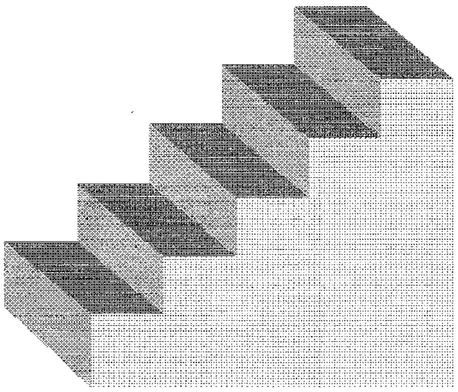

PROGRAMMING IN FORTRAN

Fourth Edition

Ömer Akgiray

e-mail:

omer.akgiray@marmara.edu.tr

CHAPTER 6: *FILES AND MORE ON FORMATS*

6.1 Records and Files

Occasionally it may be desirable to read input data from an input data file, or to write output data into an output file. Using an input data file, for example, frees the programmer from having to continually re-enter the same data while testing and debugging a program. Using an output file enables the programmer save the results of a program execution. Such an output file may later be displayed on the screen, printed on paper, or used as input to another program.

Before discussing the details of file input/output, it may be useful here to briefly describe certain terms related to files in FORTRAN. A **record** is a defined sequence of characters or values. There are three types of records in FORTRAN: **formatted records, unformatted records**, and **end-of-file records**. These terms will be explained subsequently. A sequence of records forms a **file**, of which there are two types: external and internal. An **external file** is a file stored on some external medium (e.g. a hard disk). An **internal file** is a character variable, a character array element, a character substring, or a character array that is processed as if it were an external file by using the `READ` and `WRITE` statements. Thus, an internal file is not actually a file but is treated as one. Internal files will be discussed in Section 6.11.

A file may consist of formatted records and, optionally, one end-of-file record, or it may consist of unformatted records and, optionally, one end-of-file record. In the first case, the file is referred to as a **formatted file**; in the latter case it is an **unformatted file**. A file cannot contain both formatted and unformatted records.

External files can be further classified as **sequential files** and **direct access files**. A sequential file consists of records that must be processed sequentially, starting with the first one. Each record is written after the previously written record, and records are read in the same order as they were written. The records of a direct access file, on the other hand, can be written and read in any order. Direct file access is also referred to as **random access**.

6.2 Connecting External Files: The `OPEN` Statement

Prior to writing into or reading from an *external file*, the file must first be connected to the program by an **`OPEN` statement**, which has the general form

```
OPEN(UNIT  = unit_num,
     FILE  = file_name,
     STATUS  = file_stat,
     ACCESS  = access_type,
     FORM  = format_mode,
     RECL  = record_length,
     IOSTAT  = io_status,
     BLANK  = blank_mode,
     ERR  = err_label)
```

The file specifiers `UNIT`, `FILE`, `STATUS`, etc. describe the properties of the file being connected. The `UNIT` specifier must always be present (although the keyword `UNIT` may be omitted: see below), while all the other specifiers are optional.

unit_num is an integer expression with a non-negative value less than 100. The value of *unit_num* is used in all I/O (`READ` or `WRITE`) statements which are to use that file. If the keyword `UNIT` is omitted, *unit_num* must be the first parameter.

file_name is a character expression indicating the name of the file. Acceptable values for *file_name* are dictated by the operating system being used. For example, with many versions of MS-DOS, a file can have a root name with at most 8 characters and an extension containing 3 characters.

The value of *file_stat* must be one of the strings `'NEW'` (for a new output file), `'OLD'` (for an existing file), `'SCRATCH'` (for an output file that will not be retained after the execution of the program is complete), or `'UNKNOWN'`. If omitted, `'UNKNOWN'` is assumed. If the `STATUS` specifier is set equal to `'SCRATCH'`, the file should *not* be given a name. If *file_stat* is `'OLD'` then the file *must* already exist, whereas if it is `'NEW'` it *must not* already exist. An attempt to open an existing file with `STATUS = 'NEW'`, for example, will fail. The interpretation of `UNKNOWN` status is compiler-dependent. On most systems, however, a file will be treated as `NEW` if it does not exist; it will be treated as `OLD` if it already exists. Thus the `UNKNOWN` status may be used to avoid an execution-time error due to opening an existing file with `STATUS = 'NEW'` or opening a non-existent file with `STATUS = 'OLD'`.

The value of *access_type* is either `'SEQUENTIAL'` or `'DIRECT'`. If `ACCESS` specifier is omitted, `'SEQUENTIAL'` is assumed.

The value of *format_mode* is either `'FORMATTED'` or `'UNFORMATTED'`. If the `FORM` specifier is omitted, `'FORMATTED'` is assumed for sequential files; `'UNFORMATTED'` for direct access files.

record_length is an integer expression that specifies the length of each record in a direct access file. Note that the RECL specifier is used only when ACCESS = 'DIRECT'.

io_status is an integer variable that is set to zero if no error occurs during the execution of the OPEN statement; it will be set to a compiler-dependent positive value to indicate that there was an error during the file connection process. An error may occur, for example, if the named file does not exist or is of the incorrect type.

The BLANK specifier establishes the interpretation of blanks in numeric fields for all input statements on that file. *blank_mode* is a character expression that evaluates to 'NULL' or 'ZERO'; it is set to 'NULL' to ignore blanks. Set *blank_mode* to 'ZERO' to treat blanks as zeros. If omitted, 'NULL' is assumed. The BLANK specifier is applicable only to formatted input (i.e. only to input files connected with FORM = 'FORMATTED'). The BLANK specifier establishes the interpretation of blanks in numeric fields for all input statements on the file being connected. It is also possible to specify the interpretation of blanks on a record-by-record basis by employing a BN or BZ edit descriptor.

The ERR specifier causes a transfer of control to the statement labeled *err_label* if an error occurs. Since the IOSTAT specifier can be utilized to detect an error and to take subsequent corrective action, it may not be necessary to use the ERR specifier. As a matter of fact, to minimize the appearance of labeled statements in FORTRAN programs, many programmers recommend the use the IOSTAT specifier instead of the ERR specifier to handle errors.

6.3 The READ and WRITE Statements

It may be difficult at this point to completely understand the meaning of some of the file specifiers described above[1]. You will need to study the example programs given later in this chapter to better appreciate the application of the OPEN statement and the various specifiers. Before presenting complete example programs, however, general forms of the READ and WRITE statements need to be described:

[1] Fortran 90 includes two additional file specifiers. These take the forms ACTION = *allowed_actions* and POSITION = *file_position*. Here *allowed_actions* is a character expression that evaluates to 'READ', 'WRITE', or 'READWRITE', whereas *file_position* is a character expression that must take one of the values 'REWIND', 'APPEND', or 'ASIS'.

```
READ(UNIT  = unit_spec,
     FMT  =  fmt_spec,
     END  =  end_label,
     ERR  = err_label,
     IOSTAT  =  io_status,
     REC  = rec_number) input_list

WRITE(UNIT  = unit_spec,
     FMT  =  fmt_spec,
     ERR  = err_label,
     IOSTAT  =  io_status,
     REC  = rec_number) output_list
```

where *unit_spec* specifies the file on which the READ or the WRITE operation is performed; the parameter *unit_spec* is usually a unit number (a non-negative integer); it may also be the name of an internal file. If the keyword UNIT is omitted, *unit_spec* must be the first parameter.

fmt_spec is the label of the FORMAT statement to be used in reading or writing, or the format specification itself. An asterisk (*) used as *fmt_spec* indicates list-directed input/output. Note that the FMT specifier is used with formatted files only. If the keyword FMT is omitted when reading from or writing to a formatted file, FORTRAN assumes that the second parameter is *fmt_spec*. Otherwise, all the parameters can be typed in any order. For clarity, however, it is recommended that you always include the keywords UNIT and FMT.

The optional parameter *io_status* is an integer variable. After the READ or WRITE statement is executed, the value of *io_status* will be set to zero to indicate that no errors occurred. It will be set equal to a compiler-dependent positive value to indicate that there was an error during input/output. When reading a file, *io_status* will be set to a compiler-dependent negative value if an *end-of-file record* is encountered.

The ERR specifier causes a transfer of control to the statement labeled *err_label* if an error occurs during reading or writing. The END specifier is similar, but can be used only with a READ statement. It causes a transfer of control to the statement labeled *end_label* if an end-of-file record is encountered when reading a file. This specifier can be used only with sequential files. Note that the IOSTAT specifier can be used instead of the ERR and END specifiers: *io_status* will be negative when an *end-of-file record* is encountered; it will be positive if an error occurs during input/output.

The `REC` specifier is used only when reading or writing to a direct access file. *rec_number* is an integer expression with a positive value which specifies the record number of the record to be read or written.

An asterisk, `*`, when used as *unit_spec* in a `READ` statement, designates the **default input unit**. An asterisk used as *unit_spec* in a `WRITE` statement, on the other hand, designates the **default output unit**. In most computer systems, such as personal computers and workstations, the mentioned default units will be the keyboard and the display screen, respectively. Furthermore, the default input unit is usually preconnected as unit 5, whereas the default output unit is unit 6[2]. Note that the default units will always be *preconnected* to a program, and there is no need to connect them by means of an `OPEN` statement. With most FORTRAN systems, an `OPEN` statement can be employed to connect a file with units 5 or 6. Once this is done, the relevant unit number will refer to the file in question (and, not to the default I/O unit) in that program. The asterisk (`*`), however, is permanently connected to the default I/O units, and an `OPEN` statement cannot change this.

It should be noted that the `PRINT` statement always sends it output to the default output unit. The `WRITE` statement is more flexible in that it can be used to print to other units (i.e. to internal and external files).

It follows from the information given so far that all of the statements

```
WRITE(*,*) output list
WRITE(UNIT=*,FMT=*) output list
WRITE(6,*) output list
WRITE(UNIT=6,FMT=*) output list
```

are equivalent to

```
PRINT*, output list
```

Similarly, the input statements

```
READ(*,*) input list
READ(UNIT=*,FMT=*) input list
READ(5,*) input list
READ(UNIT=5,FMT=*) input list
```

are equivalent to

```
READ*, input list
```

[2] This is compiler-dependent. On some FORTRAN systems, the default input and output units may be pre-connected with unit numbers 1 and 2, respectively. Unit number 0 may also be used for the default input unit. You can experiment with your system to find out about the default unit numbers.

The statement

```
        WRITE(*, 11) output list
```

indicates that the output will be displayed on the default output unit using the format specified by the FORMAT statement labeled 11. As such it is equivalent to

```
        PRINT 11, output list
```

which also prints *output list* on the default output unit. These last two statements are used for user-formatted output. User-formatted input (i.e. input editing) is also possible, and it will be described later in this chapter.

To read a line of data from a file with the unit number 15, one of the following statements could be used:

```
        READ(15, *) input list
        READ(UNIT=15,FMT=*) input list
```

To write a line of data into unit number 7 using the FORMAT statement labeled 23, one of the following two statements may be used:

```
        WRITE(7, 23) output list
        WRITE(UNIT=7,FMT=23) output list
```

6.4 The END FILE, BACKSPACE, and REWIND Statements

After the WRITE operation on a sequential file is completed, the **END FILE statement** should be used to mark the end of the file by writing a special **end-of-file record** at the end of the file. This statement has the following general form:

```
        END FILE(UNIT=unit_spec, ERR=err_label, IOSTAT=io_status)
```

where the UNIT, ERR, and IOSTAT specifiers are the same as those already described for use with the WRITE statement. For example, if the keyword UNIT is omitted, *unit_spec* must be the first parameter. The parameters can otherwise be typed in any order. The ERR and IOSTAT specifiers are optional. No data may be written following the end-of-file record.

When an end-of-file record is read by a READ statement, it will cause an **end-of-file condition** which can be detected by an IOSTAT or an END specifier in the READ statement. If not detected in this way an execution-time error may occur.

It was noted earlier that the records of a *sequential file* must always be processed serially. Consider, for example, the problem of writing additional records to the end of an existing sequential file (without deleting the existing records). The file is first connected to the program by an OPEN statement. When the file is opened, it will normally be positioned just before the first record in the file.(To ensure this the

REWIND statement may be used. See below.) A WRITE statement executed at this point will, therefore, overwrite the existing first record. A very important rule to remember is that *writing a record to a sequential file destroys all information in the file after that record.* (If it is required to overwrite individual records selectively within a file then the file must be opened for *direct access*.) To avoid overwriting and destroying the existing records, therefore, a sequential file must first be positioned just before the end-of-file record (and after all the data records). This can be accomplished by reading all the existing records in the file before writing new records. The first READ statement will read the very first record. The file is then positioned just before the second record, and thus the next READ statement will read the second record. As additional READ statements are executed, the file position pointer moves sequentially towards the end of the file, the final record that can be read being the end-of-file record. Recall that it is possible to detect the end-of-file record by means of a END or IOSTAT specifier used in a READ statement. Once the end-of-file record is read, the file will be positioned immediately after the end-of-file record. Since it is not possible to read or write beyond the end-of-file record, the file should be repositioned just before the end-of-file record before writing new records. This can be accomplished by using the **BACKSPACE statement** which has the following general form:

BACKSPACE(UNIT = *unit_spec*, ERR = *err_label*, IOSTAT = *io_status*)

In general, this statement causes the file to be positioned just before the preceding record. If the file position pointer is past the end-of-file record, it is repositioned just before the end-of-file record. The BACSPACE statement thus enables the program to read or overwrite the previous record. Here the UNIT, IOSTAT, and ERR specifiers are the same as those already described for use with the WRITE statement. The ERR and IOSTAT specifiers are optional.

The other file-positioning statement is the **REWIND statement**, which causes a file to be positioned just before its first record. A subsequent READ statement will start reading the file from the beginning. The general form is as follows:

REWIND(UNIT = *unit_spec*, ERR = *err_label*, IOSTAT = *io_status*)

Here again the ERR and IOSTAT specifiers are optional. It should be remembered that if a program has either read or written the end-of-file record in a *sequential* file, it cannot read or write any more records until either a BACKSPACE or a REWIND statement has repositioned the file to a point before the end-of-file record.

6.5 The CLOSE Statement

Once the READ or WRITE operation on an external file is completed, the file should be disconnected from the program using the **CLOSE statement**:

```
      CLOSE(UNIT   = unit_spec,
            ERR    = err_label,
            IOSTAT = io_status,
            STATUS = file_status)
```

where the UNIT, IOSTAT, and ERR specifiers take their usual form. The ERR, IOSTAT, and STATUS specifiers are optional. The STATUS specifier can be used to specify what will happen to the file after it is closed: *file_status* is a character expression which is either 'KEEP' or 'DELETE'. If omitted, 'DELETE' is assumed for scratch files, 'KEEP' is default for all other files. Setting STATUS='KEEP' for a scratch file is not permitted.

6.6 Sequential and Formatted External Files: Examples

The INQUIRE, OPEN, REWIND, BACKSPACE, END FILE, and CLOSE statements are executable statements, and as such they are placed within other executable statements of a program. Several examples illustrating the use of these statements in association with sequential and formatted files are presented in this section. It is recommended that you read the previous sections of this chapter again after you study these example programs.

Example 6.1:

Consider the calculation of the sum *1+2+...+i = i(i+1)/2*. The following program reads in the largest integer N for which this sum will be calculated. The program then calculates the sum for all values of *i* between 1 and N. The results are written into an output file named SUMS.OUT. Note the application of the OPEN, READ, WRITE, END FILE, REWIND, and CLOSE statements in this program.

```
      PROGRAM SUMN
C*************************************************************************
C   Prints I and the sum 1+2+...+I for all I less than or equal to
C   a user-entered integer N into an output file named by the user.
C*************************************************************************
      IMPLICIT NONE
      INTEGER N, I, ISUM, IOS
```

```
C   Prompt the user, read input
      PRINT*, 'Program to calculate 1+2+...+I for all I .LE. N.'
      PRINT*, 'Enter upper limit N (a positive integer): '
      READ*, N
C   Open output file
      OPEN(UNIT = 1, FILE = 'SUMS.OUT', FORM = 'FORMATTED',
     *     ACCESS = 'SEQUENTIAL', STATUS = 'NEW', IOSTAT = IOS)
      IF(IOS.NE.0)THEN
          PRINT*, 'An error occurred while opening output file.'
          PRINT*, 'Execution will have to be terminated.'
          STOP
      ENDIF
C   Write sums 1+2+...+I for all I .LE. N into output file
      DO 1 I= 1, N
          WRITE(UNIT = 1, FMT = 10, IOSTAT = IOS) I, I*(I+1)/2
 10       FORMAT(2I5)
          IF(IOS.NE.0) PRINT*, 'Error when writing line I =', I
  1   CONTINUE
      END FILE(UNIT = 1)
      PRINT*, 'Summation calculation and output generation completed.'
C   Echo print the contents of the output file on the screen
      REWIND(UNIT = 1)
      PRINT*, 'SUMS.OUT contains the following:'
      DO WHILE(.TRUE.)
          READ(UNIT = 1, FMT = *, END = 2, IOSTAT = IOS) I, ISUM
          IF(IOS.GT.0) PRINT*, 'Error when reading line I =', I
          IF(IOS.EQ.0) PRINT 10, I, ISUM
      END DO
  2   CLOSE(UNIT = 1)
      END
```

Notice the specifiers used in the OPEN statement. Recall that the RECL specifier is relevant only for direct access files. Since the output file is a sequential access file in this case, the RECL specifier is omitted. The BLANK specifier is omitted because it is applicable to user-formatted *input* only.

Note that the ERR specifier is omitted as well. Instead of using the ERR specifier, the IOSTAT specifier and the integer variable IOS are employed by this program for error detection. If an error occurs when executing the OPEN statement, IOS will be assigned a non-zero value. A program can detect an error during file connection and take appropriate action by utilizing the IOSTAT specifier as illustrated here. The IOSTAT specifier can be used in a similar manner with the READ, WRITE, END FILE, CLOSE, REWIND, BACKSPACE, and INQUIRE statements.

An error may occur during the execution of an OPEN statement, for example, if a file named SUMS.OUT already exists in the working directory as this would contradict the specification STATUS = 'NEW' (you can observe this by running the above program twice). The INQUIRE statement can be used to check whether or not a file with the specified name exists (see Example 6.4). An alternative that works on most compilers is to set STATUS = 'UNKNOWN' (see Exercise 7 later in this chapter).

The ACCESS specifier can actually be omitted in this example since 'SEQUENTIAL' is assumed by default. The FORM specifier could also be omitted here because 'FORMATTED' is the default for sequential files. For clarity, however, it is good practice to explicitly specify the properties of a file by including all the applicable specifiers in the OPEN statement.

Consider next the WRITE statement. There are five possible specifiers that can be used in a WRITE statement. Of these, the REC (record number) specifier is relevant only for direct access files, and is therefore omitted in the above program. The ERR specifier is omitted because the IOSTAT specifier is used for error detection.

The ERR and IOSTAT specifiers are omitted in the ENDFILE, REWIND, and CLOSE statements in the above program. This is done here for simplicity. As an exercise, modify the program by adding the IOSTAT specifier to the mentioned statements.

Example 6.2:

Consider again the linear-curve fit problem (see Example 3.5). PROGRAM LSQUAR of Chapter 3 is rewritten here as a subroutine. Note that the subroutine takes input data as arguments, carries out the required calculations, and returns the results via output arguments. That is, no input or output operations are performed within the subroutine. As a result, this same subroutine can be used regardless of how the input data are read.

```
      SUBROUTINE LSQUAR(X, Y, N, M, B)
C***************************************************************************
C  Subroutine finds the equation of the least squares line
C  for a set of data points. Variables used are:
C  X, Y     : (X(i),Y(i)) is an observed data point (i=1,...,N)
C  N        : number of data points
C  M        : slope of the line  Y = mX + b
C  B        : Y-intercept of the line Y = mX + b
C  SUMX     : sum of X's
C  SUMY     : sum of Y's
C  SUMX2    : sum of the squares of X's
C  SUMXY    : sum of the products X*Y
C  XMEAN    : mean of the X's
C  YMEAN    : mean of the Y's
C***************************************************************************
      IMPLICIT NONE
C   Dummy arguments
      INTEGER N
      REAL X(N), Y(N), M, B
C   Local variables
      INTEGER I
      REAL SUMX, SUMY, SUMX2, SUMXY, XMEAN, YMEAN
C  Initialize the sums to 0
      SUMX = 0.
      SUMXY = 0.
      SUMX2 = 0.
      SUMY = 0.
C  Calculate the necessary sums
      DO 1 I = 1, N
          SUMX = SUMX + X(I)
          SUMY = SUMY + Y(I)
          SUMXY = SUMXY + X(I)*Y(I)
          SUMX2 = SUMX2 + X(I)*X(I)
   1  CONTINUE
C  Calculate slope and intercept
      XMEAN = SUMX/N
      YMEAN = SUMY/N
      M = (SUMXY - SUMX*YMEAN) / (SUMX2 - SUMX*XMEAN)
      B = YMEAN - M*XMEAN
      RETURN
      END
```

Suppose that the input data are present in a file named XY.DAT. (Such a file might be the output of another program, or it could be prepared using a text editor.) We shall assume that the first line of XY.DAT contains a single value, namely, the number data points (i.e., number of X-Y pairs). Each of the second and subsequent lines contains a data point (a pair of real numbers). An example is provided below:

```
5
-1.0  -1.0
1.0  3.0
2.5  6.0
7.0  15.0
3.0  7.0
```

`PROGRAM LINE` given below shows how data can be read from such a file.

```
      PROGRAM LINE
C***************************************************************************
C   Main program reads input data, calls LSQUAR to find the
C   best fitting straight line to the data, prints output.
C***************************************************************************
      IMPLICIT NONE
      INTEGER Nmax
      PARAMETER (Nmax = 100)
      INTEGER I, N, IOS
      REAL X(Nmax), Y(Nmax), M, B
      OPEN(UNIT = 1, FILE = 'XY.DAT', STATUS = 'OLD', IOSTAT = IOS)
      IF(IOS.NE.0) STOP 'Error while connecting input file.'
      READ(1, *, IOSTAT = IOS) N
      IF(IOS.NE.0) STOP 'Error while reading number of data points.'
      DO 1 I = 1, N
          READ(1, *, IOSTAT = IOS) X(I), Y(I)
          IF(IOS.NE.0) STOP 'Error while reading X-Y pair.'
    1 CONTINUE
      CLOSE(UNIT = 1)
      CALL LSQUAR(X, Y, N, M, B)
      PRINT 5, M, B
    5 FORMAT(1X, 'Equation of least squares line is Y = mX + b.',/,
     %       1X, 'Slope = m =', F5.2, /, 1X, 'Y-intercept = b =', F5.2)
      END
```

The following is the output of `PROGRAM LINE` using the input file given above:

```
Equation of least squares line is Y = mX + b.
Slope = m = 2.00
Y-intercept = b = 1.00
```

Note that the keywords `UNIT` and `FMT` are omitted in the `READ` statements. When this is done, the first and the second parameters must be *unit_spec* (1 in this example) and *fmt_spec* (* in this case), respectively.

Notice also how the `IOSTAT` specifier is employed for error detection. If a data file named `XY.DAT` is not present in the directory you run this program; for example, the value of `IOS` will be nonzero after the execution of the `OPEN` statement.

Exercises:

1. When a text editor program is used to generate a data file, an end-of-file record is automatically inserted by the computer following the last line of data. Utilizing this fact and the `END` control specifier, it is possible read such a file without knowing the number of lines in advance. To demonstrate this, modify `PROGRAM LINE` by replacing the program section

```
      READ(1, *) N
      DO 1 I = 1, N
          READ(1, *) X(I), Y(I)
    1 CONTINUE
```

by the following statements:

```
      DO WHILE(.TRUE.)
          READ(1, *, END = 10) X(I), Y(I)
      END DO
   10 CONTINUE
```

Delete the first line (which contains the number of data points) from `XY.DAT`. Compile and run the program again after these modifications. As an alternative, try also the following version:

```
      DO WHILE(.TRUE.)
          READ(1, *, IOSTAT = IOS) X(I), Y(I)
          IF(IOS .LT. 0) GO TO 10
      END DO
   10 CONTINUE
```

Here `IOS` is an integer variable.

2. To obtain a table of the printable ASCII characters, type and run the following program. The first 32 characters i.e. the characters with decimal codes between 0 and 31 are skipped by the program. To type the IBM PC character '│' (within a DOS editor), hold down the ALT key and then type the number 179 using your numeric keypad. If you view the file from within a Windows editor, this character will appear as '³' (superscript 3). If you are typing the FORTRAN program using a windows editor, type '³' (which has the decimal code 179 in the Windows character set: use the ALT+0179 key combination to type it) instead of '│' (which has the decimal code 179 in the IBM PC character set). All this assumes that your processor employs an extension of the ASCII character set (see the *Remark* following Example 5.2).

```
      PROGRAM ASCII
C**************************************************************************
C   Program writes the printable US-ASCII characters to a file
C**************************************************************************
      IMPLICIT NONE
      INTEGER I, IOS
      OPEN(UNIT = 10, FILE = 'USASCII.CHR', IOSTAT = IOS)
      IF(IOS.NE.0) STOP 'Error while opening file.'
      DO 100 I = 32, 63
          WRITE(10,1) I, CHAR(I), I+32, CHAR(I+32), I+64, CHAR(I+64)
          WRITE(10,2)
  100 CONTINUE
      CLOSE(UNIT = 10)
    1 FORMAT(3(1X, I4, 1X, A, 1X, '│'))
    2 FORMAT(3(8X,'│'))
      END
```

You can view the output file `USASCII.CHR` by either typing "*edit ascii.chr*" or "*type ascii.chr*" at the DOS command prompt. The US-ASCII characters should be displayed correctly within a Windows editor as well (why?), but the line drawing character '│' used to separate the columns of the resulting table may appear as '³' (why?)

Example 6.3:

The following program uses `SUBROUTINE CBUBBLE` (cf. Chapter 5) for sorting a list of character strings. To test the program, prepare a file named `UNSORTED`. Type a word on each line of the file. The output file `SORTED` will contain the same words, sorted in ascending order.

```
      PROGRAM CORDER
C***********************************************************************
C   Program reads a character array CARR from a file named UNSORTED.
C   The elements of CARR are put into alphabetical order by CBUBBLE.
C   The sorted array CARR is written into a file named SORTED.
C***********************************************************************
      IMPLICIT NONE
      INTEGER NMAX, IOS, I, N
      PARAMETER (NMAX = 200)
      CHARACTER*40 CARR(NMAX), CWORK
      CHARACTER*1 CH
      OPEN(UNIT = 1, FILE = 'UNSORTED', STATUS = 'OLD',
     *     ACCESS = 'SEQUENTIAL', FORM = 'FORMATTED', IOSTAT = IOS)
      IF(IOS.NE.0) STOP 'Error opening input file.'
      DO 1 I = 1, NMAX
          READ(UNIT = 1, FMT = '(A)', END = 10, IOSTAT = IOS) CARR(I)
          IF(IOS.NE.0) STOP 'Error while reading input file.'
    1 CONTINUE
      PRINT 100, NMAX, NMAX
  100 FORMAT(' Read maximum number (', I4, ') records allowed'/
     *       ' before reaching end of file. The file may contain'/
     *       ' additional records. This may be a partial sort.'/
     *       ' Type S to sort the', I4, ' values read.')
      READ(UNIT = *, FMT = '(A1)') CH
      IF(CH .NE. 'S' .AND. CH.NE.'s') STOP 'Execution terminated.'
   10 CONTINUE
      CLOSE(UNIT = 1)
      N = I - 1
      CALL CBUBBLE(N, CARR, CWORK)
      OPEN(UNIT = 2, FILE = 'SORTED', STATUS = 'NEW',
     *     ACCESS = 'SEQUENTIAL', FORM = 'FORMATTED', IOSTAT = IOS)
      IF(IOS.NE.0) STOP 'Error opening output file.'
      DO 2 I = 1, N
          WRITE(UNIT = 2, FMT = '(A)', IOSTAT = IOS) CARR(I)
          IF(IOS.NE.0) STOP 'Error while writing output file.'
    2 CONTINUE
      ENDFILE(UNIT = 2)
      CLOSE(UNIT = 2)
      END
```

6.7 The INQUIRE Statement

There are occasions when it is helpful to find out the properties of a file during the execution of a program. This can be done using the **INQUIRE statement**. Most frequently this statement is employed to find out whether or not a file specified by the user exists. If an attempt is made to open a file that does not exist, an error will result. (If this error is not detected using either the ERR specifier or the IOSTAT specifier, execution of the program will terminate prematurely.) To prevent such an error and to make it possible for the user to re-enter a file name, the INQUIRE statement can be utilized. Consider, as an example, the following program segment:

```
1 PRINT*, 'Enter file name within apostrophes: '
  READ*, FNAME
  INQUIRE(FILE = FNAME, EXIST = FOUND)
  IF(.NOT.FOUND)THEN
      PRINT *, FNAME//' does not exist.'
      GO TO 1
  ELSE IF(FOUND)THEN
      OPEN(UNIT = 1, FILE = FNAME, STATUS = 'OLD')
  ENDIF
```

Here it is assumed that FOUND has been declared as a logical variable, and FNAME is a character variable of an appropriate length. After the execution of the INQUIRE statement, FOUND will have the value .TRUE. if a file with the name specified by the user (and stored in FNAME) exists; it will be .FALSE. if such a file does not exist.

An INQUIRE statement is either an **inquire-by-file** or an **inquire-by-unit**. Note that the program segment given above employs an inquire-by-file. An inquire-by-unit statement, on the other hand, includes the UNIT specifier. Exactly one (and *not* both) of the UNIT and FILE specifiers must appear in an INQUIRE statement. An INQUIRE statement may also include an IOSTAT or ERR specifier in the same form as in an OPEN statement. If the keyword UNIT is omitted, *unit_spec* must be the first parameter. Otherwise, the parameters may appear in any order. The most general form of the INQUIRE statement is as follows:

```
INQUIRE(UNIT = unit_spec,
        FILE = file_name,
        EXIST = file_existence,
        OPENED = open_status,
        NAMED =name_status,
        NAME = fname,
        NUMBER = unit_number,
        NEXTREC =record_number,
        ACCESS = access_type,
        SEQUENTIAL = yes_or_no,
        DIRECT = yes_or_no,
        FORM = format_mode,
        FORMATTED = yes_or_no,
        UNFORMATTED = yes_or_no,
        RECL = record_length,
        IOSTAT = io_status,
        BLANK = blank_mode,
        ERR = err_label)
```

The UNIT, FILE, ERR, and IOSTAT specifiers are used as described before for the OPEN statement. *io_status,* for example, is an integer variable that returns zero if the INQUIRE statement executes without any error.

file_existence, open_status, and *name_status* are logical variables. The OPENED, EXIST, and NAMED specifiers can be used in either an inquire-by-unit or an inquire-by-file, although the NAMED specifier is normally used in an inquire-by-unit operation only.

After the execution of the INQUIRE statement, *file_existence* will be .TRUE. if the specified file or unit exists; it will be .FALSE. otherwise.

open_status returns .TRUE. if the specified file is connected (i.e. open) at the time of an inquire-by-file; returns .FALSE. otherwise. In an inquire-by-unit operation, *open_status* returns .TRUE. if there is an open file connected with the specified unit; returns .FALSE. otherwise.

In an inquire-by-unit operation, *name_status* returns .TRUE. if the file connected to *unit_spec* has a name (i.e. it is not a scratch file); returns .FALSE. if the file is a scratch file or if there is no file connected with the specified unit. In an inquire-by-file operation, *name_status* returns .TRUE. if the file is open; returns .FALSE. otherwise.

fname, access_type, yes_or_no, format_mode, and *blank_mode* are character variables. Note that the values of these variables are defined by the execution of the INQUIRE statement, so that they give information about the properties of the file/unit in question. *file_name,* on the other hand, is a character expression that already has a value before the execution of the INQUIRE statement (in case of an inquire-by-file). Similarly, in case of an inquire-by-unit, *unit_spec* is an integer expression that has a fixed value before the execution of the INQUIRE statement. An asterisk (*) may also be used as *unit_spec.*

fname is a character variable used in conjunction with the NAME specifier. In an inquire-by-unit operation, *fname* returns the name of the file connected to *unit_spec*. If the file does not have a name, then *fname* is not defined. In an inquire-by-file operation, it returns the value of *file_name.*

unit_number is an integer variable used in connection with the NUMBER specifier. In an inquire-by-file operation, *unit_number* returns the unit number of the file connected to *file_name.* In an inquire-by-unit operation, it returns the value of *unit_spec*. The NUMBER specifier must not be used if UNIT = *.

The character variable *yes_or_no* returns 'YES', 'NO', or 'UNKNOWN'. Note that such a variable is used with four different specifiers: SEQUENTIAL, DIRECT, FORMATTED, and UNFORMATTED. Of course, a different character variable name (represented here by the generic name *yes_or_no*) must be used with each of these specifiers.

The character variable *access_type* returns `'SEQUENTIAL'` or `'DIRECT'`.

The character variable *format_mode* returns `'FORMATTED'` or `'UNFORMATTED'`.

The character variable *blank_mode* returns `'NULL'` or `'ZERO'`.

The `RECL` specifier can be used to determine the length of each record in a direct access file. Note that *record_length* is an integer variable. Length of a record is normally measured in bytes.

The integer variable *record_number* used with the `NEXTREC` specifier returns the number of the next record in a direct access file. It will return 1 (the number of the first record) if the file is connected but no records have yet been read or written.

All the specifiers except `UNIT` and `FILE` may be used in either an inquire-by-unit or an inquire-by-file operation. (The `UNIT` specifier cannot be used in an inquire-by-file operation, whereas the `FILE` specifier cannot be used in an inquire-by-unit operation.) For the `INQUIRE` statement to return the properties of a file, however, the file must be open at the time of the execution of the `INQUIRE` statement. This means that, if the file is not yet connected, the variables *fname, unit_number, record_number, access_type, format_mode, record_length, blank_mode,* and the four character variables represented by *yes_or_no* will not be defined upon the execution of the `INQUIRE` statement. Exceptions to this are as follows. The `EXIST` specifier can be used to check the existence of an unopened file. The `OPENED` specifier can be utilized to determine if a file is open or not. As noted before, the `NAMED` specifier can also be used with an unopened file, although such a usage may not be very meaningful: *name_status* always returns `.FALSE.` for an unopened file. The `UNIT`, `FILE`, `IOSTAT`, and `ERR` specifiers can be used with not-yet-opened files.

6.8 Formatted versus Unformatted Files

At the beginning of this chapter it was mentioned that there are three types of records in FORTRAN. We have already discussed the **end-of-file record** at some length. This record is important for files that are accessed sequentially, and it is normally written by means of the `END FILE` statement. On most `FORTRAN` systems the CLOSE statement will also perform this function in case the END FILE statement is omitted.

A **formatted record** consists of a sequence of characters selected from those that are allowed by the compiler being used. A formatted record is written by using formatted output statements. Recall that there are two types of formatted output: User-formatted output and list-directed output. Both of the following statements, for example, generate formatted records:

```
WRITE (UNIT = 1, FMT = *) X, Y, Z
WRITE (UNIT = 10, FMT = '(3F7.2)') X, Y, Z
```

Each of these statements produces a new record. A formatted I/O statement may read or write more than one record by using a suitable format. For example:

```
WRITE (UNIT = 2, FMT = '(2I4 / F7.2)') N, M, X
```

A formatted record must be read by a user-formatted or a list-directed formatted input statement.

It must thus be clear that, with the exception of end-of-file records, all the records written or read by the programs we have seen so far in this book were formatted records. A formatted record can also be generated by some means other than a FORTRAN program; e.g. it may be typed at the keyboard. Each program line of a FORTRAN source file (typed using an editor and the keyboard), for example, is a formatted record. Similarly, each line of output printed (using either list-directed or user-formatted output) into a file is a formatted record. Each data item in a formatted record is represented as a string of characters, i.e. in a form human beings and different types of computers can understand.

The internal binary representations of data are stored directly in an unformatted file. An **unformatted record** consists of a sequence of values in a form that depends on the type of computer used to generate it. Unformatted files are therefore less portable, but they can generally be processed more quickly. This is because the work involved in converting values from their internal binary representation into character form, or vice versa, is eliminated. An unformatted record is generated by an unformatted WRITE statement. For example:

```
WRITE (UNIT = 2) X1, X2, X3
WRITE (UNIT = 1, IOSTAT = IOS) M, N
```

Similarly, an unformatted record can only be read by an unformatted READ statement:

```
      READ (UNIT = 10) Y
      READ (UNIT = 10, IOSTAT = IOS) A, B
```

It should be added that an unformatted I/O statement will always read or write exactly one record. The number of data items in the input list of an unformatted READ statement should therefore be equal to or less than the number of items in the unformatted record. In the latter case, the last few items in the record are ignored.

Generally speaking, unformatted input and output are much faster than formatted input and output. This may become an important consideration when repeatedly writing or reading large data files. There are two additional benefits to using unformatted I/O. First, files generated via unformatted output are often smaller than formatted files. Second, because no conversions (from internal binary form to character form and then back to internal form) are made, the internal precision of the machine is preserved during data transfer. This is relevant for real numbers and is due to the difference in precision of the internal and the external character representations of real data. On the other hand, the particular computer system and the compiler used to generate an unformatted file determine the form of each record of that file. As a result, unformatted files are more difficult to move from one type of computer to another, i.e. they are less portable.

Example 6.4:

A large data file that will be repeatedly processed on the same (type of) computer is best defined as an unformatted file. The file named MAIL.BIN in this example represents this type of a file. It contains the names and mailing addresses of, say, the customers of a commercial organization. This list of addresses can be updated (expanded) to include the addresses of new customers by running the following program. Note that the names and addresses of new customers are read from a formatted file named UPDATE.TXT. This file may have been generated using another type of computer at a different location. For ease of transport from one type of computer to the other, it has been defined as a formatted file.

```
      PROGRAM MERGE
C***********************************************************************
C   Program merges two files MAIL.BIN and UPDATE.TXT
C   by appending the contents of UPDATE.TXT to the end
C   of the old mail list file MAIL.BIN. The program
C   generates MAIL.BIN if it does not yet exist.
C   UPDATE.TXT contains new names and addresses to be
C   added to MAIL.BIN. UPDATE.TXT is a formatted file.
C***********************************************************************
      IMPLICIT NONE
      CHARACTER*19 OLDDAT(4), NEWDAT(4)
      CHARACTER*1 CH
      LOGICAL EXISTS
      INTEGER IOS
      INQUIRE(FILE = 'MAIL.BIN', EXIST = EXISTS)
      IF(EXISTS) THEN
          OPEN(UNIT = 1, FILE = 'MAIL.BIN', FORM = 'UNFORMATTED',
     *    ACCESS = 'SEQUENTIAL', STATUS = 'OLD', IOSTAT = IOS)
          IF(IOS.NE.0) STOP 'Error while opening MAIL.BIN.'
      ELSE
          OPEN(UNIT = 1, FILE = 'MAIL.BIN', FORM = 'UNFORMATTED',
     *    ACCESS = 'SEQUENTIAL', STATUS = 'NEW', IOSTAT = IOS)
          IF(IOS.NE.0) STOP 'Error while opening MAIL.BIN.'
      ENDIF
C Make sure MAIL.BIN is positioned at the beginning of its first record.
      REWIND(UNIT = 1)
C Connect the new file UPDATE.TXT.
      OPEN(UNIT = 2, FILE = 'UPDATE.TXT', FORM = 'FORMATTED',
     *    ACCESS = 'SEQUENTIAL', STATUS = 'OLD', IOSTAT = IOS)
      IF(IOS.NE.0) STOP 'Error while opening UPDATE.TXT.'
C Position MAIL.BIN to just before the end-of-file record so that
C new names and addresses can be appended at the end of the file.
      DO WHILE(.TRUE.)
          READ(UNIT = 1, END = 10, IOSTAT = IOS) OLDDAT
          IF(IOS.NE.0) STOP 'An error occurred while reading MAIL.BIN.'
      END DO
   10 BACKSPACE(UNIT = 1)
C Read the contents of UPDATE.TXT. Write them to the end of MAIL.BIN.
      DO WHILE(.TRUE.)
          READ(UNIT = 2, FMT = 100, END = 20, IOSTAT = IOS) NEWDAT
  100     FORMAT(A/A/A/A)
          IF(IOS.NE.0)STOP 'An error occurred while reading UPDATE.TXT.'
          WRITE(UNIT = 1, IOSTAT = IOS) NEWDAT
          IF(IOS.NE.0) STOP 'An error occurred while writing MAIL.BIN.'
      END DO
   20 CLOSE(UNIT = 2)
      END FILE(UNIT = 1)
      PRINT*, 'Merge complete. Type E to see MAIL.BIN on the screen.'
      PRINT*, 'Type any other character to end execution of program.'
      READ '(A)', CH
      IF(CH.NE.'e' .AND. CH.NE.'E') THEN
          CLOSE(UNIT = 1)
          STOP 'Ending execution. Bye.'
      ENDIF
C Display the contents of MAIL.BIN on the computer screen.
      REWIND(UNIT = 1)
      DO WHILE(.TRUE.)
          READ(UNIT = 1, END = 30, IOSTAT = IOS) NEWDAT
          IF(IOS.NE.0) STOP 'An error occurred while reading MAIL.BIN.'
          PRINT '(1X,4(A,1X))', NEWDAT
      END DO
   30 CLOSE(UNIT = 1)
      END
```

To try and test this program, prepare a file named `UPDATE.TXT` using a text editor. A name and the corresponding address in this file should be typed in the following form:

```
Name Surname
Address line #1
Address line #2
Address line #3
```

A new file named MAIL.BIN will be generated during the first run. After running the program once in this way, prepare a different version of UPDATE.TXT (containing different and new names and addresses) and run the program again. This second run will update the previous version of MAIL.BIN by appending the new data at the end.

Example 6.5:

The following program further illustrates the use of the INQUIRE, OPEN, END FILE, CLOSE, BACKSPACE, and REWIND statements. Notice that the program utilizes the subprograms STRLEN, NMCASE, and UPCASE developed earlier (cf. Chapter 5). Although this is the largest program in this book, it is relatively straightforward and you should have no difficulty in understanding it.

Note that, despite its size, the program may have to be further developed and improved to be useful in practice. For example, the following features may be desirable: The ability to delete or modify an existing name and/or telephone number. The ability to search the database using a last name only to obtain a list records with the same last name, or the ability to specify a telephone number to find the corresponding name, etc. Adding home addresses, business addresses, fax numbers, e-mail addresses, etc. to the database may also be desirable.

The database is generated as a sequential-access file in this example. The use of a direct-access file may have to be considered if the program is going to be further developed. As a matter of fact, the program may have to completely re-designed to incorporate some of the above mentioned enhancements.

```
      PROGRAM NAMLST
C***************************************************************************
C   Program to generate, update, and read a database of
C   names and telephone numbers. More than one phone number
C   can be stored for a given name. The name of the file
C   to be generated, updated, or read is specified by user.
C***************************************************************************
      IMPLICIT NONE
      INTEGER IOS, NTEL, STRLEN
      CHARACTER ACT*1, REPLY*1, FNAME*12, UPCASE*1
      CHARACTER LAST*12, FIRST*12, TEL*20, SURNAM*12, NAME*12
      LOGICAL VALID, EXISTS, FOUND
C   Determine the desired program action
      VALID = .FALSE.
      DO WHILE(.NOT.VALID)
          PRINT*, 'Specify action. Type:'
          PRINT*, 'N for generating new file.'
          PRINT*, 'U for updating old file.'
          PRINT*, 'Q for query by name.'
          PRINT*, 'S to stop this program:'
          READ '(A)', ACT
          ACT = UPCASE(ACT)
          VALID = ACT.EQ.'N'.OR.ACT.EQ.'U'.OR.ACT.EQ.'Q'.OR.ACT.EQ.'S'
          IF(.NOT.VALID) PRINT *,'Please enter a valid action: N,U,Q,S?'
      END DO
      IF(ACT .EQ. 'S') STOP 'Execution terminated by request.'
C   Get filename and open it if name is valid
    1 PRINT*, 'Enter file name:'
      READ '(A)', FNAME
      INQUIRE(FILE = FNAME, EXIST = EXISTS, IOSTAT = IOS)
      IF(IOS.NE.0)THEN
          PRINT*, 'Error while looking for file: '//FNAME
          PRINT*, 'Check to ensure file name is valid.'
          GO TO 1
      ENDIF
      IF(.NOT.EXISTS .AND. (ACT.EQ.'U' .OR. ACT.EQ.'Q'))THEN
          PRINT *, FNAME(1:STRLEN(FNAME))//' does not exist.'
          GO TO 1
      ELSE IF(EXISTS .AND. (ACT.EQ.'U' .OR. ACT.EQ.'Q') )THEN
          OPEN(UNIT = 1, FILE = FNAME, FORM = 'UNFORMATTED',
     *    ACCESS = 'SEQUENTIAL', STATUS = 'OLD', IOSTAT = IOS)
          IF(IOS.NE.0) STOP 'Error while opening file.'
          REWIND(UNIT = 1)
      ELSE IF(EXISTS .AND. ACT.EQ.'N')THEN
          PRINT*, FNAME(1:STRLEN(FNAME))//
     *    ' already exists. Please type R to re-enter file name;'
          PRINT*, 'type any other character' //
     *    ' to overwrite '//FNAME(1:STRLEN(FNAME))//':'
          READ '(A)', REPLY
          IF(REPLY.EQ.'R' .OR. REPLY.EQ.'r') GO TO 1
          OPEN(UNIT = 1, FILE = FNAME, FORM = 'UNFORMATTED',
     *    ACCESS = 'SEQUENTIAL', STATUS = 'OLD', IOSTAT = IOS)
          IF(IOS.NE.0) STOP 'Error while opening file.'
          REWIND(UNIT = 1)
      ELSE IF(.NOT.EXISTS .AND.ACT.EQ.'N')THEN
          OPEN(UNIT = 1, FILE = FNAME, FORM = 'UNFORMATTED',
     *    ACCESS = 'SEQUENTIAL', STATUS = 'NEW', IOSTAT = IOS)
          IF(IOS.NE.0)THEN
              PRINT*, 'Error while opening file: '//FNAME
              PRINT*, 'Check to ensure file name is valid.'
              GO TO 1
          ENDIF
      ENDIF
C   Generate new file
      IF(ACT.EQ.'N')THEN
          CALL GETINP(1)
          END FILE(UNIT = 1)
          CLOSE(UNIT = 1)
          STOP 'File generation complete.'
      ENDIF
C   Answer query
      IF(ACT.EQ.'Q')THEN
          PRINT*, 'Enter last name:'
          READ '(A)', LAST
          PRINT*, 'Enter first name:'
          READ '(A)', FIRST
          CALL NMCASE(FIRST)
          CALL NMCASE(LAST)
          PRINT*, 'Searching database for '//FIRST(1:STRLEN(FIRST))
     *            //' '//LAST(1:STRLEN(LAST))//':'
          NTEL = 0
          FOUND = .FALSE.
          DO WHILE(.TRUE.)
              READ(UNIT = 1, END = 2, IOSTAT = IOS) SURNAM, NAME, TEL
              IF(IOS.NE.0) STOP 'Error while reading file.'
              IF(SURNAM.EQ.LAST .AND. NAME.EQ.FIRST) THEN
                  FOUND = .TRUE.
                  NTEL = NTEL + 1
                  PRINT '(1X,A,I1,A)', 'Tel-', NTEL,': '//TEL
              ENDIF
          END DO
    2     IF(.NOT.FOUND)THEN
              PRINT *, 'Reached EOF. No records for '//
     *        FIRST(1:STRLEN(FIRST))//' '//LAST(1:STRLEN(LAST))//'.'
          ENDIF
          CLOSE(UNIT = 1)
          STOP 'Name query complete.'
      ENDIF
C   Update file by adding records at the end
      IF(ACT.EQ.'U')THEN
          DO WHILE(.TRUE.)
              READ(UNIT = 1, END = 3, IOSTAT = IOS) SURNAM, NAME, TEL
              IF(IOS.NE.0) STOP 'Error while reading file.'
          END DO
    3     BACKSPACE (UNIT = 1)
          CALL GETINP(1)
          END FILE(UNIT = 1)
          CLOSE(UNIT = 1)
          STOP 'File update complete.'
      ENDIF
      END
C
```

```
      SUBROUTINE GETINP(UNUM)
      IMPLICIT NONE
      INTEGER UNUM, IOS
      CHARACTER LAST*12, FIRST*12, TEL*20, CH*1
      LOGICAL LETTER
      LETTER(CH) = (LLE('A',CH).AND.LLE(CH,'Z')) .OR.
     *             (LLE('a',CH).AND.LLE(CH,'z'))
      DO WHILE(.TRUE.)
          PRINT*, 'Enter last name (Type non-letter to stop.):'
          READ '(A)', LAST
          IF(.NOT.LETTER(LAST(1:1))) RETURN
          PRINT*, 'Enter first name:'
          READ '(A)', FIRST
          PRINT*, 'Enter tel: '
          READ '(A)', TEL
          CALL NMCASE(FIRST)
          CALL NMCASE(LAST)
          WRITE(UNIT = UNUM, IOSTAT = IOS) LAST, FIRST, TEL
          IF(IOS.NE.0) STOP 'Error while writing file.'
      END DO
      RETURN
      END
```

6.9 Carriage Control Characters

In our study of user-formatted output in Chapter 1, it was emphasized that you should avoid printing in the first column of the output line, as anything printed there may not be displayed on the screen (see Exercise 24 in Ch.1). This problem was solved by printing a blank character in the first column of the output line (by either explicitly printing the blank character `' '`, or by including `1X` as the first edit descriptor in the format specification). The term "**output line**" (also called "**output buffer**") refers to an internal memory region of the computer. The computer uses the relevant format specification to construct an output line internally in memory before actually printing the line. The first character of the buffer is called the **carriage control character**; it determines the vertical spacing for the line. The remaining characters represent the line to be actually printed. The four valid carriage control characters (also called **line-control characters** or **printer control characters**) are listed in the following table:

Character	Effect
Blank	Prints the current line immediately below the previous line (single-line spacing)
0	One blank line is left between the current line and the previous line (double-line spacing)
+	Overwrites the previous line (no paper advance)
1	Advances to top of next page

Since the first character of the output line is removed and not printed, it is important that you include an extra (carriage control) character at the beginning of each output line that is to be sent to a printer or to another output device that the compiler designates as a **printer**. This includes all line printers, some computer terminals, and display screens of personal computers. Normally, you do *not* have to use carriage control characters when writing into an output file. Likewise, the first character is *not* treated as a carriage control character in list-directed output.

Example 6.6:

You can determine whether carriage control is operative on your processor by typing and running the following simple program. This program will also let you find out which of the four control characters (' ', '+', '1' and '0') can be used to control the output on your computer's display screen.

```
      PROGRAM CARRG
      PRINT *, 'This is Line 1.'
      PRINT 1, 'Line 2.'
    1 FORMAT(' ', 45('X'), 1X, A)
      PRINT 2, 'This is expected to overwrite line 2.'
    2 FORMAT('+', A)
      PRINT 3, 'This should be printed after one blank line.'
    3 FORMAT('0', A)
      PRINT 4, 'Is this sentence printed to a new page?'
    4 FORMAT('1', A)
      END
```

If carriage control is operative, the output of the program will look like the following:

```
This is Line 1.
This is expected to overwrite line 2.XXXXXXX Line 2.

This should be printed after one blank line.
------------New Page-------------
Is this sentence printed to a new page?
```

Remark: Microsoft FORTRAN Reference Manual cited at the References section of this book states (on p.79) that "The screen behaves as if this carriage control character ['1'] is ignored." Verify this by running the above program on the Microsoft FORTRAN system. Lahey Computer Systems' Compiler behaves similarly (i.e. the line is not printed on a new "page"), except that a graphics character is displayed on the screen as the first character (try it).

If the first character of the output line is not one these four line-control characters, then the effect on the printer is not defined. However, on most processors, any other character will cause single spacing (it will have the same effect as does `1X` and `' '`), and printing will take place at the next line. (An exception to this occurs when printing the characters 3 to 9 to the screen using the Lahey compiler. See Exercise 3 below.). It should be remembered that the character in question normally will not be printed/displayed. Since a terminal and a PC's display screen do not have the capabilities as a printer for spacing, `'1'` usually becomes an invalid control character and causes single-spacing. (As remarked in Example 6.6, however, a graphics character is displayed on the screen by the Lahey compiler. The Microsoft compiler ignores `'1'` and it does not have any effect.) If the I/O system does not use carriage control, the entire contents of the output line, including the carriage control characters, will be displayed. This is what normally happens if you use carriage control characters when printing into an output file.

Example 6.7:

Forgetting to insert a carriage control character when one is expected (e.g. when the output device is a line printer or possibly a PC's display screen) causes FORTRAN to use the first character of the output line for line control instead of printing it. This can lead to unexpected results in the output. As an illustration of what can go wrong, type and run the following program using all the FORTRAN compilers available to you.

```
      PROGRAM CARRG2
      IMPLICIT NONE
      REAL X, Y
      X = 6.0
      Y = 3.0
      PRINT 1, X
      PRINT 1, Y
      PRINT 1, X*Y
      PRINT 2, Y/X
    1 FORMAT(F6.3)
    2 FORMAT(F6.4)
      END
```

If carriage control were fully operative, the result would look like the following:

```
6.000
3.000
------------New Page-------------
8.000

.5000
```

This is certainly not the intended form of the output. Note that one line is skipped before the value of `Y/X` is printed. This happens because, this value is `0.5000`, and the leading zero is interpreted as a line-control character. The output obtained (on the display screen) using the Lahey compiler is similar to this, except that the “next page” character `'1'` does not cause printing on a new “page”; a graphical character is displayed instead. The Microsoft compiler ignores `'1'` altogether (it is not displayed and it does not have any effect). In any case, the two formats should be rewritten as follows (try it):

```
    1 FORMAT(1X, F6.3)
    2 FORMAT(1X, F6.4)
```

Exercises:

3. Modify the program in Example 6.6 to print lines into a file. You should replace the `PRINT` statements by `WRITE` statements. Do not change the `FORMAT` statements. What happens? Are the carriage-control characters (`' '`, `'+'`, `'1'` and `'0'`) printed into the file? Do they affect line spacing in the file? Next, remove these characters from the format specifications. What happens?

4. Test the following program using both the Lahey and the Microsoft compilers:

```
      PROGRAM CARRG3
C   List-directed output (a control character not needed):
      WRITE( *, *) 'Testing carriage control characteristics.'
C   User-formatted output without a control character:
      WRITE( *, 1) '0 (zero)'
      WRITE( *, 1) '1 (one)'
      WRITE( *, 1) '2 (two)'
      WRITE( *, 1) '3 (three)'
      WRITE( *, 1) '4 (four)'
      WRITE( *, 1) '9 (nine)'
      WRITE( *, 1) '10 (ten)'
      WRITE( *, 1) 'a (letter a)'
C   User-formatted output using a control character:
      WRITE( *, 2) '0 (zero)'
      WRITE( *, 2) '1 (one)'
      WRITE( *, 2) '2 (two)'
      WRITE( *, 2) '3 (three)'
      WRITE( *, 2) '4 (four)'
      WRITE( *, 2) '9 (nine)'
      WRITE( *, 2) '10 (ten)'
      WRITE( *, 2) 'a (letter a)'
    1 FORMAT(A)
    2 FORMAT(1X, A)
      END
```

Make sure that you understand the differences in the actions of the two compilers. (Hint: The Microsoft compiler treats all characters other than the four control characters like the blank character. What about the Lahey compiler?)

In their book on Fortran 90, Ellis et al. remark that "When a line of output is to be sent to the output device designated as the printer, the Fortran output system will remove the first character of the line and interpret it as a printer control character which determines how much the paper is to be moved up before any remaining characters of the line are printed. This apparently bizarre behavior reflects the way in which some of the very early printers, back in the 1950s, actually worked and has remained in Fortran ever since."

6.10 Input Editing

Input formats can be employed to describe the line-by-line appearance of input data. While input editing is not very frequently needed, a basic understanding of how it is used may prove beneficial. For example, enclosing apostrophes must normally be typed when entering a character string to be read by a list-directed `READ` statement. User-formatted input, on the other hand, enables strings to be typed without enclosing apostrophes (see Examples 6.5 and 6.8). A carriage control character should not be included in a format specification used for input.

Example 6.8:

Consider the following program which writes even numbers up to a user-specified (odd or even) number `N` into a file named by the user. Note that the output file name is entered by the user interactively, i.e. it is not hard-coded in the program. The advantage is that the user can specify a different output file name during each different execution of the program without having to modify the source code.

```
      PROGRAM EVENS
C***********************************************************************
C   Prints even numbers less than or equal to a user-entered
C   integer N into an output file also specified by the user.
C***********************************************************************
      IMPLICIT NONE
      INTEGER N, NUM, IOS
      LOGICAL EXISTS
      CHARACTER *12 FNAME
      PRINT*, 'Even number generation program.'
      PRINT*, 'Enter upper limit (a positive integer): '
      READ*, N
C   Obtain a valid output file name
      IOS = 1
      DO WHILE(IOS.NE.0)
          PRINT*, 'Enter output file name: '
          READ '(A)', FNAME
          INQUIRE(FILE = FNAME, EXIST = EXISTS)
          IF(EXISTS) THEN
              OPEN(UNIT = 1, FILE = FNAME, STATUS = 'OLD', IOSTAT = IOS)
              IF(IOS.EQ.0)THEN
                  PRINT '(1X,''Overwriting existing file: '', A)', FNAME
                  REWIND(UNIT = 1)
              ENDIF
          ELSE
              OPEN(UNIT = 1, FILE = FNAME, STATUS = 'NEW', IOSTAT = IOS)
          ENDIF
      END DO
C   Write even numbers up to N into FNAME
      DO 5 NUM = 2, N, 2
          WRITE(UNIT = 1, FMT = 10) NUM
  10      FORMAT(I5)
   5  CONTINUE
      END FILE(UNIT = 1)
      PRINT*, 'Even number generation is complete.'
C   Echo the contents of FNAME on the screen
      REWIND(UNIT = 1)
      PRINT '(1X,A,1X,''contains the following:''/)',FNAME
      DO WHILE(.TRUE.)
          READ(UNIT = 1, FMT = 10, END = 20) NUM
          PRINT*, NUM
      END DO
  20  CLOSE(UNIT = 1)
      END
```

Since input-editing is employed, enclosing apostrophes for the input string (`'even.out'`) need not and should not be typed. Positive even numbers (integers) up to the user-specified number `N` are written into the output file `FNAME` using the edit descriptor `I5`. After all the output data are written, the `END FILE` statement appends an end-of-file record after the last file line. This must be the last record in any sequential file.

Note the first use of the `REWIND` statement. When a file that already exists (but is not yet connected to the program) is opened, the initial position of the file is actually not defined. To ensure that the file is positioned at the beginning of its first record (so that the first `WRITE` statement will overwrite and replace that record), the `REWIND` statement is included. This is usually not necessary because most FORTRAN compilers will position an existing file at its initial point ready to read or overwrite the first record. For maximum portability, however, it seems prudent to include the `REWIND` statement in this manner.

Normally, an output file is disconnected from a program (using a `CLOSE` statement) once its generation is complete. In this case, however, the output file `FNAME` is kept open so that the program can read its contents and display them on the display screen. The `REWIND` statement positions the position pointer for the file with unit number 1 (i.e. `FNAME`) just before the first file record. Note that the same edit descriptor (`I5`) is employed to read the integer data in the file as the one employed to write them. Once all the data in the output file are read and displayed, the file is closed using the `CLOSE` statement. A sample execution output (on the display screen) of the program is as follows:

```
Even number generation program.
Enter largest number: 16
Enter output file name: even.out
Even number generation is complete.
even.out     contains the following:

            2
            4
            6
            8
           10
           12
           14
           16
```

The contents of the output file can also be viewed by typing 'type even.out' at the DOS command prompt.

Exercises:

5. Consider Example 6.8: What happens if you omit the `REWIND` statement before reading the contents of the file?

6. Consider the `READ` statement in Example 6.8: What happens if you use the format specifications `'(I3)'` or `'(I4)'` to read the data (instead of `'(I5)'`) while still using `'(I5)'` to write the data?

7. A single `OPEN` statement with the specification `STATUS = 'UNKNOWN'` may be employed in Example 6.8. Replace the program segment

```
        INQUIRE(FILE = FNAME, EXIST = EXISTS)
        IF(EXISTS) THEN
            OPEN(UNIT = 1, FILE = FNAME, STATUS = 'OLD', IOSTAT = IOS)
            IF(IOS.EQ.0)THEN
                PRINT '(1X,''Overwriting existing file: '', A)', FNAME
                REWIND(UNIT = 1)
            ENDIF
        ELSE
            OPEN(UNIT = 1, FILE = FNAME, STATUS = 'NEW', IOSTAT = IOS)
        ENDIF
```

by the statements

```
        OPEN(UNIT = 1, FILE = FNAME, STATUS = 'UNKNOWN', IOSTAT = IOS)
        IF(IOS.EQ.0) REWIND(UNIT = 1)
```

This approach should also work when a file with the specified name (the value of `FNAME`) already exists. Actually, what happens when `STATUS = 'UNKNOWN'` is compiler-dependent. In most cases, the file will be treated as `OLD` if it already exists. If it does not exist, it will be treated as `NEW`.

To find out how your FORTRAN system behaves when `STATUS` is `'UNKNOWN'`, run the modified program as follows. Use the same output file name for two consecutive runs (during the second run a file with that name will already exist). If the program does not prompt you to re-enter a file name during either of these runs, and if the execution of the program is completed successfully, then the modified version of the program is valid. Make sure to use a unique and new file name during the first run (i.e. a file with that name should not be present in the directory you are working).

Next, try using `STATUS = 'NEW'`. If a file with the specified name already exists, the program will prompt you to type a different file name. (Note: An execution-time error would have occurred and the program would have terminated prematurely if we had not utilized the `IOSTAT` specifier and the `DO WHILE` loop to ensure that a valid file name is ultimately typed.) Alternatively, choose a new file name, and run the program twice using that same name (and

`STATUS = 'NEW')` each time. What happens in the second run? Finally, set `STATUS = 'OLD'` and run the program with a new output file name. What happens?

8. Note how the `IOSTAT` specifier is utilized in Example 6.8 to take corrective action in case of an error. If an invalid file name is typed, the program simply prompts the user to enter a different file name. As an exercise, rewrite the program without using the `IOSTAT` specifier. You can, for example, replace the entire `DO WHILE` loop by the statement

```
        OPEN(UNIT = 1, FILE = FNAME, STATUS = 'UNKNOWN')
```

If you type an invalid file name when you run the program, you will get an error message and the execution will terminate prematurely. (Note: Allowed file names depend on the operating system being used, i.e. DOS 6.22, Windows 95, etc.)

The edit descriptors used for input editing are similar to those used for output editing (cf. Chapter 1). The most frequently used edit descriptors for input editing are shown in the following table.

Edit Descriptor	Description
`Iw`	For reading the next `w` characters as an integer.
`Fw.d` `Ew.d`	For reading the next `w` characters as a real number with `d` digits after the decimal place if no decimal point is present. The value of `d` is ignored if a decimal point is present in the input field.
`A`	For reading a character value. Sufficient characters will be read to fill the input list item.
`Aw`	For reading the next `w` characters as a character string.
`nX`	For skipping `n` horizontal spaces (i.e. `n` columns are ignored).
`Tn`	To advance to position `n` of the input record before reading the next item, i.e. next item to be read starts at position `n`.
`TLn`	Next item to be read is `n` positions before the current position.
`TRn`	Next item to be read is `n` positions after the current position.
`Lw`	Read the next `w` characters as the representation of a logical value.

The edit descriptors are best explained using specific examples. Consider the following line of data:

```
123456789
```

The line contains the digits 1 through 9 in columns 1 through 9, respectively. To read this line as a single integer to be stored in the integer variable `M` one can write

```
      READ '(I9)', M
```

To read the same line as the four separate integers 12, 34, 56, and 789 one could use the following statement:

```
        READ '(I2,I2,I2,I3)', M1, M2, M3, M4
```

The statement

```
        READ '(3X,I6)', M
```

will ignore the first 3 columns and then read the next 6 as an integer; the value 456789 will consequently be stored in `M`. The statement

```
        READ '(I2,3X,I2)', M, N
```

will cause the value `12` to be stored in `M` and `67` in `N`. Columns 8 and 9 are ignored because the format only specifies the first 7 positions. All of the statements

```
        READ '(T3,I2,T9,I1,T2,I3)', M, N, K
        READ '(2X,I2,4X,I1,T2,I3)', M, N, K
        READ '(TR2,I2,TR4,I1,TL8,I3)', M, N, K
```

will read the value `34` into `M`, `9` into `N`, and `234` into `K`. Note that the `T` edit descriptor allows going back in the record and reading parts of it again. Note also that `TRn` has the same effect as `nX`. The letters `TL` followed by a number `n` specify a tab to the left, and cause the next position to be `n` positions to the left of the current position.

When used for input editing the `E` and `F` edit descriptors are interpreted in the same way. (On output, however, we know that they are different.) Using the same input line given above, the statement

```
        READ '(F9.3)', X
```

will cause the value `123456.789` to be stored in `X`. On the other hand, the statement

```
        READ '(F4.1,F2.2,F2.0,TL5,F3.1)', X, Y, Z, W
```

will cause the value `123.4` to be stored in `X`, `0.56` in `Y`, `78.0` in `Z`, and `45.6` in `W`. Consider what happens if the same statement is used to read the following line:

```
.451.78.9
```

The first edit descriptor `F4.1` requires four columns to be read, and since these (`.451`) contain a decimal point the second part of the edit descriptor is ignored and the value `.451` is stored in `X`. Similarly the value `0.7` will be stored in `Y`, `8.0` in `Z`, and `1.7` in `W`.

An `A` edit descriptor without any field width `w` is treated as though the field width was identical to the length of the corresponding input list item (a character variable). For example, if the two character variables CHR1 and CHR2 are declared by the statement

```
        CHARACTER CHR1*8, CHR2*11
```

then the following two statements will have an identical effect:

```
        READ '(A,A)', CHR1, CHR2
        READ '(A8,A11)', CHR1, CHR2
```

When an A edit descriptor is used with a field width w, there are three possibilities: (i) The length *len* of the input list item and field width w may be the same. In this case, the forms A and Aw have an identical effect, and the question of blank padding or truncation does not occur. (ii) w may be less than *len*. In this case extra blank characters will be added at the right of the input character string, and the result will be stored in the input list item. This is similar to the situation with assignment. (iii) w may be larger than *len*. In this case, the rightmost *len* characters of the input string will be stored in the input list item. Note that this is the opposite of what happens with assignment.

The L edit descriptor is used in the form Lw, and processes the next w characters to derive either a .TRUE. value or a .FALSE. value. The input field consists of optional blanks, followed by an optional period, followed by T or F. If the first non-blank character, other than a period, is not T or F, then an error will occur. Any further characters after T or F in the field are ignored. As an exercise, find out if the lower case letters (t and f) are also accepted by your FORTRAN compiler.

6.11 Internal Files

An internal file is actually not a file; it is a character variable, a character array element, a character array, or a character substring. An internal file is identified by using the character entity in question in a READ or WRITE statement in place of the unit identifier. Thus, for example, the character variable LINE used in the following program segment is an internal file:

```
        CHARACTER*20 LINE
        WRITE(UNIT = LINE, FMT = '(2F10.3)') X, Y
        READ(UNIT = LINE, FMT = '(2(F7.0,3X))') X, Y
```

The program segment generates a character string in the variable LINE consisting of the representations of the values of X and Y with three decimal places. It then reads these back into X and Y in such a way as to ignore the digits following the decimal points. The effect is thus the truncation of the fractional parts of the values.

An internal file is a means whereby FORTRAN's formatting process can be used to convert information from one format to another without the use of any external media. The program segment given above as well as the programs in Exercise 9 and Example 6.9 are examples of this application of internal files.

An internal file can also be utilized to read a record of an external file more than once without the need to backspace the external file. The entire record is first read and stored in a character variable (the internal file). The contents of the internal file can then be read as many times as required by the program. A simple illustration is presented in Exercise 10.

<u>Exercises:</u>

9. Type and run the following program. Among others, try the following values as input: `1.2345` (for `X`) and `5.6543` (for `Y`). What does the program do?

```
IMPLICIT NONE
REAL X, Y
CHARACTER*20 LINE
PRINT*, 'Enter X, Y:'
READ*, X, Y
WRITE(UNIT = LINE, FMT = '(2F10.3)') X, Y
PRINT*, 'LINE =', LINE
READ(UNIT = LINE, FMT = '(2(F8.1,2X))') X, Y
PRINT '(1X, 2(A, F10.3, 2X))', 'X=', X, 'Y=', Y
END
```

10. Type and run the following program. You should first prepare a text file named `ageinfo.txt`. The first column will contain one of the letters `M` or `F`. The second column should be left blank. If you type `M` in the first column, type an integer between 1 and 999 in columns 3 through 5.

```
PROGRAM INTFL2
IMPLICIT NONE
CHARACTER RECORD*5, GENDER*1
INTEGER AGE
OPEN(UNIT = 1, FILE = 'ageinfo.txt', STATUS = 'OLD')
READ(UNIT = 1, FMT = '(A)') RECORD
CLOSE(UNIT = 1)
READ(UNIT = RECORD, FMT = '(A1)') GENDER
IF(GENDER.EQ.'M' .OR. GENDER.EQ.'m')THEN
    READ(UNIT = RECORD, FMT = '(T3, I3)') AGE
    PRINT '(1X,A,1X,I3)', 'Age =', AGE
ENDIF
END
```

Next, modify the program by eliminating the character variable `RECORD`. Use the `BACKSPACE` statement instead so as to read the line (record) again when the specified gender is `M`.

If an internal file is a character variable, a character array element, or a character substring, then it consists of a single record. The length of this record is the same as the length of the variable, array element, or substring. If an internal file is a character array, then each array element is a record. All records have the same length, i.e. the length of array elements. In this case the entire file must be read or written by means of a single READ or WRITE statement. This is because a READ or WRITE statement on an internal file always starts at the beginning of the file. An internal file does not contain an end-of-file record. Considering the properties described here, it may be said that an internal file is a formatted sequential-access file without an end-of-file record.

Example 6.9:

The following function subprogram converts a string consisting of digits into the equivalent integer value. The value of 2*INTCH('55 '), for example, is the integer 110.

```
      INTEGER FUNCTION INTCH(STRING)
C***************************************************************************
C   Returns the integer value represented by STRING
C   STRING may contain leading and/or embedded blanks
C***************************************************************************
      IMPLICIT NONE
      CHARACTER*(*) STRING
      CHARACTER*5 FORMT
      INTEGER L
      FORMT = '(I  )'
      DO 1 L = LEN(STRING), 1, -1
          IF(STRING(L:L).NE.' ') GO TO 2
    1 CONTINUE
    2 WRITE(UNIT = FORMT(3:4), FMT = '(I2)') L
      READ(UNIT = STRING, FMT = FORMT) INTCH
      RETURN
      END
```

The input argument STRING is an assumed-length character argument. The function uses a local character variable named FORMT as an internal file. The length of STRING excluding any blank padding is first determined. It is assumed that all the non-blank characters contained in STRING are digits. The number of digits (i.e. the length of STRING excluding blank padding) is written into positons 3 and 4 within FORMT. This character variable is then used as the format specifier in the READ statement which determines the value of the function. Notice that STRING is also treated as an internal file in the READ statement. This READ statement carries out the conversion from a character string to an integer.

Note that since the format specifier FORMT cannot exceed FORMT='(I99)', the maximum length (excluding blank padding on the right) of STRING is 99. This should be sufficient for all practical purposes. You can test INTCH using the following driver routine.

```
      IMPLICIT NONE
      CHARACTER*99 STRING
      INTEGER INTCH, NUMBER
      EXTERNAL INTCH
      PRINT*, 'Enter an integer enclosed in apostrophes:'
      READ *, STRING
      NUMBER = INTCH(STRING)
      PRINT*, 'Number is =', NUMBER
      END
```

Example 6.10:

Consider the problem of converting a positive integer from its base 10 (decimal) representation into its equivalent representation in another base up to base 16. An algorithm to generate the digits of the converted number is the following: A digit of the converted number is obtained by taking the remainder of the division of the number by the base. The number is then divided by the base, with any fractional remainder discarded, and the process is repeated until the number reaches zero.

This algorithm generates the digits of the converted number starting from the rightmost digit. The following program, therefore, stores the digits of the converted number in an array named CONNUM, rather than displaying each digit as soon as it is generated. Once the conversion is finished, the digits can be displayed in the correct order.

Note that any digit of the converted number that is between 10 and 15 is displayed using the corresponding letters, A through F. This is accomplished via the character array DIGIT. For example, DIGIT(10) is 'A', DIGIT(11) is 'B', etc.

Finally, instead of displaying the converted number directly, the number is written into an internal file named TEXNUM, and the contents of this internal file are printed.

```
      PROGRAM CONVRT
C*************************************************************************
C   Program converts a positive decimal integer to another base
C   The converted number is written into an internal file
C*************************************************************************
      IMPLICIT NONE
      CHARACTER*1 DIGIT(0:15)
      DATA DIGIT /'0', '1', '2', '3', '4', '5', '6', '7',
     %            '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'/
      INTEGER NUMBER, BASE, I, J, CONNUM(64)
      CHARACTER*64 TEXNUM
      PRINT*, 'Number to be converted? '
      READ*, NUMBER
      PRINT*, 'Enter base (2 to 16): '
      READ*, BASE
      I = 1
    1 CONTINUE
          CONNUM(I) = MOD(NUMBER, BASE)
          I = I + 1
          NUMBER = NUMBER / BASE
      IF(NUMBER .NE. 0) GO TO 1
      TEXNUM = ' '
      DO 2 J = 1, I-1
          WRITE(TEXNUM(I-J:I-J), '(A1)') DIGIT(CONNUM(J))
    2 CONTINUE
      PRINT*, 'Converted number: ', TEXNUM
      END
```