MOFIA 2.0

E614 Analysis Code

I. Introduction

The framework of MOFIA has been adapted from KOFIA, the software package developed for BNL experiment E787. The first version, MOFIA 1.0 contains the code used to analyze the test data acquired with 5 x-planes and 2 y-planes in the test run of August 1997. MOFIA 1.0 includes a full software analysis package of this test data, including: calibrations, efficiency and tracking. The second version, MOFIA 1.5, includes a “skeleton” that would allow the user to access the Monte Carlo data for purposes of developing the remaining software packages for E614. Both MOFIA 1.0 and MOFIA 1.5 are written in FORTRAN 77. Since then the collaboration have decided to adopt a more modern language, FORTRAN 90, due to many attractive features in this language. This lead to the development of MOFIA 2.0. Since FORTRAN 90 compilers are able to compile FORTRAN 77 code, we decided to keep as much as we can of the MOFIA main framework code (the code adapted from E787) in FORTRAN 77. This was mainly done for two reasons: first, the code adapted from E787 is well written and tested so that changing it will unnecessarily consume too much manpower; and second, compatibility with E787 is desired since it will allow us to benefit from E787 modifications to the code. For the same reasons, we decided not to modify any of the other packages that we adapted, namely: CFM, YBOS, and CERN libraries. On the ALPHA, these packages are compiled using the f77 compiler. On LINUX, however, it was necessary to recompile them with f90, due to incompatibilities between our compiler of choice for LINUX f90, ABSOFT, and the g77 compiler. Only minor changes, however, were made to these packages.

All the E614 code written for MOFIA 1.5 was completely rewritten and reorganized in MOFIA 2.0 in order to utilize the nice features of FORTRAN 90. We have also rewritten the data structures in this version. MOFIA 2.0 also includes the geometry structures and the TDC unpacking for the proportional chambers. In addition, MOFIA 2.0 is capable of analyzing the test data of August 1999, for the UV prototype chamber pair. This last feature allowed us to test MOFIA 2.0 with real data and compare directly the results from MOFIA 2.0 and MOFIA 1.5. While the collaboration have decided to support four computer platforms for the entire E614 code, at this stage MOFIA 2.0 has only been tested on two platforms, ALPHA and LINUX. Monte Carlo and real data analysis results were identical for the two platforms.

In the discussion below, some knowledge of FORTRAN 90 is assumed. We found that the book “Upgrading to FORTRAN 90” by Cooper Redwine, provides a good introduction to FORTRAN 90, at least for those familiar with FORTRAN 77. Specific documents for the ALPHA and the ABSOFT compilers and debuggers are also useful references. For ABSOFT, this documentation may be found on any machine where ABSOFT is installed in the directory ***/usr/absoft/doc.*** In particular, the file ***FxUserGuide.pdf*** provides information on the ABSOFT ***Fx*** multi-language debugger, and the file ***F90_Reference.pdf*** provides information on the FORTRAN 90 compiler. Information on our debugger of choice for the ALPHA, ***ladebug,*** is on the web at

http://www.unix.digital.com/faqs/publications/base_doc/documentation/html/ladebug/lp.html

and information on DEC FORTRAN 90 is at

http://lin00.triumf.ca/internal/internal-links/df90/dfau.htm

II. Initialization Branch

The MOFIA 2.0 initialization branch may be divided into two parts: geometry implementation and histogram definitions as shown in Figure 1.

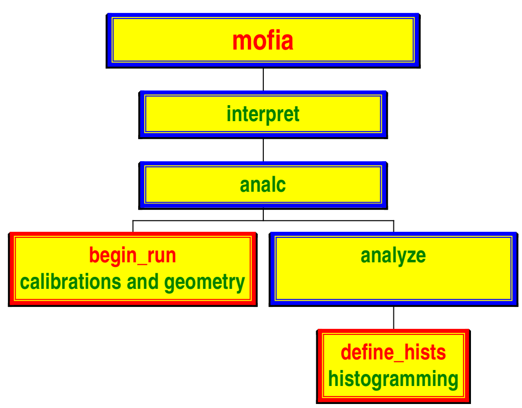

Figure 1. MOFIA 2.0 initialization branch.

II.1 Geometry and Calibrations

II.1.1 Geometry

As in MOFIA 1.5, the geometry description of the E614 detector is read in from an *ascii* data file common to both the Monte Carlo and the data analysis codes to ensure consistency. The geometry input file name has the form *dt_geo.nnnnn*, where *nnnnn* is the file's version number (for example, *dt_geo.00001* for version 1). The geometry input file is managed through the Calibration File Manager ***CFM***, where the association between version numbers and run numbers is made. This allows us to keep track of any geometry changes in the E614 detector. The module ***det_geom_mod*** contains the subroutines ***OpenGeom***, which opens the appropriate geometry file for the run number at hand (by consulting with CFM), and ***ReadDetGeom*** which reads in the geometry data file. The geometry results are saved in a common block, ***det_geom.inc***, which is common to both the analysis code and the Monte Carlo code, as is the parameter file ***det_geom.par*****,** which contains useful geometry parameters. Figure 2 shows a block diagram for the code organization in the geometry initialization branch.

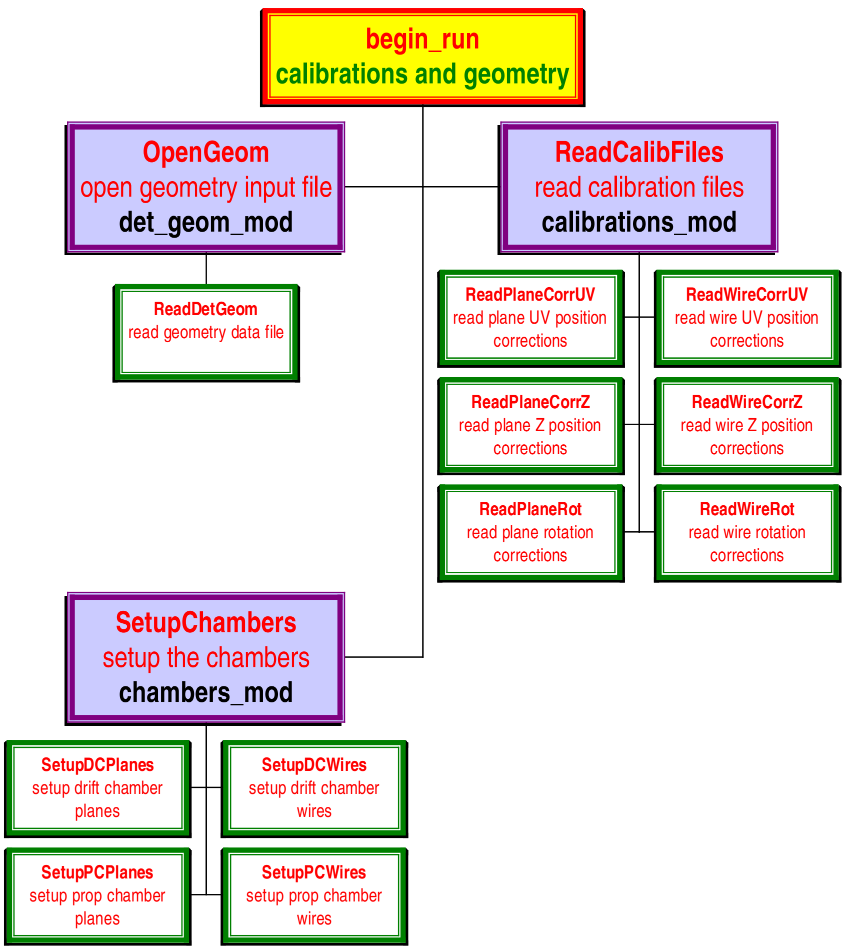

Figure 2. MOFIA 2.0 calibrations and geometry branch.

The module ***chambers_mod*** is where all the geometry structures for the drift chamber and proportional chamber are defined and filled. These structures are shown schematically in Figure 3.

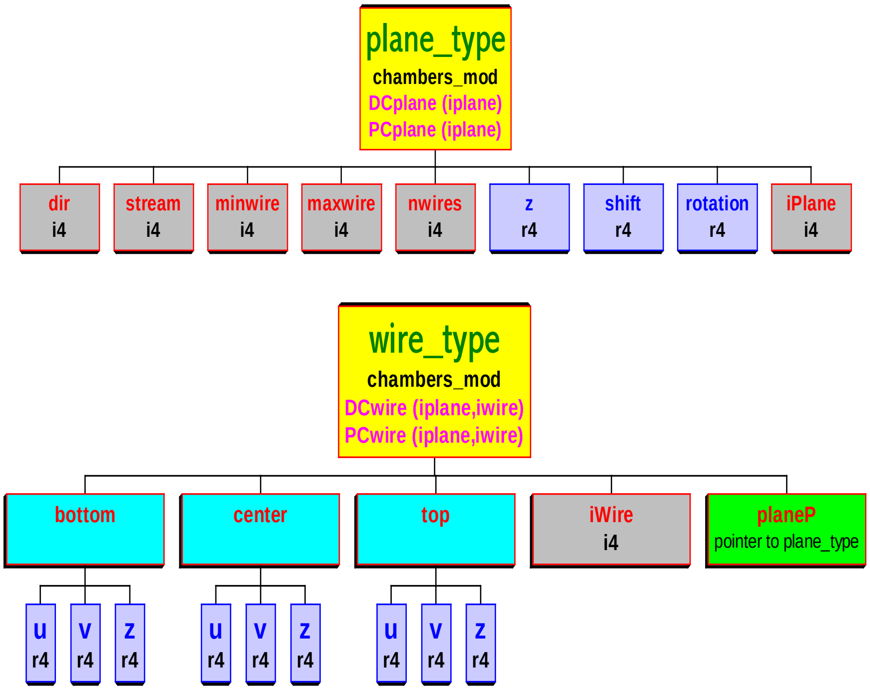

Figure 3. MOFIA 2.0 geometry structures.

The two types of structures, *plane_type* and *wire_type*, include the plane and wire geometry structures, respectively. Each type has two instantiations, one for the drift chamber and one for the proportional chamber: ***DCplane(iPlane)***, ***PCplane(iPlane)***, ***DCwire(iPlane,iWire)***, and ***PCwire(iPlane,iWire)***. Table 1 contains a brief description of the contents of the *plane_type* structure.

Plane_type	Description
dir	Coordinate measured by plane (U or V)
stream	Plane location (upstream or downstream)
MinWire	Number of the first wire in plane
MaxWire	Number of the last wire in plane
nWires	Total number of wires in plane
z	Z position of plane
shift	U or V position of plane
rotation	Angular orientation of plane
iPlane	Plane number

Table 1. A brief description of the components in the geometry structure *plane_type*.

The following are examples of how the first four components of *plane_type* are used. See ***chambers_mod*** for more details.

```
IF (DCplane(iPlane)%dir == dir%u) THEN

IF (PCplane(iPlane)%stream == stream%up) THEN

DO iWire = DCplane(iPlane)%MinWire, DCplane(iPlane)%MaxWire
```

The structures, ***dir*** with components ***u*** and ***v*** (*dir%u,dir%v*) and ***stream*** with components ***up*** and ***down*** (*stream%up,stream%down*), have also been defined in ***chambers_mod*** and are available for usage by any module or procedure that uses ***chambers_mod***. Since some of the planes will be shifted (by half a cell) for purposes of resolving the left-right ambiguity in the tracking algorithm, the component ***shift*** contains a U or V shift of that plane relative to the "unshifted" planes. The component ***rotation*** contains the angular orientation of the plane so that U planes have a rotation of -45^0, whereas V planes have a rotation of 45^0. While the component ***iPlane*** is obviously redundant when used in the form *DCplane(iPlane)%iPlane*, the reason for putting it in the structure is to allow accessing the plane number through a pointer in the hit structure, and its usefulness will become evident when the hit structure is discussed below.

Table 2 contains a brief description of the components of *wire_type*. The components ***iWire*** and ***planeP*** were introduced in this structure for the same reason that ***iPlane*** was introduced in the *plane_type* structure, and their usefulness will also become evident when the hit structure is discussed.

wire_type	Description
bottom	Wire "bottom" end point position coordinates (bottom%u,bottom%v,bottom%z)
center	Wire central position coordinates (center%u,center%v,center%z)
top	Wire "top" end point position coordinates (top%u,top%v,top%z)
iWire	Wire number
planeP	Pointer to *plane_type*

Table 2. A brief description of the components in the geometry structure *wire_type*.

As shown in Figure 2, access to the chambers module is achieved through a call from ***begin_run*** to the ***chambers_mod*** PUBLIC subroutine, ***SetupChambers***, which in turn calls PRIVATE subroutines within the ***chambers_mod*** that do the job of filling up the geometry structures. This reflects a general philosophy employed in developing the code, namely minimizing the number of entry points to the module by having a few (preferably one) PUBLIC subroutine to be called from outside the module, while the remaining subroutines in the module are made PRIVATE and can, therefore, only be called from within the module. The ***chambers_mod*** has a second entry point for testing purposes. The PUBLIC subroutine ***PrintGeom***, is accessed through a call from the subroutine ***func***, and maybe invoked by typing "*func 6*" at the MOFIA command line. When invoked ***PrintGeom*** calls private subroutines within ***chambers_mod*** that create geometry ouput data files for purposes of testing the geometry information (*dc_planes_geom.out, pc_planes_geom.out, dc_wires_geom.out, and pc_wires_geom.out).*

II.1.2 Calibrations

Figure 4 shows the calibrations data structures. As shown in this figure, these corrections include U or V and Z position shifts, for each DC plane and wire, as well as rotation corrections for each DC plane and wire. These corrections are included in the geometry structures in ***chambers_mod***, so that the components ***z***, ***shift*** and ***rotation*** in the *plane_type* structure already have these corrections in.

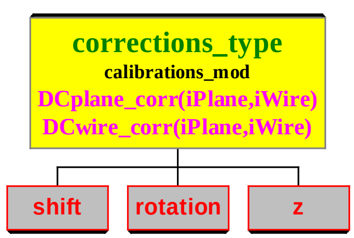

Figure 4. MOFIA 2.0 calibration structures.

Procedures in the module ***calibrations_mod*** are accessed through a call from ***begin_run*** to the module's PUBLIC subroutine ***ReadCalibFiles***, which in turn calls PRIVATE procedures within the module to open the appropriate calibration files for the run at hand (through calls to CFM) and read them in. The contents of the calibrations structures are then accessed in ***chambers_mod*** through a *USE* statement (*USE calibrations_mod*), and the appropriate corrections are made to the geometry variables.

In addition to these calibration structures, a rotation-correction parameter is installed in ***chambers_mod*** to allow for an overall rotation of the chambers (DC and PC). This parameter, ***ChamberRot***, is in the namelist ***DCset*** and may be assigned values as any other namelist variable (see subsection ***namelists***).

II.2 Histogramming

Since users tend to define a large number of histograms for purposes of testing their code or analyzing some specific data, in many cases keeping these histograms as part of the official E614 code is not practical. For one thing, CPU time will be wasted filling in histograms that most users don't need; and for another, the number of histograms created becomes large so that sorting through the histogram list to find the desired histogram becomes tedious. The decision was therefore made to provide two histogramming modules. The first, ***hists_mod***, is the module that contains the official histograms that are to be kept and used by all users. The second module, ***user_hists_mod,*** is where each user defines their own histograms. The user is responsible for keeping and maintaining their own copy of this module. The default ***user_hists_mod*** that is kept with the official E614 code will contain no histogram definitions. If you add your own histograms make sure you don't overwrite your own copy of ***user_hists_mod*** when you update your code; save this module under a different name before updating your code. The histogramming branch is shown in Figure 5.

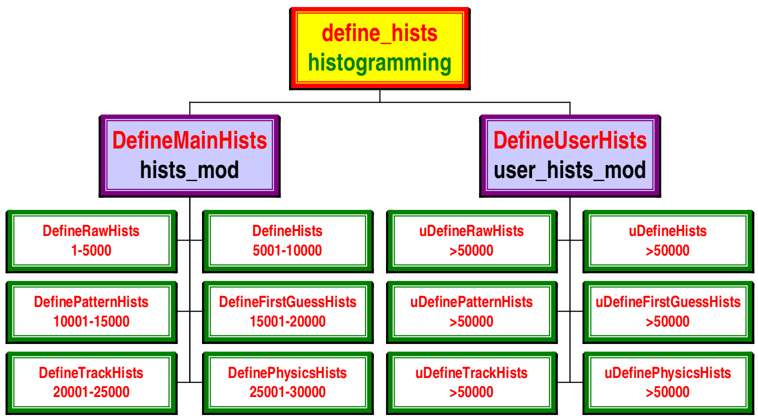

Figure 5. MOFIA 2.0 histogramming branch.

The subroutine ***define_hists*** makes two calls, one to the PUBLIC subroutine ***DefineMainHists*** in ***hists_mod***, and the other to its counterpart, ***DefineUserHists*** in ***user_hists_mod***. These subroutines, in turn, call PRIVATE subroutines within the module, one for each section of the code: raw histograms, histograms after initial filtering, pattern recognition histograms, helix fit "first guess" histograms, tracking histograms, and physics analysis histograms. As seen from figure 5 these subroutines have similar names in the ***hists_mod*** and ***user_hists_mod*** except that the letter "u" (for user) is prepended to the name in the ***user_hists_mod***.

To avoid conflicts between the user defined histogram numbers and the main histogram numbers, values between 1 and 50,000 are reserved for the main histograms; user histograms should always have numbers higher than 50,000. To avoid conflicting histogram numbers between the main histograms themselves, a range of histogram numbers has been reserved for each section of the code, as specified on the chart of Figure 5. Histogram numbers are assigned to variables in the declaration section of the histogramming modules. This has two advantages. First, if a histogram number is to be changed it only needs to be done once (rather than once where the histogram is defined and once where the histogram is filled) which reduces the potential for mismatches. Second, if this list is maintained in ascending order it becomes easy to see which histogram numbers have already been used and reduces the risk of conflicts.

III. Analysis Branch

A schematic view of the MOFIA 2.0 analysis branch is shown in Figure 6. The procedure ***dplot*** is the E614 event analysis subroutine. From ***dplot*** calls are made to analyze the event starting with the TDC unpacking, filling the histograms, filtering the event, etc.

III..1 TDC Unpacking

From ***dplot*** a call is made to the PUBLIC function ***tdcunp*** in module ***tdc_mod***, in order to unpack the TDCs. This module also contains the declarations and initializations (and filling) of the data structures associated with the TDC hit structures which are shown in Figure 7. Table 3 shows a brief description of the *tdc_type* structure. This type has two instantiations, ***DCtdc(ihit)*** and ***PCtdc(ihit)***. The first two components of the structure are the time and width of the TDC signal. The third component, ***flag***, is an integer that is assigned a zero value for normal TDC signals, and a non-zero value if the TDC signal has a peculiar characteristic (such as a leading edge but no trailing edge, etc). Table 4 contains a listing of these flags. The last two components in this structure are two pointers. The first, ***wireP***, points back to the wire geometry structure and, therefore, provides the link between the hit structure and the geometry structures. To point to the wire and plane numbers for a particular DC TDC hit, for example, we have

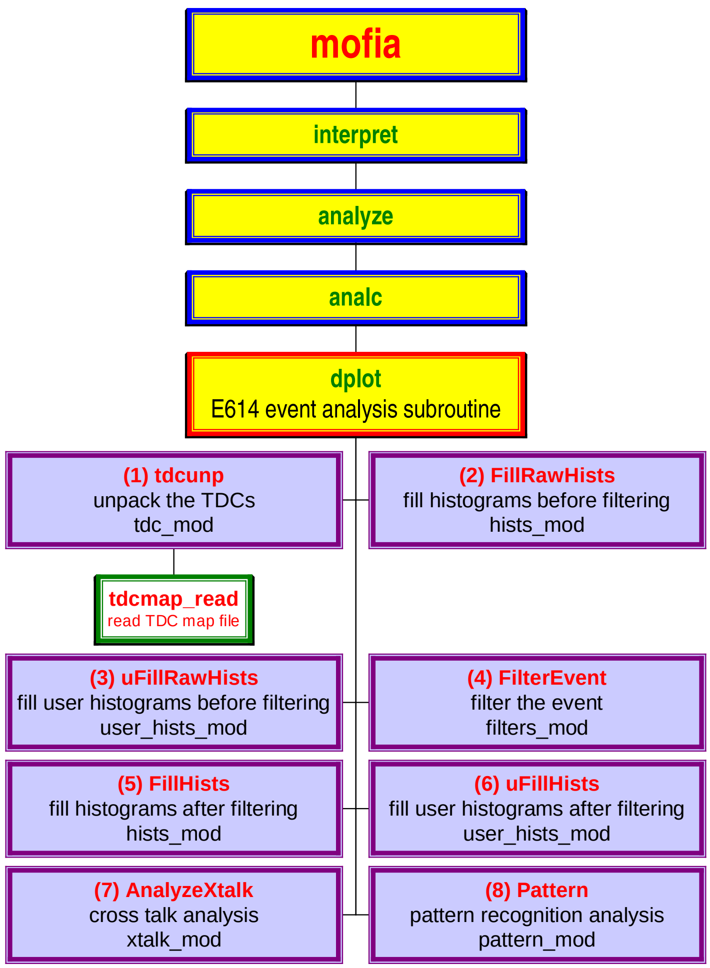

Figure 6. MOFIA 2.0 analysis branch.

```
INTEGER (i4), POINTER :: iwP, ipP

iwP => DCtdc(ihit)%wireP%iwire

ipP => DCtdc(ihit)%wireP%planeP%iplane
```

and so on. In the above example we could have defined integer variables iw and ip (instead of pointers iwP and ipP); in which case we have

```
INTEGER (i4) :: iw , ip

iw = DCtdc(ihit)%wireP%iwire

ip = DCtdc(ihit)%wireP%planeP%iplane
```

However, since copying data from a structure to a new variable is generally less efficient than using a pointer, it is recommended that the above style be used.

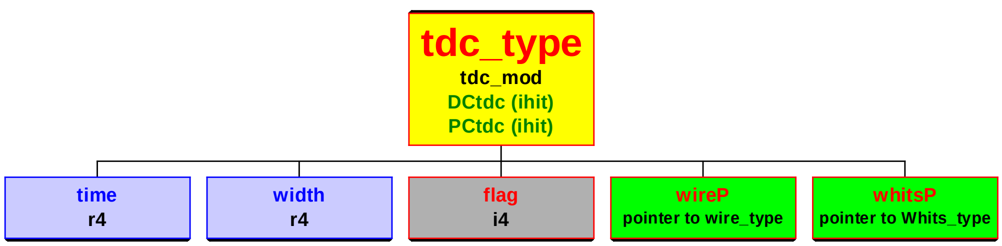

Figure 7. MOFIA 2.0 TDC hit structure.

flag **Value**	**Description**	***width*** **Value**	**Details**
0	Good hit		Trailing edge followed by leading edge on same channel
1	No leading edge	0	Trailing edge following another trailing edge
2	No leading edge	0	Last edge on channel, but it's a trailing edge
3	No leading edge	0	Last edge was a trailing edge from a different channel
4	No trailing edge	99999	First edge on channel, but it's a leading edge
5	No trailing edge	99999	Leading edge following another leading edge
6	No leading edge	0	Last edge on channel is a trailing edge
7	No trailing edge	99999	First edge on channel is a leading edge
7	Width < 0	0	Trailing edge followed by leading edge, but width < 0

Table 3 Description of the ***flag*** values in the *tdc_type* structure.

The second pointer in the *tdc_type* structure, ***whitsP***, provides the link to the *whits_type* structure shown in figure 8. This structure is also defined and filled in the module *tdc_mod*. The first element in this structure, ***nhits***, is the number of TDC hits on that wire and the second element, ***hits(index)***, contains the hit number as it appears in the hit list in the *tdc_type* structure. This is best understood through an example. The number of hits on wire 32 in

DC plane 26 is then

```
DCwhits(26,32)%nhits
```

To assign two pointers, ihitP and timeP, to the hit index and TDC time for the *second* hit on this wire we have

```
INTEGER (i4), POINTER:: ihitP
REAL (r4):: timeP
ihitP => DCwhits(26,32)%hits(2)
timeP => DCtdc(ihitP)%time
```

The above example demonstrates another case of assigning pointers to a component of a structure. While the last two lines may have been combined in one, so that

```
timeP => DCtdc(DCwhits(26,32)%hits(2))%time
```

the style of assigning a pointer makes the code more readable. It is also more efficient, since typically such statements would be inside do loops (looping over planes, wires, hits, etc), and hence the hit number is extracted from the structures **once** by assigning it to a pointer(ihitP, in this case), and the pointer is then used within that loop from that point on. Another useful technique is to assign a pointer to a structure (as opposed to a component of a structure in the examples above). This also provides faster execution time; and improves the readability of the code. For example, one can declare the pointer whitP and assign it to a structure

```
INTEGER (i4), POINTER:: whitP, nhitsP
DO iPlane = 1, Ndplanes
   DO iWire = 1, Ndwires(iPlane)
      whitP => DCWhits(iPlane,iWire)
      nhitsP => whitP%nhits
   END DO
END DO
```

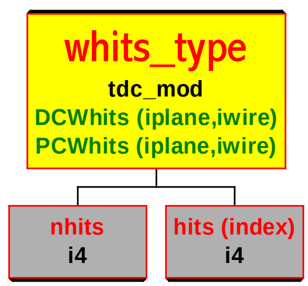

Figure 8. MOFIA 2.0 wire hits structure.

tdc_type	Description
time	TDC signal leading edge
width	TDC signal width
flag	Characteristic of the TDC signal (described in table 3)
wireP	Pointer to wire geometry structure wire_type
whitsP	Pointer to wire hits structure whits_type

Table 4. A brief description of the components of the hit structure tdc_type.

III.2 Filtering and Histogramming

Following the call to ***tdcunp***, a call is made to ***FillRawHists*** (and its user counterpart, ***uFillRawHists***) where some histograms are filled before any filtering is done. The event is then filtered by calling the PUBLIC subroutine ***FilterEvent*** in module ***filters_mod***. A call is then made to ***FillHists*** and ***uFillHists*** to fill some histograms after event filtering.

III.3 Analysis Software Under Development

Cross talk analysis is presently underway for the test data acquired in August 1999. Since the FORTRAN 90 version, MOFIA 2.0, was not ready at the time, this analysis was written using FORTRAN 77 and has not yet been converted. This will be done soon so that the next release of MOFIA will contain the cross talk analysis. The PUBLIC subroutine ***AnalyzeXtalk*** in module ***xtalk_mod*** will be called for this purpose. Once the Cross talk hits are characterized, a call will be made to the pattern recognition code then the helix fit "first guess" code, both of which are currently under development, then the tracking code and finally the physics analysis code.

Figure 9 shows the proposed data structures for the hit position, *hitpos_type*, and the tracking, *track_type*. The components ***u***, ***v***, and ***z*** of the hit position structure will be initialized to wire positions. Once the timing information is translated to distance (and the tracking is done to resolve left-right ambiguity, etc) these components will contain the actual hit position. The component ***angle*** will contain the track intersection angle in the hit cell (which is needed by GARFIELD to convert the TDC time to distance), and the component ***weight*** will contain the weight for that hit to be used by the track fitting algorithm.

The *track_type* structure will contain the number of hits in the DC and PC, ***nDChits*** and ***nPChits*** as well as the hit number as it appears in the TDC hit list in the *tdc_type* structure (see the section ***Analysis Branch*** for details and examples). The substructure ***helix*** will contain the five helix parameters, and ***ChiSqr*** the χ^2 value for the track fit.

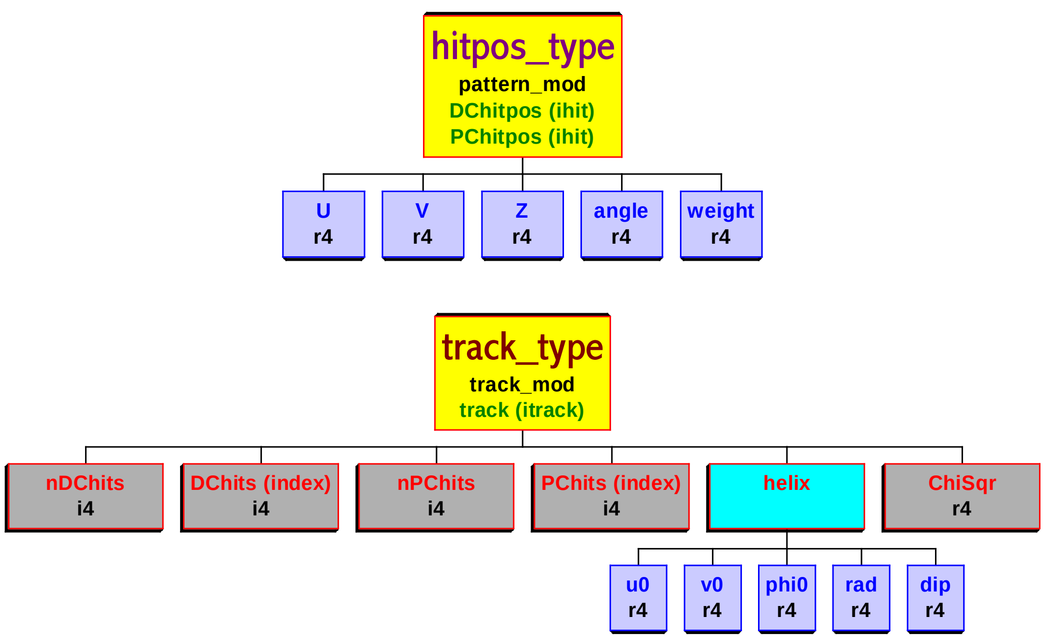

Figure 9. Proposed structures for the hit position and tracking.

IV. Source Code Organization

The source code for MOFIA 2.0 is in the directory ***mofia/2.0/source***. Several subdirectories reside in it: ***main***, ***mainf90***, ***dummy***, ***include***, ***modules***, ***photo***, and ***user***. The subdirectory ***main*** contains the MOFIA "mainframe" code (adapted from E787); this is the part that was not changed and is still in FORTRAN 77 format (the only exception is ***begin_run***, and ***define_hists*** where *USE* statements had to be introduced to allow calls to module subroutines). Code in this directory has the extension *.F* and is therefore compiled with the *f90* compiler as FORTRAN 90 *fixed format* code. The subdirectory ***mainf90*** contains the E614 main analysis code. For this release, the TDC unpacking, geometry, calibrations, and main histogramming modules reside in this subdirectory. This subdirectory will also contain other software packages once they are developed and tested. In contrast, code that is still under development and testing, as well as utility code that is needed by different users will reside in the ***user*** subdirectory. The code in the ***mainf90*** and ***user*** subdirectories carries the extension *.f90* and is therefore compiled with the *f90* compiler as FORTRAN 90 *free format* code. The ***photo*** subdirectory includes the code needed for the **ROOT** package (event display) which is written in C++. The ***photo*** directory therefore contains C++ code as well as some FORTRAN subroutines to provide the link between the MOFIA FORTRAN code and the display C++ code. The ***include*** subdirectory includes all the include files, common blocks, parameter files, and interface blocks.

When a module is compiled, the compilation process goes through two steps: first a file with extension *.mod* is created and then the *.obj* file is created. The *.mod* file resides in the ***modules*** subdirectory. These *.mod* files are needed when other modules that *USE* them are compiled to enforce type checking on all procedure calls. Finally, the subdirectory ***dummy*** contains "dummy" modules and subroutines. This serves two purposes. First, for purposes of testing (or convenience) a certain part of the code might not be required. This is achieved by calling the equivalent dummy module/subroutine to replace the actual package. Second, the use of dummy modules provides a way to simplify the compiling and linking process when interdependencies between modules in different subdirectories is present (through *USE* statements). The dummy subdirectory also contains dummy subroutines for user code such as ***my_begin_run***, ***my_end_run***, ***my_init***, etc. When the user needs to modify these subroutines, they should be copied to the user subdirectory, modified, and added explicitly to the *Makefile* (in the ***user*** subdirectory). The compiler will first look for these files in the user subdirectory, if the specified files are not present, the compiler will find the *.obj* files in ***libdummy.a*** and link with them instead.

V. Compiling MOFIA 2.0

Each of the subdirectories: ***main***, ***mainf90***, ***dummy***, and ***photo*** has its own *Makefile*. When the make file is executed, a library corresponding to this subdirectory is created (or updated) and placed in the directory ***mofia/2.0/lib***. The corresponding libraries are ***libmain.a***, ***libmainf90.a***, ***libdummy.a***, and ***libphoto.a***. Notice that the include files used by code in any of these subdirectories resides in the ***include*** subdirectory, and similarly the *.mod* files created when compiling code in any of these subdirectories resides in the ***modules*** subdirectory. In contrast, when the *Makefile* in the ***user*** subdirectory is executed, the *.mod* and *.obj* files remain in the ***user*** subdirectory (no libraries are created).

Users should never run there own code in any of the MOFIA subdirectories. Instead the code in ***mofia/2.0/source/user*** should be copied to the user's own directory (which I will call ***mymofia*** in this document to distinguish it from ***mofia/2.0/source/user***) and compiled there. If files from the other ***mofia/2.0/source*** subdirectories (***mofia/2.0/source/mofia90, mofia/2.0/source/dummy, etc***) are needed, these files should be copied over to ***mymofia*** and added to the list of *$OBJECTS* in the ***mymofia***/*Makefile*.

When compiling MOFIA the user has the option of making any of three different executables: **plain**, **mofia20** or **photo**. The first executable, **plain**, does not contain any code from the ***mainf90***, ***user***, or ***photo*** subdirectories. The second executable, **mofia20**, contains all of the MOFIA code except for the event display (***photo*** subdirectory). The last executable, **photo**, contains the entire code. To make any of these three executables the user needs to issue the command "*gmake executable*" where "*executable*" stands for **plain**, **mofia20** or **photo**. For example,

```
gmake photo
```

makes the photo executable. If only the *gmake* command is issued (with no executable name) all three: **plain**, **mofia20** and **photo** will be made.

V.1 Compiling a *debug* Version

The libraries corresponding to ***main***, ***mainf90***, ***dummy*** and ***photo*** in ***mofia/2.0/lib*** are all compiled with the *debug* flag off to allow faster execution. Since debug information may be needed at times, a debug version of all these libraries is provided in the subdirectory ***mofia/2.0/debug***. The default libraries are the non-debug ones, so that the environment variable ***MOFIA_LIBDIR*** is assigned to ***mofia/2.0/lib*** by default, and the *debug* flag is turned off when the code in the ***user*** subdirectory is compiled. To compile a debug version of MOFIA the user needs to issue the command

```
mlib f90 debug
```

before executing the *Makefile* in the ***mymofia*** subdirectory. This will assign the environment variable ***MOFIA_LIBDIR*** to ***mofia/2.0/debug*** and turn the debug flag on when compiling the ***mymofia*** code. If the user wishes to go back and recompile a non-debug version, the command

```
mlib f90
```

must be issued before executing the *Makefile*.

V.2 Module Dependencies

It would be inefficient to recompile all the code whenever a file is modified in one of the ***source*** subdirectories. On the other hand, compiling only the file modified will not be sufficient in some cases since other files might depend on it. For example, if module A is changed, one needs to find all the modules and procedures that *USE* module A and recompile them. This is also true when any of the include files is changed. In each of the subdirectories ***main***, ***mainf90***, ***dummy*** and ***photo*****,** a file called ***Dependencies*** is supplied to serve this purpose. This file contains a list of modules and include files that a *.obj* file "depends on" so that if any of these files is changed the *.f90* (or *.F*) file corresponding to this *.obj* file is recompiled. This information is made available by inserting the statement "include Dependencies" in every *Makefile* in the ***source*** subdirectories.

The ***Dependencies*** file itself is created by executing the script ***depend_f90.csh*** (which resides in the directory defined by the environment variable ***TRIUMF_ROOT***) from the ***source*** subdirectory in which the dependencies are to be found. This script goes through every file in this subdirectory and extracts all filenames that this file depends on (by examining all the *USE* and *include* statements in that file).

VI. Running MOFIA 2.0

As mentioned in the introduction, the framework for MOFIA 2.0 was not changed in this version. This section is almost identical to that in the MOFIA 1.5 "manual".

VI.1 General Commands

MTIN "argument": Assign the input file/device to argument. For example, ***MTIN "/ralph1/usr/data8/e614/data/run00092.dat"*** assigns the input file to *run00092.dat* in directory */ralph1/usr/data8/e614/data*. Similarly the command ***MTIN "/dev/mx3d"*** assigns the input to the tape drive */dev/mx3d*.

analyze "argument": Start the analysis process. The number of events to be analyzed may be used as an argument; for example ***analyze 1000*** analyzes 1000 events. If no argument is specified the entire file will be analyzed.

event "argument": Move FORWARD to the event specified and analyze it. For example ***event 237*** moves forward to event 237 and analyzes it.

show "argument": Show the current contents of the argument. For example ***show MTIN*** shows the name of the file/device that *MTIN* is assigned to. If no argument is provided, a listing of possible arguments is displayed.

@filename: Execute the command file specified by *filename*. Command files may include any MOFIA command that may be otherwise issued at the MOFIA command line. If the file extension is not specified the extension *.kcm* is assumed. Command files may also be nested (so that a command file may call another) up to 10 layers deep.

exit: Exit MOFIA

help: Run the MOFIA help facility.

VI.2 Flags

Several flags have been installed in MOFIA. To view these flags type ***show flags*** at the MOFIA command line. MOFIA responds with the following:

```
MOFIA FLAGS:

BPRINT                  = OFF
PHOTO_FLAG              = OFF
SKIM                    = OFF

      DECODE        TRACK         CUTS
DC      OFF           OFF         OFF
PC      OFF           OFF         OFF
SC      OFF           OFF         OFF
MC      OFF           OFF         OFF
```

The set command is used to change the contents of these flags. For example, to turn on the *PHOTO_FLAG* type ***set photo on*** at the MOFIA command line. To turn on the the decoding for the DC detector subsystem type ***set dc on***. To turn on the tracking and/or the cuts a qualifier following the subsystem name is needed. For example to turn on the tracking for the MC subsystem type ***set mc/track on***

VI.3 Namelists

Several namelists have been installed in MOFIA. The following commands are used to access these namelists.

show namelist "argument": Show namelist(s) and description. If no argument is provided all namelists and their descriptions are shown. If an argument is provided the contents of the namelist specified by the argument are shown as well as their respective description and default settings. For example ***show namelist DCCUTS*** shows the contents of the namelist *DCCUTS*.

namelist "argument": Access the namelist specified by the argument, for purposes of checking and/or modifying the current settings. Invoking this command puts the user inside the namelist editor. Once inside the editor, the user may check the current values of the namelist by entering "***?***", exit the editor by entering

"**&**", or change the value of a namelist variable by entering "***variable = value***".

While these commands may be typed in interactively (on the MOFIA command line), it is more convenient to have them in a command file so that the user can execute the appropriate command file (which contains the appropriate settings, flags, and cuts) for the desired analysis. A sample file, ***sample.kcm***, is provided with the code as an example. To execute this file type ***@sample*** at the MOFIA command line. The results of issuing these commands will show up on the screen; the user may also look at the log file ***mofialog.dat***.

VI.4 Functions

Several functions have also been installed in MOFIA The command: ***func "argumen****t*" allows the user to execute these functions. For example executing the command ***func 6*** prints out the drift and proportional chambers geometry files. If no argument is provided a listing of all functions and their descriptions is provided. Table 4 contains a list of all the functions currently available:

Function	Description
1	Turn ON printing for list of wire hits
2	Turn OFF printing for list of wire hits
3	Turn ON printing for tdcunp debugging
4	Turn OFF printing for tdcunp debugging
5	Write HBOOK histograms to file
6	Print DC and PC geometry files
9	Reset all histograms

Table 5. A list of functions available in MOFIA 2.0.

VI.5 Environment variables

Environment variables may also be assigned before executing MOFIA. Two convenient environment variables that come in handy for MOFIA are worth mentioning here: *MTIN* which specifies the input file/device as explained above, and *MHIST* which specifies the directory to which the hbook histograms should be written. If *MHIST* is not specified, the hbook histograms will be written to the current directory in which MOFIA is running. Some environment variables may be best to assign through the *.login* file. If multiple users are sharing the same account and wish to have their own definitions of environment variables, a file of the form *.user* (where *user* stands for the user name) would be handy. Each user would then have to *source .user* before running MOFIA.

When MOFIA is run, a log file is created with the default filename ***mofialog.dat*** and is written to the same directory in which MOFIA is running. The user may wish to change this default and may do so by assigning the environment variable *MOFIALOG* to the desired filename (which may include a directory name if the user wishes to write the log file to a different directory). This is particularly useful when the user wishes to save the log file, since otherwise, this file will be overwritten the next time MOFIA is run.

The environment variable *MOFIA_INIT* provides yet another handy tool. When this variable is assigned to a MOFIA command file (for example, to a file called ***init.kcm***) MOFIA will automatically execute this file upon entering. It is therefore handy for the user to assign this variable to a command file that contains the user customized initializations.

VII. Calibration File Manager (CFM)

E614 will also adopt the Calibration File Manager (CFM) developed for BNL experiment E787. CFM allows the user to associate run numbers with calibration files. Calibration types (plane position corrections, wire position corrections, etc.) may be defined through CFM. For example, *DC_PPC* defines the calibration type for the **D**rift **C**hamber **P**lane **P**osition **C**orrections. CFM allows up to 5 characters to identify a component (Drift Chamber, DC, in this case) and up to 3 characters to identify its attributes (Plane Position Corrections, PPC). The file name for a calibration type has the form *XXXXX_YYY.NNNNN* where *NNNNN* is a 5-digit (version) number called an *indicator*. For example *DC_PPC.00001* is a PPC file name with indicator number equal to 1. CFM is then asked to associate a run number, or a group of runs with indicator 1 for the calibration type *DC_PPC*. When MOFIA is run it checks the run number and obtains the *indicator* number and the file name for each calibration type from CFM. CFM expects all calibration files to reside in the directory defined by the environment variable ***CAL_DB***. To run CFM type ***CFM*** (outside MOFIA). Commands may be issued at the CFM command line to view the contents of CFM or modify and change the existing information. The ***help*** command provides the user with online help. For a more detailed description of CFM, please refer to the E787 document by Morgan Burke.

Several calibration files have been defined in CFM for use with MOFIA 2.0, and are listed below:

Type Number	Type	Description
1:	DC_PPC	Drift chamber UV plane position corrections
2:	DC_WPC	Drift chamber UV wire position corrections
3:	DC_RES	Drift chamber resolution parameters
4:	DC_PRC	Drift chamber plane rotation corrections
5:	DT_GEO	Detector Geometry
6:	FBC1_MAP	Mapping file for first FASTBUS crate
7:	FBC2_MAP	Mapping file for second FASTBUS crate
8:	FBC3_MAP	Mapping file for third FASTBUS crate
9:	DC_PZC	Drift chamber plane Z position corrections
10:	DC_WZC	Drift chamber wire Z position corrections
11:	DC_WRC	Drift chamber wire rotation corrections

Table 6. Calibration files currently installed in the Calibration File Manager (CFM).

VIII. Coding Conventions

The software code for the E614 experiment will be developed over many years by many members of the collaboration. It is therefore necessary to have code that is well written, easy to debug, and easy to understand. To achieve this purpose, the E614 software group adopted a set of coding conventions that all software developers should adhere to. While many of these conventions are qualitative, it is up to the good judgement of the developer to employ these conventions in the best possible way that achieves the intended purpose.

1. Choose clear and descriptive variable names. Clear and descriptive subroutine, function and module names.

It is always easier to read code that has descriptive variable names. This allows the reader to go through the code without having to go back to a comment line to remind themselves what the variable stands for. The same is true for procedure and module names.

2. Provide detailed comments throughout the code.

People with different computer skills and physics backgrounds will be reading the code (graduate students, summer students, etc). It is important to provide enough comments to make the code understandable to everyone. A well-documented program would allow the reader to go through the comments and understand the code without even being familiar with the syntax!

3. Always use implicit none to force declarations of all variables.

This allows the compiler to detect a typing error, provides a documentation of all variables used in a subroutine, and avoids misusing a variable due to FORTRAN defaults for variable names. All procedures should contain an "*IMPLICIT NONE*" statement at the top.

4. Modules should be used instead of common blocks and parameter files.

Modules provide greater flexibility than parameter files and common blocks. Variable initializations inside modules as well as module functions and subroutines allow a nicer organization of the code. The only parameter files and common blocks used in MOFIA 2.0 are the ones inherited from the E787 code, or the ones common to both the analysis and Monte Carlo code.

5. Modules should have a complete documentation of all variables.

Variables declared in modules should always be described inside the module so that the reader would know where to find the documentation, rather than having to search through the code where they are used to find their description.

6. Subroutine, function and module names should always be specified on the end statement.

This is a nice feature that FORTRAN 90 provides, although it was not made mandatory to allow for backward compatibility with FORTRAN 77.

7. Control structure names should be used with all long structures that are not likely to fit on one screen. The same should be done with do loops.

This would improve the readability of the code since long control structures (particularly multiply nested structures) are often hard to follow. This applies to all "do loops", "select case", "do while", and "if" control structures. Again, FORTRAN 90 provides the option of naming these structures, but does not make it mandatory.

8. The "intent" attribute should be used in all functions and subroutines.

This provides a nice way of telling the reader whether a subroutine's argument is an input, an output, or both. It also protects from mistakes such as changing the value of an argument in a subroutine where it is intended to be an input only. FORTRAN 90 provides this option but does not make it mandatory.

9. Interface blocks should be supplied for all external functions/subroutines.

This allows the compiler to check that the function/subroutine is being called properly (that all arguments match in type). If not, this will be detected at compile time, therefore, avoiding possible crashes due to improper usage of a subroutine. FORTRAN 90 does not make this mandatory in all cases to allow for backward compatibility with FORTRAN 77. In MOFIA 2.0 interface blocks are included in the code using a *cpp* include statement and given names with extensions "*.ifb*" (ex. *#include kerror.ifb*).

10.Internal (rather than external) functions and subroutines should be used if they are only intended to be used in the subroutine that calls them.

11.Comments on separate lines and on a statement line are allowed.

12.Keyword arguments should always be used when calling any procedure containing optional arguments (unless all arguments are used).

This avoids the confusion as to which variables are being omitted from the call.

13.Use "STATUS=replace" rather than "STATUS=unknown" when opening a file.

"*STATUS=replace*" is standard FORTRAN 90 that is not implementation dependent. If the file does not exist it is created, otherwise the existing file is deleted and a new file is created. "*STATUS=unknown*", however is implementation dependent.

14.Use whole array operations whenever possible rather than element by element assignments. However, element by element assignment could be more efficiency if several other calculations are needed in a do loop.

FORTRAN 90 supports a large number of array features not present in FORTRAN 77. In particular, whole array operations are a nice feature that makes the code more readable. Element by element assignments through do loops should be avoided whenever possible as long as it's not at the cost of efficiency.

15.The kind attribute should be used to specify all variables.

To make sure that the kind variable is used in a platform independent way the intrensic functions SELECTED_INTEGER_KIND, and SELECTED_REAL_KIND should be used to specify the precision and the range. This is already done in the module ***precisions_mod.f90***, The statement "USE precisions.f90" should be present in every module. The types defined in ***precisions_mod*** are shown in table 7.

Definition	Function Call	Usage
i2	SELECTED_INTEGER_KIND	INTEGER (i2) ::
i4	SELECTED_INTEGER_KIND	INTEGER (i4) ::
i8	SELECTED_INTEGER_KIND	INTEGER (i8) ::
r4	SELECTED_INTEGER_KIND	REAL (r4) ::
r8	SELECTED_INTEGER_KIND	REAL (r8) ::

Table 7. The *kind* attribute definitions in ***precisions_mod***.

16.Variables will not be case sensitive.

17.Lines up to 80 characters are allowed.

Although the compiler allows for up to 132 characters, lines longer than 80 characters are inconvenient for editing and printing.

18.Variable names can have up to a max of 31 characters (standard FORTRAN 90).

19.Make sure that variable names used are not cpp reserved names (ex. UNIX, MIPS, ALPHA, etc).

20.Always use the "ACTION" specifier in the open statement to specify if the file is "READ" "WRITE" or "READWRITE".

This provides a nice documentation for the reader to know how the I/O file is intended to be used, and avoids mistakes of writing output to an input file and vice versa.

21.GOTO statements and statement labels should be avoided as much as possible, in favor of the many nice control structures that FORTRAN 90 provides.

Redirecting the flow in a program is often confusing, and almost always time consuming for the reader. Using a GOTO statement forces the reader to: find the line in the code where the GOTO statement is pointing, figure out what that section of the code is doing and how it relates to where the GOTO statement is, and then go back and find where the GOTO statement that transfered the flow is to continue reading the code! Contrast this to a "cycle" or an "exit" statement in a "do loop" (for example) where you know then and there where the flow is redirected and how it relates to the rest of the code. "Do While" statements are another good example, where you know immediately upon entering the loop the condition for exiting. Short FORMATs should be included in the READ/WRITE statement. Long FORMATs, or FORMAT statements that are used more than once should be done in a separate FORMAT statement, and put as close to where they are being used as possible.

22.All features of FORTRAN 77 identified as obsolescent in FORTRAN 90 should not be used.

These features are marked for deletion in future FORTRAN standards, and have only been kept for backward compatibility with FORTRAN 77. These include a set of features that most programmers agree are not good programming techniques. For a list of obsolescent features see Appendix F in "Upgrading to FORTRAN 90" by Cooper Redwine, for example.

23.Free source format should be used. Indentations of 2 or 3 spaces should be used for statements inside do loops, if structures etc., and nested loops. However, only one statement per line should be used.

This is important for readability purposes. The indentation rule of (2 or 3 spaces) is consistent with emacs/xemacs defaults (i.e. emacs will do it for you if you hit tab on that line, as long as the program name has extension ".f90").

24.Pointer names should always end with an upper case "P".

Example: planeP, wireP.

25.All modules should have "_mod.f90" appended to the name.

Example: chambers_mod.f90, hists_mod.f90.

26.Contents of all modules should be made private by using the "PRIVATE" statement in the module. Variables and procedures that need to be accessed from outside the module should be explicitly declared public.

Making the contents of a module private protects the variables in that module from being accessed and changed by other modules that don't need them. It also prevents the use of module subroutines that are intended for internal use only. In addition, it has the advantage of blocking nested USE statements, and thereby making the module dependencies easier to sort out. For example, suppose that module B uses module A, and module C needs to use both modules A and B. If no PRIVATE statement is present in module B, module C can only contain the statement “USE B”, since module A is already used by module B. If both modules A and B are used explicitly in module C (i.e. through the statements . USE A and USE B) this will cause an error on some compilers (ALPHA) and a warning on others (ABSOFT). This module “nesting” becomes more complicated when many modules are present. The PRIVATE statement blocks nesting so that a module that is needed by another module would have to be used explicitly. In the above example, when a PRIVATE statement is inserted in module B, module C would have to explicitly contain a USE statement for both modules A and B.

27. Modules should be organized as is shown in the sample module sample_mod.f90 below. This module is included in the code for convenience.

```
MODULE sample_mod
! ==========================================================
! Author:
! Date:
! ------------------------------------------------------------------------------------------
! Description:
!.==========================================================
   ! USE statements
   USE
   IMPLICIT NONE
   PRIVATE
   ! Public Procedures
   PUBLIC ::
   ! Global variables
   PUBLIC ::
   ! Include statements
#include ""
   ! Interface blocks
INTERFACE
#include "xx.ifb"
END INTERFACE
   ! Variable declarations
CONTAINS
   SUBROUTINE yy
   END SUBROUTINE yy
END MODULE sample_mod
```

IX. Summary

The file ***f90_charts.ps*** includes a summary of the flow charts, the data structures, and a module listing including subroutines within each module and their PUBLIC/PRIVATE attribute. These pages are provided for convenience so that the user may keep them handy and refer to them when needed.