to trigger error. Averaged over whole current transfer time.

#### 4.11.3.6 void Arc::DataSpeed::set\_min\_speed (unsigned long long int *min\_speed*, time\_t *min\_speed\_time*)

Set minimal allowed speed.

##### Parameters:

*min\_speed* minimal allowed speed (Bytes per second). If speed drops and holds below threshold for *min\_speed\_time* seconds error is triggered.

*min\_speed\_time*

#### 4.11.3.7 void Arc::DataSpeed::set\_progress\_indicator (show\_progress\_t *func* = NULL)

Specify which external function will print verbose messages. If not specified internal one is used.

##### Parameters:

*pointer* to function which prints information.

#### 4.11.3.8 bool Arc::DataSpeed::transfer (unsigned long long int *n* = 0)

Inform object, about amount of data has been transferred. All errors are triggered by this method. To make them work application must call this method periodically even with zero value.

##### Parameters:

*n* amount of data transferred (bytes).

#### 4.11.3.9 void Arc::DataSpeed::verbose (const std::string & *prefix*)

Print information about current speed and amount of data.

##### Parameters:

*'prefix'* add this string at the beginning of every string.

#### 4.11.3.10 void Arc::DataSpeed::verbose (bool *val*)

Activate printing information about current time speeds, amount of transferred data.

The documentation for this class was generated from the following file:

- DataSpeed.h

## 4.12 Arc::DataStatus Class Reference

Status code returned by many [DataPoint](#) methods.

```
#include <DataStatus.h>
```

### Public Types

- enum [DataStatusType](#) {
  - [Success](#) = 0, [ReadAcquireError](#) = 1, [WriteAcquireError](#) = 2, [ReadResolveError](#) = 3,
  - [WriteResolveError](#) = 4, [ReadStartError](#) = 5, [WriteStartError](#) = 6, [ReadError](#) = 7,
  - [WriteError](#) = 8, [TransferError](#) = 9, [ReadStopError](#) = 10, [WriteStopError](#) = 11,
  - [PreRegisterError](#) = 12, [PostRegisterError](#) = 13, [UnregisterError](#) = 14, [CacheError](#) = 15,
  - [CredentialsExpiredError](#) = 16, [DeleteError](#) = 17, [NoLocationError](#) = 18, [LocationAlreadyExistsError](#) = 19,
  - [NotSupportedForDirectDataPointsError](#) = 20, [UnimplementedError](#) = 21, [IsReadingError](#) = 22,
  - [IsWritingError](#) = 23,
  - [CheckError](#) = 24, [ListError](#) = 25, [StatError](#) = 27, [NotInitializedError](#) = 29,
  - [SystemError](#) = 30, [StageError](#) = 31, [InconsistentMetadataError](#) = 32, [ReadPrepareError](#) = 32,
  - [ReadPrepareWait](#) = 33, [WritePrepareError](#) = 34, [WritePrepareWait](#) = 35, [ReadFinishError](#) = 36,
  - [WriteFinishError](#) = 37, [CreateDirectoryError](#) = 38, [SuccessCached](#) = 39, [GenericError](#) = 40,
  - [UnknownError](#) = 41 }

### Public Member Functions

- bool [Passed](#) () const
- bool [Retryable](#) () const
- void [SetDesc](#) (const std::string &d)
- std::string [GetDesc](#) () const

#### 4.12.1 Detailed Description

Status code returned by many [DataPoint](#) methods. A class to be used for return types of all major data handling methods. It describes the outcome of the method.

#### 4.12.2 Member Enumeration Documentation

##### 4.12.2.1 enum Arc::DataStatus::DataStatusType

Status codes.

##### Enumerator:

**Success** Operation completed successfully.

**ReadAcquireError** Source is bad URL or can't be used due to some reason.

**WriteAcquireError** Destination is bad URL or can't be used due to some reason.

**ReadResolveError** Resolving of index service URL for source failed.

**WriteResolveError** Resolving of index service URL for destination failed.

**ReadStartError** Can't read from source.

**WriteStartError** Can't write to destination.

**ReadError** Failed while reading from source.

**WriteError** Failed while writing to destination.

**TransferError** Failed while transferring data (mostly timeout).

**ReadStopError** Failed while finishing reading from source.

**WriteStopError** Failed while finishing writing to destination.

**PreRegisterError** First stage of registration of index service URL failed.

**PostRegisterError** Last stage of registration of index service URL failed.

**UnregisterError** Unregistration of index service URL failed.

**CacheError** Error in caching procedure.

**CredentialsExpiredError** Error due to provided credentials are expired.

**DeleteError** Error deleting location or URL.

**NoLocationError** No valid location available.

**LocationAlreadyExistsError** No valid location available.

**NotSupportedForDirectDataPointsError** Operation has no sense for this kind of URL.

**UnimplementedError** Feature is unimplemented.

**IsReadingError** [DataPoint](#) is already reading.

**IsWritingError** [DataPoint](#) is already writing.

**CheckError** Access check failed.

**ListError** File listing failed.

**StatError** File/dir stating failed.

**NotInitializedError** Object initialization failed.

**SystemError** Error in OS.

**StageError** Staging error.

**InconsistentMetadataError** Inconsistent metadata.

**ReadPrepareError** Can't prepare source.

**ReadPrepareWait** Wait for source to be prepared.

**WritePrepareError** Can't prepare destination.

**WritePrepareWait** Wait for destination to be prepared.

**ReadFinishError** Can't finish source.

**WriteFinishError** Can't finish destination.

**CreateDirectoryError** Can't create directory.

**SuccessCached** Data was already cached.

**GenericError** General error which doesn't fit any other error.

**UnknownError** Undefined.

The documentation for this class was generated from the following file:

- [DataStatus.h](#)

## 4.13 Arc::FileCache Class Reference

[FileCache](#) provides an interface to all cache operations.

```
#include <FileCache.h>
```

### Public Member Functions

- [FileCache](#) (const std::string &cache\_path, const std::string &id, uid\_t job\_uid, gid\_t job\_gid)
- [FileCache](#) (const std::vector< std::string > &caches, const std::string &id, uid\_t job\_uid, gid\_t job\_gid)
- [FileCache](#) (const std::vector< std::string > &caches, const std::vector< std::string > &remote\_caches, const std::vector< std::string > &draining\_caches, const std::string &id, uid\_t job\_uid, gid\_t job\_gid)
- [FileCache](#) ()
- bool [Start](#) (const std::string &url, bool &available, bool &is\_locked, bool use\_remote=true, bool delete\_first=false)
- bool [Stop](#) (const std::string &url)
- bool [StopAndDelete](#) (const std::string &url)
- std::string [File](#) (const std::string &url)
- bool [Link](#) (const std::string &link\_path, const std::string &url, bool copy, bool executable, bool holding\_lock, bool &try\_again)
- bool [Release](#) () const
- bool [AddDN](#) (const std::string &url, const std::string &DN, const Time &expiry\_time)
- bool [CheckDN](#) (const std::string &url, const std::string &DN)
- bool [CheckCreated](#) (const std::string &url)
- Time [GetCreated](#) (const std::string &url)
- bool [CheckValid](#) (const std::string &url)
- Time [GetValid](#) (const std::string &url)
- bool [SetValid](#) (const std::string &url, const Time &val)
- [operator bool](#) ()
- bool [operator==](#) (const [FileCache](#) &a)

### 4.13.1 Detailed Description

[FileCache](#) provides an interface to all cache operations. When it is decided a file should be downloaded to the cache, [Start\(\)](#) should be called, so that the cache file can be prepared and locked if necessary. If the file is already available it is not locked and [Link\(\)](#) can be called immediately to create a hard link to a per-job directory in the cache and then soft link, or copy the file directly to the session directory so it can be accessed from the user's job. If the file is not available, [Start\(\)](#) will lock it, then after downloading [Link\(\)](#) can be called. [Stop\(\)](#) must then be called to release the lock. If the transfer failed, [StopAndDelete\(\)](#) can be called to clean up the cache file. After a job has finished, [Release\(\)](#) should be called to remove the hard links created for that job.

Cache files are locked for writing using the FileLock class, which creates a lock file with the '.lock' suffix next to the cache file. If [Start\(\)](#) is called and the cache file is not already available, it creates this lock and [Stop\(\)](#) must be called to release it. All processes calling [Start\(\)](#) must wait until they successfully obtain the lock before downloading can begin.

The cache directory(ies) and the optional directory to link to when the soft-links are made are set in the constructor. The names of cache files are formed from an SHA-1 hash of the URL to cache. To

ease the load on the file system, the cache files are split into subdirectories based on the first two characters in the hash. For example the file with hash 76f11edda169848038efbd9fa3df5693 is stored in 76/f11edda169848038efbd9fa3df5693. A cache filename can be found by passing the URL to Find(). For more information on the structure of the cache, see the ARC Computing Element System Administrator Guide (NORDUGRID-MANUAL-20).

### 4.13.2 Constructor & Destructor Documentation

#### 4.13.2.1 Arc::FileCache::FileCache (const std::string & *cache\_path*, const std::string & *id*, uid\_t *job\_uid*, gid\_t *job\_gid*)

Create a new [FileCache](#) instance.

##### Parameters:

*cache\_path* The format is "cache\_dir[ link\_path]". path is the path to the cache directory and the optional link\_path is used to create a link in case the cache directory is visible under a different name during actual usage. When linking from the session dir this path is used instead of cache\_path.

*id* the job id. This is used to create the per-job dir which the job's cache files will be hard linked from

*job\_uid* owner of job. The per-job dir will only be readable by this user

*job\_gid* owner group of job

#### 4.13.2.2 Arc::FileCache::FileCache (const std::vector< std::string > & *caches*, const std::string & *id*, uid\_t *job\_uid*, gid\_t *job\_gid*)

Create a new [FileCache](#) instance with multiple cache dirs

##### Parameters:

*caches* a vector of strings describing caches. The format of each string is "cache\_dir[ link\_path]".

*id* the job id. This is used to create the per-job dir which the job's cache files will be hard linked from

*job\_uid* owner of job. The per-job dir will only be readable by this user

*job\_gid* owner group of job

#### 4.13.2.3 Arc::FileCache::FileCache (const std::vector< std::string > & *caches*, const std::vector< std::string > & *remote\_caches*, const std::vector< std::string > & *draining\_caches*, const std::string & *id*, uid\_t *job\_uid*, gid\_t *job\_gid*)

Create a new [FileCache](#) instance with multiple cache dirs, remote caches and draining cache directories.

##### Parameters:

*caches* a vector of strings describing caches. The format of each string is "cache\_dir[ link\_path]".

*remote\_caches* Same format as caches. These are the paths to caches which are under the control of other Grid Managers and are read-only for this process.

*draining\_caches* Same format as caches. These are the paths to caches which are to be drained.

*id* the job id. This is used to create the per-job dir which the job's cache files will be hard linked from

*job\_uid* owner of job. The per-job dir will only be readable by this user

*job\_gid* owner group of job

### 4.13.3 Member Function Documentation

#### 4.13.3.1 `bool Arc::FileCache::AddDN (const std::string & url, const std::string & DN, const Time & expiry_time)`

Store a DN in the permissions cache for the given url. Add the given DN to the list of cached DNs with the given expiry time.

**Parameters:**

*url* the url corresponding to the cache file to which we want to add a cached DN

*DN* the DN of the user

*expiry\_time* the expiry time of this DN in the DN cache

#### 4.13.3.2 `bool Arc::FileCache::CheckCreated (const std::string & url)`

Check if it is possible to obtain the creation time of a cache file. Returns true if the file exists in the cache, since the creation time is the creation time of the cache file.

**Parameters:**

*url* the url corresponding to the cache file for which we want to know if the creation date exists

#### 4.13.3.3 `bool Arc::FileCache::CheckDN (const std::string & url, const std::string & DN)`

Check if a DN exists in the permission cache for the given url. Check if the given DN is cached for authorisation.

**Parameters:**

*url* the url corresponding to the cache file for which we want to check the cached DN

*DN* the DN of the user

#### 4.13.3.4 `bool Arc::FileCache::CheckValid (const std::string & url)`

Check if there is an expiry time of the given url in the cache.

**Parameters:**

*url* the url corresponding to the cache file for which we want to know if the expiration time exists

#### 4.13.3.5 `std::string Arc::FileCache::File (const std::string & url)`

Get the cache filename for the given URL. Returns the full pathname of the file in the cache which corresponds to the given url.

**Parameters:**

*url* the URL to look for in the cache

#### 4.13.3.6 Time Arc::FileCache::GetCreated (const std::string & url)

Get the creation time of a cached file. If the cache file does not exist, 0 is returned.

##### Parameters:

*url* the url corresponding to the cache file for which we want to know the creation date

#### 4.13.3.7 Time Arc::FileCache::GetValid (const std::string & url)

Get expiry time of a cached file. If the time is not available, a time equivalent to 0 is returned.

##### Parameters:

*url* the url corresponding to the cache file for which we want to know the expiry time

#### 4.13.3.8 bool Arc::FileCache::Link (const std::string & link\_path, const std::string & url, bool copy, bool executable, bool holding\_lock, bool & try\_again)

Link a cache file to the place it will be used. Create a hard-link to the per-job dir from the cache dir, and then a soft-link from here to the session directory. This is effectively 'claiming' the file for the job, so even if the original cache file is deleted, eg by some external process, the hard link still exists until it is explicitly released by calling [Release\(\)](#).

If cache\_link\_path is set to "." or copy or executable is true then files will be copied directly to the session directory rather than linked.

After linking or copying, the cache file is checked for the presence of a write lock, and whether the modification time has changed since linking started (in case the file was locked, modified then released during linking). If either of these are true the links created during [Link\(\)](#) are deleted and try\_again is set to true. The caller should then go back to [Start\(\)](#). If the caller has obtained a write lock from [Start\(\)](#) and then downloaded the file, it should set holding\_lock to true, in which case none of the above checks are performed.

The session directory is accessed under the uid and gid passed in the constructor.

##### Parameters:

*link\_path* path to the session dir for soft-link or new file

*url* url of file to link to or copy

*copy* If true the file is copied rather than soft-linked to the session dir

*executable* If true then file is copied and given execute permissions in the session dir

*holding\_lock* Should be set to true if the caller already holds the lock

*try\_again* If after linking the cache file was found to be locked, deleted or modified, then try\_again is set to true

#### 4.13.3.9 bool Arc::FileCache::Release () const

Release cache files used in this cache. Release claims on input files for the job specified by id. For each cache directory the per-job directory with the hard-links will be deleted.



#### 4.13.3.10 bool Arc::FileCache::SetValid (const std::string & *url*, const Time & *val*)

Set expiry time of a cache file.

**Parameters:**

*url* the url corresponding to the cache file for which we want to set the expiry time  
*val* expiry time

#### 4.13.3.11 bool Arc::FileCache::Start (const std::string & *url*, bool & *available*, bool & *is\_locked*, bool *use\_remote* = true, bool *delete\_first* = false)

Start preparing to cache the file specified by url. [Start\(\)](#) returns true if the file was successfully prepared. The available parameter is set to true if the file already exists and in this case [Link\(\)](#) can be called immediately. If available is false the caller should write the file and then call [Link\(\)](#) followed by [Stop\(\)](#). It returns false if it was unable to prepare the cache file for any reason. In this case the is\_locked parameter should be checked and if it is true the file is locked by another process and the caller should try again later.

**Parameters:**

*url* url that is being downloaded  
*available* true on exit if the file is already in cache  
*is\_locked* true on exit if the file is already locked, ie cannot be used by this process  
*use\_remote* Whether to look to see if the file exists in a remote cache. Can be set to false if for example a forced download to cache is desired.  
*delete\_first* If true then any existing cache file is deleted.

#### 4.13.3.12 bool Arc::FileCache::Stop (const std::string & *url*)

Stop the cache after a file was downloaded. This method (or stopAndDelete) must be called after file was downloaded or download failed, to release the lock on the cache file. [Stop\(\)](#) does not delete the cache file. It returns false if the lock file does not exist, or another pid was found inside the lock file (this means another process took over the lock so this process must go back to [Start\(\)](#)), or if it fails to delete the lock file. It must only be called if the caller holds the writing lock.

**Parameters:**

*url* the url of the file that was downloaded

#### 4.13.3.13 bool Arc::FileCache::StopAndDelete (const std::string & *url*)

Stop the cache after a file was downloaded and delete the cache file. Release the cache file and delete it, because for example a failed download left an incomplete copy. This method also deletes the meta file which contains the url corresponding to the cache file. The logic of the return value is the same as [Stop\(\)](#). It must only be called if the caller holds the writing lock.

**Parameters:**

*url* the url corresponding to the cache file that has to be released and deleted

The documentation for this class was generated from the following file:

- FileCache.h

## 4.14 Arc::FileCacheHash Class Reference

[FileCacheHash](#) provides methods to make hashes from strings.

```
#include <FileCacheHash.h>
```

### Static Public Member Functions

- static std::string [getHash](#) (std::string url)
- static int [maxLength](#) ()

#### 4.14.1 Detailed Description

[FileCacheHash](#) provides methods to make hashes from strings. Currently the SHA-1 hash from the openssl library is used.

The documentation for this class was generated from the following file:

- FileCacheHash.h

## 4.15 Arc::FileInfo Class Reference

[FileInfo](#) stores information about files (metadata).

```
#include <FileInfo.h>
```

### 4.15.1 Detailed Description

[FileInfo](#) stores information about files (metadata).

The documentation for this class was generated from the following file:

- [FileInfo.h](#)

## 4.16 Arc::URLMap Class Reference

### Data Structures

- class `map_entry`

The documentation for this class was generated from the following file:

- `URLMap.h`

# Index

ACCESS\_LATENCY\_LARGE  
    Arc::DataPoint, 24  
ACCESS\_LATENCY\_SMALL  
    Arc::DataPoint, 24  
ACCESS\_LATENCY\_ZERO  
    Arc::DataPoint, 24  
add  
    Arc::DataBuffer, 9  
AddChecksumObject  
    Arc::DataPoint, 25  
    Arc::DataPointDirect, 35  
    Arc::DataPointIndex, 40  
AddDN  
    Arc::FileCache, 55  
AddLocation  
    Arc::DataPoint, 25  
    Arc::DataPointDirect, 35  
    Arc::DataPointIndex, 40  
AddURLOptions  
    Arc::DataPoint, 25  
Arc::CacheParameters, 7  
Arc::DataBuffer, 8  
    add, 9  
    buffer\_size, 9  
    checksum\_object, 10  
    checksum\_valid, 10  
    DataBuffer, 9  
    eof\_read, 10  
    eof\_write, 10  
    error, 10  
    error\_read, 10  
    error\_write, 10  
    for\_read, 11  
    for\_write, 11  
    is\_notwritten, 11, 12  
    is\_read, 12  
    is\_written, 12  
    set, 13  
    wait\_any, 13  
Arc::DataCallback, 14  
Arc::DataHandle, 15  
    GetPoint, 17  
Arc::DataMover, 18  
    checks, 18  
    force\_to\_meta, 19  
    secure, 19  
    set\_default\_max\_inactivity\_time, 19  
    set\_default\_min\_average\_speed, 19  
    set\_default\_min\_speed, 19  
    Transfer, 19  
    verbose, 20  
Arc::DataPoint, 21  
    ACCESS\_LATENCY\_LARGE, 24  
    ACCESS\_LATENCY\_SMALL, 24  
    ACCESS\_LATENCY\_ZERO, 24  
    AddChecksumObject, 25  
    AddLocation, 25  
    AddURLOptions, 25  
    Check, 25  
    CompareLocationMetadata, 25  
    CompareMeta, 26  
    CreateDirectory, 26  
    CurrentLocationMetadata, 26  
    DataPoint, 25  
    DataPointAccessLatency, 24  
    DataPointInfoType, 24  
    FinishReading, 26  
    FinishWriting, 26  
    GetFailureReason, 27  
    INFO\_TYPE\_ACCESS, 24  
    INFO\_TYPE\_ALL, 24  
    INFO\_TYPE\_CONTENT, 24  
    INFO\_TYPE\_MINIMAL, 24  
    INFO\_TYPE\_NAME, 24  
    INFO\_TYPE\_REST, 24  
    INFO\_TYPE\_STRUCT, 24  
    INFO\_TYPE\_TIMES, 24  
    INFO\_TYPE\_TYPE, 24  
    List, 27  
    NextLocation, 27  
    Passive, 27  
    PostRegister, 27  
    PrepareReading, 27  
    PrepareWriting, 28  
    PreRegister, 28  
    PreUnregister, 29  
    ProvidesMeta, 29  
    Range, 29  
    ReadOutOfOrder, 29  
    Registered, 29

- Resolve, [29, 30](#)
- SetAdditionalChecks, [30](#)
- SetMeta, [30](#)
- SetSecure, [30](#)
- SetURL, [30](#)
- SortLocations, [30](#)
- StartReading, [31](#)
- StartWriting, [31](#)
- Stat, [31, 32](#)
- StopReading, [32](#)
- StopWriting, [32](#)
- TransferLocations, [32](#)
- Unregister, [32](#)
- valid\_url\_options, [33](#)
- WriteOutOfOrder, [33](#)
- Arc::DataPointDirect, [34](#)
  - AddChecksumObject, [35](#)
  - AddLocation, [35](#)
  - CompareLocationMetadata, [35](#)
  - CurrentLocationMetadata, [35](#)
  - NextLocation, [35](#)
  - Passive, [36](#)
  - PostRegister, [36](#)
  - PreRegister, [36](#)
  - PreUnregister, [36](#)
  - ProvidesMeta, [36](#)
  - Range, [37](#)
  - ReadOutOfOrder, [37](#)
  - Registered, [37](#)
  - Resolve, [37](#)
  - SetAdditionalChecks, [37](#)
  - SetSecure, [37](#)
  - SortLocations, [38](#)
  - Unregister, [38](#)
  - WriteOutOfOrder, [38](#)
- Arc::DataPointIndex, [39](#)
  - AddChecksumObject, [40](#)
  - AddLocation, [40](#)
  - Check, [40](#)
  - CompareLocationMetadata, [40](#)
  - CurrentLocationMetadata, [41](#)
  - FinishReading, [41](#)
  - FinishWriting, [41](#)
  - NextLocation, [41](#)
  - Passive, [41](#)
  - PrepareReading, [41](#)
  - PrepareWriting, [42](#)
  - ProvidesMeta, [42](#)
  - Range, [42](#)
  - ReadOutOfOrder, [42](#)
  - Registered, [43](#)
  - SetAdditionalChecks, [43](#)
  - SetSecure, [43](#)
  - SortLocations, [43](#)
  - StartReading, [43](#)
  - StartWriting, [44](#)
  - StopReading, [44](#)
  - StopWriting, [44](#)
  - TransferLocations, [44](#)
  - WriteOutOfOrder, [44](#)
- Arc::DataPointLoader, [46](#)
- Arc::DataPointPluginArgument, [47](#)
- Arc::DataSpeed, [48](#)
  - DataSpeed, [48](#)
  - hold, [49](#)
  - set\_base, [49](#)
  - set\_max\_data, [49](#)
  - set\_max\_inactivity\_time, [49](#)
  - set\_min\_average\_speed, [49](#)
  - set\_min\_speed, [50](#)
  - set\_progress\_indicator, [50](#)
  - transfer, [50](#)
  - verbose, [50](#)
- Arc::DataStatus, [51](#)
  - CacheError, [52](#)
  - CheckError, [52](#)
  - CreateDirectoryError, [52](#)
  - CredentialsExpiredError, [52](#)
  - DataStatusType, [51](#)
  - DeleteError, [52](#)
  - GenericError, [52](#)
  - InconsistentMetadataError, [52](#)
  - IsReadingError, [52](#)
  - IsWritingError, [52](#)
  - ListError, [52](#)
  - LocationAlreadyExistsError, [52](#)
  - NoLocationError, [52](#)
  - NotInitializedError, [52](#)
  - NotSupportedForDirectDataPointsError, [52](#)
  - PostRegisterError, [52](#)
  - PreRegisterError, [52](#)
  - ReadAcquireError, [51](#)
  - ReadError, [52](#)
  - ReadFinishError, [52](#)
  - ReadPrepareError, [52](#)
  - ReadPrepareWait, [52](#)
  - ReadResolveError, [51](#)
  - ReadStartError, [52](#)
  - ReadStopError, [52](#)
  - StageError, [52](#)
  - StatError, [52](#)
  - Success, [51](#)
  - SuccessCached, [52](#)
  - SystemError, [52](#)
  - TransferError, [52](#)
  - UnimplementedError, [52](#)
  - UnknownError, [52](#)
  - UnregisterError, [52](#)

- WriteAcquireError, [51](#)
- WriteError, [52](#)
- WriteFinishError, [52](#)
- WritePrepareError, [52](#)
- WritePrepareWait, [52](#)
- WriteResolveError, [51](#)
- WriteStartError, [52](#)
- WriteStopError, [52](#)
- Arc::FileCache, [53](#)
  - AddDN, [55](#)
  - CheckCreated, [55](#)
  - CheckDN, [55](#)
  - CheckValid, [55](#)
  - File, [55](#)
  - FileCache, [54](#)
  - GetCreated, [55](#)
  - GetValid, [56](#)
  - Link, [56](#)
  - Release, [56](#)
  - SetValid, [56](#)
  - Start, [57](#)
  - Stop, [57](#)
  - StopAndDelete, [57](#)
- Arc::FileCacheHash, [58](#)
- Arc::FileInfo, [59](#)
- Arc::URLMap, [60](#)
- buffer\_size
  - Arc::DataBuffer, [9](#)
- CacheError
  - Arc::DataStatus, [52](#)
- Check
  - Arc::DataPoint, [25](#)
  - Arc::DataPointIndex, [40](#)
- CheckCreated
  - Arc::FileCache, [55](#)
- CheckDN
  - Arc::FileCache, [55](#)
- CheckError
  - Arc::DataStatus, [52](#)
- checks
  - Arc::DataMover, [18](#)
- checksum\_object
  - Arc::DataBuffer, [10](#)
- checksum\_valid
  - Arc::DataBuffer, [10](#)
- CheckValid
  - Arc::FileCache, [55](#)
- CompareLocationMetadata
  - Arc::DataPoint, [25](#)
  - Arc::DataPointDirect, [35](#)
  - Arc::DataPointIndex, [40](#)
- CompareMeta
  - Arc::DataPoint, [26](#)
- CreateDirectory
  - Arc::DataPoint, [26](#)
- CreateDirectoryError
  - Arc::DataStatus, [52](#)
- CredentialsExpiredError
  - Arc::DataStatus, [52](#)
- CurrentLocationMetadata
  - Arc::DataPoint, [26](#)
  - Arc::DataPointDirect, [35](#)
  - Arc::DataPointIndex, [41](#)
- DataBuffer
  - Arc::DataBuffer, [9](#)
- DataPoint
  - Arc::DataPoint, [25](#)
- DataPointAccessLatency
  - Arc::DataPoint, [24](#)
- DataPointInfoType
  - Arc::DataPoint, [24](#)
- DataSpeed
  - Arc::DataSpeed, [48](#)
- DataStatusType
  - Arc::DataStatus, [51](#)
- DeleteError
  - Arc::DataStatus, [52](#)
- eof\_read
  - Arc::DataBuffer, [10](#)
- eof\_write
  - Arc::DataBuffer, [10](#)
- error
  - Arc::DataBuffer, [10](#)
- error\_read
  - Arc::DataBuffer, [10](#)
- error\_write
  - Arc::DataBuffer, [10](#)
- File
  - Arc::FileCache, [55](#)
- FileCache
  - Arc::FileCache, [54](#)
- FinishReading
  - Arc::DataPoint, [26](#)
  - Arc::DataPointIndex, [41](#)
- FinishWriting
  - Arc::DataPoint, [26](#)
  - Arc::DataPointIndex, [41](#)
- for\_read
  - Arc::DataBuffer, [11](#)
- for\_write
  - Arc::DataBuffer, [11](#)
- force\_to\_meta
  - Arc::DataMover, [19](#)

- GenericError
  - Arc::DataStatus, 52
- GetCreated
  - Arc::FileCache, 55
- GetFailureReason
  - Arc::DataPoint, 27
- GetPoint
  - Arc::DataHandle, 17
- GetValid
  - Arc::FileCache, 56
- hold
  - Arc::DataSpeed, 49
- InconsistentMetadataError
  - Arc::DataStatus, 52
- INFO\_TYPE\_ACCESS
  - Arc::DataPoint, 24
- INFO\_TYPE\_ALL
  - Arc::DataPoint, 24
- INFO\_TYPE\_CONTENT
  - Arc::DataPoint, 24
- INFO\_TYPE\_MINIMAL
  - Arc::DataPoint, 24
- INFO\_TYPE\_NAME
  - Arc::DataPoint, 24
- INFO\_TYPE\_REST
  - Arc::DataPoint, 24
- INFO\_TYPE\_STRUCT
  - Arc::DataPoint, 24
- INFO\_TYPE\_TIMES
  - Arc::DataPoint, 24
- INFO\_TYPE\_TYPE
  - Arc::DataPoint, 24
- is\_notwritten
  - Arc::DataBuffer, 11, 12
- is\_read
  - Arc::DataBuffer, 12
- is\_written
  - Arc::DataBuffer, 12
- IsReadingError
  - Arc::DataStatus, 52
- IsWritingError
  - Arc::DataStatus, 52
- Link
  - Arc::FileCache, 56
- List
  - Arc::DataPoint, 27
- ListError
  - Arc::DataStatus, 52
- LocationAlreadyExistsError
  - Arc::DataStatus, 52
- NextLocation
  - Arc::DataPoint, 27
  - Arc::DataPointDirect, 35
  - Arc::DataPointIndex, 41
- NoLocationError
  - Arc::DataStatus, 52
- NotInitializedError
  - Arc::DataStatus, 52
- NotSupportedForDirectDataPointsError
  - Arc::DataStatus, 52
- Passive
  - Arc::DataPoint, 27
  - Arc::DataPointDirect, 36
  - Arc::DataPointIndex, 41
- PostRegister
  - Arc::DataPoint, 27
  - Arc::DataPointDirect, 36
- PostRegisterError
  - Arc::DataStatus, 52
- PrepareReading
  - Arc::DataPoint, 27
  - Arc::DataPointIndex, 41
- PrepareWriting
  - Arc::DataPoint, 28
  - Arc::DataPointIndex, 42
- PreRegister
  - Arc::DataPoint, 28
  - Arc::DataPointDirect, 36
- PreRegisterError
  - Arc::DataStatus, 52
- PreUnregister
  - Arc::DataPoint, 29
  - Arc::DataPointDirect, 36
- ProvidesMeta
  - Arc::DataPoint, 29
  - Arc::DataPointDirect, 36
  - Arc::DataPointIndex, 42
- Range
  - Arc::DataPoint, 29
  - Arc::DataPointDirect, 37
  - Arc::DataPointIndex, 42
- ReadAcquireError
  - Arc::DataStatus, 51
- ReadError
  - Arc::DataStatus, 52
- ReadFinishError
  - Arc::DataStatus, 52
- ReadOutOfOrder
  - Arc::DataPoint, 29
  - Arc::DataPointDirect, 37
  - Arc::DataPointIndex, 42
- ReadPrepareError
  - Arc::DataStatus, 52



- ReadPrepareWait
  - Arc::DataStatus, 52
- ReadResolveError
  - Arc::DataStatus, 51
- ReadStartError
  - Arc::DataStatus, 52
- ReadStopError
  - Arc::DataStatus, 52
- Registered
  - Arc::DataPoint, 29
  - Arc::DataPointDirect, 37
  - Arc::DataPointIndex, 43
- Release
  - Arc::FileCache, 56
- Resolve
  - Arc::DataPoint, 29, 30
  - Arc::DataPointDirect, 37
- secure
  - Arc::DataMover, 19
- set
  - Arc::DataBuffer, 13
- set\_base
  - Arc::DataSpeed, 49
- set\_default\_max\_inactivity\_time
  - Arc::DataMover, 19
- set\_default\_min\_average\_speed
  - Arc::DataMover, 19
- set\_default\_min\_speed
  - Arc::DataMover, 19
- set\_max\_data
  - Arc::DataSpeed, 49
- set\_max\_inactivity\_time
  - Arc::DataSpeed, 49
- set\_min\_average\_speed
  - Arc::DataSpeed, 49
- set\_min\_speed
  - Arc::DataSpeed, 50
- set\_progress\_indicator
  - Arc::DataSpeed, 50
- SetAdditionalChecks
  - Arc::DataPoint, 30
  - Arc::DataPointDirect, 37
  - Arc::DataPointIndex, 43
- SetMeta
  - Arc::DataPoint, 30
- SetSecure
  - Arc::DataPoint, 30
  - Arc::DataPointDirect, 37
  - Arc::DataPointIndex, 43
- SetURL
  - Arc::DataPoint, 30
- SetValid
  - Arc::FileCache, 56
- SortLocations
  - Arc::DataPoint, 30
  - Arc::DataPointDirect, 38
  - Arc::DataPointIndex, 43
- StageError
  - Arc::DataStatus, 52
- Start
  - Arc::FileCache, 57
- StartReading
  - Arc::DataPoint, 31
  - Arc::DataPointIndex, 43
- StartWriting
  - Arc::DataPoint, 31
  - Arc::DataPointIndex, 44
- Stat
  - Arc::DataPoint, 31, 32
- StatError
  - Arc::DataStatus, 52
- Stop
  - Arc::FileCache, 57
- StopAndDelete
  - Arc::FileCache, 57
- StopReading
  - Arc::DataPoint, 32
  - Arc::DataPointIndex, 44
- StopWriting
  - Arc::DataPoint, 32
  - Arc::DataPointIndex, 44
- Success
  - Arc::DataStatus, 51
- SuccessCached
  - Arc::DataStatus, 52
- SystemError
  - Arc::DataStatus, 52
- Transfer
  - Arc::DataMover, 19
- transfer
  - Arc::DataSpeed, 50
- TransferError
  - Arc::DataStatus, 52
- TransferLocations
  - Arc::DataPoint, 32
  - Arc::DataPointIndex, 44
- UnimplementedError
  - Arc::DataStatus, 52
- UnknownError
  - Arc::DataStatus, 52
- Unregister
  - Arc::DataPoint, 32
  - Arc::DataPointDirect, 38
- UnregisterError
  - Arc::DataStatus, 52

valid\_url\_options  
    Arc::DataPoint, [33](#)  
verbose  
    Arc::DataMover, [20](#)  
    Arc::DataSpeed, [50](#)  
  
wait\_any  
    Arc::DataBuffer, [13](#)  
WriteAcquireError  
    Arc::DataStatus, [51](#)  
WriteError  
    Arc::DataStatus, [52](#)  
WriteFinishError  
    Arc::DataStatus, [52](#)  
WriteOutOfOrder  
    Arc::DataPoint, [33](#)  
    Arc::DataPointDirect, [38](#)  
    Arc::DataPointIndex, [44](#)  
WritePrepareError  
    Arc::DataStatus, [52](#)  
WritePrepareWait  
    Arc::DataStatus, [52](#)  
WriteResolveError  
    Arc::DataStatus, [51](#)  
WriteStartError  
    Arc::DataStatus, [52](#)  
WriteStopError  
    Arc::DataStatus, [52](#)