Subject: Revised spec. for Derived Type I/O
Date: May 14, 1997

Major Changes since 97-101.

- Added separate user defined routines for Formatted vs. Unformatted I/O. Subgroup decided this would simplify writing these routines, and the improved clarity of the resulting routines is useful enough to outweigh the downside, duplicating the traversal of the derived type in twice as many routines.

- Decided to pass in ONE unit #, to represent both internal and external units. This eliminates the need to always test for which dummy argument is present (the UNIT or IFU dummy arg), simplifying these procedures considerably.

- Deleted the REC dummy argument (direct access record number). The recursive READ/WRITE calls may not specify a REC= specifier. However, specifying NON-ADVANCING for formatted I/O is required when the I/O operation is direct access (INQUIRE may be used to find out if this is a direct access file). Fortunately, most users will always want to specify non-advancing I/O for typical 2-3 component derived types.

  Note: The derived type I/O routine cannot call INQUIRE with a negative unit #. This implies the "*" external unit cannot be inquired (just like in F90).

- Unformatted I/O statements executed in the derived type I/O routine which specify the passed in unit # will act sort of like non-advancing I/O, by concatenating the "buffer" produced with the buffer started by the original I/O statement (these recursive I/O statements will not start or end a record). I need to find a better word than non-advancing to describe this behavior in the standard.

- INQUIRE by IOLENGTH will not interact with user defined I/O routines. An INQUIRE by I/O length will assume F90 semantics for derived type I/O. Unlike normal I/O, the actual data type does not determine the length of an I/O buffer needed for unformatted I/O.

- Added an intent out dummy arg, ERRMSG, which is used to return an error message to the I/O library.

- Certain dynamic formatting state information (BN, BZ, S, SP, SS, P current setting) will be saved by the Fortran I/O library BEFORE calling the user defined I/O routine, and restored (from the saved setting) after the user defined I/O routine returns.

Unresolved Issues

- Unformatted I/O is now implicitly non-advancing, in some sense. Formatted I/O isn't. An alternative is to allow non-advancing for unformatted I/O.

- At least one person has suggested allowing these routines to be called directly from a user program. I think this is confusing, since there is no initiating I/O operation, and the standardese will be awkward. Also, typically the user will use non-advancing I/O for these routines, and when called directly from a user routine, this seems less desirable.

Management Synopsis (also see the Rationale and Conceptual Model at the end of this paper):

- The provider of a derived type may also provide I/O routines for that type, called "user defined derived type I/O routines" (hereafter refered to as UDDTIO routines), which are called by the Fortran I/O library when certain conditions are met. These UDDTIO routines perform input and output of list items of a particular derived type. In essense, the effect is as if the UDDTIO 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 UDDTIO 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 the appropriate UDDTIO routine, which will actually do the I/O. Typically, the provider of a derived type (and corresponding module) would provide these UDDTIO routines as part of the module.

  NOTE: we have chosen not to implicitly overload the existing data transfer edit descriptors (I,D,E,F,G,L,...) when such an interface is visible, and call the UDDTIO routine for those edit descriptors (in addition to DTxxx edit descriptors). This capability is easy to add should we wish too, buts makes it more difficult for the user to get to the F90 functionality. Interval 2 may propose some additional syntax to address this issue.

- The UDDTIO routines will be called with a unit number, the derived type variable/value, and other misc. information. The UDDTIO routine will use normal I/O statements (READ/WRITE) on the supplied unit to read/write the derived type item's components.

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

- The UDDTIO routines will be able to inquire about, and in the most general (robust) case, might want to worry about
  - list directed vs. namelist I/O vs. a format spec.
  - the DELIM= and PAD= values for this file (accessible via INQUIRE)

  on external (positive) unit numbers.

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

Detailed Specification:

UDDTIO routines shall have the following interface (all 4 routines for a particular derived type are not required, any subset can be provided):

```
  INTERFACE FORMATTED ( READ )
    SUBROUTINE my_read_routine  (unit,                  &
                                 dtv,                   &
                                 iotype, w, d, m,       &
                                 eof, err, eor, errmsg)
      INTEGER, INTENT(IN) :: unit ! unit number
      ! the derived type value/variable
      TYPE (whateveritis), INTENT(OUT) :: dtv
      ! the edit descriptor string
      CHARACTER, (LEN=*), INTENT(IN) :: iotype
      INTEGER, OPTIONAL, INTENT(IN) :: w,d,m
      LOGICAL, INTENT(OUT) :: eof, err, eor
      CHARACTER, (LEN=*), INTENT(OUT) :: errmsg
    END
  END INTERFACE
```

```
  INTERFACE UNFORMATTED ( READ )
    SUBROUTINE my_read_routine  (unit,                  &
                                 dtv,                   &
                                 eof, err, eor, errmsg)
      INTEGER, INTENT(IN) :: unit
      ! the derived type value/variable
      TYPE (whateveritis) INTENT(OUT) :: dtv
      LOGICAL, INTENT(OUT) :: eof, err, eor
      CHARACTER, (LEN=*), INTENT(OUT) :: errmsg
    END
  END INTERFACE

  INTERFACE FORMATED ( WRITE )
    SUBROUTINE my_write_routine (unit,                  &
                                 dtv,                   &
                                 iotype, w, d, m,       &
                                 err, errmsg)
      INTEGER, INTENT(IN) :: unit
      ! the derived type value/variable
      TYPE (whateveritis), INTENT(IN) :: dtv
      ! the edit descriptor string
      CHARACTER, (LEN=*), INTENT(IN) :: iotype
      INTEGER, OPTIONAL, INTENT(IN) :: w,d,m
      LOGICAL, INTENT(OUT) :: err
      CHARACTER, (LEN=*), INTENT(OUT) :: errmsg
    END
  END INTERFACE
  INTERFACE UNFORMATED ( WRITE )
    SUBROUTINE my_write_routine (unit,                  &
                                 dtv,                   &
                                 err, errmsg)
      INTEGER, INTENT(IN) :: unit
      ! the derived type value/variable
      TYPE (whateveritis), INTENT(IN) :: dtv
      LOGICAL, INTENT(OUT) :: err
      CHARACTER, (LEN=*), INTENT(OUT) :: errmsg
    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 "dtv" dummy argument may also be given the TARGET attribute. It may not be given any other attributes.

The UDDTIO routines are called when:

- for unformatted, 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 is be an appropriate interface visible AND the list item matchs up with a "DTxxx" 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 up to 253 alphabetic characters (interspersed blanks allowed) (ex. "DTLNKLST"). The entire string of alphabetic characters, including the initial "DT", will be passed into the UDDTIO 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.

When a derived type variable/value matchs 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 that matches the definition in this paper.

The "unit" dummy argument will have the same unit value as specified by the user in the originating I/O statement for all external units except "*". When an internal unit or the "*" external unit was specified in the originating I/O statement, the "unit" dummy argument will have a processor dependent negative value.

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

The "iotype" argument (FORMATTED I/O routines only) will have the value:

- "LISTDIRECTED" if the originating I/O statement specified list directed I/O,

- "NAMELIST" if the original I/O statement contained an NML= specifier, or

- "DTxxx" if the originating I/O statement contained a format specification and the list item matched up with a DT edit descriptor, where the "xxx" is the string of alphabetic characters (if any) that actually followed "DT" in the edit descriptor.

If the original I/O statement is a READ statement, the "dtv" dummy arg should be assigned a value by the UDDTIO 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 UDDTIO 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 was a list directed, or namelist I/O statement.

The UDDTIO routines for reads shall assign a value of .FALSE. or .TRUE. to the "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 UDDTIO routine returns.

If the value .TRUE. is assigned to the "err" dummy argument, the "errmsg" dummy argument shall be defined also, before the UDDTIO routine returns.

When "err" is set to true, and the originating I/O statement did not contain an ERR= nor an IOSTAT= specifier, the processor shall attempt to output the "errmsg" value (to something) and stop execution of the program. If we add an ERRMSG= specifier to all read/write statements, this value would be returned thereto.

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, it would be appropriate to fill an output field with "*"s if "w" is too small.

When the original I/O statement was a READ, the UDDTIO routine may not READ from any other external unit other than the one passed in via the dummy arg "unit, nor WRITE to any external unit.

When the original I/O statement was a WRITE, the UDDTIO routine may not WRITE to any other external unit other than the one passed in via the dummy arg "unit, nor READ from any external unit.

Thou shalt not call BACKSPACE, ENDFILE, or REWIND while a UDDTIO routine is active.

The UDDTIO 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 UDDTIO routine which specify the same value as passed in via the "unit" dummy arg 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 UDDTIO routines. Non-advancing I/O may be used to avoid creating record boundaries.

READ statements contained in the UDDTIO routine which specify the same value as passed in via the "unit" dummy arg 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 UDDTIO routine, thru the use of non-advancing i/o.

Record positioning edit descriptors, such as TL and TR, used on "unit" while a UDDTIO routine is active, shall not cause the record position to be positioned before the record position at the time the UDDTIO routine was invoked.

A very robust UDDTIO routine may wish to use INQUIRE to determine what BLANK=, PAD= and DELIM= are for the unit.

Edit descriptors which affect subsequent edit descriptors behavior, such as BN, BZ, P, etc., are permitted in FORMATs in UDDTIO routines. The Fortran I/O library will save the state of BN, BZ, S, SP, SS, and P before calling a UDDTIO routine, and reset the library's state to those saved values when the UDDTIO routine returns. The UDDTIO routine is free to use these state changing edit descriptors without having any effect on the formatting of list items in the originating I/O list. If directed rounding mode edit descriptors are added, these will be added to the list of "saved" states.

READ and WRITE statements executed in a UDDTIO routine, or executed in a routine called (directly or indirectly) from a UDDTIO 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 from the user.

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

Conceptual Model

The key concept is that the UDDTIO 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 UDDTIO 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 accomodate all types of Fortran I/O. A robust UDDTIO routine may be quite large, but not necessarily very complicated. A simple UDDTIO routine can be written quickly for one or two forms of I/O, and extended later to handle all the possible forms of Fortran I/O.

Example for a FORMATTED(WRITE) routine:

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

RECURSIVE  SUBROUTINE my_write_routine (unit, dtv,
                                        iotype, w, d, m,
                                        err, errmsg)
  INTEGER, INTENT(IN) :: unit
  ! the derived type value
  TYPE (linkedList), TARGET, INTENT(IN):: dtv
  CHARACTER (LEN=*), INTENT(IN) :: iotype  ! the edit descriptor
  INTEGER, OPTIONAL, INTENT(IN) :: w,d,m
  LOGICAL, INTENT(OUT) :: err
  CHARACTER, (LEN=*), INTENT(OUT) :: errmsg

  TYPE (linkedList), POINTER :: ptr
  INTEGER :: ww, dd             ! local copies of w,d
  INTEGER :: en                 ! iostat= error value
  CHARACTER, (LEN=20) :: fmt    ! format specification

  err = .FALSE.

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

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

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

  ptr => dtv

  DO            ! main loop thru the linked list
    IF ( iotype == "LISTDIRECTED" ) THEN
      WRITE (unit, *,  ADVANCE="NO", ERR=99, IOSTAT=en) ptr%value
    ELSE IF ( iotype(1:2) == "DT" ) THEN
      write(unit, fmt, ADVANCE="NO", ERR=99, IOSTAT=en) ptr%value
    ELSE
      ! unrecognized i/o type
      errmsg="Unsupported I/O request:type(linkedList):"//iotype
      err = .TRUE.
      RETURN
    END IF
    IF ( ASSOCIATED (ptr%next) ) EXIT
  END DO
RETURN          ! normal exit

99 write(errmsg, "('Error writing linkedList%value, IOSTAT=',I9)") en
   err = .TRUE.
   RETURN          ! error exit
   END
```