< prev contents

4 Data Acquisition

Chapter Index

4.1 Concept

Secure Controls Framework uses the same concept as Data Acquisition Engines. In order to read or set data, the user needs to start the data acquisition job. The job has four attributes that describe the conditions of a data transfer:

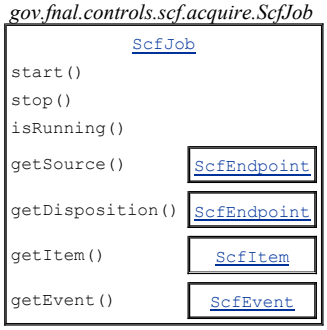
Source:
where data are going *from*;
Disposition:
where data are going *to*;
Item:
what data should be transferred;
Event:
when it should happen.

Data acquisition jobs in SCF are created inside `DataConnection` class. The most common method is:

```
public ScfJob newJob( ScfEndpoint source, ScfEndpoint disposition,
                      ScfItem item, ScfEvent event )
```

The returned object describes a job. By default the job is not running. The user must call `start` method to launch it, and may call `stop` to break. Depending on a particular event, the job can finish automatically. `isRunning` method returns *true* is the job is currently running. The current connection status can be disregarded when creating and starting jobs: formally, a job can be running on the client even if the server is unavailable. Upon connection, an internal mechinism of `DataConnection` will replicate all active jobs to the server.

Here is the structure of `ScfJob` class:



Existing instances of sources, dispositions, items, and events can be reused. `ScfJob` has no mean to control whether an attribute has already been employed in another job. Common sources, dispositions, and events can never cause problems, but this is a burden of the programmer to make sure that shared items do not mess anything up.

4.2 Job Attributes

4.2.1 Source & Disposition

All data sources and dispositions are interchangeable in SCF. Both of them are described by `ScfEndpoint` interface.

- *interface* `ScfEndpoint`
  - `AcceleratorEndpoint`
  - `ClientEndpoint`
    - `ClientCallbackEndpoint`
  - `ModelEndpoint`

The following chart gives an idea which combinations of `ScfEndpoints` are legal in a job.

Data Sources and Dispositions

		DISPOSITION		
		Accelerator	Client	Model
SOURCE	Accelerator	INVALID	VALID	INVALID
	Client	VALID	INVALID	VALID
	Model	INVALID	VALID	INVALID

For convenience, shared instances of some endpoints are available through `JobConstants` interface:

Default Endpoints

Constant Name	Value
ACCELERATOR_ENDPOINT	`AcceleratorEndpoint`
CLIENT_ENDPOINT	`ClientEndpoint`
DEFAULT_READING_SOURCE	`AcceleratorEndpoint`
DEFAULT_SETTING_DISPOSITION	

`ClientCallbackEndpoint` is a client disposition that allows the user to add one or several callback listeners:

- *interface* `CallbackListener`
  - `DataCallbackListener`
  - `PlotCallbackListener`

`DataCallbackListener` is designed for regular (not plotting) jobs to track out individual changes of the property values. As described below, all values read in a regular job are cached in corresponding data holders. So, if the program needs to know only current values (and not keep track of all changes), it does not have to use a listener. On the contary,

plotting data is not cached, and PlotCallbackListener is the only way to get the updates.

4.2.2 Item

The "item" attribute defines a subject of the data acquisition job that needs to be read or set: device channel, clock event, etc. It is described by ScfItem interface. There are two kinds of items supported: accelerator devices and clock events.

- *interface* ScfItem
  - *abstract class* Device
    - AtomicDevice
    - CompositeDevice
      - ArrayDevice
    - EmptyDevice
  - *abstract class* EventItem
    - AtomicEventItem
    - CompositeEventItem

Job items constitute complex structures that reflect an actual hierachy of entities in the data acquisition system and in the program. In fact, besides getting data from the data acquisition system, most of the programs need to store this data somewhere locally. SCF automatically caches the acquired values in cells that can be linked to some custom objects (for example, GUI components). These cells are called data holders.

- *interface* DataHolder
  - AtomicEventItem
  - *abstract class* DeviceProperty
    - AnalogAlarmProperty
    - ControlProperty
    - PlotDeviceProperty
    - ReadingProperty
    - SettingProperty
    - StatusProperty

How ScfItems and DataHolders are put together will be explained in the following sections.

4.2.3 Event

The "event" attribute specifies when (or how often) the data should be transferred from the source to the disposition. Job events are defined by ScfEvent interface.

- *interface* ScfEvent
  - ScfCommonEvent
    - ScfDefaultEvent
    - ScfOnceImmediateEvent
  - ScfContinuousSettingEvent

On DAEs, many of the events can be described with reconstruction strings. For example:

Event Reconstruction Strings

Event	Reconstruction String Format
Absolute Time event	**a**,<date>,<before>,<after>
Clock Event	**e**,<hex-number>, [**h**|**s**|**e**],<delay>
Default Data Event	u
Delta Time Event	**d**,<date1>,<date2>,<delay>
Multiple Immediate Event	**m**
Once Immediate Event	**i**
Periodic Event	**p**,<delay>,[**true**|**false**]
State Event	**s**,<di>,<value>,<delay>, [**=**|**!=**|*****|**>**|**<**]

ScfCommonEvent class uses these expressions in the constructor parameter. Thus, any DAE event can be used in SCF, as soon as it is convertable to a text (the continuous setting event is not.) ScfDefaultEvent and ScfOnceImmediateEvent are provided for convenience.

Shared instances of some events are available through JobConstants interface:

Default Events

Constant Name	Event Class
DEFAULT_EVENT	ScfDefaultEvent
DEFAULT_READING_EVENT	ScfDefaultEvent
DEFAULT_SETTING_EVENT	ScfContinuousSettingEvent

4.3 Special Numerical Values

The data acquisition system operates with values that usually are more than just numbers. In the wide sense, a measured value can include some additional attributes, such as: timestamp, unit, and representation format details. SCF has few extentions of the generic Number class in order to support this.

- Number
  - AltScalar
  - ErrorMarker
  - RawScalar
  - Scalar

These values are used in the following situations:

Usage of Special Values

Class	Usage	Attributes
AltScalar	A numerical value that has a distinct textual representation; i.e., digital status.	:: timestamp :: characters :: character colors
ErrorMarker	Error value. Used instead of all other Numbers if the reading has failed. Numerical value is always NaN.	:: timestamp :: error message :: error code :: facility code
RawScalar	Regular hexadecimal value: reading, setting, minimum, maximum, nominal, tolerance, etc.	:: timestamp :: size
Scalar	Regular decimal value: reading, setting, minimum, maximum, nominal, tolerance, etc.	:: timestamp :: unit :: long/short view; :: decimal/exponential

If the program needs to have numbers, these values can simply cosidered to be instances of java.lang.Number. For example, doubleValue() method always returns the corresponding double. JobDataFormatter.format() properly converts these values to strings. For colored strings, use JobDataRenderer.

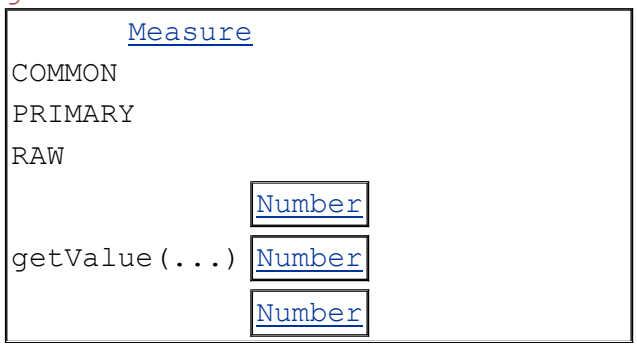
4.4 Scaling

A measured value can be usually expressed in several units. ACNET provides three scales of notations: raw, primary, and common. Raw data is that obtained directly from the ADC in a frontend. Primary (or engineering) value usually describes the ADC input in volts. Common value is something that is supposed to be measured: beam current, outside temperature; in its original units: amps, degrees, etc.

SCF provides the automated scaling for all reading values. For settings, the user must specify the corresponding scale ID along with the value.

The Measure object is used to combine several multiscale values. It has COMMON, PRIMARY, and RAW constants that identify the three common scales; and getValue() method that allows to obtain an individual numerical value on the given scale.

gov.fnal.controls.scf.items.Measure



4.5 Devices

Devices are the most common job items. There are four kinds of them:

AtomicDevice
Elementary ACNET device. Has the distinct name, like *m:outtmp*; array index; and one or several assosiated properties, like *reading*, *setting*, *analog alarm* and so on.

EmptyDevice
Placeholder that has only a text label and does not take part in the data acquisition anyhow.

CompositeDevice
Collection of any other devices (nested composite devices are permitted).

ArrayDevice
Composite device that describes an ACNET array device. Has a distinct name, start array index, and end array index. Includes necessary amount of atomic devices that have the same name and individual array indices inside the range specified.

None of the listed devices implements DataHolder interface. Instead, the values are cached in the atomic device properties. There are five of them:

- *abstract class* DeviceProperty *implements* DataHolder
  - AnalogAlarmProperty
  - ControlProperty
  - ReadingProperty
  - SettingProperty
  - StatusProperty

Each implementation of DevicePropertiy provides methods to read and/or set values of this property. Numerical values are returned as multiscale Measure objects. The properties should be created along with the atomic devices and are permanently tight with them.

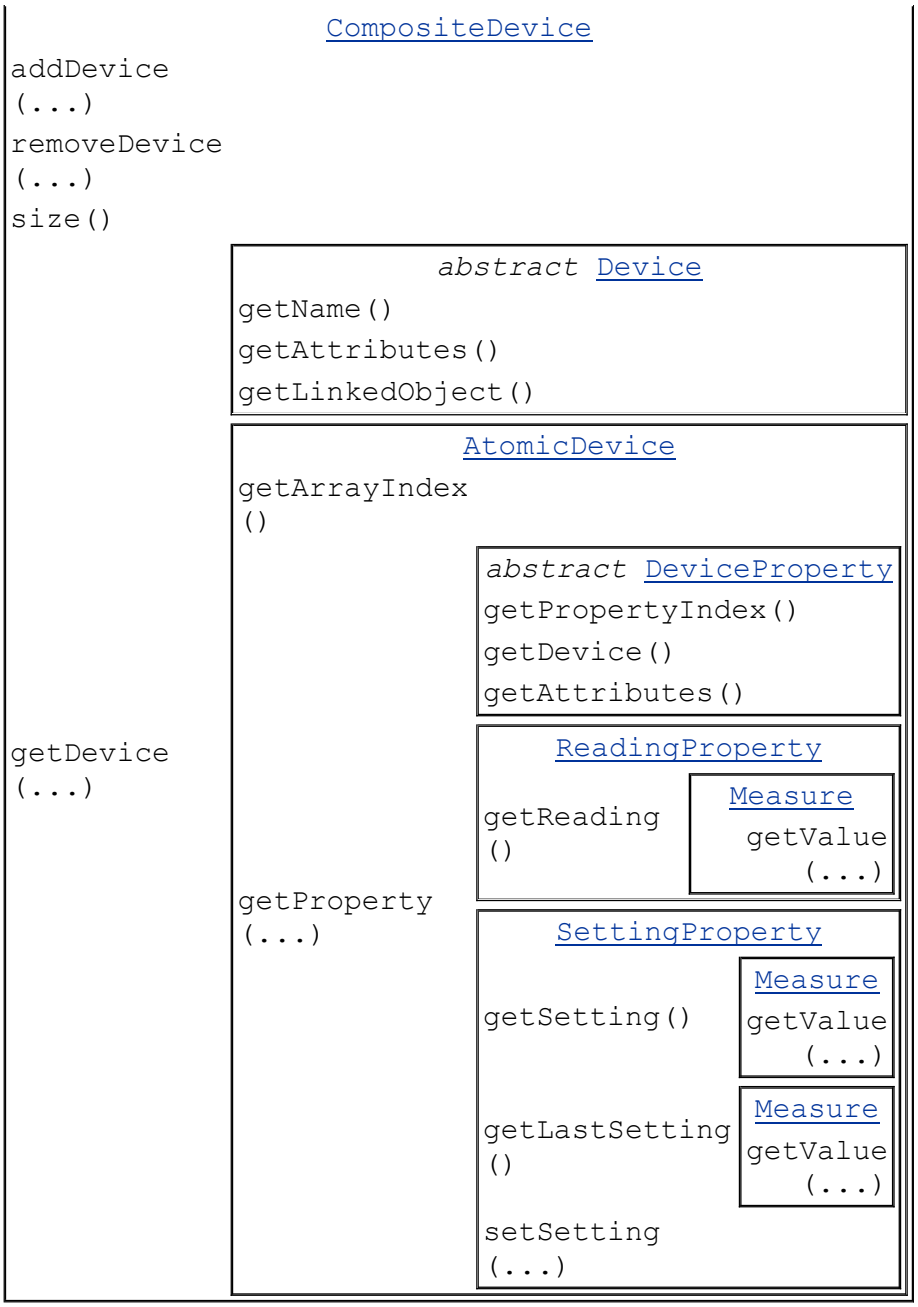
Device Property API

Class	PI Constant in DeviceProperty	Reading Methods	Settng Me
AnalogAlarmProperty	ANALOG_ALARM	:: getMinimum :: getMaximum :: hasAbort :: hasAlarm :: isMinMax :: isAlarmBypassed	:: setAla
ControlProperty	CONTROL	*NONE*	:: setOpe
ReadingProperty	READING	:: getReading	*N*
SettingProperty	SETTING	:: getSetting :: getLastSetting	:: setSet
StatusProperty	STATUS	:: getStatus	*N*

Items can be combined in one or several CompositeDevices as needed. Any device can be linked to a custom object (e.g., a GUI element) through setLinkedObject() method. Each atomic device can have some number of distinct properties. If a DataCallbackListener is used, it will be returning an array of updated device properties (actually, the property instances stay the same, only their values are changed). You can get the affected device by calling deviceProperty.getDevice(), and then get an associated GUI component through getLinkedObject().

The most general device structure is shown on the diagram:

General Device Stricture



The device names used in SCF comply with ACNET Device Request Format

The name of a device without implication of its properties, or unqualified device name, always has the colon (:) on the second position. Array indices can be specified in square brackets; omitted index is assumed to be 0. The array index must be a non-negative integer. The beginning and ending indices in a range are separated by the colon; the ending index can not exceed the beginning one. For example:

```
G:HLSLEV[9]    is an atomic device
```

```
G:HLSLEV       is the same as G:HLSLEV[0]
G:HLSLEV[0:12] is an array device
```

The name of a device property, or qualified device name, has a property qualifier on the second position.

Property Qualifiers

: ;	Reading.
_	Setting.
&	Control.
|	Status.
$	Reserver for Digital Alarm, not implemented.
@	Analog Alarm.

If you need to create a device, the are two alternatives.

4.5.1 Obtaining From The Database

It is guaranteed that the returned device exists in the system. It will have DeviceAttributes and all available properties. Each property, in its turn, will have DevicePropertyAttributes. The device can be directly used in the data acquisition job, either alone, or as a composite device item.

The procedure of obtaining devices through JNDI is described in §3.2. An ArrayDevice instance will be returned unless this is a family device. The array index will always be 0. Atomic device's deriveDevice() method should is used to create a new instance, with a different array index and subset of properties. All extended attributes of the original devices and properties will be retained.

Some trivial example:

```
// Setting up default naming factory...
System.setProperty( Context.INITIAL_CONTEXT_FACTORY,
  "gov.fnal.controls.scf.naming.InitialContextFactory" );

// Creating root context...
Context ctx = new InitialContext();

// Getting I:QC206[0] with full set of properties...
AtomicDevice d0 = (AtomicDevice)ctx.lookup( "device/i:qc206" );

// Deriving new I:QC206[14] only with READING and STATUS props...
AtomicDevice d1 = (AtomicDevice)d0.deriveDevice(
  new int[]{ DeviceProperty.READING, DeviceProperty.STATUS },
```

```
  14
);

// What is the device description?..
String dsc = d1.getAttributes().getDescription();

// How many STATUS elements in the array are allowed?...
DeviceProperty p = d1.getProperty( DeviceProperty.STATUS );
int size = p.getAttributes().getArraySize();
```

4.5.2 Parsing Device Name

The device names parser is build in DeviceRequest class. Instances of this class are used as arguments in several methods that create various kinds of devices. However, most of these methods can accept the device name as a string an then call DeviceRequest internally. It is preferred to use strings.

Depending on the argument format, Device.parse() method returns either a single atomic device, or an array device that has several atomic devices inside. In both cases, each terminal atomic device includes one property that is defined by the device name qualifier. For example,

```
// Creating single device...
AtomicDevice d0 = (AtomicDevice)Device.parse( "M:OUTTMP" );
DeviceProperty pr0 = d0.getProperty( DeviceProperty.READING );
// ... pr0 will be an instance of ReadingProperty
DeviceProperty ps0 = d0.getProperty( DeviceProperty.SETTING );
// ... ps0 will be null
int ai0 = d0.getArrayIndex();
// ... ai0 will be 0

// Creating an array device...
ArrayDevice d1 = (ArrayDevice)Device.parse( "I_QC210[0:11]" );
int size1 = d1.getSize();
// ... size1 will be 12
AtomicDevice d2 = (AtomicDevice)d1.getDevice( 5 );
DeviceProperty pr2 = d2.getProperty( DeviceProperty.READING );
// ... pr2 will be null
DeviceProperty ps2 = d2.getProperty( DeviceProperty.SETTING );
// ... ps2 will be an instance of SettingProperty
int ai2 = d2.getArrayIndex();
// ... ai2 will be 4
```

The device name can accepted by AtomicDevice constructors as an argument. In this case, the property and array index from the name can be overwritten by using other arguments. Unless property ID(s) are specified in a separate argument, the device will have the single property according to the device name qualifier. For example:

```
AtomicDevice d0 = new AtomicDevice( "I_QC210" );
```

```
DeviceProperty pr0 = d0.getProperty( DeviceProperty.READING );
// ... pr0 will be null
DeviceProperty ps0 = d0.getProperty( DeviceProperty.SETTING );
// ... ps0 will be an instance of SettingProperty
DeviceProperty pt0 = d0.getProperty( DeviceProperty.STATUS );
// ... pt0 will be null
int ai0 = d0.getArrayIndex();
// ... ai0 will be 0

AtomicDevice d1 = new AtomicDevice( "I_QC210",
    new int[]{ DeviceProperty.READING, DeviceProperty.STATUS },
    4 );
DeviceProperty pr1 = d1.getProperty( DeviceProperty.READING );
// ... pr1 will be an instance of ReadingProperty
DeviceProperty ps1 = d1.getProperty( DeviceProperty.SETTING );
// ... ps1 will be null
DeviceProperty pt1 = d1.getProperty( DeviceProperty.STATUS );
// ... pt1 will be an instance of StatusProperty
int ai1 = d1.getArrayIndex();
// ... ai1 will be 4
```

ArrayDevice constructor can also accept the device name. In this case, an array device will be created every time, even if the name describes a single one. Each terminal atomic device will have one property according to the device name qualifier.

Once an AtomicDevice is created, its properties can be updated with addProperty() and removeProperty methods. This is true for both individial atomic devices, and for array device elements. The array index must be specified in the constructor and can not be changed after that.

4.6 Event Items

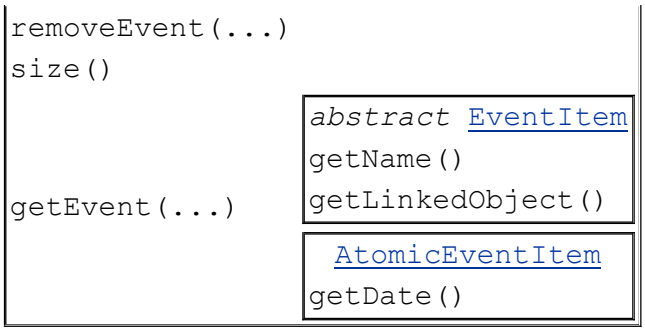
Event items can be used in the data acquisition jobs to monitor various events in the system. SCF supports all the event items that are supported on the DAE. AtomicEventItem represents an individual event, CompositeEventItem can contain several others. Event items and devices can not be combined in one job.

In order to create an atomic event item, you need to put its reconstruction string in the constructor argument. This string also is considered to be the event item name, and is returned back by getName() method. For clock events the string will be:

e,<hex-number>,[**h**|**s**|**e**],<ms-delay>

For example, *e,0F,e,100* is a soft- or hardware clock event #0F provided with 0.1s delay.

CompositeEventItem
addEvent(...)



AtomicEventItem implements DataHolder interface, that means that it also caches data received from the server. For event items, this data is a tiemstamp when the event has occured. This value can be get through getDate() method.

4.7 Plotting Jobs

In real life, client programs can get data as explained above only at a pretty low speed. Various graphical applications usually require higher rates, though less detailed data. SCF supports Fast Time Plot (FTP) and Snapshot Plot (SNP) protocols, that are used by Data Acquisition Engines to exchange this kind of information.

On a DAE, both FTP and SNP plots are defined in complicated string expressions known as plot requests. The request describes rate, duration, number of points, trigger events, and rearm conditions. These requests can be build with D43 VMS page, or manually.

In SCF, FTP and SNP data can be read in plotting jobs. The data event in these job must be an instance of ScfCommonEvent with the proper plot request (with "f" typecode) specified in its constructor. In plotting jobs, only *reading* properties are taken into the account. Received data is not cached in DeviceProperty. Instead, the samples should be get through PlotCallbackListener as an array of doubles, that holds timestamps (even items) and corresponding values (odd ones).

4.8 One-Shot Data Acquisition

If an application needs to read or set data only occasionally, QuickData class can be used. It provides simple methods to manipulate data without implicitly creating jobs. In fact, jobs are still have to be created, but this is made automatically inside the class when the user calls one of its methods. Reading jobs are stored in a pool for a certain period of time, so if the same device is

requested twice, the previous job instance will be used.

In order to receive data from QuickData class, the SCF connection must be opened as described in §2. QuickData supports only regular jobs (no FTP or SNP); only devices (no event items); and three device properties: *reading*, *setting*, and *status*.

< prev contents
security, privacy, legal