

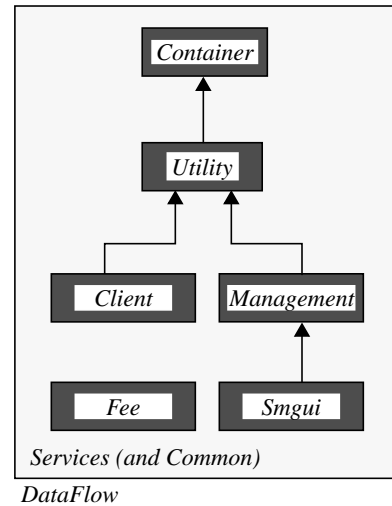
DataFlow Auxiliary Reference Manual¹

Version 1.0
December 9, 1997 10:24 am

DataFlow Core Group
and
External Developers

Abstract

This manual defines that part of DataFlow *Application Programmer Interface* (API) used to satisfy the needs of the online community both inside and outside the strict boundaries of a DataFlow platform. The Dataflow API takes the form of a collection of C++ classes organized in units of *packages*, where a package encapsulates a set of classes which cooperate in order to express a common functionality. The diagram at right shows the structure of the public DataFlow packages. This document covers only those packages which are visible beyond the core DataFlow platform boundary. These are the *Container* and *Common* packages. In order to maintain modularity, the code and documentation for these particular classes is considered outside the purvey of DataFlow, and thus the control and maintenance of these classes is under the management of the online system. Please see *DataFlow Reference Manual* for a brief description of each package and the relationships among the packages and for a description of those classes whose visibility never extends beyond platform boundaries.



1. For the current version see: www.slac.stanford.edu/BFROOT/doc/www/Computing/Online/OnlineDataFlow.html

Table of Contents

Overview	7
Required Background material and Knowledge	7
Introduction	7
odfContainer	9
odfTC	11
Overview	11
Constructor and Destructor	11
odfTC()	11
~odfTC()	12
Member Functions	12
contains()	12
iterator()	12
odfTCObject	13
Overview	13
Constructor and Destructor	13
Member Functions	13
primary()	13
secondary()	13
type()	14
odfTCIterator<T>	15
Overview	15
Constructor and Destructor	15
odfTCIterator	15
Member Functions	15
tc	15
use	15
first()	16
next()	16
previous()	16
pAdr.	16
dAdr.	17
odfdAdr	18
Overview	18
Constructor and Destructor	18
odfdAdr()	18
~odfdAdr()	18
Member Functions	18
value()	18
levelValue()	19
setLevelValue()	19
operator=()	19
operator==()	19
operator!=()	19

Constants and Defines.	20
enum dLevel	20
enum SubDetector	20
odfpAdr	21
Overview	21
Constructor and Destructor	21
odfpAdr().	21
~odfpAdr().	21
Member Functions.	21
value().	21
levelValue().	22
setLevelValue().	22
operator=().	22
operator==().	22
operator!=().	22
Constants and Defines.	23
enum pLevel	23
odfdTag	24
Constructor and Destructor	24
odfdTag().	24
~odfdTag().	24
Member Functions.	24
value().	24
levelValue().	24
setLevelValue().	25
operator=().	25
operator==().	25
operator!=().	25
odfMap	26
Overview	26
Constructor and Destructor	26
odfMap().	26
~odfMap().	27
Member Functions.	27
time().	27
pLevel().	27
print().	27
odfMapBrowser	28
Overview	28
Constructor and Destructor	28
odfMapBrowser().	28
~odfMapBrowser().	28
Member Functions.	28
up().	28
down().	28
next().	29
previous().	29

top()	29
bottom()	29
nextBranch()	29
nodes()	30
pAdr()	30
dAdr()	30
currentLevel()	30
odfMappable	32
Overview	32
Constructor and Destructor	32
Member Functions.	32
<To be determined by the design/implementation phase.>	32
odfStoredMap	33
Overview	33
Constructor and Destructor	33
odfStoredMap()	33
~odfStoredMap()	33
Member Functions.	33
<To be determined by the design/implementation phase>	33
odfCommon	34
odfTime	35
Overview	35
Constructor and Destructor	35
odfTime()	35
~odfTime()	35
Member Functions.	35
unix()	35
binary()	36
prevSubFidTime()	36
modulus()	36
operator+()	36
operator-()	37
operator==()	37
operator>()	37
operator<()	38
odfDuration	39
Overview	39
odfDuration()	39
~odfDuration()	39
Member Functions.	39
operator+()	39
operator-()	40
operator*()	40
returns: The new scaled duration.	40
operator/()	40
returns: The new scaled duration.	40

operator%()	40
returns: An unsigned short modulus.	41
Constants and Defined Types	41

List Of Figures

FIGURE 1.	Classes related to the <code>odfContainer</code> package.	9
FIGURE 2.	Classes related to the <code>odfCommon</code> package.	34

Overview

Required Background material and Knowledge

The Application Programmer Interface (API) defined within this document is derived from an Object Oriented design based on C++. The reader is assumed to have an intimate understanding of both object oriented design practice and the C++ language. In addition, a significant fraction of the API executes in a real-time environment on embedded processors within a DataFlow platform. Therefore, the reader should have a familiarity with both *VxWorks* and the principles of operation of a DataFlow platform.

Introduction

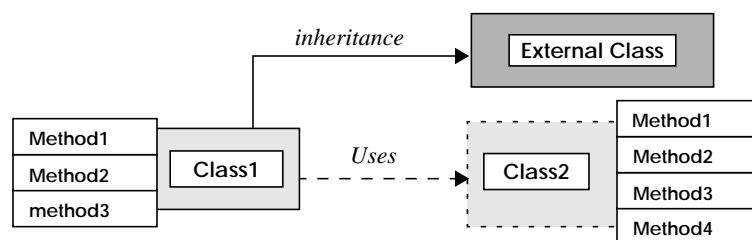
Each chapter in the document is a class description which follows a common format. First, the classes from which the described class inherits are enumerated. Second, the related classes which either use, or are used by, the described class are listed. Third, an overview, or synopsis, is provided, summarizing both the services provided by the class and its role within DataFlow.

The bulk of each class description is devoted to describing the data and member functions of the class. The first member functions described are always the class's constructor and destructor, with the remaining members listed in arbitrary order. Each member function of each class can be found in the reference manual's Table of Contents. For each member function, the various signatures of the function are listed and followed by a synopsis of the function. Following the synopsis, the arguments and return arguments are listed and a synopsis of *their* function provided. At the end of the class description, any constants and types defined by the class are listed.

The classes that make up the DataFlow API are grouped into *packages* corresponding roughly to their role within the DAS.

Accordingly, chapters of this reference manual

are grouped into sections, each section corresponding to a package description. At the beginning of each description is a diagram showing the relationships between the classes in that package. The classes are represented by shaded boxes, with their methods represented by inset unshaded boxes. Abstract base classes are represented by dashed boxes. Inheritance is indicated by a solid arrow and a *uses* relationship is indicated by a dashed arrow. Classes not in the described package are represented by darker shaded boxes.



odfContainer

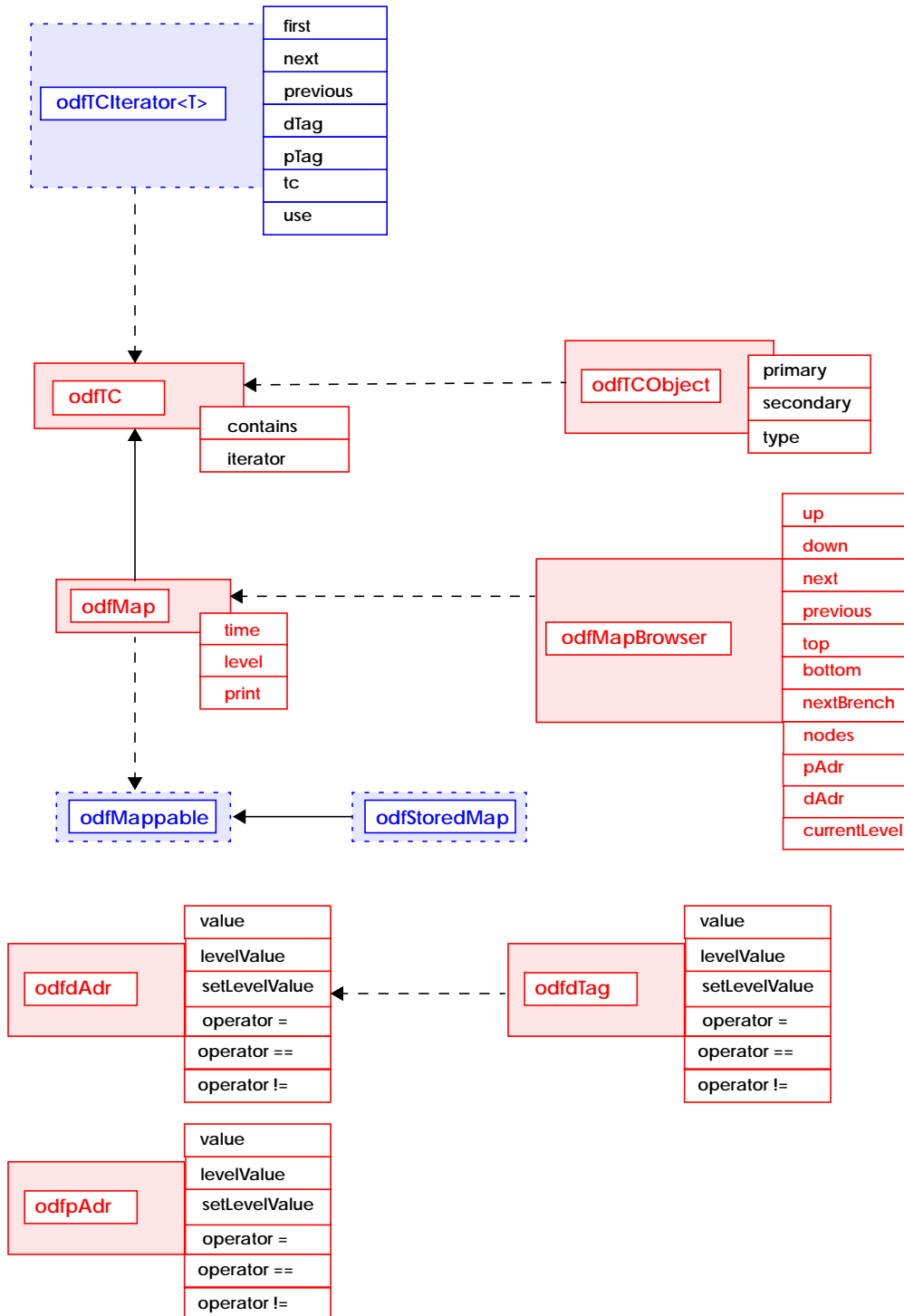


FIGURE 1. Classes related to the odfContainer package.

The `odfContainer` Package implements the organizational hierarchy that is imposed on all data transported by DataFlow. The tagged container (an object of class `odfTC`) is the building block for this hierarchy. The DataFlow System is capable of transporting tagged containers. It normally does this by using a special kind of tagged container of class `odfDatagram`.

A particular tagged container can hold a collection of objects of the same type. These objects could themselves be tagged containers. This fact allows the construction of a hierarchy of containers which reflects the fundamental organization of the data as it is built and processed by the various stages of DataFlow.

A container at the bottom of the hierarchy holds only objects which are not themselves non-empty containers. A tagged container of the base class can not directly hold additional user specified information. It can only hold other objects of a single type. To hold additional data directly in the tagged container object, simply use concrete inheritance from the `odfTC` base class and add data members and methods as appropriate.

The *kind* of children held by a tagged container is carried in a data member of the `odfTC` class. This is known as the `contains` field which is accessed by the `contains()` member function. A tagged container's children are accessed through an object of its associated iterator class (`odfTCIterator<T>`). This class is templated according to the type of the contained objects. The iterator is an abstract base class that must be implemented for each tagged container for which access is required. The specific iterator that should be used to access a tagged container's children is carried in its `iterator` field which is accessed by the `odfTC`'s `iterator()` member function.

To provide context for its contents, a tagged container has associated with it (i.e. it is *tagged* with) an address from within the Detector Hierarchy called the Detector Tag (`dTag`) and an address from within the Platform Hierarchy called the Platform Tag (`pTag`).

odfTC

Derived from: none

Related Classes: `odfTCIterator<T>`

Overview

The tagged container forms the basic object from which the event hierarchy is built. Each tagged container has a pair of tag values associated with it. These tag values are only accessed via the iterator allowing them to be either stored or calculated as appropriate.

Constructor and Destructor

`odfTC()`

```
odfTC(const odfTCObject&, Iterator)
odfTC(odfTCObject::Primary,
      odfTCObject::Secondary,
      odfTCObject::Type,
      Iterator)
```

The constructor creates an instance of a tagged container using the specified arguments to set the initial values of the container's attributes. The container is initially empty. This means that its number items is set to zero and it is undamaged. The arguments to the constructor always specify something about what the container contains (i.e. its children).

arguments:

`odfTCObject` specifies the kind of children the container holds. This value is returned by the `contains()` member function.

`Iterator` specifies which iterator to use when accessing the container's children. This value is returned by the `iterator()` member function.

`Primary` is an enumeration of the primary key to be associated with this object. (See the `odfTCObject` class for more information.)

`Secondary` is an enumeration of the secondary key to be associated with this object. (See the `odfTCObject` class for more information.)

`Type` is an enumeration with a value of either child or parent. (See the `odfTCObject` class for more information.)

~odfTC()`~odfTC()`

Member Functions

contains()`const odfTCObject& contains() const`

Returns the kind of children the container holds. The children of a given container must be homogeneous.

arguments: None.

returns: A key which describes the containers children.

iterator()`Iterator iterator() const`

Returns the kind of iterator that should be used to access the container's children.

arguments: None.

returns: A key which describes an `odfTCIterator` object which is appropriate for iterating over and accessing the containers children.

odfTCObject

Related Classes: odfTC

Overview

This class provides a generalized form of type identification for the objects in tagged containers. There need not be a one-to-one mapping between C++ types and the values of `odfTCObject`. Together with a value from `odfTC::iterator()`, `odfTCObject` provides all the information necessary to construct an `odfTCIterator` object.

Constructor and Destructor

`odfTCObject()`

```
odfTCObject(Primary, Secondary, Type)
odfTCObject(odfTCObject&)
```

arguments:

`Primary` is an enumeration of the primary key to be associated with this object.

`Secondary` is an enumeration of the secondary key to be associated with this object.

`Type` is an enumeration with a value of either child or parent.

Member Functions

`primary()`

```
Primary primary() const
```

Returns the primary lookup key used to identify the specified object.

arguments: none.

returns: A primary key value.

`secondary()`

```
Secondary secondary() const
```

Returns the secondary lookup key used to identify the specified object.

arguments: None.

returns: A secondary key value.

type()

Type type() const

Returns whether or not the object has children.

arguments: None.

returns: The enumerated value `child` indicates that the object does not have children. The enumerated value `parent` indicates that the object has children.

odfTCIterator<T>

Related Classes: odfTC

Overview

The `odfTCIterator` class specifies the behavior of an iterator for the children of a particular tagged container. An iterator derived from this base class can be used to access the contained items through a standard set of operations. These operations act on the iterator in a manner that is consistent with the way they act on simple pointers. It is important to note that this class is templated by the actual type that the iterator can access.

Note: This documentation describes the iterator as it exists in the repository. Once the `odfpAdr` and `odfdAdr` classes are added the iterator must be updated to return addresses rather than tags (there is no longer an `odfpTag`).

Constructor and Destructor

odfTCIterator

```
odfTCIterator()
odfTCIterator(odfTC*)
```

Initializes the iterator for use with a particular tagged container.

arguments: `odfTC*` is the tagged container to which this iterator will allow access.

Member Functions

tc

```
odfTC* tc() const
```

Allows access to the tagged container that is currently being browsed by this iterator without changing the iterator's state.

arguments: None.

returns: A pointer to the previous tagged container. `NULL(0)` is returned if no tagged container has been supplied.

use

```
virtual int use() =0
int use(odfTC*)
```

Initializes the iterator for use with a particular tagged container by performing the

appropriate lookup and matching of the tagged container with this iterator. The no argument version actually does the lookup on the stored tagged container. The tagged container was either provided to the constructor or to the overloaded version of use()

which then just calls use() with no arguments. The intention is that use should be called before the iterator can be used on a the tagged container. Before use is called first(), next(), and previous() should all return NULL(0) when called.

arguments: odfTC* is the tagged container to use with this iterator.

returns: Zero(0) is returned if the container and iterator are not compatible and non-zero is returned if they are compatible.

first()

```
virtual T first()=0
```

Sets the iterator to the first child item (the left-most child in the hierarchy graph).

arguments: None.

returns: The left-most child or NULL(0) if the iterator has never been given a valid tagged container.

next()

```
virtual T next()=0
```

Next returns the current child and then iterates to the next child.

arguments: None.

returns: A child object of type T (which could be a pointer) or NULL(0) if the iterator is pointing past the last child.

previous()

```
virtual T previous()=0
```

Iterates to the previous child and then returns the current child.

arguments: None.

returns: A pointer to the child or NULL(0) if the iterator is pointing past the first child.

pAdr

```
virtual odfpAdr pAdr() =0
```

Accesses the platform space address for the object currently pointed to by the iterator.

arguments: None.

returns: A platform address.

dAdr

```
virtual odfdAdr dAdr() =0
```

Accesses the detector space address for the object currently pointed to by the iterator.

arguments: None.

returns: A detector address.

odfdAdr

Related Classes: `odfpAdr`

Overview

Data is transported through the dataflow platform in tagged containers. The data stored in an `odfTC` is accessible via an `odfTCIterator<T>`. The iterator is able to return the source or destination of the datum as one of two addresses, the platform address or the detector address. These addresses are both 32-bit unsigned integers that can be manipulated using the `odfdAdr` and `odfpAdr` classes

The detector address is manipulated using the `odfdAdr` class. The hierarchy of levels in the detector address is `Detector`, `Module`, `Section`, `Chip`, `Channel`. Note that the detector address for a given channel may change with time. In order to extract the channel identifier, *detector tag*, the address must be passed to the `odfdTag` class.

Constructor and Destructor

`odfdAdr()`

```
odfdAdr()
odfdAdr(unsigned int adrValue)
odfdAdr(const odfdAdr& adr)
```

The `odfdAdr` class stores an unsigned int whose value is the address and provides functions to manipulate the address.

arguments: `adrValue` - the value of the detector address

`adr` - a detector address

`~odfdAdr()`

```
~odfdAdr()
```

Member Functions

`value()`

```
unsigned int value() const
```

arguments: none.

returns: The value of the address

levelValue()

```
unsigned char levelValue(dLevel level) const
```

arguments: `level` - The level in the detector hierarchy for which the level value is required.

returns: The value of a particular field in the address

setLevelValue()

```
odfdAdr& setLevelValue(dLevel level, unsigned char value) const
```

Set a field in the detector address. Note that the field values for all lower levels are set to 0.

arguments: `level` - the level in the detector hierarchy

`value` - the value that a field in the address is to be set to

returns: A reference to the address that has been set

operator=()

```
odfdAdr& operator=(const odfdAdr& adr)
```

arguments: `adr` - the address to copy

returns: A reference to the address that has been set

operator==()

```
int operator==(const odfdAdr& adr) const
```

arguments: `adr` - the address to be compared

returns: 0 if the values of the addresses are not equal.

operator!=()

```
int operator!=(const odfdAdr& adr) const
```

arguments: `adr` - the address to be compared

returns: 0 if the values of the addresses are equal.

Constants and Defines

enum dLevel

```
enum dLevel {Channel=0, Chip, Section, Module, Detector}
```

The levels of the detector hierarchy

enum SubDetector

```
enum SubDetector {SVT=1, DCH, DRC, EMC_BRL, EMC_ECP, IFR,  
                  TRG_TRK, TRG_ENR, TRG_GBL};
```

The sub-detector values.

odfpAdr

Related Classes: `odfdAdr`

Overview

Data is transported through the dataflow platform in tagged containers. The data stored in an `odfTC` is accessible via an `odfTCIterator<T>`. The iterator is able to return the source or destination of the datum as one of two addresses, the platform address or the detector address. These addresses are both 32-bit unsigned integers that can be manipulated using the `odfdAdr` and `odfpAdr` classes

The platform address is manipulated using the `odfpAdr` class and can only provide the address to the level of a front end element. The hierarchy of levels in the platform address is `Partition`, `Crate`, `Slot`, `Element`, `Chip`, `Channel`. Note that data-flow cannot determine values for the chip or channel, the value of an address channel field is the same as the value of that field in the detector address.

Constructor and Destructor

`odfpAdr()`

```
odfpAdr()
odfpAdr(unsigned int adrValue)
odfpAdr(const odfpAdr& adr)
```

The `odfpAdr` class stores an unsigned int whose value is the address and provides functions to manipulate the address.

arguments: `adrValue` - the value of the platform address

`adr` - a platform address

`~odfpAdr()`

```
~odfpAdr()
```

Member Functions

`value()`

```
unsigned int value() const
```

arguments: none.

returns: The value of the address

levelValue()

```
unsigned char levelValue(pLevel level) const
```

arguments: `level` - The level in the platform hierarchy for which the level value is required.

returns: The value of a particular field in the address

setLevelValue()

```
odfpAdr& setLevelValue(pLevel level, unsigned char value) const
```

Set a field in the platform address. Note that the field values for all lower levels are set to 0.

arguments: `level` - the level in the detector hierarchy

`value` - the value that a field in the address is to be set to

returns: A reference to the address that has been set

operator=()

```
odfpAdr& operator=(const odfpAdr& adr)
```

arguments: `adr` - the address to copy

returns: A reference to the address that has been set

operator==()

```
int operator==(const odfpAdr& adr) const
```

arguments: `adr` - the address to be compared

returns: 0 if the values of the addresses are not equal.

operator!=()

```
int operator!=(const odfpAdr& adr) const
```

arguments: `adr` - the address to be compared

returns: 0 if the values of the addresses are equal.

Constants and Defines

enum pLevel

```
enum pLevel {Channel=0, Chip, Element, Slot, Crate, Partition}
```

The levels of the platform hierarchy

odfdTag

Related Classes: `odfdAdr`

Overview

Data transported by the dataflow system is labelled with the detector and platform addresses. In order to provide a unique, unchanging identifier for a channel, the *detector address* can be transformed into a *detector tag* using the `odfdTag` class

Constructor and Destructor

`odfdTag()`

```
odfdTag()
odfdTag(unsigned int tagValue)
odfdTag(const odfdTag& tag)
odfdTag(const odfdAdr& adr)
```

The `odfdTag` class stores an unsigned int whose value is the detector tag and provides functions to manipulate the tag.

arguments: `tagValue` - the value of the detector tag

`tag` - a detector tag

`adr` - a detector address

`~odfdTag()`

```
~odfdTag()
```

Member Functions

`value()`

```
unsigned int value() const
```

arguments: none.

returns: The value of the detector tag

`levelValue()`

```
unsigned char levelValue(dLevel level) const
```

arguments: `level` - The level in the detector hierarchy for which the level value is required.

returns: The value of a particular field in the tag

setLevelValue()

```
odfdTag& setLevelValue(dLevel level, unsigned char value) const
```

Set a field in the detector tag. Note that the field values for all lower levels are set to 0.

arguments: `level` - the level in the detector hierarchy

`value` - the value that a field in the tag is to be set to

returns: A reference to the tag that has been set

operator=()

```
odfdTagr& operator=(const odfdTag& tag)
```

arguments: `tag` - the detector tag to copy

returns: A reference to the tag that has been set

operator==()

```
int operator==(const odfdTag& tag) const
```

arguments: `tag` - the tag to be compared

returns: 0 if the values of the tags are not equal.

operator!=()

```
int operator!=(const odfpTag& tag) const
```

arguments: `tag` - the address to be compared

returns: 0 if the values of the tags are equal.

odfMap

Derived from: `odfTC`

Related Classes: `odfMappable`, `odfStoredMap`, `odfMapBrowser`

Overview

A `odfMap` object holds a fully populated event hierarchy with the top of the hierarchy set by where in the platform the constructor was invoked or by the argument type passed to the constructor. The map can be used to explore a particular version of the event hierarchy and in particular, the relationships between the address space and the tag space. The user's interface to the map is through the `odfMapBrowser`.

A `odfMap` is normally created from a mappable object (one that inherits and implements the `odfMappable` interface class). Examples of mappable objects include `odfPlatform`, `odfPartition`, `odfCrate`¹, `odfSlot`², and `odfStoredMap`. If called with a `odfPlatform` object, `odfMap` builds the full event hierarchy of the platform. If called with a `odfPartition` object, `odfMap` creates the portion of the event hierarchy corresponding to that partition. If called with a `odfCrate` object, the event hierarchy is trimmed to correspond to only the crate specified. If called with a `odfSlot` object, an event hierarchy for a single ROM is created. In all of the previous cases, the `odfMap` object contains a snapshot of some portion of the event hierarchy at the time when the object was created. If called with a `odfStoredMap` object, `odfMap` retrieves and recreates a previous version of a map from the appropriate persistent store.

In the current implementation the map is actually instantiated from a text file.

Constructor and Destructor

`odfMap()`

```
odfMap()
odfMap(const odfMappable& object)
odfMap(odfdAdr::SubDetector detector, unsigned char module)
```

Instantiates an object of the `odfMap` class by the algorithm appropriate to `object`. For example, the constructor might simply expand the ROM's locally stored map of the attached Front-End-Electronics sections. It might also request map information from all of the components which fall under the requestor's control by using the FCTS. In this case the map is built as the request is processed and forwarded by each of the components. The build terminates with the requestor receiving a complete set of tagged containers for his branch of the event hierarchy tree.

1. The `odfCrate` class does not yet exist.
2. The `odfSlot` class does not yet exist.

arguments:

`object` - is a reference to any object which inherits from and implements the `odf-Mappable` interface. If called with no arguments the object definition is taken from the location in which the constructor was invoked.

`detector` - the sub-detector number.

`module` - the module number. The map will be instantiated from a file (`detN.map` where `det` is the detector name and `N` is the module number)

~odfMap()

```
~odfMap()
```

The destructor frees the tagged containers which were used to build the event hierarchy.

Member Functions**time()**

```
odfTime_t time() const
```

So that maps can be cross-referenced with anything else in the DAQ system, their creation time is latched and stored. This function provides this stored time to the caller.

arguments: none

returns: The DataFlow Time specifying when the map was created.

pLevel()

```
odfpAddress::pLevel pLevel()
```

arguments: none

returns: The platform level at which the map was instantiated.

print()

```
void print(ostream&)
```

Prints out the full map to a stream.

arguments: `ostream` - the stream that the map will be printed on

returns: nothing.

odfMapBrowser

Related Classes: `odfMap`

Overview

This class provides a way to navigate through an `odfMap` hierarchy.

Constructor and Destructor

`odfMapBrowser()`

```
odfMapBrowser(odfMap*)
```

Instantiate a browser on a map.

arguments: `odfMap` - a pointer to the map that is to be browsed.

`~odfMapBrowser()`

```
~odfMapBrowser()
```

Member Functions

`up()`

```
int up()
int up(odfpAdr::pLevel level)
```

Move up one level or to the specified level in the platform hierarchy. Note that the browser cannot move above the level at which it was instantiated.

arguments: `level` - the platform level that the browser should move to

returns: 0 if the browser failed to move up in the hierarchy because it is already at the node at which it was instantiated or the specified level is above the point at which the browser was instantiated.

`down()`

```
int down()
int down(odfpAdr::pLevel level)
```

Move down one level or to the specified level in the platform hierarchy.

arguments: `level` - the platform level that the browser should move to

returns: 0 if the browser failed to move down in the hierarchy because it is already at

the leaf nodes of the hierarchy or the browser is already at a lower level of the hierarchy than that requested.

next()

```
int next()
```

Move to the next node in the current level of the hierarchy that is being browsed. Note that this has the same post-increment semantics as `odfCIterator`.

arguments: none

returns: 0 if there are no more nodes at this level in the hierarchy.

previous()

Move back to the previous node in the current level of the platform hierarchy. This uses the same pre-increment semantics as `odfCIterator`

arguments: none

returns: 0 if the browser failed to move down in the hierarchy.

top()

```
void top()
```

Move the browser back to the where it was instantiated.

arguments: none

returns: nothing.

bottom()

```
void bottom()
```

Move from the current node to the bottom of the hierarchy.

arguments: none

returns: nothing.

nextBranch()

```
int nextBranch()
```

Move from the current position in the hierarchy to the next “branch” in the hierarchy.

ie. Move up one level and to the next node there if one exists, otherwise move up and try to move to the next node etc until the top of the hierarchy is reached.

arguments: none

returns: 0 if there are no more branches, the iterator then points to the point at which it was instantiated.

nodes()

```
int nodes(odfpAdr::pLevel level)
```

arguments: `level` - the level in the platform hierarchy

returns: the number of nodes in the map at the requested platform level beneath the current position.

pAdr()

```
unsigned int pAdr()
```

arguments: none

returns: The value of the platform address of the node at which the browser currently points.¹

dAdr()

```
unsigned int dAdr()
```

arguments: none

returns: The value of the detector address of the node at which the browser currently points (see footnote for `pAdr()`)

currentLevel()

```
odfpAddress::pLevel currentLevel() const
```

1. Note that due to the post-increment semantics of the browsers `next()` function, this should be extracted before `next` is called. eg:

```
unsigned int pValue;
while (pValue=browser.pAddressValue(), next()) {
    ...
}
```

arguments: none

returns: the platform level of the node that the browser is currently pointing at.

odfMappable

Derived from: none

Related Classes: odfMap

Overview

This class provides the interface for a mappable object. It defines a set of pure virtual functions which allow the `odfMap` object to create the appropriate kind of map.

Constructor and Destructor

Member Functions

<To be determined by the design/implementation phase.>

odfStoredMap

Derived from: (?PersistentStorageClass?), odfMappable

Related Classes: odfMap

Overview

This class allows maps to be stored to and retrieved from a persistent storage mechanism¹ for the conditions database. A stored map is not a map in the sense that it is not a hierarchy of tagged containers. It can be turned back into a map since it inherits from `odfMappable`.

Constructor and Destructor

odfStoredMap()

```
odfStoredMap(const PersistentStorageKey& key)
odfStoredMap(const odfMap& map)
```

Creates a map object by either accessing the persistent storage and importing the stored map by using functions provided by the Persistent Storage Class or by copying an existing map into a format appropriate for persistent storage. The constructor does not directly store the map object to persistent storage. It simply inherits from a the base class that makes it a storable object.

arguments: `key` provides the appropriate access identifier for retrieval from persistent storage. `map` is the map object which is to be copied into a form suitable for persistent storage.

~odfStoredMap()

```
~odfStoredMap()
```

Deletes the local version stored map object. The persistent version is not affected.

Member Functions

<To be determined by the design/implementation phase>

1. DataFlow does not specify the persistent storage mechanism.

odfCommon

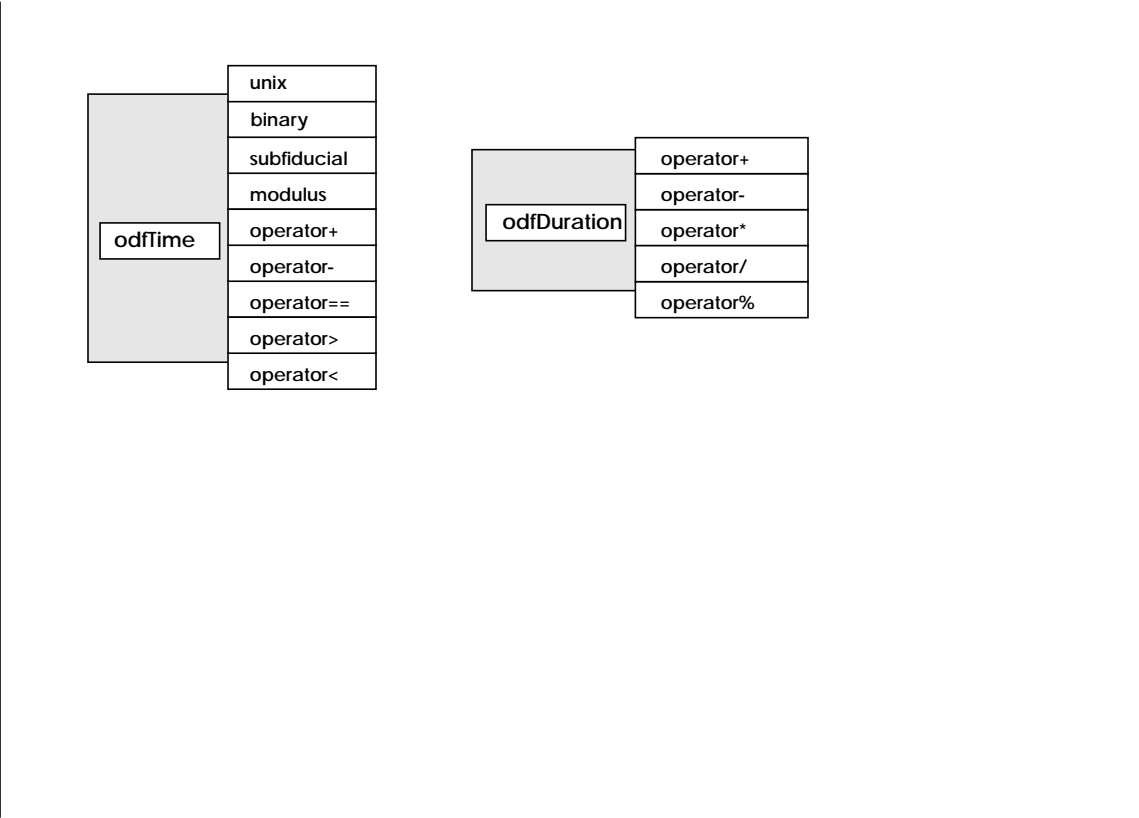


FIGURE 2. Classes related to the `odfCommon` package.

The `odfCommon` package consists of classes which may be used by all other packages, and whose functionality may be of use to the client both within and outside of the Data-flow Platform. Included here are tools for manipulating DataFlow time (`odfTime` and `odfDuration`).

odfTime

Derived from:

Related Classes: `odfDuration`, `odfBinTime`, `odfUnixTime`

Overview

The `odfTime` class provides access to DataFlow Platform time. The object's stored time value can be initialized or returned in either DataFlow binary format or Unix internal format. The class also provides unary and binary operators for relating and manipulating two `odfTime` objects. *Dataflow binary format* refers to time as a 64-bit quantity incremented at *sysclk*. *Unix internal format* refers to time as two 32-bit words, one counting seconds and the other counting microseconds. In either case, DataFlow time is calculated relative to midnight January 1, 1997 GMT. See Section 13.0 on page 54 of [1] for more information on DataFlow Platform time.

Constructor and Destructor

`odfTime()`

```
odfTime()
odfTime(odfBinTime time)
odfTime(odfUnixTime time)
odfTime(unsigned upper, unsigned lower)
```

Instantiates an object of the `odfTime` class. If no argument is present, then the object stores the current value of the time in Unix format. If an argument is present, then the object stores the specified time which can be input in either DataFlow binary format or Unix format.

arguments: Either none, or time value in either format.

`~odfTime()`

```
~odfTime()
```

Deletes an object of the `odfTime` class.

arguments: None.

Member Functions

`unix()`

```
odfUnixTime& unix()
```

Fills the object's internal time value in Unix format, *i.e.* in two long words, one of seconds and one of microseconds. If the internal Unix format is already filled (nonzero) then no action is taken.

arguments: None.

returns: The unix format value of object's time.

binary()

```
odfBinTime& binary()
```

Fills the object's internal time value in binary format, *i.e.* in two long words of *sysclks* (16.7 nsec). If the internal binary format is already filled (nonzero) then no action is taken.

arguments: None.

returns: The binary format value of object's time.

prevSubFidTime()

```
odfTime prevSubFidTime()
```

This function returns the object's time divided by the number of *sysclks* in a *subFiducial* (873). This operation is implemented as integer division with the remainder discarded. This yields the number of subFiducials that have occurred up to the time stored in the object. The value returned only has significance when the object's time is associated with an FCPM transaction.

arguments: None.

returns: The number of *subFiducials* for the time stored in this object.

modulus()

```
unsigned int modulus()
```

This function returns the remainder resulting from dividing the object's time by the number of *sysclks* in a *subFiducial* (873). This is the number of *sysclks* that have occurred since the last *subFiducial*.

arguments: None.

returns: Time within the subFiducial cycle in binary format.

operator+()

```
odfTime operator+(odfDuration time)
```

This operator returns a `odfTime` object corresponding to the sum of the object's time and the argument, where the argument specifies a duration.

arguments: `time` -a duration, or time difference.

return arguments: `odfTime` object corresponding to the sum.

operator-()

```
odfDuration operator-(odfTime time)
odfTime      operator-(odfDuration time)
```

This operator returns either a `odfTime` object corresponding to the difference of the object's time and a duration, or a duration corresponding to the difference of the object's time the value specified by another `odfTime` object.

arguments: `time` - either a duration or a reference to a `odfTime` object.

returns: either a `odfTime` object or a duration.

operator==(())

```
int operator==(odfTime& time)
int operator==(odfBinTime time)
int operator==(odfUnixTime time)
```

This operator returns a boolean value expressing the equality of the object's time and the argument, where the argument specifies a time either by reference to a `odfTime` object or by an explicit value in either binary or Unix format.

arguments: `time` -DataFlow time in the specified format.

returns: Boolean value expressing equality of two DataFlow time values.

operator>()

```
int operator>(odfTime& time)
int operator>(odfBinTime time)
int operator>(odfUnixTime time)
```

This operator returns a boolean value expressing the *greater than* relation for the object's time and the argument, where the argument specifies a time either by reference to a `odfTime` object or by an explicit value in either binary or Unix format.

arguments: `time` -DataFlow time in the specified format.

returns: Boolean value expressing the *greater than* relation for two DataFlow time values.

operator<()

```
int operator<(odfTime& time)
int operator<(odfBinTime time)
int operator<(odfUnixTime time)
```

This operator returns a boolean value expressing the *less than* relation for the object's time and the argument, where the argument specifies a time either by reference to an `odfTime` object or by an explicit value in either binary or Unix format.

arguments: `time` - DataFlow time in the specified format.

returns: Boolean value expressing the less than relation for two DataFlow time values.

odfDuration

Derived from:

Related Classes: `odfTime`

Overview

The `odfDuration` class is used to manipulate DataFlow Platform time. These objects will be created by the user by taking the difference of two `odfTime` objects. Once created, two durations can be added and subtracted. One duration can also be scaled either by multiplying by a long, or by dividing by a short. Division by a long is not currently supported because of extra coding complexity and human resource constraints.

`odfDuration()`

```
odfDuration()
```

Instantiates an object of the `odfDuration` class. Initial values are set by taking the difference of two `odfTime` objects.

`~odfDuration()`

```
~odfDuration()
```

Deletes an object of the `odfDuration` class.

arguments: None.

Member Functions

`operator+()`

```
odfDuration operator+(odfDuration time)
odfDuration operator+(unsigned long ticks)
```

Adds a duration to an existing duration OR increment a duration by an unsigned long number of ticks. Can potentially generate an error message if the 64bit result overflows (unsupported).

arguments: Either an `odfDuration` time or an unsigned long number of ticks.

returns: The new incremented duration.

operator-()

```
odfDuration operator-(odfDuration time)
odfDuration operator-(unsigned long ticks)
```

Subtracts a duration from an existing duration OR decrement a duration by an unsigned long number of ticks. Can potentially generate an error message if the subtraction generated a negative number (unsupported).

arguments: Either an `odfDuration` time or an unsigned long number of ticks.

returns: The new decremented duration.

operator*()

```
odfDuration operator*(unsigned long scale)
```

Scales a duration by an unsigned long value. This could be useful for changing your durations from sysclks to subfiducials and back.

arguments: An unsigned long scale factor.

returns: The new scaled duration.

operator/()

```
odfDuration operator/(unsigned short scale)
```

Scales a duration by an unsigned short value. This could be useful for changing your durations from sysclks to subfiducials and back. Note that only division by `unsigned short` is currently supported, because it is a simpler algorithm. Manpower and time were not present to develop a more general division algorithm.

arguments: An unsigned short scale factor.

returns: The new scaled duration.

operator%()

```
unsigned short operator%(unsigned long scale)
```

The modulus operator. Returns the remainder when a duration is divided by an unsigned short. This would be useful for calculating the number of sysclks since the last subfiducial, for example. Note that (for the same reasons mentioned above for `operator/()`) only division by `unsigned short` is currently supported.

arguments: An unsigned short divisor.

returns: An unsigned short modulus.

Constants and Defined Types

List of References

DataFlow Documents

- [1] Hamilton, R. T., M. E. Huffer, and J. L. White, *DAQ Programmer's Guide*.
- [2] Hamilton, R. T., M. E. Huffer, and J. L. White, *DataFlow Reference Manual*.
- [3] Hamilton, R. T., M. E. Huffer, and J. L. White, *The DataFlow Platform User's Guide*.

ROM Documents

- [4] Haller, G., *DAQ Readout Module - Overview*
- [5] Huffer, M. E., *Conceptual Model of the ROM from the perspective of DataFlow*
- [6] Dowdell, J., *Architecture for the BaBar DAQ Read-Out Module*
- [7] Oxoby, G., *PCI Mezzanine Card*
- [8] Dowdell, J., *BaBar DAQ Controller Card Description*
- [9] Haller, G., *Architecture for the BaBar Triggered Personality Card*
- [10] Intel, *i960RP Microprocessor User's Manual*, Part #272736-001, 1996.

FCTS Documents

- [11] Lankford, A., *Functional Requirements of the BaBar Fast Control and Timing System*.
- [12] Haller, G., *Conceptual Design of the BaBar Fast Control and Timing System*.
- [13] Sapozhnikov, L., *Architecture for the BaBar Fast Control and Timing System*.
- [14] Haller, G. and G. Oxoby, *Electronics Control and Dataflow between the Read-Out Module and the Front-End Electronics Systems*, BaBar Note 281 Version 1.1.

Wind River Systems (WRS) Documents

- [15] WRS, *User's Guide: Tornado 1.0 (UNIX Version)*.
- [16] WRS, *Tornado Release Notes (UNIX Version), 1.0, Rev. 2*.
- [17] WRS, *Wind River Products Installation Guide, Torando 1.0 (UNIX Version), Rev. 3*.
- [18] WRS, *Tornado for PowerPC Architecture Supplement, 1.0*.

- [19] WRS, *VxWorks Programmers Guide*, 5.3.
- [20] WRS, *VxWorks Reference Manual*, 5.3.
- [21] WRS, *Tornado API Guide*, 1.0 Beta.
- [22] WRS, *GNU ToolKit User's Guide*, 2.6.
- [23] WRS, *Debugging with GDB*, 4.12.

Other Third Party Documents

- [24] Radstone, *Radstone PPC603/603e/604 User Guide*.
- [25] Radstone, *Radstone PPC603/4 Rev. 004 Installation Guide*.
- [26] Tundra, Universe Chapter, *VMEbus Interface Components Manual*, spring 1996.
- [27] VITA, *The VMEbus Specification*, ANSI/IEEE STD 1014-1987, ICE 821 and 297.
- [28] EPICS Documentation.

DataFlow CDR Supporting Documents

- [29] Day, C. T., R. T. Hamilton, M. E. Huffer, and J. L. White, *The Data Flow System*.
- [30] Hamilton, R. T., *Fast Control Software Manual*.
- [31] Huffer, M. E., *The Segment Builder*.
- [32] White, J. L., *The Fragment Builder*.
- [33] Day, C. T., *The Event Builder*.

Other BaBar Documents

- [34] BaBar Collaboration, *BaBar Technical Design Report*.