Section 2

High Performance Fortran Terms and Concepts

This Section presents some rationale for the selection of Fortran 90 as HPF's base language, HPF's model of computation, and the high level syntax and lexical rules for HPF directives.

2.1 Fortran 90

The facilities for array computation in Fortran 90 make it particularly suitable for programming scientific and engineering numerical calculations on high performance computers. Indeed, some of these facilities are already supported in compilers from a number of vendors. The introductory overview in the Fortran 90 standard states:

> *Operations for processing whole arrays and subarrays (array sections) are included in the language for two principal reasons: (1) these features provide a more concise and higher level language that will allow programmers more quickly and reliably to develop and maintain scientific/engineering applications, and (2) these features can significantly facilitate optimization of array operations on many computer architectures.*
>
> — Fortran Standard (page xiii)

Other features of Fortran 90 that improve upon the features provided in FORTRAN 77 include:

- Additional storage classes of objects. The new storage classes such as allocatable, automatic, and assumed-shape objects as well as the pointer facility of Fortran 90 add significantly to those of FORTRAN 77 and should reduce the use of FORTRAN 77 constructs that can lead to less than full computational speed on high performance computers, such as `EQUIVALENCE` between array objects, `COMMON` definitions with non-identical array definitions across subprograms, and array reshaping transformations between actual and dummy arguments.

- Support for a modular programming style. The module facilities of Fortran 90 enable the use of data abstractions in software design. These facilities support the specification of modules, including user-defined data types and structures, defined operators on those types, and generic procedures for implementing common algorithms to be

used on a variety of data structures. In addition to modules, the definition of interface blocks enables the application programmer to specify subprogram interfaces explicitly, allowing a high quality compiler to use the information specified to provide better checking and optimization at the interface to other subprograms.

- Additional intrinsic procedures. Fortran 90 includes the definition of a large number of new intrinsic procedures. Many of these support mathematical operations on arrays, including the construction and transformation of arrays. Also, there are numerical accuracy intrinsic procedures designed to support numerical programming, and bit manipulation intrinsic procedures derived from MIL-STD-1753.

HPF conforms to Fortran 90 except for additional restrictions placed on the use of storage and sequence association. Because of the effort involved in producing a full Fortran 90 compiler, HPF is defined at two levels: Subset HPF and full HPF. Subset HPF is a subset of Fortran 90 with a subset of the HPF extensions. HPF is Fortran 90 (with the restrictions noted in Section 7) with all of the HPF language features.

2.2 The HPF Model

An important goal of HPF is to achieve code portability across a variety of parallel machines. This requires not only that HPF programs compile on all target machines, but also that a highly-efficient HPF program on one parallel machine be able to achieve reasonably high efficiency on another parallel machine with a comparable number of processors. Otherwise, the effort spent by a programmer to achieve high performance on one machine would be wasted when the HPF code is ported to another machine. Although SIMD processor arrays, MIMD shared-memory machines, and MIMD distributed-memory machines use very different low-level primitives, there is broad similarity with respect to the fundamental factors that affect the performance of parallel programs on these machines. Thus, achieving high efficiency across different parallel machines with the same high level HPF program is a feasible goal. While describing a full execution model is beyond the scope of this language specification, we focus here on two fundamental factors and show how HPF relates to them:

- The parallelism inherent in a computation; and
- The communication inherent in a computation.

The quantitative cost associated with each of these factors is machine dependent; vendors are strongly encouraged to publish estimates of these costs in their system documentation. Note that, like any execution model, these may not reflect all of the factors relevant to performance on a particular architecture.

The parallelism in a computation can be expressed in HPF by the following constructs:

- Fortran 90 array expressions and assignment (including masked assignment in the `WHERE` statement);
- Array intrinsics, including both the Fortran 90 intrinsics and the new intrinsic functions;
- The `FORALL` statement; and
- The `INDEPENDENT` assertion on `DO` loops.

These features allow a user to specify explicitly potential data parallelism in a machine-independent fashion. The purpose of this section is to clarify some of the performance implications of these features, particularly when they are combined with the HPF data distribution features. In addition, EXTRINSIC procedures provide an escape mechanism in HPF to allow the use of efficient machine-specific primitives by using another programming paradigm. Because the resulting model of computation is inherently outside the realm of data-parallel programming, we will not discuss this feature further in this section.

A compiler may choose not to exploit information about parallelism, for example because of lack of resources or excessive overhead. In addition, some compilers may detect parallelism in sequential code by use of dependence analysis. This document does not discuss such techniques.

The interprocessor or inter-memory data communication that occurs during the execution of an HPF program is partially determined by the HPF data distribution directives in Section 3. The compiler will determine the actual mapping of data objects to the physical machine and will be guided in this by the directives. The actual mapping and the computation specified by the program determine the needed actual communication, and the compiler will generate the code required to perform it. In general, if two data references in an expression or assignment are mapped to different processors or memory regions then communication is required to bring them together. The following examples illustrate how this may occur.

Clearly, there is a tradeoff between parallelism and communication. If all the data are mapped to one processor's local memory, then a sequential computation with no communication is possible, although the memory of one processor may not suffice to store all the program's data. Alternatively, mapping data to multiple processors' local memories may permit computational parallelism but also may introduce communications overhead. The optimal resolution of such conflicts is very dependent on the architecture and underlying system software.

The following examples illustrate simple cases of communication, parallelism, and their interaction. Note that the examples are chosen for illustration and do not necessarily reflect efficient data layouts or computational methods for the program fragments shown. Rather, the intent is to derive lower bounds on the amount of communication that are needed to implement the given computations as they are written. This gives some indication of the maximum possible efficiency of the computations on any parallel machine. A particular system may not achieve this efficiency due to analysis limitations, or may disregard these bounds if other factors determine the performance of the code.

2.2.1 Simple Communication Examples

The following examples illustrate the communication requirements of scalar assignment statements. The purpose is to illustrate the implications of data distribution specifications on communication requirements for parallel execution. The explanations given do not necessarily reflect the actual compilation process.

Consider the following statements:

```
      REAL a(1000), b(1000), c(1000), x(500), y(0:501)
      INTEGER inx(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b, inx
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: c
```

```
!HPF$ ALIGN x(i) WITH y(i+1)
      ...
      a(i) = b(i)                      ! Assignment 1
      x(i) = y(i+1)                    ! Assignment 2
      a(i) = c(i)                      ! Assignment 3
      a(i) = a(i-1) + a(i) + a(i+1)    ! Assignment 4
      c(i) = c(i-1) + c(i) + c(i+1)    ! Assignment 5
      x(i) = y(i)                      ! Assignment 6
      a(i) = a(inx(i)) + b(inx(i))     ! Assignment 7
```

In this example, the PROCESSORS directive specifies a linear arrangement of 10 processors. The DISTRIBUTE directives recommend to the compiler that the arrays a, b, and inx should be distributed among the 10 processors with blocks of 100 contiguous elements per processor. The array c is to be cyclically distributed among the processors with c(1), c(11), ..., c(991) mapped onto processor procs(1); c(2), c(12), ..., c(992) mapped onto processor procs(2); and so on. The complete mapping of arrays x and y onto the processors is not specified, but their relative alignment is indicated by the ALIGN directive. The ALIGN statement causes x(i) and y(i+1) to be stored on the same processor for all values of i, regardless of the actual distribution chosen by the compiler for x and y (y(0) and y(1) are not aligned with any element of x). The PROCESSORS, DISTRIBUTE, and ALIGN directives are discussed in detail in Section 3.

In Assignment 1 (a(i) = b(i)), the identical distribution of a and b ensures that for all i, a(i) and b(i) are mapped to the same processor. Therefore, the statement requires no communication.

In Assignment 2 (x(i) = y(i+1)), there is no inherent communication. In this case, the relative alignment of the two arrays matches the assignment statement for any actual distribution of the arrays.

Although Assignment 3 (a(i) = c(i)) looks very similar to the first assignment, the communication requirements are very different due to the different distributions of a and c. Array elements a(i) and c(i) are mapped to the same processor for only 10% of the possible values of i. (This can be seen by inspecting the definitions of BLOCK and CYCLIC in Section 3.) The elements are located on the same processor if and only if $\lfloor (i-1)/100 \rfloor = (i-1) \bmod 10$. For example, the assignment involves no inherent communication (i.e., both a(i) and c(i) are on the same processor) if $i = 1$ or $i = 102$, but does require communication if $i = 2$.

In Assignment 4 (a(i) = a(i-1) + a(i) + a(i+1)), the references to array a are all on the same processor for about 98% of the possible values of i. The exceptions to this are $i = 100k$ for any $k = 1, 2, \ldots, 9$, (when a(i) and a(i-1) are on procs(k) and a(i+1) is on procs(k+1)) and $i = 100k + 1$ for any $k = 1, 2, \ldots, 9$ (when a(i) and a(i+1) are on procs(k+1) and a(i-1) is on procs(k)). Thus, except for "boundary" elements on each processor, this statement requires no inherent communication.

Assignment 5, c(i) = c(i-1) + c(i) + c(i+1), while superficially similar to Assignment 4, has very different communication behavior. Because the distribution of c is CYCLIC rather than BLOCK, the three references c(i), c(i-1), and c(i+1) are mapped to three distinct processors for any value of i. Therefore, this statement requires communication for at least two of the right-hand side references, regardless of the implementation strategy.

The final two assignments have very limited information regarding the communication requirements. In Assignment 6 (x(i) = y(i)) the only information available is that x(i)

and `y(i+1)` are on the same processor; this has no logical consequences for the relationship between `x(i)` and `y(i)`. Thus, nothing can be said regarding communication in the statement without further information. In Assignment 7 (`a(i) = a(inx(i)) + b(inx(i))`), it can be proved that `a(inx(i))` and `b(inx(i))` are always mapped to the same processor. Similarly, it is easy to deduce that `a(i)` and `inx(i)` are mapped together. Without knowledge of the values stored in `inx`, however, the relation between `a(i)` and `a(inx(i))` is unknown, as is the relationship between `a(i)` and `b(inx(i))`.

The inherent communication for a sequence of assignment statements is the union of the communication requirements for the individual statements. An array element used in several statements contributes to the total inherent (i.e. minimal) communication only once (assuming an optimizing compiler that eliminates common subexpressions), unless the array element may have been changed since its last use. For example, consider the code below:

```
      REAL a(1000), b(1000), c(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: a, b, c
      ...
      a(i) = b(i+2)                    ! Statement 1
      b(i) = c(i+3)                    ! Statement 2
      b(i+2) = 2 * a(i+2)              ! Statement 3
      c(i) = a(i+1) + b(i+2) + c(i+3)  ! Statement 4
```

Statements 1 and 2 each require one array element to be communicated for any value of `i`. Statement 3 has no inherent communication. To simplify the discussion, assume that all four statements are executed on the processor storing the array element being assigned. [1] Then, for Statement 4:

- Element `a(i+1)` induces communication, since it is not local and was not communicated earlier;
- Element `b(i+2)` induces communication, since it is nonlocal and has changed since its last use; and
- Element `c(i+3)` *does not* induce new communication, since it was used in statement 2 and not changed since.

Thus, the minimum total inherent communication in this program fragment is four array elements. It is important to note that this is a minimum. Some compilation strategies may produce communication for element `c(i+3)` in the last statement.

2.2.2 Aggregate Communication Examples

The following examples illustrate the communication implications of some more complex constructs. The purpose is to show how communication can be quantified, but again the explanations do not necessarily reflect the actual compilation process. It is important to note that the communication requirement for each statement in this section is estimated without considering the surrounding context.

Consider the following statements:

[1] This is an optimal strategy for this example, although not for all programs.

```
      REAL a(1000), b(1000), c(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: c
      ...
      FORALL ( i = 1:1000 ) a(i) = b(i)        ! Forall 1
      FORALL ( i = 1:1000 ) a(i) = c(i)        ! Forall 2

      ! Forall 3
      FORALL ( i = 2:999 ) a(i) = a(i-1) + a(i) + a(i+1)

      ! Forall 4
      FORALL ( i = 2:999 ) c(i) = c(i-1) + c(i) + c(i+1)
```

The FORALL statement conceptually evaluates its right-hand side for all values of its indexes, then assigns to the left-hand side for all index values. These semantics allow parallel execution. Section 4 describes the FORALL statement in detail. The aggregate communication requirements of these statements follow directly from the inherent communication of the corresponding examples in Section 2.2.1.

In Forall 1, there is no inherent communication for any value of i; therefore, there is no communication for the aggregate construct.

In Forall 2, 90% of the references to c(i) are mapped to a processor different from that containing the corresponding a(i). The aggregate communication must therefore transfer 900 array elements. Furthermore, analysis based on the definitions of BLOCK and CYCLIC shows that to update the values of a owned locally, each processor requires data from every other processor. For example, procs(1) must somehow receive:

- Elements $\{2, 12, 22, \ldots, 92\}$ from procs(2);
- Elements $\{3, 13, 23, \ldots, 93\}$ from procs(3); and
- So on for the other processors.

This produces an all-to-all communication pattern similar to the pattern for transposing a 2-dimensional array with certain distributions. The details of implementing such a pattern are very machine dependent and beyond the scope of this standard.

In Forall 3, the array references are all mapped to the same processor except for the first and last values of i on each processor. The aggregate communication requirement is therefore two array elements per processor (except procs(1) and procs(10)), or 18 elements total. Each processor must receive values from its left and right neighbors (again, except for procs(1) and procs(10)). This leads to a simple shift communication pattern (without wraparound).

In Forall 4, the update of each array element requires two off-processor values, each from a different processor. The total communication volume is therefore 1996 array elements. Further analysis reveals that all elements on processor procs(k) require elements from procs(k $\ominus$ 1) and procs(k $\oplus$ 1) (MODULO(k - 2, 10) + 1 and MODULO(k, 10) + 1 respectively, so called "clock arithmetic"). This leads to a massive shift communication pattern (with wraparound).

The aggregate communication for other constructs can be computed similarly. Iterative constructs generate the sum of the inherent communication for nested statements, while

conditionals require at least the communication needed by the conditional branch that is taken. Repeated communication of the same array elements in any construct is not necessary unless the values of those elements may change.

Array expressions require an analysis similar to that for FORALL statements. In these cases, the inherent communication for each element of the result can be analyzed and the aggregate formed on that basis. The following statements have the same communication requirements as the above FORALL statements:

```
      REAL a(1000), b(1000), c(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: c
      ...
      ! Assignment 1 (equivalent to Forall 1)
      a(:) = b(:)

      ! Assignment 2 (equivalent to Forall 2)
      a(1:1000) = c(1:1000)

      ! Assignment 3 (equivalent to Forall 3)
      a(2:999) = a(1:998) + a(2:999) + a(3:1000)

      ! Assignment 4 (equivalent to Forall 4)
      c(2:999) = c(1:998) + c(2:999) + c(3:1000)
```

Some array intrinsics have inherent communication costs as well. For example, consider:

```
      REAL a(1000), b(1000), scalar
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b
      ...
      ! Intrinsic 1
      scalar = SUM( a )

      ! Intrinsic 2
      a = SPREAD( b(1), DIM=1, NCOPIES=1000 )

      ! Intrinsic 3
      a = CSHIFT(a,-1) + a + CSHIFT(a,1)
```

In general, the inherent communication derives from the mathematical definition of the function. For example, the inherent communication for computing SUM is one element for each processor storing part of the operand, minus one. (Further communication may be needed to store the result.) The optimal communication pattern is very machine-specific. Similar remarks apply to any accumulation operation; prefix and suffix intrinsics may require a larger volume based on the distribution. The SPREAD operation above requires a broadcast from procs(1) to all processors, which may take advantage of available hardware. The CSHIFT operations produce a shift communication pattern (with wraparound). This list of examples illustrating array intrinsics is not meant to be exhaustive.

There are other examples of situations in which nonaligned data must be communicated:

```
      REAL a(1000), c(100,100), d(100,100)
!HPF$ PROCESSORS procs(10)
!HPF$ ALIGN c(i,j) WITH d(j,i)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a
!HPF$ DISTRIBUTE (BLOCK,*) ONTO procs :: d
      ...
      a(1:200) = a(1:200) + a(2:400:2)
      c = c + d
```

In the first assignment, the use of different strides in the two references to a on the right-hand side will cause communication. The second assignment statement requires either a transpose of c or d or some complex communication pattern overlapping computation and communication.

A REALIGN directive may change the location of every element of the array. This will cause communication of all elements that change their home processor; in some compilation schemes, data will also be moved to new locations on the same processor. The communication volume is the same as an array assignment from an array with the original alignment to another array with the new alignment. The REDISTRIBUTE statement changes the distribution for every array aligned to the operand of the REDISTRIBUTE. Therefore, its cost is similar to the cost of a REALIGN on many arrays simultaneously. Compiler analysis may sometimes detect that data movement is not needed because an array has no values that could be accessed; such analysis and the resulting optimizations are beyond the scope of this document.

2.2.3 Interaction of Communication and Parallelism

The examples in Sections 2.2.1 and 2.2.2 were chosen so that parallelism and communication were not in conflict. The purpose of this section is to show cases where there is a tradeoff. The best implementation of all these examples will be machine dependent. As in the other sections, these examples do not necessarily reflect good programming practice.

Analyzing communication as in Sections 2.2.1 and 2.2.2 does not completely determine a program's performance. Consider the code:

```
      REAL x(100), y(100)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs:: x, y
      ...
      DO k = 3, 98
         x(k) = y(k) * (x(k-1) + x(k) + x(k+1)) / 3.0
         y(k) = x(k) + (y(k-1) + y(k-2) + y(k+1) + y(k+2)) / 4.0
      ENDDO
```

Only a few values need be communicated at the boundary of each processor. However, every iteration of the DO loop uses data computed on previous iterations for the references x(k-1), y(k-1), and y(k-2). Therefore, although there is little inherent communication, the computation will run sequentially.

In contrast, consider the following code:

```
      REAL x(100), y(100), z(100)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs:: x, y, z
      ...
!HPF$ INDEPENDENT
      DO k = 3, 98
         x(k) = y(k) * (z(k-1) + z(k) + z(k+1)) / 3.0
         y(k) = x(k) + (z(k-1) + z(k-2) + z(k+1) + z(k+2)) / 4.0
      ENDDO
```

The INDEPENDENT directive asserts to the compiler that the iterations of the DO loop are completely independent of each other and none of the data accessed in the loop by an iteration is written by any other iteration.[2] Therefore, the loop has substantial potential parallelism and is likely to execute much faster than the last example. Section 4 describes the INDEPENDENT directive in more detail.

Assignment of work to processors may itself require communication. Consider the following code:

```
      INTEGER indx(1000), inv(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: indx, inv
      ...
      FORALL ( j = 1:1000 ) inv(indx(j)) = j**2
```

(Here, indx must be a permutation of the integers from 1 to 1000 in order for the FORALL to be well-defined.) Since the processor owning element inv(indx(j)) depends on the values stored in indx, some data must be communicated simply to determine where the results will be stored. Two possible implementations of this are:

- Each processor calculates the squares for elements of indx that it owns and performs a scatter operation to communicate those values to the elements of inv where the final results are stored.
- Each processor determines the owner of inv(indx(j)) for all elements of indx that it owns and notifies those processors. Each processor then computes the right-hand side for all elements for which it received notification.

In either case, nontrivial communication must be performed to distribute the work among processors. The optimal sharing scheme, its implementation, and its cost will be highly architecture dependent.

The parallelism in a section of code may conflict with the distribution of data, thus limiting the overall performance. Consider the following code:

```
      REAL a(1000,1000), b(1000,1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK,*) ONTO procs :: a, b
      ...
      DO i = 2, 1000
         a(i,:) = a(i,:) - (b(i,:)**2)/a(i-1,:)
      ENDDO
```

[2] Many compilers would detect this without the assertion. What cases of implicit parallelism are detected is highly compiler dependent and beyond the scope of this document.

Here, each iteration of the DO loop has a potential parallelism of 1000. However, all elements of a(i,:) and b(i,:) are located on the same processor. Therefore, exploitation of any of the potential parallelism will require scattering the data to other processors. (This is independent of the inherent communication required for the reference to a(i-1,:).) There are several implementation strategies available for the overall computation.

- Redistribute a and b before the DO loop to achieve the effect of

  ```
  !HPF$ DISTRIBUTE (*,BLOCK) ONTO procs :: a, b
  ```

  Redistribute back to the original distributions after the DO loop. This allows parallel updates of columns of a, at the cost of two all-to-all communication operations.

- Group the columns of a into blocks, then operate on the blocks separately. This strategy can produce a pipelined effect, allowing substantial parallelism. It sends many small messages to the neighboring processor rather than one large message.

- Execute the vector operations sequentially. This results in totally sequential operation, but avoids overhead from process start-up and small messages.

This list is not exhaustive. The optimal strategy will be highly machine dependent.

There is often a choice regarding where the result of an intermediate array expression will be stored, and different choices may lead to different communication performance. A straightforward implementation of the following code, for example, would require two transposition (communication) operations:

```
      REAL, DIMENSION(100,100) :: x, y, z
!HPF$ ALIGN WITH x :: y, z
      ...
      x = TRANSPOSE(y) + TRANSPOSE(z) + x
```

Despite two occurrences of the TRANSPOSE intrinsic, an optimizing compiler might implement this as:

```
      REAL, DIMENSION(100,100) :: x, y, z, t1
!HPF$ ALIGN WITH x :: y, z, t1
      ...
      t1 = y + z
      x = TRANSPOSE(t1) + x
```

with only one use of transposition.

Choosing an intermediate storage location is sometimes more complex, however. Consider the following code:

```
      REAL a(1000), b(1000), c(1000), d(1000)
      INTEGER ix(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs:: a, b, c, d, ix
      ...
      a = b(ix) + c(ix) + d(ix)
```

and the following implementation strategies:

- Evaluate each element of the right-hand side on the processor where it will be stored. This strategy potentially requires fetching three values (the elements of `b`, `c`, and `d`) for each element computed. It always uses the maximum parallelism of the machine.
- Evaluate each element of the right-hand side on the processor where the corresponding elements of `b(ix)`, `c(ix)`, and `d(ix)` are stored. Ignoring set-up costs, this potentially communicates one result for each element computed. If the values of `ix` are evenly distributed, then it also uses the maximum machine parallelism.

On the basis of communication, the second strategy is better by a factor of 3; adding additional terms can make this factor arbitrarily large. However, that analysis does not consider parallel execution costs. If there are repeated values in `ix`, the second strategy may produce poor load balance. (For example, consider the case of `ix(i) = 10` for all `i`.) Minimizing this cost is a compiler optimization and is outside the scope of this language specification.

2.3 Syntax of Directives

HPF directives are consistent with Fortran 90 syntax in the following sense: if any HPF directive were to be adopted as part of a future Fortran standard, the only change necessary to convert an HPF program would be to remove the comment character and directive prefix from each directive.

H201 *hpf-directive-line* **is** *directive-origin hpf-directive*

H202 *directive-origin* **is** `!HPF$`
or `CHPF$`
or `*HPF$`

H203 *hpf-directive* **is** *specification-directive*
or *executable-directive*

H204 *specification-directive* **is** *processors-directive*
or *align-directive*
or *distribute-directive*
or *dynamic-directive*
or *inherit-directive*
or *template-directive*
or *combined-directive*
or *sequence-directive*

H205 *executable-directive* **is** *realign-directive*
or *redistribute-directive*
or *independent-directive*

Constraint: An *hpf-directive-line* cannot be commentary following another statement on the same line.

Constraint: A *specification-directive* may appear only where a *declaration-construct* may appear.

Constraint: An *executable-directive* may appear only where an *executable-construct* may appear.

Constraint: An *hpf-directive-line* follows the rules of either Fortran 90 free form (3.3.1.1) or fixed form (3.3.2.1) comment lines, depending on the source form of the surrounding Fortran 90 source form in that program unit. (3.3)

An *hpf-directive* conforms to the rules for blanks in free source form (3.3.1), even in an HPF program otherwise in fixed source form. However an HPF-conforming processor is not required to diagnose extra or missing blanks in an HPF directive. Note that, due to Fortran 90 rules, the *directive-origin* may only be the characters `!HPF$` in free source form. HPF directives may be continued, in which case each continued line also begins with a *directive-origin*. No statements may be interspersed within a continued HPF-directive. HPF directive lines must not appear within a continued statement. HPF directive lines may include trailing commentary.

An example of an HPF directive continuation in free source form is:

```
!HPF$ ALIGN ANTIDISESTABLISHMENTARIANISM(I,J,K) &
!HPF$         WITH ORNITHORHYNCHUS_ANATINUS(J,K,I)
```

An example of an HPF directive continuation in fixed source form follows. Observe that column 6 must be blank, except when signifying continuation.

```
!HPF$ ALIGN ANTIDISESTABLISHMENTARIANISM(I,J,K)
!HPF$*WITH ORNITHORHYNCHUS_ANATINUS(J,K,I)
```

This example shows an HPF directive continuation which is "universal" in that it can be treated as either fixed source form or free source form. Note that the "`&`" in the first line is in column 73.

```
!HPF$ ALIGN ANTIDISESTABLISHMENTARIANISM(I,J,K)                    &
!HPF$&WITH ORNITHORHYNCHUS_ANATINUS(J,K,I)
```