

# ***JANA: JLab Reconstruction Framework***

***David Lawrence, Jefferson Lab***

***Revision 0.2***



# Table of Contents

---

TABLE OF CONTENTS .....	3
TABLE OF FIGURES .....	5
INTRODUCTION .....	7
OBTAINING AND BUILDING JANA .....	8
QUICK START.....	10
DATA FACTORIES AND DATA OBJECTS .....	11
<i>FACTORY TAGS</i> .....	12
<i>Using Tags</i> .....	12
<i>How JANA Implements Tags</i> .....	14
<i>IDENTIFIERS</i> .....	15
<i>Using identifiers</i> .....	16
EVENT PROCESSORS.....	18
THE JAPPLICATION CLASS.....	19
<i>STANDARD COMMAND LINE OPTIONS</i> .....	19
<i>USING JAPPLICATION IN SINGLE EVENT MODE</i> .....	20
CONFIGURATION PARAMETERS .....	21
<i>USING CONFIGURATION PARAMETERS</i> .....	21
ACCESSING THE DETECTOR GEOMETRY .....	24
ACCESSING THE CALIBRATION DATABASE .....	24
INTERFACING WITH ROOT .....	25
EVENT SOURCES .....	26
<i>THE JEVENTSOURCE CLASS</i> .....	26
<i>Constructor:</i> .....	26
<i>GetEvent:</i> .....	27
<i>FreeEvent:</i> .....	27
<i>GetObjects:</i> .....	27
<i>THE JEVENTSOURCEGENERATOR CLASS</i> .....	28
<i>Description:</i> .....	28
<i>CheckOpenable(string source):</i> .....	28
<i>MakeJEventSource:</i> .....	29
SAVING OUTPUT TO A FILE .....	30
MULTI-THREADED EVENT PROCESSING .....	31
<i>USING MULTIPLE THREADS</i> .....	31
<i>HOW JANA IMPLEMENTS THREADS</i> .....	33
PLUGINS .....	36
DEBUGGING .....	38
INDEX .....	39



# Table of Figures

---

FIGURE 1: FACTORY FLOWCHART. REQUESTS FOR DATA CAN BE THOUGHT OF LIKE ORDERS TO A FACTORY.  
THE FACTORY MUST EITHER “MANUFACTURE” THE DATA, OR RETRIEVE IT “FROM STOCK”. ..... 11

FIGURE 2: TRACKING FLOW CHART. THE DTRACKCANDIDATE FACTORY CAN GET MONTE CARLO DATA  
FROM EITHER THE DTRACKHIT FACTORY WITH THE "MC" TAG (LEFT) OR THE DTRACKHIT FACTORY  
WITH AN EMPTY TAG "" (RIGHT). ..... 14

FIGURE 3: RUNNING A JANA PROGRAM WITHOUT MODIFYING ANY CONFIGURATION PARAMETERS RESULTS  
IN A MESSAGE INDICATING ONLY DEFAULT VALUES ARE BEING USED. .... 23

FIGURE 4: CONFIGURATION PARAMETER OUTPUT USING -PPRINT ..... 23

FIGURE 5: CONCEPTUAL VIEW OF HOW JANA OBJECTS ARE RELATED WHEN RUNNING MULTIPLE THREADS.  
THE JEVENTLOOP AND VARIOUS JFACTORY OBJECTS ARE ALL SPECIFIC TO A THREAD WHILE THE  
JAPPLICATION, JEVENTSOURCECODA, AND JEVENTPROCESSOR OBJECTS ARE USED BY ALL THREADS.  
..... 34



# Introduction

---

The JLab Reconstruction framework or *JANA* is a software package written in C++ that provides the mechanism by which various pieces of the reconstruction software are brought together to fully reconstruct the data. This is motivated in large part by the number of independent detector subsystems that must be processed in order to reconstruct an event. Each of the subsystems' reconstruction packages performs a similar set of actions (in no particular order):

- Read raw data in
- Obtain calibration constants from database
- Modify behavior through configuration parameters
- Provide processed data out

The JANA framework provides a standard way to pass data between packages. Data is passed using STL<sup>1</sup> vectors containing *const* pointers to data objects. By using STL, JANA adheres to a standard in the C++ programming language. By using templates, JANA ensures a level of type safety so fewer errors result and those that do are often caught at compile time. By using *const* pointers, JANA ensures only the producer of the data can change it (packages that take it as input see it as read-only).

If any of the terminology above scares you because you are unfamiliar with templates, STL vectors, etc... then don't be. One of the most important design goals for JANA is to be easy for the user to well, ...use. A few simple examples in the Quick Start section should get you going. The bulk of this manual is dedicated to documenting details about how JANA 's features are implemented.

---

<sup>1</sup> Standard Template Library

# Obtaining and Building JANA

The JANA source code can be obtained from the JANA web page at: <http://www.jlab.org/~davidl/JANA>. The most recent source however, is kept in a subversion repository on the 12gev\_phys group disk (/group/12gev\_phys/svnroot) at Jefferson Lab which can be accessed through the URL <https://phys12svn.jlab.org/repos>. To access it, you need an account on the JLab CUE<sup>2</sup>. Follow these steps to checkout and compile the code:

- ❖ **Create working directory:** All of the source code and binaries will reside in this directory. It can be named anything and placed anywhere. For example: */home/davidl/JANA*.
- ❖ **Set JANA\_HOME environment variable:** The **JANA\_HOME** variable should be set to the working directory you just made. The makefile system<sup>3</sup> uses this to find the source code and place the resulting binaries. For example<sup>4</sup>:

```
setenv JANA_HOME /home/davidl/JANA
```

- ❖ **Checkout the source:** Go into your working directory and checkout the code by doing the following:

```
cd $JANA_HOME  
svn co https://phys12svn.jlab.org/repos/trunk/src
```

Note that this assumes the account you're issuing the command from has the same username as your JLab CUE account. If not, prefix host part of the URL with your JLab account name followed by an '@'<sup>5</sup>

- ❖ **Compile the library:** Go into the *src/JANA* directory and run `gmake`:

```
cd src/JANA  
make install
```

This will build the JANA library and place it in */\${JANA\_HOME}/lib/\${OSNAME}* where *OSNAME* is the uname of the system you're working on (e.g *Linux*). The install step will also create the directory */\${JANA\_HOME}/include/JANA* and copy all of the header files into it.

---

<sup>2</sup> contact the JLab Computer Center if you need don't already have such an account

<sup>3</sup> See GlueX-doc-473 on the BMS system

<sup>4</sup> I use `tsh` in these instructions. I'll leave it to *bash* users to translate where appropriate.

<sup>5</sup> e.g. `svn co https://joe@phys12svn.jlab.org/repos/trunk/src`



❖ **Compile the debugging version of the library:**

```
make DEBUG=yes install
```

A version of the JANA library will be built with debugging symbols and placed in ``${JANA_HOME}/lib/`${OSNAME}` with the name `libJANA_d.a`.

❖ **Compile the janadump program:** Go into ``${JANA_HOME}/src/janadump` and run `make` to build the `janadump` executable:

```
cd `${JANA_HOME}/src/janadump  
make install
```

The `janadump` program is a program that will loop through events, dumping their contents to the screen in an ASCII format that is easily human readable. This relies on the `JFactory` objects' `toString()` method being defined as `janadump` doesn't know anything specific about the objects themselves.

# Quick Start

---

# Data Factories and Data Objects

The JANA framework is built upon the idea of data factories. Figure 1 gives a flowchart that illustrates the factory mechanism. The term factory comes from the mechanism used by industry to fulfill requests for manufactured products. The general idea is this: When data is requested from a factory (i.e. an order is placed) the factory's stock is first checked to see if the requested items already exist. In JANA, a factory only makes one type of object, so if the objects have already been made for this event, const pointers to them are passed back. Otherwise, it must manufacture (instantiate) the objects. The manufacturing procedure itself needs first to get the "parts" from which to build its own objects. These parts are objects produced by other factories. Eventually, one gets down to requesting objects that are not produced by a factory but rather, originate from the data source.

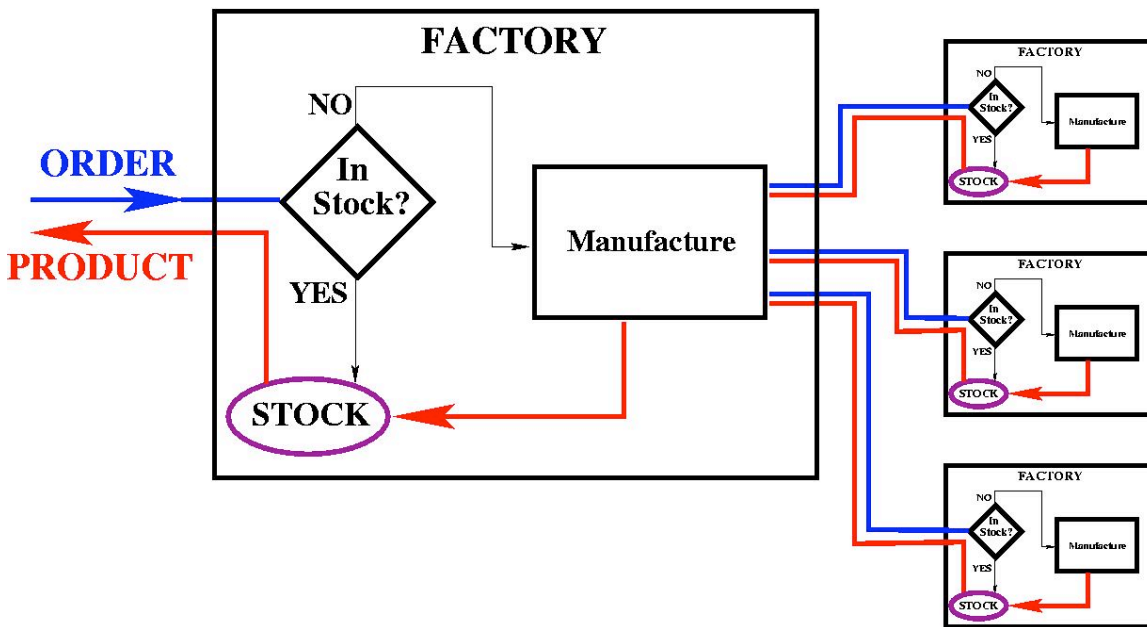


Figure 1: Factory flowchart. Requests for data can be thought of like orders to a factory. The factory must either “manufacture” the data, or retrieve it “from stock”.

There are two big advantages to the method:

1. Manufacture of data is only done “on demand” so CPU cycles are not wasted doing reconstructed values that are never used.
2. Objects are “recycled” in that subsequent requests to the same factory are just given pointers to the objects created in the first request.

## Factory Tags

It often occurs that more than one factory wants to output data objects of the same type. For example: We have a `PID`<sup>6</sup> factory that outputs a set of particle objects. Suppose there is also a `Kaon` factory that outputs particles that are likely to be `Kaons`. The C++ object types produced by both of these factories should be the same (e.g. `DParticle`). But how to distinguish between the two factories? The solution is the factory `Tag`. A `Tag` is just a string and it can be any value. All factories have a tag, but most just use the default empty string (“”). In fact the only reason to use a tag is when another factory is already producing data of the same type.

Tags are used to specify a specific factory. They only need to be unique among factories that produce the same type of data objects. *There are no built-in checks in JANA to ensure that this is the case!* Therefore, if two factories are added which produce the same data type and have the same tag, then the first one added will always be used and the second will be effectively ignored.

In the two following sections, usage of tags is discussed as well as how they are implemented in JANA.

### Using Tags

Using tags is easy. First, a factory needs to be “tagged”. This is done by simply adding a `const char* Tag()` method to the factory class. The `Tag()` method is a virtual method in the `JFactory_base` class. Without explicitly defining a `Tag()` method, the method defined in `JFactory_base` is used which just returns an empty string. Here’s an example of a factory class that has a tag:

```
class DMCTrackCandidate_factory_B:public JFactory<DMCTrackCandidate>{
public:
    DMCTrackCandidate_factory_B ();
    ~ DMCTrackCandidate_factory_B (){};
    const string toString(void);
    const char* Tag(void){return "B";}
...
};
```

Notice that the name of the class has the tag appended with an underscore(`_B`). This is a convention that helps identify the source files that make up a factory. As you can see, it can lead to some very long class names. Adhering to a convention such as this, however, is well worth it when it comes to maintenance of large coding projects.

It is also worth noting that if you use the `mkfactory` script, it will take an optional second argument that specifies a tag for the factory. This is the easiest way to make a “tagged” factory.

---

<sup>6</sup> Particle IDentification

Once you have a tagged factory, you'll surely want to use it. To do this, simply add the tag as an argument to the *Get(...)* call:

```
vector<const DMCTrackCandidate*> mctc;
eventLoop->Get(mctc, "B");
```

Factory tags can also be useful in development. For example, if you wanted to try a new PID scheme, you could place it in a tagged factory that coexists with the old one. This would allow you to compare output of the two schemes event by event.

Another place tags can be useful is when coupled with a configuration parameter to modify the source from which a factory receives its data. For example, in a tracking package, there are two ways in which Monte Carlo data can enter. One is by using the truth tags directly, the other is the “normal” way, through the individual detector packages that present the data as though it were real. Using the truth tags directly allows one to test the tracking algorithm on pristine data and to more easily match up the truth information with the tracking results. Using the other subsystems allows one to exercise the system under conditions more closely related to that of real data. Figure 2 shows the tracking flow chart for JLAB Hall-D that illustrates this. One can see that when processing Monte Carlo data, the *DTrackCandidate* factory must decide whether to take its input from the untagged *DTrackHit* factory or the one tagged “MC”. This can easily be controlled at run time via a configuration parameter. The following listing demonstrates how:

```
// constructor
DTrackCandidate::DTrackCandidate(){
    DTRACK_HIT_TAG = ""; // DTRACK_HIT_TAG is a member of DTrackCandidate
    jparms.SetDefaultParameter("TRK:DTRACK_HIT_TAG", DTRACK_HIT_TAG);
    ...
}

// evnt
derror_t DTrackCandidate::evnt(JEventLoop *loop, int event_number){
    vector<const DTrackHit*> dtrackhits;
    loop->Get(dtrackhits, DTRACK_HIT_TAG);
    ...
}
```

The value of the configuration parameter *TRK:DTRACK\_HIT\_TAG* is used as the factory tag when getting the *DTrackHit*. The default is to use the “real” data path, but if one were to run any JANA program with a *-PTRK:DTRACK\_HIT\_TAG=MC* command line option, the MC path would be used.

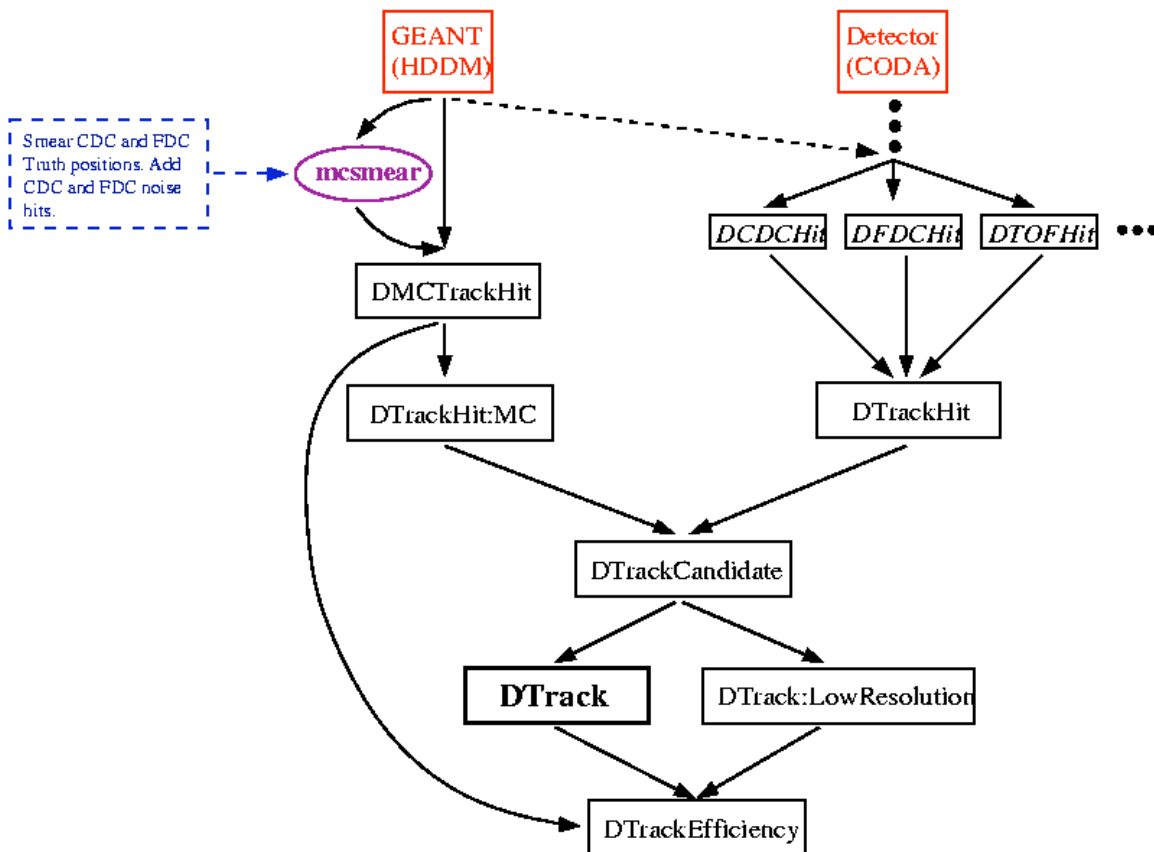


Figure 2: Tracking flow chart. The DTrackCandidate factory can get Monte Carlo data from either the DTrackHit factory with the "MC" tag (left) or the DTrackHit Factory with an empty tag "" (right).

When using tags in this way, always remember to make the default the behavior a novice would expect.

### How JANA Implements Tags

In the preceding section, the `const char* Tag()` method was described. This method is a virtual method of the `JFactory_base` class that defaults to returning an empty string. There are two places where the tag field is used to identify the factory. The first is in the `GetFromFactory()` method in `JEventLoop.h`. This gets called from the `Get()` method (also in `JEventLoop.h`). The tag gets passed into `GetFromFactory()` as a `const char*`. The code that actually searches the list of factories for the one with the right type and tag is shown below:

```

const char* className = T::className();
vector<JFactory_base*>::iterator iter=factories.begin();
JFactory<T> *factory = NULL;
for(; iter!=factories.end(); iter++){
    const char *factory_name = (*iter)->dataClassName();
    if(factory_name == className){
        if(!strcmp((*iter)->Tag(), tag)){
            factory = (JFactory<T>*)*iter;
            break;
        }
    }
}
}

```

The second place the tag can be applied is in a subclass of *JEventSource*. See the chapter on event sources for more details. In a nutshell though, some event sources can supply objects that were created by another JANA program. The objects would need to be stored with the tag identifying the factory that created them. The *GetFromFactory()* method will pass the tag into a call to *GetFromSource()* which eventually passes it to a call to the *JEventSourceXXX* object's *GetObjects()* method. There it can be used to extract objects of the correct tag (and type) from the source. This is admittedly a little complicated, but this design allows the use of object-seekable sources.

One word of caution: One of the concerns voiced when tags were added to JANA is that it might open the door to “competing standards”. For example, someone does come up with an alternative PID scheme and implements it in a tagged factory. The new scheme has some advantages over the old so those in-the-know make use of it as a standard part of their configuration while the rest of the collaboration uses the default. The point being that if a tagged factory is used to develop an alternative that turns out to be better than the current, then the new scheme needs to have its tag removed, and the old needs to either have a tag added or be retired completely.

## Identifiers

Every data object in JANA has a unique identifier attached to it. This is done through its inheritance of the *JObject* class which has a member named *id* of type *oid\_t* (for **o**bject **i**dentifier **t**ype). The *id* is used to uniquely identify an object within an event and can be used by one data object to refer to another. For example, an object representing a cluster in a calorimeter would have a list of *id*'s corresponding to the individual detector hits used to make the cluster. Similarly, a calibrated data hit may have an *oid\_t* member corresponding to the raw hit from which it came. In general a data object will have its own *id* that others can use to refer to it, and a custom set of *oid\_t*'s to refer to the objects from which it was derived.

The value of the *id* data member of a *JObject* is initialized by the *JObject* constructor. If the default constructor is used, *id* will be initialized to the object's pointer (typecast as an *oid\_t*). The *id* should never be used as a pointer though since this is not guaranteed to always be true. In particular, if the object is written to disk and read in later, the original *id* will be preserved and will therefore **NOT** be the same as the pointer. Note that using the pointer does not strictly guarantee uniqueness among *JObjects* of the event. This is because

the factory writer has the freedom to set the value of *id*. Checking for uniqueness with the creation of every *JObject* would incur an overhead that is considered unacceptable. Using the *JObject* pointer will at least guarantee a unique value for every *JObject* for which the factory author does not explicitly overwrite the value of *id*.

## Using identifiers

To use an identifier, one first needs to include a data member in the definition of the class that needs to refer to another object. For example, in the *DTrackCandidate* class there is a data member called *hitid* that holds the *id* values for all of the *DTrackHit* objects that comprise the candidate:

```
class DTrackCandidate:public JObject{
public:
    HDCLASSDEF(DTrackCandidate);

    vector<oid_t> hitid;    ///< ids of DTrackHit objects
    float x0,y0;          ///< center of circle
    float z_vertex;      ///< z coordinate of vertex
    float dphidz;        ///< dphi/dz in radians per cm
    float q;              ///< electric charge
    float p, p_trans;    ///< total and transverse mom. in GeV/c
    float phi, theta;    ///< theta and phi in radians
};
```

As the candidate is created, the *ids* of the *DTrackHit* objects are added to the list:

```
...
trackcandidate->hitid.push_back(trackhit->id);
...
```

Some methods are provided in JANA classes to help obtain pointers to the objects and the factories that created them using the object's *id*. In the most common case, one will have the *id* of the desired object and a pointer to the *JEventLoop* object, but will not have a pointer to the factory that created the desired object readily available. In this case, methods in the *JEventLoop* object can be used to search through its factories and their objects until the desired one is found. Most often, you will know the class of the desired object and so the templated *FindByID* method can be used. In the following example, the pointer to a *DTrackHit* object is obtained using the *oid\_t id*.

```
const DTrackHit* trackhit = loop->FindByID<DTrackHit>(id);
```

This is the fastest<sup>7</sup> way to search for the object pointer since the method can restrict its search to only those factories that provide the specified data type.

<sup>7</sup> Technically, if you have the pointer to the factory object already, it is faster to call the *GetByID* method of that factory directly.



If one wishes to obtain a pointer to the factory that produced the object of the given *id*, then use the *FindOwner* method. This method will return a pointer to a *JFactory\_base* object which must be `dynamic_cast<>` if one wishes to use the methods of the subclass as shown in the following example:

```
JFactory_base *fac = loop->FindOwner(id);
DTrackHit_factory *fac_th = dynamic_cast<DTrackHit_factory*>(fac);
```

The *FindOwner* method is overloaded to also accept a *JObject* pointer:

```
const DTrackHit* trackhit = loop->FindByID<DTrackHit>(id);
JFactory_base *fac = loop->FindOwner(trackhit);
DTrackHit_factory *fac_th = dynamic_cast<DTrackHit_factory*>(fac);
```

If you have the *id* of an object, but don't know the specific subclass of it, you can still obtain a pointer to it using the non-templated version of *FindByID*. This version is slower and will search through every object of every factory until it finds the object with the given *id*.

```
const JObject* obj = loop->FindByID(id);
```

Lower level methods are also provided in the *JFactory* and *JFactory\_base* classes and should be used if you already have a pointer to the factory that you know produced the object with the given *id*. Both provide a method called *GetByID(oid\_t)*. The difference between the two is that the *JFactory\_base* class can only return a *JObject* pointer while the (template) class *JFactory* can return a pointer to the subclass which actually holds the interesting data:

```
// Here, "fac" is a pointer to a JFactory_base object.
// Notice how you still have to cast the obj pointer once retrieved.
const JObject *obj = fac->GetByID(id);
const DTrackHit trackhit = dynamic_cast<const DTrackHit*>(obj);

// Here, "fac_th" is a pointer to a subclass of JFactory_base.
// The cast is done automatically for you with this method.
const DTrackHit *trackhit = fac_th->GetByID(id);
```

## Event Processors

Event Processors derive from the *JEventProcessor* class. This is the base class that implements a type of state machine that is used in the JANA framework. It defines virtual methods that are callbacks for specific conditions occur during the event processing. Namely, the five callbacks are *init*, *brun*, *evnt*, *erun*, and *fini*. The formats of these methods along with the conditions for which they are called are given in the following code snippet.

```
// Called once just before event processing begins
virtual jerror_t init(void);

// Called every time a new run number is detected.
virtual jerror_t brun(JEventLoop *eventLoop, int runnumber);

// Called every event.
virtual jerror_t evnt(JEventLoop *eventLoop, int eventnumber);

// Called every time run number changes, provided brun has been called.
virtual jerror_t erun(void);

// Called after last event of last event source has been processed.
virtual jerror_t fini(void);
```

End user code is usually implemented in a class that inherits from *JEventProcessor*. A pointer to the user's class is registered with the *JApplication* object either by passing it as an argument to the *Run(...)* method of *JApplication*, or adding it explicitly through the *JApplication*'s *AddProcessor(...)* method like so:

```
// Instantiate an object of a class derived from JEventProcessor
MyProcessor *myproc = new MyProcessor();

// It can be registered with the framework either explicitly ...
// (here, japp points to a JApplication object created earlier)
japp->AddProcessor(myproc);

// ... or when you start to process events.
japp->Run(myproc);
```

The framework keeps track of the when to call the callbacks through the *JEventProcessor* class.

# The JApplication Class

---

Every JANA application has a single JApplication object. The JApplication object directs communication between the JEventLoop, JEventSource, and JEventProcessor objects. JApplication is also responsible for creating and monitoring the event processing threads. The constructor for JApplication takes the same arguments main() does so that it can be used to parse the command-line arguments in a consistent way for all JANA applications.

Normally, a program will just pass the same arguments passed to main(int narg, char \*argv[]) into the JApplication constructor. The user may choose to modify the argument list, or even provide an empty one in order to meet the needs of the specific application. Arguments whose first character is a dash (“-“) are considered command line switches. Any switch not known to JApplication is silently ignored so that the user can customize the argument list of the program beyond the JANA default.

## Standard Command Line Options

In order to provide some level of consistency in the command line interface among JANA programs, a command line argument parser has been built into to JApplication constructor. This allows features common to all JANA programs to be accessed without having to duplicate the parse ladder in all programs. For example, the list of event sources comes from the command line. One can also set the number of processing threads ,specify configuration parameters, etc. ... via the command line. These are all handled by the JApplication parser in order to provide consistency across programs that may perform very different tasks. The following table lists the standard command line arguments accepted by JApplication.

Argument	Description
--nthreads= <i>X</i>	Tell the program to run with <i>X</i> processing threads. This will override any value compiled into the program as the second argument to JApplication::Run().
-Pkey= <i>value</i>	Set a configuration parameter. This will initially add a parameter named <i>key</i> with value <i>value</i> to the internal database JANA creates whenever a program is run. It will override the compiled in value. This option may be used multiple times.
-Pprint	Print the configuration parameter database to the screen on start up, once all of the factories have been initialized. Normally, only those parameters that differ from their

	defaults are printed.
<i>source</i>	Any argument (excluding argument 0 which is the program name) that does not start with a hyphen (-) is assumed to specify an event source (file or network based source). These arguments are added to a list and opened in the order that they appear on the command line as needed.
--so= <i>file.so</i>	Attach the shared object file <i>file.so</i> and look for the <code>InitPlugin(JApplication*)</code> routine within. The file <i>file.so</i> must be given with either the full path, or the path relative to the current working directory.
--sodir= <i>directory</i>	Look through all files in <i>directory</i> with names that follow the *.so naming convention and attach each of them as though they were passed with individual --so= arguments.
--plugin= <i>plugin</i>	Attach the specified plugin. The plugin should be the basename of the shared object file (i.e. the filename without a path and without the “.so” suffix. Plugins will be searched for in directories that comprise the plugin search path. See <code>JANA_PLUGIN_PATH</code> environment variable for details.

## Using JApplication in Single Event Mode

There is a class of programs that do not fit well in the run-with-callback paradigm JANA was designed around. These are primarily GUI (Graphical User Interface) programs that need to allow interaction with the users at each event. It is actually not entirely correct to say that GUI programs don't fit well with run-with-callback. In fact, most GUI API's implement this exact method! The problem is that both JANA and the GUI API want to implement the main event loop. Since there can be only one<sup>8</sup> main event loop, JANA has to be used in Single Event Mode so that events are processed upon request from the GUI as opposed to automatically by the `JEventLoop` object.

---

<sup>8</sup> Conner McCloud of the clan McCloud.

# Configuration Parameters

---

Configuration parameters are values used in one or more places inside the reconstruction code that one wants the option to modify without having to recompile. A distinction is made between what we call “configuration parameters” and other types of meta-data such as calibration constants. A configuration parameter has a default value that one does not expect to vary. Calibration constants, however, are expected to vary from run to run and are kept in an external database. Configuration parameters are stored as simple key-value pairs. They cannot be used to store objects or arrays. There are three areas where configuration parameters are expected to play a role: Configuration Recording, Parameter Optimization and Debugging Flags.

Configuration Recording is storing the values for all configuration parameters used by a job in the output file. This can be important since even though the default values of configuration parameters are not expected to change, one has to assume that the reconstruction code will be continually refined, occasionally leading to a new set of defaults. Registering a value as a configuration parameter will record it automatically in the output. It is often much more efficient to extract the values used from the output file than to track down the exact version of the source files and look them up by hand.

An example of Parameter Optimization would be the following: Consider charged particle tracking code, where there is a value representing the maximum number of hits a track seed can have. The name of this parameter is `TRK:MAX_SEED_HITS`. The value of this is used to decide when to stop growing a track seed and fit it. This parameter depends somewhat on the rest of the tracking code and should be optimized for the tracking efficiency. The way to do this is to process the same data set for several values of `TRK:MAX_SEED_HITS` and compare the tracking efficiencies. If this parameter were “hardwired” into the code, it would require an edit-save-compile step for every point. However, by declaring `TRK:MAX_SEED_HITS` as a configuration parameter, its value can be easily changed from the command line without recompiling.

One can also use configuration parameters to turn on debugging features that are normally bypassed in the code. For example, an extra set of histograms may be defined or maybe even the number of bins used in a histogram definition is modified. These types of changes don’t affect the reconstruction aspect, only the amount or format of the output. It should be noted that another facility exists to set the debug level for factories. See the section on *Debugging Your Code* for more.

## Using Configuration Parameters

Both key and value are stored as strings and converted when needed. Conversions are done using the `stringstream` class in templated methods of `JParameterManager`. Any variable type which can be converted by `stringstream` can be used (bool, short, int, long,

float, double, string, unsigned short ,...). Parameters are managed by a global JParameterManager object named jparms. The jparms object is instantiated globally (i.e. without new and before *main()*) so it is accessible by all code in the program. There is an extern declaration of jparms in JParameterManager.h which is included by JEventProcessor.h. Therefore, it is automatically available to all event processors and all factories (i.e. you can just use it).

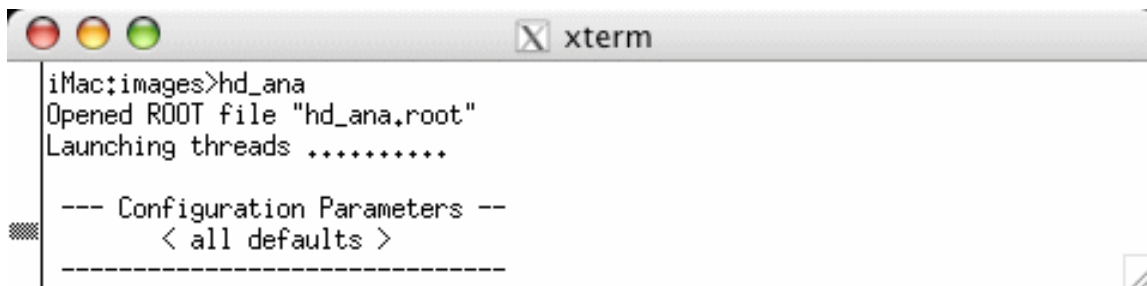
To “publish” a local parameter as a configuration parameter, use the SetDefaultParameter() method of jparms. Generally, you will initialize a data member of a factory to the default value and then call jparms.SetDefaultParameter as shown in the example below. The SetDefaultParameter method will check to see if a parameter with the given key is already defined. If it is, it will overwrite the given variable with the one already stored in jparms. If the given key does not yet exist, it will be added to jparms and initialized using the value of the given variable. Since the factories are not created until after the command line arguments have been parsed, the configuration parameters specified on the command line will take precedence and override the default. Here’s an example showing how to implement several configuration parameters.

```
//-----
DTrackCandidate_factory::DTrackCandidate_factory()
{
  // Set defaults
  MAX_SEED_DIST = 5.0;
  MAX_SEED_HITS = 10;
  MAX_CIRCLE_DIST = 2.0;
  MAX_PHI_Z_DIST = 10.0;
  MAX_DEBUG_BUFFERS = 0;
  TARGET_Z_MIN = 50.0;
  TARGET_Z_MAX = 80.0;
  TRACKHIT_SOURCE = "MC";

  jparms.SetDefaultParameter("TRK:MAX_SEED_DIST",    MAX_SEED_DIST);
  jparms.SetDefaultParameter("TRK:MAX_SEED_HITS",    MAX_SEED_HITS);
  jparms.SetDefaultParameter("TRK:MAX_CIRCLE_DIST",  MAX_CIRCLE_DIST);
  jparms.SetDefaultParameter("TRK:MAX_PHI_Z_DIST",   MAX_PHI_Z_DIST);
  jparms.SetDefaultParameter("TRK:MAX_DEBUG_BUFFERS",MAX_DEBUG_BUFFERS);
  jparms.SetDefaultParameter("TRK:TARGET_Z_MIN",     TARGET_Z_MIN);
  jparms.SetDefaultParameter("TRK:TARGET_Z_MAX",     TARGET_Z_MAX);
  jparms.SetDefaultParameter("TRK:TRACKHIT_SOURCE",  TRACKHIT_SOURCE);

  MAX_SEED_DIST2 = MAX_SEED_DIST*MAX_SEED_DIST;
}
```

JANA programs print out information about configuration parameters at startup. By default, only those parameters that differ from their default value are printed. If only default values are used, it is indicated by an “< all defaults >” message as shown in Figure 3.

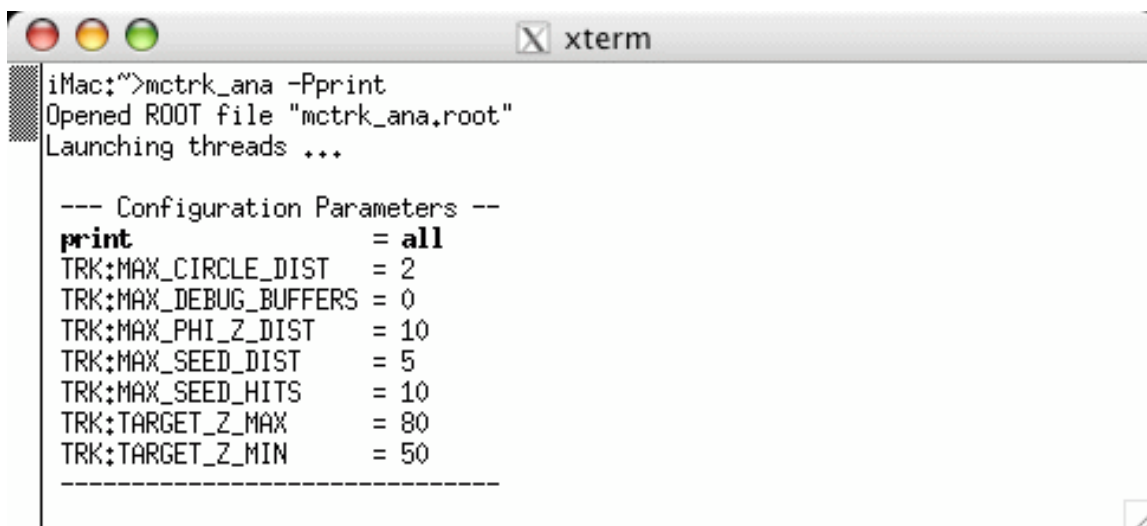


```
iMac:images>hd_ana
Opened ROOT file "hd_ana.root"
Launching threads .....

--- Configuration Parameters ---
      < all defaults >
-----
```

Figure 3: Running a JANA program without modifying any configuration parameters results in a message indicating only default values are being used.

Passing the command-line argument `-Pprint` will print all configuration parameters as shown in Figure 4.



```
iMac:~>mctrk_ana -Pprint
Opened ROOT file "mctrk_ana.root"
Launching threads ...

--- Configuration Parameters ---
print                = all
TRK:MAX_CIRCLE_DIST   = 2
TRK:MAX_DEBUG_BUFFERS = 0
TRK:MAX_PHI_Z_DIST    = 10
TRK:MAX_SEED_DIST     = 5
TRK:MAX_SEED_HITS     = 10
TRK:TARGET_Z_MAX      = 80
TRK:TARGET_Z_MIN      = 50
-----
```

Figure 4: Configuration parameter output using `-Pprint`

## Accessing the Detector Geometry

---

*<Not much has been written on this yet. Some design work is still needed>*

## Accessing the Calibration Database

---

*<Not much has been written on this yet. Some design work is still needed>*



# Interfacing with ROOT

---

The JANA framework is NOT integrated with ROOT<sup>9</sup>. This was done to provide some degree of flexibility for those wanting to use the JANA framework, but not ROOT. It is recognized, however, that a portion of the user base will want to integrate ROOT and JANA in the same application. This chapter provides some hints on how this may be done.

---

<sup>9</sup> ROOT is a C++ analysis package that provides, among other things, histogramming, fitting, and linear algebra classes. See <http://root.cern.ch> for info.

# Event Sources

---

In JANA, an event source's job is to read event-based information from a source (e.g. file) and create objects which it then hands over to the framework. The framework takes ownership of the objects the source creates and disposes of them when they are no longer needed. JANA does not have a file format of its own. Rather, JANA provides an interface through which to incorporate sources of data objects. Because of the generic interface, JANA can support an unlimited number of event source types simultaneously. This means for example that event sources can be written that obtain events from a file, shared memory, or a network socket. This also means that multiple file formats can be implemented in the same executable. For instance, one may have one file format for simulated data and another for real data. The same JANA programs will read in either format and the factories will be agnostic as to the exact source type.

To implement an event source in JANA, you will need to provide a minimum of two classes. The first inherits from `JEventSource` and the second inherits from `JEventSourceGenerator`. Each has just a few virtual methods that you must supply in order to implement the source.

## The JEventSource Class

The `JEventSource` class is responsible for actually reading the event data in from the source and creating the data objects from it. The virtual methods which must be implemented in the derived class are:

```
JEventSourceMyFormat(string source); // constructor  
  
jerror_t GetEvent(JEvent &event);  
  
void FreeEvent(JEvent &event);  
  
jerror_t GetObjects(JEvent &event, JFactory_base *factory);
```

Descriptions are as follows:

### Constructor:

The constructor is responsible for opening the source. Similarly, the destructor should close the source. The exact format of the constructor is not really specified by JANA. The constructor will actually be called from the `JEventSourceGenerator` class you

provide so all that is really required is that they are compatible. See the section on the `JEventSourceGenerator` class below for more details.

### GetEvent:

This should read in the next event from the source and fill in the `JEvent` object's fields with information about the event (see below). Note that the entire event doesn't necessarily have to be read in here. The framework is designed so that one could read in just a header and have the `JEvent` refer to the header info. This could be used, for example, to sparsely read in parts of the event from a file, saving I/O time.

The methods of `JEvent` that should be called are:

```
// Methods of JEvent class that need to be called from GetEvent(...)
void JEvent::SetJEventSource(JEventSource *source);
void JEvent::SetRunNumber(int run_number);
void JEvent::SetEventNumber(int event_number);
void JEvent::SetRef(void *ref);
```

The argument to `SetJEventSource` should always be *this*. The run number and event should be extracted from the event itself. The meaning of *ref* is defined by the specific event source being implemented. Typically though, it will be a pointer to a class, struct, or buffer of some kind that holds the event in its "raw" form.

If an event is successfully read in, then `NOERROR` should be returned (defined in `jerror.h`). Otherwise, `NO_MORE_EVENTS_IN_SOURCE` should be returned.

One should note that it is possible to get the pointer to the `JEventLoop` object that is processing this event through the `JEvent::GetJEventLoop()` method. This is useful for sources where it is more efficient to parse the raw information once, creating all data objects in a single pass through the event buffer. The `JEventLoop` pointer can be used to get pointers to the factories that will take ownership of the data objects. For these types of sources, the `FreeEvent(...)` and `GetObjects(...)` methods need not be implemented.

### FreeEvent:

This gets called once the event has been fully processed and is no longer needed. Any memory associated with the event can then be freed. This is optional and may be omitted if it is not needed.

### GetObjects:

This is where most of the work is probably done. When the framework receives a request for objects of a certain type/tag, it will first try and find the factory corresponding to it. The factory is needed only as a reservoir for the objects. If an appropriate factory is not found, one will be created automatically. This allows one to read in objects of any type without having to explicitly make a factory for them. The factory pointer is passed so that its

CopyTo(...) method may be called to insert the objects created by the event source into it. Thereby passing ownership to the factory.

In the GetObjects method, a check should first be made on what class of object is requested. Since the source presumably knows the classes it can provide, a finite and known number of checks are needed. Perhaps the best way to do this is through a dynamic\_cast of the factory pointer with a NULL check like this:

```
JFactory<DADC> fac = dynamic_cast<JFactory<DADC*>>(factory);
if(fac != NULL){
    vector<DADC*> data;

    //
    // .... create DADC objects from event and push them onto data ...
    //

    fac->CopyTo(data);
}
```

## The JEventSourceGenerator Class

The JEventSourceGenerator class is used by the framework to help it decide which JEventSource is best suited to read events from a given source. The derived class must implement three methods:

```
const char* Description(void);

double CheckOpenable(string source);

JEventSource* MakeJEventSource(string source);
```

Descriptions are as follows:

### Description:

The *Description* method is used to provide the user with information as to which JEventSource-based class is being used to read the source. This is purely informational for the end user. The intent though is that this just returns a short 1-line (possibly 1-word) description of the source.

### CheckOpenable(string source):

This method is called by the framework before opening a new source to figure out which one(s) are capable of reading it (if any). For example, if the source is a file, then the value of *source* is the filename. In that case, CheckOpenable might do something as simple as

checking the suffix of the filename or as complicated as opening the file and checking the header.

The value returned should be a number between 0 and 1 with 0 meaning “absolutely cannot read from this source” and 1 meaning “certainly can read from this source”. The framework will use the largest non-zero value to determine which `JEventSourceGenerator` to have make the `JEventSource` for this source.

### MakeJEventSource:

This gets passed a single argument which is the source name (e.g. file name) which it then passes to the constructor when creating a new `JEventSource`-based object. The pointer to the new `JEventSource` object is returned.

# Saving Output to a File

---

# Multi-Threaded Event Processing

---

Multi-threading is one of the easier concepts to grasp while being one of the harder features to implement. The difficulty in implementation arises simply from needing to get used to the idea that threads must coordinate the use of resources that they share. A thread is a single, independent process of execution. In a way analogous to how a single computer can “simultaneously” run many programs at once, a single program can have many threads that run at once. In fact, deep in the Linux kernel, threads are treated as though they are separate processes. Analysis of large data sets is a natural place to use threads as each event is independent and many events exist in a single file.

Threads have been around for a while, but their popularity has been growing in recent years as multi-processor SMP machines have become more common. In fact, threading will become necessary to take full advantage of the next generation multi-core CPU's currently being developed. The popular Intel x86 chip line has been at the 2.5-3.0GHz level for a while now and the PowerPC family has yet to break (and likely never will) the 3.0GHz limit. Work is currently being done to develop chips with large numbers of cores (20-100) on the same die. Single-threaded programs will utilize only a fraction of the available computing power on the next generation computers so multi-threading should be considered a requirement.

## Using Multiple Threads

The framework itself has multi-threading ability built in. There are two ways a program can be instructed to run with multiple event processing threads:

- Pass a second argument to the *Run()* method of *JApplication* in the source code
- Pass a `--nthreads=N` option on the command line when running the program.

All JANA programs run with a single event processing thread if neither of these is specified. Note that multiple threads can be used even on a single processor computer. You will just not see any performance gains.

The following is an example of a main routine for a JANA application that uses 4 threads by default.

```
int main(int narg, char *argv[])
{
    // Instantiate our event processor
    MyProcessor myproc;

    // Instantiate an JApplication object
    JApplication app(narg, argv);

    // Run though all events, calling our event processor's methods
    app.Run(&myproc, 4); // Tell JANA to run with 4 threads

    return 0;
}
```

The command line always takes precedence. If after compiling the above program I decided I wanted to try running it with only 1 thread, I would run it like this:

```
>hd_ana --nthreads=1 hdgeant.hddm
```

If a program is acting flaky, it's a good idea to try running it with a single thread. If it runs OK in single thread mode, that is a good indication that you are not properly locking a resource so multiple threads are colliding when accessing it. Any programs that expect to run in batch mode (like on an analysis farm) should be thoroughly tested with multiple threads before submitting a large job.

The model for multi-threading in event reconstruction is simple since it lends itself so naturally to it. The fact that events are independent suggests having a single event processed in a single thread. However, both the input(event source) and output must be shared by all threads. For example, the typical job will consist of processing all events from a file, filling histograms as you go. You start off with one input file and you want to end up with a single set of histograms in the end. The threads must be coordinated to ensure only one is trying to read an event from the source at a time. Likewise, only one thread should be modifying the histograms/trees at any given time. The mechanism for locking access to a resource like this is called a mutex (for **mutual exclusion**) and is part of the threads package.

JANA takes care of locking access to the event source, but it is up to the end user to coordinate access to output such as histograms. It can be tempting to enclose the entire contents of the `evnt()` method in a mutex lock. While this is definitely "thread-safe", it also will wipe out any gains from having multiple threads, reducing the program to essentially single threaded operation. To take advantage of threads, all of the factory data should be retrieved outside of the mutex lock. The code example below demonstrates the right way to lock the `evnt()` method.



```

#include <TThread.h>

// evnt
jerror_t MyProcessor::evnt(JEventLoop *loop, int event_number)
{
    // Do NOT place lock here! Most of the CPU time is spent in
    // the Get() calls below!

    // Grab whatever data we need from framework
    vector<const DCDCHit*> cdchits;
    vector<const DFDCHit*> fdchits;
    loop->Get(cdchits);
    loop->Get(fdchits);

    // Now lock the ROOT global area
    TThread::Lock();      // ROOT lock defined in TThread.h

    //
    // .... Fill histograms here using cdchits and fdchits
    //

    // Make sure we release the ROOT lock or the program will hang
    // on the next call to TThread::Lock()
    TThread::UnLock();   // ROOT lock defined in TThread.h
}

```

The above example demonstrates using locks for a program that produces ROOT output. For other packages, something similar must be done when running with multiple threads.

## How JANA Implements Threads

If you're used to sequential programming, then introducing threads will require a slight shift in the way you think about how programs work. The first thing to understand is that an object created in one thread can be used by other threads. More importantly, more than one thread can be "in" the same method of the same object at the same time. This is not all that different from the old idea of being "re-entrant" i.e. that a subroutine can call itself. This is an important concept in the case of event sources. When an event is requested, a `JEventSource` object must be made (if it doesn't already exist) and then an event is read from it. What happens is that the first thread to try and get an event ends up having to create the `JEventSource`<sup>10</sup> object, which it does through the `JApplication` object. `JApplication` keeps track of this so other threads can use the same `JEventSource` to obtain events. The `JApplication`, `JEventSource`, and `JEventProcessor`<sup>11</sup> objects are ones used by all threads. All threads must read from a single source (`JEventSource`), output the result to a single output (`JEventProcessor`) and the information about these is kept in a centrally accessible area (`JApplication`).

<sup>10</sup> `JEventSource` is the base class for all event sources. In reality, this will always be a subclass such as `JEventSourceEVIO`. The base class name `JEventSource` is used for simplicity here.

<sup>11</sup> Similarly to the previous footnote, this will be a subclass of `JEventProcessor`

Some objects are kept exclusive to the thread that creates them. In particular, each thread has its own independently running event loop so each has its own `JEventLoop` object. The `JEventLoop` object creates its own set of factories so each set of factories is also dedicated to a single thread. A conceptual sketch of this is shown in Figure 5. Having a separate set of factories for each set can quickly use up the memory if one is not careful. For instance, if a factory used a large 2-dimensional histogram that took up a significant, but reasonable amount of memory for a single instance of the factory object, it could require unreasonable amounts of memory to run many threads. The worst case scenario would be to cause the system to start using virtual memory that would cause the system to slow to a crawl. In that case, multiple threads would process events at a much lower rate than a single thread!

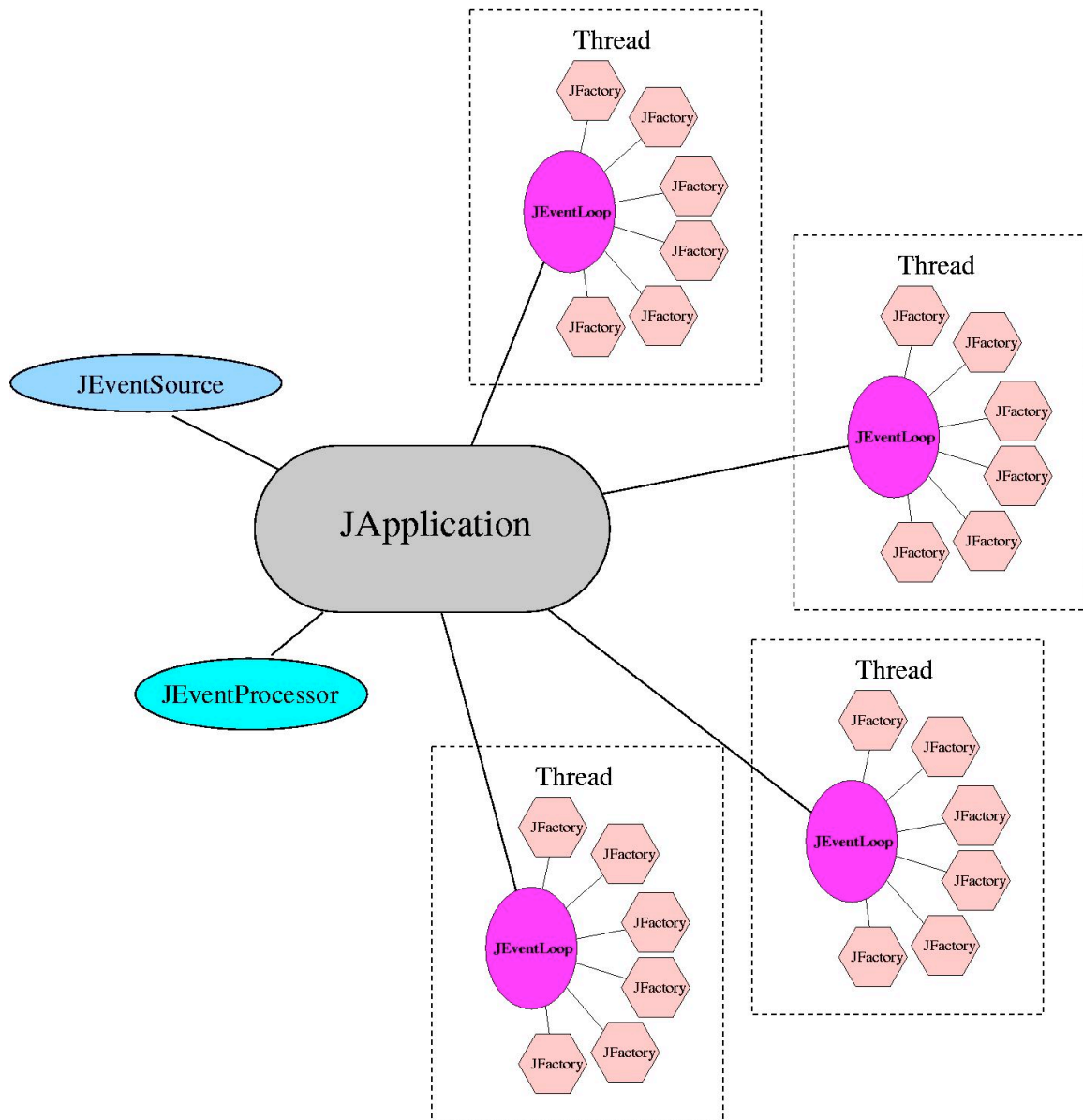


Figure 5: Conceptual view of how JANA objects are related when running multiple threads. The `JEventLoop` and various `JFactory` objects are all

specific to a thread while the `JApplication`, `JEventSource`, and `JEventProcessor` objects are used by all threads.

Threads are launched by the `JApplication::Run()` method. The posix pthreads library is used for threading so `pthread_create()` is called, passing the address of the `LaunchThread()` routine. `LaunchThread()` is a stand-alone routine that is not a member of any class. It is defined in `JApplication.cc` near the `Run()` method however since that is the only place it is used. The `LaunchThread()` routine simply creates a `JEventLoop` object, calls its `loop(...)` method then deletes the object after it returns from `loop(...)`.

All JANA programs are actually run with a minimum of 2 threads. The main thread continues in a “sleepy” loop while the processing threads do all of the work. That is to say, the main thread spends most of its time in `nanosleep()` and wakes up once every 0.5 seconds to monitor progress. Most JANA programs (when not run in batch mode) will periodically update the screen with the number of events processed and the rate at which they are being processed. The main thread is responsible for making those calculations and printing those messages to the screen. It also checks to make sure at least one processing thread is still running so that when they are all done, it can break the “sleepy” loop, print some final statistics and return from the `Run()` routine.

When the `JEventLoop` object is created, it is passed a pointer to the `JApplication` object. This is used by the `JEventLoop` object to register (and eventually deregister) itself with `JApplication`. One pitfall with this method is that if a `JEventLoop` gets stuck somehow in an infinite loop, it will never deregister itself causing the main thread to also be stuck. To address this, each `JEventLoop` passes the address of a “heartbeat” variable that it updates after processing each event. This allows the main thread to kill a processing thread if it appears to be stuck because it has not updated its heartbeat variable more than 3 seconds. Heartbeat monitoring can be turned off by setting the `monitor_heartbeat` member of the `JApplication` object to false:

```
JApplication *japp = new JApplication(narg, argv);  
japp->monitor_heartbeat = false;
```

Turning off heartbeat monitoring is necessary in programs such as event viewers or when programs are expected to read from a source that may block (e.g. from shared memory).

# Plugins

---

JANA can use event sources, factories, and event processors obtained from dynamically linked binary object files (or plugins). The ubiquitous dl library, which is installed and available by default on all major Unix platforms, is used. The dl library only provides for linking “C” style routines so the interfaces are implemented in that way.

When JANA opens a plugin, it looks for a single symbol *InitPlugin* which has the following form:

```
extern "C" {
    void InitPlugin(JApplication *japp)
    {
        // register factory generator...
        japp->AddFactoryGenerator(new MyFactoryGenerator());

        // ...and/or event source generators
        japp->AddEventSouceGenerator(new MyEventSourceGenerator());
    }
}
```

Typically, the *InitPlugin* routine will create *JEventSourceGenerator* and *JFactoryGenerator* based objects that it will then register with the application through the *JApplication::AddEventSourceGenerator* and *JApplication::AddFactoryGenerator* methods. One can also add *JEventProcessor* based objects through the *JApplication::AddProcessor* method. This would be useful for say, implementing the generation and filling of histograms in a plugin so that different histogram sets can be combined into a single output file.

C++ Objects obtained from plugins are given preference to statically linked ones. Specifically, if a plugin provides a *JFactory* object based on the same type and with the same tag as a statically linked one, the one from the plugin will be used and the statically linked one will be ignored.

There are several ways to specify what plugins (if any) a JANA program will attach and use. The first is through the command line. Using the *--plugin=name*, *--so=filename* or *--sodir=directory* options, one can specify a plugin name, a file, or directory in which to search for files respectively. The plugin name is the filename of the plugin minus the “.so” suffix. When specified in this way, the *JANA\_PLUGIN\_PATH* path is searched for the plugin. See the *JApplication* section for more info. on command-line options.

If you wish to specify an entire set of plugins, the user can set their *JANA\_PLUGIN\_DIR* environment variable. This should be a colon separated list of directory paths to search for plugins. Every file found in every directory in the path will be attached as a plugin. If a file is not a shared object, or does not contain the *InitPlugin* symbol, then it will be quietly ignored.

By default, the user is not notified of all files that are looked for as plugins. To get more verbose output, set the `JANA_PRINT_PLUGIN_PATHS` environment variable.

# Debugging

---

# Index

---

## C

configuration parameter, 13

## D

*DANA*, 1, 7, 8, 12, 13, 14, 15

*DEventLoop*, 13, 14

*DFactory\_base*, 12, 14, 15

*DTrackCandidate*, 13, 14

*DTrackHit*, 13, 14

## F

Factory, 12, 13, 14

## G

*GetFromFactory*, 14, 15

*GetFromSource*, 15

*GetObjects*, 15

## I

Identifiers, 15

## M

mkfactory, 12

## O

object-seekable, 15

## S

STL, 7

## T

Tag, 12, 14, 15

Tags, 12, 13, 14