The Formatter
A Halfway House for FORTRAN Input/Output Requests

Author:
Steven Saunders, H-P Development Engineer
General Systems Division
Cupertino, Calif.

Abstract:
The FORTRAN Formatter provides a simple intermediate level interface to the MPE File System. The Formatter provides data conversion and editing capabilities not provided by the File System. The benefits of this interface are available to SPL programmers as well as the FORTRAN compiler. The Formatter does have some special limitations and some processing overhead.

This paper provides an overview of the FORTRAN Formatter and its use. The reader is assumed to have some familiarity with FORTRAN input/output and FORMAT statements, the MPE File System, SPL, and the architecture of the HP3000. This paper is intended to be an overview and brief tutorial. The interested reader is referred to the following publications:

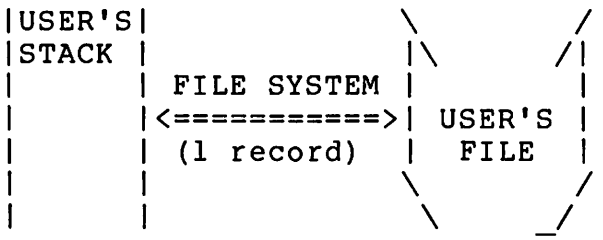
Compiler Library Reference Manual	30000-90028
FORTRAN/3000 Reference Manual	30000-90040
Systems Programming Language Reference Manual	30000-90024
MPE Intrinsics Reference Manual	30000-90010

The example programs given in the appendices of this paper have been compiled and run on a HP3000 series II. The section on the Formatter's operation is intended to provide insight, but not encouragement to try "clever" programming tricks.

General Features of the File System

The MPE File System intrinsics, hereafter simply called the File System, provide the basic mechanism to transfer data between the users's stack and external devices containing the user's files. The File System has a basic unit of data transfer called a record (fig 1). Multiple record transfers will not be discussed in this paper.

fig 1



The organization of records within a user's file is device dependent, but the File System masks most of this device dependence from the user. The device dependence the user does encounter is primarily in the area of access types. For example, not being allowed to input data from a line printer or update records in the middle of a file stored on magnetic tape.

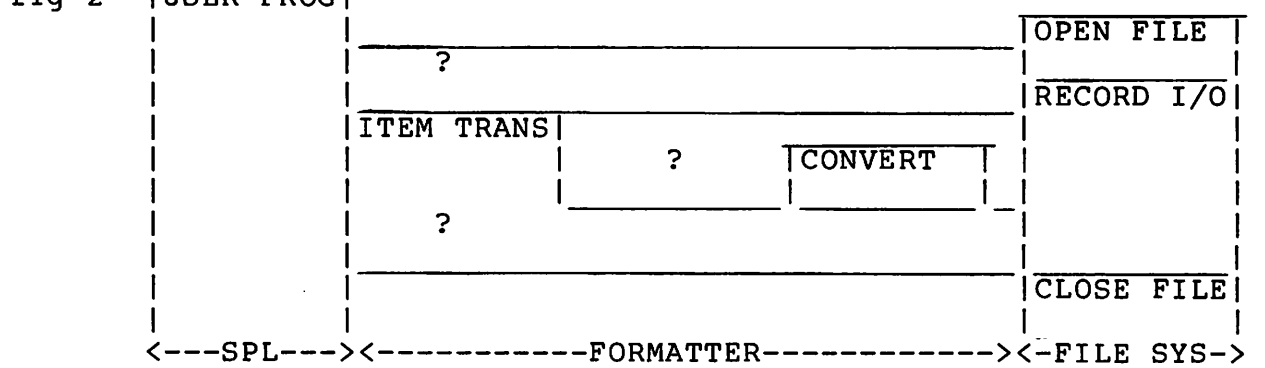
The File System does not concern itself with the contents of each record, thus no data conversions take place. The File System has no interest in the number of data items contained in each record. For example, to input a fixed number of data items from a terminal could require varying numbers of input requests (FREADs). The term "data item" is used to denote any program variable; for example, a simple variable or an array.

The Formatter as an Extension of the File System

We have seen that the File System transfers records without concern for the number or type of data items contained in each record. The user must provide the means to map his data items into File System records, but this may greatly increase the effort to develop his/her application. What is needed is an extention of the File System that will transfer data items and perform data conversions (fig 2).

The Formatter Interface to the File System

fig 2

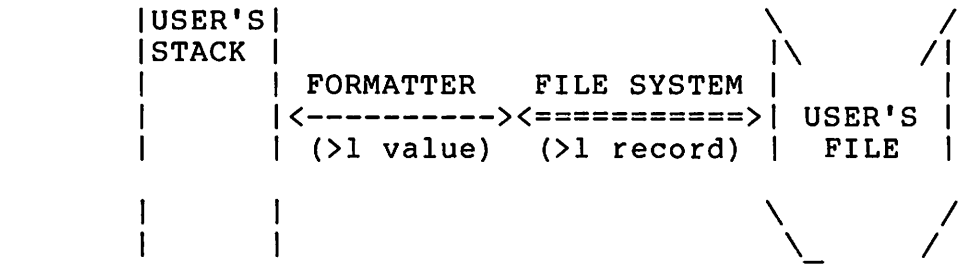


The FORTRAN Formatter intrinsics, hereafter simply called the Formatter, provide this extention of the File System. The Formatter was developed to support FORTRAN input/output requests, as described below, but it will also support intermediate level input/output requests for SPL programs. The term "intermediate level" is used to denote highlevel function without modification to the highlevel language.

General Features of FORTRAN I/O

The input-output statements in FORTRAN/3000 are READ and WRITE. Each has four forms: core-to-core, free-field, formatted, and unformatted. There exist special purpose free-field versions of READ and WRITE called ACCEPT and DISPLAY which will not be discussed further. The definitions of READ and WRITE statements are intended to be machine independent and are in terms of transfer of groups of values, not single records (fig 3).

fig 3



The four forms of READ and WRITE statements are given below along with a brief description of their features.

core-to-core:

READ (character variable,format statement#) data item list
WRITE character variable

This form causes the conversions requested in the format to be performed and the results transferred between the first character variable and the data items.

free-field:

READ (unit#, *) data item list
WRITE

This form causes default conversions to be performed and the results transferred between the FORTRAN unit# and the data items.

formatted:

READ (unit#,format statement#) data item list
WRITE character variable

This form causes the conversions requested in the format to be performed and the results transferred between the FORTRAN unit# and the data items.

unformatted:

READ (unit#) data item list
WRITE

This form causes all conversions to be suppressed and internal bit patterns transferred between the FORTRAN unit# and the data items.

The basic problem implementing all these features is deciding where the work should be done, in the user's object code or in library intrinsics.

Simplified Code Generation

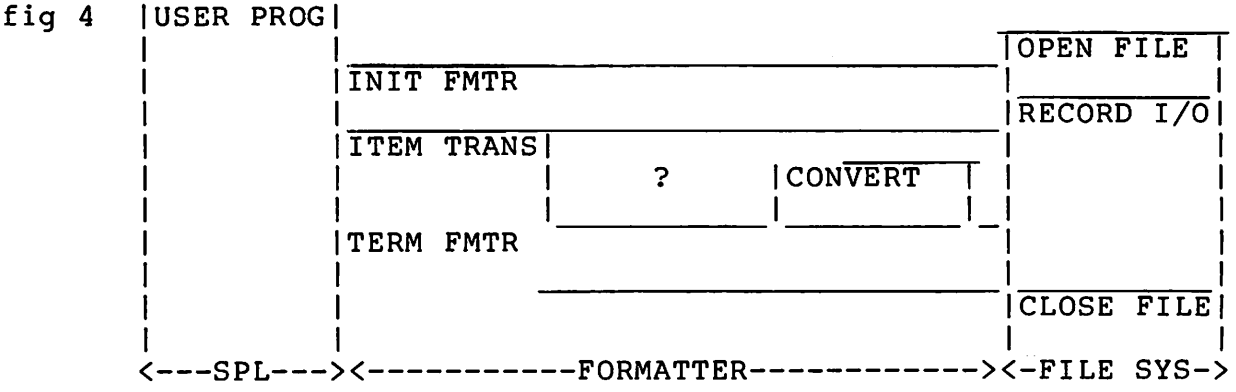
It was decided that the work required by FORTRAN I/O features would be done for FORTRAN/3000 in a set of intrinsics. This is very reasonable when we consider the segmentation features of the HP3000 architecture. This choice also removes much of the object code space requirement for input/output requests. The compiler has less object code to generate, thus it requires less time to compile input/output requests. The sharing of intrinsic code segments and smaller user code segments reduces the overall system memory requirement.

It should be noted that generalized intrinsics cannot perform as well as specially developed input/output code sequences. But the additional effort to develop (or compile) and maintain special input/output code sequences makes generalized intrinsics a better choice in almost all cases.

General Form of Formatter Requests

The SPL programmer using the Formatter to perform input/output must issue the sequence of intrinsic calls detailed below. The basic sequence is: initialize the Formatter (INIT FMTR), transfer one or several data items (ITEM TRANS) possibily with data conversion (CONVERT), and terminate the Formatter (TERM FMTR). See figure 4.

fig 4



initialize the Formatter:

```
                  file#
FMTINIT' (format,unit#,record#,iotype,last)
                  bsize,buffer
```

This intrinsic provides the Formatter with information about the type of input/output request (iotype), what data conversions to use (format), where the source/destination of the transfer is (file#/unit#/buffer), which record to access directly (record#), and what statement follows the entire sequence of intrinsic calls for returns (last). If the request is for input the first record is read by this intrinsic call. If this is the first call for a FORTRAN unit the file is opened by this intrinsic.

transfer a data item:

```
IIO' (loc)        AIIO' (dim,loc)       <<INTEGERs,LOGICALs>>
DIO' (loc)        ADIO' (dim,loc)       <<DOUBLEs>>
RIO' (loc)        ARIO' (dim,loc)       <<REALs>>
LIO' (loc)        ALIO' (dim,loc)       <<LONGs>>
SIO' (slen,loc)   ASIO' (slen,dim,loc)  <<BYTEs>>
```

These intrinsics provide the Formatter with information to transfer single or multiple (A prefixes) data values (loc), on how many multiple data values to transfer (dim), and on how long a character string is (slen). Records may be read or written by this intrinsic call.

terminate the Formatter:

TFORM'

This intrinsic provides for orderly termination of the input/output request. If the request is for output, the last record is written by this intrinsic call. The Formatter then returns to 'last' (parameter in FMTINIT' call).

The SPL programmer interested in using the Formatter is referred to the sample program in Appendix B and the Compiler Library Reference Manual.

Special Programming Considerations

Several special considerations a SPL programmer must keep in mind when using the Formatter are given as points below:

1) Calls to Formatter intrinsics must not appear in SUBROUTINEs. The reason is illustrated in the next section.

2) Calls to Formatter intrinsics must not be made by a program operating in split-stack mode. The Formatter accesses the subsystem area of DL-DB.

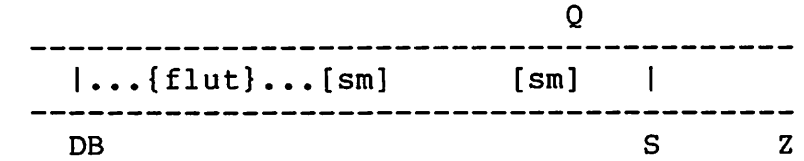
3) The program must not alter the stack marker between Formatter intrinsic calls. This includes interleaving Formatter intrinsic call sequences for different input/output requests. The reason for this is illustrated in the next section.

4) If the programmer wishes to use FORTRAN unit numbers he/she should carefully study the Compiler Library Reference Manual and the sample program in Appendix B.

Stack Games

The Formatter must convert between records, (what the File System uses), and data items, (what the programmer uses). This conversion requires a buffer to hold partial records. The buffer must be allocated on the user's stack because DL-DB has been reserved for other uses. The mechanism of this buffer allocation and its correspondence to the Formatter intrinsic call sequence are explained below. The Formatter initialization call is made and the hardware pushes a stack marker onto the stack and adjust Q, the stack marker register (fig 5).

fig 5



The Formatter initializes its global data area and allocates the buffer by incrementing S, the top of stack register (fig 6). It then places a 'fake' stack marker on the top of the stack and adjusts Q, the underscores to the left of Q denote delta-Q (fig 7).

fig 6

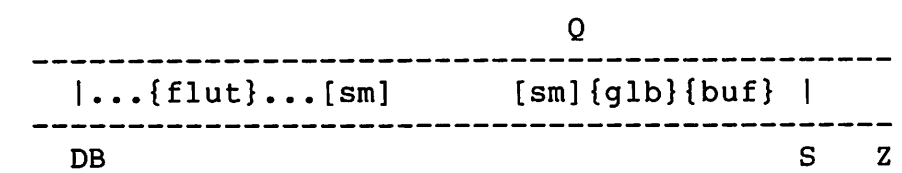
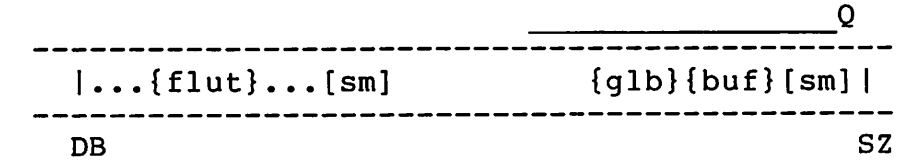


fig 7



The Formatter initialization is complete and the intrinsic returns. The hardware removes the 'fake' stack marker, leaving the buffer on the stack (fig 8). Each successive Formatter intrinsic call causes the hardware to replace the 'fake' stack marker (fig 9).

fig 8

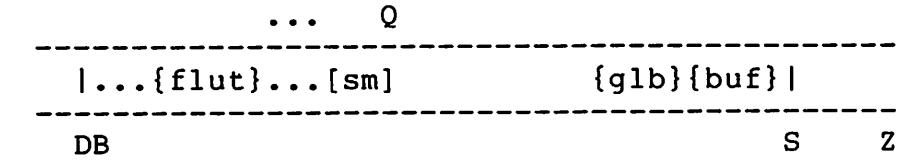
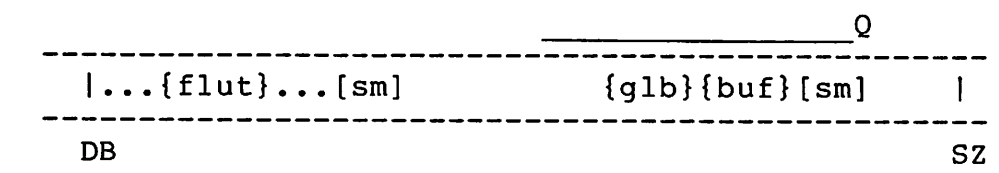
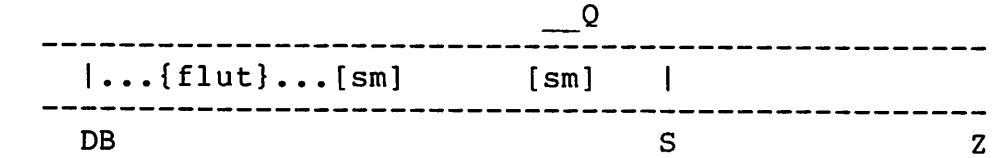


fig 9



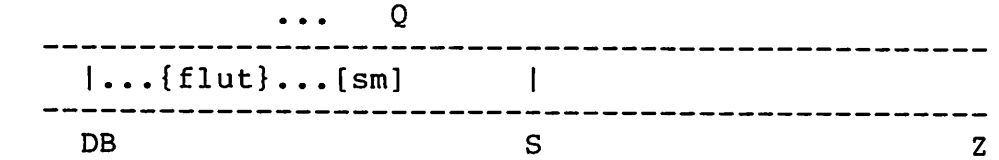
When the Formatter termination call is made or an error occurs, the value of Q is set to the 'real' stack marker (fig 10).

fig 10



The stack returns to its state before the Formatter initialization call (fig 11).

fig 11



I/O List and Format Synchronization

There is no need to synchronize a format and i/o list, or data item transfers, for either unformatted or free-field requests. Thus this section applies only to core-to-core and formatted requests.

step 0: process format until a data value must be transferred, save scan position and return

The Formatter then repeats the following steps for each Formatter call until TFORM' is called or an error occurs.

step 1: convert data value as per format specification, transfer converted value
step 2: process format until a data value must be transferred, save scan position
step 3: if more array elements to be transferred repeat step 1; otherwise return

The user's program and the Formatter behave as two parallel processes, with each executing as far as possible and then stopping to wait for the other.

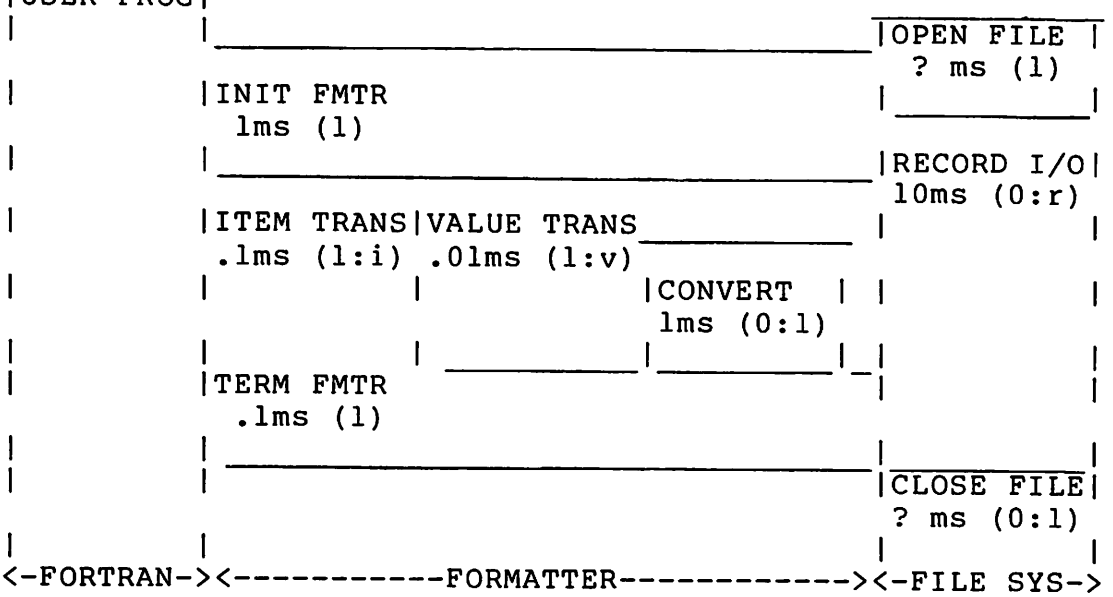
Data Conversion

The Formatter performs data conversions by translating format specifications into the parameters for either INEXT' or EXTIN'. These two intrinsics perform the actual conversions. INEXT' converts INternal bit patterns to EXTernal ASCII strings. EXTIN' converts EXTernal ASCII strings to INternal bit partterns.

Defining Processing Overhead

The definition of overhead used here is the difference in processor time for performing some action and not performing it. For example, the overhead of any statement is the amount of processor time saved if it is removed, assuming nothing else about the program's execution changes.

fig 12



The overhead of the Formatter is divided into seven (7) areas (fig 12): initialization (INIT FMTR), opening the file (OPEN FILE) , data item transfer (ITEM TRANS), data value transfer input/output (RECORD I/O), and termination (TERM FMTR). Closing a file is not included because the Formatter never closes any files. The order of magnitude overhead for each area (except opening the file) is shown in figure 12.

That figure should be interpreted this way:

1. The time shown in each box is the overhead incurred each time that box is entered. For example, each CONVERT has an overhead on the order of 1 millisecond.

2. The number(s) enclosed in parentheses indicate the number of times each box is entered for each time the box to the left is entered. For example, "(0:r)" means "entered zero to r (many) times".

The areas of overhead which must be considered for each of several types of Formatter input/output requests are discussed below.

Core-to-Core Transfers

Core-to-core transfers incur the same overhead as their corresponding formatted transfer requests, except no RECORD I/O is incurred.

Free-Field Transfers

Free-field transfers incur the same overhead as their corresponding formatted transfer requests.

Formatted Element Transfers

Formatted element transfers incur INIT FMTR, ITEM TRANS, VALUE TRANS (1 for each ITEM TRANS), CONVERT, RECORD I/O (depends on the format), and TERM FMTR overhead.

Formatted Array Transfers

Formatted array transfers incur INIT FMTR, ITEM TRANS, VALUE TRANS (for each element of each ITEM TRANS), CONVERT, RECORD I/O (depends on the format), and TERM FMTR overhead.

Unformatted Element Transfers

Unformatted element transfers incur INIT FMTR, ITEM TRANS, VALUE TRANS (1 for each ITEM TRANS), CONVERT, RECORD I/O (depends on the number and size of items), and TERM FMTR overhead.

Unformatted Array Transfers

Unformatted array transfers incur INIT FMTR, ITEM TRANS, VALUE TRANS (for each element of each ITEM TRANS), CONVERT, RECORD I/O (depends on the number, size and dimension of items), and TERM FMTR overhead.

What the Formatter Does Not Do

The Formatter does not close files because each Formatter input/output request corresponds to a single FREAD/FWRITE. The Formatter simply has no means of knowing that a file should be closed.

The Formatter has no provision to handle multiple record unbuffered input/output because it assumes the File System does all deblocking/blocking and presents/accepts only one record at a time. The best approach is to do multiple record unbuffered input/output with File System intrinsics and use Formatter intrinsics to do core-to-core transfers for each record in turn.

```
$CONTROL USLINIT,MAP,LOCATION,STAT
C
C  This is a sample FORTRAN program illustrating the use of the
C  Formatter.  Two real numbers are input (free-field) and the
C  two numbers and their sum are output (formatted)
C
      REAL a,b,c
10    CONTINUE
         READ (5,*,END=999) a,b
         c = a + b
         WRITE (6,600) a,b,c
600      FORMAT(1X,F10.3," + ",F10.3," = ",F10.3)
      GOTO 10
999   STOP
      END
```

```
$CONTROL USLINIT,MAP
BEGIN
   COMMENT
      This is a sample SPL program illustrating the use of the
      FORTRAN Formatter.  Two real numbers are input (free-field)
      and the two numbers and their sum are output (formatted).
   ;
   INTEGER ARRAY
      flut(0:2),      <<File Unit Table (used by Formatter)>>
      db'(*)=DB+0;    <<array to access DL-DB (for flut pointer)>>
   BYTE ARRAY
      format1(0:1),   <<dummy format for free-field input>>
      format2(0:35); <<format used for output>>
   REAL
      a,              <<input value>>
      b,              <<input value>>
      c;              <<sum of input values>>
   INTRINSIC
      FMTINIT',RIO',TFORM';

  <<Initialize flut>>
   flut(0) := [8/5,8/0]; <<formal designator FTN05>>
   flut(1) := [8/6,8/0]; <<formal designator FTN06>>
   flut(2) := -1;        <<flut terminator>>
   db'(-1) := @flut;     <<flut pointer>>
  <<Initialize formats>>
   MOVE format1 := " ";
   MOVE format2 := "(1X,F10.3,3H + ,F10.3,3H = ,F10.3)";
  <<Process input>>
   DO
      BEGIN <<process 2 more numbers>>
     <<read input values>>
      FMTINIT' (format1,5,0D,[8/0,8/%(2)10010001],@e'input);
      RIO' (a);
      RIO' (b);
      TFORM';
e'input:
      IF = THEN
         BEGIN <<output sum of numbers>>
         c := a + b;
         FMTINIT' (format2,6,0D,[8/0,8/%(2)00000000],@e'output);
         RIO' (a);
         RIO' (b);
         RIO' (c);
         TFORM';
e'output:
         END; <<output sum of numbers>>
      END <<process 2 more numbers>>
   UNTIL <>;
END.
```