

CDF/xxxx/xxxx/XXX/XXX
version 1.01 October 2, 2004

STNTUPLE manual

R.Culbertson, P.Murat,
Fermi National Accelerator Laboratory

A.Dominguez,
LBNL,

A.Taffard, B.Heinemann,
University of Liverpool

D.Tsybychev,
University of Florida,

J.-Y. Chung, R.Hughes, E.Thomson,
Ohio State University

B.Knuteson,
University of Chicago,

S.Sarkar,
LPNHE-Universites de Paris 6 & 7 and INFN, Rome,

C.Issever,
University of California of Santa Barbara,

K.Terashi,

A.Hamilton,
University of Alberta / IPP Canada

Abstract

This manual is well known to be incomplete. You're welcome to send us your favorite topics to be included.

1 Introduction

The name “STNTUPLE” stands for 2 related things: a micro-DST CDF data format and a set of utilities to read the data stored in this format. Originally STNTUPLE was designed to be just an extended ntuple format, it still remains such. Format of ROOT files, however, allows to minimize the difference between the ntuple and micro-DST, this is how STNTUPLE became a micro-DST.

2 Getting Started

In this section includes several examples illustrating how to start using STNTUPLE for analysis quickly.

2.1 STNTUPLE executables

Stntuple package builds several executables.

- **stnmaker.exe** is a light-weight executable, which runs on the output of ProductionExe (cdfSim, cdfGen) and produces output in STNTUPLE format. stnmaker.exe doesn't recalculate parameters of the input objects and is stable in time. Use this executable if for the generator-level studies. To build stnmaker.exe do

```
make Stntuple.stnmaker
```

stnmaker.exe is being built and validated nightly.

- **stnmaker_prod.exe** - the purpose of this executable is to generate STNTUPLE's with most up-to-date information. It runs on the output of ProductionExe, re-makes high-level objects and stores the information in STNTUPLE files. **stnmaker_prod.exe** also runs the following analysis modules, which are not part of Production: **CosmicFinderModule**, **JetProbModule**, **TrackRefitterModule**, **SecVtxModule**, **TopEventModule**

Module contents of **stnmaker_prod.exe** may change significantly from one release to another. To build this executable against CDF offline release 4.9.1:

```
setup cdfsoft2 4.9.1
newrel -t 4.9.1 dev_239
cd dev_239
addpkg Stntuple dev_239
chmod 755 Stntuple/scripts/build_stntuple
Stntuple/scripts/build_stntuple
gmake Stntuple._prod
```

The latest instructions on how to build the current version of **stnmaker_prod.exe** can be found at [STNTUPLE home page](#)

- **stnmakerFat.exe** - specialized executable for B-tag studies. Aaron knows more about its contents.

2.2 stnmaker_prod.exe

stnmaker_prod.exe is the main executable used for producing STNTUPLES. It runs on the output of ProductionExe, remakes necessary high-level objects and writes out STNTUPLE files.

Error codes:

- -101: StntupleInitElectronBlock: Too many (>100) electrons
- -102: Bob's electron w/o max Pt track
- -151: StntupleInitMuonBlock: Null tower for a muon
- -201: StntupleInitTrack: CT_ReFit failed
- -301: StntupleInitJetProbColl: no JetProbColl

2.3 B-taggers in stnmaker_prod.exe

Here we list conventions used to run B-tagging modules within stnmaker_prod.exe.

- To simplify analysis code all the B-tagging modules should be using the same jet collection and thus should be coordinated. This collection of choice is the first jet collection as declared to the StntupleMaker module. Code filling the B-tagging blocks is trying to cross-link the first jet collection declared to StntupleMakerModule, lists of B-tags and tracks.

3 STNTUPLE Primers

3.1 Setup

- Make sure that you have copied Stntuple `.rootrc` and `rootlogon.C` files into the working directory

```
cp Stntuple/scripts/.rootrc .  
cp Stntuple/scripts/rootlogon.C .
```

- make sure that you have build Stntuple shared libraries including `libStntuple_ana.so`

```
make Stntuple.shared
```

3.2 Primer#0

- Build stnmaker.exe
- run stnmaker.exe on the reconstructed file (written by the ProductionExe) to create STNTUPLE:

```
stnmaker.exe Stntuple/test/stnmaker/run_stnmaker_mc.tcl
ls -l results
total 296
drwxr-xr-x  2 murat  cdf          4096 Dec 30 10:54 ./
drwxr-xr-x 13 murat  cdf          4096 Dec 30 10:27 ../
-rw-r--r--  1 murat  cdf        121033 Dec 30 10:54 isajet_qq_today_val.stntuple
```

- start ROOT session and execute analysis script Stntuple/ana/mc_ana.C:

```
root.exe
root [1] .L shlib/$BFARCH/libStntuple_ana.so
root [0] .x Stntuple/ana/mc_ana.C
----- end job: ---- McAna
```

at this point a ROOT canvas with the filled histograms pops up on the screen

3.3 Primer#1

... you are done with primer#0

... and current directory is the topmost directory of your test release and you have STNTUPLE shared libraries built in ./shlib/\$BFARCH. Here is what you do next:

```
>root.exe
root [6] TStnAna x("/cdf/data17b/ntuples/ntupled_4.0.0_127251_1.root")
OBJ: TChainElement STNTUPLE /cdf/data17b/ntuples/ntupled_4.0.0_127251_1.root
root [7] x.PrintStat(100)
```

```
-----
..... branch name .... TotBytes ZipBytes CompFactor
-----
HeaderBlock          64.000      0.000          0          0  0.00
CalDataBlock        26767.500     8051.457    103994974    26659193  3.90
CesDataBlock        18628.200       42.453     75032714    13931349  5.39
PesDataBlock        12653.000      776.155     50178227     7376369  6.80
ClcDataBlock         428.840      112.056     1708272     1371883  1.25
CmpDataBlock         166.560       46.060      686544      442831  1.55
CmuDataBlock         1031.360      801.801     4289153     1026712  4.18
CmxDataBlock         1122.080      734.672     3923748      777857  5.04
SvxDataBlock        166034.740    68270.212    670229894   155202243  4.32
```

CprDataBlock	3185.000	0.000	12835503	4008274	3.20
TriggerBlock	172.000	0.000	688193	212744	3.23
VertexBlock	292.560	90.091	1133770	301686	3.76
TrackBlock	25405.000	11553.465	98378351	48921064	2.01
TrackLinkBlock	48.000	0.000	178251	6018	29.62
JetBlock	864.200	416.687	3419747	1296987	2.64
FatJetBlock	864.200	416.687	3419909	1301756	2.63
ElectronBlock	424.040	345.232	1519096	651658	2.33
EleLinkBlock	38.000	0.000	116842	5856	19.95
PhotonBlock	529.920	441.444	1894558	976462	1.94
ClusterBlock	14486.080	5197.959	55297931	20142201	2.75
MuonBlock	663.370	524.774	2655088	793957	3.34
TauBlock	202.080	160.035	752564	229403	3.28
TagBlock	65.000	0.000	241816	8972	26.95
MetBlock	124.000	0.000	496248	83754	5.93
XftBlock	17467.640	10808.292	68148805	11290110	6.04
SvtDataBlock	132.430	219.977	558926	203899	2.74

..... total

total event 291859.800 97254.882

DRAFT OCTOBER

3.4 Primer#2: copy events from one file to another

```
int copy() {
//-----
//  this script shows how to copy a few events from STNTUPLE file into
//  another file and drop some data blocks(by default all the data blocks
//  are copied into the output file)
//
//  "dropped" data blocks are not read in, unless explicitly requested by
//  one of the TStnModule's included into event loop. This allows to copy
//  ntuples with one (or more) branches corrupted
//
//  As no other STNTUPLE library depends on libStntuple_loop, you can check
//  it out from development and rebuild, leaving all the rest code unchanged:
//
//  cvs co -A Stntuple/loop
//  cvs co -A Stntuple/Stntuple/loop
//  setenv USESHLIBS 1
//  make Stntuple._loop
//-----
    TChain* chain = new TChain("STNTUPLE");
    chain->Add("/cdf/scratch/murat/etme01-138425-139339-s232.stn");
    TStnAna* x = new TStnAna(chain);
    TStnOutputModule* om;
    om = new TStnOutputModule("./results/junk.root");
    om->SetMaxFileSize(1000);
    om->DropDataBlock("PhotonBlock");
    om->DropDataBlock("MuonBlock");
    x->SetOutputModule(om);
    x->Run(100);
}
```

4 STNTUPLE Event Loop

Most of the STNTUPLE utilities are designed assuming the following analysis model

- analysis code is compiled into one or several shared libraries
- these libraries are loaded dynamically at run time (via rootlogon.C, for example)
- interpreted code is used only for the user interface purposes

STNTUPLE event loop, implemented by

TStnAna class,

is designed to be fully interactive. Upon construction event loop is empty. Code to be executed within the event loop - analysis modules - are inserted into event loop by the user at run time as shown in the following example

```
TStnAna* x = new TStnAna();
int filter_mode = 1;
x->AddModule(m,filter_mode);
```

It is assumed that the analysis modules are also compiled into the shared libraries which can be loaded selectively.

4.1 Methods of TStnAna

Here we describe the method provided by TStnAna class.

- **int TStnAna::AddModule(TStnModule* Module, Int_t FilteringMode) :** add Module to the end of the list of modules to be executed in the event loop. **FilteringMode** defines module execution mode (see below)
- **int TStnAna::Continue(int NEvents) :** process next NEvents starting from the next event in the chain
- **int TStnAna::ProcessRun(int RunMin, int RunMax) :** given an initialized chain, process range of runs
- **int TStnAna::ProcessEvent(int Run, int Event) :** process event with given run/event numbers
- **int TStnAna::Run(int NEvents, int RunMin, int RunMax) :** process NEvents, consider only events with run number in the range from RunMin to RunMax
- **int TStnAna::ProcessEventList(TEventList* EventList) :** process list of the tree entries specified in **EventList**, see description of ROOT class TEventList for more detail
- **int TStnAna::ProcessEventList(Int_t* EventList) :** process list events specified in **EventList** in the following format:

run1, event1, run2, event2, run3, event3, ..., runN, eventN, -1

where -1 marks end of the list

5 STNTUPLE Modules

- Base class for the analysis module implemented in the class **TStnModule**.

All the analysis modules are filters, depending on their decision execution of the event loop for a given event may be stopped. By default filtering capabilities of the module are turned off.

5.1 Modules as Filters

All the STNTUPLE modules have event filtering capabilities which allow to terminate execution of the event loop for a given event based on the decision made by the module. For a module to act as an event filter it is necessary to

- specify module filtering mode during the initialization
- specify module decision for each event
- **int TStnModule::SetFilteringMode(int Mode)** : specifies module filtering mode
 - FilteringModule = 0[default]: no filtering
 - FilteringModule=1: module acts as an event filter. If after execution of the module
 $\text{Module} \rightarrow \text{Passed}() = 0$
, the rest modules in the event loop will not be executed for this event
 - FilteringModule=2: veto mode. If after execution of the module
 $\text{Module} \rightarrow \text{Passed}() = 1$
, the rest of the event loop will not be executed for this event
- **int TStnModule::FilteringMode()** : returns value of the module filtering mode
- **int TStnModule::SetPassed(int Passed)** : specifies module decision
 - Passed = 0: event failed
 - Passed = 1: event passed

If a module is supposed to act as a filter, this method has to be called at least once from inside the module **Event** entry point

- **int TStnModule::Passed()** : returns module decision

5.2 Modules and Histograms

- TStnModule has list of histograms as one of its data members. Use TStnModule::HBook1, TStnModule::HBook2 etc to book the histograms. See Stntuple/ana/TEmFilterModule.cc as an example.
- use TStnAna::SaveHist to save histograms in the end of job. Example:

```
TStnAna* x = new TStnAna();
....
x->SaveHist('myfile.root');
```

all the histograms booked using methods of TStnModule - TStnModule::HBook*, TStnModule::HBProf - are automatically saved.

- "safe" way of registering STNTUPLE data blocks in the module:

```
int TRPhiWt::BeginJob() {

    RegisterDataBlock("TrackBlock"      , "TStnTrackBlock"      , &fTr );
    RegisterDataBlock("TrackLinkBlock"   , "TStnTrackLinkBlock"  , &fTl );
    RegisterDataBlock("SvxDataBlock"     , "TSvxDataBlock"      , &fHit);
    RegisterDataBlock("SiIsectBlock"     , "TSiIsectBlock"      , &fI );
    RegisterDataBlock("SiStripsBlock"    , "TSiStripBlock"      , &fSi );
    return 0;
}
```

If a branch which name is defined by the 1st parameter doesn't exist, the corresponding call still creates data block of a class defined by the 2nd parameter (class name) and returns it into the 3rd parameter (note "&"). The block never gets filled, no protection "if's" is necessary, all the histograms end up being empty.

6 Storing collections

6.1 Naming

- Each collection of CDF objects can be identified by 2 strings: its **description** and **name of the creating process** - see AC++ docs for more details. To reflect this mnemonics, in StntupleMaker talk-to's CDF collection names are defined as “**process_name@description**”, where the process name and the description are separated by the column, for example, **PROD@COT_Global_Tracking**.
- “@” is used instead of “:” to allow working with the corresponding branches in split mode
- it is possible to omit process name and refer to the collection by its description only, for example, **COT_Global_Tracking**. In this case default process name is assumed. The default process name can be redefined in the talk-to to StntupleMaker, for example:

```
talk StntupleMaker
  processName set L3
exit
```

6.2 The Defaults

CDF data model allows to store several collections of the same type in the event record, for example, collections of jets reconstructed by different algorithms. Same holds for tracks. By default StntupleMaker stores only one collection of each type in the STNTUPLE. The defaults are

- the default process name is **PROD**
- defTracks collection corresponding to the default process is **always** stored in the “**TrackBlock**” data block.
- **JetCluModule-cone0.4** collection corresponding to the default process is stored in the data block with the name **JetBlock**

6.3 Storing more than one collection

It is possible to store more than one collection of tracks or jets in STNTUPLE. The rules are as follows.

- tracks:
 - the following talk-to
- ```
talk StntupleMaker
 trackCollName set PROD@COT_Global_Tracking_SL COT_Global_Tracking_HL
exit
```

adds **COT\_Global\_Tracking\_SL** collection created by the process **PROD** and **COT\_Global\_Tracking\_HL** collection created by default process to the list of stored track collections. If default process name is **PROD** the collections will be stored in the data blocks named **PROD@COT\_Global\_Tracking\_SL** and **PROD@COT\_Global\_Tracking\_HL** correspondingly

- rules for the jets are slightly different. When several jet collections are specified in the talk-to, the first one is stored on the branch with predefined name - **JetBlock**, the rest collections are stored on the branches with the names defined by the names of the collections. For example,

– the following talk-to

```
talk StntupleMaker
 jetCollName set JetCluModule-cone0.7 PROD@JetCluModule-cone0.4
exit
```

instructs StntupleMaker to store jet collection **JetCluModule-cone0.7** in the data block with the name **JetBlock** and collection **JetCluModule-cone0.7** created by the process **PROD** in the data block named **PROD@JetCluModule-cone0.4**.

Therefore the difference between the tracks and the jets is that for the tracks the contents of the default track block (**TrackBlock**) is predefined, while for the jets there is no such a limitation. The reason is that **defTracks** track collection is used by the electron, muon and tau reconstruction algorithms, so storing it in the STNTUPLE on a branch with predefined name simplifies the analysis code.

## 7 STNTUPLE and DB Constants

Some run-dependent constants (luminosities, beam positions etc) are usually needed even at the latest stages of the analysis. Such constants are stored (once per run) in the STNTUPLE files and retrieved as necessary. The constants can be accessed via STNTUPLE database manager (class TStnDBManager). All the STNTUPLE “DB tables” are named and accessed by name. Tables stored in the STNTUPLE along with their respective names are listed below.

- trigger tables, table name “TriggerTable”, class TStnTriggerTable
- COT beam positions, table name “CotBeamPos”, class TStnBeamPosition.
- SVX beam positions, table name “SvxBeamPos”, class TStnBeamPosition.
- run summary constants, table name “RunSummary”, class TStnRunSummary.

The following simple example illustrates how to access the DB constants stored in STNTUPLE.

```
//-----
int TTrigAnaModule::BeginRun() {
 // talk to DB manager and retrieve the
 // trigger table for this run

 TStnDBManager* dbm = TStnDBManager::Instance();

 TStnTriggerTable* tt = (TStnTriggerTable*) dbm->GetTable("TriggerTable");

 // get COT beam position at Z=0

 TStnBeamPos* bp = (TStnBeamPos*) dbm->GetTable("CotBeamPos");
 Double_t x0 = bp->X0();
 Double_t y0 = bp->Y0();
}
```

## 8 Accessing Trigger Information

2 STNTUPLE branches (“TriggerBlock” and “TrigSimBlock”) store trigger data. Both store object of the same type - **TStnTriggerBlock** - filled with the real (“TriggerBlock”) or emulated (“TrigSimBlock”) data.

To get trigger table for a given run:

```
//-----
int TTrigAnaModule::BeginRun() {
 // talk to DB manager and retrieve the
 // trigger table for this run

 TStnDBManager* dbm = TStnDBManager::Instance();

 TStnTriggerTable* tt = (TStnTriggerTable*) dbm->GetTable("TriggerTable");
}
```

To print current trigger table interactively from the ROOT prompt:

```
root[1] TStnDBManager* dbm = TStnDBManager::Instance();
root[2] dbm->GetTable("TriggerTable")->Print();
```

### 8.1 Testing L3 bits inside STNTUPLE analysis module

Every STNTUPLE module can use trigger bits as prerequisites, such that the module would be executed only if the event passed corresponding L3 trigger paths. This can be done as follows.

- During module initialization stage define L3 trigger paths you want the module to test:

```
root[1] TWenuMonModule* m_wenu = new TWenuModule();
root[2] m_wenu->AddL3TriggerName(‘‘MUON_CMUP18’’);
```

- BeginRun entry point should include the following lines:

```
int rn = GetHeaderBlock()->RunNumber();
TStntuple::Init(rn);
TStntuple::InitListOfL3Triggers(this);
```

- Module should read TrigggerBlock branch and Event entry point should include the following:

```
fTriggerBlock->GetEntry(ientry);
int passed = TStntuple::CheckL3TriggerPath(fTriggerBlock,GetListOfL3Triggers());
if (! passed) // leave the module
```

Note that GetListOFL3Triggers() is a method of TStnModule. For a complete example see <http://cdfcodebrowser.fnal.gov/CdfCode/source/Stntuple/ana/TWenuMonMod>

## 8.2 Special Utilities

- **TStntuple::PrintListOfPassedTriggers(TStnTriggerBlock\* TriggerBlock, Int\_t Level)** : assuming that DB manager has been initialized for a given run (this happens by default within TStnAna event loop) prints list of the triggers of a given **Level** given event did pass. Set Level=0 to print passed triggers of all 3 levels. Use it as follows:

## 9 Extrapolator

STNTUPLE has simple utilities for track extrapolation. To use them one needs to build (standalone) library `libStntuple_geom.so` and load it into the ROOT session:

```
make Stntuple._geom
root
root [0] .L ./shlib/\$BFARCH/libStntuple_geom.so
```

Classes compiled into this library are defined in `Stntuple/geom` subpackage. `TSimpleExtrapolator` class has methods to extrapolate track up to CES and CMU.

### 9.1 Extrapolation to CMU

```
int TSimpleExtrapolator::SwimToCmu(int Charge,
 int NPoints,
 const Double_t* Radius,
 TTrajectoryPoint* Point)
```

Given initial point on a trajectory defined by `Point[0]` extrapolates track of a given `Charge` and returns track positions at `NPoints` different radii in array `Point`. Dimension of `Point` should thus be at least `NPoints+1`. Values of radii are stored in array `Radius`, which dimension should be `NPoints` or more. Return codes by the extrapolator are as follows:

- 0: extrapolation was successful -1:  $P \downarrow P(\min)$ : total momentum is less than minimal allowed -2:  $P_t \downarrow P_t(\min)$ : transverse momentum is less than allowed minimum -3:  $\eta$  is outside the range

When negative error code is returned, results of the extrapolation should not be relied upon.

### 9.2 Extrapolation to CES

```
int TSimpleExtrapolator::SwimToCes(TTrajectoryPoint* Point , int Charge,
 Int_t& Side , Int_t& Wedge ,
 Double_t& XWedge, Double_t& ZWedge)
```

Given initial point on a trajectory defined by `Point[0]` extrapolates track of a given `Charge` out to CES plane and returns `Side` and `Wedge` of the hit wedge and also `XWedge` and `ZWedge` - coordinates of the hit in the local coordinate system of the hit wedge. Note, that `ZWedge` is thus always positive. Global coordinates of the intersection point can be retrieved by calling

```
TSimpleExtrapolator::GetFinalDirection,
TSimpleExtrapolator::GetFinalPosition,
TSimpleExtrapolator::GetFinalMomentum,
TSimpleExtrapolator::GetFinalResults
```

### 9.3 Example of using TSimpleExtrapolator

This section gives an example of using TSimpleExtrapolator.

```
//-----
// calculate parameters of the SEED track at CES and do it once per tau
//-----
TSimpleExtrapolator* fExtrapolator = new TSimpleExtrapolator();
TTrajectoryPoint p0;
TStnTrk* trk = ...; // defined elsewhere
TLorentzVector* tmom;

double xyz[8], xw, zw, ptot;
int trk_charge, side, wedge;

xyz[0] = -trk->D0()*sin(trk->Phi0());
xyz[1] = trk->D0()*cos(trk->Phi0());
xyz[2] = trk->Z0();

tmom = trk->Momentum();

ptot = tmom->P();

xyz[3] = tmom->Px()/ptot;
xyz[4] = tmom->Py()/ptot;
xyz[5] = tmom->Pz()/ptot;

xyz[6] = 0;
xyz[7] = ptot;

p0.SetPoint(xyz);
trk_charge = trk->Charge();
fExtrapolator->SwimToCes(&p0, trk_charge, side, wedge, xw, zw);

// track coordinates at CES

float track_x_ces = xw;
float track_z_ces = zw;

```

## 10 Luminosity information

- Instantaneous luminosity for an event is stored in the header block (see TStnHeaderBlock). It is accessible from within any analysis module as follows:

```
float inst_lum = GetHeaderBlock()->InstLum()
```

luminosity is stored in units of  $cm^2 \cdot sec^{-1}$ , so the numbers are of the order of  $10^{30}$

- When stntupling job runs, TStnDBManager updates the luminosity info once per begin run record, however, db record is saved into the ntuple file only once. If events from the same run end up in 2 different files, each of the files has a db subdirectory for this run with the luminosity information.

STNTUPLE analysis job creates a list of processed runs which it updates as the execution progresses. This list doesn't have duplicates and each run appears in this list only once. As such this procedure doesn't involve any double counting and the only assumption made is that for each run we are processing all its run sections (which is almost always true modulo bad/lost files). The right way is to integrate luminosity in the stntupling job by counting each processed run section separately - this is not implemented yet, volunteers welcome and I can even show how to do it. There are also 3 histograms created by default by the analysis job - they contain delivered luminosity, online luminosity and offline luminosity for the runs which have at least one event stored in STNTUPLE (this is to be fixed)

## 11 High-Level Object ID

Classes TStnTauID, TStnElectronID etc provide utilities for identification of the high-level physics objects. We consider typical use case of these classes taking TStnTauID class as an example:

```
TStnTauID* fTauID = new TStnTauID();
int ntau = fTauBlock->NTaus();
for (int i=0; i<ntau; i++) {
 TStnTau* tau = fTauBlock->Tau(i);
 int id_word = fTauID->IDWord(tau);

 if (id_word == 0) {
 // the tau passed all the ID cuts

 }
}
```

**TStnTauID::IDWord** checks the tau identification cuts for a given tau candidate and returned integer - **id\_word** - is a bit-packed mask defining passed and failed cuts. Each ID cut implemented in TStnTauID has a bit number assigned to it. When a cut fails, the bit corresponding to it is set to 1. **id\_word=0** means that tau object passed all the selection cuts, the next example shows how to check whether a given tau object passed given ID cut:

```
TStnTauID* id = fTauID;
...
int id_word = fTauID->IDWord(tau);
if ((id_word & id->kCalIsoBit) == 0) {
 // tau candidate passed calorimetry
 // isolation cut

}
```

## 11.1 TStnTauID

`TStnTauID` class provides set of tau identification utilities. Two most important methods of the class are `TStnTauID::IDWord` and `TStnTauID::LooseIDWord`, which implement “tight” and “loose” definitions of tau leptons.

### 11.1.1 Bits and ID cuts

Assignment of the bits is described in `TStnTauID.hh` include file.

```
kDetEtaBit = 0x1 << 0,
kEtBit = 0x1 << 1,
kSeedTowerEtBit = 0x1 << 2,
kSeedTrackPtBit = 0x1 << 3,
kEmfrBit = 0x1 << 4,
kNTrk1030Bit = 0x1 << 5,
kNPi01030Bit = 0x1 << 6,
kCalIsoBit = 0x1 << 7,
kCalIso1Bit = 0x1 << 8,
kTrkIsoBit = 0x1 << 9,
kNAXSegBit = 0x1 << 11,
kNStSegBit = 0x1 << 12,
kSeedTrackD0Bit = 0x1 << 13,
kSeedTrackZ0Bit = 0x1 << 14,
kSeedTrackDzBit = 0x1 << 15,
kZCesBit = 0x1 << 16,
kCalMassBit = 0x1 << 17,
kTrkMassBit = 0x1 << 18,
kVisMassBit = 0x1 << 19,
kNMuStubsBit = 0x1 << 20, // this shouldn't really be used
kTightElectronBit = 0x1 << 21,
kTrkAngleBit = 0x1 << 22,
kPi0AngleBit = 0x1 << 23
```

### 11.1.2 Int\_t TStnTauID::IDWord(TStnTau\* Tau)

Implements the following ID cuts:

- $|\eta_{\text{detector}}| < \eta_{\text{max}}$  (default = 1.0)
- $E_t > E_{t \text{ min}}$  (default = 20.0)
- $E_{t \text{ seed tower}} > E_{t \text{ min}}$  (default = 10.0)
- $P_{t \text{ seed track}} > P_{t \text{ min}}$  (default = 4.5)
- electron rejection cut. Depending of the value of TStnTauID::fSelectElectrons this cut can be reversed.

– TStnTauID::fSelectElectrons = 0 selects electron rejection mode:

$$EM_{\text{fraction}}(\tau) < 1 - \xi/(E/p)$$

– TStnTauID::fSelectElectrons = 1 inverts the cut to select electrons:

$$EM_{\text{fraction}}(\tau) > 1 - \xi/(E/p)$$

- $N_{\text{tracks}}(10 - 30) < N_{\text{max}}$  (default :  $N_{\text{max}} = 1$ )

### 11.2 Int\_t TStnTauID::LooseIDWord(TStnTau\* Tau)

“Loose” tau identification cuts are a subset of the “tight” set of cuts:

- $|\eta_{\text{detector}}| < \eta_{\text{max}}$  (default = 1.0)
- $E_t > E_{t \text{ min}}$  (default = 20.0)
- $E_{t \text{ seed tower}} > E_{t \text{ min}}$  (default = 10.0)
- $P_{t \text{ seed track}} > P_{t \text{ min}}$  (default = 4.5)
- electron rejection cut. Depending of the value of TStnTauID::fSelectElectrons this cut can be reversed.

– TStnTauID::fSelectElectrons = 0 selects electron rejection mode:

$$EM_{\text{fraction}}(\tau) < 1 - \xi/(E/p)$$

– TStnTauID::fSelectElectrons = 1 inverts the cut to select electrons:

$$EM_{\text{fraction}}(\tau) > 1 - \xi/(E/p)$$

## 12 STNTUPLE Datasets

In this section we discuss use of the cataloged STNTUPLE datasets and describe the cataloging procedure. STNTUPLE datasets as well as the catalogs can be distributed over the network and span over multiple static file servers. They can also be stored in the DH system. Their structure is very similar to the structure of the datasets described in the CDF Data File Catalog. STNTUPLE data catalog however is implemented as a text database and its maintenance doesn't require any specialized tools and privileges.

### 12.1 Using cataloged datasets

The following example explains use of the cataloged dataset.

```
----- tau_ana.C
TStnAna* x = NULL;
TChain* chain = NULL;
TStnCatalog* catalog = NULL;
TStnDataset* dataset = NULL;

void tau_ana(const char* Book, const char* Dataset, Int_t NEvents = 0,
 int MinRun = 1, int MaxRun=9999999) {

 catalog = new TStnCatalog();
 dataset = new TStnDataset(Book,Dataset,MinRun,MaxRun);
 catalog->InitDataset(dataset);
 // dataset->Print(); // you can uncomment this line

 TStnAna* x = new TStnAna(dataset);

 x->Run(NEvents,MinRun,MaxRun);
}
----- end of tau_ana.C
....
```

The analysis driver script **tau\_ana.C** can be called from the ROOT prompt as follows:

- to process the whole dataset:

```
root [0] .x tau_ana.C('stntuple/dev_242','zewk0d');
```

- to process a single fileset:

```
root [0] .x tau_ana.C('stntuple/dev_242','ttopi:001');
```

- to process first 100 events from the fileset:

```
root [0] .x tau_ana.C('stntuple/dev_239',ttopi:001',100);
```

Book name “**file**” has special meaning and allows to initialize a dataset as consisting of a single non-cataloged file, such that TStnCatalog/TStnDataset interface can be transparently used to work with the local files:

```
dataset = new TStnDataset("STNTUPLE");
catalog = new TStnCatalog();
catalog->InitDataset('file', '/home/murat/a.root');
```

## 12.2 Format of the STNTUPLE Dataset Catalog

STNTUPLE can simultaneously use several data catalogs either local or located on a remote server. Location of the data catalog nodes has to be specified in one of **.rootrc** files (global, home or in working directory). The following example shows the corresponding fragment of the **.rootrc** file:

```
#-----
STNTUPLE catalog servers (FCDFLNx2 is a fallback for FCDFLNx3)
#-----
Stntuple.Catalog txt://fcdflnx3.fnal.gov/cdf/home/cdfopr/cafdfc
+Stntuple.Catalog txt://ncdf131.fnal.gov/home/murat/cafdfc
+Stntuple.Catalog txt://fcdflnx2.fnal.gov/cdf/home/cdfopr/cafdfc
#-----
```

Standard catalog of STNTUPLE datasets is located on FCDFLNx3.

To enable handling of files stored in DCACHE **.rootrc** file should include the following lines:

```
#-----
plugin handlers
#-----
+Plugin.TFile: ^dcache: TDCacheFile DCache "TDCacheFile(const char*,Option_t*,const char*,Int_t)"
+Plugin.TFile: ^dcap: TDCacheFile DCache "TDCacheFile(const char*,Option_t*,const char*,Int_t)"
```

- Dataset with the same name may be described in several catalogs. The first one in the sequence of the search will be used. The sequence of the search is defined by the sequence in which catalogs are listed in the **.rootrc** file. If private catalog is listed first, it will be searched first.

Datasets can be versioned. For example, we have dev\_240 and dev\_241 and dev\_242 versions of STNTUPLE's for the same datasets. STNTUPLE catalog therefore has to have multiple partitions, we use the same notation as CDF DFC - **book** - for those. STNTUPLE catalog is a directory, a **book** is a subdirectory.

For example, STNTUPLE data catalog located in /data12/murat/tgeant/test directory and described by the following directory structure

```

/data12/murat/tgeant/test:
used 8 available 1628008
drwxr-xr-x 2 murat cdf 4096 Apr 23 21:52 .
drwxr-xr-x 26 murat cdf 4096 Apr 23 21:52 ..
drwxr-xr-x 2 murat cdf 4096 Apr 23 21:52 murat
drwxr-xr-x 2 murat cdf 4096 Apr 23 21:52 stntuple

```

```

/data12/murat/tgeant/test/murat:
used 16 available 1627984
drwxr-xr-x 4 murat cdf 4096 Apr 23 21:52 .
drwxr-xr-x 4 murat cdf 4096 Apr 23 21:53 ..
drwxr-xr-x 2 murat cdf 4096 Apr 23 21:52 test1
drwxr-xr-x 2 murat cdf 4096 Apr 23 21:52 test2

```

```

/data12/murat/tgeant/test/stntuple:
used 16 available 1627984
drwxr-xr-x 4 murat cdf 4096 Apr 23 21:53 .
drwxr-xr-x 4 murat cdf 4096 Apr 23 21:53 ..
drwxr-xr-x 2 murat cdf 4096 Apr 23 21:53 dev_238
drwxr-xr-x 2 murat cdf 4096 Apr 23 21:53 dev_239

```

has 4 books: **stntuple/dev\_238**, **stntuple/dev\_239**, **murat/test1** and **murat/test2** defined. The book names should not contain “.” characters in them.

A book may have several datasets described in it, there is no limitations on the syntax of the dataset names except the one above. All the names above should be valid UNIX directory names. A full dataset name is specified by the book and by the dataset ID separated by “:”, for example: **stntuple/dev\_239:bttop0g** or **murat/test1:zee\_new**.

A dataset consists of several filesets, a fileset is a set of files located on the same file-server in the same directory. Different filesets of the same dataset can reside on different fileservers. There is no limitations on the fileset name or number of files in one fileset, other than discussed above. Dataset catalog resides in the directory, which name is defined by the names of the book and the dataset, for example, catalog of the dataset **stntuple/dev\_239:bttop0g** is located in `$(STNTUPLE_CATALOG)/stntuple/dev_239/bttop0g`.

### 12.3 Catalog Files

Dataset catalog consists of several ASCII files residing in the same directory. If dataset has N filesets in it, the dataset catalog consists onf N+1 ASCII files:

- dataset catalog: one file per dataset. It contains fileset-level description of the dataset, its name is fixed by the dataset ID. Fileset-level catalog of dataset ID **bttop0g** is stored in the file named **AAA\_CATALOG.html** . The following example explains format of the dataset catalog file:

```

#-----
fcdldata051: /cdf/scratch/cfdldata/scratch/bttop0g
fileset server name subdirectory
#-----
GI0741 fcdldata051.fnal.gov /cdf/scratch/cfdldata/scratch/bttop0g

```

```

GI0744 fcdldata051.fnal.gov /cdf/scratch/cdfdata/scratch/btop0g
GI0747 fcdldata051.fnal.gov /cdf/scratch/cdfdata/scratch/btop0g
GI0749 fcdldata051.fnal.gov /cdf/scratch/cdfdata/scratch/btop0g
GI0750 fcdldata051.fnal.gov /cdf/scratch/cdfdata/scratch/btop0g
GI0776 fcdldata051.fnal.gov /cdf/scratch/cdfdata/scratch/btop0g
GI0943 fcdldata051.fnal.gov /cdf/scratch/cdfdata/scratch/btop0g
GI0945 fcdldata051.fnal.gov /cdf/scratch/cdfdata/scratch/btop0g
GI0949 fcdldata051.fnal.gov /cdf/scratch/cdfdata/scratch/btop0g
GI1239 fcdldata051.fnal.gov /cdf/scratch/cdfdata/scratch/btop0g
GI1241 fcdldata051.fnal.gov /cdf/scratch/cdfdata/scratch/btop0g
GI1275 fcdldata051.fnal.gov /cdf/scratch/cdfdata/scratch/btop0g
#-----
end of the fileset-level catalog
#-----

```

The dataset catalog has 1 line per fileset, the first column defines the name of the fileset, the 2nd column defined the name of the remote server (may be the name of the local host) and the last column defines the name of the directory where the fileset is located. Lines starting from “#” are the comment lines. Lines starting from “!” (HLML directives) are also ignored - this allows putting in the web references.

- Catalogs of individual filesets - 1 file per fileset. The name of the file should be the same as the name of the fileset. Fileset catalog includes one non-comment line per file and its format is illustrated by the example below:

```

#-----
GI0741.*
#-----
GI0741.0 btop0g.0016.GI0741.0.s.0001 1212517997
GI0741.1 btop0g.0017.GI0741.1.s.0001 1382061686
GI0741.2 btop0g.0018.GI0741.2.s.0001 1170985582
GI0741.3 btop0g.0019.GI0741.3.s.0001 1180158114
GI0741.4 btop0g.0020.GI0741.4.s.0001 1127961129
GI0741.5 btop0g.0021.GI0741.5.s.0001 1074892152
#-----
end
#-----

```

The 1-st column is a string which includes the fileset name, the 2-nd column is the file name, the 3rd column is the file size (not used for the moment).

## 12.4 Remote Data Access

To access the STNTUPLE's stored on different platform over the network one can use ROOTD daemon. For the CDF (read-only) installation of ROOTD one can do it as follows:

```

root[0] TStnAna x('root://fcdfsgi2.fnal.gov/cdf/data05/my_file.stn');

```

Use TFile::Open to open network files interactively from the ROOT prompt. To avoid problems with accessing the remote files include call to TAuthenticate::SetGlobalUser into your **rootlogon.C**. Example:

```
TAuthenticate::SetGlobalUser('cmircea');
```

DRAFT OCTOBER 2, 2004

## 13 Good Run Lists

- by default all runs are good
- For analyses CDF Electroweak Group is using good run list described at [http://www-cdf.fnal.gov/internal/physics/ewk/tools\\_and\\_datasets/good\\_run\\_list.html](http://www-cdf.fnal.gov/internal/physics/ewk/tools_and_datasets/good_run_list.html). It is referred to as EWK (also ETF) good run list. To specify EWK good run list:

```
TStnAna* x = new TStnAna();
x->SetGoodRunList("ETF");
```

- use `TStnAna::SetGoodRunRoutine` to specify your own definition of a good run. Parameter in `TStnAna::SetGoodRunRoutine` call is a function, returning 1 for good runs and 0 for bad ones. Example:

```
int function my_good_run(int RunNumber) {
 if (RunNumber < 154799) return 1;
 else return 0;
}
```

```
TStnAna* x; // created somewhere
x->SetGoodRunRoutine(my_good_run);
```

- here is a brief example of `TStnGoodRunList` usage:

```
TStnGoodRunList grl;
TStnRunSummary* rs;

grl.Init();

int run_number = 138425;

rs = grl.GetRunSummary(run_number);
float lumi = rs->OfflineLumiRS();

// you can also do

rs->Print();
```

- technical note: `TStnGoodRunList::GetRunSummary` returns a pointer to an internally cached structure, so run summary information is available for only one run at a time

### 13.1 More about TStnGoodRunList

```
TStnGoodRunList grl;
TStnRunSummary* rs;

grl.Init();
```

DRAFT OCTOBER 2, 2004

## 14 STNTUPLE Data Blocks

This section contains description of STNTUPLE data structures (data blocks)

DRAFT OCTOBER 2, 2004

## 14.1 TStnHeaderBlock

**TStnHeaderBlock** contains event header.

- **Int\_t TStnHeaderBlock::EventNumber()** : returns event number
- **Int\_t TStnHeaderBlock::RunNumber()** : returns run number
- **Int\_t TStnHeaderBlock::SectionNumber()** : returns runsection number
- **Int\_t TStnHeaderBlock::McFlag()** : 1 for MC events, 0 for the real data
- **Float\_t TStnHeaderBlock::InstLum()** : returns instantaneous luminosity for the event in  $\text{cm}^2 \cdot \text{sec}^{-1}$ , so the returned numbers are of the order of  $10^{31}$

## TOF Data Block

The Time of Flight detector is composed of 216 scintillator bars (bar 0 at  $\phi = 0$ ) with a PMT on each end. PMTs are numbered 0-215 for the east side, and 216-431 for the west (ie. PMTs  $i$  and  $i+216$  correspond to the PMTs on either side of a bar  $i$ ). The time and charge information of each PMT is read out for each event.

The Stntuple's `TofDataBlock` contains 2 accessors and 4 methods to access the TOF information.

- **accessors**

`Short_t GetTofCharge(int pmtNum) const` returns the ADC value (charge) of the hit corresponding to the PMT number passed in `pmtNum`.

`Short_t GetTofTime(int pmtNum) const` returns the TDC value (time) of the hit corresponding to the PMT number passed in `pmtNum`.

- **methods**

`void Clear(Option_t* opt = "")` sets charge and time values to 0.

`void Print(Option_t* opt = "") const` prints charge and time values for each PMT.

`bool IsTofBarHit(int barNum, int chargeThresh=400, int timeThresh=1)` returns true if charge and time of both PMTs on a bar are above the threshold set in `chargeThresh` and `timeThresh`.

`int GetNTofHits(int chargeThresh=400, int timeThresh=1)` returns the number of bars hit in the event. A hit is defined in the same way as the `IsTofBarHit` function.

## 14.2 TStnCcosmicBlock

TStnCcosmicBlock stores output of the cosmic ray finder **CosmicFinderModule**.

Most important part of the information comes from the COT cosmic ray finder described in [1]

Cosmic finder starts from a found high-Pt muon candidate, referred to as a “leg” hereafter, it is assumed that there is not more than 2 such candidates. COT cosmic ray finder is doing 2 additional fits:

- it refits COT track of a muon candidate with floating T0
- it tries to find hits corresponding to the opposite

Below we list the most important accessors.

- **NLegs()**: number of reconstructed muons for which cosmic hypothesis has been tested and information stored in the cosmic block
- **FitDirection(Int\_t Leg, Int\_t FitType)**: fit direction corresponding to the best chi2 of the fit of **Leg**, returned values
  - 0: seed incoming, opposite leg outgoing
  - 1: seed outgoing, opposite leg incoming
  - 2: both legs outgoing (expected case)
  - 3: both legs incoming (unphysical case)

**FitType=0** : individual fit of the track, **FitType=1**: dicosmic fit

- **FitChi2(Int\_t Leg, Int\_t FitType)**: chi2 of the best fit of the cosmic candidate number **Leg** for a given **FitType**
- **FitdChi2(Int\_t Leg, Int\_t FitType)**: difference between the  $\xi^2$ 's of the next-to-best and the best **FitType** fits of the cosmic candidate number **Leg**

### 14.3 TStnElectronBlock

TStnElectronBlock stores information about the reconstructed electron candidates.

### 14.4 TStElectron

- **fDetCode** : 0 for the central EM-objects, 1 for the plug, 2 for the forward (run1, do not exist in Run 2)
- **fNTracks** : number of tracks associated with the EM object by ProductionExe
- **fBcXCes** : X-coordinate of the beam-constrained COT-only electron track, extrapolated to CES. Track = maxPtTrack;
- **fBcZCes** : Z-coordinate of the beam-constrained COT-only track, extrapolated to CES

Standard central electron ID cuts are described in CDF note 6580 [5]. Corresponding cuts implemented in TStnElectronID class.

Standard plug electron ID cuts are described in CDF note 6789 [4]. Corresponding cuts implemented in TStnElectronID class.

Leakage corrections (as of 2002, runs j 153372) for the plug and the central electrons are described in CDF note 6167 [3]

## 14.5 TStnMuonBlock

TStnMuonBlock stores information about the reconstructed muon candidates.

### 14.5.1 TStnMuon

Class **TStnMuon** represents muon candidate reconstructed by the offline muon reconstruction code. This section describes data members and accessors, which meaning is not self-explanatory. Code initializing TStnMuon variables resides in

[Stntuple/mod/InitMuonBlock.cc](#)

Here we describe some of the TStnMuon variables.

- **fCmuChi2Link**:  $\xi^2$  of the track-stub link, calculated by MuonMatchCalculator::calculateMatch, for the moment is based only on coordinates

## 14.6 TStnMetBlock

TStnMetBlock stores variables related to the calculation of the missing  $E_t$ . As there are different ways to calculate met, TStnMetBlock includes several values of MET, calculated under different assumptions. Values of MET[0:2] are precalculated, MET[3:4] can be defined by the user

- TStnMetBlock::Met(0): MET calculated at  $Z=0$
- TStnMetBlock::Met(1): MET calculated at best VXPRIIM vertex (best vertex in “VertexCollection”. If an event contains a high-Pt lepton (electron, muon or tau-candidate), Met is calculated at  $Z_0$  of the lepton’s track
- TStnMetBlock::Met(2): MET calculated at  $Z$  event with
- TStnMetBlock::Met(3): MET calculated at best Beate’s vertex
- TStnMetBlock::MetPhi(i): Phi angle of the 2-D vector of  $\cancel{E}_T$ , corresponding to i-th way of calculating  $\cancel{E}_T$ .
- TStnMetBlock::MetX(i): X-component of the 2-D vector of  $\cancel{E}_T$ , corresponding to i-th way of calculating  $\cancel{E}_T$ .
- TStnMetBlock::MetY(i): Y-component of the 2-D vector of  $\cancel{E}_T$ , corresponding to i-th way of calculating  $\cancel{E}_T$ .
- TStnMetBlock::Sumet(0): Sum  $E_t$  over all the calorimeter towers (currently at  $Z=0$ )
- TStnMetBlock::Sumet(1): Sum  $E_t$  calculated at  $Z$  of the first vertex in VertexBlock or at  $Z_0$  of the high-Pt lepton
- TStnMetBlock::Sumet(2): Sum  $E_t$  calculated at  $Z$  of the highest-Pt vertex in ZVertexBlock
- TStnMetBlock::MetSig(): significance of MET (so far calculated at  $Z=0$ )

## 14.7 TObSPBlock

TObSPBlock (OBSP - name, which roots are in CDF Run I history) stores post-simulation information: MC particles and vertices produced by the detector simulation code. For the same historical reasons TObSPBlock stores FORTRAN-type indices (starting from 1).

**Example 1:** finding MC vertex, corresponding to track Trk, reconstructed in MC event

```
//-----
int find_mc_vertex(TStnTrack* Trk) {
 // note iv-1 in the vertex index calculation

 int iobsp = Trk->ObSPNumber();

 TObSPParticle* p;
 TObSPVertex* v = NULL;

 if (iobsp >= 0) {
 p = fObSPBlock->Particle(iobsp);
 iv = p->VertexNumber();
 v = fObSPBlock->Vertex(iv-1);
 }
 return v;
}
```

## 14.8 TStnPioBlock

TStnPioBlock stores information about the reconstructed  $\pi^0$  candidates.

## 14.9 TStnPio

TStnPio class represents reconstructed  $\pi^0$  candidate.

- **Momentum** : 4-momentum of the pi0 candidate, by default is defined for  $Z=0$ , use `TStnTuple::DefinePi0Momentum` to recalculate momentum for different values of  $Z$ .
- **Z0** : Z-coordinate of the assigned vertex, defines Pi0 3-momentum
- **XCes** : X-coordinate of the `CesMatch` is stored, the sign may depend
- **ZCes** : X-coordinate of the `CesMatch` is stored, the sign may depend
- **Pi0Candidate** : transient-only backward link to mother offline Pi0, defined only at ntuple filling stage.

## 14.10 TStnTauBlock

TStnTauBlock stores information about the reconstructed tau lepton candidates.

## 14.11 TStnTau

TStnTau class describes reconstructed tau candidate. In this section we describe data members and accessor functions of TStnTau.

- **fMomentum** : 4-momentum of the tau candidate: the same as fClusterMomentum
- **fClusterMomentum** : 4-momentum of the tau candidate as defined by the calorimeter cluster and direction of the seed track ????
- **fTrackMomentum** : 4-momentum as defined by the tracks associated with the tau candidate. Pion mass is assigned to all the tracks (?????)
- **fVisMomentum** : 4-momentum as defined by the tracks and reconstructed  $\pi^0$ 's associated with the tau candidate. See [?] for more details on  $\pi^0$  reconstruction algorithm
- **fGenpMomentum** : 4-momentum by the MC generator (not defined for the tau candidates reconstructed in real data)
- **fNTracks** : not defined
- **fNTracks10** : number of tracks in the inner cone used by the tau reconstruction algorithm, which may not necessarily be 10 degrees.
- **fNTracks30** : number of tracks in the outer cone used by the tau reconstruction algorithm, which is not necessarily 30 degrees.
- **fNMuStubs** : number of muon stubs (CMU+CMP+CMX) within delta phi of 15 degrees from the direction of the  $\tau$  candidate
- **fNMuHits** : total number of muon D-hits (???) (CMU+CMP+CMX) within delta phi of 15 degrees from the direction of the  $\tau$  candidate
- **fDteta** : detector eta of the tau candidate - defined by the calorimeter cluster assuming  $Z(\text{vertex}) = 0$
- **fTowerEt[2]** :  $E_t$  's of two highest towers of the tau cluster, calculated at  $Z=?????$
- **fSumTrackP** : scalar sum of momenta of all the tracks associated with the tau candidate (within 10 deg????)

\*\*\*\*\* to be continued

TStnTau has a set of transient-only data members, those are not initialized by default and have to be set in the analysis jobs prior to their use:

- **fSeedTrack** : pointer to the seed track of tau candidate

## 14.12 TStnTrackBlock

Class **TStnTrackBlock** stores tracking information.

Matching between the tracks and the MC particles is done based on hit parentage using output of TrackObspMatch module - CdfTrackMatch object is used. If TrackObspModule has not been run upstream, matching is not done. Note, that TrackObspModule requires track hits to be present in the event record, so it either has to be run in Production, or Production tcl files should be modified to save COT hits - by default they are not saved.

Initialization of the track block is done in

[Stntuple/mod/InitTrackBlock.cc](#)

### 14.12.1 TStnTrack

Class **TStnTrack** stores parameters of the reconstructed track.

- **TStnTrack::NCotHits(Int\_t Isl)**: returns number of track hits in a given COT superlayer **Isl**
- **TStnTrack::NCotAxSeg(Int\_t MinHits)** : returns number of COT axial segments on a track with  $N_{\text{hits}} \geq \text{MinHits}$
- **TStnTrack::NCotStSeg(Int\_t MinHits)** : returns number of COT stereo segments on a track with  $N_{\text{hits}} \geq \text{MinHits}$
- **TStnTrack::Iso4()**: track “relative” isolation -

$$\text{tiso} = \frac{\sum P_t \text{ of other tracks}}{P_t}$$

- **fCMUFid**: = 1 if MuonFiducialTool considers the track to be within the CMU fiducial volume
- **fCMUFid2**: = 1 if  $x < 0$  and  $z < -3$ , where  $x$  and  $z$  are variables returned by the MuonFiducialTool
- **fCMPFid**: = 1 if MuonFiducialTool considers the track to be within the CMP fiducial volume
- **fCMPFid2**: = 1 if  $x < 0$  and  $z < -3$ , where  $x$  and  $z$  are variables returned by the MuonFiducialTool
- **fCMXFid**: = 1 if MuonFiducialTool considers the track to be within the CMP fiducial volume
- **fCMXFid2**: = 1 if  $x < 0$  and  $z < -3$ , where  $x$  and  $z$  are variables returned by the MuonFiducialTool
- **fBMUFid**: = 1 if MuonFiducialTool considers the track to be within the CMP fiducial volume
- **fBMUFid2**: = 1 if  $x < 0$  and  $z < -3$ , where  $x$  and  $z$  are variables returned by the MuonFiducialTool

- Beam constraint: applied to the tracks which have COT parents
- fAlgorithm:
  - bits 0-7 : algorithm
  - bits 8-15: COT parent algorithm
  - bits 16-23:
  - bits 24-31:

For Phoenix tracks:

- fChi2Cot has  $\xi^2$  of the Phoenix seed “energy pull” stored and returned by TStnTrack::PhoenixChi2DofSeed()

### 14.13 TPhoenixElectronsBlock

TPhoenixElectronBlock contains information about the electron candidates, reconstructed by the Phoenix algorithm [2].

DRAFT OCTOBER 2, 2004

## 14.14 SvtdataBlock

The `SvtdataBlock` is a straight-forward translation of `SVTD_StorableBank` into `Stntuple`. Only seven words which are absolutely needed for a valid SVT track are stored for each track in the data block and a number of accessors are provided to work with the `SvtdataBlock` 14.14.1. As a result, `SvtdataBlock` contributes negligibly toward the total event size in `Stntuple` is.

`SvtdataBlock` is not filled in an `Stntuple` by default; to turn on filling the block, the following statement should be added in the tcl file,

```
makeSvt set 1
```

The following example shows how to loop over SVT Tracks,

```
// Assumes SvtdataBlock handle fSvtData is valid
for (int i = 0; i < fSvtData->NSvtTracks(); i++) {
 TSvtTrack *trk = fSvtData->SvtTrack(i);
 Float_t d0 = trk->D0();
 Float_t phi0 = trk->Phi();
 Float_t chi2 = trk->Chi2();
 Float_t curv = trk->Curv();
 Float_t pt = trk->Pt();
 Int_t wedge = trk->Wedge();
 Int_t road = trk->Road();
}
```

It might be noted that XFT tracks are no longer added in `SVTD_StorableBank` and `fSvtData->NXftTracks()` identically returns 0.

### 14.14.1 API Reference

1. Event information accessors:

- **Int\_t NSvtTracks() const** – Number of tracks found by SVT
- **Int\_t NXftTracks() const** – Number of XTRP tracks including invalid ones
- **Int\_t BunchTag() const** – The 8-bit long bunch crossing number
- **Int\_t ParityBit() const** – The parity bit for the event
- **Int\_t GError() const** – The 8-bit long End Event error word
- **Int\_t L1TrigInfo() const** – 3-bit long Level 1 Trigger information from End Event word
- **Int\_t L2Bufld() const** – 3-bit The Level 2 Buffer ID from End Event word
- **Int\_t EEWord() const** – The End Event word
- **void SetEEWord(Int\_t eeword)** – Set the End Event word to `eeword`

2. The SVT track parameter accessors:

- **const Int\_t \*Words()** – Pointer to the SVT word array

- **Int\_t Word(const Int\_t i)** – The i-th SVT word
- **Float\_t D0() const** – Signed impact parameter of the track (cm)
- **Float\_t Curv() const** – Signed curvature of the track ( $\text{cm}^{-1}$ )
- **Float\_t Pt() const** – Transverse momentum of the track calculated from the curvature (GeV)
- **Float\_t Phi() const** – The azimuthal angle of the track (radian)
- **Float\_t Chi2() const** – Chi square of fit by the Track Fitter
- **Int\_t Zin() const** – ID of inner barrel hit by the track (0 – 5)
- **Int\_t Zout() const** – ID of outer barrel hit by the track (0 – 5)
- **Int\_t Wedge() const** – The phi sector traversed by the track (0 – 11)
- **Int\_t Road() const** – Road ID
- **Int\_t XftNumber() const** – The 9 most significant phi bits of the XFT track as delivered to SVT by the XTRP i.e. the XFT wedge number. Since XFT only reports one track per wedge at most, this number uniquely associate the SVT track to the original XFT one.
- **Int\_t TFStatus() const** – Track fitter status word which packs both the Fit quality and the Track Fitter errors
- **Int\_t FitQuality() const** – Quality of track fitting by the Track Fitter
- **Int\_t TFEError() const** – Track Fitter errors which consists of the following 6 bits
- **Int\_t HitOverflow() const** – Bit 0 of TF Error, more than 7 hits in one SS
- **Int\_t LayerOverflow() const** – Bit 1 of TF Error, too many layers with multiple hits
- **Int\_t CombOverflow() const** – Bit 2 of TF Error, too many combinations for fit
- **Int\_t InvalidData() const** – Bit 3 of TF Error, hit out of order, not enough SVT hits
- **Int\_t FitOverflow() const** – Bit 4 of TF Error, fit result in XFT part overflows
- **Int\_t FifoOverflow() const** – Bit 5 of TF Error, FIFO overflow
- **Int\_t ErrorOR() const** – Summary of Track Fitter errors, OR of the above 6
- **Float\_t Hit(const Int\_t layer) const** – Hit coordinates in different AM layers (0..4)(in cm)
- **Int\_t Hit16(const Int\_t layer) const** – Hit Coordinates in different AM layers (0..4) with respect to the wedge border in 16th of a strip) (At present dummy, just the 8 LSBs)
- **Int\_t HitLCF(const Int\_t layer) const** – Long cluster flag in different AM layers (0..4)
- **Int\_t HitEF(const Int\_t layer) const** – Hit existence flag in different AM layers (0..4)

## 14.15 Silicon Blocks

There are several different branches with information related to the silicon detector and tracks with silicon hits on them. As in other `STNTUPLE` branches, the data is organized as an object which inherits from `TObject` and which contains as data members a `TClonesArray` of objects which describe the fundamental unit of interest, such as a silicon cluster, and possible links (see section on `TStnLinkBlock`) to other branches.

The following sections describe the ntuple blocks that contain information on the position of all the wafers, strips, clusters, intersections of tracks with wafers, intersections of `OBSP` particles with wafers, and links between the tracks and silicon hits and intersections. These branches contain enough information to do cluster-level studies, intrinsic resolution studies, and track based studies. Examples of these can be found in the `Stntuple/ana` subdirectory: `TSiPed`, `TRPhiWt`, and `TSiExample`.

First, the fundamental objects used in the `TClonesArray` are described, and then how they fit into the larger block which is stored in a single branch.

### 14.15.1 `TStnSiDigiCode`

This class is used to give every readout unit of the silicon detector a unique identifier. It is based on the CDF class `TrackingObjects/SiData/SiDigiCode`. It is used in most of the higher level silicon data blocks in `STNTUPLE` to distinguish data coming from individual half-ladders.

- Data members
  - `UShort_t fDigiCode`: Unique code for each readout unit (also called a half-ladder). This number is fairly compact, and ranges from 0 for L00, barrel 0, phi wedge 0, to 9135 for ISL layer 7, barrel 2, phi wedge 35. Since it is fairly compact, it can be used directly as an index for arrays.
- Member functions
  - `UInt_t Barrel()`: The barrel of this half-ladder. Ranges from 0-2.
  - `UInt_t LadderSeg()`: The ladder segment. Ranges from 0-2 for L00, and 0-1 for SVX and ISL.
  - `UInt_t PhiWedge()`: Phi wedge. Ranges from 0-12 for L00 and SVX, 0-24 for ISL central barrel 1, 0-28 for ISL forward/backward barrels layer 6, and 0-35 for ISL layer 7.
  - `UInt_t Layer()`: Layer number. 0 for L00, 1-5 for SVX, 6-7 for ISL.
  - `UInt_t Side()`: Axial or stereo side of readout unit. 0 for axial, 1 for 90° or shallow stereo.
  - Setters: There are also setter functions for all the above quantities.

### 14.15.2 `TSiAlign`

This is a simple container `TVector3`'s which represents the center and normal vector of each silicon ladder segment (see Section 14.15.1) and each wafer in the half ladder.

The input files come from running `TrackingUserMods/test/siGeometryValidation.cc` with the option "outputFile" to save an ascii file of the geometry after alignment. Two example ascii files are in the CVS repository<sup>1</sup>.

You can create this object either from an ascii file using the constructor that takes a filename as input, or you can stream it in and out directly since it inherits from `TObject`.

This object defines a simple nested class, `TSiHalfLadder`, which contains the vectors for each wafer.

- Nested class `TSiHalfLadder`
  - `TVector3 h`: Global coordinate (cm) of center of each half ladder
  - `TVector3 nh`: Normal vector of center of each half ladder
  - `int n`: Number of wafers in this half ladder. L00 and SVX half ladders each have two wafers of silicon, whereas ISL half ladders have three.
  - `TVector3 w[3]`: Global position of center of each wafer in the half ladder.
  - `TVector3 nw[3]`: Normal vector of center of each wafer in the half ladder.
- Data members
  - `TMap fMap`: A map of `TSiHalfLadder`'s used as a simple lookup table for the wafer positions.
- Member functions
  - `TVector3* GetCenter(TStnSiDigiCode*)` and `GetNormal(TStnSiDigiCode*)`: Return the global coordinates and normal vector of the center of the half ladder for a given digicode.
  - `int GetNwafer(TStnSiDigiCode*)`: The number of wafers in this half ladder
  - `TVector3* GetWaferCenter(TStnSiDigiCode *digi, int iw)` and `GetWaferNormal(TStnSiDigiCode *digi, int iw)`: Return the center and normal vector of a given wafer for this half ladder.

### 14.15.3 TStnSiStrip

This class contains all the relevant information about an individual strip in the silicon detector. It is used in the ntuple branch `TSiStripBlock` (see Section 14.15.4).

Information about the strip is packed into integers and chars to save space, but is puffed up into human readable form by the streamer. Because of this feature, a `TStnSiStrip` must never be split (use split mode -1) since ROOT's default streamer will not unpack the information. What is described below are the human readable data members which are not persistent.

- Data members
  - `Int_t fStrip`: The strip number. The range of this variable depends on which layer and side it comes from, but the maximum allowed range is 0-895.

---

<sup>1</sup>See `Stntuple/db/si/siGeo_100030_1_GOOD.txt` for the ascii file that represents the silicon alignment used for the winter conferences of 2003.

- `Float_t fADC`: The pedestal subtracted charge in units of ADC counts. The range of allowed charges is -15 to 240.75 in 1/4 ADC count units.
- `Float_t fNoise`: The RMS of the pedestal (saved from the offline database) in ADC counts.
- `Float_t fdNoise`: The RMS of the difference between pedestals of neighboring strips.
- `Int_t fStatus`: 1=good strip, 0=bad strip according to offline database.
- `Int_t fOBSP[3]`: The index into the `TObSPBlock` (see Section ??) of up to three GEANT particles which contributed to the charge on this strip.
- `UShort_t fDigiCode`: The integer used as a datamember for the `TStnSiDigiCode` of the half ladder that this strip lives in (see Section 14.15.1).
- Optional data members
  - \* `bool fStreamGeometryInfo`: If this variable has been set true while making the `TSiStripBlock` (see Section 14.15.4), then the following information is streamed out for each strip. This is useful for debugging purposes and to easily make plots of the positions of individual strips before clustering.
  - \* `TVector3 fGlobal`: Global coordinates of this strip.
  - \* `Float_t fLocal`: Local coordinates. The dead-center of the half ladder is the origin of this coordinate system.

#### 14.15.4 TSiStripBlock

This is a collection of all silicon strips in the event. The `TStnSiStrip`'s (see Section 14.15.3) are stored in the flat `TClonesArray` in blocks of the same digicode (half ladder). If you want fast access to all strips in a given half ladder, you should call `TSiStripBlock::InitEvent()` to initialize the lookup tables. Then you can use `fIndexFirstHit[]` and `fIndexLastHit[]` to loop over all the strips on a half ladder.

In addition to the strips in the `TClonesArray`, the backend state and time since the previous L1 accept for each half ladder is also stored. There is also the option to turn on the streaming of the detailed geometry information of every strip (global and local coordinates). This makes the ntuple quite large, but can be useful for debugging and making plots. To turn this feature on, use the following talk-to in `StntupleMaker`: `makeSiStrips set 2`.

**When filling this block, be sure that the `StorableRun2SiStripSet` is puffed in the `PuffModule`, or that you are rerunning clustering yourself. Otherwise, there will be no strips in the event record and this branch in your ntuple will be empty.**

- Data members
  - `Int_t fNSiStrips`: Number of `TStnSiStrip`'s in the block.
  - `TClonesArray* fSiStripList`: Array of all strips.
  - `Int_t *fIndexFirstHit`: Array used as a lookup table for the index into the `TClonesArray` for the first strip on this ladder. If this half ladder has no strips, then the index is -1.

- `Int_t *fIndexLastHit`: Like previous array, but gives the index of the last hit on the half ladder. If this half ladder has no strips, then the index is -2.
  - `Int_t fNDigiCodes`: Number of half ladders with strips.
  - `Short_t *fBESState`: Backend state of this half ladder.
  - `Short_t *fDtL1A`: Time since previous L1 accept for this half ladder.
  - The other data members are streamed out, but are not generally accessed by the user. Instead they get puffed up into the lookup tables above by a call to `InitEvent()`.
- Member functions
- `Int_t InitEvent()`: Puffs up lookup tables for this event. If you aren't going to use the lookup tables, don't call this and it will save you some time.
  - `TStnSiStrip* SiStrip(int i)`: Access to the strip in `fSiStripList` with index `i`.
  - `Int_t FindStrip(int digi, int stripnum)`: Returns the index of the strip in `fSiStripList` with a given digicode and strip number. This is used primarily when filling the ntuple.

#### 14.15.5 TStnSiHit

This class contains all the relevant information about a silicon cluster (also called a hit). It is a pared down version of the CDF class `TrackingObjects/SiData/SiHit`. For a link to the `TStnSiStrip`'s that make up this hit, see the link block in `TSvxDataBlock` in Section [14.15.6](#).

- Data members
  - `TVector3 fGlobal`: Position of the hit in global coordinates. Since these are two-dimensional hits, the third coordinate is taken from the center of the half-ladder. In the case of hits that were used in a track, the global position is corrected for wafer level alignments based on the 3D track intersection with the wafer and the third dimension is taken from this intersection point.
  - `Float_t fStripNum`: Position of hit on half ladder in units of strip number.
  - `Float_t fLocal`: Same as previous local coordinate, but in units of centimeters. The origin is the center of the half ladder.
  - `Float_t fQtotal`: Total (pedestal subtracted) charge of cluster in ADC counts.
  - `Float_t fNoise`: The sum in quadrature of the noise of the strips making up the cluster.
  - `Char_t fStatus`: Bit field. Bit 0=has bad strips, bit 1=neighbors are bad. Use the member functions to test if this is a good strip.
  - `TStnSiDigiCode fDigiCode`: Digicode of the ladder that this hit lives on.
  - `Int_t fUniqueID`: Reuse this data member of `TObject` to store information on which Monte Carlo particles contributed to the charge of the hit. Use member functions to get at this.

- Member functions
  - `Int_t NObsp()`: Number of Monte Carlo particles that produced this hit.
  - `Int_t Obsp(int i)`: Index into the `TObspBlock` (see Section ??) of the *i*-th particle contributing to this hit.
  - `Bool_t Good()`: This is a good hit.
  - `Bool_t HasBadNeighbor()`: This cluster contains a strip that is next to a bad strip and should be considered suspect.

#### 14.15.6 TSvxDataBlock

This branch contains an array of all the silicon hits (`TStnSiHit`'s, see Section 14.15.5) in the event. It also has a link block to access all the strips in the `TSiStripBlock` (see Section 14.15.4) that contribute to a given hit. There are optional lookup tables which are puffed with a call to `TSvxDataBlock::InitEvent()` which enable the user to have easy access to all hits in a given half ladder (digicode).

There are also links from the hits to tracks, but this information is stored in a different block (see `TStnTrackLinkBlock` in Section 14.15.11).

- Data members
  - `Int_t fNSiHits`: Total number of `TStnSiHit`'s in this event.
  - `TClonesArray* fSiHitList`: Array of hits stored in blocks of the same digicode.
  - `Int_t* fIndexFirstHit`: Lookup table of index of first hit in a given digicode. This is created by a call to `InitEvent()`. If there is no hit in the given digicode, the index is -1.
  - `Int_t* fIndexLastHit`: Same as previous lookup table, but has the index of the last hit on the half ladder. Index is -2 if no hits on this digicode.
  - `TStnLinkBlock fSiStripLinkHit`: Link block of indexes into the `TSiStripBlock` which allows the user to have access to all strips used to create a given hit. See the example in `ana/TSiExample.cc` for a use case.
- Member functions
  - `Int_t InitEvent()`: Puffs up lookup tables for this event. If you aren't going to use the lookup tables, don't call this and it will save you some time.
  - `TStnSiHit* SiHit(int i)`: Access to the hit with index *i* in the array `fSiHitList`.
  - `Int_t IndexFirstHit()`: There are two forms of this function. One if you want to get the index of the first hit given a barrel, ladder segment, phi wedge, layer and side, and the other if you know the digicode already. This needs to have `InitEvent()` called beforehand.
  - `Int_t IndexLastHit()`: Same as previous function, but returns the index of the last hit on the half ladder.

### 14.15.7 TStnSiIsect

This object contains some information about a track intersection with a particular silicon half ladder. It can be useful for making residuals with silicon hits.

These objects are stored in the array in the `TSiIsectBlock` of Section 14.15.8. If you are looping over tracks in the track block, then you can use the links in `TStnTrackLinkBlock` (Section 14.15.11) to access these intersections.

– Data members

- `TStnSiDigiCode fDigiCode`: The digicode of the half ladder that this track intersection is coming from.
- `TVector3 fGlobal`: Global coordinates (corrected for wafer-level alignment) of track intersection with this half ladder.
- `Float_t fStripNumPhi`: Intersection point in strip units on the axial side.
- `Float_t fStripNumZ`: Intersection point in strip units on the stereo side.
- `Float_t fLocY` and `Float_t fLocZ`: Same as previous local coordinates, but in centimeters.

|                                        | Bit | If set, then passed thru active |
|----------------------------------------|-----|---------------------------------|
|                                        | 0   | phi area of axial side          |
| – <code>TBitset fActiveRegion</code> : | 1   | z area of axial side            |
|                                        | 2   | z area of stereo side           |
|                                        | 3   | phi area of stereo side         |

### 14.15.8 TSiIsectBlock

This branch contains a collection of the intersections of tracks with silicon half ladders. If you are looping over tracks, then you can use the correspondance object, `TStnTrackLinkBlock` (see Section 14.15.11), to get access to all intersections for a given track. You also have immediate access to all silicon hits for this track for comparison using that link block.

– Data members

- `Int_t fNSiIsects`: Total number of track-silicon intersections in this event.
- `TClonesArray* fSiIsectList`: Array of `TStnSiIsect`'s.

– Member functions

- `TStnSiIsect* SiIsect(int i)`: Access to the i-th intersection in the array `fSiIsectList`.

### 14.15.9 TStnSiGeantIsect

Similar to the track intersection object, `TStnSiIsect`, this object contains information of Monte Carlo particle intersections with silicon half ladders. It stores where the particle went and how much energy it lost while traversing the half ladder.

This object is a pared down version of the CDF class `PropagatedSiParticle`.

– Data members

- `TStnSiDigiCode fDigiCode`: The digicode of the half ladder that this Monte Carlo particle traversed.
- `Int_t fUniqueID`: Reuse this data member from `TObject` to store the index into the `TObsepBlock` of the particle that hit this ladder.
- `TLorentzVector fEntryMomentum`: 4-momentum of the particle at the point of entrance to the half ladder.
- `TLorentzVector fExitMomentum`: 4-momentum upon exit.
- `TVector3 fEntry` and `fExit`: The entry and exit points in global coordinates.

| Bit | If set, then passed thru active |
|-----|---------------------------------|
| 0   | Entrance in Phi area of side 0  |
| 1   | Entrance in Z area of side 0    |
| 2   | Entrance in Z area of side 1    |
| 4   | Entrance in Phi area of side 1  |
| 5   | Exit in Phi area of side 0      |
| 6   | Exit in Z area of side 0        |
| 7   | Exit in Z area of side 1        |
| 8   | Exit in Phi area of side 1      |

- `Float_t fDE`: Total energy loss in GeV in this half-ladder.
- `Float_t fRadLen`: Fraction of radiation length traversed.
- `Float_t fDistToNearPhi`: Distance from (exit-entrance)/2 to center of nearest strip. The Hall Effect should be visible in charge deposition models that include it.
- `Float_t fDistToNearZ`: Same as previous, but for strips on stereo side.
- `Int_t fInitPhi`: Initial hit phi strip.
- `Int_t fFinalPhi`: Final hit phi strip.
- `Int_t fInitZ` and `fFinalZ`: Same as previous, but stereo side.

#### 14.15.10 `TSiGeantIsectBlock`

This branch contains all the intersections of Monte Carlo particles (`TStnSiGeantIsect` of Section 14.15.9) with the silicon detector elements. It also has a link block to allow the user to have access to the particle in the `TObsepBlock` that made the intersection.

– Data members

- `Int_t fNSiGeantIsect`: Total number of `TStnSiGeantIsect`'s in this event.
- `TClonesArray* fSiGeantIsectList`: Array of intersections.
- `TStnLinkBlock fIsectLinkObsep`: Link block from OBSP index to the number of intersections that it made. This is useful if you are looping over particles in the `TObsepBlock` and you want access to the intersections in `fSiGeantIsectList`. This link block is puffed by a call to `TSiGeantIsectBlock::InitEvent()`.

- Member functions:
  - `Int_t InitEvent()`: Puffs the link block `fIsectLinkObsp` for this event.
  - `TStnSiGeantIsect* SiGeantIsect(int i)`: Access to the *i*-th intersection.

#### 14.15.11 TStnTrackLinkBlock

This is a link block which stores the correspondence between tracks and their silicon hits and intersections with the silicon detector. This branch is especially useful if you are looping over tracks in the track block, `TStnTrackBlock`, and you want access to the above information.

- Data members
  - `TStnLinkBlock fSiHitLinkTrk`: Links from tracks to silicon hits in the `TSvxDataBlock` (see Section 14.15.6).
  - `TStnLinkBlock fSiIsectLinkTrk`: Links from tracks to the track intersections in the `TSiIsectBlock` (see Section 14.15.8).

## 15 MC objects in STNTUPLE

MC information can be represented at 2 different levels - output of the MC event generators and output of the detector simulation. Correspondingly, STNTUPLE includes 2 different MC blocks:

- TGenpBlock, which contains generator-level information
- TObsepBlock, containing data corresponding to the output fo detector simulation

### 15.1 TGenpBlock

### 15.2 TObsepBlock

### 15.3 Looping over GENP particles

- The following example shows how to loop over the particles stored in GENP block

```
for (int i=0; i<fGenpBlock->NParticles(); i++) {
 TGenParticle* p = fGenpBlock->Particle(i);
 int im = p->GetFirstMother();
 if (im >= 0) {
 TGenParticle* mom = fGenpBlock->Particle(im);
 if (mom!=0) {
 int mom_id = mom->GetPdgCode();
 // do something with it

 }
 }
}
```

Note, that for incoming particles the following line

```
int im = p->GetFirstMother();
```

sets **im** to -1, which means that these particles do not have a mother. Therefore it is necessary to check that **im** is not equal to -1.

## 16 STNTUPLE Utilities

This section describes additional STNTUPLE utilities

### 16.1 Merging the histogram files

- The following routine

```
merge_stn_hist(const char* List, const char* OutputFile)
```

merges 2 or more input histogram files into the output file which has the same structure as the input ones.

- Location: libStntuple\_val.so
- Parameters:
  - List : list of STNTUPLE histogram files, for example, "a/\*.root"
  - OutputFile: name of the output histogram file

## 16.2 Validation tools

This section describes STNTUPLE validation tools. Two most typical validation tasks are

- **code validation:** a data file has been processed with some old “reference” version of the offline code. The same data file has been reprocessed with the newer version of the reconstruction code. How is it possible to find and quantify the differences between 2 different versions of the offline code?
- **data validation:** 2 different data files corresponding to the runs taken under different conditions have been processed with the same version of the reconstruction code. How it is possible to find and quantify the differences between the data?

Steps:

- run 2 STNTUPLE jobs on 2 files in question and for each job save resulting histograms into the corresponding histogram file using TStnAna::SaveHist. Suppose the names of the 2 files are **old.root** and **new.root** correspondingly.
- build validation library **libStntuple\_val.so**:

```
make Stntuple._val
```

- start interactive ROOT session, load **libStntuple\_val.so**, then do:

```
root[0] .L shlib/$BFARCH/libStntuple_val.so
root[1] compare_stn_hist("old.root","new.root",min_prob)
```

where **min\_prob** is the probability of KS test below which 2 histograms will be considered different. In the end **compare\_stn\_hist** pops up a TBrowser window and displays histograms with comparison of which has KS probability below **min\_prob**

- go to the “root” folder (double click on it with the left button) using left mouse button
- double click on **STNTUPLE\_RESULTS**. Icons with the red dot mark modules for which 2 files have different histograms continue clicking with the left button, until you get the histogram names displayed.
- click on a histogram with the right button, you get context menu
- click on “DrawEP”, see histograms from the 2 files overlaid on top of each other. In the shell window where you started ROOT, you’ll get number of entries and the probability of KS test printed.

## 17 Debugging STNTUPLE analysis job

### 17.1 Using GDB

To debug STNTUPLE analysis job:

- start emacs session, type 'Alt-X gdb'
- at EMACS prompt give the location of GDB binary, normally typing 'gdb' is enough
- type 'file root.exe' at gdb command prompt
- staying in the same window you run emacs in, start your analysis job, for example:

```
root.exe my_ana.C
```

In case you're using standard STNTUPLE .rootrc file, on startup it will print the process ID for the process to be debugged

- type 'attach PID' in GDB session, where PID is the ID of the process to be debugged, let GDB to load in the necessary shared libraries
- type 'continue' in GDB window, the debugged exec will resume
- read GDB manual for mode details

### 17.2 Diagnosing common mistakes

- **TStnAna class is not recognised by the interpreter:**

```
root[0] .x Stntuple/ana/cal_ana.C
Error: No symbol TStnAnax("results/productionMartout.root") in current
scope FILE:Stntuple/ana/cal_ana.C LINE:6
```

make sure you have the definitions of this class loaded, command

```
root[0] .class TStnAna
```

should report the list of data members and methods for this class. In case it reports an error, go back to STNTUPLE primer #1 and follow the steps there carefully to see which one you've missed

## References

- [1] Heather K. Gerberich, Ashutosh V. Kotwal and Chris Hays, Cosmic Ray Tagging using COT Hit Timing, [CDF note 6079](#)
- [2] T.Nelson, F.Snider, D.Stuart, Forward Electron Tracking with the PhoenixMods package, [CDF note 6278](#)
- [3] [CDF note 6167](#)
- [4]
- [5] [CDF note 6580](#)