Reconstruction of ZDC Pre-Processor Data and its timing Calibration

Soumya Mohapatra, Andrei Poblaguev and Sebastian White

Aug.8,2010

This note (ATL-COM-LUM-2010-027) documents the ATLAS ZDC waveform reconstruction method and the time calibration method which is used here on a series of runs with LED calibration pulses delayed in a series of 1 nsec steps using the ATLAS PHOS4[1] delay chip. The PHOS4 delay chip is widely used in ATLAS including in the Level1 Calo Pre-Processor(PPM) waveform digitizer that is used by ZDC to record data. This document will be updated whenever changes occur in this analysis framework so that users will always have a current working example available for submitting jobs.

We show first results from timing spectra in recent ATLAS collision data. Inner tracker data have been used to show the existence of satellite bunches which are spaced 2.5 nsec from the main bunch- ie 1/10 th LHC storage RF period. ZDC measurements should be extremely well suited for studying such things since the timing resolution is much better that 1 nsec and the algorithm discussed below allows reconstruction of satellites out to several 10s of meters with uniform efficiency.

The time spectra of each beam can be measured separately by plotting the reconstructed signal time of an individual module in the ZDC (ie HD0A and HD0C, corresponding to hadronic channel 0 in the side A and C ZDC calorimeters, in our examples) using an arbitrary collection of ATLAS trigger data. Distributions in the time difference and time average of these quantities are also of interest. We have implemented, in a status code, rules for selecting events to give the highest possible timing precision based on our experience with ATLAS data. Within the next week we expect to complete studies with a method that combines all 4 ZDC towers to give a time resolution of possibly $\sigma_t \to 100$ picoseconds.

Below we give a sample job preparation script that can be used by anyone in ATLAS to produce similar results. For specific questions on running this code please contact Soumya Mohapatra (smohapatra@gmail.com) who is at

CERN.

1 Introduction

All ZDC data, except for trigger information, consists exclusively of PPPM output waveforms. PHOS4 steps are assumed to be calibrated but, if calibration data are available, they could be used to correct the vector "delay". Some of our studies suggest small differences in timing behavior-such as waveform shape- between physics and LED data but, for the moment, the best calibration we have is from the LED delay scans. It is difficult to do timing calibration directly with physics data since they are concentrated in a narrow time region. This may be possible by varying the clock phase to the PPM. Also data with 2.5 nsec satellite bunches could be extremely useful for calibration with collision data.

The algorithm we have adopted for time reconstruction was chosen to give a consistent method over the full available time range. The ZDC data can be used to obtain 80 MHz sampling of the ZDC waveform, which is approximately 100 nsec wide due to the 320m long ethernet ("DRAKA") cables we used to bring signals from the tunnel. At the present time we are only working with 40 MHz samples but, as we have shown in the ZDC performance plots ATL-COM-LUM-2010-022[4] , this already gives $\sim$210 psec resolution/PMT with an algorithm designed for t$\sim$0. The purpose of the present algorithm is to extend this performance to all t.

2 Nyquist-Shannon sampling method

For a bandlimited analog signal with sparse sampling there are standard practices in electronics based on the Nyquist-Shannon sampling theorem. This was originally discussed in Claude Shannon's paper "Communication in the Presence of Noise"[2], which is a classic in information theory. Shannon derives an optimum reconstruction algorithm based on the Sinc function[3], where Sinc[x]=Sin[x]/x and->1 as x->0. This function is scaled to give roots at all other sampled points (assuming equal spacing) and each sampled point contributes a Sinc function with an amplitude equal to the sample and centered on that point to the series representation of the reconstructed waveform. Tektronix digital scopes use this algorithm on-chip to produce high resolution waveforms. Shannon's sampling theorem states

that once the step size reaches 1/(2f), where f is the highest frequency component no further information is gained by finer sampling. ZDC data, even at 80 MHz are not quite at this limit. The following timing algorithm is based on Shannon's method. The PPM data are digitized in time slices which, to the accuracy of the ATLAS clock distribution, have an interval of 25 nsec. Currently 7 time slices are read out but since we used the first slice for pedestal determination, the following discussion uses 6 slices. The code runs also on a smaller number of time slices to take into account future plans for ATLAS running.

$$time = 0, 25, .., (nslices - 1) * 25 \text{ nanoseconds} \tag{1}$$

We use a data set from PHOS4 delay scans with the following delay steps for calibration.

$$delay = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23 \text{ nsec} \tag{2}$$

To develop this algorithm and calibrate it we moved the PHOS4 delay scan data to a private area.

$$slices = Import[DelayScan.dat] \tag{3}$$

PPM time slices are then assigned to a range of PPM time bins starting from 0 in steps of 25 nanoseconds. We analyzed the 22 delay scan runs given above.

$$timeslices[[i, j, 1]] = time[[j]] \tag{4}$$

$$timeslices[[i, j, 2]] = slices[[i, j]], (j, 1, 6), (i, 1, 22) \tag{5}$$

From the raw PPM data and even approximations up to interpolation order 3 it is hard to see a clear trend in the delay steps. A gif animation of calibration data in raw form and using simple polynomial interpolation can be found at:

http://www.phenix.bnl.gov/phenix/WWW/publish/swhite/PPMData.gif

The Sinc function has roots at n*π so we scale it to give roots at neighboring sample times.

$$shannon[t] = \sum_{i=1}^{nslice} slice[i] \times Sinc[\pi \times (t - time(i))/25)] \tag{6}$$

An animated gif can be found at:

http://www.phenix.bnl.gov/phenix/WWW/publish/swhite/ShannonFilm.gif

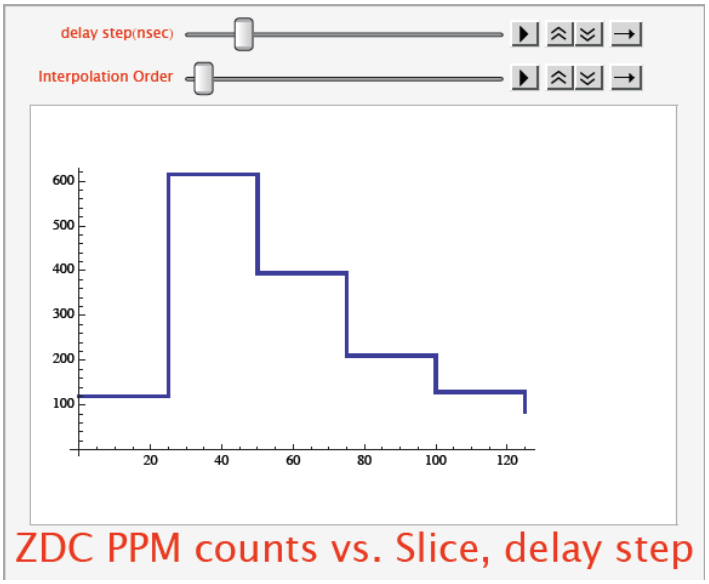


Figure 1: A typical waveform produced by the ZDC L1Calo pre-processor modules from the delay=3 nsec run.

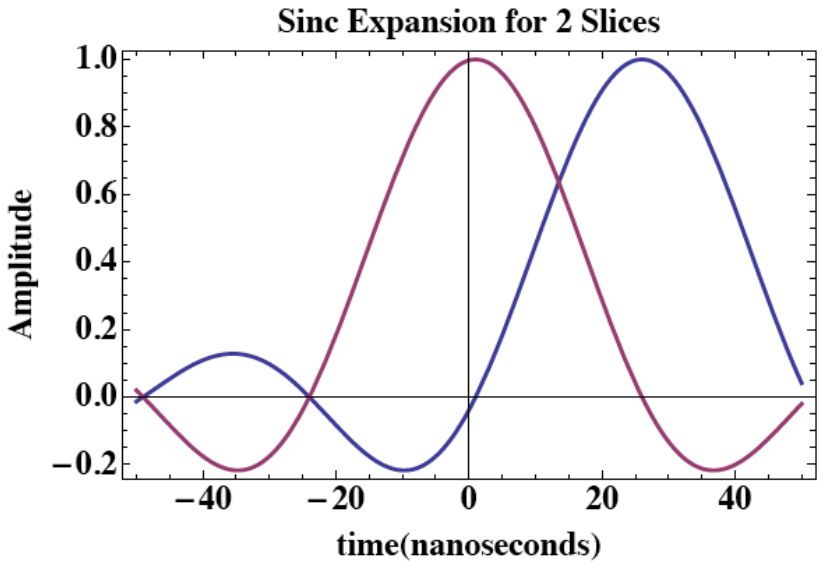


Figure 2: Example of Shannon reconstruction terms for a sampled waveform with Amplitude=1 samples at 0, 25 nanoseconds.

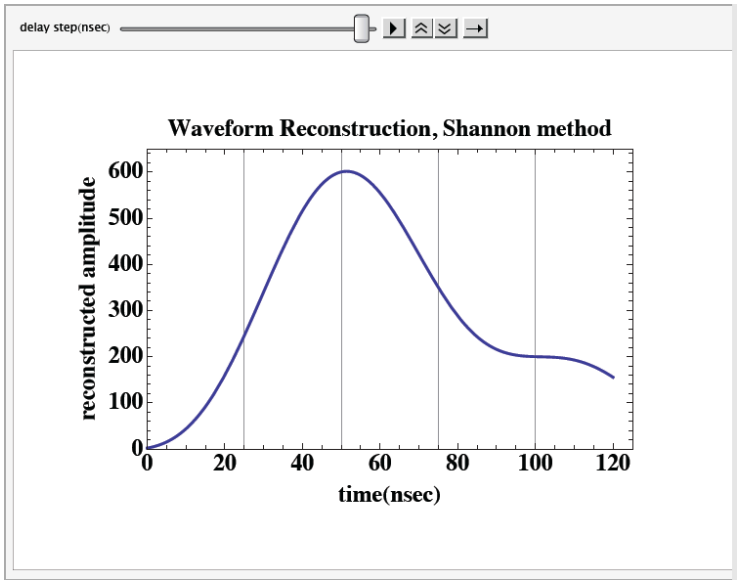


Figure 3: Reconstruction of a typical ZDC waveform.

We find time and amplitude at the Maximum of the interpolation function. NMaximize and NIntegrate are Mathematica functions that we implement in C++. Both the peak and the integral of the function give stable measurement of the energy deposit in a ZDC module - better than using 1 or 2 slices. In the ZDC signal reconstruction we describe below the default Energy method uses the peak of the shannon function.

$$a = NMaximize[shannon[t], 10 < t < 60, WorkingPrecision \to 20] \quad (7)$$

$$peakcts[[i]] = a[[1]]; peakint[[i]] = NIntegrate[shannon[t], (t, 0 \to 120]/60 \quad (8)$$

$$a2[[i]] = slices[[i, 2]]; tpeak[[i]] = t/.a[[2]] \quad (9)$$

We then plot actual PHOS4 delay time vs. the time obtained from this signal reconstruction method. There is a clear non - linearity in the interval $0 \to 25$ nsec. Without a correction one makes an error in time scale of $\sim$1.5 in the region where most physics data are concentrated.

In the following we find a piecewise function which maps PPM reconstructed time to "true" time based on the PHOS4 delays. We fit overlapping time delay regions of $0 \to 9$, $9 \to 17$ and $16 \to 22$ nanoseconds using a 3rd order polynomial function.

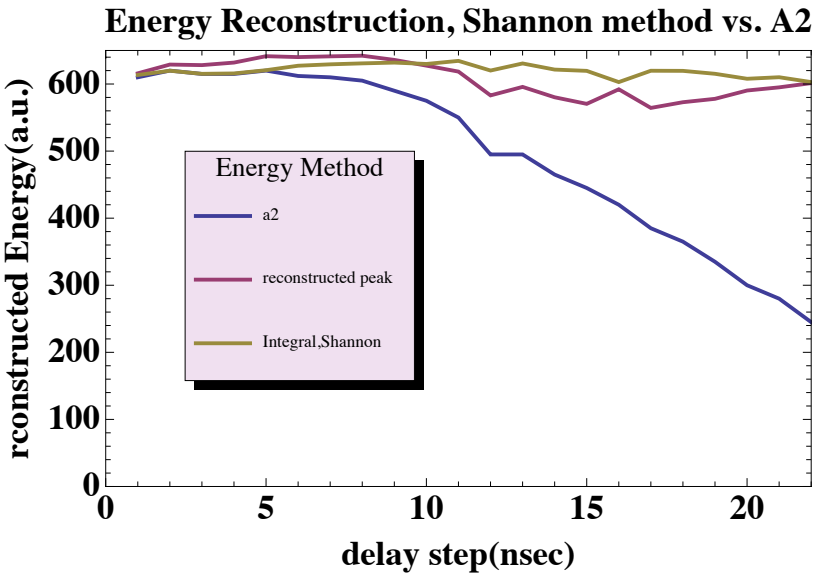


Figure 4: Reconstructed Energy for fixed amplitude light emitting diode flasher calibration data vs. delay step.

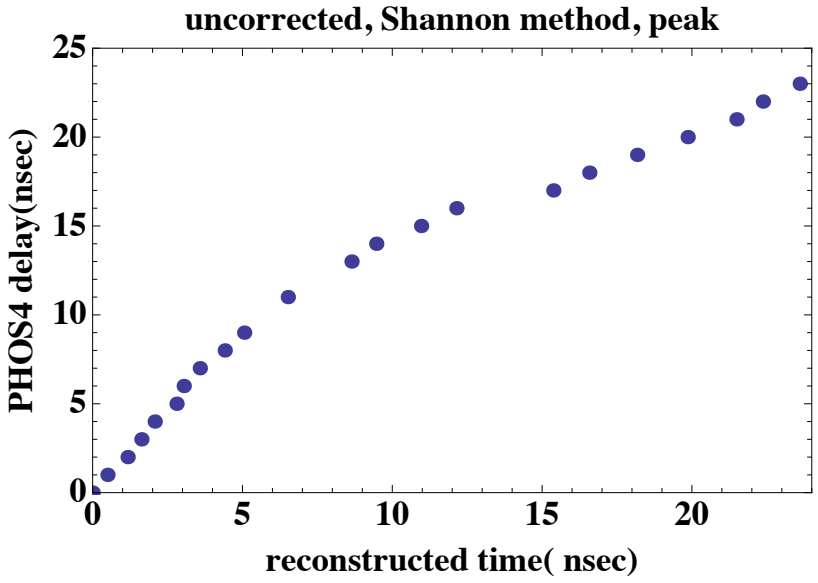
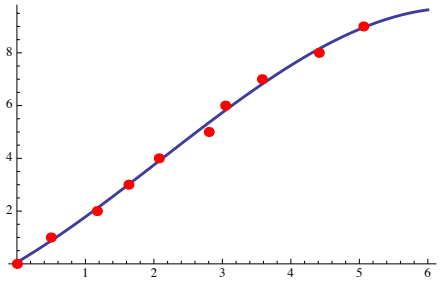
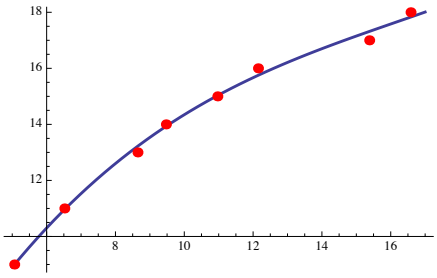


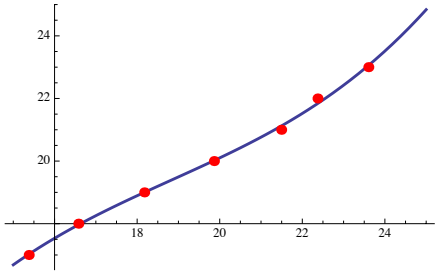
Figure 5: Reconstructed time for fixed amplitude light emitting diode flasher calibration data vs. delay step. There is a clear non-linearity when it is plotted vs. delay step.



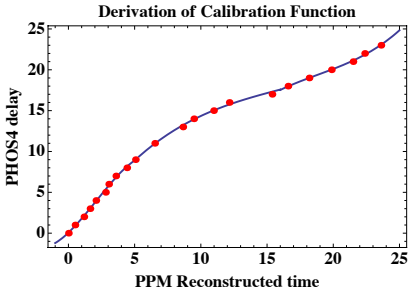
(a) Fit from 0 to 9 manoseconds.



(b) 9 to 17 nsec



(c) 16 to 22 nanosecond



(d) Piecewise fit to the full range.

Figure 6: Derivation of the correction function.

The derivative at t=0 is 1.5. Therefore an uncalibrated algorithm, like the one used in Fig.8 of ATL-COM-LUM-2010-022, will have the wrong time scale by this factor.

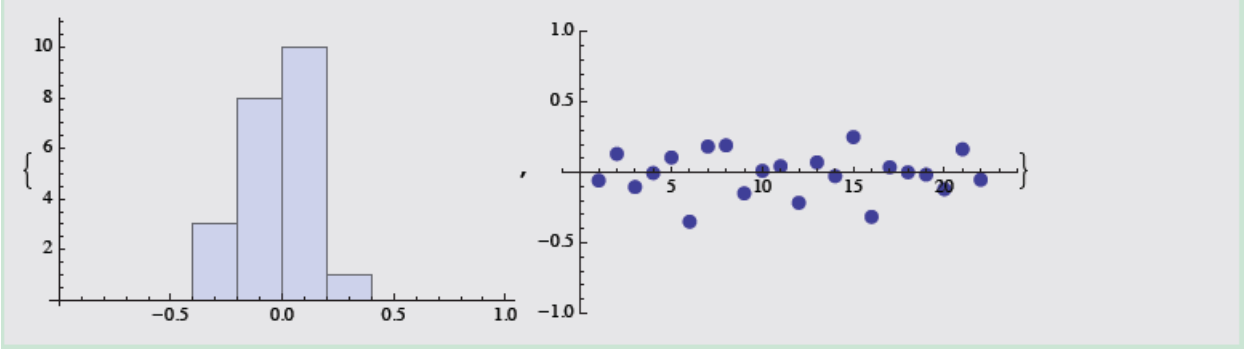


Figure 7: These fits result in reasonable residuals in our low statistics sample.

The output of our calibration program is the following function which should be used to correct PPM time, reconstructed by the above method, for physics data. It gives small residuals which are partly due to the low statistics used in this study. It represents the best knowledge we have of ZDC time calibration. The calibration could be run separately on all channels.

Since PPM digitization is periodic with a cycle of 25 nanoseconds the correction is phased to cover the regions of $t < 0$ and $t > 25$ nanoseconds also.

CalibrationFunction[t]=Piecewise[(Pfit1[t],$t < 5$),(Pfit2[t],$t > 5\&\&t < 16\}$, (Pfit3)[t],$t > 16$)] (10)

CalibrationFunction[*t*]

$$\begin{cases} 0.05683672111 + 1.53276381279t + 0.229808343735056t^2 - 0.0365636647119192t^3 & t < 5 \\ -1.2897493497 + 2.65357869933t - 0.137124014614t^2 + 0.00281211806638t^3 & t > 5\&\&t < 16 \\ -42.1868874032 + 8.6175205570t - 0.425923980606t^2 + 0.0075384950748t^3 & t > 16 \end{cases}$$

3 Addendum: Leading Edge Timing

We used the same methods for leading edge constant fraction algorithms. Below we show the behavior with constant fraction parameters of 20,50 and 75%. All require similar non-linearity corrections, as seen below. On a Macintosh G5 running *Mathematica* 7.0, with working precision of 20 the

leading edge algorithms (using FindRoot) run 10 times faster (9 milliseconds) than the peak algorithm (which uses NMaximize). The C++ code below will run faster.

$$shsub[t] = shannon[t] - peakcts[[i]] \times constantfraction \tag{11}$$

$$\text{b=FindRoot[shsub[t],(t,10),WorkingPrecision}\rightarrow 20]; \tag{12}$$

$$thalf[[i]] = t/.b[[1]] \tag{13}$$

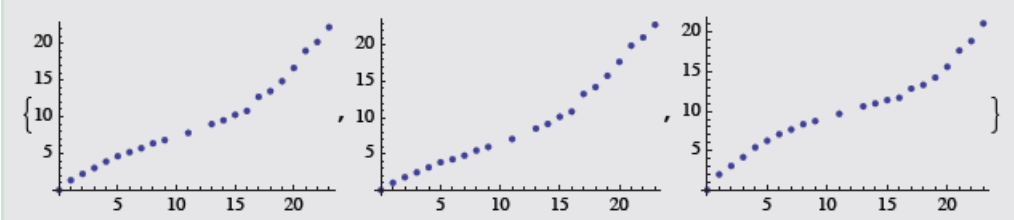


Figure 8: Delay step vs. Reconstructed time with a constant fraction method, fraction=$20, 50, 75\%$.

4 Validation of the code implementation in the ATLAS offline framework.

The following is from a collection of plots we are preparing validating the code currently checked in to SVN and also available by navigating:

https://twiki.cern.ch/twiki/bin/view/Atlas/ZeroDegreeCalorimeter

but is to always be found at:

`/afs/cern.ch/user/s/soumya/public`

5 Studies of beam structure with offline data.

These are sample plots from the sample job given below. The resolution out of the box is better than 1 nsec. Since with Andrei's optimum event selection he finds $\sigma_t \simeq 200$ picoseconds per PMT, we expect to update this example with a new package giving this in the next day.

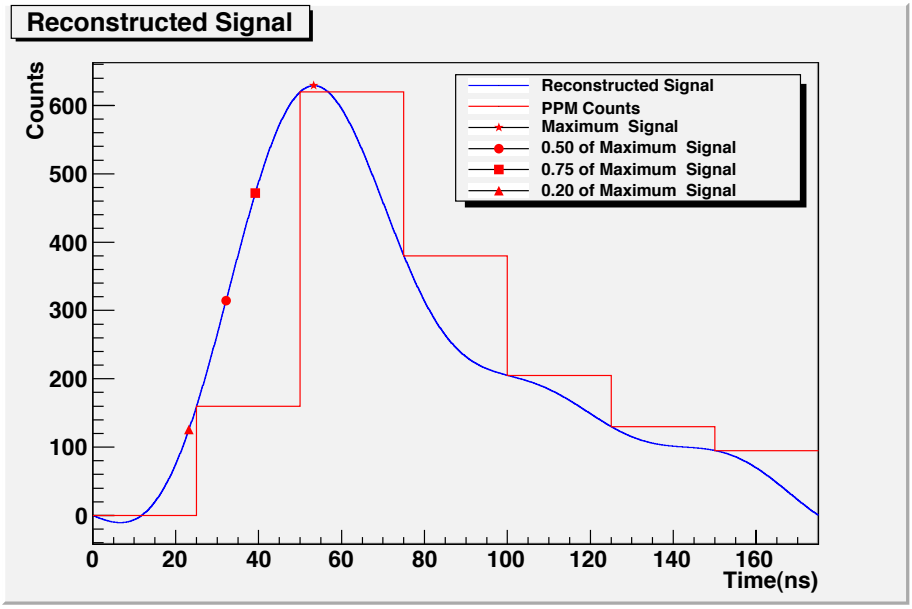


Figure 9: Signal reconstruction and time found by several optional methods available to the user.

6 Plans

The main emphasis in the next week will be to do resolution with both the single PMT algorithm and one using all PMTs. Also we will try using collision data to further tune the timing calibration.

References

[1] see N. Massol, ATLAS Calibration Delay Chip Study, ATL-ELEC-2004-001 and references therein.

[2] "Communication in the Presence of Noise",Proc. Institute of Radio Engineers, vol. 37, no. 1, pp. 10−−21, Jan. 1949. Reprinted as classic paper in: Proc. IEEE, vol. 86, no. 2, (Feb. 1998).

[3] http://reference.wolfram.com/mathematica/ref/Sinc.html

[4] ATL-COM-LUM-2010-022

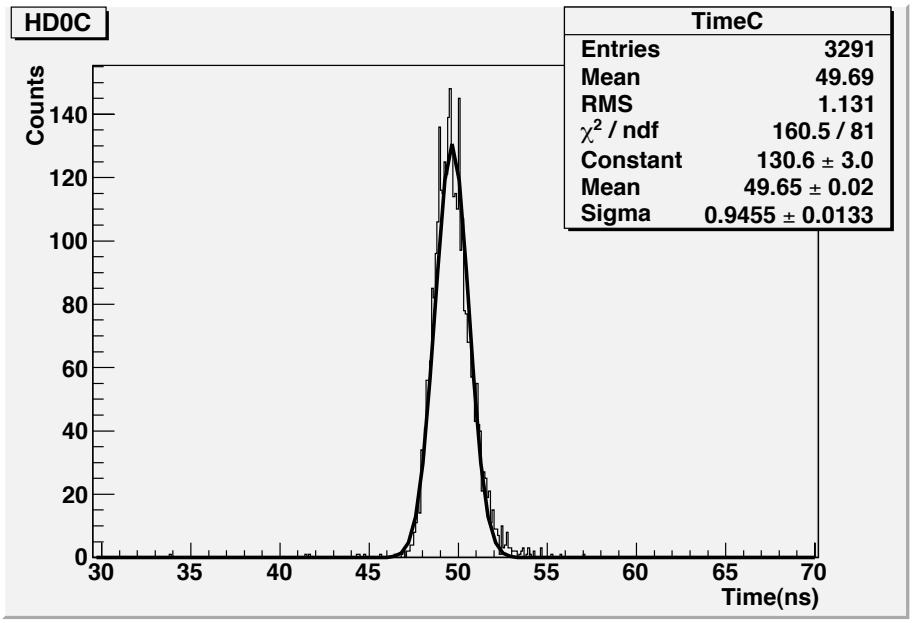


Figure 10: Reconstructed time distribution from ZDC-C PMT1 in collision data. Note there are advantages in using, instead, a time difference distribution (ie ZDC-C minus ZDC-A). This is an earlier version of the timing tune but note already the excellent fit to a gaussian.

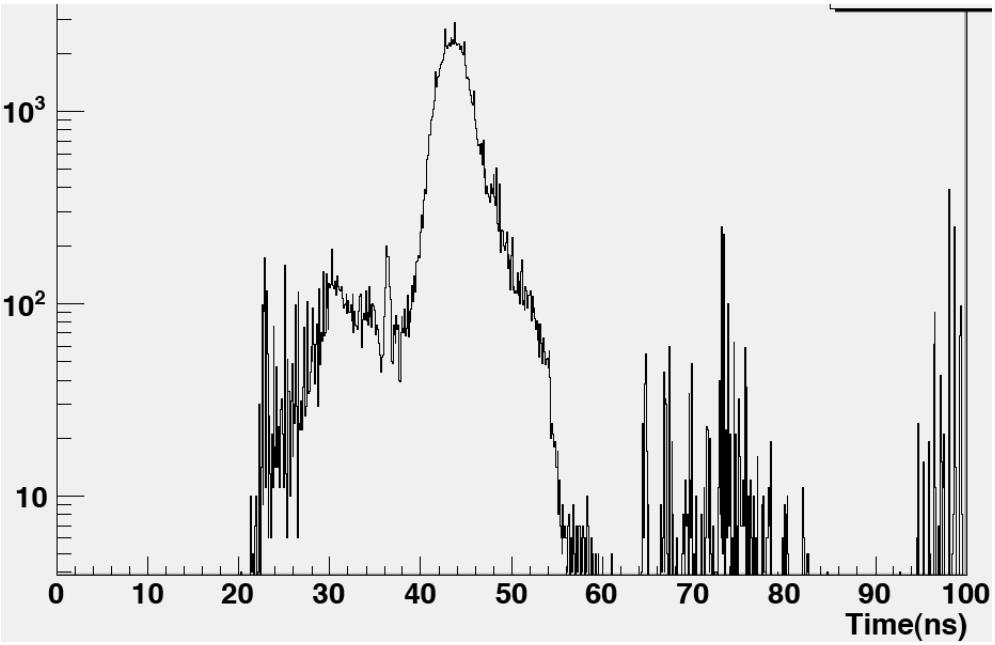


Figure 11: Sample output from the job example given below(HD0C distribution). This is a place holder since we now are getting much better resolution.

6.1 Example of a running job in ATHENA

```
from AthenaCommon.AppMgr import ServiceMgr
ServiceMgr.MessageSvc.OutputLevel = WARNING

from GaudiSvc.GaudiSvcConf import THistSvc
ServiceMgr += THistSvc()
ServiceMgr.THistSvc.Output = ["AANT DATAFILE='ZDCOff.root' OPT='RECREATE'"]


#include( "ParticleBuilderOptions/AOD_PoolCnv_jobOptions.py" )
#include( "ParticleBuilderOptions/McAOD_PoolCnv_jobOptions.py" )

import AthenaPoolCnvSvc.ReadAthenaPool
ServiceMgr.EventSelector.InputCollections = [ "ESD.pool.root" ]  #Change this to your


from AthenaCommon.AlgSequence import AlgSequence
job = AlgSequence(OutputLevel = INFO)


from ZdcRec.ZdcRecConf import ZdcRecAnalysisOff         #This is the ZDC Reconstructi
job += ZdcRecAnalysisOff( "ZDCREC",OutputLevel = INFO )

job.ZDCREC.OutputLevel = INFO
theApp.EvtMax = -1
```

6.2 the code-available on SVN at-....

6.2.1 ZdcSignalSinc.h

```
#include <math.h>

class ZdcSignalSinc {
 public:
  enum Status {e_OK=0, e_noData, e_wrongFrac, e_Overflow,
       e_wrongSignal, e_noSignal, e_localMinimum};

  ZdcSignalSinc(int);
  ~ZdcSignalSinc();
  int    process(double *,double gain=1., double ped=0.,
```

```
 double frac=1., bool corr=true);
  int    getError();
  int    getWarning();
  double getTime();
  double getAmp();

 private:
  const   int    n_Slices;
  const   double m_AmpThresh;
  const   double tClock;
  const   double Pi;
  double  m_Time;
  double  m_Amp;
  int     m_Err;
  int     m_Warn;

  bool    m_CorrFlag;

  double *m_buf;

  int     m_p;
  int     m_np;
  double  waveform(double t);
  double  findpeak(int);
  double  findpeak();
  double  fraction(double,double);

  double  tim[3],wfm[3],dt;



};
```

6.3 ZdcSignalSinc.cxx

```
#include "ZdcSignalSinc.h"

ZdcSignalSinc::ZdcSignalSinc(int nSlc)
  : n_Slices(nSlc)
```

```
    , m_AmpThresh(5.)
    , tClock      (25.)
    , Pi(4*atan(double(1.)))
{
  m_buf = new double[n_Slices];
  m_Time   =  0.;
  m_Amp    =  0.;
  m_Err    =  e_noData;
}

ZdcSignalSinc::~ZdcSignalSinc() { delete[] m_buf;};

int ZdcSignalSinc::process(double *buf, double gain, double ped,
   double frac, bool corr) {

  m_Time = 0.;
  m_Amp  = 0.;
  m_Err  = e_OK;
  m_Warn = 0;

  if (ped==0) ped=buf[0];
  for (double *p=m_buf;p<m_buf+n_Slices;p++) {
    double a = *(buf++);
    if (a>1015) {
      m_Err = e_Overflow;
      return  m_Err;
    }
    *p = a-ped;
  }

  if (frac<=0.) {
    m_Err = e_wrongFrac;
    return m_Err;
  }

  int imax = -1;
  if (m_buf[0         ]-m_buf[1         ]>m_AmpThresh) {
    imax = 0;
    m_Warn += 1;
  }
```

```
  if (m_buf[n_Slices-1]-m_buf[n_Slices-2]>m_AmpThresh) {
    if (imax<0) {
      imax = n_Slices-1;
      m_Warn += 2;
    } else {
      m_Err = e_wrongSignal;
      return m_Err;
    }
  }
  for (int i=1;i<n_Slices-1;i++) {
    double da1 = m_buf[i]-m_buf[i-1];
    double da2 = m_buf[i]-m_buf[i+1];
    if (da1>0. && da2>=0) {
      if (da1>m_AmpThresh || da2>m_AmpThresh) {
if (imax<0) imax = i;
else {
  m_Err = e_wrongSignal;
  return m_Err;
}
      }
    }
  }
  if (imax<0) {
    m_Err = e_noSignal;
    return m_Err;
  }
  if (imax==1) m_Warn += 4;

  if (imax>=2) {
    if (m_buf[imax-2]>m_AmpThresh) m_p = imax-2;
  }  else                          m_p = imax-1;
  m_np = n_Slices-m_p;

  double t_peak = findpeak(imax-m_p);
  if (m_Err) return m_Err;

  if (frac>=1.) m_Time = t_peak;
  else          m_Time = fraction(frac,t_peak);

  m_Amp = waveform(t_peak);
```

```
  if (gain==1.) {
    if (m_Amp<200.) m_Warn +=  8;
    if (m_Amp>900.) m_Warn += 16;
  } else {
    if (m_Amp< 50.) m_Warn +=  8;
    if (m_Amp>250.) m_Warn += 16;
  }
  m_Amp *= gain;
  m_Time = (m_Time+m_p)*tClock;
  if (corr) {
    int off = int(10.+m_Time/tClock)-10;
    double t = m_Time - off*tClock;

    if      (t< 5.) t =   0.056836274872111      + 1.532763686481279  *t
      +   0.229808343735056 *t*t - 0.0365636647119192 *t*t*t;
    else if (t<16.) t =  -1.28974955223497       + 2.653578699334604  *t
      -   0.1371240146140209*t*t + 0.00281211806384422*t*t*t;
    else            t = -42.18688740322650       + 8.61752055701946   *t
      -   0.4259239806065329*t*t + 0.00753849507486617*t*t*t;

    m_Time = off*tClock + t;
  }

  return m_Err;
}

int    ZdcSignalSinc::getError()   {return m_Err;}
int    ZdcSignalSinc::getWarning() {return m_Warn;}
double ZdcSignalSinc::getTime()    {return m_Time;}
double ZdcSignalSinc::getAmp()     {return m_Amp;}

double ZdcSignalSinc::waveform(double t) {
  double f = 0.;
  for (int i=0; i<m_np; i++) {
    double x = Pi*(t-i);
    if (x) f += m_buf[m_p+i]*sin(x)/x;
    else   f += m_buf[m_p+i];
  }
  return f;
}
```

```
double ZdcSignalSinc::fraction(double frac, double tpeak) {
  tim[0] = 0.;    wfm[0] = waveform(tim[0]);
  tim[1] = tpeak; wfm[1] = waveform(tim[1]);
  dt     = tpeak/2.;
  double thr = frac*wfm[1];
  while (dt>0.001) {
    double t = tim[0] + dt;
    double f = waveform(t);
    if (f>thr) {tim[1] = t; wfm[1] = f;}
    else       {tim[0] = t; wfm[0] = f;}
    dt /= 2.;
  }
  return (tim[0]+tim[1])/2.;
}

double ZdcSignalSinc::findpeak(int im) {
  tim[0] = im; tim[1]=im+0.5; tim[2]=im+1.;
  for (int i=0;i<3;i++) wfm[i]=waveform(tim[i]);
  dt = 0.5;
  double t = findpeak();
  return t;
}

double ZdcSignalSinc::findpeak() {
  if (dt<0.001) return tim[1];

  if (wfm[0]<wfm[1]) {
    if (wfm[2]<=wfm[1]) {
      dt /=2.;
      double t1 = tim[1]-dt;
      double f1 = waveform(t1);
      if (f1>wfm[1]) {
tim[2] = tim[1]; wfm[2] = wfm[1];
tim[1] = t1;     wfm[1] = f1;
      } else {
double t2 = tim[1]+dt;
double f2 = waveform(t2);
if (f2>wfm[1]) {
  tim[0] = tim[1]; wfm[0] = wfm[1];
```

```
  tim[1] = t2;      wfm[1] = f2;
} else {
  tim[0] = t1;      wfm[0] = f1;
  tim[2] = t2;      wfm[2] = f2;
}
      }
    } else {
      tim[0] = tim[1]; wfm[0] = wfm[1];
      tim[1] = tim[2]; wfm[1] = wfm[2];
      tim[2] += dt;     wfm[2] = waveform(tim[2]);
    }
  } else {
    if (wfm[2]<=wfm[1]) {
      tim[2] = tim[1]; wfm[2] = wfm[1];
      tim[1] = tim[0]; wfm[1] = wfm[0];
      tim[0] -= dt;    wfm[0] = waveform(tim[0]);
      return findpeak();
    } else {
      m_Err = e_localMinimum;
      return 0.;
    }
  }
  return findpeak();
}
```