STAR

Offline Library TRS Long Writeup

Tpc Response Simulator

User Guide and Reference Manual

Revision: 1.3

Date: 1999/01/24 20:58:59

Contents

Part I

User Guide

1 Philosophy and Motivation

With the decision at the STAR collaboration meeting in January 1998 at LBNL that a migration to the C++ programming language and Object-Oriented (OO) coding practices will be made, it was suggested that the TPC slow simulator be the first package to be re-engineered using these techniques and principles. This project was to be a prototype for such future developments within the collaboration, both in its design and structure. A requirements documents was quickly prepared after the meeting and work began on simple design, prototyping, and evaluation of various commercial and public domain class libraries.

The result of these initial studies was the development of the StarClassLibrary[1] which drew on the developments of CLHEP but relied heavily on the ANSI standard[2] C++ Standard Library and specifically the Standard Template Library or STL. This library provides several physics containers (three- and four- vectors) as well as matrices random number generators, a system of units, etc.). It was upon this development that the TPC Response Simulator (TRS) was built.

The goal of re-engineering TRS was two-fold:

- generate an object-oriented package using C++ in the STAR computing environment.
- add detail and develop the simulation tools parallel to the commissioning schedule of the STAR TPC.

In order to add flexibility to the package it was designed to support two types of simulation from a single interface; that is, a very detailed microscopic simulation of the processes occurring in the STAR-TPC and a faster parameterized version which could be used for large scale processing. In order to do this the design concentrated on isolating the physics processes of interest and how the data could be stored in generic data containers. These ideas, as well as a preliminary class design were drawn up in a design document[3] which was discussed at the STAR collaboration meeting in August 1998 at BNL. This document was the basis for the design and developments for the first version of TRS. This document is a report of these developments as well as a guide for its use in its current form.

2 Organization of the Manual

This manual is a description of the TPC Response Simulator (TRS) package. The first part of the manual is a **User Guide** which describes the platform dependencies, package organization and general description of the design and basic implementation of the components within the package. The subsequent parts make up the **Reference Manual** which is a detailed description of the component classes of TRS. This is broken into four different sections, the first two of which describe the generic data base and coordinate classes used in the package, which are not necessarily specific to TRS, but which are of interest to the general STAR software environment and could be used in any software package. If these gain acceptance within the collaboration, they can be moved to a more accessible location such as the StarClassLibrary The last two parts deal with the TRS specific classes which are broken into two separate components—the **containers** and **processes**.

[1] http://star.physics.yale.edu/SCL
[2] http://www.cygnus.com/misc/wp/nov97/
[3] see http://www.rhic.bnl.gov/STAR/html/comp_l/simu/TpcRespSim/src/ps/TrsDesign.ps

3 Platforms and Compilers

The TPC Response Simulator (TRS) was developed on an HP workstation since HP currently provides the closest ANSI compliant compiler and standard libraries. However the target platforms on which the code will ultimately run are LINUX and SUN . Red Hat LINUX 5.0 and higher versions provide very close to fully ANSI compliant compilers with only exceptions[4] being not fully supported. The current SUN compiler (CC v4.2) however, is far from compliant and the large scale use of macros populate the code to allow for operation in a SOLARIS environment. A modified standard C++ library from Object Space (v2.0.2) was created in order to make up for the vendor short-comings. This library is currently available for distribution from the STAR CVS repository. It is foreseen that the new SUN compiler (CC v5.0) which is scheduled for release in March of 1999 should large solve many of the current problems and inconveniences of the SUN compiler.

TRS has been tested and the example programs that utilize the TRS classes currently compile on the following platforms:

1. HP-UX 10.20:
    - aCC A.01.06 or higher versions. The standard library supplied from the vendor is the basis of most other libraries in the industry.
2. Red Hat Linux 5.0 or higher versions:
    - egcs 1.0.2 (g++).
3. Solaris 2.4–2.6:
    - CC 4.2 and Object Space 2.0.2 modified.

Tests with Visual C++ 5.0 have not been attempted since the StarClassLibrary (SCL) does not currently build due to the lack of support of templated member functions and a broken overloading mechanism. It is hoped that this will be addressed with v6.0 which has been released (but not tested). In the interim, non-templated container classes have been written and incorporated into theSCL, so that this should now be possible. No efforts have been spent to write, port, or adapt the code to SGI-IRIX nor IBM-AIX and such developments are not foreseen unless there is significant user demand.

4 Organization of the TRS Package

All documentation, code, and header files of TRS are contained in the **CVS** repository at BNL in the `$STAR/StRoot` section in a directory named ./St_trs_Maker.[5]. There are several subdirectories in which the code is then distributed. The header files as well as the source code in contained in the ./include and ./src directories respectively. The ./local directory contains the necessary **GNUMakefile**s in order to compile stand-alone shared libraries such that code can be run outside the ROOT/STAR framework. This was

[4] and some member functions of minimal importance in the STL.
[5] To be changed to: StTrsMaker/

initially done to avoid the complications of the constantly evolving STAR infrastructure, (i.e. STAF/ROOT) however it has been retained in order to aid in parallel development projects which require the use of some classes within TRS, but do not necessarily require or warrant the use of the complete STAR infrastructure.

Example programs as well as those to monitor and control the developments within the TRS classes are contained in the directory ./examples. This directory also contains the necessary **GNUMakefile**s for compilation. To avoid confusion, both the TRS shared libraries and the executables from the shared libraries are kept in a directory denoted by the system architecture derived from the STAR environment variable **$STAR_SYS**. Finally the documentation is stored in the ./doc directory. This currently consists of this manual which is written in LaTeX. A class browser which illustrates the function prototypes, also used with the SCL is also a possibility in the future. The developments that occur in TRS are kept on the STAR WWW page (see: http://www.rhic.bnl.gov/STAR/html/comp_l/simu/TpcRespSim/src/Welcome.html).

All makefiles are especially written for the GNU make utility (gmake) and rely on certain features only available therein. This is also the reason why we do not follow the standard naming scheme but named them GNUmakefile. Note that these **Makefile**s do not contain any hard-wired file names; the dependencies are rather created on-the-fly at compile time. The libraries should also be generated from within the ”official” STAR **Makefile**s.

5 Requirements and Installation of TRS

The TPC Response Simulator (TRS) requires the Standard C++ Library, including the STL as well as the STAR Class Library (SCL). TRS was structured such that if the SCL will compile and run on a system, so too will TRS.

TRS is part of the official STAR software distribution and is therefore present in the actual STAR software releases. If you want to install TRS locally on your machine you should continue to read the following three sections which explains how to obtain, compile and install the code.

5.1 The source code

The TPC Response Simulator is under **CVS** control at BNL. It can be accessed via afs:

1. Obtain an afs token: `klog -cell rhic`.

2. Make sure `$CVSROOT` is set properly. It should be if the STAR setup scripts were run at login time: (i.e. `CVSROOT = /afs/rhic/star/packages/repository`)

3. Check-out package into your current working directory:
   `cvs checkout StRoot/StTrsMaker`

5.2 Compilation

In order to create the library and (optionally) the referring examples you must first have the STAR Class Library (SCL) installed and compiled. For instructions see the SCL manual[6] . After the SCL is installed and compiled, the directory with the header files and shared libraries must be set in the TRS **GNUMakefile.architecture**. If the installation is away from BNL, a local (or through afs) copy of the SCL can be used. In order to use this, a flag (`AT_BNL`) can also be set in the **GNUMakefile.architecture**, otherwise, the BNL installation of the SCL is assumed to be used.

Please note, however, that TRS is not yet supported on all STAR platforms. See section 3 for the current platforms that have been tested. To compile:

- set the correct system flag:
  In csh, tcsh:
  `setenv STAR_SYS` *system-flag*.
  In sh, ksh, bash, zsh:
  `export STAR_SYS=`*system-flag*.
  where *system-compiler* can be one of:
  - `i386_linux2`
  - `i386_redhat50`
  - `i386_redhat51`
  - `i386_redhat52`
  - `hp_ux102`
  - `sun4x_54`
  - `sun4x_55`
  - `sun4x_56`
- `cd StRoot/StTrsMaker/local`
- `gmake`
  compiles the code into a shared (or archived) library.

This creates a shared (or archived library) against which the test programs (or your own) can link against. A library is located under a directory with the same name as the `STAR_SYS` environment variable. This setup is best suited for developers working with, and on, TRS.

Hint: In case your platform is not supported and you encounter problems with your native C++ compiler you can always use the GNU compiler (egcs or gcc depending on your installation) by defining `STAR_SYS=i386_linux2`. Note also that "unsupported platform" does not necessarily mean that the SCL/TRS will not compile, but rather that it was never tested and that the makefiles contain no parameters specific for this environment. In general it should be sufficient to adjust the settings in GNUmakefile.architecture (see also section 6).

[6] Post-script version available at http://star.physics.yale.edu/SCL

6 Macros

The TPC Response Simulator (TRS) is coded under the assumption that all ANSI features, and a standard Library, including the STL, are available, and that there is access to the Star Class Library . If the compiler used is fully ANSI compliant TRS will compile without any modifications, as will the STAR Class Library (see the SCL manual for specific details).
Because the new C++ ANSI standard pushes the limits of current compiler technology, a number of compiler and Standard C++ Library features are often missing or implemented in a way that differs from the ANSI standard.
In order to use TRS on those systems various macros were defined which either disable certain features, e.g. exception handling, or use slightly modified (and less elegant) code. The following macros are used throughout the SCL and TRS. If the SCL and TRS is installed properly they should be defined according to your platform/compiler. Please note that the same flags should be used for the compilation of both the SCL and TRS!

6.1 Generic Macros

The following macros are also used in the Star Class Library .

`ST_NO_MEMBER_TEMPLATES`: defined if the compiler does not support template member functions.

`ST_NO_EXCEPTIONS`: defined if the compiler does not support exception handling. **NOTE:** This flag is recommended to be defined at all times in order to avoid inefficiencies due to the extended memory required for exception handling.

`ST_NO_NUMERIC_LIMITS`: defined if the STL class numeric_limits is not available (it is usually located in the `<limits>` header file).

`ST_NO_TEMPLATE_DEF_ARGS`: defined if the compiler does not support template default arguments

`ST_NO_NAMESPACES`: definded if the compiler does not support multiple namespaces.

`ST_OLD_CLHEP_SYSTEM_OF_UNITS`: user defined if one must use units as defined in CLHEP v1.2 (use of this macro is strongly discouraged).

`NO_HBOOK_INIT`: restricts the automatic initialization of HBOOK memory (SCL specific).

`RWSTD_BOUNDS_CHECKING`: used to check that the bounds of STL containers[7] are not exceeded.

Furthermore several platform idiosynchrocies and shortcomings must be dealt with in a very direct manner. As such, macros are utilized which are defined by the compilers. For:

- SUN → __sun
- HP → __hp
- Linux → GNC_GCC

[7]Only for Rogue-Wave implementations of the STL

6.2 TRS Specific Macros

The following macros are only used in TRS specific classes.

ST_CHECK_SECTOR_BOUNDS: defined if it is desired to check that the indices do not over step the boundaries for pad rows, pad numbers, or time bins.

7 Documentation

The documentation of TRS consists of the **User Guide and Reference Manual** (i.e. this document) is located in ./StRoot/StTrsMaker/doc/tex, and (shortly) a HTML class browser in StRoot/StTrsMaker/doc/html. These are not automatically made during the installation but must be created separately as follows:

- `cd StRoot/StTrsMaker/local`
- `gmake doc`

The class index and all other HTML pages referenced therein will be generated automatically from the current code.

8 Overview and Design

The TRS design document[8] sets out the design, requirements and philosophy of TRS, as well as the first simple description of the algorithms proposed for use in the various processes that are modeled.

The initial prototyping phase led to the development of two types of classes that make up the core of the physics and simulation components of TRS:

- **Processes** – physical processes that model an aspect of the physics occuring in the chamber such as charge transport, wire chamber operation and signal generation. These are defined in detail in section 8.1.
- **Containers** – structures which contain the various types and forms of data required in the time evolution of the simulation. These are defined in section 16.

In the class design of the package, each physical process, as described above, was mapped to a single class, as were the containers. This provides a simple factorization between processes which allows them to function independently of one another. It also allows the isolation of single processes for detailed study independent of the effect of other processes. As an example, previous attempts to simulate the behavior of a wire chamber use a parameterization (f) of a pad response function from empirical relations such as:

$$f(x) \sim e^{-(x-x_o)^2/(2\sigma^2)} \tag{1}$$

[8] http://www.rhic.bnl.gov/STAR/html/comp_l/simu/TpcRespSim/src/ps/TrsDesign.ps

where σ incorporates terms which depend on the orientation of the trajectory of a particle, the transport properties of ionization through a gaseous medium, pad plane geometry, electronics properties, etc. As such it is very difficult to isolate the effects of a specific process, and parameters which affected by many different processes result. This reduces the physical significance of tunable parameters and while this may be fine for large scale production of data, for understanding of the effects of physical processes, it is not very useful. This does not need to be the case since the physics processes important in the operation of a TPC are independent of not only the granularity of the simulation but also the level of detail of the simulation. As an example, one needs to transport the ionization to the read-out plane independent of whether it is a single electron or segment of charge, however the detail of whether one calculates the drift velocity at every point on a fine mesh grid and propagate the charge across the grid, or do a simple projection of the charge onto the pad plane. As such, the above processes can define an interface from which the *FAST* parameterizations and the *SLOW* microscopic simulation can utilize. This allows the use of the same code and interface with only a switch (or flag) which can be used to set the detail of the simulation. This allows the same infrastructure to handle various levels of detail without resorting to separate packages for each. Rather the same interface will mean the same function names will be called, with only the implementations which vary.

As such, the **processes** are the actual physical events that occur inside the chamber and operate on data which are stored in **containers**. Thus a general practice was followed that input data would be passed to a **process** class via a **container** and the processor would either mutate the data within the containers or produce data that would be stored more naturally in a new or separate **container** for the next physics process. As such, although the functions of these two types of classes fulfill two different needs within the package, the relation between them is very close. These relations are explained in terms of very physical means in the following two sections (sections 8.1 and 8.2),

8.1 Physical Processes

There are four main types of **processes** that were identified as being necessary to model in order to reproduce the operation of a TPC in TRS. When discussing the mode of operation of a TPC they become evident:

- *Ionization Transport* – charge transport of the ionization. deposited in the active region to the readout chambers.
- *Charge Collection* – electron/ionization collection on the sense wires of the multi-wire proportional chamber (MWPC).
- *Analog Signal Generation* – charge induction on the pad plane and the generation of the time evolution of the analog signals on the pads.
- *Digital Signal Generation* – Digital conversion of analog signals.

A brief overview of the processes are given below, and more detailed remarks are given in section 9.

8.1.1 Ionization Decomposition and Transport

The ionization transport takes the charge deposited in an active volume within a TPC detector and transports it through the field cage structure of the TPC to the read-out plane. The charge which is deposited can either be generated externally, by GEANT or internally given knowledge about parameters of the the fill gas in the chamber such as the mean free path and ionization potential.

In the most common mode of operation, the ionization of the particle tracks will be described by GEANT, which will provide the amount of energy deposited (dE) over a given path-length (ds), by a particle with a momentum $\vec{p}$. Given the average ionization potential of the gas, the total number of electrons can be calculated such that the transport can be done at the segment level (dE) or the single electron level. This provides a mechanism which allows the possibility to distinguish between a detailed microscopic simulation and a macroscopic parameterization; that is, the granularity of the simulation can take on a range from the single electron level to a charge segment containing many 10s of electrons. By varying the length of the segment that is transported (and subsequently processed), the granularity of the simulation can be varied. The ionization can then be distributed on the pad-plane according to the distributions which characterize the effects of diffusion. The role of the charge transporter is to alter the x and y positions according to the transverse diffusions, the z position to reflect the drift time, with the effects of longitudinal diffusion folded in) and the amount of charge that actually reached the read-out plane.

8.1.2 Charge Collection and Amplification

Once the electrons arrive at the read-out plane, they must be collected by the individual anode/sense wires of the multi-wire proportional chamber (MWPC). This is where the avalanche process multiplies the signal of several 10s of electrons to several 10^3-10^5, depending on the potential on the wires. This operation is somewhat of a hybrid as it is really occurring within a container, however the processes occur in the amplification stage require the knowledge of the wire grid structure, and so the processes are incorporated here for an efficient implementation.

8.1.3 Analog Signal Generation

Once the charge is multiplied at the field wires of the MWPC, the amount of charge induced on the pads produces an analog signal which varies as a function of time. These events can be modeled via two processes:

- charge induction on the pads.
- charge sampling by the electronics.

Given the amount of charge on the wires and the geometry of the pad plane, the induced charge on any arbitrary pad can be calculated. From this quantity of charge, the signal at the output of the pre-amplifier/shaper can be determined given the response (i.e. transfer function) of the analog electronics. Although the time evolution of the signal that is developed on the wires is nearly entirely due to the motion of the positive ions away from the wire, the number of electrons produced provides a measure of the total amount of charge that is available to be induced on the pad plane. The shaping width of the electronics provide an estimate of the fraction of charge that is really observed. In reality only a fraction of the total charge is seen because of the small mobility of the positive charged ions and it takes a long time for the signal to finally decay (the famous 1/T tail). As such the shaping properties of the analog electronics (i.e. pre-amplifier and pulse shaping) plays an important role in the amount of charge actually measured. As such, given the total amount of charge induced on the pads, the electronics will differentiate the signal, and be sensitive to a certain fraction of the total charge. In the case of STAR, it is of the order of 45% of the charge. This analog charge can then be distributed into time bins modeling the behavior of the switch capacitor array (SCA)

8.1.4 Digital Signal Generation

After the analog charge has been distributed into the time bins on each pad, it can be digitized. This is an important property which is to be modeled because STAR uses a non-linear 8-bit ADC which contains the information of a 10-bit linear scale. Both the analog and digital signal generation components operate on the signals which are distributed in time bins, however there functions and operations as well as the information contained differ enough to justify breaking the processes into two distinct types of processes. The digital information can be compressed much more and require a format that is accessible via the *Data Decoder*.[9]

8.2 Data Containers

The containers required for TRS are essentially defined by the physics processes as described in section 8.1. Container classes are used rather than simple data structures because it is more simple to incorporate book-keeping and simple functions which are closely matched to the structure of the container into a class. In essence the containers establish a kind of *I/O* interface between the separate processes and allow the same ”physics framework” to exist, independent of the granularity or detail of the simulation. As a concrete example the physics of the ionization transport of a charge segment through the TPC field cage is independent of whether the segment in question is a single electron or macroscopic charge cloud. As such the containers are designed in a very generic manner that would facilitate implementation of the physics processes at various levels of detail and granularity. The containers required for TRS are:

- *Charge Segment* – the input to the program is an amount of energy (dE) deposited in a TPC volume with length (ds), by a particle of momentum $\vec{p}$. In essence it is a object definition of a *g2t_tpc_hit*-like structure.
- *Charge Mini Segment* – a fraction of a Charge Segment (above) which can be, in the limit, the position of a single electron. This is transported to the read-out plane.
- *Wire Bin Entry* – Position and charge of an ionization segment (Charge Mini Segment) at the wire plane after the ionization transport.
- *Wire Histogram* – all anode/sense wires which make up the MWPC which facilitates the charge collection.
- *Analog Signal* – structure containing the time (centroid) and amplitude of a signal on a pad. This can function as storage for both analog and digital signals
- *Analog Sector* – contains analog signals on pads and is the main working structure for calculations involving the charge induced on the pads.
- *Digital Sector* – contains digital signals on pads and contains the output data of the simulation.

[9] see DAQ interface to Offline, M. Shultz et al.

The containers and the processes are meant to be as independent as possible and only overlap when concerns of efficiency over ride the design guidelines. The functions of the containers are described in more detail below:

8.2.1 Charge Segment

The purpose of the slow simulator is to transform ionization segments (usually produced by GEANT, or some type of external program) to pixel, or Raw TPC Data. As such the input will nominally be a quantity of energy (dE) deposited by a track over a finite path length (ds) at a coordinate ($\vec{x}$). This information was previously wrapped in two C/Fortran structures called *g2t_tpc_hit* and *g2t_tpc_track*. In the context of `TRS` it is necessary, and more convenient, to have this data wrapped in a class structure. This allows the same possibilities of data access as simple structures, however the functionality goes beyond data encapsulation. It also allows the use of more flexible utilities such as the `St`ThreeVector to keep track of the position and momentum of track segment, as well as the utilities these classes provide. Furthermore operations that are intrinsic to the segment itself such as rotation, fragmenting (segment splitting) (*Note:* splitting requires knowledge of the gas parameters as well), etc. can also be associated with the object making it more "complete". These operations are necessary for rotating the charge segment into the sector 12 reference frame of the TPC and splitting the ionization into smaller segments to increase the granularity of the simulation.

8.2.2 Charge Mini Segment

The charge mini segment is a fragment of a charge segment (above) after it has been split. It can contain the complete charge segment or a fraction of it. Each mini segment is distributed within the path length of the charge segment (ds) onto a helical trajectory according to the mean free path of the particle within the gas (medium). It is this mini segment on which the charge transporter *process* will operate. As such the relevant data members are the position of the segment and the number of electrons. Access and set functions are provided so that the transporter has read and write access to these quantities.

8.2.3 Wire Bin Entry

Once the charge reaches the anode wire grid it must be collected on the sense/anode wires. In order to distribute the ionization onto the wires after diffusion effects are introduced (in the charge transporter), the ionization must be repackaged such that a specific quantity of ionization can be assigned to a position on the anode wires. Thus the *wire bin entry* is made to collect an amount of charge, q in the vicinity of an anode wire at a position $\vec{x}$, which can be assigned to a wire.

8.2.4 Wire Histogram

The wire histogram is a class which models the charge collection on single wires in a controlled manner; that is, charge is assigned through the wire bin entry class. The functionality of this class goes beyond simply keeping track of the amount of ionization collected on a single wire, but also possesses functions necessary to calculate gas gain amplification of each cluster. The reason this *"process"* is associated with a container is that the structure and layout of the wire grid, which is necessary in the wire histogram is also necessary to be able to do gas gain. Thus instead of imposing overhead of redundant class construction, it was incorporated here. This is one of the advantages of Object-Oriented design. Each charge cloud can now be used to induce a signal on the pad plane.

8.2.5 Analog Signal

An electronic signal, regardless of it being analog or digital, can be characterized by a pair of numbers which represents the time (centroid) and total charge. This class provides both access and set functions to the data. The storage of an arbitrary signal is the purpose of this class. In order for full reconstruction of an analog form however, a functional form must also be specified. No matter, the final output of the `TRS` package is pixel data; that is a given amplitude at a given time which is indexed by a pad-row and pad number. Keeping the time allows the flexibility of zero-suppressing any data with a negligible amplitude. This structure is used to keep all the signals that are generated on the pad plane, both analog and digital. In this sense the term *analog* is probably a misnomer, but it is foreseen in the future to introduce another digital signal which does not require **float** precision of the stored values. This should improve the storage requirements for a single event.

8.2.6 Analog Sector

The final output of the simulator is pixel data which is indexed by pad-row, pad number, and perhaps time-bin. The sector is a container which can store an indeterminate number analog signals indexed by these quantities. As such after a specific charge is induced onto the cathode pads, the analog sector keeps track of all intermediate stages of calculation before digitization occurs. Please note that although both analog or digital signals can be stored, only analog information is stored within this structure. The data is accessible either by pad-row, or single pad indices. Iterators are also provided within the class to facilitate traversal of the structure.

8.2.7 Digital Sector

The final output of the simulator is pixel data which is indexed by pad-row, pad number. The digital sector is a container which can store an indeterminate number of digital (8-bit) signals indexed by these quantities. The complete simulation of a complete detector is 24 separate digital sectors.

8.3 Auxiliary Components

By itself the containers and processes must be supplemented by various administrative and book-keeping type classes. Classes are required in order to keep track of the containers as well as which type of simulation (i.e. fast or slow) is being used. This is the job of a program manager which takes the form the the StTrsMaker.cxx in the STAR environment, however, no STAR/ROOT specific functions that are used that require it to be run in this framework. The ”Maker” also functions as an event reader which provides the I/O interface between the raw input (GEANT) data and the output pixel data. As such, it is used as a ”writer” which can direct the produced data into the appropriate place. As with the manager, these classes can be provided externally. In addition to these types of classes, there is also a need for simple library functions which give access to experimental constants from data bases and a means to transform between the basic coordinate types. It was decided that such classes should also be developed in the context of TRS for STAR.

A prominent design goal of the TRS package was to have no hard coded numerical constants within the code. Rather they would be accessed from a data base, which was an undefined quantity at the beginning of the development period of TRS. Specifically it was not known whether access to the data would be through GEANT, ROOT, or a BABAR like conditional data base derived from Objectivity. In order to be flexible to simultaneous developments from the Grand Challenge developments which was charged with large scale data base access for the RHIC experiments, several data base interfaces classes were defined. These took the form as purely virtual abstract classes which defined the type and names of parameters to which TRS required access. An abstract class was used to ensure that independent of the implementation, that is whether an Objectivity, ROOT, GEANT, etc. was used, the same parameters and names would exist and this would make the code stable against evolving developments and implementations. The requirements for data bases can be split into four or five different types. These are:

- Geometry – geometrical parameters and dimensions of the detector, pad-plane, field cage, etc.
- Slow Control – parameters which must be monitored or controlled such as drift velocity, voltages, and environmental conditions.
- Electronics – all electronics parameters such as gains, sampling times, frequencies, gains, etc.
- Magnetic Field.
- Gas/Ionization requires knowledge of the parameters of the gas.

Along with these data base interfaces, various transformation routines between different coordinate systems were required. Currently three types of coordinates are implemented and used:

- TPC Raw Pad Coordinate – defined by a sector, pad row, pad, and time bucket.
- TPC Local Coordinate – defined by a three-vector with respect to an origin at the center of the TPC.
- STAR Global Coordinate – defined by a three-vector which is the global coordinate system, fixed by a survey of the detector with respect to the magnet iron.

Although the simulation is done within the TPC local coordinate system, there is a need to also define a global coordinate system as the magnetic field (as well as any other detector) will be linked through this coordinate system. It should be noted that the geometry for this transformation has not been implemented into TRS yet.

The coordinates and transformations between each system requires the knowledge of the geometry of the chamber in general. As such the coordinate transformations were designed so that any reference to these parameters would be through an interface which could access any Grand Challenge infrastructure defined for STAR in the future.

Although the coordinate transformations and data base interfaces are not specific to the TRS package, they were required before a real implementation could be developed. As such the implementation of these classes was done in a TPC specific manner, however the design incorporated ideas such that the framework could be easily extended to other detector sub-systems. It is foreseen that in the near future these classes (or subsequent versions) will be incorporated into the SCL with access to the official STAR data base infrastructure, rather than the simple TRS framework. Below the data base interfaces and the coordinate transformations are described in more detail.

8.3.1 Data Base Interfaces

The same common design for all data base interfaces was implemented. A purely abstract base class, from which any concrete class must inherit from, defined the necessary functionality of these access classes. This provides a constant user interface, independent of the evolution of the developments in data base implementation. This was seen as a necessity to keep TRS stable against this evolution. Each concrete class is implemented as a singleton class[10] for several reasons. First, in the initial implementation, these classes are initialized each time an instance is created and this can be very time consuming if all values must be assigned to data members within the class. Also because the many classes which model the physics require access to the data bases, it is necessary to ensure that several *different* data base implementations are **NOT** used simultaneously. A singleton class ensures the same functions are used homogeneously in the code. Thus for reasons of efficiency, and robustness it was decided to use this design pattern. It should be mentioned that this mechanism can be extended to allow for multiple versions of geometry which will be important once real survey data must be reconciled with experimental data.

In the current implementation of the simple data bases, the classes are a collection of inline access functions, along with some simple diagnostic features that allows the user to print out the values associated with all member functions. It should also be noted that to ensure consistency in the units, the `SystemOfUnits` class from the SCL are used through out.

Geometry– Contains the geometrical parameters which define the dimensions and construction of the detector. For the TPC, these parameters describe the field cage, pad plane, and wire chamber dimensions.

Slow Control– Contains all parameters which are monitored and/or controlled by the experiment. These include not only environmental parameters such as temperature and pressure but also detector operating conditions, such as voltage, drift velocity etc.

Electronics– All electronics parameters and specifications such as nominal gain, sampling time, shaping time, etc. are included. This interface can also be extended to include derived parameters such as channel gains and time offsets derived from calibration runs. It should be noted that these types of parameters may also be, or rather be, located in the Slow Controls data base if they are found to fluctuate over the course of the experiment. **NOTE:***The individual channel time offsets are not yet implemented.*

Magnetic Field– Provides access to the magnetic field components at any position specified by its global coordinate. It should be noted that it is currently assumed that the TPC Local Coordinate system and the

[10] see: E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995.

Global system coincide exactly.

Gas/Ionization– This is currently not contained in an interface but a generic stand-alone class which contains the gas parameter listings for three gases:

- Ar
- P10 (Ar:CH_4 90:10)
- $NeCO_2$ (90:10)

Parameters for a HeC_2H_6 (50:50) mixture are pending. Should it become necessary, the $\frac{dE}{dx}$ class can be split into two components—a gas data base and the calculation portion. Currently it is implemented as a single class, for reasons of efficiency. See 15.3 for more details.

8.3.2 Coordinate Transformations

The transformation between coordinate systems requires information of not only the geometrical layout of the detector, but also control parameters. It should be noted that even calibration offsets due to sector alignment and electronics offsets may also be important. Although it is possible to incorporate such values in the current design, they are not currently implemented.

In general, when a transformation is required, there is no need for knowledge of the intermediate steps of the calculations. One knows the name of the coordinate available, and the one required. In order to make the transformations simple, a functor[11] was defined which encapsulated all the real transformation calculations behind a single overloaded operator "()". This structure allows the same form of the function call between any of the six allowed transformations:

- TPC Raw ↔ TPC Local
- TPC Local ↔ Global
- TPC Raw ↔ Global

with the last being trivial calls of the former two. As mentioned previously, the transformations are dependent on parameters which are accessible via the data bases. Thus in order to instantiate the transformation functor, an `StTpcGeometry` and `StTpcSlowControl` data base (interface) needs to be specified. Currently the individual electronic channel time offsets are not implemented, but it is a straight forward addition.

9 Mode of Simulation

This section describes in detail the processes that are modeled, and the algorithms and methods that implement the physics. Technical details of the implementation are not included as this is discussed in sec-

[11] A generic name for a class which contains only functions with few or no data members.

tions 15–16. As mentioned previously a single process is mapped directly to a single class structure, where possible.

9.1 Ionization Process

The classes that deal with the raw ionization components within the package is capable of manipulating ionization distributions either generated from an external package (i.e. GEANT), or given the momentum vector of a particle, generate the ionization in the active detector volume internally according to basic parameters of the fill gas in the chamber.

In normal operations, the ionization for each TPC sub-volume is taken according to the `g2t_tpc_hit` structures. This large segment is then broken up into pieces, according to an exponential distribution which has a mean value characterized by the mean free path, between ionizing interactions, of a charged particle within the gas. This distribution, which specifies the distance to the next interaction is given by:

$$f(d) = Ke^{-d/\bar{d}} \tag{2}$$

where $\bar{d}$ is the mean free path. It is derived from the number of ionizing collisions which occur per unit distance, $\bar{n}$. The mean free path is then given by $1/\bar{n}$. This gives rise to a distribution of ionizing interactions per unit distance which is Poissonian in character.

At each interaction point a single "primary" electron is produced. These electrons will have an energy distribution given by the Rutherford scattering formula. In the case of free e^-e^- scattering (or at high momentum transfer where the binding energy can be neglected), this implies an energy (E) distribution which varies as E^{-2}. Because the electrons are not, strictly speaking, free but rather bound to the atoms of the medium (in our case the TPC gas) via a finite potential, there is a modification to this energy dependence which is medium dependent. This dependence has been measured experimentally and can be quite accurately parameterized by a function of the form E^{-n}.[12] For Ar, it is found experimentally that n=2 is a good approximation. A similar behavior is expected for P10 which is 90% Ar. For lighter gases, n increases. For example, Ne requires n=2.2, and He, n=2.6. This "stiffening" of the primary electron energy spectrum is a consequence of the higher binding energies of the lighter gases, however it has the desirable effect of decreasing the width of the Landau distribution as lower Z gases are used. The reason is that although fewer secondary electrons are produced, the spread in the energy distribution is smaller, and this compensates for the degradation in statistics. For this reason, resolution of dE/dx measurements are nearly constant, independent of the gas type used (to first order).

The energy dependence of the primary electrons is important because it determines the amount of subsequent ionization that is generated. For those primaries that have an energy above the ionization potential of the medium, which is typically the order of 10s of eV, subsequent electrons (secondaries) can be generated. The number of secondaries N_{sec} is then given by:

$$N_{sec} = (E_p - I_p)/W \tag{3}$$

where E_p is the energy of the primary electron, I_p is the ionization potential of the medium and W is the amount of energy necessary to create an $e^- - ion$ pair. This is an "effective" quantity which is very difficult to calculate from first principles and therefore must be measured for each gas mixture individually. It is found

[12] see: H. Fischle et al., NIM **A301** (1991) 202; and B. Lasiuk NIM **A409** (1998) 402.

that in most cases, except where the energy of the particle is very small, the amount of energy required to produce an $e^- - ion$ pair is independent of the energy of the incident particle. This is seen in figure 1.[13]

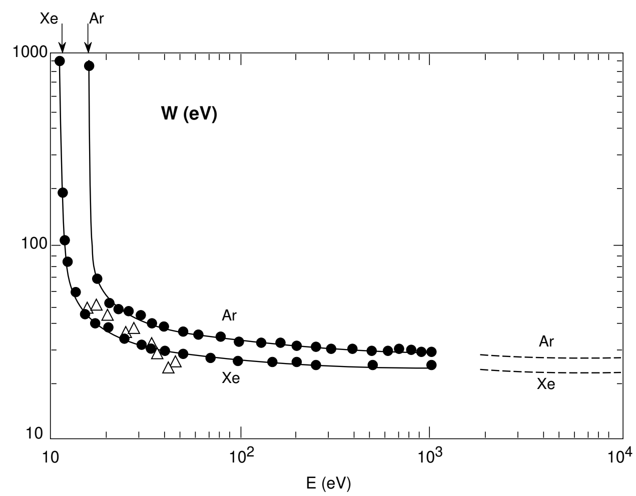

Figure 1: The average energy W spent (in eV) for the creation of one electron-ion pair in Ar and Xe as a function of the incident energy of the ionizing particle (in this case, an electron). Dashed lines are extrapolations to higher energies.

The total number of electrons produced per interaction is then given as the sum of the primaries and secondaries. Thus calculating the number of electrons generated in a specific length convolutes two distributions—a Poissonian which determines the number of interactions, and an E^{-n} which determines the majority of the yield. This procedure results in a Landau-like distribution—which is expected for the yield of ionization in a given path length. Results for Ar and Ne are shown in figure 2. Once the ionization is generated, or distributed into smaller segments, it may then be transported to the multi-wire proportional chamber (MWPC).

9.2 Charge Transporter

The charge, or ionization, in the segments that have been produced must be transported to the sense (anode) wires of the MWPC for read-out. This transport is modeled in the **Charge Transporter** component of TRS, and three distinct processes are described:

- E × B Effect.
- Transverse and Longitudinal Diffusion.
- Charge Loss *through*

[13] From W. Blum and L. Rolandi *Particle Detection with Drift Chambers* Springer-Verlag 1994.

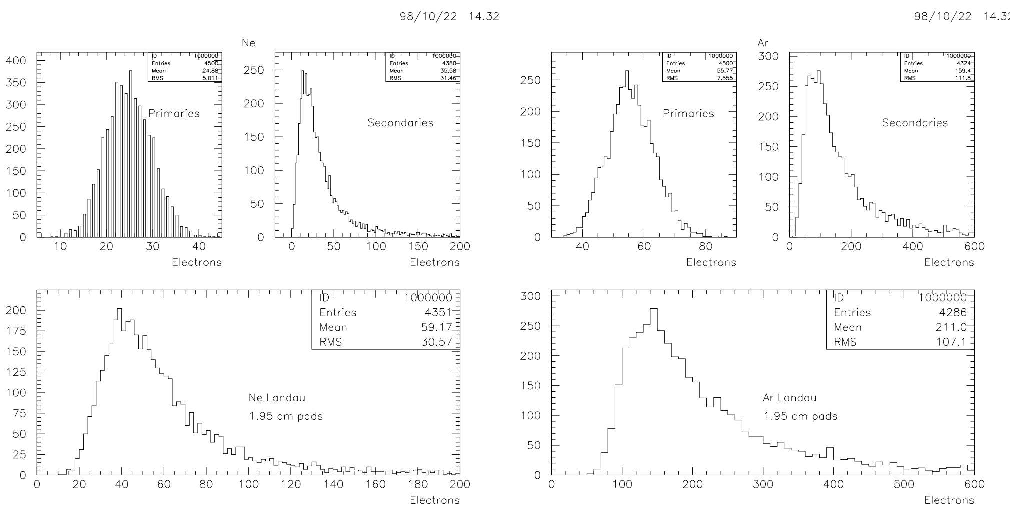

Figure 2: Ionization statistics for primary, secondary, and total yield distributions for Ne (left panel) and Ar (right panel). All yields are from a path length of 1.95 cm which is the pad length in the outer part of the TPC super-sector.

- Absorption through attachment (O_2).
- Gating Grid Transparency.

and it is possible to control each process independently. The charge transporter calculates the position of the charge segment/electron at the z-plane of the anode/sense wire grid and the amount of charge that arrives. This charge must then be collected and amplified on the anode wires. In the current implementation (i.e. *FAST*) the charge is simply projected in the z coordinate to the position of the anode/sense wire plane; the underlying assumption that the ionization follows the $\vec{E}$ field lines and no distortions are visible. This will be refined in future versions.

9.2.1 E $\times$ B Effect

NOTE: not currently implemented!

A misalignment between the electric and magnetic fields or inhomogeneities in the magnetic field can give rise to the ionization following trajectories which are not parallel to the $\vec{E}$ field lines. This introduces distortions into the trajectories of ionization left in the chamber. These are collectively known $\vec{E} \times \vec{B}$ distortions. In the general case of a charged particle in the presence of an electric and magnetic field, it will not follow a trajectory which is described by the Langevin equation:

$$\frac{d\mathbf{v}}{dt} = \frac{q}{m}(\mathbf{E} + \mathbf{v} \times \mathbf{B}) - \frac{1}{\tau}\mathbf{v} \tag{4}$$

where q is the charge of the particle, m is its mass, **v** is its velocity, **E** the electric (drift) field, **B** the magnetic field, and τ is the average time between collisions with the molecules in the medium. The last term is essentially a frictional force that limits the maximum average drift velocity. The steady state solution (i.e. $\frac{d\mathbf{v}}{dt} = 0$) is given by:

$$\mathbf{v} = \frac{\mu|\mathbf{E}|}{1+\omega^2\tau^2}(\hat{\mathbf{E}} + \omega\tau(\hat{\mathbf{E}} \times \hat{\mathbf{B}}) + \omega^2\tau^2(\hat{\mathbf{E}} \cdot \hat{\mathbf{B}})\hat{\mathbf{B}}) \tag{5}$$

where μ $(= \frac{e\tau}{m})$ is the electron mobility and ω $(= \frac{eB}{m})$ is the cyclotron frequency. Although in the ideal field cage structure of a TPC, the electric and magnetic fields are parallel and the drift direction is defined by the electric field vector. Deviations from this idealization introduces velocity components in the orthogonal directions.

Equation 5 allows the drift velocity to be calculated at any arbitrary spatial point given knowledge regarding the electro-magnetic field vectors and the mobility of electrons in the gas. The distortions can then be modeled either by numerically integrating the equation over the drift time of the ionization, or parameterizing the displacement of the ionization in the plane orthogonal to the $\vec{E}$ field.

9.2.2 Diffusion

The effects of diffusion are due to thermal motion of the molecules in the gas and multiple scattering of the ionization in the transport from the position of deposition in the chamber to the read-out or sense wires on the pad plane. While the mean position of a charge segment may be transported through a volume with arbitrary electro-magnetic fields according to equation 5, its profile will broaden in proportion to the drift length, or more appropriately, the square root of its drift length. It is possible to parameterize the evolution of the size of the charge distribution (its width) in terms of a diffusion coefficient, which is in principle, different in the transverse (σ_T) and longitudinal (σ_L) directions. This distinction is important because the evolution of the diffusion coefficient in the transverse direction also depends on the presence of an magnetic field where it is reduced according to:

$$\sigma_{\mathbf{T}}(\mathbf{B}) = \frac{\sigma_T}{1+\omega^2\tau^2} \tag{6}$$

Once the charge segment/cloud is transported to the read-out plane, each segment (or sub-component) can be distributed according to a Gaussian (or any other) distribution characterized by a width derived from the diffusion constants. Increasing the granularity of the charge distribution will better reproduce single electron statistical fluctuations.

9.2.3 Charge Absorption

The absorption of charge is a complex process that can be attributed to many different mechanisms.[14] For our application we consider a simple parameterization of charge attachment in a gas where trace amounts of oxygen are present. The probability (P) for attachment to occur in a specified drift time, t, is given by:

$$P(t) = 1 - e^{-At} \tag{7}$$

[14] Details contained in a soon to be added appendix.

where A specifies an *attachment rate* given by:

$$A = P_{O_2} \cdot P_M \cdot C_{O_2,M} \tag{8}$$

where P_{O2} and P_M are the partial pressures of the oxygen and TPC gas respectively. $C_{O2,M}$ specifies an *attachment coefficient* which is a function of the gas in question and the reduced electric field (E/p). For the case of STAR, a value of C = 10.2 μs^{-1} bar^{-2} has been deduced.[15] To give an idea of the order of magnitude, for a concentration of oxygen of 50 ppm and an ionization cloud drifting for 50 μs, the charge loss is expected to be approximately 2.5%. This charge loss can be applied at the single electron level or an extended charge cloud where a fraction of the total is lost.

9.2.4 Wire Grid Transparency

As the charge enters the region of the wire grids in the TPC, there is a non-zero probability that the charge may not be transmitted due to the potential configuration on the wire grids. In the STAR TPC there are three wire grids:

- anode/sense—responsible for charge collection and amplification.
- zero/frisch—defines the boundary of the field cage and allows a sink for ion collection in the amplification process.
- gate—switch which controls passage of ionization from the active TPC volume to the sense/anode wire plane.

The passage of ionization is controlled by the potential which is applied to the gate wires. Basically by setting the appropriate potential on this grid, the drift $\vec{E}$ lines can be made to terminate on the gating grid (zero transparency) or at the anode wires (full transparency). The transparency is the fraction of lines that terminate on the anode wires compared to those that terminate on the gating grid.

A general expression for the transparency of a mono-stable switched gating grid was deduced such that the full range from 0-100% transmission can be modeled. It should be noted that the effects of a bi-polar switching grid (which is the actual construction of the STAR-TPC) should not influence the values at more than the $\sim$5% level. The expressions involved require only the geometry of the wire grids and the voltages set on the gating grid and high-voltage plane. An example of the transmission curve is shown in figure 3.

9.3 Charge Collection

According to the position of the charge segment/electron at the anode wire plane, the ionization can be collected by the nearest wire. The appropriate time delays for charge collection should a segment not be projected directly on top of the wire position can be calculated. Once the charge is assigned to a wire, the gas amplification can occur.

[15] soon to be added appendix.

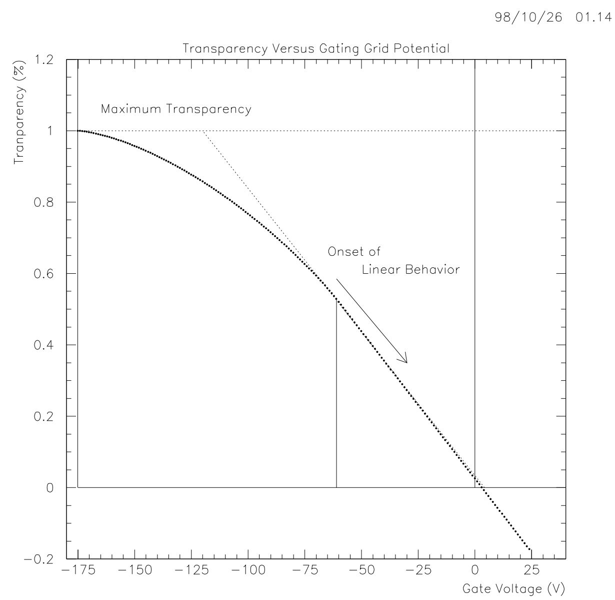

Figure 3: Wire Grid Transparency Calculation for nominal wire potentials and STAR outer sector read-out chamber geometry.

9.3.1 Gas Gain Amplification at Sense Wires

The process of gas amplification at the sense wires is modeled by the Raether distribution,[16] or, as it is denoted in Blum and Rolandi[17], the Yule-Furry process which describes the fluctuations in the amplification an amount of charge q with an exponential distribution:

$$p(q) = \frac{1}{\bar{q}} e^{-q/\bar{q}} \tag{9}$$

Subsequent theoretical refinements to this simple expression which were made in order to take into account effects like the asymmetric growth of the avalanche profile as well as saturation effects. This resulted in several distribution, the Polya distribution being the most popular. It requires an additional parameter to supplement the mean amplification ($\bar{q}$). The effect of the additional parameter is to suppress the small amplification factors. As this parameter tends to zero, the Raether distribution is recovered. Experimentally, perfectly exponential behavior is seen at low to moderate gas gains (i.e.$<10^4$), in parallel plate geometry, however at gas gains above 10^5, slight deformation from exponential shape is observed. This is probably attributable to self-saturation effects which become important because of space charge. Since the TPC is generally operated at low gas gains, the simple Raether distribution was deemed acceptable, however the Polya function may be substituted if there is need.

[16] see http://www.rhic.bnl.gov/STAR/html/comp_l/simu/TpcRespSim/src/literature.html
[17] W. Blum and L. Rolandi *Particle Detection with Drift Chambers* Springer-Verlag 1994.

It should be noted that in TPC (or any drift chamber) operation, the effect of the fluctuations in gas gain is to simply degrade the attainable space-point resolution, and for this purpose the exact functional form of the avalanche yields is not absolutely critical. This degradation is the physics that the implementation of the gas-gain fluctuations will attempt to address, and the Raether distribution should be quite acceptable in this regard.

Once the charge has been amplified, the amount of charge induced on the cathode pad plane can then be calculated.

9.4 Analog Signal Generation

The Analog Signal Generator has three main functions:

- Determine the charge induced on single pads from the charge “collected” on the anode wires.
- Sample the induced charge signals in time according to the electronics (i.e pre-amplifier/shaper) response.
- Distribute the **analog** charge into time bins according to the parameters of the switched capacitor array (SCA).

9.4.1 Charge Induction

The charge induced on a grounded pad plane by a point charge q located a distance d above the plane can be calculated by the method of images. The charge density (σ) on the plane is given by:

$$\sigma(x,y) = \frac{1}{2\pi}\frac{qd}{((x-x_o)^2+(y-y_o)^2+d^2)^{3/2}} \tag{10}$$

where the charge q is located at a position (x_o,y_o). However this expression is not the solution for the charge induced on the pad plane of a wire chamber. In the case of a MWPC the charge, which is at the position of the anode wire, is generally surrounded by *two* cathode planes—one from above and one from below.[18] Thus in order to calculate the charge density induced on the pad plane, all higher order multi-pole terms must be incorporated. This results in the following expression:

$$\sigma(x,y) = \frac{1}{2\pi}\sum_{n}^{\infty}\frac{(2n+1)qd}{((x-x_o)^2+(y-y_o)^2+((2n+1)d)^2)^{3/2}} \tag{11}$$

Doing the sum, (and integrating over all y) yields a function which describes the charge distribution induced on the pad plane at a distance x from the wire where x_o is the position of the charge q:

$$\sigma(x,y) = \frac{q}{4d\cosh(\frac{\pi(x-x_o)}{2d})} \tag{12}$$

This function, an inverse hyperbolic cosine as shown in equation 12 is called an *Endo function*. Note that the derivation put no limits on the extent of the pad in the y direction (i.e. $\frac{w}{l}=0$). The effect of finite geometry

[18] Also the charge is in the form of a line charge associated with a linear charge density, not a discrete point charge.

of segmented cathodes can be accounted for with the addition of another parameter. The same function can be written in a form more instructive for our purposes:

$$\sigma(x) = K_1 \frac{1 - \tanh^2(\frac{\pi(x-x_o)}{4d})}{1 + \tanh^2(\frac{\pi(x-x_o)}{4d})} \tag{13}$$

where K_1 is a normalization constant. Equation 13 can be used to generalize the Endo function in order to account for finite geometry effects of the segmented cathode pads. That is, each pad has a finite width/length ratio. This can be accounted for with the addition of another constant K_2:

$$F(x) = K_1 \frac{1 - \tanh^2(\frac{\pi(x-x_o)}{4d})}{1 + K_2 \tanh^2(\frac{\pi(x-x_o)}{4d})} \tag{14}$$

This is the generalized solution to the distribution of charge induced on a grounded pad plane by a point/line charge, and is usually dubbed, the Gatti function. The Gatti function is most prevalently used for pad chamber responses. Although more accurate, the price that is paid is the increase in time required to integrate equation 14 over the rather simple expression which can be deduced from the integral of 12 A comparison of the Gatti and Endo functions to a Gaussian (for reference) are given in figure 4

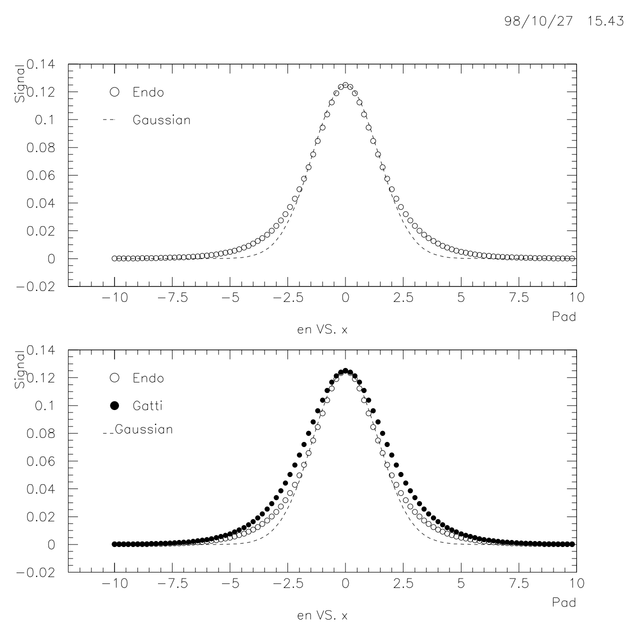

Figure 4: Comparison of Gaussian, Endo, and Gatti functions for profiles of the pad-response-function.

For the case of the TPC, the quantity of interest is the total amount of charge (Q) induced per pad which means one must integrate these functions, which specify the charge density, according to the pad dimensions.

Thus for a point charge (equation 10), the integral is:

$$Q = \frac{1}{2\pi} \int_{yl}^{yu} \int_{xl}^{xu} dxdy \frac{qd}{((x - x_o)^2 + (y - y_o)^2 + d^2)^{3/2}} \tag{15}$$

where xl and xu denotes the lower and upper bounds of the pad in the x direction respectively. Similarly yl and yu denotes the same in the y direction. Similar integrals can be constructed given any arbitrary charge density (σ). The advantage of using such functions is that they allow the production of longer tails which have non-Gaussian characteristics. The tails are an important characteristic to understand as they determine the efficiency of the ionization collection which is very important in the study of ionization collection for $\frac{dE}{dx}$ resolution. Furthermore these integrals are well defined given the the position of the charge and the coordinates of the pads. Thus charge induced on adjacent pads as well as rows can be also be calculated quite simply in this formalism. Illustrations of some of these function are shown in figures 4.[19]

Please keep in mind that the fact that the signal on the wire is due almost exclusively to the motion of the charged ions **away** from the wire. In order to calculate the total amount of charge induced on the pad however, it is possible to use the number of electrons as a quantity, even though they are not the physical reason for the charge induction. This point will be reiterated in the electronics response of this description.

9.4.2 Sample the Signal in Time

Once the amount and centroid of the charge distribution on each pad is determined, this charge can be sampled in time corresponding to the analog electronics response. The signals are generated by super-imposing each analog signal from each avalanche which induces a signal on the pad plane. This allows the shaping time of the electronics can be varied independently of the width of the pad response function which is the strength of this simulation methodology. Currently four types of sampling are possible:

- delta function.
- Symmetric Gaussian.
- Asymmetric Gaussian.
- Parameterized STAR response.

A function also exists where a fractional scale of the total charge integral can be added as an Undershoot/Unrestored baseline component in the signal which has the effect of convoluting effects from the long 1/T tail. This is an important point and one that actually blurs the line of physics and simulation. In reality the time evolution of the signal that is developed on the wires is nearly entirely due to the **motion of the positive ions** away from the wire. This produces a signal with a long 1/T tail. For the STAR geometry the signal is $\sim$62 μs. In order to make a detector faster, the signal is differentiated after a characteristic time—the shaping time of the pre-amplifier. Although the detector response becomes faster, the trade-off is that only a fraction of the total charge is seen by the downstream electronics (i.e. the ADC). This fraction F, is given by:

$$F = \frac{ln(1 + \frac{t_m}{t_o})}{ln(1 + \frac{t_s}{t_o})} \tag{16}$$

[19] For more details see: http://www.rhic.bnl.gov/STAR/html/comp_l/simu/TpcRespSim/src/literature.html

where t_m is the length of time the undifferentiated signal would persist (i.e. $\sim$62 μs), t_o is the characteristic time of the signal development (i.e. $\sim$1 ns), and t_s is the shaping time of the pre-amplifier (i.e. $\sim$180 ns). For STAR this implies the order of 45% of the charge is distributed into time bins by the SCA. More importantly is that the signal shape is dominated very strongly by the shaping properties of the electronics. Thus the long time response of the chamber is parameterized in the electronics processing component of the simulator, rather than modeling the motion of the positive ions.

The symmetric Gaussian response, with the effect of an unrestored baseline due to under/over shoot is illustrated in figure 5 where the time evolution of a series of 20 signals, of identical amplitude, are induced

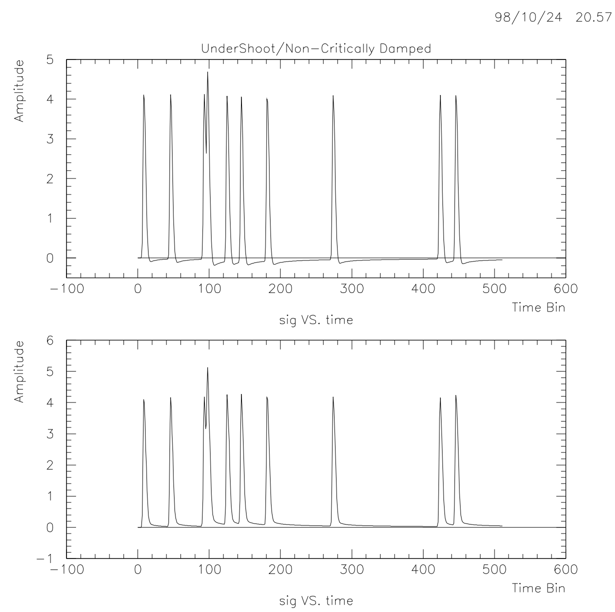

Figure 5: Asymmetric Gaussian Electronics Response with pedestal suppressed showing both undershoot (top panel) and an under damped baseline restoration (bottom panel).

on a single pad.[20] The pulses are simply added using the principle of super-position. The modeling of an unrestored baseline is very important since the pulse height (actually the integral) of the signal is used as a measure of the velocity of the particle (via $\frac{dE}{dx}$ information). An unrestored baseline due to undershoot can result in an effective loss of charge in a high-multiplicity environment. Conversely an unrestored baseline due to under-damping will result in too much charge being observed at large drift distances. Both effects will reduce the attainable resolution if not taken into account.

Once the complete functional form of the signals induced on a pad over the read-out period of the TPC electronics, the analog charge can be distributed into discrete time bins. This sampling simulates the behavior of the switched capacitor array (SCA) in the front-end electronics. This is done by integrating the amount of charge in a time interval Δt, which is determined from the SCA sampling frequency. Chamber noise as

[20] More examples can be seen at: http://www.rhic.bnl.gov/STAR/html/comp_l/simu/TpcRespSim/src/literature.html

well as electronic noise can be added at this point.

9.5 Digital Signal Generation

Once the analog charge is distributed into time bins on the pads, the digitization can occur. This is currently a simple conversion from voltage to ADC counts. Other features such as the addition of a pedestal and non-linearities in the ADC can also be added. In fact, any characteristic of the digital electronics can be added and will be independent of the analog signal sampling. This is very close to the way the front-end electronics are designed where the digital and analog electronic components are separated.

10 Physics Limitations

There are still some components either not implemented, or completely missing from TRS in its current form. It is hoped that these will be incorporated as the understanding of TRS evolves. Most critically is the noise. Although some preliminary work exists for this modeling, it has currently not reached a mature enough state where it can be added. Some obvious short-comings are listed below.

- $\vec{E} \times \vec{B}$ in not tested although code exists and can be ”plugged” in.
- Noise at the chamber and electronics level does not yet exist.
- A function (or look-up table) is needed for the non-linear ADC response.
- The pad geometry is not incorporating the fractional pads at the border of the sectors.
- The gas gain is currently independent of the wire and the position of the avalanche on the wire.
- Effects of space charge in the charge transport stage of the simulation.
- No positional dependence in the transparency of the gating grid exists. Currently a constant transparency, independent of position is calculated.

It is almost certain that as experience is gained with the package, this list will be further expanded, however the code is written in a flexible manner such that these type of additions should be relatively straight forward.

11 Known Limitations

To be seen...

12 Support and Reporting Bugs

Currently TRS is supported by a small group. Hopefully as more people begin to use it and add to it, there will be a larger support base for it. If there is a problem or bug, report it to one or more of the following:

- starsas-l@bnl.gov
- starsofi-l@bnl.gov
- startpc-l@bnl.gov
- brian.lasiuk@yale.edu
- thomas.ullrich@yale.edu

Part II

Reference Manual

This reference manual is broken into six sections. The first section describe the *Data Base interfaces and implementations* within TRS. The second describes the *Coordinate systems* and the *Coordinate transformation functor*. The next two sections describe the *Containers* and *Physics Processes* that are accessible and utilized in TRS. The fifth section is not yet complete but will include the *Administrative* type classes that are utilized while the final section contains an *Example* and describes how the package is used in an example program. Within the categories, as much as possible, the listing of classes are alphabetical, and closely related classes which either inherit or are derived are cross-referenced.

As mentioned previously, heavy use of the SCL is made within TRS, so that manual is also a useful reference. It can be found at: http://star.physics.yale.edu/SCL. As such, many conventions adopted in the SCL are carried through into TRS in order to ensure compatibility. As **namespaces** are not currently supported widely among vendor or commercial compilers, the current STAR conventions advocate the addition of prefixes to indicate the scope of a class. **St** denotes all STAR specific classes. This prefix is supplemented with **Trs** to denote TRS specific classes where appropriate. As an extension, the prefix **Tpc** is also used occasionally.

13 Data Base and Interfaces

Following are descriptions of the four different data base interfaces and implementations.

13.1 StMagneticField

Summary Interface which defines access methods to the magnetic field components.

Synopsis Purely abstract class, no instantiation is possible.

Description Class StMagneticField is an abstract class that defines the interface that is used to access the magnetic field components.

Persistence None

Related Classes The implementation of the magnetic field data base is done in the class **StSimpleMagneticField**. See section 13.2.

Dependencies Requires `StGlobalCoordinate` from TRS`StThreeVector` from the SCL .

Public Constructors None

Public Virtual Operators None

Public Virtual Member Functions `virtual const StThreeVector<double>& at(const StGlobalCoordinate& gp)`
Provides access to the magnetic field components in an StThreeVector at a position, gp specified in the STAR global coordinate system.

13.2 StSimpleMagneticField

Summary Implementation of a simple data base which provides the magnetic field components which are read from an ASCII file.

Synopsis `#include "StSimpleMagneticField.hh"`
requires `StGlobalCoordinates.hh` from TRS. `StThreeVector.hh` as well as `StGetConfigValue` are required from the SCL. The `SystemOfUnits` class in the SCL is also used to ensure a uniform usage of the unit types.

Description Class StSimpleMagneticField is a concrete class that implements methods defined by the abstract base class `StMagneticField` (see section 13.1). Values of the field are accessed given a global coordinate (`StGlobalCoordinate`) as described in section 14.1. The field value is returned as a `StThreeVector` which is currently of double precision. The implementation of the ”simple” data base uses the `StGetConfigValue` utility from the SCL which parses an ASCII file which reads the numerical value of the magnetic field specified by a key word. The field values are kept as data members, and the access functions simply return the value. This initialization is done in the *private* constructor(s). Currently a constant field with the field components specified as (B_x,B_y,B_z) given as (0,0,.5) Tesla, at any coordinate, is implemented. The data base is implemented as a singleton class which protects the code against multiple distinct copies of the data base parameters in the code. As such the class constructors are implemented as *private* data members which are called via a *public* member function:

Persistence None

Related Classes The base class which defines the interface is specified in the class `StMagneticField`. See section 13.1.

Public Constructors

`static StMagneticField* instance(const char* file)`
Returns a pointer of type `StMagneticField`. The *static* designation implies at most, one instance of this can occur. The pointer will be returned if and only if a file name (file) suitable to initialize the class is specified. Such a file is provided in TRS in the **run** directory (run/example.conf). Subsequent declarations of magnetic fields can be made in the code, but once the first instance is created, the same pointer will be returned.

```
static StMagneticField*
instance(const StThreeVector<double>& B)
```
Returns a pointer of type `StMagneticField`. The *static* designation implies at most, one instance of this can occur. The pointer will be returned if and only if an StThreeVector, B is supplied which will specify the field components at every spatial location. This is useful if a constant field is required. Subsequent declarations of magnetic fields can be made in the code, but once the first instance is created, the same pointer will be returned.

`static StMagneticField* instance()`
Returns the pointer of type StMagneticField should an instance of the class have

been made previously, otherwise a filename or three-vector need be specified for the first initialization.

Private Constructor

The constructors are hidden from direct call to ensure that only one instance of the data base is made. As such, the constructors are only called through member functions as described above. The *actual* constructors which are called are:

`StSimpleMagneticField(const char* file)`
Called from the member function `instance(const char* file)` which is only invoked if a previous instance is not detected. The parsing of the ASCII file is done in this constructor to initialize the data members. The file is provided in the **run** directory of the package.

`StSimpleMagneticField(const StThreeVector<double>& B)`
Called from the member function *instance(const StThreeVector<double>& B)* if and only if a previous instance is not detected. The data member which stores the magnetic field is initialized in this constructor.

`StSimpleMagneticField()`
Never called, rather the member function `instance()` returns the `StMagneticField` pointer which was created by a previous instance. No initialization of data members is done in this constructor.

Public Operators

None

Public Member Functions

Following are implementations of the functions defined in the interface `StMagneticField`.

```
const StThreeVector<double>&
          at(const StGlobalCoordinate& gp) const
```
Returns the magnetic field components in an `StThreeVector` at a position, gp specified in the STAR global coordinate system.

```
const StThreeVector<double>&
          at(const StThreeVector<double>& gp) const
```
Returns the magnetic field components in an `StThreeVector` at a position, gp specified in the STAR global coordinate system.

Examples

```
#include <iostream.h>
#include <unistd.h>    // needed for access()
#include <string>

// SCL
#include "SystemOfUnits.h"

// TRS
#include "StCoordinates.hh"
#include "StSimpleMagneticField.hh"

int main()
{
    // Check File access

    string magFile("../run/example.conf");
```

```
    if (access(magFile.c_str(),R_OK)) {
        cerr << "ERROR:\n" << magFile << " cannot be opened" << endl;
        exit(1);
    }

    // Create an instance of the DataBase

    StMagneticField *magDb =
        StSimpleMagneticField::instance(scFile.c_str());

   // Print the data base to the screen

   magDb->print();

   // Access to the field at a spatial point

   StGlobalCoordinate
      myCoordinate(0.*centimeter, 0.*centimeter, 0.*centimeter);
   StThreeVector<double> field = magDb->at(myCoordinate);

   cout << "Magnetic Field at " << myCoordinate << " is "
        << field << " T." << endl;

    return 0;
}
```

Programs Output:

```
To be run
```

13.3 StTpcElectronics

Summary Interface which defines access functions to electronics specific constants or parameters.

Synopsis Purely abstract class, no instantiation is possible.

Description Class `StTpcElectronics` is an abstract base class that defines the interface that is used to access all TPC electronics related parameters. This includes both the analog and digital components of the TPC electronics.

Persistence None

Related Classes The implementation of the electronics data base is done in the class `StTpcSimpleElectronics`.

Public Constructors None

Public Virtual Operators None

Public Virtual Member Functions

Analog Electronics

`virtual double nominalGain() const`
Provides access to the nominal gain of the pre-amplifier in mV/fC.

`virtual double channelGain(int s,int r,int p) const`
Provides access to the individual channel gain of the pre-amplifier indexed by the sector (s), pad row (r), and pad (p). This function should be more relevant once an electronics gain calibration has been done.

`virtual double channelGain(StTpcPadCoordinate& c) const`
Provides access to the individual channel gain of the pre-amplifier indexed given a StTpcPadCoordinate (c). See section 14.4.

`virtual double shapingTime() const`
Provides access to the shaping time of the pre-amplifier shaper.

`virtual double samplingFrequency() const`
Provides access to the sampling frequency of the switched capacitor array (SCA) .

Digital Electronics

`virtual double adcConversion() const`
Provides access to the nominal ADC conversion value.

`virtual double adcConversionCharge() const`
Provides access to the nominal ADC conversion value.

`virtual int averagePedestal() const`
Provides access to the nominal pedestal value for the ADC.

`virtual int pedestal(int s,int r, int p, int t) const`
Provides access to the pedestal value for a single ADC channel indexed by a sector (s), pad row (r), pad (p), and time bin (t). Should be more relevant once a real electronics calibration has been done.

`virtual int pedestal(StTpcPadCoordinate&) const`
Provides access to the pedestal value for a single ADC channel specified by a raw pad coordinate. See section refsec:rawCoordinate.

Diagnostic

`virtual void print(ostream& = cout) const`
Prints the values of all constants accessible by public access functions to an output file stream. The default is the screen.

Examples None

13.4 StTpcGeometry

Summary Interface which defines access functions to geometrical specific constants or parameters.

Synopsis Purely abstract class; instantiation is not possible.

Description Class `StTpcGeometry` is an abstract base class that defines the interface that is used to access all geometrical parameters related to the TPC including the field cage, pad-plane, and read-out chamber.

Persistence None

Related Classes The implementation of the geometry data base is done in the class `StTpcSimpleGeometry`.

Public Constructors None

Public Virtual Operators None

Public Virtual Member Functions

Rows

`virtual int numberOfRows() const`
Provides access to the number of rows in a single super sector.

`virtual int numberOfInnerRows() const`
Provides access to the number of rows in the inner part of a super sector.

`virtual int numberOfInnerRows48() const`
Provides access to the number of rows in the inner part of a super sector where the row pitch is 48 mm.

`virtual int numberOfInnerRows52() const`
Provides access to the number of rows in the inner part of a super sector where the row pitch is 52 mm.

`virtual int numberOfOuterRows() const`
Provides access to the number of rows in the outer part of a super sector.

`virtual double innerSectorRowPitch1() const`
Provides access to the pitch of the rows in the 8 innermost pad rows of the inner super sector.

`virtual double innerSectorRowPitch2() const`
Provides access to the pitch of the rows in the 5 outermost pad rows of the inner super sector.

`virtual double outerSectorRowPitch() const`
Provides access to the pitch of the rows in the outer part of a super sector.

`virtual int numberOfPadsAtRow(int r) const`
Provides access to the total number of pads in row, r of a super sector.

`virtual double radialDistanceAtRow(int r)`
Provides access to the radial distance to the center of the mid-point of a pad row, r in a super sector.

Time buckets
`virtual int numberOfTimeBuckets() const`
Provides access to the number of time bins on a single pad.

Pads
`virtual double innerSectorPadWidth() const`
Provides access to the geometric width of a single pad in the inner part of a super sector.

`virtual double outerSectorPadWidth() const`
Provides access to the geometric width of a single pad in the outer part of a super sector.

`virtual double innerSectorPadLength() const`
Provides access to the geometric length of a single pad in the inner part of a super sector.

`virtual double outerSectorPadLength() const`
Provides access to the geometric length of a single pad in the outer part of a super sector.

`virtual double innerSectorPadPitch() const`
Provides access to the pitch of the pads in the inner part of a super sector.

`virtual double outerSectorPadPitch() const`
Provides access to the pitch of the pads in the outer part of a super sector.

Sector Dimensions
`virtual double innerSectorEdge() const`
Provides access to the radial distance of the edge of the inner part of a super sector closest to the inner field cage.

`virtual double outerSectorEdge() const`
Provides access to the radial distance of the edge of the outer part of a super sector closest to the outer field cage.

`virtual double ioSectorSpacing() const`
Provides access to the distance between the inner and outer parts of a super sector.

Wire Plane
`virtual double anodeWireRadius() const`
Provides access to the radius of the anode sense wires of the MWPC.

`virtual double frischGridWireRadius() const`
Provides access to the radius of the wires which make up the zero potential wire grid (frisch grid) of the MWPC.

`virtual double gateWireRadius() const`
Provides access to the radius of the wires which make up the gating grid of the MWPC.

`virtual double anodeWirePitch() const`
Provides access to the pitch of the anode wires in the MWPC.

`virtual double frischGridPitch() const`
Provides access to the pitch of the wires which make up the zero potential (frisch) grid of the MWPC.

`virtual double gatePitch() const`
Provides access to the pitch of the wires which make up the gating grid of the MWPC.

`virtual double innerSectorAnodeWirePadPlaneSeparation() const`
Provides access to the separation distance between the anode wires and the pad plane in the inner part of a super sector.

`virtual double innerSectorFrischGridPadPlaneSeparation() const`
Provides access to the separation distance between the zero potential (frisch) wire grid and the pad plane in the inner part of a super sector.

`virtual double innerSectorGatingGridPadPlaneSeparation() const`
Provides access to the separation distance between the gating grid wire grid and the pad plane in the inner part of a super sector.

`virtual double outerSectorAnodeWirePadPlaneSeparation() const`
Provides access to the separation distance between the anode wires and the pad plane in the outer part of the super sector.

`virtual double outerSectorFrischGridPadPlaneSeparation() const`
Provides access to the separation distance between the zero potential (frisch) wire grid and the pad plane in the outer part of the super sector.

`virtual double outerSectorGatingGridPadPlaneSeparation() const`
Provides access to the separation distance between the gating grid wire grid and the pad plane in the outer part of the super sector.

`virtual int numberOfInnerSectorAnodeWires() const`
Provides access to the number of anode wires in the inner part of the super sector.

`virtual double firstInnerSectorAnodeWire() const`
Provides access to the radial distance to the first anode wire at the centroid of the inner part of a super sector.

`virtual double lastInnerSectorAnodeWire() const`
Provides access to the radial distance to the last anode wire at the centroid of the inner part of a super sector.

`virtual double`
` innerSectorAnodeWire(int w) const`
Provides access to the radial distance to the anode wire number, w.

`virtual int`
` numberOfOuterSectorAnodeWires() const`
Provides access to the number of anode wires in the outer part of a super sector.

`virtual double`
` firstOuterSectorAnodeWire() const`
Provides access to the radial distance of the first anode wire in the outer part of a super sector.

`virtual double`
` lastOuterSectorAnodeWire() const`
Provides access to the radial distance to the last anode wire in the outer part of a super sector.

`virtual double`
` outerSectorAnodeWire(int w)`
Provides access to the radial distance to the anode wire w in the outer part of a super sector.

General – Field Cage
`virtual double frischGrid() const`
Provides access to the z-position of the zero potential (frisch) grid with respect to center of the TPC volume.

`virtual double driftDistance() const`
Provides access to the maximum drift distance of ionization within the field cage volume.

`virtual double ifcRadius() const`
Provides access to the radial distance to the inner field cage electrodes.

`virtual double ofcRadius() const`
Provides access to the radial distance to the outer field cage electrodes.

`virtual double endCapZ() const`
Provides access to the z-coordinate of the end-cap of the TPC.

`virtual bool`
` acceptance(StThreeVector<double>& c) const`
Provides a boolean value that indicates whether a position as specified by the `StThreeVector`, c lies within the confines of the TPC field cage/active-volume.

`virtual void print(ostream& os = cout) const`
Diagnostic function which prints all constants within the data base accessible by member functions to a file stream. The default is the screen.

13.5 StTpcSimpleElectronics

Summary

Implementation of a simple data base which provides electronics specific constants or parameters which are read from an ASCII file.

Synopsis

`#include "StTpcSimpleElectronics.hh"`
Requires `StGetConfigValue` from the SCL. The `SystemOfUnits` class in the SCL is also used to ensure a uniform usage of the unit types.

Description

Class `StTpcSimpleElectronics` is a concrete class which implements methods for the access functions defined in the abstract base class `StTpcElectronics`. The implementation of the "simple" data base uses the `StGetConfigValue` utility from the SCL which parses an ASCII file, provided in the **run** directory (run/electronics.conf) and reads the numerical value of the parameters which are specified by a key word. The parameters are kept as data members, and the access functions simply return these values. The data base is implemented as a singleton class which protects the code against multiple distinct copies of the data base parameters in the code. As such the class constructors are implemented as *private* data members which are called via a public member function.

Persistence

None

Related Classes

The base class which defines the interface is specified in the class `StTpcElectronics`.

Public Constructors

`static StTpcElectronics* instance(const char* file)`
Returns a pointer of type `StTpcElectronics`. The *static* designation implies at most, one instance of this can occur. The pointer will be returned if and only if a file name (file) suitable to initialize the class is specified. Such a file is provided in TRS in the **run** directory. Subsequent declarations of magnetic fields can be made in the code, but once the first instance is created, the same pointer will be returned.

`static StTpcElectronics* instance()`
Returns the pointer of type `StTpcElectronics` should an instance of the class have been made previously, otherwise a filename must be specified for the first initialization.

Private Constructors

The constructors are hidden from direct call to ensure that only one instance of the data base is made. As such, the constructors are only called through member functions as described above. The constructors which can be called are:

`StTpcSimpleElectronics(const char* file)`
Called from the member function `instance(const char* file)` which is only invoked if a previous instance is not detected. The parsing of the ASCII file is done in this constructor to initialize the data members.

`StTpcSimpleElectronics()`
Never called; rather the member function `instance()` returns the `StElectronics`

pointer which was created by the previous instance. No initialization of data members is done.

Public Operators

None

Public Member Functions

Following are implementations of the functions defined in the interface `StTpcElectronics`.

Analog Electronics

`double nominalGain() const`
Returns the nominal gain of the pre-amplifier. The value of 16 mV/fC is taken from STAR Note #230.

`double channelGain(int s,int r,int p) const`
Returns the individual channel gain of the pre-amplifier in mV/fC indexed by the sector (s), pad row (r), and pad (p). Should take on more importance when the first calibration runs for the electronics are available.

`double channelGain(StTpcPadCoordinate& c) const`
Returns the individual channel gain of the pre-amplifier in mV/fC indexed by given a StTpcPadCoordinate, c. See section 14.4. Should take on more importance when the first calibration runs for the electronics are available.

`double shapingTime() const`
Returns the shaping time of the pre-amplifier shaper. A value of 180 ns is specified in STAR Note #230.

`double samplingFrequency() const`
Returns the sampling frequency of the switched capacitor array (SCA) . The value of 16 mV/fC is taken from STAR Note #230.

Digital Electronics

`double adcConversion() const`
Returns the nominal ADC conversion value.

`double adcConversionCharge() const`
Returns the nominal ADC conversion value.

`int averagePedestal() const`
Returns the nominal pedestal value for the ADC.

`int pedestal(int s, int r, int p, int t) const`
Returns the pedestal value for a single ADC channel indexed by a sector (s), pad row (r), pad (p), and time bin (t). Should take on more importance when the electronics calibration is done.

`int pedestal(StTpcPadCoordinate&) const`
Returns the pedestal value for a single ADC channel specified by a raw pad coordinate. See section 14.4.

Diagnostic

`void print(ostream& = cout) const`
Prints the values of all constants accessible by public access functions to an output file stream. The default is the screen.

Examples

```
#include <iostream.h>
#include <unistd.h>    // needed for access()

#include <string>

#include "StTpcSimpleElectronics.hh"

int main ()
{
    // Check File access

    string electronicsFile("../run/electronics.conf");
    if (access(electronicsFile.c_str(),R_OK)) {
        cerr << "ERROR:\n" << electronicsFile << " cannot be opened" << endl;
        cerr << "Exitting..." << endl;
        exit(1);
    }

    // Instantiation of the DataBase

    StTpcElectronics *electronicsDb =
        StTpcSimpleElectronics::instance(electronicsFile.c_str());

    // print out the parameters contained in the db

    electronicsDb->print();

    return 0;
}
```

Programs Output:

```
To be run
```

13.6 StTpcSimpleGeometry

Summary Implementation of a simple data base which provides geometrical parameters specific to the TPC field cage, pad-plane, and wire-planes.

Synopsis `#include "StTpcSimpleGeometry.hh"`
Requires `StThreeVector` class as well as `StGlobals` from the SCL . As with the other data base implementations the `SystemOfUnits` from the SCL is also used to ensure consistent set of units is utilized. .

Description Class `StTpcSimpleGeometry` is a concrete class which implements methods for the access functions defined in the abstract base class `StTpcGeometry`. The implementation of the "simple" data base uses the `StGetConfigValue` utility from the SCL which parses an ASCII file, contained in the **run** directory (run/TPCgeo.conf) and reads the numerical value of the parameters which are specified by a key word. These parameters are taken from two engineering drawings.[21] The parameters are kept as data members, and the access functions simply return these values. This initialization is done in the *private* constructors.

Persistence None

Related Classes The implementation of the geometry data base is done in the class **StTpcSimpleGeometry**.

Public Constructors The data base is implemented as a singleton class which protects the code against multiple distinct copies of the data base parameters in the code. As such the class constructors are implemented as *private* data members which are called via a public member function:

`static StTpcGeometry* instance(const char* file)`
Returns a pointer of type `StTpcGeometry`. The *static* designation implies at most, one instance of this can occur. The pointer will be returned if and only if a file name (file) suitable to initialize the class is specified. Such a file is provided in TRS in the **run** directory (run/TPCgeo.conf). Subsequent declarations of magnetic fields can be made in the code, but once the first instance is created, the same pointer will be returned.

`static StTpcGeometry* instance()`
Returns the pointer of type `StTpcGeometry` should an instance of the class have been made previously, otherwise a filename must be specified for the first initialization.

Private Constructors The constructors are hidden from direct call to ensure that only one instance of the data base is made. As such, the constructors are only called through member functions as described above. The constructors which can be called are:

`StTpcSimpleGeometry(const char* file)`
Called from the member function `instance(const char* file)` which is

[21] RHIC-STAR-TPC Inner and Outer Sector Pad Plane Configuration. DWG: 24A0221 Rev B, Rev C. Accessible at: http://www.rhic.bnl.gov/STAR/html/tpc_l/tpc.html

only invoked if a previous instance is not detected. The parsing of the ASCII file is done in this constructor to initialize the data members.

`StTpcSimpleGeometry()`
Never called; rather the member function `instance()` returns the `StTpcGeometry` pointer which was created by the previous instance. No initialization of data members is done.

Public Operators

None

Public Member Functions

Following are implementations of the functions defined in the interface `StTpcGeometry`.

Rows

`int numberOfRows() const`
Returns the number of rows in a single super sector.

`int numberOfInnerRows() const`
Returns the number of rows in the inner part of a super sector.

`int numberOfInnerRows48() const`
Returns the number of rows in the inner part of a super sector where the row pitch is 48 mm.

`int numberOfInnerRows52() const`
Returns the number of rows in the inner part of a super sector where the row pitch is 52 mm.

`int numberOfOuterRows() const`
Returns the number of rows in the outer part of a super sector.

`double innerSectorRowPitch1() const`
Returns the pitch of the rows in the 8 innermost pad rows of the inner super sector.

`double innerSectorRowPitch2() const`
Returns the pitch of the rows in the 5 outermost pad rows of the inner super sector.

`double outerSectorRowPitch() const`
Returns the pitch of the rows in the outer part of a super sector.

`int numberOfPadsAtRow(int r) const`
Returns the total number of pads in row, r of a super sector.

`double radialDistanceAtRow(int r)`
Returns the radial distance to the center of the mid-point of a pad row r in a super sector.

Time buckets

`int numberOfTimeBuckets() const`
Returns the number of time bins on a single pad.

Pads

`double innerSectorPadWidth() const`
Returns the geometric width of a single pad in the inner part of a super sector.

`double outerSectorPadWidth() const`
Returns the geometric width of a single pad in the outer part of a super sector.

`double innerSectorPadLength() const`
Returns the geometric length of a single pad in the inner part of a super sector.

`double outerSectorPadLength() const`
Returns the geometric length of a single pad in the outer part of a super sector.

`double innerSectorPadPitch() const`
Returns the pitch of the pads in the inner part of a super sector.

`double outerSectorPadPitch() const`
Returns the pitch of the pads in the outer part of a super sector.

Sector Dimensions

`double innerSectorEdge() const`
Returns the radial distance of the edge of the inner part of a super sector closest to the inner field cage.

`double outerSectorEdge() const`
Returns the radial distance of the edge of the outer part of a super sector closest to the outer field cage.

`double ioSectorSpacing() const`
Returns the distance between the inner and outer parts of a super sector.

Wire Plane

`double anodeWireRadius() const`
Returns the radius of the anode sense wires of the MWPC.

`double frischGridWireRadius() const`
Returns the radius of the wires which make up the zero potential wire grid (frisch grid) of the MWPC.

`double gateWireRadius() const`
Returns the radius of the wires which make up the gating grid of the MWPC.

`double anodeWirePitch() const`
Returns the pitch of the anode wires in the MWPC.

`double frischGridPitch() const`
Returns the pitch of the wires which make up the zero potential (frisch) grid of the MWPC.

`double gatePitch() const`
Returns the pitch of the wires which make up the gating grid of the MWPC.

```
double
  innerSectorAnodeWirePadPlaneSeparation() const
```
Returns the separation distance between the anode wires and the pad plane in the inner part of a super sector.

```
double
  innerSectorFrischGridPadPlaneSeparation() const
```

Returns the separation distance between the zero potential (frisch) wire grid and the pad plane in the inner part of a super sector.

```
double
  innerSectorGatingGridPadPlaneSeparation() const
```
Returns the separation distance between the gating grid wire grid and the pad plane in the inner part of a super sector.

```
double
  outerSectorAnodeWirePadPlaneSeparation() const
```
Returns the separation distance between the anode wires and the pad plane in the outer part of the super sector.

```
double
  outerSectorFrischGridPadPlaneSeparation() const
```
Returns the separation distance between the zero potential (frisch) wire grid and the pad plane in the outer part of the super sector.

```
double
  outerSectorGatingGridPadPlaneSeparation() const
```
Returns the separation distance between the gating grid wire grid and the pad plane in the outer part of the super sector.

```
int
  numberOfInnerSectorAnodeWires() const
```
Returns the number of anode wires in the inner part of the super sector.

```
double
  firstInnerSectorAnodeWire() const
```
Returns the radial distance to the first anode wire at the centroid of the inner part of a super sector.

```
double
  lastInnerSectorAnodeWire() const
```
Returns the radial distance to the last anode wire at the centroid of the inner part of a super sector.

```
double
  innerSectorAnodeWire(int w) const
```
Returns the radial distance to the anode wire, w.

```
int numberOfOuterSectorAnodeWires() const
```
Returns the number of anode wires in the outer part of a super sector.

```
double
  firstOuterSectorAnodeWire() const
```
Returns the radial distance of the first anode wire in the outer part of a super sector.

```
double
  lastOuterSectorAnodeWire() const
```
Returns the radial distance to the last anode wire in the outer part of a super sector.

```
double
   outerSectorAnodeWire(int w) const
```
Returns the radial distance to the anode wire, w in the outer part of a super sector.

General – Field Cage

`double frischGrid() const`

Returns the z-position of the zero potential (frisch) grid with respect to center of the TPC volume.

`double driftDistance() const`

Returns the maximum drift distance of ionization within the field cage volume.

`double ifcRadius() const`

Returns the radial distance to the inner field cage electrodes.

`double ofcRadius() const`

Returns the radial distance to the outer field cage electrodes.

`double endCapZ() const`

Returns the z-coordinate of the end-cap of the TPC.

```
bool
  acceptance(StThreeVector<StDouble>& c) const
```

Returns a boolean value that indicates whether a position as specified by the three-vector c lies within the confines of the TPC field cage/active-volume.

`void print(ostream& os = cout) const`

Diagnostic function which prints all constants within the data base accessible by member functions to a file stream.

Example

```
#include <iostream.h>
#include <unistd.h>    // needed for access()

#include <string>

#include "StTpcSimpleGeometry.hh"

int main ()
{
    // Check File access

    string geoFile("../run/TPCgeo.conf");
    if (access(geoFile.c_str(),R_OK)) {
        cerr << "ERROR:\n" << geoFile << " cannot be opened" << endl;
        cerr << "Exitting..." << endl;
        exit(1);
    }

    // Instantiate the DataBase

    StTpcGeometry *geomDb =
        StTpcSimpleGeometry::instance(geoFile.c_str());

    // print out the data base parameters

    geomDb->print();

    return 0;
}
```

Programs Output:

```
To be run
```

13.7 StTpcSimpleSlowControl

Summary Implementation of a simple data base which provides slow control or monitored parameters.

Synopsis `#include "StTpcSimpleSlowControl.hh"`
As with the other data base implementations the `SystemOfUnits` from the SCL is also used to ensure consistent set of units is utilized. .

Description Class `StTpcSimpleSlowControl` is a concrete class which implements methods for the access functions defined in the abstract base class `StTpcSlowControl`. The implementation of the ”simple” data base uses the `StGetConfigValue` utility from the SCL which parses an ASCII file and reads the numerical value of the parameters which are specified by a key word. The parameters are kept as data members, and the access functions simply return these values. This initialization is done in the *private* constructors.

Persistence None

Related Classes The base class which defines the interface is specified in the class **StTpcSlowControl**. See section 13.8.

Public Constructors The data base is implemented as a singleton class which protects the code against multiple distinct copies of the data base parameters in the code. As such the class constructors are implemented as *private* data members which are called via a public member function:

`static StTpcSlowControl* instance(const char* file)`
Returns a pointer of type StTpcSlowControl. The *static* designation implies at most, one instance of this can occur. The pointer will be returned if and only if a file name (file) suitable to initialize the class is specified. Such a file is provided in TRS in the **run** directory (run/sc.conf). Subsequent declarations of magnetic fields can be made in the code, but once the first instance is created, the same pointer will be returned.

`static StTpcSlowControl* instance()`
Returns the pointer of type StTpcSlowControl should an instance of the class have been made previously, otherwise a filename must be specified for the first initialization.

Private Constructors The constructors are hidden from direct call to ensure that only one instance of the data base is made. As such, the constructors are only called through member functions as described above. The constructors which can be called are:

`StTpcSimpleSlowControl(const char* file)`
Called from the member function `instance(const char* file)` which is only invoked if a previous instance is not detected. The parsing of the ASCII file is done in this constructor to initialize the data members.

`StTpcSlowControl()`
Never called; rather the member function *`instance()`* returns the StSlowCon-

trol pointer which was created by the previous instance. No initialization of data members is done.

Public Operators None

Public Member Functions Following are implementations of the functions defined in the interface `StTpcSlowControl`.

Environment

`double hallTemperature() const`
Returns the environmental temperature within the experimental hall.

`double hallPressure() const`
Returns the environmental pressure within the experimental hall.

Voltages

`double driftVoltage() const`
Returns the voltage applied to the membrane which produces the drift field voltage.

`double innerSectorAnodeVoltage() const`
Returns the voltage applied to the anode wires in the inner part of a super sector.

`double innerSectorGatingGridVoltage() const`
Returns the voltage applied to the gating grid wires in the inner part of a super sector.

`double outerSectorAnodeVoltage() const`
Returns the voltage applied to the anode wires in the outer part of a super sector.

`double outerSectorGatingGridVoltage() const`
Returns the voltage applied to the gating grid wires in the outer part of a super sector.

Diagnostic

`void print(ostream& os = cout) const`
Diagnostic feature which prints all the parameters accessible via public member functions to an output file stream. Default is the screen.

Example

```
#include <iostream.h>
#include <unistd.h>    // needed for access()

#include <string>

#include "StTpcSimpleSlowControl.hh"

int main ()
{
    // Check File access

    string scFile("../run/sc.conf");
    if (access(scFile.c_str(),R_OK)) {
        cerr << "ERROR:\n" << scFile << " cannot be opened" << endl;
        cerr << "Exitting..." << endl;
        exit(1);
    }

    // Instantiate the DataBase
```

```
    StTpcSlowControl *scDb =
        StTpcSimpleSlowControl::instance(scFile.c_str());

    // print out the data base parameters

    scDb->print();

    return 0;
}
```

Programs Output:

```
To be run
```

13.8 StTpcSlowControl

Summary Interface which defines access functions to conditional parameters that are monitored during a physics run.

Synopsis Purely abstract class, no instantiation is possible.

Description Class `StTpcSlowControl` is an abstract base class that defines the interface that is used to access all TPC slow control related parameters. This includes both detector and environmental parameters.

Persistence None

Related Classes The implementation of the slow control data base is done in the class `StTpcSimpleSlowControl`. See section 13.7.

Public Constructors None

Public Virtual Operators None

Public Virtual Member Functions

Environment

`virtual double hallTemperature() const`
Provides access to the environmental temperature within the experimental hall.

`virtual double hallPressure() const`
Provides access to the environmental pressure within the experimental hall.

Voltages

`virtual double driftVoltage() const`
Provides access to the voltage applied to the membrane which produces the drift field voltage.

```
virtual double
  innerSectorAnodeVoltage() const
```
Provides access to the voltage applied to the anode wires in the inner part of a super sector.

```
virtual double
  innerSectorGatingGridVoltage() const
```
Provides access to the voltage applied to the gating grid wires in the inner part of a super sector.

```
virtual double
  outerSectorAnodeVoltage() const
```
Provides access to the voltage applied to the anode wires in the outer part of a super sector.

```
virtual double
  outerSectorGatingGridVoltage() const
```
Provides access to the voltage applied to the gating grid wires in the outer part of a super sector.

Diagnostic

`virtual void print(ostream& os = cout) const`

Diagnostic feature which prints all the parameters accessible via public member functions to an output file stream os.

14 Coordinates and Transformations

Following are descriptions of the three different coordinate systems as well as the functor which relates them. The transformations between the three systems are all encapsulated in the overloaded "()" operator. The transformation routines requires the use of the data base implementations described in section 13.

14.1 StGlobalCoordinate

Summary

Definition of a global coordinate given by an `StThreeVector` of double precision. This defines the global STAR coordinate system which is deduced from the survey measurements of the detectors.

Synopsis

`#include "StGlobalCoordinate.hh"`
Requires `StThreeVector` as well as `StGlobals` from the SCL. These are included internally.

Description

Class `StGlobalCoordinate` stores a position as a three vector (i.e. (x,y,z)). This is the coordinate system that the detector subsystem as well as the magnetic field will be related through. It will be necessary when following track trajectories between detectors. No units are implied or assumed. These must be specified by the user.

Persistence

None

Related Classes

The header file "StTpcCoordinates.hh" contains the *include* directives for all coordinate systems. The transformation class or functor "StCoordinateTransform" is also related. See section 14.2.

Public Constructors

```
StGlobalCoordinate()
```
Creates an instance with the coordinate vector initialized to (0,0,0).

```
StGlobalCoordinate(const double x, const double y,
                                   const double z)
```
Creates an instance with the coordinate vector initialized to (x,y,z).

```
StGlobalCoordinate(const StThreeVector<double>& v)
```
Creates an instance with the coordinates initialized to the values as specified by the `StThreeVector` v.

Public Operators

None

Public Member Functions

```
const StThreeVector<double>& pos() const
```
Inline access function which returns the coordinate in thr form of an StThreeVector. Single components can only be accessed through member function of the StThreeVector.

Non-Member Operators

```
ostream& operator<<(ostream& os,
                    const StGlobalCoordinate& c)
```

Allows the printing of an StGlobalCoordinate to an output file stream without accessing each individual component.

Example

```
#include "StThreeVector.hh"
#include "StGlobalCoordinate.hh"

int main()
{
    StThreeVector<double> v2(1,2,3);

    StGlobalCoordinate coordinate1(9,8,7);
    StGlobalCoordinate coordinate2(v2);

    cout << "Coordinate 1 " << coordinate1 << endl;
    cout << "Coordinate 2 " << coordinate2 << endl;

    return 0;
}
```

Programs Output:

```
To be run
```

14.2 StTpcCoordinateTransform

Summary Functor which contains the routines necessary to transform any of the three coordinate systems as specified in this section (section 14).

Synopsis `#include "StTpcCoordinateTransform.hh"`
The functor utilizes the classes `StThreeVector`, `StMatrix`, and `StGlobals` from the SCL. . In addition, because the coordinate systems are related via geometrical layout of the detector, the data base classes are also required at instantiation—both `StTpcGeometry` and `StTpcSlowControl` are required. As with all classes contained within the framework of TRS, the `SystemOfUnits` from the SCL is also utilized.

Description Class `StGlobalCoordinate` stores a position as a three vector (i.e. (x,y,z)). This is the coordinate system that the detector subsystem as well as the magnetic field will be related through. It will be necessary when following track trajectories between detectors. No units are implied or assumed. These must be specified by the user.

Both the geometrical parameters from the TPC geometry data base and several parameters (i.e. drift velocity) from the slow control database are required for the transformations to be physically meaningful. These data bases must be created outside the class and passed via the constructor.

Persistence None

Related Classes The header file ”StTpcCoordinates.hh” contains the *include* directives for all coordinate systems.

Public Constructors

```
StCoordinateTransform(StTpcGeometry* geoDb,
                                  StTpcSlowControl* scDb)
```

Instantiates the coordinate transformation functor.

Public Operators The only operations publicly accessible are the “()” operators which have been overloaded to facilitate the transformation between the different coordinate systems. This allows the same form and usage syntax independent of the coordinate systems under transformation and relieves the user from calling the actual conversion routines in the correct order. The operator takes two arguments, the coordinate system one has available as the first, and the desired coordinate system as the second. Six such overloads are currently available, two of which are trivial calls of the other four:

RAW TPC Coordinate ↔ TPC Local Coordinates

```
void operator()
  (const StTpcPadCoordinate& raw,
                        StTpcLocalCoordinate& loc)
```

Transforms an `StTpcPadCoordinate` (see 14.4) to an `StTpcLocalCoordinate` (see 14.3). Transformation involves several intermediate steps which are contained in private member functions. The y axis is orthogonal to the pad row direction. The row number then directly maps from the y coordinate and the pad number directly maps from the x coordinate. The z coordinate is obtained from the drift velocity. The coordinate is then rotated through the appropriate angle to coincide with the sector in question.

```
void operator()
  (const StTpcLocalCoordinate& loc,
                              StTpcPadCoordinate& raw)
```

Transforms an `StTpcLocalCoordinate` (see 14.3) to an `StTpcPadCoordinate` (see 14.4). The inverse of the previous transformation, the coordinate if rotated such that the new y axis is orthogonal to the pad row direction. The y coordinate directly maps to a pad row and the x axis to a specific pad. The time bin is related to the z position via the drift velocity. As above these calculations are buried in private member functions.

TPC Local Coordinate ↔ STAR Global Coordinates

```
void operator()
  (const StTpcLocalCoordinate& loc,
                             StGlobalCoordinate& glo)
```

Transforms an `StTpcLocalCoordinate` (see 14.3) to an `StGlobalCoordinate` (see 14.1). The coordinates are shifted by the appropriate amounts specified by the detector survey and alignment calibration procedure. Currently, in the absence of these numbers, the local and global coordinate systems coincide.

```
void operator()
  (const StGlobalCoordinate& glo,
                            StTpcLocalCoordinate& loc)
```

Transforms an `StGlobalCoordinate` (see 14.1) to an `StTpcLocalCoordinate` (see 14.3). The coordinates are shifted by the appropriate amounts specified by the detector survey and alignment calibration procedure. As mentioned above, currently, in the absence of these numbers, the local and global coordinate systems coincide.

RAW TPC Coordinate ↔ STAR Global Coordinates

```
void operator()
  (const StTpcPadCoordinate& raw,
                                StGlobalCoordinate& glo)
```

Transforms an `StTpcPadCoordinate` (see 14.4) to an `StGlobalCoordinate` (see 14.1). This is facilitated through two subsequent calls of the above operators: raw → local; local → global.

```
void operator()
  (StGlobalCoordinate& glo,
             const StTpcPadCoordinate& raw)
```

Transforms an `StGlobalCoordinate` (see 14.1) to an `StTpcPadCoordinate` (see 14.4). This is facilitated through two subsequent calls of the above operators: global → local; local → raw.

Public Member Functions None

Non-Member Operators None

Example

```
#include <iostream.h>
#include <unistd.h>
#include <string>

#include "StTpcSimpleGeometry.hh"
#include "StTpcSimpleSlowControl.hh"

#include "StCoordinates.hh"
#include "StTpcCoordinateTransform.hh"

int main()
{
    // Make instance of Data Bases
    // Check File access

    string geoFile("../run/TPCgeo.conf");
    if (access(geoFile.c_str(),R_OK)) {
        cerr << "ERROR:\n" << geoFile << " cannot be opened" << endl;
        cerr << "Exitting..." << endl;
        exit(1);
    }

    string scFile("../run/sc.conf");
    if (access(scFile.c_str(),R_OK)) {
        cerr << "ERROR:\n" << scFile << " cannot be opened" << endl;
        cerr << "Exitting..." << endl;
        exit(1);
    }

    // Instantiate the DataBases

    StTpcGeometry *geomDb =
        StTpcSimpleGeometry::instance(geoFile.c_str());

    StTpcSlowControl *scDb =
        StTpcSimpleSlowControl::instance(scFile.c_str());


    // Pad coordinate (sector, pad row, pad, timebin)
    StTpcPadCoordinate   raw(12.,4.,4.,312.);
    StTpcLocalCoordinate local;
    StGlobalCoordinate   global;

    StTpcCoordinateTransform transformer(geomDb, scDb);

    transformer(raw,local);
    cout << "raw: " << raw << " --> local " << local << endl;

    transformer(local, global);
    cout << "local: " << local << " --> global " << global << endl;

    transformer(global, raw);
    cout << "global: " << global << " --> raw " << raw << endl;

    return 0;
```

```
}
```

Programs Output:

```
To be run
```

14.3 StTpcLocalCoordinate

Summary Definition of a local coordinate given by an `StThreeVector` of double precision. Defined with respect to the origin at the center of the TPC.

Synopsis `#include "StGlobalCoordinate.hh"`
Requires `StThreeVector` as well as `StGlobals` from the SCL. These are included internally. All the coordinates are included in the header file ”StCoordinates.hh”.

Description Class `StTpcLocalCoordinate` stores a position as a three vector (i.e. (x,y,z)). This is the coordinate system that is defined with respect to an origin that is located at the geometrical center of the TPC. No units are implied or assumed. These must be specified by the user.

Persistence None

Related Classes The header file ”StTpcCoordinates.hh” contains the `include` directives for all coordinate systems. The transformation class or functor ”StCoordinateTransform” is also related. See section 14.2.

Public Constructors

`StTpcLocalCoordinate()`
Creates an instance with the coordinates initialized to (0,0,0).

```
StTpcLocalCoordinate(const double x,
                     const double y, const double z)
```
Creates an instance with the coordinates initialized to (x,y,z).

`StGlobalCoordinate(const StThreeVector<double>& v)`
Creates an instance with the coordinates initialized to the values as specified by the `StThreeVector`, v.

Public Operators None

Public Member Functions

`const StThreeVector<double>& pos() const`
Inline access function which returns the coordinate in thr form of an StThreeVector. Single components can only be accessed through member function of the StThreeVector.

Non-Member Operators

```
ostream& operator<<(ostream& os,
                    const StTpcLocalCoordinate& c)
```
Allows the printing of an StTpcLocalCoordinate to an output file stream without accessing each individual component.

Example

```
#include "StThreeVector.hh"
#include "StTpcLocalCoordinate.hh"

int main()
{
    StThreeVector<double> v2(1,2,3);

    StTpcLocalCoordinate coordinate1(9,8,7);
```

```
    StTpcLocalCoordinate coordinate2(v2);

    cout << "Coordinate 1 " << coordinate1 << endl;
    cout << "Coordinate 2 " << coordinate2 << endl;

    return 0;
}
```

Programs Output:

```
To be run
```

14.4 StTpcPadCoordinate

Summary Definition of a raw TPC coordinate indexed by a sector number, pad row number, pad number and time bin. All are given as integers.

Synopsis `#include "StTpcPadCoordinate.hh"`
Requires `StGlobals` from the SCL. It is included internally.

Description Class `StTpcPadCoordinate` stores a raw position indexed by the elements of the detector. The coordinate is uniquely specified by four (integer) numbers (integers)— a sector number (1–24), a pad row (1–45), a pad number (1–192), and a time bin (1–512). For compactness, a more efficient choice can be made for the data types.

Persistence None

Related Classes The header file ”StTpcCoordinates.hh” contains the *include* directives for all coordinate systems. The transformation class or functor ”StCoordinateTransform” is also related. See section 14.2.

Public Constructors

`StTpcPadCoordinate()`
Creates an instance with the coordinates initialized to (0,0,0,0).

```
StTpcPadCoordinate(const int s,
                   const int r, const int p, const int tb)
```
Creates an instance with the coordinates initialized to (s,r,p,tb) where s, r, p, and tb denote sector, pad row, pad, and time bin respectively.

Public Operators None

Public Member Functions

`const int sector() const`
Inline access function which returns the sector number.

`const int row() const`
Inline access function which returns the pad row number.

`const int pad() const`
Inline access function which returns the pad number.

`const int timeBucket() const`
Inline access function which returns the time bucket.

`void setSector(int s)`
Function which assigns the sector to be s.

`void setRow(int r)`
Function which assigns the row to be r.

`void setPad(int p)`
Function which assigns the pad to be p.

`void setTimeBucket(int tb)`
Function which assigns the time bucket to be tb.

Non-Member Operators

```
ostream& operator<<(ostream& os,
               const StTpcPadCoordinate& c)
```

Allows the printing of an `StTpcPadCoordinate` to an output file stream without accessing each individual component.

Example

```
#include "StTpcPadCoordinate.hh"

int main()
{

    StTpcPadCoordinate coordinate1(9,8,7,312);

    int sector = coordinate.sector();
    cout << "sector = " << sector << endl;

    coordinate.setPad(4);
    cout << "pad = " << pad << endl;

    cout << "Coordinate 1 " << coordinate1 << endl;


    return 0;
}
```

Programs Output:

```
To be run
```

15 Physics Processes

Following are descriptions of the four groups of processes that are modeled in TRS at present. These include the classes responsible for the ionization calculations (`StTrsDeDx` – see section 15.3), charge transport of the ionization through the field cage to the read out plane (`StTrsChargeTransporter` – see sections 15.2, 15.6, and 15.9), analog signal generation (see sections 15.1, 15.5, and 15.8), and the digital signal generation (see sections 15.4, 15.7, and **??**). The process of charge collection is best done in a container and is described in section 16.

15.1 StTrsAnalogSignalGenerator

Summary Abstract class which defines an interface for the functions necessary to induce charge onto the TPC pad place given a quantity of charge on an anode wire above it. Furthermore, after the charge is induced on the pad plane, the analog sampling of the charge must be calculated and distributed into time bins.

Synopsis Abstract class, no instantiation is possible.

Dependencies Requires the geometry (section 13.4), slow control (section 13.8), and electronics (section 13.3) data base classes as well as the wire plane (section 16.6) , and the sector where the charge will distributed (section 16.5).

Description Class `StTrsAnalogSignalGenerator` is an abstract class which defines the necessary functions to facilitate the modeling of the physics processes that occur after charge collection occurs on the anode wires. This entails the charge induction on the pad plane which is then read out by an analog pre-amplifier and stored in discrete time bins by a switched capacitor array (SCA) . The base class keeps track of the data base information as well as the range of pads and rows that have a signal induced per single charge cluster on the anode wires. It is possible to set a threshold which reduces the number of signals stored as well as specifying to suppress time bins with no signal present.

Simple initialization and process flags are set via public member functions to enable or disable specific processes. It should be assumed that all processes are by default turned **off** unless set otherwise by the user. No statistical processes occur within this realm so no random number generators are needed nor accessible. This may change with the addition of random noise generation which is currently not implemented.

It should be noted that the derived analog signal generators are implemented as singleton classes so that there is no confusion to which signal generator is being used.

Persistence None

Related Classes Concrete classes are `StTrsFastAnalogSignalGenerator` (see section 15.5) and `StTrsSlowAnalogSignalGenerator` (see section 15.8).

Public Constructors

```
StTrsAnalogSignalGenerator(
 StTpcGeometry* geo, StTpcSlowControl* sc,
```

```
StTpcElectronics* elec, StTrsSector& sec)
```

Called from derived classes which sets flags for calculations and stores data base class pointers—the geometry db (geo), slow control db (sc), the electronics db (elec), and magnetic field db (mag). The class cannot be instantiated alone as it contains virtual member functions which must be implemented in a derived class.

Public Operators

None

Public Member Functions

Charge Induction

```
virtual void
  inducedChargeOnPad(StTrsWireHistogram& wires)
```

Provides an interface for determining the amount of charge that is induced on a single pad given a quantity of charge collected on the anode wire plane. Given an `StTrsWireBinEntry` which specifies an amount of charge collected on a wire at a specific position, the charge can be generated (using a specific functional form) on a single pad.

Charge Sampling

```
virtual void sampleAnalogSignal()
```

Provides interface for sampling an analog signal given the centroid and maximum amplitude of the signal on a pad.

```
virtual double signalSampler(double t,
                             StTrsAnalogSignal& sig)
```

Provides interface for distributing charge across multiple time bins given the time bin (t) and the centroid and amplitude of the signal (sig). Provides the amount of integrated charge within the time bin specified by t.

Switches

```
void setDeltaPad(int dp)
```

Sets the total number of pads (dp) in a given row which will have the amount of charge induced on them from a single charge deposition at the wire plane. Default is 0; that is, the charge is only induced on the pad directly below the charge.

```
void setDeltaRow(int dr)
```

Sets the total number of rows (dr) in a given row which will have the amount of charge induced on them from a single charge deposition at the wire plane. No cross coupling between the inner and outer part of the super sector is currently allowed. Default is 0; that is, the charge is only induced on the pad directly below the charge.

```
void setSignalThreshold(int thr)
```

Sets the threshold (in fC) of the signal size which must be exceeded in order for the results to be stored.

```
void setSuppressEmptyTimeBins(bool v)
```

Sets flag to write out only time bins that are above the signal threshold.

Non-Member Operators

None

Example

see section 15.8 and 15.5.

15.2 StTrsChargeTransporter

Summary Abstract class which provides an interface for the functions necessary to transport an `StTrsMiniChargeSegment` (see section 16.4).

Synopsis Abstract class, no instantiation is possible.

Dependencies Requires the geometry (section 13.4), slow control (section 13.8), and magnetic field (section 13.1) data base classes as well as the ionization information of the gases (section 15.3). Internally requires the random number generators which are contained in the SCL.

Description Class `StTrsChargeTransporter` is an abstract class which defines the necessary functions to facilitate charge transport through a gas volume with the possible presence of electro-magnetic fields. Simple initialization and process flags are set via public member functions to enable or disable specific processes. It should be assumed that all processes are by default turned **off** unless set otherwise by the user. Statistical processes are generally handled by the use of random number generators which are contained in the Star Class Library . These are contained in the base class as *static* data members in order to ensure time is not wasted in the instantiation of random seeds.

It should be noted that the derived charge transporters are implemented as singleton classes so that there is no confusion to which transporter is being used.

Persistence None

Related Classes Concrete classes are `StTrsFastChargeTransporter` (see section 15.6) and `StTrsSlowChargeTransporter` (see section 15.9).

Public Constructors

```
StTrsChargeTransporter(
  StTpcGeometry* geo, StTpcSlowControl* sc,
  StTrsDeDx* dedx, StMagneticField* mag)
```

Called from derived classes which sets flags for calculations and stores data base class pointers—the geometry db (geo), slow control db (sc), the ionization/gas information (dedx), and magnetic field db (mag). The class cannot be instantiated alone as it contains virtual member functions which must be implemented in a derived class.

Public Operators None

Public Member Functions

```
virtual void transportToWire(StTrsMiniChargeSegment& seg)
```

Provides interface for transporting an `StTrsMiniChargeSegment` seg to the z position of the wire plane. The z-position of the mini segment must be changed to reflect the drift distance and the amount of charge may be altered to reflect any charge loss.

```
virtual double chargeAttachment(double l)
```

Provides interface for charge attachment value to be calculated. Given a drift length l, the fraction of charge can be calculated. Users may used any parameter from the data bases that are stored within the class. The *protected* data member **mAttachment** (double precision) is assigned a value which determines the fraction of charge loss. A boolean flag which is set by the member function **setChargeAttachment(bool v)** determines whether the charge attachment calculation is applied.

`virtual double wireGridTransmission()`
Provides interface for calculation of the fraction of charge lost due to the transport of ionization through a wire grid (i.e. the gating grid). The member function must set the protected data member **mTransparency** (double precision) which specifies the fraction of charge lost. The charge loss is applied if a boolean flag is set via the public member function *setGatingGridTransparency*.

Switches
`void setChargeAttachment(bool v)`
Sets a flag which determines whether the charge will be attenuated via attachment processes during transport through the field cage. Default is FALSE.

`void setGatingGridTransparency(bool v)`
Sets a flag which determines whether the charge will be attenuated due to a finite wire grid transparency. Default is FALSE.

`void setTransverseDiffusion(bool v)`
Sets a flag which determines whether the charge will be distributed at the pad plane according to the diffusion properties of the gas in the transverse direction only. Default is FALSE.

`void setLongitudinalDiffusion(bool v)`
Sets a flag which determines whether the charge will be distributed at the pad plane according to the diffusion properties of the gas in the longitudinal direction only. Default is FALSE.

`void setExB(bool v)`
Sets a flag which determines whether the charge will be transported incorporating the effects of the magnetic field. Default is FALSE.

`double transparencyCalculation()`
An implementation for a mono-stable switched gating grid using the gating grid geometry and voltages. A constant value (independent of position in the chamber) is returned in the private data member **mTransparency**. The calculation in hidden in **private** member functions. It is done only once, regardless of the number of times it is called, when the public member function `setGatingGridTransparency(bool v)` is set to TRUE.

Non-Member Operators None

Example see section 15.6 and 15.9

15.3 StTrsDeDx

Summary Contains parameters regarding various gas mixtures of interest for STAR in addition to the functions required to calculate parameters and quantities related to ionization production of charged particles through such mixtures.

Synopsis `#include "StTrsDeDx.hh"`

Description Class `StTrsDeDx` has two components. It contains constants and parameters related to (currently) three types of gas mixtures which allow calculations to be done regarding both transport properties of ionization through the medium, but also the production of ionization itself. Currently constants for three gas mixtures are included:

- $NeCO_2$ (90:10).
- P10; $ArCH_4$ (90:10).
- Ar.

Parameters needed for transport properties include diffusion coefficients in both the longitudinal and transverse directions. The magnetic field dependence on these parameters is also calculable. Attachment coefficients due to electro-negative impurities (i.e. O_2 and H_2O) are also stored. (in a soon to be included appendix).

Ionization parameters that are stored include the mean average ionization yield per unit length, ionization potential, average number of pairs produced per unit length etc. This allows the modeling of energy loss through ionization processes from first principles (i.e. the mean free path) for any gas mixture. Given a specific length, the total number of interactions, as well as secondary electrons produced can be calculated completely within the class. The Bethe-Bloch parameterization which specifies the amount of ionization relative to a minimum ionizing particle is calculable within the class. Since ionization is a statistical procedure, there are also three *static* random number generators which allow the generation of:

- Poissonian Distribution.
- Gaussian Distribution.
- Flat Distribution.

Persistence None

Related Classes None

Dependencies This class requires the random number generators from the SCL are required along with the `SystemOfUnits`. These are included internally.

Public Constructors `StTrsDeDx()`
Default constructor is not accessible.

`StTrsDeDx(const string& gas, double l=1.95*centimeter,)`
Creates an instance with the gas constants selected by the string gas. Either ”Ar”, ”Ne”, or ”P10” (”p10”) can be specified. A sample (i.e. pad) length is specified by the second parameter l. The type of gas must be specified while the sample length has a default value of 19.5 mm. **NOTE: Units are taken to be mm by default!**

Enumerated Types

```
StElectron(primaries, secondaries, total,
                                    numberOfElectrons)
```

Public Operators

None

Public Member Functions

`double W() const`
Inline access functions which returns the average energy deposition to produce a single electron-ion pair.

`double padLength() const`
Inline access function which returns the current sample length.

`void setPadLength(double len)`
Allows one to set the sample length to any arbitrary length. Use `SystemOfUnits` to remove ambiguity in sample length.

`double transverseDiffusionCoefficient() const`
Inline access function returns the transverse diffusion constant. No magnetic field is currently assumed.

`double longitudinalDiffusionCoefficient() const`
Inline access function returns the longitudinal diffusion constant.

`double attachmentCoefficient() const`
Inline access function which returns the oxygen attachment coefficient.

`double nextInteraction()`
Returns the relative position of the next interaction from a statistical distribution of the mean free path of a charged particle in the gas specified in the constructor.

`int primary(double bg = 3)`
Returns the number of primary electrons produced in a sample length as specified in the constructor due to ionization interactions by a particle with a relativistic value of $\beta\gamma$. By default it is taken as 3 which is a minimum ionizing particle. The number of ionizations is calculated from a Poissonian distribution with a mean which is calculated from a parameter which specifies the mean number of ionizations per unit length in the specific gas.

`int secondary(double* E)`
Returns the number of secondary electrons generated in a sample length as specified in the constructor from ionization processes subsequent to the initial primaries. The energy of the primary is calculated (from a statistical distribution), and given the ionization potential of the gas, the number of subsequent electrons can be calculated. A slight medium dependence to the expectation of the Rutherford (E^{-2}) is introduced. The energy of the primary electron is returned as a pointer value (E).

`double betheBloch(double bg) const`
The fraction of ionization relative to minimum ionizing is returned given the relativistic velocity ($\beta\gamma$) of the particle. The parameterization used is taken from Walenta et al.[22]

`void electrons(vector<int>&, double bg = 3)`
Returns the total number of electrons produced in a sample length specified in the constructor in a vector with three components. The first entry stores the number of primaries, the second stores the number of secondaries, and the third stores the total number generated.

`void print(ostream& os = cout)`
Diagnostic feature which allows one to print the contents of the parameters stored for the gas of interest within the class to a file stream. The default file stream is the screen.

Non-Member Operators None

Example

```
#include <fstream.h>
#include <unistd.h>
#include <vector>
#include <string>

#include "StGlobals.hh"
#include "SystemOfUnits.h"
#include "StThreeVector.hh"

#ifndef ST_NO_NAMESPACES
using namespace units;
#endif

#include "StTrsDeDx.hh"

int main()
{
    int    numberOfTracks  = 1000;
    int    numberOfSamples = 45;    // number of pads
    float  subSegments     = 1.;    // break sample into

    double padLength = 1.15*centimeter;
    cout << "subSegments= " << subSegments << endl;

    string gas("Ar");
    StTrsDeDx myELoss(gas,padLength);
    StTrsDeDx subELoss(gas,(padLength/subSegments));

    myELoss.print();

    int ii,jj,kk;
#ifndef ST_NO_TEMPLATE_DEF_ARGS
    vector<int> sum;
#else
    vector<int, allocator<int> > sum;
#endif
```

[22] A. H. Walenta et al., NIM **161** (1979) 45.

```
    double bg = .1;

    for(jj=1; jj<=5; jj++) {
        double increment = bg;
        for(kk=1; kk<10; kk++) {
            bg+=increment;

            cout << "bg " << bg
                 << \"Relative Energy Loss \" << (myELoss.betheBloch(bg)) << endl;
        }
        bg+=increment;
    }


    //Create tracks:
    for(int itrack=0; itrack<numberOfTracks; itrack++) {
        for (int isample=0; isample<numberOfSamples; isample++) {

            sum.resize((StTrsDeDx::numberOfElectrons),0);
            myELoss.electrons(sum);

            cout << "Track " <<  static_cast<float>(itrack)           << endl;
            cout << " primaries   " <<  (sum[StTrsDeDx::primaries])   << endl;
            cout << " secondaries " <<  (sum[StTrsDeDx::secondaries]) << endl;
            cout << " total       " <<  (sum[StTrsDeDx::total])       << endl;

            int totalInSubsegment = 0;
            for(int isubsample=0; isubsample<subSegments; isubsample++) {
                sum.resize((StTrsDeDx::numberOfElectrons),0);
                subELoss.electrons(sum);
                totalInSubsegment += sum[StTrsDeDx::total];

                cout << " sub: " <<  (totalInSubsegment);
            }
        } // isample
    }     //  itrack

    return 0;
}
```

15.4 StTrsDigitalSignalGenerator

Summary Abstract class which provides an interface for the functions necessary to simulate the digitization process in the TPC front-end electronics.

Synopsis Abstract class, no instantiation is possible.

Dependencies Requires the electronics (section 13.3) data base class `StTpcElectronics` as well as the sector information where the analog information is stored `StTrsSector` (section 16.5) and the output data structure `StTrsDigitalSector` (section 16.3).

Description Class `StTrsDigitalSignalGenerator` is an abstract class which defines the necessary functions to facilitate the modeling of the processes that occur in the digitization of the analog signals which occupy the time buckets in a sector. This is currently implemented as a simple proportionality constant. In the near future, pending an appropriate function, the non-linear characteristics of the STAR ADC will be implemented. **Digital noise is also not currently implemented.** The base class keeps track of the data base information. No flags currently exist to toggle the active processes, however they should be added upon implementation of various noise generators. No statistical processes occur within this realm so no random number generators are needed nor accessible. This may change with the addition of random noise generation. It should be noted that the derived digital signal generators are implemented as singleton classes so that there is no confusion to which signal generator is being used.

Persistence None

Related Classes Concrete classes are `StTrsFastDigitalSignalGenerator` (see section 15.7) and `StTrsSlowDigitalSignalGenerator` (see section **??**).

Public Constructors

```
StTrsDigitalSignalGenerator(StTpcElectronics* elec,
             StTrsSector& sec,StTrsDigitalSector& digSec)
```

Called from derived classes which stores the electronics data base class pointers. The class cannot be instantiated alone as it contains virtual member functions which must be implemented in a derived class.

Public Operators None

Public Member Functions

```
virtual void digitizeSignal() const
```

Provides interface for digitizing the signals stored in the `StTrsSector` which is stored as a protected data member, by reference.

```
virtual void addWhiteNoise() const
```

Provides interface for producing random white noise into the data at the digital level.

```
virtual void addCorrelatedNoise() const
```

Provides interface for producing correlated noise into the data at the digital level.

Non-Member Operators None

Example see section 15.7 and **??**.

15.5 StTrsFastAnalogSignalGenerator

(*Not currently implemented*)

Summary Implements the virtual functions in the class `StTrsAnalogSignalGenerator` (section 15.1). **NOTE:** This class is not currently implemented!!!

Synopsis `#include "StTrsFastAnalogSignalGenerator.hh"`
Requires `StTpcGeometry` (section 13.4 `StTpcSlowControl` (section 13.8), `StTrsDeDx` (section 15.3) `StMagneticField` (section 13.1 .

Description Class `StTrsFastAnalogSignalGenerator` is meant to implement the three virtual functions specified in the class `StTrsAnalogSignalGenerator` (section 15.1 with fast (parameterized) algorithms which reproduce the statistical mean behavior of the processes in question. The emphasis should be on speed and efficiency as it is foreseen that many full events will be modeled with these algorithms. Currently it is not implemented as the **Slow** version of the implementation is being developed. The implementation will be made as a singleton so that multiple transporters cannot exist within the same program.

Persistence None

Related Classes The header file ”StTrsAnalogSignalGenerator.hh” contains the base class and ”StTrsSlowAnalogSignalGenerator.hh” contains a detailed microscopic version of the implementation.

Dependencies As mentioned above, this class requires `StTpcGeometry` (section 13.4 `StTpcSlowControl` (section 13.8), `StTpcElectronics` (section 13.3 . The output is written to an `StTrsSector` .

Public Constructors

```
StTrsAnalogSignalGenerator* instance(
    StTpcGeometry* geo, StTpcSlowControl* sc,
    StTpcElectronics* elec, StTrsSector& sec)
```

Checks whether or not a previous instance of the class exists and either returns a pointer to that instance or calls the private constructor and returns a pointer to it.

```
StTrsAnalogSignalGenerator* instance()
```

Checks whether or not a previous instance of the class exists and either returns a pointer to that instance or returns an error message.

Protected Constructors

```
StTrsFastAnalogSignalGenerator()
```

No default constructor is accessible.

```
StTrsFastAnalogSignalGenerator(
   StTpcGeometry* geo, StTpcSlowControl* sc,
   StTpcElectronics* elec, StTrsSector& sec)
```

Will create an instance of the fast analog signal generator. Initialization and bookkeeping variables are done in the base class (section 15.1).

Public Operators None

Public Member Functions

`void inducedSignalOnPad(StTrsWireHistogram& plane)`

`void sampleAnalogSignal()`

`void signalSampler()`

Non-Member Operators None

Example None yet.

15.6 StTrsFastChargeTransporter

Summary Implements the virtual functions in the class `StTrsChargeTransporter` (section 15.2).

Synopsis `#include "StTrsFastChargeTransporter.hh"`
Requires `StTpcGeometry` (section 13.4 `StTpcSlowControl` (section 13.8), `StTrsDeDx` (section 15.3) `StMagneticField` (section 13.1 .

Description Class `StTrsFastChargeTransporter` is meant to implement the three virtual functions specified in the class `StTrsChargeTransporter` (section 15.2 with fast (parameterized) algorithms which reproduced that statistical mean or af the process parameters. Currently the member function `transportToWire()` simple projects the mini charge segment onto the pad plane. A parameterization of the field cage distortions and effects of the magnetic field are required to go beyond this. The charge attachment is done as a statistical mean rather than at the single charge level, and no positional dependence is allowed. All flags which determine the processes which will be modeled are contained in the base class. The implementation is a singleton so that multiple transporters cannot exist within the same program.

Persistence None

Related Classes The header file "StTrsChargeTransporter.hh" contains the base class and "StTrsSlowChargeTransporter.hh" (not yet implemented) contain are more microscopic approach to the implementation.

Dependencies As mentioned above, this class requires `StTpcGeometry` (section 13.4 `StTpcSlowControl` (section 13.8), `StTrsDeDx` (section 15.3) `StMagneticField` (section 13.1 . An `StTrsMiniChargeSegment` is the object that is transported. `StGlobals` and `StThreeVector` from the SCL are also required. These are included internally.

Public Constructors

```
StTrsChargeTransporter* instance(
  StTpcGeometry* geo, StTpcSlowControl* sc,
  StTrsDeDx* dedx, StMagneticField* mag)
```

Checks whether or not a previous instance of the class exists and either returns a pointer that instance or calls the private constructor and returns a pointer to it.

Protected Constructors

```
StTrsChargeTransporter()
```

No default constructor is accessible.

```
StTrsChargeTransporter(
     StTpcGeometry* geo, StTpcSlowControl* sc,
     StTrsDeDx* dedx, StMagneticField* mag)
```

Creates an instance of the fast charge transporter. Initialization and book-keeping variables are done in the base class (section 15.2).

Public Operators None

Public Member Functions

`void transportToWire(StTrsMiniChargeSegment& seg)`
Member function projects the position of a mini charge segment onto the pad plane. The z-position is modified to reflect the drift distance of the ionization segment. The quantity of charge is also adjusted depending on whether the appropriate boolean flags have been set. The charge attenuation must be calculated in the two other virtual functions.

`void chargeAttachment(double l)`
Given a drift length, the member function sets a protected data member **mAttachment** (double precision) to the fractional value of the charge loss expected.

`void wireGridTransmission()`
Applies the fractional value stored in the protected data member **mTransparency** to the charge in the mini segment to allow for the charge loss.

Non-Member Operators

None

Example

Do not run this example! It is meant for illustrative purposes only. For a real example consult section 18.

```
// DataBases
#include "StTpcSimpleGeometry.hh"
#include "StTpcSimpleSlowControl.hh"
#include "StTpcSimpleElectronics.hh"
#include "StSimpleMagneticField.hh"
#include "StTrsDeDx.hh"

// processes
#include "StTrsFastChargeTransporter.hh"

int main()
{
    //
    // after the Data bases have been created

    StTrsChargeTransporter *trsTransporter =
        StTrsFastChargeTransporter::instance(geomDb, scDb, &myEloss, magDb);

    // Set Flags:
     trsTransporter->setChargeAttachment(true);
     trsTransporter->setGatingGridTransparency(true);
     trsTransporter->setTransverseDiffusion(true);
     trsTransporter->setLongitudinalDiffusion(true);
     trsTransporter->setExB(false);

    //
    // After mini Segments are generated
    //

     trsTransporter->transportToWire(aMiniSegment);

    // collect the charge, generate the signals...
}
```

15.7 StTrsFastDigitalSignalGenerator

Summary Implements the virtual functions in the class `StTrsDigitalSignalGenerator` (section 15.4).

Synopsis `#include "StTrsFastDigitalSignalGenerator.hh"`
Requires `StTpcElectronics` (section 13.3.

Description Class `StTrsFastDigitalSignalGenerator` implements the three virtual functions specified in the class `StTrsDigitalSignalGenerator` (section 15.4 with fast (parameterized) algorithms which reproduced that statistical behavior of the digitization. Currently the member function `digitizeSignal()` calculates the ADC value given the amount of charge in a given time bucket assuming a constant conversion factor independent of channel number. Several member functions which generate noise components require development. The implementation is a singleton so that multiple transporters cannot exist within the same program.

Persistence None

Related Classes The header file ”StTrsDigitalSignalGenerator.hh” contains the base class and ”StTrsSlowDigitalSignalGenerator.hh” (not yet implemented) contain are more microscopic approach to the implementation.

Dependencies As mentioned above, this class requires `StTpcElectronics` (section 13.3 charge collected on the sense wires. The output is written as analog signal (section **??**) into a container class which is an `StTrsSector` indexed by an `StTpcPadCoordinate`. As with all classes, use of `SystemOfUnits` is made where appropriate.

Public Constructors

```
StTrsDigitalSignalGenerator*
    instance(StTpcElectronics* elec, StTrsSector& sec)
```

Checks whether or not a previous instance of the class exists and either returns a pointer to that instance or calls the private constructor and returns a pointer to it. Base class contains the data base instances as well as the output container (by reference).

```
StTrsDigitalSignalGenerator* instance()
```

Checks whether or not a previous instance of the class exists and returns a pointer to that instance or returns an error message.

Protected Constructors

```
StTrsFastDigitalSignalGenerator()
```

No default constructor is accessible.

```
StTrsFastDigitalSignalGeneratorStTpcElectronics* elec,
                                            StTrsSector& sec)
```

Creates an instance of the fast digital signal generator. Initialization and bookkeeping variables are kept in the base class (section 15.4).

Public Operators None

Public Member Functions

`void digitizeSignal()`
Loops over all time bins contained in the sector and applies a conversion constant which converts the charge (quantified in mV) to an ADC value.

`void addWhiteNoise()`
Produces random white noise into the data at the digital level. (Not currently implemented).

`virtual void addCorrelatedNoise()`
Produces correlated noise into the data at the digital level. (Not currently implemented).

Non-Member Operators

None

Example

Do not run this example! It is meant for illustrative purposes only. For a real example consult section **??**.

```
// DataBases
#include "StTpcSimpleGeometry.hh"
#include "StTpcSimpleFastControl.hh"
#include "StTpcSimpleElectronics.hh"
#include "StSimpleMagneticField.hh"
#include "StTrsDeDx.hh"

// processes
#include "StTrsFastDigitalSignalGenerator.hh"

int main()
{
    // after the Data bases have been created

    StTrsDigitalSignalGenerator *trsDigitalSignalGenerator =
        StTrsFastDigitalSignalGenerator::instance(electronicsDb, sector);

    // create a Sector:
    StTrsSector *sector = new StTrsSector(geomDb);

    // after charge transport and charge collection
    // Generate the ANALOG Signals on pads

    trsDigitalSignalGenerator->digitizeSignal();
}
```

15.8 StTrsSlowAnalogSignalGenerator

Summary Implements the virtual functions in the class `StTrsAnalogSignalGenerator` (section 15.1).

Synopsis `#include "StTrsSlowAnalogSignalGenerator.hh"`
Requires `StTpcGeometry` (section 13.4 `StTpcSlowControl` (section 13.8), `StTrsDeDx` (section 15.3) `StMagneticField` (section 13.1 .

Description Class `StTrsSlowAnalogSignalGenerator` implements the three virtual functions specified in the class `StTrsAnalogSignalGenerator` (section 15.1 with fast (parameterized) algorithms which reproduced that statistical mean or af the process parameters. Currently the member function `inducedChargeOnPad()` calculates the amount of charge induced on a specified pad through a (selectable) analytic expression as described in section 9 The charge is then sampled in time after amplification. Charge is distributed into time bins according to a specific electronics response. Integer flags which determine the extent of the pad and row cross coupling are set in the base class. The implementation is a singleton so that multiple transporters cannot exist within the same program.

Persistence None

Related Classes The header file "StTrsAnalogSignalGenerator.hh" contains the base class and "StTrsSlowAnalogSignalGenerator.hh" (not yet implemented) contain are more microscopic approach to the implementation.

Dependencies As mentioned above, this class requires `StTpcGeometry` (section 13.4 `StTpcSlowControl` (section 13.8), `StTrsDeDx` (section 15.3) `StMagneticField` (section 13.1 . An `StTrsWireHistogram` is the object that is required for the input as this contains all the charge collected on the sense wires. The output is written as analog signals (section **??**) in a container class which is an `StTrsSector` indexed by an `StTpcPadCoordinate`. As with all classes, use of `SystemOfUnits` is made where appropriate.

Public Constructors

```
StTrsAnalogSignalGenerator* instance(
  StTpcGeometry* geo, StTpcSlowControl* sc,
  StTpcElectronics* elec, StTrsSector& sec)
```

Checks whether or not a previous instance of the class exists and either returns a pointer to that instance or calls the private constructor and returns a pointer to it. Base class contains the data base instances as well as the output container (by reference).

`StTrsAnalogSignalGenerator* instance()`
Checks whether or not a previous instance of the class exists and returns a pointer to that instance or returns an error message.

Protected Constructors `StTrsSlowAnalogSignalGenerator()`
No default constructor is accessible.

```
StTrsSlowAnalogSignalGenerator(
  StTpcGeometry* geo, StTpcSlowControl* sc,
  StTpcElectronics* elec, StTrsSector& sec)
```
Creates an instance of the slow analog signal generator. Initialization and book-keeping variables are kept in the base class (section 15.1).

Public Operators

None

Enumerated Data Types

`StDistribution`
Used to specify the functional form of the charge distribution. Values include: **endo**, **gatti**, **dipole**, **unknown**.

`StSignal`
Used to specify the functional form of the electronics response. Values include: **delta**, **symmetricGaussianApproximation**, **symmetricGaussianExact**, **symmetricGaussianExact**, **realShaper undefined**.

Public Member Functions

`void inducedChargeOnPad(StTrsWireHistogram& hist)`
Creates an analog signal which denotes the amplitude and centroid (in time) of a pulse generated by an **StTrsWireBinEntry**. Filled into the private data member *mSector*. The functional forms of the charge distributions are specified by *private* member functions that are called internally.

```
double signalOnPad(double xo, double yo, double xl,
                   double xu, double yl, double yu)
```
Calculates the charge with coordinates specified in a Cartesian system from xl → xu in the x-direction and yl → yu in the y-direction. Note that the value returned must be scaled by the appropriate amplitude of the charge cluster. The actual functional forms are specified by *private* member functions which are called internally.

`void sampleAnalogSignal()`
Given the centroid and amplitude of all signals on a pad, the signal from the analog electronics is deduced and distributed into time bins. Currently there is provisions to add purely random noise, however it is foreseen to implement algorithms which produced series and parallel noise for more accurate simulations. The results are written into the container *StTrsSector*.

`double signalSampler(double tb, StTrsAnalogSignal& sig)`
Selects the functional form of the analog electronics response. These functions must be implemented as *private* data members and return an amplitude (double precision) given the centroid and total charge.

`void setChargeDistribution(StDistribution dist)`
User Selection of the functional form of the charge distribution. Must be specified by an `StDistribution` which is an enumerated data type.

`void setElectronicSampler(StSignal dist)`
User Selection of the functional form of the signal sampler. Must be specified by an `StSignal` which is an enumerated data type.

Non-Member Operators

None

Example Do not run this example! It is meant for illustrative purposes only. For a real example consult section 18.

```
// DataBases
#include "StTpcSimpleGeometry.hh"
#include "StTpcSimpleSlowControl.hh"
#include "StTpcSimpleElectronics.hh"
#include "StSimpleMagneticField.hh"
#include "StTrsDeDx.hh"

// processes
#include "StTrsSlowAnalogSignalGenerator.hh"

int main()
{
    // after the Data bases have been created

    StTrsAnalogSignalGenerator *trsAnalogSignalGenerator =
        StTrsSlowAnalogSignalGenerator::instance(geomDb,
                                          scDb, electronicsDb, sector);
    // set the flags

    trsAnalogSignalGenerator->setDeltaRow(0);
    trsAnalogSignalGenerator->setDeltaPad(0);
    trsAnalogSignalGenerator->setSignalThreshold(.0001);
    trsAnalogSignalGenerator->setSuppressEmptyTimeBins(true);

    // create a Sector:
    StTrsSector *sector = new StTrsSector(geomDb);

    // after charge transport and charge collection
    // Generate the ANALOG Signals on pads

    trsAnalogSignalGenerator->inducedChargeOnPad(theWirePlane);
}
```

15.9 StTrsSlowChargeTransporter

Summary Implements the virtual functions in the class `StTrsChargeTransporter` (section 15.2). **NOTE:** This class is not currently implemented!!!

Synopsis `#include ``StTrsSlowChargeTransporter.hh''`
Requires **StTpcGeometry** (section 13.4 **StTpcSlowControl** (section 13.8), **StTrsDeDx** (section 15.3) **StMagneticField** (section 13.1 .

Description Class `StTrsSlowChargeTransporter` is meant to implement the three virtual functions specified in the class `StTrsChargeTransporter` (section 15.2 with detailed microscopic algorithms which model the processes as close as possible. Little regard is made for time or efficiency. It is not foreseen that more than a few tracks will be modeled with these algorithms, but it is meant to study and tune parameterizations that can be used in the fast charge transporter. Currently the member function `transportToWire()` simply projects the mini charge segment onto the pad plane. It is foreseen that the drift velocity of the ionization at and position within the field cage will be done via the Langevin equation, and propagated using a Runge-Kutta solution to the equation. The charge attachment should be done at the single electron level given a probability for absorption. The implementation should be made as a singleton so that multiple transporters cannot exist within the same program.

Persistence None

Related Classes The header file ”StTrsChargeTransporter.hh” contains the base class and ”StTrsFastChargeTransporter.hh” contains a parameterized version of the implementation.

Dependencies As mentioned above, this class requires `StTpcGeometry` (section 13.4 `StTpcSlowControl` (section 13.8), `StTrsDeDx` (section 15.3) `StMagneticField` (section 13.1 . An `StTrsMiniChargeSegment` is the object that is transported. `StGlobals` and `StThreeVector` from the SCL are also required. These are included internally.

Public Constructors

```
StTrsChargeTransporter* instance(
  StTpcGeometry* geo, StTpcSlowControl* sc,
  StTrsDeDx* dedx, StMagneticField* mag)
```

Checks whether or not a previous instance of the class exists and either returns a pointer that instance or calls the private constructor and returns a pointer to it.

Protected Constructors

```
StTrsChargeTransporter()
```

No default constructor is accessible.

```
StTrsChargeTransporter(
  StTpcGeometry* geo, StTpcSlowControl* sc,
  StTrsDeDx* dedx, StMagneticField* mag)
```

Creates an instance of the fast charge transporter. Initialization and book-keeping variables are done in the base class (section 15.2).

Public Operators None

Public Member Functions

```
void transportToWire(StTrsMiniChargeSegment& seg)

void chargeAttachment(double l)

void wireGridTransmission()
```

Non-Member Operators None

Example None Possible.

16 Physics Containers

Following are descriptions of the different containers used for the storage of the data generated at the various stages of the simulation

16.1 StTrsAnalogSignal

Summary Provides storage for an arbitrary signal characterized by an amplitude and a time.

Synopsis `#include "StTrsAnalogSignal.hh"`

Description Class `StTrsAnalogSignal` provides a way to wrap both analog and digital signal for insertion into the sector structure which is the complete raw data of the TPC. It characterizes a signal by its centroid (in time) and either total integrated charge or charge at the peak. It is the responsibility of the user to make this distinction. Access functions for both components are provides as well as set functions which allow manipulation of each component. No use of the `SystemOfUnits` class is implied or expected. In addition a comparison operator () is provided which returns the smallest of two signals determined by the time value. It is foreseen in the future to make this a templated class such that optimization can be made based on whether analog or digital signals are being used.

Persistence None

Related Classes `StTrsSector` is implemented such that it stores data in the form of `StTrsAnalogSignal`

Public Constructors

`StTrsAnalogSignal()`
Default constructor: time and amplitude components are initialized to zero.

`StTrsAnalogSignal(float t, float amp)`
Time and amplitude components are initialized to the values of t and amp respectively.

Public Operators None

Public Member Functions

`float time() const`
Returns the value of the time component.

`float amplitude() const`
Returns the value of the amplitude component.

`float setTime(float t)`
Sets the value of the time component to the value specified by t.

`float setAmplitude(float amp)`
Sets the value of the amplitude component to the value specified by amp.

`void scaleAmplitude(float f)`
Scales the value of the amplitude component from the current value by a factor f.

Non-Member Operators

`ostream& operator<<(ostream& os, const StTrsAnalogSignal& sig)`
Allows the printing of an StTrsAnalogSignal to an output file stream without accessing each individual component. Default is the screen.

`bool operator()(StTrsAnalogSignal& a, StTrsAnalogSignal& b)`
Returns a boolean value based on whether the signal a has a time component less than the value of b (true).

Example

Programs Output:

```
To be run
```

16.2 StTrsChargeSegment

Summary An object-oriented translation of the *g2t_tpc_hit* structure which functions as an input container for the simulator.

Synopsis `#include "StTrsChargeSegment.hh"`
Requires `StTpcGeometry`, `StTpcSlowControl`, `StMagneticField` data base classes. Also requires `StTrsDeDx` and `StTrsMiniSegment`. Requires *StThreeVector*, *StMatrix* from the SCL.

Description The input to the simulator is generally taken from an external program which will dominantly be GEANT. The information required for input is the amount of ionization (dE) deposited over a path length (ds) at a position $\vec{x}$ of a track t. This is the information necessary to construct a charge segment. The class is also able to rotate the charge segment to the sector 12 reference system which allows a direct mapping from the pad-row number to the y coordinate and the pad number to the x coordinate. Calculations are much simpler in this system. The class is also able to split the segment into smaller fragments given the parameters of the gas from the ionization class (see `StTrsDeDx` in section **??**). This allows finer granularity of the simulation process.

Persistence None

Related Classes A list of `StTrsMiniChargeSegment` is constructed by calling the member function `split()`.

Public Constructors

`StTrsChargeSegment()`
Constructs a charge segment with all components initialized to zero.

```
StTrsChargeSegment(StThreeVector<double>& pos,
  StThreeVector<double>& mom, g2t_tpc_hit* g2t)
```
Constructs a charge segment with a momentum, mom at a position, pos. The number of electrons and path length are also kept in data members and must be specified...

Public Operators None

Public Member Functions

`StThreeVector<double>& position() const`
Returns the position of the centroid of the segment.

`StThreeVector<double>& momentum() const`
Returns the momentum of the track at the centroid of the segment.

`double dE() const`
Returns the energy deposited in the segment.

`double ds() const`
Returns the length of the track segment.

`void rotate(StTpcGeometry* geoDb, StTpcSlowControl* scDb)`
Alters the position of the charge segment relative to the position in the local coordinate system of sector 12.

```
void split(StTrsDeDx* dedx, StMagneticField* magDb,
   int n, double len, list<StTrsMiniChargeSegment>* segs)
```

Splits a charge segment into n subsegments of length len and returns each into a list, segs.

Non-Member Operators None

Example

Programs Output:

```
To be run
```

16.3 StTrsDigitalSector

Summary Storage for the pixel data in digital, zero compressed format.

Synopsis `#include "StTrsDigitalSector.hh"`
Requires `StTpcGeometry`, and `StTpcPadCoordinate`.

Description The final output of the simulator is pixel data which is indexed by pad-row, pad number, and perhaps time-bin. The `StTrsDigitalSector` is a container which can store an indeterminate number of `signed char` indexed by these quantities which represent 8-bit ADC values. Access is provided by the pad number of a given row and by a complete pad row. The class also provides methods to add entries with a single function call.

Persistence None

Related Classes `StTrsAnalogSignal` is the only type of structure that the sector can store.

Public Constructors

`StTrsDigitalSector()`
Default constructor cannot be called!

`StTrsDigitalSector(StTpcGeometry* geoDb)`
Constructs a sector with the number of pad-rows and pads contained in a STAR sector. The numbers are read from the geometry database specified by geoDb.

Public Operators None

Public Member Functions

`vector<char>& timeBinsOfRowAndPad(int r, int p)`
Returns all the ADC values contained on pad p of row r in an STL vector.

`typedef vector<vector<char>> tpcPadRow`
`tpcPadRow& padsOfRow(int r)` Returns all the ADC values on the pads in row r. The signals are accessible by an additional pad number.

`typedef vector<tpcDigitalPadRow>> tpcSector`
`tpcDigitalSector& rows()`
Returns the complete digital sector.

`int numberOfRows() const`
Returns the number of pad-rows contained by the digital sector.

`int numberOfPadsInRow(int r) const`
Returns the number of pads in row r contained by the digital sector.

`void clear()`
Clears all ADC values from the sector.

`void assignTimeBins(int r, int p, vector<char>& sig)`
Assigns a set of time bins sig to the pad p of row r. **Indices start from 1!**.

```
void assignTimeBins(StTpcPadCoordinate& c,
                        vector<char>& sig)
```
Assigns a set of time bins sig to the pad and row r specified by the raw TPC coordinate c.

Non-Member Operators None

Example

Programs Output:

```
To be run
```

16.4 StTrsMiniChargeSegment

Summary A sub-segment of a charge segment which determines the granularity of the simulation.

Synopsis `#include "StTrsMiniChargeSegment.hh"`
Requires `StThreeVector`, `StMatrix` from the SCL.

Description The mini segment is a fragment of a charge segment (above) after it has been split. It can contain the complete or a component of the segment. It is distributed onto a helical trajectory over the path length (dl) as specified in the charge segment. It is this mini segment on which the charge transporter will operate. As such it stores the position of the mini segment and the number of electrons contained therein. Access functions for each component exist as to does a method for adjusting the components after transport has been completed.

Persistence None

Related Classes An `StTrsChargeSegment` is fragmented to produce mini segments.

Public Constructors

`StTrsMiniChargeSegment()`
Default constructor initializes the position and amount of ionization and path length over which ionization is distributed to zero.

`StTrsMiniChargeSegment(StThreeVector<double>& x, double de, double dl`
Initializes the position of the mini segment to x and amount of ionization de deposited over a path length dl to the specified values.

Public Operators None

Public Member Functions

`const StThreeVector<double>& position() const`
Returns the position of the mini segment.

`StThreeVector<double>& position()`
Allows assignment of the position of the mini segment.

`double dl() const`
Returns the value of the path length.

`const double charge() const`
Returns the value of the amount of charge deposited in the mini segment.

`void setCharge(double c)`
Allows the value of the charge c to be specified.

Non-Member Operators

`ostream& operator<<(ostream& os, const StTrsMiniChargeSegment& seg)`
Allows the printing of an StTrsMiniChargeSegment to an output file stream without accessing each individual component. Default is the screen.

Example

Programs Output:

```
To be run
```

16.5 StTrsSector

Summary

Storage for the pixel data in both analog and digital format.

Synopsis

#include "StTrsSector.hh"
Requires StTpcGeometry, StTrsAnalogSignal, and StTpcPadCoordinate.

Description

The final output of the simulator is pixel data which is indexed by pad-row, pad number, and perhaps time-bin. The StTrsSector is a container which can store an indeterminate number StTrsAnalogSignals indexed by these quantities. Access is provided by the pad number of a given row and by a complete pad row. The class also provides methods to add entries with a single function call.

Persistence

None

Related Classes

StTrsAnalogSignal is the only type of structure that the sector can store.

Public Constructors

StTrsSector()
Default constructor cannot be called!

StTrsSector(StTpcGeometry* geoDb)
Constructs a sector with the number of pad-rows and pads contained in a STAR sector. The numbers are read from the geometry database specified by geoDb.

Public Operators

None

Public Member Functions

vector<StTrsAnalogSignal>& timeBinsOfRowAndPad(int r, int p)
Returns all the analog signals contained on pad p of row r in an STL vector.

typedef vector<vector<StTrsAnalogSignal>> tpcPadRow
tpcPadRow& padsOfRow(int r) Returns all the signals on the pads in row r. The signals are accessible by an additional pad number.

typedef vector<tpcPadRow>> tpcSector
tpcSector& rows()
Returns the complete sector.

int size() const
Returns the number of pad-rows contained by the sector.

int numberOfRows() const
Returns the number of pad-rows contained by the sector. (See also size()).

int numberOfPadsInRow(int r) const
Returns the number of pads in row r contained by the sector.

void clear()
Clears all analog signals from the sector.

void addEntry(int r, int p, StTrsAnalogSignal& sig)
Adds a signal sig, to row r, and pad p. **Indices start from 1!**

void addEntry(StTpcPadCoordinate& c, StTrsAnalogSignal& sig)
Adds a signal sig, to the row and pad as specified by the raw coordinate c.

```
void assignTimeBins(int r, int p, vector<StTrsAnalgoSignal>& sig)
```
Assigns a set of time bins sig to the pad p of row r. **Indices start from 1!**.

```
void assignTimeBins(StTpcPadCoordinate& c,
                vector<StTrsAnalgoSignal>& sig)
```
Assigns a set of time bins sig to the pad and row r specified by the raw TPC coordinate c.

Non-Member Operators None

Example

Programs Output:

```
To be run
```

16.6 StTrsWireHistogram

Summary Contains all the charge collected on the anode wires and provides a mechanism to amplify this charge via gas gain amplification.

Synopsis `#include "StTrsWireHistogram.hh"`
Requires the data base classes `StTpcGeometry` and `StTpcSlowControl`. Only `StTrsWireBinEntrys` can be stored in the histogram. Also requires the `StThreeVector` and `Random` classes of the SCL.

Description The *wire histogram* is a class in which charge can be assigned to single wires in a controlled manner; that is, charge is assigned through the *wire bin entry* class. The functionality of this class goes much beyond simply keeping track of the amount of ionization collected on a single wire, but also possesses functions necessary to facilitate gas gain amplification of each cluster. The reason this *"process"* is associated with a container is that the structure and layout of the wire grid, which is necessary in the *wire histogram* is also necessary to be able to do gas gain. Thus instead of imposing overhead of redundant class construction, it was incorporated here. This is one of the advantages of Object-Oriented design. Each charge cloud can now be used to induce a signal on the pad plane.

Persistence None

Related Classes Only `StTrsWireBinEntrys` can be stored in the histogram.

Public Constructors The wire histogram is implemented as a singleton class which protects the code against multiple distinct copies of the wire histogram in the code. As such the class constructors are implemented as *private* data members which are called via a public member function:

`StTrsWireHistogram()`
Cannot be called.

`StTrsWireHistogram(StTpcGeometry* geoDb, StTpcSlowControl* scDb)`
Called from the member function *instance()*.

Private Constructors

```
static StTrsWireHistogram*
   instance(StTpcGeometry* geoDb, StTpcSlowControl* scDb)
```

Returns a pointer of type StTrsWireHistogram. The *static* designation implies at most, one instance of this can occur. Subsequent declarations of StTrsWireHistograms can be made in the code, but the same pointer will be returned.

Public Operators None

Public Member Functions

`int min() const`
Returns the first wire with at least one entry.

`int max() const`
Returns the last wire with at least one entry.

Charge Collection
`void addEntry(StTrsWireBinEntry& bin)`

Adds an *StTrsWireBinEntry* bin to the histogram according to the position of the entry.

`void clear()`
Removes all entries from the histogram.

`void setDoTimeDelay(bool v)`
Sets a flag to explicitly calculate the time offset for those electrons that do not project directly on an anode wire. Calculation is done in a private member function.

Wire Operations
`double wireCoordinate(int w)`
Returns the y coordinate of the wire number w as specified in the sector 12 coordinate frame.

`vector<StTrsWireBinEntry>& getWire(int w)`
Returns in an STL container, all the entries on a wire specified by the wire number w.

`vector<vector<StTrsWireBinEntry>>& getWireHistogram()`
Returns all the entries on all wires.

Gas Gain
`void setDoGasGain(bool v)`
Sets a flag according the the value v whether gas gain amplification should be done. Actual gas gain is done in private member function.

`void setDoGasGainFluctuations(bool v)`
Sets a flag according the the value v whether gas gain amplification should be done with fluctuations according to the Raether distribution. Actual gas gain is done in private member function.

`double avalanche(int n)`
Performs the gas gain amplification according to the status of the flags set on wire n. Actual gas gain is done in private member function.

Non-Member Operators

`ostream& operator<<(ostream& os, const StGlobalCoordinate& c)`
Allows the printing of an StGlobalCoordinate to an output file stream without accessing each individual component.

Example

```
#include "StThreeVector.hh"
#include "StGlobalCoordinate.hh"

int main()
{

    return 0;
}
```

Programs Output:

```
To be run
```

16.7 StTrsWireBinEntry

Summary Container such that the charge arriving at the anode wire plane can be conveniently entered into the `StTrsWireHistogram`.

Synopsis `#include "StTrsWireBinEntry.hh"`
Requires `StGlobalCoordinates`. Also requires `StThreeVector` from the SCL.

Description Once the charge reaches the anode wire grid it must be collected on the wires. As such the charge mini segment, which has a ionization distribution which is in general, orthogonal to the wire direction, distributes the charge on the pad plane in a Gaussian-like distribution to model the effects of diffusion. This charge must then be collected on the wires and so, must be repackaged such that a specific quantity of ionization can be assigned to a position on the anode wires. Thus the *wire bin entry* is made to collect an amount of charge, q in the vicinity of an anode wire at a position $\vec{x}$, which can be assigned to a wire.

Persistence None

Related Classes Class `StTrsWireHistogram` only stores objects of type `StTrsWireBinEntry`.

Public Constructors

`StTrsWireBinEntry()`
Creates an object with the position and number of electrons initialized to zero.

`StTrsWireBinEntry(StThreeVector<double> pos, float ne)`
Creates an object with the position pos and number of electrons ne.

Public Operators None

Public Member Functions

`StThreeVector<double>& position() const`
Returns the position of the ionization cluster.

`StThreeVector<double>& position()`
Allows the position of the ionization cluster to be set.

`float numberOfElectrons() const`
Returns the number of electrons in a single cluster.

`void setNumberOfElectrons(float n) const`
Allows the number of electrons within the cluster to be set to a definite value n.

`void scaleNumberOfElectrons(float f) const`
Allows the number of electrons within the cluster to be scaled by a factor f.

Non-Member Operators

`ostream& operator<<(ostream& os, const StGlobalCoordinate& c)`
Allows the printing of an StGlobalCoordinate to an output file stream without accessing each individual component.

Example

```
#include "StThreeVector.hh"
#include "StGlobalCoordinate.hh"

int main()
```

```
{

    return 0;
}
```

Programs Output:

```
To be run
```

17 Management and Auxiliary Classes

Following are descriptions of the management classes which are used to coordinate and oversee the simulation procedure.

17.1 StTrsMaker

Summary

Synopsis `#include "StTrsMaker.hh"`
Requires

Description

Persistence None

Related Classes

Public Constructors `StTrsMaker()`

Public Operators None

Public Member Functions `const StThreeVector<double>& pos() const`
Inline access function which returns the coordinate in thr form of an StThreeVector. Single components can only be accessed through member function of the StThreeVector.

Non-Member Operators None

Example see examples

18 Physical Examples

Example

```
#include <iostream.h>
#include <unistd.h>    // needed for access()
#include <fstream.h>

#include <string>
#include <vector>
#include <utility>    // pair
#include <algorithm>  // min() max()

// SCL
#include "Randomize.h"
#ifdef DIAGNOSTICS
#include "StHbook.hh"
#endif
// General TRS
#include "StCoordinates.hh"
#include "StTpcCoordinateTransform.hh"

// TRS
// db
#include "StTpcSimpleGeometry.hh"
#include "StTpcSimpleSlowControl.hh"
#include "StTpcSimpleElectronics.hh"
#include "StSimpleMagneticField.hh"
#include "StTrsDeDx.hh"

// processes
#include "StTrsFastChargeTransporter.hh"
#include "StTrsSlowAnalogSignalGenerator.hh"
#include "StTrsFastDigitalSignalGenerator.hh"

// containers
#include "StTrsAnalogSignal.hh"
#include "StTrsWireBinEntry.hh"
#include "StTrsWireHistogram.hh"

#include "StTrsSector.hh"


#define VERBOSE 1
#define ivb if(VERBOSE)

void printPad(StTrsSector *a)
{
    ostream_iterator<StTrsAnalogSignal> out(cout,",");

    for(int irow=0; irow<a->size(); irow++)
        cout << irow << " " <<  "<" << a[irow].size() << "> ";

//      for(int ipad=0; ipad<a[irow].size(); ipad++) {
//          cout << irow << " " << ipad << "<" << a[irow][ipad].size() << "> ";
//          //copy(a[irow][ipad].begin(),a[irow][ipad].end(),out);
//          cout << endl;
//      }
```

```
}

// Sort the "analogSignal"s on a pad according to the time
// void sortTime(sector& a)
// {
//     for(int irow=0; irow<a.size(); irow++)
//      for(int ipad=0; ipad<PADS[irow]; ipad++) {
//          sort(a[irow][ipad].begin(),a[irow][ipad].end(),comp_last());
//      }
// }


/* -------------------------------------------------------------------- */
/*                        Main Program                                  */
/* -------------------------------------------------------------------- */
int main (int argc,char* argv[])
{

    //
    // Make the DataBase
    //
    // Check File access
    //
    string geoFile("../run/TPCgeo.conf");
    if (access(geoFile.c_str(),R_OK)) {
        cerr << "ERROR:\n" << geoFile << " cannot be opened" << endl;
        //shell(pwd);
        cerr << "Exitting..." << endl;
        exit(1);
    }

    string scFile("../run/sc.conf");         // contains B field
    if (access(scFile.c_str(),R_OK)) {
        cerr << "ERROR:\n" << scFile << " cannot be opened" << endl;
        cerr << "Exitting..." << endl;
        exit(1);
    }

    string electronicsFile("../run/electronics.conf");
    if (access(electronicsFile.c_str(),R_OK)) {
        cerr << "ERROR:\n" << electronicsFile << " cannot be opened" << endl;
        cerr << "Exitting..." << endl;
        exit(1);
    }

    //
    // The DataBases
    //
    StTpcGeometry *geomDb =
        StTpcSimpleGeometry::instance(geoFile.c_str());

    StTpcSlowControl *scDb =
        StTpcSimpleSlowControl::instance(scFile.c_str());
    scDb->print();

    StMagneticField *magDb =
        StSimpleMagneticField::instance(scFile.c_str());
```

```
    StTpcElectronics *electronicsDb =
        StTpcSimpleElectronics::instance(electronicsFile.c_str());


    string gas("Ar");
    StTrsDeDx myEloss(gas);


    //
    // create a Sector:
    //
    StTrsSector *sector = new StTrsSector(geomDb);

    //
    // Processes
    //
    StTrsChargeTransporter *trsTransporter =
        StTrsFastChargeTransporter::instance(geomDb, scDb, &myEloss, magDb);
    // set status:
//     trsTransporter->setChargeAttachment(true);
//     trsTransporter->setGatingGridTransparency(true);
//     trsTransporter->setTransverseDiffusion(true);
//     trsTransporter->setLongitudinalDiffusion(true);
//     trsTransporter->setExB(true);

    StTrsWireHistogram *theWirePlane =
        StTrsWireHistogram::instance(geomDb, scDb);
//     theWirePlane->setDoGasGain(true);  // True by default
//     theWirePlane->setDoGasGainFluctuations(false);
//     theWirePlane->setDoTimeDelay(false);

    StTrsAnalogSignalGenerator *trsAnalogSignalGenerator =
        StTrsSlowAnalogSignalGenerator::instance(geomDb, scDb, electronicsDb, sector);
//     trsAnalogSignalGenerator->setDeltaRow(0);
//     trsAnalogSignalGenerator->setDeltaPad(0);
//     trsAnalogSignalGenerator->setSignalThreshold(.0001);
//     trsAnalogSignalGenerator->setSuppressEmptyTimeBins(true);
    //     ??should the type of function be an option ???

    StTrsDigitalSignalGenerator *trsDigitalSignalGenerator =
        StTrsFastDigitalSignalGenerator::instance(electronicsDb, sector);

    //
    // Generate the Ionization
    //

    float maxDistance = geomDb->lastOuterSectorAnodeWire();
    PR(maxDistance);
    float zPosition = 1.*meter;
    float position = 52.*centimeter;
    float dS;
    do {
        dS = myEloss.nextInteraction();
//      PR(dS);
        position += dS;
```

```
    if(position>maxDistance) break;

    double primaryEnergyDistribution;
    int totalElectrons = myEloss.secondary(&primaryEnergyDistribution) + 1;
    cout << endl;
    PR(totalElectrons);

    // Make a StTrsMiniChargeSegment (the thing that must be transported)
    StTrsMiniChargeSegment
        aMiniSegment(StThreeVector<double>(0, position, zPosition),
                     totalElectrons,  // q
                     0);              // dl
    PR(aMiniSegment);

    //
    // TRANSPORT HERE
    //
    trsTransporter->transportToWire(aMiniSegment);
    PR(aMiniSegment);

    //
    // CHARGE COLLECTION AND AMPLIFICATION
    //
    StTrsWireBinEntry anEntry(aMiniSegment.position(), aMiniSegment.charge());
    PR(anEntry);
    theWirePlane->addEntry(anEntry);

}while(true);

cout << "\a***************************\a\n" << endl;


//
// Generate the ANALOG Signals on pads
//

trsAnalogSignalGenerator->inducedChargeOnPad(theWirePlane);


//
// Sample the ANALOG Signals that were induced on pads
//
trsAnalogSignalGenerator->sampleAnalogSignal();


//
// Digitize the Signals
//
trsDigitalSignalGenerator->digitizeSignal();


//
// Write it out!
//

cout << "Write out the Sector " << endl;
string outPutFile("output");
```

```
    ofstream to(outPutFile.c_str());
    if(!to) {
        cerr << "Cannot open output file " << outPutFile << endl;
        exit(1);
    }

    for(irow=1; irow<=sector->numberOfRows(); irow++)
        for(ipad=1; ipad<=sector->numberOfPadsInRow(irow); ipad++) {
            tpcTimeBins currentPad = sector->timeBinsOfRowAndPad(irow,ipad);

            for(tIter  = currentPad.begin();
                tIter != currentPad.end();
                tIter++) {

                to << irow << '\t' << ipad << '\t' << (tIter->time()) << " " << (tIter
            }
        }

    return 0;
}
```

Programs Output:

```
To be run
```

Index

U

V

W