



D0 Framework Software Tutorial

Reiner Hauser
Michigan State University
Feb 11, 2003



This tutorial covers...

- ...how to setup and initialize your work area
- ...how to create and checkout packages
- ...how to compile and link libraries and binaries
- ...how to write a D0 framework package for analysis
- ...how to access physics objects in your analysis
- ...how to apply corrections to physics objects
- ...how to calculate luminosities



This tutorial doesn't cover...

- ...an introduction into object-oriented programming or C++.
- ...which triggers you should use for your analysis
- ...what are the correct cuts to apply to your physics objects

It is technical in nature, assuming that you know some C++, explaining you *how* to do certain things inside the D0 software framework, but not *what* (that's left to you).

Don't run the examples blindly and expect them to do something useful for you...



The D0 Software Environment

- The D0 software consists of many components:
 - External software (root, OpenInventor, python,gcc....)
 - D0 specific software written by us.
- All these pieces are packaged up in so-called *products* of the UPD/UPS system.
 - Some products are supported Fermilab-wide (e.g. ROOT)
 - Others are specific to D0
- UPS supports multiple sources for products and allows to install multiple versions of a product at the same time.



UPS and setup

- The `setup` command can be used to make a certain product available to your session:
 - **`setup python`**
 - uses the version declared *current* in UPS configuration.
 - **`setup python v2_2_1`**
 - uses the specific version you require
 - **`setup -t d0tools`**
 - uses the version declared the *test* version in UPS
- The typical effect of a `setup` command is a change of your `PATH` and other environment variables. (type 'env' before and after a setup).
- UPS products can depend on each other.



Setup a D0 Release

- A D0 release is a UPS product depending on many other UPS products.
- Version numbers are like this:
 - `setup D0RunII p13.08.00`
 - `setup D0RunII t03.03.00`

This is a *production* release.
13 is the major, 08 the minorversion of the release.
Major release are done every three to four months.

This is a *test* release. The minor version changes every week.

Avoid **setup D0RunII current**
It will set your work environment to the release du jour, and you will get surprises the next week if you don't know what you are doing.



Your work area

- The area where you are doing your work is expected to have a certain structure.
- You can have as many work areas as you like, but use only one at a time.
- Here is how you create a new one:

Use your own machine name and user id here !!!

```
% cd /work/wensley-clued0/rhauser
% setup D0RunII p13.08.00
% newrel -t p13.08.00 work
% cd work
% d0setwa
```

Use the same version number for **setup** and **newrel**

The work directory now has a couple of subdirectories and files. The hidden file `.base_release` contains the version number.



Using an existing work area

If you have an existing work area, just do the **setup** and **d0setwa** there:

```
% cd /work/wensley-clued0/rhauser/work  
% setup D0RunII p13.08.00  
% d0setwa
```

If you are switching to a different release version, either create a new working area and copy things over, or change the `.base_release` file and:

```
% rm -rf bin/* lib/* tmp/* rcpdb/*  
% setup D0RunII t03.06.00  
% d0setwa
```

If in doubt, go to your work area and **d0setwa** again. Usually it won't hurt and often you have just forgotten it.



CVS Packages

- The DO software is structured into many packages that are under version control via CVS.
- To check out an existing package in your work area:
 - **addpkg -h analysis_tutorial**
 - checks out the head version of the package (may be untagged and may not work)
 - **addpkg em_evt**
 - checks out the version that corresponds to the release you have setup
 - **addpkg em_evt v00-15-26**
 - checks out a specific version of the package



Package Versions

- CVS keeps a revision version for every file; every change you commit is recorded.
- Package versions are a symbolic name that applies to the whole package, not individual files inside the package.
- You can *tag* a package (if you have the permissions):

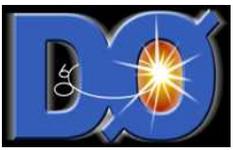
No, you can't tag this specific package if you're not that the author/responsible

```
% cvs rtag analysis_tutorial v01-02-10
```

These tags are what you have to submit to the release managers to have your package included in the release.

For a list of package versions for your release, look at:

```
$SRT_PUBLIC_CONTEXT/D0reldb/inventory
```



Release area and work area

- The release area is shared by all users on a machine and is readonly .
- After a setup you can find it when looking at
 - `% ls $SRT_PUBLIC_CONTEXT`
- Your work area is kept in
 - `% ls $SRT_PRIVATE_CONTEXT`
- The release area and your work area have a similar structure.
- Whenever the build or run-time system looks for something, it first checks your local work area, then the release area !



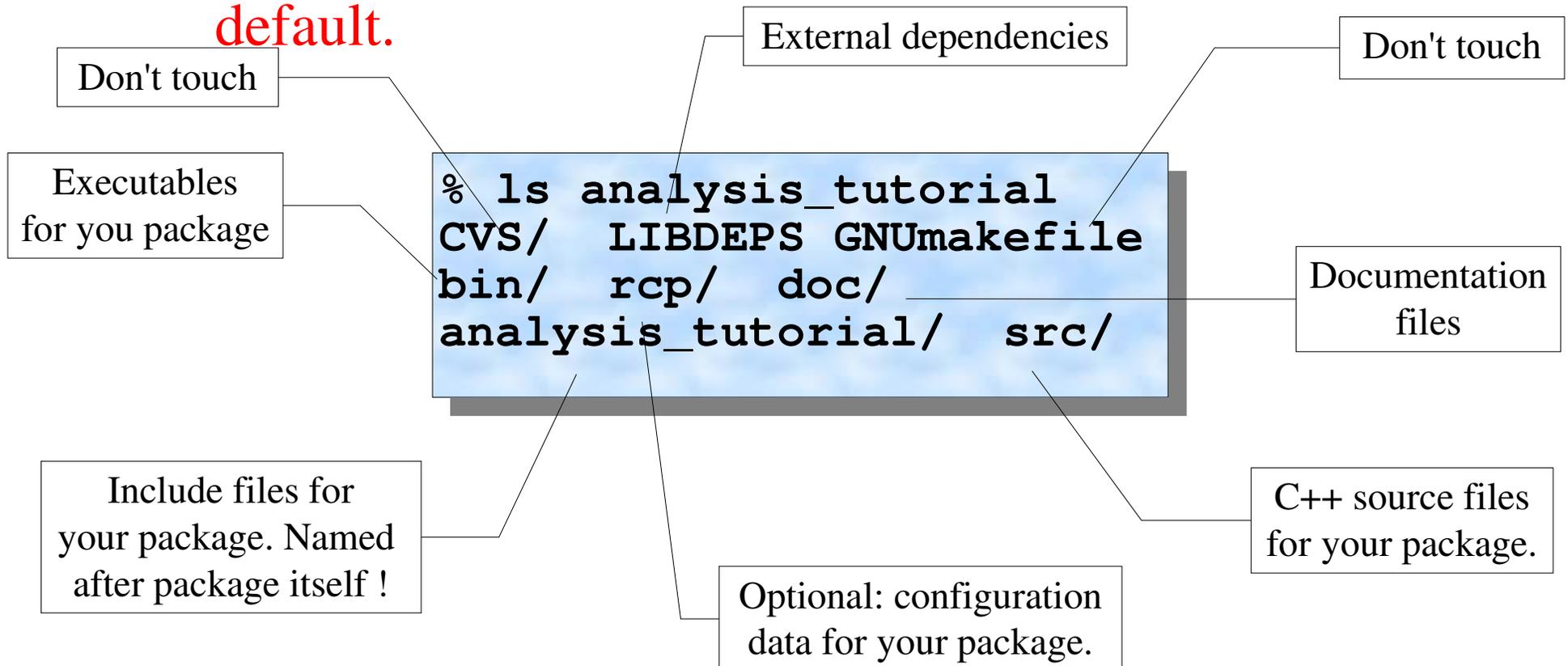
Looking up things...

- C++ include files are searched in
 - `$SRT_PRIVATE_CONTEXT/include`
 - then `$SRT_PUBLIC_CONTEXT/include`
- Libraries are searched in
 - `$SRT_PRIVATE_CONTEXT/lib/...`
 - then `$SRT_PUBLIC_CONTEXT/lib/...`
- Binaries are searched in
 - `$SRT_PRIVATE_CONTEXT/bin/...`
 - then `$SRT_PUBLIC_CONTEXT/bin/...`
- RCP files are searched in
 - `$SRT_PRIVATE_CONTEXT/<package>/rcp`
 - then `$SRT_PUBLIC_CONTEXT/<package>/rcp`



Package structure

- Every CVS package appears as a subdirectory in your work area.
- The structure of a package is standardized
 - you can change it but it is easiest to stick with the default.





More on the package structure

The **src** directory has a number of magic files that tell the build system what to do: you don't have to write any Makefiles yourself.

Most of
your Code

A list of all your .cpp files except component tests and those in OBJECT_COMPONENTS. The entries are without the .cpp extensions !

```
% ls analysis_tutorial/src
COMPONENTS                OBJECT_COMPONENTS
AccessChunks.cpp          AccessChunks_t.cpp
AccessTrigger.cpp         AccessTrigger_t.cpp
Minimal.cpp               Minimal_t.cpp
RegMinimal.cpp
[...more of the same...]
```

A special C++ file only needed for D0 framework programs. This goes into OBJECT_COMPONENTS

Component tests, one for each of your source files.



Ready to go...

- From your work area, type
 - **% make all**
 - to compile and link all packages in your area in the correct order.
 - **% make clean**
 - to delete and clean up any of the produces object files, libraries and executables
- These are the only commands that are always safe.
- If you have modified only one package and you're sure none of the others needs recompiling, you can do
 - **% make analysis_tutorial.all**



Optimized Builds

- For production code you want to build your executable with compiler optimizations switched on.
- Setup a release as usual
 - **srt_setup SRT_QUAL=maxopt**
 - executables now go into
 - `bin/Linux2.4-GCC_3_1-maxopt`

This is missing after a normal setup



More on LIBDEPS

- Originally you had to specify all direct dependencies on other packages here.
- This is no longer necessary, the build system can figure this out for itself.
- The exception are libraries which are not part of the D0 software, but external UPS packages. To find the correct header files, you have to put the package name into LIBDEPS !
 - e.g. add a line with 'root' if you want to include headers files from ROOT
 - It's not always obvious to a beginner what an external package is...



The bin directory

A list of binaries to build, without the .cpp extension. Here just 'tutorial_x'.

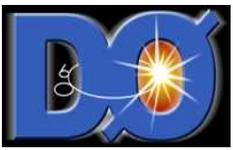
Libraries to link the executable with. Usually just the name of your own package.

```
% ls analysis_tutorial/bin
CVS/      BINARIES  LIBRARIES
OBJECTS  tutorial_x.cpp
```

A list of D0 framework packages to include in the binary.
An example would be the RegMinimal from the src directory.

Your main() program if you're building a simple C++ program. Empty when you're building a D0 framework program (really...)

This will build an executable **tutorial_x** and leave it in the bin directory of your work area.
You can start it by simply typing **tutorial_x ...arguments**



Some actual source code

```
#ifndef MINIMAL_HPP_  
#define MINIMAL_HPP_  
  
#include <string>  
  
#include "framework/Package.hpp"  
#include "framework/interfaces/Flow.hpp"  
#include "framework/interfaces/StandAlone.hpp"  
  
namespace edm {  
    class Event;  
}  
  
namespace d0tutorial {  
  
    class Minimal : public fwk::Package,  
                  public fwk::Analyze,  
                  public fwk::JobSummary  
    {  
        [...actual class declaration...]  
    };  
}  
  
#endif // MINIMAL_HPP_
```

Usual header protection

Include all standard headers that you make use of in your header itself.

This is how to access other D0 packages, here one called 'framework'

But don't include classes whose full declaration you don't need here. Just forward declare them.

Choose your own namespace to separate your code from others and avoid naming collisions.

analysis_tutorial/Minimal.hpp



D0 framework packages

- So far we had UPS products and CVS packages. Now we introduce a third type of package to maximize confusion:
 - *A D0 framework package is a C++ class that follows certain rules and implements certain interfaces.*
 - A single CVS package can contain multiple D0 framework packages.
 - *The D0 framework contains the actual main() program and calls your C++ objects according to rules specified in configuration files.*
 - Your D0 framework package is typically one of many called in the processing of one event.
 - *By combining existing and new packages you make use of other peoples code and add your own contributions.*
 - Our example analysis code will be in the form of a D0 framework package.



Back to our header file...

[...]

```
namespace d0tutorial {  
  
  class Minimal : public fwk::Package,  
                 public fwk::Analyze,  
                 public fwk::JobSummary  
  {  
  public:  
    Minimal(fwk::Context *ctx);  
    virtual ~Minimal();  
  
    virtual fwk::Result analyzeEvent(const edm::Event& event);  
  
    virtual fwk::Result jobSummary();  
  
    [...stuff omitted here...]  
  
};  
}
```

A class becomes a framework package by inheriting from certain base classes. The most basic one is Package which you always need. The others each define a virtual routine that you have to implement.

Your constructor must look like this.

This virtual function comes from fwk::Analyze. It is an example of an interface that takes an Event as argument.

analysis_tutorial/Minimal.hpp

This virtual function comes from fwk::JobSummary. It takes no arguments.



More things to do....

[...]

```
namespace d0tutorial {  
  
    class Minimal : public fwk::Package,  
                   public fwk::Analyze,  
                   public fwk::JobSummary  
    {  
    public:  
  
        [...stuff omitted here...]  
  
        virtual std::string packageName() const { return package_name(); }  
        static const std::string package_name() { return "Minimal"; }  
        static const std::string version() { return "$Id$"; }  
    private:  
        bool          _debug;  
        unsigned int  _events_processed;  
    };  
}
```

You must have these 3 lines in each of your classes. The package_name() must correspond to your class name.

CVS will replace this with something else the first time you checked your code in.

analysis_tutorial/Minimal.hpp

This is the first of our own code: private member variables that we are going to use. Note that they are prefixed by an underscore.



Now to the implementation...

```
#include "framework/Registry.hpp"
#include "edm/Event.hpp"
#include "analysis_tutorial/Minimal.hpp"

namespace d0tutorial {

    FWK_REGISTRY_IMPL(Minimal, "$Name$");

    Minimal::Minimal(fwk::Context *ctx)
        : fwk::Package(ctx),
          fwk::Analyze(ctx),
          fwk::JobSummary(ctx),

          _debug(packageRCP().getBool("Debug")),
          _events_processed(0)
    {
        log()(ELInfo, "Minimal") << "Instance = "
            << instanceName()
            << " created"
            << endmsg;
    }

    Minimal::~Minimal()
    {
    }

    [...]
}
```

src/Minimal.c

We need this for a macro to register our package.

We now need the full declaration of everything that we declared forward in the header.

This tells the framework that the implementation of Minimal is here.

We don't care what Context is, just pass it to all our superclasses.

This initializes our member variables.

This issues a log message with severity level 'info'. We never use std::cout or std::cerr directly !



Our interfaces...

```
namespace d0tutorial {  
[...]  
fwk::Result Minimal::analyzeEvent(const edm::Event& event)  
{  
    if(_debug) {  
        out() << instanceName()  
            << " : AnalyzeEvent called"  
            << std::endl  
  
        << instanceName()  
        << " : RunNumber = "  
        << (long )event.collisionID().runNumber()  
        << std::endl  
  
        << instanceName()  
        << " : EventNumber = "  
        << (long )event.collisionID().eventNumber()  
        << std::endl;  
    }  
    ++_events_processed;  
    return fwk::Result::success;  
}  
[...]  
}
```

This function is called for every event.

Notice that the Event is const: you are not supposed to modify it.

Another way to produce output without using std::cout

Our first access to an event !!!

Except when filtering (coming later), always return success here...

src/Minimal.cpp



The final touch...

```
namespace d0tutorial {  
[...]  
    fwk::Result Minimal::jobSummary()  
    {  
        out() << packageName()  
              << '/' << instanceName()  
              << " : processed "  
              << _events_processed  
              << " events" << std::endl;  
        return fwk::Result::success;  
    }  
[...]  
}
```

src/Minimal.cp

This function is called once per job. It takes no parameters.

You typically use it to print summary information.

```
#include "framework/Registry.hpp"  
namespace d0tutorial {  
    using namespace fwk;  
    FWK_REGISTRY_DECL(Minimal)  
}
```

src/RegMinimal.cpp

Just follow this magic recipe for every framework package you write.

We're done with our first framework program !



Interfaces again...

- There are many interfaces defined for various tasks, some with, some without parameters.
 - Just follow the same approach we have used here
 - We will make use of only a few; we will ignore all interfaces which typically modify an event (that's what you use in d0reco).
 - Only **filterEvent (const edm::Event&)** is special in that returning **failure** instead of **success** will not terminate the program, but just end the processing of this event.



Configuration with RCP

- RCP files are text files where parameters are stored.
- RCP has an underlying database (also text based at the moment).
- Any program can make use of the RCP database and extract parameters out of it.
- Framework packages need some special entries in the RCP files which tell it how to put the pieces together.



Simple RCP example

```
# this is a comment
// this is a comment, too

string PackageName = "Minimal"

bool Debug = true

float anArray = ( 1.0, 2.0, 5.6 )
```

This tells the framework to construct an object of type *Minimal*.

The other entries are the parameters for our package.

rcp/minimal.rcp

```
Minimal::Minimal (fwk::Context *ctx)
    : fwk::Package (ctx),
      fwk::Analyze (ctx),
      fwk::JobSummary (ctx),

      _debug (packageRCP () .getBool ("Debug")),
      _events_processed (0)
{
    [...]
}
```

Inside our package we have access to a method packageRCP() that returns an object which we can use to access the parameters.

The RCP object has methods for every datatype that can be used in the RCP file.

src/Minimal.cpp



The framework RCP

```
string InterfaceName = "process"  
  
string Packages = "read minimal"  
  
RCP read      = <io_packages      ReadEvent>  
RCP minimal   = <analysis_tutorial minimal>
```

Required for framework RCP

The list of framework packages to run.

This is an existing package to read events.

For each package, the RCP file that contains the package parameters.

`rcp/runMinimal.rcp`

This will be the `instanceName()` that we print out in our code

We are ready to run !

Our executable name

Name of the framework RCP file, required

```
% tutorial_x -rcp analysis_tutorial/rcp/runMinimal.rcp \  
-input_file recoT_all_0000163972_mrg_200-204.raw_p13.06.01
```

A thumbnail file



Some notes...

- Our program will run on reco output, DST and thumbnail files.
- This is because we don't access any of the information inside the event, only the event number and run number...
- For thumbnail files we will need additional packages after the reading to have the data in the format we want it.
- To understand this we have to know how Events are stored in memory.



D0OM

- d0om is D0's persistency mechanism.
- With a few lines of code you can make any C++ class persistent, i.e. you can store objects of the class in a file and retrieve them later.
- All raw, reco, DST and thumbnail files are d0om files.
- E.g. the ReadEvent package that we used in our first example can read d0om files and insert them into the data flow of the framework so that they can be processed by the other packages.



Event Data Model (EDM)

- The EDM defines the representation of events in memory.
 - What is contained in an event
 - How to add new data to an event (e.g. during reconstruction)
 - How to access specific data in an event
 - It keeps track of dependencies between parts of the event (e.g. which other parts were used to create the tracks in this one)
 - It keeps track of which RCP parameters were used in the creation of a certain part of an event.



EDM (2)

- The smallest unit of an Event that the EDM knows of is called a '*Chunk*'.
- Different Chunks contain different types of data (e.g. a *JetChunk* contains jets, a *MuonParticleChunk* muons...)
- An event can contain more than one chunk of the same type at the same time (e.g. each jet reco algorithm produces a *JetChunk*, but with different parameters)
- Chunks are the units that can be inserted, deleted etc. in an event.



Using the EDM

```
#include "framework/Registry.hpp"
#include "edm/Event.hpp"
#include "analysis_tutorial/AccessChunks.hpp"
[...]
```

void AccessChunks::access_muon(const edm::Event& event)

```
{
    using namespace edm;
    using namespace muonid;

    typedef std::vector<MuonParticle>::const_iterator const_iterator;

    TKey<MuonParticleChunk> key;
    THandle<MuonParticleChunk> handle = key.find(event);

    if(handle.isValid()) {
        [...do something with handle...]
    }
}
```

Include the header files for the physics object type and chunks you're interested in

Most classes live in a separate namespace.

Define a key for the Chunk

Use the key to find the chunk.

See if we were successful. If yes use the chunk...

src/AccessChunks.cpp



Accessing Muons...

```
void AccessChunks::access_muon(const edm::Event& event)
{
  [...]

  if(handle.isValid()) {
    const std::vector<MuonParticle> *particles = handle->getParticles();
    for(const_iterator it = particles->begin();
        it != particles->end();
        ++it) {

      const MuonParticle& obj = *it;

      float pT = obj.pT();

      const MuonQualityInfo *quality = obj.qualInfo();

      int nhit = quality->nhit();
      int nseg = quality->nseg();
    } // for
  } // if (handle.isValid())
}
```

Get the list of muon objects.

Iterate over muons

Use a shortcut to access the muon

src/AccessChunks.cpp

Work with the Muon qualities
some are in a separate object
called MuonQualityInfo



Accessing tracks

```
void AccessChunks::access_tracks(const edm::Event& event)
{
    using namespace edm;
    using namespace vertex;
    typedef std::vector<ChargedParticle>::const_iterator const_iterator;

    TKey<ChargedParticleChunk> key;
    THandle<ChargedParticleChunk> handle = key.find(event);

    if(handle.isValid()) {
        const std::vector<ChargedParticle> *particles = handle->getParticles();

        for(const_iterator it = particles->begin();
            it != particles->end();
            ++it) {
            const ChargedParticle& obj = *it;

            float pT = obj.pT();
            [...]
        }
    }
}
```

src/AccessChunks.cpp

Follow the same structure for every chunk where you expect only one chunk of a given type per event....



Missing ET

```
void AccessChunks::access_met(const edm::Event& event)
{
    using namespace edm;
    // there is no namespace missingET...

    TKey<MissingETChunk> key;
    THandle<MissingETChunk> handle = key.find(event);
    if(handle.isValid()) {

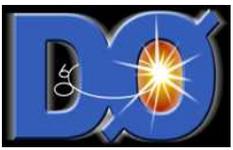
        // Note: no 'const' qualifier here...
        MissingET *met = handle->getMissingET();

        float MET      = met->getMET();
        float scalarET = met->getScalarET();

        _met->Fill(MET); // fill histogram
    }
}
```

src/AccessChunks.cpp

There are no particles here, but only a single object. It should be const, but isn't, so in principle you can change it... Don't do it, this is a design error in the MissingET class.



Accessing EM particles

```
[...]  
#include "edm/TKey.hpp"  
  
namespace edm {  
    class Event;  
}  
  
namespace emid {  
    class EmparticleChunk;  
}  
[...]  
namespace d0tutorial {  
  
    class AccessChunks : public fwk::Package,  
                        public fwk::Analyze,  
                        public fwk::JobSummary  
    {  
    private:  
        edm::TKey<emid::EMparticleChunk> _emKey;  
        [...]  
    };  
}
```

We now need the TKey header file here.

We still do not need the chunk definition !

A key that we will initialize and reuse.

analysis_tutorial/AccessChunks.hpp



Initializing the TKey from RCP

```
#include "em_evt/EMparticleChunkSelector.hpp"
[...]
```

```
AccessChunks::AccessChunks(fwk::Context *ctx)
[...]
```

```
{
    using namespace edm;
    using namespace std;
    using namespace emid;

    RCP rcp = packageRCP();

    // Initialize key for selecting the correct EM chunk
    RCP emidAlgoRCP = rcp.getRCP("EMid_Algo");
    vector<string> emidNestedRCP = rcp.getVString("EMid_SearchRCPs");

    EMparticleChunkSelector emAlgoSel(emidAlgoRCP, emidNestedRCP);
    _emKey = edm::TKey<emid::EMparticleChunk>(emAlgoSel);

    [...]
}
```

This retrieves the algorithm parameter from the RCP file.

This initializes the TKey with a EM specific Selector object.

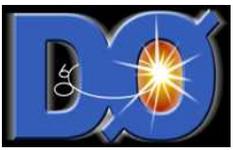
src/AccessChunks.cpp

```
string PackageName = "AccessChunks"
[...]
```

```
RCP EMid_Algo = < emreco EMReco-scone-id >
string EMid_SearchRCPs = ("clusterer", "HMReco",)
```

This selects the simple cone algorithm

rnp/chunks.cpp



The rest is business as usual...

```
void AccessChunks::access_em(const edm::Event& event)
{
    using namespace edm;
    using namespace emid;

    typedef std::vector<EMparticle>::const_iterator const_iterator;
    THandle<EMparticleChunk> handle = _emKey.find(event);
    if(handle.isValid()) {
        const std::vector<EMparticle> *particles = handle->getParticles();
        for(const_iterator it = particles->begin(); it != particles->end(); ++it) {
            const EMparticle& obj = *it;
            int id = abs(obj.typeID());

            if(id == 10 || id == 11) {
                float pT = obj.pT();
                float eta = obj.eta();
                float hm8 = obj.HMx8();
                float isolation = obj.isolation();
                float em_fraction = obj.emfrac();
                if(obj.has_track_match()) {
                    float chi2 = obj.track_match_chi2prob();
                    // get best track
                    const vertex::ChargedParticle *track = obj.getPtrChp();
                }
            }
        }
    }
}
```

Use the initialized member variable here

Restrict yourself to the methods
marked as user interface...
This is new after p13.xx.xx !

src/AccessChunks.cpp



Similar for JetChunk

```
AccessChunk::AccessChunk(fwk::Context *ctx)
[... ]
{
[... ]
    string jetType          = rcp.getString("JetAlgo_type");
    vector<float> jetValues = rcp.getVFloat("JetAlgo_values");
    vector<string> jetNames = rcp.getVString("JetAlgo_names");

    if ( jetValues.size() != jetNames.size() ) {
        log()(EError, "JetAlgoSelection")
            << "Inconsistent RCP values for the jet selection, disable it"
            << endl;
        jetNames.clear();
        jetValues.clear();
    }

    jetid::JetAlgoInfo jetAlgoInfo(jetType, jetValues, jetNames);
    _jetKey = Tkey<JetChunk>(JetChunkSelector(jetAlgoInfo));
}
```

src/AccessChunks.cpp

```
string PackageName = "AccessChunks"
[... ]
string JetAlgo_type = "PreSCilcone"
string JetAlgo_names = ( "towers" "coneSize" "Radius_of_Cone" "Min_Jet_ET" )
float JetAlgo_values = ( 0. 0.3 0.5 8. )
```

rcp/chunks.cpp

Jet chunks are described by a set of parameter names and values plus the algorithm type



Rest as usual...

```
void AccessChunks::access_jet(const edm::Event& event)
{
    using namespace edm;
    using namespace jetid;

    typedef std::vector<Jet>::const_iterator const_iterator;

    THandle<JetChunk> handle = _jetKey.find(event);

    if(handle.isValid()) {
        const std::vector<Jet> *particles = handle->getParticles();

        for(const_iterator it = particles->begin();
            it != particles->end(); ++it) {
            const Jet& obj = *it;

            float pT           = obj.pT();
            float emfraction    = obj.emETFraction();
            float hotcellratio = obj.hotcellratio();
            int    n90          = obj.n90();

        }
    }
}
```

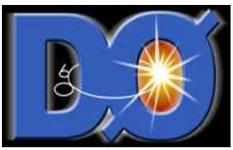
Use the
predefined key

src/AccessChunks.cpp



Relationships between Chunks

- A chunk can contain objects that in turn can contain pointers to other objects etc.
- An object in one chunk may not contain a pointer to an object in another chunk. Otherwise the two would be strongly tight together: you couldn't read/write them independently !
- If you want to reference an object in another chunk you have to go through an intermediate step: chunks only store an *index* to objects in other chunks which can be converted into *pointers* before you use them.



Example: Finding track matches

```
void AccessChunks::access_em(const edm::Event& event)
{
    using namespace emid;

    const EMparticle::ChpIndices& tracks = obj.chpindices();
    for(EMparticle::ChpIndices::const_iterator it = tracks.begin();
        it != tracks.end();
        ++it) {

        LinkPtr<ChargedParticleChunk, ChargedParticle> trk(*it);

        // Check whether this is a valid pointer to a track
        if ( trk.isValid() ) {
            float momentum = trk->p();
        }
    }
}
```

This is a really a vector of LinkIndex objects

This turns the index into a pointer like object...

src/AccessChunks.cpp

We first have to check if the conversion succeeded

Then we just use it like a pointer to ChargedParticle



Other chunks in the event

- More physics objects not covered here: bcJets, Taus
- Global information: state of the various magnets, luminosity block number
- Trigger information
 - TMBTriggerChunk (in thumbnail files)
 - L3Chunk
 - L1L2Chunk
 - L1 cal, muon, ctt, trigger framework
 - All L2 global and preprocessor objects



ID Group Corrections

- The various Physics ID group produce certified cuts and corrections which you can/should apply to the data.
- We look at the Muon ID, EM ID and Jet Energy Scale corrections.
- Each one uses a different approach...
- Some of them work exactly the same when using TMBTrees, others differ in their initialization.
- The following examples are for framework programs only...



EM ID Corrections

```
string InterfaceName = "process"  
  
string Packages = "geo read config unptmb mucand em1 em2 analyze"  
[...]  
RCP em1 = <emreco EMReco-scone-postprocess>  
RCP em2 = <emreco EMReco-cnn-postprocess>  
  
RCP mucand = <analysis_tutorial MuoCandReco>  
  
RCP analyze = <analysis_tutorial corrections>
```

rcp/runCorrections.cpp

Add these two packages before your analyze package

That's it !

From now on we are using the corrected values.

Caveat: as of Feb 11, 2003 you have to check out 11 packages by hand and compile them locally in your work area.



Muon ID Corrections

```
Void AccessCorrections::muo_cand(const edm::Event& event)
{
    using namespace edm;
    using namespace muonid;
    typedef MuoCandidateVectorD00M::const_iterator const_iterator

    TKey<MuoCandidateChunk> key;
    THandle<MuoCandidateChunk> handle = key.find(event);

    if(handle.isValid()) {
        int nTight = 0;
        const MuoCandidateVectorD00M& particles = handle->getMuoCandidates();
        for(const_iterator it = particles.begin(); it != particles.end(); ++it) {
            const MuoCandidate& obj = *it;
            float pT      = obj.pT();
            float pTcorr  = obj.pTCorr();

            int whitsA    = obj.whitsA();
            int whitsBC   = obj.whitsBC();
            int shitsA    = obj.shitsA();
            int shitsBC   = obj.shitsBC();

            if(obj.isTight()) nTight++;
        }
    }
}
```

A new chunk type !

Add MuoCandReco to your framework RCP (see previous slide) and include MuoCandidateChunk

New methods

muo_cand is not in a release. You have to check it out by hand and compile it locally.

These would be useful with the original MuonParticle as well...

src/AccessCorrections.cpp



JES Corrections

```
#include "jetcorr/ParticleJetCorr.hpp"
[...]
```

```
Void AccessCorrections::jet_corr(const edm::Event& event)
{
    using namespace edm;
    using namespace jetid;
    typedef vector<Jet>::const_iterator const_iterator;
    THandle<JetChunk> handle = _jetKey.find(event);

    int cone_type = check_jet_type(handle);
    if(cone_type == 0) return;

    if(handle.isValid()) {
        const std::vector<Jet> *particles = handle->getParticles();
        for(const_iterator it = particles->begin(); it != particles->end(); ++it) {
            const Jet& obj = *it;
            // last 2 arguments:
            // 1 = RunII 0.7 cone jet, 2 = 0.5 cone
            // 0 = Data, 1 = mixture MC, 2 == plate MC
            ParticleJetCorr pjc(obj.E(), obj.eta(), obj.detEta()/10, cone_type, 0);
            pjc.calc();
            float C, dCstat, dCsys;
            double Enew;
            pjc.forJets(&C, &dCstat, &dCsys, &Enew);
            float deltaET;
            pjc.forMET(&deltaET);
        }
    }
}
```

Only available for certain jet types. RunII 0.5/0.7 cone

JES code doesn't know anything about D0 framework. Same code can be used in root.

src/AccessCorrections.cpp



What chunks are in a file ?

- In the previous examples we accessed various chunk types.
- None of them exists in a thumbnail file !
 - Chunks can be created on-the-fly and be used to pass information from one package to another.
- All the uncorrected chunks exist in reco output:
 - *EMparticleChunk*, *JetChunk*, *MissingETChunk*, *MuonParticleChunk*
- When reading a thumbnail file, these chunks are recreated on the fly to look (almost) like reco output.
 - The code we wrote will work on reco and DST files as well !



Thumbnail Unpacking

- Thumbnail files have only *ThumbnailChunk*, *TMBTriggerChunk*, *L1L2Chunk*, *HistoryChunk* (+ MC...)
- All our other chunks are created by the thumbnail unpacker; however, some information may be missing compared to reco output !
- You never access *ThumbnailChunk* itself

This package does the necessary unpacking. Just leave it out for DST or reco files !

```
string InterfaceName = "process"

string Packages = "geo read config unptmb mucand em1 em2 analyze"
[...]
```

RCP geo	=	<geometry_management	geometry_management>
RCP read	=	<io_packages	ReadEvent>
RCP config	=	<run_config_fw	RunConfigPkg>
RCP unptmb	=	<thumbnail	UnpThumbNail>

```
[...]
```

rcp/runCorrections.cpp



Controlling Unpacking

- In many cases the thumbnail unpacking dominates your total processing time !
- You can selectively turn off the unpacking of chunks you don't use, e.g. if you are making some simple cuts.
- Be careful, since some of the corrections require access to more than one chunk (e.g. the muon ID corrections needs jets, tracks etc)

```
string PackageName = "UnpThumbNailPkg"  
[...]
```

```
bool doCps=true  
bool doFps=true  
bool doVtx=true  
bool doChp=true  
bool doEM=true  
bool doMuon=true  
bool doTau=true  
bool doJet=true  
bool doMET=true  
bool doL3=true  
bool dobcJet=true  
bool doRecomputeBC=true  
[...]
```

Example: When doing thumbnail skimming the B physics group uses only muons: they can process 80 event/s compared to 20 events/s when unpacking everything.



Accessing Trigger Information

For thumbnails use the TMBTriggerChunk to get trigger information: you can do this before unpacking the other chunks !

```
#include "run_config_fw/TMBTriggerChunk.hpp"
[...]
```

```
void AccessTrigger::access_triggers(const edm::Event& event)
{
    using namespace edm;

    TKey<TMBTriggerChunk> key;
    THandle<TMBTriggerChunk> handle = key.find(event);

    if(handle.isValid()) {

        // Get the trigger list
        _triggers = handle->getTrigList();

        // Get the luminosity block number for this event
        const thumbnail::Global& global = handle->global();
        _luminosity_block_numbers.insert(global.get_lumblk());

    } else {
        // make sure the list is empty
        _triggers.clear();
        log()(ELwarning, "access_trigger")
            << "No TMBTriggerChunk found" << endmsg;
    }
}
```

A list of trigger names

We save every luminosity block number that we encounter...

src/AccessTrigger.cpp



Checking if a trigger fired

Checking if a single trigger or one out of a list of triggers fired is now trivial:

```
bool AccessTrigger::trigger_fired(const std::string& trigger)
{
    return (std::find(_triggers.begin(), _triggers.end(), trigger) != _triggers.end());
}

bool AccessTrigger::trigger_fired(const std::vector<std::string>& trigger_list)
{
    using namespace std;

    for(vector<string>::const_iterator it = trigger_list.begin();
        it != trigger_list.end();
        ++it) {
        if(trigger_fired(*it)) return true;
    }
    return false;
}
```

src/AccessTrigger.cpp



Filtering Events

```
class AccessTrigger : public fwk::Package,  
                    public fwk::Filter,  
                    public fwk::Analyze,  
                    public fwk::JobSummary,  
                    public fwk::FileClose  
{  
public:  
    AccessTrigger(fwk::Context *ctx);  
    virtual ~AccessTrigger();  
  
    virtual fwk::Result filterEvent(const edm::Event& event);  
    virtual fwk::Result fileClose(const fwk::FileInfo& info);  
[...]
```

Two new interfaces
we implement

`analysis_tutorial/AccessTrigger.hpp`

```
fwk::Result AccessTrigger::filterEvent(const edm::Event& event)  
{  
    ++_events_read;  
  
    access_triggers(event);  
  
    if(_user_triggers.size() > 0 && !trigger_fired(_user_triggers)) {  
        return fwk::Result::failure;  
    } else {  
        ++_events_passed;  
        return fwk::Result::success;  
    }  
}
```

This variable is read
from a RCP file

`src/AccessTrigger.cpp`

This is one of the
places where we can
return failure. The
rest of the processing
will be skipped.



Reuse existing code

- The example only works on thumbnails.
- It doesn't allow you to use L3 information
- There already exists a package that implements all this including checks for Monte Carlo data:
 - `analysis_utilities/D0TriggerFilter.hpp`
 - use this when you want to filter or tag events
 - `analysis_utilities/D0TriggerDecoder.hpp`
 - use this when you want to have access to the trigger information
- We make use of this later, but don't go into details here...(see Marco's tutorial).



Accessing L1 Results

```
void AccessTrigger::access_l1l2(const edm::Event& event)
{
    using namespace edm;

    TKey<L1L2Chunk> key;
    THandle<L1L2Chunk> handle = key.find(event);

    if(handle.isValid()) {

        // Access L1 Calorimeter information
        l1cal_reco l1cal = handle.ptr()->Ret_l1cal();

        int    i = 0;
        float  l1_total = l1cal.l1cal_tot(i);
        float  l1_em     = l1cal.l1cal_em(i);
        float  l1_eta    = l1cal.l1cal_tot_eta(i);
        float  l1_phi    = l1cal.l1cal_tot_phi(i);

        // Access L1 CTT information
        l1ctt_reco l1ctt = handle.ptr()->Ret_l1ctt();

        for(int i = 0; i < l1ctt.trk_size(); i++) {
            _l1_ctt->Fill(l1ctt.l1trk_pt(i));
        }
    }
}
```

[...]

src/AccessTrigger.cpp

Both L1 and L2 information is in the L1L2Chunk.

We look at the leading tower in the L1 CAL here



Accessing L2 Results

```
// We start with the global EM objects
vector<l2gblEMObj_reco> l2em_objs = handle.ptr()->Ret_l2gblEMObj();

if(l2em_objs.size() > 0) {

    for(vector<l2gblEMObj_reco>::iterator it = l2em_objs.begin();
        it != l2em_objs.end();
        ++it) {
        float pT = (*it).pt();
        _l2_em->Fill(pT);
    }

    // Plot difference between leading tower and L2 EM object
    // with highest pT
    _l1l2_diff->Fill(l2em_objs[0].pt() - l1_total);
    _l1l2_eta->Fill((l2_eta(l2em_objs[0].ieta()) - l1_eta);

    float delta_phi = fabs(l2_phi(l2em_objs[0].iphi()) - l1_phi);
    if(delta_phi > M_PI/2) delta_phi -= M_PI/2;
    _l1l2_phi->Fill(delta_phi);
}
```

L2 eta and phi use
their own internal
coordinate system !!

[...mercifully skipped here: how to access preprocessor objects...]

src/AccessTrigger.cpp

```
float l2_eta(int eta)
{
    return eta * 8.0/160 - 4.0;
}

float l2_phi(int phi)
{
    return phi * 2 * M_PI/160;
}
```

src/AccessTrigger.cp



Accessing L3 Results

```
void AccessTrigger::access_l3(const edm::Event& event)
{
    using namespace edm;

    THandle<L3Chunk> handle = _l3Key.find(event);
    if(handle.isValid()) {
        typedef L3Chunk::L3ChunkPhysToolMap::const_iterator map_iterator;
        const L3Chunk::L3ChunkPhysToolMap& tool_map = handle->getPhysToolMap();

        // Loop over all tools
        for(map_iterator tool = tool_map.begin();
            tool != tool_map.end();
            ++tool) {
            // Loop over list of results

            for(std::list<L3PhysicsResults*>::const_iterator it = (*tool).second.begin();
                it != (*tool).second.end();
                ++it) {

                if(L3PhysicsResults *result = *it) {
                    use_l3_result(result);
                }
            }
        }
    }
}
```

There are maps for filters and tools.

An entry in the map contains the name of the tool and a list of results.

src/AccessTrigger.cpp



Using L3 PhysicsResults

Some methods are common

You have to cast this pointer to a subclass of L3PhysicsResults

```
void AccessTrigger::use_l3_result(L3PhysicsResults *result)
{
    float detEta = result->get_detectorEta();
    float Et     = result->get_ET();
    if(L3ElePhysicsResults *ele = dynamic_cast<L3ElePhysicsResults*>(result)) {
        float emfrac      = ele->get_emFraction();
        float isolation    = ele->get_isolation();
        if(ele->PsMatch()) {
            const L3CPSCluster& cluster = ele->get_PsCluster();
        }
    }
    else if(L3JetsPhysicsResults *jet = dynamic_cast<L3JetsPhysicsResults*>(result)) {
        float emETfrac      = jet->get_emETfraction();
        float hotcellreacion = jet->get_hotcellratio();
    }
    else if(L3MuonPhysicsResults *muon = dynamic_cast<L3MuonPhysicsResults*>(result)) {
        int missA  = muon->get_nhitmissA();
        int missBC = muon->get_nhitmissBC();
    }
    else if(L3TauPhysicsResults* tau = dynamic_cast<L3TauPhysicsResults*>(result)) {
        float emfrac      = tau->get_emFraction();
        float isolation    = tau->get_isolation();
    }
    else if(L3MEtPhysicsResults *met = dynamic_cast<L3MEtPhysicsResults*>(result)) {
    [...]
    } else {
        log()(ELwarning, "L3_result") << "Unknown l3 type"
            << result->get_name() << endmsg;
    }
}
```

src/AccessTrigger.cpp



Putting things together...

- We now have seen examples of how to access most kinds of data.
- Now you can write your package to do all the filtering, cuts, analysis you want....
- However:
 - **Do not reinvent the wheel.**
 - Many of the things you need are already available in one form or the other
 - We go through a list of useful framework packages:
 - reading, filtering, tagging, writing, using SAM, producing TMBTrees.



ReadEvent

- Source: `io_packages`
- `bin/OBJECTS`: `ReadEvent`
- Many of its RCP options can be overridden on the command line !

Means: add this to your OBJECTS file if you want to use the package

All the options for specifying input files

This is the one you want for testing...

Useful if program crashes on certain events

Use this for good/bad run selection, e.g. for missing ET.

```
// Allowed formats for input file names and lists.
//
// 1. A single filename.
// 2. A wildcard filename.
// 3. A space-separated list of files or wildcards.
// 4. An rcp vector of filenames or wildcards: ("file1" "file2")
// 5. A list file (e.g. listfile:mylist.dat)
//
// Command line overrides (only for first ReadEvent package).
//
// RCP parameter      Command line option
//-----
// InputFile          -input_file <file>
// SkipEvents         -skip_events <n>
// NumEvents          -num_events <n>
// MaxEventsPerFile  -per_input_file <n>
// NumFiles           -num_files <n>
// OnlyCollids        -only_collids <run1> <event1> <run2> <event2> ...
// SkipCollids        -skip_collids <run1> <event1> <run2> <event2> ...
// OnlyRuns           -only_runs <run1> <run2> ...
// SkipRuns           -skip_runs <run1> <run2> ...
```

`io_packages/rcp/ReadEvent.rcp`



D0TriggerFilter

- Source: analysis_utilities
- bin/OBJECTS: RegD0TriggerFilter

```
string PackageName = "D0TriggerFilter"

string Usage = "Filter"

string InputMode = "List"
string Triggers = ( "EM_HI" "EM_HI_2CEM5" )
[...]
```

```
// Trigger decoder configuration.
RCP TriggerDecoder = <analysis_utilities D0TriggerDecoder>
```

analysis_utilities/rcp/D0TriggerFilter.rcp

```
// Chunk used for extracting the list of the L3 triggers
// fired in the event (L3Chunk or TMBTriggerChunk). The
// latter is available only when reading the thumbnail,
// but it can be decoded without decoding the rest of the
// event. NB: the TMBTriggerChunk does not contain mark
// and pass triggers.
string useChunk = "TMBTriggerChunk"

// Enable or disable trigger cuts on MC events.
bool MCtrigger = false
```

analysis_utilities/rcp/D0TriggerSelector.rcp

This will filter events bases on the specified trigger names. The package can also tag events.

There are many more options. Look at the commented RCP file

This references a second RCP file that specifies how to decode the trigger information



Tagging an event...

- Any package can add an arbitrary label called a *Tag* to an event (even if the Event is const !)
- This can be used to tell other packages further down the processing chain which events they should consider.
- You can apply tags to specific chunks as well.

```
// tagging an event is trivial
if([ my complicated condition is true])
    event.tag("MyTag");

[...in another package...]

// checking for tags, too
edm::TagCollection tags;
tags.push_back("MyTag");
if(event.hasTag(tags)) {
    // do something with even
}
```



Applying cuts to objects

- A common task is to apply your own cuts to objects, then write out the interesting events.
- You know how to hard-code the cuts in the code
 - but every change requires recompilation
- You know how to use RCP to parameterize the values of your cuts
 - but that doesn't allow you to change the condition itself
- The ObjectTag package allows you to tag an event based on cuts that you specify in an RCP file.
 - No recompilation required, easy change of cuts
 - Variants of this approach are used in the tmb skimming of at least three physics groups.
 - It is not infinitely flexible, though. If you reach its limits, you have to code things yourself.



ObjectTag

- Source: np_tmb_stream
- bin/OBJECTS: RegObjectTag
 - can use EMparticle, MuonParticle, Jet, MissingET, MuoCandidate in selection

```
string PackageName = "ObjectTag"
```

```
// the tag to produce for the event  
string Tag = "2EM"
```

```
// list of trigger names,  
// empty means no trigger selection applied  
string Trigger = ( "EM_HI" )
```

```
// The list of cuts to apply  
string Cuts = ( "Cut1" "Cut2" )
```

```
string Cut1 = ( "EM" "Pt > 15.0 && AbsEta < 2.4 && emfrac > 0.9 && isolation < 0.15" )  
string Cut2 = ( "EM" "Pt > 10.0 && emfrac > 0.9" )
```

```
// EM id algorithm to use for key
```

```
string EMid_SearchRCPs = ("clusterer", "HMReco", )
```

```
np_tmp_stream/rcp/cut1.rcp
```

This is the tag added to the event if it passes the cuts.

This uses D0TriggerSelector internally, independent of filtering.

Your list of cuts

The conditions for your cuts.

You can actually mix and match object types



WriteEvent

- Source: io_packages
- bin/OBJECTS: WriteEvent
 - Writes events in d0om format. Again lots of options. Understands tags for events and/or chunks !

```
// 1. A single filename.
// 2. A wildcard filename (can only overwrite existing).
// 3. A space-separated list of files or wildcards.
// 4. An rcp vector of filenames or wildcards: ("file1" "file2")
// 5. A list file (e.g. listfile:mylist.dat)
// 6. A filename pattern (a string containing a percent (%) expression).
// The following percent expressions are recognized:
// %n - File sequence number (starts at zero, and is incremented each time
//      a new filename is generated).
// %f - Current input filename (not including directory).
// %e - Current input file extension (final '.' and succeeding chars.).
// %r - Current input file root name (name minus extension & directory).
// %d - Current input file directory.
// %D - A string representing the current date (ddmmmyyyy).
// %T - A string representing the current time (hhmmss).
//
// RCP parameter      Command line option
//-----
// OutputFile         -output_file <file>
// MaxEventsPerFile   -max_per_file <n>
```

io_packages/rcp/WriteEvent.rcp



Using SAM

- Source: sam_manager
- bin/OBJECTS: RegSAMManager
 - Just insert it before ReadEvent
 - Enable/disable it by changing the UseSAM flag in the RCP file if you want to just leave it there, or
 - remove it from the package list if you don't need it

See this afternoon tutorial for more about SAM !



Generating TMBTrees

- Source: tmb_analyze/tmb_tree/thumbnail
- bin/OBJECTS:

- RegLinkedPhysObjReco
- RegTMBTreePkg
- RegTMBCorePkg
- RegTMBBCJetPkg
- RegTMBTreeMCPkg
- RegTMBTRefsPkg

This basically puts TMBAnalyze_x into your program...

```
string Packages = "geo read config unptmb [...] your packages... ] links tmb_tree  
                  tmb_core tmb_bcjet tmb_mc tmb_refs"
```

```
[...]  
RCP unptmb    = <thumbnail UnpThumbNail>  
RCP links     = <linked_physobj LinkedPhysObjReco>  
RCP tmb_tree  = <tmb_tree_maker TMBTreePkg>  
RCP tmb_core  = <tmb_tree_maker TMBCorePkg>  
RCP tmb_bcjet = <tmb_bcjet TMBBCJetPkg>  
RCP tmb_mc    = <mc_analyze TMBTreeMCPkg>  
RCP tmb_refs  = <tmb_analyze TMBTRefsPkg>
```

```
tmb_analyze/rcp/runTMBTreeMaker.rcp
```



Luminosity

- Every event has a luminosity block number associated with it.
- This block number changes (at least) every 60 seconds.
- It references entries in a database and can be used to check if this was a good luminosity block or not (if not, don't use it in your analysis !)
- To calculate the luminosity for your trigger you must have the list of all luminosity blocks of all the data from which your final sample was taken !!!
- => Somebody must keep track of this when you/your physics group skims events !!!



Preferred solution: use SAM

- SAM can find the luminosity block number range for you original sample since it keeps track of the *parentage* of files
 - ✓ raw data file (in SAM)
 - ✓ reco file (in SAM)
 - ✓ thumbnail file (in SAM)
 - ✓ skimmed thumbnail (if stored back into SAM !!!)
 - user skimmed file (usually on your work disk !!!)
- Note that each level might have merged multiple input files, so there is no 1-to-1 correspondence !
- You have to be careful to keep track of the last step !
 - Remember those datasets are constantly growing...



Caveat

- It seems that every group that does thumbnail skimming uses a slightly different approach to keep track of this numbers; so I won't go through any of these...
- However, the idea is always the same: keep track of all luminosity block numbers that are encountered during a skim and make them available with the final sample.
 - Alternatively, keep the filenames of the last set of inputs that is stored in SAM
- It is often straightword to modify the root macros provided by the luminosity group to use the custom format; and your physics group should have already done it.



lm_access_pkg

- Source: lm_access_pkg
- bin/OBJECTS: RegLmAccess

```
string PackageName = "LmAccess"

// this directory contains files with luminosity block numbers
string parentage_path = "$SRT_PRIVATE_CONTEXT/lbns"

string lumfile = "my_summary.lum"

// what kind of luminosity is this? Reconstructed includes offline checks,
// Recorded is what was taken online. For p13, Recorded is probably fine.
string lm_type = "Recorded" //Triggered, Recorded,
                    // Reconstructed, ReconstructedDontCheckL3

// if reconstructed, what kind of data?
string filetype = "thumbnail" // unknown, raw, reconstructed, roottuple, thumbnail

string goodRunsList = "goodrunslst"
string badRunsList = "none"

bool filterBadRuns = true // Don't even look at events from bad runs
string triggers = ( "EM_HI_2EM5" "EM_HI_2EM5_EMFR8" "EM_HI_2EM5_SH"
                    "EM_HI_2EM5_SH_TR" "EM_HI_2EM5_TR" "EM_HI" "EM_HI_EMFR8"
                    "EM_HI_SH" "EM_HI_SH_TR" "EM_HI_TR" "EM_MX" "EM_MX_EMFR8"
                    "EM_MX_SH" "EM_MX_TR" "2EM_HI")
```

lm_access_pkg/rcp/LmAccess.rcp



Example: NP skims

1. We want to run over the di-em sample (which is mostly in SAM)
2. Run a script from the `lm_access` package to generate the parentage files
 - `makeDimensionParentageList.py`
`'data_tier thumbnail-bygroup and`
`file_name 2em%p1305%'`
3. Put the directory location into `parentage_path` (see previous slide)
4. Put `lm_access_pkg` into your framework RCP
5. Run over the specified data set
 - each file we encounter should have a corresponding parentage file from step 2



Output

- Output looks something like this (didn't use good/bad run lists and ran over small sample):

```
% cat my_summary.lum
Luminosity summary
/rooms/dining/ethomas/np_skims/p13.04.00/2mu/01-DEC-2002/2mu_p130400_Dec01_ID12184
End of Job Summary
LmType : Recorded
Filetype :thumbnail
      EM_HI_2EM5 =      6098.13 mub-1      Good Events: 0      Bad Events: 0
EM_HI_2EM5_EMFR8 =      6098.13 mub-1      Good Events: 0      Bad Events: 0
      EM_HI_2EM5_SH =      6098.13 mub-1      Good Events: 0      Bad Events: 0
EM_HI_2EM5_SH_TR =      6098.13 mub-1      Good Events: 0      Bad Events: 0
      EM_HI_2EM5_TR =      6098.13 mub-1      Good Events: 0      Bad Events: 0
          EM_HI =      6098.13 mub-1      Good Events: 0      Bad Events: 0
      EM_HI_EMFR8 =      6098.13 mub-1      Good Events: 0      Bad Events: 0
          EM_HI_SH =      6098.13 mub-1      Good Events: 0      Bad Events: 0
      EM_HI_SH_TR =      6098.13 mub-1      Good Events: 0      Bad Events: 0
          EM_HI_TR =      6098.13 mub-1      Good Events: 0      Bad Events: 0
          EM_MX =      6098.13 mub-1      Good Events: 0      Bad Events: 0
      EM_MX_EMFR8 =      6098.13 mub-1      Good Events: 0      Bad Events: 0
          EM_MX_SH =      6098.13 mub-1      Good Events: 0      Bad Events: 0
          EM_MX_TR =      6098.13 mub-1      Good Events: 0      Bad Events: 0
          2EM_HI =      6098.13 mub-1      Good Events: 0      Bad Events: 0
```



Trying it yourself...

- Examples are in *analysis_tutorial*
 - see next slide
- You need tons of additional packages, many of which are not in the release (or not the right version in the release).
- *analysis_tutorial* will therefore be never in a release itself...
 - But you can use the things described here with D0ChunkAnalyze or other programs.



Package List (Feb 11, 2003)

```
[...setup your work area...]  
% setup d0cvs  
% addpkg -h analysis_tutorial  
% addpkg emcandidate v00-01-04  
% addpkg em_util v00-02-76  
% addpkg em_evt v00-15-26  
% addpkg emreco v00-12-20  
% addpkg hmreco v00-06-13  
% addpkg muo_cand p13-br-03  
% addpkg jetcorr v04-00-01  
% addpkg thumbnail  
% addpkg tmb_analyze  
% addpkg tmb_tree  
% addpkg bc_eTagreco  
% addpkg bcjet_evt  
% addpkg empartfit  
% addpkg linked_physobj  
% addpkg lm_access v00-03-02  
% addpkg -h lm_access_pkg  
% make all
```

You can go for lunch now...



clued0 hack of the day...

- The linker that will come with gcc-3.2 is greatly improved...
- Unfortunately it seems to take a few more releases until we switch...
- Before you compile in an older release, do
 - `setenv COMPILER_PATH /usr/local/ld-hack`
 - or in bash
 - `export COMPILER_PATH=/usr/local/ld-hack`
- This should improve your link times on machines with less memory, especially when using a test release with gcc (in one example from 15 minutes to about 3 mins...)