To: X3J3/WG5
From: Rich Bleikamp (/io)
Subject: Functional Specification for Derived Type I/O
Date: Jan. 17, 1997

Revised version of X3J3/96−177. Look for "|"s in the left margin for significant changes.

One possible activity for WG5/X3J3 at the Feb. ’97 meeting is to start on the edits to F95 for this proposal. Any volunteers?

Changes since 97−177:

- added UNFORMATTED I/O support. Useful to avoid the restrictions such as "no pointers", ... which limits F90 functionality for unformatted derived type I/O.

  Since the so−called "user defined formatting routines" now support unformatted I/O, I have renamed them "user defined I/O routines".

- Allow internal I/O in one of these user−defined I/O routines. Or in any routine called therefrom.

- Added a Rationale.

- Added a Conceptual Model (how will this be implemented).

- Added a small example routine.

Unresolved Issues

- Should the "err", "eor", and "eof" dummy arguments be a derived type? Or is logical type sufficient?

- How can the writer of a user defined I/O routine debug anything, without the ability to WRITE stuff out?

- Should the I/O statements executed within a user defined I/O routine be IMPLICITLY non−advancing? This would only work for sequential formatted I/O.

  How do we describe UNFORMATTED and direct access I/O? Non−advancing is only allowed for formatted sequential I/O, so we need some other term to describe how we insert/extract characters from the middle of a record in the user defined I/O routine.

  Perhaps we need to describe these nested/recursive I/O calls in other terms, to avoid confusion with normal Fortran I/O.

- Should we allow a "text error string" to be returned when an ERR is to be reported, in case the original I/O statement did not have an ERR= or IOSTAT= ?

- Should we add an IOSTAT variable, so specific values can be passed back to the user?

- Having to handle internal and external units separately is inconvenient. Should we only pass in a unit ?

This document is the (proposed) functional specification for enhanced derived type I/O. The goals are to provide a powerful and portable way to encapsulate I/O support in a MODULE which defines a derived type. This I/O support is provided by simple, easy to use extensions to the traditional Fortran READ and WRITE statements.

This specification, in a earlier form, was approved as the generally correct approach for supporting derived type I/O at X3J3 meeting 139.

Management Synopsis:

- The provider of a derived type may also provide two I/O routines, called "user defined I/O routines", which are called by the Fortran I/O library when certain conditions are met. These user supplied routines handle input and output of a list item of derived type. In essense, the effect is as if the user defined I/O routines were substituting list items into the original I/O list (where the derived type item was), and adding edit descriptors into the middle of the original format specification, under control of the provided routines.

- The F90 way of doing formatted and unformatted I/O on derived types still works the same as before. Only the presence of an interface for the appropriate user defined I/O routine triggers this new functionality.

- FORMATs have a new edit descriptor, "DT". When the I/O library encounters this, it must match up with a derived type list item. The I/O library will call a user supplied I/O routine, which will actually do the I/O. Typically, the provider of a derived type would provide these user defined I/O routines.

- The user supplied procedures (one for READs, one for WRITEs), will be called with a unit number, the derived type variable/value, and other misc. information. The procedure will use normal I/O statements (READ/WRITE) on the supplied unit to read/write the derived type item.
  This use of "recursive" I/O will be restricted to this particular feature of the language. Internal I/O will be permitted in the user defined I/O routines.

- Full support for complicated data structures is provided. These user defined I/O routines can invoke themselves indirectly thru formatted I/O (to traverse a linked list for example), and can invoke the user defined I/O routines for another derived type (indirectly, thru formatted I/O) to handle nested derived types.
| Internal I/O may be used to easily construct character
| string values.

- The user supplied procedures will be able to inquire about, and in the most general (robust) case, have to worry about:
  | - Formatted and unformatted I/O
  - list directed and namelist I/O
  - sequential and direct access I/O
  - non-advancing and advancing I/O
  - the DELIM= and PAD= values for this file (accessible via INQUIRE)

- List directed and NAMELIST I/O will also call these same user supplied routines under certain, F90
| compatible circumstances. (when an appropriate interface
| is visible)

Detailed Specification:

User defined I/O routines shall have the following interface:

```
  INTERFACE FORMAT ( READ )
    RECURSIVE SUBROUTINE my_read_routine  (unit,
                                           ifu,
                                           dtv,
                                           iotype, w, d, m,
                                           rec, eof, err, eor)
      INTEGER, OPTIONAL :: unit
      CHARACTER (LEN=*), OPTIONAL  :: ifu
      TYPE (whateveritis) dtv   ! the derived type value/variable
      CHARACTER (*) iotype      ! the edit descriptor string
      INTEGER, OPTIONAL :: w,d,m
      INTEGER, OPTIONAL :: rec
      LOGICAL :: eof, err, eor
    END
  END INTERFACE
  INTERFACE FORMAT ( WRITE )
    RECURSIVE  SUBROUTINE my_write_routine (unit,
                                            ifu,
                                            dtv,
                                            iotype, w, d, m,
                                            rec, err)
      INTEGER, OPTIONAL :: unit
      CHARACTER (LEN=*), OPTIONAL  :: ifu
      TYPE (whateveritis) dtv   ! the derived type value/variable
      CHARACTER (*) iotype      ! the edit descriptor string
      INTEGER, OPTIONAL :: w,d,m
      INTEGER, OPTIONAL :: rec
      LOGICAL :: err
    END
  END INTERFACE
```

where the actual specific routine names (my_xxx_routine above) and the dummy argument names may be chosen by the user. These routines shall not be invoked directly by the users program.

The user defined I/O routines are called when:

| - for unformatted i/o, list directed, and namelist i/o,
| an appropriate interface for the derived type of a
| particular list item is visible

| - for I/O statements with a <format-specification>,
| there must be an appropriate interface AND the
| list item must match up with a "DT" edit descriptor.

What the user defined I/O routines are passed:

If the original I/O statement specified list directed I/O, the "iotype" argument will have the value "LISTDIRECTED". If the original I/O statement specified NAMELIST I/O, the "iotype"
| argument will have the value "NAMELIST". When the original
| I/O statement specified UNFORMATTED I/O, the "iotype" argument
| will have the value "UNFORMATTED".

When the original I/O statement included a format-specification, then the user defined I/O routines are accessible via the new "DT" edit descriptor.

A new edit descriptor, "DT", with the usual (optional) "[w[.d[.m]]]" widths is provided for use with format specifications. It must match up with a variable/value of a derived type.

The DT characters may be followed by an arbitrary (up to 253?) number of alphabetic characters (interspersed blanks allowed) (ex. "DTLNKLST"). The entire string of alphabetic characters, including the initial "DT", will be passed into the user defined I/O routine (the "iotype" argument).

This argument will be converted to UPPERCASE and have all blanks removed. The user can support different types of formatting for one derived type via this extended edit descriptor.

For example, the consecutive characters after the "DT" could be used to request different formatting rules for consecutive components in the derived type, or different formatting rules for nested derived types, etc.

If a derived type variable/value is specified in an I/O list, and that variable/value will match up with a "DT" edit descriptor, the user must have also provided the matching read/write procedure for that derived type , with a visible interface thats matches the definition in this paper. These procedures are called the "user defined I/O routines". If such an interface is visible, the derived type item may either match a "DT" edit descriptor or use original F90 conventions.
When the user defined I/O routines are called, either "unit" or "ifu" will be present, but not both.

If "unit" is present, the original I/O statement specified an external unit (possibly *), and all I/O statements for
| external units in the user defined I/O routine
| (including INQUIRE) shall specify this dummy argument for the
| UNIT= specifier.   (we used to only require the same value)

The "unit" dummy argument, if present, contains a processor dependent value, that may, or may not, be the same unit number specified by the user in the original I/O statement.

Note that an INQUIRE statement cannot be executed when "unit" is absent.

| If "ifu" is present, the original I/O statement specified an
| internal unit, and all I/O statements in the user defined
| I/O routine shall specify an internal unit specifier.
| If the dummy argument "ifu" is used as the unit specifier,
| the I/O statement processes the record(s) from the original
| I/O statement which triggered this user defined I/O routine.
| Other internal unit variables will behave as if no other I/O
| were active.

| Note that "ifu" may not have any obvious relationship with
| the internal unit specified in the original I/O statement
| (i.e. "ifu" may not point to the original internal unit
| in any discernable manner).

If the original I/O statement is a READ statement, the "dtv" dummy arg should be assigned a value by the user defined I/O "read" routine.

If the original I/O statement is a WRITE or PRINT, the "dtv" dummy arg contains the value of the list item from the original I/O statement, to be output by the user defined I/O routine.

The "w", "d", and "m" arguments contain the user specified values from the FORMAT (i.e. FORMAT(DT12.5.2)). If the user did not specify "w", "d", and/or "m", those dummy arguments will not be present. They will not be present if the original I/O statement specified unformatted, list directed, or namelist i/o.

The "rec" dummy arg will be present if the original I/O statement contained a REC= sepcifier, and will not be present otherwise. Note that the READ or WRITE statements for "unit" or "ifu" contained in the user defined I/O routine shall contain a REC=rec specifier if dummy arg "rec" is present, and shall not contain a REC= specifier otherwise.

The user defined I/O routines for reads shall assign a value of .FALSE. or .TRUE. to the "end", "err", "eof", and "eor" dummy args. The value assigned to these dummy arguments shall determine whether or not the corresponding condition will be triggered in the I/O library when the user defined I/O routine returns.

In the absence of an appropriate visible interface in the
| scope of the I/O statement, unformatted, list-directed, and namelist I/O will behave as it did in Fortran 90.

When an appropriate interface is visible for a particular derived type, and either:

| 1. The original I/O statement specified unformatted, list directed, or namelist I/O, OR

2. the original I/O statement specified a FORMAT and the list item of derived type matches up with a "DT" edit descriptor, THEN

the restrictions on derived type I/O, such as no private components, all components must be defined, no ultimate components with the pointer attribute, etc. do not apply to the list item of derived type, but
the normal rules in F95 still apply, about not referencing undefined entities, not referencing/defining POINTERS which are not associated, etc.

If NO appropriate interface is visible for a particular derived type, the processor will perform "F90" style I/O, and a "DT" edit descriptor which matches that derived type list item will cause an error (at runtime possibly).

When F90 style I/O is selected, all the old F90 restrictions on derived type list items still apply.

The users routine may chose to interpret the "w" argument as a field width, but this is NOT required. If it does so, it would be appropriate, but not required, to fill an output field with "***"s if the value does not fit.

When the original I/O statement was a READ, the user defined I/O routine may only do READs. Similarly for WRITE.

The user defined I/O routines ARE permitted to use a FORMAT with a DT edit descriptor, for handling components of the derived type which are themselves a derived type. List directed and NAMELIST I/O are also permitted for the "recursive" I/O statement.

WRITE statements contained in the user defined I/O routine will insert the characters "written" into the record started by the original WRITE statement, starting at the position in the record where the last edit descriptor left off. Record boundaries may be created by WRITE statements in the user defined I/O routine. Non-advancing I/O may be used to avoid creating record boundaries.

READ statements contained in the user defined I/O routine for read will "pick up" in the current record, where the last edit descriptor from the original I/O statement left off. Multiple records can be read, and the current position can be left within a record by the READ statement in the user defined I/O routine, thru the use of non-advancing i/o.

A very robust user defined I/O routine may need to use INQUIRE to determine what BLANK=, PAD= and DELIM= are for the specified unit.

Edit descriptors such as BN, BZ, P, etc., are permitted in FORMATs in user defined I/O routines, and have the same effect as if they had been present in the original FORMAT.

| READ and WRITE statements executed in a user defined I/O
| routine, or executed in a routine called (directly or
| indirectly) from a user defined I/O routine shall not
| have an ASYNCHRONOUS specifier.

Rationale

The desire to allow users to implement new data types in a MODULE requires additional language features, including I/O support. The provider of a module which implements a new datatype needs to be able to also provide I/O support.
The approach chosen extends existing Fortran features to support derived types, is fairly easy to use, bypasses the restrictions on derived type I/O present in Fortran 90, and allows the I/O support to be bundled with the MODULE which supplies the derived type definition and implements the operations thereon. This also provides the ability to protect these I/O operations.

The use of visible interfaces to trigger this functionality helps preserve Fortran 90 compatability, since no Fortran program can specify such an interface.

Conceptual Model

The key concept is that the user defined I/O routines can, more or less, be viewed as adding individual components into the middle of the original item list, and edit desciptors into the middle of the original format-specification (if any). They also have full control over how input values are processed, and how values are represented on output.
They can do so in an intelligent, dynamic, and arbitrarily complex manner. They can also avoid the restrictions on F90 derived type I/O (pointers, etc.), handle nested derived types, and support complex data structures (such as linked lists).

The user defined I/O routines provide a familar mechanism, Fortran I/O statements, to insert data into an output record, and to retrieve values from an input record.

The user of a derived type uses familiar Fortran syntax to activate this capability. Usually, the user only needs to "USE" the appropriate module, and possibly insert some "DT" edit descriptors into their format-specifications.

All of the hard work is done by the provider/writer of the derived type. Once that hard work is done, many users can easily adapt their programs to use it.

The interface provides all the information necessary to accomadate all types of Fortran I/O. A robust user defined I/O routine will be quite large, but not necessarily very complicated. A simple user-defined I/O routine can be written quickly, and extended later to handle all the possible forms of Fortran I/O.

```
|   -------------------------------------------------------------
|   Example:  ( this has not been syntax checked yet )

|   TYPE linkedList
|     TYPE (linkedList), POINTER :: next
|     INTEGER :: value
|   END TYPE linkedList

|   RECURSIVE  SUBROUTINE my_write_routine (unit,
|                                          ifu,
|                                          dtv,
|                                          iotype, w, d, m,
|                                          rec, err)
|     INTEGER, OPTIONAL :: unit
|     CHARACTER (LEN=*), OPTIONAL  :: ifu
|     TYPE (linkedList), TARGET:: dtv  ! the derived type value
|     CHARACTER (*) iotype             ! the edit descriptor string
|     INTEGER, OPTIONAL :: w,d,m
|     INTEGER, OPTIONAL :: rec

|     TYPE (linkedList), POINTER :: ptr
|     INTEGER :: ww, dd             ! local copies of w,d
|     CHARACTER (LEN=20) :: fmt     ! format specification
|
|     err = .FALSE.

|     IF ( iotype == "NAMELIST" ) THEN
|       ! namelist I/O not supported yet
|       err = .TRUE.
|       RETURN
|     END IF

|     ! handle the optional "w" and "d" arguments
|     IF ( present ( w ) ) THEN
|       ww = w
|     ELSE
|       ww = 10
|     END IF

|     IF ( present ( d ) ) THEN
|       dd = d
|     ELSE
|       dd = 1
|     END IF

|     ! if we will need a format-spec, build it now
|     IF ( iotype(1:2) == "DT" ) THEN
|       ! build a Format string for use later
|       write(fmt, "'(I',I4,1x,I4,')'" ) ww, dd  ! (Iw.d)
|     END IF

|     ptr => dtv
|     DO            ! main loop down the linked list

|       IF ( PRESENT ( unit ) ) THEN
|         ! external I/O
|         IF ( iotype == "UNFORMATTED") THEN
|           IF ( PRESENT ( rec ) ) THEN
|             WRITE (unit, REC=rec, ERR=99) ptr%value
|           ELSE
|             WRITE (unit, ERR=99) ptr%value
|           END IF
|         ELSE IF ( iotype == "LISTDIRECTED" ) THEN
|           WRITE (unit, *, ADVANCE="NO", ERR=99) ptr%value
|         ELSE IF ( iotype(1:2) == "DT" ) THEN
|           IF ( PRESENT ( rec ) ) THEN
|             write(unit, fmt, REC=rec, ERR=99) ptr%value
|           ELSE
|             write(unit, fmt, ADVANCE="NO", ERR=99) ptr%value
|           END IF
|         ELSE
|           ! unrecognized i/o type
|           GO TO 99
|         END IF
|       ELSE
|         ! assume internal I/O
|         !   remember, direct access (rec=) is prohibited on
|         !   internal files, simplifies the stuff below
|         IF ( iotype == "UNFORMATTED") THEN
|           WRITE (ifu, ERR=99) ptr%value
|         ELSE IF ( iotype == "LISTDIRECTED" ) THEN
|           WRITE (ifu, *, ADVANCE="NO", ERR=99) ptr%value
|         ELSE IF ( iotype(1:2) == "DT" ) THEN
|           write(ifu, fmt, ADVANCE="NO", ERR=99) ptr%value
|         ELSE
|           ! unrecognized i/o type
|           GO TO 99
|         END IF
|       END IF
|       IF ( ASSOCIATED (ptr%next) ) EXIT
|     END DO
|   RETURN          ! normal exit

|99 err = .TRUE.
|   RETURN          ! error  exit
    END
```