Chapter 7

Input/Output Processing

This chapter contains the following subsections:

- "Records"
- "I/O Statements"
- "Files"
- "Methods of File Access"
- "Units"

Input statements copy data from external media or from an internal file to internal storage. This process is called *reading*. Output statements copy data from internal storage to external media or to an internal file. This process is called *writing*.

The Fortran input/output (I/O) facilities give you control over the I/O system. This section deals primarily with the programmer−related aspects of I/O processing, rather than with the implementation of the processor−dependent I/O specifications.

See Chapter 1 of the *Fortran 77 Programmer's Guide* for information on extensions to Fortran 77 that affect I/O processing.

Records

A *record* is simply a sequence of values or characters. Fortran has three kinds of records:

- formatted
- unformatted
- endfile

A *record* is a logical concept; it does not have to correspond to a particular physical storage form. However, external media limitations can also limit the allowable length of records.

Formatted Records

A *formatted record* contains only ASCII characters and is terminated by a carriage−return or line−feed character. Formatted records are required only when the data must be read from the screen or a printer copy.

A formatted record can be read from or written to only by formatted I/O statements. Formatted records are measured in characters. The length is primarily a function of the number of characters that were written into the record when it was created, but it may be limited by the storage media or the CPU. A formatted record may have a length of zero.

Unformatted Records

Unformatted records contain sequences of values; both character and noncharacter are not terminated by any special character and cannot be accurately comprehended in their printed or displayed format. Generally, unformatted records use less space than formatted records and thus conserve storage space.

An unformatted record can be read from or written to only by unformatted I/O statements. Unformatted records are measured in bytes. That length is primarily a function of the output list used to write the record but may be limited by the external storage media or the CPU. An unformatted record can be empty.

Endfile Records

An *endfile record* marks the logical end of a data file. Thus, it can only be the last record of a file. An endfile record does not contain data and has no length. An endfile record is written by an **ENDFILE** statement.

When a program is compiled with **-vms_endfile**, an endfile record consists of a single character, Control D. In this case, several endfile records can exist in the same file and can be anywhere in the file. Reading an endfile record will result in an end−of−file condition being returned, but rereading the same file will read the next record, if any.

I/O Statements

The I/O statements that Fortran uses to transfer data can be categorized by how the data translated during the transfer, namely, as *formatted*, *list−directed*, and *unformatted I/O.*

Unformatted Statements

An unformatted I/O statement transfers data in the noncharacter format during an I/O operation. Unformatted I/O operations are usually faster than formatted operations, which translate data into character format.

In processing formatted statements, the system interprets some characters, for example, the line−feed character, as special controls and eliminates them from input records. Therefore, unformatted statements must be used when *all* characters in a record are required.

The absence of a format specifier denotes an unformatted data transfer statement, as shown by the **WRITE** statement in the following example:

```
    program MakeIndex
    character*12  word
    open (2, file='v',form='formatted')
    open (unit=10, status='new', file='newv.out",
   +       form='unformatted')
116 read (2,666, end=45) word
    write (10) word
    go to 116
45  close (10)
end
```

In the above example, formatted records are read into the variable *word* from the input file attached to unit 2 and then written unformatted to the output file attached to unit 10.

Formatted Statements

A *formatted I/O* statement translates all data to character format during a record transfer. The statement contains a *format specifier* that references a **FORMAT** statement; the **FORMAT** statement contains descriptors that determine data translation and perform other editing functions. Here is an example of two formatted **WRITE** statements:

```
    program makeindex
    character*18  message
    message = 'Hello world'
    write (6,100) message
    write (6,100) 'hello world'
100 format (a)
end
```

Note that both statements contain the format specifier 100, which references a format statement with an *A* character descriptor. (Chapter 9, "Format Specification," describes the descriptors in detail.) Both statements perform the same function, namely, writing the following message to the unit 6 device:

```
HELLO WORLD
```

List−Directed Statements

An I/O statement is *list directed* when an asterisk is used in place of a format specifier. A list−directed I/O statement performs the same function as a formatted statement. However, in translating data, a list−directed statement uses the declared data type rather than format descriptors in determining the format.

The following two list−directed **WRITE** statements perform the same function as the formatted **WRITE** statements in the example for formatted output above.

```
program makeindex
character*18  message
message = 'hello world'
write (6,*) message
write (6,*) 'hello world'
end
```

In this example, the variable *message* in the first **WRITE** statement determines that output is in character format; the character constant `Hello World` in the second statement makes this determination.

Files

A file is a sequence of records. The processor determines the set of files that exists for each executable program. The set of existing files can vary while the program executes. Files that are known to the operating system do not necessarily exist for an executable program at a given time. A file can exist and contain no records (all files are empty when they are created). I/O statements can be applied only to files that exist.

Files that have names are called *named files.* Names are simply character strings.

Every data file has a position. The position is used by I/O statements to tell which record to access and is changed when I/O statements are executed.

The terms used to describe the position of a file are

Initial point The point immediately before the first record.

Terminal point The point immediately after the last record.

Current record The record containing the point where the file is positioned. There is no current record if the file is positioned at the initial point (before all records) or at the terminal point (after all records) or between two records.

Preceding record

The record immediately before the current record. If the file is positioned between two records (so there is no current record), the preceding record is the record before the file position. The preceding record is undefined if the file is positioned in the first record or at the initial point.

Next record The record immediately after the current record. If the file is positioned between two records (so there is no current record), the next record is the record after the file position. The next record is undefined if the file position is positioned in the last record or at the terminal point.

This section discusses the two kinds of files: internal files and external files.

External Files

An *external file* is a set of records on an external storage medium (for example, a disk or a tape drive). A file can be empty, which means it can contain zero records.

Internal Files

An *internal file* is a means of transferring data within internal storage between character variables, character arrays, character array elements, or substrings.

An internal file is always positioned at the beginning of the first record before data transfer. Records are read from and written to by sequential access of formatted I/O statements only.

The following simple example shows how to use an internal file transfer to convert character and integer data.

```
program conversion
character*4   CharRep
integer       NumericalRep
```

```
NumericalRep = 10
C
C    example 1
C
     write (CharRep, 900) NumericalRep
900  format (i2)
     CharRep = '222'
C
C    example 2
C
     read (CharRep, 999) NumericalRep
999  format (i3)
     end
```

In the first example, the contents of **NumericalRep** are converted to character format and placed in **CharRep**. In the second example, the contents of **CharRep** are converted to integer format and placed in **NumericalRep**.

Methods of File Access

The following methods of file access are supported:

- sequential
- direct
- keyed

External files can be accessed using any of the above methods. The access method is determined when the file is opened or defined.

Fortran 77 requires that internal files must be accessed sequentially.

As an extension, the use of internal files in both formatted and unformatted I/O operations is permitted.

Sequential Access

A file connected for *sequential access* has the following properties:

- For files that allow only sequential access, the order of the records is simply the order in which they were written.
- For files that also allow direct access, the order of files depends on the record number. If a file is written sequentially, the first record written is record number 1 for direct access, the second written is record number 2, and so on.
- Formatted and unformatted records cannot be mixed within a file.
- The last record of the file can be an endfile record.
- The records of a pure sequential file must not be read or written by direct−access I/O statements.

Direct Access

A file connected for *direct access* has the following properties:

- A unique *record number* is associated with each record in a direct−access file. Record numbers are positive integers that are attached when the record is written. Records are ordered by their record numbers.
- Formatted and unformatted records cannot be mixed in a file.
- The file must not contain an endfile record if it is direct access only. If the file also allows sequential access, an endfile record is permitted but will be ignored while the file is connected for direct access.
- All records of the file have the same length. When the record length of a direct−formatted file is one byte, the system treats the files as ordinary system files, that is, as byte strings in which each byte is addressable. A **READ** or **WRITE** request on such files consumes/produces bytes until satisfied, rather than restricting itself to a single record. Note that to produce a record length of one byte, the program must be compiled with the **-old_rl** option.
- Only direct−access I/O statements can be used for reading and writing records. An exception is made when sequential I/O statements are used on a direct−unformatted file, in which case the next record is assumed. List−directed formatting is not permitted on direct−access files.
- The record number cannot be changed once it is specified. A record can be rewritten, but it cannot be deleted.
- Records can be read or written in any order.

Keyed Access

A file connected for *keyed access* has the following properties:

- Only files having an indexed organization can be processed using the keyed−access method.
- A unique character or integer value called a *key* is associated with one or more fields in each record of the indexed access file. The fields are defined when the file is created with an **OPEN** statement. Each **READ** statement contains a key to locate the desired record in the indexed file.
- You can intermix keyed access and sequential access on the same opened file.

Units

Files are accessed through *units.* A unit is simply the logical means for accessing a file. The file−unit relationship is strictly one to one: files cannot be connected to more than one unit and vice versa. Each program has a processor−dependent set of existing units. A unit has two states: connected and disconnected.

Connection of a Unit

A *connected unit* refers to a data file. A unit can be connected implicitly by the processor or explicitly by an **OPEN** statement. If a unit is connected to a file, the file is connected to the unit. However, a file can be connected and not exist. Consider, for example, a unit preconnected to a new file. A *preconnected unit* is already connected at the time the program execution begins. See the section on preconnected files in Chapter 1 of the *Fortran 77 Programmer's Guide* for these default connections.

Disconnection of a Unit

A unit can be *disconnected* from a file by a **CLOSE** statement specifying that particular unit.