RAL-92-076

Science and Engineering Research Council
Rutherford Appleton Laboratory
Chilton DIDCOT Oxon OX11 0QX

RAL-92-076

HUMPF Users Guide

P Cahill R Edgecock S M Fisher C N P Gee J C Gordon T Kidd J Leake D J Rigby and J H C Roberts

December 1992

HUMPF Users Guide

P. Cahill, R. Edgecock, S. M. Fisher,

C. N. P. Gee, J. C. Gordon, T. Kidd,

J. Leake, D. J. Rigby and J. H. C. Roberts

18 November 1992

This document introduces users to HUMPF (Heterogeneous Unix Montecarlo Production Facility). The work was carried out as part of an IBM/SERC Joint Study Agreement.

Contents

Chapter 1

Introduction

The Heterogeneous Unix Montecarlo Production Facility (HUMPF) simplifies the running of particle physics simulation programs on Unix workstations.

Montecarlo is the largest consumer of IBM CPU capacity within the Atlas centre at RAL. It is likely that the future computing requirements of the LEP and HERA experiments cannot be satisfied by the IBM 3090 system. HUMPF adds extra capacity, and can be expanded with minimal effort.

Montecarlo programs are CPU-bound, and make little use of the vector or I/O capacity of the IBM 3090. Such programs are therefore excellent candidates to use the spare capacity of powerful workstations. The main data storage is still handled centrally by the IBM 3090 and its peripherals. The HUMPF facility is suitable for any program with a similar profile.

1.1 What is HUMPF?

HUMPF is an environment which includes, along with the Unix operating system:

- The Network Queueing System (NQS). CERN has purchased NQS with a license allowing them to modify and redistribute it to collaborating institutes.
- Access to tapes on VM by the 'tape' command. This is a utility produced by CCD as part of this project. It uses their Virtual Tape Protocol (VTP) running over TCP/IP to make centrally managed tapes accessible to Unix machines.
- Access to disk files on VM using the Network File System (NFS). This also runs over TCP/IP and makes files on a remote machine appear to be part of the local file system.
- Transfer of complete disk files using 'ftp'. This service is available between Unix machines and those VAX/VMS and IBM machines which support TCP/IP.
- An up to date copy of the CERN program library.

The tools have been installed on most Unix machines in PPD. Please contact the authors if HUMPF is not available on your system.

HUMPF makes full use of the data storage facilities accessible from the IBM system. These include the tape robot accessed via the transparent tape system, and disk space which is virtually unlimited as mini-disks spend most of their time on tape and occupy

physical disk space only when they are accessed. Storing Unix files on VM mini-disks lightens the systems maintenance load on the Unix side as there is much less data to back up.

This document is addressed to those who wish to install and run a Montecarlo program using the HUMPF facilities. Much of the information can also be found in the 'man pages' (the Unix on-line help system). The document is not meant to be a Unix introduction, though it does contain some information to help users who are not yet fluent in Unix.

The NQS batch system is explained below, along with the procedures for accessing centrally managed tapes. The use of NFS and ftp for transferring or accessing disk files on different machines is discussed and there is a short section on how to get a program going.

Chapter 2

The job submission system, NQS

NQS, the Network Queueing System, is a network-based batch job submission system that runs across a wide variety of Unix implementations. Jobs may be processed on the user's local system or on a remote system. The user may request that a job runs on a particular machine or class of machine.

A user job is written as a 'shell script', which is the Unix term for a command file. The script may also contain a header describing the resources needed. The most important features of three of the NQS commands are described here.

qsub to submit a batch job

qstat to display the status of the batch queues

qdel to delete a batch job

2.1 qsub – submit a batch job

qsub submits a batch job to NQS. The user will normally give the name of the script file as an argument. If no file name is specified, commands to be executed in the batch job are read directly from the standard input (stdin), terminated with control-D (the Unix end-of-file indicator). If the script requires parameters, the qsub command can be used without arguments to allow script name and parameters to be entered as a single command line for the job.

The script file is spooled, so that later changes will not affect previously queued batch jobs.

If the batch job is successfully submitted, its job-id is displayed on the terminal. A job-id is always of the form: seqno.hostname, where seqno refers to the sequence number assigned to the job by NQS, and hostname refers to the name of the originating machine. This identifier is used throughout NQS to identify the job, no matter where it is in the network.

2.1.1 The environment

The job will, in due course, execute on some machine chosen according to the queue to which it is submitted and other controls. The script will start execution in the home directory of that machine, as if the user had logged in to that machine and executed the script. The owner of the process is the submitter of the job.

Several environment variables are set up by NQS for the job.

The variables HOME, SHELL, PATH, LOGNAME (not all systems), USER (not all systems), and MAIL are set from the user's password file entry, as though the user had logged directly into the execution machine.

The variable ENVIRONMENT is set to the value BATCH, so that shell scripts and the user's .profile (Bourne shell) or .cshrc and .login (C-shell) scripts can test for batch job execution as appropriate.

The variables QSUB_WORKDIR, QSUB_HOST, QSUB_REQNAME, and QSUB_REQID denote the working directory at the time the job was submitted, the name of the originating host, the name of the job and the job-id.

2.1.2 Control of output

By default, all stdout output for the batch job is saved in a file whose name consists of the first seven characters of the job-name (normally the name of the script) followed by the characters: '.o' and the job sequence number portion of the job-id. This file will be returned to the machine that originated the batch job in the current working directory, as defined when the batch job was first submitted. The file is normally sent to its destination at the end of the job, but can be written directly, allowing it to be read during execution, if the **-ro** option is specified.

The stderr file, which receives error messages, is handled in a similar way. It has a file name including '.e' instead of '.o' and may be written directly with the **-re** option.

The job script must take responsibility for any other file movements needed between the submitting and execution machines – only the standard input, output and error files are moved automatically.

2.1.3 Options in the script file

Options can be specified both in the qsub command line and also within the first comment block inside the batch job script file. Command line options take precedence, so embedded options are defaults for the job if no command line value is given.

If the value of an option has two or more tokens separated by white-space characters, the value must be placed within double quotes as in `-a "July, 4, 2026 12:31"` (or otherwise escaped) such that qsub and the shell will interpret the entire value as a single character string.

The use of embedded options within a script file is illustrated below:

```
#
#  Batch job script example:
#
#  @$-a "11:30pm" -lT"22:00, 20:00"
#              # Run job after 11:30,
#              # and set a maximum CPU time of 22 minutes
#              # Send a warning signal after 20 CPU minutes
#  @$-mb -me   # Send mail at beginning and end of
#              # job execution.
#  @$-q batch1 # Queue job to queue batch1
#
make all
```

2.1.4 Syntax

qsub [*option* ...] [*script_file*]

-a *time* : Do not run the batch job before the specified date and/or time.

The syntax accepted for the *time* parameter is flexible. It is possible to specify the date as a weekday name (e.g. 'Tues'), or as one of the strings 'today' or 'tomorrow'. Weekday and month names can be abbreviated. The default 24 hour clock may be overridden by 'am' or 'pm'

Some valid *time* examples are:

```
-a "01-Jan-1986 12am"
-a "Tuesday, 23:00:00"
-a "11pm tues"
```

-lT *maxtime* [, *warntime*] : Set a maximum and an optional warning cumulative CPU time limit for all the processes that make up the job. If the CPU time exceeds *maxtime*, then all the processes in the job will be stopped. The ability to act on a warning limit is supported only for some Unix systems. When such a warning limit is exceeded, a signal should be delivered to one or more of the processes of the running job. The job will stop immediately on receipt of a signal unless it includes a signal handling routine – see the `'signal'` man pages for further details.

-mb : Send mail to the user on the originating machine when the job starts executing.

-me : Send mail to the user on the originating machine when the job ends.

-p *priority* : Assign a queue priority to the job to define the relative ordering of jobs within the queue. The specified *priority* must be an integer in the range [0..63] where a value of 63 defines the highest *priority*. The relative ordering of jobs within a queue does not always determine the order in which the jobs will be run. The scheduler is allowed to make exceptions to the job ordering for efficient machine resource usage.

-q *queue* : Specify the queue to which the batch job is to be submitted. If no -q *queue* is given, the value of the environment variable QSUB_QUEUE is used, or failing that the system default is taken.

-r *job-name* : Assign the specified *job-name* to the job. By default the name is taken from the name of the script file which was submitted (or STDIN). The character 'R' always precedes the job-name if it would otherwise begin with a digit. Job-names are truncated to 15 characters.

-re : Write the stderr file directly instead of spooling it at the end of the job.

-ro : Write the stdout file directly instead of spooling it at the end of the job.

2.2 qstat – display status of NQS queue(s)

qstat displays information about the NQS queues and the jobs they contain.

2.2.1 Syntax

qstat [**-a**] [**-l**] [**-m**] [**-x**] [*queue-name ...*] [*queue-name@host-name ...*]

Information may be requested for a specific queue or queues. If none is specified the current state of each NQS queue on the local host is displayed. Queues may be specified either as queue-name or queue-name@host-name. In the absence of a host-name specifier, the local host is assumed.

The general state of a queue is defined by two principal properties of the queue, which qstat displays as part of the queue header. The first property determines whether jobs can be submitted to the queue. If this has the value ENABLED then jobs can be submitted. The second property describes the execution state of the queue. When the queue is functioning normally then it should be either RUNNING if a job is running or INACTIVE if there is no work for it at the moment.

For each selected queue, qstat displays the queue header followed by information about jobs in the queue. The following options are available to modify the default, which is a summary of your jobs:

-a : Shows all jobs instead of just yours.

-m : Jobs are shown in a medium-length format.

-l : Jobs are shown in a long format.

-x : The queue header is shown in an extended format.

2.3 qdel – delete or signal NQS job(s).

2.3.1 Syntax

qdel [**-k**] *job-id ...*

qdel deletes one or more queued (not executing) NQS jobs whose job–id is listed on the command line. Additionally, if the option **-k** is specified, a kill signal will be sent to the job if it is running. This will cause the receiving job to exit and be deleted.

The job–id of an NQS job is displayed when the job is submitted, and can be obtained later by the qstat command.

Chapter 3

Access to centrally managed tapes

The Virtual Tape Protocol (VTP) system enables programs and applications running in workstations to read and write data on magnetic tapes in the main tape library as if the tapes were mounted on a local tape drive. The communication path is as transparent as possible between a 'named pipe' on the workstation, and the server software, currently installed on VM, which actually does I/O to the tapes[1]. Because the named pipe behaves to the user and applications like a conventional Unix file, it is not normally necessary for the user to make any changes to the application code to access tapes over VTP.

The system works with any format of tape, since it simply views a tape as a series of bytes and tape marks. It leaves the problem of formatting the data on the tape to the application, just as a real tape drive does. An alternative would be to transfer whole tapes of data onto the workstation disks and process it there. The remote tape system has all the advantages of record-level I/O: no need to transfer large amounts of data which will not be used; and high speed positioning within a tape. It is however as fast as other access methods for bulk transfers.

Both IBM Standard Label (SL) and No Label (NL) tapes are supported. The SL tapes may have labels in IBM EBCDIC format or the ANSI ASCII format.

3.1 The tape command

Access to tapes via VTP is provided by the HUMPF 'tape' command. The easiest way to use the tape command is to read or write from a named pipe – which appears to FORTRAN like an ordinary file. Alternatively, the 'tape' command can read from stdin or write to stdout (and so may be piped in the Unix style) or it can be used to read or write directly from or to a normal Unix file.

The 'tape' command works in units of one file. In read mode it reads until a tape mark is found, and in write mode it writes a single file on the tape.

Access to tapes is controlled at the server end by the Tape Management System (TMS). You must use TMS, which currently has an interface to VM/CMS but not to HUMPF, to ensure that the **userid** you specify on the 'tape' command is allowed to perform the requested read or write action on the tape with name **volid**. See section 3.2 below for details.

[1] The server could in principle interface directly to real tape drives. Instead it takes advantage of the VM 'Transparent tape Staging System', which provides an interface (almost) identical to tapes on real tape drives, but makes effective use of disk and the 3480 robot to reduce the number of real drives required and reduce the delays and contention between users associated with using real tape drives.

3.1.1 Syntax

tape *volid userid [password]* [*option ...*]

Where:

volid : The name of the tape you wish to use. This name may be up to 6 characters long.

userid : A valid username on the server. This username may be up to 8 characters long. The user must have the desired level of TMS access to the tape as explained above.

password : Optionally, the password for the specified *userid* on the server. The preferred security method, described below in section 3.2 avoids the need for passwords. Passwords should not be used in job scripts.

The *volid, userid*, and optional *password* must be in that order with interspersed options as desired. Each option must be preceded by a '-'.

Options **-r**, **-w**, **-rp** and **-wp** define the basic mode of operation and are mutually exclusive. **-r** is the default.

-r : Reads from tape to the standard output, or to a named file if the **-f** option is specified.

-w : Writes to tape from the standard input, or from a named file if the **-f** option is specified.

-rp : Reads data from the tape through a named pipe.

-wp : Writes data onto the tape through a named pipe.

The following options require values. No space is allowed between the option character and its value.

-f*fname* : Specifies *fname* as the name of the file (for **-r** and **-w** options) or as the name of the pipe (for **-rp** and **-wp** options). Default values of *fname* are:

-r : stdin
-w : stdout
-rp : taperead
-wp : tapewrite

-l*label* : The type of tape labels to use. Specify **-lnl** for no label (NL) tapes. Specify **-lsl** for standard label (SL) tapes. The default value is **-lsl.**

-m*filenum* : Move the tape to the file with sequential position *filenum* on the tape before reading or writing. The default value is **-m1** which corresponds to the first file on the tape.

-n*dsn* : Use *dsn* as the data set name, otherwise the name is taken from *fname.*

-b*blksize* : The size of the block to be used on the tape. The default value is **-b8192.** The *blksize* may be set from 1 to 65535.

3.1.2 Examples

These examples all relate to a tape with a *volid* 'mytape' and belonging to server *userid* 'auser' with appropriate tape access (see section 3.2 below).

```
tape mytape auser
```

causes the first file on the standard label tape named mytape to be displayed on the terminal. This is generally not a good idea unless the tape contains a small file of ASCII data. If it contains 'binary' data then a more useful command would be:

```
tape mytape auser | od -x | more
```

which pipes into 'od' to create a hexadecimal dump and then into 'more' to paginate the output.

The next example shows the use of named pipes. This is probably the most useful style of working for those who want to read and write data from FORTRAN programs.

```
tape -rp mytape auser -ftapein &
tape -wp mytap2 auser -ftapeout &
myprog
```

The first command makes a tape file available for reading. The command line is terminated by an '&' to run tape concurrently in a background process which is able to keep the data flowing down the pipe as the program removes data from it. The program sees a file called tapein in the current working directory, so it may be read by a FORTRAN program with an open statement like:

```
OPEN(LUN,FILE='tapein',...)
```

The second of the pair of tape commands is similar but allows a tape file to be opened for writing. After connecting the tapes to the pipes the program (myprog in this case) can be started. If the program closes the files properly the background tape jobs will be stopped and the pipes deleted. If the job crashes you will need to do this yourself.

```
tape mytape auser -foutput
```

causes the first file on the standard label tape named mytape to be copied in the file 'output'. This could require a lot of disk space on the Unix workstation.

```
tape -w mytape auser -finfile -nbert
```

will cause the file 'infile' to be written onto the standard label tape named mytape as the first file on the tape. The dataset name 'bert' will be used in the file header rather than 'infile'.

```
cat myfile | tape -w mytape auser -lnl -m2
```

will write the data from the file 'myfile' onto the NL tape named mytape as the second file on the tape.

3.2 Authorising Tape Access

Access to tapes in the central tape store is controlled by the Tape Management System (TMS). This section explains how the access check works and how to enable access from Unix systems.

Before any tape access is permitted, an appropriate TMS protection group must be set up. The following CMS commands create a new TMS tape protection group named MYPROT under account 1234, grant control (and implied read and write) access to userid AUSER, and apply the protection rules to tapes number 888881 to 888888 inclusive.

```
TMS PROTECT MYPROT ACC 1234 CREATE
TMS PROTECT MYPROT ACC 1234 AUTH GRANT CONTROL USERID AUSER
TMS PROTECT MYPROT ACC 1234 AUTH GRANT WRITE   USERID TAPENET
TMS PROTECT MYPROT ACC 1234 APPLY SET 888881 - 888888
```

Further details of TMS commands are available via the CMS command FIND TMS.

The following discussion relates to a request from a user with a Unix username of auser_ix who wishes to read tape with **volid** 888888 using the access rights of IBM **userid** auser:

```
tape -r 888888 auser
```

The tape command extracts the current Unix username using a secure system call. The username, **userid** and **volid** are then sent to the server.

The server authenticates the IBM **userid** using the AUTHRTY mechanism on the IBM mainframe. The nominated IBM **userid** must authorise the remote username at the remote host to inherit the tape access privileges as specified in TMS for the IBM **userid.**

3.2.1 AUTHRTY NAMES Entry Format

The user AUTHRTY NAMES file is checked using the following information and tags:

The (authenticated) username at the remote host	:user.
The host from which the request originated	:command.
The tape for which access is requested	:target.
The mode (read or write) for which access is requested	:group.

The tokens TAPER and TAPEW are used to specify read or write access. Checks are all carried out under the system name TAPENET. The following example entry allows the remote user 'auser_ix' on the system at IP address 130.246.8.1 (the machine arcu.cc.rl.ac.uk) to access only **volid** 888888 in Read mode:

```
:nick.
   :system.TAPENET
   :command.82F60801
   :target.888888
   :group.TAPER
   :user.auser_ix
```

The present AUTHRTY checking mechanism supports strings of no more than 8 characters, so the IP address must be encoded as a hexadecimal string. The conventional IP format will be adopted when longer character strings are supported. Meanwhile, an IBM command procedure HEXNET EXEC U is available to convert between normal and hexadecimal IP addresses.

Wild characters may be used to simplify AUTHRTY entries. The check for access (TAPER or TAPEW) and the volid may be replaced with an asterisk to allow any access to any tape. The AUTHRTY wild character '+' may also be used. For example, an entry :target.8888++ would allow access to a sub-set of tapes starting with 8888 and followed by any two characters.

A general entry might be:

```
:nick.
   :system.TAPENET
   :command.82F628++
   :target.*
   :group.*
   :user.auser_ix
```

This will allow the remote user 'auser_ix' access to any tape (belonging to AUSER) in any mode provided the request comes from a machine in the RAL PP domain (130.246.40.xxx).

Chapter 4

Remote disk files

Data on remote disk files can be accessed either by linking to the remote file system over NFS or by copying the files locally via ftp. For a large file which is being updated regularly, NFS is probably better, but for a small stable file which is to be read many times it may be better to take a local copy using ftp. The examples here assume that the remote files are on VM/CMS, but they could equally be on any Unix, Vax or other system with NFS software.

4.1 NFS – The Network File System

NFS is a facility for sharing files between machines. It might be used for example to access a calibration data set stored on a VM mini-disk.

In general, NFS allows a remote tree of files to be grafted into the local file system. The remote directory replaces an existing local directory, so it is customary to keep an empty local directory for this purpose. If a non-empty local directory is used, everything in the directory becomes inaccessible while the remote directory is mounted. A VM mini-disk is viewed as a remote file tree containing a single directory.

For example:

```
mkdir ~/vm
mount ib.cc.rl.ac.uk:pubxu.211,ro,record=binary ~/vm
```

creates a new directory, vm, below your home directory then mounts the 211 disk belonging to user pubxu on ib.cc.rl.ac.uk. The disk is mounted read-only in binary mode (see below), and is assumed to have a read password of ALL so a read password need not be specified.

The next example unmounts the disk and mounts another disk in read-write mode in place of it. The mountpw command sends the mini-disk write password before the mount command is issued. The NFS server on VM will remember the mini-disk/password pair for about five minutes, during which time the mount command should be issued.

For example:

```
umount ~/vm
mountpw ib.cc.rl.ac.uk:pubxu.215,pass=writeit
mount   ib.cc.rl.ac.uk:pubxu.215,rw,record=binary  ~/vm
```

will unmount an existing mini-disk and mount a new one in read-write mode with a write password.

Many Unix versions do not allow ordinary users to mount a file system nor to export one (to make it available for mounting elsewhere). This is a potential problem accessing VM disks, because each desired mini disk must be individually mounted.

There are a few restrictions when files are written to VM mini-disks via NFS. In particular, it is not possible to create sub-directories on a VM mini-disk. As file names are more restricted under VM than under Unix, a file acting as a look-up table is created on the mini-disk to define the name conversion. A file with a name 'FNAME FTYPE' on VM which is not in this look-up table is presented to Unix as `fname.ftype`. Use of Unix file names conforming to VM file naming conventions will generally help (human) users, but is not necessary.

Unix has no concept of record length in files, so the record length is ignored on reading. Files are always written with `RECFM F LRECL 1`, but can easily be modified to a more convenient record length for reading under CMS. For example, a 36000 byte file 'FNAME FTYPE D' will be written with 36000 1-byte records, but is converted to ten 3600 byte records with the CMS command

```
CHLRECL FNAME FTYPE D 3600
```

A disk accessed with `record=binary` can be used as an extension of the Unix filing space to hold any type of file since data are stored on the VM disk without conversion. The alternative `record=text` is suitable for text files which are stored as EBCDIC on VM and converted to and from ASCII for the Unix side. Text files can of course be stored on a record=binary mounted mini-disk, but as characters will be represented in ASCII they will be difficult to work with from VM. If a mini-disk contains some files to be accessed in binary mode and some in text mode, the disk can be mounted twice as separate Unix directories.

Unfortunately it is a feature of the current IBM software that the umount request is ignored, so you have to log on to VM/CMS and tell VMNFS to detach the disks. The CP command:

```
SMSG VMNFS 1M QUERY
```

will display a list of disks known on the VM side, and

```
SMSG VMNFS 1M DETACH PUBXU.211
```

will detach PUBXU's 211 minidisk. The need to issue DETACH commands will be removed when VMNFS version 2.2 is installed.

This feature of the software makes writing disk files rather unpleasant, because you can mount the disk for reading many times and VMNFS will just get another read link to the disk – but VMNFS only allows one write link.

VMNFS takes a private copy of a minidisk directory, so the Unix side will not see directory changes (eg new files written by CMS) after the read link is established. The command

```
SMSG VMNFS 1M REFRESH PUBXU.211
```

instructs VMNFS to take a fresh copy of PUBXU's 211 minidisk directory.

4.2 ftp – File Transfer Protocol

It may be preferable to copy frequently accessed files to the target Unix machine if disk space permits, using ftp. Unfortunately, ftp implementations do vary quite a lot. In general, is wise to move to the appropriate directory before issuing an `ftp` command. The name of the target machine should be given on the command line, for example enter the command `ftp ib.cc.rl.ac.uk` to perform transfers to or from the machine with internet address ib.cc.rl.ac.uk.

When connection has been established, ftp will prompt for a command. Details of commands may be obtained from the man pages. Useful commands include:

help	print help information
cd	change remote directory
ls	list remote directory
ascii/binary	select 'ASCII' or 'binary' transfer mode
type	see whether ascii or binary transfer mode
get/put	get/put a file
mget/mput	get/put several files
prompt	toggle to switch on or off prompts from mget/mput
quote	send a message to the remote server
quit	get out of ftp

The file on Unix is just a string of bytes. The quote command can be used to send a message to the remote server to say how the file is to be created there. For example to issue SITE commands on VM:

```
quote SITE FIX 3600
```

causes files transferred to VM to be RECFM F LRECL 3600.

It does not appear to be possible to do the same thing when transferring a file to VMS. Instead after transferring the file it should be converted by using a file such as CONVERT.FDL shown below:

```
RECORD
BLOCK_SPAN          yes
CARRIAGE_CONTROL    none
FORMAT              fixed
SIZE                3600
```

to convert old_file to new_file by the command:

```
$ EXCHANGE/NET/FDL=CONVERT old_file new_file
```

4.2.1 anonymous ftp

If you just want to take a copy of public files then you can often use a user name 'anonymous' or 'ftp'. Not all installations support anonymous access, and a password may or may not be requested. Many systems request your real name or network address instead of a password. If you are prompted for a password when using anonymous ftp to a VAX/VMS system, try entering a null password (just hit return).

Chapter 5

Preparing and running a program

To run a FORTRAN program under Unix, you need to know a little about a shell (Korn shell, C-shell or Bourne shell) and an editor.

5.1 Editors

Various editors are available. The standard Unix editor `vi` has only its availability to commend it. Distributed with the X window software is a simple editor, `xedit`, which is easy to use and makes use of the mouse, but will not do very much. The third editor to mention is emacs from the Free Software Foundation. It is powerful – but as it was designed to run on the dumbest of dumb terminals needs strange control sequences to make it do things. You must at least know Control-x Control-c to exit from it. It works well with X windows and there is an edt emulation available for those unable to retrain their ageing fingers.

5.2 Preparing the input for the compiler

Common blocks may either be expanded (by PATCHY or some other mechanism) or 'include' files can be used. The C preprocessor can be run by the fortran compiler driver, giving the interesting possibility to use conditional code without the weight and mystery of PATCHY/CMZ. Using `\#include` rather than `include` also allows you to specify at compile time in which directory(s) are the files to be included. If you wish to create a compiled library, each FORTRAN routine should be in a separate file. A file containing many routines may be split using `fsplit` – but note that fsplit will not recognise labelled END statements (which are legal FORTRAN).

5.3 Compiling and Linking

The FORTRAN compiler (f77 on most machines, xlf on the IBM RS6000, fort77 on the HP9000/700) is not actually a compiler at all but a program which will run the C preprocessor, the C compiler, the FORTRAN compiler, the FORTRAN optimizer and the loader. The f77 command is probably the most non-standard command in Unix, and is also rather tricky as its options have to be passed down to the programs it calls. A consequence of this is that erroneous options tend to be ignored rather than giving an error. The **-v** (verbose) flag will allow you to see what the f77 command is doing. f77 requires a list of files, which it processes according to the part of the filename after the

last dot. The .c files are passed to the C compiler, .f files to the FORTRAN compiler, and .o files are passed straight to the loader. Libraries are introduced by a **-l** option., e.g. `-lpacklib` which will selectively load routines from the file libpacklib.a. (The lib prefix is not a typing error!) Useful options to look for in the man pages include:

-c : compile only

-g : debug control

-O : optimization control

-static : ensures static allocation of all variables. For those who have got all their FORTRAN SAVE statements correct this is not needed – but for the average HEP code this option is essential as the more modern Unix compilers often, by default, allocate the space for variables when they are needed and give them up when the routine is exited.

-cpp : run the c preprocessor on .f files.

5.4 Libraries

The .o files may be stored in libraries. Use the `ar` command to build the library. For loading several files initially use the **-q** flag which prevents the program wasting time checking for duplicate entries. The ar provided with some versions of Unix does not produce a symbol definition file (index) to the library; instead you build the library by adding in many files and then index it in one go. Other versions of Unix maintain the index for you. You should get a message if the library has no index and you try to use it. Possible commands to build the index are `ranlib` or `ar ts`

5.5 The CERN library

The cern library is found in the tree with the root `/cern` – for example `/cern/pro/lib/libpacklib.a` is the production version of packlib. Instead of 'pro' the name of that version of the cern program library may be specified instead.

5.6 Running the program and tidying up

The program is by default called `a.out` and generally lacks the bit which allows it to be executed. To change this, if the file is called `a.out`, enter `chmod +x a.out` If your program does not run investigate the debugger, dbx, which may have a nice window style front end on your machine.

When you are happy with your program you can use the strip command on object files, libraries and programs to remove the symbol table information. This will reduce the size of the executable file on disk, but also means that a symbolic traceback cannot be produced should the program abend.

5.7 Simple example of a script

The first line of a script (Unix command file) should contain the name of the shell program which should interpret it. For example a csh script will start off:

```
#! /bin/csh
```

For this reason you can mix Bourne (sh), Korn (ksh) and C (csh) shell scripts as you wish. Included with humpf are two sets of demonstration scripts, one using the C shell as this is currently the most commonly used shell in scientific work, and one using the the Korn shell which, being closer to the POSIX shell, will probably take over.

To prepare the demonstration, issue the following commands (from any shell):

```
mkdir my_humpf_demo                          # make a directory for the demo
cd my_humpf_demo                             # make it current directory
cp -r /usr/local/lib/humpf/* .               # copy some files there
mv ksh/* .                                   # and move shell scripts up
mv humpf_configure.decs humpf_configure # rename configuration file
rm humpf_configure.*                         # get rid of the rest
cd ..                                        # return to the parent directory
```

The text following the # are comments. These are legal only if the commands are executed from a script and not interactively. The fourth line copies the ksh version of the scripts. (It could be replaced by `mv csh/* .` to get the csh scripts instead.) The fifth and sixth lines rename the appropriate configuration file, and delete the other configuration files. Files available for the various machines are:

Decstation	: humpf_configure.decs
Hewlett Packard	: humpf_configure.hp
RS6000	: humpf_configure.aix

The following commands will run the demo from the Korn shell:

```
cd my_humpf_demo                             # go to your humpf_demo directory
. humpf_configure                            # run the configuration file
./humpf_demo 809401 fisher                   # run the demo
cd ..                                        # return to the original directory
```

The second line runs the humpf_configure file in the current shell so that the environment variables it sets are not lost at the end of the script. (The equivalent csh command is: `source humpf_configure`)

The third line starts `./` which just specifies the current directory to ensure that humpf_demo is found correctly (independently of how your PATH environmment variable is set)

You can clean up afterwards with `rm -rf my_humpf_demo` which will destroy my_humpf_demo and all the files it contains recursively without asking any questions.

The file humpf_configure for ksh on a Decstation is:

```
#! /bin/ksh
set -a
#
# Define environment variables for HUMPF
#
# This is the decstation version
#
# Set compiler and flags
#
COMPILER_NAME=f77
COMPILER_OPTIONS="-g -w1 -assume backslash"
LOAD_OPTIONS=
#
# Set cern library version
#
CERN_VERSION=pro
#
# Define command to randomise a libarary
#
RLIB=ranlib
```

The `set -a` command on the second line ensures that all variables are 'exported' and so become 'environment variables' which can be used elsewhere. Note that the variable assignments have no spaces around the = sign. (The equivalent csh script uses the `setenv` command to set environment variables.)

The ksh version of humpf_demo starts:

```
#! /bin/ksh -p
#
# humpf-demo a script to demonstrate some of the humpf tools
#
function error_exit
{
        echo $@;
        exit;
}

[ ${#*} -eq 2 ] ||
   error_exit 'Must have parameters volser and tmsuser'
[ ${#COMPILER_NAME} -gt 0 ] ||
   error_exit 'Must . humpf_configure to set COMPILER_NAME etc'
```

The -p flag on the first line ensures that any script as defined by your ENV environment variable will not be run, as this could upset the script.

The error_exit function which is defined here is used by the next two statements to generate an error message and exit. In these two statements, the expression within square brackets if true satisfies the 'or' condition || and so the second expression which is the call to error_exit will not be evaluated. This ensures that the two parameters $1 and $2 have been provided and that the environment variable COMPILER_NAME has been set by the humpf_configure script.

The humpf_demo script continues ...

```
volser=$1                              # get the tape number
tmsuser=$2                             # the 'owner' of the tape
echo Will use tape $volser via TMS entry for user $tmsuser

./make_generate                        # Make the generate program

[ -a numbers ] && rm numbers           # remove numbers file if it exists
./generate                             # run the program
od -x numbers | head                   # hex dump the file and take the head
ls -lF numbers                         # and have a look at it.

rm numbers                             # remove numbers file,
mknod numbers p                        # replace it by a named pipe,
ls -lF numbers                         # and have a look at it.

tape -w $volser $tmsuser -fnumbers & # start background job to write tape
./generate                             # Run generate

./make_libintegrate                    # make the mc integration library
./make_integrate                       # make the integrate program

tape -r $volser $tmsuser -fnumbers & # start background job to read tape
./integrate                            # Run the integrate program
rm numbers                             # and finally remove pipe file.
```

After saving and displaying the input parameters, humpf_demo runs another script, make_generate, which builds the program called generate.

The line `[ -a numbers ] && rm numbers` is similar to the earlier case with `||` but here the `&&`, i.e. 'and', ensures that `rm numbers` is only executed if `[ -a numbers ]` is true i.e. if the file exists.

The rest of the script executes the generate program which writes to a file called numbers. The beginning of this file is displayed on the terminal as a hex dump. The numbers file is then replaced by a pipe file of the same name, and the same program run again after starting up the tape command in the background to write the data from the pipe on to a tape.

The make_generate script[1] is very simple:

```
#! /bin/ksh -p
#
# make_generate a ksh script to build the generate program
#
$COMPILER_NAME $COMPILER_OPTIONS generate.f -L/cern/$CERN_VERSION/lib -lgenlib ||
   exit
mv a.out generate
chmod +x generate
```

[1] The Unix make utility could also be used to build a program rather than such a script.

The FORTRAN code generate.f is shown below:

```
      PROGRAM GENERATE
      IMPLICIT NONE

      INTEGER I,IOS
      REAL    X,RN32
C
C       Generate 1000 random numbers and store them on a file
C
      OPEN(10,FILE='numbers',FORM='UNFORMATTED',STATUS='UNKNOWN',
     +     IOSTAT=IOS)
      IF(IOS .NE. 0)THEN
         PRINT *,'Failed to open file'
         STOP
      ENDIF

      DO 10 I = 1,10000
        X = RN32(I)
        IF(I .LT. 10)PRINT *,'Random number is ',X
        WRITE(10)X
 10   CONTINUE
C
C       Write a flag on the end of the file (a negative number)
C
      WRITE(10)-1.0

      END
```

The OPEN statement in the FORTRAN opens the file with STATUS='UNKNOWN' to ensure that the program will write to an ordinary file or to a pipe. It does not matter whether the tape program is run first or second and either or both may be run in the background.[2].

As a single Unix pipe can only transmit data and not control information, a process reading from a pipe cannot use the END= on the READ statement of a FORTRAN routine to know when there is no more data.[3] As a consequence, the reading program must use some other means to know when to stop reading. This is not a problem in practice, because formats such as FZ of ZEBRA have end of file information before the physical end of file. In the case of this humpf_demo, a negative word is used to flag the end of file.

[2]The option on the tape command to allow it to create the pipe and destroy it at the end is not used. In principle it can lead to synchronisation problems if the application program opens the file first with STATUS=UNKNOWN it will create an ordinary file and the tape command will fail to open a pipe of the same name.

[3]This is only a problem for FORTRAN (which is record oriented in its i/o). In C the routine read returns 0 as the number of bytes read to signal an end of file when attempting to read from an empty pipe if no process has the pipe open for writing. This is fortunate, as otherwise the tape command would not stop when it was reading from the pipe and writing to a tape.

5.8 Unix version dependency

Below are listed for some machines:

- the command to compile with a reasonable set of options, including the debugger. Better performance for stable programs will be obtained by removing debug options and enabling optimisation.
- whether to escape a '\' character in the FORTRAN source by a second '\'
- the set of libraries to accompany grafX11 which is the X11 HIGZ library
- the command to index a library

RS6000(AIX)

```
Fortran: xlf -g -w -qextname -qcharlen=30000 -NQ30000 \
             -NT120000 -NP1000 -ND20000 -NN100000
\\     : yes
X11    : -lX11 -lm
index  : ranlib
```

Decstation (Ultrix, Dec f77 Compiler)

```
Fortran: f77 -g -w1 -assume backslash
\\     : no
X11    : -lX11 -lcursesX -lm
index  : ranlib
```

Silicon Graphics

```
Fortran: f77 -g2 -O0 -G3 -static
\\     : yes
X11    : -lXm -lXt -lX11 -lPW
index  : not needed
```

Sun (SunOs)

```
Fortran: f77 -i4 -w -g
\\     : yes
X11    : -lX11 -lm
index  : ranlib
```

Apollo Dn10000(Domain)

```
Fortran: f77 -w -g
\\     : yes
X11    : -L/usr/lib/X11 -lX11 -lm
index  : not needed
```

Hewlett Packard 9000/700(HP-UX)

```
Fortran: fort77 -c -K -w -g +ppu
\\     : no
X11    : -L/usr/lib/X11R4 -lX11 -lm
index  : not needed
```

5.9 Support for FORTRAN I/O

This section discusses use of FORTRAN binary i/o on different systems.

The Unix file system is not as record oriented as FORTRAN. Unix treats files as a sequence of characters rather than a collection of records, and record handling is provided by the FORTRAN run-time system. Care is needed in particular when moving binary data files between Unix and non-Unix systems.

Below is illustrated a code fragment which writes a binary record, and a hexadecimal dump (using `od -x`) of the output which is produced on different systems:

```
      PROGRAM HUMPF_TEST
      REAL * 4 F
      CHARACTER * 4 NAME
      INTEGER * 4 I
      DATA F/10.0/,NAME/'FRED'/,I/25/
      OPEN(UNIT=10,FILE='TEN',FORM='UNFORMATTED',STATUS='NEW')
      WRITE(10) F,NAME,I
      END
```

```
Sun 3/160     0000 000c 4120 0000 4652 4544 0000 0019 0000 000c
Dn10000       0000 000c 4120 0000 4652 4544 0000 0019 0000 000c
HP9000/700    0000 000c 4120 0000 4652 4544 0000 0019 0000 000c
RS6000        0000 000c 4120 0000 4652 4544 0000 0019 0000 000c

Decstation    000c 0000 0000 4120 5246 4445 0019 0000 000c 0000

VAX(VMS)      0e00 0300 2042 0000 4652 4544 1900 0000
IBM(VM/CMS)   0014 0000 0010 0000 41a0 0000 c6d9 c5c4 0000 0019
```

In the Unix systems, the 12 bytes of user data are preceeded and followed by a record length specifier `0000 000c` (the byte order is different on the Decstation). In the non-Unix systems, only a leading record length specifier is present, but extra bytes are inserted to help in mapping FORTRAN records to file system records. Furthermore, the representation of floating point and character values differ.

Index