The ADAM Concept

David Groep, Wim Heubers, Eddy Jans, Ronald Starink, Jos Steijger

January 25, 1996
The ADAM Concept

A tutorial introduction to the use of the AmPS Data Analysis Method

Nationaal Instituut voor Kernfysica en Hoge Energie Fysica 1995
ADAM development group, adam@nikhef.knhef.nl
NIKHEF Internal report 96-036

The AmPS Data Analysis Method was designed by David Groep, Wim Heubers, Eddy Jans, Ronald Starink and Jos Steijger.

UNIX is a trademark of AT&T Bell Laboratories
SunOS and Solaris are trademarks of SUN Microsystems Inc.
## Contents

1 The structure of ADAM .................................................. 1
   1.1 Kernel and modules ............................................. 1
   1.2 Types of modules ............................................... 3
   1.3 Exchanging information ........................................ 4

2 Communicating Information ........................................... 7
   2.1 A common market – the Public Variable Space ................. 7
      2.1.1 Naming of PVS variables ................................. 8
      2.1.2 Types of variables ....................................... 9
      2.1.3 Constants and event dependent data ...................... 9
      2.1.4 Querying the type of a variable ......................... 10
      2.1.5 Database .................................................. 10
   2.2 Using event fragments .......................................... 12
   2.3 Information Exchange .......................................... 14
      2.3.1 Supplying information .................................... 15
      2.3.2 Using information ........................................ 16
   2.4 Target ................................................................ 16

3 The analysis phase ...................................................... 18
   3.1 What kind of events should be analysed ...................... 18
      3.1.1 Rejecting events and statistics .......................... 19
   3.2 What should be calculated ...................................... 19

4 Miscellaneous .......................................................... 21
   4.1 Initialisation and termination .................................. 21
   4.2 Compiling an ACI loadable module ............................. 22
   4.3 Logging messages ................................................ 23
   4.4 Errors and warnings ............................................ 24

5 Development tutorial .................................................. 26
   5.1 The fake event ................................................... 26
   5.2 Our design specifications ...................................... 27
   5.3 The database .................................................... 27
   5.4 The initialisation and termination section ..................... 27
      5.4.1 Exporting quantities ........................................ 28
      5.4.2 Getting the database information ........................ 29
      5.4.3 Obtaining the event fragment ............................. 30
      5.4.4 Registering our analysis routine ......................... 30

January 25, 1996
CONTENTS

5.5 The analysis phase ........................................... 31
5.6 How to save you from keying in this example ........... 32

A Specifications ................................................. 33
   A.1 Input ......................................................... 33
   A.2 Analysis ...................................................... 33
   A.3 Output ......................................................... 34
   A.4 Database ...................................................... 35
   A.5 Coordinate system (reference frame) ..................... 35
   A.6 Implementation ............................................. 36

B Naming conventions for detector modules ................. 37
Chapter 1

The structure of ADAM

ADAM, the *AmPS Data Analysis Method*, is a new data analysis package for data obtained at the AmPS facility. It is designed in a strictly modular way: a fixed base kernel, which offers mainly communications and interfacing services, and a series of “plug-in” modules, like detectors, filters and output formats for the more specific parts.

1.1 Kernel and modules

The ADAM kernel defines a framework for distributing raw information to the various detector modules, for communication among those detectors and facilities to make selections on the raw or processed data and finally produce output for further analysis.

The kernel is composed of several *subsystems*. A subsystem performs a specific task for ADAM, either directly (top level) or as a service to other subsystems. Typical examples of a top level subsystem are the event reader, database processor or the statistics routines. Lower level subsystems include the log- and error facilities.

The use of each subsystem is formally defined in an *Application Programming Interface* definition, in short *API*. This definition lists all possible calls one can make to a subsystem, the arguments one should use and the results of such a call. The word *API* is also frequently used to refer to the list of functions or to the subsystem itself.

Not all of the ADAM subsystems are of interest to arm developers. In practice, you’ll work mainly with the following API’s:

- **ADAM Communications Interface (ACI)** initialisation and calling of the routines for analysis from the loadable modules, distribution of event fragments to various detector modules.
- **Information Exchange Library (IEL)** communication of intermediate results from various detectors – like vertex and timing – and target corrections.
- **Logging library (LOG)** Writing messages to the ADAM logfile.
CHAPTER 1. THE STRUCTURE OF ADAM

Public Variable Space (PVS) Obtaining information from the database, exporting intermediate and final results for use by other modules.

Statistics library (STL) Keeping track of counting and rejection statistics.

The interface to each subsystem is described in detail in other documents available from the ADAM development group. The relevant function will be discussed in this introductory guide when appropriate.

To use the API in your arm code, you should include the appropriate definitions from a header file. This file is called `adam.h` and resides in the ADAM include directory (normally `//global/glance/adam/include`, or `$ADAMHOME/include`). This general include file lists all definitions of all ADAM API calls. To make compilation faster you can choose to include only certain subsystems in the list by setting `define`'s before your include statement. The list of possible defines is:

- `INCL_LOG`
- `INCL_STL`
- `INCL_ERR`
- `INCL_IEL`
- `INCL_PVS`
- `INCL_TAR`
- `INCL_ACI`
- `INCL_ALL`

The `INCL_ALL` directive means the inclusion of all subsystems and thus makes all other defines unnecessary.

| The use of the Adam header file |
| A typical piece of ADAM source code starts with: |
| ```c
    #define INCL_ALL
    #include <adam.h>
    ...
``` |

or, in case you only need e.g. the statistics and the PVS:

```c
    #define INCL_PVS
    #define INCL_STL
    #include <adam.h>
    ...
```

The detectors, filters and output modules are not part of ADAM itself. Instead, they are more like libraries, programmed in such a way that they can be linked to ADAM when necessary. These dynamic modules are called ACI loadable modules, after the name of the subsystem which manages them.

| loadable modules |

It is good to realise, that programming an ACI loadable module is fundamentally different from writing a stand-alone program. The basis of an ACI loadable module is the call-back function. A module asks the ADAM kernel that it be called when something has to be done, e.g. analyse an event fragment for that specific detector. This is done using an API call to AciRegisterProc, giving the address of the call-back function.
CHAPTER 1. THE STRUCTURE OF ADAM

to be called and the arm id. The kernel will record that a function for that specific task exists and will call that function when appropriate.

Registration of the call-back routines is done in the initialisation section of the module, which will always be called when ADAM starts (before any events are processed). This initialisation section has a fixed name (the same for every loadable module): ModInitialise. At the end of the analysis, just before ADAM terminates, the routines called ModTerminate from all loaded modules will be called.

A call-back routine for a parameter block

In the event stream, parameter blocks are identified by a special arm identifier, e.g. 0xf000. Suppose you want to extract information from such a parameter block to put it in the log file. To accomplish this, you write an ACI loadable module to process those parameter 'events'. In the initialisation section you register the call-back procedure for events with arm-id 0xf000. If this function is called e.g. ArmParabloHandler, you will register it with the call:

AciRegisterProc(ACI_SAP, ArmParabloHandler, 0xf000);

From that moment on, whenever an event with arm-id 0xf000 is found, the function ArmParabloHandler will be called. This function can be something like:

int ArmParabloHandler()
{
    LogWrite("* An Arm parameter block was found");
    return 0;
}

In general, there will be at least one such call-back routine for every arm-id present in the datafile. Besides, other types of call-back routine are present to produce different types of output (HBOOK histograms, ntuples) and/or to filter the events before or after analysis.

Figure 1.1 shows a schematic view of the ADAM design.

1.2 Types of modules

There are four different types of ACI loadable modules:

prefilter this type of module will register a prefilter routine. This module will typically look at the general characteristics of the event (e.g. coincidence level) and can in this way decide quickly whether or not to analyse the event. This cut could of course also be done at the end, after all analysis, but fast rejecting will save a lot of time.

detector a 'detector' module looks at the event fragment with a specific arm identifier and reconstructs relevant physical quantities from it,

\[\text{See the ACI description (separate document) for the syntax of the call.}\]
like energy at the interaction point, angle of emission, etc. The relevant quantities, which may also be intermediate results, deemed of interest for further analysis, or debug information, is exported to a shared area.

**physics** these modules are used to do post-processing on the quantities produced by the detector modules. The name is to some degree misleading, as the only available “physics” module will be a cutter. physics modules can look at all quantities made public by other modules or the kernel (detector quantities, parablo information etc.)

**output** output modules take the quantities produced by the detector modules and place them into an output file. This may be a histogram, an ntuple or any other type of output. Several output modules exist, and may be combined to satisfy the needs of the experimentalist.

Any quantity made public (*exported*) by a detector module may be put in the output, without the need for changes in the source code for the detector.

**Analysis stages**

Each of these modules registers call-back procedures of their own type, except for detector modules. Detector modules have two types of call-back procedures: *Single Arm (SAP)* and *Post Information Exchange (PIEP)*, which will be discussed in the next section. The registered routines are called in the above mentioned sequence (prefilter, SAP, PIEP, physics and output).

It is important to note that all modules are completely independent; they have no way of knowing which other modules are present during the analysis. The only thing with global impact that ACI loadable modules can do, is to reject the event (see below).

### 1.3 Exchanging information

Due to the way ADAM was designed, it is impossible to use information from one detector module directly in another detector. On the other hand, it is in many cases absolutely essential that intermediate results like a vertex are known to detectors which have no resolution at all in that specific quantity. To counter that obstacle, the *Information Exchange Library* and the two-stage analysis was introduced.

**Single Arm**

Detectors with good resolution in an interesting quantity like the vertex should calculate this quantity as soon as possible, based only on their raw event fragment. The result should be exported to the shared area (PVS) with a predetermined name\(^2\). This stage is called *Single Arm Analysis*, and should be done in the first analysis stage (SAP).

**Post Information Exchange**

After all single arm analysis for all detectors is finished, a second round is made. In this round, all PIEP’s will be called. These PIEP’s may use the raw event fragment and any quantity calculated by the SAP of the

---

\(^2\)Please refer to section 2.3.1 and appendix B for precise naming rules.
detector\(^3\). Besides, calls may be done to the *Information Exchange Library* (IEL). The Library will, to the best of its abilities, provide the arms with general information about the event, esp. the vertex. To calculate the general information, the IEL will use the intermediate results exported by the Single Arm analysis procedures from different arms. The quality of those intermediate results directly determines the quality of the IEL results. If no additional information is present for an event, a reasonable default value is returned to the calling PIEP.

It is now obvious that no calls to the IEL can be made from the SAP. At this stage, one can be sure that there is no general information, as the order in which the arms are executed is arbitrary\(^4\). Also, the importance of exporting 'interesting' intermediate results a.s.a.p. is clear. If those results would be saved till the end of the second analysis phase, the analysis would block.

### Communicating a vertex from BigBite to Hadron

A typical spectrometer, like BigBite or the QDQ, has a good vertex resolution. In case of extended targets, this knowledge of the interaction point is of great importance for detectors like the Recoil Detector or Hadron, which have no resolution at all with respect to that quantity. Before any useful information can be extracted from the Hadron data, the vertex should be known so as to reconstruct the proton track, on which the energy determination is also based. With only the raw event fragment, hadron can do very little.

The SAP for the Hadron will therefore be almost empty. Analysis should be postponed until the vertex is available.

BigBite on the other hand, can readily calculate the vertex to reasonable accuracy, given the hit patterns of the wire chambers. In fact, there is no quantity BigBite cannot calculated given its event fragment: the BB module will contain only a SAP, no PIEP is needed. At the end of the BB SAP, the routine will export its vertex line to the pool, under a predefined name.

In the second analysis stage, the Hadron PIEP will be called. At that moment, the IEL function `IelGetVertex` may be used to query the interaction point of the current event. This IEL call will, in turn, recombine the vertex line the BB SAP has put in the PVS, recombine it with the beam line and determine the most likely interaction point. The coordinates of this point will be returned to the Hadron PIEP.

In this way, the Hadron module will not know who presented the IEL with the knowledge of the vertex. The result is just used, the details of the calculation hidden from the hadron detector module developer.

---

\(^3\)It is the responsibility of the detector developer to conserve any interesting quantities between the two calls

\(^4\)but see the section on optimisation later on
CHAPTER 1. THE STRUCTURE OF ADAM

Figure 1.1: ADAM schematic structure of the kernel and loadable modules

- Public Variable Space (PVS)
  - Initialisation
  - Register call-back
  - Register quantities
  - SAP
    - Stand Alone Proc
  - PIEP
    - Post Info. Exchange
  - TERMINation
  - DETECTOR MODULE CODE

- Event fragments
- DataBase
- Intermediate results
- Final results
- per event
- to cutter and output

ADAM KERNEL
- initialisations
- unpack evt
- select coinc
- call SAP’s
- call PIEP’s
- call cutter & output
- terminations statistics out

Information Exchange Library (IEL)
Chapter 2

Communicating Information

During the analysis, you will use information from various sources: constants from a database, raw event fragments from the datafile, intermediate results from the Information Exchange Library. On the other hand, the analysis will produce new information: intermediate results, interesting physical quantities and statistical data. This chapter discusses the different sources of information, the ways to access that information and how to export new information to the outside world.

2.1 A common market – the Public Variable Space

All information, except for the raw event fragments, that must be used by more than one module, is communicated via the Public Variable Space (PVS). Every single piece of information stored in the PVS is named, i.e. all references to that information are based on its symbolic name. In contrast to ‘regular’ global variables declared in the ACI loadable modules, PVS variables are visible from everywhere in ADAM\(^1\).

Just like a ‘regular’ variable, a PVS variable should be declared before it can be used. But in contrast to ‘regular’ events, PVS variables are declared with an API call: \texttt{PvsRegister}. The arguments to this call are:

- the symbolic name of the variable
- the type of the variable
- its length, in case of a vector, otherwise zero

All calls to \texttt{PvsRegister} should be done in the initialisation phase. So if you plan to use a quantity, e.g. \texttt{rec.energy} in your analysis routines, you call the \texttt{PvsRegister} routine for that quantity in the initialisation section,

\(^1\)Due the the characteristics of the implementation, even variables explicitly declared \texttt{external} in ACI loadable modules are private to the module in which they were declared. They are global only in the sense that all objects within the module share the variable.
although you'll never use the quantity there. You can write variables during the initialisation, termination and analysis phases.

### Querying data from the PVS

To extract a piece of information from the PVS, one must use one of the PVS API calls. Although this may seem complicated at first, in fact the use of PVS variables is made easy through a series of tailor calls for specific tasks.

For example, imagine you want to know the moment at which an event occurred, with respect to the beamgate. This (predefined) event quantity is stored in the PVS with the symbolic name `evt.coarsetime`. It is also stated in the documentation\(^*\) that this is a long integer. To get the value of the coarsetime for the current event, you do a PVS API call: `PvsQueryInt`, like:

```c
...        
   printf("The coarsetime is \%d.", 
         PvsQuery("evt.coarsetime"));
```

One can also store the coarsetime in a variable. Trivially, the extension of the above-mentioned call in this case would be:

```c
int tm;         
   tm=PvsQueryInt("evt.coarsetime");
```

But alternatively, one could use the more general call:

```c
PvsQuery("evt.coarsetime",\&tm);
```

The first call is referred to as a **typed call**, the second is the more general call. Typed calls are only available for integer, float data and pointers stored in the PVS.

\(^*\)see *Public Variables – a comprehensive list*

### Naming of PVS variables

The name of a PVS variable consists of the 26 lowercase characters and the digits 0 – 9 and consists of four **fields**, separated by dots. Only the first three characters in each field are significant, but it is encouraged to use longer variable names, for enhanced comprehensibility. Uppercase characters can be used to give better readability, but they are mapped to their lowercase equivalents before use. A PVS name may not start with a digit.

Some examples of valid PVS variable names are:

- `aci.modules.detector`
- `evt.coarsetime`
- `h4.gain.l1`
- `bb.vertexline.phi`

**Significant character pitfall** but invalid names like `eva@heaven.org` or `h4.gain.H1_01T` give undefined behaviour. Note that the latter example also points at a hidden pitfall: `h4.gain.H1_01T` is identical to `h4.gain.H1_11B`, which is probably not what you wanted. If you want to represent a large series of numbers, use
vector variables (like a vector h4.gain.h1T with 40 elements).

2.1.2 Types of variables

Five basic variable types are known to the PVS:

<table>
<thead>
<tr>
<th>type</th>
<th>symbol</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>CHAR</td>
<td>an 8 bit integer</td>
</tr>
<tr>
<td>short</td>
<td>SHORT</td>
<td>a 16 bit integer</td>
</tr>
<tr>
<td>long</td>
<td>LONG</td>
<td>a 32 bit integer</td>
</tr>
<tr>
<td>floating point</td>
<td>FLOAT</td>
<td>a 32 bit IEEE floating point number</td>
</tr>
<tr>
<td>‘void’ pointer</td>
<td>PTR</td>
<td>a 32 bit pointer</td>
</tr>
</tbody>
</table>

Signed and unsigned

All integer variables are available in a signed and an unsigned variant. In practice, the use of signed and unsigned is almost useless: in the general calls, the value is copied to the specified address and no type checking can be done. In case of the typed calls, however, the sign for shorts and chars is observed when converting them to a signed long integer. Unsigned long integers are treated as signed longs and should be cast by the calling routine (`myvar=(ULONG)PvsQueryInt(...)`).

Arrays

From the five base types, arrays may be formed. The maximum length of an array is 255 elements. The elements are stored consecutively, and thus an array of characters can be used as a string, provided it is null-terminated. Two dimensional arrays are not allowed. An array is registered by specifying the modifier PVST_ARRAY in the call to PvsRegister, like:

```
PvsRegister("recoil.SiX.charge",PVST_FLOAT|PVST_ARRAY,32);
```

After this call, you have an array of 32 floats named recoil.SiX.charge. Arrays can be set only with the general call PvsSet:

```
char *PvsSet(const char * variablename,
             void * address);
```

With this call, the entire array is copied from address to the PVS memory. For example:

```
float charges_Si_X[32];
...
PvsSet("recoil.SiX.charge",charges_Si_X);
```

To make null-terminated strings easier to handle, the modifier PVST_STRING can be specified. With this modifier set, a PvsQuery call will copy data from the array only until the first null character. Remaining elements are not copied. PVST_STRING can only be specified with the character type.

2.1.3 Constants and event dependent data

All ‘global’ data is stored in the PVS. There is however a significant difference between constants (mainly imported from the database) and quantities that change every event.

\(^2\)see Public Variable Space API definition
This difference is expressed with three modifiers:

**PVST_RDONLY** This variable can only be assigned a value once. All variables read from the database have this modifier set. If you try to set a database constant to another value (with a PvsSet or typed variant thereof), it results in an error.

**PVST_EVENT** All variables for which this modifier is set are cleared after the current event is processed. They can be set again during the analysis of the next event. As it is an error to query the value of a variable which is not set, it warns the user of unassigned variables in the analysis.

**PVST_REREG** Explicit permission to register an already registered variable again, possibly with a new type. You’ll probably never use it.

2.1.4 Querying the type of a variable

Especially in the case of database variables, it is of interest to see whether or not a variable was defined, and if so, which type it has. An attempt to use a variable which does not exist is not an error, but leads to a warning. Execution can continue, but the global variable adamerr will be set to E_NOTFOUND. In that case, you can decide yourself what to do. If you absolutely need this variable, you can terminate ADAM yourself by calling the ErrTrap function, after displaying an appropriate error message (with ErrReport). On the other hand you can decide to use a default value and continue (but be careful to produce sensible results. It’s better to stop too soon, than to continue and make someone look at nonsense data for days, or worse...).

You can also query the type and modifiers of a variable, to see whether it really is what you expect. The API call PvsQueryType returns the complete PVS type, including modifiers. You can and this result with three masks:

**PVST_M_TYPE** selects the base type of the variable: char, short, long, float or pointer.

**PVST_M_QUALITY** selects the ARRAY, STRING and UNSIGNED modifiers

**PVST_M_MODF** selects the protection modifiers: EVENT, REREG, RDONLY and the USED flag.

This PvsQueryType call can also conveniently be used to see whether a variable exists. If it does not exist, PvsQueryType will return zero, otherwise the result will always be non-zero.

2.1.5 Database

At the startup of ADAM, the ‘database’ is read from disk and all the information contained therein is copied into the PVS. This means that the
detector (and other) loadable modules can use the information from the
database without the need to read it physically from disk. To use a
database constant, query its value from the PVS.
All information about the detector, the particle ID and everything else,
should be specified in the database, not directly in the source code. If a
need for such a constant arises, put its value in the database (or a relevant
include file) with a name, chosen by you, which identifies the constant.
Rules for naming a constant are described in the document *The ADAM
database*.
In the source code, you query the value of this constant with a regular
PvsQuery call or one of the typed equivalents. If the constant is not
present in the database, the variable will not be available from the PVS.
It’s up to the developer to determine whether or not analysis can continue
without this constant.

**Using database information**

Assume you have a database which contains the following line:

```
h4.angle,float: 57.7 // degrees
```

To use this value in your analysis code, you do:

```c
...  
angle=PvsQueryFloat("h4.angle");  
...
```

If you want to verify that the constant is really there, use a PvsQueryType
call and test whether the return value is different from zero:

```c
if(PvsQueryType("h4.angle"))  
  angle=PvsQueryFloat("h4.angle");  
else {  
  printf("HELP: Assuming angle of 90 deg for H4!!");  
  angle=90.0;  
}
```

**Warning:** Although the database recognises an array of strings, ‘string’ is
not a PVS base type, and thus an array of strings does not exist. The
database type ‘array of strings’ is treated like an array of pointers in the
PVS. The string space is separately *malloced* by the database reader. A
single database string however maps to the PVS string type.
CHAPTER 2. COMMUNICATING INFORMATION

Using strings from the database

Suppose you want to use a variable series of setup files for your detector. The names of those setup files should be specified in the database. As the database is read into the PVS at the start of ADAM, you should use PVS calls to get the names of those files. Assume you named your database variable `h4.charon.crates` and you want to open all setup files, one by one:

```c
FILE *setupfile;
char filename[MAXPATHLEN];
char *filenamearray[];
int filecount,i;

if(PvsQueryType("h4.charon.crates") & PVST_STRING) {
    /* single file */
PvsQuery("h4.charon.crates",filename); /*get name*/
    filecount=1
} else { /* should be an array of pointers to string */
    filecount=PvsQueryLength("h4.charon.crates");
    filenamearray=(char**)malloc(filecount*sizeof(char*));
PvsQuery("h4.charon.crates",filenamearray);
}
for(i=0;i<filecount;i++) {
    if(filecount==1) setupfile=fopen(filename,"r");
    else setupfile=fopen(filenamearray[i],"r");
    /* use the setupfile here */
    fclose(setupfile);
}
```

Of course, if you would have had only one setup file, the above would have been much simpler. You can specify in the database (by explicit typing) that a variable contains only one string, no array.

2.2 Using event fragments

*Event fragments*

ADAM can be used with EventBuilder, parablo-enhanced EventBuilder and arm-singles datafiles. Datafiles are processed by the ADAM kernel, and so the detector modules need not bother with the format of the datafile as a whole. The routines from the ACI loadable modules only see *event fragments*. An event fragment is a piece of data originating from a single arm. In the case of arm singles data, this would be the complete event, but if the input file is EventBuilder output, the event will be split into one or more fragments, one fragment for every arm-id. The shared data from the coincidence detector is processed by the kernel and only some of it is available (in a controlled way) to the detector routines.
To use event fragments in your analysis, you will first have to reserve a fragment space. This should be done in the initialisation section of your module with a call to AciGetFragSpace:

```c
char *AciGetFragSpace(unsigned short armid, int length);
```

This will reserve a maximum of `length` bytes of memory for the event fragment with identifier `armid`. From then on, if such an event fragment is found in the input file, it will be stored. How this event fragment is stored is hidden in the implementation. The starting address may vary from event to event and thus must be queried before analysis can start with a call to:

```c
EVTARMINFO *AciQueryEventInfo(unsigned short armid,
                                EVTARMINFO *buffer);
```

The EVTARMINFO buffer contains the starting address of the event fragment, along with some other useful information and is defined by the event subsystem as:

```c
typedef struct _evtarminfo {
    char *fragment;
    int length;
    unsigned short armid;
    unsigned long extractiontime;
    unsigned long finetime;
    int reserved;
} EVTARMINFO, *PEVTARMINFO;
```

The field named `reserved` may not be referenced or used by developers of detector modules. `fragment` points to the first byte of the event fragment corresponding to `armid`.

It is an error to query information about an event fragment for which no space was reserved. In the termination section of your module, you should release any fragment space with a call to AciReleaseFragSpace.

**Note:** Requesting fragment space in your initialisation section does not imply that one of your analysis routines will be called. Therefore you need to register them with AciRegisterProc. Likewise, registering a call-back routine does not imply that event fragments can be used by that routine. You need to reserve space with AciGetFragSpace.
Setting up an analysis routine

A simple framework to analyse event fragments for arm-id 0x0b00 is set up in the following piece of code:

```c
int SingleArm()

{  
    EVTARMINFO eventinfo;
    
    AcQueryEventInfo(0x0b00, &eventinfo);
    printf("Length was %d bytes, first word: %04x ...",
            eventinfo.length, (int)((USHORT)eventinfo.fragment)[0]);
    return 0;
}

int Initialise()

{  
    AcGetFragSpace(0x0b00, 1024); /*max 1Kb fragment*/
    AcRegisterProc(ACI_SAP, SingleArm, 0x0b00);
    return 0;
}
```

*See also the more elaborate example later on*

**Extraction time**

The EVTARMINFO structure also contains the field *extraction time*. This field is only useful in case of EventBuilder data and will always contain a zero in case of an arm-singles datafile (but also in the case of EventBuilder data the extraction time may of coarse be zero!). The extraction time is a combination of the coarse time of the event relative to the beamgate and the fine time generated by the channel module of the CoincDetector. Its resolution in 6.25 ns per tick. Analysis routines (typically the SAP) may choose to export this quantity (preferably as a float) on request as an additional arm/debug quantity. The advised name for this debug variable is *detectorname*.extractiontime (like in bb.extr).

### 2.3 Information Exchange

As we already discussed in chapter one, there is a need to exchange intermediate results between different detector modules during the analysis of an event. These quantities are strictly defined and the sharing of those quantities is regulated by the Information Exchange Library, or IEL. The relevant quantities are (as foreseen now):

- vertex (three spatial coordinates)
- time of flight
• polarisation of the electron beam (as reconstructed by a detector)\(^3\)

### 2.3.1 Supplying information

A detector should supply interesting intermediate results via the PVS. To allow the IEL to retrieve the data, a fixed naming convention should be used. The first field of the PVS variable name identifies the detector (e.g., \texttt{bb} for BigBit, \texttt{rec} for recoil, see the document \textit{ADAM naming conventions for information exchange} and appendix B. The remaining fields describe the quantity exported. The IEL variables should be registered in the initialisation section of the detector module.

For the three relevant quantities the definitions are:

**vertex** a detector, which is able to determine a \textit{vertexline} with reasonable accuracy (e.g. a magnetic spectrometer), should export this vertexline in \textit{hall coordinates}\(^4\). These hall coordinates are represented in three spatial coordinates of a point and the two angles \(\theta\) and \(\phi\):

\[
\begin{align*}
\text{detname}.iel.\text{vertexline}.x \\
\text{detname}.iel.\text{vertexline}.y \\
\text{detname}.iel.\text{vertexline}.z \\
\text{detname}.iel.\text{vertexline}.theta \\
\text{detname}.iel.\text{vertexline}.phi
\end{align*}
\]

**time of flight** The time of detection of the particle relative to the GPM timestart signal in nanoseconds. The raw ‘event time’ is available (in fineticks) in the arm-info block\(^5\) and can be converted to nanoseconds with the \texttt{IelRealTime()} function. The corrected event time should be exported as:

\[
\text{detname}.iel.timecorrection
\]

and is thus, symbolically, equal to \texttt{RealTime(arminfo.finetime)} + \texttt{correction}.

**polarisation** This quantity is not currently defined.

If more than one detector exports the same intermediate quantity, the IEL library tries to combine these two results into a best combined value. We strongly advise that the export of intermediate results is a configurable option in your detector module (i.e. a switch in the database), so that one can disable a detector from this mechanism when a more accurate one is available.

\[\textit{Vertex reconstruction}\]

\[\text{Only if a detector should be able to determine the polarisation on the incoming electron. The polarisation of an individual particle belongs only within one arm and/or its own output. If the polarisation is the same for more than a few events and is not determinable by a detector, parameter blocks should be used.}\]

\[\text{\cite{AdamNamingConventions}}\]

\[\text{\cite{AciQueryFragInfo}}\]
specified in the database) to produce a vertex. If more than one vertexline is available, and two of them are (almost) parallel, then lines who are the most perpendicular (including the beam line) are used.

2.3.2 Using information

To use information from other arms in your PIEP, you make calls to the IEL subsystem. On the first invocation of an IEL routine, the relevant intermediate results are combined into the required quantity (all vertex lines together form a vertex). This result will be the value of the quantity during the analysis of this event. If no intermediate results are available from other arms, a sensible default value is returned.

**vertex**

All interactions take place at (0,0,0) unless reconstructed otherwise from one or more vertex lines. To get the spatial vertex, use the IEL call:

```c
SPOINT *ielGetVertex(SPOINT *buffer);
```

The buffer will be filled with the x, y and z coordinates of the interaction vertex (in the hall reference frame).

**timing**

All events take place at time zero (0) unless a timing is exported by an arm, in which case the event will take place at that time. To use the event time in your arm analysis (i.e. to determine PID based on time of flight) use:

```c
float IelGetEventTime();
```

This will return the event time in ns.

### Calculating the relative time for your event

Assume that you have calculated all timing corrections for your own detector (e.g. recoil) and stored the result (in ns) in the variable `timecor`. In your output, you probably want the time relative to the moment of interaction. Assuming also that you have queried your fragment information and stored it in `fraginfo`, the following code would export the 'detection' time for your particle:

```c
... PvsSetFloat("rec.time",IelRealTime(fraginfo.firstime) IelGetEventTime()+timecor);
```

2.4 Target

In the same way that detector modules are independent of each other, the detectors modules are also independent from the target. To use information from the target, like the calculation of energy losses due to

---

6This property is planned for a future release

7In your arm information structure, the trigger time of your detector is specified in ticks. You can use the function IelRealTime to convert from ticks to nanoseconds in your analysis.
interactions of the outgoing particles and the target material, the detector module should call routines from the IEL subsystem. These calls will in turn request that a special target module calculates the energy loss, time of flight, etc. Note that the target is not an ACI loadable module, but can be replaced ‘on the run’ by specifying a different target module in the database (entry target.executable).

The quantities calculated by a detector would be the quantities at the target, so every detector module should correct for energy losses and time of flight by calling the IEL routines.

To make a proper correction for the energy loss and time of flight in the target, two calls will have to be made to the target module. Because the target module only implements the target proper, the exit point of your detected particle from the target depends on the vertex and the flight direction. Your target module should determine time-of-flight and energy loss up to the exit point, the target module then corrects for energy losses and t.o.f. suffered from the interaction to this exit point.

Target information is returned in a user-supplied buffer of type IELTARINFO:

```c
typedef struct _ieltarinfo {
    int size;       /* significant size of structure in bytes (=4*7=28) */
    SPOINT exitpos; /* exit position of particle from target in hall coord */
    float tof;      /* time of flight from interaction to exit point in ns */
    float energy;  /* energy of particle at interaction point in MeV */
    float dalpha;  /* estimate of multiple scattering angle straggling (rad) */
} IELTARINFO, *PIELTARINFO;
```

You obtain target information by calling

```c
int IelReconstruct(PSLINE pline, PCHAR pidname,
                    float ebox, PIELTARINFO buf);
```

If, on entry, the field `buf.size` is set to zero, the target will supply all information based on your input parameters. If you specify a size different from zero, only that many bytes of the buffer will be filled. Other properties will not be calculated. Note that in future kernel revisions the size of this buffer may change and your module may be made more compatible by always specifying the number of bytes requested. If you request more bytes than the target module can supply, an error is generated. The size must be a multiple of 4 bytes.

To determine the initial energy of your particle, you will typically make a first call to the target requesting 16 bytes to get the exit point. Then calculate the box energy up to that point and then request the initial energy and t.o.f. with the correct box energy.

The `PID` is a symbolic constant defining the type of particle detected. See the document `ADAM particle information` for the names of those constants. This document also describes how to obtain information about any other type of particle, so you will not have to specify any general particle properties (masses, charge etc.) yourself.

P.S. The `IelCalcTargetEnergy` call has been superseded and phased out.
Chapter 3

The analysis phase

If you want to make your detector module a useful tool for analysis, you should adhere to the guidelines laid out in this chapter. This chapter describes the quantities one should calculate, how to reject events which do not match the required particle identification etc.

3.1 What kind of events should be analysed

Particle ID

As stated in the definition, the data is analysed for one specific reaction. This means, that a detector should reject particles which do not match the predefined particle ID requirement: in a \((e,e'p)\) reaction, any pion events in the electron spectrometer should be rejected.

The type of particle required in a detector should be specified in the database, if a detector is capable of detecting more than one type of particle.

A detector analysis routine (both the SAP and the PIEP) may set other criteria to reject an event, e.g. a lower and upper limit on the wire chambers of a spectrometer (for selecting a specific energy). These kinds of restriction should be implemented in the arm analysis, but the limits in this case should be defined in the database.

Cuts based on information from one arm only should be done in the code of that arm. Cutting those events in a later stage (e.g. by the cutter module) will take an enormous amount of time spent analysing an event which will be rejected anyway.

Analysis order

For the same reason, it is advised to put detectors with a low efficiency (high percentage of events will be rejected) earlier in the chain. This can be done by changing the order in which the modules are specified in the database (field aci.modules.detector).

Also, rejecting events in the SAP is better then waiting until the PIEP. Even if you cannot do any real analysis in your SAP (because you need information from the IEL), rejecting events on some crude criteria is rewarded with a significant speed increase.
3.1.1 Rejecting events and statistics

To reject an event, it is enough to return a non-zero value from the SAP or the PIEP. When a non-zero return value is encountered by the kernel, no other modules in the analysis chain are called. As output modules also form part of this chain, the event is lost.

Statistics

If you decide to reject an event for some reason, be sure to maintain a count of rejected events. It is crucial to have later on during the analysis (efficiency determination etc). A general statistics library subsystem (STL) is available. By registering a statistics counter with the StlRegister function from the library, you can be sure that the counter is written in the logfile for every file analysed (and at the end the summed counts over all files). You are requested to use the statistics library and not write statistics messages to the logfile yourself. Once you have registered a counter, you can increment it by one with a call to StlIncr.

LogFile dump

You need not bother about the writing of the statistics counters to the logfile: it is done automatically, based on the statistics library setupfile (see the separate document describing the format of the setupfile). If a specific counter is not mentioned in the setupfile, it will nevertheless be written out. This minimises the risk of losing statistics information due to a bad setup file. The setupfile however adds additional functionality to the dump (like simple arithmetic operations on the statistics).

All calls to StlRegister should be done in the initialisation section of your module, as the statistics counters are a special form of PVS variables.

Counting your event fragments

You will probably want to know how many event fragments you processed during the analysis. To accomplish this using the STL subsystem, you call, in your initialisation section:

```c
...  
StlRegister("recoil.count");
...  
and later in your SAP procedure:

...  
StlIncr("recoil.count");
```

3.2 What should be calculated

required quantities

According to the specifications laid out in appendix A, the analysis phase should yield at least:

- momentum of the detected particle at the interaction point
- energy of the detected particle at the interaction point
- emission angles of the particle in the reference frame in radians.
- time relative to some corrected trigger time.
- polarisation of the detected particle (*details under consideration*)
CHAPTER 3. THE ANALYSIS PHASE

These quantities should be stored in the PVS under a predefined name, the first field consisting of the detector identifier (see appendix B), the following fields indicating the quantity. The following names are defined:

<table>
<thead>
<tr>
<th>Quantity</th>
<th>assigned name</th>
</tr>
</thead>
<tbody>
<tr>
<td>momentum</td>
<td>detname.momentum</td>
</tr>
<tr>
<td>energy</td>
<td>detname.energy</td>
</tr>
<tr>
<td>angles</td>
<td>detname.theta</td>
</tr>
<tr>
<td></td>
<td>detname.phi</td>
</tr>
<tr>
<td>time</td>
<td>detname.time</td>
</tr>
</tbody>
</table>

ADAM output should be given in natural units where possible: energy in MeV, momentum in MeV/c, angles in rad, distances in cm and times in ns. Besides the above mentioned required quantities, detector analysis Local and debug quantities modules may export any number of other quantities. One can think of hit patterns, differential light outputs, collected charges, etc. These quantities can often be useful in the further analysis and it is therefore encouraged to make the export of those quantities possible. To keep output ntuples, histograms etc. of modest size, you should however make the export of those “debug” quantities optional, i.e. implement a database switch to enable or disable the generation of the corresponding PVS variables. If a quantity appears in the setupfile of the ntuple or histogram module, but the variable is not registered in the PVS, it is ignored and does not appear in the output. This mechanism is useful when combined with the option in the detector modules to enable or disable verbose information. If you select, in the arm-code by means of a database switch, to add or remove certain ‘debug’ quantities, you do not have to change your output setup files. Remember that all PVS variables must be registered in the initialisation section of your module.
Chapter 4

Miscellaneous

This chapter describes a number of routines which are not absolutely necessary for the analysis, but make the development easier. They also ensure that the module at least roughly conforms to the ADAM default behaviour. Please read this section before starting with the examples! In the examples (next chapter) many of these routines will be used without further explanation.

4.1 Initialisation and termination

The functions ModInitialise and ModTerminate are called by the ADAM kernel when a module is loaded (initialisation) and when the analysis is done and a module will no longer be called (terminate). ModInitialise is called with two arguments:

\[
\text{int ModInitialise(int argc, char *argv[])};
\]

The arguments are taken from the database. Each entry in the list of loadable modules is divided in a list of arguments, separated by whitespace. This list of arguments, with argc entries, whose pointers are stored in argv, is passed to the module. The arguments are in a static space, which is overwritten after ModInitialise terminates. If you want to preserve the arguments, copy them to a local variable.
CHAPTER 4. MISCELLANEOUS

Using ‘commandline’ arguments in a module

Assume you use a database which contains the following line:

```
aci.mod.det:
    "/global/glance/adam/contrib/bigbite.mod -debug full"
```

The ADAM kernel will load the ACI loadable module specified by the first argument: `/global/glance/adam/contrib/bigbite.mod`. The initialisation function will be called with the arguments:

```
argc 3
argv[0] "global/glance/adam/contrib/bigbite.mod"
argv[1] "-debug"
argv[2] "full"
argv[3] NULL
```

The function `ModTerminate` is called without arguments.

4.2 Compiling an ACI loadable module

As stated in chapter one, ACI loadable modules differ from regular stand-alone programs. This means that also in compiling and linking a module, special steps must be taken.

Position independent code

ACI loadable modules are linked to the ADAM kernel at runtime. By a call to the operating system\(^1\), the *dynamic library* (i.e. loadable module) is included in the process address space of ADAM. Because the position where this module will be loaded is not known in advance, it should be compiled without any assumptions about the executable position (in case of a regular `a.out` file, the position is determined in advance by the linker `ld(1)`). This is called *position independent code*, or PIC.

Scope of symbols

All functions, variables etc. (in short *symbols* which will be used by the calling process, the ADAM kernel, should be in a position independent piece of code. Functions and variables which are local to the module need not be, even should *not* be in *pic* code: there is no need at all to share those variables of functions with the kernel or other modules. Those ‘local’ functions should be compiled as a regular object and bound to the *pic* code. Only two symbols in your module should be position independent: the initialisation function `ModInitialise()` and the terminate function `ModTerminate`. Those are the only two functions of which the symbolic name is used by ADAM. The call-back functions are registered at runtime and their address passed to ADAM at a moment at which they are already loaded. They are not called by symbol and should be in regular code.

Using the precompiled wrapper

If you are implementing an ACI loadable module and do not want to bother about position independence, you can use the precompiled wrapper object. This object defines the two public functions `ModInitialise` and `ModTerminate` as wrappers: the initialisation function will immediately call a (local) function `Initialise`, and `ModTerminate` will call a local function `Terminate`. The wrapper is compiled as *pic* code, all other objects should

\(^1\)you may want to reference the manual page for `dlopen(3X)`

January 25, 1996
be compiled as regular code. All arguments to `ModInitialise` are passed unchanged to `Initialise`. When linking your loadable module, you include this wrapper object in the list of objects to process. You can use the `acinmkf` command to create a makefile which automatically uses the wrapper object and correctly links all objects together in an ACI loadable module.

If an attempt is made to load an ACI loadable module twice, the operating system will detect this and implicitly use the already loaded version. As all variables (also the local ones) will be shared, this is probably not what you want! Reloading is as simple as mentioning a certain module twice in the list of detectors!

If you want to use the same module for more then one detector (e.g., for all Hadron detectors you develop one module), you should make physical copies of the final module and specify different copies in the database.

---

**Reusing ACI loadable modules**

Assume you have a Hadron detector module. The type of hadron event to analyse (1, 2, 3 or 4) is determined by the name of the module (e.g., "hadron1"). Suppose you compiled your module to something named "hadron.mod", to make all the required modules, you should type:

```
cp hadron.mod hadron1.mod
cp hadron.mod hadron2.mod
cp hadron.mod hadron3.mod
cp hadron.mod hadron4.mod
```

**Warning:** making symbolic or hard links is not enough! You should physically copy the disk file, otherwise it will still be recognised as 4 copies of the same module.

This does not apply to Solaris version 1.x (SunOS 4.1.2, 4.1.3 and 4.1.3_U1).

---

**Stripping**

The symbols contained in the dynamic module are essential for linking it to the kernel at runtime. You should therefore not 'strip' the module after linking (note that stripping is also done by the `install(1)` program when the `-s` option is specified).

---

### 4.3 Logging messages

During the entire run of ADAM, a logfile is maintained by the kernel. This logfile contains messages of general interest and the dumps from the statistics library. Modules may also write messages to this general logfile via the `LOG` subsystem. The only relevant call to this subsystem is:

```
int LogWrite(const char /*fmt*/, ...);
```

The `LogWrite` function can be called like the `printf` function. The format specifier `fmt` follows the same rules. See `printf(3V)` for a description.
4.4 Errors and warnings

Whenever the ADAM kernel detects an error, the global variable \texttt{adamerr} is set to the corresponding error code and the routine you called will return with a special return value indicating an error.

Most errors are so severe, that they require immediate termination of the analysis. Some others allow that buffers ed, be flushed. The following severity levels exist:

<table>
<thead>
<tr>
<th>code</th>
<th>symbol</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>level is unused</td>
</tr>
<tr>
<td>1</td>
<td>INTRN</td>
<td>impossible condition reached, internal failure. Terminate immediately and unconditionally.</td>
</tr>
<tr>
<td>2</td>
<td>FATAL</td>
<td>impossible action, implementation limitation. This error is mainly user-caused and indicates a malformed request or an action that is not allowed. ADAM terminates immediately and unconditionally.</td>
</tr>
<tr>
<td>3</td>
<td>TERM</td>
<td>action not allowed. The error was made in the user code (i.e. a loadable module), but ADAM is stable, so all termination routines will be called and output flushed. No more events are processed.</td>
</tr>
<tr>
<td>4</td>
<td>ERROR</td>
<td>request could not be satisfied (e.g. PVS variable does not exist). The routine will return an error code to the calling module and the global variable \texttt{adamerr} will be set.</td>
</tr>
<tr>
<td>5</td>
<td>WARN</td>
<td>a warning was issued.</td>
</tr>
</tbody>
</table>

Whenever an error of severity 1–3 is encountered, a message stating the cause of the error is written to the log file and to the standard error stream (UNIX descriptor 2). ADAM terminates by a call to \texttt{ErrTrap}.

Errors with severity 4 and up do not terminate ADAM. Instead, it is the responsibility of the user code to check all return values, or the value of \texttt{adamerr}. If an error was raised, the user code can report the default error message with a call to:

\begin{verbatim}
ErrReport(const char *fmt, ...);
\end{verbatim}

An error message is then printed containing the internal description of the error as well as the user specified explanation. \texttt{fmt} and the variable argument list are in the form of \texttt{printf}. The resulting message will be e.g.:

\begin{verbatim}
ErrReport("energy conversion factor ...
... fake.conv.energy required");
\end{verbatim}

\texttt{ERROR 0007(4): PVS variable not found}

If you deem the error to be too severe to continue, you can break ADAM from the user code. Please write an appropriate message to both the logfile and the standard error stream with \texttt{ErrReport}.

To stop ADAM, use the call:
ErrTrap(int severitylevel);
with the severity levels stated above. You should use symbolic names. The
are the names of the classes from the table above, prefixed with TRAP, so:
ErrTrap(TRAP_TERM);
This call is the only allowed way to stop ADAM. If you have situations in
which you want to terminate the analysis (an error defined by you), you
should write a message to standard error (stderr), the logfile (with
LogWrite) and stop ADAM with a call to ErrTrap.
Beware: only severity levels 1-3 cause ADAM to terminate, the higher
levels do nothing. It is not allowed to trap with severity INTRN.
Chapter 5

Development tutorial

In this chapter, we will guide you through the entire process of developing an ACI loadable module, exporting quantities and using the database. For simplicity, we did not take a real event type, but we designed one ourselves. This ‘fake event’ contains physical quantities. Almost; we still have to do some simple calculations to get at the final result...

We will use the simple programming interface (the precompiled wrapper) from section 4.2. Before starting with this tutorial (and in general whenever you want to do something with ADAM):

- append /global/glance/adam/bin to your PATH
- append /global/glance/adam/man to your MANPATH

For example, type:

```bash
PATH=/global/glance/adam/bin:$PATH
MANPATH=/global/glance/adam/man:$MANPATH
export PATH MANPATH
```

5.1 The fake event

The fake event directly represents physical quantities, like it was a very intelligent detector. Its layout is as follows:

<table>
<thead>
<tr>
<th>offset (b)</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>short</td>
<td>energy (in units of 0.0625 MeV). The detected energy varies slightly with the position of the vertex, it drops linearly with the distance of the interaction point and the origin.</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
<td>reserved</td>
</tr>
<tr>
<td>4</td>
<td>short</td>
<td>theta (in units of 0.01 rad)</td>
</tr>
<tr>
<td>6</td>
<td>short</td>
<td>phi (in units of 0.01 rad)</td>
</tr>
<tr>
<td>8</td>
<td>short</td>
<td>particle ID: 0=electron, 1=pi+, 2=other</td>
</tr>
<tr>
<td>10</td>
<td>short</td>
<td>charge of the particle</td>
</tr>
</tbody>
</table>
As you can see from the definition, this detector is utterly non-physical. It illustrates however the basic properties of the ADAM analysis. For instruction purposes we will however treat this detector like a real one. We will calculate the box energy, based on the vertex, perform the target energy loss corrections and export a 'good' energy at the interaction point to the outside world.

The conversion factors in the event may vary slightly in time, so they should not be hardcoded, but be put in the database. The detector name for this fake event generator is fak, its armid is 0x0D000.

5.2 Our design specifications

From the fake detector data, we want to extract the energy, momentum and angles for one specific type of particle (see specifications for detector modules in appendix A). All constants, including the PID are specified in the database.

The five output quantities will be called: fake.energy, fake.momentum, fake.theta, fake.phi and fake.time. We will also export the charge of the particle as “debug” information.

5.3 The database

The database in this case should include all constants needed by the fake detector module (like the conversion factors), the path of the modules to be loaded and the names of the setup files. Assume we only want to make histograms from the energy, momentum and angles. We need only to add the hbook histogrammer module.

The database, named fake.db, could look like this:

/* ADAM database for the fake detector arm, example */
aci.module.detector: "/global/glance/adam/contrib/fake.mod"
aci.module.output: "/global/glance/adam/lib/hbook.mod"
counter.interval: 1000  // show the counter every 1000 events

// HBOOK setupfile (see later on)
hbook.suf: "/global/glance/adam/db/hbook.suf"

// entries for the fake detector
fake.conversion.energy, float: 0.0625 // MeV per tick
fake.conversion.angle, float: 0.01  // rad per tick
fake.ecorerection.vertex, float: 1   // MeV/cm
fake.pid: 0    // electron detector

5.4 The initialisation and termination section

If you use the precompiled wrapper, the initialisation and termination routines are called:
CHAPTER 5. DEVELOPMENT TUTORIAL

Initialise(int argc, char *argv[])
Terminate()

The most simple module routine looks like the following. Lets name it fake.c:

/* This is a very simple demo module */
#define INCL_ALL
#include <adam.h>

int Initialise(argc,argv)
    int argc;
    char *argv[];
{
    LogWrite("* FAKE detector loaded with %d arguments\n",argc);
    return 0;
}

int Terminate()
{
    return 0;
}

Both the Initialise and the Terminate must be there, although they may be empty.
To make a working module from this piece of code, put it in a separate directory and cd there. Then type:

    acimkmf MODULE=fake.mod

This will create a makefile called Makefile for you. To make your new module, type

    make

This will compile the source code shown above, and link it together with the wrapper object. The resulting file will be fake.mod in the current directory.

Installing modules
To make this module publicly available for everybody, execute a make install. This will copy the module fake.mod to the directory /global/glance/adam/ contrib. This directory works like /tmp: everybody can write to it, but you cannot delete or modify someone else's modules.

starting ADAM
To start your analysis with adam, you would type

    adam fake.db fakedatafile

As there is no faka datafile available, you will not be in the position to actually test this fake module. To do some testing, start developing a module for your own detector. For a complete description of ADAM commandline options see the manaulpage for ADAM(1A).

5.4.1 Exporting quantities

As stated in the specifications, we want to export five quantities. This means that we should register five variables in the PVS during our
initialisation function. Most of those quantities are floats, one integer (the charge). In our initialisation section, we add:

```c
...  
PvsRegister("fake.energy",PVST_FLOAT|PVST_EVENT,0);  
PvsRegister("fake.momentum",PVST_FLOAT|PVST_EVENT,0);  
PvsRegister("fake.theta",PVST_FLOAT|PVST_EVENT,0);  
PvsRegister("fake.phi",PVST_FLOAT|PVST_EVENT,0);  
PvsRegister("fake.charge",PVST_LONG|PVST_EVENT,0);  
PvsRegister("fake.time",PVST_FLOAT|PVST_EVENT,0);  
...  
```

These quantities are only exported. Our detector module need not bother about histograms, ntuples and the like. It is only in the setup file of the *hb00k* module we should specify those quantities and the way they will be represented.

**Statistics**

During our analysis, we will most likely encounter events which do not match the required particle identification specified in the database. Those events should be rejected. A count of those rejected events should be maintained using the STL subsystem.

We also want to maintain a count of how many events we saw in the input. To register both counters, we add in the initialisation section:

```c
...  
StlRegister("fake.count"); ///* number of events seen */  
StlRegister("fake.reject.pid"); ///* rejected on PID */  
...  
```

### 5.4.2 Getting the database information

We also need our database entries during the analysis phase. Therefore, we get their value from the database for further use (we store them in local variables to reduce the overhead introduced by repeatedly querying the PVS for large numbers of variables). At the beginning of our module, we add some local variables, so it will look like:

```c
#include <adam.h>  
static float e_convert,thph_convert;  
static float e_vertex_corr;  
static int pid_required=0; ///* defaults to electrons */  
...  
```

And in the initialisation section, we put:

```c
...  
if(!PvsQuery("fake.conv.energy",&e_convert)) {  
    ErrReport("database entry fake.conv.energy required");  
    ErrTrap(TRAP_TERM);  
}  
```
if(!PvsQuery("fake.conv.angle",&thph_convert)) {
    ErrReport("database entry fake.conv.angle required");
    ErrTrap(TRAP_TERM);
}
if(!PvsQuery("fake.ecorr.vertex",&e_vertex_corr)) {
    ErrReport("database entry fake.conv.angle required");
    ErrTrap(TRAP_TERM);
}

if(PvsQueryType("fake.pid")) /* valid database entry for PID */
    pid_required=PvsQueryInt("fake.pid");
else
    LogWrite("* FAKE - analysing for default \ particle type ELECTRON\n");
    ...

5.4.3 Obtaining the event fragment

To do any useful analysis at all, we need the event fragment of armid 0xD000. We know our fragment size will never exceed 12 bytes (in fact, it will always be this size). To allocate space for our event fragment and extract it from the data stream, we add the AciGetFragSpace call to our initialisation routine:

    ...
    AciGetFragSpace(0xD000, 12);
    ...

5.4.4 Registering our analysis routine

Because in the analysis the position of the vertex is required, and we cannot determine the vertex ourselves, we need two analysis stages. If we are analysing singles, you might suppose that the analysis totally breaks. However, in absence of a vertex-determining detector, the IEL subsystem assumes the vertex at (0,0,0).

We now need to register the two analysis routines. The SAP will check for particle ID, so we can reject inappropriate events as soon as possible. The PIEP will do the real calculations. We'll call our SAP FakeSap, our PIEP FakePiep. To register those two call-back functions, we add to the initialisation:

    ...
    AciRegisterProc(ACI_SAP,FakeSap,0xD000);
    AciRegisterProc(ACI_PIEP,FakePiep,0xD000);
    ...

and add at the top of the initialisation routine:

    int FakeSap(), FakePiep();
5.5 The analysis phase

In the first analysis stage, you can do nothing more then check the particle ID. Also the information about the event (fragment address, length and timing) is queried and stored in a static structure to preserve that info for the PIEP.

In the global section of the file, we include (e.g. below the include directive):

```c
...  
#include <math.h>

static EVTARMINFO fraginfo;  
...
```

In the SAP we use the following code (comments shown in code):

```c
int FakeSap()  
{
    int pid;

    AcqQueryEventInfo(0xD000, &fraginfo);
    pid=((unsigned short *)fraginfo.fragment)[4]; /* byte offset 8 */
    if(pid != pid_required) {  
        StlIncr("fake.reject.pid"); /* increment statistics */
        return 1;  
    }  
    return 0; /* preserve the event and continue */
}
```

In the second phase, we use the event fragment (the information was stored in a local variable by the SAP already) and do a call to the IEL subsystem.

```c
int FakePiep()  
{
    float energy, momentum, theta, phi;
    float distance, time;
    int part_type;
    char *pidname;
    SPOINT vertex;
    SLINE vline;
    IELTARINFO tarinfo;

    energy = e_convert*((unsigned short *)fraginfo.fragment)[0];
    theta = thph_convert*((unsigned short *)fraginfo.fragment)[2];
    phi = thph_convert*((unsigned short *)fraginfo.fragment)[3];

    /* calculate distance based in vertex */
    IelGetVertex(&vertex);
```

January 25, 1996
distance = sqrt((vertex.x*vertex.x) +
                (vertex.y*vertex.y) +
                (vertex.z*vertex.z));

/* the real box energy depends linearly on the vertex position */
energy += distance*e_vertex_corr;

/* perform target corrections, based on pid */
switch(((unsigned short *)fraginfo.fragment)[4]) {
  case 0: /* electrons */
    pidname = "e-";
    break;
  case 1: /* pions */
    pidname = "piplus";
    break;
}

tarinfo->size=24; /* calculate everything upto and incl. energy */
vline.th=theta;
vline.ph=phi;
IelReconstruct(&vline, pidname, energy, &tarinfo);

momentum=0.; /* this is real fake. See the document
  ADAM particle information and calculate it from
  the energy */
time=IelRealTime(fraginfo.finetime)+IelGetEventTime();

/* export quantities */
PvsSet("fake.energy",&energy);
PvsSet("fake.momentum",&momentum);
PvsSet("fake.theta",&theta);
PvsSet("fake.phi",&phi);
PvsSet("fake.time",&time);
PvsSetInt("fake.charge",
           (int)((unsigned short *)fraginfo.fragment)[5]);

return 0; /* keep the event */
}

5.6 How to save you from keying in this example

A compilable version of this example is stored in the directory
/global/glance/adam/lib/demo. Copy this file fake.c to an empty
directory, make sure you appended your PATH and MANPATH correctly
and type acimkmf MODULE=fake.mod. The warning about adam.h not
found can be ignored.
Good luck!
Appendix A

Specifications

The formal specifications for ADAM were fixed at a meeting on July 18, 1995 and are reproduced below for reference.

The Amps Data Analysis Method (ADAM) intends to become the modular analysis package for electron scattering data measured at AmPS. ADAM is designed in a highly modular fashion, to ensure a well maintainable and expandable analysis package for the future. This means that all parts of the analysis that are specific for a certain detector, target or output format (or experiment) are not part of the standard code of ADAM. Instead, they are considered as plug-in modules and have to be selected in the database in order to be used. In this way, changing or replacing modules doesn’t affect the general system. The communication between the modules is done via well-defined protocols.

A.1 Input

The input for ADAM may be either data in event builder format (including parablo blocks), or may consist of arm singles (as generated by the filter-program from spy-data). Data are read from disk or from standard input (convenient for using it in combination with the filter-program). A possible future expansion is to enable reading data directly from gltape. It will be possible to analyse data from any number of files, and write the output to one file.

Parablo blocks in the data stream will cause an update of the relevant parameters in the runtime copy of the variables space describing the experimental conditions.

A.2 Analysis

In the database the type of reaction for which the data will be analysed is selected. This includes specifying the particle type that must be detected in each detector. If one of the detectors cannot reconstruct a track corresponding to a particle of the required type, the event will be rejected. Every detector should find at maximum one track (for the correct particle type) that can be used in the further analysis. If a detector finds multiple
tracks, it is up to the programmer of the code to choose one, i.e. the one that made the trigger, or to reject them both.

All detector codes must work independent of the experimental conditions; there is no knowledge (neither available nor obtainable) about any other detector or the target.

The detector codes consist of four parts: an initialisation routine, a Stand Alone Procedure (SAP), a Post Information Exchange Procedure (PIEP), and a termination routine. The initialisation and termination routines are executed just once, but the SAPs and PIEPs are executed for each event. The SAP performs the analysis based on the single arm event fragment only, without any information from other arms. The calculated quantities possibly being of interest to other arms, are submitted to the Information Exchange Library (IEL). After the SAPs of all arms have been executed, the PIEPs are called. The PIEPs are allowed to ask the IEL for information from other arms (like the vertex or timing). Using this information, each PIEP should be able to complete the analysis of its event.

Every arm must calculate the following quantities:

- momentum and energy of the particle
- the angles theta and phi in the reference frame (see below)
- optionally, the polarisation

For reconstruction of the vertex, at least one arm must produce a vertex line. In combination with the direction of the beamline, it is then possible to calculate the vertex. The analysis of other arms may depend on the vertex. Energy loss corrections in the target will be performed by library functions. For energy loss in the detectors, a set of functions will be offered to the arm developers which will perform these calculations. All arms may calculate additional quantities which are convenient in the further analysis, or may be used for debugging purposes.

ADAM will only produce single arm quantities in its output, i.e. there will not be any experiment dependent physics in ADAM. ADAM will not perform kinematic corrections; this may be done, however, in the arm code of the electron detector. The output of ADAM can be used for further analysis using PAW-macros or dedicated programs, containing the experiment dependent physics routines.

### A.3 Output

The output of ADAM is taken care of by plug-in modules, analogous to the arm analysis codes. These modules may be replaced or expanded when the output data is desired in a new format. The output modules read from a setup file which quantities should appear in their output stream. For each event, the values of these quantities are acquired from the system and then written to the output in the selected format (HBOOK histograms, ntuples, DATAN, ...).
For log output and bookkeeping of statistics, the arms have to use library functions taking care of this. These library functions will perform all statistics operations needed. Reject statistics will not be stored within the various arm codes, but in the statistics library (STL).

Before the output modules are called, a fraction of the data may be selected by a cutter-module. This module uses a setup file to put rude cuts on quantities (only lower and upper limit). Only those events, for which all cut conditions are satisfied are selected for output.

### A.4 Database

A new database format will be introduced to structure the contents in a more transparent fashion. A general set of functions to read the database will be offered in a library. These functions may also be used in other programs (e.g. experiment dependent physics). The CSG will provide a (graphical) database editor for the new database.

The most important properties of the new database format are:

- it will be in plain text (ASCII)
- the data may be structured in 4 levels of hierarchy
- usage of `#include` statements gives the opportunity to combine various parts of detector databases into one experimental database without loss of structure
- data may be of one of the following types:
  - floating point
  - integer
  - string
  - arrays of the above
- as much information as possible will be stored in the database

The database will also contain information about the detector geometry. This will facilitate adjusting the analysis parameters if there are changes in the setup. It is easier to implement these modifications via the database than in the source code.

### A.5 Coordinate system (reference frame)

See also the drawing of the reference frame. The reference frame in which all quantities must be expressed has its z-axis along the beamline. The positive y-axis is the vertical direction (upward). To make a right handed system, the positive x-axis is in the horizontal plane, pointing to the outside of AmPS. The origin of this frame is determined in the ITP-hall by markers on the walls.

The angle theta is the angle between the vector under consideration and the z-axis. Its value lies in the interval $[0, \pi]$. The angle phi defines the azimuthal angle of this vector. Phi is restricted to the interval $[0, 2\pi]$. 

January 25, 1996
A.6 Implementation

ADAM will be implemented in C. The arm developers should conform to this, as far as interaction with their outside world, like ADAM and I/O, is concerned.
Appendix B

Naming conventions for detector modules

The first fieldname for all PVS variables associated with a specific detector should be assigned according to the following list. Note that as only the first three characters are significant, you can extend your name to make the sourcecode more readable. You can of coarse also mention those longer names in your own arm documentation.

<table>
<thead>
<tr>
<th>Detector name</th>
<th>first fieldname</th>
<th>example (imaginary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QDQ spectrometer</td>
<td>qdq</td>
<td>qdq.wire.x1</td>
</tr>
<tr>
<td>QDD spectrometer</td>
<td>qdd</td>
<td>qdd.adc1</td>
</tr>
<tr>
<td>QDQ+</td>
<td>qpl</td>
<td>qpl.aero.pm4</td>
</tr>
<tr>
<td>Hadron 1</td>
<td>h1</td>
<td>h1.diff.11.01</td>
</tr>
<tr>
<td>Hadron 2</td>
<td>h2</td>
<td>h2.uctime</td>
</tr>
<tr>
<td>Hadron 3</td>
<td>h3</td>
<td>h3.adc.h2.24</td>
</tr>
<tr>
<td>Hadron 4</td>
<td>h4</td>
<td>h4.tcor.3</td>
</tr>
<tr>
<td>RUU TOF</td>
<td>tof</td>
<td>tof.veto</td>
</tr>
<tr>
<td>Recoil</td>
<td>rec</td>
<td>rec.charge.SiX</td>
</tr>
<tr>
<td>Range Telescope</td>
<td>rt</td>
<td>rt.energy</td>
</tr>
<tr>
<td>Calorimeter</td>
<td>cal</td>
<td>cal.energy</td>
</tr>
<tr>
<td>BigBite</td>
<td>bb</td>
<td>bb.momentum</td>
</tr>
<tr>
<td>HARP</td>
<td>har</td>
<td>harp.veto</td>
</tr>
<tr>
<td>Forward Angle Detector</td>
<td>fad</td>
<td>fad.theta</td>
</tr>
<tr>
<td>Neutron bars at ITH</td>
<td>nb</td>
<td>nb.geo.dist</td>
</tr>
</tbody>
</table>

For specific beam properties, which are continuously measured, the name beam is reserved. Although you are not allowed to use any other first fieldname than the one assigned to your detector, we list the fieldnames explicitly reserved for other purposes:

aci ACI  bee beam  boo boot  cfi coincfilter  
cou counter cut cutter  db database  ecf enhanced c.f.  
evt event lib hbook  iel info exch mtu ntuples  
par par block phy physics  pvs PVS  scf symbolic c.f.  
sdr SDRF  sha shared  stl statlib  tar target  

January 25, 1996