To: X3J3
From: Rich Bleikamp
Subject: Revised syntax and partial edits for Async I/O
Date: May 14, 1997

(a revision of J3/97-102)

Issues resolved in this revision:

- ID= variables will be default integer type

- The list items in asynchronous I/O lists are now required to be evaluated (i.e. an address or descriptor computed) when the async I/O statement is executed. This requirement is implicit in the lack of rules about when implied DO variables can be referenced/modified, and the value of such variables after an I/O statement is executed. Some explicit statement of this intent is probably needed to help implementors.

  This means a statement such as READ(...)n,(a(i), j=1,n) will probably never be performed asynchronously. This change solves a class of problems, such as using function calls in subscript calculations in the I/O list, where such a function (even if PURE) can read a global variable. Such behavior is extremely difficult to prohibit in standardese, and not useful (in subgroups opinion).

  This also allows simplification of the standardese having to do with implied DO variables, and when they can be redefined or referenced. I/O statements with implied DO loops can be still be performed in an asynchronous manner, but the processors I/O library cannot read/write from/into the implied DO variables after the async I/O statement is executed.

- A dummy argument with the ASYNC attribute now forces the containing procedure to have a visible explicit interface. This allows the processor to know that it must use call by address/descriptor. A dummy argument is required to have the ASYNC attribute if the corresponding actual argument has the ASYNC attribute.

Remaining issues:

- The presence of an ASYNC attribute for a dummy arg in a visible interface for a called routine requires the processor to pass by address or pass by descriptor. We need to prohibit those cases where such parameter passing mechanisms violate other parts of the standard, or cannot be implemented reasonably. These include vector valued subscripts actual arguments when the indexed array was in the I/O list of a pending async I/O operation, and those cases where interp 125 requires copyin/copyout (if any).

"Notes to the reader" are not notes to be included in the standard. Text to be included in the standard is either "quoted" or indented.

Edits to 96-007R1:

In rule 214 (specification-stmt), add:
or asynchronous-stmt

In rule R426 (component-attr-spec), add:
or ASYNCHRONOUS

In rule R503 (attr-spec), add:
or ASYNCHRONOUS

and add a new section (page 57):

5.1.2.12 ASYNCHRONOUS attribute

The ASYNCHRONOUS attribute may be specified for any variable, in any scoping unit.

A variable that :

1) is used in an asynchronous data transfer statement input/output list, or

2) is in a namelist group that is used in an asynchronous data transfer statement, and is actually read or written by that data transfer statement, or

3) is specified in a SIZE= specifier in an asynchronous data transfer statement

or is associated with such a variable shall have the ASYNCHRONOUS attribute, or be a subobject of an object with the ASYNCHRONOUS attribute, in a given scoping unit, if :

1) that variable is referenced, defined, or used as an actual argument in a scoping unit other than the scoping unit containing the asynchronous data transfer statement, and

2) any executable statement in such a scoping unit might be executed while the asynchronous data transfer operation is pending.

A variable with the ASYNCHRONOUS attribute (implicitly or explicitly) shall not be passed as an actual argument unless the corresponding dummy argument has the ASYNCHRONOUS attribute.

Note: A pending data transfer operation exists when a READ or WRITE statement with the ASYNCHRONOUS specifier is executed, but the corresponding wait operation has not yet been executed.

Note to reader: we allow any variable to have the asynchronous attribute so users can remove ASYNCHRONOUS specifiers from data transfer statements without having to delete the ASYNCHRONOUS attribute.

Note: The ASYNCHRONOUS attribute is similar to the VOLATILE attribute provided by some processors, and is intended to facilitate traditional code motion optimizations in the presence of asynchronous input/output.

Variables in asynchronous input / output lists implicitly have the ASYNCHRONOUS attribute in the scoping unit of that asynchronous READ or WRITE statement, but shall have the ASYNCHRONOUS attribute in other scoping units when those variables are referenced, defined, or otherwise used in a scoping unit, and ANY executable statements in that scoping unit might be executed while the asynchrounous I/O is pending. Other variables associated (argument and storage association) with such variables must also have the ASYNCHRONOUS attribute under those same circumstances.
-- End Note

Add a new section, 9.2.10 (and renumber 9.2.10 and later sections):

9.2.10 ASYNCHRONOUS statement

```
R5xx  asynchronous-stmt is  ASYNCHRONOUS  [::]
                                          <object-name-list>
```

The ASYNCHRONOUS statement specifes the ASYNCHRONOUS attribute for a list of objects.

In rule R905 (OPEN statement connect-spec), add, after PAD= (on its own line)(pg. 140):

```
  or  ASYNCHRONOUS
```

Add section 9.3.4.11 (page 142/143):

9.3.4.11 ASYNCHRONOUS specifier in the OPEN statement

If the ASYNCHRONOUS specifier is specified for a unit in an OPEN statement, then READ and WRITE statements for that unit may include the ASYNCHRONOUS specifier in the control information list.

The presence of an ASYNCHRONOUS specifier in a READ or WRITE statement permits, but does not require, a processor to perform the data transfer asynchronously. The WAIT, CLOSE, and file positioning statements may be used to wait for asynchronous data transfer operations to complete, and the INQUIRE statement may be used to inquire whether or not asynchronous data transfer operations have completed.

Note to the reader: the above rules imply only external unit input / output (not including the "*" unit) may specify an ASYNCHRONOUS specifier for READs and WRITEs, since internal files and the "*" external unit are not OPENed.

In section 9.3.5 (CLOSE statement), page 143, add the following paragraph and notes after line 5:

Execution of a CLOSE statement causes the processor to wait for all pending data transfer operations for the specified unit to complete.

If a CLOSE statement is executed for a unit with pending data transfer operations, that CLOSE statement is considered to be the corresponding wait operation for the READ or WRITE statements that initiated those pending data transfer operations, and the CLOSE statement is considered to be a data transfer statement for purposes of end of file, end of record, and error processing.

Deleted a big paragraph that discussed when a variable needed the asynchronous attribute.

In rule 912 (io-control-spec) (page 144), add:

or ASYNCHRONOUS
or ID = <scalar-default-int-variable>

Add the following constraint after the constraint on line 19, page 145:

Constraint: An ASYNCHRONOUS specifier shall be present if an ID= specifier is present.

Constraint: An ASYNCHRONOUS specifier shall not be specified if the <io-unit> is an <internal-file-unit> or "*".

Note to the reader: the first constraint implies an ID= specifier, typically used in a corresponding WAIT statement, is NOT required in an asynchronous READ or WRITE statement. The user would have to CLOSE the unit (or execute another wait operation) before referencing any storage locations in an input list or namelist, and to NOT define any storage locations referenced by an output list or namelist in an output statement. This allows a knowledgeable user to READ or WRITE massive amounts of data to a file, without ever waiting for completion, as long as they close the file or perform some other wait operation before modifying or referencing any storage locations referenced by an input / output list or namelist.

Insert a new section:
In section 9.4.1.9 (page 147), first sentence, insert

without an ASYNCHRONOUS specifier

before "terminates", and add the following as the last sentence of that paragraph:

If an ASYNCHRONOUS specifier is present, the variable specified in the SIZE= specifier, if any, will become defined, with the value described above, when the wait operation corresponding to the non-advancing input statement is executed.

Note: A CLOSE, INQUIRE or a file positioning statement, as well as a WAIT statement, can be a wait operation (9.3.5).

9.4.1.10 Asynchronous specifier

The ASYNCHRONOUS specifier indicates that this data transfer operation can be performed asynchronously. Records read or written by asynchronous data transfer statements will be read, written, and processed in the same order as they would have been if the data transfer statement did not contain the ASYNCHRONOUS specifier.

The ASYNCHRONOUS specifier shall not be present in a READ or WRITE statement unless the OPEN statement for the unit referenced in the READ or WRITE statement contained an ASYNCHRONOUS specifier.

When a data transfer statement with the ASYNCHRONOUS specifier is executed, the program shall not execute any statements that would cause any variable in the input / output list, namelist, or the variable specified in a SIZE= specifier to become undefined as described in 14.7.6, until the corresponding wait operation is performed. When a namelist group name is specified in data transfer statement with the ASYNCHRONOUS specifier, any variables in the namelist group that are not actually read or written by the data transfer statement are not subject to the restrictions described in this paragraph.

When a data transfer statement with the ASYNCHRONOUS specifier is executed, the program shall not execute any statements that would cause the pointer association status of any variable in the input / output list, namelist, or a variable specified in the SIZE= specifier to change, or would cause any such variable to become associated with a different target, as described in 14.6.2, until the corresponding wait operation is performed. When a namelist group name is specified in a data transfer statement, variables in the namelist group not actually read or written by the data transfer statement are not subject to the restrictions described in this paragraph.

Note: These last two restrictions ensure that certain variables referenced in asynchronous data transfer statements must still exist and reference the same storage locations when the corresponding wait operation is performed, including the implicit CLOSE for open units when a program is exiting.

When an input data transfer statement with the ASYNCHRONOUS specifier is executed, the input list or namelist items, and the variable specified in the SIZE= specifier, if any, become undefined until the corresponding wait operation is executed (9.3.5, 9.5). When a namelist group name is specified in a data transfer statement, variables in the namelist group not actually read by the data transfer statement do not become undefined.

When a data transfer statement with the ASYNCHRONOUS specifier is executed, the item list or namelist items shall not be redefined until the corresponding wait operation is executed (9.3.5, 9.5). When a namelist group name is specified in such an data transfer statement, variables in the namelist group not written by the data transfer statement may be redefined before the corresponding wait operation.

When a READ statement with the ASYNCHRONOUS specifier is executed, the program shall not execute any procedure call where any variable :

1) in the input / output list or namelist, or
3) specified in a SIZE= specifier,

or subobject or parent object thereof, is passed as an actual argument, until the corresponding wait operation is executed, unless :

1) the actual argument passed does not include any storage location defined or referenced by the data transfer statement,

2) the corresponding dummy argument is an assumed shape array, or

3) the corresponding dummy argument has the ASYNCHRONOUS attribute.

Note: This restriction prevents interactions between actual arguments passed with so-called copyin/copyout semantics and asynchronous I/O.

Insert a new section 9.4.1.11:

9.4.1.11 ID= specifier

The ID= specifier identifies a variable that is assigned a processor dependent value during the execution of an asynchronous data transfer statement. This value can be used in a WAIT statement to force the processor to wait for a particular data transfer operation to complete.

In section 9.4.4, list item (5), change "namelist" to

namelist, except that if the ASYNCHRONOUS= specifier was also present, the entities specified in the input/output list or namelist become undefined

In section 9.4.4, list item (8), change "defined" to

defined, except that a variable specified in a SIZE= specifier becomes undefined if an ASYNCHRONOUS specifier was also specified

In section 9.4.4.4, page 152, before the paragraph that starts "On output ...", insert the following paragraphs:

If an ASYNCHRONOUS specifier is specified in a data transfer statement, the actual list processing and data transfers may occur during execution of the input statement, during execution of the corresponding wait operation, or anywhere in-between. The data transfer operation is considered to be a pending data transfer operation until a corresponding wait operation is performed.

If an ASYNCHRONOUS specifier is specified on an input statement, the list items or namelist variables, and the variable specified in the SIZE= specifier, if any, become undefined until the corresponding wait operation is executed (9.3.5, 9.5). When a namelist group name is specified in a data transfer statement, variables in the namelist group not actually read by the input statement do not become undefined.

If an ASYNCHRONOUS specifier is specified on an output statement, the list items or namelist variables shall not be redefined until the corresponding wait operation is executed (9.3.5, 9.5). When a namelist group name is specified in an output statement, variables in the namelist group not actually written by the data transfer statement are not subject to the restrictions described in this paragraph.

When a data transfer operation is performed asynchronously, any errors that would have caused the ERR= branch on a non-asynchronous READ or WRITE to be taken, and the IOSTAT variable to be defined with a non-zero value, may instead occur during execution of the corresponding wait operation (a WAIT, CLOSE, INQUIRE or file positioning statement) and take the ERR= branch of that wait operation instead. If an ID= specifier is not present in the initiating READ or WRITE statement, the errors may occur during the execution of any subsequent data transfer statement for that same unit, and not just during the corresponding wait operation.

Insert a new section 9.5, and renumber every section thereafter appropriately:

9.5 WAIT statement

Execution of a WAIT statement causes the processor to wait for one of more previously initiated (pending) asynchronous data transfers to complete.

```
R919  <wait-statement> is  WAIT (<wait-spec-list>)

R920  <wait-spec> is  [UNIT = ]
                         <external-file-unit>
                  or  IOSTAT =
                         <scalar-default-int-variable>
                  or  ERR = <label>
                  or  END = <label>
                  or  EOR = <label>
                  or  ID = <scalar-default-int-variable>
```

Constraint: A <wait-spec-list> shall contain exactly one <external-file-unit> specifier, and may contain at most one of each of the other specifiers.

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier shall be the first item in the <wait-spec-list>.

(note to Richard Maine: insert other appropriate constraints, similar to the position-spec constraints, and one for the END=label branch target)

The IOSTAT=, ERR=, and END= specifiers are described in x, x, and x respectively.

If an ID= specifier is not present, the processor waits for all pending data transfers on the specified unit to complete, if any. If an ID= specifier is present, the processor waits for the corresponding READ or WRITE operation to complete. The corresponding READ or WRITE operation is that READ or WRITE that returned the same value for the ID= specifier for the specified unit. The value specified for the ID= specifier shall be a value returned by a READ or WRITE statement for the specified unit, for which a corresponding wait operation has not been executed.

The data transfer operation specified in the corresponding READ or WRITE statement may happen when the WAIT statement is executed, when the corresponding READ or WRITE statement was previously executed, or anytime in-between. The WAIT statement is considered to be a data transfer statement for purposes of end of file, end of record, and error processing.

Note: The CLOSE , INQUIRE, and file positioning statements, as well as the WAIT statement, can be a "wait" operation.

Note: If an asynchronous READ attempts to read beyond the end of a file, then the end of file condition may occur either during execution of the READ statement or during execution of the corresponding wait operation. If the end of file condition occurs during the wait operation, and there is not an END= or IOSTAT= specifier in the statement that is the corresponding wait operation, then program execution terminates. Error conditions are handled in a similar manner.

and renumber all subsequent rules.

In the old section 9.5 (File Positioning statements), add the following after the last sentence in that section:

Execution of a file positioning statement causes the processor to wait for all pending data transfer operations for the specified unit to complete.

If a file positioning statement is executed for a unit with pending data transfer operations, that statement is considered to be the corresponding wait operation for the READ or WRITE statements that initiated the pending data transfers, and is also considered to be an data transfer statement for purposes of end of file, error, and end of record processing.

In section 9.6.1, add the following to rule 924:
  or ID = <scalar-default-int-variable>
  or PENDING = <scalar-default-logical-variable>

and add these constraints around line 40 on page 156:

Constraint: The ID= and PENDING= specifiers shall not appear in an INQUIRE statement if the FILE = specifier is present.

Constraint: If an ID= specifier is present, a PENDING= specifier shall also be present.

On page 159, add section 9.6.1.23

9.6.1.23   ID= and PENDING= specifiers in the INQUIRE statement

If an ID= specifier is not present in an INQUIRE statement, the variable specified in the PENDING= specifier is assigned the value true if there are any pending asynchronous data transfers for the specified unit that have not completed. If an ID= specifier is present, the variable specified in the PENDING= specifier is assigned the value true if the data transfer identified by the ID= specifier for the specified unit has not yet completed. In all other cases, the variable specified in the PENDING= specifier is set to false.

When the variable specified in the PENDING= specifier is set to false, then any pending data transfer operations for this unit are considered to have completed, and this INQUIRE is the corresponding wait operation for the corresponding READ or WRITE statements. When an ID= specifier is present, the corresponding operation is the READ or WRITE statement identified by the unit and ID= specifier value. When an ID= specifier was not present, then this INQUIRE statement is the corresponding wait operation for all pending data transfer operations for the specified unit. When an INQUIRE statement is considered to be a wait operation, it is also considered to be a data transfer statement for purposes of end of file, end of record, and error processing.

In section 9.6.1.14, add the following sentence as the last sentence of the paragraph.

If there are pending data transfer operations for the specified unit, the value assigned to the variable specified in a NEXTREC= specifier is computed as if all the pending data transfers had already completed.

Note to the reader: the POSITION= specifier does not appear to need any edits.

Note to the reader. In section 14, we discuss events causing definition and undefinition of variables. In item (3) of 14.7.5, we discuss when input causes an item to be defined, in terms of when the data is transferred, so no edit is needed in (3). Note that the second part of (3) applies to internal units, which cannot be written to asynchronously.

In section 12.3.1.1, add this item under the list (2)
(f) A dummy argument that has the ASYNCHRONOUS attribute, or

and delete the trailing " or" from item (e) in that list.

In section 14.7.5, item (5), change "an input/output statement" to "an input/output statement without the ASYNCHRONOUS specifier".

In section 14.7.5, item (8), change "statement" to "statement without an ASYNCHRONOUS specifier".

In section 14.7.5, insert this new item (9), and renumber the remaining items:
(9) Execution of a READ statement containing both an ASYNCHRONOUS and a SIZE= specifier may cause the variable specified in the SIZE= specifier to become defined, or the corresponding wait operation may cause that variable to become defined. Either the READ statement or the corresponding wait operation will cause that variable to become defined.

In section 14.7.6, item (4), change "input/output statement" to "input/output statement or its corresponding wait operation".

In section 14.7.6, item (5), change "input/output statement" to "input/output statement or its corresponding wait operation".

In section 14.7.6, item (7), change "input statement" to "input statement or its corresponding wait operation".

In section 14.7.6, add a new item (16) (the editor may relocate to another part of the list if desired):
Execution of a READ or WRITE statement with the ASYNCHRONOUS specifier causes all variables in the item list or namelist, and the variable specified in the SIZE= specifier, if any, to become undefined. Variables in a namelist group specified in such a READ or WRITE statement that are not actually read or written by the data transfer statement do not become undefined.

Rationale for Asynchronous I/O: may be inserted in the appropriate annex if desired.

Rather than limit support for asynchronous I/O to what has been traditionally provided by facilities such as BUFFERIN-BUFFEROUT, this standard builds upon existing Fortran syntax. This permits alternative approaches for implementing asynchronous I/O, and simplifys the task of adapting existing standard conforming programs to utilize asynchronous I/O.

Not all processors will actually perform I/O asynchronously, nor will every processor that does, be able to handle data transfer statements with complicated I/O item lists in an asynchronous manner. Such processors can still be standard conforming. Hopefully, the documentation for each Fortran processor will describe when, if ever, I/O will be performed asynchronously.

Conceptual Model

This proposal accomodates at least two different conceptual models for asynchronous I/O.

Model 1: the processor will perform asynchronous I/O when the item list is simple (perhaps one contiguous named array) and the I/O is unformatted (possibly MAGTAPE). The implementation cost is reduced, and this is the scenario most likely to be beneficial on traditional "big-iron" machines.

Model 2: The processor is free to do any of the following:
on output, create a buffer inside the I/O library, completely formatted, and then start an async write of the buffer, and immediately return to the next statement in the program. The processor is free to wait for previously issued WRITEs, or not.

OR

pass off the I/O list addresses to another processor/process, that will process the list items independently of the processor which executes the users code. There is still an ordering requirement on list item processing, to handle things like READ (...) N,(a(i),i=1,N). The addresses of the list items must be computed before the async I/O READ/WRITE statement completes.

One source of confusion is the role of the ID= values and wait operations. The standard allows a user to issue an large number of async I/O requests, without waiting for any of them to complete, and to then wait for any or all of them. It may be impossible, and undesirable to keep track of each of these I/O requests individually.

The proposed support does not require all requests to be tracked by the runtime library. When the user does NOT specify an ID= specifier on a READ or WRITE, the runtime is free to forget about this particular request once it has successfully completed. If it gets an ERR or END condition, the processor is free to report this during any I/O operation to that unit.

When an ID= specifier is present, the runtime is required to keep track of any END or ERR conditions for that specific I/O request. However, if the I/O request succeeds without any exceptional conditions occuring, then the runtime can forget about that ID= value if it wishes. Typically, I except a runtime to only keep track of the last request made, or perhaps a very few. Then, when a user WAITs for a particular request, either the library knows about it (and does the right thing w.r.t. error handling, etc.), or will assume it is one of those requests that successfully completed and was forgotten about (and will just return without signaling any end/err conditions). It is encumbent on the user to only pass in valid ID= values. There is no requirement on the processor to detetct invalid ID= values.

There is of course, a processor dependent limit on how many outstanding I/O requests which generate an END or ERROR conditions can be handled before the processor runs out of memory to keep track of such stuff.

The restrictions on the SIZE= variables are designed to allow the processor to update such variables at any time (after the request has been processed, but before the WAIT operation), and to then forget about them. That's why there is no SIZE= specifier allowed in the various WAIT operations. Only exceptional conditions (errors or EOFs) are expected to be tracked by individual request by the runtime, and then only if an ID= specifier was present.

The EOR= specifier has not been added to the WAIT operations. Instead, the IOSTAT variable will have to be queried after a WAIT operation to handle this situation. This choice was made because an EOR condition is not perceived to be an exceptional condition, like those that trigger and END= or ERR= branch. This particular choice is philosophical, and was not based on significant technical difficulties.

Note that the requirement to set the IOSTAT variable correctly requires an implementation to remember which I/O requests got an EOR condition, so that a subsequent wait operation will return the correct IOSTAT value. This means there is a processor defined limit on the number of outstanding I/O requests (non-advancing) which got an EOR condition (constrained by available memory to keep track of this info, similar to END/ERR conditions).