MICHIGAN ATLAS MONITORED DRIFT CHAMBER PRODUCTION DATABASE

FEBRUARY 4, 2000

HOMER A. NEAL, SHAWN MCKEE AND CHUNHUI HAN
DEPARTMENT OF PHYSICS
UNIVERSITY OF MICHIGAN
ANN ARBOR, MICHIGAN 48109

THE UNIVERSITY OF MICHIGAN ATLAS MONITORED DRIFT CHAMBER PRODUCTION DATABASE

TABLE OF CONTENTS

THE UNIVERSITY OF MICHIGAN ATLAS MONITORED DRIFT CHAMBER PRODUCTION DATABASE

1. ABSTRACT

We describe herein the design philosophy and the implementation details of the University of Michigan ATLAS MDT Production Database. Details are given that would be useful to other institutes wishing to establish a similar facility. We present also a comparison between relational and object oriented databases with regard to their utility for managing production information and analysis.

2. INTRODUCTION

Over the next six years the ATLAS Collaboration will need to construct, test, and commission a massive high precision muon spectrometer consisting of more than one million readout channels. The proper performance of this instrument is absolutely critical to the success of the experiment, particularly since one of the key physics goals is the detection of the Higgs boson, and the four-muon decay is expected to be one of the principal discovery modes.

The University of Michigan has assumed responsibility for the R/D, prototyping and production of 36,000 drift tubes and assembly of the tubes into 96 chambers in the forward muon spectrometer. These chambers consist of the longest drift tubes used in the experiment and are subject to a unique variety of design and production challenges. The principal issue is whether the inherent sag of the drift chambers and their wires will permit the achievement of the 80 micron spatial resolution required. This problem will be most critical for the long chambers and remaining within the parameter design envelopes will require careful monitoring of the fabrication process at each stage.

The success of the chamber construction will depend critically on the creation and maintenance of a comprehensive production database that is to contain the relevant evolutionary history and characterization of each muon chamber component.

Production is to start in 2000, initial chamber deliveries to CERN will occur in September 2000 and these will continue through December 2004, at which time installation in the ATLAS Detector will begin, with the target launch of the experiment being July, 2005.

There are eight distinct stages in the Ann Arbor tube production facility -- the tube assembly station (Wiring), the Leak Checking Station, the Dark Current and Cosmic Ray Station, the Dark Current Station, the X-ray Station, the EMMI Station (wire location), the Resistance Station and the Frequency (Tension) Station. Details of the activities performed at each station are provided in Attachment I. The structure of our production database reflects the steps followed in the assembly and testing the chamber components.

Following the completion of a group of chambers in Ann Arbor, they will be crated into a seaworthy container and shipped to CERN. Upon arrival at CERN they will be unpacked and subjected to a set of tests to insure that all vital parameters remain within specified tolerances.

We present herein the philosophy that guided us in the development of this database, and reference the repository of code that may be required as upgrades are made to the package in the years ahead.

There are several novel elements in our approach. One is a rather close linkage between our ACCESS database and PAW *(a CERN analysis package)*, designed to facilitate extensive analysis of the chamber data in an environment that is familiar to a large number of the Project's physicists. Another is the use of a tandem database structure where appropriate subsets of the final local data is routinely captured and made available to the ATLAS Global MDT database. Indeed, this arrangement will permit ongoing upgrades to the main local database, while yet providing a stable feed to the Global database. An additional unique feature is the provision we have made to insure that the essential functions of the database are accessible remotely via the web.

Other details will be described herein that we hope will be of value to other institutes as they design their databases.

3. OUTLINE OF PAPER

In Section 4 we discuss the guiding principles we used in establishing our database. The exact manner which we chose to implement the database is described in Section 5, including the database tables, forms and fields used.

In Section 6 we discuss the various external interfaces available for interacting with the database, beginning with a description of the core Microsoft tools and ending with an extensive discussion of how to display database contents on the web. As part of this discussion we also describe the techniques used to extract data from the LabWindows applications running on the local "Stations", by using Perl scripts.

Section 6.1.3 shows how one can use PAW to carry out analyses using data contained in the database. Section 6.1.4 describes how we provide information to the ATLAS-wide global production database. In Section 7 we discuss considerations relating to the use of object-oriented databases in the production environment, and we make a detailed comparison of the steps required to extract certain information using our standard ACCESS database and its Objectivity companion which we created. Finally, we discuss in Section 8 future related topics we may wish to examine.

Appendix 1 contains information on the structure of the database, including the detailed definitions of the tables and fields. An example of a Visual Basic program to generate frequency distributions from within ACCESS is given in Appendix 4 Examples of the text files produced at the various stations are shown in Appendix 2 Perl script files for grabbing input data from these text files are given in Appendix 3. An example of a PAW session using an NTUPLE from the database is shown in Appendix 5. The code used to generate NTUPLEs from tables is in Appendix 6. Appendix 7 shows a set of typical plots generated nightly by the application that updates the database. Appendix 8 demonstrates how to create a DSN (Data Source Name). Appendix 9 provides an overview of how to setup another production site based upon our model.

4. DATABASE DESIGN PHILOSOPHY

There are several basic principles we have tried to adhere to as the database structure was established. Many of these will, at first, appear to be obvious. But, even a significant fraction of these are subject to a reasonable debate.

To give just one example, in this day and age of fast computers and cheap data storage cost, one might think it reasonable to measure and record each and every conceivable bit of information about every component that goes into the muon chambers. After all, it may be argued, we may wish to have access to a particular data element five years from now, even though we see no use for it now. The competing argument is that if we are

not careful we will drown in useless information and will be led into a false state of believing we can accept wide variances in chamber parameters now, since we presumably could correct for most maladies in the future.

Against this background, we have decided to adopt the following set of guidelines:

- We will collect and store every piece of data that is known to bear directly on the principal physics performance of the muon chamber
- We will collect and store chamber production data related to the raw material parameters, production conditions and production procedures, including all information required by the ATLAS-wide muon coordinating groups.
- We will automate each and every measurement possible. Each manual entry of a measured quantity to our database must be justified and approved by the MDT group.
- The database is to be available to any authorized user via the web. That is, a key member of the UM-MDT group should be able to log on to the web anywhere in the world and, after sufficient authorization checks, have the same functionality as if he or she was in the production lab.
- Database backup should occur nightly.
- The initial database application package will be based upon Microsoft ACCESS.
- An ACCESS database, or whatever other application is specified by ATLAS, will be explicitly maintained to provide an interface to the ATLAS MDT global database.
- Whenever possible, the database structure, even though relational in basic character, will be developed in full recognition of the current features of object oriented databases. This is done because of known benefits for OO databases, because we know the ultimate ATLAS event storage paradigm will be that of OO databases, and because of the likelihood that database migration of the future will probably be toward OO databases.
- A local R/D effort will be maintained to look at OO based approaches, both to aid us in keeping our database structure as OO "aware" as possible, and to keep open the possibility of coupling OO project management software to our data recording efforts at a later time. The current lead effort will be directed toward the CRISTAL package being developed by a group in CMS.

5. UM DATABASE IMPLEMENTATION OVERVIEW

There are two basic "views" established for our production database: one for intimate operations on the database host machine and one for the typical user, who will normally deal with the database via a web interface. In most instances the direct host machine work will be carried out by the database administrator or by Project leaders.

The direct database interface welcomes the user as he or she enters the initial form, and does not let the user out of its sight until he or she formally exits. In previous generations of database systems, this escort service

was seldom an important consideration since developers wrote applications and the user ran the applications. Now, given the extreme power of programs like ACCESS to both run applications and to alter them, developers must be cautious that the adventurous user is always clear about his or her identity – user or erstwhile developer.

To better demonstrate this point, consider that a user can navigate to a particular table either through the established form presented by the application, or by using the ACCESS menu bar that simply allows one to open the table. In one case, the formal application would permit one to change data only if it satisfies certain validation criteria. In the manual direct approach, data might be changed without any restrictions.

In Figure 1 the basic design of the database is presented.

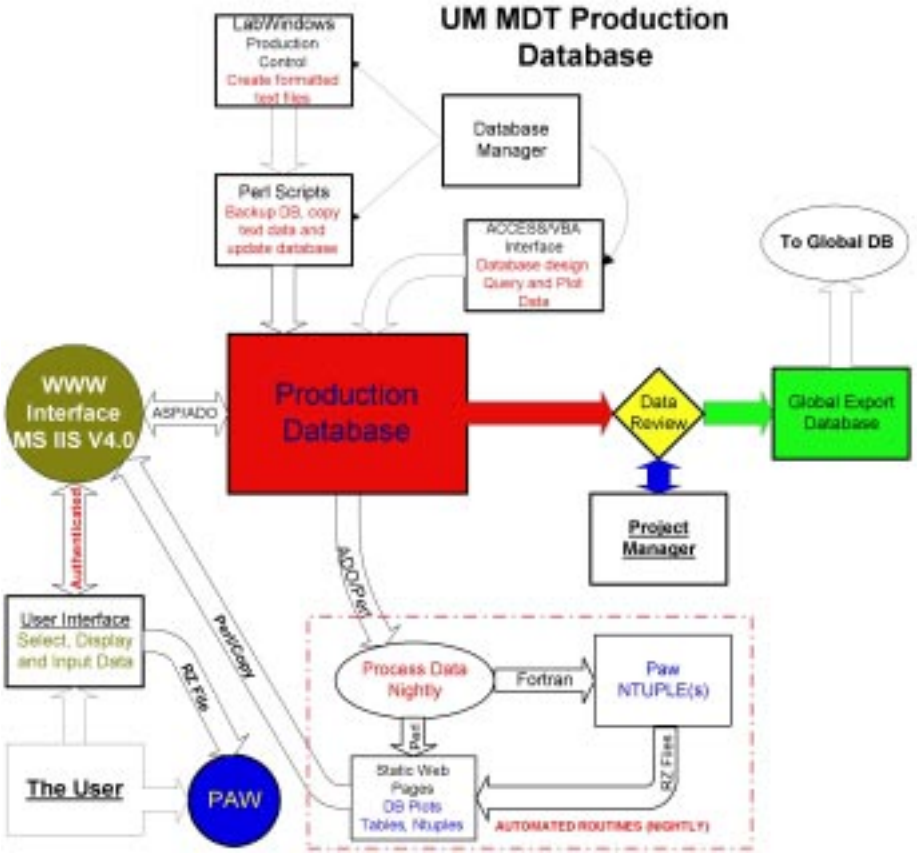


Figure 1 Functional schematic of the current UM production database and interfaces

5.1 DATA INPUT

The list of permanent tables and fields comprising our production database are given in Appendix 1. Data are inserted into the database either manually or through a quasi-automatic process where data stations output results of their measurements to structured text files [See "UM DAQ System for MDT Production" by Qichun and Zhou, submitted as an ATLAS note]. These text files are then loaded into the database through express actions of the database manager (or a monitoring package, which he or she controls). In the latter step one has the option of not only coordinating the daily updates, but of also executing certain consistency checks before the uploading takes place.

5.2 DATA VIEWING AND DATA QUERY

For those accessing the database through the host terminal, the data in any one of the ACCESS tables can be viewed by selecting the table via the Main Menu. Likewise, any table can be queried by following the path provided from the Main Menu. For complex, multi-table queries, a full SQL query can be entered by following the SQL path from the Main Menu.

To realize the full potential of the database, one must be able to easily access its data for purposes of analysis. Since ACCESS is fully integrated with Visual Basic, any simple analysis can be conducted via a Visual Basic script. We have exploited this feature by creating a frequency distribution package that runs entirely within ACCESS. This particular analysis tool was chosen because one of the most direct tests of the reasonableness of some parameter for that tube is to compare the value of that parameter with the distribution of that parameter for all other tubes in the class. Code that performs the frequency distribution analysis is given in Appendix 4. It is included as an example of how packages of this type can be constructed.

We should note, however, that Visual Basic programs will likely have limited utility in analyzing more detailed questions based on the database. For that reason, we have also built in a method through which users can have the database produce NTUPLES for import into PAW. Since the bulk of our community is fluent in PAW, we believe that this bridge is the most efficient way to provide large number of physicists with the database contents. More details on the use of this feature are provided in a subsequent section.

5.3 FILLING OF GLOBAL DATABASE REFLECTOR

To provide an official database which contains data for asynchronous input to the ATLAS Global database, we are creating a database that is separate from, but driven by, our main production database. Our Global Database Reflector differs from the working Production database in the following ways:

1) It is based on the set of tables and fields specified by ATLAS centrally. (Our main production database will have many other fields for local use, and they will be arranged in a table structure more suited to the measurement sequence of our fabrication and testing stations).

2) Only measurements that have been reviewed by, and approved by, our MDT Coordinator staff will be moved into the reflector database.

3) Data moved in to the Reflector will not be recalled or revised by our group, except through whatever protocol is established by ATLAS for such changes.

Details on the reflector are given in a later section.

5.4 WEB INTERFACE

Given the distributed nature of our production and testing facility, as well as the fact that key individuals in our group must be able to query the status of the fabrication at any time and from any place in the world, we have given high priority to providing full database access and control to authenticated users via the web.

The entire interface with the database will, of course, appear differently from the direct host-based interface described here. One reason has to do with security considerations. Another technical reason is that the web user is really not running ACCESS on the host machine, but rather is being provided with various published services. We provide more details about how the features of the web interface in a subsequent section.

5.5 STRUCTURE OF THE DIRECT HOST-BASED DATABASE INTERFACE

In the early stages of the design of the production database the authors made a sharp distinction between the type of access that would be provided the regular user and the access provided the database administrator. While the latter would have full privileges through any perceived interface, including a special host-based interface, the typical user would be channeled to a protected path on a "need-to-know" basis. In the end that philosophy was changed to encompass a single web-based pathway to accessing the database, with the only difference between users being that certain operations would require passwords. However, since some other installations may prefer having a host-based interface, we share below some of our early design considerations.

The direct host-based interface to the database was conceived to be through a set of integrated forms. These forms handle the authentication of the user, and guidance to the proper database functions required to meet the wishes of the user. One can browse the database, seek help on any part of the production process, receive an update on the production status, do data analysis, or any other activity, by choosing from a set of options provided.

The initial "Welcome" form one sees when starting the database application is reproduced below.

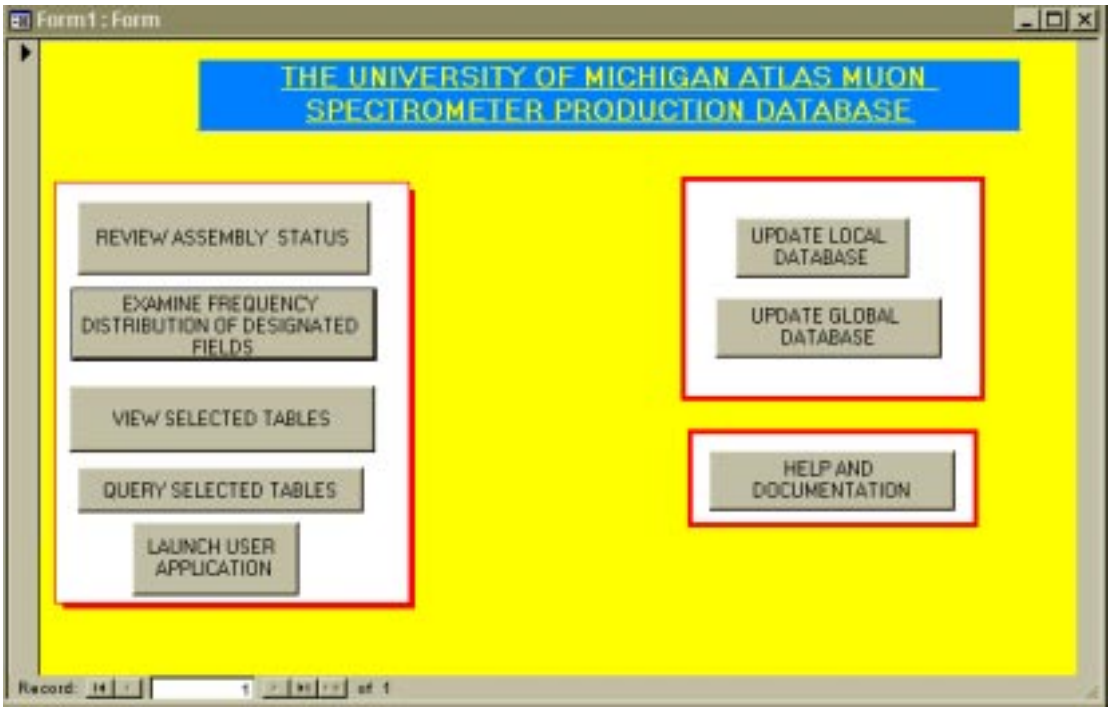


Figure 2 Initial "Welcome" form for host-based interface

The goal of this opening Form is to receive the user, provide him or her with the guideposts needed to get to the task they are trying to accomplish. At the same time, there are protections built in to prevent the unauthorized changing of database data. This is done by having one form call another, or by having a form open a specified table.

The complete form/table structure of the direct host-based database interface is in Figure 3 below. This figure provides details about the navigation from one form to the next, starting with the Main Menu.(The complementary web-based interface will be presented in a later section.)

If one is at the Main Menu (the location of the user after opening the database), there are options to go to a form where the assembly status can be reviewed, to a form where a frequency plot can be made of a specified variable, to a form where a query can be initiated, to a form where a user application can be launched, where the local database can be updated, where the local global database reflector can be updated, or where help can be obtained on any production issue. In some instances the user is taken to an intermediate form, such as represented in the middle column, in order for more information to be collected before executing the primary request. An example is the frequency distribution application, where the user needs to supply the name of the database, table and field, as well as the minimum and maximum value to be used in the frequency plot.

The rightmost column describes the type of output provided as the result of pursuing a particular menu choice. As an example, the request for the status of the assembly produces a report in a fixed format. In the case of a request for a frequency plot, the output can either be a screen display, or a printed copy.

Thus, Figure 3 provides a map of the program steps associated with a given selection made from the main menu.

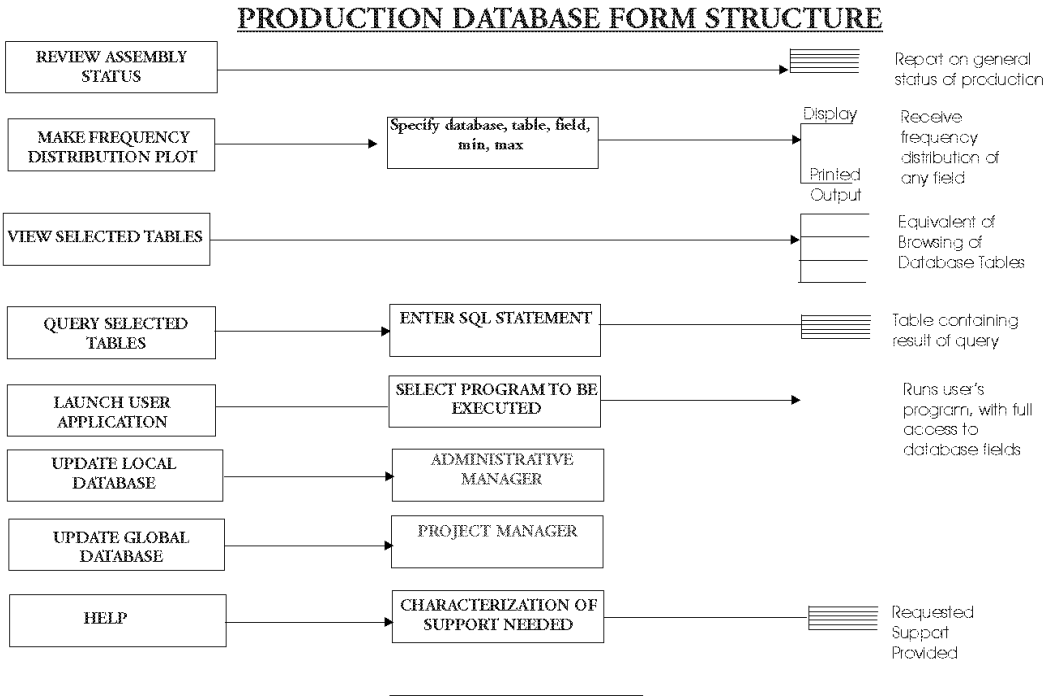


Figure 3 Schematic of host-base interface layout

6. SPECIFIC EXTERNAL INTERFACES

6.1 TOOLS AND TECHNIQUES

After creating the database structure, the next step is to decide how to actually put our data into it. This could be done via ACCESS, either using direct data entry or by importing files, but this has a number of shortcomings. First, the data entry method is not general but is completely tied to ACCESS. This would make changing the underlying database in the future that much more difficult. Second, the import capabilities are limited to specific formats. In our production line we have a combination of computer-generated data and

operator input data whose format may vary. Third, using a direct ACCESS entry would limit our ability to automatically check the incoming data for validity and log the results for future reference.

The primary concern is that we isolate the production data from the methods used to store and access the data. This allows us the freedom to update or change components without having to rebuild our whole software setup. We also want to utilize the extensive amount of software that has been designed to access and present data, thereby allowing us to focus on the details of our requirements.

To address these concerns we felt it best to pursue standard technologies that have been created to solve such problems, like Microsoft's Universal Data Access (UDA, see http://www.microsoft.com/Data/default.htm). From Microsoft's product description:

> *"Universal Data Access is Microsoft's strategy for providing access to information across the enterprise. … Universal Data Access provides high-performance access to a variety of information sources, including relational and non-relational, and an easy to use programming interface that is tool and language independent."*

Since we are using ACCESS on a Microsoft platform we have chosen to utilize Microsoft's MDAC (Microsoft Data Access Components) to enable Universal Data Access. These components include ActiveX Data Objects (ADO), Remote Data Service, (RDS, formerly known as Advanced Database Connector or ADC), OLE DB, and Open Database Connectivity (ODBC).

UDA is only software "glue" which still needs to be implemented with some programming language. Even restricting ourselves to a Microsoft platform we have a number of obvious choices: Visual Basic, Visual C/C++, FORTRAN, Java and Perl.

All the visual tools have the advantage that they are UDA "aware" and provide a GUI (Graphical User Interface) for coding applications. The disadvantage is that these same languages are not very portable to non-Microsoft platforms. FORTRAN is widely known and used in physics, but is being phased out in favor of newer languages. Perl, and to a lesser extent Java, are available for almost any operating system and this is a very strong point in favor of using either. Also both can be object-oriented languages and both are available at little or no cost.

We have chosen to implement UDA with Perl (Practical Extraction and Reporting Language) for a number of reasons. First, it is freely available and supported on a wide range of platforms. Perl is maybe the only programming language found on Amiga's and Cray's and almost everything in between. Second, it is very easy to learn and use and is very powerful in its capabilities. In most programming languages, one has to declare types, variables and subroutines to be used before writing the first line of code. For complex problems demanding complex data structures this is a good idea, but for many simple everyday problems one may want a programming language where one can simply say what is desired to be done. Perl is such a language. Of course if one wants to be specific and declare everything, that can be done too. Our last reason for choosing Perl is that it scales beyond the task at hand. We will be able to use Perl (or PerlScript) when we come to providing the data on the WWW (see "WWW Access" below).

6.1.1 PROCEDURES EMPLOYED FOR DATA INPUT

Our production data is generated daily at a number of stations as text files (see Appendix 2 for examples). This allows each station to run de-coupled from the database and the database server, providing an

environment where problems with the remote database server do not prevent a station from doing its production work. By using text files to record the data we also allow the database architecture to be changed or upgraded independent of the production station software.

The task is to process this data nightly and enter all valid data into the database. At the same time, we want to note and classify unusual events that occur during data processing for each station. We will describe the full nightly sequence below after discussing some of the implementation details.

We have attempted to isolate the required functionality into a number of Perl scripts. For data extraction we have a specific Perl script for each production station which is responsible for parsing that station's text file. Likewise, we have specific Perl routines that enter each station's data into the corresponding ACCESS tables. Each of these routines can refer to a set of common utility routines we have developed for UDA database work: opening the ODBC data source (see Appendix 8 for details on ODBC), checking for errors, querying the database, etc.

Part of Perl's power becomes obvious when we need to perform specific tasks with different input data repeatedly. For instance, creating a new data record for a table is a fundamental task, which we repeat for each station's data. However, each station has different data and different ACCESS tables to be filled. We have a Perl subroutine called CreateRecord that has two arguments: Table and Fields. This routine can fill any ACCESS table just by passing the table name and the corresponding Fields hash, as will be shown below.

One nice aspect of Perl is its hash variable types, which are well meshed to database tables, fields and values. For example, if we have a table in ACCESS called ResistOperation with 4 fields: `IdTube, ResistN, ResistS` and `ResistDate`, we can easily mimic this in Perl with a hash data type:

```
$ResistOperation{IdTube} = 1;
$ResistOperation{ResistN} = 0.001;
$ResistOperation{ResistS} = 0.002;
$ResistOperation{ResistDate} = "8/18/99 12:43";
```

We could then create a new record in ACCESS for this table with the following code:

```
$Table = "ResistOperation";
$istat = CreateRecord($Table,\%ResistOperation);
if (!$istat){print "CreateRecord failed with $istat\n"}
```

It is important to note that we don't have to input every field of a table in this call…only the ones we have data for or have chosen to fill.

With this introduction as to how the current software tools are implemented, we can take up in more detail the nightly processing of data. There is a main Perl script, called `nightly_db.pl`, which is responsible for implementing all nightly operations for the production database. The production database is "attached" to a DSN called "UM_DB". This means that any software can "connect" to the data source by referring to that name, even during the update process. The first task is to copy the current production database file (called production.mdb) to a backup directory to allow recovery in the case of catastrophic errors. The backup file names are of the form production_yyyy_mm_dd.mdb. A backup failure halts the update process immediately.

Once the database is backed up, we scan for new production text files. We use Perl's database interface to maintain a status flag for all input text files. We have defined specific directories on each station's computer to hold the production text data files. In addition, each text file has a specific name format like <sid>_yyyy_mm_dd_hh_mm.dat to allow easy identification of the files. (<sid> is a station identifier string

like “wire” or “xray”) A scan of files matching the naming convention is performed for each station and a list is assembled. Our Perl file status database is then checked for each found file and all “new” text files are entered into the status database with `$dbstat{<file>} = 0`.

The status database is implemented very simply as a hash connected to a file. In practice this means we check `$dbstat{<filename>}` to determine the status of file `<filename>`. The current values of `$dbstat` are:

-5 = Unable to rename file after processing
-2 = Error processing this file during database update
-1 = Error during copy of this file from remote station computer
0 = File found on remote station computer but not yet copied
1 = File copied to database server
2 = File successfully processed into database (no warnings or errors)
3 = File successfully processed into database (warnings encountered)
4 = File successfully processed into database (errors encountered)
5 = Empty file

All files that have status 0 or –1 are copied to the database server computer’s input directory for processing. Note that we always work with local copies of the station text files. The originals are kept on each station as a backup. Also, each station has its own directory on the server for storing the text files. Once all files are copied from a station to the specific server directory, we attempt to process them into the database. Any files with status 1 or –2 are input to the update procedure. During processing, each file has an update logfile created that records any messages, warnings or errors encountered as part of the file processing.

A count of all records processed, warnings and errors is kept for each file. As a result of processing each file can have a status of –2, 2, 3 or 4. Files with status 2 or 3 are moved to a subdirectory called “Processed”. In addition, the logfile name is changed to reflect warnings or errors found. The update logfile starts out named <filename>.log. If warnings are encountered the name becomes <filename>_warnings.log and if errors are encountered it becomes <filename>_errors.log. The logfile is moved if the input text data file is moved, e.g., to the processed directory.

After all processing of text files completes, we update or recreate any tables which are generated as the result of queries. We have a separate Perl script which submits all of the make table queries in the correct order. The queries are stored inside the ACCESS database, rather than generated in the Perl scripts, since we feel the queries should be stored with the data. The generated tables summarize production status or create commonly requested data sets for ease of use, e.g., summary records of all tubes passing all QA tests, etc.

The third step we will discuss in greater detail in the next section. This step is responsible for generating all the static HTML pages and plots for use with the WWW interface.

A global log file is kept which records a summary line for each processed file at each station. This log file is then emailed to a list of people if any warnings or errors were encountered during processing. Otherwise a success email is sent instead.

An example of a data input Perl script is provided in Appendix 3, along with the URL of all the other scripts.

6.1.2 WWW ACCESS

A critical aspect of any database system is the user's ability to retrieve information. Of course the database manager has direct access to the database, but a key question for us is how do the many potential users get equivalent access? The WWW was developed to address the problem of making information widely available. There exist many standard tools for publishing data from a database to the WWW. ACCESS can publish tables and forms to HTML or active data technologies like IDC (Internet Database Connector) or ASP (Active Server Pages). In addition, a number of web authoring tools are available to aid in the process of creating Web content.

We have chosen ASP as our primary web-publishing tool for a number of reasons. ASP is the latest server-based technology from Microsoft designed to create interactive HTML pages for a WWW site. When accessed, ASP files produce up-to-date HTML information for WWW browsers. VBScript, JavaScript or PerlScript can provide the underlying programming functionality, as the programmer chooses. It is also very easy to use ADO and ODBC to publish specific content from a database, leveraging our database input programming for output uses.

For general access to our production database we have found an ASP based product which quickly and easily "publishes" our database to the WWW: ASP-db. There is a free version available for download from http://www.aspdb.com and commercial versions of varying capability for purchase. To demonstrate how easy it is to accomplish this task, here listed below is the complete code to access to any table in our production database, as well as filtering and download options:

```
<% Response.Buffer = True %>
   <HTML>
   <BODY>
   <CENTER>
   Welcome to the UM ATLAS Production Database Page.<P>
<%
 Set MyDb=Server.CreateObject("ASPdb.Free") ' Create the ASP-db object
 MyDb.dbQuickProps="1;UM_DB;*;grid;4,auto,lightgrey" ' Set its std prop.
 MyDb.dbDBType = "SQL"       ' It is an SQL database
 MyDb.dbNavigationItem = "top,prev,next,bottom,gridrow,filter,download,color,reload"

 MyDb.aspdbfree        ' Display it!
%>

   </CENTER>
   </BODY>
   </HTML>
```

The following figures show our current WWW production database interface. The examples documented here use Internet Explorer V5 with authentication enabled. The first figure shows the login box which appears to validate the user:

Figure 4 Authentication popup box for secure WWW access

After successfully authenticating, the user is given a choice of options concerning our production database.

Currently, the user has seven choices:

- They can view plots of production QA/QC information which are updated nightly
- Download a copy of the current export version of the ACCESS database (representing the version we "publish" for export to the main Global ACCESS database at Rome.
- Download an NTUPLE of the Summary_Table (described in Appendix 1 and 5). This NTUPLE is also generated nightly.
- Use ASP-db to interact with specific tables inside the ACCESS production database. This is implemented with the simple code fragment above.
- Get a listing of all the current Perl scripts in use and have the option to download any of them.
- Read a draft (eventually final) copy of this paper
- Examine the current production status via ACCESS exported ASP tables (see below)

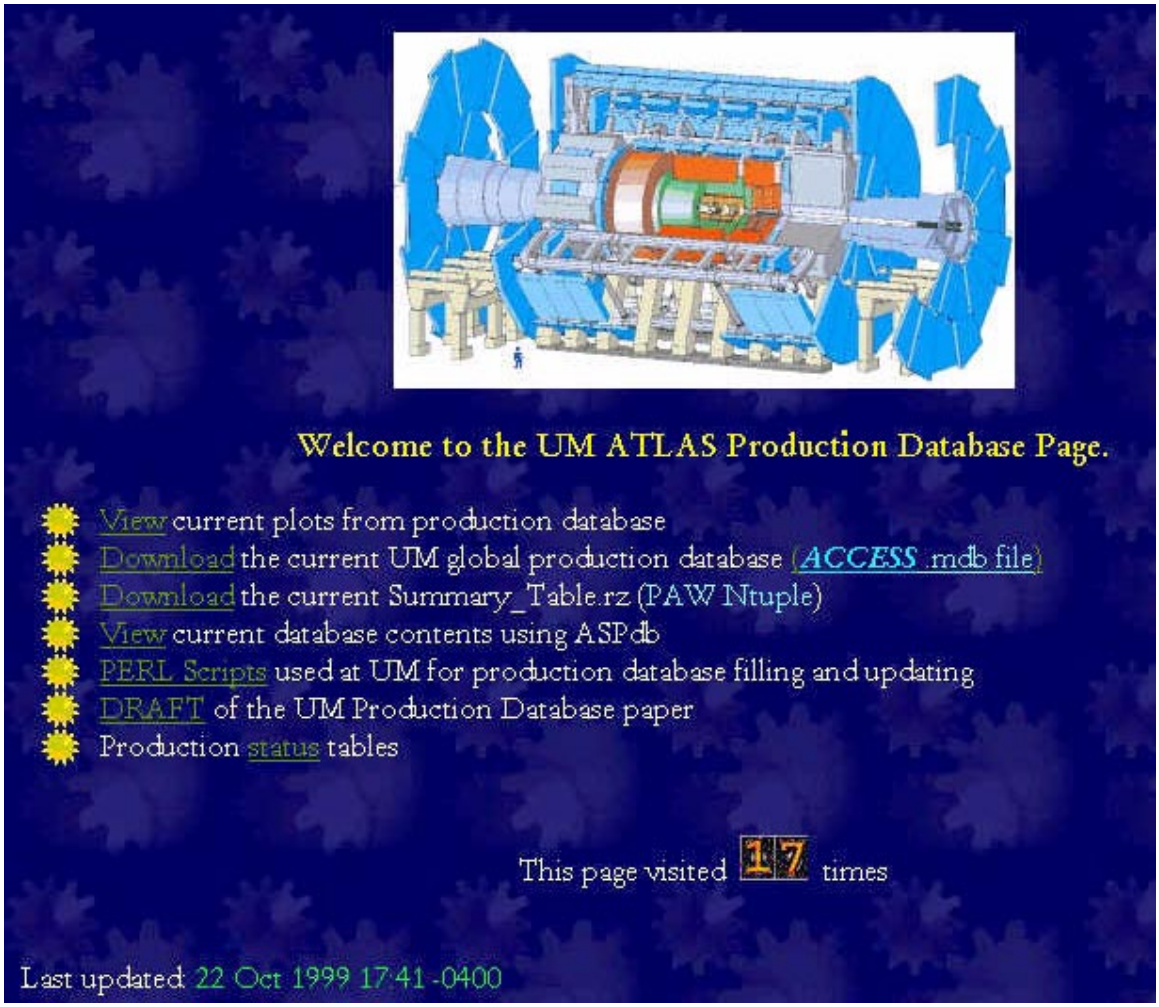


Figure 5 Main WWW page for the UM production database

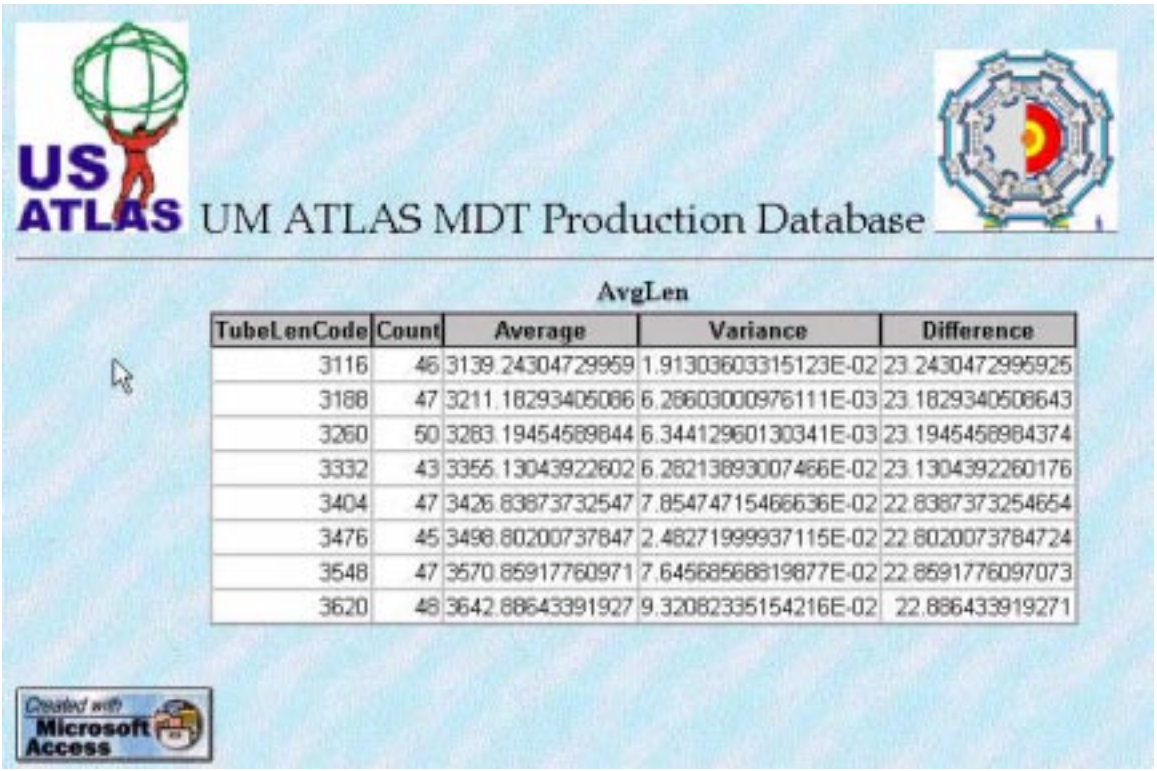
By selecting the "View" link on the main page the user arrives at the ASP-db interface page:

Figure 6ASP-db WWW page interface example

Figure 7 ACCESS Summary_Table shown on WWW using ASP-db

Any column of data can be sorted in ascending or descending order simply by clicking on the column. The data can be downloaded in to a local file for further processing or filtered in place using ASP-db.

Of course, ACCESS itself allows publishing directly to the WWW. By selecting a table or form and "Saving As…" an ASP file, we can create active WWW pages which remain connected to the ODBC data source. We have created a default template for our ACCESS exported tables and forms and an example of exporting the AvgLen query is shown here:



TubeLenCode	Count	Average	Variance	Difference
3116	46	3139.24304729959	1.91303603315123E-02	23.2430472995925
3188	47	3211.18293405086	6.28603000976111E-03	23.1829340508643
3260	50	3283.19454589844	6.34412960130341E-03	23.1945458984374
3332	43	3356.13043922602	6.28213893007466E-02	23.1304392260176
3404	47	3426.83873732547	7.85474715466636E-02	22.8387373254654
3476	45	3498.80200737847	2.48271999937115E-02	22.8020073784724
3548	47	3570.85917760971	7.64568568819877E-02	22.8591776097073
3620	48	3642.88643391927	9.32082335154216E-02	22.886433919271

Figure 8 Example of ACCESS exporting a table to an ASP page (Table is AvgLen)

The preceding table remains current, in that changes to the table in the ACCESS production.mdb file are reflected in the WWW table when it is viewed. This table shows an interesting systematic which arose during tube production: the absolute length scale shifted between production of tubes of length 3332 and 3404 (Also see the NTUPLE example in Appendix 5).

One of the primary shortcomings we are working on is the inability to plot frequency distributions through a WWW interface. Ideally the user could select a set of data and then click on a specific field to get a plot of that field's frequency distribution. One can imagine additional nice features that should be added to make the WWW interface a full analysis tool.

We have partially overcome this limitation by generating a set of useful plots during the nightly database processing. This allows a graphical view of the data for specifically chosen variables to be exported to static WWW pages each night. In addition, we generate an HBOOK NTUPLE version of our production summary table for downloading into PAW (see the next section and example in Appendix 5).

6.1.3 PAW IMPLEMENTATION

We recognize the need of those involved in the production process to visualize the data stored in the database. They should have a simple way to examine histograms of frequency distributions, and to make virtually unlimited complex cuts on the data. To achieve this goal, one can write his or her own programs, using Microsoft Visual Studio programming tools, which have powerful visualization functions, or we can turn to already mature scientific software analysis packages. As mentioned above, we have produced a Visual Basic Program to make a simple frequency distribution of any numeric database field for any dataset produced with user imposed filters. But, for more general analysis purposes, we have provided a PAW interface. PAW, because it is most familiar to high-energy physicists, and because it provides versatile and powerful capabilities for data analysis and visualization, is used as a bridge between the somewhat arcane database world and the analysis world in which most physicists live.

PAW (Physics Analysis Workstation) is a data analysis and presentation system included in the CERNLIB software package developed at CERN. It provides interactive graphical presentation and statistical or mathematical analysis, working on objects familiar to physicists such as histograms, NTUPLEs, vectors, etc. PAW is closely coupled with the FORTRAN language. It provides a set of commands acting on specific objects, and all the commands are organized in a useful tree structure. For complex operations, one can use Macro files, which are a set of stored command lines that can be created or modified with any text editor.

For sophisticated data structures, PAW uses NTUPLEs to store and analyze data. An NTUPLE is a set of events, where for each event the value of a number of variables is recorded. NTUPLEs are a very convenient tool for analyzing statistical datasets. An NTUPLE can be viewed as a table with each row corresponding to one event and each column corresponding to given variables. The interesting properties of the data in an NTUPLE can normally be expressed as distributions of NTUPLE variables or as correlations between two or more of these variables.

An NTUPLE is the basic type of data used in PAW. An NTUPLE is made available to PAW by opening a direct access file, which is created with a program using HBOOK. A storage area for an NTUPLE may also be created directly using NTUPLE/CREATE. Data may then be stored in the allocated space using the NTUPLE/LOOP or NTUPLE/READ commands.

At the same time, in our production database we have a lot of properties associated with a tube object. So it is natural to export the data stored in the ACCESS database to a row-wise NTUPLE file, in which each NTUPLE corresponds to the dataset of a tube. Then users can open PAW interface, read in the NTUPLEs from the data file, and make proper data analysis. An example of the use of an NTUPLE in analyzing database contents is provided in Appendix 5.

A Perl script (shown in Appendix 6) controls the process of making tables into an NTUPLE. The script is responsible for querying the database about a specified table, retrieving all field names and table contents and outputting a formatted text file. The script then runs the `Make_ntuple.for` code (also in Appendix 6), processing the text file into an RZ file containing the NTUPLE. It is important to note that any ACCESS table can be automatically converted to an NTUPLE with this method.

Notes:

1) Authorized users can get PAW from CERN by the ftpsite: asisftp.cern.ch

2) PAW supports Unix-like systems(HP-UX, Solaris, PC Linux, etc), and Windows NT/95 Systems.

3) Useful information can be found at the PAW webpage: http://wwwinfo.cern.ch/asd/paw

6.1.4 UM EXPORT DATABASE AND ITS INTERFACE TO UM MAIN PRODUCTION DATABASE

As indicated in the introductory section, we have decided to maintain a separate database, tightly coupled to the main database, which is to be accessible to any members of the collaboration, and especially the managers of the global muon database, who wish to view and/or retrieve information about the status of the UM muon chamber production. Having two databases permits us to insert a formal certification process before providing data (some of which may be of a preliminary nature) to a wider audience. In addition, a separate database also permits locally controlled changes in the main database to take place without creating problems for the global database, which would only be changed in synchronization with centrally orchestrated ATLAS changes.

The logical relation between the two databases is shown in Figure 1. Only authorized individuals within our group transfer data from the Main database to the UM-Export Database using a set of Perl scripts designed to select and transfer the data. Only "completed" tubes are copied into the export database. A "completed" tube is either a tube which has passed all QA/QC checks and is ready to go into a chamber or one which has terminally failed a check and will be discarded. Note that there is a third possibility, that a tube fails a QA/QC check but may be recoverable through additional intervention. Until such intervention is complete we withhold that tube's data.

Once the data is "published" to the export database, we lock all of our corresponding local data, since we have ceded control of it to the global system. This is done via a data record field which indicates whether or not the corresponding data record has been used to "publish" data or not. Modifications are not allowed to records that have this field checked. Mechanisms are under discussion for ways to insure that managers of the global database are aware of any actions to correct errors or to update information.

The exact format of the global database and the process for transferring the data to master copy of the global database has been defined. We provide a URL which "points" to the export database which we periodically update. The Rome group can then "scan" this database, looking for additions that are then incorporated into the global database.

7. OBJECT ORIENTED DATABASE CONSIDERATIONS

There has been much discussion of the benefits of object oriented databases over the more traditional relational databases. Clearly, before embarking upon the establishment of a new relational database for use in a high energy physics experiment that will extend into the next decade, one must carefully assess what opportunities will be missed by not adopting an OO database paradigm.

" The very root of ODBMS technology is to store complex objects and the complex relationships between them. People familiar with relational technology sometimes suggest that objects are simply rows in tables, and the data members in an object are like the columns that make up a row. While this model appears to hold up at first glance, further examination shows that objects have many other characteristics that distinguish them from being represented as rows of data members. Specifically, objects have methods and direct relationships to other objects. And, quite importantly, objects can exist independently, in both a logical and physical sense, from any object of the same class. Clearly, this is a departure from the tabular storage of information found in RDBMS, which group all items of the same type in a single location. "

excerpt from "Choosing an Object Database," whitepaper at Objectivity website

Another important consideration, however, is that one should not take a relatively simple problem and turn it into a major one just to satisfy a desire to use only the latest programming paradigms. Here we outline why we think it appropriate to begin our database effort using the relational database ACCESS, while conducting ongoing R&D to determine the appropriateness of a possible later conversion to an OO facility.

With an object oriented database one would hope to gain the benefits of:

- Object Persistency
- Class Inheritance
- Code reusability
- Easy access to stored data by analysis routines
- Easy federation of databases so they can be accessed by many users remotely
- Easy communication between high level analysis packages, which will surely use OO based code, and the raw production data, which may on occasion need to be retrieved by the high level package.

If one wishes to argue that none of these features are relevant to the design of the production databases, cogent arguments can be advanced.

- Regarding persistency, there is no problem. A manufactured drift tube is going nowhere, except to its known place in a chamber, or a junk pile. And a relational database that is properly constructed can keep track of every tube with no difficulty.
- The benefits of inheritance are primarily associated with applications where there are many classes of objects with meaningful relationships. For the production database there are few such object classes. There are endplugs, wires, tubes and chambers. Moreover, objects in each class are simply related. For example, here are short tubes and longer tubes.
- Benefits of being able to reuse code are clear – but there is very little code to be re-used in the production database. In the most elemental stage, we mainly want to look at data and calculate simple variances,

It is clearly of value to be able to simply access the database's fields and tables from any Fortran or C++ program a user may write. This opens up the database to virtually any kind of analysis desired. But, as we will show below, rather simple interfaces can be written for ACCESS that will provide that functionality. One cost of that functionality is the execution speed due to the necessity of making database transactions in an environment that is foreign to the application. But, given the scope of the database and the suitability of a relaxed response rate to most queries, it is hard to see that this is a driving justification for an OO database.

While database federation is desirable, achieving such a federation is possible with both relational and OO databases.

Finally, it is possible to imagine the benefits of a high level C++ program being able to reach seamlessly into a particular production database to extract parameters to permit a re-calculation to be made of a certain physics variable. But one should question whether this is even an option that should be desired. The pathway for such a reconstruction change should probably be through other channels, where parameters would be revised using a system that would be perfectly comfortable using a regular relational database.

So, one can see why it is possible to be satisfied that a relational database would be the appropriate vehicle for recording and displaying production data for the muon chambers.

But, before leaving the topic, we should note that there are some attractive features of viewing the chamber elements as strict objects, rather than just entities that occupy a row in a spreadsheet.

When, for example, a drift tube is viewed as an object, we can be relatively unrestricted in what kind of data is kept for that tube. Not only can the tube tell us how long it is, what the tension is of its wire, but it can almost as easily share with us logbook images of the terrible time we had in getting it to work. As time changes we can alter what features of its rich past that should be shared with its colleagues in the chamber.

We wished to not only guess what the differences would be between an ACCESS rendition of the production database and a OO rendition, but to actually assess the differences in a real world setting. To this end we have actually constructed a Objectivity version of the database as well. Some of the pluses and minuses of the OO approach will be discussed in the next section.

8. OODBMS STUDY

Nothing could inform us better about the pros and cons of OODBMS than going through the process of building one explicitly for our application. We thus set about to use Objectivity to build a parallel object database for our tube production, and we used it to solve some real problems such as generating a report on the current tube production.

In our object database, each tube is an object. Since a tube may have repeated measurements in one station, we have to dynamically allocate space for the new measurement data. We did that by using measurement classes. The tube object does not actually hold data; it contains associations (that is, logical links to other classes) to instances of measurement classes. Whenever the tube undergoes a measurement in any of the stations, the database will create a new measurement object to store the data, thus creating a linked list of measurement objects.

We defined the following persistent classes in our object database:

- Tube: corresponding to a tube in the production

- CrOperation: a measurement class storing data in the Cr Station measurement
- DCOperation: a measurement class storing data in the DC Station measurement
- LeakChkOperation: a measurement class storing data in the LeakChk Station measurement
- ResistOperation: a measurement class storing data in the Resist Station measurement
- WirOperation: a measurement class storing data in the Wir Station measurement
- XrayOperation: a measurement class storing data in the Xray Station measurement

In order to reduce data redundancy, we also defined Run classes and Parameter classes for each station.

We have left out the FreqOperation and EmmiOperation tables, since they were not used when the study was initiated.

Each class corresponds to a table in our ACCESS database. The structure of our object database is similar to that of the ACCESS database because, first, we want it to store all the data in our current ACCESS database; and second, because the structure of our ACCESS database matches the actual procedures of the tube measurements. It is quite natural to have the object database reflect such procedures as well.

Here is the definition of some key classes in Objectivity:

```
class Tube: public ooObj{
   private:
        int IdTube;
        int SCode;        // Status code
        long Rjcode;
        long TubeLayerLoc;
        long Location;
        char RecycleEndPlugs; // y -- yes; n -- no
        ooVString comment;

   public:
            // Below are associations to measurement classes.
        ooRef(CrOperation) CrData : copy(delete);
        ooRef(DCOperation) DCData : copy(delete);
        ooRef(LeakChkOperation) LeakChkData : copy(delete);
        ooRef(ResistOperation) ResistData : copy(delete);
        ooRef(WirOperation) WirData : copy(delete);
        ooRef(XrayOperation) XrayData : copy(delete);

        Tube(long Id); // constructor
        ooRef(Layer) p_layer <-> p_tube[]; /reference to the higher level: layer

        int put_stuff(char* name, char* value, int count);
   // data retrieval methods
        int get_count(char* name);
        int get_latest(char* name);
        long get_IdTube(){ return IdTube; }
        int get_SCode(){ return SCode; }
        long get_Rjcode(){ return Rjcode; }
        long get_TubeLayerLoc(){ return TubeLayerLoc; }
        long get_Location() { return Location; }
};

class DCOperation : public ooObj{
   private:
```

```
        int MyID;
        int ADCChan;
        float DarkCurAve;
        float DarkCurSDev;
        float DarkCurMax;
        ooVString StartDT;
        ooVString EndDT;
    public:
        ooRef(DCRun) RunData : copy(delete);
        int put_stuff(char* name, char* value);
        int get_MyID(){ return MyID; }
        int get_ADCChan();
        float get_DarkCurAve(){ return DarkCurAve; }
        float get_DarkCurSDev();
        ooVString get_StartDT(){ return StartDT; }
        ooVString get_EndDT();
        // Association for the next measurement
        ooRef(DCOperation) nextOne : copy(delete);
};

class DCRun : public ooObj{
    private:
        long ID;
        ooVString Operator;
        ooVString StartTime;
        ooVString EndTime;
        int StationID;
        ooVString InputFile;
    public:
        ooRef(DCParam) ParamData : copy(delete);
        int put_stuff(char* name, char* value);

};

class DCParam : public ooObj{
    private:
        long ID;
        float DarkCurPercAr;
        float DarkCurPercCO2;
        float HVOp;
        float PrsIn;
        float PrsOut;
        float TempStart;
        float TempEnd;
    public:
        int put_stuff(char* name, char* value);
};
```

Class definitions for the other stations are similar.

Since these classes inherit from the base class (ooObj), they are persistent classes in Objectivity. Their data are kept in the database after the program ends, in contrast to the volatile classes whose data will be lost.

8.1 A COMPARISON BETWEEN ACCESS AND OBJECTIVITY

As stated above, our goal in preparing duplicate ACCESS and Objectivity databases was to assess their relative efficacy in solving real problems. Here we examine an example of how to generate a summary table in ACCESS and Objectivity. We have previously encountered the need for a summary table for the status of each tube at each station. This summary table should contain the latest measurement of each tube at each station, and show the number of measurement it underwent at each station.

We want to use the Join Table query to generate that summary table. So we add a reference field for each station at the tube table pointing to the latest measurement at each "Operation" table.

The complicated part in this task is that sometimes we do not want to use the latest measurement at a station for a specific tube (for example, the data in the latest measurement at that station is shown not to be the one that should be definitive). So we added a check box field in the tube table for each station, indicating whether we want to use the latest measurement record. If the box is checked, the desired record ID is indicated, instead of the latest record ID, in the reference field for that station.

Another complication is that in the Xray Station, we measure both ends of the tube: south and north, and keep separate records in the "XrayOperation" table. While in the summary table, we want to have each record show the data for both ends.

To accomplish these tasks in ACCESS, we use a set of SQL statements to update the record ID fields in the Tube table. First we use Make Table query to generate an intermediate table for each station, giving the latest (or preferred) measurement data. Here is the list of those queries:

Query for Cr Station (MakeCrTemp):

```
SELECT IdTube, count(IdTube) AS [Count], max(ID) AS BestID INTO CrTemp
FROM CrOperation
GROUP BY IdTube;
```

Query for DC Station (MakeDCTemp):

```
SELECT IdTube, count(IdTube) AS [Count], max(ID) AS BestID INTO DCTemp
FROM DCOperation
GROUP BY IdTube;
```

Query for LeakChk Station (MakeLeakChkTemp):

```
SELECT IdTube, count(IdTube) AS [Count], max(ID) AS BestID INTO LeakChkTemp
FROM LeakChkOperation
GROUP BY IdTube;
```

Query for Resist Station (MakeResistTemp):

```
SELECT IdTube, count(IdTube) AS [Count], max(ID) AS BestID INTO ResistTemp
FROM ResistOperation
GROUP BY IdTube;
```

Query for Wiring Station (MakeWiredTemp):

```
SELECT IdTube, count(IdTube) AS [Count], MAX(WirOperation.ID) AS BestID INTO WiredTemp
FROM WirOperation
GROUP BY WirOperation.IdTube;
```

Query for Xray Station at North End (MakeXrayTempN):

```
SELECT IdTube, count(IdTube) AS CountN, max(XrayDT) AS XrayDTN, max(ID) AS BestIDN INTO XrayTempN
FROM XrayOperation
WHERE TubeEnd="n" or TubeEnd="N"
GROUP BY IdTube;
```

Query for Xray Station at South End (MakeXrayTempS):

```
SELECT IdTube, count(IdTube) AS CountS, max(XrayDT) AS XrayDTS, max(ID) AS BestIDS INTO XrayTempS
FROM XrayOperation
WHERE TubeEnd="s" or TubeEnd="S"
GROUP BY IdTube;
```

Then we use six Update queries to update the Tube table:

Query for Cr Station (UpdateCr):

```
UPDATE tube, CrTemp SET Tube.CrID = [CrTemp].[BestID]
WHERE [Tube].[IdTube]=[CrTemp].[IdTube] And [Tube].[CrOver]=No;
```

Query for DC Station (UpdateDC):

```
UPDATE tube, DCTemp SET Tube.DCID = [DCTemp].[BestID]
WHERE [Tube].[IdTube]=[DCtemp].[IdTube] And [Tube].[DCOver]=No;

Query for LeakChk Station (UpdateLeakChk):
UPDATE tube, LeakChkTemp SET Tube.LeakChkID = [LeakChkTemp].[BestID]
WHERE [Tube].[IdTube]=[LeakChktemp].[IdTube] And [Tube].[LeakChkOver]=No;

Query for Resist Station (UpdateResist):
UPDATE tube, ResistTemp SET Tube.ResistanceID = [ResistTemp].[BestID]
WHERE (((tube.IdTube)=[ResistTemp].[IdTube]) AND (([Tube].[ResistanceOver])=No));

Query for Wiring Station (UpdateWire):
UPDATE tube, WiredTemp SET Tube.WiredID = [WiredTemp].[BestID]
WHERE [Tube].[IdTube]=[Wiredtemp].[IdTube] And
   [Tube].[WiredOver]=No;

Query for Xray Staion at North End (UpdateXrayN):
UPDATE tube, xrayTempN SET Tube.xrayNID = [xrayTempN].[BEstIDN]
WHERE [Tube].[IdTube]=[xraytempN].[IdTube] And
   [Tube].[xraySOver]=No;

Query for Xray Station at South End (UpdateXrayS):
UPDATE tube, xrayTempS SET Tube.xraySID = [xrayTempS].[BEstIDS]
WHERE [Tube].[IdTube]=[xraytempS].[IdTube] And
   [Tube].[xraySOver]=No;
```

Finally, we used Make Table query to generate the summary table. We have to use two queries to achieve that because of the “South and North” complication mentioned above.

```
First Query (Summary_Table_half):
SELECT Tube.IdTube, Tube.SCode, DCOperation.DarkCurAve, DCTemp.Count AS DCCnt,
LeakChkOperation.LeakDT, LeakChkOperation.LeakRate, LeakChkTemp.Count AS LCCnt, ResistOperation.ResistDT,
ResistOperation.ResistN, ResistOperation.ResistS, ResistTemp.Count AS RCnt, WirOperation.WirStartDT,
WirOperation.TubeLenCode, WirOperation.TubeLen, WirOperation.WirFreq, WirOperation.Tension, WiredTemp.Count
AS WCnt, XrayOperation.XrayDT AS XrayDTS, XrayOperation.XrayOff AS XrayOffS, XrayTempS.CountS AS CntS
INTO Summary_Table_half
FROM (((((((((Tube LEFT JOIN DCTemp ON tube.idtube=DCTemp.IdTube) LEFT JOIN DCOperation ON
Tube.DCID=DCOperation.ID) LEFT JOIN LeakChkTemp ON Tube.IdTube=LeakChkTemp.IdTube) LEFT JOIN
LeakChkOperation ON Tube.LeakChkID=LeakChkOperation.ID) LEFT JOIN ResistTemp ON
Tube.IdTube=ResistTemp.IdTube) LEFT JOIN ResistOperation ON Tube.ResistanceID=ResistOperation.ID) LEFT
JOIN WiredTemp ON Tube.IdTube=WiredTemp.IdTube) LEFT JOIN WirOperation ON Tube.WiredID=WirOperation.ID)
LEFT JOIN XrayTempS ON Tube.IdTube=XrayTempS.IdTube) LEFT JOIN XrayOperation ON
Tube.XraySID=XrayOperation.ID
WHERE Tube.SCode = 1 OR Tube.SCode > 4;

Second Query (Summary_Table):
SELECT Summary_Table_half.*, XrayOperation.XrayDT AS XrayDTN, XrayOperation.xrayOff AS XrayOffN,
XRayTempN.CountN AS CntN INTO Summary_Table
FROM (Summary_Table_half LEFT JOIN xrayTempN ON Summary_Table_half.IdTube=XrayTempN.IdTube) LEFT
JOIN XrayOperation ON XrayTempN.BestIDN=XrayOperation.ID
WHERE Summary_Table_half.SCode = 1 OR Summary_Table_half.SCode>4;
```

Finally we generated the summary table. We used nine Make Table queries, eight intermediate tables and seven Update Table queries to get the result.

Of course, we can write external programs to handle this. But in our program, we have to include some database access tools to access the data.

Now here is how we do it in Objectivity. Since in our object database schema, we use linked lists to store the measurement objects, there is no need to keep a “record ID” for each station. Instead, we just keep an association pointer to point to the first measurement. But in order to deal with the “Not the Latest One”

complication, we have to keep a record to indicate the desired measurement, if we don't want to see the latest measurement data.

Now the Tube class definition becomes:

```
class Tube: public ooObj{
   private:
         int IdTube;
         int SCode;        // Status code
         long Rjcode;
         long TubeLayerLoc;
         long Location;
         char RecycleEndPlugs; // y -- yes; n -- no
         ooVString comment;

   public:
            // Below are associations to measurement classes.
         ooRef(CrOperation) CrData : copy(delete);
         ooRef(DCOperation) DCData : copy(delete);
         ooRef(LeakChkOperation) LeakChkData : copy(delete);
         ooRef(ResistOperation) ResistData : copy(delete);
         ooRef(WirOperation) WirData : copy(delete);
         ooRef(XrayOperation) XrayData : copy(delete);

         Tube(long Id); // constructor
         ooRef(Layer) p_layer <-> p_tube[]; /reference to the higher level: layer
         int put_stuff(char* name, char* value, int count);

// New part: indicate the desired measurement instance that we want to see in the summary table.
         ooRef(CrOperation) CrDesired :copy(delete);
         ooRef(DCOperation) DCDesired :copy(delete);
         ooRef(LeakChkOperation) LeakChkDesired : copy(delete);
         ooRef(ResistOperation) ResistDesired : copy(delete);
         ooRef(WirOperation) WirDesired : copy(delete);
         ooRef(XrayOperation) XraySDesired : copy(delete);
         ooRef(XrayOperation) XrayNDesired : copy(delete);
// End of new part.

   // retrieve data methods
         int get_count(char* name);
         int get_latest(char* name);
         long get_IdTube(){ return IdTube; }
         int get_SCode(){ return SCode; }
         long get_Rjcode(){ return Rjcode; }
         long get_TubeLayerLoc(){ return TubeLayerLoc; }
         long get_Location() { return Location; }
};
```

Here is the code that we generate the summary table in Objectivity:

```
// this function omitted the database initialization in the program, which should
// be done before accessing the data.
void genrate_summary_table(){
   int i;
   char pred[20];
   FILE* stream; // will write the result summary table to this text file.
   ooItr(Tube) TubeItr;
   ooItr(DCOperation) DCItr;
   ooItr(LeakChkOperation) LeakChkItr;
   ooItr(ResistOperation) ResistItr;
   ooItr(WirOperation) WirItr;
   ooItr(XrayOperation) XrayItr;
   stream=fopen("summary.txt", "w");
```

```
    fprintf(stream,
"IdTube;SCode;DarkCurAve;DCCnt;LeakDT;LeakRate;LCCnt;ResistDT;ResistN;ResistS;RCnt;WirStartDT;TubeLenCo
de;TubeLen;WirFreq;Tension;WCnt;XrayDTS;XrayOffS;CntS;XrayDTN;XrayOffN;CntN\n");
    TubeItr.scan(contHandle);
    while(TubeItr.next()){
        fprintf(stream, "%d;%d;", TubeItr->get_IdTube(),
                                        TubeItr->get_SCode());
        if(TubeItr->DCDesired().isNull())
            i=TubeItr->get_latest("DCOperation");
        else
            i=(TubeItr->DCDesired()).get_MyID();
            if(i==0)
            fprintf(stream, ";;");
        else{
            sprintf(pred, "MyID==%d",i);
            DCItr.scan(contHandle, oocUpdate, oocPublic, pred);
            fprintf(stream, "%f;%d;", DCItr->get_DarkCurAve(),
                                        TubeItr>get_count("DCOperation"));
        }
        fprintf(stream, "\t");
        if(TubeItr->LeakChkDesired().isNull())
            i=TubeItr->get_latest("LeakChkOperation");
        else
            i=(TubeItr->LeakChkDesired()).get_ID();
        if(i==0)
            fprintf(stream, ";;;");
        else{
            sprintf(pred, "ID==%d",i);
            LeakChkItr.scan(contHandle, oocUpdate, oocPublic, pred);
            fprintf(stream, "%s;%f;%d;",
                (const char *)(LeakChkItr->get_LeakDT()),
                LeakChkItr->get_LeakRate(),
                TubeItr->get_count("LeakChkOperation"));
        }
        fprintf(stream, "\t");
        if(TubeItr->ResistDesired().isNull())
            i=TubeItr->get_latest("ResistOperation");
        else
            i=(TubeItr->ResistDesired()).get_ID();
        if(i==0)
            fprintf(stream, ";;;");
        else{
            sprintf(pred,"ID==%d",i);
            ResistItr.scan(contHandle, oocUpdate, oocPublic, pred);
            fprintf(stream, "%s;%f;%f;%d;",
                (const char *)(ResistItr->get_ResistDT()),
                ResistItr->get_ResistN(),
                ResistItr->get_ResistS(),
                TubeItr->get_count("ResistOperation."));
        }
        fprintf(stream, "\t");
        if(TubeItr->WirDesired().isNull())
            i=TubeItr->get_latest("WirOperation");
        else
            i=(TubeItr->WirDesired()).get_ID();
        if(i==0)
            fprintf(stream,";;;;;");
        else{
            sprintf(pred,"ID==%d",i);
            WirItr.scan(contHandle, oocUpdate,oocPublic, pred);
            fprintf(stream, "%s;%d;%f;%f;%f;%d;",
                (const char *)(WirItr->get_WirStartDT()),
                WirItr->get_TubeLenCode(),
                WirItr->get_TubeLen(),
                WirItr->get_WirFreq(),
                WirItr->get_Tension(),
                TubeItr->get_count("WirOperation"));
        }
```

```
            fprintf(stream, "\t");
            if(TubeItr->XraySDesired().isNull())
                  i=TubeItr->get_latest("XrayOperation_S");
            else
                  i=(TubeItr->XraySDesired()).get_ID();
            if(i==0)
                  fprintf(stream,";;;");
            else{
                  sprintf(pred,"ID==%d",i);
                  XrayItr.scan(contHandle, oocUpdate, oocPublic, pred);
                  fprintf(stream,"%s;%f;%d;",
                        (const char*)(XrayItr->get_XrayDT()),
                        XrayItr->get_XrayOff(),
                        TubeItr->get_count("XrayOperation_S"));
            }
            fprintf(stream,"\t");
            if(TubeItr->XrayNDesired().isNull())
                  i=TubeItr->get_latest("DCOperation_X");
            else
                  i=(TubeItr->XraySDesired()).get_ID();
            if(i==0)
                  fprintf(stream,";;;\n");
            else{
                  sprintf(pred,"ID==%d",i);
                  XrayItr.scan(contHandle, oocUpdate, oocPublic, pred);
                  fprintf(stream, "%s;%f;%d;\n",
                        (const char*)(XrayItr->get_XrayDT()),
                        XrayItr->get_XrayOff(),
                        TubeItr->get_count("XrayOperation_N"));
            }
      }
      fclose(stream);
}
```

We observe that the code uses a number of methods (or functions) of different classes:

Tube::get_latest(char* name) → returns the latest measurement object in the database.

Tube::get_count(char* name) → returns the number of measurements carried out in one station.

Based on the above two ways we use in ACCESS and Objectivity and our experience in using both of them, we can summarize the difference between the two database management systems.

One difference between Objectivity and ACCESS database is how we write programs to implement some functions on top of them. Objectivity provides a more natural way to integrate the database with your C++ (or Java) programs. One can apply object-oriented methodology to it: using encapsulation to protect the data, using inheritance to share the same interfaces, and so on. One can put the useful functions inside the persistent class definition, thus integrating the implementation with the database.

Another difference is how to make queries to the database. ACCESS provides Query by Example (QBE) interface, and SQL tools to do that. While in Objectivity, the usual way is to use iterators for a certain class and select qualified objects. Because of its close integration with the programming language (C++ or Java), we can make sophisticated query requirement. In ACCESS we have to use external programs and use those database access API's (such as ODBC and Microsoft ADO) to do that.

In general, relational database provides a simple and elegant way to store data. However, it is not suitable to express such data attributes as encapsulation, inheritance, polymorphism, references among objects, and

complex data structure. The advantage of OODBMS lies in its support for considerably richer data types than is available in a relational DBMS: it supports user-defined abstract data types (ADT's), structured types, and inheritance.

However, such data type requirement does not exist in our production database: Some simple Ntuples can represent all the data. And the way to access the data in the ACCESS database does not cause too much inconvenience. This makes us believe that ACCESS can suffice the database task in our chamber production.

9. PLANNED FUTURE EXPLORATIONS

9.1.1 CRISTAL

CRISTAL (Cooperating Repositories and an Information System for Tracking Assembly Lifecycles) is used extensively in CMS production. Its architecture is based on Workflow and Product Data Management techniques. It uses OODBMS (currently Objectivity) as the repository for the construction and assembly data and incorporates CORBA technology as the mechanism for distributing functions.

A CRISTAL system should consist of three parts: the Central System, the Local Center, and the Operator Station. The Central System defines the definition of production parts, the workflow of the production, and populates the definitions to the local sites. It also stores all the data sent from the local production sites. The Local Center applies new production scheme, register instrument, and monitors the local production. And Operator Stations will involve direct production and measurement.

Since CRISTAL is dealing with distributed production activity, it uses the CORBA (Common Object Request Broker Architecture) to transfer objects in the OODBMS. CORBA is an industry standard for the development and deployment of applications in distributed, heterogeneous environments. The data stored in a local center will be automatically sent to the central system for storage.

We find the most attractive feature of CRISTAL is its capability to control the workflow of the production. You can centrally define the steps in the process of production and measurement, and define the allowed ranges of measurement data, which provides a quality assurance. Though many functions of CRISTAL are still being implemented, you now can view the data stored in the database, and export the data into an Excel spreadsheet.

9.1.2 ACCESS UPGRADE PATHS

We have tried to isolate the individual software components of our production database system from one another. Data generation is independent of data storage and retrieval, which is independent of data presentation and analysis. We may find that during production ACCESS has undesirable limitations (size of data stored, performance, ease of access, multiple users, etc.) and we can then consider upgrade options to remedy the problem(s). The most obvious change would be to upgrade to the next version of ACCESS: ACCESS 2000. This should be almost painless, even if we hadn't been careful to isolate storage/retrieval from the data source and from presentation and analysis, since Microsoft has to consider backward compatibility when designing new software versions. If performance, data size or multiple simultaneous users are problems, SQL Server V7.0 is a much more likely upgrade path. Fortunately, there are tools within SQL Server that convert ACCESS databases to SQL databases with minimal user intervention. Since we have chosen to rely on UDA (and ADO + ASP), our presentation and data inputs tools will require NO changes.

The only required change is to modify the ODBC data source to use the SQL driver, rather than the ACCESS driver.

9.1.3 WEB PLOTTING PACKAGES

As we pointed out earlier, a significant shortcoming of our WWW interface is a lack of plotting capability. We are researching tools that will allow us to plot frequency distribution using ASP and ADO. This is ongoing work and we will publish our findings in a future paper.

10. USEFUL READING

10.1 ACCESS

- "Platinum Edition Using Access 97", Que Corporation, 1997.
- "Access Database Design and Programming", O'Reilly & Associates, 1997.

10.2 PERL

- "Learning Perl on Win32 Systems", O'Reilly & Associates, 1997.
- "Programming Perl", O'Reilly & Associates, 1997.

10.3 ADO

- "ADO 2.0 Programmers Reference", Wrox Press, 1998. (New version ADO 2.1 out now, 1999.)

10.4 ASP

- "Professional Active Server Pages 2.0", Wrox Press, 1998.
- "Developing ASP Components", O'Reilly & Ass

11. APPENDICES

11.1 APPENDIX 1

11.1.1 SUMMARY OF THE STRUCTURE OF THE CURRENT PRODUCTION DATABASE, AND A FULL LISTING OF ALL THE FIELDS AND TABLES IN THE CURRENT PRODUCTION DATABASE:

The structure of the production database reflects the current measurement procedures. All the measurements are carried out in eight main “stations”:

1) Wiring Station
2) Leak Checking Station
3) Dark Current Station
4) Cosmic Ray Station
5) X-ray Station
6) Emmi Station
7) Resistance Station
8) Frequency Station

As the name suggests, each station carries out one specific task. Each station is manned with one to two operators. Each station generates separate text data files, which will be processed by Perl programs to input data into the database.

We define a “run” for a period of measurement in each station. During a “run”, the measurement in that station is associated with a set of parameters with fixed values. For example, each “run” in the Dark Current station is associated with such parameters as the argon percentage in dark current gas (DarkCurPercAr), CO2 percentage in the dark current gas (DarkCurPercCO2), Operational HV (HVOp), average input pressure during this run, and so on. Since they are constant data in that “run”, we set up a separate parameter table for each station to reduce data redundancy in the database.

Here is the listing of the tables and their fields for each station:

11.1.1.1 Wiring Station

Table 1: WirOperation

Id	Primary Key
Idtube	Tube Id From Barcode
Wirerun	Pointer To Wiring Run Table
Tubem1code	Manufacture Id #1
Tubem2code	Manufacture Id #2
Wirstartdt	Timestamp Start Of Tube Wiring
Tubelencode	Barcode Of Tube Length (Mm)
Humidity	Humidity At Wiring (%)
Pressure	Pressure At Wiring (Psi)
Wirtemp	Temperature At Wiring (°C)
Wirdt	Timestamp For Wiring Operation
Swaghv	Hv For Swagging (Kv)
Cleancode	Cleaning Code (1=None, 2=Alcohol, 3= Scotch-Brite+ Alcohol)
Wirfreq	Measured Wire Frequency (Hz)
Wirfreqtemp	Temperature During Frequency Measurement (°C)
Wirfreqdt	Timestamp For Wire Frequency Measurement
Tubelen	Measured Tube Length (Mm)
Tubelendt	Timestamp For Tube Length Measurement
Tension	Calculated Wire Tension (G)
Wirenddt	Timestamp For End Of Wiring Operation

Table 2: Wirrun

Runrecord	The Run Number For A Wiring Station Session
Wirparam	The Wiring Station Parameters For This Run
Operator1	First Operator For This Session
Operator2	Second Operator For This Session
Starttime	Begin Run Date/Time
Endtime	End Run Date/Time
Stationid	Pointer To Station Id Table
Inputfile	Filename Of Input File

Table 3: Wirparam

Wirparam	Wiring Table Parameters
Wirspool	Wire Batch Spool Lot Number
Wirgold	Percentage Of Gold On Wire
Plugprodbat	Plug Production (Format:Pluga_Producer_Plugb)
Tension	Desire Tension (G)
Tubelendgth	Mdt Desired Tube Length For This Setup (Mm)
Frqdrive	Driving Frequency For Wire Tension Test (Hz)
Frqpuldur	Driving Frequency Pulse Duration (S)
Frqwait	Waiting Time After Driving Pulse (S)
Frqsamp	Sampling Frequency For Tension Determination (Khz)
Stretchten	Wire Stretching Tension (G)
Strechtime	Wire Stretching Time (S)
Pincrimp	Pin Crimping Diameter (Microns)

11.1.1.2 Leak Check Station

Table 4: Leakchkoperation

Id	Primary Key
Idtube	Tube Id From Barcode
Leakrate	Gas Leak Rate
Leakraten	Gas Leak Rate On North End Of Tube
Leakrates	Gas Leak Rate On South End Of Tube
Leakbody	Gas Leak Rate On Body Of Tube
Leakdt	Timestamp For Leak Check
Leakrun	Pointer For Leak Check Run Table

Table 5: Leakchkrun

Runrecord	The Run Number For A Leak Check Station Session
Crparam	The Leak Check Station Parameters For This Run
Operator	First Operator For This Session
Starttime	Begin Run Date/Time
Endtime	End Run Date/Time
Stationid	Pointer To Station Id Table
Inputfile	Filename Of Input File

Table 6: Leakchkparam

Leakchkparam	Parameter Pointer
Machinesettings	Leak Check Machine Parameters In Compressed Format

11.1.1.3 Dark Current Station

Table 7: Dcoperation

Id	Primary Key
Idtube	Tube Id From Barcode
Adcchan	Adcchannel Number For Test
Darkcurave	Average Dark Current
Darkcursdev	Standard Deviation Of Dark Current
Darkcurmax	Maximum Dark Current During Run
Startdt	Timestamp For Start Of Operation
Enddt	Timestamp For End Of Operation
Dcrun	Pointer To Dark Current Run Table

Table 8: Dcrun

Runrecord	The Run Number For A Dark Current Station Session
Dcparam	The Dark Current Station Parameters For This Run
Operator	First Operator For This Session
Starttime	Begin Run Date/Time
Endtime	End Run Date/Time
Stationid	Pointer To Station Id Table
Inputfile	Filename Of Input File

Table 9: Dcparam

Dcparam	Parameter Pointer
Darkcurpercar	Argon Percentage In Dark Current Gas
Darkcurpercco2	Co2 Percentage In Dark Current Gas
Hvop	Operational Hv (Depending From Gas Mixture)

Prsin	Average Input Pressure During Run
Prsout	Average Output Pressure During Run
Tempstart	Average Temperature During Run
Tempend	Average Temperature During Run

11.1.1.4 Cosmic Ray Station

Table 10: Croperation

Id	Primary Key
Idtube	Tube Id From Barcode
Cosmicrate	Total Number Of Events/Run Time In This Tube
Cosmictdcmin	Minimum Tdc Channel For Run
Cosmictdcmax	Minimum Tdc Channel For Run
Cosmicchi2	Chi-Squared Value For “Shape” Analysis
Startdt	Timestamp For Start Of Operation
Enddt	Timestamp For End Of Operation
Crrun	Pointer To Cosmic-Ray Run Table

Table 11: Crrun

Runrecord	The Run Number For A Dark Current Station Session
Crparam	The Cosmic Ray Station Parameters For This Run
Operator	First Operator For This Session
Starttime	Begin Run Date/Time
Endtime	End Run Date/Time
Stationid	Pointer To Station Id Table
Inputfile	Filename Of Input File

Table 12: Crparam

Crparam	Parameter Point
Crgasmix	Gas Mixture Percentage Of Each Gas
Crgaspurityar	Argon Purity Code
Crgasco2	Co2 Purity Code
Atmprsmin	Minimum Atmospheric Pressure During Run
Atmprsmax	Maximum Atmospheric Pressure During Run (Mbar)
Hvop	Operational Hv (Depending From Gas Mixture)
Prsinmin	Minimum Input Pressure During Run (Psi)
Prsinmax	Maximum Input Pressure During Run (Psi)
Prsoutmin	Minimum Output Pressure During Run (Psi)
Prsoutmax	Maximum Output Pressure During Run (Psi)

Tempmin	Minimum Temperature During Run (K)
Tempmax	Maximum Temperature During Run (K)
Threshold	Tdc Discriminator Threshold (Mv)
Tdcgain	Nominal Tdc Gain
Adcgain	Nominal Scanning Adc Gain

11.1.1.5 X-Ray Station

Table 13: Xrayoperation

Id	Primary Key
Idtube	Tube Id From Barcode
Tubeend	Which Tube End Is Being Measured (N Or S)
Xraydy	Measured Offset In “Y” Direction (Microns)
Xraydz	Measured Offset In “Z” Direction (Microns)
Xraydy180	Measured Offset In “ Y 180 Degrees” Direction (Microns)
Xraydz180	Measured Offset In “ Z 180 Degrees” Direction (Microns)
Xrayoff	Measured Radial Offset (Microns)
Xraydt	Timestamp For Measurement
Xrayrun	Pointer To Xray Run Number

Table 14: Xrayrun

Runrecord	The Run Number For An Xray Station Session
Xrayparam	The Xray Station Parameters For This Run
Operator	First Operator For This Session
Starttime	Begin Run Date/Time
Endtime	End Run Date/Time
Stationid	Pointer To Station Id Table
Inputfile	Filename Of Input File

Table 15: Xrayparam

Xrayparam	Parameters Set Id
Xrayhv	Hv On X-Ray Tube (V)
Xraydur	Duration Of X-Ray Pulse (S)
Xrayenergy	Energy Of X-Rays (Kev)
Xrayccdgate	Ccd Gate Time (S)

11.1.1.6 Emmi Station

Table 16: Emmioperation

Id	Primary Key
Idtube	Tube Id From Barcode
Tubeend	Which Tube End Is Being Measured (N Or S)
Emmidy	Measured Offset In “Y” (Volts)
Emmidz	Measured Offset In “Z” Direction (Volts)
Emmidy180	Measured Offset In “ Y 180 Degrees” Direction (Volts)
Emmidz180	Measured Offset In “ Z 180 Degrees” Direction (Volts)
Emmioff	Measured Radial Offset (Microns)
Emmidt	Timestamp For Measurement
Emmirun	Pointer To Emmi Run Number

Table 17: Emmirun

Runrecord	The Run Number For An Emmi Station Session
Emmiparam	The Emmi Station Parameters For This Run
Operator	First Operator For This Session
Starttime	Begin Run Date/Time
Endtime	End Run Date/Time
Stationid	Pointer To Station Id Table
Inputfile	Filename Of Input File

Table 18: Emmiparam

Emmiparam	Parameters Set Id
Ncalib	Microns/Millivolt North End
Scalib	Microns/Millivolt South End
Fudge	Fudge Factor Applied To Determine Offset Values From Measurements (Offset = Offset*Fudge)

11.1.1.7 Resistance Station

Table 19: Resistoperation

Id	Primary Key
Idtube	Tube Id From Barcode
Resistn	Resistance Measurement 1 End-Plug To Tube, North-End
Resists	Resistance Measurement 1 End-Plug To Tube, South-End
Resistdt	Date/Time Of Resistance Measurement On Tube
Resistrun	Pointer To Resistance Run Table

Table 20: Resistrun

Runrecord	The Run Number For A Resistance Station Session
Resistparam	The Resistance Station Parameters For This Run
Operator	First Operator For This Session
Starttime	Begin Run Date/Time
Endtime	End Run Date/Time
Stationid	Pointer To Station Id Table
Inputfile	Filename Of Input File

Table 21: Resistparam

Resistparam	Parameter Pointer
Meter	Ohm Meter Type

11.1.1.8 Frequency Station

Table 22: Freqoperation

Id	Primary Key
Idtube	Tube Id From Barcode
Frequency	Measured Frequency (Hz)
Freqtemp	Temperature During Measuremen (C)
Freqdt	Timestamp For Start Of Operation
Freqrun	Pointer To Frequency Run Table

Table 23: Freqrun

Runrecord	The Run Number For A Frequency Station Session
Freqparam	The Frequency Station Parameters For This Run
Operator	First Operator For This Session
Starttime	Begin Run Date/Time
Endtime	End Run Date/Time
Stationid	Pointer To Station Id Table
Inputfile	Filename Of Input File

Table 24: Freqparam

Freqparam	Parameter Pointer
Freqdrive	Driving Frequency For Wire Tension Test (Hz)
Freqpuldur	Driving Frequency Pulse Duration (S)
Freqwait	Waiting Time After Driving Pulse (S)
Freqsamp	Sampling Frequency For Tension Determination (Khz)

Above are all the tables corresponding to each measurement station. We have a “tube” table, which contains the general data for a tube.

11.1.1.9 The Structure Of The Tube Table

Table 25

Idtube	Tube Id (Bar Code First Field)
Scode	Tube Status Code
Idlayer	Layer Id – Pointer To Layer (At Assembly Time)
Tubelayerloc	Tube Location In Layer (At Assembly Time)
Location	Current Tube Location (Points To Location Table)
Wiredid	Id From “Best” Wiroperation Measurement
Wiredover	Override (By Hand) Id For Best Measurement
Leakchkid	Id From “Best” Leakchkoperation Measurement
Leakchkover	Override (By Hand) Id For Best Measurement
Crid	Id From “Best” Croperation Measurement
Crover	Override (By Hand) Id For Best Measurement
Dcid	Id From “Best” Dcoperation Measurement
Dcover	Override (By Hand) Id For Best Measurement
Xraysid	Id From “Best” Xrayoperation South Measurement
Xraysover	Override (By Hand) Id For Best Measurement
Xraynid	Id From “Best” Xrayoperation North Measurement
Xraynover	Override (By Hand) Id For Best Measurement
Resistid	Id From “Best” Resistoperation Measurement
Resistover	Override (By Hand) Id For Best Measurement
Qcsafetest	D/Nd - Safety Test (Done/Not Done)
Repeatmeasmarker	If Yes, There Are Extra Measurements In Rep_Meas
Commentmarker	If Yes, Comment(S) Exist In The Comment Table
Freqid	Id From “Best” Freqoperation Measurement
Freqover	Override (By Hand) Id For Best Measurement
Emmiid	Id From “Best” Emmioperation Measurement
Emmiover	Override (By Hand) Id For Best Measurement

Finally, our group needs a summary table showing the status of the current production. We list this table here; it is used for our comparison between access and objectivity (see the part of oodbms study).

11.1.1.10 Summary Table

Table 26

Idtube	Tube Id
Scode	Status Code For The Tube
Darkcurave	Average Dark Current
Dccnt	Measurement Count In Dc Station
Leakdt	Timestamp For Leak Check
Leakrate	Gas Leak Rate
Lccnt	Measurement Count In Leakchk Station
Resistdt	Timestamp For Resistance Measurement
Resistn	Resistance Measurement 1 End-Plug To Tube, North-End
Resists	Resistance Measurement 1 End-Plug To Tube, South-End
Rcnt	Measurement Count In Resist Station
Wirstartdt	Timestamp For Wiring Measurement
Tubelencode	Barcode Of Tube Length (Mm)
Tubelen	Measured Tube Length (Mm)
Wirfreq	Measured Wire Frequency (Hz) Just After Wiring
Tension	Inferred Wire Tension (G) Just After Wiring
Wcnt	Measurement Count At Wir Station
Freq	Frequency (Hz) During Remeasurement
Freqdt	Timestamp For Frequency Remeasurement
Freqtemp	Temperature (C) During Frequency Remeasurement
Fcnt	Number Of Frequency Remeasurements
Xraydts	Timestamp For Xray Measurement, South-End
Xrayoffs	Measured Radial Offset (Microns), South-End
Cnts	Measurement Count At Xray Station, South-End
Xraydys	Y Offset On Tube South End (Microns)
Xraydzs	Z Offset On Tube South End (Microns)
Xraydtn	Timestamp For Xray Measurement, North-End
Xrayoffn	Measured Radial Offset (Microns), North-End
Cntn	Measurement Count At Xray Station, North-End
Xraydyn	Y Offset On Tube North End (Microns)
Xraydzn	Z Offset On Tube North End (Microns)

11.2 APPENDIX 2

11.2.1 EXAMPLES OF STATION TEXT FILES

11.2.1.1 Wiring Station

```
Operator: Garth Heutel and Jim Degenhardt
Starting at 15:41:18 8-9-1999
12345 2.97 350.0 0.0 0.0 24.0
10100 11 138981 15:42:5 3332.0
10100 0.0 15:47:18 11.22 2
10100 45.166016 0.0 15:50:39
10100 3355.17 15:50:55 359.910095
10100 1 15:51:0
10082 11 138981 15:53:36 3332.0
10082 0.0 15:53:39 11.22 2
10082 60.058594 0.0 15:53:43
10082 3437.98 15:53:54 668.439021
10082 2 15:55:0 Small scratch on inside of tube, north end. Did not string.
10097 11 138981 15:58:18 3332.0
10097 0.0 15:58:20 11.22 2
10097 24.536133 0.0 15:58:23
10097 3437.99 15:58:36 111.564558
10097 2 16:2:0 Small scratch on inside of tube, south end. Did not string.
10099 11 138981 16:2:11 3332.0
10099 0.0 16:12:21 11.22 2
10099 45.043945 0.0 16:14:30
10099 3355.24 16:14:49 357.982318
10099 1 16:15:0
10098 11 138981 16:17:18 3332.0
10098 0.0 16:21:32 11.22 2
10098 45.043945 0.0 16:23:29
10098 3355.18 16:23:46 357.969415
10098 1 16:24:0
10087 11 138981 16:26:10 3332.0
10087 0.0 16:30:3 11.22 2
10087 45.166016 0.0 16:32:32
10087 3355.15 16:32:49 359.905771
10087 1 16:33:0
10096 11 138981 16:35:0 3332.0
10096 0.0 16:38:39 11.22 2
10096 45.288086 0.0 16:40:51
10096 3355.16 16:41:10 361.85601
10096 1 16:41:0
10095 11 138981 16:43:35 3332.0
10095 0.0 16:49:5 11.22 2
10095 59.936523 0.0 16:49:9
10095 3465.56 16:49:19 676.530529
10095 2 16:50:0 South end not crimped.
Ended at: 16:50:27 8-9-1999
```

11.2.1.2 Leak Check Station

```
Operator: Qiao Li and NONE
Starting at 11:10:31 8-17-1999
A0B0C00004D000009E000188F355G053H0501100I04L11000100000051N292J1005K090O294P2
4Q25R19S0180T0172U0079V1100W2308X21Y0010Z0001[0010\0010]140056250^0007_0041`1
00a0010b000384c0010d0001e239f255g054062h51j0020k00l1
10315 3.1e-005 1.7e-007 0.00012 0 11:41:5
10312 8.0e-006 0.0 0.0 0 11:51:48
Ended at 13:11:59 8-17-1999
```

11.2.1.3 DarkCurrent (CosmicRay not yet running)

```
Operator: levin
 Start time: 1999 9 3 13 38
 Parameters: Argon-C02 93 7 3200 44.9 44.7 -273.392706 -273.392706
 10417 1 17.924 27.994 110.413
 10418 2 55.467 37.698 124.939
 10415 3 103.634 24.839 124.939
 10423 4 -0.237 0.136 0.183
 10427 5 -0.626 0.124 -0.244
 10428 6 -0.508 0.127 -0.122
 10451 7 1.169 5.269 29.907
 10453 8 -0.480 0.128 -0.122
 10450 9 -0.344 0.123 0.000
 10307 10 -0.188 0.159 0.366
 10324 11 -0.472 0.118 -0.061
 10335 12 2.230 0.553 3.906
 10328 13 -0.428 0.116 -0.122
 10332 14 -0.306 0.131 0.610
 10302 15 109.672 17.225 124.939
 10444 16 -0.376 0.127 0.793
 10279 17 -0.834 0.130 -0.427
 10289 18 0.592 0.128 0.977
 10304 19 0.681 0.124 1.099
 10308 20 1.332 1.907 15.381
 10298 21 0.471 0.129 0.854
 10326 22 0.380 0.119 0.732
 10329 23 0.067 0.125 0.427
 10281 24 -0.441 0.120 -0.061
 10323 25 -0.301 0.124 0.061
 10321 26 -0.202 0.119 0.183
 10327 27 -0.228 0.125 0.183
 10305 28 0.023 0.127 0.427
 10310 29 -0.458 0.525 13.855
 10300 30 -0.575 0.127 -0.061
 10295 31 -0.520 0.133 -0.061
 10445 32 2.859 6.704 28.259
 10313 33 1.000 0.000 1.000
 10283 34 0.325 0.149 0.793
```

```
 10315 35 0.122 0.138 0.549
 10282 36 10.896 14.770 56.458
 10309 37 0.102 0.162 0.610
 10317 38 -0.093 0.137 0.305
 10320 39 0.009 0.132 0.366
 10435 40 0.120 0.137 0.549
 10410 41 4.847 10.642 47.607
 10422 42 4.448 6.694 38.086
 10433 43 0.287 0.162 2.075
 10457 44 0.898 0.297 3.235
 10452 45 0.081 0.137 0.488
 10455 46 0.696 0.145 1.160
 10436 47 0.098 0.151 0.549
 10454 48 0.700 0.278 1.709
 End time 1999 9 6 12 19
```

11.2.1.4 Xray Measurement Station

```
Operator: Garth Heutel
Started at 17:11:26 on 08-05-1999
Run parameters:
Xray Voltage = 100kV, Xray Exposure Time 1s
CCD exposure time 1.50s, Xray Energy 0eV
9861  n  -13.6 -16.3 -17.4 2.5   9.6   17:16:36
9861  s  -16.3 7.3   0.9   -23.9 17.8  17:20:18
9828  n  -5.9  -12.4 11.5  2.2   11.4  17:28:07
9828  s  -14.9 4.4   -1.7  -4.0  7.9   17:32:45
9818  n  -0.2  -17.3 -8.9  1.7   10.5  17:39:54
9818  s  -5.7  9.0   -13.8 -19.6 14.9  17:44:22
9814  n  5.5   -9.8  -26.5 -22.5 17.2  17:51:55
9814  s  -16.4 -8.9  0.9   -14.7 9.1   17:56:29
9823  n  -8.1  -3.0  -8.5  -4.6  0.8   18:03:58
9823  s  6.1   2.6   -6.8  -25.5 15.4  18:08:25
Ended at 18:08:27 on 08-05-1999
```

11.2.1.5 EMMI Measurement Station

```
Operator: qiao and none
Start: 19991019 124701
n_calib 2.8010 s_calib 2.6123 fudge 1.0650
10421 n 0.723 0.742 0.742 0.725 4.93 124701
10421 s 3.921 3.984 4.080 4.033 33.84 124701
10421 n 0.706 0.686 0.674 0.652 8.87 124942
10421 s 3.948 4.041 4.136 4.087 39.47 124942
10421 n 0.647 0.679 0.662 0.652 5.82 125343
10421 s 3.999 4.038 4.136 4.092 29.94 125343
10421 n 0.637 0.667 0.657 0.647 5.25 125916
10421 s 4.001 4.048 4.136 4.094 28.96 125916
10421 n 0.637 0.664 0.654 0.630 7.26 130133
```

```
10421 s 4.006 4.045 4.153 4.097 31.64 130133
End: 19991019 130151
```

11.2.1.6 Resistance Measurement Station

```
Operator: diehl
Started at 14:15:03 on 08-24-1999
10066 0.02  0.04  14:15:26
10101 0.03  0.03  14:15:41
10075 0.03  0.02  14:16:02
10065 0.02  0.02  14:16:14
10076 0.03  0.03  14:16:28
10050 0.01  0.03  14:16:40
10064 0.03  0.04  14:16:51
10089 0.04  0.03  14:17:08
10087 0.01  0.02  14:17:21
10081 0.02  0.58  14:17:53
Ended at 14:17:57 on 08-24-1999
```

11.2.1.7 Frequency Measurement

```
Operator: Jim Degenhardt and Jim Degenhardt
Starting at 8:42:51 8-24-1999
42.0 1.0 0.0625 4.0
9697 47.851563 13:43:8 19.904865
9708 47.851563 13:44:31 19.852974
9694 47.851563 13:45:43 19.905625
9952 45.898438 13:49:8 19.905715
10286 42.96875 13:51:14 19.853645
10325 42.96875 13:53:1 19.905535
10314 42.96875 13:55:23 19.957025
10322 42.96875 13:58:16 19.800994
10312 42.96875 14:1:26 19.801575
10285 42.96875 14:4:41 19.801575
10306 42.96875 14:7:58 19.385152
10334 42.96875 14:9:1 19.749014
10293 42.96875 14:10:24 19.853555
10447 42.480469 14:12:5 20.270115
10318 43.457031 14:15:21 19.800994
10288 42.96875 14:16:39 19.853555
10287 42.96875 14:18:10 19.800994
10301 42.96875 14:19:55 19.800994
10060 44.921875 14:24:57 19.749014
10188 43.945313 14:26:40 19.800994
Ended at 14:27:38 8-24-1999
```

11.3 APPENDIX 3

11.3.1 EXAMPLE OF A PERL SCRIPT USED IN ACQUIRING INPUT DATA

Below is one of many Perl scripts used to import data from the Labwindows generated text files for each station. This example shows the Leak Check station acquisition as a specific example. Each station has analogous code. The main entry points are named <station>_convert, where <station> represents the name of each station. This routine opens a database connection, verifies the input file exists and read in the data using a Read<station>Data subroutine which understands the station's text file format. After validating the data, the process_<station> subroutine is called to actually create the database record(s) in ACCESS.

```
#
# Converts LABWINDOWS leak text file into formatted fields for ACCESS
#
use OLE;
use Win32::ADO;

do "db_subs.pl";

sub leakchk_convert {

 $ODBC = shift;
 $filename = shift;

 $debug = 0;
 if ($filename =~ /leak_(\d\d\d\d)_(\d{1,2})_(\d{1,2})_(\d{1,2})_(\d{1,2})\.dat/) {
    $mon = $2;
    $day = $3;
    $year = $1;
    $time = "$4:$5";
    $name = "leak\_$1\_$2\_$3\_$4\_$5";
  $file = $name."\.dat";
#  print "\t file ($file) is from year=$year, mon=$mon, day=$day, time=$time\n";
#
# We now have a log file to process into ACCESS..
#

    foreach $key (keys %LeakChkParam) {
     delete $LeakChkParam{$key};
    }
    foreach $key (keys %LeakChkRun) {
     delete $LeakChkRun{$key};
    }
  @ldata = ();
    $istat = ReadLeakChkData($filename,\%LeakChkParam,\%LeakChkRun,\@ldata);

    $cnt = $#ldata+1; # Get "real" count of records
    if ($cnt <= 0) {
     $MAIN::ecnt++;
     $MAIN::errors[$MAIN::ecnt] = " No data in file $file\n";
     return 0;
    } else {
     $MAIN::cnt = $cnt;
    }

    if ($debug == 1) {
     print " Read in $filename, istat=$istat\n";
     foreach $key (keys %LeakChkParam) {
            print " LeakChkParam key $key = $LeakChkParam{$key}\n";
     }
     foreach $key (keys %LeakChkRun) {
            print " LeakChkRun key $key = $LeakChkRun{$key}\n";
     }
     print " Found $cnt records in file..\n";
```

```
    for ($i=0; $i<$cnt; $i++) {
            print "ldata[$i][0-5] = ";
            for ($j=0; $j<6; $j++) {
             print "$ldata[$i][$j] ";
            }
            print "\n";
    }
  next;
 }

#   print " Found $cnt leakchk measurements..\n";

# Open database..
    OpenDSN($ODBC);

    $table="LeakChkParam"; # Table to check..exists with these params?
    $key = "LeakChkParam";
    $LeakChkParam = CheckRecord($table,$key,\%LeakChkParam);
    if ($debug == 1) {
     foreach $key (keys %LeakChkParam) {
            print " LeakChkParam key $key = $LeakChkParam{$key}\n";
     }
     print " LeakChkParam = |$LeakChkParam|\n";
    }
    if ($LeakChkParam eq "") {
# Need a new record..get current largest $key value and increment by 1
     $table = "LeakChkParam";
     $LeakChkParam{$key} = GetFieldMax($table,$key)+1;
     if (! ($istat=CreateRecord($table,\%LeakChkParam)) ) {
            $MAIN::ecnt++;
            $MAIN::errors[$MAIN::ecnt]=" Error in CreateRecord LeakChkParam on file $file, stat=$istat";
            return 0;
     }
     $LeakChkParam = $LeakChkParam{$key};
    }

# Create new LeakChkRun record? This might be a re-scan of this text file..check!
  $LeakChkRun{"LeakChkParam"} = $LeakChkParam;
    $LeakChkRun{"StationID"} = GetStationID("LeakChk Station");
    $LeakChkRun{"InputFile"} = $file;
    if ($debug == 1) {
     foreach $key (keys %LeakChkRun) {
            print " LeakChkRun key $key = $LeakChkRun{$key}\n";
     }
  }
    $table="LeakChkRun";
    $key = "RunRecord";
  $LeakChkRun = CheckRecord($table,$key,\%LeakChkRun);
    if ($LeakChkRun ne "") {
# We have processed this run already!!!
   $MAIN::wcnt++;
     $MAIN::warnings[$MAIN::wcnt] = "Already processed file $file as xray run $XrayRun!\n";
     return 1;
    } else {
# Create new run record
     $table = "LeakChkRun";
     $LeakChkRun{$key} = GetFieldMax($table,$key)+1;
     $LeakChkRun = $LeakChkRun{$key};
     if (! (my $istat=CreateRecord($table,\%LeakChkRun)) ) {
      $MAIN::ecnt++;
      $MAIN::errors[$MAIN::ecnt]=" Error in CreateRecord LeakChkRun on file $file, stat=$istat";
      return 0;
     }
  }
# Get Tube table for LeakChk update..
    $table="LeakChkOperation";

# Loop over all measurements in this run
```

```
  for ($i=0;$i<$cnt;$i++) {

#================================================================================================
# Create Tube table first..
# Init %Tube..
   foreach $key (keys %Tube) {
    delete $Tube{$key};
   }
      $table="Tube";
      $key = "IdTube";
      $Tube{IdTube} = $Idata[$i][0];
      if ($Tube{IdTube} <= 0) {
       $MAIN::wcnt++;
       $MAIN::warnings[$MAIN::wcnt] = " Bad tube id $Tube{IdTube}..skipping..\n";
       next;
      }
# Have we done this tube already?
      $IdTube2 = CheckRecord($table,$key,\%Tube);
#    print " IdTube2 = $IdTube2, IdTube = $Tube{IdTube}\n";

      $Site="Michigan";
      $Room="PRL_2208";
      $RackBox="Rack_2";
      $ShelfLay=1;
      $Tube{Location} = GetLocation($Site,$Room,$RackBox,$ShelfLay);

# Check if we have this Tube record..
      if ($IdTube2 == $Tube{IdTube}) {
#      print " Tube $IdTube2 already entered in Tube table..updating values\n";
       $table="Tube";
       $key = "IdTube";
       $keyvalue = $IdTube2;
       foreach $key2 (keys %Tube) {
              $field = "$key2";
              $fvalue = $Tube{$key2};
              $istat = UpdateField($table,$key,$keyvalue,$field,$fvalue);
              if ($istat != 1) {
               $MAIN::ecnt++;
               $MAIN::errors[$MAIN::ecnt] = " Error in UpdateField, istat=$istat\n\t
(table,key,keyvalue,field,fvalue)=$table,$key,$keyvalue,$field,$fvalue\n";
              }
       }
       $istat = 1;
      } else {
       $table="Tube";
       $key = "IdTube";
       $MAIN::wcnt++;
       $MAIN::warnings[$MAIN::wcnt] = "Creating Tube, IdTube=$Tube{IdTube}, IdTube2= $IdTube2..\n";
       $istat = CreateRecord($table,\%Tube);
      }
#================================================================================================

# We have found this tube..process the current measurement..
      for ($j=0;$j<7;$j++) {
      $hLeakChk[$j] = $Idata[$i][$j];
      }
      if ($LeakChkRun < 1) {
       $MAIN::ecnt++;
       $MAIN::errors[$MAIN::ecnt] = "Bad LeakChkRun = |$LeakChkRun| found!\n";
       return 0;
      }
      $hLeakChk[7] = $LeakChkRun;
      $istat = process_LeakChk(\@hLeakChk);
      if ($istat != 1) {
       $MAIN::ecnt++;
       $MAIN::errors[$MAIN::ecnt] = "Error processing entry $i, istat=$istat";
      }
  }
```

```
 CloseDSN();
} else {
    $MAIN::ecnt++;
    $MAIN::errors[$MAIN::ecnt] = "File $filname has bad name syntax\n";
    return 0;
}
return 1;
}

sub process_LeakChk {
#
# This subroutine processes an leak measurement.
#
# We must also create any comment fields which exist
#
# $istat = process_LeakChk(\@values); # where values are LeakChkOperation DB values
#
 local (*values) = @_;

# Must "map" from measured key names to DB names..
 $DBnames[0] = "IdTube";
 $DBnames[1] = "LeakRate";
 $DBnames[2] = "LeakRateN";
 $DBnames[3] = "LeakRateS";
 $DBnames[4] = "LeakBody";
 $DBnames[5] = "LeakDT";
 $DBnames[6] = "LeakRun";

# Swap "comment" with LeakRun
 my $tmp = $values[6];
 $values[6] = $values[7];
 $values[7] = $tmp; # now this is comment
 chomp($values[7]);

# Initialize Fields..
 foreach $key (keys Fields) {
    delete $Fields{$key};
 }
# Load Fields
 for (my $i=0;$i<7;$i++) {
    if ($values[$i]) {
     $Fields{$DBnames[$i]} = $values[$i];
    }
 }

# Exists?
 my $table = "LeakChkOperation";
 my $key = "ID";
 if (my $ID=CheckRecord($table,$key,\%Fields)) {
    $MAIN::ecnt++;
    $MAIN::errors[$MAIN::ecnt] = " process_LeakChk: Entry already exists with ID=$ID!\n";
    return 0;
 } else {
# Put it in table!
  if (my $istat=CreateRecord($table,\%Fields)) {
# Handle comment if it exists..
  if ($values[7]) {
      my $IdTube = $values[0];
      my $source = GetStationID("LeakChk Station");
      my $comment = $values[7];
      my $timestamp = $values[5];
      if (! ($istat=process_comment($IdTube,$source,$comment,$timestamp))) {
       $MAIN::ecnt++;
       $MAIN::errors[$MAIN::ecnt] = " process_LeakChk: Error entering comment for tube $IdTube, stat=$istat\n";
       return 0;
      }
  }
  } else {
```

```
    $MAIN::ecnt++;
    $MAIN::errors[$MAIN::ecnt] = "Error creating LeakChkOperation record, stat=$istat\n";
    return 0;
 }
 }
 return 1;
}

sub ReadLeakChkData {
#
#   $istat = ReadLeakChkData($filename,\%LeakChkParam,\%LeakChkRun,\@ldata);
#
#
 local ($filename, *LeakChkParam, *LeakChkRun, *ldata) = @_;

# Check for existence of file
 if (! -e "$filename") {
    $MAIN::ecnt++;
    $MAIN::errors[$MAIN::ecnt] = " File $filename not found!\n";
    return 0;
 }

 open(IN,"$filename");
 $np = 0;
 $cnt = -1;
 $nstart = 0;
 my $start_time = 0;
 my $start_date = "";
 my $end_time = 0;
 my $end_date = "";
 $nend = 0;

# Load up default values
 $LeakChkRun{Operator} = "Li Chao";
 $filename =~ /leak_\d\d(\d\d)_(\d{1,2})_(\d{1,2})_(\d{1,2})_(\d{1,2})\.dat/;
 $DefDT = "$2/$3/$1 $4\:$5";
 $LeakChkParam{MachineSettings} = "";
# print " ============================ $filename ====================\n";
 while ($line=<IN>) {
    $np++;
#  print " ----$np------ \n";
#   print "$line\n";
# Start run record?
    if ($line =~ /^(?:Operator:)?\s*([^\d\s]+)\s+(\S+)?/ && $np==1) {
     $nstart++;
#    print " Found start run record!\n $line\n";
     ($nstart > 1) and $error[$ecnt++]="Extra start found in file $filename";
#   print " Operator #2 = |$2|\n";
     $LeakChkRun{Operator} = $1." ".$2;
# Trim any blanks at the end

    }
    if ($line =~ /([\d\:]+)/ && $np==2) {
#    print " Found start time-stamp record!\n $line\n";
     $start_time=$1;
  }
  if ($line =~ /([\d]+-[\d-]+)/ && $np==2) {
     $start_date=$1;
#    print " Found start date record |$start_date|\n";
  }
  if ($start_date ne "" && $start_time != 0 && !$LeakChkRun{StartTime}) {
     $start_date =~ /(\d{1,2})-(\d{1,2})-\d\d(\d\d)/;
     $start_date = "$1/$2/$3";
     $LeakChkRun{StartTime}="$start_date $start_time";
#    print " StartTime=$LeakChkRun{StartTime}\n";
    }
    if ($line =~ /^(\w{100,})/) {
#    print " Found leakchk parameters\n$line";
```

```
      $LeakChkParam{MachineSettings} = $1;
     }
     if ($line =~ /^(?:End)?.*([\d]+\:[\d\:]+)/ && !$LeakChkRun{EndTime}) {
      if ($line =~ /([\d]+\:[\d\:]+)/) {
       $end_time=$1;
      }
      if ($line =~ /([\d]+-[\d-]+)/) {
            $end_date=$1;
      }
      if ($end_time != 0 && $end_date ne "") {
#     print " Found end time-stamp record!\n $line\n";
       $end_date =~ /(\d{1,2})-(\d{1,2})-\d\d(\d\d)/;
       $LeakChkRun{EndTime}="$1/$2/$3 $end_time";
#      print " EndTime=$LeakChkRun{EndTime}\n";
      }
     }

#
# $tube $resN $resS $resDT [$comment]
# Computer format
# $tube $leakrate $leakrateN leakrateS $bodyPass $timestamp $comment
#
     if ($line =~ /^(\d{1,6})(?!(\:|\d|-))/) { # Scan for tube ID..
      $cnt++;
#     print " LINE: Matched $1\n";
      ($ldata[$cnt][0],$ldata[$cnt][1],$rest) = split(/\s+/,$line,3);
      if ($rest) {
# Process computer generated info..
            if ($rest !~ /^\s+$/) {
             ($ldata[$cnt][2],$ldata[$cnt][3],$ldata[$cnt][4],$ldata[$cnt][5],$ldata[$cnt][6]) = split(/\s+/,$rest,5);
             $ldata[$cnt][5] = "$start_date $ldata[$cnt][5]";
            }
            chomp($ldata[$cnt][6]);
      } else {
            chomp($ldata[$cnt][1]);
      }
     }
 }
 $cnt++; # Get real count
# print " Found $cnt records in file\n";
# print " File $file had $np lines and $error{$file} errors\n";
 if ($LeakChkRun{StartTime} eq "") {
     $LeakChkRun{StartTime} = $DefDT;
 }
 close(IN);
 return 1;
}
```

A complete list of the Perl scripts used in our production, as well as the capability to download the files, is available at: http://atwww.physics.lsa.umich.edu/aspdb/perl_scripts.htm.

11.4.1 EXAMPLE OF VBA PROGRAMMING; FREQUENCY DISTRIBUTION APPLICATION

At the early stage of our development, we used Microsoft Visual Basic to access the data in the ACCESS database, and used it to generate report and frequency distribution histograms of the data. The VB program first brings out an interface:

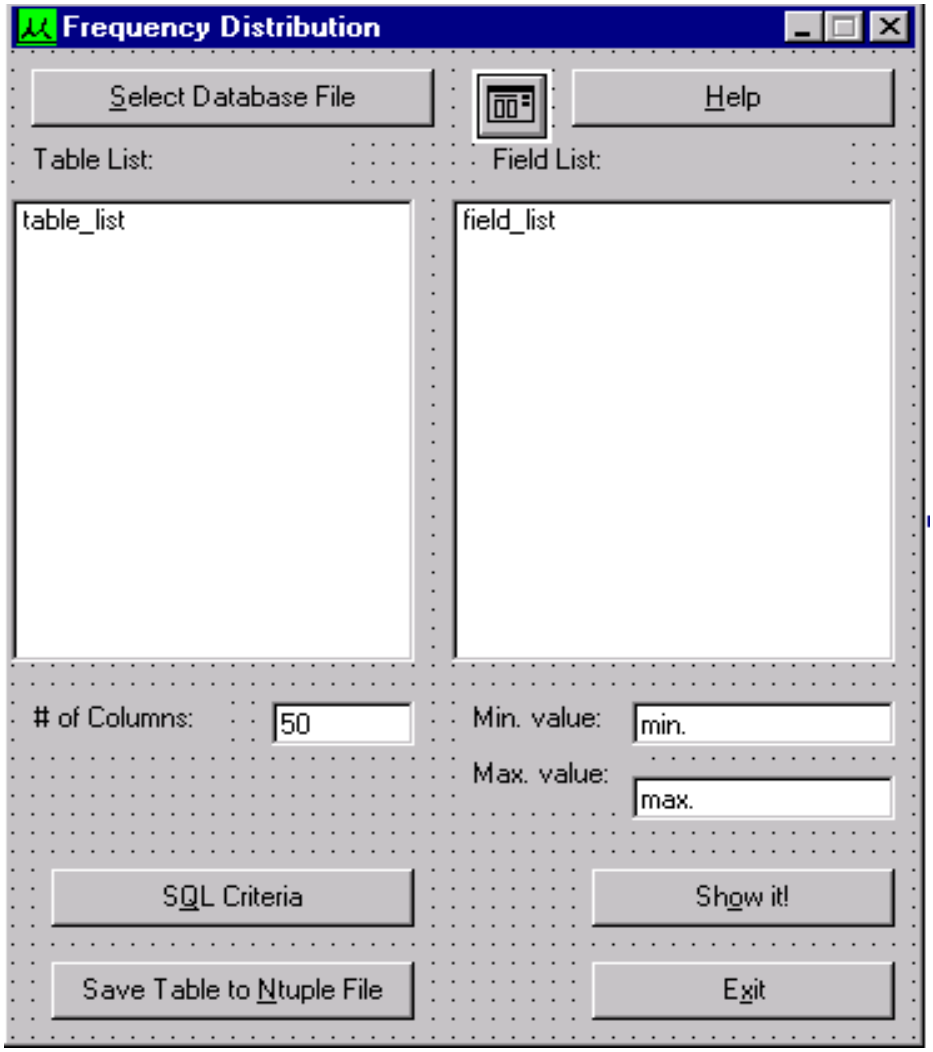


Figure 9 Data entry interface from Frequency Distribution program

As the image shows, you can set some properties of the output histogram, like the number of columns, and min/max value of the cuts. You can write SQL statements to select the data, and save the output data to an Ntuple file, which can be read and analyzed by PAW. Here is a sample histogram:

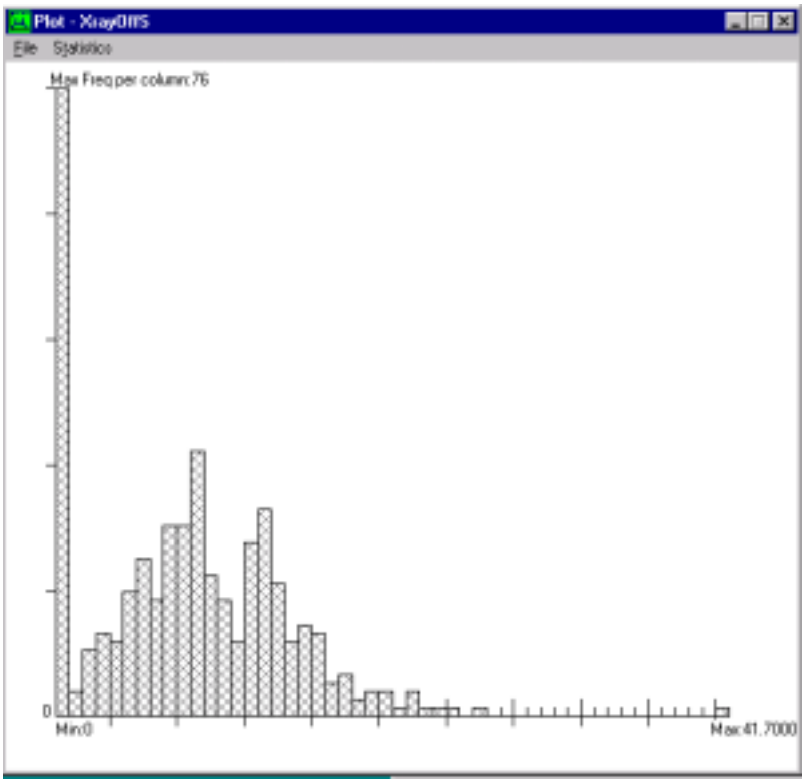


Figure 10 A frequency distribution plot showing Xray Offset (microns) for all north tube ends

Visual Basic provides many ways to access the data inside an ACCESS database. We used Microsoft DAO (Data Access Object) models to do that. The following snippet shows how to access the data:

```
Dim dbMydb As Database
Dim recMyRec As Recordset
Dim sDBPath As String
Dim sSQL As String
Dim sField As String
Dim sfileName as String
…

‘ Set the SQL statement for selecting records
sSQL = “SELECT * FROM Tubes WHERE IdTube > 200”
Set dbMydb = OpenDatabase(sDBPath)     ‘ open the database
‘ A recordset object represents a record in the chosen table.
Set recMyRec = dbMydb.OpenRecordset(sSQL, dbOpenDynaset)

For I = 0 to recMyRec.Fields.Count
   Print recMyRec.Fields(i).name
            ‘ print out the names for each field in this table.
Next

‘ The following code reads data from a test file, and put it into the
‘ Database.
Open sFile For Input As #1
For I = 0 To recMyRec.RecordCount
            While (Not char Like Chr(13))
          Entry = Entry + input(1, #1)
            Wend
   RecMyRec.Fields(“Scode”) = Entry
Next
```

Though Visual Basic is very flexible, and highly integrated with ACCESS, but for the reason stated before, we finally switched to PAW implementation.

11.5.1 AN EXAMPLE OF A PAW ANALYSIS SESSION.

We show an example of using the downloaded Summary_Table.rz file inside PAW using the Windows NT version. We first “double-click” on PawNT.exe in our \cern\bin directory and hit enter at the workstation type request:

Figure 11 Example of PAW command window using WinNT

The “set 2buf 11” is to implement “backing-store” which saves any graphics in the HIGZ window if they become covered by other windows. We then open the ntuple file and print the contents:

‘PAW> H/FILE 25 Summary_Table.rz’,
‘PAW> NT/PRINT 1’

See the plot on the next page for the NTUPLE contents. We now prepare to plot some data by executing a local PAW setup file and then create a plot.

‘PAW> EXEC Setup.kumac’
‘PAW> NT/PLOT 1.TubeLen-TubeLenc-23.15%WirStart abs(TubeLen-TubeLenc-23.15)<5’

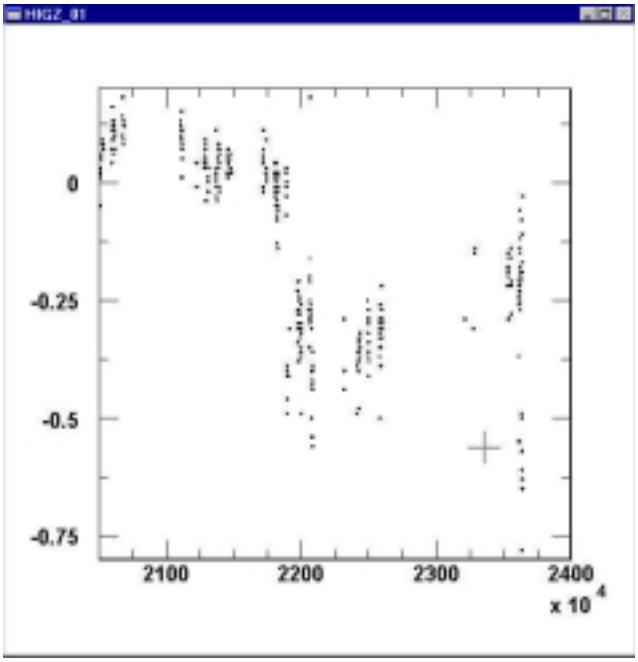
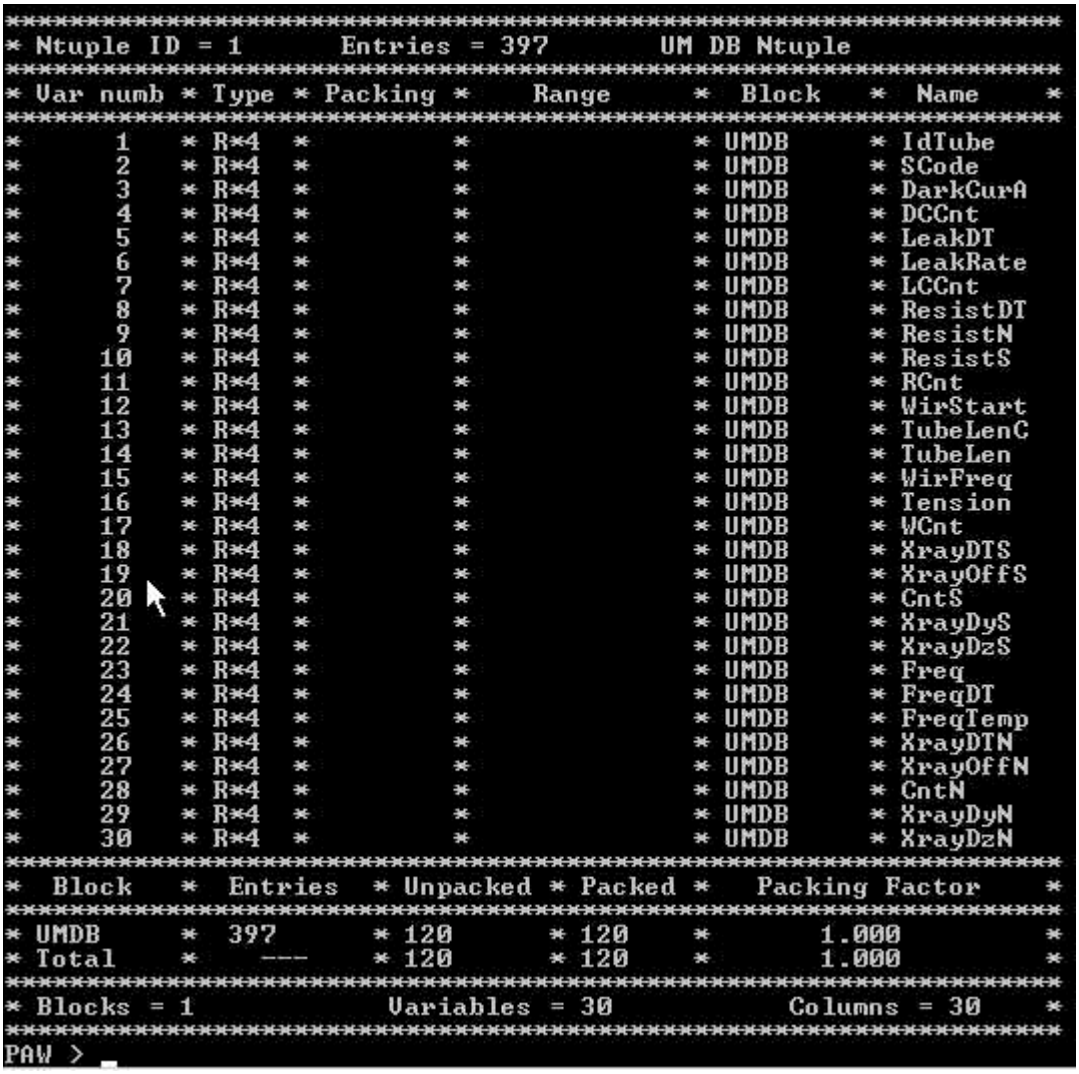


Figure 12 NTUPLE plot of measured length minus calculated length (mm) versus time (sec)



```
*****************************************************************
* Ntuple ID = 1      Entries = 397       UM DB Ntuple
*****************************************************************
* Var numb * Type * Packing *    Range     *  Block   *  Name    *
*****************************************************************
*      1   * R*4  *          *              * UMDB    * IdTube
*      2   * R*4  *          *              * UMDB    * SCode
*      3   * R*4  *          *              * UMDB    * DarkCurA
*      4   * R*4  *          *              * UMDB    * DCCnt
*      5   * R*4  *          *              * UMDB    * LeakDT
*      6   * R*4  *          *              * UMDB    * LeakRate
*      7   * R*4  *          *              * UMDB    * LCCnt
*      8   * R*4  *          *              * UMDB    * ResistDT
*      9   * R*4  *          *              * UMDB    * ResistN
*     10   * R*4  *          *              * UMDB    * ResistS
*     11   * R*4  *          *              * UMDB    * RCnt
*     12   * R*4  *          *              * UMDB    * WirStart
*     13   * R*4  *          *              * UMDB    * TubeLenC
*     14   * R*4  *          *              * UMDB    * TubeLen
*     15   * R*4  *          *              * UMDB    * WirFreq
*     16   * R*4  *          *              * UMDB    * Tension
*     17   * R*4  *          *              * UMDB    * WCnt
*     18   * R*4  *          *              * UMDB    * XrayDTS
*     19   * R*4  *          *              * UMDB    * XrayOffS
*     20   * R*4  *          *              * UMDB    * CntS
*     21   * R*4  *          *              * UMDB    * XrayDyS
*     22   * R*4  *          *              * UMDB    * XrayDzS
*     23   * R*4  *          *              * UMDB    * Freq
*     24   * R*4  *          *              * UMDB    * FreqDT
*     25   * R*4  *          *              * UMDB    * FreqTemp
*     26   * R*4  *          *              * UMDB    * XrayDTN
*     27   * R*4  *          *              * UMDB    * XrayOffN
*     28   * R*4  *          *              * UMDB    * CntN
*     29   * R*4  *          *              * UMDB    * XrayDyN
*     30   * R*4  *          *              * UMDB    * XrayDzN
*****************************************************************
*  Block   *  Entries  * Unpacked * Packed *   Packing Factor    *
*****************************************************************
* UMDB     *  397      * 120      * 120    *       1.000         *
* Total    *    ---    * 120      * 120    *       1.000         *
*****************************************************************
* Blocks = 1           Variables = 30          Columns = 30     *
*****************************************************************
PAW > _
```

Figure 13 PAW printout of Summary_Table NTUPLE showing variables

This example shows how to plot a calculated variable versus time (delta-length versus seconds since January 1, 1999 00:00). In this example, you can see a systematic shift in the offsets for different sets of tube lengths. This problem is understood as a shift in the absolute calibration scale between different batch of tube runs.

11.6 APPENDIX 6

11.6.1 CODE USED TO CREATE NTUPLES FROM ACCESS TABLES

```
#
# db_to_nt.pl – Perl script for creating an NTUPLE from ACCESS table
#
# Creates an SQL query returning a recordset which is "written" to a
# text/data file
#
# Must also convert all time fields to seconds since EPOCH (see below)
#
# Format defined as follows (see Make_ntuple.for)
#
# Line 1: N -- Number of variables in table
# Line 2-(N+1): <varnames> (Trimmed to 8 characters)
# Line N+2-<end>: <data> ! All fields for each record (record / line)
#

use OLE;
use Win32::ADO;
use Time::Local;

$table="Summary_Table";
foreach $arg (@ARGV) {
 print " Passed argument $arg..setting table..\n";
 $table = $arg;
}
$fname = "$table.txt";

do "db_subs.pl";

$sec=0;
$min=0;
$hour=0;
$mday=1;
$mon=0;
$year=99;
$epoch = timelocal($sec,$min,$hours,$mday,$mon,$year);

$istat = OpenDSN("UM_DB");
if (!$conn) {
 print " No DSN(datasource) open!\n";
 return;
}
@fields = GetFields($table);
open(OUT,">$fname");
$nflds=$#fields+1;
#
# Build format field for printf below
#
$format="";

print OUT "$nflds\n";
print "Found $nflds fields from $table\n";
$i=0;
foreach $f (@fields) {
 $type[$i] = "R";
 if ($f =~ /DT/) {
    $index[$ndates++] = $i;
    $type[$i] = "I";
 }
 $i++;
 print " Field $i is $f\n";
 print OUT "$f\n";
# Augment format
 $format .= "%15.7e ";
}
$format .= "\n";

$sql = "SELECT * FROM $table";
$rs = $conn->Execute($sql);
if (!$rs) {
 $Errors = $conn->Errors();
 print "Errors in Execute on DSN=UM_DB:\n";
 foreach $error (keys %$Errors) {
```

```
   print $error->{Description},"\n";
 }
 exit 0;
}
while (!$rs->EOF()) {
 for ($i=0;$i<$nflds;$i++) {
    $vals[$i] = $rs->Fields($i)->value;
    for ($j=0;$j<$ndates;$j++) {
     if ($i==$index[$j]) {
# We have to "fix" this date field to seconds since epoch
        if ($vals[$i] =~
/(\d{1,2})\/(\d{1,2})\/(\d\d)\s+(\d{1,2})\:(\d{1,2})\:(\d{1,2})\s+(.M)/i) {
#        print " Converting $fields[$i] with value $vals[$i]\n";
         $sec = $6;
         $min = $5;
         $hours = $4;
         $mday = $2;
         $mon = $1;
         $year = $3;
         if ($7 =~ /PM/i && $hours<12) {
            $hours+=12;
         }
#        print " sec=$sec min=$min hours=$hours mday=$mday mon=$mon
year=$year\n";
         $vals[$i] = timelocal($sec,$min,$hours,$mday,$mon,$year)-$epoch;
         } else {
#      print " Bad field?: Converting $fields[$i] with value $vals[$i]\n";
       }


     }
    }
# Put in -1.0 for missing values
    if ($vals[$i] eq '') {
#    print " Found null field $fields[$i] value $i=|$vals[$i]|\n";
     $vals[$i] = -1.0;
    }
# Insure all $vals are REAL
    if ($vals[$i] =~ /\./) {
# OK
    } else {
     $vals[$i] = $vals[$i]."\.0";
    }
 }
 printf OUT "${format}", @vals;
 $rs->MoveNext;
}
$rs->Close();
close(OUT);
#
# Now run Make_ntuple.exe to create .rz file..
#
$stat = system("umdb_ntuple.exe $fname");
if ($stat != 0) {
 print " Return error on umdb_ntuple.exe, stat=$stat\n";
} else {
#
# Now we should have a valid ntuple RZ file
#
 $rname = $fname;
 $rname =~ s/\.txt/\.rz/;
 print " Output in $rname\n";
}
exit;
```

Below is the FORTRAN code responsible for creating an NTUPLE from the text file generated above:

```
      PROGRAM MAKE_NTUPLE

      USE DFPORT ! Allows use of LNBLNK
      USE DFLIB ! Allows use of GETARG/NARGS
C
C This routine creates a CWN from an input text
C file <infile>.txt. The output file is named <infile>.rz
C
```

```
C The input file is generated by a Perl script (db_to_nt.pl)
C which converts an ACCESS table to a text file with the
C following format. (The db_to_nt.pl also runs this program)
C
C Format of input text file is as follows:
C Line 1: N     ! where N is the number of variables (fields)
C Line 2-(N+1): <variables> ! Names of fields (trimmed to 8 characters)
C Line N+2:  <data>  ! N fields of data (1 record)
C ..             ! NOTE: Format is nvars(E15.7,1x)
C Line M:  <data>  ! Last line of data
C
C Must link against packlib.lib and kernlib.lib (Add to project
C link settings library list)
C
C Output file contains a CWN (RWN style) with ID=1
C
C Author: Shawn McKee (smckee@umich.edu)

      IMPLICIT NONE


      INTEGER nmax
      PARAMETER (nmax=200) ! Recommended max for HBOOKNC

      CHARACTER infile*80,outfile*80
      INTEGER lin,lout,ID,nvars
      INTEGER I,J

      INTEGER NPAWC, icycle, istat, ipaw
      CHARACTER*8 chtags(nmax)
      REAL rwn(nmax)
      INTEGER NWPAWC
      PARAMETER (NWPAWC=1000000)
      COMMON /PAWC/IPAW(NWPAWC)

      SAVE rwn

      CALL HLIMIT(NWPAWC)

      IF (NARGS().LT.2) THEN
       PRINT *,' Input file name to process:'
       READ(*,*)infile
      ELSE
          CALL GETARG(1,infile)
      ENDIF
      lin = LNBLNK(infile)

      OPEN(10,FILE=infile(1:lin),READONLY,STATUS='OLD',
     +  FORM='FORMATTED',ERR=200)
      READ(10,*)nvars
      IF (nvars.gt.nmax) THEN
          PRINT *,' Too many variables (>',nmax,')!..stopping'
          STOP
      ENDIF
      DO I = 1, nvars
          READ(10,'(A8)')chtags(I)
      ENDDO

      DO I = 1, MIN(INDEX(infile,'.'),lin)
          outfile(i:i) = infile(i:i)
      ENDDO
      lout=LNBLNK(outfile)
C     print *,' lout=',lout
      outfile(INDEX(outfile,'.')+1:INDEX(outfile,'.')+2) = 'rz'


      CALL HROPEN(1,'UMDB',outfile(1:lout),'N',1024,istat)
      ID = 1
      CALL HBOOKNC(ID,'UM DB Ntuple',nvars,'UMDB',rwn,chtags)

      DO WHILE (.TRUE.)
          READ(10,'(<nvars>(e15.7,1x))',ERR=100,IOSTAT=istat)
     +    (rwn(j), j=1,nvars)
C         print *,' rwn(13),rwn(14)=',rwn(13),rwn(14)
          CALL HFNT(ID)
      ENDDO

  100 CLOSE(10)
```

```
      IF (ISTAT.NE.-1) THEN
       PRINT *,' Finished file '//infile(1:lin),' ISTAT=',istat
      ENDIF

      CALL HROUT(ID, ICYCLE, ' ')
      CALL HREND('UMDB')

C     PRINT *,' Output in '//outfile(1:lout)
      GOTO 300
200   PRINT *,' Unable to find input file '//infile(1:lin)
300   CONTINUE
      END
```

11.7 APPENDIX 7

11.7.1 EXAMPLES OF PLOTS GENERATED NIGHTLY AFTER DATABASE UPDATING.

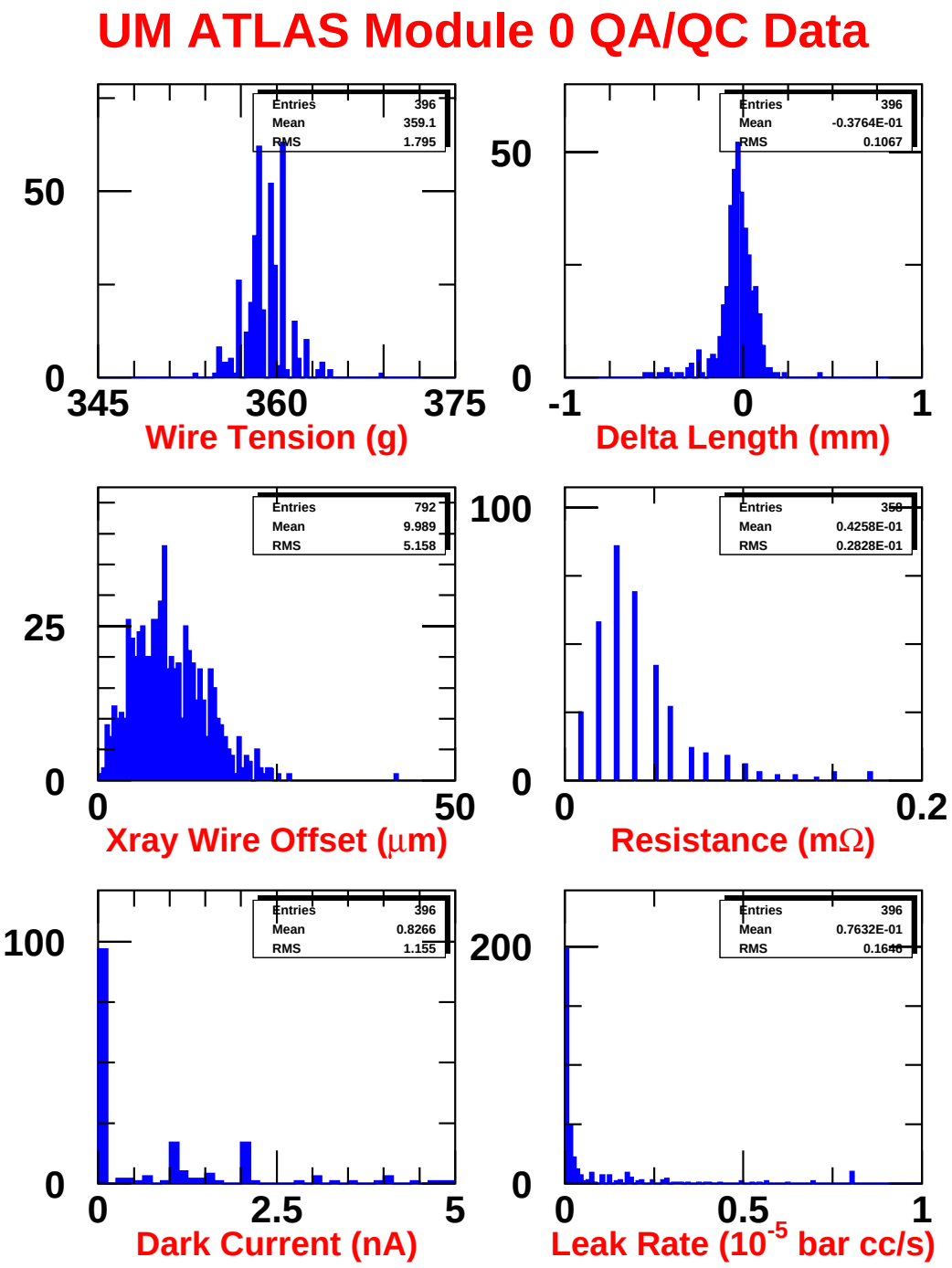


Figure 14Some QA/QC plots which are automatically recreated nightly

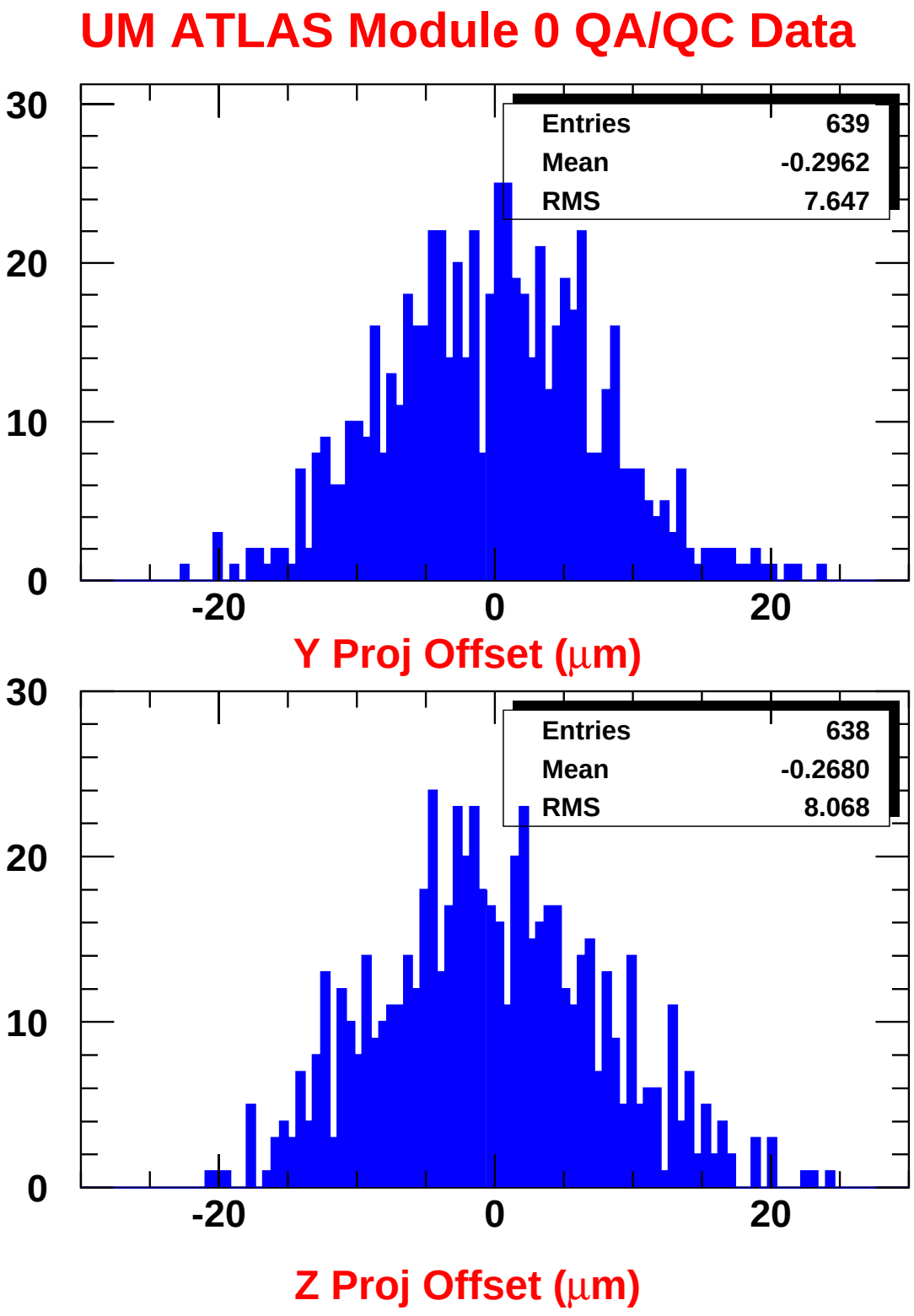


Figure 15 Plots of Y/Z projections for X-ray measurements regenerated nightly

11.8 APPENDIX 8

11.8.1 ODBC: EXPLANATION AND SETUP

ODBC (Open DataBase Connectivity) is a widely accepted API (Application Programming Interface) for database access. By defining such an API, the user of the database is freed from having to know the specific implementation details of particular database programs. This also frees the user from dependence upon a specific database program, since any other database could be used with the same code as long as it provides an ODBC interface.

After defining our ACCESS production database, the next step was to provide the ODBC data source for our data filling and data querying routines. In Windows NT, we open the control panel and double-click the ODBC Data Sources icon. This brings up the following window:

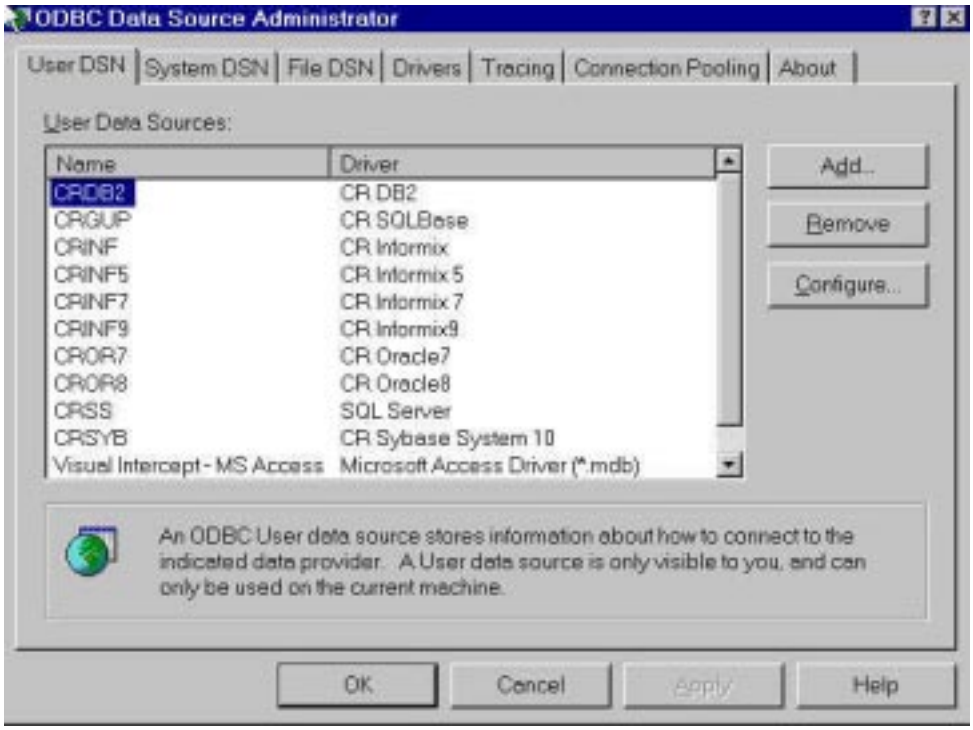


Figure 16 The WinNT ODBC data source configuration window

We wish to add a system-wide data source, so we select the System DSN tab and then click the Add button (shown in the next graphic):

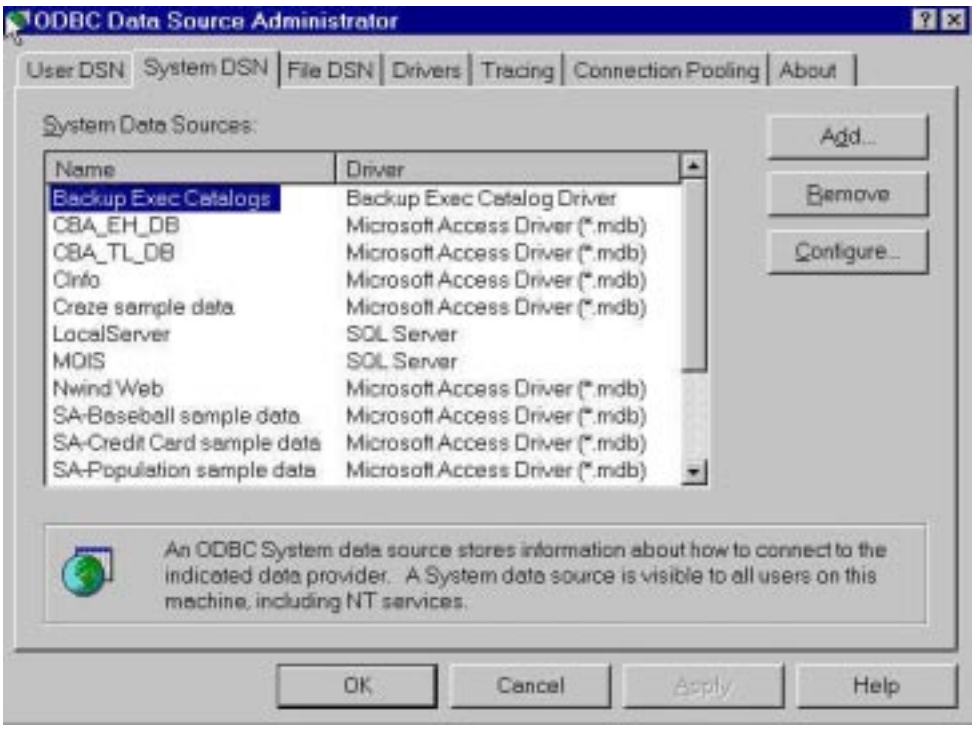


Figure 17 The System DSN tab from the WinNT ODBC32 configuration

The Add dialog then appears. We have chosen to "publish" our DSN (Data Source Name) as 'UM_DB'.

This DSN is then connected to our ACCESS MDT production database. Whenever we need to access the database (for input, queries or output) we can use ODBC (and ADO/RDS) to provide the connection.

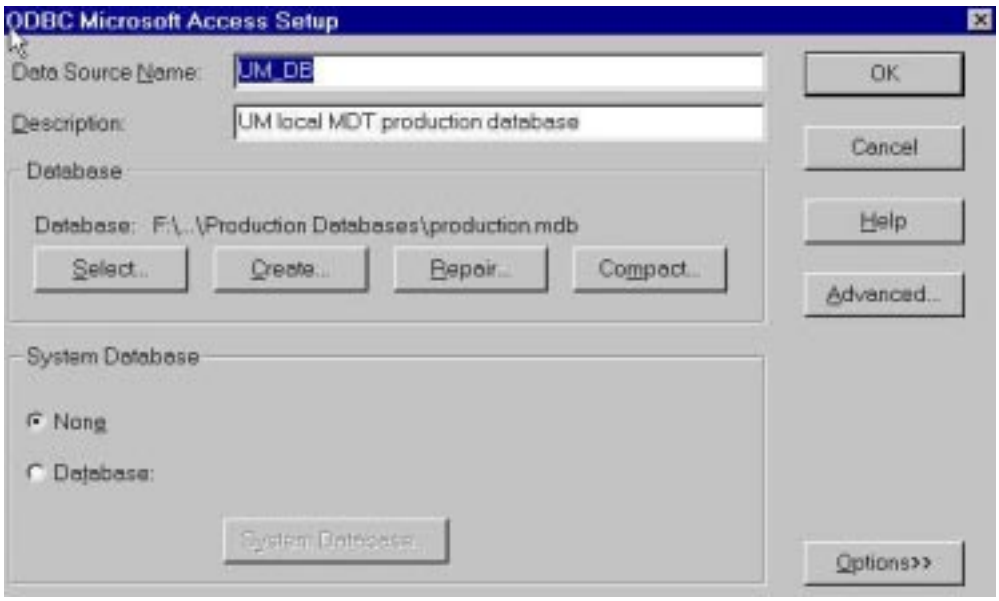


Figure 18 WinNT DSN configuration interface showing UM DSN creation

11.9 APPENDIX 9

11.9.1 SETTING UP ANOTHER SITE BASED UPON THE UM CONFIGURATION

Anyone interested in adapting our setup for their own production can follow the directions in this appendix. The first step is getting a Windows NT machine properly configured. You will need:

- Windows NT Server V4.0 SP4+ (We are using SP5) (See http://support.microsoft.com/servicedesks/servicepacks/servicepacks.htm)
- Microsoft ACCESS 97 with Service Release 2a (Get the SR2a pack from: http://officeupdate.microsoft.com/articles/sr2fact.htm)
- To use the Internet aspects you will need Internet Explorer V5 as a client (supports authentication) and Internet Information Server V4 (comes with NT Server) to act as a WWW server.
- The current Microsoft Data Access Components (MDAC) which you can get at http://www.microsoft.com/data (We have version 2.1.2.4202.3)
- Perl (freely available at http://www.activestate.com)

You must then setup at least two directory areas, one for the WWW pages and one to keep the database and related input files. We have created an area under our internet "root" directory on C:\Inetpub\WWWroot for our WWW pages. This can be setup with Microsoft Frontpage or by using the Internet Service Manager. The production database area can be located elsewhere.

The production database area should have the following directories created:

- Text_data_files
- Export
- Backup

In the following we refer to this directory as the "root" of the production database. You can then copy all our Perl scripts (available from: http://atwww.physics.lsa.umich.edu/aspdb/perl_scripts.htm) into the root area. You will also need to retrieve an empty (template) version of our local production database from http://atwww.physics.lsa.umich.edu/aspdb/blank.asp)

Once you have completed these steps you are ready to customize the configuration for your site. Our site has the production stations defined in this paper with specific formats for the text input files. Your site may be significantly different and so will require extensive changes. At the least, you will need to modify the input file locations and name formats in the Get_files.pl and other places. The Reset_db.pl script will "initialize" your production.mdb file from blank.mdb and create a set of directories for the text input file archiving. Many of the Perl scripts depend upon stored procedures (queries) in the production.mdb file, so you will need to become familiar with the query content of the database. You will also need to define a DSN for your version of the production.mdb following the instructions in appendix 8.

This paper should serve as the primary documentation for the configuration of your site. However, for anyone trying to configure another production site based upon our setup, we strongly recommend examining the Perl scripts and ACCESS queries to understand the overall structure. In case of problems or difficulties, feel free to contact Shawn McKee at smckee@umich.edu.

13. FIGURES

14. ATTACHMENT I:

14.1 ASSEMBLY AREA DEVELOPMENT

The tube and chamber assembly area development at Michigan has been a great challenge for the group since we are building the largest chambers (6m long) among the US muon chamber production sites.

Chamber construction is done on a granite surface table that provides an exceedingly flat and stable surface for assembly. The first difficulty to be faced is the installation of the 645×290 cm^2, 60,000 lb granite table. The floor of our high bay area requires reinforcement to support the table weight, and the door of the high bay needs widening to permit entry of the table. Utilization of the high bay room for large chamber assembly is highly non-trivial. The additional complication is the presence of an electrical transformer room under the high bay which sets additional constraints on the floor reinforcement options. For example, there is an excluded zone around the transformers where pillars may not be placed. As a consequence we had to consider many different designs with the University and rigging company engineers to work out an economical design for the high bay room renovation. Initial construction ("phase I") is nearly finished which will allow us to install the large granite table by the end of March 1999. An assembly room ("phase II") with very good temperature and humidity control will be built after phase I is complete. The design work for phase II is also quite challenging since all the original ductwork and electric wiring must be relocated in the high bay to permit construction of the large assembly room required to handle the 6 meter long chambers. Considerable manpower has been expended during the design process to hold down costs. The design work has been finished and the construction team has been identified. The assembly room is expected to be ready by the end of June, 1999. Our specific utilization of the high bay area for assembly and testing of chambers has been carefully considered. A full-sized dummy chamber has been used to model the movement and handling of chambers.

In parallel with the chamber assembly room construction, we have developed a tube assembly room, which is located in the second floor of the high bay building. Special efforts were required to install two large and heavy optical tables into the tube assembly room. For ease of tube handling an additional door was added to the tube room.

In addition to the assembly room preparations, we have acquired tooling for tube and chamber assembly and testing. Well-designed facilities and tools are crucial to ensure that chambers with the required precision are made on time. The completed work and production plans are described below.

14.2 TUBE ASSEMBLY AND TESTING

We will construct 40,000 drift tubes with lengths ranging from 3.1 m to 5.9 m over a period of 5 years. This goal requires that the tube production rate should reach 50 tubes a day. The tubes must meet high quality specifications which are summarized in table 1. This table also indicates the measurement techniques used to test these specifications. Much progress has been made in defining the QA test techniques, which are presented in a later section of this proposal. To meet the tube quality specifications, we need to develop exceedingly reliable and reproducible techniques for tube assembly. This section first discusses the tube assembly equipment and techniques, and then the testing facilities and procedures.

Item	Specification	Test Method
Tube Length	$\Delta L < 100$ μm	Linear encoder
Wire position	$\Delta x < 25$ μm	X-ray image
Wire tension	375 g ± 2%	Measure fundamental vibration mode
Leak rate	$< 10^{-5}$ bar cc/sec	Helium leak detector with fixture
Dark current	< 5 nA	Precision ammeter
Cosmic-ray rate	$<2 \times$ expected rate	Discriminator and scaler

Table 1: Quality control specifications for MDT tubes

14.2.1 TUBE COMPONENTS

A monitored drift tube consists of three parts: an aluminum tube of diameter 3 cm and 400 μm wall thickness; a pair of endplugs which hold the wire and which provide electrical and gas connections; and a 50 μm gold-plated tungsten-rhenium wire.

The endplug is injection-molded Noryl with an inner brass tube and an outer aluminum cylinder. The brass tube houses a wire positioning device called a ``tango". A tango is a 6 mm long cylinder with 5 perpendicular surfaces along its length leaving a 60 μm hole down the center for the wire. The tango positions the wire to an accuracy of 10 μm with respect to the outer reference surface of the endplug. The inner brass cylinder protrudes out the back of the endplug into a threaded tube on which is attached the gas manifold and ends in a crimp tube to hold the wire. The outer aluminum cylinder provides a surface on which the aluminum tube is swaged to provide mechanical and electrical connection between the tube and endplug. An o-ring provides the gas seal for the tube. The aluminum surface also has a raised 4 mm precision surface of diameter 30.01 ± 0.01 mm which provides the precision reference surface for use in tube placement.

14.2.2 TUBE ASSEMBLY AND TEST STATIONS

We have constructed a complete tube assembly station, a gas leak test station and a cosmic ray test station in our tube construction room. The tube assembly station consists of two optical tables connected end-to-end to form a single 25 foot table as shown in picture 4. The two tables are coupled together with a 25 foot aluminum U-channel that ensures temperature expansions or contractions of the tube are mirrored by movements of the optical tables.

Figure 4: Picture of wiring table

On the optical table are two platforms which hold equipment for inserting and attaching endplugs and wire. One of these platforms is fixed (shown in Figure 5), and the other rides on a 2m linear actuator, whose position is accurate to 1 μm. Mounted on each platform is a 30 cm vacuum chuck which holds the tubes during wiring. Also located on the vacuum chuck is an endplug swaging coil for crimping the endplug onto the tube by electromagnetic swaging, and an automatic endplug insertion fixture, driven by an air actuator. A pair of precision wire crimping jaws are directly mounted on the endplug insertion fixture to crimp the wire into the small tube that holds the wire on the endplug. On the fixed platform there is also a wire tension sensor and a 30 cm linear actuator to control the tension on the wire. The wire spool is mounted at the other end of the movable table. Raw and assembled tubes are stored in two long racks and mounted on one of the long sides of the table. A control computer and associated electronics are mounted on one end of the table. The computer runs control software to operate various actuators and

sensors during the assembly process and to create a database entry of tube parameters for future reference.

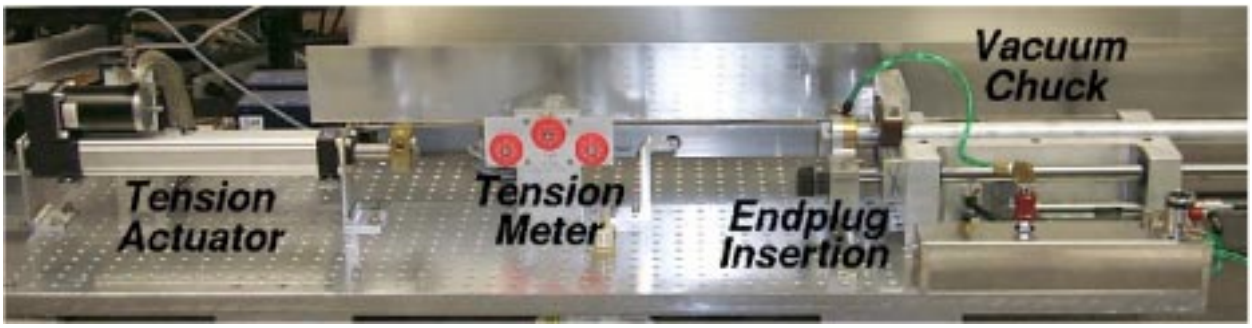


Figure 5: Picture of fixed wiring platform

14.2.3 WIRING PROCEDURE

The wiring procedure has been fully tested using the set-up described in the previous section. We have still to demonstrate tube assembly at the required speed and quality specifications. We briefly describe the wiring procedure below.

Tube assembly is initiated by scanning the bar code of a raw tube into the control computer. Based on the tube ID (which indicates the tube length), computer will set the movable plate to the correct place by moving the long actuator. A raw tube will be taken from the lower rack and placed on two vacuum chucks and held firmly in the vacuum chuck. The wire is then pulled by a vacuum technique from one end of the tube to the other end. The next step is to insert the wire via a current of air into the two endplugs. After the wire is inserted into the endplug, the endplugs are inserted into the tube and the electromagnetic swager is activated causing the tube end to be crimped around the endplug.

With the endplugs crimped on the tube, the wire can be tensioned and crimped. This crimping is accomplished by a pair of vice jaws which have been mounted on the outside of the endplug holder. The computer controls the tensioning by reading the tension on a meter and moving the actuator to tension the wire. The wire is initially tensioned to 450g for 30 seconds to pre-tension the wire. After pretensioning the tension is reduces to 375g, and the wire is crimped.

The whole tube assembly process will take 2 workers about 6 minutes per tube. Allowing for daily calibration, setup and breaks, we expect to make about 50 tubes per day in full production. The quality of the wiring has been checked using the test stations. We describe these tests in the following sections.

14.2.4 TUBE QUALITY ASSURANCE TESTS

The finished tube is immediately put through a series of quality control tests. The first two of these tests are performed in a V-channel which is mounted on the wiring table itself (see Fig. 4). The tube length measurement is done by putting one end of the tube against a stop and the measuring the position of the other end with a computer-read linear encoder, accurate to 10 μm. For wire tension measurement an oscillating signal with a frequency near the expected resonance of the tube is fed into the wire. The driving signal on the wire is then removed and the induced electrical oscillations (waveforms) are read by the computer. This induced signal is then Fourier analyzed to determine the resonant frequency. These two tests take less than one minute. The tube is then queued for leak testing, wire position measurement, and high voltage testing.

It has been realized for long time that the leak test is the most time consuming part of the tube test since we plan to test the entire length of the tube for leaks. A common method is to build a large vacuum tank and test a group tubes together. This method is very hard to implement since it requires very large space for the large tank required for our long tubes. It is also hard to pump a good vacuum with a big tank volume. Instead, we have developed a relatively cheap and fast test technique to test single tubes. It handles different length tubes and requires about 5-6 minutes per tube. In the test procedure, the tube is pressurized to three atmospheres with helium gas, and then a helium gas detector and two different fixtures are then used to detect the tube leak rate. One fixture checks for leak at the joint where the endplug is swaged onto the tube by using a vacuum technique, and a separate fixture is used to look for pin-hole leaks over the length of the tube by using a sniffer technique. This second fixture is slowly moved over the length of the tube via a motorized track to check for leaks. Figure 6 show a picture of this leak detector. Our leak testing method has been adopted by the ATLAS group at the University of Washington, and is being considered by some other groups such as Freiburg, Germany.

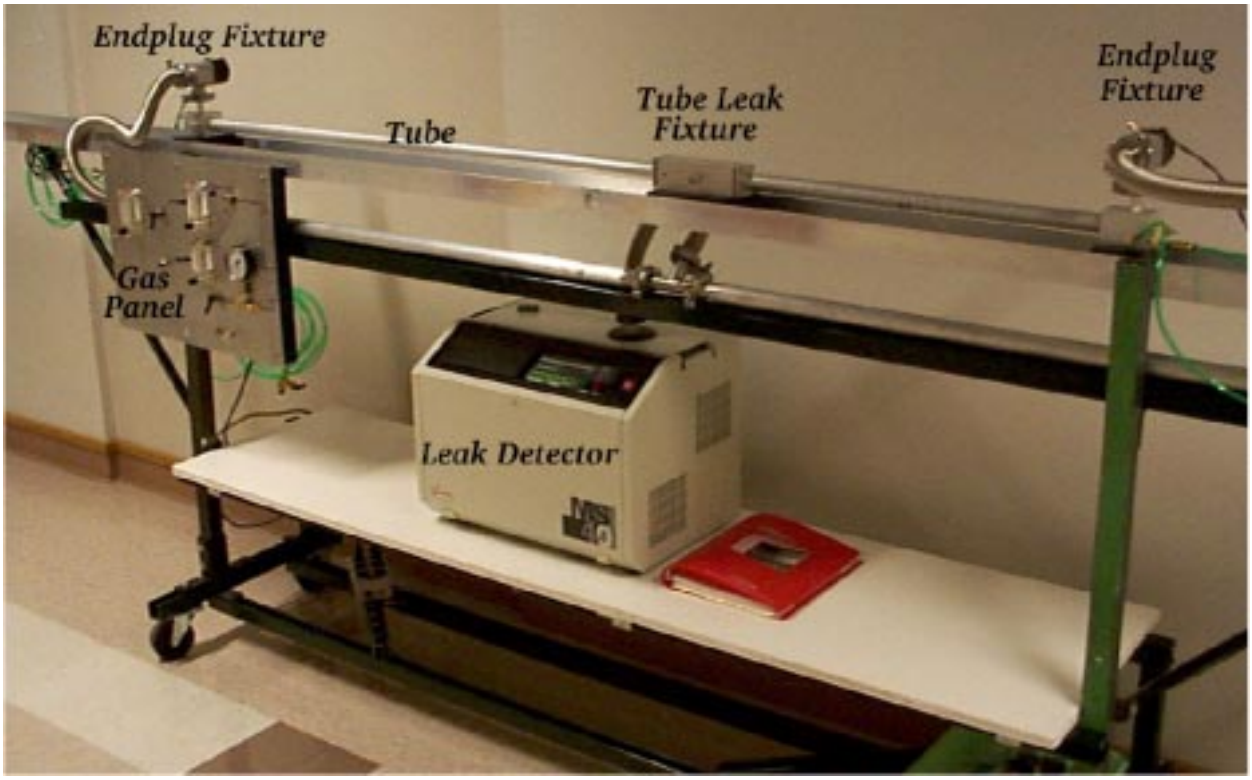


Figure 6: Picture of the Michigan leak detector

The tubes passed leak tests are then taken to an adjacent room to be x-rayed to survey the wire position near the endplug. The endplugs are held securely and the tube is x-rayed relative to several reference wires with a CCD image array connected to a computer. Two orthogonal views of the tube are taken permitting a full 3D reconstruction of the wire position, with an accuracy of 2-3 μm.

After x-raying, the tubes are tested for leakage current and cosmic-ray rate. We have constructed a 50 tube test station, which permits overnight testing of one day's production. We have designed the test electronics board, that connect the tubes to the high voltage, current amplifier/discriminators, and scalars. Micro-ammeters monitor leakage current on each tube and the scalers the cosmic-ray rate.

14.3 CHAMBER CONSTRUCTION

Michigan will construct 104 chambers over next 5 years, building one chamber every two weeks. Maintaining high mechanical precision in chamber assembly is as crucial in chamber production as it is in tube construction. In the past year our major effort has focused on evaluating and prototyping the assembly tools. The completed work and the production procedures are described below.

14.3.1 CHAMBER COMPONENTS

Chambers consist of 6 tube layers with 64 tubes per layer. The 6 layers are divided into two 3-tube multi-layers which are glued on either side of a spacer frame. The endcap chambers are trapezoidal in shape with 8 different tube lengths per chamber. The endcaps come in two different trapezoid shapes with corner angles of 76° for the "long" chambers and 81.5° for the "short" chambers. The spacer frame consists of 2 long beams and 3 cross beams as shown in figure 7. The spacer frame provides some structural support for the tubes, as well as providing locations for the three attachment points which are used to connect a chamber to the ATLAS support structure. Within the spacer frame are mounted three RASNIK optical alignment devices. These devices use a CCD to view a coded mask through a lens to monitor deformations of the chamber. At the ends of the chambers are gas manifolds, HV fanout, and readout electronics.

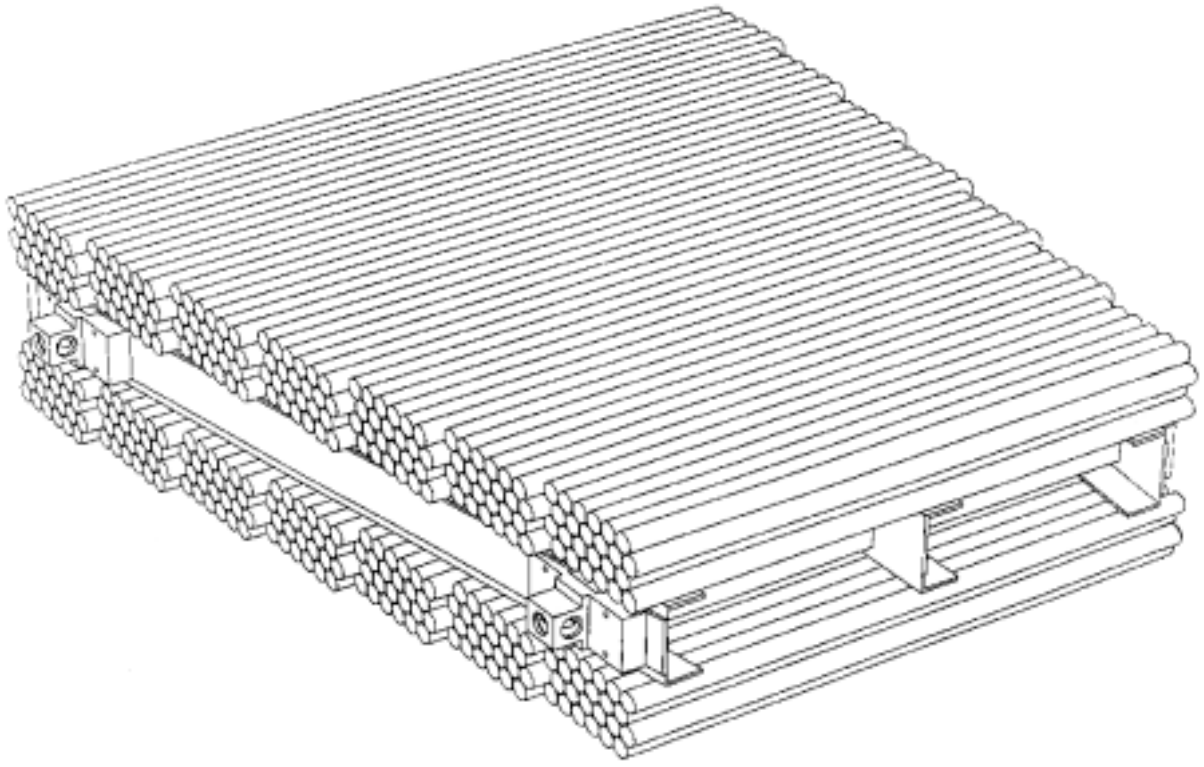


Figure 7: Endcap MDT chamber

14.3.2 CHAMBER ASSEMBLY STATION AND TOOLING

A temperature and humidity controlled assembly room is under the construction in the high bay. This room will include temperature control to ±1° centigrade and relative humidity to 45 ± 5%. This room will house the granite table ($2.90 \times 6.45 \times 0.41$ m^3) which has been polished to a flatness of 10 μm, as well as a large rolling table will be placed next to the granite table. The granite surface plate will be used for assembling the chambers, and the rolling table will be used during assembly to hold a glued multi-layer and spacer and to transport a completed chamber out of the assembly room via the large double doors. An 2-ton overhead crane in the room is designed to move a chamber between two tables.

The construction status and major chamber assembly tools are listed below:

- A set of precision jigs (V-shaped combs) will be mounted and aligned on the granite table to hold a layer of 64 tubes using suction cups. The jigging design was done by our collaborators at Brandeis University. At Michigan we have developed efficient machining tools, and successfully prototyped the first comb-base. Precision measurements have shown that our comb-base meets the design requirement very well. After all the straight combs are machined at Michigan, the final precision jigging assembly work will be carried out at Brandeis University. For jigging alignment we have constructed a laser positioning system, including the software development for the measurement. The large vacuum system has also been designed and is under construction.

- An automatic gluing machine with 3D motion will be used to glue the tubes together. After thorough investigation, we have designed a large, but economical motion system and determined the alignment and mounting method. The motion rails have been mounted on the granite table. The glue dispenser system was designed by the Max-Planck Institute in Munich, Germany, and we are now prototyping and testing this system at Michigan. We expect to test the gluing procedure and timing in next three months.

- A special rigid frame called a “stiffback” will be used pick up tube multi-layers during construction. This frame is similar to a spacer frame, except that it much stiffer. A stiffback has three precision vacuum chucks to pick up tube layers via aluminum strips which have been glued the top tube layer. A different size stiffback is used for each chamber size. The machining work for the stiffback will be shared by Boston University and Michigan machine shops.

- A set of “sphere blocks” which will be used to set the relative heights and offsets of tube layers during gluing. The sphere blocks and towers permit a precise and reproducible setting of the multi-layer to the base plate separation. The sphere block machining work and final assembly work will be shared by the Harvard machine shop and MIT Lincoln Lab.

14.3.3 CHAMBER CONSTRUCTION PROCEDURE

The chamber assembly cycle is a complex procedure taking two weeks. The general sequence of building a chamber is to glue together two 3 tube multi-layers which are then glued on either side of the spacer frame. The time scale of assembly is largely set by the glue curing time which permits only one layer to be glued each day. Therefore, to make a 6 layer chamber glued to a spacer frame takes approximately two weeks.

To start chamber production tubes are placed on the combs and held down via the suction cups on the combs. Care is taken that the tube ends are all properly aligned, and that none of the tubes are touching. After the first layer of tubes in a multi-layer is properly layout on the combs, the glue is applied to the gap between tubes. Three aluminum strips are then glued transversely across the top of the layer of tubes. These strips are used to pick up the tube layer with the stiffback via its vacuum chuck. The vacuum chucks achieve metal to metal contact with the strips so as to maintain a precise and reproducible alignment between the tube layer and stiffback. This alignment is critical for setting the proper spacing and offset between tube layers.

The first layer will be picked up by the stiffback after the glue cure overnight. The next layer of tubes then is then laid out on the combs. This second layer must be glued not just to itself, but also to the first layer. The glue is therefore laid down in 3 beads: one bead between the tubes, and beads at 30° up the side of 2 adjacent tubes. After the glue has been applied, the first layer of tubes is lowered onto the second layer. In order to set the height and offset between the 2 layers, the stiffback is lowered onto the six sphere blocks located on the 4 corners and at the centers of the long side of the chamber. The sphere blocks are set to their lowest height to glue layer 2 onto layer 1.

The glue is again allowed to cure overnight before the first 2 layers are picked up with the stiffback. A third layer of tubes is placed on the combs and glued to the first two layers in the same way as the second layer was glued to the first. The only difference is the height of the sphere blocks which are set to the proper height and offset for the third layer. After the first multi-layer is glued and cured it is transferred to the rolling table. Another 3 tube multi-layer is then build on the granite table in exactly the same manner as the first multi-layer.

When 2 multi-layers are completed they are glued to a spacer frame. The completed second multi-layer is left on the granite table. A spacer frame is lowered by the stiffback via the vacuum chucks and glued to the second multi-layer via the aluminum strips which had previously used to engage the stiffback. A third set of sphere block towers are used to set the proper separation between the spacer frame and the multilayer.

Finally, the first multi-layer is moved into position with the stiffback and glued to the other multi-layer and spacer frame. The fourth set of sphere block towers is used to set the height of the upper multi-layer as it is glued onto the spacer frame (see figure 8).

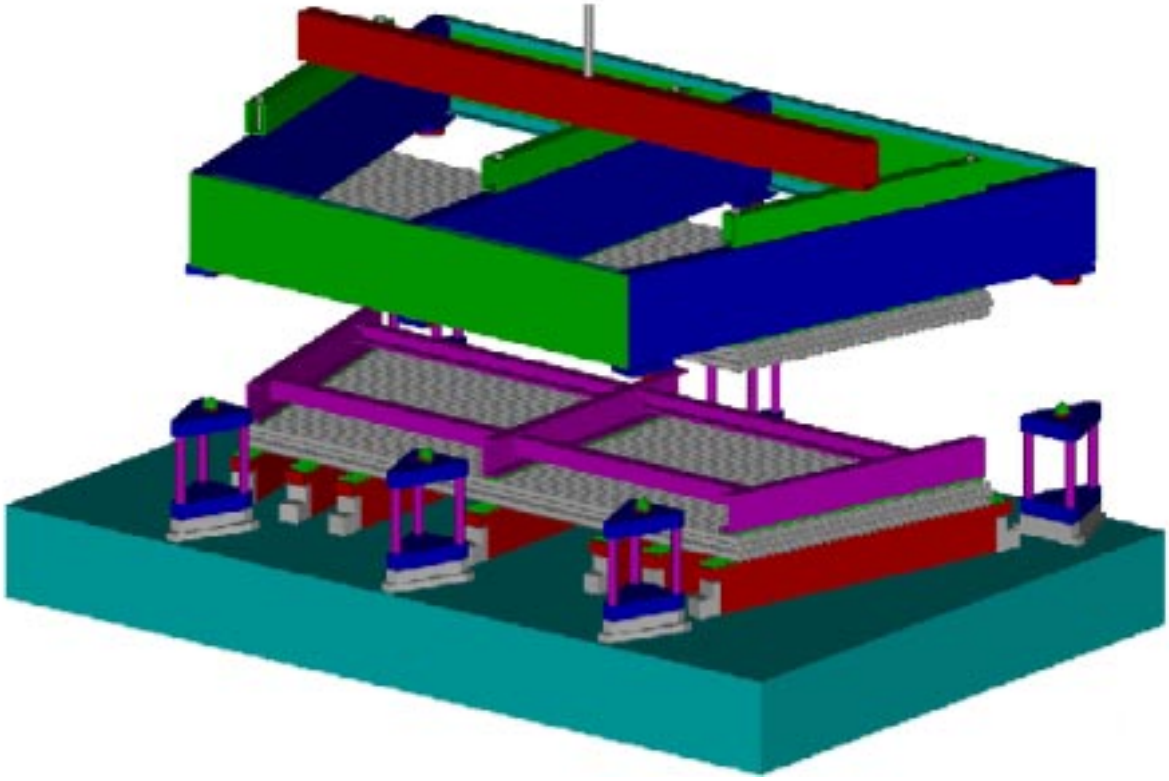


Figure 8: Chamber assembly -- attachment of spacer frame

Once the chamber assembly is complete, it will be moved to the wheeled table. This table is then rolled out into the high bay chamber test area for final installation of gas manifolds, electronics, and testing.

The above procedure is the planned procedure. As we have done with the wiring procedure, we expect to carry out very intensive tests to fine tune and modify tools to ensure that chamber assembly achieves the designed precision and expected production rate.

14.3.4 CHAMBER QUALITY CONTROL TESTS

Chamber quality control tests will be more challenging than that for tubes because of large size and complexity of the chambers. The required tests are listed below:

RASNIK Calibration:

The RASNIK system is used to monitor chamber deformations. It provides only a relative measurement so therefore a baseline measurement must be made with an undeformed chamber. While the chamber is still resting on the granite table the RASNIKs are read out to provide this baseline CCD image.

Gas Connection and Test:

Tube will be gas connected in parallel by 768 individual gas jumpers and two overall gas manifolds. Chamber gas leakage checking normally takes enormous time. As we have done for the tube gas leakage test, we will develop an efficient way to test and monitor the chamber gas system. In fact, we have designed and prototyped an auto feedback pressure sensor system and used it in our tube tests. The leak test will apply the sniffer technique similar as that used in the tube test.

Cosmic-ray test:

We plan to do a full operational test of each chamber via a cosmic-ray test stand. The test stand will have two layers of scintillator for a trigger, with the chamber sandwiched between them. The chamber will operated be under standard ATLAS conditions, i.e. with 3 atmospheres of Ar/CO_2 (93%/7%) at 3080 V. Our plan is to measure each tube's TDC spectrum in the cosmic ray test. This will enable us to identify clearly chamber operation problems.

X-ray survey:

After all the tests are done in the chamber test station, we will ship the chambers to CERN for x-ray tomograph test. This tomograph performs a 3D survey of the entire chamber and determines the positions of all the chamber wires. Each chamber assembly site must demonstrate that it can meet the chamber design specifications before mass production of chambers can begin. Our first chamber is scheduled to ship to CERN in February, 2000.

14.4 MDT DELIVERY MILESTONES

The milestones for Michigan MDT production facilities, chamber design, fabrication, and installation have been specified in the Memorandum of Understanding between the University of Michigan and the U.S. ATLAS Project Office [18]. The key milestones for MDT chamber construction are listed in Table 2.

Milestone	Date
Tube assembly station complete	5/15/99
Chamber assembly station complete	11/15/99
Tube test station complete	6/1/99
Chamber test station complete	12/20/99
Module 0 chamber assembly start	12/6/99
Approve Module 0 complete	1/30/00
Chamber mass production start	2/1/00
Fabrication of 104 chambers complete	12/15/04
Installation of chambers complete	12/15/05

Table 2: Michigan MDT production milestones

We will make every effort to carry out our institutional responsibilities consistent with the overall ATLAS schedule as indicated in the above milestones.