CDF/DOC/COMP_UPG/CDFR/5294
Version 3.15.0
April 25, 2001

Getting Started with CDF Run 2 Offline

Ken Bloom
University of Michigan

Abstract

We describe parts of the Run 2 offline system for new users. After reading this note, a user should be able to run an `AC++` program, write their own module and include it in an executable, and access some analysis-level data. The instructions are by no means comprehensive, but are complete enough for an average user to start working with the data. This note will be updated as the offline system evolves.

1 Preliminaries

The Run 2 offline system is now available on a large number of CDF computers. At Fermilab, you might want to use `fcdfsgi2`, the new Run 2 analysis machine. You may also have the offline system on a computer at your home institution, or in your trailer office.

The very first thing you must do is `source ~cdfsoft/cdf2.cshrc`, to define UPS commands such as `setup`, which is used to define environment variables and paths for various software packages. You might want to keep this `source` statement in your `.cshrc` file, so you never have to think about it.

Then, various environment and path variables are defined with the command `setup cdfsoft2`. (Some of these variables may conflict with those needed for other software development efforts; only do this in sessions where you are doing CDF code development, and don't put it in any login scripts.) With just those two words, you are set up to use the current production release of the offline software, version 3.15.0 as of this writing. You should use a production release unless you think that you need to be on the absolute cutting edge – the frozen has been "extensively" validated, and is guaranteed to link, run, and give "reasonable" results.

If you prefer a different release, include the version name in your setup command, *e.g.* `setup cdfsoft2 development`. The development release is the most up-to-date version of the code; it is not guaranteed to link or run or do anything sensible. You should only use development if you think you need the absolutely most recent version of the software; it is by its nature very unstable. On `fcdfsgi2` and some other Fermilab computers, including the online machines, a release called "rawhide" is maintained; it is a copy of the development release from the day that it most recently linked. It is guaranteed to link, but won't necessarily run in any sensible way. The rawhide release is only marginally more stable than development; it too can change from day to day. Most users do not need this release. If you are a beginner, stick with a production release, even if someone tells you otherwise.

What if you want to have the stability of a frozen release, but you want to use code that is more recent than the current production release, *i.e.* 3.15.0? Your best option is the latest integration release. This is a frozen release that is created every week, with code that is approved by the librarians for each package. You can setup for the latest integration release with the command `setup -t cdfsoft2` on `fcdfsgi2`.

A useful environment variable that is declared when you `setup cdfsoft2` is `CDFSOFT2_DIR`, which is effectively the top-level directory of the release. In this directory you will find subdirectories for all the packages (about 230 at last count) that are in the release, and under each package name you will find the source code. Each package contains code for some particular task. One subdirectory is `$CDFSOFT2_DIR/bin/<OpSys>-<CompVer>`, which contains some precompiled executables for your use. Here, `<OpSys>` is set to the name and version of your computer's operating system, and `<CompVer>` is the name and version of the compiler, which will vary from platform to platform; on `fcdfsgi2`, for instance, `<OpSys>-<CompVer>` will be `IRIX6-KCC_4_0`. This subdirectory gets added to your Unix path, so you will be able to run these executables just by typing their names (without the directory name in front) on your Unix command line.

One very useful tool is the Web-based code browser which can be accessed from the CDF offline Web page. By default you get the development version of the code, but there are links at the top of the page to frozen releases. What you see is essentially the directory structure in `CDFSOFT2_DIR`, but from there on down everything is extensively cross-referenced, so you can easily hop from a routine name in one file to the source code for that routine. There are also search functions; the "free text" search is most likely to get you to where you want to go, but is all-inclusive and will give plenty of links that you don't want.

As you navigate through the directories of source code, you will find that a given package has two subdirectories of greatest importance. The first of these has the same name as the package itself. This is where you will typically find the "header" files for the classes in that package, with the suffix `.hh`. These files are where classes are declared, with their methods and data members. The other important subdirectory has the name `src`, and this is where the corresponding source code files, with the suffix `.cc`, reside. The roles of these files will be described further in later sections.

2 AC++ Concepts

2.1 The Structure of AC++

`AC++` is the uniform framework that is used throughout CDF to process and analyze our data files. This is ultimately a complicated task, involving file management through the operating system, and the conversion of sequences of bits in those files into variables and data structures that have physics information. `AC++` is meant to take care of all of these low-level tasks for you, so you can concentrate on physics. `AC++` is a family of classes (routines, if you prefer the Fortran nomenclature) which have space set aside for code that you write yourself for data analysis. Thus, you may create as many `AC++` executables as you desire, each with a different piece of user-written code.

Just like the old Analysis Control from Run 1, `AC++` thinks in terms of modules, paths, and streams. The user is allowed to configure all of these elements at run time, which provides a lot of flexibility and power, along with all the advantages and disadvantages that that implies.

When you start an `AC++` program, you are given a command prompt: `AC++>`. This is where you enter all the commands described below. Commands are interpreted with the TCL interpreter, and thus any TCL commands should be valid, should you happen to know any TCL commands. If you don't feel like typing the same commands every time you run your program, you can put them all in a file, and just give the file name on your Unix command line: `myProgram myCommands.tcl`, where `myProgram` is the executable name and `myCommands.tcl` is the file with the commands. This file can also be read in from the `AC++` prompt: `source myCommands.tcl`. Note that abbreviated commands are not allowed, but `AC++` supports tab completion for many commands, module names, and file names.

An `AC++` module is a chunk of code that is used to analyze events. Typically, a single module will carry out a single well-defined task, such as event reconstruction in a particular subdetector. Each `AC++` program can have an arbitrary number of modules of your choice linked into it, and you choose at run time which of them you want to use. The `AC++` command `show module` will tell you which modules are available in the executable that you are running. Each module typically comes with a “begin job” routine that is executed when you start processing data, an “event” routine that is called on every data event, and an “end job” routine that is called when you are done processing data. The “event” routine gets to look at something called the “event record,” which comes from your input file and holds, in some fashion, all the data for a particular event. What the event record really is is discussed later in this note. Any module can write new data to the event record (but can’t modify data that are already there!), so another module that is downstream can look at the output of the earlier module. The rule is that modules are only supposed to communicate with each other by writing to and reading from the event record, so structures like common blocks are not (supposed to be) allowed. It is possible to define a module as a “filter.” If the filtering is enabled, this module will decide which of the input data events will be written to the output file.

Modules can have parameters that you can set at run time. This is done using a “talk-to.” You can invoke the talk-to of a module with the command `talk ThisModule`, where of course `ThisModule` is the name of the module. This will bring you to a menu of commands for that module, and at this point it is helpful to type `help` or `show` to see what commands are available. Some commands will give you a submenu with more commands. Parameters are usually set with a command that has the syntax `ParameterName set newValue`. If you have set the new value successfully, `AC++` typically just responds with a blank line. If you gave an invalid command, you will get an error message that might be informative. The most common mistake is to give the `newValue` in the wrong format. `AC++` is picky about talk-to parameters having the right type, *i.e.* integer, floating point, or boolean.

More commands to control modules can be found with the command `module help`. A module can be completely disabled with `module disable ThisModule`. Modules can also be “cloned” with `module clone ThisModule NewModule`. If you want to run a particular module twice with two different sets of parameters (the cone size for jet clustering is a standard example), you can make a clone of the module at run time, with a new name, and then specify different parameters (*via* the talk-to) for the two copies.

There are two special kinds of modules that have particularly important talk-tos: the input and output modules. The former is responsible for opening the event file that you wish to read, and the latter is responsible for writing out events that may have new data attached, or may have been filtered, to an output file. `DHInput` and `DHOutput` are, as of 3.11.0, the default input and output modules for use with files that are in `ROOT` format, superseding `FileInput` and `FileOutput`. (The “DH” stands for “data handling.”) These modules can be used not just to read or write files on local disk, but also to read and write files, filesets, and datasets in the data catalog. You must talk to these modules to specify the input and output file names. Thus, you will definitely be talking to an input module every time you run an `AC++` program.

When you talk to `DHInput`, you can specify local input files with the syntax `input file InputFile.root`, just as was done for the `FileInput` module. If you are running on a machine with access to the data catalog, you can use the more powerful `include` command to read files that are in the catalog; the use of `include` will be described below. (You can still read in a local disk file with `include`, but you must have a `/` somewhere in your file name, or else the module will think that you are trying to read a file in the data catalog.) The syntax for output modules will be discussed below.

When talking to the input module, you might find a need to create a "hidden list," which is a list of objects in the event record that should be hidden from your code. For instance, if you are running on a file that has been through both Level 3 and production, you may not want to examine the Level 3 results at all. (If Level 3 found a muon, and production found the same muon again, you don't want to think that there are two muons in the event.) To create a hidden list, use the command `hideList set <process name>`, where process name is `L3` for Level 3 and `PROD` for production.

There are other input and output modules besides the standard ones that are meant for use with the different file formats that exist in the online and Level 3 systems. It is possible for multiple input or output modules to be available in a single `AC++` executable, but only one of each can be active at a given time. You can specify which one is active with the command `module input MyInputModule` or `module output MyOutputModule`.

There are also several other modules which are (or should be) included in every `AC++` program; they are called "managers" and handle tasks such as error logging, talking to the calibration and geometry databases, interacting with output histogram files, and so on. These don't interact with the data in the same sort of way.

When you decide at run time which modules you will use, you need to tell `AC++` their names and what order you want them to run in. (Obviously, you would want to run the tracking modules before you run the muon modules.) This is done by defining a "path." By default, `AC++` creates a path with all of the available modules that is given the name `AllPath`. Since each `AC++` executable can have different modules linked in, each executable will have a different definition of `AllPath`. You can use this path, or define your own if you don't want to use all of the modules, or want to run them in a different order. The syntax for creating a path is `path create PathName module1 module2 module3 ...`, where the module names are listed in the order you want them executed. Whenever you define your own path, be sure that `ManagerSequence` is at the beginning of it. A sequence is a collection of modules, and this particular sequence includes all the necessary manager modules mentioned earlier.

Once a path is defined, it needs to be "enabled" for it to be active. When you start `AC++`, there are no paths enabled, so you must enable one via a command with the syntax `path enable PathName`. It is possible to define and enable multiple paths. An event that comes through `AC++` will go through every enabled module in every enabled path. A single module can appear in multiple paths, but it will only be run once on each event no matter how many times it appears. However, if you have cloned the module, each copy of the module runs once. The command `show path` will show you all the paths that are defined, and which ones are enabled. If you run an `AC++` program and it appears to do nothing, you probably forgot to enable a path; this is the most common `AC++` mistake that you will make. (So don't forget!)

Events that are to be sent to an output data file, with any additional information that has been added to the event record during processing, go out through a "stream." Streams must be defined, and each stream must be associated with a path. Since there can be multiple enabled paths, a single `AC++` job can have multiple output files, with events that may have gone through different modules, or modules that have different parameter sets. If different paths have different filter modules, then events could be directed to some output files but not to others. Streams are defined in the talk-to of the output module. This is a two-step process. First, a stream is created and associated with an output file with a command of the syntax `output create MyStream output.root`, where `MyStream` is the (completely arbitrary) stream name and `output.root` is the output file. Then, the stream is associated with a path with a command of the syntax `output path MyStream MyPath`. If you are using the `DHOutput` module for this, there must be either a `.` or a `/` in your file name, or else the module will try to put the file in the data catalog.

2.2 AC++ in Action

Let's try running a simple AC++ program that has already been compiled for you, called AC++Dump, which allows you to dump out the contents of data banks or other "storable objects." You run it by typing its name, AC++Dump, on your Unix command line.

Here is a set of commands that can be used with this program to dump the contents of the EVCL and LRID banks (strictly speaking, the "storable banks," as we shall see), which give some of the most basic information about each event (such as run and event number). In addition, we write the events to an output file, for demonstration purposes.

```
path enable AllPath
talk DHInput
    input file $env(VAL_DATA_DIR)/top175.root
    exit
talk DHOutput
    output create mystream output.root
    output path mystream AllPath
    exit
talk EventDump
    bankList set EVCL LRID
    classList set EVCL_StorableBank LRID_StorableBank
    exit
begin -nev 5
exit
```

Let's review what happens. First, we enable the default path; remember that no paths are enabled when you start. Since EventDump is the only active module in AllPath, and event dumping is all we want to do, that path will do the job. Then we talk to the DHInput module, and specify the input file name. We need to use $env() to make AC++ translate the VAL_DATA_DIR environment variable on fcdfsgi2. The command show input would allow us to verify that we had given the right file name. Then we talk to the DHOutput module to specify an output file. We see the two-step process, creating the stream and associating it with an output file, and then associating the stream with a path. At this point, we could type show output to see a list of streams and output files.

Then, we talk to the module that we are interested in, EventDump. How do you know what commands are available in the talk-to? Type help to see all the options, and consider trying some of them. We invoke the bankList command, setting a list of banks that we want dumped out in hex format. We choose the EVCL and LRID banks, which have event header information. In addition, we invoke the classList command, setting a list of classes that we want dumped in a prettier format. If we want to print particular banks, we must give the full class names, which are EVCL_StorableBank and LRID_StorableBank in this case. This procedure should work for any of the four-letter named banks. You could also add non-bank classes to this list, such as the various analysis-level "storable objects," and you should try doing so on a file of reconstructed events. Other interesting EventDump commands are list, which will give a list of all stored objects in the event, and summary, which will give a more concise list.

After leaving the talk-to, we tell AC++ to begin processing the data, but only to do five events. And that is it; the program should open the data file, and give both a hex dump and a formatted dump of the EVCL and LRID banks. These particular hex dumps are not terribly instructive, but can be generally useful for debugging. If we want to dump five more events, we could give the command `cont -nev 5`.

To verify that you made the output file, you could run the program again and use `output.root` as your input file. It should have five events in it.

3 Data Catalog

Large samples of both real detector data and simulated events are kept in the data catalog. You have direct access to the catalog from `fcdfsgi2` through the `DHInput` module.

Within the catalog, data are organized into datasets, each of which is composed of several filesets, each of which are in turn composed of several files. In the `DHInput` module, you can read any of these, using the `include dataset`, `include fileset`, and `include file` commands.

It is the first of these that is the easiest to use. Dataset ID's typically are six letters long. The first letter indicates the data logging stream that fed this dataset. Data from stream A end up in datasets that start with `a` and so forth. The last letter indicates what has been done to the data; `r` is for raw data, `a` is for reconstructed data, `b` is for data that has been reconstructed on a second pass through the farms, and so forth. The most interesting datasets to you (for the moment) are most likely `aphysr` and `aphysa`.

To analyze the data for a particular run in a particular dataset, give the command

```
include dataset <dataset ID> run=<run#>
selectEvents set run=<run#>
```

Note that you need to specify the run number twice. The first line selects files that contain data from that run, and possibly other runs. The second line ensures that you only see events from the run of interest.

You can specify multiple runs by adding multiple conditions to these lines, *e.g.* `run>=<run#1> run<=<run#2>`. If you wish to specify a particular event, you can do so on the `selectEvents` line, with `event=<event#>.` However, you will have to scan all the files of this to find that particular event, which could take some time.

Simulated data files may be more instructive from a beginner's standpoint, since they contain "data" from a complete detector, and have physics processes that can be reconstructed. A stable set of simulation files are those that were generated as part of the second Mock Data Challenge, which can be found in `/cdf/data02/s5/mc/data/mdc2/sim` on `fcdfsgi2`. Details about the generation can be found on the simulation Web page, but remember that these files are now rather old, and may not be so useful to you. These files are the output of the detector simulation, not of reconstruction, so you will not find any analysis-level data structures there. (Once you have gotten through this note, you may want to try running `ProductionExe` on one of these files to see what happens.) These files are just small samples of our total Mock Data Challenge production. Larger samples are kept in the data catalog. They are split into datasets by physics topic, and different run numbers within each dataset have a different physics process. Again, see the simulation Web page for more details about these files.

Much more information about the use of the `DHInput` and `DHOutput` modules can be found in the `DHMods/doc` area in the code browser.

4 Setting Up Your Programming Environment

4.1 Test Releases

Now that you have tried an `AC++` program that someone else wrote, compiled, and linked, you should try to do these things for yourself. To do so, you must set up a "test release" in a directory where you have access. It is where the Run 2 software system will keep various files it needs to do the compiling and linking; it is essentially a skeletal copy of the full offline release, with some of the same directory names, but none of the 230 packages that are in the real release.

The command to make a test release is

```
newrel -t <base release> <new release name>
```

The `-t` option says that this is a test release, not a full release. You have to give the name of a base release that will be the basis of your test release, and that should be the release that you want to use; it can be any release that is installed on the computer where you are working, such as development or 3.15.0. You also have to give a name for your release; you will end up with a new subdirectory called `<new release name>`. This name is arbitrary, and you can have multiple, independent test releases with different names. Then, `cd` to this new directory to use your test release. Among the subdirectories of your test release are `lib` and `bin`, which is where the library and executable (binary) files that you make will be kept. There is also a `GNUmakefile`, which is used to drive the compiling and linking process.

Your library and binary files will be very large, and while you will want to keep your test release and the source code therein on a disk that is regularly backed up, you will probably want to keep the large files in a scratch area where disk quotas are not enforced. This is easily accomplished if you have a file called `.srtrc` in your home area. The `newrel` script will use this file to make soft links from your `lib`, `bin`, and `tmp` areas to a scratch disk. On `fcdfsgi2`, feel free to copy `~bloom/.srtrc` to your area, and modify it if you think you need to. (You probably don't.)

4.2 Code Repository

But at this point, there is no code for you to work with. All of the offline code, including some sample `AC++` modules, is kept in a CVS repository on `cdfsga`. You have read-only access to this repository from any machine. If you need write access for particular packages, you will first need to arrange permissions with the CDF code management group, and then have a valid Kerberos ticket whenever you modify the repository.

You can check out a "package" of code and work on it with the command

```
addpkg <package name>
```

By default, this command gives you the version of the package that is in the base release you specified. If you are working from development, that means that you always get the latest-greatest version. If you are working from a frozen release, you will get the version from the time the release was frozen. If you want the latest-greatest in that case, use the `-h` option (for "head" of the repository) with this command.

One of the dangers of working with the development release is that it can change from day to day. In particular, if you have made changes in a package that you checked out which are not compatible with changes made in other packages, your programs may fail to link or start crashing. You are responsible for keeping your test release up to date. (By the same token, it is dangerous to add a package from the head of the repository to a test release based on a frozen release.)

To do this, give the command `cvs update` from the top directory of the release, or `cvs update <package name>` if you only want to update a particular package. The CVS system will then sort out which files need updating. It will give you a list of files that it is updating in your release because they changed in the repository (indicated by the letters `U` or `P`), files that you have modified but have not changed in the repository (indicated by `M`), new files that the repository does not know about (indicated by `?`), and files that you have modified in such a way that they are in conflict with the repository (indicated by `C`). If the conflicts are minor, CVS may choose to modify your file to resolve them. You might not want want this to happen. To see what CVS intends to do without it actually happening, insert the `-n` flag between `cvs` and `update`. After updating, your release will be consistent with the repository at that moment.

4.3 Compiling, Linking, and Running

The `CDFSOFT2_DIR` directory shows you what packages are in the base release, and all of those are available to you. However, the package that we are using as an example to get started with is not in any release, and is therefore not visible there. It is called `ExampleMyModule`, and since it is not in a release you will need the `-h` option to check out a copy. This package is a fine enough jumping-off point. You could also make a completely new package in your test release with the `newpkg` script, and then add code to it, but the `GNUmakefile` requires some modifications to make your programs compile and build, so start with `ExampleMyModule`, since it is in proper shape already.

When you add that package, you will be given a new subdirectory called `ExampleMyModule`, and it comes with code for an `AC++` module called `MyModule`, which attempts to make a J/ψ mass plot, and for another module called `ExampleTrackAnalysis`, which makes histograms of some charged-particle tracking quantities. The former module relies on storable banks as analysis-level data structures, a mode of working that should go away, but it does make a pretty plot from Run 1 data. You should not study this code in any detail. We use `MyModule` merely to get started, and then discuss `ExampleTrackAnalysis` in detail below.

To compile the source code and link an executable, type `gmake` from the top-level directory of your test release. This will activate the makefiles in your test release, and should be completed in a few of minutes, at most, for this particular package. Whenever you make any changes to your code, or do a `cvs update`, or if any libraries that you link to have changed (which can happen overnight if you are working from the development release), you should return to this directory and `gmake` again. The system is pretty smart about only recompiling the files that need to be recompiled, so the process will be relatively quick on successive iterations. You can also tell `gmake` to perform only part of the building process. For example, `gmake nobin` will do everything except build the executables. `gmake bin` will then do that last step. In addition, you can tell `gmake` to only do particular steps of the process on particular packages, if you have more than one package in your test release. If you wanted only to build the executables in `ExampleMyModule`, for instance, type `gmake ExampleMyModule.bin`. Some packages have "test" executables that are not made as part of the regular build process; these can be built with the `gmake tbin` command.

Linking an executable takes an unfortunately long amount of time, especially on `fcdfsgi2`. There is a mechanism to get around this that is available for the first time in version 3.12.0: dynamic linking. In this scheme, the code for the packages in your test release are compiled as shared libraries, rather than static libraries, and are linked into your executable at run time, rather than at build time. Thus, you only need to build the executables for your test release *once.* If you make any changes to the code in your test release, you only have to recompile it (`gmake nobin` only), and your changes will be linked into the executable at run time, without you having to relink the executable. To enable dynamic linking, `setenv USESHLIBS 1` *before* you `setup cdfsoft2`, and it will work the next time you try to build your test release. You will save many minutes each time that you change your code!

Now you will have an executable program with the puzzling name `ExampleMyModule_test` in the `bin/<OpSys>-<CompVer>` subdirectory of your test release. Note that this area is in your path, and it's ahead of the similarly-named area in the base release, so you can run this program just by typing its name without the directory name in front of it. You can give commands interactively while running, or you might want to try using the TCL file that has been provided with the package. To do this, type

```
ExampleMyModule_test ExampleMyModule/run.tcl
```

on your Unix command line. (If you are not running on `fcdfsgi2`, you will need to change the input file name in the TCL file.) Most of the commands in the TCL file will look familiar. (The `run.tcl` file will activate only `MyModule`, while `run_track.tcl` will activate only `ExampleTrackAnalysis`. Feel free to try both!)

The program will take a minute or so to run, and when you are done you will have a file called `appexample.hbook` in your area. (Common pitfall – if `appexample.hbook` already exists, your program will crash! Be sure to remove it. The TCL files in the package will do this for you.) This is in fact an `HBOOK` file, so you can open it in `PAW`. Like the old Analysis Control, each module puts its histograms in a different subdirectory of the file; that way, no one has to worry about which module is using what histogram numbers. The subdirectories have the same names as their modules, so `cd` to the `MYMODULE` subdirectory, and there you will find your first Run 2 histograms (filled with Run 1 data).

So what happened here? How did you make the executable? How can you add other modules to it? What happened in the module? How can you do all of this yourself?

5 Assembling Your Executable

How did you end up with an executable called `ExampleMyModule_test`? You can get some idea of how this happened in the file `ExampleMyModule/GNUmakefile`, which defines what happens for this package when you type `gmake`. Makefiles are notoriously inscrutible, but towards the bottom you will see the line `BINS = ExampleMyModule_test`, which means that a binary file (*i.e.* executable) with this name will be created.

In each package, you are allowed to make as many binaries as you want, but, for complicated reasons, only one of them is allowed to have a different name than its source code file. We see such an example here. `ExampleMyModule_test` is defined as your one and only `COMPLEXBIN`, and `BINCCFILES = BuildExampleMyModule_test.cc` indicates that this file is going to be part of it. We will see in a moment how this file is the basis for the executable.

If you are willing to let the executable have the same name as the file with the source code, life is easier. If you want to add executable with the name `MyExe`, all you need to do is have a source file called `MyExe.cc`, and to add the word `MyExe` to the `BINS` line. **Exercise:** Make a copy of `BuildExampleMyModule_test.cc` with a different name, and try making an executable with that name.

But what goes on in the source code in `BuildExampleMyModule_test.cc`? Taking a look inside, we find that this file has the code for several methods of a class called `AppUserBuild`, another puzzling name. A few quick comments on C++ will be helpful. For our purposes, a C++ class is a collection of variables, and functions ("methods," in object-oriented parlance) that have access to these variables, stored together somewhere in memory. Somewhere else in the offline system, the

AppUserBuild class is declared, with a given set of method names, and the input arguments to the methods. There are other pieces of code out there that will try to call these methods with these arguments at particular times during the execution of your program. However, just what happens when these AppUserBuild methods are called has NOT been specified – that is left for you to do! And it is in the methods of AppUserBuild that you get to define what modules will be included in your executable. The "beauty" of this is that some other routines elsewhere in the AC++ structure can call methods of this class which will always have the same name, but each user gets to define what those functions do for their own executable.

The class declaration is done in a header file which must be "included" in the file with the actual source code, so that the source code file itself knows what methods it is supposed to be defining. In this particular file, BuildExampleMyModule_test.cc, you can see the line #include "Framework/APPUserBuild.hh", which points to the header file in question.

The bottom line of this discussion is that the AppUserBuild class is the moral equivalent of main for an AC++ program. This is the spot where you get to define what modules go into your particular executable. This particular file, BuildExampleMyModule_test.cc, gives a perfectly fine example of how to include modules. The action all takes place in the constructor for the class. The first thing that happens is that a bunch of modules are added via the routine addCDFrequiredModules. These include the input and output modules and the manager modules, so you always want to have this line. The routine addAllStorableObjects makes your executable aware of the various data structures that may be in your input file. Then, three other modules are added with a function called add – HepHbookManager, MyModule, and ExampleTrackAnalysis. Just what the syntax means requires another lecture on C++, which we will skip for simplicity. But if you want to add an additional module to this executable, just follow the same syntax as for the modules that are already there.

How does AppUserBuild know what these modules are? They are defined in their header files, so those header files must also be included. You can see that in the set of lines under the "Collaborating Class Headers" comment. In Fortran, you used include files to make different subroutines aware of external variables in common blocks. In C++, you use header files to make different classes aware of external classes and their functions. So, to add an additional module to this executable, make sure that you have the header file listed in the include lines, and then just follow the same syntax as for the other modules. Actually, there is one more finer point – the GNUmakefile may not know where to look for the compiled code for these other modules when you try to link everything; it has to know what library of compiled code must be searched. In that file, you will find a bunch of lines that look like override LINK_FrameMods += ExampleMyModule. (Note that this is the GNUmakefile for ExampleMyModule, not the one at the top of your test release.) You may need to add more such lines, in which FrameMods gets replaced by the name of the library containing the module that you want to link in. Typically, this library name is the same as the name of the package containing the code, but there are a few exceptions. FrameMods happens to be one of them; this package makes several libraries, such as FrameMods_hbook, FrameMods_root and FrameMods_dump. We hope that there will soon be a replacement for this mysterious mechanism.

There are a lot of AC++ modules that have already been written for possible inclusion in executables. The offline group has been reasonably good about putting them in packages that have the word Mods in their name, like TrackingMods and MuonMods. Use the code browser to help find your way around.

Exercise: BuildExampleMyModule_test.cc includes some comments that suggest how to change your output from an HBOOK file to a ROOT file. See if you can follow them; this will give you a chance to try swapping which modules are used. Also, try adding a completely new module to this executable, such as the event dumping module. When you are done making these changes to

BuildExampleMyModule_test.cc (and the GNUmakefile!), type gmake to recompile and relink the code, and run the resulting executable to see if your changes worked.

6 Inside ExampleTrackAnalysis: How to Make a Histogram

Let's take a look at what is going on inside ExampleTrackAnalysis, which serves as a simple example of making histograms with analysis-level data structures. (Remember, this module is linked into ExampleMyModule_test, and you can activate this module with run_track.tcl.) Now, every AC++ module is itself (an instance of) a C++ class that has several methods associated with it. It ultimately inherits from a base class, called AppModule, that represents a generic module. What happens inside the executable is that AC++ maintains a list of the modules to be run (provided by you in AppUserBuild!), and then it calls the right method of the class at the right time for each of the modules on the list. Here, the "right time" means at the start of the job, or every time a new event is read, or any other point in the progress of your analysis job where a module should perform a particular task. So as long as each module has a uniform set of method names, this will all work very simply, even if you don't understand this explanation.

What are these uniformly-named methods? You can see the entire list in the header file for AppModule, which is in the Framework package. When you make your own module class that inherits from AppModule, you are allowed to override any of these methods to make them do what you want. The ones you will most likely want to override are beginJob, which is called when you start your analysis program, and event, which is called for each new event in the data file. There are also beginRun, endRun, endJob, and abortJob methods, which are called at the times implied by their names. To override one of these methods, you must declare it in the header file for your own module, and then you must define what it does in the source code file.

Let us examine ExampleTrackAnalysis.hh, which is the header file for this class, defining its methods and data members. First, we see that the class ExampleTrackAnalysis inherits from a base class called HepHistModule. HepHistModule ultimately inherits from AppModule, so this class does also. Any module that will be producing histograms must inherit from this base class. In frozen releases previous to 3.5.0, HepHistModule had a non-regular set of method names; these are still available for backwards compatibility, but should not be used.

Note that we are using the HepTuple histogramming package in this module. HepTuple was written to be an interface to either HBOOK or ROOT, and it is straightforward to change between the two. (In fact, this is what you did in the last exercise – just switch from the HepHbookManager to the HepRootManager in your AppUserBuild class.) In the long run, many users will probably want to use ROOT directly, without the HepTuple interface; we will document this in the future.

The first method declared for the ExampleTrackAnalysis class is the constructor, which is in fact what got called in AppUserBuild. You will note that it comes with default arguments for the name of the module that the user will see in AC++, and its description. In AppUserBuild, where the constructor is called, these defaults can be overridden, but aren't in our example. There are also the beginJob and event methods, where most of the work will be done. The other methods are not of great importance for getting started. However, if there were any other methods of AppModule that you wanted to override, *e.g.* if you wanted to print out some end-of-job summary information in endJob, you would need to declare them in this header file.

At the bottom, there are several data members that are declared private. This means that other classes don't have access to them, but since they are defined in the header file, they are defined globally within this class, so every method has access to them. Some are declared as HepHist1D*, which means that they are pointers (currently uninitialized) to one-dimensional histograms, and there is also a HepNtuple*, a pointer to an ntuple. If you wanted to add additional histograms to this module, you would declare them similarly.

Now let's move on to the guts of the code in ExampleTrackAnalysis.cc. Here there is a chunk of code for each method that is defined in the header file. In some cases, there is not much to it – nothing is done in the destructor, and in the constructor we merely initialize the base class with its constructor (always be sure to do that in any module you write) and initialize the talk-to (discussed below).

Obviously, a lot more action is going on in beginJob, where histograms are booked, and event, where histograms are filled. If you want to book a histogram, you need to have memory allocated in the right place so that the histogram ends up in the right file, and so forth. This is handled through one of the "manager" modules, which you make contact with through the HepFileManager class. The beginJob method has a line that gives you access to it. Then, shortly below, you see the statements that book an ntuple and histograms, associating them with the pointers that had been declared in the header file. This is done through methods of the file manager. These pointers now point to something that actually exists, so you can work with these objects. Note the syntax for booking histograms, which is a little different from what we are used to in HBOOK. The order of arguments is the title, the number of bins, the lower and upper edges, and the ID number. The ID number is in fact an optional argument – if you don't specify one, HepTuple will choose one for you. (Optional arguments of a method are always placed at the end of the list in C++.) You can now use this histogram pointer variable (not the ID number) to refer to your histogram elsewhere in the module. The syntax needed to create a 2D histogram (of type HepHist2D) is quite similar. As you can see, making ntuples with the HepTuple package is unfortunately complicated.

Finally, we get to the event method. This method gets called for each new event in the data stream. Here we loop over reconstructed tracks (or CdfTrack objects, really), and fill the ntuple and histograms with some of their parameters. We will explain below how one goes about accessing data, but you can get a schematic idea of what is going on – methods of various objects are being called to get the information that you want. But at the end, we call the accumulate method of the histogram pointers, and the capture and storeCapturedData methods of the ntuple pointer, and that is what captures the data into the histograms and ntuple. This is all you have to do. AC++ will worry about writing your histograms out to a file when you are done.

Exercise: Add more histograms to this module.

7 Advanced Module Features

The ExampleTrackAnalysis module also has a talk-to, and it can serve as filtering module. Here, we explain how to incorporate these features into your own module.

7.1 Writing a Talk-To

If you want to have parameters in your module that a user can set at run time, it is pretty easy to write a talk-to. (If you have ever done this for Run 1 code, you will find it much easier in Run 2!). Data structures that hold the talk-to information are kept in classes that start with the name AbsParm, and can be found in the Framework package. If you want to define a talk-to variable of an arbitrary type, you can use the AbsParmGeneral class. When you declare the variable, use the syntax AbsParmGeneral<type> myParm, where type is the data type. There are some pre-written classes that have particular variable types already set. For instance, AbsParmDouble will hold a double-precision variable that can be set with a talk-to, and AbsParmBool will do the same with a boolean variable.

As an example, imagine that you wanted to have a p_T cut that is set by the user. First, declare the variable in the `private:` section of the header file of your module, like so:

```
        AbsParmGeneral<float> _ptCut;
```

Then, three things should be done in the constructor method of your module. First, you must initialize it, by telling it the name that will appear in the talk-to menu in `AC++`, telling it what module it belongs to (this module itself, obviously, which you refer to with the variable `this`), and its initial value. Second, you can (and probably should) add a description of the variable, which will be printed when the user types `help` in the talk-to. Third, you must append it to the modules list of talk-to commands, which is done with the `append` method of the command list, which you access with the function `commands()`. Here is what a constructor would look like with these three steps in place. Note that the first step is done as part of the initialization of the constructor.

```
ExampleTrackAnalysis::ExampleTrackAnalysis(
    const char* const theName,
    const char* const theDescription )
    : HepHistModule( theName, theDescription ),
      _ptCut("ptCut",this,0.0)
{
  _ptCut.addDescription("\t Save event if there exists a \n
                        \t track with pT above this value.");
  commands()->append(&_ptCut);
}
```

Now the user will be able to set the value of `_ptCut` in the talk-to. Your code will want to access the value later on. This is done via the `value` method of the `AbsParm` classes:

```
        float ptCut_value = _ptCut.value();
```

Now you can use `ptCut_value` as you would any other floating-point number.

7.2 Filter Modules

If you wish to use your module as a filter, it must inherit from `AppFilterModule`, which inherits from `AppModule`. As it happens, `HepHistModule` inherits from `AppFilterModule`, so any histogramming module can also be a filter module.

In the `event` method, you should construct a boolean variable that indicates whether the event should be kept or not. For safety purposes, it is probably best to initialize it as false: `bool filter_pass = false`. At the end of `event`, you tell this module whether or not the event passed the filter:

```
        this->setPassed(filter_pass);
```

That is all that you have to do in the code. At run time, you must enable this module as a filtering module. At the `AC++>` prompt, give the command `filter MyModule on` to enable it. You can also use a module as an anti-filter; the command `filter MyModule veto` will cause `AC++` to save all events that failed the filter.

8 The Event-Data Model

The "event-data model," or EDM, describes how data within each event are stored within memory, and on disk. Over the past two years, there has been much effort on CDF to build a better EDM, one that takes full advantage of C++, requires less detailed knowledge of the data format on the part of the average user, and has safety features that keeps data from getting accidentally overwritten in memory.

You may not have realized that CDF ever had an EDM, but there has always been one. In Run 1, the EDM was implemented through `YBOS`. From anywhere in your executable, you had full access to a very large, single-indexed array which held all the data for each event. For instance, each track would have a number of data elements associated with it, which would appear, in a specified order, at some location in the big array. These contiguous collections of array elements were called "banks." When you reached a new event, this array was refreshed from disk. To find a particular datum, such as the momentum of a particular track, you called a function that gave you an index into the array that pointed you to the start of the track bank. Then you had to know how far down from the start of the bank to find the momentum. If there were many tracks, they could only be distinguished by the position of their banks in the array. This array could be overwritten by any routine. If the event were to be written out to the data file, the array elements, in their current state, would be put on disk. Among the disadvantages of this EDM are that the user had to know the organization of each bank, and banks could be modified by one routine without a later routine ever knowing about it.

The EDM for Run 2 looks a little different. Here we describe the various concepts that you need to know, or might want to know, to find your way around your data.

8.1 Event Record

The `event` method of every `AC++` module has the input argument `AbsEvent* anEvent`. This pointer to the "event record" is your gateway to any data that are in this particular event. (Actually, `anEvent` is globally visible from any class; if you include `AbsEnv/AbsEnv.hh`, you can access it through `AbsEnv::instance()->theEvent()`, but this is not encouraged.) The event record pointer is refreshed for each new event in a data file.

You can imagine that the event record is a big container holding all sorts of different data structures. These data structures are defined by C++ classes, and the name of the structure in the event record is the same as the name of the class. Since each one of these structures in the record is an "object," in the C++ sense, we will refer to them that way. An object that is in the event record, and thus can be read from and written to a file, is required to inherit from the `StorableObject` class.

Each of the objects in the event record is *unique*. Associated with each is a unique number that is assigned when the object is added to the event record. Even if a particular object is removed from the record, its number cannot be reused. But you never have to know what this number is – these objects can also be identified by C++ class name (what you will probably do most frequently), by a text string that is associated with each object, or by a label that identifies what process (e.g. Level 3, production) created the object. Each of the objects in the event record are also *non-modifiable*. If you feel you have the need to modify objects in the event record, you should make your own copy, modify that copy, and put the new objects into the event record.

Creating a storable object and putting into the event record is not that difficult, but the average user won't be doing it that often; typically, the production executable puts objects into the record, and the end user reads them. Reading an object from the event record is therefore something you must do in your AC++ module if you want to look at the data, so we describe how to do it here.

8.2 Event-Record Iterators and Handles

There are two steps required to examine a particular object in the event record. The first step is to actually locate your object within the event record, a container that can hold many different objects. This is done by creating an "iterator" (the C++ equivalent of a looping index) that can be used to step through the event record. Why might one want to do such stepping? While every object in the event record is unique, there may be several objects of each class. For instance, there might be several collections of tracks in the event record, each created by a different tracking algorithm. The iterator will allow you to step through the record and access each collection in turn. Of course, some of the objects in the record will be one-of-a-kind, like an EVCL bank.

Imagine that you are looking for all objects of type MyObject in the event record. You make the iterator, which we will call obj_iter, like so:

```
EventRecord::ConstIterator obj_iter(anEvent, "MyObject");
```

This line essentially says, "Make me an iterator that steps through all instances of MyObject in anEvent." It's a ConstIterator because objects that you are getting are constant, *i.e.* non-modifiable, as all objects in the event record must be.

"All instances of MyObject in anEvent" really means all instances that are visible to your module, *i.e.* those that are not on the hidden list. If your file has been processed multiple times, *e.g.* by both Level 3 and production, you could have multiple copies of the the same object in the event record, and you will see both of them, unless you specify that one or more processes be on the hidden list when you talk to the input module.

The variable obj_iter now is associated with the first MyObject that is found in the event. But how do you know if there is in fact such a MyObject in the event? After you create the iterator, you can check if it is valid, with the method obj_iter.is_valid(). If the method returns a value of true, then the iterator is useable. If there is more than one such object, you can increment the iterator, and it will point to the next one: ++obj_iter. (The iterator is smart enough to not point to objects in the event record that are not of type MyObject.) Before you use the iterator, you should *always* check that it is valid; once you have stepped through every MyObject in the event, it won't be.

The above method of creating the iterator is not the only method. You can also create the iterator by specifying any of the identifiers of a storable object, such as the description text string or the creating process. Better still, you can create logical combinations of these "selectors" with the usual &&, ||, and ! operators, which could allow you to specify both an object name and a description string.

The second step is to go from this iterator to something that you can use to actually access MyObject. This is done by creating a "handle" that is used to grab the object. A handle is very much like a pointer, but has some code underneath it which handles memory management. (This is an important issue when creating storable objects.) A handle (in this case, a ConstHandle, since this object is coming out of the event record) for MyObject is made like so:

```
ConstHandle<MyObject> obj_hndl(obj_iter);
```

This line says, "Make a handle for MyObject out of obj_iter. Once you have the handle, you can treat it like a pointer to MyObject, and call methods of MyObject in the usual way, with the dereferencing operator (better known as ->):

```
int myObjectSize = obj_hndl->size();
```

(If `MyObject` has a method `size()`, of course.) If you prefer the usual method operator, better known as `.`, you can convert your handle into a reference:

```
const MyObject& object = *obj_hndl;
int myObjectSize = obj.size();
```

Note for advanced C++ users: A potentially useful feature of handles is that you can use them to take advantage of class inheritance. Suppose that `MyObject` inherits from a class called `MyObjectParent`. Then, you can make a handle out of the event record that will act like a pointer to a `MyObjectParent` instead of a `MyObject`:

```
ConstHandle<MyObjectParent> parentobj_hndl(obj_iter);
```

This technique is used in programs such as `Edm_ObjectLister`, which gives you a list of all the objects in each event in a data file. The program iterates through every object in the event record, and makes a `ConstHandle<StorableObject>` for each of them. It can then call methods of `StorableObject`, and never has to know about the specific features of each object in the event. Of course, if you do use this technique, you cannot call the methods of the child class `MyObject`.

8.3 Links

Analysis-level objects are often built from a number of lower-level objects. For instance, a muon has a track associated with it. The muons are kept inside a different storable object than the tracks. How can you find your way from one storable object to another?

The EDM provides a means for this, called a link. Links behave like handles, except that they can also be added to the event record as part of a storable object. Each muon carries a link to the track associated with it. Thus, when you have pulled an muon out of the event record, you can immediately get another handle (the link) for some other object in the record (the track). You can then use the link as if it is a pointer to the object, just like you would do with the handle. The syntax of the class name is even the same – a link to a `MyObject` is of type `Link<MyObject>`. Examples of links can be found in the discussion of electrons and muons below.

9 Using CDF Data Structures

After the raw data have been run through the production executable, there will be a standard set of objects in the event record. Here, we give some examples of how to use them.

While all of these structures can be found in the event record using the techniques of the previous section, some of them come with shortcuts that allow you to condense the two steps (making the iterator, then making the handle) into one. The examples given below have, wherever possible, taken the necessary steps so that the `.` operator can be used, rather than `->`; other pieces of example code may not take these steps, and therefore not use the same syntax.

Of course, each class will have its own methods, which can be found in the header file for the class. Be sure to put `#include` statements in your code if you want to use these methods.

9.1 Storable Banks

The `StorableBank` classes are used to hold raw detector data. It is unlikely that you will need to access any of these to do a data analysis, but you will if you want to do low-level detector studies. Any storable bank can be obtained from the event record by the means described above.

All of these classes have names of the form `XXXX_StorableBank`, where `XXXX` is the four-letter name of the bank, just like in YBOS. And the data are structured in the same way as YBOS banks: a one-dimensional array.

Any element of any bank can be obtained if you know its position in the array. However, many banks have functions that will allow you to get particular elements without knowledge of the position. These functions will naturally differ from bank to bank, so look at the header file for the appropriate class. Unfortunately, the `StorableBank` classes have very challenging header files, and you may want to sit down with someone who is an expert with that particular bank.

Here is a quick example of how to access a bank, using the tools described in the previous section. The `XXXX_StorableBank` and its methods are completely fictional, created for the purpose of this example.

```
EventRecord::ConstIterator XXXX_iter(anEvent,"XXXX_StorableBank");

while (XXXX_iter.is_valid()) {
   ConstHandle<XXXX_StorableBank> XXXX_hndl(XXXX_iter);
   const XXXX_StorableBank& XXXX_bank = *XXXX_hndl;
   int version = XXXX.version();
}
```

9.2 Tracks

Tracks in CDF can be reconstructed with a great variety of tracking algorithms. However, all of these track types are represented in the event record by a single class, `CdfTrack`. The event record can contain not just single tracks, but "collections" of tracks, each of which has tracks that have been found with a particular set of algorithms, such as COT-only algorithms, or algorithms that involve the silicon detectors. Thus, it is possible to pull an entire collection out of the event record.

The storable objects, are really "wrappers" around the actual collections, where the tracks really are. (The wrapper is needed to satisfy some requirements of the EDM; the actual collection is not directly storable.) Thus, once you have pulled the storable object out of the event record, you will need to pull the actual collection of tracks out of the storable object, which constitutes a third step beyond the usual two-step process. The CDF wrapper objects typically provide a `contents()` method that gives you access to the actual collection.

For a data analysis, you will want the "best," or "default," set of tracks, which would have the greatest amount of information possible (both COT and silicon hits, if both are available), without any duplication. This set of tracks will be distributed across the different collections that may be in the event record. Thus, it is also possible to create a track "view," which consists of pointers to tracks that are in different collections.

Here is how to get a "default" set of tracks out of the event; the default tracks are defined here as those which do not have any other tracks derived from them. For instance, if a COT track has been found, and then silicon hits are added to it to make a new track, the latter will be in the default set and the former will not. In the tracking code, a shortcut is made to condense the usual two-step process into one, so that the event-record iterator is hidden from you. In this code fragment, we loop over the default tracks, and extract various quantities of interest. Additional functions can be found in the `CdfTrack` header file, and in the header files for the classes it inherits from, such as `SimpleReconstructedTrack`, in the `TrackingObjects` package.

```
CdfTrackView_h view_hndl;   //This will be the handle for the view.
```

```
  //This line does the two-step process, which makes view_hndl useful.
  //Note also we check that this came out OK.

CdfTrackView::Error trackStatus = CdfTrackView::defTracks(view_hndl);
if (trackStatus == CdfTrackView::OK) {

    //Pull the actual collection of tracks view out of the handle.

    const CdfTrackView::CollType & tracks = view_hndl->contents();

    //Now we can actually iterate over the tracks, by constructing
    //a CdfTrackView iterator.

    for (CdfTrackView::const_iterator itrack=tracks.begin();
      itrack!=tracks.end(); ++itrack) {

      //This double-dereference looks strange, but things get easier!

      const CdfTrack & trk = **itrack;

      //Here are some functions you may want to use:
      float pt = trk.pt();
      float phi0 = trk.phi0();
      float d0 = trk.d0();
      float z0 = trk.z0();
      float eta = trk.pseudoRapidity();
      HepVector p = trk.momentum();
      int algorithmNumber = trk.algorithm().value();
      int numCOTHits = trk.numCTHits();
      int numSIHits = trk.numSIHits();

      //You can also look at properties of the parent of this
      //track, if the parent exists.

      float parentPt;
      if (trk.parent().is_nonnull()) parentPt = trk.parent()->pt();
    }
}
```

Why do we have to do a double-dereference to get to something that is of type `CdfTrack`? This is because the `CdfTrackView` is a collection of pointers to tracks. The first dereference is needed to convert the iterator to the pointer, and the second is needed to convert the pointer to the `CdfTrack`. This double-dereferencing scheme will be common to any "view" of reconstructed objects.

9.3 Calorimetry

There are three kinds of calorimetry data that might appear in an analysis. The first is the energy in each tower in the entire detector. In the calorimetry code, each tower is represented by a `CalTower` object, and the collection of all towers is stored in the event record as a container called `CalData`.

CalData is the equivalent of the Run 1 TOWE bank. The second kind of data is jet objects, which are formed through the clustering of towers. Each jet is represented by a CdfJet object, and the entire collection is stored in the event record as a container called CdfJetColl. As in the track case, there can be many collections of jets, because there can be many jet-clustering algorithms. You may need to specify which collection you want. Finally, there is the missing transverse energy ($\not\!\!E_T$), which is kept in the CdfMet object in the event record.

Here are some examples of how to access these data, with a few basic methods for each object. The complete list of methods can be found in the header files for these classes in the CalorObjects, Jet, and Met packages.

```
//First make a handle for the data.  The two-step process is handled
//with just one line.

CalData_ch calData_hndl; //ConstHandle for the CalData container
CalData::Error calStatus = CalData::find(calData_hndl); //Two-step process
if (calStatus == CalData::OK) {

    for (CalData::ConstIterator itower = calData_hndl->begin();
         itower != calData_hndl->end(); ++itower) {

        const CalTower& tower = *itower;
        float energy = tower.energy();
        float emEnergy = tower.emEnergy();
        float hadEnergy = tower.hadEnergy();
        int iEta = tower.iEta();
        int iPhi = tower.iPhi();
    }
}

//Now get the CdfJets.  We specify the "description" to get jets found
//with a particular algorithm.  Here, we choose the results of the
//standard JetCluModule, which uses a cone of R = 0.7.  After that,
//this looks very similar to the tracking example.

CdfJetColl_ch jetColl_hndl;
std::string description = "JetCluModule"; //not necessarily what you want
CdfJetColl::Error jetStatus = CdfJetColl::find(jetColl_hndl,description);
if (jetStatus == CdfJetColl::OK) {

    const JetList& jets = jetColl_hndl->contents();
    for (ConstJetIter ijet = jets.begin(); ijet != jets.end(); ++ijet) {

        const CdfJet& jet = *ijet;
        float jetEt = jet.et();
        float jetEta = jet.eta();
        float jetPhi = jet.phi();
        HepLorentzVector jetP = jet.fourMomentum();
    }
}

//Finally, we get the missing energy.

CdfMet_ch mEt_hndl;
CdfMet::Error metStatus = CdfMet::find(mEt_hndl);
if (metStatus == CdfMet::OK) {

    const CdfMet& mEt = *mEt_hndl;
    float missingEt = mEt.met();
    float sumEt = mEt.etSum();
    float xMEt = mEt.exSum();
    float yMEt = mEt.eySum();

}
```

9.4 Photons and Electrons

In CDF, photons and electrons are both treated as "electromagnetic objects," since both are defined by their deposits in the EM calorimeter. A photon is an EM energy cluster without an associated track, while an electron typically has an associated track, except in the far-forward plug where track reconstruction is inefficient. Therefore, both of these physics objects are represented by the `CdfEmObject` class. Like the tracks, you can access these objects through a `CdfEmObjectView`. Here is an example of how to access this view, and how to access information about electrons and photons. See the header files for `CdfEmObject` and `EmCluster` in the `ElectronObjects` package for the complete list of methods.

```
CdfEmObjectView::handle emObjView_hndl;
CdfEmObjectView::Error eleStatus =
   CdfEmObjectView::defEmObjects(emObjView_hndl);
if (eleStatus == CdfEmObjectColl::OK) {

   const CdfEmObjectView::collection_type emObjs = emObjView_hndl->contents();

   for (CdfEmObjectView::const_iterator
          emIter = emObjs.begin(); emIter != emObjs.end(); ++emIter) {

      const CdfEmObject& emObject = **emIter;
      const EmCluster& emCluster = *(emObject.getEmCluster());

      HepLorentzVector emP4 = emCluster.fourMomentum();
      double hadOverEm = emCluster.hadEm();
      double eta = emCluster.emEtaEvent();
      double detectorEta = emCluster.emEtaDetector();
      double phi = emCluster.emPhi();
      double emEt = emCluster.emEt();

      const CdfTrackView& tracks = emObject.matchingTracks();
      Link<CdfTrack> maxPtTrack = emObject.maxPtTrack();
      double trackPt;
      if (maxPtTrack.is_nonnull()) trackPt = maxPtTrack->pt();
   }
}
```

9.5 Muons

A working version of the CdfMuon class is now available. Just like many of the other physics objects, the CdfMuons live in a CdfMuonColl in the event record. However, it is better to make a CdfMuonView, which behaves much like the CdfTrackView.

The CdfMuon itself holds links to one or more MuonStub objects, and to a CdfTrack. It also carries information about the quality of each track-stub match, in the form of a MuonMatchData object, and calorimeter and isolation quantities (currently unfilled). Here is an example of how to access muons and some of this information; see the header files and documentation in the MuonObjects package for more details.

```
CdfMuonView_h muonView_hndl;
bool muonStatus = CdfMuonView::allMuons(muonView_hndl);
if (muonStatus == true) {

   for (CdfMuonView::const_iterator muIter =
          muonView_hndl->contents().begin();
        muIter != muonView_hndl->contents().end(); ++muIter) {

      const CdfMuon& muon = **muIter;

      HepLorentzVector muP4 = muon.fourMomentum();

      //Be sure to check that the muon has a stub in a particular
      //system before trying to access data for that system!
      //cmu can be replaced by cmp, cmx, cmu.

      if (muon.hasCmu()) {
         const MuonMatchData& cmuMatch = muon.cmu();
         float drphi = cmuMatch.drphi();
         const MuonStub* cmuStub = muon.stub(CdfMuon::CMU);
      }

      ConstLink<CdfTrack> track = muon.bestTrack();
      double pT = track->pt();
   }
}
```

10 Acknowledgements

Colleagues and friends at Michigan and Chicago have given helpful feedback, starting from the beta version of this document. Many examples have been borrowed from David Dagenhart, Fedor

Ratnikov, Pierre Savard, Liz Sexton-Kennedy, and Rick Snider. Andy Beretvas, Frank Chlebana, Isi Dunietz, Rob Harris, Stephen Haywood, Young-Kee Kim, Art Kreymer, Marjorie Shapiro, and Tony Vaiciulis have also made useful comments and suggestions. The system described in this note only works because of the years of effort by members of the offline group.

A Quick Guide to Commands

A.1 Preliminaries

`source ~cdfsoft/cdf2.cshrc`	Always do this first!
`setenv USESHLIBS 1`	Enable dynamic linking
`setup cdfsoft2 [<release name>]`	Always do this second!

A.2 `AC++` Commands

`<program> <TCL file>`	Run `AC++` program with command file
`help`	Print available help information
`show`	Show status
`show module`	List available modules
`show path`	List available paths
`module input <input module>`	Specify the input module
`module output <output module>`	Specify the output module
`module disable <module name>`	Disable a module
`talk <module name>`	Enter the talk-to for a module
`<parameter name> set <value>`	Syntax for setting parameters
`filter <module name> <on/off/veto>`	Set filter status
`input file <file name>`	Specify input file in input module
`include dataset "<dataset ID>" run=<run#>`	Specify files in data catalog
`selectEvents set run=<run#>`	Specify run number in those files
`hideList set <process name>`	Create a hidden list
`output create <stream name> <file name>`	Create an output stream
`output path <stream name> <path name>`	Define path for output stream
`path create <path name> <mod1> [<mod2> ...]`	Create a path
`path enable <path name>`	Enable a path
`begin [-nev <number of events>]`	Start analyzing events
`cont [-nev <number of events>]`	Continue analyzing events
`exit`	Exit `AC++`

A.3 Test Releases

`newrel -t <base release> <new release name>`	Create test release
`addpkg [-h] <package name>`	Add package to test release
`cvs update [<package name>]`	Update development-based release
`cvs update -D "<date>" -Pd [<package name>]`	Update rawhide-based release
`gmake`	Complete build of test release
`gmake nobin`	Build everything except executables
`gmake bin`	Build only executables; don't recompile
`gmake <package name>.all`	Do complete build of one package
`gmake <package name>.<action>`	Partial build of one package