JAVAIOC

M.R. Kraimer, 2826 Lake Ave, Osseo MI, 49266, USA

Abstract

EPICS [1] is a set of Open Source software tools, libraries, and applications developed collaboratively. It is used worldwide to create distributed soft real-time control systems for scientific instruments such as particle accelerators, telescopes, and other large scientific experiments. An IOC (Input/Output Controller) is a network node that controls and/or monitors a collection of devices. An IOC contains a memory resident real-time database. The real-time database has a set of "smart" records. Each record is an instance on a record of a particular type. JavaIOC is a JAVA implementation of an EPICS IOC. It is like a Version 3 EPICS IOC, but extends the data types to support structures and arrays.

OVERVIEW

A JavaIOC [2] is an IOC implemented in Java. A running JavaIOC has a "smart" memory resident soft real-time database. It is called smart because a record can be processed. The data is composed of Process Variable (PV) Data, which is structured data composed of 0 or more fields. Each field is accessed via interface PVField. PV Data is described in more detail in the next section.

Associated with each PVField is a Database Field (DBField) interface. A DBField and extensions, e.g. DBRecord provide additional features including record locking and monitoring of puts to any field in a record.

Each record instance has an associated record processor. The record processor supports synchronous and asynchronous processing.

Each record instance has an associated support module. When the record processor is asked to process a record, it calls the process method provided by the support. The support determines the semantics of record processing, Since each field can optionally have associated support, support can call other support. Support can link to hardware, other records, or other entities.

Event and Periodic Database scanners are provided. An event scanner requests that a set of records be processed when a named event occurs. A periodic scanner requests that a set of records be processed at a regular interval.

A JavaIOC has a Database Definition Database (DBD) which has definitions for the following: structure, recordType, and support. A structure definition describes the fields contained in a structure; recordType describes a top level structure; support locates a Java factory that creates support. XML (Extensible Markup Language) is used to create DBD instances. A parser is provided that reads the XML definitions and puts the result into the DBD. DBD instances can be added to a running IOC.

A JavaIOC has an IOCDB (IOC Database) which holds record instances. An XML parser reads record instance files and by also using the DBD definitions creates new record instances. New record instances can be added to a running IOC.

The data in JavaIOC records is accessed via Channel Access (CA). CA accesses the PV data, i.e. PV Data is designed to be exported. NOTE: remote CA is only partially implemented.

The JavaIOC provides a set of generic structure and recordTypes as well as generic support.

The remainder of this paper provides more details about PV Data and about support.

PROCESS VARIABLE DATA

A Process Variable (PV) Database contains record instances where each instance is a structured set of data. A record instance is a top level structure. A structure contains zero or more fields where each field holds data of one of the supported data types.

PV Data Types

The supported data types are: boolean, byte, short, int, long, float, double, string, array, and structure. The types boolean, ..., double are Java primitive types. String is a Java String. An array has an element type that is any of the supported types. A structure has zero or more fields with each field having any of the supported types. Since the array elements can be an array or a structure and since a structure field can be an array or a structure, arbitrarily complex structures are supported. The types boolean, ..., string are defined as primitive. The types byte, ..., double are defined as numeric.

Introspection Interfaces

Field is the base introspection interface. Array and Structure extend Field. Field methods include getType, getFieldName, and getSuportName. Array adds method getElementType. Structure adds getStructureName and getFields. Field and Structure also have a number of convenience methods.

PV Data Interfaces

PVField is the base interface for accessing PV Data. Every PV Data field, including a structure, array, and a record instances has an associated PVField, or extension, that provides access to the data. Among the PVField methods are: getField, getParent, and getPVRecord.

PVBoolean, ..., PVString each extend PVField by providing get and put methods which provide access to the data itself.

PVStructure extends PVField. It provides method getPVFields, which gets the array of PVField interfaces for the fields of the structure. It also provides a number of convenience interfaces. PVRecord extends PVStructure and provides method getRecordName.

PVArray extends PVField and is the base for all array data. PVBooleanArray, ..., PVStructureArray each extend PVArray by providing get and put methods that get and put a sub array. Each has an associated interface BooleanArrayData, ..., StructureArrayData. The get methods expose an underlying Java array via the associated interface. The array interfaces satisfy two important requirements: 1) an array can efficiently be copied to another array of the same type, and 2) an array implementation can expose only a sub array. Two examples where these requirements are important are passing an array over a network and a transient digitizer. In both cases a large array must be transferred in chunks.

PV Factories

Factory FieldFactory creates introspection instances and returns interfaces that can be attached to data fields. So far there has been no need to provide other implementations but base classes are provided.

Factory PVDataFactory creates an implementation of PVField and all it's extensions. It is often desirable to replace the default implementations. Base classes are provided to aid implementation.

Factory ConvertFactory returns an interface Convert, which provides conversion methods. All reasonable conversions between the supported data types are implemented. In addition conversion to/from Java types are provided.

DATA MODEL

The data model has two objectives: 1) support general purpose client tools such as display managers, archivers, and alarm handlers; 2) allow generic support.

The data model is simple: All related data appears together in a structure.

Client tools are normally interested in a data value and possible properties for the data. The data model requires that a structure holding information intended for a client tool have a field named "value". Every other field in the structure, except a null structure, is a property of the value field and the field name is the property name. A null structure is a structure that has no fields and no support.

In support of the model, interface PVField has methods findProperty and getPropertys(). These look for fields but exclude null structure fields. Examples appear later.

Support can be attached to any field. Record support just happens to be the support that is attached to a record instance. If the support is generic it also supports an embedded structure. Generic support looks for fields, by name, that it requires. It does this during initialization by using the introspection interfaces.

Associated with each primitive type and each array of primitive types, the JavaIOC provides a DBD structure and recordType definition. These are generic definitions. Depending on how record instances are defined, a record instance can just hold data, can be an input record, can be an output record, etc. The following is the definition for recordType double.

```
<structure name="double" supportName="generic" >
  <field name="value" type="double" />
  <field name="alarm" type="structure" />
  <field name="timeStamp" type="structure" />
  <field name="input" type="structure" />
  <field name="valueAlarm" type="structure" />
  <field name="output" type="structure" />
  <field name="display" type="structure" />
  <field name="control" type="structure" />
  <field name="history" type = "structure" />
</structure>
```

By default, every field except value is a null structure. Unless a record instance overrides the default, a null structure field is not a property of value.

If a record instance is defined as:

```
<record name = "dataOnly" type = "double" />
```

then dataOnly is a record that has a double value, which has no properties.

If a record instances is defined as:

```
<record name = "example" type = "double" >
   <alarm structureName = "alarm" />
   <timeStamp structureName = "timeStamp" />
   <display structureName = "display" >
      <units>volts</units>
      <limit structureName = "doubleLimit" >
         <low>0.0</low>
         <high>10.0</high>
     </limit>
   </display>
</record>
```

then example is a record instance that has a double value and value has the properties alarm, timeStamp, and display. If a client has a "PVRecord pvRecord" interface to the record than it can ask for:

```
PVField                pvField                    =
pvRecord.findProperty("value");
```

It can than ask for and get a non null result for:

```
PVField pvAlarm = pvField.getProperty("alarm");
PVField pvUnits = pvField.getProperty(
    "display.units");
PVField pvLowLimit = pvField.getProperty(
    "display.limit.low");
```

GENERIC SUPPORT

As a first example of generic support consider the support named "generic". It is the default support for structure double defined in the previous section. It is also the default support for most structure and recordType definitions. It looks at each field in the structure(or recordType) to which it is attached. When the process method of generic is called, it calls the process method of each field that has support.

As mentioned above the double recordType can be used to create a record instance that is an input record. The following is an analog input record:

```
<record name = "ai" type = "double" >
  <input structureName = "linearConvertInput" >
     <input structureName = "pdrvInt32Input" >
        <!--- definitions for portDriver -->
     </input>
     <linearConvert supportName =
               "linearConvertInput">
        <engUnitsHigh>10.0</engUnitsHigh>
        <engUnitsLow>0.0</engUnitsLow>
     </linearConvert>
  </input>
</record>
```

To support this the JavaIOC defines a structure named linearConvertInput, which has fields named value, input, and linearConvert. Structure linearConvertInput has generic as it's support. pdrvInt32Input is a structure and associated support for portDriver, which is a facility for connecting to hardware. When the above record is processed the following happens.

- Support for the record calls the support for input.
    - Input support calls the support for input.input.
        - It gets a value and puts it into input.value. This is the raw value.
    - Input support calls the support for linearConvert.
        - It gets the value from input.value, converts it to engineering units, and puts the result into value.

The JavaIOC supports "device" records. For example a powerSupply structure can be defined as:

```
<structure name="powerSupply"
       supportName="generic" >
  <field name="power" type="structure
       structureName = "double" />
  <field name="voltage" type="structure
       structureName = "double" />
  <field name="current" type="structure
       structureName = "double" />
</structure>
```

A record that is an array of power supplies can be defined as:

```
<recordType name = "powerSupplyArray"
       supportName="generic" >
  <field name = "supply" type = "array"
       elementType = "structure" />
</recordType>
```

Assuming a record instance is properly defined a client could ask for the following and get a non-null return:

pvRecord.findProperty("supply[0]power.value");
pvRecord.findProperty("supply[1].power.value.display");
...

The first request is a request to find the value of the power for the first supply. The second is a request to find the display property of the value of the power for the second supply.

REFERENCES

[1] EPICS: http://www.aps.anl.gov/epics
[2] JavaIOC: http://epics-doc.desy.de/ioc/javaIOC