Introduction to the f77 I/O Library

David L. Wasley

University of California, Berkeley
Berkeley, California 94720

ABSTRACT

The f77 I/O Library implements ANSI 1978 Fortran standard input and output with a few minor exceptions. Where the standard is vague, we have tried to provide flexibility within the constraints of the UNIX† operating system. The I/O Library was written originally by Peter J. Weinberger at Bell Labs. A number of logical extensions and enhancements have been provided by this author.

April 1983

Revised for 4.3 BSD, September, 1985 by J. Berkman.

† UNIX is a trademark of Bell Laboratories.

-- --

Introduction to the f77 I/O Library

David L. Wasley

University of California, Berkeley
Berkeley, California 94720

The f77 I/O library, libI77.a, includes routines to perform all of the standard types of Fortran input and output. Several enhancements and extensions to Fortran I/O have been added. The f77 library routines use the C stdio library routines to provide efficient buffering for file I/O.

1. Fortran I/O

The requirements of the ANSI standard impose significant overhead on programs that do large amounts of I/O. Formatted I/O can be very ‘‘expensive’’ while direct access binary I/O is usually very efficient. Because of the complexity of Fortran I/O, some general concepts deserve clarification.

1.1. Types of I/O and logical records

There are four forms of I/O: **formatted, unformatted, list directed,** and **namelist.** The last two are related to formatted but do not obey all the rules for formatted I/O. There are two types of ‘‘files’’: **external** and **internal** and two modes of access to files: **direct** and **sequential.** The definition of a logical record depends upon the combination of I/O form, file type, and access mode specified by the Fortran I/O statement.

1.1.1. Direct access external I/O

A logical record in a **direct** access **external** file is a string of bytes of a length specified when the file is opened. Read and write statements must not specify logical records longer than the original record size definition. Shorter logical records are allowed. **Unformatted** direct writes leave the unfilled part of the record undefined. **Formatted** direct writes cause the unfilled record to be padded with blanks.

1.1.2. Sequential access external I/O

Logical records in **sequentially** accessed **external** files may be of arbitrary and variable length. Logical record length for **unformatted** sequential files is determined by the size of items in the iolist. The requirements of this form of I/O cause the external physical record size to be somewhat larger than the logical record size. For **formatted** write statements, logical record length is determined by the format statement interacting with the iolist at execution time. The ‘‘newline’’ character is the logical record delimiter. Formatted sequential access causes one or more logical records ending with ‘‘newline’’ characters to be read or written.

1.1.3. List directed and namelist sequential external I/O

Logical record length for **list directed** and **namelist** I/O is relatively meaningless. On output, the record length is dependent on the magnitude of the data items. On input, the record length is determined by the data types and the file contents. By ANSI definition, a slash, ‘‘/’’, terminates execution of a list directed input operation. Namelist input is terminated by ‘‘&end’’ or ‘‘$end’’ (depending on whether the character before the namelist name was ‘‘&’’ or ‘‘$’’).

1.1.4. Internal I/O

The logical record length for an **internal** read or write is the length of the character variable or array element. Thus a simple character variable is a single logical record. A character variable array is similar to a fixed length direct access file, and obeys the same rules. **Unformatted** and **namelist** I/O are not allowed on ‘‘internal’’ files.

1.2. I/O execution

Note that each execution of a Fortran **unformatted** I/O statement causes a single logical record to be read or written. Each execution of a Fortran **formatted** I/O statement causes one or more logical records to be read or written.

A slash, ‘‘/’’, will terminate assignment of values to the input list during **list directed** input and the remainder of the current input line is skipped. The standard is rather vague on this point but seems to require that a new external logical record be found at the start of any formatted input. Therefore data following the slash is ignored and may be used to comment the data file.

Direct access list directed I/O is not allowed. **Unformatted internal** I/O is not allowed. **Namelist** I/O is allowed only with **external sequential** files. All other flavors of I/O are allowed, although some are not part of the ANSI standard.

Any I/O statement may include an **err=** clause to specify an alternative branch to be taken on errors and/or an **iostat=** clause to return the specific error code. Any error detected during I/O processing will cause the program to abort unless either **err=** or **iostat=** has been specificed in the program. Read statements may include **end=** to branch on end-of-file. The end-of-file indication for that logical unit may be reset with a **backspace** statement. File position and the value of I/O list items is undefined following an error.

2. Implementation details

Some details of the current implementation may be useful in understanding constraints on Fortran I/O.

2.1. Number of logical units

Unit numbers must be in the range 0 − 99. The maximum number of logical units that a program may have open at one time is the same as the UNIX system limit, currently 48.

2.2. Standard logical units

By default, logical units 0, 5, and 6 are opened to ‘‘stderr’’, ‘‘stdin’’, and ‘‘stdout’’ respectively. However they can be re-defined with an **open** statement. To preserve error reporting, it is an error to close logical unit 0 although it may be reopened to another file.

If you want to open the default file name for any preconnected logical unit, remember to **close** the unit first. Redefining the standard units may impair normal console I/O. An alternative is to use shell redirection to externally re-define the above units. To re-define default blank control or format of the standard input or output files, use the **open** statement specifying the unit number and no file name (see § 2.4).

The standard units, 0, 5, and 6, are named internally ‘‘stderr’’, ‘‘stdin’’, and ‘‘stdout’’ respectively. These are not actual file names and can not be used for opening these units. **Inquire** will not return these names and will indicate that the above units are not named unless they have been opened to real files. The names are meant to make error reporting more meaningful.

2.3. Vertical format control

Simple vertical format control is implemented. The logical unit must be opened for sequential access with **form = ′print′** (see § 3.2). Control codes ‘‘0’’ and ‘‘1’’ are replaced in the output file with ‘‘\n’’ and ‘‘\f’’ respectively. The control character ‘‘+’’ is not implemented and, like any other character in the first position of a record written to a ‘‘print’’ file, is dropped. The **form = ′print′** mode does not recognize vertical format control for **direct formatted, list directed,** or **namelist** output.

An alternative is to use the filter *fpr*(1) for vertical format control. It replaces ‘‘0’’ and ‘‘1’’ by ‘‘\n’’ and ‘‘\f’’ respectively, and implements the ‘‘+’’ control code. Unlike **form = ′print′** which drops unrecognized form control characters, *fpr* copies those characters to the output file.

2.4. File names and the open statement

A file name may be specified in an **open** statement for the logical unit. If a logical unit is opened by an **open** statement which does not specify a file name, or it is opened implicitly by the execution of a **read**, **write**, or **endfile** statement, then the default file name is *fort.N* where *N* is the logical unit number. Before opening the file, the library checks for an environment variable with a name identical to the tail of the file name with periods removed.† If it finds such an environment variable, it uses its value as the actual name of the file. For example, a program containing:

```
open(32,file='/usr/guest/census/data.d')
read(32,100) vec
write(44) vec
```

normally will read from */usr/guest/census/data.d* and write to *fort.44* in the current directory. If the environment variables *datad* and *fort44* are set, e.g.:

```
% setenv datad mydata
% setenv fort44 myout
```

in the C shell or:

```
$ datad=mydata
$ fort44=myout
$ export datad fort44
```

in the Bourne shell, then the program will read from *mydata* and write to *myout*.

An **open** statement need not specify a file name. If it refers to a logical unit that is already open, the **blank=** and **form=** specifiers may be redefined without affecting the current file position. Otherwise, if **status = ′scratch′** is specified, a temporary file with a name of the form *tmp.FXXXXXX* will be opened, and, by default, will be deleted when closed or during termination of program execution.

It is an error to try to open an existing file with **status = ′new′** . It is an error to try to open a nonexistent file with **status = ′old′** . By default, **status = ′unknown′** will be assumed, and a file will be created if necessary.

By default, files are positioned at their beginning upon opening, but see *fseek*(3f) and *ioinit*(3f) for alternatives. Existing files are never truncated on opening. Sequentially accessed external files are truncated to the current file position on **close**, **backspace**, or **rewind** only if the last access to the file was a write. An **endfile** always causes such files to be truncated to the current file position.

2.5. Format interpretation

Formats which are in format statements are parsed by the compiler; formats in **read**, **write**, and **print** statements are parsed during execution by the I/O library. Upper as well as lower case characters are recognized in format statements and all the alphabetic arguments to the I/O library routines.

If the external representation of a datum is too large for the field width specified, the specified field is filled with asterisks (∗). On **E**w.d**E**e output, the exponent field will be filled with asterisks if the exponent representation is too large. This will only happen if ‘‘e’’ is zero (see appendix B).

On output, a real value that is truly zero will display as ‘‘0.’’ to distinguish it from a very small nonzero value. If this causes problems for other input systems, the **BZ** edit descriptor may be used to cause the field following the decimal point to be filled with zero’s.

Non-destructive tabbing is implemented for both internal and external formatted I/O. Tabbing left or right on output does not affect previously written portions of a record. Tabbing right on output causes unwritten portions of a record to be filled with blanks. Tabbing right off the end of an input logical record is an error. Tabbing left beyond the beginning of an input logical record leaves the input pointer at the beginning of the record. The format specifier **T** must be followed by a positive non-zero number. If it is

†Periods are deleted because they can not be part of environment variable names in the Bourne shell.

not, it will have a different meaning (see § 3.1).

Tabbing left requires seek ability on the logical unit. Therefore it is not allowed in I/O to a terminal or pipe. Likewise, nondestructive tabbing in either direction is possible only on a unit that can seek. Otherwise tabbing right or spacing with **X** will write blanks on the output.

2.6. List directed output

In formatting list directed output, the I/O system tries to prevent output lines longer than 80 characters. Each external datum will be separated by two spaces. List directed output of **complex** values includes an appropriate comma. List directed output distinguishes between **real** and **double precision** values and formats them differently. Output of a character string that includes ‘‘\n’’ is interpreted reasonably by the output system.

2.7. I/O errors

If I/O errors are not trapped by the user’s program an appropriate error message will be written to ‘‘stderr’’ before aborting. An error number will be printed in ‘‘[ ]’’ along with a brief error message showing the logical unit and I/O state. Error numbers < 100 refer to UNIX errors, and are described in the introduction to chapter 2 of the UNIX Programmer’s Manual. Error numbers ≥ 100 come from the I/O library, and are described further in the appendix to this writeup‡. For internal I/O, part of the string will be printed with ‘‘|’’ at the current position in the string. For external I/O, part of the current record will be displayed if the error was caused during reading from a file that can backspace.

3. Non-‘‘ANSI Standard’’ extensions

Several extensions have been added to the I/O system to provide for functions omitted or poorly defined in the standard. Programmers should be aware that these are non-portable.

3.1. Format specifiers

B is an acceptable edit control specifier. It causes return to the logical unit’s default mode of blank interpretation. This is consistent with **S** which returns to default sign control.

P by itself is equivalent to **0P** . It resets the scale factor to the default value, 0.

The form of the **E**w.d**E**e format specifier has been extended to **D** also. The form **E**w.d.e is allowed but is not standard. The ‘‘e’’ field specifies the minimum number of digits or spaces in the exponent field on output. If the value of the exponent is too large, the exponent notation **e** or **d** will be dropped from the output to allow one more character position. If this is still not adequate, the ‘‘e’’ field will be filled with asterisks (∗). The default value for ‘‘e’’ is 2.

An additional form of tab control specification has been added. The ANSI standard forms **TR**n, **TL**n, and **T**n are supported where *n* is a positive non-zero number. If **T** or n**T** is specified, tabbing will be to the next (or n-th) 8-column tab stop. Thus columns of alphanumerics can be lined up without counting.

A format control specifier has been added to suppress the newline at the end of the last record of a formatted sequential write. The specifier is a dollar sign ($). It is constrained by the same rules as the colon (:). It is used typically for console prompts. For example:

```
write (*, "('enter value for x: ',$)")
read (*,*) x
```

Radices other than 10 can be specified for formatted integer I/O conversion. The specifier is patterned after **P,** the scale factor for floating point conversion. It remains in effect until another radix is specified or format interpretation is complete. The specifier is defined as [n]**R** where $2 \le n \le 36$. If *n* is omitted, the default decimal radix is restored.

‡ On many systems, these are also available in *help f77 io_err_msgs*.

The format specifier **O**m.n may be used for an octal conversion; it is equivalent to 8**R,I**m.n,10**R**. Similarly, **Z**m.n is equivalent to 16**R,I**m.n,10**R** and may be used for an hexadecimal conversion;

In conjunction with the above, a sign control specifier has been added to cause integer values to be interpreted as unsigned during output conversion. The specifier is **SU** and remains in effect until another sign control specifier is encountered, or format interpretation is complete.† Radix and ‘‘unsigned’’ specifiers could be used to format a hexadecimal dump, as follows:

```
2000  format ( SU, 8Z10.8 )
```

3.2. Print files

The ANSI standard is ambiguous regarding the definition of a ‘‘print’’ file. Since UNIX has no default ‘‘print’’ file, an additional **form=** specifier is now recognized in the **open** statement. Specifying **form = ′print′** implies **formatted** and enables vertical format control for that logical unit (see § 2.3). Vertical format control is interpreted only on sequential formatted writes to a ‘‘print’’ file.

The **inquire** statement will return **print** in the **form=** string variable for logical units opened as ‘‘print’’ files. It will return -1 for the unit number of an unconnected file.

If a logical unit is already open, an **open** statement including the **form=** option or the **blank=** option will do nothing but re-define those options. This instance of the **open** statement need not include the file name, and must not include a file name if **unit=** refers to a standard input or output. Therefore, to re-define the standard output as a ‘‘print’’ file, use:

```
open (unit=6, form=′print′)
```

3.3. Scratch files

A **close** statement with **status = ′keep′** may be specified for temporary files. This is the default for all other files. Remember to get the scratch file’s real name, using **inquire** , if you want to re-open it later.

3.4. List directed I/O

List directed read has been modified to allow tab characters wherever blanks are allowed. It also allows input of a string not enclosed in quotes. The string must not start with a digit or quote, and can not contain any separators (‘‘,’’, ‘‘/’’, blank or tab). A newline will terminate the string unless escaped with \. Any string not meeting the above restrictions must be enclosed in quotes (‘‘ " ’’ or ‘‘ ′ ’’).

Internal list directed I/O has been implemented. During internal list reads, bytes are consumed until the iolist is satisfied, or the ‘‘end-of-file’’ is reached. During internal list writes, records are filled until the iolist is satisfied. The length of an internal array element should be at least 20 bytes to avoid logical record overflow when writing double precision values. Internal list read was implemented to make command line decoding easier. Internal list write should be avoided.

3.5. Namelist I/O

Namelist I/O is a common extension in Fortran systems. The **f77** version was designed to be compatible with other vendors versions; it is described in ‘‘A Portable Fortran 77 Compiler’’, by Feldman and Weinberger, August, 1985.

---

†Note: Unsigned integer values greater than (2∗∗31 - 1), can be read and written using **SU**. However they can not be used in computations because Fortran uses signed arithmetic and such values appear to the arithmetic unit as negative numbers.

4. Running older programs

Traditional Fortran environments usually assume carriage control on all logical units, usually interpret blank spaces on input as ‘‘0’’s, and often provide attachment of global file names to logical units at run time. There are several routines in the I/O library to provide these functions.

4.1. Traditional unit control parameters

If a program reads and writes only units 5 and 6, then including **–lI66** in the f77 command will cause carriage control to be interpreted on output and cause blanks to be zeros on input without further modification of the program. If this is not adequate, the routine *ioinit*(3f) can be called to specify control parameters separately, including whether files should be positioned at their beginning or end upon opening.

4.2. Ioinit()

Ioinit(3f) can be used to attach logical units to specific files at run time, and to set global parameters for the I/O system. It will look for names of a user specified form in the environment and open the corresponding logical unit for **sequential formatted** I/O. Names must be of the form **PREFIX***nn* where **PREFIX** is specified in the call to *ioinit* and *nn* is the logical unit to be opened. Unit numbers < 10 must include the leading ‘‘0’’.

Ioinit should prove adequate for most programs as written. However, it is written in Fortran–77 specifically so that it may serve as an example for similar user-supplied routines. A copy may be retrieved by ‘‘ar x /usr/lib/libU77.a ioinit.f’’. See § 2.4 for another way to override program file names through environment variables.

5. Magnetic tape I/O

Because the I/O library uses stdio buffering, reading or writing magnetic tapes should be done with great caution, or avoided if possible. A set of routines has been provided to read and write arbitrary sized buffers to or from tape directly. The buffer must be a **character** object. **Internal** I/O can be used to fill or interpret the buffer. These routines do not use normal Fortran I/O processing and do not obey Fortran I/O rules. See *topen*(3f).

6. Caveat Programmer

The I/O library is extremely complex yet we believe there are few bugs left. We’ve tried to make the system as correct as possible according to the ANSI X3.9–1978 document and keep it compatible with the UNIX file system. Exceptions to the standard are noted in appendix B.

Appendix A

I/O Library Error Messages

The following error messages are generated by the I/O library. The error numbers are returned in the **iostat=** variable. Error numbers < 100 are generated by the UNIX kernel. See the introduction to chapter 2 of the UNIX Programmers Manual for their description.

100 *error in format*
See error message output for the location of the error in the format. Can be caused by more than 10 levels of nested parentheses, or an extremely long format statement.

101 *illegal unit number*
It is illegal to close logical unit 0. Unit numbers must be between 0 and 99 inclusive.

102 *formatted i/o not allowed*
The logical unit was opened for unformatted I/O.

103 *unformatted i/o not allowed*
The logical unit was opened for formatted I/O.

104 *direct i/o not allowed*
The logical unit was opened for sequential access, or the logical record length was specified as 0.

105 *sequential i/o not allowed*
The logical unit was opened for direct access I/O.

106 *can't backspace file*
The file associated with the logical unit can't seek. May be a device or a pipe.

107 *off beginning of record*
The format specified a left tab beyond the beginning of an internal input record.

108 *can't stat file*
The system can't return status information about the file. Perhaps the directory is unreadable.

109 *no* ∗ *after repeat count*
Repeat counts in list directed I/O must be followed by an ∗ with no blank spaces.

110 *off end of record*
A formatted write tried to go beyond the logical end-of-record. An unformatted read or write will also cause this.

111 *truncation failed*
The truncation of an external sequential file on **close**, **backspace**, **rewind**, or **endfile** failed.

112 *incomprehensible list input*
List input has to be just right.

113 *out of free space*
The library dynamically creates buffers for internal use. You ran out of memory for this. Your program is too big!

114 *unit not connected*
The logical unit was not open.

115 *invalid data for integer format term*
Only spaces, a leading sign and digits are allowed.

116 *invalid data for logical format term*
Legal input consists of spaces (optional), a period (optional), and then a ‘‘t’’, ‘‘T’’, ‘‘f’’, or ‘‘F’’.

117 *′new′ file exists*
You tried to open an existing file with ‘‘status=′new′’’.

118 *can’t find ′old′ file*
You tried to open a non-existent file with ‘‘status=′old′’’.

119 *opening too many files or unknown system error*
Either you are trying to open too many files simultaneously or there has been an undetected system error.

120 *requires seek ability*
Direct access requires seek ability. Sequential unformatted I/O requires seek ability on the file due to the special data structure required. Tabbing left also requires seek ability.

121 *illegal argument*
Certain arguments to **open**, etc. will be checked for legitimacy. Often only non-default forms are looked for.

122 *negative repeat count*
The repeat count for list directed input must be a positive integer.

123 *illegal operation for unit*
An operation was requested for a device associated with the logical unit which was not possible. This error is returned by the tape I/O routines if attempting to read past end-of-tape, etc.

124 *invalid data for d, e, f or g format term*
Input data must be legal.

125 *illegal input for namelist*
Column one of input is ignored, the namelist name must match, the variables must be in the namelist, and the data must be of the right type.

Appendix B

Exceptions to the ANSI Standard

A few exceptions to the ANSI standard remain.

Vertical format control

The ‘‘+’’ carriage control specifier is not fully implemented (see § 2.3). It would be difficult to implement it correctly and still provide UNIX-like file I/O.

Furthermore, the carriage control implementation is asymmetrical. A file written with carriage control interpretation can not be read again with the same characters in column 1.

An alternative to interpreting carriage control internally is to run the output file through a ‘‘Fortran output filter’’ before printing. This filter could recognize a much broader range of carriage control and include terminal dependent processing. One such filter is *fpr*(1).

Default files

Files created by default use of **endfile** statements are opened for **sequential formatted** access. There is no way to redefine such a file to allow **direct** or **unformatted** access.

Lower case strings

It is not clear if the ANSI standard requires internally generated strings to be upper case or not. As currently written, the **inquire** statement will return lower case strings for any alphanumeric data.

Exponent representation on Ew.dEe output

If the field width for the exponent is too small, the standard allows dropping the exponent character but only if the exponent is > 99. This system does not enforce that restriction. Further, the standard implies that the entire field, ‘‘w’’, should be filled with asterisks if the exponent can not be displayed. This system fills only the exponent field in the above case since that is more diagnostic.

Pre-connection of files

The standard says units must be pre-connected to files before the program starts or must be explicitly opened. Instead, the I/O library connects the unit to a file on its first use in a **read**, **write**, **print**, or **endfile** statement. Thus **inquire** by unit can not tell prior to a unit number use the characteristics or name of the file corresponding to a unit.

Table of Contents

1. Fortran I/O 1
1.1. Types of I/O and logical records 1
1.1.1. Direct access external I/O 1
1.1.2. Sequential access external I/O 1
1.1.3. List directed and namelist sequential external I/O 1
1.1.4. Internal I/O 1
1.2. I/O execution 2
2. Implementation details 2
2.1. Number of logical units 2
2.2. Standard logical units 2
2.3. Vertical format control 2
2.4. File names and the open statement 3
2.5. Format interpretation 3
2.6. List directed output 4
2.7. I/O errors 4
3. Non-‘‘ANSI Standard’’ extensions 4
3.1. Format specifiers 4
3.2. Print files 5
3.3. Scratch files 5
3.4. List directed I/O 5
3.5. Namelist I/O 5
4. Running older programs 6
4.1. Traditional unit control parameters 6
4.2. Ioinit() 6
5. Magnetic tape I/O 6
6. Caveat Programmer 6
Appendix A: I/O Library Error Messages 7
Appendix B: Exceptions to the ANSI Standard 9

Introduction to the f77 I/O Library

David L. Wasley

University of California, Berkeley
Berkeley, California 94720

ABSTRACT

The f77 I/O Library implements ANSI 1978 Fortran standard input and output with a few minor exceptions. Where the standard is vague, we have tried to provide flexibility within the constraints of the UNIX† operating system. The I/O Library was written originally by Peter J. Weinberger at Bell Labs. A number of logical extensions and enhancements have been provided by this author.

April 1983

Revised for 4.3 BSD, September, 1985 by J. Berkman.

† UNIX is a trademark of Bell Laboratories.

Introduction to the f77 I/O Library

David L. Wasley

University of California, Berkeley
Berkeley, California 94720

The f77 I/O library, libI77.a, includes routines to perform all of the standard types of Fortran input and output. Several enhancements and extensions to Fortran I/O have been added. The f77 library routines use the C stdio library routines to provide efficient buffering for file I/O.

1. Fortran I/O

The requirements of the ANSI standard impose significant overhead on programs that do large amounts of I/O. Formatted I/O can be very ‘‘expensive’’ while direct access binary I/O is usually very efficient. Because of the complexity of Fortran I/O, some general concepts deserve clarification.

1.1. Types of I/O and logical records

There are four forms of I/O: **formatted, unformatted, list directed,** and **namelist.** The last two are related to formatted but do not obey all the rules for formatted I/O. There are two types of ‘‘files’’: **external** and **internal** and two modes of access to files: **direct** and **sequential.** The definition of a logical record depends upon the combination of I/O form, file type, and access mode specified by the Fortran I/O statement.

1.1.1. Direct access external I/O

A logical record in a **direct** access **external** file is a string of bytes of a length specified when the file is opened. Read and write statements must not specify logical records longer than the original record size definition. Shorter logical records are allowed. **Unformatted** direct writes leave the unfilled part of the record undefined. **Formatted** direct writes cause the unfilled record to be padded with blanks.

1.1.2. Sequential access external I/O

Logical records in **sequentially** accessed **external** files may be of arbitrary and variable length. Logical record length for **unformatted** sequential files is determined by the size of items in the iolist. The requirements of this form of I/O cause the external physical record size to be somewhat larger than the logical record size. For **formatted** write statements, logical record length is determined by the format statement interacting with the iolist at execution time. The ‘‘newline’’ character is the logical record delimiter. Formatted sequential access causes one or more logical records ending with ‘‘newline’’ characters to be read or written.

1.1.3. List directed and namelist sequential external I/O

Logical record length for **list directed** and **namelist** I/O is relatively meaningless. On output, the record length is dependent on the magnitude of the data items. On input, the record length is determined by the data types and the file contents. By ANSI definition, a slash, ‘‘/’’, terminates execution of a list directed input operation. Namelist input is terminated by ‘‘&end’’ or ‘‘$end’’ (depending on whether the character before the namelist name was ‘‘&’’ or ‘‘$’’).

1.1.4. Internal I/O

The logical record length for an **internal** read or write is the length of the character variable or array element. Thus a simple character variable is a single logical record. A character variable array is similar to a fixed length direct access file, and obeys the same rules. **Unformatted** and **namelist** I/O are not allowed on ‘‘internal’’ files.

1.2. I/O execution

Note that each execution of a Fortran **unformatted** I/O statement causes a single logical record to be read or written. Each execution of a Fortran **formatted** I/O statement causes one or more logical records to be read or written.

A slash, ‘‘/’’, will terminate assignment of values to the input list during **list directed** input and the remainder of the current input line is skipped. The standard is rather vague on this point but seems to require that a new external logical record be found at the start of any formatted input. Therefore data following the slash is ignored and may be used to comment the data file.

Direct access list directed I/O is not allowed. **Unformatted internal** I/O is not allowed. **Namelist** I/O is allowed only with **external sequential** files. All other flavors of I/O are allowed, although some are not part of the ANSI standard.

Any I/O statement may include an **err=** clause to specify an alternative branch to be taken on errors and/or an **iostat=** clause to return the specific error code. Any error detected during I/O processing will cause the program to abort unless either **err=** or **iostat=** has been specificed in the program. Read statements may include **end=** to branch on end-of-file. The end-of-file indication for that logical unit may be reset with a **backspace** statement. File position and the value of I/O list items is undefined following an error.

2. Implementation details

Some details of the current implementation may be useful in understanding constraints on Fortran I/O.

2.1. Number of logical units

Unit numbers must be in the range 0 − 99. The maximum number of logical units that a program may have open at one time is the same as the UNIX system limit, currently 48.

2.2. Standard logical units

By default, logical units 0, 5, and 6 are opened to ‘‘stderr’’, ‘‘stdin’’, and ‘‘stdout’’ respectively. However they can be re-defined with an **open** statement. To preserve error reporting, it is an error to close logical unit 0 although it may be reopened to another file.

If you want to open the default file name for any preconnected logical unit, remember to **close** the unit first. Redefining the standard units may impair normal console I/O. An alternative is to use shell re-direction to externally re-define the above units. To re-define default blank control or format of the standard input or output files, use the **open** statement specifying the unit number and no file name (see § 2.4).

The standard units, 0, 5, and 6, are named internally ‘‘stderr’’, ‘‘stdin’’, and ‘‘stdout’’ respectively. These are not actual file names and can not be used for opening these units. **Inquire** will not return these names and will indicate that the above units are not named unless they have been opened to real files. The names are meant to make error reporting more meaningful.

2.3. Vertical format control

Simple vertical format control is implemented. The logical unit must be opened for sequential access with **form = ′print′** (see § 3.2). Control codes ‘‘0’’ and ‘‘1’’ are replaced in the output file with ‘‘\n’’ and ‘‘\f’’ respectively. The control character ‘‘+’’ is not implemented and, like any other character in the first position of a record written to a ‘‘print’’ file, is dropped. The **form = ′print′** mode does not recognize vertical format control for **direct formatted, list directed,** or **namelist** output.

An alternative is to use the filter *fpr*(1) for vertical format control. It replaces ‘‘0’’ and ‘‘1’’ by ‘‘\n’’ and ‘‘\f’’ respectively, and implements the ‘‘+’’ control code. Unlike **form = ′print′** which drops unrecognized form control characters, *fpr* copies those characters to the output file.

2.4. File names and the open statement

A file name may be specified in an **open** statement for the logical unit. If a logical unit is opened by an **open** statement which does not specify a file name, or it is opened implicitly by the execution of a **read**, **write**, or **endfile** statement, then the default file name is *fort.N* where *N* is the logical unit number. Before opening the file, the library checks for an environment variable with a name identical to the tail of the file name with periods removed.† If it finds such an environment variable, it uses its value as the actual name of the file. For example, a program containing:

```
open(32,file='/usr/guest/census/data.d')
read(32,100) vec
write(44) vec
```

normally will read from */usr/guest/census/data.d* and write to *fort.44* in the current directory. If the environment variables *datad* and *fort44* are set, e.g.:

```
% setenv datad mydata
% setenv fort44 myout
```

in the C shell or:

```
$ datad=mydata
$ fort44=myout
$ export datad fort44
```

in the Bourne shell, then the program will read from *mydata* and write to *myout*.

An **open** statement need not specify a file name. If it refers to a logical unit that is already open, the **blank=** and **form=** specifiers may be redefined without affecting the current file position. Otherwise, if **status = ′scratch′** is specified, a temporary file with a name of the form *tmp.FXXXXXX* will be opened, and, by default, will be deleted when closed or during termination of program execution.

It is an error to try to open an existing file with **status = ′new′** . It is an error to try to open a nonexistent file with **status = ′old′** . By default, **status = ′unknown′** will be assumed, and a file will be created if necessary.

By default, files are positioned at their beginning upon opening, but see *fseek*(3f) and *ioinit*(3f) for alternatives. Existing files are never truncated on opening. Sequentially accessed external files are truncated to the current file position on **close**, **backspace**, or **rewind** only if the last access to the file was a write. An **endfile** always causes such files to be truncated to the current file position.

2.5. Format interpretation

Formats which are in format statements are parsed by the compiler; formats in **read**, **write**, and **print** statements are parsed during execution by the I/O library. Upper as well as lower case characters are recognized in format statements and all the alphabetic arguments to the I/O library routines.

If the external representation of a datum is too large for the field width specified, the specified field is filled with asterisks (∗). On **E**w.d**E**e output, the exponent field will be filled with asterisks if the exponent representation is too large. This will only happen if ‘‘e’’ is zero (see appendix B).

On output, a real value that is truly zero will display as ‘‘0.’’ to distinguish it from a very small nonzero value. If this causes problems for other input systems, the **BZ** edit descriptor may be used to cause the field following the decimal point to be filled with zero’s.

Non-destructive tabbing is implemented for both internal and external formatted I/O. Tabbing left or right on output does not affect previously written portions of a record. Tabbing right on output causes unwritten portions of a record to be filled with blanks. Tabbing right off the end of an input logical record is an error. Tabbing left beyond the beginning of an input logical record leaves the input pointer at the beginning of the record. The format specifier **T** must be followed by a positive non-zero number. If it is

---

†Periods are deleted because they can not be part of environment variable names in the Bourne shell.

not, it will have a different meaning (see § 3.1).

Tabbing left requires seek ability on the logical unit. Therefore it is not allowed in I/O to a terminal or pipe. Likewise, nondestructive tabbing in either direction is possible only on a unit that can seek. Otherwise tabbing right or spacing with **X** will write blanks on the output.

2.6. List directed output

In formatting list directed output, the I/O system tries to prevent output lines longer than 80 characters. Each external datum will be separated by two spaces. List directed output of **complex** values includes an appropriate comma. List directed output distinguishes between **real** and **double precision** values and formats them differently. Output of a character string that includes ‘‘\n’’ is interpreted reasonably by the output system.

2.7. I/O errors

If I/O errors are not trapped by the user’s program an appropriate error message will be written to ‘‘stderr’’ before aborting. An error number will be printed in ‘‘[ ]’’ along with a brief error message showing the logical unit and I/O state. Error numbers < 100 refer to UNIX errors, and are described in the introduction to chapter 2 of the UNIX Programmer’s Manual. Error numbers ≥ 100 come from the I/O library, and are described further in the appendix to this writeup‡. For internal I/O, part of the string will be printed with ‘‘|’’ at the current position in the string. For external I/O, part of the current record will be displayed if the error was caused during reading from a file that can backspace.

3. Non-‘‘ANSI Standard’’ extensions

Several extensions have been added to the I/O system to provide for functions omitted or poorly defined in the standard. Programmers should be aware that these are non-portable.

3.1. Format specifiers

B is an acceptable edit control specifier. It causes return to the logical unit’s default mode of blank interpretation. This is consistent with **S** which returns to default sign control.

P by itself is equivalent to **0P** . It resets the scale factor to the default value, 0.

The form of the **E**w.d**E**e format specifier has been extended to **D** also. The form **E**w.d.e is allowed but is not standard. The ‘‘e’’ field specifies the minimum number of digits or spaces in the exponent field on output. If the value of the exponent is too large, the exponent notation **e** or **d** will be dropped from the output to allow one more character position. If this is still not adequate, the ‘‘e’’ field will be filled with asterisks (∗). The default value for ‘‘e’’ is 2.

An additional form of tab control specification has been added. The ANSI standard forms **TR**n, **TL**n, and **T**n are supported where *n* is a positive non-zero number. If **T** or n**T** is specified, tabbing will be to the next (or n-th) 8-column tab stop. Thus columns of alphanumerics can be lined up without counting.

A format control specifier has been added to suppress the newline at the end of the last record of a formatted sequential write. The specifier is a dollar sign ($). It is constrained by the same rules as the colon (:). It is used typically for console prompts. For example:

```
write (∗, "(′enter value for x: ′,$)")
read (∗,∗) x
```

Radices other than 10 can be specified for formatted integer I/O conversion. The specifier is patterned after **P,** the scale factor for floating point conversion. It remains in effect until another radix is specified or format interpretation is complete. The specifier is defined as [n]**R** where $2 \le n \le 36$. If *n* is omitted, the default decimal radix is restored.

‡ On many systems, these are also available in *help f77 io_err_msgs*.

The format specifier **O**m.n may be used for an octal conversion; it is equivalent to 8**R,I**m.n,10**R**. Similarly, **Z**m.n is equivalent to 16**R,I**m.n,10**R** and may be used for an hexadecimal conversion;

In conjunction with the above, a sign control specifier has been added to cause integer values to be interpreted as unsigned during output conversion. The specifier is **SU** and remains in effect until another sign control specifier is encountered, or format interpretation is complete.† Radix and ‘‘unsigned’’ specifiers could be used to format a hexadecimal dump, as follows:

```
2000  format ( SU, 8Z10.8 )
```

3.2. Print files

The ANSI standard is ambiguous regarding the definition of a ‘‘print’’ file. Since UNIX has no default ‘‘print’’ file, an additional **form=** specifier is now recognized in the **open** statement. Specifying **form = ′print′** implies **formatted** and enables vertical format control for that logical unit (see § 2.3). Vertical format control is interpreted only on sequential formatted writes to a ‘‘print’’ file.

The **inquire** statement will return **print** in the **form=** string variable for logical units opened as ‘‘print’’ files. It will return -1 for the unit number of an unconnected file.

If a logical unit is already open, an **open** statement including the **form=** option or the **blank=** option will do nothing but re-define those options. This instance of the **open** statement need not include the file name, and must not include a file name if **unit=** refers to a standard input or output. Therefore, to re-define the standard output as a ‘‘print’’ file, use:

```
open (unit=6, form=′print′)
```

3.3. Scratch files

A **close** statement with **status = ′keep′** may be specified for temporary files. This is the default for all other files. Remember to get the scratch file’s real name, using **inquire** , if you want to re-open it later.

3.4. List directed I/O

List directed read has been modified to allow tab characters wherever blanks are allowed. It also allows input of a string not enclosed in quotes. The string must not start with a digit or quote, and can not contain any separators (‘‘,’’, ‘‘/’’, blank or tab). A newline will terminate the string unless escaped with \. Any string not meeting the above restrictions must be enclosed in quotes (‘‘ " ’’ or ‘‘ ′ ’’).

Internal list directed I/O has been implemented. During internal list reads, bytes are consumed until the iolist is satisfied, or the ‘‘end-of-file’’ is reached. During internal list writes, records are filled until the iolist is satisfied. The length of an internal array element should be at least 20 bytes to avoid logical record overflow when writing double precision values. Internal list read was implemented to make command line decoding easier. Internal list write should be avoided.

3.5. Namelist I/O

Namelist I/O is a common extension in Fortran systems. The **f77** version was designed to be compatible with other vendors versions; it is described in ‘‘A Portable Fortran 77 Compiler’’, by Feldman and Weinberger, August, 1985.

---

†Note: Unsigned integer values greater than (2∗∗31 - 1), can be read and written using **SU**. However they can not be used in computations because Fortran uses signed arithmetic and such values appear to the arithmetic unit as negative numbers.

4. Running older programs

Traditional Fortran environments usually assume carriage control on all logical units, usually interpret blank spaces on input as ‘‘0’’s, and often provide attachment of global file names to logical units at run time. There are several routines in the I/O library to provide these functions.

4.1. Traditional unit control parameters

If a program reads and writes only units 5 and 6, then including **–lI66** in the f77 command will cause carriage control to be interpreted on output and cause blanks to be zeros on input without further modification of the program. If this is not adequate, the routine *ioinit*(3f) can be called to specify control parameters separately, including whether files should be positioned at their beginning or end upon opening.

4.2. Ioinit()

Ioinit(3f) can be used to attach logical units to specific files at run time, and to set global parameters for the I/O system. It will look for names of a user specified form in the environment and open the corresponding logical unit for **sequential formatted** I/O. Names must be of the form **PREFIX***nn* where **PREFIX** is specified in the call to *ioinit* and *nn* is the logical unit to be opened. Unit numbers < 10 must include the leading ‘‘0’’.

Ioinit should prove adequate for most programs as written. However, it is written in Fortran–77 specifically so that it may serve as an example for similar user-supplied routines. A copy may be retrieved by ‘‘ar x /usr/lib/libU77.a ioinit.f’’. See § 2.4 for another way to override program file names through environment variables.

5. Magnetic tape I/O

Because the I/O library uses stdio buffering, reading or writing magnetic tapes should be done with great caution, or avoided if possible. A set of routines has been provided to read and write arbitrary sized buffers to or from tape directly. The buffer must be a **character** object. **Internal** I/O can be used to fill or interpret the buffer. These routines do not use normal Fortran I/O processing and do not obey Fortran I/O rules. See *topen*(3f).

6. Caveat Programmer

The I/O library is extremely complex yet we believe there are few bugs left. We’ve tried to make the system as correct as possible according to the ANSI X3.9–1978 document and keep it compatible with the UNIX file system. Exceptions to the standard are noted in appendix B.

Appendix A

I/O Library Error Messages

The following error messages are generated by the I/O library. The error numbers are returned in the **iostat=** variable. Error numbers < 100 are generated by the UNIX kernel. See the introduction to chapter 2 of the UNIX Programmers Manual for their description.

100 *error in format*
See error message output for the location of the error in the format. Can be caused by more than 10 levels of nested parentheses, or an extremely long format statement.

101 *illegal unit number*
It is illegal to close logical unit 0. Unit numbers must be between 0 and 99 inclusive.

102 *formatted i/o not allowed*
The logical unit was opened for unformatted I/O.

103 *unformatted i/o not allowed*
The logical unit was opened for formatted I/O.

104 *direct i/o not allowed*
The logical unit was opened for sequential access, or the logical record length was specified as 0.

105 *sequential i/o not allowed*
The logical unit was opened for direct access I/O.

106 *can’t backspace file*
The file associated with the logical unit can’t seek. May be a device or a pipe.

107 *off beginning of record*
The format specified a left tab beyond the beginning of an internal input record.

108 *can’t stat file*
The system can’t return status information about the file. Perhaps the directory is unreadable.

109 *no * after repeat count*
Repeat counts in list directed I/O must be followed by an * with no blank spaces.

110 *off end of record*
A formatted write tried to go beyond the logical end-of-record. An unformatted read or write will also cause this.

111 *truncation failed*
The truncation of an external sequential file on **close**, **backspace**, **rewind**, or **endfile** failed.

112 *incomprehensible list input*
List input has to be just right.

113 *out of free space*
The library dynamically creates buffers for internal use. You ran out of memory for this. Your program is too big!

114 *unit not connected*
The logical unit was not open.

115 *invalid data for integer format term*
Only spaces, a leading sign and digits are allowed.

116 *invalid data for logical format term*
Legal input consists of spaces (optional), a period (optional), and then a ‘‘t’’, ‘‘T’’, ‘‘f’’, or ‘‘F’’.

117 *′new′ file exists*
You tried to open an existing file with ‘‘status=′new′’’.

118 *can’t find ′old′ file*
You tried to open a non-existent file with ‘‘status=′old′’’.

119 *opening too many files or unknown system error*
Either you are trying to open too many files simultaneously or there has been an undetected system error.

120 *requires seek ability*
Direct access requires seek ability. Sequential unformatted I/O requires seek ability on the file due to the special data structure required. Tabbing left also requires seek ability.

121 *illegal argument*
Certain arguments to **open**, etc. will be checked for legitimacy. Often only non-default forms are looked for.

122 *negative repeat count*
The repeat count for list directed input must be a positive integer.

123 *illegal operation for unit*
An operation was requested for a device associated with the logical unit which was not possible. This error is returned by the tape I/O routines if attempting to read past end-of-tape, etc.

124 *invalid data for d, e, f or g format term*
Input data must be legal.

125 *illegal input for namelist*
Column one of input is ignored, the namelist name must match, the variables must be in the namelist, and the data must be of the right type.

Appendix B

Exceptions to the ANSI Standard

A few exceptions to the ANSI standard remain.

Vertical format control

The ‘‘+’’ carriage control specifier is not fully implemented (see § 2.3). It would be difficult to implement it correctly and still provide UNIX-like file I/O.

Furthermore, the carriage control implementation is asymmetrical. A file written with carriage control interpretation can not be read again with the same characters in column 1.

An alternative to interpreting carriage control internally is to run the output file through a ‘‘Fortran output filter’’ before printing. This filter could recognize a much broader range of carriage control and include terminal dependent processing. One such filter is *fpr*(1).

Default files

Files created by default use of **endfile** statements are opened for **sequential formatted** access. There is no way to redefine such a file to allow **direct** or **unformatted** access.

Lower case strings

It is not clear if the ANSI standard requires internally generated strings to be upper case or not. As currently written, the **inquire** statement will return lower case strings for any alphanumeric data.

Exponent representation on Ew.dEe output

If the field width for the exponent is too small, the standard allows dropping the exponent character but only if the exponent is > 99. This system does not enforce that restriction. Further, the standard implies that the entire field, ‘‘w’’, should be filled with asterisks if the exponent can not be displayed. This system fills only the exponent field in the above case since that is more diagnostic.

Pre-connection of files

The standard says units must be pre-connected to files before the program starts or must be explicitly opened. Instead, the I/O library connects the unit to a file on its first use in a **read**, **write**, **print**, or **endfile** statement. Thus **inquire** by unit can not tell prior to a unit number use the characteristics or name of the file corresponding to a unit.

Table of Contents

1. Fortran I/O 1
   1.1. Types of I/O and logical records 1
      1.1.1. Direct access external I/O 1
      1.1.2. Sequential access external I/O 1
      1.1.3. List directed and namelist sequential external I/O 1
      1.1.4. Internal I/O 1
   1.2. I/O execution 2
2. Implementation details 2
   2.1. Number of logical units 2
   2.2. Standard logical units 2
   2.3. Vertical format control 2
   2.4. File names and the open statement 3
   2.5. Format interpretation 3
   2.6. List directed output 4
   2.7. I/O errors 4
3. Non-‘‘ANSI Standard’’ extensions 4
   3.1. Format specifiers 4
   3.2. Print files 5
   3.3. Scratch files 5
   3.4. List directed I/O 5
   3.5. Namelist I/O 5
4. Running older programs 6
   4.1. Traditional unit control parameters 6
   4.2. Ioinit() 6
5. Magnetic tape I/O 6
6. Caveat Programmer 6
Appendix A: I/O Library Error Messages 7
Appendix B: Exceptions to the ANSI Standard 9