Accessing Process Variables in Control System Studio

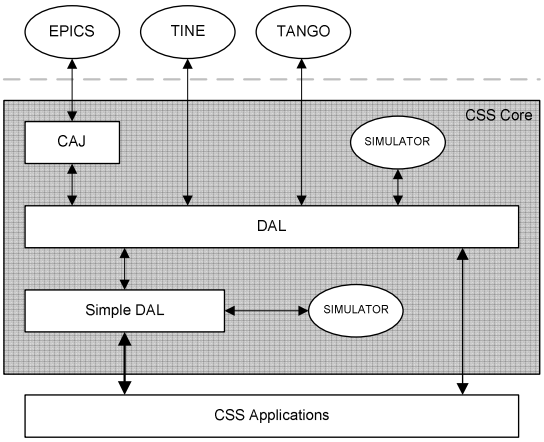
Sven Wende

C1 WPS GmbH
Vogt-Kölln-Straße 30
D-22527 Hamburg
[sven.wende]@c1-wps.de

1 Abstract 2
2 Process variable address syntax 3
2.1 Syntax for EPICS 3
2.2 Syntax for simulated channels 5
3 SimpleDAL API 6
3.1 How to obtain a connection service instance 7
3.2 How to create process variable addresses 7
3.3 How to read process variable values once 7
3.4 How to write process variable values once 8
3.5 How to listen to process variables permanently 8
4 Monitor open connections 9

1 Abstract

The Control System Studio (CSS) is an Eclipse RCP based development platform and the fundament for many applications. As most of these applications deal with process variables and connections to control systems, the CSS Core provides the necessary APIs for a convenient start.



The heart of the connection APIs is the Data Access Layer (DAL) which was developed by Cosylab. It uses CAJ (Channel Access Java), which is a pure Java implementation of the Channel Access protocol, to connect to EPICS control systems. In the (near) future it will be possible to connect to TINE and TANGO control systems through DAL as well. A TINE integration is already available as Beta. DAL is an inherent part of the CSS Core but can also be used as a library in any other Java application.

SimpleDAL is a connection layer built on top of DAL. It provides a slim, less complex API that allows for a much easier start for developers dealing with process variables in their applications. Using SimpleDAL implies a certain syntax for process variable addresses that enables applications to make use of the following features:

- access different control systems (e.g. TINE and EPICS) in one application
- use characteristics, a concept for resource saving access to record fields
- query process variables in different types
- use simulated channels
- address system functions as process variable

This paper describes the SimpleDAL API, the implied process variable syntax and a view that comes with the CSS core which provides on overview for all channels that are currently in use.

2 Process variable address syntax

The general syntax of a process variable address ist defined as follows. For a better understanding of the notation we refer you to literature on the Extended Backus-Naur[1] Form which is the metasyntax [2]we use here.

```
[1] address ::= [protocol] id [type]

[2] protocol ::= ('dal-epics' | 'dal-tine' | 'dal-tango' | 'local') '://'

[3] id::= (letter | specialcharacter) +

[4] type ::= ', ' ('double' | 'int' | 'long' | 'string' | 'enum')

[5] letter::= 'A' | ... | 'Z' | 'a' | ... | 'z'

[6] specialcharacter::= ':' | '/' | '\' | '.' | '[' | ']'

[7] number ::= digitWithoutZero (digit)*

[8] digit ::= '0' | ... | '9'

[9] digitWithoutZero ::= '1' | ... | '9'
```

As you can see there are 3 optional and 1 mandatory fragments that constitute a full process variable address (line 1).

The `protocol` (line 2) is optional and defines the connection protocol. If you don´t specify a prefix, a default protocol is chosen according to the settings of the `CSS-Core/Control-System` preference page.

The `id` (line 3) is mandatory. It must be a globally unique name identifiying the information you want to address.

The `type` (line 5) is optional, too. It can be used to specify the expected return type for channel values explicitly.

2.1 Syntax for EPICS

When EPICS channels are addressed, we can concretise line 3 to:

```
[3a] id::= recordname ['.' fieldname] [characteristic]

[10] recordname ::= (letter | specialcharacter)+

[11] fieldname ::= (letter)+

[12] characteristic ::= '[' (letter)+ ']'
```

[1] e.g. http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf
[2] in short: `x` = a variable, `'x'` = literal, `(x)` = grouping, `(x)*` = x may occur zero or more times, `(x)+` = x may occur zero or more times, `a | b` = a or b, `[a]` = a is optional

As you can see an EPICS process variable is always identified by its `recordname` (line 10) which is therefore mandatory.

Optionally a `fieldname` can be provided to address a single field of a record (see line 11). If no `fieldname` is provided the address defaults to the `.VAL` field.

The `characteristic` (line 12) is optional as well. If defined it allows for accessing additional information of a record without establishing a new connection. All characteristics of the same record share the same connection. So in general it is a good idea to use characteristics whenever possible to save system resources. To give you an idea of what´s already possible with characteristics take a look at the following list:

- `[Position]` - position
- `[Description]` – long description
- `[displayName]` – short description
- `[propertyType]` - type
- `[resolution]` – number of bits used for ADC conversion of analog value when sampled
- `[minimum]` – minimum allowed value
- `[maximum]` – maximum allowed value
- `[graphMin]` – minimum allowed value when displayed (e.g. in a chart)
- `[graphMax]` – maximum allowed value when displayed (e.g. in a chart)
- `[format]` – C print-f style format that is used to render the value
- `[units]` – units of the value
- `[scaleType]` – scale type for plotting (linear or logarithmic)
- `[warningMax]` – upper warning limit
- `[warningMin]` – lower warning limit
- `[alarmMax]` – upper alarm limit
- `[alarmMin]` – lower alarm limit
- `[sequenceLength]` – sequence length
- `[enumValues]` – enum value array (returns Object[])
- `[enumDescriptions]` – enum value descriptions (returns String[])
- `[bitDescriptions]` – bit descriptions (returns String[])
- `[conditionWhenSet]` – active bit significance
- `[conditionWhenCleared]` - inactive bit significance
- `[bitMask]` – bits relevance

In the following we provide some examples to help you getting a better understanding of the described syntax.

```
mychannel
mychannel.VAL
mychannel.LOLO
mychannel[minimum]
mychannel[timestamp]
dal-epics://mychannel
dal-epics://mychannel[maximum]
dal-epics://mychannel, String
dal-epics://mychannel[enumDescriptions], enum
dal-tine://mychannel
```

Figure 1: examples for channels

2.2 Syntax for simulated channels

The `local://` protocol can be used to address simulated channels. Simulated channels live in the local Java Virtual Machine (JVM) and are created on demand. A simulated channel is born on the first connection attempt and dies immediately when it is no longer needed.

At the moment there are the following 4 different kinds of simulated channels:

- channels with a fix static value
- channels generating random numbers
- channels counting numbers down
- channels accessing system functions

To make use of those channels you have to apply certain rules when choosing the process variable address. We concretise the syntax for the `id` to:

```
[3b] id ::= [name] [random | countdown | system]

[13] name ::= (letter | specialcharacter)+

[14] random ::= 'RND:' number ':' number ':' number

[15] countdown ::= 'COUNTDOWN:' number ':' number ':' number

[16] system ::= 'SINFO:' info ':' number

[17] info::= 'ApplicationId' | 'HostId' | 'QualifiedHostname' | 'UserId' |
                'MaxMemory' | 'FreeMemory' | 'TotalMemory' | 'SystemTime'
```

An arbitrary name can be used to create a static simulated channel.

To establish a random number generating channel you have to provide 3 numbers that define the range of random values (no. 1+2) and the update frequency in milliseconds (no. 3) .

Constructing a countdown channel is similar to random number channels. The first two numbers define the range of values (from – to) and the third number defines the update frequency in milliseconds.

To access system information you have to provide one of the following (self explaining) identifiers followed by a number for the update frequency again.

- `ApplicationId`
- `HostId`
- `QualifiedHostname`
- `UserId`
- `MaxMemory`
- `FreeMemory`
- `TotalMemory`
- `SystemTime`

Take a look at the following examples to get the idea.

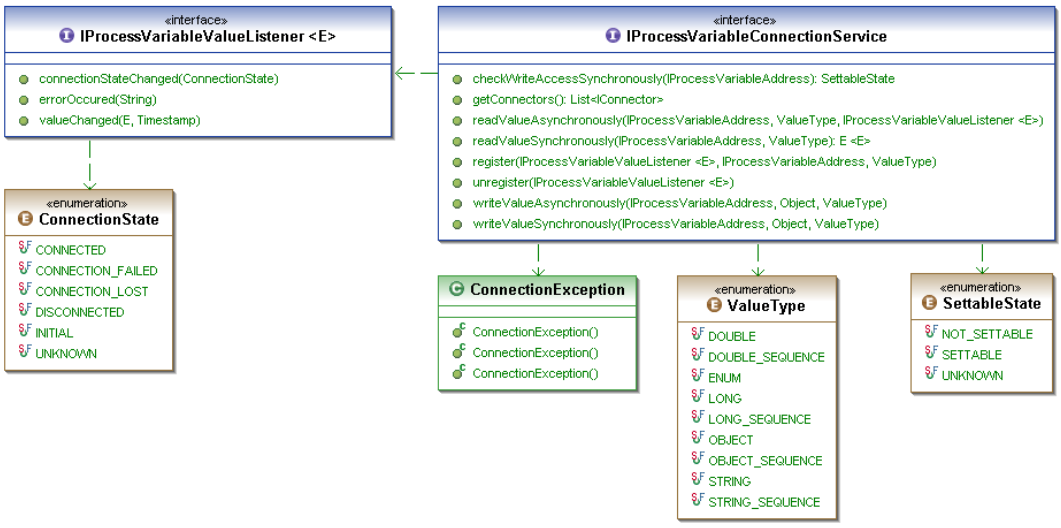
`local://mychannel` (static)
`local://mychannel[minimum]` (static)
`local://mychannel RND:1:100:1000` (random number between 1 and 100 each second)
`local://mychannel RND:2:3:100` (random number between 2 and 3 each 100 milliseconds)
`local://mychannel COUNTDOWN:100:0:1000` (counts down from 100 to 0 per second)
`local://mychannel COUNTDOWN:9:0:60000` (counts down from 9 to 0 at 1 minute update rate)
`local://mychannel SINFO:FreeMemory:10000` (delivers free memory each 10 seconds)
`local://mychannel SINFO:HostId:60000` (delivers host id each minute)

Figure 2: examples for local channels

3 SimpleDAL API

The following class diagramm illustrates the most important parts of the SimpleDAL API. The main service interface `IProcessVariableConnectionService` can be used to establish connections to process variables using the following access patterns:

- read a single value (asynchronously + synchronously)
- write a single value (asynchronously + synchronously)
- check write access (synchronously)
- register a listener (`IprocessVariableValueListener`) that permanently receives updates



We will now provide some comprehensive code examples that demonstrate the API usage.

3.1 How to obtain a connection service instance

It is preferable to share the same service instance accross all CSS applications. Therefore the `getProcessVariableConnectionService()` method of the factory (see code snippet) delivers a singleton instance of the `IProcessVariableConnectionService`.

```
// get a service instance (all applications using the same shared instance will share channels, too)
IProcessVariableConnectionService service = ProcessVariableConnectionServiceFactory
                                            .getDefault().getProcessVariableConnectionService();
```

Figure 3: source code for obtaining a service instance

3.2 How to create process variable addresses

When using the `IProcessVariableConnectionService` interface you have to provide addresses for process variables. As you can see in the following code snippet, instances of `IProcessvariableAddress` can be created easily, using a factory as well.

```
// get a factory for process variable addresses
ProcessVariableAdressFactory pvFactory = ProcessVariableAdressFactory.getInstance();

// create a process variable address
IProcessVariableAddress pv = pvFactory.createProcessVariableAdress("dal-epics://myproperty");
```

Figure 4: source code for creating process variable addresses

3.3 How to read process variable values once

When want to read the values or properties of a process variable you are free to choose between synchronous and asynchronous access.

If you decide for a synchronous call, please consider that the current thread is blocked until the according channel has answered or a certain timeout has expired. The method returns the received value directly or throws an exception in case of any errors.

```
try {
   // read value synchronously
   Double value = service.readValueSynchronously(pv, ValueType.DOUBLE);
} catch (ConnectionException e) {
   e.printStackTrace();
}
```

Figure 5: source code for synchronous read access to a process variable

For asynchronous calls a listener has to be provided. When the according channel answers, the listeners callback methods are invoked. Errors are reported via a callback method, too.

```
// create a listener
IProcessVariableValueListener<Double> listener = new ProcessVariableValueAdapter<Double>() {
     public void valueChanged(Double value, Timestamp timestamp) {
          System.out.println(value);
     }

     public void errorOccured(String error) {
          System.out.println(error);
     }
};

// use listener for asynchronous calls
service.readValueAsynchronously(pv, ValueType.DOUBLE, listener);
```

Figure 6: source code for asynchronous read access to a process variable

3.4 How to write process variable values once

Write access to process variables is similar to read access and can be done in synchronous or asynchronous calls as well. Please ensure that the provided value and the `ValueType` parameter are compatible.

```
// write value synchronously
try {
  service.writeValueSynchronously(pv, 98.0, ValueType.DOUBLE);
} catch (ConnectionException e) {
  e.printStackTrace();
}
```

Figure 7: source code for synchronous write access to a process variable

```
IProcessVariableWriteListener writeListener = new IProcessVariableWriteListener() {
  public void error(Exception error) {
      System.out.println(error.toString());
   }

   public void success() {
      System.out.println("ok");
   }
};

// write value asynchronously
service.writeValueAsynchronously(pv, 98.0, ValueType.DOUBLE, writeListener);
```

Figure 8: source code for asynchronous write access to a process variable

3.5 How to listen to process variables permanently

A common usecase for CSS applications is to observe a channel permanently. To do so you have to register a `IProcessVariableValueListener`.

```
// create a listener
IProcessVariableValueListener<Double> listener = new ProcessVariableValueAdapter<Double>() {
     public void valueChanged(Double value, Timestamp timestamp) {
          System.out.println(value);
     }

     public void errorOccured(String error) {
          System.out.println(error);
     }
};

// register listener for permanent updates
service.register(listener, pv, ValueType.DOUBLE);
```

Figure 9: source code for permanent asynchronous read access to a process variable

SimpleDAL will keep all channel connections open as long as there are any listeners registered. To help saving system resources, please unregister your listener if it´s no longer needed. For safety SimpleDAl references all listeners weakly. When you forget to unregister explicitly, SimpleDAL will remove the listener from it´s internal list as soon as it is garbage collected by the JVM.

```
// unregister a listener explicitly
service.unregister(listener);

// unregister a listener implicitly
listener = null;
```

Figure 10: sourcecode for unregistering permanent listeners

4 Monitor open connections

Open the „Process Variable Connections“ view to lists all open connections. Each line is prefixed by an icon that expresses the connection state by its color and the connection protocol by its text. A green icon represents a working connection whereas a red icon stands for a failing connection. When an icon gets white, the connection is about to be closed because no listeners are attached.

Additionally the view displays the connection state and the latest received value. For convenience you can filter the connections by connection state by using one of the toolbar buttons.

You can also use the context menu on each connection to link the corresponding process variable to other CSS applications or close the connection manually.

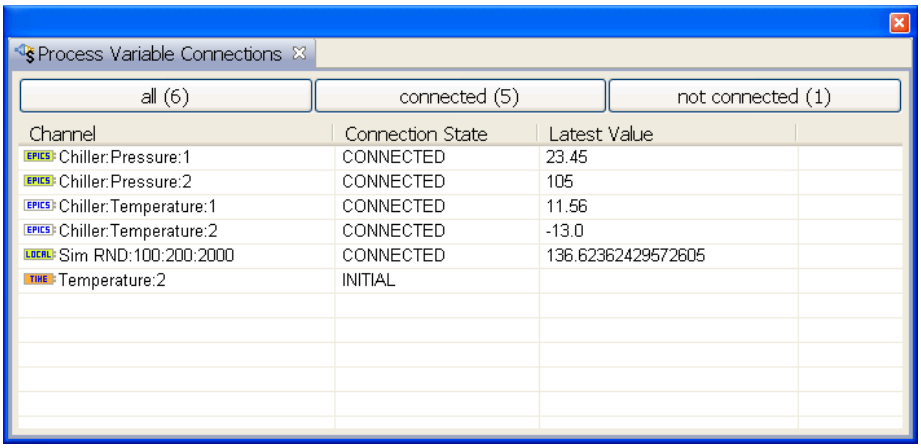


Figure 11: process variable connection view