DASH: A Data Analysis Shell.
Manual For MACRO Users
V2.1

Edward Kearns
Department of Physics, Boston University

December 2, 1993

1 Introduction

High energy physics collaborations rely on an assortment of computing tools. Many of these are "standard" packages written and maintained at international laboratories. There is a package used at Fermilab, Brookhaven and some other places called *Analysis_Control* [1] that helps to organize analysis into modules and provides a flexible structure for controlling the analysis. The most sophisticated example of its use might be the CDF production analysis, which was the first stage of data processing and produced multiple output streams for a diverse set of physics topics. It relies on the UIPACK package for the user interface and the YBOS package for memory management.

However, at MACRO we use CERN products[2]: FFREAD or KUIP for user interface and ZEBRA (or the simpler DST common block) for memory management. There is no analog to Analysis_Control in the CERN distribution. So I have written a simplified version of Analysis Control using CERN tools; it is called DASH (Data Analysis Shell). It is a friendly interactive environment, with online help, useful HBOOK utilities, KUIP-macros and command line editing. A typical use would be to run interactively on a few events, perhaps using a event dump module to debug your analysis, and then when you know what you are doing, to use the same executable in a batch job.

By the way: DASH is completely independent of MACRO analysis- it does not know anything about DST's, ERPs, tracks et cetera. However, DASH can control *modules* which may read a DST event, find a track, make a histogram. DASH is a user interface shell which surrounds the subroutines that do the real work. As an analogy to DREAM, each module is like a DUSER subroutine. The difference is that there can be several modules and you can run them in any order. It is not intended to replace DREAM (the physics analysis routines); it could theoretically replace the user interface (which is based on the outmoded CERN package FFREAD). This would require rearranging the DREAM analysis into modules; a fair bit of work, but potentially worthwhile.

DASH has already been put to extensive use during its testing phase [3]. It has been used for Muon Analysis DST data as well as MACRO ZEBRA data. There seems to be a growing interest in expanding its use. This manual will explain its features and show you how to use it.

1.1 A simple example session

DASH provides a command line interface for turning on and off physics analysis that is coded in a suitably modular fashion. It controls the input and output of event based data. A simple session might go as follows:

```
DASH> path trgsel track stdcuts mycuts myana   | specify the analysis
DASH> talk trgsel choose muon           | select the muon trigger
DASH> filter on trgsel                  | require events pass muon-trigger
DASH> filter on stdcuts                 | require standard cuts
DASH> filter on mycuts                  | require events pass mycuts
DASH> filter failed mycuts              | only events that fail finalcuts
DASH> book track                        | keep tracking histograms
DASH> book myana                        | make histograms in myana
DASH> output on scratch:bad_events.dst  | write out events
DASH> input run005759.dst               | specify input file
DASH> analyze
DASH> input run005760.dst
DASH> analyze
...
DASH> hist/write scratch:bad_events.hbook    | save histograms to disk
DASH> exit
```

In the above example, `trgsel`, `track` and `stdcuts` are standard modules that the whole collaboration uses. The modules `mycuts` and `myana` are specific to this analysis. In this example the person is making histograms for events that fail his final cuts and writing the bad events to disk.

1.2 Philosophy

The philosophy behind this structure is to compartmentalize distinct analysis tasks. DASH provides a control structure for turning things on and off. Typically, each module runs independently and exchanges only event information, using `COMMON` or ZEBRA storage. One module should not need to know the value of a variable in another.

DASH is intended to work well interactively as well as in batch. Because of this, it is considered polite to keep your module quiet: do not print junk all over the screen as you do analysis. If you want diagnostic printout, see the print-level capabilities listed later in this document.

DASH becomes most useful when people can share modules. Although the talk entrypoint is a convenient way to set cuts, please make sure that your module does sensible analysis by default. It would also be nice if you provided a response to the command: `TALK MYMODULE HELP`.

As always, it is impossible to keep documentation up to date with analysis code. The best and most accurate documentation is the code itself. So if you intend to share your modules, please use long descriptive variable names and lots of comments. Include files with descriptive parameter names are also a good way to self-document data structures.

2 Terminology

MODULE: A module is a set of *entrypoints* gathered under a single name. For example, `trigsel` could be the name of a module that checks trigger bits.

PATH: This defines the order in which analysis modules are executed. The analysis modules can make histograms or ntuples. They may add reconstructed banks to the data structure. They may also set a flag indicating whether the event is "good" or "bad"; an optional *filter* will allow or prevent the event from going on to the next module.

READ-MODULE: This is a special module executed before the first module in the analysis path. It is expected to read in each event and put the event data somewhere that the analysis modules can access. This happens mostly outside the intervention of DASH. However DASH does expect the read-module to report the run and event number.

WRITE MODULE: This is a special module executed after the last module in the analysis path. It is expected to write out each event that it acts on. Unlike reading, writing out events is optional and may be controlled with the `OUTPUT` command.

ENTRYPOINT: An entrypoint is one of several instances when your module gets called. The most important is the **event entrypoint**, which gets called once per event. The list of entrypoints and when they get called:

INIT	When the executable program begins running
BEGINRUN	When a new run begins
EVENT	For each event
ENDRUN	When a run ends
BOOK	When you give the `BOOK` command
TALK	When you give the talk command
FINI	When the executable program stops running

A module is not required to use all of the entrypoints, or any particular entrypoint.

FILTER: This is a capability that any module may have. Inside a module, the user may call `DASH_PASS_EVT` or `DASH_FAIL_EVT`. In the first case, the analysis will continue on to the next module in the path (which might be the write module). In the second case, no further modules will be executed for this event; instead, the program will go on to the next event and begin the analysis path at the beginning.

TALK: This entrypoint is called when you give the `TALK` command. It is intended to give you interactive access to a module, for example to set flags or parameters.

DECLARE: DASH works by storing the addresses of the entrypoints of a module; this is called *declaration*. This involves a special subroutine that you write and link with your program.

3 How to write a DASH module

First, choose a name for your module. Second, write FORTRAN "subroutines" for each entrypoint you wish to use; the subroutines should have no arguments. The subroutines will access event data either through common blocks or ZEBRA banks.

I put "subroutines" in quotes, because I prefer to use the FORTRAN `ENTRY` statement. Let me tell you how it works: you write a single subroutine (with one `END` statement), but inside it are several `ENTRY` statements, each with a `RETURN` statement. You call (actually, you tell DASH to call) the `ENTRY` statements. The convenience is that the value of variables used in the subroutine are available to all of the entrypoints without a common block.

Here is an example, using all of the DASH entrypoints:

```
SUBROUTINE EXAMPLE                    ! Never actually called
IMPLICIT NONE                         ! You should always use this
INCLUDE 'MUA93_COMMON.INC'            ! The event data lives here
INTEGER LEN
CHARACTER*8 VERB
REAL CUT1, CUT2

ENTRY EXAMPLE_INIT                    ! Called when the program starts
CALL LOAD_MY_CONSTANTS
RETURN

ENTRY EXAMPLE_BEGINRUN                ! Called for a new run number
NPASSED = 0
RETURN

ENTRY EXAMPLE_EVENT                   ! Called for each event
CALL HFILL(1000,XYZ,0.,1.)
IF (XYZ .GT. CUT1) THEN
   NPASSED = NPASSED + 1
END IF
```

```
      RETURN

      ENTRY EXAMPLE_ENDRUN                 ! Called at end of run
      WRITE(*,*) ' N passed = ',NPASSED
      RETURN

      ENTRY EXAMPLE_BOOK                   ! c.l.i. command BOOK EXAMPLE
      CALL HBOOK1(1000,'The XYZ variable',
     &            100,0.,10.,0.)
      RETURN

      ENTRY EXAMPLE_TALK                   ! DASH> TALK EXAMPLE CUT1 1.5
      CALL KUGETC(VERB,LEN)
      IF (VERB(1:4) .EQ. 'CUT1') THEN
         CALL KUGETR(CUT1)
      ELSE IF (VERB(1:4) .EQ. 'HELP') THEN
         WRITE(*,*) ' Valid commands are: '
         WRITE(*,*) ' CUT1 '
      END IF
      RETURN

      ENTRY EXAMPLE_FINISH                 ! Called when program ends
      WRITE(*,*) ' I love DASH! '
      RETURN

      END
```

So, in a FORTRAN program, when the line `CALL EXAMPLE_BEGINRUN` occurs, the subroutine `EXAMPLE` begins with the line `NPASSED = 0`. You could replace every `ENTRY` with `SUBROUTINE`, but then you would have to declare the local variables separately and pass them around somehow. If you are more comfortable with explicit subroutine statements, go ahead and do it this way; it will work just the same.

3.1 What to put in each entrypoint

INIT The init entrypoint is called once, just after the program begins executing. This entrypoint is intended as a grand-initialization for your module. Because this entrypoint is called for every linked module, you should consider carefully if you need to do any initialization here at all. This is because the user may choose not to include your module in the analysis path; so if you try to open some file that is not there, then you will cause an error that may obstruct the user. I suggest you test for such initialization elsewhere, such as the event entrypoint, with `IF (.NOT. INIT) THEN....`

BEGINRUN The beginrun entrypoint is called whenever a beginrun record is read. If there are no beginrun records in the data, DASH will call beginrun whenever the run number changes.

EVENT The event entrypoint is where most of your analysis will probably occur. It is called whenever an event record is read.

ENDRUN The endrun entrypoint is called whenever an endrun record is read. In the case where there are no endrun records, there are two subtleties to discuss. DASH will call endrun whenever the run number changes, but before the beginrun entrypoint, so you can accumulate statistics or something similar. However, the current event in memory will be the first event of the new run. Therefore, if you want to record the time of the last event, you should do it in the event entrypoint and use the stored value in the endrun entrypoint. There is also the issue of what to do with the last event of the last file (no new run number will happen to trigger endrun). See the command `DASH/SET/EOF_ENDRUN` for details.

BOOK The book entrypoint is called when you issue the DASH command `BOOK`. See the command reference for more details and some useful features to help you organize your histograms. See also the DASH `HIST` commands for helpful utilities.

TALK The talk entrypoint is used to communicate with your module from the command line interface. It is called when you issue the DASH command `TALK <module> <command>`. You should write the code in the talk module so that it interprets `<command>` using KUIP: see the example modules to see how.

FINISH The finish entrypoint is called for all linked modules when the program ends. The comments under INIT apply here too.

OPEN This entrypoint is only present in read-modules and write-modules. It is called when the INPUT or OUTPUT command is given. See a well commented read/write module for more detail.

CLOSE This entrypoint is only present for read-modules and write-modules. It is called when appropriate, for example when the read-module hits an end-of-file.

4 How to tell DASH about your module

There is a subroutine that you have to provide. The subroutine name is `DASH_DECL`, but it is best if you name the file `MYPROGRAM.FOR`, where `MYPROGRAM` is what you will call your executable. `MYPROGRAM.FOR` will have a very short main `PROGRAM` section which calls `DASH_CLI`. The rest of the file is a subroutine named `DASH_DECL` that DASH will call. A complete example (including comments) is included in appendix A.

The DASH_DECL subroutine is used to list the entrypoints of all of the modules you intend to link with your program. Lets say you have copied a working example from someone and wish to add your own analysis module. Here are the modifications you make to the DASH_DECL subroutine. First, make subroutine calls which *declare* to DASH the name of your module and which entrypoints you have written:

```
c
```

```
c      Declarations for module MYANA:
c      ==============================
       CALL DASH_DEC_MOD('MYANA','INI',MYANA_INIT)
       CALL DASH_DEC_MOD('MYANA','BEG',MYANA_BEGINRUN)
       CALL DASH_DEC_MOD('MYANA','EVT',MYANA_EVENT)
       CALL DASH_DEC_MOD('MYANA','END',MYANA_ENDRUN)
       CALL DASH_DEC_MOD('MYANA','TAL',MYANA_TALK)
       CALL DASH_DEC_MOD('MYANA','BOO',MYANA_BOOK)
       CALL DASH_DEC_MOD('MYANA','FIN',MYANA_FINISH)
```

You also must declare (to FORTRAN) the subroutine names as `EXTERNAL`:

```
       EXTERNAL MYANA_INIT
       EXTERNAL MYANA_BEGINRUN
       EXTERNAL MYANA_EVENT
       EXTERNAL MYANA_ENDRUN
       EXTERNAL MYANA_TALK
       EXTERNAL MYANA_BOOK
       EXTERNAL MYANA_FINISH
```

The other important thing you do in `DASH_DECL` is to declare the size of CERN arrays and initialize the packages. So you will typically see lines like as follows, which you can change as needed:

```
       INTEGER PAW
       COMMON/PAWC/PAW(800000)
       ...
       CALL HLIMIT(760000)
       CALL KUINIT(40000)
```

You may see other modules declared in the `DASH_DECL` you start with. Leave them in if you want them in your program, delete the lines if you do not. You may see other `DASH_DEC_xxx` routines; see the section on utility routines below for an explanation of more advanced use.

4.1 How to link your program using VMS

You should have the logical name `DASH$LIBRARY` defined. This is defined with DCL by `@disk:[xxx.DASH]DASH_SETUP`. You will need to know where DASH is installed on your computer to specify "disk" and "xxx". `DASH_SETUP` also creates a symbol called `DASHLINK` that you use to link your program.

Specify your modules for linking in a *options file*. This is a standard VMS feature, but perhaps you have not used it before. An options file is simply a list of files, just as they might follow a link command:

```
$ LINK MYPROGRAM, MYSUBROUTINE, -
       USEFUL_SUB, UTILITY$LIBRARY:UTILITY/LIB
```

This is the same as:

```
$ LINK MYPROGRAM,MY_OPTFILE/OPT
```

if the file `MY_OPTFILE.OPT` consists of:

```
MYSUBROUTINE, -
USEFUL_SUB, UTILITY$LIBRARY:UTILITY/LIB
```

Now that you understand options files, make one that lists your declaration subroutine and your module and any other modules you intend to use. You actually link with the command:

```
$ DASHLINK <program-name> <options-file>
```

By convention, the name of the options-file should be the same as your program-name (see also how to name `DASH_DECL`- these are merely suggestions for bookkeeping). `DASHLINK` will link to the CERN libraries PACKLIB and KERNLIB for you. A perfectly acceptable alternative is for you to copy `DASHLINK.COM` to your area and use it as a template. You can put all the link directives into your own command (`.COM`) file.

4.2 How to link your program using Unix

Every UNIX machine has a facility called `make`. This can be very helpful and powerful, although there are certainly some subtleties. It will contain the directives for linking to DASH, CERN and other libraries. There are places in the Makefile where you specify the names of your FORTRAN files. The `make` facility will take care of the rest. An example `Makefile` is included in appendix B.

5 More DASH Features

Most of the features are described in the DASH command reference. But there are a few things I will describe in more detail here.

5.1 DASHLOGON and BATCH files

A DASH executable is started very similar to the way PAW is started. If you have a file in your home area called `.dashlogon.kumac` (Unix) or `DASHLOGON.KUMAC` (VMS), it will be executed. If you wish to redirect it to a different file, you may set the environment variable `DASHLOGON` (Unix) or logical name `DASH$LOGON` (VMS). It may also be specified on the command line with the `-l` option (Unix) or with `/LOGON=logonf.kumac` (VMS).

There are two commands that you may want to place in your logon file to override the default behavior of KUIP. The command `HELP -N` turns off the new default HELP behavior which pops you into an editor whenever you want help. Most help commands are relatively short. Also `SET/BREAK TB` is preferred when debugging since it will cause errors to provide a traceback of routines and line numbers. KUIP's default behavior is to trap errors and return to the command line interface. See `HELP HELP` and `HELP SET/BREAK` in a current version of PAW for the latest details.

You may also specify a file of DASH commands with the `-b` (Unix) or BATCH=batchf.kumac command-line options. This is the same as typing the command `EXEC batchf` as the only command of a job.

See `KUARGS` in the KUIP manual for more information.

5.2 How to filter events with your module

You may wish to filter events; that is, to interrupt the analysis path depending on whether an event passes your cuts. To do this, call `DASH_PASS_EVT` or `DASH_FAIL_EVT` somewhere in your event entrypoint. After your module returns control to DASH, the value of a flag will be tested. If you "failed" the event, execution of the analysis path will cease and a new event will be read in. If you "passed" the event, the next module in the analysis path will be called.

The above action is controlled by the `FILTER` command. If you give the command `FILTER ON MYANA`, then the filtering will happen. You can disable it with `FILTER OFF MYANA`. Filtering is assumed to be off for module `MYANA` by default, unless `DASH_DEC_FLT('MYANA')` is called in `DASH_DECL`.

You may invert the behavior of your filtering from the command line by specifying `FILTER FAILED MYANA`. This turns filtering on, but events that "pass" the filter criteria are not processed further, whereas events that "fail" the filter continue on for further analysis and/or writing to disk.

5.3 Print and debug levels

Although most switches (such as the value of cuts) should be controlled through the talk entrypoint, DASH does have a few special lines of communication set up. One can set two special "levels" from the command line: `DASH> SET/PRINT_LEVEL n` and `DASH> SET/DEBUG_LEVEL n`.

The print-level is intended to control diagnostic printing that you may include in your module. The print-level is initially zero; by convention you should print nothing but serious error messages. Setting the print level higher may then increase the amount of informational or diagnostic printing. Inside your analysis module you check the current print-level like this:

```
INTEGER DASH_PRI_LEV
EXTERNAL DASH_PRI_LEV
...
IF (DASH_PRI_LEV() .GE. 1) WRITE(*,*) ' Something useful '
```

The debug-level works the same way, but you would control FORTRAN code that actually does something besides print information.

5.4 Command Line Interface

Since DASH uses KUIP, you automatically obtain the many benefits provided by KUIP. This is the same interface package used by PAW, so even if you have not programmed with it, you may already be familiar with it. It gives you a command history (up-arrow) and command line editing. You can have MACRO files (`.kumac`). It has an integrated help facility.

A good command line interface is most helpful during the development phase of your analysis program. You are able to use the DASH commands to start up your analysis and analyze a single event and then interactively work with it. Using the `CALL` command, you can repeatedly invoke an event entrypoint, which can be helpful when debugging some code. Using `PATH`, you can include in the analysis path event-dump or other diagnostic tools. Using `RUNLIST`s, you can go right to an interesting event in a data file.

The transition to batch jobs which analyze large amounts of data is straightforward. The same commands you use interactively go in the batch file. Plus, the commands can be broken up into KUIP MACRO's- for example one might have `MYINIT.KUMAC` which sets up things that many jobs would want.

5.5 Histograms

Since HBOOK is an unchallenged standard, it was reasonable to build a certain amount of support directly into the DASH commands. Most importantly, you are provided a command to write the histograms out to a filename you specify on the command line. This can happen at the end of your job or between input files if that is useful to your analysis (the `HIST/RESET` command is probably helpful here).

You might decide to remake a histogram with different limits than those you put in your code. You can do this without recompiling and relinking using `HIST/DELETE and HIST/HBOOK1`, where the latter command respecifies that histogram.

Since one of the philosophical strengths of DASH is sharing modular code, DASH helps you manage histograms across modules. The command `BOOK MYANA` calls the book entrypoint for module `MYANA` and the histograms will be saved in the top level directory of the HBOOK output file (`//PAWC`). You may optionally specify a HBOOK subdirectory for a module: `BOOK MYANA BANANA`, which will put those histograms in `//PAWC/BANANA`. A convenient shortcut is to specify the optional second argument as "`*`". Then, `BOOK MYANA *` will use the modulename as the directory name and put the histograms in `//PAWC/MYANA`.

The way DASH manages histogram subdirectories is to call `HCDIR` before and after every entry into your module. Using histogram subdirectories helps avoid ID number conflicts as well as provides you with a simple heirarchy that makes browsing your histogram output file with PAW a little easier.

5.6 Ntuples, how to implement

Memory-based ntuples are implemented exactly like histograms. In the book entrypoint you call `HBOOKN` and in your code you call `HFN`. This is the easiest way to deal with relatively small ntuples. For example, this works quite well for accumulating endrun statistics, where each ntuple row contains the run number, run length, efficiencies etc. The ntuple will be saved (with your histograms) to the filename of your choice when you give the `HIST/WRITE` command. The `HIST/WRITE` command simply gets the filename of the DASH command line and calls `HRPUT(0,HBKFIL,'T')`.

But if you wish to make an ntuple summarizing information over thousands of events on an event by event basis, the standard technique is to make a *disk-based ntuple*. The problem for DASH is that information about the top directory name of the RZ file (`CHRZPA`) must be coordinated between `HROPEN` and `HBOOKN`. Since `HBOOKN` must be coded by the user, the user should also take care of calling `HROPEN`. Once the ntuple is booked and associated with a directory in an RZ-file, the user never has to worry about which hbook subdirectory he is in when filling the ntuple. Therefore, these ntuples will not interfere with the automatic subdirectory features discussed in section 5.5.

In addition, the user must call `HREND` at the end of his job to cause the last flush from memory to RZ file. In this case, you must `HCDIR` to the top directory of the RZ file first. This would best be done in the FINI entrypoint, although it may be useful to have a way to close these files available from the TALK entrypoint as well.

The user may specify the filename for HROPEN as a fixed name, or calculate it from the run number or date. The most flexible choice would be to control the filename from the `TALK` command (with a good default to handle the case where the `TALK` command has not been given). See appendix C for sample code.

In summary: since disk-based ntuples require management of the top level directory of the RZ file as well as the file name, I have left this completely up to the module-author. If new ways to handle this become available in HBOOK, they might be incorporated into future DASH features.

5.7 Analysis Control and Bookkeeping

Much of the control and bookkeeping advantages are built-in by KUIP, discussed above. But there are also DASH commands that allow you to set up and monitor your analysis.

Using RUNLIST/INCLUDE or RUNLIST/EXCLUDE, you can set up a job to skip or jump to certain runs, events or blocks of events. You can also use the SKIP command to skip a certain number of events, without regard to the exact run and event numbers. One possible use is to make up a KUIP macro file, GOODRUNS.KUMAC, that picks "good runs" out of a directory of all runs, such as this exerpt shows:

```
INPUT WSGS03$DKA100:[MACRODATA.DST]RUN005530.DSTE ; ANA
INPUT WSGS03$DKA100:[MACRODATA.DST]RUN005531.DSTE ; ANA
INPUT WSGS03$DKA100:[MACRODATA.DST]RUN005532.DSTE ; ANA
```

Note that we have taken advantage of KUIP's feature allowing multiple commands to be put on one line if separated by a semi-colon.

When running over a large number of events, it is reassuring to have some sort of notification at regular intervals. The COUNT commands give you many ways to accomplish this. See the examples in the command reference for suggestions. In addition, the STATUS command will report how many events were analyzed at each module in the analysis path (as well as giving other descriptive information about the modules linked and in the path).

5.8 Using HIGZ Graphics

If you would like to make a module perform a graphic display using the CERN package HIGZ, you need to make some initialization calls and pass around a variable that records the workstation type. Since the initialization should happen only once, but you may have more than one display module, you should make the initialization calls in DASH_DECL. The workstation type can be reported to DASH via a call to DASH_DEPOSIT('KWTYPE',KWTYPE). Then, any analysis module that needs to know the value can call DASH_INQUIRE('KWTYPE',KWTYPE) and the value will be passed back.

In DASH_DECL, you should make the following calls:

```
INTEGER KWTYPE
...
CALL IGINIT(0) ! Initialize HIGZ
CALL IGWKTY(KWTYPE) ! Get workstation
CALL IGSSE(6,KWTYPE) ! Errors go to Unit 6
CALL DASH_DEPOSIT('KWTYPE',KWTYPE)
```

If you wish to write a robust graphics module, you will want to use the value of `KWTYPE` to keep track of whether you are writing to an open graphics display or a metafile. See Colin Okada's `MUAPICT` module for example code; this is included in the Muon Analysis DST library (MADLIB) (see appendix D).

At the end of the program, you should shut down the graphics gracefully with `IGEND`. The best place to do this is in the main program after the call to `DASH_CLI`. These calls are in the example `DASH_DECL` in appendix A, but are commented out.

6 Troubleshooting

In this section, I will review the points that seem to be the most frequent causes of trouble when beginning to use DASH.

- If you have an access or segmentation violation associated with calling an entrypoint, this is probably DASH attempting to call an entrypoint that is nonexistant. These problems typically occur when you are adding a new module or adding an entrypoint to an existing module. There are three likely causes:
  - In `DASH_DECL`, the entrypoint is declared to DASH with `DASH_DEC_MOD`, but is not declared `EXTERNAL`.
  - The entrypoint is declared in `DASH_DECL` but not coded in the module, or coded with a different name.
  - The entrypoint is declared in `DASH_DECL` and coded in the module, but you have not linked to the module object file. This should cause an unsatisfied external link error. Fix your options file (VMS) or Makefile (Unix).
- You have added a new module, there are no link errors, but it is not present when you look for it with the `STATUS` command. You have forgotten to add the module in `DASH_DECL`: you need to add calls to `DASH_DEC_MOD` for each entrypoint and declare them `EXTERNAL`.
- Most DASH commands can be given in any order. But watch out for the "`*`" shortcut. For `BOOK`, `FILTER` and `CALL`, the "`*`" represents "all modules in the analysis path". Therefore, this shortcut should be used `after` you set up the analysis path.
- The analysis path should not include the read-module and write-module. The `PATH` command is used to specify the analysis modules in between. The read-module always comes first for each event, and the write-module always comes last. Usually there is only one read-module and write-module for a given DASH executable and you never have to specify anything; if you have more than one, see `SET/READ` and `SET/WRITE` for more information.
- Some problems may be related to the changing state of the CERN packages, KUIP in particular. For example, the way KUIP handles optional arguments in release 93C introduced a bug that means you have to specify all the arguments for `RUNLIST/INCLUDE`

and RUNLIST/EXCLUDE. This is not a problem in earlier versions and will be fixed in release 93D. Another problem that seems to occur after a new CERN release is that the code in DASH_DEF needs to be rebuilt. This is the CLI definition routine and is built from DASH_DEF.CDF using the program KUIPC.

7 Installation

If you are working on VAXGS::, DASH is already installed. You will find the library, documentation and command files in DISK$MACRO2:[MACROUSA.DASH]. The subdirectories such as [.V21] are for managing release versions. The current "production" version of the library will be in the top-level directory.

If you wish to install DASH on a remote VAX, you may simply copy the library, DASHLIB.OLB, from the above area, along with the documentation and command files.

If you wish to install DASH with source code, which is a necessity for Unix installation, you can take the file dashv21.tar from the [MACROUSA.DASH.V21] directory. Use ftp in binary mode. More information is included in AAA_INSTALLING_DASH.TXT in the same directory.

A starting library of modules and an example program can be found in [MACROUSA.MAD]. See AAA_*.TXT and appendix D for more information.

Good luck and have fun!

8 DASH Command Reference

Data Analysis Shell V2.0

8.1 INPUT inpfil [opts]

```
INPFIL     C 'File Name:'
OPTS       C 'Optional Parameters:' D=' '
```

Specify the input file for events.
The open entrypoint of the read-module will intercept the file name. It may optionally intercept further parameters. See the documentation or code for the current read-module for more information.

Example: `DASH> input run005759.dst`

8.2 OUTPUT flag [outfil opts]

```
FLAG       C 'ON/OFF:' R='ON/OFF'
OUTFIL     C 'File Name:'
OPTS       C 'Optional Parameters:' D=' '
```

Specify the output file for events.
The open entrypoint of the write-module will intercept the file name. It may optionally intercept further parameters. See the documentation or code for the current write-module for more information.

Example: `DASH> output on good_events.dst`
Example: `DASH> output off`

8.3 PATH [mod]

```
MOD        C 'Module Name(s):'
```

Specify the analysis path.
The argument should be a list of valid module names, separated by spaces. Use the STATUS command to see what modules are linked with the current executable. You should not explicitly include the read-module or write-module; they are controlled by INPUT and OUTPUT. The default analysis path is determined in the declaration subroutine; use the STATUS command to see the current analysis path.

Example: `DASH> path trgsel calib track mycuts myana1`

8.4 BOOK modnam [dir]

```
MODNAM     C 'Module Name:' D='*' R='Any declared module'
DIR        C 'Hist Dir:' D=' '
```

Call booking entrypoint for specified module.
MODNAM=* will book histograms for all modules in the analysis path. Optional DIR will put histograms in a subdirectory of //PAWC; DIR=* will use module name for the directory name.

Example: `DASH> book myana1`
Example: `DASH> book * *`

8.5 ANALYZE [nana]

```
NANA       I 'Analyze how many events?:' D=0
```

Analyze this many events.
NANA omitted or =0 will analyze entire file.

Example: `DASH> ana 100`

8.6 SKIP nskip

```
NSKIP      I 'Skip how many events?:' D=1
```

Skip this many events without analyzing.

Example: `DASH> skip 10`

8.7 FILTER flag modnam

```
FLAG       C 'ON,OFF:' D='ON' R='ON,OFF,FAILED'
MODNAM     C 'Module Name:' D='*'
```

Turn on/off the filter for a module.
MODNAM=* will set the filter on/off for all modules in the analysis path. The default for a module is determined by the module when it is declared; use the STATUS command to see the current state. FLAG=FAILED will select events that fail a filtering module rather than those that pass.

Example: select triggers, turn off some cuts, write bad events

```
DASH> path trgsel track mycuts myana finalcuts
DASH> talk trgsel select mutrigger
DASH> book myana
DASH> filter on trgsel
DASH> filter off mycuts
DASH> filter failed finalcuts
DASH> output on bad_events.dat
```

8.8 TALK modnam [cmdline]

```
MODNAM     C 'Module Name:'
CMDLIN     C 'Command-line:' D=' '
```

Call TALK entrypoint for module.
The TALK entrypoint can then process the CMDLINE (using KUGETC, KUGETI and KUGETR) and set flags or parameters specific to the module.

Example: `DASH> talk mycuts e_min 12.  e_max 1000.`
Example: `DASH> talk track 3dtrack on`

8.9 CALL entryp [modnam]

```
ENTRYP     C 'Entrypoint:'
MODNAM     C 'Module Name:' D='*'
```

Call any entrypoint for module.
This command forces the specified entrypoint to be called now. You may specify the module name, or use MODNAM=* to call the entrypoint for all modules in the analysis path.

Example: `DASH> call endrun myana`

8.10 STATUS

Informative printout about the analysis program.
The names of the modules are listed vertically, in the order in which they will be executed. Modules marked OFF are linked in but are not in the current analysis path. The entrypoints for each module are indicated as follows:

```
X = exists             B = booked
F = filtering on      -F = filter on failed events
READ = read-module     WRIT = write-module
```

8.11 RUNLIST

Include or exclude runs/events from analysis.

8.11.1 RUNLIST/INCLUDE run [event run2 event2]

```
RUN        I 'Run number:'
EVENT      I 'Event number:'
RUN2       I 'Run number high range:'
EVENT2     I 'Event number high range:'
```

Include runs, events or ranges thereof for analysis.
If EVENT is not specified, the whole run will be included. RUN2 may be optionally specified to include a range of runs; again, if EVENT2 is not specified, every event will be included. Using the INCLUDE command will automatically exclude all other runs and events; therefore, one typically does not mix INCLUDE and EXCLUDE commands.

Example: list of special events

```
DASH> runlist/incl 5659 392
DASH> runlist/incl 5779 472
 ...
```

8.11.2 RUNLIST/EXCLUDE run [event run2 event2]

```
RUN        I 'Run number:'
EVENT      I 'Event number:'
RUN2       I 'Run number high range:'
EVENT2     I 'Event number high range:'
```

Exclude runs, events or ranges thereof from analysis.
If EVENT is not specified, the whole run will be excluded. RUN2 may be optionally specified to exclude a range of runs; again, if EVENT2 is not specified, every event will be excluded. Note: KUIP now allows you to skip arguments. You may skip the `EVENT` argument to specify a list of runs (see the example below).

Example: list of bad runs

```
DASH> runlist/excl 6666
DASH> runlist/excl 6789 run2=6792
...
```

Example: exclude events in run 5488 where detector went crazy (2012:3777)

```
DASH> runlist/excl 5488 2012 5488 3777
```

8.11.3 RUNLIST/CLEAR

Initialize the runlist (analyze all events).
This command resets DASH so that it analyzes all events.

8.11.4 RUNLIST/PRINT

Print the runlist to the screen.

8.12 COUNT

Count number of events or runs or files processed.

8.12.1 COUNT/PRINT

Print out counters now.

8.12.2 COUNT/CLEAR

Reset counters to zero.

8.12.3 COUNT/EVENTS flag [type mod]

```
FLAG       C 'ON/OFF:' R='ON,OFF'
TYPE       C 'Type=' R='READ,ANALYZE,WRITE'
MOD        I 'Modulo=' D=1
```

Print event count every MOD events.
Use TYPE to specify counting events (R)ead, (A)nalyzed or (W)ritten, default is analyzed.

Example: `DASH> count/events on mod=1000`

8.12.4 COUNT/RUNS flag

```
FLAG       C 'ON/OFF:' R='ON,OFF'
```

Print event count at end of each run.

Example: `DASH> count/run on`

8.12.5 COUNT/FILES flag

```
FLAG        C 'ON/OFF:' R='ON,OFF'
```

Print event count at end of each file.

Example: `DASH> count/file on`

8.13 HIST

HBOOK Histogram control commands.

8.13.1 HIST/WRITE hbkfil

```
HBKFIL      C 'File Name:' D='HISTOGRAMS.HBOOK'
```

Output histograms to a file.
This uses HRPUT, see the HBOOK manual for more details.

Example: `DASH> h/write myana.hbook`

8.13.2 HIST/PRINT id [dir]

```
ID          I 'Id:' D=0
DIR         C 'Hist Dir:' D=' '
```

Print a histogram (ID=0 prints all histograms).
See HPRINT in HBOOK the manual. Optional DIR will print the histograms in the specified directory.

Example: `DASH> h/print 100`
Example: `DASH> h/print 0 MYANA`

8.13.3 HIST/RESET id [title dir]

```
ID          I 'Id:' D=0
TITLE       C 'Title:' D=' '
DIR         C 'Hist Dir:' D=' '
```

Reset histograms (ID=0 resets all histograms).
You may optionally change the title also. See HRESET in the HBOOK manual. Optional DIR will RESET the histograms in the specified directory.

Example: `DASH> h/reset 0`
Example: `DASH> h/reset 2030 MYANA`

8.13.4 HIST/DELETE id [dir]

```
ID         I 'Id:' D=0
DIR        C 'Hist Dir:' D=' '
```

Delete histograms (ID=0 deletes all histograms).
See HDELET in HBOOK manual. Optional DIR will delete histograms in the specified directory.

Example: `DASH> h/delete 0`
Example: `DASH> h/print 2031 MYANA`

8.13.5 HIST/HBOOK1 id chtitl nx xmi xma vmx [dir]

```
ID         I 'id:'
CHTITL     C 'title:'
NX         I 'number of bins:'
XMI        R 'x lower edge of first bin:'
XMA        R 'x upper edge of last bin:'
VMX        R 'vmx:'
DIR        C 'Hist Dir:' D=' '
```

Book histogram using HBOOK1.
This would be used to rebin a histogram after HIST/DELETE. See HBOOK1 in the HBOOK manual for more detail. Optional DIR will book the histogram in the specified directory.

Example: rebin a histogram without recompiling and relinking

```
DASH> book myana
DASH> h/delete 1001
DASH> h/book1 1001 "Transverse Energy" 100 0. 100. 0.
```

8.13.6 HIST/HBOOK2 id chtitl nx xmi xma ny ymi yma vmx [dir]

```
ID         I 'id:'
```

```
CHTITL     C 'title:'
NX         I 'number of x-bins:'
XMI        R 'x lower edge of first bin:'
XMA        R 'x upper edge of last bin:'
NY         I 'number of y-bins:'
YMI        R 'y lower edge of first bin:'
YMA        R 'y upper edge of last bin:'
VMX        R 'vmx:'
DIR        C 'Hist Dir:' D=' '
```

Book histogram using HBOOK2.
This can be used to rebin a histogram after HIST/DELETE. See HBOOK2 in the HBOOK manual for more detail. Optional DIR will book the histogram in the specified directory.

Example: rebin a histogram without recompiling and relinking

```
DASH> book myana *
DASH> h/delete 2010 MYANA
DASH> h/book2 2010 "delta x vs delta y" 100 -5. 5. 100 -5. 5. 0. MYANA
```

8.14 SET

Set various control parameters.

8.14.1 SET/PRINT_LEVEL prtlev modnam

```
PRTLEV     I 'Print Level:' D=0
MODNAM     C 'Module Name:'
```

Specify the print-level for a module.
The module may call the integer functon `DASH_PRI_LEV()` and depending on the returned value print a lot (or a little) extra information.

Example: `DASH> set/print 1 mycuts`

8.14.2 SET/DEBUG_LEVEL dbglev modnam

```
DBGLEV     I 'Debug Level:' D=0
MODNAM     C 'Module Name:'
```

Specify the debug-level for a module.
The module may call the integer function `DASH_DBG_LEV()` and depending on the returned value execute special code to assist in debugging.

Example: `DASH> set/debug 3 myana`

8.14.3 SET/EOF_ENDRUN onoff

```
ONOFF      C 'On/Off:' D='ON'
```

Set flag to call endrun at end-of-file.
By default, endrun entrypoints are called when the file ends. If you have runs which span files, you may not want this; issue the command in the example below. Note: in this case, if there are no separate endrun records, DASH will not know to call endrun after the last file you analyze. So you should finish your job with CALL ENDRUN * (see CALL command).

Example: `DASH> set/eof_end off`

8.14.4 SET/READ_MODULE modnam

```
MODNAM      C 'Module Name:'
```

Choose a read-module.
This command can be used to select a primary read-module. Typically, one is determined by default in `DASH_DECL`; however if one is using several data formats (for example in a translation program), one may wish to select a different one.

Example: `DASH> set/read alt_read`

8.14.5 SET/WRITE_MODULE modnam

```
MODNAM      C 'Module Name:'
```

Choose a write-module.
This command can be used to select a primary write-module. Typically, one is determined by default in `DASH_DECL`; however if one is using several data formats (for example in a translation program), one may wish to select a different one.

Example: `DASH> set/write alt_write`

9 DASH Library Routines Reference

Most DASH routines are for internal use only. The following routines may be called by the user.

· `DASH_DEC_MOD(MODNAM,ENTRYP,SUBROU)`
This declares a module named by the character string `MODNAM` to have an entrypoint specified by the character string `ENTRYP` that is accessed through the address of `SUBROU`, which should be an `EXTERNAL` routine (`ENTRY` or `SUBROUTINE` statement). `DASH_DEC_MOD` should be called in the user-provided `DASH_DECL` subroutine. The string `ENTRYP` should be one of the following values (only the first two characters are significant): `'INIT'`, `'BEGINRUN'`, `'EVENT'`, `'ENDRUN'`, `'BOOK'`, `'TALK'`, `'FINI'`, `'OPEN` or `'CLOSE'`. Example:

```
    EXTERNAL MYMODULE_EVENT
...
CALL DASH_DEC_MOD('MYMODULE','EVENT',MYMODULE_EVENT)
```

· `DASH_DEC_PAT(MODNAME)`
This places the module `MODNAME` in the default analysis path (the analysis path the executable will start-up with before any `PATH` command is given. It may be overridden by a `PATH` command given on the DASH command line. `DASH_DEC_PAT` should be called in the user-provided `DASH_DECL` subroutine.

· `DASH_DEC_FLT(MODNAME)`
This turns filtering on for module `MODNAME` by default. It may be overridden with `FILTER <ON/OFF>` commands given on the DASH command line. `DASH_DEC_FLT` should be called in the user-provided `DASH_DECL` subroutine.

· `DASH_DEC_REA(MODNAME)`
· `DASH_DEC_WRI(MODNAME)`
This declares the module `MODNAME` to be the default read/write module. It may be overridden with the commands `SET/READ` or `SET/WRITE`, although this will only be necessary in executables with more than one possible read-module or write-module (such as programs which translate between ZEBRA and binary DST files). These calls should be made in `DASH_DECL`.

· `DASH_PASS_EVT()`
· `DASH_FAIL_EVT()`
These subroutines are called in the event entrypoint of an analysis module. `DASH_PASS_EVT` tells DASH to continue analyzing the event with the next module. `DASH_FAIL_EVT` tells DASH to discontinue analyzing this event and to restart the analysis with the next event. These actions will be enabled if filtering is turned on (either through the `FILTER ON` command or by default with `DASH_DEC_FLT`. These actions will be ignored if filtering is off for this module.

· `INTEGER FUNCTION DASH_PRI_LEV()`
This function should be called from within analysis modules. `DASH_PRI_LEV` returns an integer equal to the print-level set on the command line with `SET/PRINT` command. If the print-level is greater than zero, the module may print out informative information.

· `INTEGER FUNCTION DASH_DBG_LEV()`
This function should be called from within analysis modules. `DASH_DBG_LEV` returns an integer equal to the debug-level set on the command line with `SET/DEBUG` command. If the debug-level is greater than zero, the module may execute alternative code that is helpful in debugging.

· `DASH_DEPOSIT(KEYWRD,VALUE)`
This function is usually only called by read-modules and write-modules. It is used to report to tell DASH the current run number (`KEYWRD = 'RUN'`), event number (`KEYWRD = 'EVENT'`). Other keywords are: `'TYPREC'`, the record type; and `READOK` and `WRITOK`, which reports if the i/o for reading or writing an event was successful.

· `DASH_INQUIRE(KEYWRD,VALUE)`
This may be used to ask DASH to report the current run number (`KEYWRD = 'RUN'`) or event number (`KEYWRD = 'EVENT'`).

10 References

References

[1] A Beginner's Guide to Analysis_Control and Build_Job, M. Shapiro, E. Sexton-Kennedy and D. Quarrie, CDF Note 384.

[2] The CERN manuals are available by anonymous ftp from `asis01.cern.ch` in the directory `cernlib/doc/ps.dir`.

[3] I would like to thank the following testers for putting DASH through the paces, uncovering bugs and generally giving back good ideas on how it should work: Rob Cormack, Colin Okada, Chris Walter, Bob Nolty, Gary Ludlam and John Hong. I think they may have even got some useful work out of it along the way.

[4] Managing Projects with **make**, A. Oram and S. Talbott, O'Reilly & Associates, Inc. (1991)

A EXAMPLE.FOR: An example DASH_DECL subroutine

```
      PROGRAM EXAMPLE
      CALL DASH_CLI
c     CALL IGEND ! Uncomment if using HIGZ
      END

      SUBROUTINE DASH_DECL
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c THIS SUBROUTINE SHOULD BE CALLED DASH_DECL, HOWEVER THE FILE NAME
c SHOULD BE DESCRIPTIVE OF THE EXECUTABLE, eg. MYPROGRAM_DECL.FOR
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c This is the declaration subroutine for a DASH executable. You, the
c programmer provides this subroutine and DASH will call it on startup.
c This contains subroutine calls that are used to make a list of all
c the modules one can use and which entry-points they have. Possible
c entrypoints: INIT BOOK EVENT BEGINRUN ENDRUN TALK FINI. Read/write
c modules will also have OPEN and CLOSE entrypoints. In the calls to
c DASH_DEC_MOD you need only specify the first few characters of
c these keywords.
c
c *** To add an analysis module, the user must modify this routine,
c *** adding 1) EXTERNAL declarations for each entrypoint and
c *** and 2) a call to DASH_DEC_MODULE for each entrypoint.
c
c In addition, one module be declared a read-module, from which events
c are read in. Similarly, one module should be declared a write module
c to write events out.
c
c *** You should also allocate arrays and initialize packages in this
c *** subroutine, such as PAW, HBOOK and KUIP.
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      IMPLICIT NONE
      INTEGER PAW
      COMMON/PAWC/PAW(800000)
      INTEGER KWTYPE
c-----------------------------------------------------------------------
      EXTERNAL MADREAD_EVENT
      EXTERNAL MADREAD_OPEN
      EXTERNAL MADREAD_CLOSE
      EXTERNAL MADREAD_TALK
c-----------------------------------------------------------------------
      EXTERNAL MADWRITE_EVENT
      EXTERNAL MADWRITE_OPEN
      EXTERNAL MADWRITE_CLOSE
      EXTERNAL MADWRITE_TALK
```

```
c-----------------------------------------------------------------------
      EXTERNAL MADDUMP_EVENT
      EXTERNAL MADDUMP_TALK
c-----------------------------------------------------------------------
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c     Initialize packages:
c     ====================
      CALL HLIMIT(760000)
      CALL KUINIT(40000)
c
c     Initialize graphics (if using HIGZ):
c     ====================================
C      CALL IGINIT(0) ! Initialize HIGZ
C      CALL IGWKTY(KWTYPE) ! Get workstation
C      CALL IGSSE(6,KWTYPE) ! Errors go to Unit 6
C      CALL DASH_DEPOSIT('KWTYPE',KWTYPE)
c
c     Declarations for module MADREAD:
c     ================================
      CALL DASH_DEC_MOD('MADREAD','EVNT',MADREAD_EVENT)
      CALL DASH_DEC_MOD('MADREAD','OPEN',MADREAD_OPEN)
      CALL DASH_DEC_MOD('MADREAD','CLOS',MADREAD_CLOSE)
      CALL DASH_DEC_MOD('MADREAD','TALK',MADREAD_TALK)
      CALL DASH_DEC_REA('MADREAD')
c
c     Declarations for module MADDUMP:
c     ================================
      CALL DASH_DEC_MOD('MADDUMP','EVNT',MADDUMP_EVENT)
      CALL DASH_DEC_MOD('MADDUMP','TALK',MADDUMP_TALK)
c
c     Declarations for module MADWRITE:
c     =================================
      CALL DASH_DEC_MOD('MADWRITE','EVNT',MADWRITE_EVENT)
      CALL DASH_DEC_MOD('MADWRITE','OPEN',MADWRITE_OPEN)
      CALL DASH_DEC_MOD('MADWRITE','CLOS',MADWRITE_CLOSE)
      CALL DASH_DEC_MOD('MADWRITE','TALK',MADWRITE_TALK)
      CALL DASH_DEC_WRI('MADWRITE')
c
      RETURN
      END
```

B Linking: Examples using Unix and VMS.

Under Unix, the make facility is the best method for compiling and linking. Shown below is a simple Makefile that builds the example executable included with the Muon Analysis DST library for DASH. See the Unix man page on make for more information; also [4] is a good general reference.

```
# Makefile for "example"- example program using MAD and DASH

MADDIR = /macro/users/kearns/mad
DASHDIR = /macro/users/kearns/dash
CERNDIR = /sgibin/cernlib/cern/pro/lib

FFLAGS = -g -O0 -static

LOADLIBES = -L$(MADDIR) -lmad \
-L$(DASHDIR) -ldash \
-L$(CERNDIR) -lpacklib -lkernlib

FOBJS = example.f

example: $(FOBJS)
$(FC) -g -o example $(FOBJS) $(LOADLIBES)
```

Under VMS, the linking can be handled by the DASHLINK command, which is created as a symbol if you execute DASH_SETUP.COM in the DASH directory. Alternatively, you may create your own com-file to handle linking. If you do use DASHLINK, you provide an options file that lists all the code (modules, private libraries) you wish to link in (as explained in section 4.1. The rather trivial options file for the example program consists of:

```
EXAMPLE,DISK$MACRO2:[MACROUSA.MAD]MADLIB/L
```

DASHLINK.COM takes two arguments. The first is the name of the executable to create and the second is the options file:

```
$ LOCDIR = F$PARSE(``''P1''',,,''DEVICE'')+F$PARSE(``''P1''',,,''DIRECTORY'')
$ LINK/DEB/EXE='P1' -
  'LOCDIR''P2'/OPT, -
  DASH$LIBRARY:DASHLIB/L,CERN_ROOT:[LIB]PACKLIB/L,KERNLIB/L
```

C Disk-based ntuples: example code

This is example code for handling the bookkeeping of the RZ filename and top level directory. The DASH command would be `TALK MYMODULE RZFILE myfilename.rz N`, where the final argument specifies a new RZ file. See the HBOOK manual for more detail. As coded here, this would open the file, but booking would not occur until the command `BOOK MYMODULE` is given. The final call to `HREND` would occur on program exit or could be specified interactively with `CALL FINI MYMODULE`.

```
      CHARACTER*80 RZFILENAME
      DATA RZFILENAME/'MYMODULE.RZ'/ ! Default filename
...
      ENTRY MYMODULE_TALK
c
c     Get first word after TALK in command line and act based on it:
c     ==============================================================
      CALL KUGETC(VERB,LEN)
      IF (VERB(1:3) .EQ. 'RZF') THEN
c
c        Get the filename and CHOPT options from the command line:
c        ---------------------------------------------------------
         CALL KUGETC(RZFILENAME,LEN)
         CALL KUGETC(CHOPT,LEN)
         CALL HROPEN(1,'MYNTUPL',RZFILENAME,CHOPT,LRECL,ISTAT)
         RZOPEN = .TRUE.
      ELSE
         IF (VERB(1:4) .NE. 'HELP') THEN
            WRITE(*,*) 'Unrecognized MYMODULE command', VERB
         END IF
         WRITE(*,*) 'Valid MYMODULE commands: '
         WRITE(*,*) 'RZFILE <filename>'
         WRITE(*,*) 'HELP'
      END IF
      RETURN

      ENTRY MYMODULE_BOOK
c
c     Book NTUPLE- but only if file is open!:
c     =======================================
      IF (RZOPEN) THEN
         CALL HBOOKN(30,'MYNTUPL',3,'//MYANA',1000,CHTAGS)
      ELSE
         WRITE(*,*) 'RZ file not open! Try TALK MYMODULE HELP'
      END IF
      RETURN
```

```
ENTRY MYMODULE_FINI
IF (RZOPEN)
   CALL HCDIR('//MYANA')
   CALL HREND('MYANA')
   RZOPEN = .FALSE.
ELSE
   WRITE(*,*) 'No RZ file to close'
END IF
RETURN
```

D MADLIB: A library of example modules

```
AAA_README_ABOUT_MAD                          E. Kearns, G. Ludlam, C. Okada
============================================================================

This documents the files in the Muon Analysis DST (MAD) library. It
includes the standard MUA (also Muon Analysis) code for reading the DST
files made on VAXGS. The library contains several DASH modules as well
as utility subroutines called by those modules. You may link to this
library and use the utility subroutines outside of the DASH environment-
you might find the print routines especially useful.

To use these routines on VAXGS, simply link to [MACROUSA.MAD]MADLIB/L.
You should also point your include statements to the include files in
this directory. This directory will always be the current official
release, the source code and development versions will be kept in
subdirectories such as [.V21]. To install on a remote computer, see the
instructions in [.V21]AAA_HOW_TO_INSTALL.TXT.

In this area I have included a copy of the DST common block that you can
point to. The latest version of the DST common block now holds all of
the variables used in the past (e.g. IMBU data) as well as the current
version (whatever that might be). So regardless of which read
subroutines you use, all of your analysis code should now *** include
'dst_common.inc' ***. Hint: use lower case filenames to make migrating
to unix easier.

MADREAD  is a DASH read-module that reads the unformatted DST events into
         the DST common block. It calls the MUAREAD subroutine which
         contains the big read statement that describes the DST
         structure. You will see several versions of MUAREAD in this
         area as well as hooks in the MADREAD module to select which one
         to call. This was put in when there were multiple file versions;
         the current MUAREAD is advertised to read all versions past and
         present (.TDST and .DSTE for example); this is the routine called
         by default, so do not worry about the extra routines and hooks.

MADWRITE is a DASH write-module that writes the DST information back out
         in the same compressed form that MUADST does. It calls MUAWRITE
         which contains the big read statement changed into a write
         statement.

MADDUMP  is a DASH module that dumps printed information from the DST
         common block to the screen. It contains a TALK entrypoint which
         you can use to turn off information you are not interested in.
```

MUAPICT is a DASH module which uses HIGZ to make a graphics display of the information in the MUADST common block. If you include this module, be sure to perform the HIGZ initialization as described in the DASH manual. You will find that initialization also in EXAMPLE.FOR (but commented out).

DST2ZEB is a DASH module that converts the DST information into a ZEBRA structure. The ZEBRA structure mostly contains the compressed information, but due to overhead, is about twice as big as the binary DST files. The primary purpose is to write exchange format files for transfer to Unix. This directory contains INCLUDE files you should use that define the offsets into the bank for each variable. The file ZEB_COMMON.INC sets up the ZEBRA store and declares the arrays IQ and Q that contain the banks.

ZEB2DST is a DASH module that converts the ZEBRA structure back into the DST common block.

ZREAD is a DASH read-module that reads in ZEBRA structures. It will actually read in any ZEBRA structure, but special hooks have been put in to recognize ZEB2DST banks and RAWR banks (MACRO raw data). It will do byte-swapping on MACRO RAWR if necessary (the DREAM routine BYTSWP is included in this library). The TALK entrypoint can call of the DZ diagnostic subroutines. Try TALK ZREAD DZSURV after reading in an event. The record length is 900 bytes by default, but can be specified such as TALK ZREAD LREC 5400 (LREC appropriate for MACRO raw data). The CHOPT for the FZFILE call is 'LXP' by default, but can be specified as the optional third argument to the INPUT command; for example: to read events from tape under unix, we use: INPUT /dev/nrtape TPLX.

ZWRITE is a DASH write-module that writes out ZEBRA structures. It has the same interface as ZREAD for specifying LREC and CHOPT, but it does not share these variables- they must be set separately.

MUAPRINT is a set of utility subroutines that print the variables in the DST common. It is aclled by MADDUMP, however you may use this outside of DASH by calling MUAPRINT_ERP(LUN) and so on.

Since these modules are useful to convert the DST common block into ZEBRA banks for exchange with UNIX computers, let me provide an example and remind you how DASH handles multiple READ/WRITE-modules. This is an instance where you would use the rare DASH command SET/READ and SET/WRITE. Let us assume that you built an executable linking in MADREAD, MADWRITE (declared primary read/write modules in subroutine DASH_DECL with DASH_DEC_REA and DASH_DEC_WRI) as well as ZREAD and

ZWRITE. Here are example sessions:

```
(Convert DST to ZEBRA)
DASH> INPUT RUNxxxxxx.DST
DASH> PATH DST2ZEB
DASH> SET/WRITE ZWRITE
DASH> OUTPUT ON RUNxxxxxx.ZDST
DASH> ANA

(Convert ZEBRA to DST)
DASH> SET/READ ZREAD
DASH> INPUT RUNxxxxxx.ZEB
DASH> PATH DST2ZEB
DASH> OUTPUT ON RUNxxxxxx.DST
DASH> ANA
```

Note: this is also an unusual instance where the order of the commands matters somewhat. You must SET/READ before INPUT and SET/WRITE before OUTPUT so that the files are opened correctly by ZEBRA.

Here are some notes about working with MACRO-style ZEBRA raw data (GJL): ZWRITE writes the data out with bytes in the VMS order. That is, the data is the same as if it had come directly off of a data tape. So when reading in UNIX, it will generally be necessary to swap the bytes. We check using the data whether or not this is necessary by comparing the header with the bank. (The header is always in the correct order.) ZREAD does this automatically.

When opening a file to write out in DASH, there is generally no need to specify the CFOPEN options. The correct (default) options are LXO. If for some reason one wishes to write out the data in 'SWAPPED' format (the format that UNIX understands), then do the following in DASH: TALK ZWRITE SWAP 0. This will turn off the 'unswapping'.

For MACRO data, the record length must be specified (5400 bytes, currently) because the default behavior for ZREAD and ZWRITE is set for the DST ZEBRA files (900 bytes). Use TALK ZREAD LREC 5400 and TALK ZWRITE LREC 5400 to make this correct.

Here is a synopsis of how to correctly read in and out macro data:

```
(start dash program)
TALK ZREAD LREC 5400
TALK ZWRITE LREC 5400
INP FILENAME  (for a file on disk)
     or INP /dev/nrtape tplx (for a file on tape)
OUTPUT ON OUTFILE
```

Very simple. Now, this output file should be readable by DREAM on the same machine. To read in the file on a VMS machine, do the following:

First, FTP the file to the VMS machine in BINARY mode. This will transfer the file in 512 byte blocks. That is, the BLOCKSIZE will now be 512. However, we need the blocksize to be 21600 (which is 4 times the record-length (5400)). So we have to run a command file to change that. The command file is printed on p. 56 of the ZEBRA FZ reference manual. It exists on buphyc::buphyf$dub0:[macro.ludlam.anal.wfd] and is called sgitovax.com. It uses the vms utility EXCHANGE. One can specify the BLOCKSIZE here, and it needs to be 21600. (The example in the manual gives 3600.) When that is done, the file is DREAM readable.