MCFIO : A machine independent I/O system for Monte-Carlo studies.

P Lebrun
Bphys memo

April 12, 1996

Abstract

An I/O system based on the XDR protocol from the RPC software has been written to exchange Physics data files across various machines. direct access capabilities have been implemented, such as the automatic searches for specific triggers. This note is a brief user's guide and reference manual.

Version 2.0: Full implementation of direct access stream and sequential streams. Specific structure, such as STDHEP or TRACK_OFFLINE have been moved to another directory of McFast, so that MCFIO becomes a stand-alone package. Disk-based NTuples are the outstanding new feature for this version.

1 Introduction and Scope of MCFIO

The basic requirement is to be able to read/write MCFast data structures (of "event blocks") to mass media in a machine transportable fashion. By MCFast data structure we mean things like instances of the STDHEP COMMON, the track records, and other data blocks created by this program. It is implicitly understood that we want to save/restore the values for these data structures on an event by event basis. By mass media, we either mean disks or tape, both sequential or direct access must be supported by MCFast.

A machine transportable (or machine independent) image of this data, once written on a specific platform, can be read from another machine without any code changes, or setting of specific environment variables.

Additional capabilities are:

- direct access: accessing specific events or data blocks within an event without having to scan through the entire file. The file must be on a direct access media (e.g., on disk) and set for direct access.
- Support for a variety of sequential media: MCFIO must be able to interface to 8mm, DLT's, or other types of serial media.
- Support for standard ASCII tape labels, if media is sequential (e.g., tape).
- Multiple I/O streams per MCFast process, running quasi-concurrently, without interference or other limitations.
- Full versioning: each Data Block is stamped by a unique version 4-characters string, that can be set by the routine encoding the block. This should guarantee to implement backward compatibility for each block.
- Disk-based Ntuple: Based on a defined data model, COMMON block or C-like data structures can be easily mapped to MCFIO Blocks, without user-written XDR filter. The data language description for these Ntuples is also saved on files. Therefore, within the context of this data model, MCFIO files are entirely sef-described.

The MCFIO direct access files are basically UNIX files. Each opened file corresponds to a stream. Currently, up to 20 concurrent streams can be opened at any given time (this is just a PARAMETER in the software, and can be redefined as needed). A given stream is either input or output, but not both: no updates or changes from output to input are allowed without closing the stream and reopening it.

The structure of an MCFast event, from the I/O subsystem perspective, is extremely simple: it consists of a header and flat list of blocks, each block contains a structure (or a tree of structures). An event without data blocks is a valid event. There is no limit on the number of blocks per event per stream, but a maximum number of blocks must be declared while opening an output stream. No duplicate blocks are allowed: for instance, it will not be possible to store two distinct snapshots of the STDHEP COMMON block in a single event. The user will have to create a second clone of STDHEP (since it will most likely have a slightly different set of variables, it is a different block). Blocks can be fetched/stored in an event in random order. Some additional blocks, such as tape labels, header or, for direct access, "Event table" blocks will have to have to be defined, but since they are used internally, the user needs to know about them. Unlike FarFalla, or other C^{++} tools, we do not maintain a hierarchical tree of structures describing the event for the user. A file consists of an ensemble of events and an event consists essentially of a flat list of Blocks. It is up to the user to define a hierarchical structure within these blocks. Such a limited design comes from the fact that MCFIO must be callable from FORTRAN (F77), a language that does not gracefully dynamically allocate hierarchical tree structures.

An MCFIO event does not necessarily correspond to a single HEP collision: multiple interactions in a single "event" can easily be supported, for instance by cloning blocks with similar structures but different contents. By MCFIO blocks, we mean a C or VAX FORTRAN data structure that is mapped to a set of XDR filters. Rules and examples to implement new MCFIO blocks will be described later.

The package is based on two distinct tools:

- XDR: The machine-independent data representation is achieved through a set of translation based on the XDR Networking package. The External Data Representation (XDR) library functions are used to represent data structures in a machine independent form. The XDR protocol is documented in the ARPA Network Information Center's RFC1014. An accessible version of this documentation has been published by O'Reilly & Associates, Inc[1] XDR can handle data stream directly on files or to a virtual memory segment that can be written to sequential media at a later stage. XDR is used simply to 'convert' machine dependent images of character strings, floating point, double precision, etc... quantities to a byte stream that can be decripted on all UNIX (or VAX, if need be (God forbid)) workstations. A real example on how this can be implemented will be shown later.

[1] Power Programming with RPC, by John Bloomer, ISBN:0-93715-773

- The Fermilab RBIO package, in order to transfer an event buffer from memory to sequential media. Sequential media can be either files on disk, which have an internal record structure so that they can easily be transferred from/to tape using RBIO calls, or a genuine sequential device, e.g., 8mm tapes of DLT. These tapes will be labelled. Most if not all sequential I/O peripherals support variable length records, which makes it possible to fully contain an event in a sequence of blocks, without wasting space. Thus, each record contains at most one event, an event may consist of more than one record. One event consists of a contiguous bunch of records, each event starts at a record boundary. This choice has been made for ease of implementation, and related performance consideration. As XDR supports standard Unix I/O with the capability of "seeking" position in the stream, RBIO would add little for direct access devices ,and is not used if direct access files (e.g. UNIX files that are not record oriented.) are accessed.

Note that the basic routines to build an event and doing the event I/O, within the event loop, can be done using the exact same call for sequential and direct access. Obviously, some functionality such as skipping over data blocks or entire events is not available for sequential streams, but if one simply wants to go from one event to the next, the user does not need to change his code in the event loop, only the opening and closing calls change from direct to sequential.

The package is written in C, and has a FORTRAN binding (The C binding would be trivial to write; however, we currently have no customers for it). Casual users obviously do not need to learn C or XDR software. Some knowledge of C and XDR are required to implement new MCFIO blocks.

Many examples are provided as testing material with the package. They are briefly described at the end of the tutorial section. The last section consist of a crude reference manual.

2 Tutorial and Examples

This section explains how to modify a standard user MCFast Code. Note that some of these calls could be moved to the "system" part of MCFast (since, currently the MCFast system directly handles the reading of STDHEP events, this should probably be considered.) . For sake of clarity, let us assume now that the user is in charge. This example is fairly lengthy, obviously many options are far from minimalist.

Let us assume that the mandatory minimalist job consist of reading a disk file with only STDHEP COMMON blocks. FORTRAN line marked with a * are mandatory calls, and refer to the *core* MCFIO system.

2.1 In User Initialization (usr_init)

Code:

```
Notes....  VAX FORTRAN CODE

1*)  'include mcfio.inc'

2)       integer ostr_tape, ostr_disk, istr_old
         COMMON/USER_IO/ostr_disk, istr_old

3)       Character*80 device, title, filename, blocktitle
         Character*6 label
         integer numblocks, block_id, blocks_ids(3), max_evts

4)       integer BLOCKCER, mcfxdr_CerenkovPrelim
         integer size_CerenkovBlock
         external mcf_xdr_CerenkovPrelim, size_CerenkovBlock
         PARAMETER (BLOCKCER=16001)

4*)      call mcfio_Init

1)       call mcfio_printdictionary

5*)      filename ='bpsi_isajet.dat'
         istr_old = mcfio_OpenReadDirect(filename)

5)       call mcfio_InfoStreamChar(istr_old, MCFIO_TITLE, title)
```

```
      print *, ' Title of STDHEP run ', title
      call mcfio_InfoStreamInt(istr_old,
     &              MCFIO_NUMBLOCKS, numblock)
      if (numblock .lt. 1) then
           print *, ' Not much on this file, bail out'
           call mcfio_close(istr_old)
           return.. or stop
      end if

      call mcfio_InfoStreamInt(istr_old,
     &                          MCFIO_BLOCKIDS, block_ids)
      if (block_ids(1) .ne.
     &     MCFIO_STDHEP .or. numblock .ne. 1) then
        print *,
     &   ' STDHEP is not the first and only block, bail out'
        call mcfio_close(istr_old)
         return... or stop
      end if

4)    titleblock = 'Cerenkov Hits, Preliminary Version
      block_cer = mcfio_DefineUserBlock(BLOCKCER,
    &       title,MCFIO_xdr_CerenkovPrelim, size_CerenkovBlock)
      max_evts = 10000
6) ostr_disk = mcfio_OpenWriteDirect(filename,
    &              title, 'No Comments',max_evts, blocks_ids, 3)
```

Notes

1. The insertion of an include file is necessary, as we use strong casting. More importantly, parameter settings and information gathering about the Stream, events or blocks are done through subroutine calls, which is a safe, sound mechanism, that implements *information hiding* and *data encapsulation*, so that the user interface is not bound to a particular version of the inner data structures. A subroutine has been provided to print the Parameter Dictionary.

2. The user must keep track of the MCFIO I/O stream indices. In this example, istr_old and ostr_disk are two integers, referring to two distinct data streams : one input, to read an old disk file with STDHEP blocks, and one output to disk. These integers will be referenced later in the event loop. An MCFIO I/O stream will be mapped to either a RBIO jfn (see RBIO manual), or to a stdio XDR stream, in the MCFIO package.

3. Local Variable definition.

4. Variable and Option settings in MCFIO: There is only one initialization routine, setting all parameter default values. This routine *mcfio_init* can be called more than once in a given run, in which case all previous settings are lost. Another setting in this example consists of declaring a "User defined block". The user must define a unique identifier. A title for the block is also a good idea. Finally, the user must write an XDR filter for his private data structure, and a little routine wich is called by MCFIO to get the current size of the data block to preallocate memory, in the case of sequential streams.

5. The input stream definition for this example. Obviously, none of these calls are mandatory, but in a non-trivial example, we need at least one! This is a direct access stream, one will be be able to access particular events in random order. Although not required, we also wish to check that this is not a trivial file, and the unique block per event is STDHEP. If the file is know to be valid, safe and sound, it is possible to read it faster using *mcfio_OpenReadMapped*, which bypasses the rather slow UNIX STDIO system, used in *mcfio_OpenReadDirect*

6. One output stream definition, going to disk. Once again, a title for the data set is probably a good idea. This is not mandatory, the user can always leave it blank (implemented by a *%val(0)* argument value, on most systems).

2.2 Getting Events

Depending on the exact layout of the job, this code must be inserted in usr_generate and/or usr_analysis.

Code:

```
       integer ostr_all, ostr_sel, istr_old
       COMMON/USER_IO/iostr_tape, iostr_disk, istr_old

8*) include 'stdhep.inc'
       integer xdr_mcfast_stdhep
       external xdr_mcfast_stdhep

       integer iostat, event_number
       integer store_numer, run_number
       integer trigger_mask

7a*)   iostat = mcfio_NextEvent(istr_old)
7a)    call mcfio_InfoEvent(istr_old,
     &             MCFIO_EVENTNUMBER, event_number)
       call mcfio_InfoEvent(istr_old,
     &             MCFIO_TRIGGERMASK, trigger_mask)

7b)    event_number = 553
       iostat = mcfio_SpecificEvent(istr_old, event_number,
    &                      0,0,0)

7c)    trigger_mask = 129
       iostat = mcfio_NextSpecificEvent(istr_old, 0,
    &                      0,0,trigger_mask)

8*)    if (iostat .gt. 0)
    &    iostat = mcfio_Block(istr_old, MCFIO_STDHEP,
    &    xdr_mcfast_stdhep)
       if (iostat .gt. 0) ......
```

7. mcfio_NextEvent is the basic routine to read or write an event from a specific MCFIO I/O stream. The routine will get the event header, the user can now inquire event property and decode specific blocks.

Returns -1 if there are problems (e.g. , I/O error or end of the stream..), 1 if a new event has been read or written. Once the event header is in memory, one can gets information about its unambiguous identifiers, such as run numbers, store or spill numbers and so forth. Note that we use a subroutine call to do this rather than a FORTRAN COMMON, to avoid version dependencies on the exact contents of the event header. Since the Input stream is direct access, one can search for a specific event starting from the beginning of the file (SpecificEvent) or from the current stream position (NextSpecificEvent). MCFIO internally keeps a table of these key event identifiers. It will position the XDR stream to the requested location without reading sequentially through all the events. It might have to read a few more of these tables, but this will be much faster than going through the entire file. If the value of one of those event identifiers is set to 0, this means that any value will do. Hence, if event_number, spill_number, run_number and trigger_mask are all set to 0, this is equivalent to a call to *NextEvent*.

8. mcfio_Block is the basic routine to ENCODE/DECODE a specific block, in our case, STDHEP. Returns 1 if O.K., -1 if the block is missing or there are I/O problem. The first argument is simply the stream index, the second is the unique block identifier, and the last argument is the external subroutine that performs the XDR transfer. Obviously, from a logical stand point, this last argument is redundant: the package could have figured out internally which routine to call! But this implementation would have implied that *all* FORTRAN COMMON blocks be always loaded, an unnecessary cost of memory. To avoid this, it is up to the user to declare the COMMON Block (including file stdhep.inc), and declare explicitly the appropriate XDR filtering routine. These routines check for consistency with respect to the Block I.D.

2.3 Saving Events (end of usr_analysis)

For events of interest, once the user-event analysis is done, one now has to block and buffer the events out. So, at the end of user_analysis :

```
\vspace{.1in}
 Code:
```

```
8)      include 'stdhep.inc'
        integer xdr_mcfast_stdhep
        external xdr_mcfast_stdhep

4)      integer BLOCKCER, mcfxdr_CerenkovPrelim
        external MCFIO_xdr_CerenkovPrelim
        PARAMETER (BLOCKCER=16001)

9a)     if( mcfio_Block(ostr_disk,
     &      MCFIO_SDTHEP,xdr_mcfast_stdhep ) .lt. 1) goto 91
        if (mcfio_Block(ostr_disk,
     &      BLOCKCER, MCFIO_xdr_CerenkovPrelim ).lt. 1) goto 91

10)     call mcfio_SetEventInfo(ostr_disk,
     &      MCFIO_STORENUMBER, 236)

9b)     if (mcfio_NextEvent(ostr_disk) .lt. 1) goto 91

        return
 9)     print *, ' I/O problem for stream to tape '
        ???
```

9. For sake of simplicity, the subroutine calls for input and output are identical. Since the MCFIO stream index points to an internal data structure that carries the properties of the stream, there is no confusion. The blocking routine mcfio_Block must be called *before* the routine mcfio_NextEvent. NextEvent, in the context of an output stream, flush the completed event to tape or disk.

10. Note that the event number, or other unambiguous event descriptors are not specified throught NextEvent. This can be done through a specific call, similarly to getting information about the event. By default, upon writing, if no specific setting is done, all event identifiers remain unchanged except the event number, which gets increased by 1.

2.4 Termination (in user_endjob)

```
 Code:

11)      call mcfio_Rewind(istr_old)
11*)     call mcfio_Close(istr_old)
         call mcfio_Close(0)
```

11. Input streams can be rewound at any time. The user must also close the output streams before exiting, as one needs to flush out the last few buffer(s), update the last event table and update the file header (only if the stream is direct access). A call *mcfio_close(0)* closes all the active streams. Once a stream is closed, all pertinent information about that stream is lost. The user may re-open the same file, if need be.

2.5 Other Examples and related Testing

Seven short testing programs have been written, and can be used as examples. The first one *t1*, writes a short file, with only STDHEP COMMON. Event number 13 looks interesting, he received a non-trivial trigger mask. Example *t2* read the file generated by *t1* and dumps some STDHEP contents. Example *t3* will look specifically for a given type of trigger, event 13 should be selected. Example *t4* writes a new file (test4.dat), with two different blocks. Non-trivial values for the store numbers are given.

3 NTuples

Implemented only in V2.0. V2 is entirely backward compatible: the poetry presented above is still applicable. Ntuple declarations and filling can be safely added to existing code, without re-organization or limitations on the type of Blocks defined above.

The coding and maintenance of an ever growing number of XDR filters, even if automatically generated through dbin, was not considered an acceptable, long term solution to data machine independent persistency. Meanwhile, the "generalized HBOOK4 Ntuples" were found to be a successful tool at CERN, as well as at Fermilab. The underlying data model of such NTuples is essentially a loosely structured F77 COMMON, which is instanciated one time per HEP *event*. The internal organization of the data on an F77 direct access file can be optimized for histogramming a particular variable within such a data block ("column-wise" Ntuple), or conversely, to read out the entire instance of the COMMON for a selected class of events. However, HBOOK4, being an F66 based software package, can not easily take advantage of modern UNIX facilities, such as true dynamical allocation of virtual memory segments, XDR data representation and so forth. Thus, a complete re-writing of this package is necessary. In addition, one would like to refine and specialized the Ntuple data model a bit, and make the implementation more robust, allowing for versioning, for instance. An other important goal is to provide a way to safely cross the FORTRAN - C language barrier, at the data level. At a later stage, data validation could be added. Finally, doing this the most efficient way while maintaining a machine independent data representation is a real challenge.

3.1 The MCFIO NTuple data model

A typical data structure in HEP, particularly in heavy quark decay, is actually fairly simple, and does not feature a lot of advanced data structure arrangement familiar to C programmers such as trees of lengthy, arbitrary depth,[2] circular structures and so forth. In practice, particles decay or production dynamics are studied at a well defined stage of the hadronization or weak decay process, involving a limited number of levels in the global event hierarchy: one typically has a bunch of tracks, lumped into a Vertex or sub-event. Some variables of fixed size described the property of the vertex, such as position in space, others are assigned to tracks, such as momentum. The number of track in each instance of this sub-event may vary from event to event.

Thus, a typical physics analysis is often based on a two-level data struc-

[2]The STDHEP package features such a decay tree display, which is indeed an exception

ture: a fixed size part, containing at least a key variable, which tells us how many tracks are present in a particular instance (This variable will be referred to as the Multiplicity), and a variable size part, containing the "tracks". In order to satisfy F77 requirements, (and/or save time freeing and reallocating the data structure), it is a good tradition to place an upper limit to this "track" multiplicity; The vertex or the event is judged unusable if a too high multiplicity is found. Thus, in memory, the structure could take a fixed space. This is not a strict requirement, the C application interface allows for user driven dynamic allocation of the space reserved for the variable sized part of the data structure. In order to save disk-space, one may wish to write out only the relevant information, which typically varies in size on an event by event basis.

In our data structure, a given variable is therefore indexed, or, alternatively, is instanciated only once. In our example, we have only one value for the Vertex chi-square, while the track distance of closest approach to this vertex is a property of the track and is therefore indexed. Once again this type of data structure does not support arbitrarily nesting of substructures: we simply stop at one level deep:$vertex- > tracks$. However, within the context of MCFIO, one can build a hierarchical tree of multiple instances of such generalized Ntuple Data structure, pretty much the same way one can define a tree for a set HistoScope items (Histograms, memory based Ntuples). If the user wishes to chain various MCFIO Ntuple instances together, he has to do on his own, no data integrity tools are provided. He simply has to include the relevant "POINTER" variables in the fixed size part of the data structure, and relocated these pointers explicitly in his code.

Currently, MCFIO places rigid boundaries between HEP "events": there are simply no provision to assemble fractions of events together into some higher level data structures. One could think it would be useful to collect all the track momenta, for all events within the stream, into some large array that can be quickly scanned and histogrammed. Such utilities could be added at later releases, however, ultimately, it is my opinion that the physicists will spend most of his time studying one event at time, respecting basic Quantum Mechanics rules (e.g., it does not make much sense to compute masses based on tracks from different events! And sometimes, this is particularly true even if one is studying backgrounds, due to multiplicity variations and coherence effects!). However, once again, the API is flexible enough to allow the user to write code that would save part of an event data, to be used at a later stage.

3.2 The implementation of MCFIO NTuples in FORTRAN or C

Although our Ntuple Data Model is quite simple, in VAX FORTRAN (or F90) and in C, it already has multiple possible implementations: one could treat the information about tracks (indexed variables) in parallel arrays. In rather plain FORTRAN[3], one has :

```
        INTEGER MAXNTR
        PARAMETER (MAXNTR = 10)
        CHARACTER*16 VNAME
        REAL POSXYZ(3), CHI_V, PARAMTR(8,MAXNTR)
        DOUBLE PRECISION DIST_CLOSEST(MAXNTR)
COMMON/MYVERTEX/ NTR_IN_VERTEX, VNAME, POSXYZ, CHI_V,
     &     PARAMTR, DIST_CLOSEST
```

where PARAMTR, the track parameters of relevance in this study, and DIST_CLOSEST, the closest distance approach are multidimensioned arrays, with the slowest varying index being common to both variables, and is mapped to the track number index, running from 1 to MAXNTR. This type of COMMON has been extensively used in E687, and probably in other experiments as well. However, in more modern languages, the following construct is a bit more rational:

```
     structure /MYVERTEX_t_struct/
            Real                PARAMTR(8)
            Double Precision  DIST_CLOSEST
      end structure
     structure /MYV_struct/
            integer          NTR_IN_VERTEX
            character*16     VNAME
```

[3]Except long variable names, good habits die hard!

```
        Real            POSXYZ(3)
        Real            CHI_V
        record /MYVERTEX_t_struct/ track(MAXNTR)
end structure
```

The C syntax would be different, of course, but the semantics are the same, Thus, one clearly needs a little utility that translates these Data Descpritors from one language to another. Dbin, written by Torre Wenaus, does that, in addition it automatically creates intialization and relevant I/O codes (including XDR filters). In MCFIO, the data types used in structures have been extended to support all VAX FORTRAN (or SGI FORTRAN) variable types within a structure. A small GUI has been written to define, clone, maintain these data descriptors. The name of this utility is *ntuBuild*. Through *ntuBuild*, the user creates an ASCII file containing this information (saved in the original dbin format), and generates the corresponding ".inc" or ".h" include file. He can then include this information in his/her MCFIO user code.

This type of GUI, producing exclusively textual information, seems a-priori superfluous. However, it nicely implements the much needed verification mechanism to make sure the include file(s) are consistent with the model described above. Such a GUI has been written for the convenience of the MCFIO authors, as well as for the end-user: no further verifications are needed once a user-written MCFIO application uses the ASCII files produced by *ntuBuild*. Finally, versioning of these data structures has been implemented: the user must declare explicitly and update the version of his/her data structure. The tool has been automated, but it is the responsability of the user to decide which ASCII token is used, MCFIO simply ensures consistency between the token found in the *ntuBuild* output file and the one declared by the user in the MCFIO application, at run time. This will become more apparent in the example and reference sections below. Let us start by briefly describing the *ntuBuild* MOTIF-based GUI, then discuss a brief example/tutorial on how to invoke, store and retrieve these NTuples.

3.3 ntuBuild

Since a complete online help facility is provided (or will be...) , this description will be kept extremely brief. To start this application, you type from a UNIX shell:

```
fnpspg.fnal.gov % ntuBuild <filename>
```

The argument *filename* is optional, it simply refers to an existing ddl file describing a previously defined data descriptor. Note: these dbin files must have been created by *ntuBuild* in order to be read in by *ntuBuild*. Once again, this is to ensure consistency. The little structure mentioned above has been entered into *ntuBuild*, as shown on figure 1 and 2.

3.4 ntuBrowse

Once a Data Description File (.ddl) has been composed, along with it's autogenerated F77 or C include file, the developers can include them in his application, and data can be generated. An MCFIO file can then be browsed to check this data. As the Ntuples are self-described data structures, only the data file itself (which has in it's header the full content of the .ddl file) is needed to interpret that data. Thus, an MCFIO data browser becomes a real possibility. To use it, simply type :

```
fnpspg.fnal.gov % ntuBrowse <filename>
```

The argument *filename* is optional, it simply refers to an existing MCFIO file containing the data. You can check the file header, scan Ntuple data and eventually create a volatile HistoScope histogram to "pre-analyze" the data.

3.5 Tutorial

Let us now incorporate this data in a program that generates the data, e.g. creates an Ntuple. After opening a stream for write, direct access[4], at initialization time, the user declares these NTuples to this existing stream:

[4]sequential access might be supported at a later release, eventually

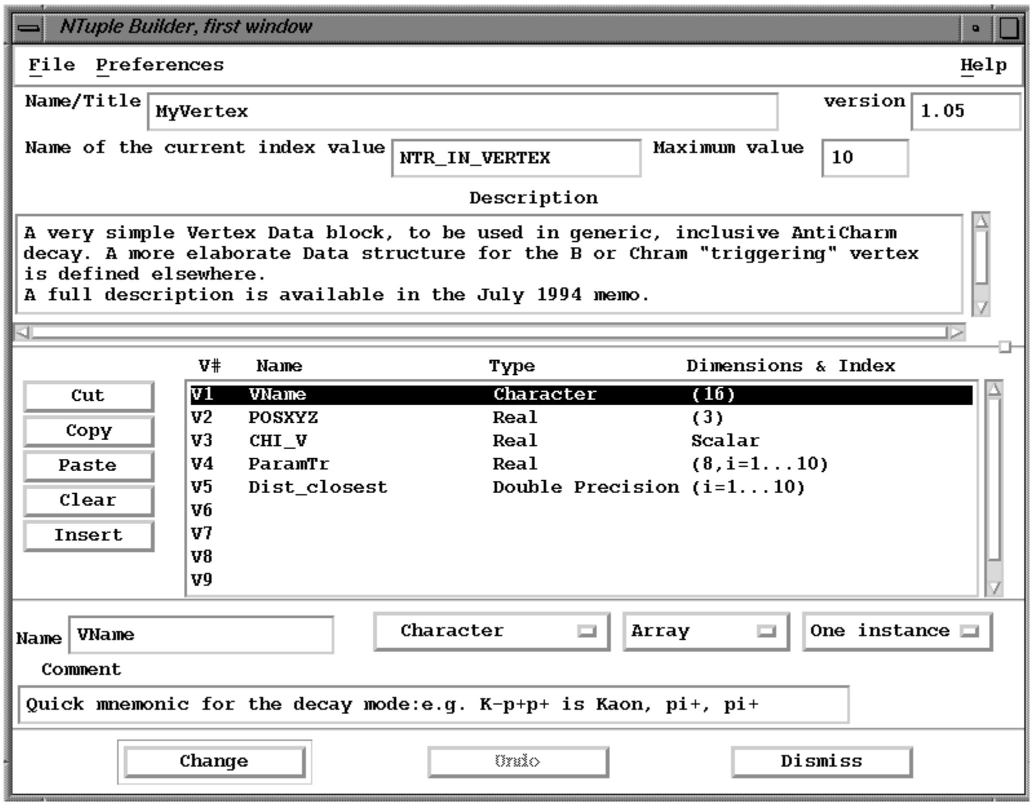

Figure 1: The ntuBuild panel, displaying the small data structure mentioned in the text. The language syntax is Fortran, the comments fields have been filled. This structure is now ready to saved.

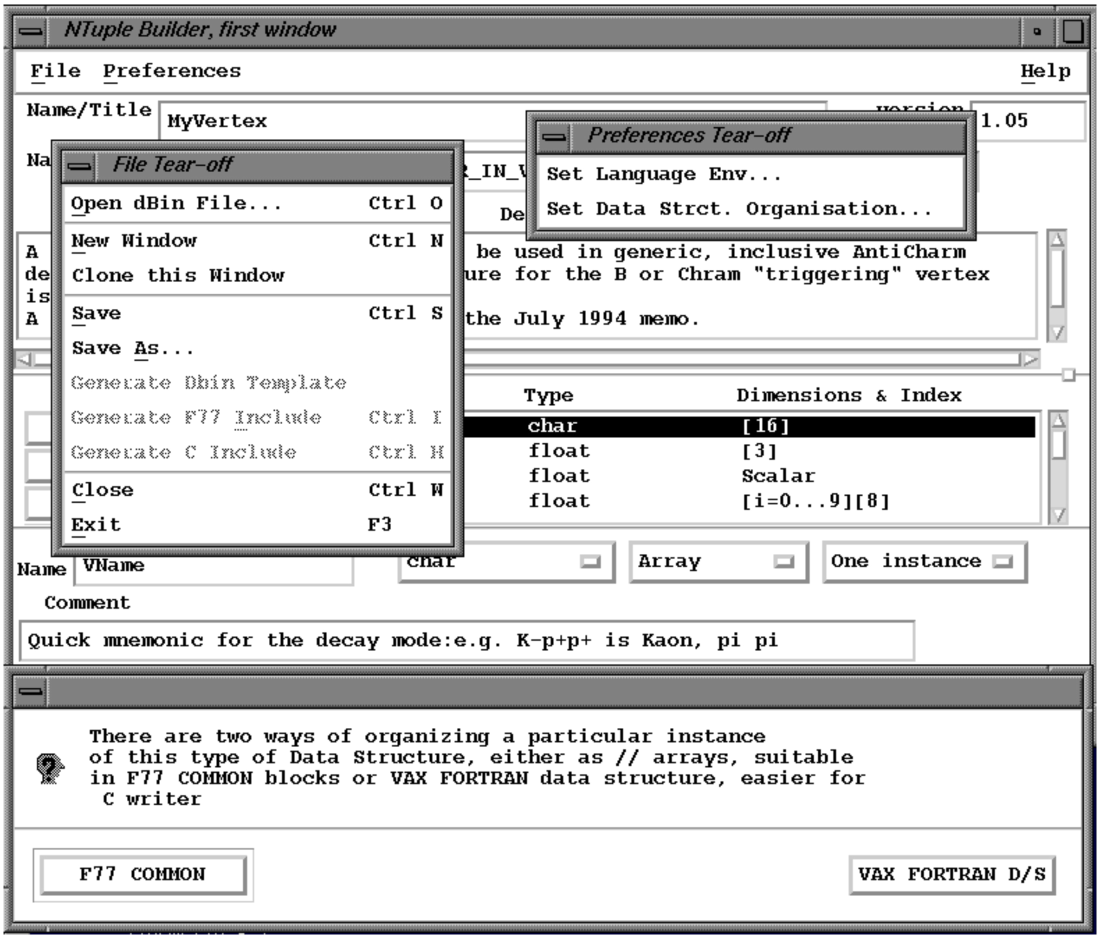

Figure 2: The ntuBuild panel, displaying the same structure, using the C syntax, The File and Preference pulldown menu have been teared of. The last item in the preference menu has been activated, displaying the Dialog Box that lets you set the structure organization (bottom). Note also, that at this stage, the structure could be saved, and once that is done, the include files can be generated.

```
integer ostr_all
character *60 filename
character *80 title
integer blkids(3), ndeclared
integer NUMTRIES
PARAMETER (NUMTRIES = 1500)
include 'myvertex.inc'
include 'myBstuff.inc'


ostr_all = mcfio_OpenWriteDirect(filename, title, %val(0),
     &            NUMTRIES, blkids, 0)
        if (ostr_all .eq. -1) then
           print *, ' Can t open output file, give up '
           stop
        end if
        blkids(1) = mcfio_DeclareNtuple(22,
     &                               'AntiCharm Vertex, Central',
     &                               'Vertex/Tertiaries',
     &                                 ostr_all, 'myvertex.db')

        blkids(2) = mcfio_DeclareNtuple(122,
     &                               'AntiCharm Vertex, Forward',
     &                               'Vertex/Tertiaries',
     &                                 ostr_all, 'myvertex.db')
     .......
        blkids(3) = mcfio_DeclareNtuple(55,
     &                               'AntiCharm Vertex, Central',
     &                               'Vertex/BigStuff',
     &                                 ostr_all, 'myBstuff.db')
     ......
        ndeclared = ....
        if (mcfio_EndDeclNTuples(ostr_all) .ne. ndeclared) then
            print *, 'Problem with stream initialisation '
            stop
        end if
```

This Application User Interface is very similar to the HistoScope one: each Ntuple can be characterized by a user-chosen numerical token, the UID (first argument in *mcfio_DeclareNtuple*, a title (2nd arg.) and a category string). Each declared Ntuple must be associated to a stream. And finally, one has to refer to the Data Descriptor, saved in the file *myvertex.db*. Note that the same Vertex Data Model can be used for the Forward or Central analysis, thus, we made reference to *myvertex* twice. This declaration routine returns an integer, used later to quickly access the NTuple.

It is necessary to cleanly end this declaration phase (*mcfio_EndDeclNTuples*), in order to write all this information into the file header of the stream. We now are ready to fill these NTuples, in the event loop.

```
myvertex_c.version = '1.05'
        do i = 1, NUMTRIES
          call find_central(i,....)
          if (mcfio_NTuple(ostr_all,blkids(1), myvertex_c.version)
     &           .eq. -1) then
            print *, 'Problem with Encoding Forward Vertex '
            stop
          end if

          call find_forward(i,....)
          if (mcfio_NTuple(ostr_all,blkids(2), myvertex_c.version)
     &           .eq. -1) then
            print *, 'Problem with Encoding Forward Vertex '
            stop
          end if
```

The first line, before the event loop start is mandatory, as it implements the check of the versions upon write. Note that the token "1.05" can be seen on the upper right corner in panel display on figure 1. Thus, this information has been already downloaded to this MCFIO job. However, the code displayed above is a different file, whose content could become incompatible with *myvertex.db, myvertex.inc*. Thus, this check is non-trivial.

A better mechanism that would detect incompatibility between tokens could be devised in coordination with the code management system, if dummy externals are introduced, could even be detected at link time rather than at run time. However, the first in implementing versioning is to make provisions for it at the data level, and that's what is done in this release.

The routines *find_central,* find_forward is your own business, it is assumed in this context that they will fill (or re-fill) the COMMON MYVERTEX. The subroutine *mcfio_NTuple* downloads the content of this COMMON into the stream *ostr_all*, and implements one "fill" for the NTuple referred to by the index blkids. Note that a third argument is needed: the address of the first word of the COMMON MYVERTEX. For FORTRAN programs, this is a fixed address, it can not change. However, since a C API is also implicitly provided (remember that the MCFIO package is written in C), this allows C writer to dynamically allocate the heap for the MYVERTEX structure, and keep track of multiple instance of that structure:

```
p_central = (MyVertex_struct *) malloc(sizeof(MyVertex_struct));
     find_central(i,p_central,....);
     if (mcfioC_NTuple(ostr_all,blkids, p_central->version)
&           ==  -1) {
       printf ("Problem with Encoding central Vertex ");
       exit(0);
    }

p_forward = (MyVertex_struct *) malloc(sizeof(MyVertex_struct));
     find_forward(i,p_forward,....);
     if (mcfioC_NTuple(ostr_all,blkids, p_forward->version)
&           ==  -1) {
       printf ("Problem with Encoding Forward Vertex ");
       exit(0);
    }
```

No special action is necessary to close an MCFIO stream that contains NTuples. Once a file has been generated, we now can read back this information. The initialization stage looks like:

```
istr_old = mcfio_OpenReadDirect('testN1.dat')
call mcfio_InfoStreamInt(istr_old, MCFIO_NUMNTUPLES, numNTuples)
print *, ' Number of Ntuples for this stream ',  numNTuples
c
nn = mcfio_GetNTupleIds(istr_old, blkids, MAX_NUMNTUPLE)
if (nn .gt. MAX_NUMNTUPLE) then
   print *, ' Too many NTuple on this stream, gave up'
   stop
end if
c
c We are interested only in the forward stuff for now, I remember
c that user ID is 122, I'll check the title and category to make sure.
c
        ntu_forward = -1
do i = 1, numNTuples
    print *, ' Ntuple Sequential number ',i
    ll =  mcfio_GetNtupleCategory(blkids(i), answer)
    print *, '         Category  ', answer(1:ll)
    ll =  mcfio_GetNtupleTitle(blkids(i), answer)
    print *, '         Title  ', answer(1:ll)
    ll =  mcfio_GetNtupleName(blkids(i), answer)
    print *, '         Template Name (.db file) ', answer(1:ll)
    if (mcfio_GetNTupleUID(blkids(i)) .eq. 122)
     &           ntu_forward = blkids(i)
end do
c
If (ntu_forward .eq. -1) then
    print *, ' Can not find Ntuple number 122 '
    stop
end if
```

As the Data Descriptors are saved in the stream itself, there is no point referring to the *myvertex.db* file; the *OpenReadDirect* call is identical to the one used in the previous release. However, we need to know something about the NTuple defined in that stream, and get the handle to access contents of these structures. Hence the list of calls made after the Open statement. Once the variable *ntu_forward* is filled, we can start the event loop, and dumping the contents from the stream directly into the COMMON MYVERTEX for further analysis:

```
        do i = 1, numevts_real
          if (mcfio_NextEvent(istr_old) .ne. MCFIO_RUNNING) then
             print *,'  Error reading next event '
             call mcfio_InfoStreamInt(istr_old, MCFIO_STATUS, inum)
             if (inum .eq. MCFIO_EOF ) then
                 print *, ' Reached EOF, Close and Stop '
         call mcfio_Close(istr_old)
       stop
             else
               print *, ' Unrecognized Status, stat = ',  inum
             end if
             stop
          end if

          if (mcfio_NTuple(istr_old, ntu_forward, myvertex_c.version)
     &            .eq. -1) then
            print *, 'Problem decoding Ntuple with forward Vertex'
            stop
          else
             call analyze_forward(...)

          end if
        end do
```

Note that, as for the *mcfio_Block* routine, the synopsis for *mcfio_NTuple* is independent of the indirection of the stream. This is the minimal code one could use to study an MCFIO stream. However, if for some reasons one is interested in only one type of vertices, and wishes to decode the stream a bit faster, consider replacing the call *mcfio_NTuple* by the following code:

```
      if (mcfio_NTupleMult(istr_old, ntu_forward, myvertex_c.version)
     &           .eq. -1) then
        print *, 'Problem decoding Multiplicity from Ntuple ntu_forward'
        stop
      end if
      if (myvertex_c.NTR_IN_VERTEX .eq. 3) then
         if (mcfio_NTuple(istr_old, ntu_forward, myvertex_c.version)
     &           .eq. -1) then
            print *, 'Problem decoding Ntuple with forward Vertex'
            stop
           else
            call analyze_forward(...)

           end if
       end do
```

In this example, the user only wishes to look at 3-prong vertices. Other types of selection is possible, as every variable in the structure is directly addressable, using the routines *mcfio_NTupleVar, mcfio_NTupleSubVar, mcfio_NTupleSubStruct*. These routines are only valid for input streams, obviously. Let us also mention that they are optimized for Read performance, not user-friendlyness in reporting error messages. In fact, few checks on the arguments are made in order to gain speed. Only the requested variable(s) are decoded[5]. Note that XDR seeking(s) must be done to access the correct location in the stream.

4 Reference Section

4.1 Core routines

4.1.1 mcfio_init

```
      subroutine mcfio_init
```

[5]However, if the variable is an array, the entire array is decoded.

Initialization routine; no arguments. It is no longer necessary to call this routine, it is done implicitely upon opening the first I/O stream. It can be safely disregarded, but it does not hurt to leave your code as it is.

4.1.2 mcfio_OpenReadDirect

```
integer function mcfio_OpenReadDirect(filename)
```

Opens an MCFIO stream for input, direct access. *filename*, a character string (max. length 255), must refer to an existing standard UNIX file resinding on disk. Returns -1 if unsuccessful, or the MCFIO stream index used for future reference and access for this stream. The user must keep this information in its code.

4.1.3 mcfio_OpenReadMapped

```
integer function mcfio_OpenReadMapped(filename)
```

Opens an MCFIO stream for input, direct access, using virtual memory mapping. *filename*, a character string (max. length 255), must refer to an existing standard UNIX file resinding on disk. Returns -1 if unsuccessful, or the MCFIO stream index used for future reference and access for this stream. The user must keep this information in its code. This routine allows the system to consider the user file as an extension of the virtual memory space allocated to the process. As a consequence, while accessing information, one simply rely on the paging/swapping UNIX system. The rather slow UNIX STDIO system is bypassed, reading a file may on some system be up to 5 to 7 times faster than the ReadDirect routine, which is based on STDIO. However, there is at least one known drawback to this memory mapping: if the file is incomplete or corrupted, an illegal read operation (e.g. beyond the end of the file, for instance) results in a segmentation violation fault, with no warning nor chance of recovery.

4.1.4 mcfio_OpenWriteDirect

```
      integer function mcfio_OpenWriteDirect(filename,
     &                       title, comment,
     &                       numevt_expected, block_ids, numblocks)
```

Opens an MCFIO stream for output, direct access. *filename*, a character string (max. length 255), will refer to a standard UNIX file created on disk. Returns -1 if unsuccessful, or the MCFIO stream index used for future reference and access for this stream. The user must keep this information in its code. Additional input arguments are :

- *title:* Character string, maximum length of 80 characters, a mnemonic name for the stream.
- *comment:* A comment line, maximum length of80 characters. Both *title* and *comment* are optional, e.g., the value *%val(0)* is allowed (interpreted in C as a NULL pointer).
- *numevt_expected:* Integer. The expected number of events for this output file.
- *block_ids:* Integer array. The expected block identifier list for this run. This array must have a length of at least *numblocks*
- *numblocks:* Integer. The number of blocks in this list.

4.1.5 mcfio_OpenReadSequential

```
      integer function mcfio_OpenReadSequential(device,
     &            label, filenumber)
```

Opens an MCFIO stream for input, sequential access. The stream corresponds to a sequence of physical records, written to tape or to disk. Returns -1 if unsuccessful, or the MCFIO stream index used for future reference and access for this stream. The user must keep this information in its code. The Input device selection is controlled by the first two arguments in the most inelegant manner:

- *device:* A character string (maximum length 255), will refer to a sequential UNIX device or a UNIX filename, depending on the value of *label*. If *label* has the value "Disk", then it is understood the *device* points to a UNIX file name, and obviously no label is written. Else, *device* points to a tape drive, for instance */dev/rmt/tps0d5nrnsv.8500* points to an 8mm EXABYTE device, located on SCSI bus 0, device number 5, no rewind device, no swap, used in the 8500 mode.

- *label:* Character string. For tapes, a standard ACII label tape, made of 6 non-control characters (e.g. ZB1025).

- *filenumber:* Integer. The sequential file number on the tape. This corresponds to the "Sequence" key in RBIO.

4.1.6 mcfio_OpenWriteSequential

```
      integer function mcfio_OpenWriteSequential(device,
     &           label, filenumber, title, comment,
     &            numevt_expected, blockids, numblocks)
```

Opens a stream opened for output, sequential mode. Returns -1 if unsuccessful, or the MCFIO stream index used for future reference and access for this stream. The user must keep this information in its code.

- *device:* A character string (maximum length 255), will refer to a sequential UNIX device or a UNIX filename, depending on the value of *label*. If *label* has the value "Disk", then it is understood the *device* points to a UNIX file name, and obviously no label is written. Else, *device* points to a tape drive, for instance */dev/rmt/tps0d5nrnsv.8500* points to an 8mm EXABYTE device, located on SCSI bus 0, device number 5, no rewind device, no byte swap, used in the 8500 mode.

- *label:* Character string. For tapes, a standard ACII label tape, made of 6 non-control characters (e.g. ZB1025).

- *title:* Character string, maximum length of 80 characters, a mnemonic name for the stream.

- *comment:* A comment line, maximum length of 80 characters. Both *title* and *comment* are optional, e.g., the value *%val(0)* is allowed (interpreted in C as a NULL pointer).

- *numevt_expected:* Integer. The expected number of events for this output file.

- *block_ids:* Integer array. the expected block identifier list for this run. This array must have a length of at least *numblocks*

- *numblocks:* Integer. The number of blocks in this list.

4.1.7 mcfio_DeclareNTuple

```
      integer function mcfio_DeclareNTuple(uid, title, category,
   &           ostream, db_filename)
```

Declares an NTuple template Data Descriptor (DD) to a specific Output Stream. Invalid for Input stream, as these DD's are already part of the MCFIO input streams. The *uid, title* and *category* are additional identifiers used to locate the NTuple in an overall hierachical structure of the MCFIO event. All arguments are input:

- *uid:* Integer. uid defines a user-specified identification number which can later be used along with the category to specify the item created.

- *title:* Character, (length less than 80 characters), a title that can be displayed in the MCFIO data browser. Used also as a mnemonic for the user to distinguish various intances based on the same DD template.

- *category:* Character (length less than 255 characters)), a string defining a hierarchical "location" for the NTuple. Subcategories can be specified using the "/" character as in Unix file specifications. If a category or subcategory does not exist, it is created. If the category argument is blank (either " ", or ' ') the NTuple will appear in the top level category.

- *ostream:* Integer. The Stream index, as returned by a *mcfio_OpenWriteDirect* call.
- *db_filename:* Character. A UNIX filename, pointing to a dbin template containing the Data Descriptor. These files are produced by the ntuBuild utility, or by the MCFIO browser.

Returns an id number NTupleId to be used at filling time, to refer to the NTuple.

4.1.8 mcfio_EndDeclNTuples

```
integer function mcfio\_EndDeclNTuples(ostream)
```

Completes the declaration phase of the Ntuples for a specific output stream. Invalid for input stream. This call must be placed in order to store the DD(s) onto the streams. The file header is updated, the stream is now ready to accept data, the event loop starts. *ostream* is an Integer, Input, the stream index as returned by the *Open_WriteDirect* call.

4.1.9 mcfio_NextEvent

```
integer function mcfio_NextEvent(iostream)
```

The access routine to set or get an event to or from a stream. *iostream* is integer, returned by an MCFIO_Open* routine. For input direct access stream, mcfio will get the XDR position for the next event, either from the internal table, or, if need be, will get the next event table in the stream. For input sequential, RBIO will be called to get a string or tape records, into virtual memory. The XDR event filter will allow us to decode the event number, store, and so forth. For output streams, this will "close" the current event. If direct access, MCFIO will update the XDR locators on the file. If sequential, a set of records will be written on tape. In addition, the current event number for this output stream will be incremented by one, and a new (empty!) XDR event block will be generated to be ready to flush the next event.

If the next event has indeed been transferred, mcfio_NextEvent returns MCFIO_RUNNING. If not, mcfio_NextEvent returns -1 if an I/O error has been encountered, or MCFIO_EOF is the stream is Input, and ran past the End of File mark.

4.1.10 mcfio_Block

```
      integer function mcfio_Block(iostr,
   &                     blockid, XDR_filtercode)
```

The access routine to set or get a data block to or from a stream. Data can be directly decoded or encoded from the targeted COMMON block without extraneous copying (direct access streams). Data is converted through a set of XDR filters. As module *NextEvent*, this routine is valid for input and output streams. Note that, for input streams, *Block* must be called after *NextEvent*, while for output streams, it must be called before. All arguments are input:

- *iostr:* Integer. the stream index, as returned by an MCFIO_Open* call
- *blockid:* Integer. The unique block identifier number. Used essentially for self consistency checks.
- *XDR_filtercode:* Integer, external (to be casted later on to something else..) The XDR external routine to convert the data.

The return code is the same as *NextEvent*: -1 if a problem occurred, else 1.

4.1.11 mcfio_NTuple

```
      integer function mcfio_NTuple(ioStream,
   &                    NtupleId, Struct_name.version)
```

Reads or writes an NTuple (referred to by the id, *NTupleId*) instance onto/from the MCFIO stream referred to by *iostream*. To be called from the event loop. The last argument is in essence the address of the structure or COMMON from/to the data is coming or going. Note that, for output streams, it is necessary to update the *Struct_name.version* Character token to the correct version token found in the Data Descriptor. All arguments are input:

- *ioStream:* Integer. The MCFIO stream index, as returned by an *mcfio_Open** call.
- *NTupleId:* Integer. The NTuple Id number, as returned by a call to *mcfio_DeclareNtuple* (Output streams) or *mcfio_GetNTupleIds* (Input Streams).
- *Struct_name.version:* Character. In practice, this is simply used as a POINTER, MCFIO will start encoding or decoding based on his knowledge of the Date Descriptors. The version token being always the first variable in the structure or COMMON.

The return code the same as *NextEvent*: -1 if a problem occurred, else 1.

4.1.12 mcfio_NTupleMult

```
      integer function mcfio_NTupleMult(iStream,
     &                   NtupleId, Struct_name.version)
```

Reads the Ntuple multiplicity (referred to by the id *NTupleId*) from stream referred to by *iStream*. By multiplicity, we mean number of sub-structures or "tracks" if the Ntuple object is a "vertex". All the input are arguments, already described in *mcfio_NTuple*.

4.1.13 mcfio_NTupleVar

```
      integer function mcfio_NTupleVar(iStream,
     &                   NtupleId, ivar,  Struct_name.version)
```

Reads a specific variable in an Ntuple (referred to by the id *NTupleId*) from the stream referred to by *iStream*. A variable can be just a scalar or an array defined in the *ntuBuild* panel. The argument *ivar* refers to the variable number on that panel, once the DD has been saved. If the organization of the Ntuple is the type parallel arrays, and if the variable is indexed (e.g. belongs to a "track" rather than to a "vertex"), all instances of the variables are retrieved. This routine is not applicable to an indexed variable used in the other type of organization based on substrucure. No checks are made, to ensure optimum speed!.

4.1.14 mcfio_NTupleSubVar

```
      integer function mcfio_NTupleSubVar(iStream,
     &              NtupleId, ivar,  subIndex,  Struct_name.version)
```

Reads a specific indexed variable in an Ntuple (referred to by the id *NTupleId*), organized by sub-structures rather than parallel arrays, from the stream referred to by *iStream*. A variable can be just a scalar or an array defined in the *ntuBuild* panel. The argument *ivar* (Integer) refers to the variable number on that panel, once the DD has been saved. The argument *subIndex* (Integer) refers to the index of the substructure (e.g. the "track") index. No checks are made, to ensure optimum speed.

4.1.15 mcfio_NTupleSubStruct

```
      integer function mcfio_NTupleSubStruct(iStream,
     &              NtupleId, subIndex,  Struct_name.version)
```

Reads a specific substructure instance (e.g. all the information concerning a specific "track") in an Ntuple (referred to by the id *NTupleId*), organized by sub-structures rather than parallel arrays, from the stream referred to by *iStream*. *subIndex* (Integer) is the "track" index.

4.1.16 mcfio_Close

```
subroutine  mcfio_Close(iostr)
```

Closes a stream. This is mandatory if the user expects to re-open the file, or, to complete the file header if the stream is output, direct access. Only 1 input argument:

- *iostr:* Integer. The stream index, as returned by an MCFIO_Open* call. *iostr = 0* is also valid, in which case all active streams are closed.

```
subroutine  mcfio_CloseSequentialFile(iostr)
```

Closes a file on a sequential stream. This call is only relevant for tape device, when the user wishes to keep the stream active. The file on the tape is closed (e.g. an EOF is written), and the user may call mcfio_OpenWriteSequential (or Read, but the stream must be kept in the same mode, e.g., an input stream can not be turned into an output stream) again, pointing to the same device and using the same label. This avoids closing the stream entirely, and having to reopening it, a sequence that takes time in RBIO.

- *iostr:* Integer. The stream index, as returned by an MCFIO_Open* call.

```
subroutine  mcfio_CloseSequentialTape(iostr)
```

Closes a sequential stream for good. All previous knowledge about this stream is lost. Upon reopening the same tape device, a rewind will occur. Can also be used if the sequential stream points to a UNIX file.

- *iostr:* Integer. The stream index, as returned by an MCFIO_Open* call.

4.2 Utilities

This subsection is devoted to ancillary modules used to set or obtain information about a stream, an event, a Block or an Ntuple.

4.2.1 mcfio_PrintDictionary

```
subroutine  mcfio_PrintDictionary
```

Prints the current list of parameters controlling the streams, describing the events, blocks and so forth...

4.2.2 mcfio_InfoNumStream

```
integer function  mcfio_InfoNumStream(iostrs, nmax)
```

Returns the current number of active streams and their corresponding indices in the integer array *iostrs*, which must be dimensioned to at least nmax (to be used in case you lost the Stream indices in your program).

4.2.3 mcfio_InfoStreamInt

```
subroutine  mcfio_InfoStreamInt(iostr, key, ival)
```

Returns the current integer value for a parameter associated with a given stream. The list of such parameters and their key values can be found by printing the dictionary. For instance, after opening an input stream, one can find the number of events in the file using the key MCFIO_NUMEVTS. Arguments are straightforward, two input, one output

- *iostr:* Integer. The stream index, as returned by an MCFIO_Open* call.

- *Key:* Integer. The keyword, as defined in the Dictionary, implemented in the *mcfio.inc* include file.
- *ival:* The returned value.

4.2.4 mcfio_InfoStreamChar

```
subroutine  mcfio_InfoStreamChar(iostr, key, answer)
```

Returns the current character string value for a parameter associated with a given stream. The list of such parameters and their key values can be found by printing the dictionary. For instance, after opening an input stream, one can find the title corresponding to the file using the key MCFIO_TITLE. Arguments are straightforward, two input, one output.

- *iostr:* Integer. The stream index, as returned by an MCFIO_Open* call.
- *Key:* Integer. The keyword, as defined in the Dictionary, implemented in the *mcfio.inc* include file.
- *answer:* Character string, the returned answer.

4.2.5 mcfio_InfoEventInt

```
subroutine  mcfio_InfoEventInt(iostr, key, ival)
```

Returns the current integer value for a parameter associated with the current event for a given stream. The list of such parameters and their key values can be found by printing the dictionary. For instance, after opening an input stream and reading an event one can find the number of blocks in this event using the key MCFIO_NUMBLOCKS. Arguments are straightforward, two input, one output

- *iostr:* Integer: the stream index, as returned by an MCFIO_Open* call.

- *Key:* Integer: The keyword, as defined in the Dictionary, implemented in the *mcfio.inc* include file.
- *ival:* the returned value.

4.2.6 mcfio_InfoEventChar

```
      subroutine  mcfio_InfoEventChar(iostr, key, answer)
```

Returns the current character string value for a parameter associated with the current event for a given stream. The list of such parameters and their key values can be found by printing the dictionary. See *InfoStreamChar* for argument list.

4.2.7 mcfio_GetBlockName

```
      subroutine  mcfio_GetBlockName(blockid, answer)
```

Returns the Block title/name in the character string answer for a specific block referred to by integer *blockid*.

4.2.8 mcfio_DefineUserBlock

```
      subroutine  mcfio_DefineUserBlock(blockid, name,
     &              XDR_filter, current_size)
```

Obsolete, no effect..

4.2.9 mcfio_GetNtupleIds

```
      integer function mcfio_GetNTupleIds(ioStream,
     &             NtupleIds, MAX_NUMNTUPLE)
```

For a given stream (Input or Output), returns the number of declared NTuples, and fill the array *NTupleIds*, containing the NTuple id's. MAX_NUMNTUPLE is an integer argument set to the size of the *NtupleIds* array (to avoid overwrites, basically).

4.2.10 mcfio_GetNtupleUID

```
integer function mcfio_GetNTupleUID(NtupleId)
```

Returns the UID of a defined or declared NTuple referred to by *NTupleId*.

4.2.11 mcfio_GetNtupleTitle

```
integer function mcfio_GetNTupleTitle(NtupleId, title)
```

Returns the length of the title string and fills that string *title*, for an Ntuple referred to by *NtupleId*.

4.2.12 mcfio_GetNtupleCategory

```
integer function mcfio_GetNTupleCategory(NtupleId, title)
```

Returns the length of the category string and fills that string *title*, for an Ntuple referred to by *NtupleId*.

4.2.13 mcfio_GetNtupleName

```
integer function mcfio_GetNTupleName(NtupleId, title)
```

Returns the length of the Data Descriptor name string and fills that string *title*, for an Ntuple referred to by *NtupleId*. This Data Descriptor name appears as "namw/title" on the *ntuBuild* panel, and indeed refers to a short mnemonic for the Data descriptor, it is also the name of the COMMON block associated with the NTuple.

4.2.14 mcfio_SetEventInfo

```
subroutine  mcfio_SetEventInfo(iostr, key, ival)
```

Sets a requested value for a parameter associated with the current event for a given stream. The list of such parameters and their key values can be found by printing the dictionary. For instance, after opening an output stream and optionally writing some data blocks out, one can set the trigger mask for this event by using the key MCFIO_TRIGGERMASK. Arguments are straightforward, all input:

- *iostr:* Integer. The stream index, as returned by an MCFIO_Open* call.
- *Key:* Integer. The keyword, as defined in the Dictionary, implemented in the *mcfio.inc* include file.
- *ival:* The set value.

4.2.15 mcfio_SpecifiEvent

```
      integer function mcfio_SpecificEvent(iostream,
     &                 event_number, store_number, run_number,
     &                 trigger_mask)
```

The access routine to get a specific event from a direct access input stream. The position of the stream is set to the first event encountered in the stream, starting from the beginning of the stream (e.g., after an implicit rewind), that satisfy the requested event criteria. Arguments (all inputs) are:

- *iostream:* Integer. Returned by an MCFIO_Open* routine.

- *event_number:* Integer. The event number, as set by automatically by MCFIO upon writing the file, or by a call to *mcfio_SetEventInfo*.

- *store_number:* Integer. The store number, as set by a call to *mcfio_SetEventInfo*. item *run_number:* Integer. The store number, as set by a call to *mcfio_SetEventInfo*.

- *trigger_mask:* Integer. The user defined trigger mask, as set by a call to *mcfio_SetEventInfo*.

If one of these arguments is set to 0, this means that no selection is applied on that particular argument. If all are set to 0, this routine will return the first event on the file. If the requested event has indeed been found and its header successfully decripted, mcfio_SpecificEvent returns MCFIO_RUNNING. If not, mcfio_SpecificEvent returns -1 if an I/O error has been encountered, or MCFIO_EOF is the stream ran past the End of File mark.

4.2.16 mcfio_NextSpecifiEvent

```
      integer function mcfio_NextSpecificEvent(iostream,
     &                     event_number, store_number, run_number,
     &                     trigger_mask)
```

Same as *mcfio_SpecificEvent*, but starts from the current position of the stream.

4.2.17 mcfio_Rewind

```
      subroutine  mcfio_Rewind(iostr)
```

Rewinds a stream. For direct access, this is just setting one pointer, while for sequential device, it may take some times. Only one input argument:

- *iostr:* Integer. The stream index, as returned by an MCFIO_Open* call.

4.2.18 mcfio_SetForSaveDecoding

```
subroutine  mcfio_SetForSaveDecoding(Yes_or_No)
```

Relevant only for input stream containing NTuples. By default, MCFIO assumes it knows nothing about the virtual memory locations of the various variables in an Ntuple. MCFIO decodes first the Data Descriptors, and for each of them write a little C program that computes the relative addresses of variable within the structure. This could be time consuming. If the user knows that the NTuples have been written on the same machine, using the same Operating System, and the same compiler, he/she can by-pass that operation and trust the relative address and length found in the file header. By default, it is set for save decoding. *Yes_or_No = .TRUE.*. It is often possible to speed up the initialization phase by call this routine with *Yes_or_No = .FALSE.*.

4.3 User Defined XDR filter for private Data Blocks

The simplest way to start is to consider the routine that filter the STDHEP COMMON block. This module is already part of the current version of MCFIO, you do not have to rewrite it, and can be used as a template. First, we start by translating the FORTRAN COMMON to a C structure:

```
      INTEGER NMXHEP
      PARAMETER (NMXHEP=4000)
      COMMON/HEPEVT/NEVHEP,NHEP,ISTHEP(NMXHEP),IDHEP(NMXHEP),
     &JMOHEP(2,NMXHEP),JDAHEP(2,NMXHEP),PHEP(5,NMXHEP),VHEP(4,NMXHEP)
      INTEGER NEVHEP,NHEP,ISTHEP,IDHEP,JMOHEP,JDAHEP
      REAL PHEP,VHEP
      SAVE /HEPEVT/
C...      NEVHEP       - event number
C...      NHEP         - number of entries in this event
```

```
C...      ISTHEP(..)   - status code
C...      IDHEP(..)    - particle ID, P.D.G. standard
C...      JMOHEP(1,..) - position of mother particle in list
C...      JMOHEP(2,..) - position of second mother particle in list
C...      JDAHEP(1,..) - position of first daughter in list
C...      JDAHEP(2,..) - position of last daughter in list
C...      PHEP(1,..)   - x momentum in GeV/c
C...      PHEP(2,..)   - y momentum in GeV/c
C...      PHEP(3,..)   - z momentum in GeV/c
C...      PHEP(4,..)   - energy in GeV
C...      PHEP(5,..)   - mass in GeV/c**2
C...      VHEP(1,..)   - x vertex position in mm
C...      VHEP(2,..)   - y vertex position in mm
C...      VHEP(3,..)   - z vertex position in mm
C...      VHEP(4,..)   - production time in mm/c
```

... becomes:

```
#define NMXHEP 4000

extern struct hepevt {
int nevhep; /* The event number */
int nhep; /* The number of entries in this event */
int isthep[NMXHEP];  /* The Particle id */
int idhep[NMXHEP];      /* The particle id */
int jmohep[NMXHEP][2];    /* The position of the mother particle */
int jdahep[NMXHEP][2];    /* Position of the first daughter... */
float phep[NMXHEP][5];    /* 4-Momentum, mass */
float vhep[NMXHEP][4];    /* Vertex information */
} hepevt_;
```

We are now in position to write the required function:

```
#include <stdio.h>
#include <string.h>
#include <sys/param.h>
```

```
#include <rpc/types.h>
#include <sys/types.h>
#include <rpc/xdr.h>
#include <limits.h>
#include <float.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "mcfio_Dict.h"
#include "mcf_xdr.h"
#include "stdhep.h"
#define FALSE 0
#define TRUE 1
/*
** These include files are probably needed, to be able to do math,
** get the time and date if you feel like, and most important,
** access the XDR filter protocols. These are described in your
** workstation documentation ( for instance, a SGI documentation set,
**  XDR is described in the IRIX Network Programming Guide.
*/

bool_t xdr_mcfast_stdhep_(XDR *xdrs, int *blockid,
   int *ntot, char** version)

/*  Translate the HEPEVT COMMON block from the STDHEP package to/from
**  an XDR stream. Note that we do not allocate memory, because we fill
**  directly the COMMON.  Also, the mcfio will allocate the space for the
** string version.
**
** Arguments:
**
** XDR *xdr : pointer to an active XDR filter, set by an Open
** routine.
** Blockid : The unique block identifier, in this case,
** must be set to MCFIO_STDHEP.
** Used for consistency and check only
**
```

```
** ntot : Returns the total number of bytes to be filtered.
** Getting the exact correct number is not essential
** in this case, because the XDR filters won't rely
** on it, they use their internal pointers. This is
** used merely as a user-driven mechanism to make sure
** we did not forget a particular item in the STDHEP
** COMMON, and/or to compute the efficiency of the
** filtering.
**
** version : Returns the version for this block, a 4-character
** string.
 */

{

    unsigned int nn, nn2, nn4, nn5;
    int *idat;
    float *dat;

    if (xdrs->x_op == XDR_ENCODE) {
       if (*blockid != MCFIO_STDHEP) {
          fprintf (stderr, "mcf_Stdhep_xdr: Inconsistent Blockid %d \n ",
           (*blockid));
          return FALSE;
       }
/*
** Obviously this routine is good for Input and Output Stream.
** The XDR structure gives us which indirection is used through
**      the member xdrs->op
*/
       nn = (unsigned int) hepevt_.nhep;
       nn2 = 2 *(unsigned int) hepevt_.nhep;
       nn4 = 4 *(unsigned int) hepevt_.nhep;
       nn5 = 5 *(unsigned int) hepevt_.nhep;
       *ntot = 20 + 4 * (2 * nn + 2* nn2 + nn4 + nn5);
/*
**    This is an estimate of the total number of bytes used.
```

```
**
       strcpy(*version, "1.05");
       }

     if     ( (xdr_int(xdrs, blockid) &&
              xdr_int(xdrs, ntot) &&
              xdr_string(xdrs, version, 4) &&
              xdr_int(xdrs, &(hepevt_.nevhep)) &&
                xdr_int(xdrs, &(hepevt_.nhep))) == FALSE) return FALSE;
/*
**
** The header of the block has been filtered. We now in position
** to use this information to check the blockid and/or the
** version for Inpute stream
*/

     if ((xdrs->x_op == XDR_DECODE) && ( *blockid != MCFIO_STDHEP) ) {
          fprintf (stderr, "mcf_Stdhep_xdr: Inconsistent Blockid %d \n ",
           (*blockid));
          return FALSE;
     }
/*
** We now filter the data itself. We go array by array, setting a
** pointer, and filter.. We could write this in a more compact
** way, but it won't help to debug the stuff !
*/

     idat = hepevt_.isthep;
     if     ( xdr_array(xdrs, (char **)  &idat,
                 &nn, NMXHEP, sizeof(int), xdr_int) == FALSE) return FALSE;
     idat = hepevt_.idhep;
     if     ( xdr_array(xdrs, (char **)   &idat,
                 &nn, NMXHEP, sizeof(int), xdr_int) == FALSE) return FALSE;
     idat = (int *) hepevt_.jmohep;
     if     ( xdr_array(xdrs, (char **)   &idat,
                 &nn2, 2*NMXHEP, sizeof(int), xdr_int) == FALSE) return FALSE;
     idat = (int *) hepevt_.jdahep;
```

```
    if     ( xdr_array(xdrs,  (char **)   &idat,
                &nn2, 2*NMXHEP, sizeof(int), xdr_int) == FALSE) return FALSE;
    dat = (float *)  hepevt_.phep;
    if     ( xdr_array(xdrs,  (char **)   &dat,
                &nn5, 5*NMXHEP, sizeof(int), xdr_float) == FALSE) return FALSE;
    dat = (float *) hepevt_.vhep;
    if     ( xdr_array(xdrs, (char **)    &dat,
                &nn4, 4*NMXHEP, sizeof(int), xdr_float) == FALSE) return FALSE;
    return TRUE;
}
```

The real way to do this, of course, is to send an e-mail to the author of this infamous package, describing the USER COMMON, yell at him, and he will do it for you (That is, if you yell hard enough).