Date: 5 May 1997
To: J3
From: Van Snyder
Subject: Comments about J3/97-101, "Derived Type I/O"

First, I must apologize for putting this on the table at such a late date.

On page 1, paragraph -3, 97-101 asks:

> How do we describe UNFORMATTED and direct access I/O? Non-advancing is only allowed for formatted sequential I/O....

I prefer relaxing the restrictions on non-advancing I/O, if possible, as opposed to inventing a new special mechanism to substitute for it in this special case.

On page 2, in the first paragraph, 97-101 asks:

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

My tentative answer is "yes." But this may cause difficulty in producing status error messages. What if the user defined derived type I/O routine passes back an IOSTAT value "invented out of thin air" instead of one produced by an I/O statement in the routine? It will always be impossible to deprive users of sharp knives with which they might cut their own throats. I'd be happy with a warning that IOSTAT values not gotten from I/O statements may elicit bogus messages from the routines that fetch/display status error messages.

On page 3, 97-101 proposes INTERFACE FORMAT (READ).

Since the intended use of this interface apparently has been extended to unformatted I/O, I would prefer INTERFACE INPUT or INTERFACE READ, and, in parallel, INTERFACE OUTPUT or INTERFACE WRITE instead of INTERFACE FORMAT (WRITE).

I would prefer to have two routines, one for I/O on external files, and one for I/O on internal files (character arrays). Since the "system" I/O library has already determined whether I/O is directed to an internal or external file, it would make more sense to call different routines rather than to call the same routine (effectively under selection of an IF block, with exactly

one of "unit" or "ifu" present in the call in each branch), and then have the user's routine test *again* whether I/O is directed to an internal or external file.

On page 5, in paragraph 4, 97-101 contains:

> If "unit" is present, the original I/O statement specified an external unit (possibly *)...

This implies that the "system" I/O library *must* be able to produce a unit number corresponding to *, and insure that output directed to (input received from) that unit have the correct time sequence of data. It's not a great stretch for implementors to provide an intrinsic function that returns the unit number(s) for writing (and reading) unit *. It's presumably also required for "PRINT" and "READ" (with no unit, not even *). A proposal to provide this capability was approved by /misc but judged by the full committee at WG5/J3 meeting 140 to be "a good idea, but there's not enough time." Should it be upgraded to "approved MTE?"

An alternative is to specify unit numbers for those purposes. Since negative unit numbers are presently prohibited, the standard could specify a few negative unit numbers for this purpose without invalidating any presently standard-conforming programs – e.g. -1 is the unit for READ or READ (*...), -2 is the unit for PRINT and WRITE (*...), and -3 is the unit for output to "stderr" if such a concept exists on the system, else it's the same as using unit -2. This would take less editing than introducing several new intrinsic functions, and wouldn't "pollute the name space."

On page 6, in paragraph 2, 97-101 specifies:

> The "w", "d" and "m" arguments contain the user specified values from the format (i.e. (sic) FORMAT(DT12.5.2)). If the user did not specify "w", "d" or "m" those dummy arguments will not be present....

Optional arguments are very convenient for interface to reusable software in which the user knows at the time the calling program is composed exactly which subset of arguments are desired. In the case of a "system" I/O routine (the one that calls the user's derived type I/O routine), they cause a combinatorial-explosion-induced nightmare for implementors, because they *must* cater to all possible combinations of present and absent arguments – with n optional arguments, the calling routine needs to have

IF/CASE blocks that lead to one of 2^n different calls (or be able to construct and then execute calling sequences that describe present/absent optional arguments "on the fly" – but that's the subject of another of my proposals, which didn't even get out of subgroup).

I think it would ease implementors jobs if the interface were defined with fewer optional arguments, and more values that say "this value wasn't specified by the user" – e.g. negative values for "w," "d" or "m" can't be specified in a FORMAT statement, and so could mean "absent." This may entail *extra* arguments in some cases (e.g. a LOGICAL argument LA that says whether argument A was specified), but I think it's worth it.

Another alternative is a simulation of optional arguments I've used since about 1974, called an "option vector." These are easy to set up, even "on the fly," are easy and efficient to process, and they're portable. I can give examples if anybody is interested but isn't familiar with the idea yet.

If we *insist* on the OPTIONAL argument approach, could "w," "d" and "m" be elements of an array with 0, 1, 2 or 3 elements. This change alone would decrease the number of IF/CASE blocks in the calling routine by a factor of 8.

On page 7, in paragraph 2, 97-101 specifies:

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

Would it be a real headache for developers if we allowed WRITE's on different units (or different internal files) if the original I/O statement was a READ, and conversely for WRITE? The present proposal appears to allow READ's on different units during READ's, so is WRITE on a different unit much of an extension? BTW, this answers the question raised on page 1, in paragraph 2 of *Unresolved issues*: "How can the writer of a user defined I/O routine debug anything, without the ability to WRITE stuff out?"

On page 7, in paragraph -3, 97-101 remarks:

> 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.

Could INQUIRE be extended to allow inquiring after many of the things that are presently proposed to be delivered by way of optional arguments?

This is another strategy to reduce the implementors' headaches due to "combinatorial explosion."

On page 7, in paragraph -1, 97-101 specifies:

> 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 the ASYNCHRONOUS specifier.

This is presumably because the I/O routine can't be *sure* that the OPEN for that unit had the ASYNCHRONOUS specifier. If the presence or absence of the ASYNCHRONOUS specifier could be determined, most attractively by way of INQUIRE, would asynchronous I/O in user defined I/O routines cause a problem?

Nit Picking

For stylistic consistency, I'd prefer "CHARACTER (LEN=*) iotype" instead of "CHARACTER (*) iotype".

Shouldn't "... I/O statements in the user ..." in paragraph -3 on page 5 be "... I/O statements *executed* in the user ..."?

The "i.e." in line 2 of paragraph 2 on page 6 should be "e.g." "i.e." means *id est*, Latin for "that is," while "e.g." means *example grati*, Latin for "for example."

The "end" in line 2 of paragraph 4 on page 6 should be removed – it's superfluous and redundant.

In line 4 of the first paragraph of *Rationale* on page 8, "to also" should be "also to." (I also must guard against a tendency to carelessly split infinitives.)

In line 2 of paragraph -1 on page 8, "accomadate" should be "accomodate".