Miriad

Multichannel Image Reconstruction,
Image Analysis and Display

Programmers Guide

Bob Sault and Neil Killeen

18 Jun 2014

See http://www.atnf.csiro.au/computing/software/miriad

Contents

List of Tables

Chapter 1

Program Development

Programs are developed in the usual manner, making calls to the *MIRIAD* subroutine library. It may be convenient to pass code through the RATTY preprocessor before compiling. RATTY preprocesses a few language extensions into standard Fortran, and flags a few bad programming practises. When linking, the *MIRIAD* object library, libmir, is used.

1.1 General Programming Conventions

MIRIAD tasks should be written in Fortran-77. Though you should program in standard FORTRAN, two extensions will generally be needed by the programmer. Firstly *MIRIAD* uses both upper and lower case character strings (Fortran-77 strictly supports only upper case characters. However almost all compilers support both upper and lower case, and it would be a reasonably simple preprocessing job to convert all of *MIRIAD* to a strictly upper case system if the need ever arises). Generally *MIRIAD* routines are case-sensitive, with lower case being preferred.

Secondly *MIRIAD* tasks should use the `maxdim.h` include file where appropriate. This include file defines a parameter, `maxdim`, which gives the maximum image dimension that a task should be prepared to accept. Currently this is set to 4096, but by using `maxdim` to define needed storage, it should be reasonably easy to rebuild all *MIRIAD* tasks to handle larger images. The include file also defines a parameter `maxbuf`, which is a guide to the maximum amount of internal data storage that a program should contain.

Despite the encouragement to use this include file, programmers are generally discouraged from using include files and common blocks. This is far from a strict rule. Avoid them if you can.

See Chapter 3 for a description of some utilities which simplify the development of code.

1.2 In-Line Documentation

Documentation for *MIRIAD* tasks and subroutines is included as comments within the body of the code (delimited by special "directives"). This documentation is stripped out by the `doc` program to produce a `.doc` file. This `.doc` file is then used by the on-line help facilities and the manual generation utilities.

This documentation should be at the head of the source code. In FORTRAN notation, the documentation "directives" are:

```
c=  [routine name] [one-line description]   (for programs)
c*  [routine name] [one-line description]   (for subroutines)
c&  programmer ID
```

```
c:  comma-separated list of categories
c+
c   start of multi-line description block
c@  keyword                                 (for tasks)
c   multi-line keyword description          (for tasks)
c<  standard keyword                        (for tasks - deprecated form)
c--
```

For FORTRAN, comment lines can begin with either an uppercase or a lowercase `c`. In-line documentation in C is analogous, except that comment lines begin with a `/*`. Note also that once a `/*` is entered, everything until the next `*/` is considered a comment; it is the programmer's responsibility to determine where to place the `*/`.

The entries in the comma-separated list of categories should be:

For executables:

General	Utility	Data Transfer	Visual Display
Calibration	uv Analysis	Map Making	Deconvolution
Plotting	Map Manipulation	Map Combination	Map Analysis
Profile Analysis	Model Fitting	Tools	Other

For subroutines:

Baselines	Calibration	Convolution	Coordinates
Display	Error-Handling	Files	Fits
Fourier-Transform	Gridding	Header-I/O	History
Image-Analysis	Image-I/O	Interpolation	Least-Squares
Log-File	Low-Level-I/O	Mathematics	Model
PGPLOT	Plotting	Polynomials	Region-of-Interest
SCILIB	Sorting	Strings	Terminal-I/O
Text-I/O	Transpose	TV	User-Input
User-Interaction	Utilities	uv-Data	uv-I/O
Zeeman	Other		

A Program Example

By way of illustration, below is the in-code documentation for *MIRIAD* task `varplot` which uses the "directives" noted previously.

```
c= varplot - Plot uv variables
c& lgm
c: uv analysis, plotting
c+
c       VARPLOT makes X,Y plots selected variables from a uv data set.
c       Only integer, real, and double precision variables maybe plotted.
c       When curser is in the plot window, the following keys are active:
c          X - expand window in X to give one column of plots
c          Y - expand window in Y to give one row of plots
c          Z - expand window in both X and Y to show only one plot
c          N - step to "next" plot in x or y or both depending on expansion
c@ vis
c       Miriad uv data-set. No default.
c@ device
c       PGPLOT plotting device. No default.
c@ xaxis
c       Name of variable to be plotted along x-axis. Default is ut time.
c@ yaxis
c       Name of variable to be plotted along y-axis. No default.
c@ multi
```

```
c       Make multiple plots or a single plot? Yes yields multiple plots,
c       No yields a single plot with multiple lines as needed. Default
c       is yes.
c@ compr
c       Compress number of x or y variables to be plotted by averaging
c       over spectral windows. Currently only SYSTEMP can be averaged.
c--
```

The task's name is "varplot", its one-line description is "Plot uv variables", the responsible programmer is "lgm", and the program is categorized as both a "uv analysis" program and a "plotting" program. It has a general description (the text following the `c+` line), and it has 6 keywords that the user may set: "vis", "device", "xaxis", "yaxis", "multi", and "compr".

A Subroutine Example

By way of illustration, below is the in-code documentation for *MIRIAD* subroutine axistype, which uses the "directives" noted previously.

```
c* Axistype - Find the axis label and plane value in user friendly units
c& mchw
c: plotting
c+
        subroutine AxisType(lIn,axis,plane,ctype,label,value,units)
c
        implicit none
        integer lIn,axis,plane
        character ctype*9,label*13,units*13
        double precision value
c
c Find the axis label and plane value in user friendly units.
c
c  Inputs:
c    lIn        The handle of the image.
c    axis       The image axis.
c    plane      The image plane along this axis.
c  Output:
c    ctype      The official ctype for the input axis.
c    label      A nice label for this axis.
c    value      The value at the plane along this axis.
c    units      User friendly units for this axis.
c--
```

Note that the programmer has woven executable code into the documentation (the lines that are not commented out): anything between the `c+` and the `c--` is considered to be part of the documentation, even though the lines are actually part of the subroutine code itself.

A subroutine source code file (or a program source code file) may contain multiple subroutines, each documented as above.

1.3 Code History

All source code files should contain comments (near the start of the file) describing the creation and modification history. This is quite important in the *MIRIAD* development environment, where programs are spread across many computers and programmers are separated by many miles.

Below are typical history comments (taken from the "key" routines, subroutine key.for):

```
c************************************************************************
```

```
c  The key routines provide keyword-oriented access to the command line.
c
c  History:
c    rjs    6jun87    Original version.
c    bs     7oct88    Converted it to use iargc and getarg. Added -f flag.
c    rjs    8sep89    Improved documentation.
c    nebk  10sep89    Added mkeyr.  I think rjs will not like it (Too right!).
c    rjs   19oct89    Major rewrite to handle @ files.
c    rjs   15nov89    Added keyf routine, and did the rework needed to support
c                     this. Added mkeyf. Modified mkeyr.
c    pjt   26mar90    Added mkeya. like mkeyr (again, bobs will not like this)
c    pjt   10apr90    some more verbose bug calls.
c    rjs   23apr90    Made pjt's last changes into standard FORTRAN (so the
c                     Cray will accept it).
c    pjt   10may90    Make it remember the programname in keyini (se key.h)
c                     for bug calls - introduced progname
c    rjs   22oct90    Check for buffer overflow in keyini.
c    pjt   21jan90    Added mkeyi, variable index is now idx, exp is expd
c----------------------------------------------------------------------
```

1.4 Task Version Identification

The first executable statement of a program should be to print out a version identification. The following is typical:

```
c************************************************************
       program Clean
          .
          .
          .
       character version*(*)
       parameter(version='Clean: version 1.0 26-jan-90')
          .
          .
          .
       call output(version)
```

This gives a version and the date when the task `CLEAN` was last modified. Additionally, this version identification should be included in any history generated by the task.

Typically, if the program produces a new dataset from existing ones, it is also recommended to pass this version along. Suppose a dataset is identified by the identifier `tin`, you will often find the following style of code at the end after the new dataset has been written:

```
        call hisopen(tin,'append')
        call hiswrite(tin,version)
        call hisinput(tin,'itemize')
        call hisclose(tin)
```

1.5 Task Version Identification RCS/CVS style

An alternative style can be used with the new `versan` subroutine:

```
c************************************************************
```

```
      program Itemize
         .
         .
         .
      character version*80, versan*80
         .
      version = versan('itemize',
     *  '$Id$')
         .
         .
```

Notice the subroutine `output` does not need to be called anymore. In addition, each time the source code is updated via RCS (the ATNF source repository) or CVS (the CARMA source repository), this string is automatically updated. During runtime the user will then see:

```
% itemize in=map0
itemize: Version 1.4, 2008/02/19 20:06:29 UTC
.
.
```

Chapter 2

MIRIAD Subroutine Library

2.1 Task Parameters

```
subroutine keyini
subroutine keyr(keyword,value,default)
subroutine keyd(keyword,value,default)
subroutine keyi(keyword,value,default)
subroutine keyl(keyword,value,default)
subroutine keya(keyword,value,default)
subroutine keyf(keyword,value,default)
subroutine mkeyr(keyword,values,nmax,n)
subroutine mkeyi(keyword,values,nmax,n)
subroutine mkeya(keyword,values,nmax,n)
subroutine mkeyf(keyword,values,nmax,n)
logical function keyprsnt(keyword)
subroutine keyfin
```

The `key` routines are used to get task parameters, which describe the processing that is to be performed. Typically the `key` routines will be called in the first few lines of the task, and never called again. All checking for the validity of the parameters should be carried out at this time.

`Keyini` initializes the `key` routines, and must be the first routine called. Similarly `keyfin` tidies up, and closes down. `Keyr, keyd, keyi, keyl` and `keya` return the value of a task parameter from the user. Their inputs are `keyword` (a character string) and `default`, the default value for the parameter. *The keyword must be in lower case.* The task parameter is returned in `value`. `Keyi`, `keyr`, `keyl`, `keyd` and `keya` are used for integer, real, logical, double precision and character values respectively. Only one value is returned at a time. The `keyf` routine is like `keya`, except that the string entered by the user is treated as a file name, and wildcard expansion is performed. The `key` routines can be called several times, giving the same keyword, and each successive call will get the next value associated with the keyword.

For example, if TRC is defined by the user as:

```
% TRC = 45,50
```

then the code:

```
call keyi('trc',n1,1)
```

```
call keyi('trc',n2,1)
```

will return the values 45 and 50 to `n1` and `n2` respectively.

These routines always return a value, even if it is only the default value. To determine if a parameter was actually set by the use, the `keyprsnt` routine can be called. This returns `.true.` if a value for the keyword still remains. An alternate way to test if a value is still present is to use a default which is clearly illegal (e.g. a blank string for a file name, or 0 for a pixel index).

The `mkey` routine return all values entered by the user for that particular keyword where all the values of the keyword are of the same data type. For these routines `values` is an array of `nmax` elements. The number of values returned (which may be zero) is given by `n`.

2.2 Error Handling

Many *MIRIAD* routines perform error checking internally, and bomb out if an error is detected. Other *MIRIAD* routines pass back a status value (generally the last subroutine argument). A status value of zero indicates success, -1 indicates end-of-file, and a positive value indicates some other error (what the positive values indicate is system dependent). Two routines can be called to indicate an error:

```
subroutine bug(severity,text)
subroutine bugno(severity,number)
```

Here `severity` is a single character, being either `'w', 'e'` or `'f'`, meaning warning, error and fatal respectively. When `bug` or `bugno` is called with a fatal error, it will not return. Rather it will cause the task to exit. For routine `bug`, `text` is a character string describing the error. For routine `bugno`, `number` is a status value, returned by a *MIRIAD* routine.

2.3 Text I/O

Though standard Fortran-77 routines would appear to be adequate for text i/o, there are invariably minor differences between systems, mainly related to carriage control. Additionally placing them in a module of routines forces the programmer to follow the 'handle' convention.

```
subroutine output(text)
subroutine txtopen(handle,name,status,iostat)
subroutine txtread(handle,text,length,iostat)
subroutine txtwrite(handle,text,length,iostat)
subroutine txtclose(handle)
```

`Output` prints `text` (a character string) on the users terminal.

`Txtopen` opens a text file (passing back a handle) with name `name`. `Status` can be either `'old'` or `'new'`. When opening a new file, any old files which exist with the same name may be deleted. `Txtread` and `txtwrite` read and write a character string, `text`, of `length` characters. `Length` is passed back from `txtread`, whereas it is passed into `txtwrite`. It may be zero in either case. All these routines return an i/o status variable, `iostat`.

`Txtclose` closes the file.

2.4 General Data Set Handling

MIRIAD makes few distinctions between what some systems (e.g. GIPSY, AIPS and FITS) call "data" and "header". Instead a data "file" (usually called a data set in this document) consists of a collection of items. Some items are small (a few bytes) whereas others are large (e.g. the collection of pixels in an image). All items are accessed by their name (a lower case string of up to 8 alphanumeric characters. Underscore and dollar characters are to be avoided). Convention dictates the names assigned to the various items within a data set. For example, the item name "image" is always used to store the pixel data of an image, "naxis" is the number of dimensions in an image, and "naxis1" is the number of pixels along the first axis of an image. New sorts of data items can be invented as the need arises.

What would be conventionally call header variables, are stored as small items. The naming convention is close to the FITS standard (though the names are lower case).

The following sections describe routines that in some way package together several calls to the lower level i/o routines. Their first argument is a "handle" returned by one of the open routines.

2.5 UV Data Sets

2.5.1 General

At the very least, a uv data set can be viewed as a sequence of correlation records, with associated u and v coordinates, time and baseline number. Associated with each correlation is a flag, indicating whether the correlation is believed to be good or bad.

The *MIRIAD* uv data structure required a more general structure. Unfortunately this is more complicated and somewhat cumbersome for simple cases. A uv data set can be viewed as an ordered (generally time ordered) stream of named records or "variables". There are markers in this data stream, to indicate when several variables change "simultaneously" (i.e. they correspond to the same time). Each variable consists of an array of values, the type of which can be either integer, real or double precision, etc. Correlation data, u and v coordinates, time and baseline numbers are specific examples of variables. Because of the special nature of these variables, special routines are available to simplify accessing them. A list of the variables that may be present in a uv data set is given in Appendix I.

In addition to this variable stream, a uv file will contain a file giving flagging information.

It should be noted that "variables" and "items" are quite distinct. For a particular data set, variables vary, or at least may vary, whereas data items are fixed. The notion of variables is unique to uv data sets, whereas all *MIRIAD* data sets are composed of data items. The stream of uv variables is implemented as three data items, called `visdata`, `vartable` and `flags`.

There is a "miriad" of uv routines. The routines used to access and manipulate a uv data set are given in the following table.

2.5.2 Open, Close and Rewind

The routine `uvopen` opens a uv data set and readies it for access. Here `dataname` is a string giving the data set's name. `Status` can be either `'old'` or `'new'`, depending whether an old data set is being read, or a new data set is being created. `Tno` is an integer handle passed back by the open routine (and is used for all future access to the data set). The routine `uvclose` closes the data set. The routine `uvrewind` positions a uv file at its beginning, and allows it to be read again.

2.5.3 Reading and Writing Visibilities

The routines `uvread` and `uvwrite` are routines used to read and write the correlation data (and the associated flagging information). Here `preamble` is an array of four double precision elements, `data` is an

```
subroutine uvopen(tno,dataname,status)
subroutine uvread(tno,preamble,data,flags,n,nread)
subroutine uvflgwr(tno,flags)
subroutine uvwrite(tno,preamble,data,flags,n)
subroutine uvclose(tno)
subroutine uvrewind(tno)

subroutine uvwread(tno,data,flags,n,nread)
subroutine uvwwrite(tno,data,flags,n)

subroutine uvgetvra(tno,varname,data)
subroutine uvgetvri(tno,varname,data,n)
subroutine uvgetvrr(tno,varname,data,n)
subroutine uvgetvrd(tno,varname,data,n)
subroutine uvgetvrc(tno,varname,data,n)

subroutine uvrdvra(tno,varname,data,default)
subroutine uvrdvri(tno,varname,data,default)
subroutine uvrdvrr(tno,varname,data,default)
subroutine uvrdvrd(tno,varname,data,default)
subroutine uvrdvrc(tno,varname,data,default)

subroutine uvprobvr(tno,varname,type,length,update)

subroutine uvputvra(tno,varname,data)
subroutine uvputvri(tno,varname,data,n)
subroutine uvputvrr(tno,varname,data,n)
subroutine uvputvrd(tno,varname,data,n)
subroutine uvputvrc(tno,varname,data,n)

subroutine uvtrack(tno,varname,switches)
integer function uvscan(tno,varname)
logical function uvupdate(tno)
subroutine uvmark(tno,onoff)
subroutine uvcopyvr(tin,tout)
subroutine uvnext(tno)

subroutine uvset(tno,object,type,n,p1,p2,p3)
subroutine uvselect(tno,object,p1,p2,flag)
subroutine uvinfo(tno,object,data)
```

Table 2.1: UV Data Subroutines

array of n complex elements, whereas flags is an array of n logical values. Preamble, data and flags are output from uvread, whereas they are input to uvwrite. The four elements of preamble are the u coordinate, v coordinate (measured in nanoseconds), time (Julian date) and baseline number. The baseline number, Bl, is calculated as:

$$Bl = 256A_1 + A_2$$

where A_1 and A_2 are the numbers of the first and second antennae respectively (antenna numbers vary from 1 to $N_{antenna}$). The array data is used to store the complex correlation data, whereas the logical values of the array flags indicate whether the corresponding correlation is deemed good or bad (true or false, respectively). For uvread, n limits the number of correlations that can be read; the actual number of correlations read is passed back as nread. Uvread can perform a number of additional processing steps – see the description of uvset (Section 2.5.11).

The flags of a data file can be modified using the uvflgwr subroutine. When called, the flags associated with the previous call to uvread and uvwrite are changed to those values given in the flags array. Using uvflgwr when reading a visibility file, is the method used to develop flagging tasks. Currently uvflgwr has the limitation that the linetype is either 'channel' or 'wide', and that the 'start' and 'width' linetype parameters are 1 (see Section 2.5.11). Also uvflgwr aborts if no flagging file exists

2.5.4 Reading and Writing Continuum Visibilities

MIRIAD uv datasets can contain both spectral and continuumn visibility data simultaneously. When both are present, the user/programmer will normally select which of these data to read, using the uvset routine (see Section 2.5.11). However this allows only one "linetype" to be read and written at a time. The uvwread and uvwwrite allow the programmer to read and write the continuum data independently of the "linetype". These routines completely bypass linetype processing. They should be used only when both a particular linetype, and the continuum data are required. Uvwread should be called after the call to uvread, whereas uvwwrite should be called before the call to uvwrite.

The routine uvwflgwr is the "wide" equivalent of uvflgwr. That is, by calling uvwflgwr, you can overwrite the flags associated with the previous call to uvwread and uvwwrite. Note that uvwflgwr aborts if no flagging file exists.

2.5.5 Direct Access to UV Variables

The main routines to access the variables are the uvgetvr and uvputvr routines. These read or write (respectively) an array of variables of given name (varname). The data read or written are in the array data (which can be a character string or an integer, real or double precision array, depending on the routine called). Exactly n values are read or written. Except for character strings, it is a fatal error, when reading, if n does not agree with the actual number of values for the variable). For a character string, the string is blank padded on a read (no n parameter is needed).

When reading an old data set, the uvgetvr routines should not be called before the first call to uvread. When creating a data set, the initial values for all variables should be written, by calls to the uvputvr routines, before the first call to the uvwrite routine (see Section 2.5.7 for some enlightenment on this issue).

Often it occurs that we are interested in a variable which has a single value, but we are not sure if the variable is present in the dataset. It would be possible to handle this with uvprobvr (Section 2.5.8) and uvgetvr. If the variable is not present, then we would want to use a default value. The uvrdvr routines package these three steps. It returns the value of the variable, data. If the variable is missing from the data stream, the default value, default, is returned. One disadvantage is that the uvrdvr routines only every return a single value (the first value in a multi-element variable).

The routine uvread functions by scanning through the variable streams, and returns with its results, when the "correlation data" ("corr" or "wcorr") is encountered. If you are not interested in reading the correlation data (i.e. if you do not intend calling uvread), then the uvscan routine can be used to scan through the variable stream until another variable is encountered. Actually uvscan may well read

somewhat past the desired variable, until it has read all variables which changed simultaneously with the desired one. `Uvscan` returns 0 is the variable was successfully found, -1 if an end of file was encountered, or 1 if the variable was not found. Note that it may not make a great deal of sense to intermix calls to `uvscan` and `uvread`.

2.5.6 Variable Override

Occassionally it is useful for the user to override the value of a uv variable. This is especially useful if the value of the variable is both wrong and important! When uvio opens an old file, for each variable name that is present in the uv file, it checks if there is a corresponding item with the same name. If there is, then the value of the item is used to override the value of the variable. Unfortunately the item must consist of a single number, and this single number will be copied into each value of the variable (if the variable consists of several elements).

2.5.7 UVNEXT

The uv i/o routines need to know what variables change simultaneously. all variables that have changes simultaneously. Conversely routine `uvwrite` assumes that all variables that change simultaneously with the variables that it writes, have already been written with routine `uvputvr`. Hence if the programmer is using `uvscan`, `uvread` or `uvwrite`, then simultaneity is not a concern, as long as variables are read after `uvscan/uvread`, and written before `uvwrite`. The routine `uvnext` provides better control, where this is needed. For an input file, a call to `uvnext` causes the next set of variables (which change simultaneously) to be read. For an output file, `uvnext` causes a marker to be written into the data, to indicate that variables written after the call did not change simultaneously with variables written before the call.

2.5.8 Determining UV Variables and Their Characteristics

Routine `uvprobvr` checks for the existence of a variable, and returns information about it. The string `varname`, the variable name, is input. The single character `type` is output, being either 'a', 'r', 'd', 'c', 'i', 'j' or ' ' (a blank), which indicates (respectively) that the variable is of type string (ascii), real, double precision, complex, integer, short integer or (in the case of the blank) that the variable is not present in the data-set. The output integer `length` gives the number of elements in the variable, and the output logical `update` indicates whether the variable has been updated 'recently' (see Section 2.5.10). Both `length` and `update` have no meaning if the variable is not present in the data-set. Before a variable is first read, `length` will be zero, and `update` will be `.false.`.

There is no special routine to return a complete list of the variables present in a uv data set, however this information is present in the item "`vartable`". This is a text file with each line consisting of two fields separated by a blank. The first is the "type" (either a, r, d, c, i or j) of the variable, the second is the variables name. The following section of FORTRAN lists the variables present in a uv data set.

```
character var*12,name*(?)
integer iostat,tno,item

call uvopen(tno,name,'old')
call haccess(tno,item,'vartable','read',iostat)
call hreada(item,var,iostat)
dowhile(iostat.eq.0)
  call output(var(3:10))
  call hreada(item,var,iostat)
enddo
call hdaccess(item,iostat)
call uvclose(tno)
```

Object	Type	N,P1,P2,P3
data	channel	nchan, start, width, step
	wide	nchan, start, width, step
	velocity	nchan, start, width, step
reference	channel	—, start, width, —
	wide	—, start, width, —
	velocity	—, start, width, —
coord	wavelength	—, —, —, —
	nanosec	—, —, —, —
planet	—	—, plmaj, plmin, plangle
selection	amplitude	n, —, —, —
	window	n, —, —, —

Table 2.2: Arguments to UVSET, for STATUS=OLD

2.5.9 Keeping Track of UV Variables

With many variables streaming past, there is a need to keep track on some particular variables. It would be rather inefficient and laborious to need to continually call `uvprobvr`, to check on particular variables. The `uvtrack` routine is used to instruct the uv routines to keep track of when certain variable changes its value, and to perform special processing on these variables at a later stage. Typically `uvtrack` would be called soon after `uvopen`, marking all the variables of particular interest. The special processing that the uv routines perform is dictated by the `switches` argument. This is a string, consisting of several characters, each character representing a particular processing step to be taken. Currently there are two switches – `u` and `c`. The `u` switch is used by `uvupdate`, whereas the `c` switch is used by `uvcopyvr`.

The routine `uvupdate` returns a `.true.` value if one of the variables, marked with the `u` switch, has been updated "recently" (see Section 2.5.10).

The routine `uvcopyvr` copies variables marked with the `c` switch, from the input dataset (given by `tin`) to the output dataset (given by `tout`) if they have changed "recently" (see Section 2.5.10). You need only mark the variables in the input dataset.

2.5.10 When Do UV Variables Change?

A uv variable can change its value in any part of the uv variable stream. So it can change its value after each call to `uvread`, `uvnext` or `uvscan`. The uv routines which need to know if a uv variable has changed (`uvprobvr`, `uvcopyvr` and `uvupdated`) normally (i.e. by default) work on whether the particular variable(s) of interest has changed since the last "mark" in the uv stream. By default any routine which causes more of the uv stream to be read (`uvread`, `uvscan` and `uvnext`) move this marker to the current point in the uv stream, before reading more. The `uvmark` routine provides greater control at marking the position in the stream. Calling

```
call uvmark(tno,.true.)
```

sets the marker at the current position in the uv file, and disables `uvread`, etc, from resetting the marker. Calling

```
call uvmark(tno,.false.)
```

also sets the marker at the current position, and enables `uvread`, etc, to reset the marker on each call.

2.5.11 Massaging Steps Performed by UVREAD – UVSET

As mentioned above, the `uvread` routine can perform, at the programmers request, extra processing steps on the visibility data. These steps consist of averaging and resampling frequency channels, uv coordinate

conversion and some corrections for planet observations. The steps are requested by calls to `uvset`. In the call to `uvset`, the argument `object` (a string) gives the general processing step that is being requested. The `type` argument (another string) gives more specific details, and the arguments `n` (integer) and `p1`, `p2` and `p3` (reals) give any numerical values needed.

Note that the set-up given by `uvset` only becomes correctly activated during the next call to `uvread`. Before this next call, the setup is in a somewhat nebulous state. So you should not expect various other routines associated with `uvread` to work as expected until after the next call to `uvread`. Associated routines include `uvflgwr` and `uvinfo`.

Table 2.2 summarizes the possible values of the arguments to `uvset`. Here the column titled "Object" and "Type" are the possible string values that `object` and `type` can take on. The third column gives the meaning for the parameters `n,p1,p2,p3`. Dashes in the third column indicate that the arguments value is ignored in this particular call. While several processing can be performed simultaneously (several calls to `uvset` will be needed to specify them all), others are mutually inconsistent. When mutually inconsistent steps are requested, the last requested step is honored. Each processing step requires further explanation.

`object='data'` This gives operations on the spectral data. Type `'channel'` selects the channels to be returned, and possible averaging together of the channel data. If the original channels are numbers from 1 to N, then, by using `type='channel'`, `uvread` will return *nchan* massaged channels, where channel i of the massaged channels is formed by averaging *width* channels of the original data, starting at channel $(i-1)\cdot step + start$. If `uvset` is called with *nchan* being zero, all channels are selected (note that this only makes sense if *start*, *step* and *width* are all 1).

`type='wide'` is similar, but uses the continuum data rather than the spectral data.

`type='velocity'` is also similar, returning a weighted sum of the spectral data. However in this case *start*, *width* and *step* are given in units of km/s (rather than channels). This is particularly useful if the spectrometer setup is not constant throughout the data or there is no Doppler tracking, and so the velocity of a given channel changes. Note that `'channel'`, `'wide'` and `'velocity'` are mutually exclusive. The default is `'channel'` (or `wide` if there is no spectral data in the file), with start,increment and width of 1.

If there are fewer than *nchan* channels, then dummy channels, which are flagged as bad, are added. If *nchan* is specified as 0, then `uvread` will return as many channels as possible.

`object='reference'` The "reference line" is a spectral channel, or an average of spectral channels, which the main data is divided by. Typically the reference line would be a strong point source (e.g. a maser). The resultant data is essentially normalized and shifted, but it also has atmospheric-based and instrument-base calibration problems removed. The extra parameters needed to describe the reference line is the same as for `object='data'`, except that the number of channels, and the increment is ignored (there is only ever one reference line). The default is not to have a reference line.

`object='coord'` This sets the units of the u and v coordinates returned in the preamble. Using `'wavelength'` or `'nanosec'` sets the units of the returned u and v. For `'wavelength'`, the sky frequency used is that of the first channel returned. The default value is `'nanosec'`.

`object='planet'` This causes the u and v coordinates to be scaled and rotated, and the correlation values to be scaled, to adjust for changes when observing planets. The parameters `plmaj, plmin` and `plangle` give the reference size (arcseconds) and position angle (degrees) of the planet. If the reference size is 0, then the size of the first selected data record is used.

`object='selection'` This gives extra control over the uv selection process (see 2.6). Currently there is only one possible type, `'amplitude'`, which enables or disables the amplitude selection process. If the argument `n` is positive, then amplitude selection is applied (i.e. the normal action), otherwise amplitude selection is not applied.

2.5.12 Setting Up UVWRITE – UVSET

`object='corr'` The uv routines allow the correlation data to be stored on disk, either as floating point numbers, or as 16 bit integers with an associated scale factor. The 16 bit format roughly halves

Object	Type	N,P1,P2,P3
corr	c	—, —, —, —
	j	—, —, —, —
data	channel	—, —, —, —
	wide	—, —, —, —

Table 2.3: Arguments to UVSET, for STATUS=NEW

Object	Units	P1,P2
time	Julian date	tmin,tmax
dra	radians	dramin, dramax
ddec	radians	ddecmin, ddecmax
ra	radians	ramin, ramax
dec	radians	decmin, decmax
uvrange	wavelengths	uvmin, uvmax
uvnrange	nanoseconds	uvmin, uvmax
pointing	arcseconds	pntmin, pntmax
visibility		vismin, vismax
increment		incr, –
on		state, –
polarization	FITS code	polval, –
amplitude		ampmin, ampmax
frequency	GHz	freqmin, freqmax
source		
window		win, –
antennae		ant1, ant2
or		–, –
and		–, –
clear		–, –

Table 2.4: Arguments to UVSELECT

the disk space required, but slows the read and write operation, and can cause precision problems. On the first call to `uvwrite`, the uv routines decide on the format to use, using a simple rule. The `uvset` call can be used to override this rule. To be of use, it must be called before the first call to `uvwrite`. The `type` argument is a single character, being `'r'` or `'j'`, which instructs floating point or scaled integers, respectively, to be used.

`object='data'` This conrtols whether `UVWRITE` writes the data to the `corr` or `wcorr` variable. The default is to write it to the `corr` item (i.e. it assumes that the data is spectral, rather than continuum data).

2.5.13 Selection Steps Performed by UVREAD – UVSELECT

Another function performed by `uvread` is skip or flag data that is not required. The routine `uvselect` is used to instruct `uvread` on which data are to be selected and rejected. Normally the programmer will not call `uvselect` directly, but will use the `SelInput` and `SelApply` routines (see Section 2.6). The Users Manual gives a description about the way the user normally interacts with these routines.

Generally `uvselect` will be called many times, each call giving a different selection or discard criteria. The routines `SelInput` and `SelApply` merely sequentially parse and pass the user-given criteria to `uvselect`. Hence the 'grammar' of the sequence of calls to `uvselect` (i.e. use of "or", or multiple occurrences of criteria based on the same parameter) is the same as the 'grammar' of the user-specified task parameter. The 'grammar' will not be repeated here.

The argument `object` is a string giving the parameter on which a select/discard criteria is based. The arguments `p1` and `p2` (both double precision) give added numerical parameters. Generally `p1,p2` give a range of parameter values to select or discard. Note that the units of the values are consistent with the

Object	Units	No. Values Returned
restfreq	GHz	nread
velocity	km/s	nread
bandwidth	GHz	nread
frequency	GHz	nread
sfreq	GHz	nread
visno		1
line		6
amprange	(flux units)	3

Table 2.5: Arguments to UVINFO

units of the underlying uv variables. These are not necessarily the most convenient units for the user, and so the user interface (given by SelInput and SelApply) performs some conversion between user-units and program-units.

There are a few additions objects, when compared with the `SelInput` and `SelApply` routines. These include the `ra` and `dec` objects, which give the pointing centre RA and DEC (after `dra` and `ddec` have been taken into account). Another is the `and` operator. `Uvselect` treates `and` and `or` as having identical precedence, and handles these operators in the order in which they are given. Beware of this lack of precedence.

The `clear` object causes the selection criteria to be reset to its default of selected everything.

The argument `flag` determines whether data which matches the associated criteria is to be selected (`flag=.true.`) or discarded (`flag=.false.`).

For example, to select data with Julian days 2444239.5 to 2444240.5 (i.e. data for 1 January, 1980), use:

```
call uvselect(tno,'time',2444239.5d0,2444240.5d0,.true.)
```

To select all data, except for 1 January, 1980, use:

```
call uvselect(tno,'time',2444239.5d0,2444240.5d0,.false.)
```

Note: In the `'antennae'` criteria, an antennae number of 0 is treated as a "match-all" number.

2.5.14 Getting Information After UVREAD

The routine `uvinfo` returns information about the data returned by the last call to `uvread`. The argument `object` is a character string indicating the information that is desired. The argument `data` is a double precision array, containing the returned information. Possible values for `object` are:

`'restfreq'` `Data` contains the rest frequency (GHz) for each channel returned by `uvread`.

`'velocity'` `Data` contains the velocity (km/s) for each channel returned by `uvread`.

`'frequency'` `Data` contains the frequency (GHz) of the channel returned by `uvread`, after removing the doppler contribution.

`'bandwidth'` `Data` contains the bandwidth (GHz) of each channel returned by `uvread`.

`'sfreq'` `Data` contains the sky frequency (GHz) of each channel returned by `uvread`.

`'visno'` `Data` contains a single number, which is the visibility number (running from 1 upwards) of the last channel read.

`'amprange'` `Data` contains three values. The first value indicate the sort of amplitude selection that was requested for this record, and the second and third values give a flux range. Possible value of data(1) are -1 (data outside the range [data(2),data(3)] were rejected), 0 (no amplitude selection was active) or +1 (data inside the range [data(2),data(3)] were rejected).

Object	Units
time	Julian day.
antennae	Baseline number = 256*ant1 + ant2. One of ant1 or ant2 can be zero.
uvrange	Wavelengths.
uvnrange	Nanoseconds.
visibility	Visibility number (1 relative).
dra	Radians.
ddec	Radians.
pointing	Arcseconds.
amplitude	Same as correlation data.
window	Window Number.

Table 2.6: Arguments to SelProbe

For example, consider the following code fragment.

```
integer maxchan
parameter(maxchan=512)
integer tno,nread
complex data(maxchan)
logical flags(maxchan)
double precision preamble(4),velocity(maxchan)
     .
     .
     .
call uvread(tno,preamble,data,flags,maxchan,nread)
call uvinfo(tno,'velocity',velocity)
```

After the call to `uvinfo` will contain the velocity of each channel read by `uvread`.

2.6 UV Selection – SelInput and SelApply

```
subroutine SelInput(key,sels,maxsels)
logical function SelProbe(sels,object,value)
subroutine SelApply(tno,sels,flag)
```

These routines are the usual programmer interface to the uv selection routines. They perform the parsing and checking of the user input, and the calling of the `uvselect` routine to actually implement the selection process. For more information see uv selection in the Users Manual, and the `uvselect` routine in this Programmers Manual.

`SelInput` calls the `keya` routine to get the user-specified selection criteria. This criteria is then broken into an intermediate form. The argument `key` is the keyword to be used. Generally it should be `'select'`. The real array `sels` (of size `maxsels` elements) is used to hold the intermediate form of the selection.

`SelApply` takes a selection criteria, in its intermediate form, and calls the `uvselect` routine to apply it. The argument `flag` determines whether criteria is actually to be used for selection (`flag=.true.`), or rejection (`flag=.false.`).

`SelProbe` returns information about whether uv data with a particular parameter value may have been selected. It does not guarantee that such data might exist in any particular data file. It also has the limitation that information is not present to convert "uvrange" and "uvnrange" calls into each other. These should be treated with caution. The `sels` array is the intermediate form returned by `SelInput`, and `value` is a double precision value, giving the parameter value that is of interest. The `object` argument determines the meaning (and the units) of this value. Possible values are given in Table 2.6.

Flag	Meaning
'r'	Get reference linetype specification (keyword 'ref').
's'	Get Stokes/polarisations (keyword 'stokes').
'd'	Perform input selection (keyword 'select').
'l'	Get data linetype specification (keyword 'line').
'p'	Apply planet rotation and scaling.
'w'	Return u and v in wavelengths.
'1'	Default number of channels is 1.
'c'	Apply selfcal gain solutions.
'x'	Data must be cross-correlation data.
'a'	Data must be auto-correlation data.
'b'	Input must be a single file.

Table 2.7: Flag Values for the uvDatInp Call

Note that this does not support all objects to `uvselect`. The name must be given in full (no abbreviations and case is significant).

2.7 The UVDAT routines

The `uvdat` routines are a layer of routines which sit on top of the `uvio` routines. They are used to read old uv data-sets. They perform a number of functions commonly used in handling uv data. The services include:

- Retrieve a number of standard task parameters (dealing with uv files) from the user. These include the `vis`, `line`, `select`, `stokes` and `ref` keywords. The uvDat routines support processing of several visibility files, and simplifies the book-keeping involved in tracking the several files.
- Apply antenna gain solutions to the data on the fly, if applicable.
- Perform polarization conversion steps, if required.

The `uvdat` routines still allow the programmer to use most of the `uvio` routines, to get the best of both worlds. Routines available are:

```
subroutine uvDatInp(key,flags)
logical function uvDatOpn(tno)
subroutine uvDatCls()
subroutine uvDatRd(preamble,data,flags,n,nread)
subroutine uvDatWRd(data,flags,n,nread)
subroutine uvDatGti(object,ival)
subroutine uvDatGtr(object,rval)
subroutine uvDatGta(object,aval)
subroutine uvDatSet(object,ival)
logical function uvDatPrb(object,dval)
```

The `uvDatInp` routine is called to setup the uvDat routines, and to retrieve the user parameters. The `key` argument (a character string) gives the name of the keyword to use to retrieve the input visibility data-set name. Normally it will be `'vis'`. The `flags` argument specifies what processing steps are to be performed, and which user parameters to retrieve. The `flags` argument is a character string, each character representing a processing step or parameter retrieval request. The legitimate characters are given in Table 2.7.

The `uvDatInp` subroutine should be called when retrieving task parameters (i.e. between calls to `keyini` and `keyfin`. It does not open any files.

The logical function uvDatOpn is responsible for opening the requested uv data-set, and performing most of the initialization steps (e.g. calling uvio routines to set the selection, linetype and planet processing options requested by its caller and by the user). uvDatOpn returns .true. if a visibility file was successfully opened. Otherwise it returns .false., indicating that there are no more files to process. uvDatOpn also returns the handle of the opened uv data-set, in the variable tno. The routine uvDatCls closes the opened uv data-set. When dealing with several files, the caller will go through the sequence: uvDatOpn, read data, uvDatCls, until uvDatOpn returns .false. (indicating no more files).

After opening, the uvDatRd routine can be used to read through the data. The arguments are the same as the uvread call, except that the file handle (tno) is not required. The routine uvDatWRd is equivalent to the uvwread routine – that is it reads the "wide" channels, ignoring the current linetype.

The uvDatGt routines are a set of inquiry routines, for the caller to determine what is going on inside the uvdat routines. Blurb-blurb.

The uvDatSet routine is used for the caller to instruct uvdat on what to do. Blurb-blurb

The uvDatPrb routine is used in exactly the same way (with the same restrictions) as the SelProbe routine – it is used to determine information about the selection criteria in force. Its arguments are the same as the SelProbe routine, except that the sels array is not needed (this array is stored internally in the uvdat routines.

2.8 Image Data Sets

These routines access image data sets.

```
subroutine xyopen(tno,dataname,status,naxis,nsize)
subroutine xyclose(tno)
subroutine xyread(tno,index,array)
subroutine xywrite(tno,index,array)
subroutine xysetpl(tno,naxis,nsize)
```

Here xyopen opens the image data set, and readies it for reading or writing. Dataname is the name of the data set, status is either 'old' or 'new', depending on whether an old data set is being opened to be read, or a new data set is being created. Naxis gives the dimension of the nsize array. Naxis is always an input parameter. Nsize gives the size, along each axis of the image. When opening an old data set, nsize is filled in by xyopen, and passed back to the caller. For a new data set, nsize must be set to the size of the desired image before the open routine is called. The argument tno is the handle passed back by the open routine, and is used in all subsequent calls to identify the data set.

Xyclose closes the data set.

Xyread and xywrite read or write a single row of the image. The row number is given by index, and the pixel data is stored in array (a real array). By default, xyread and xywrite access a row in the first image of a multi-image data set. The routine xysetpl is used to change this default to another image. In this naxis gives the dimension of the nsize array, and nsize is an integer array giving the indices along the third, fourth, etc, dimension of access. For example, to access the n'th image in a cube, use:

```
call xysetpl(tno,1,n)
```

2.9 Image Coordinate System

MIRIAD defines and stores image coordinate system information in a similar fashion to AIPS and FITS. Most cubes will have coordinates along its three axis of RA, DEC and velocity. The item ctype

Ctype	Crval	Crpix	Cdelt	Equation
RA---xxx	α_0	i_0	$\Delta\alpha$	$\alpha = \alpha_0 + \Delta\alpha/\cos(\delta_0)(i - i_0)$
DEC--xxx	δ_0	i_0	$\Delta\delta$	$\delta = \delta_0 + \Delta\delta(i - i_0)$
VELO-xxx	v_0	i_0	Δv	$v = v_0 + \Delta v(i - i_0)$
Others	x_0	i_0	Δx	$x = x_0 + \Delta x(i - i_0)$

Table 2.8: Image Coordinate System

gives the type of coordinate along a particular axis, whereas `crval, crpix` and `cdelt` give the value of the coordinate at the reference pixel, the value of the reference pixel, and the increment between pixels, respectively. Unlike AIPS and FITS, RA and DEC are given in radians, and velocity is given in km/sec. RA and DEC will generally need to be converted to hours,minutes,seconds, or degrees,minutes,seconds, before being presented to the user.

Table 2.8 gives quite approximate formulae for converting from pixel number to an astronomical coordinate. For more accurate formulae, see AIPS Memo No. 27, "Non-linear Coordinate Systems in AIPS" (Eric Greisen).

For example, the following code fragment calculates RA, DEC and velocity.

This code fragment checks that the axis are in the order RA, DEC then velocity (i.e. the normal ordering) and aborts if they are not. Smarter code would allow them in any order, and would probably treat any unrecognized ctype as a linear coordinate system. It uses default values (if values of crval, crpix and cdelt are missing) of 0, 1 and 1. Probably better default values could be chosen, though if ctype is an item, then we can be fairly certain that the other parameters will also be present.

2.10 Region of Interest and Pixel Blanking

It is a common situation for the user to want to process a limited subset of an image. It is also common for an image to contain pixels which are "blanked" or have an "undefined value". *MIRIAD* routines are available to treat these two pixel selection operations together. The input to these routines are the task parameters which describe the subregion the user is interested in, and a masking item that may be associated with an image. The output is a description of the pixels selected.

Each image dataset may have a masking item. This is a bitmap containing a flag for each pixel, indicating whether the pixel is good or bad. For bad pixels, the pixel value actually stored in the image is not defined, though it will be a legal or typical value. Zero, or the value of the pixel before blanking, is a good choice.

2.10.1 Regular Regions of Interest

Generally a minimum of three routines are needed to process even "regular" regions of interest. By "regular" we mean that the region of interest is describable by a bottom left corner and top right corner (i.e. it is a filled grid). The routines of interest are:

```
subroutine BoxInput(key,dataset,boxes,maxboxes)
subroutine BoxSet(boxes,naxis,nsize,flags)
subroutine BoxInfo(boxes,naxis,blc,trc)
```

All routines require an integer array, `boxes`, which is used to accumulate a description of the region of interest. Its size is given by argument `maxboxes`. The required size is a function of the complexity of the region of interest, etc. Typically 1024 elements should be adequate.

`BoxInput` reads the task parameters that the user gives to specify the region of interest. The way this is specified is moderately general and (consequently) complex. See the description in the users guide. Included is the ability to give the region specification in a variety of units. To convert between some units and absolute pixels, `BoxInput` needs information about the coordinate system being used (e.g.

```
 character ctype1*8,ctype2*8,ctype3*8
 double precision crval1,crpix1,cdelt1,crval2,crpix2,cdelt2
 double precision crval3,crpix3,cdelt3
 double precision alpha,delta,v
 integer i,j,k
     .
     .
 (assume i,j,k contains the grid coordinate of interest)
     .
 call rdhda(tno,'ctype1',ctype1,' ')
 if(ctype1(1:5).ne.'RA---')
*    call bug('f','First axis is not RA')
 call rdhda(tno,'ctype2',ctype2,' ')
 if(ctype2(1:5).ne.'DEC--')
*    call bug('f','Second axis is not DEC')
 call rdhda(tno,'ctype3',ctype3,' ')
 if(ctype1(1:5).ne.'VELO-')
*    call bug('f','Third axis is not velocity')
 call rdhdd('crval1',crval1,0.d0)
 call rdhdd('crval2',crval2,0.d0)
 call rdhdd('crval3',crval3,0.d0)
 call rdhdd('crpix1',crpix1,1.d0)
 call rdhdd('crpix2',crpix2,1.d0)
 call rdhdd('crpix3',crpix3,1.d0)
 call rdhdd('cdelt1',cdelt1,1.d0)
 call rdhdd('cdelt2',cdelt3,1.d0)
 call rdhdd('cdelt3',cdelt3,1.d0)
 alpha = crval1 + cdelt1/cos(crval2)*(i-crpix1)
 delta = crval2 + cdelt2            *(j-crpix2)
 v     = crval3 + cdelt3            *(k-crpix3)
```

Table 2.9: Coordinate System Code Example

parameters `crval`, `crpix` and `cdelt`). `BoxInput` determines these from the *MIRIAD* data-set given by `dataset`. Normally this is the name of the image dataset which we are interested in. If `dataset` is blank, `BoxInput` still functions correctly, but cannot perform unit conversion.

`BoxInput` breaks up the specification into an intermediate form, and stores it in the `boxes` array. The keyword associated with the task parameter is `key`, which would normally be `'region'`. If the `key` argument is blank, `BoxInput` does not attempt to get a region-of-interest specification, but instead just initialises the `boxes` array with the default region-of-interest.

The programmer passes to `BoxSet` information about the size of the image of interest. This is given in the integer array `nsize`, which consists if `naxis` elements (as with the corresponding arguments to `xyopen`). The `flags` argument is an input character string, giving information about the default region of interest. It can consist of:

- q The default region of interest is the inner quarter of the image.
- 1 The default region of interest consists of the first plane only.
- s The region of interest must be "regular". If it is not, `BoxSet` generates a warning message, and will use the bounding box of the selected region.

`BoxInfo` returns integer arrays `blc` and `trc`, which give the bottom left corner and top right corner of the region which encloses the overall region of interest. Both these arrays are of size `naxis` integers.

2.10.2 Arbitrary Regions of Interest and Blanking Information

In addition to the above routines, there are three routines to allow treatment of more complex regions of interest. These are:

```
      subroutine BoxMask(tno,boxes,maxboxes)
      logical function BoxRect(boxes)
      subroutine BoxRuns(naxis,nsize,flags,boxes,
     *        runs,maxruns,nruns,xblx,xtrc,yblc,ytrc)
```

`BoxMask` requests that an image mask be ANDed with the region requested by the user. The input to the routine is the image dataset handle, `tno`, whereas the `boxes` array is an input/output integer array used to accumulate region of interest information. `BoxMask` can be called multiple times, each time ANDing in a new image mask. This is an optional routine. Typically it would be called for each of the input images, so that "blanked" pixels in the input images would be excluded from the region of interest.

`BoxRect` returns `.true.` if the region of interest is rectangular (i.e. whether the region of interest is entirely described by `blc` and `trc`.

`BoxRuns` returns the region selected for a particular plane. The input arguments `naxis` and `nsize` are analogous to the same arguments in the `xysetpl` routine. `runs` is an integer array of size $3 \times maxruns$. On output it indicates which pixels in the plane are to be process. `Runs` consists of `nruns` entries of the form:

```
j,imin,imax
```

This indicates that pixels (imin,j) to (imax,j) are to be processed. All entries are non-overlapping. There may be zero or many entries for a particular value of j. The table is in increasing order of j and imin. On output, the integers `xblc, xtrc, yblc` and `ytrc` give the bottom left and top right corners, in x and y, of the smallest subimage which contains the selected pixels. The `flags` argument, an input character string, indicates some extra options. These are:

- r Make the coordinates, returned in the `runs` table, relative to `(xblc,yblc)`.

2.10.3 Reading and Writing Blanking Information

Though the box routines are the preferred way to read blanking information, it is possible to read the blanking information associated with an image directly. Also a routine is needed to write blanking information. Blanking information can be read and written by two sets of routines. These routines are:

```
subroutine xyflgrd(tno,index,mask)
subroutine xyflgwr(tno,index,mask)
subroutine xymkrd(tno,index,runs,n,nread)
subroutine xymkwr(tno,index,runs,n)
```

Here tno is the image handle returned by xyopen, index gives the row number (analogous to xyread and xywrite). Analogous to the working of xyread and xywrite, the plane of the masking file that the routines access is set with the xysetpl routine.

The xyflgrd and xyflgwr routines read and write the logical array mask. The mask array has a .true. value if the corresponding pixel is good, or .false. if it is bad (or blanked).

The xymkrd and xymkwr routines read and write a "runs" array. The runs array is a table of the form:

```
imin,imax
```

where pixels imin to imax are good (not blanked). Note that the size of runs is n integers of n/2 pairs of ordinates. Runs is input to xywrite and output from xyread. For xyread, nread returns the number of elements of runs that have been filled in.

Because many images are completely good (i.e. no blanked pixels), it would be superfluous to always carry around blanking information. Hence the mask containing the blanking information need not exist. If it does not exist, the read routines described above will return indicating that all pixels are good. A programmer can check if the mask exists, using the logical function hdprsnt:

```
logical exists
logical hdprsnt
     .
     .
     .
exists = hdprsnt(tno,'mask')
```

2.11 Scratch Files

```
subroutine scropen(tno)
subroutine scrread(tno,buffer,offset,length)
subroutine scrwrite(tno,buffer,offset,length)
subroutine scrclose(tno)
```

The scratch i/o routines are used to create and read and write from a scratch file containing real-valued data. Here tno is a handle passed back by the open routine, and must be used in all subsequent calls to the scratch i/o routines. Scrread and scrwrite read and write scratch data to/from the real array buffer. Length values are accessed, starting at the offset given by offset (for a scratch file of N real values, offset can vary from 0 to $N-1$).

2.12 General Item Routines

These routines read and write "small" items (items consisting of a single value), or perform some operation on an entire item.

In many processing systems these routines would be called "header handling" routines. This has led to the rather misleading use of the letters "hd" as a component of each of the following routine names.

```
subroutine rdhda(tno,itemname,value,default)
subroutine rdhri(tno,itemname,value,default)
subroutine rdhdr(tno,itemname,value,default)
subroutine rdhdd(tno,itemname,value,default)
subroutine rdhdc(tno,itemname,value,default)
subroutine wrhda(tno,itemname,value)
subroutine wrhdi(tno,itemname,value)
subroutine wrhdr(tno,itemname,value)
subroutine wrhdd(tno,itemname,value)
subroutine wrhdc(tno,itemname,value)
logical function hdprsnt(tno,itemname)
subroutine hdcopy(tin,tout,itemname)
subroutine hdprobe(tno,itemname,descr,type,n)
```

Before these can be accessed, the data set must be opened with either `xyopen`, `uvopen` or `hopen`. `Tno` is the handle returned by these open routines. The name of the item to access is given by `itemname` (a string). These names should be up to 8 lowercase alphanumeric characters, and (where possible) they should conform to the FITS standard. The `rdhd` routines read an item, returning its value in `value`. The routines `rdhda, rdhdi, rdhdr, rdhdd` and `rdhdc` return a string, integer, real, double precision or complex value, respectively. If the item is not found, the `rdhd` routines return the default value given by `default` (note `value` and `default` should be a string, integer, real, double precision or complex values, depending on the routine called). Similarly the `wrhd` routines save the value of an item.

The logical function `hdprsnt` can be called to determine if a particular item exists.

The routine `hdcopy` copies an item from one data set to another. The item can be arbitrarily large. `Hdcopy` copies from the data set, whose handle is given by `tin`, to the data set whose handle is `tout`.

The routine `hdprobe` is used to probe the characteristics of an item. The string `descr` returns a brief description of the item (intended for people, not programs, to read). If the item consists of a single value (or string), the description gives the value. Otherwise the description gives the type and size of the item (in human readable form). The string `type` is one of `'nonexistent', 'integer*2', 'integer', 'real', 'double', 'complex', 'character', 'text'` or `'binary'` (unknown type). The integer `n` gives the number of elements in the item. If the item does not exist, `type` returns `'nonexistent'` and `n` returns zero.

2.13 History Item

Most data sets will have an associated history item. This is a text file containing a description of the data set, and the processing that has been performed on it. The routines to access the history item are as follows:

```
subroutine hisopen(tno,status)
subroutine hisread(tno,line,eof)
subroutine hiswrite(tno,line)
subroutine hisinput(tno,taskname)
```

```
subroutine hisclose(tno)
```

`Hisopen` must be called before the history item can be accessed. Here `tno` is the handle passed back by a previous call to `xyopen` or `uvopen`, and `status` is a string which can be either `'read'`, `'write'` or `'append'`. `Hisread` and `hiswrite` can be called to read from, or append to the history item a line at a time (`line` is a character string). When reading, the logical value `eof` turns true when the end of the history item is encountered. `Hisinput` writes a number of history comments, giving the command line input parameters. This is a particularly useful routine to summarize user inputs. For `hisinput`, `taskname` is the name of the task. `Hisclose` closes the history item.

Note that there is no routine to copy a history file from one dataset to another. This is easily performed using the `hdcopy` routine described in the previous section.

For uniformities sake, history comments follow a standard format. The following is an example of the history comments written by the task DEMOS:

```
DEMOS: Miriad DeMos: version 1.0 30-apr-90'
DEMOS: Input parameters are
DEMOS:   vis=uvn
DEMOS:   map=mosclean
DEMOS: Pointing offset used (arcsec):   24.0 -30.0
```

Note that all comments start with the task name. The first comment gives a version date of the program. The next three lines were generated by `HisInput`, and the last line was an extra comment.

2.14 Low Level I/O Routines

All i/o routines described above are built on top of a set of low level routines. In *MIRIAD*, a "data set" is a collection of named items. Each item is an unstructured, byte addressable collection of data. It is up to the higher level software to impose some structure to the data items, and to provide a better interface for the high level programmer. The higher level routines should be adequate for most programmers. Though direct use of the following low level routines is discouraged, there are some instances where they may be needed.

```
subroutine hopen(tno,dataname,status,iostat)
subroutine hclose(tno,iostat)
subroutine haccess(tno,item,itemname,status,iostat)
subroutine hdaccess(item,iostat)
subroutine hdelete(tno,itemname,iostat)
integer function hsize(item)

subroutine hreada(item,buffer,iostat)
subroutine hreadb(item,buffer,offset,length,iostat)
subroutine hreadj(item,buffer,offset,length,iostat)
subroutine hreadi(item,buffer,offset,length,iostat)
subroutine hreadr(item,buffer,offset,length,iostat)
subroutine hreadd(item,buffer,offset,length,iostat)

subroutine hwritea(item,buffer,iostat)
subroutine hwriteb(item,buffer,offset,length,iostat)
subroutine hwritej(item,buffer,offset,length,iostat)
subroutine hwritei(item,buffer,offset,length,iostat)
subroutine hwriter(item,buffer,offset,length,iostat)
subroutine hwrited(item,buffer,offset,length,iostat)
```

Only two operations can be performed on a data set as a whole, namely to open and to close it. The routines to perform these are `hopen` and `hclose`. Here `tno` is a handle passed back by `hopen`, `dataname` is a string giving the name of the data set, and `status` is either `'old'` (when accessing an old data set) or `'new'` (when creating a data set). `Iostat` is a error indicator, being zero if the operation was successful. Because of buffering performed by the i/o routines, it is very important to close a data set when it is no longer needed.

Before any item can be read or written, it must be "opened" with the `haccess` routine. The inputs to this are: `tno` (the handle passed back by `hopen`); `itemname`, a string giving the item name; `status`, a string which can be `'read'`, `'write'` `'append'` or `'scratch'` (if an item is opened with `status='scratch'`, an item is created, but then destroyed when the item is closed). The outputs from `haccess` are firstly a handle, `item`, which is used to perform i/o on the item, and secondly a status return, `iostat`. All items should be closed down, with `hdaccess`, before the data set as a whole is closed.

The `hdelete` subroutine deletes an item. The integer function `hsize` returns the size of an item in bytes.

There are a group of i/o routines provided to read and write items. Because items are stored in a machine independent format, there are separate i/o routines for each data type. Each i/o routine performs the conversion between the external (disk) format and the hosts internal format. For example, the `hreadr` routine reads real numbers. On disk, *MIRIAD* reals are stored as IEEE floating point numbers. Internally, however, reals are stored in the hosts machines "real number format". The `hreadr` routine performs the conversion. The read/write routines know nothing about the type of the data it is accessing. The caller must know this, and call the appropriate read/write routine.

All read/write routines take the handle `item` (passed back by `haccess`) as their first argument, and pass back an i/o status as the last argument. The second argument is a buffer, which can be either a character string (`hreada, hwritea, hreadb, hwriteb`), an integer array (`hreadj, hwritej, hreadi, hwritei`), real (`hreadr, hwriter`) or double precision (`hreadd, hwrited`). `Hreada` and `hwritea` read/write a text file, performing i/o on a line at a time. Text files are stored using a line-feed character to delimit the end of a line (i.e. the normal UNIX convention, or the VMS Stream_LF convention). The routines `hreadb` and `hwriteb` perform i/o on bytes, no conversion being performed – these routines should rarely be needed). The routines `hreadj` and `hwritej` assume the integers are externally stored as 16 bit quantities, whereas `hreadi` and `hwritei` assume the integers are externally stored as 32 bit quantities. Internally, integers are always stored in the hosts standard integer format (`int` in C, and an `INTEGER` in FORTRAN. Real and double precision values are externally stored in IEEE 32 bit and 64 bit floating point format. All read/write routines (except `hreada` and `hwritea`) take (as inputs) a byte `offset` and byte `length` as their third and fourth arguments. Both `offset` and `length` must be a multiple of the size of the data type being accessed (e.g. they must be a multiple of 4 for reals, or 8 for double precision numbers). Apart from this alignment restriction, data items can be read in a random fashion.

All I/O routines pass back an i/o status indicator. A value of zero indicates success, -1 indicates end-of-file, and any other values indicate some other error (that is system dependent).

2.15 Numeric Routines

The following numeric routines should be used in preference to others that do the same function, as these are reasonably optimized. Generally these are tailored to the target machine (e.g. on the Cray these are based on Cray SCILIB routines, on the Convex, they are partially based on VECLIB routines).

2.15.1 FFT Routines

There are three FFT routines, namely:

```
subroutine fftrc(in,out,sign,n)
subroutine fftcr(in,out,sign,n)
subroutine fftcc(in,out,sign,n)
```

These perform one-dimensional FFTs. In all cases, `sign` is the sign of the exponent in the transform (i.e. a `sign` of -1 is conventionally viewed as a forward transform), and `n` is a power of 2 giving the length of the (full) sequence. $\frac{1}{N}$ scaling is never performed (it is up to you to scale at the best time). `In` is the input array, and `out` is the output array. These routines evaluate:

$$Out(l) = \sum_{k=1}^{N} In(k) \exp(\pm j \frac{2\pi}{N}(k-1)(l-1))$$

where k and l vary from 1 to N. `Fftrc` transforms a real sequence (i.e. `in` is a real array) and outputs only the first $N/2+1$ complex values. No information is lost because of the conjugate symmetry of FFTs of real sequences. Conversely `fftcr` takes a complex sequence of length $N/2+1$ and produces a real sequence of length N. Finally `fftcc` performs a complex to complex transform, with both input and output being of length N.

2.15.2 Min and Max Value Routines

There are two routines to find the index of the minimum and maximum values of an array, namely:

```
integer function IsMin(n,data,step)
integer function IsMax(n,data,step)
```

2.15.3 WHEN and ISRCH Routines

```
subroutine whenxxx(n,array,step,target,index,nindex)
integer function isrchxxx(n,array,step,target)
```

The `when` routines return the indices of all locations in an array that have a "true relational value" to the target. See page 4-64 of the Cray "Library Reference Manual" for more information. The `isrch` routines return the first location in an array that has a "true relational value" to the target. See page 4-59 of the Cray "Library Reference Manual". For both routines, "`xxx`" can be one of `ieq, ine, ilt, ile, igt, ige, feq, fne, flt, fle, fgt` or `fge`. These give the type of `array` (Integer or Floating), and the relation (equal, not equal, less than, etc).

Because these are well optimized routines, these routines should be used in preference to straightforward FORTRAN code that performs an equivalent function.

2.15.4 Blas and Linpack Routines

The Blas routines are "Basic Linear Algebra Subprograms", while the Linpack routines are a suite of linear equation solving routines. It is common to find these routines, in an optimized form, on vector computers. Additionally they are public domain routines in standard FORTRAN, making them easy to port. The *MIRIAD* library (or a system library) has these routines in their single precision real and complex forms. There are a number of good references on Blas and Linpack routines available (e.g. 4-1 and 4-38 in the Cray "Library Reference Manual", or the Convex "VECLIB Users Manual").

Chapter 3

Utility Programs

A few programs have been developed to simplify the development of good portable code.

3.1 FLINT

Flint is a FORTRAN program checker. It aims at uncovering programming bugs and bad programming practises. Given FORTRAN source as input, Flint produces warnings on the terminal and in a listing file (flint.log). Flint checks for variables which are unused, uninitialized or undeclared. It also checks that the number, type and intent of arguments passed to a subroutine remains consistent amongst all calls (an arguments intent is whether its value is input to, or output from a routine).

```
flint [-acdfhjkrsux2?] [-I incdir] [-o file] [-l] file ...
```

There are many command line switches (though the defaults are usually adequate). Invoking Flint with the "?" switch causes it to print out brief information about Flint, and all the command line switches. For example:

```
flint -?
```

will give some help information.

3.1.1 Making Flint Quieter

Flint assumes that the programmer follows a fairly strict programming convention. If the programmer does not, then Flint can produce a voluminous number of warning messages. There are a number of switches which help quieten Flint to manageable proportions.

–c Allow interwoven continuation and comment lines. Normally Flint generates warnings about this (even though it is standard FORTRAN) An unpleasant side effect of allowing interwoven comment and continuation lines is that comments are not copied to the listing file.

–d Do not warn when variables are not explicitly declared. Normally Flint insists that all variables must be explicitly declared with a 'REAL', 'INTEGER', etc, statement.

–f Do not generate warnings about lines greater than 72 characters long or with an odd number of quotes. When checking if a line is greater than 72 characters, Flint interprets tabs as if the equivalent number of spaces had been typed (the VMS compiler treats a tab as a single character, rather than as multiple spaces).

−h By default, Flint does not understand hollerith data. The `-h` flag instructs Flint to recognize hollerith data as integers.

−j Do not check if a variable has been initialized before being used. If a variable is not in a DATA, COMMON, PARAMETER or SAVE statement, and if it is not a dummy subroutine argument, then Flint normally attempts to check that the variable is initialized before it is used. It does this by checking that it is assigned to at a earlier line in the routine than where it is first used. This is not necessarily correct in routines with EQUIVALENCE statements, a maze of gotos, or which have conditional blocks within loops. Applying this same rule to subroutine arguments provides Flint with a technique for determining subroutine argument intent. Disabling initialization checking will also disable these intent-determining rules, and so should eliminate spurious warnings about inconsistent subroutine argument intent.

−k Suppress warnings about common blocks with possible alignment problems.

−r Do not warn about seemingly redundant variables. Redundant variables are those that are assigned to but never otherwise used. A common source of most messages is when a value returned by a subroutine is never used. For example, Flint will complain if you ignore values returned in an array used by a subroutine for scratch storage, or if values returned by a general subroutine are ignored in specific cases.

−s Load definitions of FORTRAN-IV and specific intrinsic functions. By default, Flint only knows about the standard FORTRAN-77 generic intrinsic functions (e.g. COS, REAL, MAX). It does not know specific function names or obsolete FORTRAN-IV functions (e.g. DCOS, FLOAT, AMAX0). The 's' flag causes Flint to load the definitions of these as well.

−u Do not generate a warning about variables which do not seem to be used. By default Flint produces such a warning for local variables (but not COMMON or PARAMETER variables).

−x Allow extended subroutine and variable names. By default Flint warns if subroutine or variable names are longer than 8 characters, or if they contain underline or dollar characters. Using this flag allows names of arbitrary length, with dollar and underline characters.

3.1.2 Other Flags and Arguments

−a This causes flint to include a crude cross-reference map in its log file.

−I This flag is used to give the name of a directory to search for include files. When it encounters an 'include' statement, flint first checks the local directory for the include file, and then checks other directories given by the `-I` switch, in the order they were present on the command line. This is also equivalent to `-i`.

−? This causes some help information to be printed.

3.1.3 Bugs and Shortcomings

Flint has a number of bugs and shortcomings, in addition to those described above. Flint is intended to be run on a FORTRAN that a compiler would accept. During the early stages of program development, a good FORTRAN compiler is a far better tool to find programming problems. Flint bugs and shortcomings include:

- IMPLICIT statements are ignored.
- EQUIVALENCE statements are rcudely treated.
- A number of archaic, poor or non-standard FORTRAN features are not recognized. These include the ASSIGN statement and assigned GOTO, alternate return statements, PAUSE, ENTRY, DECODE, ENCODE, many archaic forms of i/o statements, NAMELIST and many VMS extensions.

- While BYTE, INTEGER*2, INTEGER*4, REAL*4, REAL*8, etc statements are recognized, a warning message is generated, Flint otherwise treats these variables as either standard INTEGERs or REALs.
- Columns 72 to 80 of an input line (if they are present) are assumed to be part of the FORTRAN statement, rather than a comment field. Flint generates a warning if a line contains more than 72 characters.

Flint recognizes the following extensions, and digests them without complaint:

- DO/ENDDO and DOWHILE/ENDDO constructs.
- INCLUDE statements.
- Lower case characters are considered equivalent to upper case. Tabs are treated as if they were the equivalent number of spaces. The full printable ascii character set can appear in character strings.
- A comment line starting with the hash character (#).
- The INTENT statement (see the section on Interface Definition Libraries).

3.1.4 Determining the Intent of Subroutine Arguments

Whenever Flint finds a call to a subroutine, or the source of a subroutine, it attempts to determine whether a subroutine argument is either input or output (this is called an arguments intent). Usually Flint will build up its knowledge of a particular routines arguments by analyzing many uses of the subroutine, and generate a warning whenever inconsistent use is noted.

Note that Flint expects an argument's intent to be either input or output or input/output. Some routines use an argument as input in some situations and output in others. Such routines may confuse Flint, and it may well generate spurious messages about the variable in the subroutine call being uninitialized or redundant, or that the arguments intent is inconsistent.

Normally Flint performs a single pass, analyzing files in the order that they are given on the command line. When analyzing the early part of the input source, Flints knowledge of argument intent is very poor. Thus some problems (uninitialized and redundant variables) may be missed. There are two recipes to partially avoid this. Firstly files on the command line should be ordered with the files containing low level subroutines first, and high level programs and application code last (generally interface definition libraries should go first). Additionally inside files, the lower level routines should be first, and higher level routines last. Secondly, the user can ask Flint to perform two passes of the input files, by using the `-2` flag. Two passes will not necessarily correctly determine the intent of all arguments (in general N passes will always be enough for program with subroutine calls to a depth of N).

The following contrived example shows the worst case sort of behavior, where Flint builts up its knowledge of argument intent quite slowly.

```
subroutine a(x)
real x
call b(x)
end
subroutine b(x)
real x
call c(x)
end
subroutine c(x)
real x
write(*,*)x
end
```

It would require three passes for Flint to determine that the intent of argument `x` in subroutine `a` was input (the first pass would determine the intent of `x` in `c`, the second would determine the intent of `x` in `b`). Usually Flint determines argument intents by other means than just allowing information to bubble up from the lowest level to the higher level routines. But in the above example, none of these could be applied. One of the techniques is derived from initialization checking (described under the `-i` flag). When determining intent accurately is important, and if the initialization checking algorithm is failing (i.e. producing spurious messages), then initialization checking should be disabled.

Small changes can significantly change the number of passes needed to uncover all intent information. For instance, in the example above, if the ordering of the routines was reversed (i.e. `c` first and `a` last), or if a piece of code such as

```
call a(1.0)
```

appeared before subroutine `a`, then Flint would require only one pass.

3.1.5 Interface Definition Libraries

It is a common (normal) situation in program development to have a library of standard functions and routines. Though Flint knows the interface definitions (i.e. number, type and intent of each argument) of standard FORTRAN functions, it is desirable for users to build and use a file (or files) describing the interfaces to their own library. This section describes the features in Flint to achieve this.

Flint can output a text file, in a pseudo-FORTRAN, which contains interface descriptions of all the routines that it has encountered in this run. The format is:

```
flint -o file ....
```

Each routine in the output file will probably contain many '`INTENT`' statements (part of the FORTRAN-8X standard). INTENT is used to indicate whether a subroutine argument is input to or output from a routine (or both), or whether this is unknown. Typically a routine description would look like:

```
subroutine example(arg1,arg2,arg3,arg4)
real arg1,arg2
integer arg3,arg4
intent(in) arg1,arg3
intent(out) arg3,arg4
intent(unknown) arg2
end
```

Here Flint believes `arg1` is input, `arg4` is output, and `arg3` is both input and output. Flint could not determine whether `arg2` was input or output.

The user can edit an interface file created by Flint, or create an entire interface description by hand.

Such an interface description file appears as normal FORTRAN to Flint, so it can be input to Flint (to teach Flint about a users standard routine interfaces) on subsequent uses of Flint. While these interface definitions could be input along with normal source files, preceding the interface file with the `-l` (library) switch has the advantage that the source of this file does not appear in the listing file. Warning messages generated by analyzing this file are also suppressed.

The following example generates a interface description file, `library.dat`, from source files `routine1.for`, `routine2.for` and `routine3.for`. The interface definition is then used in a subsequent Flint run on `program.for`.

```
flint -o library.dat routine1.for routine2.for routine3.for
flint -l library.dat program.for
```

3.2 RATTY

RATTY is a FORTRAN preprocessor, which is intended to simplify the job of developing code that is to run on several machines. RATTY checks for some non-ANSI or dubious programming practises, secondly it converts some FORTRAN extensions into standard FORTRAN, and thirdly it handles some compiler directives associated with conditional compilation or code optimization.

Apart from the extra trouble of running the preprocessor before compilation, preprocessors are generally a nuisance, in that they make the code that is developed and the code that is debugged different (both compile and run time debugging). This tends to complicate the debugging process. To this end, RATTY attempts to make the minimum changes necessary to code to get it to compile. Comments and indentation are retained in the output, so that the output should be nearly as readable as the input. Additionally when conditional compilation directives are avoided, it will be found that much code does not need to be preprocessed on a number of compilers.

3.2.1 The Command

The command is:

```
% ratty [-s system] [-I incdir] [-D symbol] [-b] [-?] input output
```

"Input" is a text file containing a mildly extended FORTRAN, whereas "output" is the resultant standard (?) code. The default input and output are the standard input and standard output. A few command line flags are also recognized.

-s The string following this flag indicates the target compiler. RATTY performs some special processing for the following compilers:

vms VAX/VMS FORTRAN compiler.

cft Cray FORTRAN compiler for both CTSS and COS.

f77 UNIX FORTRAN-77 compiler.

convex Convex C-1 compiler.

fx Alliants compiler.

trace Multiflow Trace computers.

sun Sun computers.

RATTY assumes that target compilers other than these are strict FORTRAN-77 compilers.

-I The string following this flag indicates an alternate directory to search for include files. The -I flag can occur several times, giving several directories. When opening include files, first the current directory is check, and then each directory given by the -I flag is check in the order in which they appeared on the command line.

-D The name following this flag is treated as if it appeared in a #define statement. Note that unlike the cc compiler, a space is required between the -D and the name.

-b If this flag is given, every backslash in the input is converted to two backslashes in the output. This is useful when the target compiler treats backslash as an "escape character" (i.e. several UNIX FORTRAN compilers).

? This causes some help (on using ratty) to be printed.

3.2.2 Language Extensions

RATTY coverts converts the following language extensions into standard FORTRAN, where necessary.

- Tab characters are replaced with the corresponding number of blanks.
- DO/ENDDO and DOWHILE/ENDDO can be replaced with standard FORTRAN-77 loops.
- The IMPLICIT NONE and IMPLICIT UNDEFINED statements may be converted to the form required by the local compiler, or eliminated altogether.
- INCLUDE files are expanded.

3.2.3 Optimization Directives

Vector computers occasionally need directives to help optimize some loops. The needed directives, however, vary between manufacturers. RATTY takes optimization directives in a standard form and converts them to the form recognized by the target compiler. A directive is introduced by either a 'c#' or a '#' starting a line (i.e. starting in column 1), for example:

```
c#directive
```

or

```
#directive
```

Either form can be used, but the 'c#' form is preferable, as the directive will be seen as a comment to a standard compiler. The directives are:

ivdep This instructs the compiler to ignore any apparent vector dependencies in the immediately following loop. This is modestly commonly used.

maxloop nn This indicates that the loop count of the immediately following loop will not exceed 'nn'. This enables the compiler to optimize some short loops. For example

```
c#maxloop 32
    do i=1,n
       .
       .
    enddo
```

indicates that the do loop will not execute more than 32 times.

nooptimize This directive appears just before a do-loop that should not be optimized.

3.2.4 Conditional Compilation Directives

When coding a task for several machines, it is sometimes desirable that different code is executed for different machines. For example, to achieve good efficiency, the inner loops may need to be different on a scalar and vector machine.

The basic syntax is the same as the optimization directives (i.e. 'c#' or '#' starting a line, followed by the directive). However, as code using conditional compilation will have to be passed through RATTY to produce the correct code, the '#' form is preferred. This will be seen as an error by a standard compiler, thus preventing code from accidently no being passed through the preprocessor.

There are four directives of interest, namely `ifdef, ifndef, else` and `endif`. Like the cpp preprocessor, `ifdef` and `ifndef` pass the following section of code to the compiler if a particular "symbol" is defined

or not. Currently there are only one or two symbols that are defined. Firstly a symbol with the compiler target name is defined (the target compiler name is gained from the command line -s flag). Secondly a symbol "vector" is defined if the target computer is a vector machine. For example the command line:

```
% ratty -s cft input output
```

causes the symbol 'cft' and 'vector' to become defined. Then, given the following code fragment from "input", the first section is copied to "output", whereas the second is discarded.

```
#ifdef cft
   .
   .  Use this code on the Cray.
   .
#else
   .
   .  This code on all other machines.
   .
#endif
```

Appendix A

Image Items

In this Appendix we will describe the items that can be present in an image dataset. *MIRIAD* utilities such as `prthd`, `imlist`, `itemize`, `puthd`, `copyhd` and `delhd` may be used to browse or modify these 'header variables'.

Note that the units used in *MIRIAD* are not always FITS-like, *e.g.* frequencies are stored in GHz in *MIRIAD*, whereas in FITS one finds Hz. For example, *MIRIAD* keeps velocity in km s^{-1}, sky coordinates in radians and frequencies in GHz.

Table A.1: Item names in *MIRIAD* image datasets

Item	Type	Units	Description/Comments
bmaj,bmin	real	radians	Beam major and minor axis FWHM.
bpa	real	degrees	Beam position angle, measured east from north.
bunit	character		The units of the pixels.

btype	character		The type of the image. Possible values:
			`intensity` – Normal map.
			`beam` – Point spread function.
			`depolarisation_ratio`
			`fractional_polarisation`
			`optical_depth`
			`polarised_intensity`
			`position_angle`
			`rotation_measure`
			`spectral_index`

cdelt1,cdelt2,...	double	⇒ ctype	The increment between pixels.

cellscal	character		Latitude/longitude cell scaling convention with frequency/velocity. Possible values are:
			`CONSTANT`
			`1/F`

crpix1,crpix2,...	real	pixels	The pixel coordinate of the reference pixel.
crval1,crval2,...	double	radians km s^{-1} GHz	The coordinate value at the reference pixel. if ctype represents a celestial coordinate. if ctype `VELO` or `FELO` if ctype `FREQ`
ctype1,ctype2,...	character		The type of the nth axis (as FITS CTYPE).
datamin,datamax	real	⇒ bunit	The minimum and maximum pixel values.
epoch	real	years	The epoch of the coordinate system.
history	text		A text item containing the history of processing performed to the data set.
image	real	⇒ bunit	The pixel data.
llrot	double	radians	Eastward rotation relative to north of the sky grid relative to the pixel grid.
lstart,lstep,lwidth	real		
ltype	character		The linetype used in making the map.
mask	integer	#	Bitmap used to determine which pixels in the image have been blanked.
mostable	binary		Mosaic information table.
naxis	integer	#	Number of dimensions.
naxis1,naxis2,...	integer	#	Number of pixels along each dimension.
niters	integer	#	The total number of deconvolution iterations performed on the image.
object	character		source name
observer	character		observers name
obsra,obsdec	double	radians	The apparent RA and DEC of the observation centre.
obstime	double	Julian date	The mean observing time.
pbfwhm	real	arcsec	Gaussian primary beam size (deprecated).
pbtype	character		Primary beam type. Standard values include: `HATCREEK, VLA, ATCA, WSRT`
restfreq	double	GHz	The rest frequency of the observed data.
telescop	character		Telescope name. Standard values include: `HATCREEK, VLA, ATCA, WSRT`
vobs	real	km s^{-1}	Velocity of the observatory with respect to the rest frame, during the observation.

Appendix B

UV Variables

B.1 UV Dataset

A *MIRIAD* uv dataset is composed of a collection of items and '$u-v$ variables'. The variables are parameters that are known at the time of the observation, and include measured data, and the description of the observation set up (*e.g.* correlator set up and observing centers).

Table B.1 gives a list of the items that are used to build up a *MIRIAD* uv dataset.

The *Programmers Guide* contains more detailed information on how a visibility dataset is constructed, this Appendix only reports which variables can be found in the item `visdata`. The text item `vartable` contains an ordered (for quick indexing) list of all the variables which exist in the `visdata` item.

A list of all items in a visibility dataset is summarised in Table B.1 below. A list of all the uv variables can be obtained with the *MIRIAD* program `uvlist` or `uvio` for the brave of heart.

The storage **types** (2nd column) in the table below are:

```
A -- ascii (NULL terminated)
R -- real (32 bit IEEE)
D -- double (64 bit IEEE)
C -- complex (2 * 32 bit IEEE)
I -- integer (32 bit twos complement)
J -- short (16 bit twos complement)
K -- long (64 bit  twos complement) *** not currently used in visdata ***
```

They are the same as the data type in the first column of the `vartable` item in a *MIRIAD* uv dataset.

Variables with two dimensions have the first dimension varying fastest, the usual FORTRAN notation.

NB: The formal version of this document is recorded as "*January 3, 2008*".

Table B.1: *MIRIAD* items in a uv visibility dataset

Item name	Type	Description
obstype	ascii	value: '`cross`', '`auto`' or '`mixed`'
history	text	history text file (in principle editable)
vartable	text	lookup table for all uv variables (do not edit!)
visdata	mixed	data stream of uv variables
flags	integer	optional flags for narrowband data
wflags	integer	optional flags for wideband data
gains	mixed	antenna gain table (delhd this item to disable gain table)
nfeeds	integer	number of feeds on each antenna
ntau	integer	Number of delay/spectral index terms per antenna in 'gains'
nsols	integer	number of records in 'gains'
ngains	integer	number of antenna gains in each record of 'gains'
interval	double	gain interpolation time tolerance (days)
leakage	complex	polarization leakage parameters
freq0	double	reference frequency for delay terms
freqs	mixed	frequency set up description table for 'bandpass'
bandpass	complex	bandpass function gains (delhd this item to disable passband corrections)
nspect0	integer	number of windows in the bandpass function
nchan0	integer	total number of channels in the bandpass function

Name	Ty	Units	Comments
airtemp	R	centigr.	Air temperature at observatory
antaz(nants)	D	deg.	azimuth of antennas (BIMA was using 0=south CARMA will use 0=north)
antdiam	R	meters	Antenna diameter
antel(nants)	D	deg.	elevation of antennas
antpos(nants, 3)	D	nanosec	Antenna equatorial coordinates, with X along the local meridian (not Greenwich)
atten(nants)	I	dB	Attenuator setting (Hat Ck/CARMA) datatype R ???
axismax(2,nants)	R	arcsec	Maximum tracking error in a cycle. axismax(1,?) is azimuth error, axismax(2,?) is the elevation error.
axisoff(nants)	R	nanosec	Horizontal offset between azimuth and elevation axes (CARMA)
axisrms(2,nants)	R	arcsec	RMS tracking error. axisrms(1,?) is azimuth error, axisrms(2,?) is the elevation error.
baseline	R	-	The current antenna baseline Baseline is stored as $256 * ant1 + ant2$ or $2048 * ant1 + ant2 + 65536$ The uv coordinates are calculated as $uvw = xyz(ant2) - xyz(ant1)$. Note that this is different from the AIPS/FITS convention (where $uvw = xyz(ant1) - xyz(ant2)$). When writing this variable, software must ensure that $ant1 < ant2$. `baseline` is also known as `preamble(4)` or `preamble(5)` depending if you have uv or uvw data resp.
bin	I	-	Pulsar bin number.
cable(nants)	D	nanosec	measured length of IF cable (Hat Ck)
calcode	A	-	ATCA calcode flag
chi or chi(nants)	R	radians	Position angle of the X feed relative to the sky. This is the sum of the parallactic angle and the `evector` variable. If only one value is present, all antennas are assumed to have identical values.
chi2	R	radians	Second feed angle variation (SMA)
coord(*)	D	nanosec	uv(w) baseline coordinates ?? what epoch ?? `coord` is also known as `preamble(1:2)` or `preamble(1:3)` depending if you have uv or uvw data resp.

corbit	R	-	Number of correlator bits (Hat Ck)
corbw(2)	R	MHz	Correlator bandwidth setting (Hat Ck) Must take the values 1.25, 2.5, 5.0, 10.0, 20.0, 40.0 & 80.0 MHz.
corfin(4)	R	MHz	Correlator LO setting before Doppler tracking (Hat Ck) This is the LO frequency at zero telescope velocity Must be in the range 80 to 550 MHz.
cormode	I	-	Correlator mode (Hat Ck). Values are: 1 : 1 window /sideband by 256 channels 2 : 2 windows/sideband by 128 channels 3 : 4 windows/sideband by 64 channels, single sideband 4 : 4 windows/sideband by 64 channels, double sideband
coropt	I	-	Correlator option (Hat Ck) 0 means cross-correlation 1 means auto-correlation Same as the `obstype` item?
corr(nchan)	J or R	-	Correlation data `corr` is really a complex quantity, but the data stream variable can be stored otherwise for efficiency.
cortaper	R	-	On-line correlation taper (Hat Ck) This is the value at the edge of the window The value is from 0-1.
dazim(nants)	R	radians	Offset in Azimuth. (CARMA)
ddec	R	radians	Offset in declination from `dec` in `epoch` coordinates. The actual observed DEC is calculated as dec + ddec.
dec	R or D	radians	Declination of the phase center/tangent point. See `epoch` for coordinate definition. See also `obsdec`
delay(nants)	D	nanosec	delay setting at beginning of integration (Hat Ck)
delay0(nants)	R	nanosec	delay offset for antennas (Hat Ck)
deldec	R or D	radians	Declination of the delay tracking center. See `epoch` for coordinate definition.
delev(nants)	R	radians	Offset in Elevation (CARMA)
delra	R or D	radians	Right ascension of the delay tracking center. See `epoch` for coordinate definition.
dewpoint	R	centigr.	Dew point at weather station (Hat Ck)
dra	R	radians	Offset in right ascension from `ra` in `epoch` coords. The actual observed RA is calculated as ra + dra/cos(dec).
epoch	R	years	A badly named variable – this defines the mean equinox and equator for the equatorial coordinates `ra`, `dec`, `dra` and `ddec`. The epoch of the coordinates is actually the observing time. Values less than 1984.0 are Besselian with coordinates in the FK4 system. Values greater than 1984.0 are Julian with coordinates in the FK5 system. You will typically find 1950.0 or 2000.0 here.
evector or evector(nants)	R	radians	Position angle of the X feed, to the local vertical. If only one value is present, all antennas are assumed to be identical.
focus(nants)	R	volts	Focus setting (Hat Ck)
freq	D	GHz	Rest frequency of the primary line
freqif	D	GHz	? (Hat Ck only?)
inttime	R	seconds	Integration time (see also time)
ischan(nspect)	I	-	Starting channel of spectral window
ivalued(nants)	I	?	Delay step (Hat Ck) Used in an attempt to calibrate amp and phase vs. delay.
jyperk	R	Jy/K	The efficiency Jy/K, calculated during online calibration
jyperka(nants)	R	sqrt(Jy/K)	Antenna based Jy/K, calculated during online calibration (Hat Ck)
latitud	D	radians	Geodetic latitude of the observatory.
lo1	D	GHz	First local oscillator (Hat Ck/CARMA) `lo1` is in the range 70 GHz - 115 GHz for 3mm.

lo2	D	GHz	Second local oscillator (Hat Ck)
longitu	D	radians	Longitude of the observatory.
lst	D	radians	Local apparent sidereal time.
modedesc	A	-	Correlator mode description (CARMA only) Example: 500-32-8-X-X-X-X-X
mount or mount(nants)	I	-	The type of antenna mounts. If only one value is given, all antennas are assumed to be the same. Possible values are: 0: Alt-az mount. 1: Equatorial mount. 2: X-Y. 3: orbiting. 4: bizarre.
name	A	-	ATCA raw RPFITS file name.
nants	I	-	The number of antennas Following variables use a dimension of `nants`: antpos(nants, 3) focus(nants) phaselo[1-2](nants) phasem1(nants) systemp(nants, nspect) wsystemp(nants, nwide) temp(nants, ntemp) tpower(nants, ntpower) axisrms(2,nants) dazim(nants) delev(nants) The antennas are always numbered starting at 1.
nbin	I	-	Total number of pulsar bins.
nchan	I	-	The total number of individual frequency channels The following variables have the dimension of nchan: corr(nchan)
npol	I	-	The number of simultaneous polarisations
nschan(nspect)	I	-	Number of channels in spectral window
nspect	I		Number of spectral windows Following variables use a dimension of `nspect`: ischan(nspect) nschan(nspect) restfreq(nspect) sdf(nspect) sfreq(nspect) systemp(nants, nspect)
ntemp	I	-	Number of antenna thermisters Following variables use a dimension of `ntemp`: temp(nants, ntemp)
ntpower	I	-	Number of total power measurements The following variable depends on `ntpower`: tpower(nants,ntpower) `ntpower` is currently 1, could be more later.
nwide	I	-	Number of wideband channels Variables which depend on `nwide` are: wfreq(nwide) wwidth(nwide) wcorr(nwide) wsystemp(nants,nwide)
obsdec	D	radians	Apparent declination of the phase centre/tangent point at time of observation. See also `dec`
observer(*)	A	-	The name of the observer
obsline(*)	A	-	The name of the primary spectral line of interest to the observer
obsra	D	radians	Apparent right ascension of the phase centre/tangent point at time of observation. See also `ra`
on	I	-	Either 1, 0 or -1, for on, off pointing, and Tsys spectrum resp. for auto-correlation data.

operator(*)	A	-	The name of the current operator
pbfwhm	R	arcsec	(Deprecated) Primary Beam at Full Width Half Maximum For Hat Ck, it is approximately 11040.0/`lo1`.
pbtype(*)	A	-	Primary beam type to be used in imaging.
phaselo1(nants)	R	radians	Antenna phase offset (Hat Ck/CARMA)
phaselo2(nants)	R	radians	Second LO phase offset (Hat Ck/CARMA)
phasem1(nants)	R	radians	IF cable phase (Hat Ck/CARMA)
plangle	R	degrees	Planet angle
plmaj	R	arcsec	Planet major axis (note units)
plmin	R	arcsec	Planet minor axis
pltb	R	Kelvin	Planet brightness
pntdec	R or D	radians	Declination of the pointing center. See `epoch` for coordinate definition.
pntra	R or D	radians	Right ascension of the pointing center. See `epoch` for coordinate definition.
pol	I	-	Polarization type of the correlation data. Values follow the AIPS/FITS convention, viz: 1: Stokes I 2: Stokes Q 3: Stokes U 4: Stokes V -1: Circular RR -2: Circular LL -3: Circular RL -4: Circular LR -5: Linear XX -6: Linear YY -7: Linear XY -8: Linear YX
precipmm	R	mm	Mm of precipitable water vapour in the atmosphere.
pressmb	R	millibar	atmospheric pressure.
project(*)	A	-	The name of the current project
purpose(*)	A	-	Scientific intent or purpose For CARMA: B=bandpass, F=flux, G=gain (phase/amp) P=polarization, R=radio pointing, S=science target, O=other
ra	R or D	radians	Right ascension of the phase center/tangent point. See `epoch` for the definition of the coordinate system. See also `obsra`
rain	R	mm	The current amount of water in the rain gauge. The rain gauge is emptied at 9:00 AEST (ATCA).
refpnt(2,nants)	R	arcsec	Reference pointing offsets. refpnt(1,?) is azimuth offset, refpnt(2,?) is the elevation offset.
relhumid	R	%	Relative Humidity at observatory
restfreq(nspect)	D	GHz	Rest frequency for each spectral window. This may be zero for continuum observations.
rmspath	R	microns	RMS path variation (CARMA, for HatCrk units were %) see also smonrms
sctype	A	-	Scan type (ATCA?)
sdf(nspect)	D	GHz	Change in frequency per channel
sfreq(nspect)	D	GHz	Sky frequency of (center of) first channel in window
smonrms	R	μm	ATCA seeing monitor rms value (see also rmspath)
source(*)	A	-	The name of the source
srv2k(nants)	R	?	??? (Hat Ck)
systemp or systemp(nants) or systemp(nants,nspect)	R	Kelvin	Antenna system temperatures
tau230	R	-	Optical depth at 230 GHz, as measured with the ... system (Hat Ck/CARMA)
tcorr	I	-	HasTsys correction has been applied (0:none, 1:applied) (CARMA, ATNF)

telescop(*)	A	-	The telescope name. Some standard values are:
			`'ATCA'`
			`'HATCREEK'`
			`'VLA'`
			`'WSRT'`

temp (nants, ntemp)	R	centigr.	Antenna thermistor temperatures (Hat Ck)
themt(nants)	R	Kelvin	temperature of the hemt amplifier (Hat Ck)
tif2(nants)	R	Kelvin	temperature of IF amplifier (Hat Ck)
time	D	days	The time (nominally UT1) stored as a Julian date. For example, noon on Jan 1, 1980 is 2,444,240.0! `time` is also known as `preamble(3)` or `preamble(4)` depending if you have uv or uvw data resp. time is the beginning of an integration with length inttime
tpower (nants, ntpower)	R	volts	Total power measurements (Hat Ck)
trans	R	K	CARMA
tscale	R	-	Optional correlation scale factor Used only when `corr` is stored as J (16 bits).
tsis(nants)	R	Kelvin	temperature of the SIS mixers (Hat Ck)
tsky	R	-	CARMA
ut	D	radians	The time since midnight Universal time (nominally UT1).
veldop	R	km s^{-1}	The sum of the radial velocity of the observatory (in the direction of the source, with respect to the rest frame) and the nominal systemic radial velocity of the source.
veltype(*)	A	-	Velocity rest frame. Possible values for `veltype` are: VELO-LSR: rest frame is the LSR VELO-HEL: rest frame is the barycentre VELO-OBS: rest frame is the observatory FELO-LSR: rest frame is the LSR (deprecated) FELO-HEL: rest frame is the barycentre (deprecated)
version(*)	A	-	The current hardware/software version Current options: oldhat, newhat For carma: x.y.z
vsource	R	km s^{-1}	Nominal radial systemic velocity of source. Positive velocity is away from observer.
wcorr(nwide)	C	-	Wideband correlations. The current ordering is: wcorr(1:2) are the digital LSB and USB. wcorr(3:4) are the analog LSB and USB.
wfreq(nwide)	R	GHz	Wideband correlation average frequencies (center?)
wind	R	km/h	Wind speed in km/h (ATCA)
winddir	R	deg	Wind direction (where the wind is blowing from) (note: originally encoded as 'N', 'SE', 'W', etc.)
windmph	R	mph	Wind speed - in imperial units!
wsystemp or wsystemp(nants) or wsystemp(nants,nwide)	R	K	System temperature for wide channels.
wwidth(nwide)	R	GHz	Wideband correlation bandwidths
xsampler (3,nants,nspect)	R	percent	X sampler statistics (ATCA).
xtsys(nants,nspect)	R	Kelvin	System temperature of the X feed (ATCA).
xtsysm(nants,nspect)	R	Kelvin	???
xyamp(nants,nspect)	R	Jy	On-line XY amplitude measurements (ATCA).
xyphase (nants,nspect)	R	radians	On-line XY phase measurements (ATCA).
ysampler (3,nants,nspect)	R	percent	Y sampler statistics (ATCA).
ytsys(nants,nspect)	R	Kelvin	System temperature of the Y feed (ATCA).
ytsysm(nants,nspect)	R	Kelvin	???

B.2 Telescope specific notes

A reminder on some telescope specific variables

B.2.1 ATCA

```
calcode
name
rain
sctype
smonrms
wind
xsampler(3,nants,nspect)
xtsys(nants,nspect)
xyamp(nants,nspect)
xyphase(nants,nspect)
ysampler(3,nants,nspect)
ytsys(nants,nspect)
```

B.2.2 CARMA

```
dazim(nants)
delev(nants)
modedesc
axisrms       "skyErr"  -- temporary sqrt(2) issue
axisoff
lo1 changes, phaselo1=0
lo2 still absent
purpose
```

B.2.3 SMA

```
chi2
```

B.2.4 BIMA/Hat Creek

Although the telescope name is for historic reasons called `HATCREEK`, they are really the 6m `BIMA` antennae, but while this array was operational at the Hat Creek site in Northern California. The following UV variables were specificially used for this array, although some of them moved to CARMA as well:

```
atten(nants)
cable(nants)
corbit
corbw(2)
corfin(4)
cormode
coropt
cortaper
delay(nants)                    carma
delay0(nants)
dewpoint
focus(nants)
freqif
ivalued(nants)
lo1                             carma
lo2
phaselo1(nants)                 carma
phaselo2(nants)                 carma
phasem1(nants)                  carma
rmspath                         carma
srv2k(nants)
tau230                          carma
temp(nants, ntemp
themt(nants)
tif2(nants)
tpower(nants, ntpower)
tsis(nants)
```

B.3 Examples

```
% ls -l 3c273/
total 1808
-rw-r--r--    1 teuben   teuben      49952 Oct 12  1998 flags
-rw-r--r--    1 teuben   teuben        136 Jul 24  1998 header
-rw-r--r--    1 teuben   teuben      48700 Oct 12  1998 history
-rw-r--r--    1 teuben   teuben        671 Jul 24  1998 vartable
-rw-r--r--    1 teuben   teuben    1725300 Jul 24  1998 visdata
-rw-r--r--    1 teuben   teuben       1760 Jul 30  1998 wflags

% itemize in=3c273
Itemize: Version 31-JUL-97
  nwcorr   = 13608
  ncorr    = 387072
  vislen   = 1725304
  obstype  = crosscorrelation
  history    (text data, 48704 elements)
  visdata    (binary data, 1725300 elements)
  vartable   (text data, 675 elements)
  flags      (integer data, 12487 elements)
  wflags     (integer data, 439 elements)

% uvio 3c273
uvio Version 16-jan-01 rjs
0x       0 FILE: 3c273
0x       0 SIZE: project   Count=12,Type=a
0x       8 DATA: project   n196d028.cal
0x      18 SIZE: source    Count=5,Type=a
0x      20 DATA: source    3C273
0x      30 SIZE: ra        Count=1,Type=d
0x      38 DATA: ra                 3.26861624
0x      48 SIZE: dec       Count=1,Type=d
0x      50 DATA: dec              0.03582093395
0x      60 SIZE: vsource   Count=1,Type=r
0x      68 DATA: vsource                    0
0x      70 SIZE: plmaj     Count=1,Type=r
0x      78 DATA: plmaj                      0
0x      80 SIZE: plmin     Count=1,Type=r
.....
0x    12b0 DATA: tif2              34.60030746
0x    12e8 SIZE: pol       Count=1,Type=i
0x    12f0 DATA: pol       -6
0x    12f8 SIZE: wcorr     Count=18,Type=c
0x    1300 DATA: wcorr            0.1456700563        -0.1822117269
0x    1398 SIZE: tscale    Count=1,Type=r
0x    13a0 DATA: tscale         0.0009044817416
0x    13a8 SIZE: corr      Count=1024,Type=j
0x    13b0 DATA: corr      0
0x    1bb8 SIZE: coord     Count=2,Type=d
0x    1bc0 DATA: coord            -96.06886361
0x    1bd8 SIZE: time      Count=1,Type=d
0x    1be0 DATA: time              2450671.342   97AUG10:20:12:01.1
0x    1bf0 SIZE: baseline  Count=1,Type=r
0x    1bf8 DATA: baseline                  260
0x    1c00 ========== EOR (1) ========
0x    1c08 DATA: wcorr            0.1115201488         0.1867246479
0x    1ca0 DATA: tscale          0.001088747638
0x    1ca8 DATA: corr      0
0x    24b0 DATA: coord             19.81238265
0x    24c8 DATA: baseline                  516
0x    24d0 ========== EOR (2) ========
0x    24d8 DATA: wcorr             0.106826134        -0.0437762402
0x    2570 DATA: tscale         0.0009396394016
0x    2578 DATA: corr      0
0x    2d80 DATA: coord            -8.372860208
0x    2d98 DATA: baseline                  261
0x    2da0 ========== EOR (3) ========
...
0x   15068 DATA: baseline                 2314
0x   15070 ========== EOR (36) ========
```

```
0x   15078 DATA: obsra              3.268015211
0x   15088 DATA: obsdec           0.03609215511
0x   15098 DATA: chi              -0.7118714452
0x   150a0 DATA: tpower             19.17340851
0x   150d8 DATA: ut                 5.289289984
0x   150e8 DATA: lst                2.459578844
0x   150f8 DATA: axisrms           0.2067008913
0x   15160 DATA: focus              7.899638176
0x   15198 DATA: delay              289.0346413
0x   15200 DATA: antaz              1.040362579
0x   15268 DATA: antel             0.5775128425
0x   152d0 DATA: themt                      475
0x   15308 DATA: tif2               34.53898239
0x   15340 DATA: wcorr            0.02284911089        -0.1492281109
0x   153d8 DATA: tscale         0.0009604850202
0x   153e0 DATA: corr      0
0x   15be8 DATA: coord              -95.98970399
0x   15c00 DATA: time                2450671.342   97AUG10:20:12:13.0
0x   15c10 DATA: baseline                   260
0x   15c18 ========== EOR (37) ========
0x   15c20 DATA: wcorr              0.2651385069        0.05663052946
0x   15cb8 DATA: tscale         0.0009315458592
0x   15cc0 DATA: corr      0
0x   164c8 DATA: coord              19.82889086
0x   164e0 DATA: baseline                   516
0x   164e8 ========== EOR (38) ========
0x   164f0 DATA: wcorr              0.2428172529         0.1108904481
.....
```

```
As of version 4.1.0 we will have two types of installation:
As of version 4.2.0 we will require flex, needed by wcslib

1) What we now call "old-style" installation:

You will want to execute the $MIR/install/install.miriad script to
install miriad, it also has a number of checks to deal with system
dependant things (e.g. linux vs. solaris). We expect a different, and
easier to use, installation method sometime during the Miriad V4
development, which will be based on the GNU 'autoconf' toolkit.

This method will be deprecated some time in the future when it starts to
conflict with the method below.

2) The new style uses the GNU auto tools and libtool. Some reasonably modern
   version of this is needed. You might otherwise need to locally install
   m4, autoconf, automake and libtool (in that order) for a developer version.

Quick summary. On the first compile:

        ./autogen.sh
        ./configure --prefix=`pwd`/build  F77=gfortran
                 (prefix is just an example, and F77= can also be left off
        make
        make install

If you have just updated the source code, you can skip the first
several steps:

        make (will recompile only those tasks that have changed)
        make install

To set up your shell to run Miriad tasks:

        source build/lib/miriad/automiriad.csh (if using a C shell)
        source build/lib/miriad/automiriad.sh (if using a Bourne shell)

It is recommended that you read the detailed instructions below.

================================================
Detailed instructions for the new build system
================================================

If you are getting Miriad directly from CVS, you will first need to
create the build scripts. (If a file called 'configure' exists in this
directory, you can skip this step.) Run:

        setenv LIBTOOLIZE glibtoolize    (only on mac)
        ./autogen.sh

This requires the GNU "auto tools": libtool, autoconf, and
automake. If 'autogen.sh' fails, the output will likely be hard to
interpret; see the section "Handling Auto Tools Errors" at the bottom
of this file for suggestions.

Once the build scripts are created, you run the 'configure' script to
probe your system's capabilities and generate the build instructions
for your particular machine:

        ./configure --prefix=`pwd`/build  (prefix is just an example)

The configure script takes several options that you may wish to use:
```

```
        --prefix : Controls where the Miriad executables are
          installed. Defaults to /usr/local. The above example will
          install the executables in a directory called 'build' within
          the Miriad source tree.

        --with-telescope=NAME : Specifies which telescope's data
          Miriad is expected to reduce. Sets the various MAX*
          variables to values appropriate to the given
          telescope. Possible values for NAME are: ata, atnf, bima,
          carma, lofar, fasr, gmrt, sma, and wrst.

        --help : Shows other options that 'configure' accepts.

Configure may exit with an error message if it doesn't know how to
handle your system, but will know about virtually all modern computer
hardware. Sometimes errors encountered in running the 'autogen.sh'
script won't manifest themselves until you run 'configure'; if you get
a mysterious error message, you should probably ask for help as
suggested in the "Handling Auto Tools Errors" section at the end of
this file. Once the configure script has been run successfully, use
the 'make' program to compile Miriad:

        make            (generates executables and support files)
        make install    (places the files inside the prefix directory)

(The compile system should work with several vendor's implementations
of make, but GNU make is recommended; it may be called 'gmake' on your
system.)

IMPORTANT: If you are using Miriad from CVS and update the sources,
you only need to rerun 'make' and 'make install' to update your Miriad
programs. You do NOT need to rerun 'autogen.sh' or 'configure'. Make
will recompile as few files as possible while ensuring that the whole
suite is up-to-date, saving time when there are small, incremental
updates to the source code.

Once Miriad is installed, you can set up a shell's environment to run
Miriad tasks by sourcing one of the following files:

        source <prefix>/lib/miriad/automiriad.csh (if using a C shell)
        source <prefix>/lib/miriad/automiriad.sh  (if using a Bourne shell)

where <prefix> is the value of the --prefix option to
'configure'. (These files are equivalent to MIRRC.<hosttype> or
MIRSRC.<hosttype> from the old-style build system.) The script will add the
directory containing the Miriad binaries to your shell's path,
allowing you to run the Miriad tasks. It will also set the following
environment variables:

        MIR : the location of the Miriad source code tree.

        MIRCAT : the location of some support Miriad data files.

        MIRPROG : the directory inside which all of the Miriad
          task source code is stored.

        MIRSRC : the directory inside which all Miriad source code is
          stored.

        MIRSUBS : the directory inside which the source code of the
          Miriad subroutine library is stored.

        MIRBIN : the directory in which the Miriad task binaries are
```

```
          installed.

        MIRLIB : the directory in which the Miriad library files are
          installed.

        MIRINC : the directory in which the Miriad include files are
          installed.

        MIRDOC : the directory in which the Miriad documentation files
          are installed.

        MIRPDOC : the directory containing the Miriad task
          documentation files in particular.

        MIRSDOC : the directory containing the Miriad subroutine
          documentation files in particular.

        MIRNEWS : $MIR/news, a nonexistant file. (Defined for
          compatibility with MIRRC.local.)

        PGPLOT_DIR : the directory containing the 'pgxwin_server'
          executable used by the PGPLOT plotting package. This is a
          slight abuse of the PGPLOT_DIR environment variable.

        PGPLOT_FONT : the location of the 'grfont.dat' resource file
          used by the PGPLOT plotting package.

        PGPLOT_RGB : the location of the 'rgb.txt' resource file used
          by the PGPLOT plotting package.

The 'automiriad' scripts also source the scripts $MIR/MIRRC.local (or
$MIR/MIRSRC.local) and $HOME/.mirrc (or $HOME/.mirsrc) if they
exist. This allows you to personalize your Miriad environment if you
so desire.

================================================
Tips for the new build system
================================================

1) Shortcuts to automiriad

To save typing you may want to put something like the following alias
into your shell's initialization scripts:

        alias mirenv="source <prefix>/lib/miriad/automiriad.csh"

(if you use a C shell). Then you can just type 'mirenv' to set up your
shell to run Miriad tasks.

2) Separate build directories

If you wish to compile different versions of Miriad, or merely wish to
keep the Miriad source directories clean of built files, you can
create a separate "build directory" and run 'configure' from that
directory:

        mkdir $HOME/miriad-build
        cd $HOME/miriad-build
        /path/to/miriad/source/configure --prefix=/opt/miriad4.1
        make
        make install

In this example, all of the built files will go in $HOME/miriad-build,
```

```
while the build system will be smart enough to find the source code in
the directory containing the configure script that you ran. If you
wish to update your Miriad installation, you can run 'cvs up' in the
Miriad source code directory, then run 'make' and 'make install' in
the build directory.

One very useful aspect of this capability is that you can use one copy
of Miriad source code to build several different versions of Miriad.
For instance, you could compile Miriad for two different telescopes:

        mkdir $HOME/miriad-build-carma
        cd $HOME/miriad-build-carma
        /path/to/miriad/source/configure --with-telescope=carma \
          --prefix=/opt/miriad-carma
        make install

        mkdir $HOME/miriad-build-ata
        cd $HOME/miriad-build-ata
        /path/to/miriad/source/configure --with-telescope=ata \
          --prefix=/opt/miriad-ata
        make install

3) Installation directories

The autotools-based Miriad install uses directory names that will not
pollute the <prefix> directory in unexpected ways. That is, you can
configure Miriad with a prefix of /usr or /usr/local and not worry
about overwriting standard system files. The installation directories
that Miriad uses are:

        <prefix>/bin : Miriad task binaries.

        <prefix>/lib : Miriad support shared libraries.

        <prefix>/lib/miriad : System-specific Miriad support files
          that are not shared libraries.

        <prefix>/lib/pgplot-miriad-remix : System-specific PGPLOT
          support files that are not shared libraries.

        <prefix>/libexec : Support executables for Miriad not
          intended to be run directly by the user.

        <prefix>/include/miriad-f : Miriad Fortran include files.

        <prefix>/include/miriad-c : Miriad C include files.

        <prefix>/share/miriad : System-independent Miriad support
          files.

        <prefix>/share/pgplot-miriad-remix : System-independent PGPLOT
          support files.

That being said, because Miriad installs so many files into
<prefix>/bin, it is suggested that you install it into its own prefix,
such as $MIR/build or /opt/miriad.

4) Traditional PGPLOT_DIR

A consequence of the above directory structure is that the PGPLOT data
files are installed in locations different than what a traditional
PGPLOT installation uses. The 'automiriad' scripts account for this,
but if you would rather have the traditional single PGPLOT_DIR, you
```

```
can just copy the appropriate files into that directory manually:

        cd <PGPLOT_DIR of your choice>
        cp <prefix>/share/pgplot-miriad-remix/* .
        cp <prefix>/libexec/* .

You'll then want to create a $MIR/MIRRC.local or a $HOME/.mirrc to
override the PGPLOT environment variable settings that the
'automiriad' scripts create.

5) Preserving the source directory

If you are used to compiling software with the usual trio of
'configure ; make ; make install', you may be used to being able to
delete the source code once your software has been installed. Miriad
will still function if you delete its source code, but it's
recommended that you keep the source tree around. It contains a lot of
uninstalled data files that useful and, more importantly, the source
code to all of the tasks, which you will probably find very useful in
understanding what Miriad is doing or tweaking existing tasks to fit
your particular needs.

6) Orthogonality with old-style build system

You should currently be able to compile Miriad using both the
old-style and new-style build systems at the same time. The new-style
build system will put the compiled files in different places than the
old-style one, and the files needed to make the new-style system work
don't overwrite the old-style build system files.

However, once the new-style Miriad is installed, the two versions
should be functionally identical, except for the name of the shell
script that you source: MIRRC.<hosttype> in the old style,
automiriad.csh in the new style. All of the environment variables
defined by MIRRC are exported by the new script and have the same
semantics, so long as you don't delete the Miriad source code. (See
item (5) in this section.)

To be extra sure that old-style and new-style builds aren't
interfering with each other, try using a separate build directory as
described in item (2) in this section.

7) Other 'make' targets

Because Miriad's Makefiles are generated by 'automake', the standard
GNU makefile targets are supported:

        clean : Delete most built files.

        dist : Generate a gzipped TAR file of the source code
          tree. (Not guaranteed to include all necessary files, since
          the main method of distributing Miriad is CVS.)

and so on.

8) Building subsets of Miriad

If you only want to compile a subset of the Miriad tasks, you can run
'make' and 'make install' inside the directory containing the source
code for those tasks. For instance, if you've change the source code
only of the 'clean' task, you can save some time by running:

        cd $MIRPROG
```

```
        cd deconv
        make
        make install

as opposed to running 'make' from $MIR.

9) More portable binaries

If gfortran is your Fortran compiler, you can force it to be run with
the -static-libgfortran flag to statically link the Fortran libraries,
which will help the generated MIRIAD binaries be compatible with
systems which only have g77 installed. This can be accomplished with

        ./configure F77="gfortran -static-libgfortran" ...

This option, however, may lead to build failures on some systems.
Initial investigations suggests that it works on Fedora machines but
not on SuSE machines.

================================================
Handling Auto Tools Errors
================================================

The GNU auto tools are very good at providing portability to a large
number of systems, but they are notorious for their fragility and the
impenetrable errors messages they can generate.

If the 'autogen.sh' or 'configure' script fails and gives you an error
message that you don't immediately know how to handle, the most
efficient course of action is probably to consult to consult with your
local system administrator or a knowledgeable Linux
programmer. Fortunately, someone familiar with the auto tools can
resolve most of the problems they cause without specific knowledge of
the program being compiled.

When asking for help, it is vital that your provide the COMPLETE and
EXACT output of a fresh run of both the 'autogen.sh' and 'configure'
scripts, as well as the EXACT command lines that you used. It will
often be helpful to provide the COMPLETE 'config.log' file that
'configure' generates. If the person you're asking is not familiar
with building Miriad, you might suggest that they read this file to
get an overview of Miriad's build system.

Even if you give your helper the most complete possible information,
you will probably have to iterate a couple of times to make sure that
the problem is solved. Please be considerate of your helper's time and
other priorities as they work through the problem with you.
```

Index