



DANA:

Hall-D Data Analysis Framework

David Lawrence, Jefferson Lab

Revision 0.1

Table of Contents

TABLE OF CONTENTS	3
TABLE OF FIGURES	5
INTRODUCTION	7
OBTAINING AND BUILDING DANA	7
QUICK START	9
DATA FACTORIES AND DATA OBJECTS	9
<i>FACTORY TAGS</i>	10
<i>Using Tags</i>	10
<i>How DANA Implements Tags</i>	12
<i>IDENTIFIERS</i>	13
<i>Methods for using identifiers</i>	14
EVENT PROCESSORS	15
THE DAPPLICATION CLASS	15
<i>STANDARD COMMAND LINE OPTIONS</i>	16
<i>USING DAPPLICATION IN SINGLE EVENT MODE</i>	17
CONFIGURATION PARAMETERS	17
<i>USING CONFIGURATION PARAMETERS</i>	18
ACCESSING THE DETECTOR GEOMETRY	20
ACCESSING THE CALIBRATION DATABASE	20
INTERFACING WITH ROOT	20
EVENT SOURCES	20
<i>THE HDDM EVENT SOURCE</i>	21
SAVING OUTPUT TO A FILE	21
MULTI-THREADED EVENT PROCESSING	21
<i>USING MULTIPLE THREADS</i>	21
<i>HOW DANA IMPLEMENTS THREADS</i>	23
SHARED OBJECTS	25
DEBUGGING YOUR CODE	27
INDEX	29

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". 9

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). 12

FIGURE 3: RUNNING A DANA PROGRAM WITHOUT MODIFYING ANY CONFIGURATION PARAMETERS RESULTS
IN A MESSAGE INDICATING ONLY DEFAULT VALUES ARE BEING USED. 19

FIGURE 4: CONFIGURATION PARAMETER OUTPUT USING -PPRINT 20

FIGURE 5: CONCEPTUAL VIEW OF HOW DANA OBJECTS ARE RELATED WHEN RUNNING MULTIPLE THREADS.
THE DEVENTLOOP AND VARIOUS DFACTORY OBJECTS ARE ALL 24

Introduction

The Hall-**D Analysis** framework or *DANA* 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
- Provide processed data out
- Obtain calibration constants from database
- Modify behavior through configuration parameters

The DANA framework provides a standard way to pass data between packages. Data is passed using STL¹ vectors with *const* pointers to the data objects. By using STL, DANA adheres to a standard in the C++ programming language. By using templates, DANA ensures a level of type safety so fewer errors result and those that do are often caught at compile time. By using *const* pointers, DANA 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 DANA 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 DANA's features are implemented.

If you happen to be looking at DANA as a framework for processing data not related to Jefferson Lab's Hall-D, then that should be pretty easy to do too. The one thing that would really need to be changed is the *DEventLoop* constructor which calls things like `BCAL_init()`, `CDC_init()`, ... etc. which install the initial set of factories. If you're interested in trying this, contact me (davidl@jlab.org), I'd be eager to help.

Obtaining and Building DANA

The Hall-D reconstruction code is kept in a CVS repository on the Hall-D group disk (`/group/halld/Repositories/cvsroot`) at Jefferson Lab. To access it, you need an account

¹ Standard Template Library

on the JLab CUE which belongs to the halld unix group². 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. In my account, I use */home/davidl/HallD*.
- ❖ **Set `HALLD_HOME` environment variable:** The `HALLD_HOME` variable should be set to the working directory you just made. The makefile system³ uses this to find the source code and place the resulting binaries. So, for my account this is set to */home/davidl/HallD*. For example⁴:

```
setenv HALLD_HOME /home/davidl/HallD
```

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

```
cd $HALLD_HOME
setenv CVS_RSH `which ssh`
setenv CVSROOT cvs.jlab.org:/group/halld/Repositories/cvsroot
cvs co src include
```

Note that this assumes the account you're issuing the command from has the same username as your JLab CUE account. If not, prefix `CVSROOT` with your JLab account name followed by an `@`⁵

- ❖ **Move the *include* directory:** Hopefully this step will be removed in the near future. But for now, you'll need to move the include directory that you just checked out into the `src` directory (that you also just checked out).

```
mv include src
```

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

```
cd src/libraries
gmake
```

This should build all of the libraries and place them in *HALLD_HOME/lib/\$OSNAME* where *\$OSNAME* is the uname of the system you're working on (e.g *Linux*).

- ❖ **Compile the executables:** Go into *src/programs/Analysis* and run `gmake` in the *hd_dump*, *hd_ana*, and *hdview* directories:

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

³ See GlueX-doc-473 on the BMS system

⁴ I use `tcsh` in these instructions. I'll leave it to `bash` users to translate where appropriate.

⁵ e.g. `joe@cvs.jlab.org:/group/halld/Repositories/cvsroot`


```
cd $HALLD_HOME/src/programs/Analysis
gmake -C hd_dump
gmake -C hd_ana
gmake -C hdview
```

The executables will be placed in the *HALLD_HOME/bin/\$OSNAME* directory. These programs do the following:

- hd_dump Dump ASCII output to screen from all factories
- hd_ana Generates a couple of example \$ROOT\$ histograms
- hdview Graphical drawing of data

Quick Start

Data Factories and Data Objects

The DANA framework is built upon the idea of data factories. Figure 1 gives a flowchart that illustrates the factory mechanism. The term factory even 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).

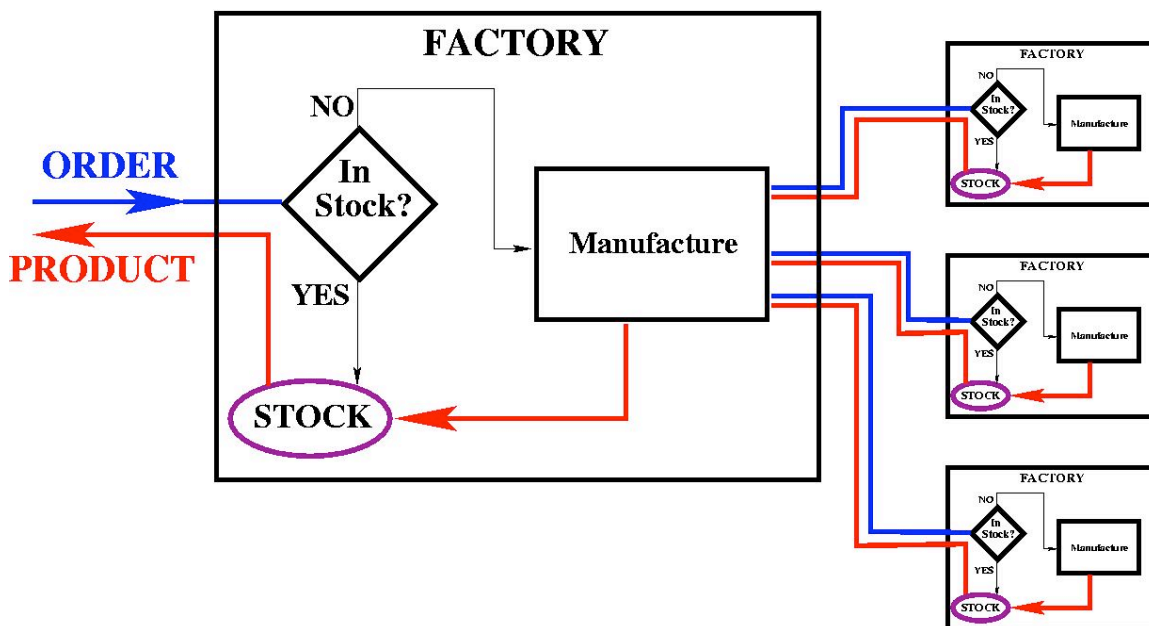


Figure 1: Factory flowchart. Requests for data can be thought of like orders to a factory. The factory must either “manufacture” the data, or retrieve it “from stock”.

Factory Tags

It often occurs that more than one factory wants to output data objects of the same type. For example: We have a PID⁶ 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 (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 DANA 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 and how they are implemented in DANA.

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 DFactory_base class. Without explicitly defining a Tag() method, the method defined in DFactory_base is used which just returns an empty string. Here’s an example of a factory class that has a tag:

```
class DFactory_DMCTrackCandidate_B:public DFactory<DMCTrackCandidate>{
public:
    DFactory_DMCTrackCandidate_B();
    ~DFactory_DMCTrackCandidate_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 code maintenance.

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.

⁶ 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 the 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 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 member of DTRACK_HIT_TAG
    dparms.SetDefaultParameter("TRK:DTRACK_HIT_TAG", DTRACK_HIT_TAG);
    ...
}

// evnt
derror_t DTrackCandidate::evnt(DEventLoop *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 DANA program with a *-PTRK:DTRACK_HIT_TAG=MC* command line option, the MC path would be used.

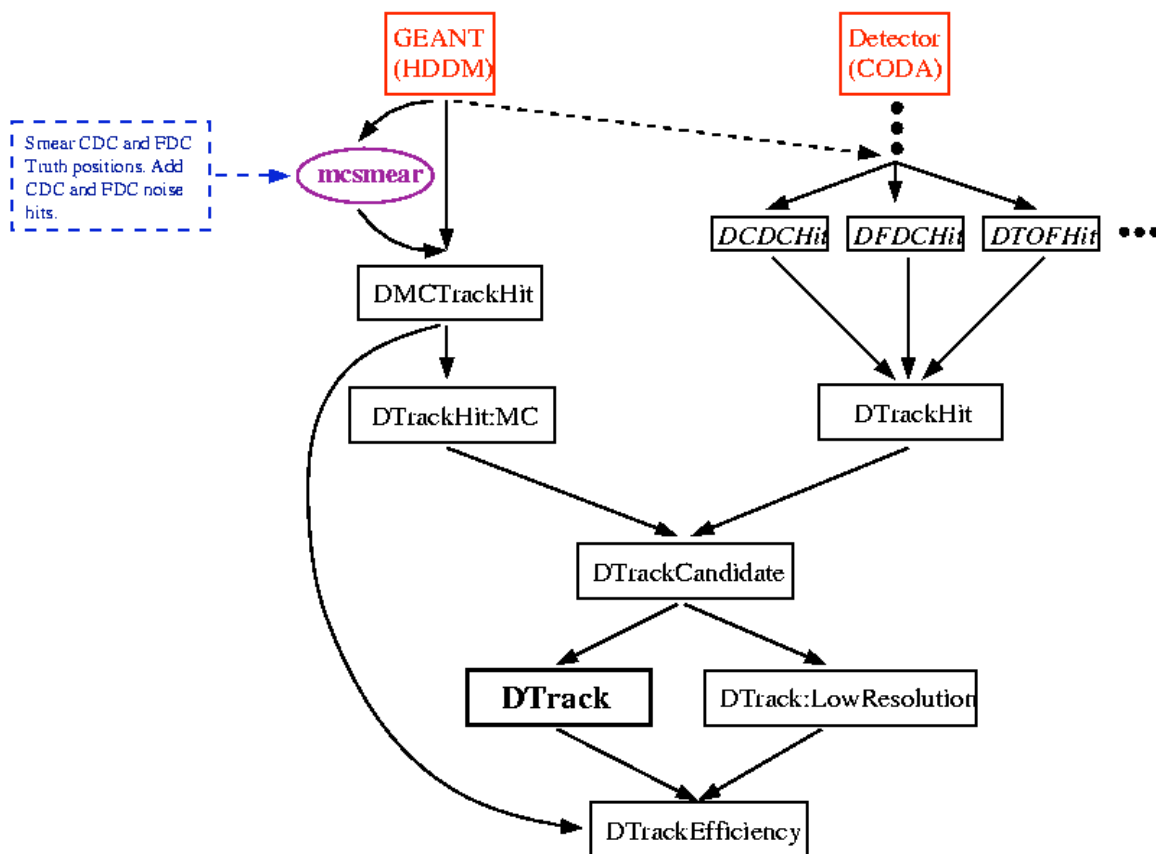


Figure 2: Tracking flow chart. The DTrackCandidate factory can get Monte Carlo data from either the DTrackHit factory with the "MC" tag (left) or the DTrackHit Factory with an empty tag "" (right).

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

How DANA Implements Tags

In the preceding section, the `const char* Tag()` method was described. This method is a virtual method of the `DFactory_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 `DEventLoop.h`. This gets called from the `Get()` method (also in `DEventLoop.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<DFactory_base*>::iterator iter=factories.begin();
DFactory<T> *factory = NULL;
for(; iter!=factories.end(); iter++){
    const char *factory_name = (*iter)->dataClassName();
    if(factory_name == className){
        if(!strcmp((*iter)->Tag(), tag)){
            factory = (DFactory<T>*)*iter;
            break;
        }
    }
}
}

```

The second place the tag can be applied is in a subclass of *DEventSource*. See the chapter on event sources for more details. In a nutshell though, some event sources can supply objects that were created by another DANA 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 *DEventSourceXXX* 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 DANA 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 new needs to either have a tag added or be retired.

Identifiers

Every data object in DANA has a unique identifier attached to it. This is done through its inheritance of the *DObject* 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 *DObject* is initialized by the *DObject* 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 *DObjects* 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 *DObject* would incur an overhead that is considered unacceptable. Using the *DObject* pointer will at least guarantee a unique value for every *DObject* 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 DObject{
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 DANA 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 *DEventLoop* object, but will not have a pointer to the factory that created the desired object readily available. In this case, methods in the *DEventLoop* 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 fastest⁷ way to search for the object pointer since the method can restrict its search to only those factories that provide the specified data type.

⁷ 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 *DFactory_base* object which must be `dynamic_cast<>` if one wishes to use the methods of the subclass as shown in the following example:

```
DFactory_base *fac = loop->FindOwner(id);
DFactory_TrackHit *fac_th = dynamic_cast<DFactory_TrackHit*>(fac);
```

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

```
const DTrackHit* trackhit = loop->FindByID<DTrackHit>(id);
DFactory_base *fac = loop->FindOwner(trackhit);
DFactory_TrackHit *fac_th = dynamic_cast<DFactory_TrackHit*>(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 DObject* obj = loop->FindByID(id);
```

Lower level methods are also provided in the *DFactory* and *DFactory_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 *DFactory_base* class can only return a *DObject* pointer while the (template) class *DFactory* can return a pointer to the subclass:

```
DFactory_base *fac = loop->FindOwner(id);
DFactory_TrackHit *fac_th = dynamic_cast<DFactory_TrackHit*>(fac);

const DObject *obj = fac->GetByID(id);
const DTrackHit *trackhit = fac_th->GetByID(id);
```

Event Processors

The DApplication Class

Every DANA application has a single *DApplication* object. The *DApplication* object directs communication between the *DEventLoop*, *DEventSource*, and *DEventProcessor* objects. *DApplication* is also responsible for creating and monitoring the event processing threads. The constructor for *DApplication* takes the same arguments `main()` does so that

it can be used to parse the command-line arguments in a consistent way for all DANA applications.

Normally, a program will just pass the same arguments passed to `main(int narg, char *argv[])` into the `DApplication` 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 not known to the `DApplication` parser are ignored. In this way, the specific application can implement its own parser in addition to the standard one.

Standard Command Line Options

In order to provide some level of consistency in the command line interface among DANA programs, a command line argument parser has been built into the `DApplication` constructor. This allows features common to all DANA 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 `DApplication` 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 `DApplication`.

Argument	Description
<code>--nthreads=<i>X</i></code>	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 <code>DApplication::Run()</code> .
<code>-Pkey=value</code>	Set a configuration parameter. This will initially add a parameter named <i>key</i> with value <i>value</i> to the internal database DANA creates whenever a program is run. It will override the compiled in value. This option may be used multiple times.
<code>-Pprint</code>	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.
<code>--so=<i>file.so</i></code>	Attach the shared object file <i>file.so</i> and look for event sources, and factory initialization

	routines within. The file <i>file.so</i> must be given with either the full path, or the path relative to the current working directory.
<code>--sodir=directory</code>	Look through all files in <i>directory</i> with names that follow the <i>lib*.so</i> naming convention and attach each of them as though they were passed with individual <code>--so=</code> arguments.

Using DApplication in Single Event Mode

There is a class of programs that do not fit well in the run-with-callback paradigm DANA 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 DANA and the GUI API want to implement the main event loop. Since there can be only one⁸ main event loop, DANA has to be used in Single Event Mode so that events are processed upon request from the GUI as opposed to automatically by the DEventLoop object.

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 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: In the tracking code, there is a value representing the maximum number of hits a track seed can have. The

⁸ Conner McCloud of the clan McCLoud.

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 `DParameterManager`. 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 `DParameterManager` object named `dparms`. The `dparms` object is instantiated globally (i.e. with out new and before `main()`) so it is accessible by all code in the program. There is an extern declaration of `dparms` in `DParameterManager.h` which is included by `DEventProcessor.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 `dparms`. Generally, you will initialize a data member of a factory to the default value and then call `dparms.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 `dparms`. If the given key does not yet exist, it will be added to `dparms` 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.

```

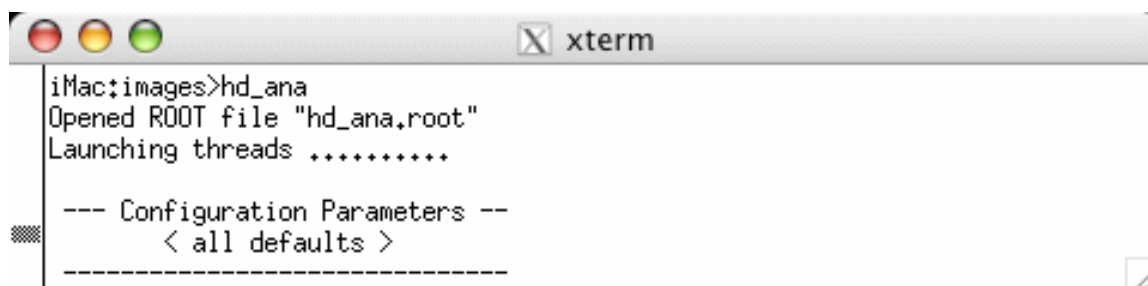
//-----
DFactory_DTrackCandidate::DFactory_DTrackCandidate()
{
    // Set defaults
    MAX_SEED_DIST = 5.0;
    MAX_SEED_HITS = 10;
    MAX_CIRCLE_DIST = 2.0;
    MAX_PHI_Z_DIST = 10.0;
    MAX_DEBUG_BUFFERS = 0;
    TARGET_Z_MIN = 50.0;
    TARGET_Z_MAX = 80.0;
    TRACKHIT_SOURCE = "MC";

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

    MAX_SEED_DIST2 = MAX_SEED_DIST*MAX_SEED_DIST;
}

```

DANA 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.



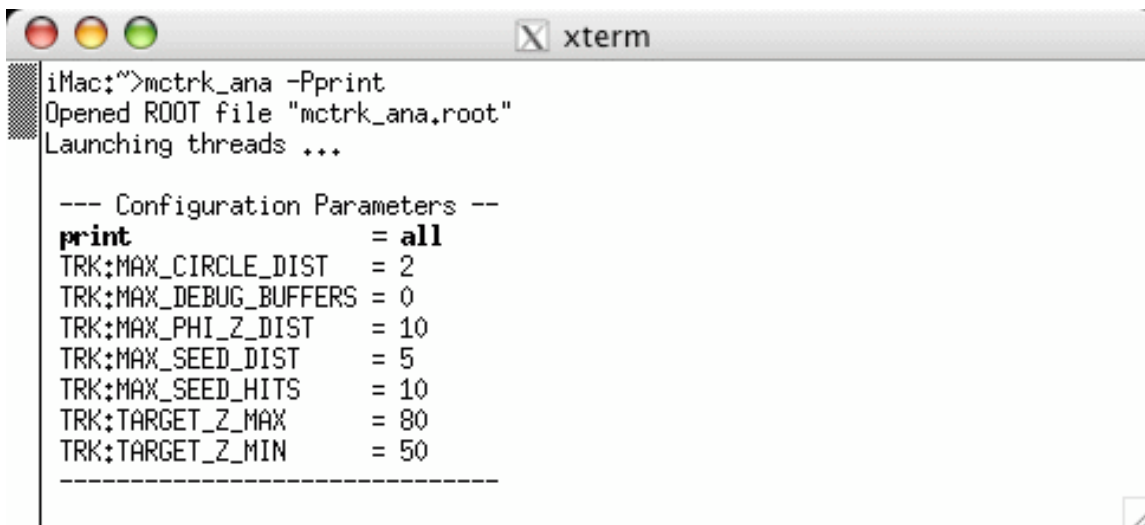
```

iMac:images>hd_ana
Opened ROOT file "hd_ana.root"
Launching threads .....
--- Configuration Parameters ---
< all defaults >

```

Figure 3: Running a DANA 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

Accessing the Calibration Database

Interfacing with ROOT

The DANA framework is NOT integrated with ROOT⁹. This was done to provide some degree of flexibility for those wanting to use the DANA framework, but not ROOT. It is recognized, however, that a portion of the user base will want to integrate ROOT and DANA in the same application. This chapter provides some hints on how this may be done.

Event Sources

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

The HDDM Event Source

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 *DApplication* in the source code
- Pass a `--nthreads=N` option on the command line when running the program.

All DANA 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 DANA application that uses 4 threads by default.

```

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

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

    // Run though all events, calling our event processor's methods
    app.Run(&myproc, 4); // Tell DANA 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 data analysis is simple since it lends itself so naturally to it. The fact that events are independent leads to having a single event processed in a single thread. However, both the input and output event sources 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 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.

DANA 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>

// evtnt
derror_t MyProcessor::evnt(DEventLoop *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 which produces ROOT output. For other packages, something similar must be done when running with multiple threads.

How DANA 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 "reentrant" 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 DEventSource 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 DEventSource¹⁰ object, which it does through the DApplication object. DApplication keeps track of this so other threads can use the same DEventSource to obtain events. The DApplication, DEventSource, and DEventProcessor¹¹ objects are ones used by all threads. All threads must read from a single source(DEventSource), output the result to a single output(DEventProcessor) and the information about these is kept in a centrally accessible area (DApplication).

¹⁰ DEventSource is the base class for all event sources. In reality, this will always be a subclass such as DEventSourceHDDM. The base class name DEventSource is used for simplicity here.

¹¹ Similarly to the previous footnote, this will be a subclass of DEventProcessor

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 DEventLoop object. The DEventLoop 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!

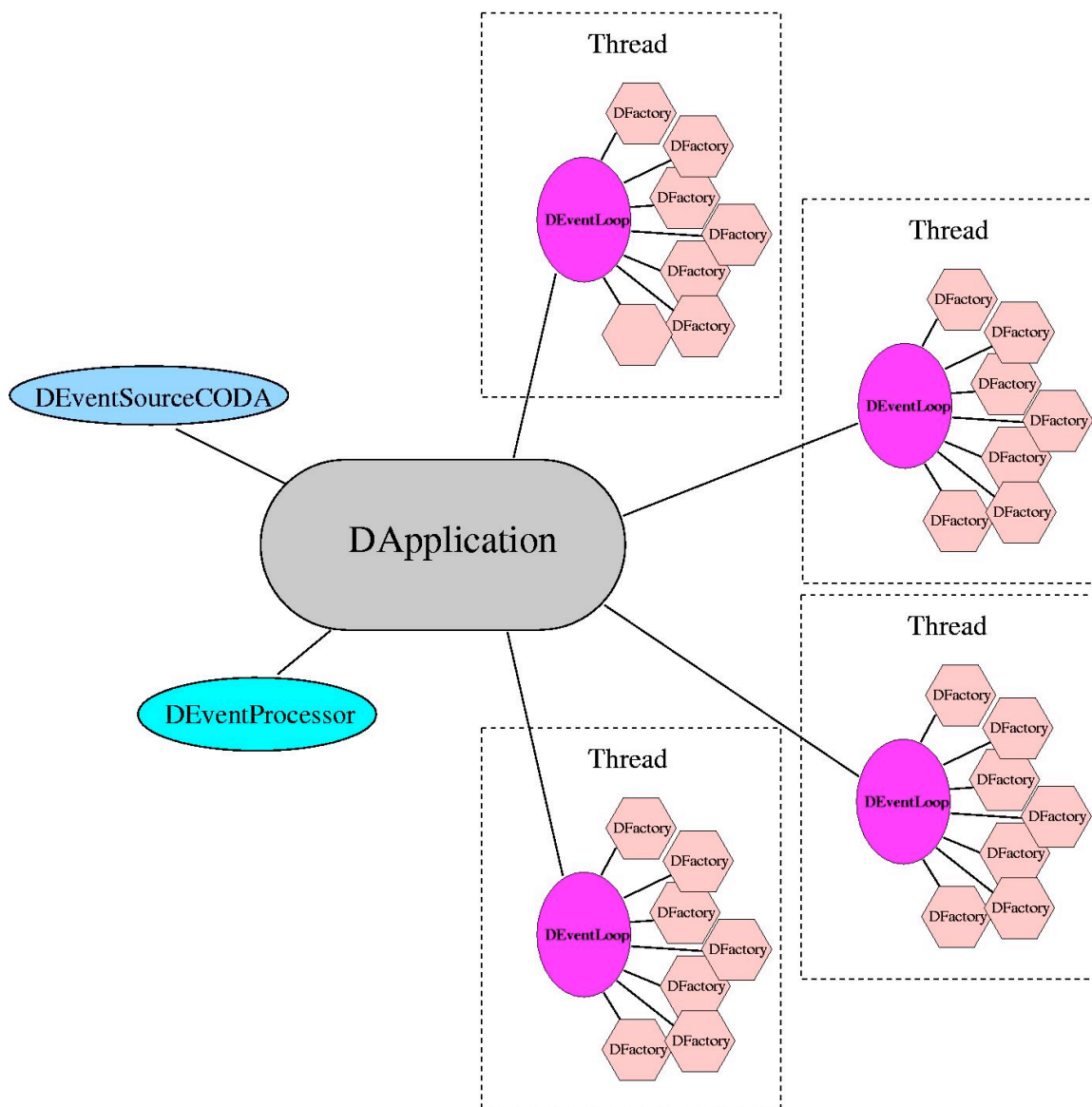


Figure 5: Conceptual view of how DANA objects are related when running multiple threads. The DEventLoop and various DFactory objects are all

specific to a thread while the DApplication, DEventSourceCODA, and DEventProcessor objects are used by all threads.

Threads are launched by the `DApplication::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 `DApplication.cc` near the `Run()` method however since that is the only place it is used. The `LaunchThread()` routine simply creates a `DEventLoop` object, calls its `loop()` method then deletes the object after it returns from `loop()`.

All DANA objects 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, it spends most of its time in `nanosleep()` and wakes up once every 0.5 seconds to monitor progress. Most DANA 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 `DEventLoop` object is created, it is passed a pointer to the `DApplication` object. This is used by the `DEventLoop` object to register (and eventually deregister) itself with `DApplication`. One potential fallback to this method is that if a `DEventLoop` gets stuck somehow in an infinite loop, it will never deregister itself causing the main thread to also be stuck. This is a current limitation that may need to be addressed in the future.

Shared Binary Objects

DANA can use event sources, factories, and event processors obtained from dynamically linked binary object files. The seemingly standard `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.

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

DEventSource in a shared binary object

To include an object of a class derived from *DEventSource* in a shared object, the *GetDEventSourceType* and *MakeDEventSource* symbols must be defined. The following example illustrates how to do this:

```
extern "C" {
extern GetDEventSourceType_t GetDEventSourceType;
extern MakeDEventSource_t MakeDEventSource;
}

const char* GetDEventSourceType(void){
    return "DEventSourceHDDM";
}

DEventSource* MakeDEventSource(const char* name)
{
    return new DEventSourceHDDM(name);
}
```

The *GetDEventSourceType_t* and *MakeDeventSource_t* types are defined in *DApplication.h*.

Note that for event sources, simply adding the type to the list is not sufficient to ensure that it is used. For each source that the program tries to open, it calls the static *GuessSourceType* method of *DEventSource* to determine the subclass of *DEventSource* that should be instantiated to handle it.

DFactory in a shared binary object

To add *DFactory* objects to a process from a shared binary object, the *InitFactories* symbol must be defined.

```
extern "C" {
extern InitFactories_t InitFactories;
}

void InitFactories(DEventLoop* eventLoop)
{
    cout<<"Adding factories from shared object ..."<<endl;
    eventLoop->AddFactory(new DFactory_DMCTrackHit());

    // Add more factories here id needed!
}
```

The *InitFactories* routine will be called for every *DEventLoop* object created, or in other words, for every thread. Because the factories added by the *InitFactories* routine are added to the front of the list, they will be found first and therefore used in preference to factories statically linked into the executable. Thus, this mechanism can be used to effectively *replace* a factory in an existing executable.

DEventProcessor in a shared binary object

To add *DEventProcessor* objects to a process from a shared binary object, the *InitProcessors* symbol must be defined.

```
extern "C" {
extern InitProcessors _t InitProcessors;
}

void InitProcessors (DApplication * app)
{
    cout<<"Adding processors from shared object ..."<<endl;
    app->AddProcessor(new DEventProcessor_bcal_hists());
}
```

Debugging Your Code

Index

C

configuration parameter, 11

D

DANA, 1, 7, 10, 11, 12, 13

DEventLoop, 7, 11, 12

DFactory_base, 10, 12, 13

DTrackCandidate, 11, 12

DTrackHit, 11, 12

F

Factory, 10, 11, 12

G

GetFromFactory, 12, 13

GetFromSource, 13

GetObjects, 13

I

Identifiers, 13

M

mkfactory, 10

O

object-seekable, 13

S

STL, 7

T

Tag, 10, 12, 13

Tags, 10, 11, 12