Polarimeter Note
August 13, 2003

The TPOL Software

Jenny Böhme
DESY
Jenny.Boehme@desy.de

Abstract

This note describes the software package "daq" of the Transverse Polarimeter at HERA II. The main emphasis is on the offline analysis software as well as on the storage of status information and analysis results in the ORACLE database.

Contents

Chapter 1

Introduction

In the HERA storage rings at DESY, Hamburg, electrons (or positrons) with an energy of 27.5 GeV collide with protons carrying 920 GeV in two interaction regions, where the experiments H1 and ZEUS measure the products of the collisions. Furthermore two fixed target experiments, HERMES and HERA–B, make use of only the electron or proton beam, respectively. The electron beam at HERA can be tuned such that it becomes transversely polarised via the Sokolov–Ternov effect [1]. At HERA I (up to the year 2000), only HERMES used special chains of magnets rotating the spin of the electrons in order to achieve longitudinal polarisation at its interaction region. After HERA had been upgraded for higher luminosity (HERA II), also H1 and ZEUS installed spin rotators in order to make use of the polarised lepton beam. The degree of polarisation is measured by two polarimeters. The Transverse Polarimeter (TPOL) measures the transverse polarisation near the HERA–B interaction region, and the Longitudinal Polarimeter (LPOL) measures the longitudinal polarisation between the HERMES spin rotators. While the LPOL remained more or less unchanged awaiting a later upgrade, the TPOL has been improved substancially during the luminosity upgrade in order to deliver a polarisation measurement of increased precision not only to HERMES, but also to the H1 and ZEUS experiments. The upgraded polarimeter itself as well as the data analysis is described in a detail in other documents [2]. This note intends to give an overview of the new software package of the TPOL (called `daq`), which comprises besides the actual data acquisition and monitoring software also the online and offline analysis, data storage and the control software. The main part of package is written in C++. Exeptions are the GUI for controlling and monitoring the polarimeter, which uses java, as well as some parts of the offline analysis software, which makes use of fortran packages like HBOOK [3] and MINUIT [4].

The data storage proceeds on two levels. The complete raw data are stored in the disk cache [5] mass storage system, which is run by the DESY IT devision, under /pnfs/tpol/public/[year]/[month]. All montoring information as well as the online and offline analysis results are stored in an ORACLE [6] database, also run by DESY–IT. The `daq` package contains interfaces to directly access both data sources from your program.

The whole package is stored in a CVS–repository in afs. If you wish to access the code, please contact Stefan Schmitt (`Stefan.Schmitt@desy.de`) or Jenny Böhme

(`Jenny.Boehme@desy.de`). If you have problems with the code, if you find bugs or if you have suggestions for improvements, please also contact the above persons or (for the offline part) Benno List (`Benno.List@desy.de`).

This note is organised as follows: Chapter 2 gives you an overview of the existing sub-packages and libraries. In chapter 4, the offline analysis part is then described in more detail, presenting the most important classes. Chapter 5 is dedicated to the database, describing the general access to the database as well as the most important tables.

Chapter 2

The Code Organisation

2.1 Accessing the Code

The whole package `daq` is stored in a CVS–repository under afs at the following location: `CVSROOT=/afs/desy.de/user/p/pol2000/CVSroot/` (or if you don't have afs on your computer: `CVSROOT=:ext:pol2000pctpol01:/afs/desy.de/user/p/pol2000/CVSroot/`. To be granted the necessary afs rights, please contact Stefan Schmitt (`Stefan.Schmitt@desy.de`) or Jenny Böhme (`Jenny.Boehme@desy.de`). Detailed information on CVS can be found in [7], however the steps to access the TPOL code should be mentioned here. After having set the shell variable CVSROOT to its above value (in case of remote access you need also to set CVS_RSH=ssh), all you have to do is to execute `cvs checkout daq`. This will install in your current directory a subdirectory called `daq`, which contains the whole code package. To make sure that you always have the most recent version, it is important to update your copy of the code frequently (especially if you have requested a change!) using `cvs update daq`.

2.2 The Directory Structure

In the `daq` directory you find the following files and directories:

- `CVS`: directory containing bookkeeping information for CVS
- `comis`: directory containing a few fortran functions for offline analysis, which are wrapped in C++ classes, but which can also be used interactively from paw
- `doc`: expert documentation on hardware components
- `driver`: drivers for several hardware components
- `include`: header files for the C++ part of the software. There is a subdirectory for each TPOL library.

- `java`: the java part, i.e. the TPOL monitor program. For documentation of this part see [8]
- `kumac`: a few PAW kumacs for monitoring purposes
- `linux`: executables and libraries for linux will be written here
- `lynxos`: executables and libraries for lynxos will be written here
- `sql`: SQL macros defining the ORACLE tables
- `src`: source code for the C++ part, again one subdirectory per package
- `sweeper`: program to sweep read out
- makefiles: the makefile is generated automatically from several input makefiles, see next section
- `cshosts.csv, daqparameters.init, exports.csv, lumibus.setup, motor.speed, tpol.hosts`: several steering or setup files

2.3 The Makefiles

The actual `makefile` includes several other makefiles. The largest part, containing all dependencies for each object file in each library, `makefile.tpol` is generated with the command `make newmakefile` by running the program `tpolmake`, which reads `makefile.tpol.old` as input. The `makefile` distinguishes between different operating systems, therefore the environment variable OSTYPE needs to be set accordingly (i.e. OSTYPE=linux for linux PCs). The included makefiles do the following:

- `makefile.paths`: defines the paths to the diffenrent subdirectories with header and source files. In case new subpackages are added, their paths have to be defined here.
- `makefile.libs`: in the first part the libraries are defined, where most libraries correspond to one of the source subdirectories. In the second part, more important for the user, the executables have to be listed together with the libraries they need. If you wish to add your own executable, you have to add a line here like:
  `# tpolexe [name of executable] [list of libraries]`
- `makefile.$OSTYPE`: defines operating system depending settings like compiler, optimisation level, warning level, preprocessor flags
- `makefile.flags`: contains not system depending compiler settings, mainly paths of directories containing the header files

Furthermore, the makefile contains explicitly the depedencies for a few exceptional libraries or executables, like the `newmakefile` command, short cuts to make several executables at once, some libraries for the MFCC and for the NETMEX connection. Most important exception for the user is probably the dependency list of the comis function library. If you add a new fortran function to the `comis` directory, you have to add its object file explicitly to this list in the `makefile`. The object file is then created automatically by the corresponding suffix rule.

Having generated the makefile, each executable can be built by executing `make linux/[name of executable]`. The first compilation of all the libraries may take some time. Afterwards, only files that changed or that depend on changed files are recompiled. Anytime you add new classes or when you include a header file in another place, you should rerun `make newmakefile`, in order to get the dependencies correct and thus recompile the right files. If a header file which is included indirectly through several other files is changed, not all affected files are detected, which leads to strange compiler errors, usually complaining about errors in the virtual table. In such a case it might be necessary to touch by hand all .C files in the package you are working on (or to remove the corresponding library) in order to force a complete recompilation of this package.

Usually, the program generating the makefile works fine and does not need to be changed. There is however one exception: if you use a new **system** header file that has not been used before in the TPOL code (there is no need to check this yourself, the makefile will complain), you should add this new system header file in src/executables/tpolmake.C (search for example for `math.h` and treat your file the same) and recompile the tpolmake program.

2.4 The executables and your own code

The source code of all executables can be found in `daq/src/executables`. Besides several test programs there are programs for the developping work on the offline analysis and of course the actual daq programs for handling the data. You will probably soon write your own executable for your special purpose. You can of course also write your own classes. Depending on their purpose, you may either put them into one of the existing packages or create a new package. (Re–)reading the makefile section might help to avoid a few pit falls. If you wish to store your new classes or executables in the CVS repository, you need the proper password which you can get from Stefan or Jenny, but we probably will be curious to learn what you want to do....

Chapter 3

The Packages

3.1 The C++ Packages

This chapter gives an overview on the existing subpackages in alphabetical order. The `offline` and `oracle` packages are described in more detail in the next chapters. For all packages, the header files containing the class definition are stored in `daq/include/[package name]` and the source files containing the implementations can be found in `daq/src/[package name]`.

3.1.1 analysis

The `analysis` package implements the fast online analysis.

3.1.2 data

The `data` package contains classes defining and handling the raw data.

3.1.3 hbook

The `hbook` package provides a simple wrapper for the HBOOK [3] library as well as an object oriented interface to some of the most important HBOOK functionalities. This package has been provided by Benno List. For details see [9].

3.1.4 message

The `message` package provides classes handling messages exchanged between DAQ programs and hardware components.

3.1.5 mfcc

The `mfcc` package contains some MFCC code.

3.1.6 minuit

The `minuit` package contains a C++ wrapper for the MINUIT [4] fitting package – again special thanks to Benno List!

3.1.7 netmex

The `netmex` package takes care of the communication with the HERA information exchange system NETMEX.

3.1.8 network

The `network` package handles communication between the TPOL components.

3.1.9 offline

The `offline` package contains classes for offline analysis. It is described in more detail in the next section.

3.1.10 oracle

The `oracle` package provides an interface to the ORACLE database. It is described in section 5

3.1.11 pilot

The `pilot` package implements the TPOL autopilot, wich is realised using final state machines.

3.1.12 runstatus

The `runstatus` package contains the classes holding all status information of the TPOL. A runstatus object is stored at the beginning and at the end of each data taking cycle with the raw data. When ever status information changes, the DAQ notifies the oracleclient, who will store the new status in the database. For some classes filtering algorithms have been implemented to keep the amount of data written to ORACLE at a sensible level.

3.1.13 statusdb

The `statusdb` package provides an online database, which is stored locally on the tpol online PC. It serves as back up in case the connection to ORACLE does not work for some time. The functionality is basic and only a limit history is kept. Users should always access the ORACLE database!

3.1.14 sync

The `sync` package contains some online stuff.

3.1.15 user

The `user` package contains some online stuff.

3.1.16 vme

The `vme` package contains some online stuff.

3.2 The comis Package

Some functions used in the offline analysis are provided in FORTRAN, to allow usage as comis functions in PAW. Often there is a wrapper funtion for fitting purposes named the same as the core function but with "fit" in the beginning. They are not mentioned explicitely in the list below. Examples for calling FORTRAN functions from C++ can be found in the offline package, e.g. compton.C.

- brems.f: cross section for bremsstrahlung as function of the photon's energy, see also newbrems.f
- bremsscreened.f: cross section for bremsstrahlung as function of the photon's energy taking into account screening effects by rest gas atoms.
- dsigmade.f: cross section of Compton scattering as function of the scattared photon's energy.
- dsigmadphide.f: cross section of Compton scattering as function of the scattared photon's energy and azimutal angle.
- dsigmadyde.f: cross section of Compton scattering as function of the scattared photon's energy and vertical coordinate.

- newbrems.f: More general implementation of cross section for bremsstrahlung as function of the photon's energy.
- polynom.f: polynomial of degree up to 20.
- smear.f: dsigmade.f smeared with calorimeter resolution.
- smearbrems.f: brems.f smeared with calorimeter resolution.
- smearnewbrems.f: newbrems.f smeared with calorimeter resolution.
- smearscreened.f: bremsscreened.f smeared with calorimeter resolution.
- trig.f: parametrisation of trigger threshold of calorimeter.

3.3 The java Package

The java subdirectory contains the code of the monitoring display of the TPOL and of the run control. For further documentation see [8]

Chapter 4

The Offline Analysis Software

This chapter gives a few hints on the usage of the offline analysis package. The description is kept on a technical level. A scientific discussion of the offline analysis can be found in [2].

Since the TPOL collects Compton events for about a minute in order to perform one polarisation measurement, histograms play a central role in the analysis. The offline package comprises its own set of histogram classes (called "spectrum..."). Besides the usual histogramming functionality they provide also some features special to polarisation analysis. The key issue of the offline analysis is fitting the double differential cross–section to the measured spectra, giving measurements of the system's parameters as well as of the polarisation as results. One iteration in this fit comprises calculation of the theorectical cross section according to the given parameters, applying several steps of smearing and coordinate tranformation and finally comparing the result bin by bin to the data and calculating the corresponding χ^2. All these steps are implemented here. The actual minimisation procedure is handled by MINUIT [4].

Another subset of classes in this package helps reading different data formats, allowing access not only to the recent data, but also to HERA I data and MC, being stored as HBOOK [3] files.

4.1 Imporant Classes to know

In this section the most important classes are described in more detail. The description starts with higher level classes playing a central role in extracting the polarisation and continues with more technical (but nevertheless very useful) classes.

4.1.1 The Event

In the context of the TPOL analysis the word "event" often does not refer to a single compton scattering, but to a pair of data taking cycles with the laser switched on and off, leading to one polarisation measurement. Correspondingly the `Event` class provides access to the raw and status data of one "LASER ON" and one "LASER OFF" cycle specified in the constructor either as single files or as a runfile and the numbers of the cycles that should be used. The `Event` performs background subtraction (subtracting the "LASER OFF" spectra from the "LASER ON" spectra taking into account the actual time interval of each cycle including deadtime corrections) as well as the energy calibration of the calorimeter. Via maps it provides access to all kind of histograms using easy to remember strings (see class `HistoIndex`) instead of cryptic histogram id numbers.

4.1.2 Methods to Calculate the Polarisation

When analyzing the data offline, several methods can be used to extract the polarisation. For each method a class is derived from the (abstract) base class `Polarisation`, implementing its own `calculate` method. Each algorithm is expected to provide results for beam polarisation, difference of the linear components of the laser polarisation between the right and left handed laser state, beam position relative to the calorimeter and the beam spot size. The following choices are implemented:

- pololdanalysis: reruns the online analysis, but using the current data for calibration instead of the data from the previous event.
- polfitmethodbase: base class for all methods using the 2D fit.
- polfitmethod: full offline analysis comprising calibration and polarisation fits, derived from polfitmethodbase. In the calculate method details of the fits can be specified, e.g. which parameters should be fixed or bounded, which bins should be fitted, which base id number should be used for HBOOK output and so on.
- polfitpolonly: like polfitmethod, but performs only the polarisation assuming the calibration is known sufficiently well.

4.1.3 The Offline Fit

The actual fit function is implemented in the `PolFitter` class. It is derived from the `FitFun2D`, which is itself a daughter of `FitFun`. For each `FitFun`, the `operator()` is overloaded with the actual function to be fitted. For details of the general fitting mechanism see [2]. For creating a `PolFitter` object, you need to provide the following information:

- a base ID number for HBOOK output,
- 2D spectra corresponding to the three cross section terms Σ_0, Σ_1 and Σ_{2Y} and functions of energy and vertical coordinate of the scattered photon,
- the binnings for the energy axis, the y axis and the η axis,

- an η–y–transformation object and
- two vectors containing the start values of the fit parameters and their errors.

When the `operator()` of `PolFitter` is called, the cross–section spectra are added and transformed according to the parameters and the resulting twodimensional spectrum is returned. Since some of the transformations are very time consuming, they are stored as special kind of maps (”resampling maps”) in the `PolFitter`, so that they have to be calculated only at the first call to the `operator()` or when one of their specific parameters has changed.

4.1.4 The Spectrum Classes

There are two histogram classes in the offline package: `Spectrum` for one–dimensional histograms and `Spectrum2D` for two–dinesional histograms[1]. As first order approximation, you can imagine a spectrum as an HBOOK histogram, although it is a lot more clever, especially concerning fitting related issues. For a full list of functionality see `daq/include/offline/spectrum.h and spectrum2d.h`

[1]For technical purposes there are derived classes which can be filled from the raw data format: `SpectrumH` and `Spectrum2DH`.

Chapter 5

The ORACLE Database

The database itself is maintained centrally by DESY IT. The polarimeter group is responsible for creating and filling the tables needed to store the desired information. In case of performance problems or other technical questions to the database itself, contact the IT division.

5.1 Tables and Classes

The existing tables store not only the online and offline polarisation, but also all status information of the TPOL. The table layout follows closely the class structure of the `runstatus` subpackage. In most cases, there is one table for each statusdata class, containing the data members of this class as columns and named with the capital letter part of the corresponding class name (i.e. `POLFITStatusData` → `POLFIT`). Whenever this status object is updated by the DAQ, a new row is added to this table. If however a statusdata class contains information on several hardware elements which usually change independent of each other (for example stepping motors), separate tables of identical structure are used for each independent component. In this case, a serial number is added to the table name, i.e. `MOTOR0, MOTOR1, MOTOR2, ...`. To get a quick listing of all tables and column, the web interface can be used (pctpol03.desy.de/ pol2000).

All tables are indexed by their unixtime stamp, which is internally multiplied by 100 and added to a version number, which allows several entries for the same second. However this feature is not really used (yet?).

5.2 Accessing the ORACLE database

Both read and write access is implemented in a generic way for all tables using the ProC precompiler, which allows to write SQL statements within the C++ code. The actual access follows the so called Method 4 of Dynamic SQL allowing to generate the SQL statements at runtime. The code can be found in `daq/src/oracle/oracle.pc`. The corresponding oracle.C file is generated by the precompiler (just type `make oracle.C` in the `oracle` directory) – don't edit it directly!

The specification which variables should be written to which table in which order is given in the StatusData classes, namely in the method `defineVariables`. If variables are added to or removed from a table, this method has to be adapted in the corresponding StatusData class.

In case another database (for example MySQL) is desired, you just need to derive a corresponding class from the DataBase base class (from which also the oracle class is derived), implement the read and write methods and then create an object of this class whenever access to this database is desired. The StatusData classes don't require any changes for this purpose.

5.2.1 Writing to ORACLE

The tables in the ORACLE database are filled by an executable called `oracleclient`, which should be running all the time and which is notified by the DAQ whenever new data arrives. This program has been running up to now on Jenny's desktop PC (flc13). It should be put in a more official place, i.e. a dedicated TPOL computer where it can also be monitored and restarted automatically. In case the connection to ORACLE is interrupted for some time, there is another executable `dbtooracle`, which takes the unixtime range for which the data haven't been written to ORACLE and copies them from the online database.

5.2.2 Reading from ORACLE

There are several methods to read from the ORACLE database. For very simple queries, which only need information from one table, you can use the web interfaces mentioned above to get a plot, a rowwise ntuple or an ascii table of your query results.

For more complicated queries or when accessing the database from another program, you should use the DAQ framework. Basically, you just need to create an `oracle` object and an instance of the `StatusData` class corresponding to the table you would like to access. Then you just call its `fromDatabase` method with the `oracle` object as argument (for the full argument list see `daq/include/runstatus/statusdata.h`).

Often the access to a large amount of entries is needed. This is much faster if all entries are fetched in one query from the database. On the other hand, it is desirable to allow some user interaction for each entry separately, for example add it to an ntuple. This can be achieved by using visitor classes derived from `daq/include/oracle/statusdatavisitor.h`, as for example `daq/include/oracle/polfitvisitor.h`. If such a visitor object written by the user is given to the `fromDatabase` method, its `operator()` is called automatically for each entry read from the database. User tasks which should be performed once in the beginning or in the end can be put in the constructor or destructor, respectively. An example for a main program reading all polarisation values for a given time range and writing them into an ntuple can be found in `daq/src/executables/oracletest8.C`

5.3 Adding new Tables

The tables have been created interactively using the `sqlplus` commandline interface to the database. The corresponding SQL macros are stored in the directory `daq/sql`. If a new table is needed, use these macros as example for adding a new one. Having created the new table, you will need a class that contains the corresponding data – and which is able to handle the read and write acess. The easiest way to achieve this is to derive your class from the `StatusData` base class. Then you only have to implement the `name` and `defineVariables` methods.

5.4 What is still missing

Since the question which polarisation measurement (TPOL, old LPOL, new LPOL, online, offline, averages, ...) will only be resolved with time once serious data taking has started, it is planned to introduce a master table, which always points the user automatically to the table containing the values considered best by the pol2000 group for the time period in question.

With relational databases, this should be implemented using an additional key table. Then, the master table contains for each unixtime just a short number which codes the actual table name. The meaning of the codes is then stored only once in the key table.

These two tables could then be used via a `BESTPOLStatusData` class (plus a corresponding visitor if desired) which then looks up the table in the `name` method to query from the master table instead of using a hard coded name.

Bibliography

[1] A.A.Sokolov, I.M.Ternov, V.V.Mikhailin, Izv. Vuz. Fiz. **4** (1976) 7.

[2] J.Böhme, *A New Analysis for the Transverse Polarimeter at HERA II*, 2003. http://pctpol03.desy.de/ pol2000

[3] *HBOOK reference manual (Version 4.23),* Program Library Y250, CERN, 1995.

[4] F. James, *MINUIT reference manual (Version 94.1),* Program Library D506, CERN, 1994.

[5] Disk Cache (dcache), see DESY–IT web pages.

[6] Oracle Database at DESY, see DESY–IT web pages.

[7] Per Cederqvist *et al.*, *Version Management with CVS*, 1992.

[8] S.Schmitt, *TPOL Monitor Online Manual*, http://pctpol03.desy.de/ pol2000/monitor/manual.html.

[9] B.List, *A C++ Interface for HBOOK*, https://www-h1.desy.de/ blist/c++/.