Input/Output

Input/Output

* Input/output (i/o) can be more a lot more flexible than just reading typed input from the terminal window and printing it back out to a screen.
* Fortran allows for multiple file streams.
* Fortran allows multiple representations of the data for i/o,
* Fortran allows multiple approaches to the sequencing of i/o.

Multiple File Streams

A keyword nearly universal to all Fortran i/o statements is the **Logical Unit**

```
write(11,*)u          ! u written to file associated with logical unit 11
write(12,*)v          ! v written to file associated with logical unit 12

integer :: lun=3
read(lun,*) n            ! logicial unit can be a variable
```

Default filename associated with logical unit lun is fort.lun (fort.1 1, fort.1 2 fort.3). Compilers may vary!

Some More Definitions

* **File** - a collection of data
* Data is organized into **records**, which may be **formatted** (character representation), **unformatted** (machine binary representation), or denote an end of file.

The Open Statement

The **Open** statement associates a logical unit with a specific file:

```
OPEN([UNIT=]< integer >,&
    FILE=< filename >, IOSTAT=< status_tag >, &
    STATUS=< status >, ACCESS=< method >,&
    FORM=< format >,&
    ACTION=< mode >, RECL=< int-expr >)
```

Examples:

```
OPEN(unit=10,file='input.u',form='formatted')
OPEN(21,file='output.dat',form='unformatted',status='OLD')
```

The Open Statement (cont)

More on open keywords:

IOSTAT: a returned integer variable, zero for successful execution;
other values for various errors.
STATUS: character string - ‘UNKNOWN’ (default); ‘OLD’; ‘NEW’;
‘REPLACE”; ‘SCRATCH’
ACCESS: ‘SEQUENTIAL’ (default) or ‘DIRECT’
FORM: ‘FORMATTED’ or ‘UNFORMATTED’
POSITION: ‘ASIS’ (default) or ‘REWIND’ or ‘APPEND’
RECL: record length for direct access i/o
ACTION: actions one can take with the file - ‘READWRITE’ (default);
“READ”; or ‘WRITE’

One can open an already connect file to change its properties

The Close Statement

The **Close** statement terminates the connection of a file to a logical unit.

```
Close([UNIT=]< integer >,&
    IOSTAT=< status_tag >, &
    STATUS=< status >)
```

IOSTAT: integer returned containing theerror status of the call, zero if no errors

STATUS: what to do with the closed file - ‘KEEP’ (default) or ‘DELETE’

The Write Statement

The **Write** statement does output generally:

```
WRITE([UNIT=]< integer >,&
   [FMT=]< format >, IOSTAT=< status_tag >, &
   END=< label >, ERR=< label >, &
   ADVANCE=< advance_mode >, REC=< record_num >)
```

END and **ERR** are obsolete, avoid!

Examples:

```
WRITE(11,'(5e15.8)')t(:)
WRITE(IOSTAT=status_int,UNIT=lun, ADVANCE='NO')t
WRITE(ERR=909,UNIT=11)t
.
.
909 CONTINUE
```

The Read Statement

The Read statement does input:

```
OPEN([UNIT=]< integer >,&
    [FMT=]< format >, IOSTAT=< status_tag >, &
    END=< label >, ERR=< label >, &
    ADVANCE=< advance_mode >, REC=< record_num >)
```

Examples:

```
READ(11,’(5e15.8)’)t(:)
READ(IOSTAT=status_int,UNIT=lun, ADVANCE=’NO’)t
READ(ERR=909,END=910,UNIT=11)t
.
.
909 CONTINUE
.
.
910 CONTINUE
```

Formatting

The Format specifier is used in read, write and print statements.

* - default, or list-directed formatting (space or comma delimited)
    - f (floating point) for printing of reals
        - syntax: fw.d
            - w = total number of positions
            - d = number of places after the decimal point
            - the decimal point occupies a position, as does a minus sign
    - e (exponential) for large or small real numbers - ew.d
        - d = number of digits in mantissa
    - a (alphanumeric) for character strings
    - i (integer) for integer - can use iw.d format, where the d will pad in front of the value with zeroes

Examples:

exp_format1.f90, exp_format2.f90, exp_format3.f90

Unformatted I/O

* When the file is opened with form='unformatted' the binary will be read/written in the machine representation. Use no format specifier!
* Warning! Different machines may have different representations - big_endian vs. little_endian; the latter is generally found on PC chips.

Sequential vs. Direct Access

* In sequential access the end of record is marked in the file.
* As name implies, each read/write proceeds to next record - exception when ADVANCE='NO' used.
* Can move file position with Position statements

Direct Access

* Must open file with ACCESS='DIRECT' and specify a record length (recl) (generally in bytes)
* You go directly where you wish in the file by specifying the record number (rec=n) in the READ/WRITE
* Multiple jobs/processes can access the file without interference

Other Useful Statements

The Inquire statement can get information about a file. You may inquire by **unit**, or by **filename**:

```
INQUIRE([UNIT=]< integer >,&
    EXIST=< logical >, IOSTAT=< integer >, &
    NAME=< character >, OPENED=< logical >)

INQUIRE([FILE=]< filename >,&
    EXIST=< logical >, IOSTAT=< integer >, &
    NUMBER=< integer >, OPENED=< logical >)  ! plus many more available
                                             !  arguments
```

Position statements:

```
REWIND lun;        REWIND (UNIT=lun, IOSTAT=status_int)
BACKSPACE lun;     BACKSPACE (UNIT=lun, IOSTAT=status_int)
ENDFILE lun;       ENDFILE(UNIT=lun, IOSTAT=status_int)
```

Other Useful Statements

Namelist i/o, a type of formatted i/o (deprecated):

```
logical :: dopbp
integer :: ijtlen
NAMELIST /pbplist/ dopbp,ijtlen
open(unit=2,file='namel.pbp',form='formatted')
read(2,pbplist)
```

```
>cat namel.pbp
 &pbplist
 dopbp=.true.
 IJTLEN=4
 &END
```

Internal files: unit is a program variable rather than a file, no open statement used.

```
character (len=4) :: year
write(unit=cyear,fmt='(i4.4)') 1989
```

I/O Libraries

* Typically, with standard fortran i/o statements, when someone sends you a file, he must also send you a readme about the contents (which variables, dimensions, format, etc.) or some code kernel for reading.
* It sure would be nice if the data in the files were 'self-describing' with the use of 'meta-data.'
* I/O libraries are publicly available that can do this: NetCDF, HDF.

NetCDF

* Something of a standard for climate/meteorogical data - http://www.unidata.ucar.edu/software/netcdf/
* Includes command line utilities to inspect the files (ncdump)
* Many graphics packages can read it (IDL)
* NCO (http://nco.sourceforge.net/) is a set of utilities to manipulate netcdf files
* Fortran subroutine calls are used to read/write/inquire about the data.

NetCDF Philosophy

* NetCDF (network Common Data Form) is a set of interfaces for array-oriented data access and a freely-distributed collection of data access libraries for C, Fortran, C++, Java, and other languages. The netCDF libraries support a machine-independent format for representing scientific data. Together, the interfaces, libraries, and format support the creation, access, and sharing of scientific data.

* NetCDF data is:

* • *Self-Describing*. A netCDF file includes information about the data it contains.
* • *Portable*. A netCDF file can be accessed by computers with different ways of storing integers, characters, and floating-point numbers.
* • *Direct-access*. A small subset of a large dataset may be accessed efficiently, without first reading through all the preceding data.
* • *Appendable*. Data may be appended to a properly structured netCDF file without copying the dataset or redefining its structure.
* • *Sharable*. One writer and multiple readers may simultaneously access the same netCDF file.
* • *Archivable*. Access to all earlier forms of netCDF data will be supported by current and future versions of the software.

NetCDF examples

```
include "netcdf.inc"
status = nf_open(                                                   &
       “infile.nc", nf_nowrite, ncidin)
status = nf_inq_ndims( ncidin, ndims)
status = nf_inq_nvars( ncidin, nvars)
do n = 1,ndims
  status = nf_inq_dim(ncidin, n, dimname, dimlen)
enddo
do n = 1,nvars
  status = nf_inq_var(ncidin, n, varname, vartype, vardims, &
                vardimids, varnatts )
  do k = 1,varnatts
    status = nf_inq_attname(ncidin, n, k, attname)
  enddo
  if(vartype.eq.nf_float)                                           &
        status = nf_get_var_real(ncidin, n, float_1din )
enddo
```