| | | | |
|---|---|---|---|
| **PRI Name** | eWON Java IOServer interface | | |
| **Access** | OEM, DIST, ALL | | |
| **Since revision** | | | |
| **PRI Number** | PRI-0024-0 | **Build** | 38 |
| **Mod date** | 16. Jun. 2014 | | |

**Preliminary Reference Information**

# eWON Java IOServer interface


***Summary:***
User manual for creation of IOServer in Java

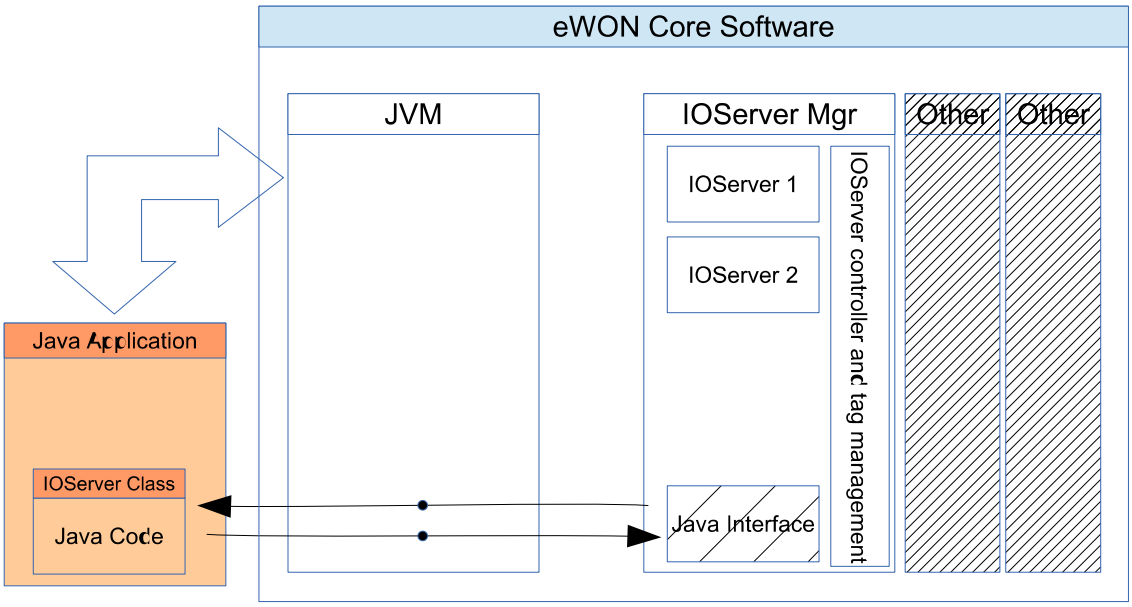

***Table of content***

| | | | |
|---|---|---|---|
| **PRI Name** | eWON Java IOServer interface | | |
| **Access** | OEM, DIST, ALL | | |
| **Since revision** | | | |
| **PRI Number** | PRI-0024-0 | **Build** | 38 |
| **Mod date** | 16. Jun. 2014 | | |

**Preliminary Reference Information**

# Introduction

## *Software Structure*

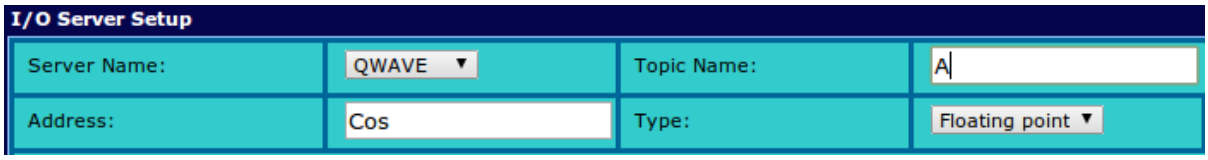
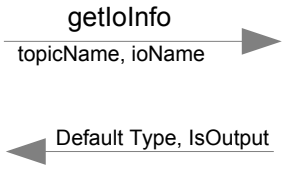


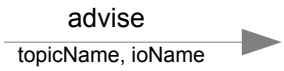
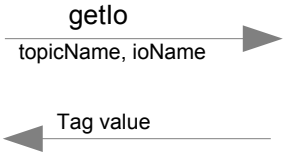
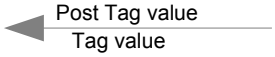
## *Software overview*

In this section you will have an overview of the different interactions existing between the eWON Tags operations and the Java IOServer implementation.

Some concepts of the IOServer library (IOValue, IOTag) will then be explained.

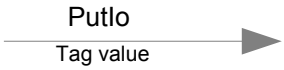
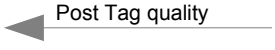
The following table shows the operations taking place in the eWON Core firmware, resulting from user's configuration or from tag management and the operations taking place in the user created Java IOServer.

| **EWON Core side** | | **Java IO Server Side** |
|---|---|---|
| **<u>User creates a new tag:</u>**<br>  <br><br>When you create a new tag, you have to define the IOServer, the “Topic Name”, the “IO Address” and the type used by the eWON to store the tag value (tag type may be different from the default type used by the IOServer to manage the tab – for example a 16 bit modbus register can be stored as float in the eWON).<br>When the tag is created, a number of operations are triggered in the eWON Core. | | |
| Get Io Information:<br>The first request the eWON core will ask the IOServer is: what's the default type for the given tag, and is the tag “writable” (isOutput==true)? Otherwise, the tag is only read-only. |  <br> | IOTag **onGetIoInfo**(String topicName, String ioName)<br><br>The hook in the IOServer class will create an IOTag object, Default type and isOutput attribute will be provided during creation. |

| | | |
|---|---|---|
| Advise[1] Tag:<br>Request monitoring and write access to the tag. |  | void **onAdviseIo**(IOTag ioTag)<br><br>The IOTag created during onGetIoInfo is passed as parameter, and from this moment you are supposed to monitor the tag value&quality and post updates to the eWON Core.<br>You also need to update the ioTag initial value. |
| Get IO:<br>The eWON Core will update the initial tag value by reading the tag's current value. |  | IOValue **onGetIo**(IOTag ioTag)<br><br>The IOTag created during onGetIoInfo is passed as parameter.<br>The Java IOServer will return the current IOTag's value, this is done automatically, with the current value of the IOTag object that you have to update when required.<br>REM: you can set the initial tag quality to “invalid” if you need to wait a first poll to return a valid value. |
| **<u>The tag value is changing in the device:</u>**<br><br>One of the advised tag's value has changed, you need to post this new value to the eWON.<br>IOTag's values must only be posted if their value has changed[2]. | | |
| |  | This operation is initiated by the Java code when required by calling:<br><br>void **postIoChange**(IOTag ioTag) |
| Core Post IO Change: | | |

1 Advise a tag means the eWON wants to monitor the tag and receive update when the tag value changes. From the IO Server point of view, the tag needs to be monitored only when it has been advised.
In the eWON, a tag will never be read or written if it has not been advised first.

2 This is important because if you post the value every time it is read from the device for example, you will overflow the IO queues, and you would possibly log the tag in the historical logging even if not necessary.

| | | |
|---|---|---|
| The new value received from the IO server is posted in the eWON IO queue for processing (alarm, historical, etc.) | | |
| **The eWON needs to change the tag value:**<br>  | | |
| PutIO:<br>eWON Core will post the tag value to the IO Server.<br>REM: this new value will only become effective in the eWON when the value is received back from the IO server by the Post Tag value operation (see above) |  | void **onPutIo**(IOTag ioTag, IOValue ioValue)<br><br>This hook in the IOServer class receives the new value to apply to the IOTag.<br>The value is passed as an IOValue object which contains the value (with advised type – float, integer, Dword, boolean).<br>At this point, you should update the internal IO Server tag's value (writing it to a field bus or whatever), then post back the updated value via **postIoChange** or let your polling mechanism do that (depending on your specific case). |
| | | |
| **The tag quality is changing:**<br>  <br>If your IOServer code detects that one of the tag quality is changing, for example, either you loose the communication with a device and the quality becomes BAD or you recover it and it becomes GOOD, then you need to post this update to the eWON Core. | | |
| |  | This operation is initiated by the Java code when required by calling:<br><br>void **postQuality**(IOTag ioTag)<br><br>The ioTag.setQuality must be |

| | | |
|---|---|---|
| | | called before posting. |
| | | |

**The tag is deleted in the eWON:**

Delete Selected Tag

When the tag is deleted in the eWON core, the Java IOServer will receive an unadvised request.

| | | |
|---|---|---|
| | Unadvise IO<br>Tag value → | void **onUnadviseIo**(IOTag ioTag)<br><br>The IOTag created during onGetIoInfo is passed as parameter.<br>At this time you have to loose track of the given ioTag, remove it from your polling process. The ioTag object (created during onGetIoInfo) will automatically be deleted when you return from your hook. |

## *Main IOServer's classes*

The Java IOServer support is provided by 3 different classes.

### *I The IOTag*

This class maintains all the attributes of a tag (its quality, its value, its advised status). It is used to communicate with many of the IOServer's class primitives.

If you need to store additional details about your specific IOServer's tag, you should ***extend*** the IOTag class to create your own class and instantiate an object of your own class in the **onGetIoInfo** function you will write.

### *II The IOValue*

This is a very simple class designed to hold the value of a tag.

This class represents a value stored with one of the following types: Integer, Dword, Float, Boolean. These are the types currently supported by the eWON tags.

We could have used a standard **double** type to hold the value of all tags, but it would imply the following drawbacks:

- All tag manipulations would force **double** computation making the Java code CPU intensive.
- In case a new type not compatible with **double** would be introduced, we would not be able to manage it.

### *III IOServer*

This is the parent class of your IOServer, it is an abstract class in which some hooks still need to be defined (like onGetIoInfo, onAdviseIo, onUnadviseIo,..).

This class will receive events from the Java Event Handler and will call the required hooks to provide answers to the eWON Core.

This class also contains the primitives that will let you post updates to the eWON core.

## Java IO Server structure: by examples

The Java Doc contains a detailed description of each class and member for the IOServer, IOTag and IOValue, but we will introduce here a very basic IOServer example to show an overview of a typical IOServer implementation.

This example implements an IOServer with the following tags:

REM: there is a global “mAngle” value that counts from 0 to 360 by steps of 10 at a 1second rate (1000 msec sleep in the thread loop)

Topic is A or B (not used but checked)

| Tag IO Address | Computed value | Range |
|---|---|---|
| sin | mAmpl * sin(mAngle) | -mAmpl → +mAmpl |
| cos | mAmpl * cos(mAngle) | -mAmpl → +mAmpl |
| int32 | mAngle - 180 | (-180 → 180) |
| uint32 | mAngle | (0 → 360) |
| bool | True if mAngle < 180 | True then False (50%) |
| | | |

mAmpl is editable with the IOServer configuration.

## I Instanciate and start your IOServer

```
private static void TestIOServer()
{
   TestIOServer  testIOServer;

   System.out.println("Test: com.ewon.ewonitf.IOServer");

   testIOServer = new TestIOServer();
   try {
      testIOServer.registerServer();
   } catch (Exception ex) {
      System.out.println("Test: com.ewon.ewonitf.Loader");
   }
   DefaultEventHandler.runEventManager();
}
```

This requires essentially 2 things:

- Create your IOServer: testIOServer = new TestIOServer();
- Register the IOServer: testIOServer.**registerServer**();

During the register process, the eWON Core will add a new IOServer to the IOServers' list, if a Java IOServer with the same name exists, it will be replaced by the new one, in that case, all the tags will be disabled and re-enabled, this is essentially provided for debugging purpose to be able to reload the IOServer without restarting the eWON (see also “Java IO Server registration“ on page 13 and “Debugging“ on page 14).

The eWON Core will then call the IOServer.**onRegister** hook where you can add some specific execution.

The onRegister hook defaults to empty, because most initializations can be done in the IOServer constructor.

At the end of the onRegister call, the Java process will signal the IOServer as ready.

As it is an events based mechanism, do not forget to run the EventManager at the end of the instantiation.

## II IOServer declaration

```
public class TestIOServer extends IOServer implements Runnable {
```

Your IO Server will extend the IOServer class. If you have a background processing, you can make it runnable to instantiate a thread.

## III Constructor

```
public TestIOServer()
{
   super();
   setServerName("JIOS");
   /* setTagHelper must NOT be called, the tag helper defaults to "" */
   setTagHelper("{\"topic\":{ \"A\":{},\"B\":{}}}");

   mAngle = 0;
   Thread t=new Thread(this);
   t.start();
}
```

Here we

- define the IOServer name: setServerName("JIOS")
- Configure the tag helper: setTagHelper("{\"topic\":{ \"A\":{},\"B\":{}}}");
  The tag helper will provide the valid list of topics in the user interface, it will also provide an optional help for the IO address syntax. This syntax is described in another manual (Not available yet) and is not mandatory.
- For this particular example, we also initialize the global angle and start our IOServer thread.

## IV Manage the configuration

```
public void onPassConfig(boolean applyConfig, boolean checkConfig) throws Exception
{

   float fBuff=Float.parseFloat(getConfigParam("ampl","1.0"));
   if (checkConfig)
      if (fBuff<0)
         throw new Exception("Amplitude cannot be negative");
   if (applyConfig)
```

```
        mAmplitude = fBuff;
}
```

This function should be defined when your IOServer needs to be configured.

The configuration for the Java IOServer is entered as free text (as the eWON IOServer configuration).

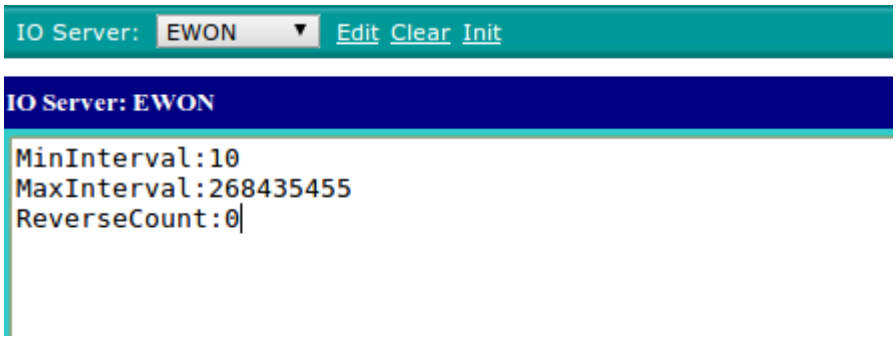
Each line of the text has the following syntax:

*ParamName:ParamValue*

ParamName is a string

ParamValue is interpreted as string also.

Example for eWON IO Server:




The **onPassConfig** function will receive 2 parameters:

- applyConfig: the given config must be applied
- checkConfig: the options passed in the configuration must be validated.

The configuration text itself is maintained internally by the IOServer class, and you have access to the **getConfigParam** function to read each parameter.

## *V Advise (subscribe) sequence: "onGetIoInfo"*

```
public IOTag onGetIoInfo(String topicName, String ioName) throws Exception
{

IOTag ioTag = null;

   if (!topicName.equals("A") && !topicName.equals("B"))
      throw new Exception("invalid Topic");
```

```
    if (ioName.equals("sin") || ioName.equals("cos") )
       ioTag = new IOTag(topicName, ioName, IOValue.DATATYPE_FLOAT32, true);
    ...
    ...

    if (ioTag == null)
       throw new Exception("invalid ioname");

    //if you don't want the tag to have an initial quality == GOOD
    //ioTag.setQuality(IOTag.QUALITY_BAD);

    return ioTag;
}
```

Above is a partial example of onGetIoInfo.

The function must check the topicName and ioName, validate them, then create an IOTag object with the correct default type for the tag and the correct “is output” attribute.

As mentioned earlier, the IOTag is supposed to maintained the state of your tag while it is monitored, if you need to hold other details about your tag, you should **extend** IOTag to create your own augmented class.

Then the IOTag object created is returned and will be maintained by the IOServer class.

Every further calls from the IOServer will directly provide the IOTag instead of the topicName and the ioName (which can be read from the IOTag via IOTag.getIoName and IOTag.getTopicName functions).

### *VI Advise (subscribe) sequence: “onAdvise”*

```
public void onAdviseIo(IOTag ioTag) throws Exception
{

    if (ioTag.getIoName().equalsIgnoreCase("sin"))
       mSinTag = ioTag;
    ...
    ...

}
```

This is the second step of the “advise” sequence, it confirms that the tag must be monitored.

You will receive the IOTag object created during onGetIoInfo.

This IOTag object should be stored in a list where your IOServer can find the “items” to monitor, read their values (by any field bus mean, or by computation from other information) and post these updates back to the eWON.

In this example, there is a discrete number of possible ioName, and there is a dedicated variable for each of them, when the variable is *null*, it means the tag is NOT advised.

## VII Un-advise: “onUnadvise”

```
public void onUnadviseIo(IOTag ioTag) throws Exception
{

   if (ioTag.getIoName().equals("sin"))
      mSinTag = null;
   ...
   ...
}
```

Obviously, this function is called when a tag is deleted from the eWON, and you don't need to monitor it any more.

In this function, you will typically remove the IOTag from a list of all your advised tags.

In this example, we simply put the corresponding member's var to *null*.

## VIII Tag value changed by the eWON Core: “onPutIo”

```
public void onPutIo(IOTag ioTag, IOValue ioValue) throws Exception
{
   ioTag.updateTag(ioValue);
   postIoChange(ioTag);
   System.out.println("JIOS onPutIo: "+ioTag.getIoName());

}
```

This function is called when the tag value is changed by the eWON core.

The IOTag object is passed, with the new value.

At this point, you can apply the new value to the IOTag object (updateTag as in the example above), then this new value must be posted to the eWON core (postIoChange)

or you can write the new value to the "device" via fieldbus and wait for your IO Server to read back the new value and post it.

In any case, the ioValue must be written in the internal IOTag value via updateTag, this new value will only be applied in the eWON core when the postIoChange will be called.

## *The functions you "can"/ "will need to" write*

### *I Abstract*

public abstract void onPassConfig(boolean applyConfig, boolean checkConfig) throws Exception;

public abstract IOTag onGetIoInfo(String topicName, String ioName) throws Exception;

public abstract void onAdviseIo(IOTag ioTag) throws Exception;

public abstract void onUnadviseIo(IOTag ioTag) throws Exception;

### *II Re-definable*

public void onRegister() throws Exception

public void onPutIo(IOTag ioTag, IOData ioData) throws Exception

## *Starting the IOServer program*

### *I Java IO Server registration*

When the Java register function is executed, the new IO server will be registered and this leads to some interaction with the eWON configuration.

There are typically 3 cases:

- The IOServer is autolaunched at boot time
- The IOServer is launched after the eWON as booted (debug or manual)
- The IOServer is re-launched (registered a second time – debugging purpose)

When autolaunched at boot time – with jvmrun and #IOSERVER (see “Autorun at boot time“ on page 15), the eWON will load the configuration (config.txt) normally, but will not enable the tags (meaning that the tags will not be advised yet) until the Java IOServer has started. As soon as the Java IOServer is registered and ready, the tags are enabled.

When the IO Server is registered after the eWON has booted or if the IOServer is re-registered (because you restart the Java during development): all the tags are disabled, then the eWON configuration (config.txt) is cleared inside the eWON, then the configuration is reloaded and all the tags are re-enabled.

**NOTE** If you boot the eWON with Java IOServer tags defined, but the Java IOServer is not started, you will have errors logged, and these tags quality will be invalid. But as soon as you start the Java, as the configuration is reloaded, your tags will become valid again.

## II Debugging

As described in this document, when the eWON Core needs to communicate with the Java, it will trigger an event in the Java, and wait for a reply from the IOServer's Java code.

If the Java is not responding within a specific time, the eWON will consider it hung up and will mark the Java IOServer as “dead”, it will then stop sending commands to the Java IOServer until it is re-registered.

You do not want this to happen, and in a final IOServer you must design your code to reply within the timeout interval in every situations (even if the device associated to this IO server is for example not responding).

During debugging, you may need to step in your Java code, while the Core is waiting for a reply, this could make the Java code appear as dead.

This is why, during debug, you should increase the reply timeout to a huge value, it will prevent other IOServers from running, but as you are focused on your Java IOServer design, this is not a problem.

The function **setJavaReplyTimeout** can be used for this purpose.

Example:

```
public void onRegister() throws Exception
{
   if (DEBUG)
      setJavaReplyTimeout(1000000);   //Let Java freeze the eWON while debugging.
}
```

During debugging, you may need to restart the IOServer multiple times. As described in “Java IO Server registration“ on page 13, this is acceptable but should not be used in a production environment.

### *III Autorun at boot time*

The standard eWON mechanism is used to start the IOServer at boot time → the **jvmrun** file (see PRI-0002-eWON Java Toolkit User guide).

There is though a small difference with the case of an IOServer: the eWON core needs to make sure Java is loaded before it can enable the tags defined in the configuration (when booting), otherwise we would have invalid IO Tags reported in the event file, and the tag would be invalid until the Java is started.

In order to signal that the jvmrun contains an IOServer start, the **#IOSERVER** line must be added at the beginning of the file.

This is a **jvmrun** example complying with this requirement.

```
#IOSERVER
#jvmrun file, place it in /usr
#This file will trigger execution of the JVM at boot time
-heapsize 1M -classpath  /usr/javaEtkTest.jar -emain TestMain
```