High Performance Fortran Language Specification

High Performance Fortran Forum

January 31, 1997
Version 2.0

The High Performance Fortran Forum (HPFF), with participation from over 40 organizations, met from March 1992 to March 1993 to define a set of extensions to Fortran called High Performance Fortran (HPF). Our goal was to address the problems of writing data parallel programs for architectures where the distribution of data impacts performance. While we hope that the HPF extensions will become widely available, HPFF is not sanctioned or supported by any official standards organization. The HPFF had a second series of meetings from April 1994 to October 1994 to consider requests for corrections, clarifications, and interpretations to the Version 1.0 HPF document and also to develop user requirements for possible future changes to HPF. A third set of meetings took place From January 1995 through December 1996 to incorporate features recommended to meet user needs identified in the 1994 meetings.

This document contains all the technical features proposed for the version of the language known as HPF Version 2.0. This copy of the draft was processed by LaTeX on January 31, 1997.

HPFF encourages requests for interpretation of this document, and comments on the language defined here. We will give our best effort to answering interpretation questions, and general comments will be considered in future HPFF language specifications.

Please send interpretation requests to `hpff-interpret@cs.rice.edu`. Your request is archived and forwarded to a group of HPFF committee members who attempt to respond to it.

The text of interpretation requests becomes the property of Rice University.

Contents

Acknowledgments

The High Performance Fortran Forum (HPFF) is a coalition of industrial and academic groups working to suggest a set of standard extensions to Fortran that provide support for high performance programming on a wide variety of machines, including massively parallel SIMD and MIMD systems and vector processors. From its beginning, HPFF has included most vendors delivering parallel machines, a number of government laboratories, and university research groups. Public input has been encouraged. This document defining HPF 2.0 is the third in a series of documents resulting from the HPFF. HPF 2.0 is intended to be a language portable from workstations to massively parallel supercomputers while being able to express the algorithms needed to achieve high performance on specific architectures. HPF 2.0 builds on the efforts of the previous HPFF meetings, primarily in 1992 and 1994. Specific acknowledgments for the many people who contributed to the previous versions of HPF are included in Annex D.

HPFF 2 Acknowledgments

The HPF 2.0 version of the document was prepared during series of meetings in 1995–1996. A number of people shared technical responsibilities for the activities of these meetings:

- Ken Kennedy, Convener and Meeting Chair;
- Mary Zosel, Executive Director;
- Rob Schreiber, Organizer for Control Subgroup;
- Piyush Mehrotra and Guy Steele, Organizers for Distribution Subgroup;
- David Loveman, Organizer for External Subgroup;
- Chuck Koelbel, Editor, assisted by multiple committee members (names later).

Attendance at the HPFF 2 meetings included the following people from organizations that were represented two or more times.

John Levesque, Marc Baber	Applied Parallel Research
Ian Foster	Argonne National Laboratory
Jaspal Subhlok	Carnegie Mellon University
Jim Cowie	Cooperating Systems
Andy Meltzer	Cray Research
Henk Sipps, Will Denissen	Delft University of Technology
David Loveman, Carl Offner	Digital Equipment Corporation
Joel Williamson	Hewlett Packard/Convex

Henry Zongaro	IBM
Piyush Mehrotra	Institute for Computer Applications in Science & Engineering
Mary Zosel	Lawrence Livermore National Laboratory
Bob Boland, Ralph Brickner, Joe Fasel	Los Alamos National Laboratory
J. Ramanujam, Arun Venkatachar	Louisiana State University
Steve Hammond	National Center for Atmospheric Research
Yoshiki Seo	NEC
Alok Choudhary	Northwestern University/Syracuse University
P. Sadayappan,	Ohio State University
Larry Meadows, Zeki Bozkus	The Portland Group, Inc.
Robert Schreiber	RIACS and Hewlett Packard
Ken Kennedy, Charles Koelbel	Rice University
Hon Yau, Tom Haupt	Syracuse University
Guy Steele	Sun Microsystems
Paula Vaughn	Mississippi State University
Carol Munroe, Harvey Richardson	Thinking Machines Corporation
Scott Baden, Steve Fink, Jay Boisseau	University of California, San Diego
Robert Babb	University of Denver
Joel Saltz, Paul Havlak, Manuel Ujaldon	University of Maryland
Jerrold Wagener	X3J3 and University of Oklahoma
Barbara Chapman, Siegfried Benkner, Guy Robinson	University of Vienna

An important activity of HPFF 2 meetings was the processing of the many items submitted for comment, correction, and interpretation. General thanks go to all who submitted comments and interpretation requests. The following people submitted items that resulted in changes to the document text: Yasuharu Hayashi, Michael Hennecke, Larry Meadows, Carol Munroe, Eiji Nunohiro, Carl Offner, Rob Schreiber, David Singleton, Shigeru Uzuhara, Clint Whaley, and—last, but most prolific—Henry Zongaro.

Special thanks go to Chuck Koelbel at Rice University for continued maintenance of the HPFF mailing lists.

Theresa Chatman and staff at Rice University were responsible for meeting planning and organization and Danny Powell continued to handle financial details of the project.

HPFF 2 received direct support for research and administrative activities from grants from ARPA, DOE, and NSF.

Part I

Introduction

This major section describes the organization of the document as a whole. It also defines terms and concepts that are common to High Performance Fortran version 2.0 (described in Part II) and the HPF Approved Extensions (described in Part III). Therefore, it provides necessary background for the succeeding sections.

Section 1

Overview

This document specifies the form and establishes the interpretation of programs expressed in the High Performance Fortran (HPF) language. It is designed as a set of extensions and modifications to the established International Standard for Fortran. At the time of publication of this document, the version of the standard used as a base is informally referred to as "Fortran 95" (ISO/IEC 1539:1997). References to that document are made as follows: Section 13.11.6 in that document is referred to here as F95:13.11.6.

In this overview Section of the document, we outline the goals and scope of the language, introduce the HPF language model, highlight the main features of the language, describe the changes between HPF 1.1 and HPF 2.0, and provide a guide to the rest of this document.

1.1 Goals and Scope of High Performance Fortran

The primary goals behind the development of the HPF language include:

- Support for data parallel programming (single threaded, global name space, and loosely synchronous parallel computation);
- Portability across different architectures;
- High performance on parallel computers with non-uniform memory access costs (while not impeding performance on other machines);
- Use of Standard Fortran (currently Fortran 95) as a base;
- Open interfaces and interoperability with other languages (e.g., C) and other programming paradigms (e.g., message passing using MPI).

Secondary goals include:

- Implementation feasibility within a limited time span;
- Provision of input to future standards activities for Fortran and C;
- Provision of an evolutionary path for adding advanced features to the language in a consistent manner.

The first version of the language definition, HPF 1.0 was released in May 1993. A number of language features that were defined in HPF 1.0 have now been absorbed into the Fortran 95 language standard (e.g., the `FORALL` statement and construct, and `PURE` procedures). These features are therefore no longer detailed in the definition of HPF 2.0. Information about the evolution of the HPF language (through versions 1.0, 1.1, and 2.0) and an enumeration of the differences between HPF 2.0 from HPF 1.1 may be found in subsection 1.4.

1.2 HPF Language 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 shared-memory machines and distributed-memory machines may use 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. Some of the fundamental factors affecting the performance of a parallel program are the degree of available parallelism, exploitation of data locality, and choice of appropriate task granularity. HPF provides mechanisms for the programmer to guide the compiler with respect to these factors.

The first versions of HPF were defined to extend Fortran 90. HPF 2.0 is defined as an extension to the current Fortran Standard (Fortran 95). Future revisions of HPF will include and be consistent with advances in the Fortran standards, as they are approved by ISO.

Building on Fortran, HPF language features fall into four categories:

- HPF directives;
- New language syntax;
- New library routines; and
- Language changes and restrictions.

HPF directives appear as structured comments that suggest implementation strategies or assert facts about a program to the compiler. When properly used, they affect only the efficiency of the computation performed, but do not change the value computed by the program. The form of the HPF directives has been chosen so that a future Fortran standard may choose to include these features as full statements in the language by deleting the initial comment header.

A few new language features have been defined as direct extensions to Fortran syntax and interpretation. The new HPF language features differ from HPF directives in that they are first-class language constructs and can directly affect the result computed by a program.

The HPF library of computational functions defines a standard interface to routines that have proven valuable for high performance computing. These additional functions include those for mapping inquiry, bit manipulation, array reduction, array combining scatter, prefix and suffix, and sorting.

A small number of changes and restrictions to Fortran 95 have also been defined. The most significant restrictions are those imposed on the use of sequence and storage association, since they are not compatible with the data distribution features of HPF. It is however possible to retain sequence and storage association semantics in a program by use of certain explicit HPF directives.

1.2.1 Data Mapping Directives

The fundamental model of parallelism in HPF is that of single-threaded data-parallel execution with a globally shared address space. Fortran array statements and the `FORALL` statement are natural ways of specifying data parallel computation. In addition, HPF provides the `INDEPENDENT` directive. It can be used to assert that certain loops do not carry any dependences and therefore may be executed in parallel.

Exploitation of data locality is critical to achieving good performance on a high-performance computer, whether a uniprocessor workstation, a network of workstations, or a parallel computer. On a Non-Uniform-Memory-Access (NUMA) parallel computer, the effective distribution of data among processor memories is very important in reducing data movement overheads. One of the key features of HPF is the facility for user specification of data mapping. HPF provides a logical view of the parallel machine as a rectilinear arrangement of abstract processors in one or more dimensions. The programmer can specify the relative alignment of elements of different program arrays, and the distribution of arrays over the logical processor grid. Data mapping is specified using HPF directives that can aid the compiler in optimizing parallel performance, but have no effect on the semantics of the program. This is illustrated by the following simple example.

```
      REAL A(1000,1000)
!HPF$ PROCESSORS procs(4,4)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO procs :: A
      DO k = 1, num_iter
         FORALL (i=2:999, j=2:999)
           A(i,j) = (A(i,j-1) + A(i-1,j) + A(i,j+1) + A(i+1,j))/4
         END FORALL
      END DO
```

The code fragment describes a simple Jacobi relaxation computation using a two-dimensional floating-point array `A`. The HPF directives appear as structured comments. The `PROCESSORS` directive specifies a logical 4×4 grid of processors `proc`. The `DISTRIBUTE` directive recommends that the compiler partition the array `A` into equal-sized blocks along each of its dimensions. This will result in a 4×4 configuration of blocks each containing 250×250 elements, one block per processor. The `PROCESSORS` and `DISTRIBUTE` directive are described in detail later in Section 3.

The outer `DO k` loop iterates over `num_iter` Jacobi relaxation steps. The inner loop uses the Fortran 95 `FORALL` construct. It specifies the execution of the loop body for all values of `i` and `j` in the range 2 through 999. The semantics of the `FORALL` require that the right-hand-side expressions for all iterations (i.e., for all values of `i` and `j` between 2 and 999) be evaluated before any of the assignments to the left-hand-side variables are performed.

When targeted for execution on a distributed-memory machine with 16 processors, an HPF compiler generates SPMD code, with each processor locally containing a part of the global array `A`. The outer `k` loop is executed sequentially while the inner `FORALL` is executed in parallel. Each processor will require some "boundary" elements of `A` that reside in partitions mapped to the local memories of other processors. Primitives to achieve the necessary inter-processor communication are inserted by the HPF compiler into the generated SPMD code. The single-threaded data-parallel model with a global name-space makes it convenient for the programmer to specify the strategy for parallelization and data partitioning at a higher level of abstraction. The tedious low-level details of translating from an abstract global name space to the local memories of individual processors and the management of explicit inter-processor communication are left to the compiler.

The following example illustrates some of the communication implications 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 code fragment:

```
      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 recommends that `x(i)` and `y(i+1)` be stored on the same processor for all values of `i`, regardless of the actual distribution chosen by the compiler for `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` specifies that for all `i`, corresponding elements of `a(i)` and `b(i)` should be mapped to the same processor. Therefore, execution of this statement requires no communication of data values between processors.

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 from the definitions of `BLOCK` and `CYCLIC` in Section 3.) The elements are located on the same processor if and only if $\lfloor(\mathtt{i}-1)/100\rfloor = (\mathtt{i}-1) \bmod 10$. For example, the assignment involves no inherent communication (i.e., both `a(i)` and `c(i)` are on the same processor) if $\mathtt{i} = 1$ or $\mathtt{i} = 102$, but does require communication if $\mathtt{i} = 2$.

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 $\mathtt{i} = 100 * \mathtt{k}$ 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 $\mathtt{i} = 100 * \mathtt{k} + 1$ 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)`). This statement requires communication. only for "boundary" elements on each processor,

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 required at runtime for 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))`.

1.3 Overview of HPF 2.0 Language Features

The language defined in this document consists of two main parts:

- The HPF 2.0 Language (Part II)
- HPF 2.0 Approved Extensions (Part III)

The HPF 2.0 language includes features that are expected to be implementable within a year of release of the language specification. These include basic data distribution features, data parallel features, intrinsic and library routines, and the extrinsic mechanism. The Approved Extensions include advanced features that meet specific needs, but are not likely to be supported in initial compiler implementations.

1.3.1 HPF 2.0 Language Features

1.3.1.1 Data Distribution Features (Sections 3 and 4)

Most parallel and sequential architectures attain their highest speed when the data accessed exhibits locality of reference. The sequential storage order implied by Fortran standards often conflicts with the locality demanded by the architecture. To avoid this, HPF includes features that describe the co-location of data (`ALIGN`) and the partitioning of data among memory regions or abstract processors (`DISTRIBUTE`). Compilers may interpret these annotations to improve storage allocation for data, subject to the constraint that semantically every data object has a single value at any point in the program. Section 4 defines how the mapping features interact across subprogram boundaries.

While a goal of HPF is to maintain compatibility with Fortran, full support of Fortran sequence and storage association, however, is not compatible with the goal of high performance through distribution of data in HPF. Sections 3 and 4 describe restrictions and directives related to storage and sequence association.

1.3.1.2 Data Parallel Execution Features (Section 5)

To express parallel computation explicitly, HPF defines the `INDEPENDENT` directive. It asserts that the statements in a particular section of code do not exhibit any sequentializing dependences; when properly used, it does not change the semantics of the construct, but may provide more information to the language processor to allow optimizations. A `REDUCTION` clause can be used with the `INDEPENDENT` directive to identify variables that are updated by commutative and associative operations. This facilitates the utilization of parallelism with reduction operations, in the context of loops where the order of accumulation of updates to a variable is insignificant.

1.3.1.3 Extrinsic Program Units (Section 6)

Because HPF is designed as a high-level machine-independent language, there are certain operations that are difficult or impossible to express directly. For example, an application may benefit from finely-tuned systolic communications on certain machines; HPF's global address space does not express this well. HPF defines the Extrinsic mechanism to facilitate interfacing with procedures written in other paradigms, such as explicit message-passing subroutine libraries or in other languages, such as C.

1.3.1.4 Intrinsic Functions and Standard Library (Section 7)

Experience with massively parallel machines has identified many basic operations that are useful in parallel algorithm design. The Fortran array intrinsics address some of these. HPF adds several classes of parallel operations to the language definition as intrinsic functions and as standard library functions. In addition, several system inquiry functions useful for controlling parallel execution are provided in HPF.

1.3.2 HPF 2.0 Approved Extensions

1.3.2.1 Extensions for Data Mapping (Section 8)

The extended mapping features permit greater control over the mapping of data, including facilities for dynamic realignment and redistribution of arrays at run-time (`REALIGN, REDISTRIBUTE, DYNAMIC` directives), mapping of data among subsets of processors, mapping of pointers and components of derived types, and support for irregular distribution of data (`GEN_BLOCK` and `INDIRECT` distributions). In addition, mechanisms are defined that permit the programmer to provide information to the compiler about the range of possible distributions an array might take (`RANGE` directive) and the amount of buffering to be used with arrays involved in stencil-based nearest-neighbor computations (`SHADOW`).

1.3.2.2 Extensions for Data and Task Parallelism (Section 9)

The `ON` directive facilitates explicit computation partitioning. The site of recommended execution of a computation can be specified either as an explicitly identified subset of a processor arrangement, or indirectly as the set of processors onto which a data object or template is mapped.

In order to assist the compiler in generating efficient code, the `RESIDENT` directive is defined, to be used in conjunction with an `ON` directive by the programmer. It can be used to assert that all accesses to the specified object within the scope of the `ON` directive are to be found locally on the executing processor. The `TASK_REGION` directive allows the user to specify the concurrent execution of different blocks of code on disjoint processor subsets.

1.3.2.3 Extensions for Asynchronous I/O (Section 10)

In order to permit overlap of I/O with computation, an extension has been defined for asynchronous `READ/WRITE` of direct, unformatted data. This is done through an additional I/O control parameter in the Fortran `READ/WRITE` statement that specifies non-blocking execution and a new statement (`WAIT`).

1.3.2.4 Extensions to Intrinsic and Library Procedures (Section 12)

The approved extensions to the HPF intrinsics and library routines relate mostly to mapping inquiry procedures. Some new inquiry routines are defined and other routines defined by the HPF 2.0 language are extended to facilitate inquiry about extended mapping features, such as mapping to processor subsets, `GEN_BLOCK`, `INDIRECT` and `DYNAMIC` distributions. A generalization of the Fortran `TRANSPOSE` intrinsic is also defined.

1.3.2.5 Approved Extensions for HPF Extrinsics (Section 11)

A number of specific extrinsic interfaces are defined in Section 11 as approved HPF 2.0 extensions. These include interfaces to facilitate interoperability with other languages (e.g., C and FORTRAN 77) as well as interfaces for different models of parallelism (`LOCAL` for SPMD parallel, and `SERIAL` for single-process sequential). Library routines useful in the extrinsic models are defined in Section 11.7. Additional extrinsic interfaces that are formally recognized by the HPF Forum, but not defined and maintained by the Forum, are included in Annexes F and G. The policy and mechanism for formal recognition of such extrinsic interfaces is described in Annex E.

1.4 Changes from HPF 1.1

HPF 2.0 differs from HPF 1.1 in a number of ways:

Repartitioning of the Language: The new document describes two components: the HPF 2.0 language (which is expected to be widely and relatively rapidly implemented) and the set of Approved Extensions (which are not part of HPF 2.0 but may be included in future implementations in response to user demand, as the compilation technology matures.)

Features Now in Standard Fortran: Fortran, instead of Fortran 90 is now defined as the base language for extensions; this implies that HPF includes all features added to Fortran at the 1995 revision. With this revision, a few HPF 1.1 features are now part of the Fortran standard, and hence no longer appear as HPF extensions to Fortran.

Features Removed or Restricted in HPF 2.0: Some features of HPF 1.1, that have not been implemented to date, have been removed from the language because experience has shown that the simplicity gained by doing so outweighs the advantage of the features.

Elimination of the HPF Subset: Unlike HPF 1.1, HPF 2.0 no longer has a recommended minimal subset for faster implementation (i.e. Subset HPF), although the original HPF 1.1 Subset is documented in an annex.

Features Moved to Approved Extensions: A few language features have been moved from HPF 1.1 to the category of Approved Extensions.

New Features of HPF 2.0: A few new features have been added to the base language.

New Approved Extensions: A number of further new features are defined as approved extensions to the language.

Recognized Externally-Supported HPF Extrinsics: Finally, the document acknowledges a new category, HPF-related `EXTRINSIC` interfaces, that are recognized as meeting appropriate standards for such interfaces, but are not included as Approved Extensions. Responsibility for the content of each such interface is assumed by the organization proposing it rather than by the HPF Forum.

Each of these categories is summarized in the following subsections.

1.4.1 Repartitioning of the Language

The HPF Forum had two important goals that were sometimes in conflict:

- Providing advanced language capabilities that users had requested.
- Allowing fast compiler development by vendors.

One compromise made to satisfy both goals was to divide the language definition into two parts. HPF 2.0 is very similar to HPF 1.1, and is expected to be efficiently implemented by a number of vendors within approximately one year from the appearance of this document. Advanced features that require more implementation effort are collected as Approved Extensions. Implementors are encouraged to support these features as rapidly as possible, and users are encouraged to speed this process by making their wishes known to the vendors.

1.4.2 Features Now in Standard Fortran

The following features, which formed part of HPF 1.1, have been removed from the document because they are now part of ISO Fortran:

- The `FORALL` statement and construct;
- The `PURE` attribute for procedures;
- Extensions to the `MINLOC` and `MAXLOC` intrinsics to include an optional `DIM` argument.

1.4.3 Features Removed or Restricted in HPF 2.0

The following features have been removed from the language:

- Sequential arrays may no longer be explicitly mapped;
- In any procedure call in which distributed data may require redistribution, the procedure must now have an explicit interface;
- The treatment of the `INHERIT` directive has been simplified in that it is no longer possible to specify both `INHERIT` and `DISTRIBUTE` together.
- The treatment of pointers has been simplified.

1.4.4 Features Moved to Approved Extensions

The `DYNAMIC` attribute and the `REDISTRIBUTE` and `REALIGN` statements have been moved to the Approved Extensions.

1.4.5 New Features of HPF 2.0

The following new constructs have been introduced in HPF 2.0:

- The `REDUCTION` clause for `INDEPENDENT` loops;
- The new `HPF_LIBRARY` procedures `SORT_DOWN, SORT_UP`.

1.4.6 New Approved Extensions

The Approved Extensions include the following features not part of HPF 1.1:

- Mapping of objects to *processor subsets*;
- Explicit mapping of pointers and components of derived types;
- New distribution formats: `GEN_BLOCK` and `INDIRECT`;
- New directives: `RANGE`, `SHADOW`, `ON`, `RESIDENT`, `TASK_REGION`;
- Additional intrinsic procedures: `ACTIVE_NUM_PROCS`, `ACTIVE_PROCS_SHAPE`, and a generalized `TRANSPOSE` intrinsic;
- New `HPF_LIBRARY` procedures: `HPF_MAP_ARRAY` and `HPF_NUMBER_MAPPED`; revision of procedures `HPF_ALIGNMENT, HPF_DISTRIBUTION` and `HPF_TEMPLATE`;

- Support for asynchronous I/O with a new statement `WAIT`, and an additional I/O control parameter in the Fortran `READ/WRITE` statement;
- Extensions to the `EXTRINSIC` facilities to support interoperability with C and FORTRAN 77.

1.4.7 Recognized Externally-Supported HPF Extrinsics

Two externally supported extrinsic interfaces are recognized in this document:

- HPF_CRAFT: providing an SPMD paradigm with HPF features;
- The Fortran 77 Local Library: defining library support for calling Fortran 77 procedures in local mode.

Section 2

Notation and Syntax

This chapter describes the notational conventions employed in this document and the syntax of HPF directives.

2.1 Notation

This document uses the same notation as the Fortran 95 standard. In particular, the same conventions are used for syntax rules. BNF descriptions of language features are given in the style used in the Fortran standard. To distinguish HPF syntax rules from Fortran rules, each HPF rule has an identifying number of the form H*snn*, where *s* corresponds to the section number and *nn* is a two-digit sequence number. Nonterminals not defined in this document are defined in the Fortran standard. Also note that certain technical terms such as "storage unit" are defined by the Fortran standard.

As previously noted in Section 1, a reference of the form F95:2.4.7 in the text refers to Section 2.4.7 of the Fortran 95 standard.

Part III describes the approved extensions. In some cases this requires extending the syntax rules already introduced in an earlier section. In particular, the syntax rules here are often supersets of similar syntax rules in Part II; in these cases, the names of the nonterminals include the suffix *-extended*. Thus, when a non-terminal such as *name* is redefined it is referred to as *name-extended* under the proviso that any reference to *name* is to be replaced by *name-extended* in the rest of the syntax rules.

When a constraint or restriction in Part II is modified by an approved extension, this fact is noted, in the text, and a forward reference is provided. A downward-pointing double arrow is used in the margin (as here) to highlight such a forward reference. ⇓

Each such modification (in Part III) contains a backward reference to the original language in Part II that is modified. An upward-pointing double arrow is used in the margin (as here) to highlight such a backward reference. ⇑

> *Rationale.* Throughout this document, material explaining the rationale for including features, for choosing particular feature definitions, and for making other decisions, is set off in this format. Readers interested only in the language definition may wish to skip these sections, while readers interested in language design may want to read them more carefully. (*End of rationale.*)

> *Advice to users.* Throughout this document, material that is primarily of interest to users (including most examples of syntax and interpretation) is set off in this format.

Readers interested only in technical material may wish to skip these sections, while readers wanting a more tutorial approach may want to read them more carefully. (*End of advice to users.*)

Advice to implementors. Throughout this document, material that is primarily of interest to implementors is set off in this format. Readers interested only in the language definition may wish to skip these sections, while readers interested in compiler implementation may want to read them more carefully. (*End of advice to implementors.*)

2.2 Syntax of Directives

HPF directives are consistent with Fortran 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 replace the directive-origin with blanks.

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	*inherit-directive*
		or	*template-directive*
		or	*combined-directive*
		or	*sequence-directive*
H205	*executable-directive*	**is**	*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 free form (F95:3.3.1.1) or fixed form (F95:3.3.2.1) comment lines, depending on the source form of the surrounding Fortran source form in that program unit. (F95:3.3)

An *hpf-directive* is case insensitive and 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 language processor is not required to diagnose extra or missing blanks in an HPF directive. Note that, due to Fortran rules, the *directive-origin* in free source form must be the characters `!HPF$`. 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.

The blanks in the adjacent keywords `END FORALL` and `NO SEQUENCE` are optional, in either source form.

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 that 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)
```

Part III introduces new directives, both specifications and executable ones, for the approved extensions to HPF 2.0. These are included below:

H206 *specification-directive-extended* **is** *processors-directive*
or *subset-directive*
or *align-directive*
or *distribute-directive*
or *inherit-directive*
or *template-directive*
or *combined-directive*
or *sequence-directive*
or *dynamic-directive*
or *range-directive*
or *shadow-directive*

H207 *executable-directive-extended* **is** *independent-directive*
or *realign-directive*
or *redistribute-directive*
or *on-directive*
or *resident-directive*

The following rule extends rule R215 of Fortran 95:

H208 *executable-construct-extended* **is** *action-stmt*
or *case-construct*
or *do-construct*
or *if-construct*
or *where-construct*
or *on-construct*
or *resident-construct*
or *task-region-construct*

Part II

High Performance Fortran Language

This major section describes the syntax and semantics of features of the High Performance Fortran language, version 2.0. Some technical terms used herein are defined in Part I; otherwise this description is self-contained. Part III builds upon this material.

Section 3

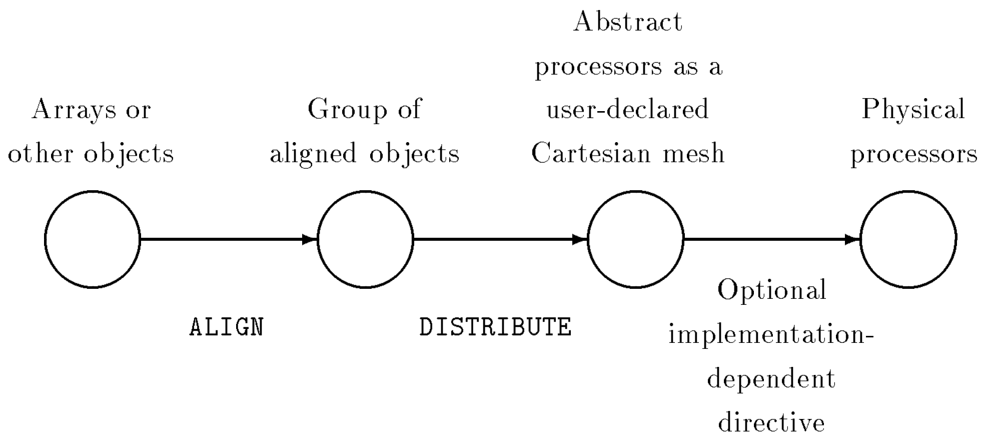
Data Mapping

HPF data alignment and distribution directives allow the programmer to advise the compiler how to assign array elements to processor memories. This section discusses the basic data mapping features applicable, particularly those that are meaningful within a single scoping unit. Section 4 discusses features that apply when mapped variables appear as procedure arguments.

3.1 Model

HPF adds directives to Fortran to allow the user to advise the compiler on the allocation of data objects to processor memories. The model is that there is a two-level mapping of data objects to memory regions, referred to as "abstract processors." Data objects (typically array elements) are first *aligned* relative to one another; this group of arrays is then *distributed* onto a rectilinear arrangement of abstract processors. (The implementation then uses the same number, or perhaps some smaller number, of physical processors to implement these abstract processors. This mapping of abstract processors to physical processors is implementation-dependent.)

The following diagram illustrates the model:


Arrays or other objects
Group of aligned objects
Abstract processors as a user-declared Cartesian mesh
Physical processors
ALIGN
DISTRIBUTE
Optional implementation-dependent directive

The underlying assumptions are that an operation on two or more data objects is likely to be carried out much faster if they all reside in the same processor, and that it may be possible to carry out many such operations concurrently if they can be performed on different processors.

Fortran provides a number of features, notably array syntax, that make it easy for a compiler to determine that many operations may be carried out concurrently. The HPF directives provide a way to inform the compiler of the recommendation that certain data objects should reside in the same processor: if two data objects are mapped (via the two-level mapping of alignment and distribution) to the same abstract processor, it is a strong recommendation to the implementation that they ought to reside in the same physical processor. There is also a provision for recommending that a data object be stored in multiple locations, which may complicate any updating of the object but makes it faster for multiple processors to read the object.

There is a clear separation between directives that serve as specification statements and directives that serve as executable statements (in the sense of the Fortran standards). Specification statements are carried out on entry to a program unit, as if all at once; only then are executable statements carried out. (While it is often convenient to think of specification statements as being handled at compile time, some of them contain specification expressions, which are permitted to depend on run-time quantities such as dummy arguments, and so the values of these expressions may not be available until run time, specifically the very moment that program control enters the scoping unit.)

The basic concept is that every array (indeed, every object) is created with *some* alignment to an entity, which in turn has *some* distribution onto *some* arrangement of abstract processors. If the specification statements contain explicit specification directives specifying the alignment of an array `A` with respect to another array `B`, then the distribution of `A` will be dictated by the distribution of `B`; otherwise, the distribution of `A` itself may be specified explicitly. In either case, any such explicit declarative information is used when the array is created.

Advice to implementors. This model gives a better picture of the actual amount of work that needs to be done than a model that says "the array is created in some default location, and then realigned and/or redistributed if there is an explicit directive." Using `ALIGN` and `DISTRIBUTE` specification directives doesn't have to cause any more work at run time than using the implementation defaults. (*End of advice to implementors.*)

In the case of an allocatable object, we say that the object is created whenever it is allocated. Specification directives for an allocatable object may appear in the *specification-part* of a program unit, but take effect each time the object is created, rather than on entry to the scoping unit.

Alignment is considered an *attribute* (in the Fortran sense) of a data object. If an object `A` is aligned with an object `B`, which in turn is already aligned to an object `C`, this is regarded as an alignment of `A` with `C` directly, with `B` serving only as an intermediary at the time of specification. We say that `A` is *immediately aligned* with `B` but *ultimately aligned* with `C`. If an object is not explicitly aligned with another object, we say that it is ultimately aligned with itself. The alignment relationships form a tree with everything ultimately aligned to the object at the root of the tree; however, the tree is always immediately "collapsed" so that every object is related directly to the root.

Every object that is the root of an alignment tree has an associated *template* or index space. Typically, this template has the same rank and size in each dimension as the object associated with it. (The most important exception to this rule is dummy arguments with the `INHERIT` attribute, described in Section 4.4.2.) We often refer to "the template for an array," which means the template of the object to which the array is ultimately aligned. (When an explicit `TEMPLATE` (see section 3.7) is used, this may be simply the template to which the array is explicitly aligned.)

The *distribution* step of the HPF model technically applies to the template of an array, although because of the close relationship noted above we often speak loosely of the distribution of an array. Distribution partitions the template among a set of abstract processors according to a given pattern. The combination of alignment (from arrays to templates) and distribution (from templates to processors) thus determines the relationship of an array to the processors; we refer to this relationship as the *mapping* of the array. (These remarks also apply to a scalar, which may be regarded as having an index space whose sole position is indicated by an empty list of subscripts.)

Every object is created as if according to some complete set of specification directives; if the program does not include complete specifications for the mapping of some object, the compiler provides defaults. By default an object is not aligned with any other object; it is ultimately aligned with itself. The default distribution is implementation-dependent, but must be expressible as explicit directives for that implementation. Identically declared objects need not be provided with identical default distribution specifications; the compiler may, for example, take into account the contexts in which objects are used in executable code. The programmer may force identically declared objects to have identical distributions by specifying such distributions explicitly. (On the other hand, identically declared processor arrangements *are* guaranteed to represent "the same processors arranged the same way." This is discussed in more detail in section 3.6.)

Sometimes it is desirable to consider a large index space with which several smaller arrays are to be aligned, but not to declare any array that spans the entire index space. HPF allows one to declare a `TEMPLATE`, which is like an array whose elements have no content and therefore occupy no storage; it is merely an abstract index space that can be distributed and with which arrays may be aligned.

An object is considered to be *explicitly mapped* if it appears in an HPF mapping directive within the scoping unit in which it is declared; otherwise it is *implicitly mapped*. A mapping directive is an `ALIGN`, or `DISTRIBUTE`, or `INHERIT` directive, or any directive that confers an alignment, a distribution, or the `INHERIT` attribute.

Note that we extend this model in Section 8 to allow dynamic redistribution and remapping of objects.

3.2 Syntax of Data Alignment and Distribution Directives

Specification directives in HPF have two forms: specification statements, analogous to the `DIMENSION` and `ALLOCATABLE` statements of Fortran; and an attribute form analogous to type declaration statements in Fortran using the "`::`" punctuation.

The attribute form allows more than one attribute to be described in a single directive. HPF goes beyond Fortran in not requiring that the first attribute, or indeed any of them, be a type specifier.

H301 *combined-directive* **is** *combined-attribute-list* :: *combined-decl-list*

H302	*combined-attribute*	**is**	`ALIGN` *align-attribute-stuff*
		or	`DISTRIBUTE` *dist-attribute-stuff*
		or	`INHERIT`
		or	`TEMPLATE`
		or	`PROCESSORS`
		or	`DIMENSION ( `*explicit-shape-spec-list*` )`
H303	*combined-decl*	**is**	*hpf-entity* [`(` *explicit-shape-spec-list* `)`]
		or	*object-name*
H304	*hpf-entity*	**is**	*processors-name*
		or	*template-name*

The `INHERIT` attribute is related to subroutine call conventions and will be discussed in Section 4.

Constraint: The same kind of *combined-attribute* must not appear more than once in a given *combined-directive*.

Constraint: If the `DIMENSION` attribute appears in a *combined-directive*, any entity to which it applies must be declared with the HPF `TEMPLATE` or `PROCESSORS` type specifier.

The following rules constrain the declaration of various attributes, whether in separate directives or in a *combined-directive*.

If the `DISTRIBUTE` attribute is present, then every name declared in the *combined-decl-list* is considered to be a *distributee* and is subject to the constraints listed in section 3.3.

If the `ALIGN` attribute is present, then every name declared in the *entity-decl-list* is considered to be an *alignee* and is subject to the constraints listed in section 3.4.

The HPF keywords `PROCESSORS` and `TEMPLATE` play the role of type specifiers in declaring processor arrangements and templates. The HPF keywords `ALIGN`, `DISTRIBUTE`, and `INHERIT` play the role of attributes. Attributes referring to processor arrangements, to templates, or to entities with other types (such as `REAL`) may be combined in an HPF directive without having the type specifier appear.

No entity may be given a particular attribute more than once.

Dimension information may be specified after an *hpf-entity* or in a `DIMENSION` attribute. If both are present, the one after the *object-name* overrides the `DIMENSION` attribute (this is consistent with the Fortran standard). For example, in:

```
!HPF$ TEMPLATE,DIMENSION(64,64) :: A,B,C(32,32),D
```

`A`, `B`, and `D` are 64×64 templates; `C` is 32×32.

Directives mapping a variable must be in the same scoping unit where the variable is declared.

If a specification expression includes a reference to the value of an element of an array specified in the same specification-part, any explicit mapping or `INHERIT` attribute for the array must be completely specified in prior specification-directives. (This restriction is inspired by and extends F95:7.1.6.2 in the Fortran standard, which states in part: If a specification expression includes a reference to the value of an element of an array specified in the same specification-part, the array bounds must be specified in a prior declaration.)

A comment on asterisks: The asterisk character "`*`" appears in the syntax rules for HPF alignment and distribution directives in three distinct roles:

- When a lone asterisk appears as a member of a parenthesized list, it indicates either a collapsed mapping, wherein many elements of an array may be mapped to the same abstract processor, or a replicated mapping, wherein each element of an array may be mapped to many abstract processors. See the syntax rules for *align-source* and *align-subscript* (see section 3.4) and for *dist-format* (see section 3.3).
- An asterisk appearing in an *align-subscript-use* expression represents the usual integer multiplication operator.
- When an asterisk appears before a left parenthesis "(" or after the keyword WITH or ONTO, it indicates a descriptive or transcriptive mapping for dummy arguments of subprograms (see Section 4) and for mapping of pointers under the approved extensions (see section 8.8). ⇓

An asterisk can also be used in the PASS_BY attribute in an interface block to describe dummy arguments passed by reference to an extrinsic routine written in C (see Section 11.2).

3.3 The DISTRIBUTE Directive

The DISTRIBUTE directive specifies a mapping of data objects to abstract processors in a processor arrangement. For example,

```
      REAL SALAMI(10000)
!HPF$ DISTRIBUTE SALAMI(BLOCK)
```

specifies that the array SALAMI should be distributed across some set of abstract processors by slicing it uniformly into blocks of contiguous elements. If there are 50 processors, the directive implies that the array should be divided into groups of $\lceil 10000/50 \rceil = 200$ elements, with SALAMI(1:200) mapped to the first processor, SALAMI(201:400) mapped to the second processor, and so on. If there is only one processor, the entire array is mapped to that processor as a single block of 10000 elements.

The block size may be specified explicitly:

```
      REAL WEISSWURST(10000)
!HPF$ DISTRIBUTE WEISSWURST(BLOCK(256))
```

This specifies that groups of exactly 256 elements should be mapped to successive abstract processors. (There must be at least $\lceil 10000/256 \rceil = 40$ abstract processors if the directive is to be satisfied. The fortieth processor will contain a partial block of only 16 elements, namely WEISSWURST(9985:10000).)

HPF also provides a cyclic distribution format:

```
      REAL DECK_OF_CARDS(52)
!HPF$ DISTRIBUTE DECK_OF_CARDS(CYCLIC)
```

If there are 4 abstract processors, the first processor will contain DECK_OF_CARDS(1:49:4), the second processor will contain DECK_OF_CARDS(2:50:4), the third processor will contain DECK_OF_CARDS(3:51:4), and the fourth processor will contain DECK_OF_CARDS(4:52:4). Successive array elements are dealt out to successive abstract processors in round-robin fashion.

Distributions are specified independently for each dimension of a multidimensional array:

```
      INTEGER CHESS_BOARD(8,8), GO_BOARD(19,19)
!HPF$ DISTRIBUTE CHESS_BOARD(BLOCK, BLOCK)
!HPF$ DISTRIBUTE GO_BOARD(CYCLIC,*)
```

The CHESS_BOARD array will be carved up into contiguous rectangular patches, which will be distributed onto a two-dimensional arrangement of abstract processors. The GO_BOARD array will have its rows distributed cyclically over a one-dimensional arrangement of abstract processors. (The "*" specifies that GO_BOARD is not to be distributed along its second axis; thus an entire row is to be distributed as one object. This is sometimes called "on-processor" distribution.)

The DISTRIBUTE directive may appear only in the *specification-part* of a scoping unit and can contain only a *specification-expr* as the argument to a BLOCK or CYCLIC option.

The syntax of the DISTRIBUTE directive is:

H305	*distribute-directive*	**is**	DISTRIBUTE *distributee dist-directive-stuff*
H306	*dist-directive-stuff*	**is**	*dist-format-clause* [*dist-onto-clause*]
H307	*dist-attribute-stuff*	**is**	*dist-directive-stuff*
		or	*dist-onto-clause*
H308	*distributee*	**is**	*object-name*
		or	*template-name*
H309	*dist-format-clause*	**is**	(*dist-format-list*)
		or	* (*dist-format-list*)
		or	*
H310	*dist-format*	**is**	BLOCK [(*scalar-int-expr*)]
		or	CYCLIC [(*scalar-int-expr*)]
		or	*
H311	*dist-onto-clause*	**is**	ONTO *dist-target*
H312	*dist-target*	**is**	*processors-name*
		or	* *processors-name*
		or	*

The full syntax is given here for completeness. However, some of the forms are discussed only in Section 4. These "interprocedural" forms are:

- The last two options of rule H309 (containing the * form)
- The last two options of rule H312 (containing the * form)

Constraint: An *object-name* mentioned as a *distributee* must be a simple name and not a subobject designator or a *component-name*.

Constraint: An *object-name* mentioned as a *distributee* may not appear as an *alignee*.

Constraint: An *object-name* mentioned as a *distributee* may not have the POINTER attribute.

Constraint: An *object-name* mentioned as a *distributee* may not have the TARGET attribute.

Constraint: If the *distributee* is scalar, the *dist-format-list* (and its surrounding parentheses) must not appear. In this case, the statement form of the directive is allowed only if a *dist-format-clause* of "*" is present.

Constraint: If a *dist-format-list* is specified, its length must equal the rank of each *distributee* to which it applies.

Constraint: If both a *dist-format-list* and a *dist-target* appear, the number of elements of the *dist-format-list* that are not "*" must equal the rank of the specified processor arrangement.

Constraint: If a *dist-target* appears but not a *dist-format-list*, the rank of each *distributee* must equal the rank of the specified processor arrangement.

Constraint: If either the *dist-format-clause* or the *dist-target* in a DISTRIBUTE directive begins with "*" then every *distributee* must be a dummy argument.

Constraint: Any *scalar-int-expr* appearing in a *dist-format* of a DISTRIBUTE directive must be a *specification-expr*.

Advice to users. Some of the above constraints are relaxed under the approved extensions (see Section 8): mapping of derived type components (relaxes constraint 1), and mapping of pointers and targets (relaxes constraints 3, 4, and 9). (*End of advice to users.*) ⇓

Note that the possibility of a DISTRIBUTE directive of the form

!HPF$ DISTRIBUTE *dist-attribute-stuff* :: *distributee-list*

is covered by syntax rule H301 for a *combined-directive*.

Examples:

```
!HPF$ DISTRIBUTE D1(BLOCK)
!HPF$ DISTRIBUTE (BLOCK,*,BLOCK) ONTO SQUARE:: D2,D3,D4
```

The meanings of the alternatives for *dist-format* are given below.

Define the ceiling division function CD(J,K) = (J+K-1)/K (using Fortran integer arithmetic with truncation toward zero.)

Define the ceiling remainder function CR(J,K) = J-K*CD(J,K).

The dimensions of a processor arrangement appearing as a *dist-target* are said to *correspond* in left-to-right order with those dimensions of a *distributee* for which the corresponding *dist-format* is not *. In the example above, processor arrangement SQUARE must be two-dimensional; its first dimension corresponds to the first dimensions of D2, D3, and D4 and its second dimension corresponds to the third dimensions of D2, D3, and D4.

Let d be the size of a *distributee* in a certain dimension and let p be the size of the processor arrangement in the corresponding dimension. For simplicity, assume all dimensions have a lower bound of 1. Then BLOCK(m) means that a *distributee* position whose index along that dimension is j is mapped to an abstract processor whose index along the corresponding dimension of the processor arrangement is CD(j,m) (note that $m \times p \geq d$ must be true), and is position number m+CR(j,m) among positions mapped to that abstract processor. The first *distributee* position in abstract processor k along that axis is position number 1+m*(k-1).

The block size m must be a positive integer.

`BLOCK` by definition means the same as `BLOCK(CD(`d`,`p`))`.

`CYCLIC(`m`)` means that a *distributee* position whose index along that dimension is j is mapped to an abstract processor whose index along the corresponding dimension of the processor arrangement is `1+MODULO(CD(`j`,`m`)-1,`p`)`. The first *distributee* position in abstract processor k along that axis is position number `1+`m`*(`k`-1)`.

The block size m must be a positive integer.

`CYCLIC` by definition means the same as `CYCLIC(1)`.

`CYCLIC(`m`)` and `BLOCK(`m`)` imply the same distribution when $m \times p \geq d$, but `BLOCK(`m`)` additionally asserts that the distribution will not wrap around in a cyclic manner, which a compiler cannot determine at compile time if m is not constant. Note that `CYCLIC` and `BLOCK` (without argument expressions) do not imply the same distribution unless $p \geq d$, a degenerate case in which the block size is 1 and the distribution does not wrap around.

Suppose that we have 16 abstract processors and an array of length 100:

```
!HPF$ PROCESSORS SEDECIM(16)
      REAL CENTURY(100)
```

Distributing the array `BLOCK` (which in this case would mean the same as `BLOCK(7)`):

```
!HPF$ DISTRIBUTE CENTURY(BLOCK) ONTO SEDECIM
```

results in this mapping of array elements onto abstract processors:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	8	15	22	29	36	43	50	57	64	71	78	85	92	99	
2	9	16	23	30	37	44	51	58	65	72	79	86	93	100	
3	10	17	24	31	38	45	52	59	66	73	80	87	94		
4	11	18	25	32	39	46	53	60	67	74	81	88	95		
5	12	19	26	33	40	47	54	61	68	75	82	89	96		
6	13	20	27	34	41	48	55	62	69	76	83	90	97		
7	14	21	28	35	42	49	56	63	70	77	84	91	98		

Distributing the array `BLOCK(8)`:

```
!HPF$ DISTRIBUTE CENTURY(BLOCK(8)) ONTO SEDECIM
```

results in this mapping of array elements onto abstract processors:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	9	17	25	33	41	49	57	65	73	81	89	97			
2	10	18	26	34	42	50	58	66	74	82	90	98			
3	11	19	27	35	43	51	59	67	75	83	91	99			
4	12	20	28	36	44	52	60	68	76	84	92	100			
5	13	21	29	37	45	53	61	69	77	85	93				
6	14	22	30	38	46	54	62	70	78	86	94				
7	15	23	31	39	47	55	63	71	79	87	95				
8	16	24	32	40	48	56	64	72	80	88	96				

Distributing the array `BLOCK(6)` is not HPF-conforming because $6 \times 16 < 100$.

Distributing the array `CYCLIC` (which means exactly the same as `CYCLIC(1)`):

```
!HPF$ DISTRIBUTE CENTURY(CYCLIC) ONTO SEDECIM
```

results in this mapping of array elements onto abstract processors:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
97	98	99	100												

Distributing the array `CYCLIC(3)`:

```
!HPF$ DISTRIBUTE CENTURY(CYCLIC(3)) ONTO SEDECIM
```

results in this mapping of array elements onto abstract processors:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	4	7	10	13	16	19	22	25	28	31	34	37	40	43	46
2	5	8	11	14	17	20	23	26	29	32	35	38	41	44	47
3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48
49	52	55	58	61	64	67	70	73	76	79	82	85	88	91	94
50	53	56	59	62	65	68	71	74	77	80	83	86	89	92	95
51	54	57	60	63	66	69	72	75	78	81	84	87	90	93	96
97	100														
98															
99															

Note that it is perfectly permissible for an array to be distributed so that some processors have no elements. Indeed, an array may be "distributed" so that all elements reside on one processor. For example,

```
!HPF$ DISTRIBUTE CENTURY(BLOCK(256)) ONTO SEDECIM
```

results in having only one non-empty block—a partially-filled one at that, having only 100 elements—on processor 1, with processors 2 through 16 having no elements of the array.

The statement form of a `DISTRIBUTE` directive may be considered an abbreviation for an attributed form that happens to mention only one *distributee*; for example,

`!HPF$ DISTRIBUTE` *distributee* `(` *dist-format-list* `) ONTO` *dist-target* is equivalent to

```
!HPF$ DISTRIBUTE ( dist-format-list ) ONTO dist-target :: distributee
```

Note that, to prevent syntactic ambiguity, the *dist-format-clause* must be present in the statement form, so in general the statement form of the directive may not be used to specify the mapping of scalars.

If the *dist-format-clause* is omitted from the attributed form, then the language processor may make an arbitrary choice of distribution formats for each template or array. So the directive

```
!HPF$ DISTRIBUTE ONTO P :: D1,D2,D3
```

means the same as

```
!HPF$ DISTRIBUTE ONTO P :: D1
!HPF$ DISTRIBUTE ONTO P :: D2
!HPF$ DISTRIBUTE ONTO P :: D3
```

to which a compiler, perhaps taking into account patterns of use of `D1`, `D2`, and `D3` within the code, might choose to supply three distinct distributions such as, for example,

```
!HPF$ DISTRIBUTE D1(BLOCK, BLOCK) ONTO P
!HPF$ DISTRIBUTE D2(CYCLIC, BLOCK) ONTO P
!HPF$ DISTRIBUTE D3(BLOCK(43),CYCLIC) ONTO P
```

Then again, the compiler might happen to choose the same distribution for all three arrays.

In either the statement form or the attributed form, if the `ONTO` clause is present, it specifies the processor arrangement that is the target of the distribution. If the `ONTO` clause is omitted, then a implementation-dependent processor arrangement is chosen arbitrarily for each *distributee*. So, for example,

```
      REAL, DIMENSION(1000) :: ARTHUR, ARNOLD, LINUS, LUCY
!HPF$ PROCESSORS EXCALIBUR(32)
!HPF$ DISTRIBUTE (BLOCK) ONTO EXCALIBUR :: ARTHUR, ARNOLD
!HPF$ DISTRIBUTE (BLOCK) :: LINUS, LUCY
```

causes the arrays `ARTHUR` and `ARNOLD` to have the same mapping, so that corresponding elements reside in the same abstract processor, because they are the same size and distributed in the same way (`BLOCK`) onto the same processor arrangement (`EXCALIBUR`). However, `LUCY` and `LINUS` do not necessarily have the same mapping because they might, depending on the implementation, be distributed onto differently chosen processor arrangements; so corresponding elements of `LUCY` and `LINUS` might not reside on the same abstract processor. (The `ALIGN` directive provides a way to ensure that two arrays have the same mapping without having to specify an explicit processor arrangement.)

In a given environment, for some distributions, there may be no appropriate processor arrangement.

3.4 The ALIGN Directive

The `ALIGN` directive is used to specify that certain data objects are to be mapped in the same way as certain other data objects. Operations between aligned data objects are likely to be more efficient than operations between data objects that are not known to be aligned (because two objects that are aligned are intended to be mapped to the same abstract processor). The `ALIGN` directive is designed to make it particularly easy to specify explicit mappings for all the elements of an array at once. While objects can be aligned in some cases through careful use of matching `DISTRIBUTE` directives, `ALIGN` is more general and frequently more convenient.

The `ALIGN` directive may appear only in the *specification-part* of a scoping unit and can contain only a *specification-expr* as a *subscript* or in a *subscript-triplet*.

The syntax of `ALIGN` is as follows:

H313	*align-directive*	**is**	`ALIGN` *alignee align-directive-stuff*
H314	*align-directive-stuff*	**is**	(*align-source-list*) *align-with-clause*
H315	*align-attribute-stuff*	**is**	[(*align-source-list*)] *align-with-clause*
H316	*alignee*	**is**	*object-name*
H317	*align-source*	**is**	:
		or	*
		or	*align-dummy*
H318	*align-dummy*	**is**	*scalar-int-variable*

Constraint: An *object-name* mentioned as an *alignee* must be a simple name and not a subobject designator or a *component-name*.

Constraint: An *object-name* mentioned as an *alignee* may not appear as a *distributee*.

Constraint: An *object-name* mentioned as an *alignee* may not have the `POINTER` attribute.

Constraint: An *object-name* mentioned as an *alignee* may not have the `TARGET` attribute.

Constraint: If the *alignee* is scalar, the *align-source-list* (and its surrounding parentheses) must not appear. In this case the statement form of the directive is not allowed.

Constraint: If the *align-source-list* is present, its length must equal the rank of each *alignee* to which it applies.

Constraint: An *align-dummy* must be a named variable.

Constraint: An object may not have both the `INHERIT` attribute and the `ALIGN` attribute.

Advice to users. Some of the above constraints are relaxed under the approved extensions (see Section 8): mapping of derived type components (relaxes constraint 1) and mapping of pointers and targets (relaxes constraints 3 and 4). (*End of advice to users.*) ⇓

Note that the possibility of an `ALIGN` directive of the form

```
!HPF$ ALIGN align-attribute-stuff :: alignee-list
```

is covered by syntax rule H301 for a *combined-directive*.

The statement form of an `ALIGN` directive may be considered an abbreviation of an attributed form that happens to mention only one *alignee*:

```
!HPF$ ALIGN alignee ( align-source-list ) WITH align-spec
```

is equivalent to

```
!HPF$ ALIGN ( align-source-list ) WITH align-spec :: alignee
```

If the *align-source-list* is omitted from the attributed form and the *alignee*s are not scalar, the *align-source-list* is assumed to consist of a parenthesized list of ":" entries, equal in number to the rank of the *alignees*. Similarly, if the *align-subscript-list* is omitted from the *align-spec* in either form, it is assumed to consist of a parenthesized list of ":" entries, equal in number to the rank of the *align-target*. So the directive

```
!HPF$ ALIGN WITH B :: A1, A2, A3
```

means

```
!HPF$ ALIGN (:,:) WITH B(:,:) :: A1, A2, A3
```

which in turn means the same as

```
!HPF$ ALIGN A1(:,:) WITH B(:,:)
!HPF$ ALIGN A2(:,:) WITH B(:,:)
!HPF$ ALIGN A3(:,:) WITH B(:,:)
```

because an attributed-form directive that mentions more than one *alignee* is equivalent to a series of identical directives, one for each *alignee*; all *alignee*s must have the same rank. With this understanding, we will assume below, for the sake of simplifying the description, that an `ALIGN` directive has a single *alignee*.

Each *align-source* corresponds to one axis of the *alignee*, and is specified as either ":" or "*" or a dummy variable:

- If it is ":", then positions along that axis will be spread out across the matching axis of the *align-spec* (see below).

- If it is "*", then that axis is *collapsed*: positions along that axis make no difference in determining the corresponding position within the *align-target*. (Replacing the "*" with a dummy variable name not used anywhere else in the directive would have the same effect; "*" is merely a convenience that saves the trouble of inventing a variable name and makes it clear that no dependence on that dimension is intended.)

- A dummy variable is considered to range over all valid index values for that dimension of the *alignee*.

The `WITH` clause of an `ALIGN` has the following syntax:

H319	*align-with-clause*	**is**	WITH *align-spec*
H320	*align-spec*	**is**	*align-target* [(*align-subscript-list*)]
		or	* *align-target* [(*align-subscript-list*)]
H321	*align-target*	**is**	*object-name*
		or	*template-name*
H322	*align-subscript*	**is**	*int-expr*
		or	*align-subscript-use*
		or	*subscript-triplet*
		or	*
H323	*align-subscript-use*	**is**	[[*int-level-two-expr*] *add-op*] *align-add-operand*
		or	*align-subscript-use add-op int-add-operand*
H324	*align-add-operand*	**is**	[*int-add-operand* *] *align-primary*
		or	*align-add-operand* * *int-mult-operand*
H325	*align-primary*	**is**	*align-dummy*
		or	(*align-subscript-use*)
H326	*int-add-operand*	**is**	*add-operand*
H327	*int-mult-operand*	**is**	*mult-operand*
H328	*int-level-two-expr*	**is**	*level*-2-*expr*

The full syntax is given here for completeness. However, some of the forms are discussed only in Section 4. These "interprocedural" forms are those using the second option of rule H320 (containing the * form).

Constraint: An *object-name* mentioned as an *align-target* must be a simple name and not a subobject designator or a *component-name*.

Constraint: An *align-target* may not have the OPTIONAL attribute.

Constraint: If the *align-spec* in an ALIGN directive begins with "*" then every *alignee* must be a dummy argument.

Constraint: In an *align-directive* any *int-expr*, *int-level-two-expr*, *int-add-operand* or *int-mult-operand* must be a specification expression.

Constraint: Any *subscript* or *stride* in a *subscript-triplet* that is an *align-subscript* in an *align-directive* must be a specification expression.

Constraint: Each *align-dummy* may appear at most once in an *align-subscript-list*.

Constraint: An *align-subscript-use* expression may contain at most one occurrence of an *align-dummy*.

Constraint: A *scalar-int-variable* that is used as an *align-dummy* may not appear anywhere in the *align-spec* except where explicitly permitted to appear by virtue of the grammar shown above. Paraphrased, one may construct an *align-subscript-use* only by starting with an *align-dummy* and then doing additive and multiplicative things to it with integer specification expressions that contain no *align-dummy*.

Constraint: A *subscript* within an *align-subscript* may not contain occurrences of any *align-dummy*.

Constraint: An *int-add-operand*, *int-mult-operand*, or *int-level-two-expr* must be of type integer.

⇓ *Advice to users.* Some of the above constraints are relaxed under the approved extensions (see Section 8): mapping of derived type components (relaxes constraint 1), mapping of pointers (relaxes constraint 3) and remapping of data objects (relaxes constraints 4 and 5). (*End of advice to users.*)

The syntax rules for an *align-subscript-use* take account of operator precedence issues, but the basic idea is simple: an *align-subscript-use* is intended to be a linear (more precisely: affine) function of a single occurrence of an *align-dummy*.

For example, the following *align-subscript-use* expressions are valid, assuming that each of J, K, and M is an *align-dummy* and N is not an *align-dummy*:

```
J    J+1      3-K      2*M     N*M         100-3*M
-J   +J       -K+3     M+2**3  M+N         -(4*7+IOR(6,9))*K-(13-5/3)
M*2  N*(M-N)  2*(J+1)  5-K+3   10000-M*3   2*(3*(K-1)+13)-100
```

The following expressions are not valid *align-subscript-use* expressions:

```
J+J  J-J       3*K-2*K  M*(N-M)  2*J-3*J+J  2*(3*(K-1)+13)-K
J*J  J+K       3/K      2**M     M*K        K-3*M
K-J  IOR(J,1)  -K/3     M*(2+M)  M*(M-N)    2**(2*J-3*J+J)
```

The *align-spec* must contain exactly as many *subscript-triplets* as the number of colons (":") appearing in the *align-source-list*. These are matched up in corresponding left-to-right order, ignoring, for this purpose, any *align-source* that is not a colon and any *align-subscript* that is not a *subscript-triplet*. Consider a dimension of the *alignee* for which a colon appears as an *align-source* and let the lower and upper bounds of that dimension be *LA* and *UA*. Let the corresponding subscript triplet be *LT*:*UT*:*ST* or its equivalent. Then the colon could be replaced by a new, as-yet-unused dummy variable, say J, and the subscript triplet by the expression (J-*LA*)**ST*+*LT* without affecting the mapping specified by the directive. However, the colon form additionaly requires that the axes must conform, which means that

$$\max(0, UA - LA + 1) \;=\; \max(0, \lceil (UT - LT + 1)/ST \rceil)$$

must be true. (This is entirely analogous to the treatment of array assignment.)

To simplify the remainder of the discussion, we assume that every colon in the *align-source-list* has been replaced by new dummy variables in exactly the fashion just described, and that every "*" in the *align-source-list* has likewise been replaced by an otherwise unused dummy variable. For example,

```
!HPF$ ALIGN A(:,*,K,:,:,*) WITH B(31:,:,K+3,20:100:3)
```

may be transformed into its equivalent

```
!HPF$ ALIGN A(I,J,K,L,M,N) WITH B(I-LBOUND(A,1)+31,       &
!HPF$            L-LBOUND(A,4)+LBOUND(B,2),K+3,(M-LBOUND(A,5))*3+20)
```

with the attached requirements

```
SIZE(A,1) .EQ. UBOUND(B,1)-30
  SIZE(A,4) .EQ. SIZE(B,2)
 SIZE(A,5) .EQ. (100-20+3)/3
```

Thus we need consider further only the case where every *align-source* is a dummy variable and no *align-subscript* is a *subscript-triplet*.

Each dummy variable is considered to range over all valid index values for the corresponding dimension of the *alignee*. Every combination of possible values for the index variables selects an element of the *alignee*. The *align-spec* indicates a corresponding element (or section) of the *align-target* with which that element of the *alignee* should be aligned; this indication may be a function of the index values, but the nature of this function is syntactically restricted (as discussed above) to linear (precisely: affine) functions in order to limit the complexity of the implementation. Each *align-dummy* variable may appear at most once in the *align-spec* and only in certain rigidly prescribed contexts. The result is that each *align-subscript* expression may contain at most one *align-dummy* variable and the expression is constrained to be a linear function of that variable. (Therefore skew alignments are not possible.)

An asterisk "*" as an *align-subscript* indicates a replicated representation. Each element of the *alignee* is aligned with every position along that axis of the *align-target*.

Rationale. It may seem strange to use "*" to mean both collapsing and replication; the rationale is that "*" always stands conceptually for a dummy variable that appears nowhere else in the statement and ranges over the set of indices for the indicated dimension. Thus, for example,

```
!HPF$ ALIGN A(:) WITH D(:,*)
```

means that a copy of `A` is aligned with every column of `D`, because it is conceptually equivalent to

for every legitimate index j, align `A(:)` *with* `D(:,`*j*`)`

just as

```
!HPF$ ALIGN A(:,*) WITH D(:)
```

is conceptually equivalent to

for every legitimate index j, align `A(:,`*j*`)` *with* `D(:)`

Note, however, that while HPF syntax allows

```
!HPF$ ALIGN A(:,*) WITH D(:)
```

to be written in the alternate form

```
!HPF$ ALIGN A(:,J) WITH D(:)
```

it does *not* allow

```
!HPF$ ALIGN A(:) WITH D(:,*)
```

to be written in the alternate form

```
!HPF$ ALIGN A(:) WITH D(:,J)
```

because that has another meaning (only a variable appearing in the *align-source-list* following the *alignee* is understood to be an *align-dummy*, so the current value of the variable J is used, thus aligning A with a single column of D).

Replication allows an optimizing compiler to arrange to read whichever copy is closest. (Of course, when a replicated data object is written, all copies must be updated, not just one copy. Replicated representations are very useful for small lookup tables, where it is much faster to have a copy in each physical processor but without giving it an extra dimension that is logically unnecessary to the algorithm. (*End of rationale.*)

By applying the transformations given above, all cases of an *align-subscript* may be conceptually reduced to either an *int-expr* (not involving an *align-dummy*) or an *align-subscript-use*, and the *align-source-list* may be reduced to a list of index variables with no "*" or ":". An *align-subscript-list* may then be evaluated for any specific combination of values for the *align-dummy* variables simply by evaluating each *align-subscript* as an expression. The resulting subscript values must be legitimate subscripts for the *align-target*. (This implies that the *alignee* is not allowed to "wrap around" or "extend past the edges" of an *align-target*.) The selected element of the *alignee* is then considered to be aligned with the indicated element of the *align-target*; more precisely, the selected element of the *alignee* is considered to be ultimately aligned with the same object with which the indicated element of the *align-target* is currently ultimately aligned (possibly itself).

More examples of ALIGN directives:

```
      INTEGER D1(N)
      LOGICAL D2(N,N)
      REAL, DIMENSION(N,N):: X,A,B,C,AR1,AR2A,P,Q,R,S
!HPF$ ALIGN X(:,*) WITH D1(:)
!HPF$ ALIGN (:,*) WITH D1:: A,B,C,AR1,AR2A
!HPF$ ALIGN WITH D2:: P,Q,R,S
```

Note that, in a *alignee-list*, the alignees must all have the same rank but need not all have the same shape; the extents need match only for dimensions that correspond to colons in the *align-source-list*. This turns out to be an extremely important convenience; one of the most common cases in current practice is aligning arrays that match in distributed ("parallel") dimensions but may differ in collapsed ("on-processor") dimensions:

```
      REAL A(3,N), B(4,N), C(43,N), Q(N)
!HPF$ DISTRIBUTE Q(BLOCK)
!HPF$ ALIGN (*,:) WITH Q:: A,B,C
```

Here there are processors (perhaps `N` of them) and arrays of different sizes (3, 4, 43) within each processor are required. As far as HPF is concerned, the numbers 3, 4, and 43 may be different, because those axes will be collapsed. Thus array elements with indices differing only along that axis will all be aligned with the same element of `Q` (and thus be specified as residing in the same processor).

In the following examples, each directive in a group means the same thing, assuming that corresponding axis upper and lower bounds match:

```
!Second axis of X is collapsed
!HPF$ ALIGN X(:,*) WITH D1(:)
!HPF$ ALIGN X(J,*) WITH D1(J)
!HPF$ ALIGN X(J,K) WITH D1(J)

!Replicated representation along second axis of D3
!HPF$ ALIGN X(:,:) WITH D3(:,*,:)
!HPF$ ALIGN X(J,K) WITH D3(J,*,K)

!Transposing two axes
!HPF$ ALIGN X(J,K) WITH D2(K,J)
!HPF$ ALIGN X(J,:) WITH D2(:,J)
!HPF$ ALIGN X(:,K) WITH D2(K,:)
!But there isn't any way to get rid of *both* index variables;
! the subscript-triplet syntax alone cannot express transposition.

!Reversing both axes
!HPF$ ALIGN X(J,K) WITH D2(M-J+1,N-K+1)
!HPF$ ALIGN X(:,:) WITH D2(M:1:-1,N:1:-1)

!Simple case
!HPF$ ALIGN X(J,K) WITH D2(J,K)
!HPF$ ALIGN X(:,:) WITH D2(:,:)
!HPF$ ALIGN (J,K) WITH D2(J,K):: X
!HPF$ ALIGN (:,:) WITH D2(:,:):: X
!HPF$ ALIGN WITH D2:: X
```

3.5 Allocatable Arrays and Pointers

A variable with the `ALLOCATABLE` attribute may appear as an *alignee* in an `ALIGN` directive or as a *distributee* in a `DISTRIBUTE` directive. Such directives do not take effect immediately, however; they take effect each time the array is allocated by an `ALLOCATE` statement, rather than on entry to the scoping unit. The values of all specification expressions in such a directive are determined once on entry to the scoping unit and may be used multiple times (or not at all). For example:

```
      SUBROUTINE MILLARD_FILLMORE(N,M)
      REAL, ALLOCATABLE, DIMENSION(:) :: A, B
!HPF$ ALIGN B(I) WITH A(I+N)
!HPF$ DISTRIBUTE A(BLOCK(M*2))
      N = 43
      M = 91
      ALLOCATE(A(27))
      ALLOCATE(B(13))
      ...
```

The values of the expressions N and M*2 on entry to the subprogram are conceptually retained by the ALIGN and DISTRIBUTE directives for later use at allocation time. When the array A is allocated, it is distributed with a block size equal to the retained value of M*2, not the value 182. When the array B is allocated, it is aligned relative to A according to the retained value of N, not its new value 43.

Note that it would have been incorrect in the MILLARD_FILLMORE example to perform the two ALLOCATE statements in the opposite order. In general, when an object X is created it may be aligned to another object Y only if Y has already been created or allocated. The following example illustrates several related cases.

```
      SUBROUTINE WARREN_HARDING(P,Q)
      REAL P(:)
      REAL Q(:)
      REAL R(SIZE(Q))
      REAL, ALLOCATABLE :: S(:),T(:)
!HPF$ ALIGN P(I) WITH T(I)                      !Nonconforming
!HPF$ ALIGN Q(I) WITH *T(I)                     !Nonconforming
!HPF$ ALIGN R(I) WITH T(I)                      !Nonconforming
!HPF$ ALIGN S(I) WITH T(I)
      ALLOCATE(S(SIZE(Q)))                      !Nonconforming
      ALLOCATE(T(SIZE(Q)))
```

Three ALIGN directives are not HPF-conforming because the array T has not yet been allocated at the time that the various alignments must take place. The four cases differ slightly in their details. The arrays P and Q already exist on entry to the subroutine, but because T is not yet allocated, one cannot correctly prescribe the alignment of P or describe the alignment of Q relative to T. (See Section 4 for a discussion of prescriptive and descriptive directives.) The array R is created on subroutine entry and its size can correctly depend on the SIZE of Q, but the alignment of R cannot be specified in terms of the alignment of T any more than its size can be specified in terms of the size of T. It *is* permitted to have an alignment directive for S in terms of T, because the alignment action does not take place until S is allocated; however, the first ALLOCATE statement is nonconforming because S needs to be aligned but at that point in time T is still unallocated.

When an array is allocated, it will be aligned to an existing object or template if there is an explicit ALIGN directive for the allocatable variable. If there is no explicit ALIGN directive, then the array will be ultimately aligned with itself. It is forbidden for any other object to be ultimately aligned to an array at the time the array becomes undefined by reason of deallocation. All this applies regardless of whether the name originally used in the ALLOCATE statement when the array was created had the ALLOCATABLE attribute or the POINTER attribute.

Pointers cannot be explicitly mapped in HPF and thus can only be associated with objects which are not explicitly mapped. When used for allocation, the compiler may choose any arbitrary mapping for data allocated through the pointer. Explicit mapping of pointers is allowed under the approved extensions (see section 8.8). Also, the relationship of pointers and sequence attributes is described in section 3.8.

⇓

3.6 The PROCESSORS Directive

The `PROCESSORS` directive declares one or more rectilinear processor arrangements, specifying for each one its name, its rank (number of dimensions), and the extent in each dimension. It may appear only in the *specification-part* of a scoping unit. Every dimension of a processor arrangement must have nonzero extent; therefore a processor arrangement cannot be empty.

In the language of F95:14.1.2 in the Fortran standard, processor arrangements are local entities of class (1); therefore a processor arrangement may not have the same name as a variable, named constant, internal procedure, etc., in the same scoping unit. Names of processor arrangements obey the same rules for host and use association as other names in the long list in F95:12.1.2.2.1 in the Fortran standard.

A processor arrangement declared in a module has the default accessibility of the module.

> *Rationale.* Because the name of a processor arrangement is not a first-class entity in HPF, but must appear only in directives, it cannot appear in an *access-stmt* (`PRIVATE` or `PUBLIC`). If directives ever become full-fledged Fortran statements rather than structured comments, then it would be appropriate to allow the accessibility of a processor arrangement to be controlled by listing its name in an *access-stmt*. (*End of rationale.*)

If two processor arrangements have the same shape, then corresponding elements of the two arrangements are understood to refer to the same abstract processor. (It is anticipated that implementation-dependent directives provided by some HPF implementations could overrule the default correspondence of processor arrangements that have the same shape.)

If directives collectively specify that two objects be mapped to the same abstract processor at a given instant during the program execution, the intent is that the two objects be mapped to the same physical processor at that instant.

The intrinsic functions `NUMBER_OF_PROCESSORS` and `PROCESSORS_SHAPE` may be used to inquire about the total number of actual physical processors used to execute the program. This information may then be used to calculate appropriate sizes for the declared abstract processor arrangements.

H329	*processors-directive*	**is**	`PROCESSORS` *processors-decl-list*
H330	*processors-decl*	**is**	*processors-name* [(*explicit-shape-spec-list*)]

Examples:

```
!HPF$ PROCESSORS P(N)
!HPF$ PROCESSORS Q(NUMBER_OF_PROCESSORS()),       &
!HPF$            R(8,NUMBER_OF_PROCESSORS()/8)
!HPF$ PROCESSORS BIZARRO(1972:1997,-20:17)
!HPF$ PROCESSORS SCALARPROC
```

If no shape is specified, then the declared processor arrangement is conceptually scalar.

Rationale. A scalar processor arrangement may be useful as a way of indicating that certain scalar data should be kept together but need not interact strongly with distributed data. Depending on the implementation architecture, data distributed onto such a processor arrangement may reside in a single "control" or "host" processor (if the machine has one), or may reside in an arbitrarily chosen processor, or may be replicated over all processors. For target architectures that have a set of computational processors and a separate scalar host computer, a natural implementation is to map every scalar processor arrangement onto the host processor. For target architectures that have a set of computational processors but no separate scalar "host" computer, data mapped to a scalar processor arrangement might be mapped to some arbitrarily chosen computational processor or replicated onto all computational processors. (*End of rationale.*)

An HPF compiler is required to accept any PROCESSORS declaration in which the product of the extents of each declared processor arrangement is equal to the number of physical processors that would be returned by the call NUMBER_OF_PROCESSORS(). It must also accept all declarations of scalar PROCESSOR arrangements. Other cases may be handled as well, depending on the implementation.

For compatibility with the Fortran attribute syntax, an optional "::" may be inserted. The shape may also be specified with the DIMENSION attribute:

```
!HPF$ PROCESSORS :: RUBIK(3,3,3)
!HPF$ PROCESSORS, DIMENSION(3,3,3) :: RUBIK
```

As in Fortran, an *explicit-shape-spec-list* in a *processors-decl* will override an explicit DIMENSION attribute:

```
!HPF$ PROCESSORS, DIMENSION(3,3,3) ::      &
!HPF$            RUBIK, RUBIKS_REVENGE(4,4,4), SOMA
```

Here RUBIKS_REVENGE is $4 \times 4 \times 4$ while RUBIK and SOMA are each $3 \times 3 \times 3$. (By the rules enunciated above, however, such a statement may not be completely portable because no HPF language processor is required to handle shapes of total sizes 27 and 64 simultaneously.)

Returning from a subprogram causes all processor arrangements declared local to that subprogram to become undefined. It is not HPF-conforming for any array or template to be distributed onto a processor arrangement at the time the processor arrangement becomes undefined unless at least one of two conditions holds:

- The array or template itself becomes undefined at the same time by virtue of returning from the subprogram.
- Whenever the subprogram is called, the processor arrangement is always locally defined in the same way, with identical lower bounds and identical upper bounds.

Rationale. Note that this second condition is slightly less stringent than requiring all expressions to be constant. This allows calls to NUMBER_OF_PROCESSORS or PROCESSORS_SHAPE to appear without violating the condition. (*End of rationale.*)

Variables in COMMON or having the SAVE attribute may be mapped to a locally declared processor arrangement, but because the first condition cannot hold for such variables (they don't become undefined), the second condition must be observed. This allows COMMON variables to work properly through the customary strategy of putting identical declarations in each scoping unit that needs to use them, while allowing the processor arrangements to which they may be mapped to depend on the value returned by NUMBER_OF_PROCESSORS. (See section 3.8 for further information on mapping common variables.)

Advice to implementors. It may be desirable to have a way for the user to specify at compile time the number of physical processors on which the program is to be executed. This might be specified either by an implementation-dependent directive, for example, or through the programming environment (for example, as a UNIX command-line argument). Such facilities are beyond the scope of the HPF specification, but as food for thought we offer the following illustrative hypothetical examples:

```
!Declaration for multiprocessor by ABC Corporation
!ABC$ PHYSICAL PROCESSORS(8)
!Declaration for mpp by XYZ Incorporated
!XYZ$ PHYSICAL PROCESSORS(65536)
!Declaration for hypercube machine by PDQ Limited
!PDQ$ PHYSICAL PROCESSORS(2,2,2,2,2,2,2,2,2,2)
!Declaration for two-dimensional grid machine by TLA GmbH
!TLA$ PHYSICAL PROCESSORS(128,64)
!One of the preceding might affect the following:
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
```

It may furthermore be desirable to have a way for the user to specify the precise mapping of the processor arrangement declared in a PROCESSORS statement to the physical processors of the executing hardware. Again, this might be specified either by a implementation-dependent directive or through the programming environment (for example, as a UNIX command-line argument); such facilities are beyond the scope of the HPF specification, but as food for thought we offer the following illustrative hypothetical example:

```
!PDQ$ PHYSICAL PROCESSORS(2,2,2,2,2,2,2,2,2,2,2,2,2)
!HPF$ PROCESSORS G(8,64,16)
!PDQ$ MACHINE LAYOUT G(:GRAY(0:2),:GRAY(6:11),:BINARY(3:5,12))
```

This might specify that the first dimension of G should use hypercube axes 0, 1, 2 with a Gray-code ordering; the second dimension should use hypercube axes 6 through 11 with a Gray-code ordering; and the third dimension should use hypercube axes 3, 4, 5, and 12 with a binary ordering. (*End of advice to implementors.*)

3.7 The TEMPLATE Directive

The TEMPLATE directive declares one or more templates, specifying for each the name, the rank (number of dimensions), and the extent in each dimension. It must appear in the *specification-part* of a scoping unit.

In the language of F95:14.1.2 in the Fortran standard, templates are local entities of class (1); therefore a template may not have the same name as a variable, named constant, internal procedure, etc., in the same scoping unit. Template names obey the rules for host and use association as other names in the list in F95:12.1.2.2.1 in the Fortran standard.

A template declared in a module has the default accessibility of the module.

> *Rationale.* Because the name of a template is not a first-class entity in HPF, but must appear only in directives, it cannot appear in an *access-stmt* (`PRIVATE` or `PUBLIC`). If directives ever become full-fledged Fortran statements rather than structured comments, then it would be appropriate to allow the accessibility of a template to be controlled by listing its name in an *access-stmt.* (*End of rationale.*)

A template is simply an abstract space of indexed positions; it can be considered as an "array of nothings" (as compared to an "array of integers," say). A template may be used as an abstract *align-target* that may then be distributed.

H331	*template-directive*	**is**	`TEMPLATE` *template-decl-list*
H332	*template-decl*	**is**	*template-name* [(*explicit-shape-spec-list*)]

Examples:

```
!HPF$ TEMPLATE A(N)
!HPF$ TEMPLATE B(N,N), C(N,2*N)
!HPF$ TEMPLATE DOPEY(100,100),SNEEZY(24),GRUMPY(17,3,5)
```

If the "`::`" syntax is used, then the declared templates may optionally be distributed in the same *combined-directive.* In this case all templates declared by the directive must have the same rank so that the `DISTRIBUTE` attribute will be meaningful. The `DIMENSION` attribute may also be used.

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,*) ::    &
!HPF$                                WHINEY(64,64),MOPEY(128,128)
!HPF$ TEMPLATE, DIMENSION(91,91) :: BORED,WHEEZY,PERKY
```

Templates are useful in the particular situation where one must align several arrays relative to one another but there is no need to declare a single array that spans the entire index space of interest. For example, one might want four $N \times N$ arrays aligned to the four corners of a template of size $(N+1) \times (N+1)$:

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) :: EARTH(N+1,N+1)
      REAL, DIMENSION(N,N) :: NW, NE, SW, SE
!HPF$ ALIGN NW(I,J) WITH EARTH( I , J )
!HPF$ ALIGN NE(I,J) WITH EARTH( I ,J+1)
!HPF$ ALIGN SW(I,J) WITH EARTH(I+1, J )
!HPF$ ALIGN SE(I,J) WITH EARTH(I+1,J+1)
```

Templates may also be useful in making assertions about the mapping of dummy arguments (see Section 4).

Unlike arrays, templates cannot be in `COMMON`. So two templates declared in different scoping units will always be distinct, even if they are given the same name. The only way for two program units to refer to the same template is to declare the template in a module that is then used by the two program units.

Templates are not passed through the subprogram argument interface. The template to which a dummy argument is aligned is always distinct from the template to which the actual argument is aligned, though it may be a copy (see section 4.4.2). On exit from a subprogram, an HPF implementation arranges that the actual argument is aligned with the same template with which it was aligned before the call.

Returning from a subprogram causes all templates declared local to that subprogram to become undefined. It is not HPF-conforming for any variable to be aligned to a template at the time the template becomes undefined unless at least one of two conditions holds:

- The variable itself becomes undefined at the same time by virtue of returning from the subprogram.
- Whenever the subprogram is called, the template is always locally defined in the same way, with identical lower bounds, identical upper bounds, and identical distribution information (if any) onto identically defined processor arrangements (see section 3.6).

> *Rationale.* Note that this second condition is slightly less stringent than requiring all expressions to be constant. This allows calls to `NUMBER_OF_PROCESSORS` or `PROCESSORS_SHAPE` to appear without violating the condition. (*End of rationale.*)

Variables in `COMMON` or having the `SAVE` attribute may be mapped to a locally declared template, but because the first condition cannot hold for such variable (they don't become undefined), the second condition must be observed.

3.8 Storage and Sequence Association

HPF allows the mapping of data objects across multiple processors in order to improve parallel performance. Fortran specifies relationships between the storage for data objects associated through `COMMON` and `EQUIVALENCE` statements, and the order of array elements during association at procedure boundaries between actual arguments and dummy arguments. Otherwise, the location of data is not constrained by the language.

`COMMON` and `EQUIVALENCE` statements constrain the alignment of different data items based on the underlying model of storage units and storage sequences:

> *Storage association is the association of two or more data objects that occurs when two or more storage sequences share or are aligned with one or more storage units.*
> — Fortran Standard (F95:14.6.3.1)

The model of storage association is a single linearly addressed memory, based on the traditional single address space, single memory unit architecture. This model can cause severe inefficiencies on architectures where storage for data objects is mapped.

Sequence association refers to the order of array elements that Fortran requires when an array expression or array element is associated with a dummy array argument:

> *The rank and shape of the actual argument need not agree with the rank and shape of the dummy argument, ...*
> — Fortran Standard (F95:12.4.1.4)

As with storage association, sequence association is a natural concept only in systems with a linearly addressed memory.

As an aid to porting FORTRAN 77 codes, HPF allows codes that rely on sequence and storage association to be valid in HPF. Some modification to existing FORTRAN 77 codes may nevertheless be necessary. This section explains the relationship between HPF data mapping and sequence and storage association.

3.8.1 Storage Association

3.8.1.1 Definitions

1. `COMMON` blocks are either *sequential* or *nonsequential*, as determined by either explicit directive or compiler default. A sequential `COMMON` block has a single common block storage sequence (F95:5.5.2.1).

2. An *aggregate variable group* is a collection of variables whose individual storage sequences are parts of a single storage sequence.

   Variables associated by `EQUIVALENCE` statements or by a combination of `EQUIVALENCE` and `COMMON` statements form an aggregate variable group. The variables of a sequential `COMMON` block form a single aggregate variable group.

3. The *size* of an aggregate variable group is the number of storage units in the group's storage sequence (F95:14.6.3.1).

4. Data objects are either *sequential* or *nonsequential*. A data object is *sequential* if and only if any of the following holds:

   (a) it appears in a sequential `COMMON` block;

   (b) it is a member of an aggregate variable group;

   (c) it is an assumed-size array;

   (d) its type is a sequence type;

   (e) it is a subobject of a sequential data object; or

   (f) it is declared to be sequential in an HPF `SEQUENCE` directive.

   A sequential object can be storage associated or sequence associated; nonsequential objects cannot.

5. A `COMMON` block contains a sequence of *components*. Each component is either an aggregate variable group, or a variable that is not a member of any aggregate variable group. A sequential `COMMON` block contains a single component. A nonsequential `COMMON` block may contain several components each of which may be a sequential variable, an aggregate variable group, or a nonsequential variable.

3.8.1.2 Examples of Definitions

```
!Example 1:
      COMMON /FOO/ A(100), B(100), C(100), D(100), E(100)
      DIMENSION X(100), Y(150), Z(200)
      EQUIVALENCE ( A(1), Z(1) )
```

```
!Four components: (A, B), C, D, E
!Sizes are: 200, 100, 100, 100

!Example 2:
      COMMON /FOO/ A(100), B(100), C(100), D(100), E(100)
      DIMENSION X(100), Y(150), Z(200)
      EQUIVALENCE ( A(51), X(1) ) ( B(100), Y(1) )
!Two components (A, B, C, D), E
!Sizes are: 400, 100

!Example 3:
      COMMON /FOO/ A(100), B(100), C(100), D(100), E(100)
      DIMENSION X(100), Y(150), Z(200)
!HPF$ SEQUENCE /FOO/
!The COMMON has one component, (A, B, C, D, E)
!Size is 500
```

The `COMMON` block `/FOO/` is nonsequential in Examples 1 and 2. Aggregate variable groups are shown as components in parentheses.

3.8.2 The SEQUENCE Directive

A `SEQUENCE` directive is defined to allow a user to declare explicitly that data objects or `COMMON` blocks are to be treated by the compiler as sequential. (`COMMON` blocks are by default nonsequential. Data objects are nonsequential unless Definition 4 of Section 3.8 applies.) Some implementations may supply an optional compilation environment where the `SEQUENCE` directive is applied by default. For completeness in such an environment, HPF defines a `NO SEQUENCE` directive to allow a user to establish that the usual nonsequential default should apply to a scoping unit or to selected data objects and `COMMON` blocks within the scoping unit.

H333	*sequence-directive*	**is**	`SEQUENCE` [[`::`] *association-name-list*]
		or	`NO SEQUENCE` [[`::`] *association-name-list*]
H334	*association-name*	**is**	*object-name*
		or	**/** [*common-block-name*] **/**

Constraint: An object name or `COMMON` block name may appear at most once in a *sequence-directive* within any scoping unit.

Constraint: Only one sequence directive with no *association-name-list* is permitted in the same scoping unit.

A sequential pointer can be associated only with sequential objects. A nonsequential pointer can be associated only with nonsequential objects.

3.8.2.1 Storage Association Rules

1. A *sequence-directive* with an empty *association-name-list* is treated as if it contained the names of all implicitly mapped objects and `COMMON` blocks in the scoping unit that

cannot otherwise be determined to be sequential or nonsequential by their language context.

2. A sequential object may not be explicitly mapped.

3. No explicit mapping may be given for a component of a derived type having the Fortran `SEQUENCE` attribute. Note that this rule is applicable only under the approved ⇓ extensions since components of derived types cannot be explicitly mapped in HPF.

4. If a `COMMON` block is nonsequential, then all of the following must hold:

   (a) Every occurrence of the `COMMON` block has exactly the same number of components with each corresponding component having a storage sequence of exactly the same size;

   (b) If a component is a nonsequential variable in *any* occurrence of the `COMMON` block, then it must be nonsequential with identical type, shape, and mapping attributes in *every* occurrence of the `COMMON` block; and

   (c) Every occurrence of the `COMMON` block must be nonsequential.

3.8.2.2 Storage Association Discussion

Advice to users. Under these rules, variables in a `COMMON` block can be mapped as long as the components of the `COMMON` block are the same in every scoping unit that declares the `COMMON` block.

Correct Fortran programs will not necessarily be correct without modification in HPF. The use of `EQUIVALENCE` with `COMMON` blocks can impact the mappability of data objects in subtle ways. To allow maximum optimization for performance, the HPF default for data objects is to consider them mappable. In order to get correct separate compilation for subprograms that use `COMMON` blocks with different aggregate variable groups in different scoping units, it will be necessary to insert the HPF `SEQUENCE` directive.

As a check-list for a user to determine the status of a data object or `COMMON` block, the following questions can be applied, in order:

- Does the object appear in some explicit language context which dictates that the object be sequential (e.g. `EQUIVALENCE`) or nonsequential?
- If not, does the object appear in an explicit mapping directive?
- If not, does the object or `COMMON` block name appear in the list of names on a `SEQUENCE` or `NO SEQUENCE` directive?
- If not, does the scoping unit contain a nameless `SEQUENCE` or `NO SEQUENCE`?
- If not, is the compilation affected by some special implementation-dependent environment which dictates that names default to `SEQUENCE`?
- If not, then the compiler will consider the object or `COMMON` block name nonsequential and is free to apply data mapping optimizations that disregard Fortran sequence and storage association.

(*End of advice to users.*)

Advice to implementors. In order to protect the user and to facilitate portability of older codes, two implementation options are strongly recommended. First, every implementation should supply some mechanism to verify that the type and shape of every mappable array and the sizes of aggregate variable groups in `COMMON` blocks are the same in every scoping unit unless the `COMMON` blocks are declared to be sequential. This same check should also verify that identical mappings have been selected for the variables in `COMMON` blocks. Implementations without interprocedural information can use a link-time check. The second implementation option recommended is a mechanism to declare that data objects and `COMMON` blocks for a given compilation should be considered sequential unless declared otherwise. The purpose of this feature is to permit compilation of large old libraries or subprograms where storage association is known to exist without requiring that the code be modified to apply the HPF `SEQUENCE` directive to every `COMMON` block. (*End of advice to implementors.*)

Section 4

Data Mapping in Subprogram Interfaces

In this Section, phrases such as "the caller must pass..." are constraints on the implementation (i.e., on the generated code produced by the compiler), not on the source code produced by the programmer.

4.1 Introduction

This introduction gives an overview of the ways in which mapping directives interact with argument passing to subprograms. The language used here, however, is not definitive; the subsequent subsections of this Section contain the authoritative rules.

In addition to the data mapping features described in Section 3, HPF allows a number of options for describing the mapping of dummy arguments.

The mapping of each such dummy argument may be related to the mapping of its associated actual argument in the calling main program or procedure (the "caller") in several different ways. To allow for this, mapping directives applied to dummy arguments can have three different syntactic forms: *prescriptive*, *descriptive*, and *transcriptive*.

HPF provides these three forms to allow the programmer either to specify that the data is to be left in place, or to specify that during the execution of the call the data must be automatically remapped into a new and presumably more efficient mapping for the duration of the execution of the called subprogram.

The meaning of these forms is as follows:

prescriptive The directive describes the mapping of the dummy argument. However, the actual argument need not have this mapping. *If it does not*, it is the responsibility of the compiler to generate code to remap the argument as specified, and to restore the original mapping on exit. This code may be generated either in the caller or in the called subprogram; the requirements for explicit interfaces in Section 4.6 insure that the necessary information will be available at compile time to perform the mapping in either place.

Prescriptive directives are syntactically identical to directives occurring elsewhere in the program. For instance, if A is a dummy argument,

```
!HPF$ DISTRIBUTE A (BLOCK, CYCLIC)
```

is a prescriptive directive.

descriptive Descriptive syntax has exactly the same meaning as prescriptive syntax, except that in addition it amounts to a weak assertion by the programmer that the actual argument requires no remapping.

The assertion is characterized as "weak" because if it is false, the program is still standard-conforming. In such a case, the compiler must generate the appropriate remapping.

If the compiler can prove that the assertion is false, or if the compiler cannot verify that it is true, it may issue a warning or informational diagnostic message.

Advice to users. The purpose of descriptive, as opposed to prescriptive, directives is simply to provide a possible way for the compiler to report information to the programmer that may be useful in program development and debugging. Note that any diagnostic message that may be produced as a result of the use of descriptive directives is not a portable feature of this language. In particular, there are instances in which no remapping is needed but where this fact would be impossible or highly non-trivial for a compiler to ascertain. Different compilers may well emit messages in different circumstances; and there is no requirement that any such messages be emitted at all. (*End of advice to users.*)

Descriptive directives look like prescriptive directives, except that an asterisk precedes the description. For instance,

```
!HPF$ DISTRIBUTE A *(BLOCK, CYCLIC)
```

is a descriptive directive.

transcriptive The mapping is unspecified. The called subprogram must accept the mapping of the argument as it is passed. Of course this means that (the implementation of) the caller must pass this mapping information at run-time.

Transcriptive directives are written with a single asterisk for distributions and processor arrangements; for instance

```
!HPF$ DISTRIBUTE A *
!HPF$ DISTRIBUTE B * ONTO *
```

are transcriptive directives. The `INHERIT` directive (see Section 4.4.2) is used to specify a transcriptive alignment.

Both distribution formats and processor arrangements can be specified prescriptively, descriptively, or transcriptively. Alignment is more complicated, because of the need to specify the template with which the dummy is aligned. This template may be unspecified (in this case of course there is no `ALIGN` directive), in which case it is the *natural template* of the dummy. ("Natural template" is defined in Section 4.4.1 below.) Otherwise, one of the following disjoint possibilities must be true:

- The template is explicitly specified by a prescriptive `ALIGN` directive.

- The template is explicitly specified by a descriptive `ALIGN` directive.
- The template is *inherited*. This is specified by giving the dummy the `INHERIT` attribute (described in Section 4.4.2 below). This implicitly specifies the template to be a copy of the template with which the corresponding actual argument is ultimately aligned; further, the alignment of the dummy with that template is the same as that of the corresponding actual. This is, in effect, a transcriptive form of alignment.

This is restated more precisely in Section 4.4.1 below.

Advice to users. Although it is possible to write some combinations of mapping directives that are partially prescriptive and partially transcriptive, for instance, there is probably no virtue in doing so. The point of these directives is to enable the compiler to handle any necessary remapping correctly and efficiently. Now remapping can happen for one or more of the following reasons:

- to make the alignment of the actual and the dummy agree;
- to make the distribution of the actual and the dummy agree;
- to make the processor arrangement of the actual and the dummy agree.

For most machines, there is no real difference in the cost of remapping for any of these reasons. It is therefore a better practice (for readability, at least) to make a mapping either purely transcriptive, purely prescriptive, or purely descriptive.

While transcriptive mappings can be useful in writing libraries, they impose a run-time cost on the subprogram. They should therefore be avoided in normal user code. (*End of advice to users.*)

4.2 What Remapping Is Required and Who Does It

If there is an explicit interface for the called subprogram and that interface contains prescriptive or descriptive mapping directives for a dummy argument, and if a remapping of the corresponding actual argument is necessary, the call should proceed as if the data were copied to a temporary variable to match the mapping of the dummy argument as expressed by the directives in the explicit interface. The template of the dummy will then be as declared in the interface.

If there is no explicit interface, then no remapping will be necessary; this is a consequence of the requirements in Section 4.6.

The reader should note that for reasons of brevity, not all such required explicit interfaces are included in the code fragments in this Section.

An overriding principle is that *any remapping of arguments is not visible to the caller.* That is, when the subprogram returns and the caller resumes execution, all objects accessible to the caller after the call are mapped exactly as they were before the call. It is not possible for a procedure to change the mapping of any object in a manner visible to its caller.

Advice to users. Some Approved Extensions relax this restriction; see for instance Sections 8.6 and 8.8. (*End of advice to users.*)

⇓

4.3 Distributions and Processor Arrangements

In a DISTRIBUTE directive where every *distributee* is a dummy argument, either the *dist-format-clause* or the *dist-target*, or both, may begin with, or consist of, an asterisk.

- Without an asterisk, a *dist-format-clause* or *dist-target* is prescriptive; the clause describes a distribution and constitutes a request of the language processor to make it so. This might require (the implementation of) either the caller or the called subprogram to remap or copy the actual argument on entry at run time in order to satisfy the requested distribution for the dummy.
- Starting with an asterisk, a *dist-format-clause* or *dist-target* is descriptive. Such a directive is equivalent in every respect to a prescriptive directive, except that if the compiler cannot verify that no remapping of the actual is required, it may issue a diagnostic message to that effect. See Section 4.1 for further information on this point.
- Consisting of only an asterisk, a *dist-format-clause* or *dist-target* is transcriptive; the clause says nothing about the distribution but constitutes a request to the language processor to copy that aspect of the distribution from that of the actual argument. (The intent is that if the argument is passed by reference, no movement of the data will be necessary at run time.)

It is possible that, in a single DISTRIBUTE directive, the *dist-format-clause* might have an asterisk but not the *dist-target*, or vice versa.

4.3.1 Examples

These examples of DISTRIBUTE directives for dummy arguments illustrate the various combinations:

```
!HPF$ DISTRIBUTE URANIA (CYCLIC) ONTO GALILEO
```

The language processor should do whatever it takes to cause URANIA to have a CYCLIC distribution on the processor arrangement GALILEO.

```
!HPF$ DISTRIBUTE POLYHYMNIA * ONTO ELVIS
```

The language processor should do whatever it takes to cause POLYHYMNIA to be distributed onto the processor arrangement ELVIS, using whatever distribution format it currently has (which might be on some other processor arrangement).

```
!HPF$ DISTRIBUTE THALIA *(CYCLIC) ONTO *FLIP
```

The language processor should do whatever it takes to cause THALIA to have a CYCLIC distribution on the processor arrangement FLIP; the programmer believes that the actual is already distributed in this fashion and that no remapping is required.

```
!HPF$ DISTRIBUTE EUTERPE (CYCLIC) ONTO *
```

The language processor should do whatever it takes to cause EUTERPE to have a CYCLIC distribution onto whatever processor arrangement the actual was distributed onto.

```
!HPF$ DISTRIBUTE ERATO * ONTO *
```

The mapping of ERATO should not be changed from that of the actual argument.

Note that DISTRIBUTE ERATO * ONTO * does not mean the same thing as

```
!HPF$ DISTRIBUTE ERATO (*) ONTO *
```

This latter means: distribute ERATO * (that is, on-processor) onto whatever processor arrangement the actual was distributed onto. The processor arrangement is necessarily scalar in this case.

4.3.2 What Happens When a Clause Is Omitted

One may omit either the *dist-format-clause* or the *dist-onto-clause* for a dummy argument. This is understood as follows:

If the dummy argument has the INHERIT attribute (see Section 4.4.2), then no distribution directive is allowed in any case: the distribution as well as the alignment is inherited from the actual argument.

In any other case in which distribution information is omitted, the compiler may choose the distribution format or a target processor arrangement arbitrarily.

Here are two examples:

```
!HPF$ DISTRIBUTE WHEEL_OF_FORTUNE *(CYCLIC)
```

The programmer believes that the actual argument corresponding to the dummy argument WHEEL_OF_FORTUNE is already distributed CYCLIC. The compiler should insure that the mapping of the passed data is in fact CYCLIC, and remap it if necessary if it is not. It may in addition be remapped onto some other processor arrangement, but there is no reason to; most likely the programmer would be surprised if such a remapping occurred.

```
!HPF$ DISTRIBUTE ONTO *TV :: DAVID_LETTERMAN
```

The programmer believes that the actual argument corresponding to the dummy argument DAVID_LETTERMAN is already distributed onto TV in some fashion. The compiler should insure that this is so, and make it so if it is not. The distribution format may be changed as long as DAVID_LETTERMAN is kept on TV. (Note that this declaration must be made in attributed form; the statement form

```
!HPF$ DISTRIBUTE DAVID_LETTERMAN ONTO *TV         !Nonconforming
```

does not conform to the syntax for a DISTRIBUTE directive.)

4.4 Alignment

4.4.1 The Template of the Dummy Argument

Here we describe precisely how to determine the template with which the dummy argument is ultimately aligned:

Templates are not passed through the subprogram argument interface. A dummy argument and its corresponding actual argument may be aligned to the same template only if that template is accessible in both the caller and the called subprogram either through host association or use association. In any other case, the template with which a dummy argument is aligned is always distinct from the template with which the actual argument is aligned, though it may be a copy (see Section 4.4.2). On exit from a procedure, an HPF implementation arranges that the actual argument is aligned with the same template with which it was aligned before the call.

The template of the dummy argument is arrived at in one of three ways:

- If the dummy argument appears explicitly as an *alignee* in an `ALIGN` directive, its template is the *align-target* if the *align-target* is a template; otherwise its template is the template with which the *align-target* is ultimately aligned.
- If the dummy argument is not explicitly aligned and does not have the `INHERIT` attribute (described in Section 4.4.2 below), then the template has the same shape and bounds as the dummy argument; this is called the *natural template* for the dummy.

  (Thus, all the examples in Section 4.3 use the natural template.)
- If the dummy argument is not explicitly aligned and does have the `INHERIT` attribute, then the template is "inherited" from the actual argument according to the following rules:
  - If the actual argument is a whole array, the template of the dummy is a copy of the template with which the actual argument is ultimately aligned.
  - If the actual argument is an array section of array *A* where no subscript is a vector subscript, then the template of the dummy is a copy of the template with which *A* is ultimately aligned.
  - If the actual argument is any other expression, the shape and distribution of the template may be chosen arbitrarily by the language processor (and therefore the programmer cannot know anything *a priori* about its distribution).

  In all of these cases, we say that the dummy has an *inherited template.*

4.4.2 The INHERIT Directive

The `INHERIT` directive specifies that a dummy argument should be aligned to a copy of the template of the corresponding actual argument in the same way that the actual argument is aligned.

H401	*inherit-directive*	**is**	`INHERIT` *inheritee-list*
H402	*inheritee*	**is**	*object-name*

Constraint: An *inheritee* must be a dummy argument.

Constraint: An *inheritee* must not be an *alignee.*

Constraint: An *inheritee* must not be a *distributee.*

Advice to users. The first of these three constraints is relaxed for pointers under the approved extensions (see Section 8.8). (*End of advice to users.*)

⇓

The INHERIT directive causes the named subprogram dummy arguments to have the INHERIT attribute. Only dummy arguments may have the INHERIT attribute. An object must not have both the INHERIT attribute and the ALIGN attribute. An object must not have both the INHERIT attribute and the DISTRIBUTE attribute. The INHERIT directive may appear only in a *specification-part* of a scoping unit.

The INHERIT attribute specifies that the template for a dummy argument should be inherited, by making a copy of the template of the actual argument. Moreover, no other explicit mapping directive may appear for an argument with the INHERIT attribute: the INHERIT attribute implies a distribution of DISTRIBUTE * ONTO * for the inherited template. Thus, the net effect is to tell the compiler to leave the data exactly where it is, and not attempt to remap the actual argument. The dummy argument will be mapped in exactly the same manner as the actual argument; the subprogram must be compiled in such a way as to work correctly no matter how the actual argument may be mapped onto abstract processors.

Note that if A is an array dummy argument, the directive

```
!HPF$ INHERIT A
```

is more general than

```
!HPF$ DISTRIBUTE A * ONTO *
```

for the following reason: The INHERIT directive states that the (inherited) template with which A is aligned is distributed * ONTO *, but that A may be aligned in some non-trivial manner with that template. On the other hand, the DISTRIBUTE directive states that A is aligned trivially with its natural template, which in turn is distributed * ONTO *.

For example, the following code is not permitted:

```
!HPF$ PROCESSORS P(2)
      REAL, DIMENSION(100) :: A
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A

      CALL FOO(A(1:50))

      ...

      SUBROUTINE FOO(D)
      REAL, DIMENSION(50) :: D
!HPF$ DISTRIBUTE D *            ! Nonconforming
```

The transcriptive distribution for D is nonconforming because the natural template for D is not distributed BLOCK. On the other hand, it would be correct to replace the illegal directive by

```
!HPF$ INHERIT D
```

4.4.2.1 Examples

Here is a straightforward example of the use of INHERIT:

```
      REAL DOUGH(100)
!HPF$ DISTRIBUTE DOUGH(BLOCK(10))
      CALL PROBATE( DOUGH(7:23:2) )
      ...
      SUBROUTINE PROBATE(BREAD)
      REAL BREAD(9)
!HPF$ INHERIT BREAD
```

The inherited template of `BREAD` has shape [100]; element `BREAD(I)` is aligned with element `5 + 2*I` of the inherited template, and that template has a `BLOCK(10)` distribution.

More complicated examples can easily be constructed. It is important to bear in mind that the rank of the inherited template may be different from the rank of the dummy, and it might even be different from the rank of the actual. For instance, one might have a program containing the following:

```
      REAL A(100,100)
!HPF$ TEMPLATE T(100,100,100)
!HPF$ DISTRIBUTE T(BLOCK,CYCLIC,*)
!HPF$ ALIGN A(I,J) with T(J,3,I)
      CALL SUBR(A(:,7))
      ...
      SUBROUTINE SUBR(D)
      REAL D(100)
!HPF$ INHERIT D
```

In this case, the dummy `D` has rank 1. It corresponds to a 1-dimensional section of a 2-dimensional actual `A`, which in turn is aligned with a 2-dimensional section of a 3-dimensional template `T`. The template of `D` is a copy of this three-dimensional template. `D` is aligned with the section `(7, 3, :)` of this inherited template. Thus, the "visible" dimension of the dummy `D` is distributed `*`, although if the call statement had been

```
      CALL SUBR(A(7,:))
```

for instance, the "visible" dimension of the dummy would be distributed `BLOCK`.

4.4.3 Descriptive ALIGN Directives

The presence or absence of an asterisk at the start of an *align-spec* has the same meaning as in a *dist-format-clause*: it specifies whether the `ALIGN` directive is descriptive or prescriptive, respectively.

If an *align-spec* that does not begin with `*` is applied to a dummy argument, the meaning is that the dummy argument will be forced to have the specified alignment on entry to the subprogram. This may require (the implementation of) either the caller or the subprogram to temporarily remap the data of the actual argument or a copy thereof.

Note that a dummy argument may also be used as an *align-target*.

```
      SUBROUTINE NICHOLAS(TSAR,CZAR)
      REAL, DIMENSION(1918) :: TSAR,CZAR
!HPF$ INHERIT :: TSAR
!HPF$ ALIGN WITH TSAR :: CZAR
```

In this example the first dummy argument, `TSAR`, remains aligned with the corresponding actual argument, while the second dummy argument, `CZAR`, is forced to be aligned with the first dummy argument. If the two actual arguments are already aligned, no remapping of the data will be required at run time. If they are not, some remapping will take place.

If the *align-spec* begins with "`*`", then the *alignee* must be a dummy argument. The "`*`" indicates that the programmer believes that the actual argument already has the specified alignment, and that no action to remap it is required at run time. (As before, there is no requirement that the programmer's belief is correct, and the compiler must generate a remapping if one appears to be necessary, just as in the case of a prescriptive alignment.) For example, if in the above example the alignment directive were changed to

```
!HPF$ ALIGN WITH *TSAR :: CZAR
```

then the programmer is expressing a belief that no remapping of the actual argument corresponding to `TSAR` will be necessary.

It is not permitted to say simply "`ALIGN WITH *`"; an *align-target* must follow the asterisk. (The proper way to say "accept any alignment" is `INHERIT`.)

If a dummy argument has no explicit `ALIGN` or `DISTRIBUTE` attribute, then the compiler provides an implicit alignment and distribution specification, one that could have been described explicitly without any "assertion asterisks".

4.4.3.1 Example

If the `INHERIT` directive is not used, explicit alignment of a dummy argument may be necessary to insure that no remapping takes place at the subprogram boundary. Here is an example:

```
      LOGICAL FRUG(128)
!HPF$ PROCESSORS DANCE_FLOOR(16)
!HPF$ DISTRIBUTE (BLOCK) ONTO DANCE_FLOOR::FRUG
      CALL TERPSICHORE(FRUG(1:40:3))
```

The array section `FRUG(1:40:3)` is mapped onto abstract processors in the following manner:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1			25												
	10			34											
		19													
4			28												
	13			37											
		22													
7			31												
	16			40											

Suppose first that the interface to the subroutine `TERPSICHORE` looks like this:

```
      SUBROUTINE TERPSICHORE(FOXTROT)
      LOGICAL FOXTROT(:)
!HPF$ INHERIT FOXTROT
```

The template of FOXTROT is a copy of the 128 element template of the whole array FRUG. The template is mapped like this:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	9	17	25	33	41	49	57	65	73	81	89	97	105	113	121
2	10	18	26	34	42	50	58	66	74	82	90	98	106	114	122
3	11	19	27	35	43	51	59	67	75	83	91	99	107	115	123
4	12	20	28	36	44	52	60	68	76	84	92	100	108	116	124
5	13	21	29	37	45	53	61	69	77	85	93	101	109	117	125
6	14	22	30	38	46	54	62	70	78	86	94	102	110	118	126
7	15	23	31	39	47	55	63	71	79	87	95	103	111	119	127
8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128

FOXTROT(I) is aligned with element 3*I-2 of the template.

Suppose, on the other hand, that the interface to TERPSICHORE were to look like this instead:

```
      SUBROUTINE TERPSICHORE(FOXTROT)
      LOGICAL FOXTROT(:)
!HPF$ DISTRIBUTE FOXTROT(BLOCK)
```

In this case, the template of FOXTROT is its natural template; it has the same size 14 as FOXTROT itself. The actual argument, FRUG(1:40:3) is mapped to the 16 processors in this manner:

Abstract processor	Elements of FRUG
1	1, 2, 3
2	4, 5, 6
3	7, 8
4	9, 10, 11
5	12, 13, 14
6–16	none

That is, the original positions (in the template of the actual argument) of the elements of the dummy are as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1			9												
	4			12											
		7													
2			10												
	5			13											
		8													
3			11												
	6			14											

This layout (3 elements on the first processor, 3 on the second, 2 on the third, 3 on the fourth, ...) cannot properly be described as a BLOCK distribution. Therefore, remapping will take place at the call.

Remapping can be avoided without using INHERIT by explicitly aligning the dummy to a declared template of size 128 distributed BLOCK:

```
      SUBROUTINE TERPSICHORE(FOXTROT)
      LOGICAL FOXTROT(:)
!HPF$ PROCESSORS DANCE_FLOOR(16)
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK) ONTO DANCE_FLOOR::GURF(128)
!HPF$ ALIGN FOXTROT(I) WITH GURF(3*I-2)
```

Advice to users. The advantage of this latter technique is that, where it can be used, it gives the compiler more information; this information can often be used to generate more efficient code. (*End of advice to users.*)

4.5 Equivalence and Partial Order on the Set of Mappings

The set of mappings of named objects is endowed with a partial order modulo a certain equivalence. Roughly speaking, if P and Q are two mappings, then to say that Q is a *specialization* of P (i.e., "Q is below P" in this ordering) is to say that P is partially specified and that Q is one of the mappings that is consistent with P. This notion is used below in Section 4.6, and also in Section 8.8.

Advice to users. Since these conditions are complex to state, it is worth noting that if you always provide explicit interfaces (which, as explained below, is quite easy and generally happens automatically) and if you don't use mapped pointers (an Approved Extension, explained below in Section 8.8), then you need not read this Section. (*End of advice to users.*) ⇓

The precise definition is as follows.

First, we define a notion of equivalence for *dist-format* specifications:

1. Using the notation $\equiv$ for the phrase "is equivalent to",

`BLOCK`	$\equiv$	`BLOCK`	
`CYCLIC`	$\equiv$	`CYCLIC`	
`*`	$\equiv$	`*`	
`BLOCK(`*n*`)`	$\equiv$	`BLOCK(`*m*`)`	iff m and n have the same value
`CYCLIC(`*n*`)`	$\equiv$	`CYCLIC(`*m*`)`	iff m and n have the same value
`CYCLIC`	$\equiv$	`CYCLIC(1)`	

2. Other than this, no two lexically distinct *dist-format* specifications are equivalent.

This is an equivalence relation in the usual mathematical sense.

Now we define the partial order on mappings: Let `S` ("special") and `G` ("general") be two data objects.

The mapping of `S` is a *specialization* of the mapping of `G` if and only if either:

1. `G` has the `INHERIT` attribute, or
2. `S` does not have the `INHERIT` attribute, and the following constraints all hold:
   (a) `S` is a named object, and
   (b) The shapes of the ultimate align targets of `S` and `G` are the same, and
   (c) Corresponding dimensions of `S` and `G` are mapped to corresponding dimensions of their respective ultimate align targets, and corresponding elements of `S` and `G` are aligned with corresponding elements of their respective ultimate align targets, and
   (d) Either
      i. The ultimate align targets of both `S` and `G` are not explicitly distributed, or
      ii. The ultimate align targets of both `S` and `G` are explicitly distributed. In this case, the distribution directive specified for the ultimate align target of `G` must satisfy one of the following conditions:
         A. It has no *dist-onto-clause*, or
         B. It has a *dist-onto-clause* of "`ONTO *`", or
         C. It has a *dist-onto-clause* specifying a processor arrangement having the same shape as that explicitly specified in a distribution directive for the ultimate align target of `S`.

         and must also satisfy one of the following conditions:
         A. It has no *dist-format-clause*, or
         B. It has a *dist-format-clause* of "`*`", or
         C. Each *dist-format* is equivalent (in the sense defined above) to the *dist-format* in the corresponding position of the *dist-format-clause* in an explicit distribution directive for the ultimate align target of `S`.

With this definition,

- Any mapping of a named object is a specialization of itself.

- If A, B, and C are named objects, and if the mapping of A is a specialization of the mapping of B and the mapping of B is a specialization of the mapping of C, then the mapping of A is a specialization of the mapping of C.

That is, the specialization relation, as applied to mappings of named objects, is reflexive and transitive, and it can therefore be applied to produce an equivalence relation on the set of mappings of named objects: two such mappings can be said to be equivalent iff each is a specialization of the other. With this definition, the specialization relation yields a partial ordering on the set of mappings of named objects, modulo equivalence. The `INHERIT` mapping is the unique maximal element in this partial order.

4.6 Conditions for Omitting Explicit Interfaces

Under certain conditions, an explicit interface for a subprogram is not required. The conditions in Fortran under which this is allowable are tightened considerably for HPF programs that use mapping directives.

Advice to users. These conditions are complex. The important thing to realize is that you don't have to read any of this if you have an explicit interface. So if there is any doubt in your mind, just make sure you have an explicit interface. (*End of advice to users.*)

An explicit interface is required *except* when all of the following conditions hold:

1. Fortran does not require one, *and*
2. No dummy argument is distributed transcriptively or with the `INHERIT` attribute, *and*
3. For each pair of corresponding actual and dummy arguments, either:
   (a) They are both implicitly mapped, or
   (b) They are both explicitly mapped and the mapping of the actual argument is a specialization of the mapping of the dummy argument,

   and
4. For each pair of corresponding actual and dummy arguments, either:
   (a) Both are sequential, or
   (b) Both are nonsequential.

Rationale. This has the following consequences:

- A plain Fortran program (i.e., with no HPF directives) will be HPF-conforming without the need to add additional interfaces, at least in a compilation environment in which all variables are sequential by default. This is insured by items 1, 2, 3(a), and 4(a).
- If remapping is necessary, this fact will be visible to the caller. Thus the implementation may choose to have all remapping performed by the caller.

(*End of rationale.*)

Advice to users. This requirement pushes the user strongly in the direction of always providing explicit interfaces. This is a good thing—explicit interfaces allow many errors to be caught at compile-time and greatly speed up the process of robust software development.

Note that an explicit interface can be provided in three ways:

1. A module subprogram has an explicit interface.
2. An internal subprogram has an explicit interface.
3. An explicit interface may be provided by an interface block.

In addition, an intrinsic procedure always has an explicit interface by definition.

The idiomatic Fortran way of programming makes extensive use of modules; every subprogram, for instance, can be in a module. This provides explicit interfaces automatically, with no extra effort on the part of the programmer. It should very seldom be necessary to write an interface block. (*End of advice to users.*)

4.7 Characteristics of Procedures

The characteristics of dummy data objects and function results as given in the Fortran standard (F95:12.2) are extended to include also the *HPF-characteristics* of such objects, which are defined recursively as follows:

- A processor arrangement has one HPF-characteristic: its shape.
- A template has up to three HPF-characteristics:
  1. its shape;
  2. its distribution, if explicitly stated;
  3. the HPF-characteristic (i.e., the shape) of the processor arrangement onto which it is distributed, if explicitly stated.
- A dummy data object has the following HPF-characteristics:
  1. its alignment, if explicitly stated, as well as all HPF-characteristics of its align target;
  2. its distribution, if explicitly stated, as well as the HPF-characteristic (i.e., the shape) of the processor arrangement onto which it is distributed, if explicitly stated.
- A function result has the same HPF-characteristics as a dummy data object. Specifically, it has the following HPF-characteristics:
  1. its alignment, if explicitly stated, as well as all HPF-characteristics of its align target;

2. its distribution, if explicitly stated, as well as the HPF-characteristic (i.e., the shape) of the processor arrangement onto which it is distributed, if explicitly stated.

Rationale. In case an explicit interface is given by an interface block, the Fortran standard specifies what information must be specified in that interface block; it does this using the concept of a Fortran *characteristic*. Characteristics of dummy data objects, for instance, include their types. Characteristics must be specified in interface blocks; F95:12.3.2.1 in the Fortran standard states

> An interface body specifies all of the procedure's characteristics and these shall be consistent with those specified in the procedure definition ...

Normally, an interface block for a procedure is a textual copy of the appropriate declarations of that procedure. This Section simply says that such a textual copy must include any explicit mapping directives relevant to dummy arguments of the procedure. (*End of rationale.*)

4.8 Argument Passing and Sequence Association

For actual arguments in a procedure call, Fortran allows an array element (scalar) to be associated with a dummy argument that is an array. It furthermore allows the shape of a dummy argument to differ from the shape of the corresponding actual array argument, in effect reshaping the actual argument via the procedure call. Storage sequence properties of Fortran are used to identify the values of the dummy argument. This feature, carried over from FORTRAN 77, has been widely used to pass starting addresses of subarrays, rows, or columns of a larger array, to procedures. For HPF arrays that are potentially mapped across processors, this feature is not fully supported.

4.8.1 Sequence Association Rules

1. When an array element or the name of an assumed-size array is used as an actual argument, the associated dummy argument must be a scalar or specified to be a sequential array.

   An array-element designator of a nonsequential array must not be associated with a dummy array argument.

2. When an actual argument is an array or array section and the corresponding dummy argument differs from the actual argument in shape, then the dummy argument must be declared sequential and the actual array argument must be sequential.

3. An object of type character (scalar or array) is nonsequential if it conforms to the requirements of Definition 4 of Section 3.8.1.1. If the length of an explicit-length character dummy argument differs from the length of the actual argument, then both the actual and dummy arguments must be sequential.

4. Without an explicit interface, a sequential actual may not be associated with a nonsequential dummy and a nonsequential actual may not be associated with a sequential dummy. (This item merely repeats part of Section 4.6).

4.8.2 Discussion of Sequence Association

When the shape of the dummy array argument and its associated actual array argument differ, the actual argument must not be an expression. There is no HPF mechanism for declaring that the value of an array-valued expression is sequential. In order to associate such an expression as an actual argument with a dummy argument of different rank, the actual argument must first be assigned to a named array variable that is forced to be sequential according to Definition 4 of Section 3.8.1.1.

4.8.3 Examples of Sequence Association

Given the following subroutine fragment:

```
SUBROUTINE HOME (X)
DIMENSION X (20,10)
```

By rule 1

```
CALL HOME (ET (2,1))
```

is legal only if `X` is declared sequential in `HOME` and `ET` is sequential in the calling procedure.

Likewise, by rules 2 and 4

```
CALL HOME (ET)
```

requires either that `ET` and `X` are both sequential arrays or that `ET` and `X` have the same shape and (in the absence of an explicit interface) have the same sequence attribute.

Rule 3 addresses a special consideration for objects of type character. Change of the length of character objects across a call, as in

```
CHARACTER (LEN=44) one_long_word
one_long_word = 'Chargoggagoggmanchaugagoggchaubunagungamaugg'
CALL webster(one_long_word)

SUBROUTINE webster(short_dictionary)
CHARACTER (LEN=4) short_dictionary (11)
    !Note that short_dictionary(3) is 'agog', for example
```

is conceptually legal in Fortran. In HPF, both the actual argument and dummy argument must be sequential. (Chargoggagoggmanchaugagoggchaubunagungamaugg is the original Nipmuc name for what is now called Lake Webster in Massachusetts.)

Section 5

INDEPENDENT and Related Directives

The HPF INDEPENDENT directive allows the programmer to give information to the compiler concerning opportunities for parallel execution. The user can assert that no data object is defined by one iteration of a DO loop and used (read or written) by another; similar information can be provided about the combinations of index values in a FORALL statement. Such information is sometimes valuable to enable compiler optimization, but may require knowledge of the application that is available only to the programmer. HPF therefore allows a user to make these assertions, and the compiler may rely on them in its translation process. If the assertion is true, the semantics of the program are not changed; if it is false, the program is not HPF-conforming and has no defined meaning.

In contrast to HPF 1.0, the INDEPENDENT assertion of HPF 2.0 allows reductions to be performed in INDEPENDENT loops, provided the reduction operator is a built-in, associative and commutative Fortran operator (such as .AND.) or function (such as MAX). It is often the case that a data parallel computation cannot be expressed in HPF 1.0 as an INDEPENDENT loop because several loop iterations update one or more variables. In such cases parallelism may be possible and desirable because the order of updates is immaterial to the final result. This is most often the case with accumulations, such as the following loop:

```
DO I = 1, 1000000000
    X = X + COMPLICATED_FUNCTION(I)
END DO
```

This loop can run in parallel as long as its iterations make their modifications to the shared variable X in an atomic manner. Alternatively, the loop can be run in parallel by making updates to temporary local accumulator variables, with a (short) final phase to merge the values of these variables with the initial value of X. In either case, the computation is conceptually parallel, but it cannot be asserted to be INDEPENDENT by the strict definition found in HPF 1.0.

It is worth mentioning that Fortran now includes several means to express data parallel computation:

- Array assignments, including the WHERE statement.

- Elemental invocation of intrinsic and user-defined functions.

- The FORALL statement and construct, including element-wise invocation of PURE functions.
- Transformational intrinsics such as SUM and TRANSPOSE.

FORALL and PURE were adopted by Fortran from HPF version 1.0. As these are all now part of Fortran, they are not discussed separately in this document.

5.1 The INDEPENDENT Directive

The INDEPENDENT directive can precede an indexed DO loop or FORALL statement. It asserts to the compiler that the iterations in the following DO loop or the operations in the following FORALL may be executed independently—that is, in any order, or interleaved, or concurrently—without changing the semantics of the program.

The INDEPENDENT directive precedes the DO loop or FORALL for which it asserts behavior, and is said to *apply* to that loop or FORALL. The syntax of the INDEPENDENT directive is

H501	*independent-directive*	**is**	INDEPENDENT [, *new-clause*] [, *reduction-clause*]
H502	*new-clause*	**is**	NEW (*variable-name-list*)
H503	*reduction-clause*	**is**	REDUCTION (*reduction-variable-list*)
H504	*reduction-variable*	**is**	*array-variable-name*
		or	*scalar-variable-name*
		or	*structure-component*

Constraint: The first non-comment line following an *independent-directive* must be a *do-stmt*, *forall-stmt*, or a *forall-construct*.

Constraint: If the first non-comment line following an *independent-directive* is a *do-stmt*, then that statement must contain a *loop-control* option containing a *do-variable*.

Constraint: If either the NEW clause or the REDUCTION clause is present, then the first non-comment line following the directive must be a *do-stmt*.

Constraint: A *variable* named in the NEW or the REDUCTION clause and any component or element thereof must not:

- Be a dummy argument;
- Have the SAVE or TARGET attribute;
- Occur in a COMMON block;
- Be storage associated with another object as a result of appearing in an EQUIVALENCE statement;
- Be use associated;
- Be host associated; or
- Be accessed in another scoping unit via host association.

Constraint: A variable that occurs as a *reduction-variable* may not appear in a *new-clause* in the same *independent-directive*, nor may it appear in either a *new-clause* or a *reduction-clause* in the range (i.e., the lexical body) of the following *do-stmt*, *forall-stmt*, or *forall-construct* to which the *independent-directive* applies.

Constraint: A *structure-component* in a *reduction-variable* may not contain a *subscript-section-list*.

Constraint: A variable that occurs as a *reduction-var* must be of intrinsic type. It may not be of type `CHARACTER`.

> *Rationale.* The second constraint means that an `INDEPENDENT` directive cannot be applied to a `WHILE` loop or a simple `DO` loop (i.e., a "do forever"). An `INDEPENDENT` in such cases could only correctly describe a loop with zero or one trips; the potential confusion was felt to outweigh the possible benefits. (*End of rationale.*)

When applied to a `DO` loop, an `INDEPENDENT` directive is an assertion by the programmer that no iteration can interfere with any other iteration, either directly or indirectly. The following operations define such interference:

- Any two operations that assign to the same atomic object interfere with each other. (A data object is called *atomic* if it contains no subobjects.)
  - Exception: If a variable appears in a `NEW` clause, then operations assigning values to it in separate iterations of the `DO` loop *do not* interfere. The reason for this is explained in Section 5.1.2.
  - Exception: If a variable appears in a `REDUCTION` clause, then assignments to it by reduction statements in the range of the `DO` loop *do not* interfere with assignments to it by other reduction statements in the same loop. The reason for this is explained in Section 5.1.3.

  Operations that assign to objects include:

  - Assignment statements assign to their left-hand side and all its subobjects.
  - `ASSIGN` statements assign to their integer variables.
  - `ALLOCATE` and `DEALLOCATE` statements with the `STAT=` specifier assign to the `STAT` variable.
  - `DO` statements assign to their indices.
  - I/O statements with the `IOSTAT=` specifier assign to the `IOSTAT` variable. They may also assign to other objects, as described below.
  - Asynchronous `READ` and `WRITE` statements (as described in Section 10) assign to their `ID=` variable.
  - `READ` statements assign to all variables in their input item list and any variables accessed at runtime through their `NAMELIST`. `READ` statements with the `SIZE=` specifier assign to the `SIZE` variable.
  - `INQUIRE` statements assign to all variables in their specifier list, except the `UNIT` and `FILE` specifiers.

  - Compound statements (e.g., `IF` statements) cause assignments to objects if their component statements do.
  - Subprogram invocations cause assignments to objects if operations in the subprogram execution do.

- An operation that assigns to an atomic object interferes with any operation that uses the value of that object.
  - Exception: If a variable appears in a `NEW` clause, then operations assigning values to it in one iteration of the `DO` loop *do not* interfere with uses of the variable in other iterations. The reason for this is explained in Section 5.1.2.
  - Exception: If a variable appears in a `REDUCTION` clause, then assignments to it by reduction statements in the range of the `DO` loop *do not* interfere with the allowed uses of it by reduction statements in the same loop. The reason for this is explained in Section 5.1.3.

Any expression that computes the value of a variable uses that object. This includes uses on the right-hand side of assignment statements, uses in subscripts on the left-hand side of assignment statements, conditional expressions, specification lists for I/O statements, output lists for `WRITE` statements, allocation shape specifications in `ALLOCATE` statements, and similar situations.

> *Rationale.* These are the classic Bernstein conditions to enable parallel execution. Note that two assignments *of the same value* to a variable interfere with each other and thus an `INDEPENDENT` loop with such assignments is not HPF-conforming. This is not allowed because such overlapping assignments are difficult to support on some hardware, and because the given definition was felt to be conceptually clearer. Similarly, it is not HPF-conforming to assert that assignment of multiple values to the same location is `INDEPENDENT`, even if the program logically can accept any of the possible values. In this case, both the "conceptually clearer" argument and the desire to avoid indeterminate behavior favored the given solution. (*End of rationale.*)

- An `ALLOCATE` statement, `DEALLOCATE` statement, `NULLIFY` statement or pointer assignment statement interferes with any other access, pointer assignment, allocation, deallocation, or nullification of the same pointer. In addition, an `ALLOCATE` or `DEALLOCATE` statement interferes with any other use of or assignment to the object that is allocated by `ALLOCATE` or deallocated by `DEALLOCATE`.

> *Rationale.* These constraints extend Bernstein's conditions to pointers. Because a Fortran pointer is an alias to an object or subobject rather than a first-class data type, a bit more care is needed than for other variables. (*End of rationale.*)

- Any transfer of control to a branch target statement outside the body of the loop interferes with all other operations in the loop.

- Any execution of an `EXIT`, `STOP`, or `PAUSE` statement interferes with all other operations in the loop.

Rationale. Branching (by `GOTO` or `ERR=` branches in I/O statements) implies that some iterations of the loop are not executed, which is drastic interference with those computations. The same is true for `EXIT` and the other statements. Note that these conditions do not restrict procedure calls in `INDEPENDENT` loops, except to disallow taking alternate returns to statements outside the loop, executing a `STOP`, or executing a `PAUSE`. (*End of rationale.*)

- Any two file I/O operations except `INQUIRE` associated with the same file or unit interfere with each other. Two `INQUIRE` operations do not interfere with each other; however, an `INQUIRE` operation interferes with any other I/O operation associated with the same file.

  Rationale. Because Fortran carefully defines the file position after a data transfer or file positioning statement, these operations affect the global state of a program. (Note that file position is defined even for direct access files.) Multiple non-advancing data transfer statements affect the file position in ways similar to multiple assignments of the same value to a variable, and is disallowed for the same reason. Multiple `OPEN` and `CLOSE` operations affect the status of files and units, which is another global side effect. `INQUIRE` does not affect the file status, and therefore does not affect other inquiries. However, other file operations may affect the properties reported by `INQUIRE`. (*End of rationale.*)

- Any data realignment or redistribution performed by subprogram invocation (see Section 4) interferes with any access to or any other remapping of the same data.

  Rationale. Remapping may change the processor storing a particular array element, which interferes with any assignment or use of that element. This applies even though the remappings are "undone" when the call returns. During the execution of the call, the homes of the array elements have changed, thus interfering with accesses in the caller, accesses in other invocations of the same procedure, and remappings of the array due to another procedure call. (*End of rationale.*)

  Advice to users. Data remapping performed by the `REALIGN` and `REDISTRIBUTE` aproved extensions also causes interference under this rule. See Chapter 8.5 for details. (*End of advice to users.*)

The interpretation of `INDEPENDENT` for `FORALL` is similar to that for `DO`: it asserts that no combination of the `FORALL` indices assigns to an atomic storage unit that is read by another combination. A `DO` and a `FORALL` with the same body are equivalent if they both have the `INDEPENDENT` directive. This is illustrated in Section 5.1.1.

If a procedure is called from within an `INDEPENDENT` loop or `FORALL`, then any local variables in that procedure are considered distinct on each call unless they have the `SAVE` attribute. This is consistent with the Fortran standard. Therefore, uses of local variables without the `SAVE` attribute in calls from different iterations do not cause interference as defined above.

Advice to implementors. A conforming Fortran implementation can often avoid creating distinct storage for locals on every call. The same is true for an HPF implementation; however, such an implementation must still interpret `INDEPENDENT` in the same way. If locals are not allocated unique storage locations on every call, then the `INDEPENDENT` loop must be serialized to respect these semantics (or other techniques must be used to avoid conflicting accesses). (*End of advice to implementors.*)

Note that all these rules describe interfering behavior; they do not disallow specific syntax. Statements that appear to violate one or more of these restrictions are allowed in an `INDEPENDENT` loop, if they are not executed due to control flow. These restrictions allow an `INDEPENDENT` loop to be executed safely in parallel if computational resources are available. The directive is purely advisory and a compiler is free to ignore it if it cannot make use of the information.

Advice to implementors. Although the restrictions allow safe parallel implementation of `INDEPENDENT` loops, they do not imply that this will be profitable (or even possible) on all architectures or all programs. For example,

- An `INDEPENDENT` loop may call a routine with explicitly mapped local variables. The implementation must then either implement the mapping (which may require serializing the calls, under some implementation strategies) or override the explicit directives (which may surprise the user).
- An `INDEPENDENT` loop may have very different behavior on different iterations. For example,

```
!HPF$ INDEPENDENT
      DO i = 1, 3
        IF (i.EQ.1) CALL F(A)
        IF (i.EQ.2) CALL G(B)
        IF (i.EQ.3) CALL H(C)
      END DO
```

  This poses obvious problems for implementations on SIMD machines.
- An `INDEPENDENT` loop may call a subroutine that accesses global mapped data. On distributed-memory machines, generating the communication to reference the data may be challenging, since there is in general no guarantee that the owners of the data will also call the subroutine.

In all cases, it is the implementation's responsibility to produce correct behavior, which may in turn limit optimization. It is recommended that implementations provide some feedback if an `INDEPENDENT` assertion may be ignored. (*End of advice to implementors.*)

5.1.1 Visualization of INDEPENDENT Directives

Graphically, the `INDEPENDENT` directive can be visualized as eliminating edges from a precedence graph representing the program. Figure 5.1 shows some of the dependences that may normally be present in a `DO` and a `FORALL`. (Most of the transitive dependences are not shown.) An arrow from a left-hand-side node (for example, "`lhsa(1)`") to a right-hand-side node ("`rhsb(1)`") means that the right-hand side computation might use values assigned in the left-hand-side node; thus the right-hand side must be computed after the left-hand side completes its store. Similarly, an arrow from a right-hand-side node to a left-hand-side node means that the left-hand side may overwrite a value needed by the right-hand side

```
DO i = 1, 3
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END DO
```

```
FORALL ( i = 1:3 )
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END FORALL
```

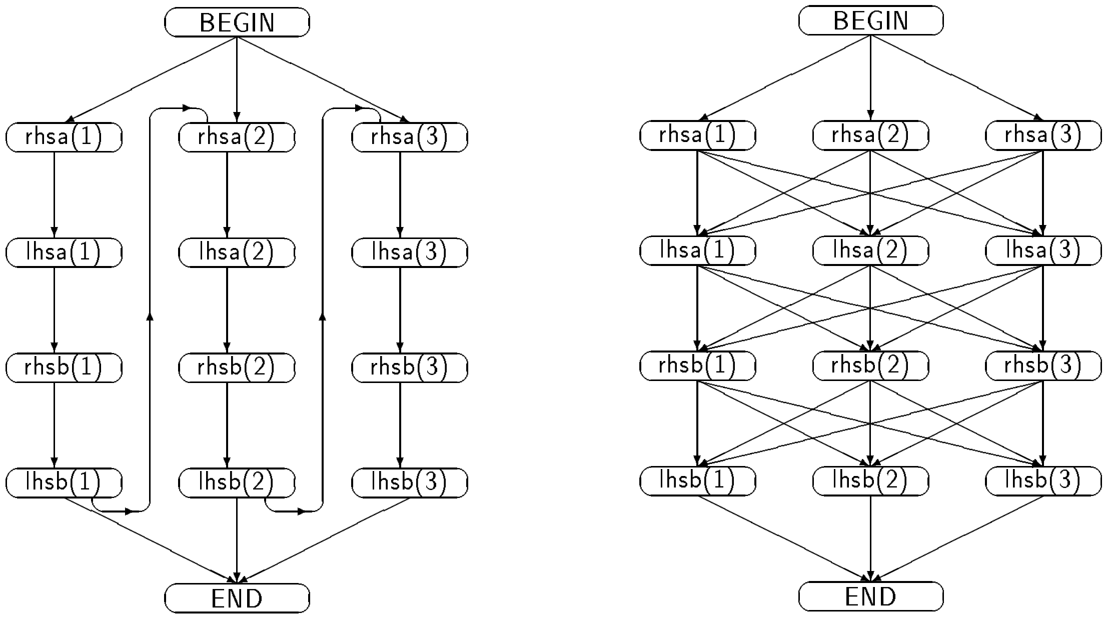


Figure 5.1: Dependences in `DO` and `FORALL` without `INDEPENDENT` assertions

```
!HPF$ INDEPENDENT
DO i = 1, 3
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END DO
```

```
!HPF$ INDEPENDENT
FORALL ( i = 1:3 )
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END FORALL
```

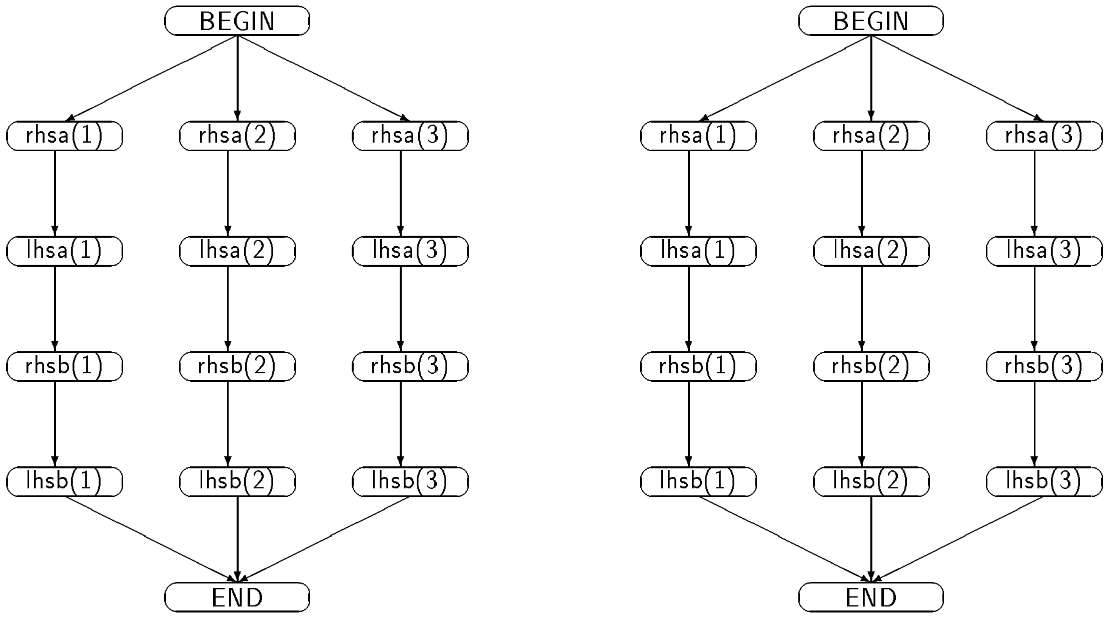


Figure 5.2: Dependences in `DO` and `FORALL` with `INDEPENDENT` assertions

computation, again forcing an ordering. Edges from the BEGIN and to the END nodes represent control dependences. The INDEPENDENT directive asserts that the only dependences that a compiler need enforce are those in Figure 5.2. That is, the programmer who uses INDEPENDENT is certifying that if the compiler enforces only these edges, then the resulting program will be equivalent to the one in which all the edges are present. Note that the set of asserted dependences is identical for INDEPENDENT DO and FORALL statements.

The compiler is justified in producing a warning if it can prove that one of these assertions is incorrect. It is not required to do so, however. A program containing any false assertion of this type is not HPF-conforming, thus is not defined by HPF, and the compiler may take any action it deems appropriate.

5.1.1.1 Examples of INDEPENDENT

```
!HPF$ INDEPENDENT
DO i = 2, 99
  A(I) = B(I-1) + B(I) + B(I+1)
END DO
```

This is one of the simplest examples of an INDEPENDENT loop. (For simplicity, all examples in this section assume there is no storage or sequence association between any variables used in the code.) Every iteration assigns to a different location in the A array, thus satisfying the first condition above. Since no elements of A are used on the right-hand side, no location that is assigned in the loop is also read, thus satisfying the second condition. Note, however, that many elements of B are used repeatedly; this is allowed by the definition of INDEPENDENT. This loop is INDEPENDENT regardless of the values of the variables involved.

```
!HPF$ INDEPENDENT
FORALL ( I=2:N ) A(I) = B(I-1) + B(I) + B(I+1)
```

This example is equivalent in all respects to the first example.

```
!HPF$ INDEPENDENT
DO I=1, 100
  A(P(I)) = B(I)
END DO
```

This INDEPENDENT directive asserts that the array P does not have any repeated entries (else they would cause interference when A was assigned). The DO loop is therefore equivalent to the Fortran statement

```
A(P(1:100)) = B(1:100)
```

5.1.2 NEW Variables

The NEW clause asserts that the named variables act as private variables to each iteration of the INDEPENDENT loop. That is, there would be no interfering assignments and uses in the loop, and thus no change in ghe behavior of the program, if new objects were created for the NEW variables for each iteration of the DO loop and those objects were destroyed at the end of each iteration. Thus, no values flow into NEW variables from execution before the loop, no values flow from NEW variables to execution after the loop, and (most importantly) no values flow from one iteration to another through NEW variables.

Advice to users. A pointer or allocatable variable may appear in a NEW clause. The interpretation of the paragraph above, in these cases, is that one should not rely on the value, the association status, or the allocation status of such a variable on entry to the loop; rather, such variables should be allocated or pointer assigned in the loop body before they are used. It would also be advisable to deallocate or nullify such a variable in the loop body after its last use as well. (*End of advice to users.*)

Rationale. NEW variables provide the means to declare temporaries in INDEPENDENT loops. Without this feature, many conceptually independent loops would need substantial rewriting (including expansion of scalars into arrays) to meet the rather strict requirements described above. Note that a temporary must be declared NEW only at the innermost lexical level at which it is assigned, since all enclosing INDEPENDENT assertions must take that NEW into account. Note also that index variables for nested DO loops must be declared NEW; the alternative was to limit the scope of an index variable to the loop itself, which changes Fortran semantics. FORALL indices, however, are restricted by the semantics of the FORALL; they require no NEW declarations. (*End of rationale.*)

5.1.2.1 Examples of NEW

```
!HPF$ INDEPENDENT, NEW(I)
DO I = 1, 10
   A(I) = B(I-1)
END DO
```

This example would be correct either with or without the NEW clause; in either case, the compiler could confidently parallelize the assignments to array A. Additionally however, the NEW clause asserts that the loop index I is not used after the completion of the loop. Some compilers may be able to use this information to avoid updating replicated copies of I on other processors, or to enable other optimizations.

```
!HPF$ INDEPENDENT, NEW (I2)
DO I1 = 1,N1
   !HPF$ INDEPENDENT, NEW (I3)
   DO I2 = 1,N2
      DO I3 = 2,N3   ! The inner loop is NOT independent!
          A(I1,I2,I3) = A(I1,I2,I3) - A(I1,I2,I3-1)*B(I1,I2,I3)
      END DO
   END DO
END DO
```

The inner loop is not independent because each element of A is computed from the preceding one. However, the two outer loops are independent because they access different elements of A. The NEW clauses are required, since the inner loop indices are assigned and used in different iterations of the outermost loops.

5.1.3 REDUCTION Variables and Statements

The REDUCTION clause asserts that the named variables are updated in the INDEPENDENT loop by a series of operations that are associative and commutative. Furthermore, the intermediate values of the REDUCTION variables are not used within the loop (except, of course, in the updates themselves). Thus, the value of a REDUCTION variable after the loop may be computed as the result of a reduction tree.

> *Rationale.* REDUCTION variables provide the means to accumulate values generated in an INDEPENDENT loop. Without this feature, the programmer must store update information in a temporary array whose size is equal to the number of loop iterations, and then use an intrinsic reduction function or XXX_SCATTER library function after the loop. The problem with this approach is that the temporary array may be excessively large. (*End of rationale.*)

The semantics of reductions are discussed in detail in Section 5.1.4. This section defines correct syntax.

Any variable whose name occurs as a *reduction-variable* is said to be *protected* while the immediately following DO loop is active (i.e. being executed). It may not be referenced while the loop in which it is protected is active, with one exception. It may occur in special locations in assignment statements of a special form, and these statements must be in the range (i.e. the lexical body) of the loop. In particular, it may not occur in any HPF directive, including the variable list in a NEW clause. This includes any NEW clause in the same INDEPENDENT directive.

A reduction statement is an assignment statement of the following special form that occurs in the range of an independent DO loop for which the name of its reduction variable occurs in a reduction clause. This description is not part of the grammar of HPF; rather, it serves to define the restricted assignment statements in which reduction variables are allowed.

H505	*reduction-stmt*	**is**	*variable* = *variable mult-op mult-operand*
		or	*variable* = *add-operand* * *variable*
		or	*variable* = *variable add-op add-operand*
		or	*variable* = *level*-2-*expr* + *variable*
		or	*variable* = *variable and-op and-operand*
		or	*variable* = *and-operand and-op variable*
		or	*variable* = *variable or-op or-operand*
		or	*variable* = *or-operand or-op variable*
		or	*variable* = *variable equiv-op equiv-operand*
		or	*variable* = *equiv-operand equiv-op variable*
		or	*variable* = *reduction-function* (*variable* , *expr*)
		or	*variable* = *reduction-function* (*expr* , *variable*)
H506	*reduction-function*	**is**	MAX
		or	MIN
		or	IAND
		or	IOR
		or	IEOR

Constraint: The two occurances of *variable* in a *reduction-stmt* must be textually identical.

The first two assertions of Section 5.1 account for the fact that the occurrences of reduction variables in their allowed positions in reduction statements do not cause interference between iterations of an INDEPENDENT loop. Any other assignment to or reference to a reduction variable *does* interfere with the reduction statement; this includes occurrences in subprograms and in the *expr* part of a reduction statement.

A variable that is updated by reduction statements in an independent loop must be protected by explicit appearance in a reduction clause. This clause must appear in the `INDEPENDENT` directive for the outermost independent loop that

- Contains the reduction statement;
- Does not have a `NEW` clause naming the reduction variable; and
- Lies within the innermost independent loop, if any, that contains the reduction statement and does have a `NEW` clause naming the reduction variable.

If the same variable is updated by two or more reduction statements, then the operators in those statements must be in the same class (e.g. both must be an *add-op* if one is).

Advice to users. When a reduction statement is executed, some nest of `DO` loops will be active. If there are several nested `INDEPENDENT DO` loops surrounding the reduction statements in which the variable is updated, which one is the right one to get the reduction clause? The answer is the outermost one, subject to the constraint that a reduction variable may not appear in a `NEW` clause for that loop or a contained loop. Consider

```
!HPF$ INDEPENDENT, NEW(J), REDUCTION(X)
DO I = 1, 10
    !HPF$ INDEPENDENT
    DO J = 1, 20
        X = X + J
    END DO
END DO
```

It would be incorrect to move the reduction clause to the inner `INDEPENDENT` directive. Since `X` is updated by reduction operations (twenty times) for *each* iteration of the outer loop, it does not have a well-defined value until the completion of the outer loop. (*End of advice to users.*)

The *reduction-variable* reference may be an array element or array section. The two references that occur in a reduction statement must be lexically identical. The Fortran rules of operator precedence and the use of parentheses in the expression must ensure that the reduction operator is the top-level operator (i.e. it is evaluated last) on the right-hand side. Therefore,

```
X = X * A + 1
```

is not a correctly formed reduction statement.

Note that the syntax of the `INDEPENDENT` directive does not allow an array element or array section to be designated as a reduction variable in the reduction clause. Even though such a subobject may occur in a reduction statement, it is the entire array or character variable that is treated as a reduction variable.

The allowed reduction operators and functions are all associative (in their mathematical definitions, even though the usual implementations of the arithmetic operators by Fortran language processors and the underlying hardware are not).

In most cases, only one operator will be used in the reduction statements (if there are more than one) that update a given reduction variable. It is sensible, however, to use `+` and `-` together on the same reduction variable: mathematically, subtraction is just addition of the additive inverse. For example:

```
!HPF$ INDEPENDENT, REDUCTION(X)
      DO I = 1, 100
        X(IDX(I,1)) = X(IDX(I,1)) + Y(I)
        X(IDX(I,2)) = X(IDX(I,2)) - Y(I)
      END DO
```

The same is true for multiplication (`*`) and division (`/`). No other mixing of operators is allowed.

Advice to users. While it is true that

```
X = I + X
```

is permitted as a reduction statement, for most purposes

```
X = X + I
```

is stylistically cleaner. (*End of advice to users.*)

5.1.4 Semantics and Implementation of Reduction

HPF specifies an allowed parallel implementation of an `INDEPENDENT DO` loop with reduction statements, thereby specifying the semantics of such a loop.

Just as the result of the Fortran intrinsic function `SUM` is defined to be a implementation-dependent approximation to the sum of the elements of its argument array, the value of a reduction variable on exit from its `INDEPENDENT DO` loop is likewise not completely specified by HPF. One possible value is that which would have been computed by sequential execution of the loop, but other implementation-dependent approximations to this value may be produced. Any such implementation-dependent value is, however, an approximation to the value produced by sequential execution of the loop. If rounding error, underflow, and overflow do not occur, it will be identical to that value.

Advice to users. If overflow, underflow, or rounding occur, this is one of the few places where an HPF directive in a conforming program may cause that program to produce different output. However, the same problems occur in other systems that attempt to parallelize these operations, for the same reasons. (*End of advice to users.*)

Since no reference to a protected reduction variable can occur except in a reduction statement, it is not necessary to define the values that these variables may have while protected.

Advice to users. The following "advice to implementors" is useful for understanding the behavior of an `INDEPENDENT` loop with reduction statements. (*End of advice to users.*)

Advice to implementors. In the discussion in this section, the term "processor" means a single physical processor or a group of physical processors that together sequentially execute some or all of the iterations of an independent loop.

We describe a simple implementation mechanism that applies to commutative reduction operations. On entry to an independent loop, every executing processor allocates a private accumulator variable associated with each variable in the reduction clause on the `INDEPENDENT` directive, and initializes it to the identity element for the corresponding intrinsic reduction operator. The private accumulator variable has the same shape, type, and kind type parameter as the reduction variable.

The identity elements for the intrinsic operators are defined in Table 5.1.

Operator	Identity Element
`+`	`0`
`-`	`0`
`*`	`1`
`/`	`1`
`.AND.`	`.TRUE.`
`.OR.`	`.FALSE.`
`.EQV.`	`.TRUE.`
`.NEQV.`	`.FALSE.`

Table 5.1: Identity elements for intrinsic reduction operators.

Function	Identity element
`IAND(I,J)`	`NOT(0)` (all one-bits)
`IOR(I,J)`	`0`
`IEOR(I,J)`	`0`
`MIN(X,Y)`	the positive number of largest absolute value that has the same type and kind type parameter as the reduction variable
`MAX(X,Y)`	the negative number of largest absolute value that has the same type and kind type parameter as the reduction variable

Table 5.2: Identity elements for intrinsic reduction functions.

The intrinsic functions that may be used as reduction functions are listed, together with their identity elements, in Table 5.2.

Each processor performs a subset of the loop iterations; when it encounters a reduction statement, it updates its own accumulator variable. A processor is free to perform its loop iterations in any order; furthermore, it may start an iteration, suspend work on it, do some or all of the work of other iterations, and resume work on the suspended iteration. However, any update of a private accumulator variable occurs through the execution of a reduction statement, and reduction statements *are* executed atomically.

The final value of the reduction variable is computed by combining the private accumulator variables with the value of the reduction variable on entry to the loop, using the reduction operator. The ordering of this reduction is language-processor dependent, just as it is for the intrinsic reduction functions (`SUM`, etc.).

As an example, consider:

```
      REAL Z

      Z = 5.
!HPF$ INDEPENDENT, REDUCTION(Z)
      DO I = 1, 10
        Z = Z + I
      END DO
```

The final value of `Z` will be 5 + (1+2+3+4+5+6+7+8+9+10) = 60; the order in which the additions occur is not specified by HPF.

For a second example, here is a `SUM_SCATTER` done as an independent loop:

```
!HPF$ INDEPENDENT, REDUCTION(X)
DO I = 1, N
    X(INDEX(I)) = X(INDEX(I)) - F(I)
END DO
```

The implementation will most likely make a private copy on every processor of an accumulator array `XLOCAL` of the same type and shape as `X`, and initialize it to zero. Each iteration will subtract the value of `F(I)` from its own `XLOCAL(INDEX(I))`. To create the final result, the implementation must combine all the private accumulator arrays with the initial value of `X`. The combining operator is the same as the reduction operator, namely addition, so that the result is the sum of the initial value of `X` and the accumulator arrays. The implementation has the option of using a sparse data structure to store only the updated elements of the local accumulator.

In an MPI based implementation, the `MPI_REDUCE` function could be used for this task. (*End of advice to implementors.*)

5.2 Further Examples of INDEPENDENT Directives

```
      !HPF$ INDEPENDENT
      DO I = 1, 10
          WRITE (IOUNIT(I),100) A(I)
      END DO
  100 FORMAT ( F10.4 )
```

If `IOUNIT(I)` evaluates to a different value for every value of `I` from 1 to 10, then the loop writes to a different I/O unit (and thus a different file) on every iteration. The loop is then properly described as independent. On the other hand, if `IOUNIT(I)=5` for all `I`, then the assertion is in error and the directive is not HPF-conforming.

```
!HPF$ INDEPENDENT, NEW (J)
      DO I = 2, 100, 2
      !HPF$ INDEPENDENT, NEW(VL, VR, UL, UR)
        DO J = 2 , 100, 2
          VL = P(I,J) - P(I-1,J)
          VR = P(I+1,J) - P(I,J)
          UL = P(I,J) - P(I,J-1)
          UR = P(I,J+1) - P(I,J)
          P(I,J) = F(I,J) + P(I,J) + 0.25 * (VR - VL + UR - UL)
        END DO
      END DO
```

Without the `NEW` clause on the `J` loop, neither loop would be independent, because an interleaved execution of loop iterations might cause other values of `VL`, `VR`, `UL`, and `UR` to be used in the assignment of `P(I,J)` than those computed in the same iteration of the loop. The `NEW` clause, however, specifies that this is not true if distinct storage units are used in each iteration of the loop. Using this implementation makes iterations of the loops independent of each other. Note that there is no interference due to accesses of the array `P` because of the stride of the `DO` loop (i.e. `I` and `J` are always even, therefore `I-1`, etc., are always odd.)

When loops are nested, a reduction variable may need to be protected in an independent outer loop even though the reduction operations in which it occurs are nested inside an inner loop. Moreover, the inner loop and any intervening loops may or may not be independent.

```
!   Nested Loop Example 1.  Inner loop is sequential

      X = 10
OUTER: DO WHILE (X < 1000)   ! this loop is sequential
         !HPF$ INDEPENDENT, NEW(J), REDUCTION(X)
MIDDLE:  DO I = 1, N
INNER:     DO J = 1, M
             X = X + J
             !  Note that it would be incorrect to refer to X
             !  here, except in another reduction statement
           END DO INNER
         !  Note that it would be incorrect to refer to X
         !  here, except in another reduction statement
         END DO MIDDLE
         PRINT *, X
       END DO OUTER
```

Since the variable `X` occurs in a reduction clause for loop `MIDDLE`, it is a protected reduction variable throughout that loop, including inside the inner loop. If `INNER` had an `INDEPENDENT` directive, it would be incorrect to include `X` in a `REDUCTION` or a `NEW` clause of that directive.

The outermost loop is not independent, and so X need not and cannot be protected in that part of its range outside the middle loop.

A variable that occurs in a NEW clause must not be a reduction variable in the same or a containing loop, although it may be used as a reduction variable in a contained loop:

```
!   Nested Loop Example 2.  Outer loop NEW clause.

!HPF$ INDEPENDENT, NEW(I)
OUTER:  DO K = 1, 100
          !HPF$ INDEPENDENT, NEW (J,X)
MIDDLE:   DO I = 1, N
            X = 10
            !HPF$ INDEPENDENT, REDUCTION(X)
INNER:       DO J = 1, M
               X = X + J**2
               !  Note that it would be incorrect to refer to X
               !  here, except in another reduction statement
             END DO INNER
             Y(I) = X
          END DO MIDDLE
        END DO OUTER
```

Here, X is a protected reduction variable only in the inner loop.

```
INTEGER, DIMENSION(M) :: VECTOR

!HPF$ INDEPENDENT, REDUCTION(X, Y)
      DO I = 1, N-4
        X(I:I+4) = X(I:I+4) + A(I)   ! As many as 5 updates
        Y(VECTOR) = Y(VECTOR) + B(I,1:N)
      END DO
```

Note that the compiler, if it distributes iterations of this loop in a block-wise manner, will not need to make a private copy of the entire array X on each processor.

If a statement that has the form of a reduction statement occurs while an independent loop is active, but the updated variable is not a protected reduction variable, then the programmer is guaranteeing that no two iterations of the independent loop will update the same location. For example:

```
!HPF$ INDEPENDENT
      DO I = 1, N
        ! X is NOT a reduction variable, but
        ! I know there are no repeated values in INDX(1:N)
        ! Updates will be written directly to X(INDX(I))
        X(INDX(I)) = X(INDX(I)) + F(I)
        ! I also guarantee that the condition in the IF statement
        ! is true for at most one value of I.
        IF (A(I) > B(I)) Y = Y + 1
      END DO
```

Section 6

Extrinsic Program Units

The HPF global model of computation extends (and restricts) Fortran to provide programmers with the Fortran model of computation implementable efficiently on a wide class of hardware architectures with, in general, multiple processors, multiple memories with non-uniform access characteristics, and multiple interconnections. This model of computation presents a single logical thread of control, including Fortran's data parallel features such as array syntax and the `FORALL` statement, and data visibility defined by the scoping rules of Fortran. In particular, this model does not require the use of low-level features such as threads libraries and explicit message passing to exploit such architectures. Programmers expect their HPF compilers to generate efficient code by using HPF's features to assist in mapping data and computation to the given hardware architecture.

This chapter defines the *extrinsic* mechanism by which HPF program units may use non-HPF program units that don't use the HPF global model. It describes how to write an explicit interface for a non-HPF procedure and defines the caller's assumptions about handling distributed and replicated data at the interface. This allows the programmer to use non-HPF language facilities, for example, to descend to a lower level of abstraction to handle problems that are not efficiently addressed by HPF, to hand-tune critical kernels, or to call optimized libraries. Such an interface can also be used to interface HPF to other languages, such as C.

6.1 Overview

An HPF program may need to call a procedure implemented in a different programming model or in a different programming language. A procedure's *programming model* might provide:

- a single logical thread-of-control where *one* copy of the procedure is conceptually executing and there is a single locus of control within the program text; this model is called *global* when the underlying target hardware has (potentially) multiple processors or memories and is called *serial* when the underlying target hardware is treated as a uniprocessor (or a single node in a multiprocessor),

- multiple threads-of-control, one per processor, each thread executing the same procedure; this model is called *local* or, more generally, SPMD (Single Program, Multiple Data), or

- some other model, not discussed here, such as multiple threads-of-control, perhaps with dynamic assignment of loop iterations to processors or explicit dynamic process forking, where there is, at least initially upon invocation, *one* copy of the procedure that is conceptually executing but that may spawn multiple loci of control, possibly changing in number over time, within the program text.

A *programming language* provides a specific syntax (language features), semantics (meanings), and pragmatics (purposes). Examples of programming languages include Fortran (an ANSI and ISO standard—the most recent revision is expected to be approved by 1997), HPF (a specification of extensions and restrictions to Fortran), Fortran 77 (a previous ANSI and ISO standard), C, C++, Java, Visual Basic, and COBOL.

A program unit's language and model, when taken together, constitute its *extrinsic kind*. This *extrinsic kind* may be specified explicitly by an *extrinsic-prefix* or implicitly by the selection of a compiler and its invocation with a particular set of compiler options. Thus, one might view the compiler as providing a *host scoping unit* as defined by Fortran. For example, a program unit compiled by an HPF compiler will be of extrinsic kind `HPF`. Alternatively, its extrinsic kind may be specified explicitly by an *extrinsic-prefix* such as `EXTRINSIC(HPF)` or `EXTRINSIC(LANGUAGE='HPF',MODEL='GLOBAL')` .

6.2 Declaration of Extrinsic Program Units

6.2.1 Function and Subroutine Statements

An *extrinsic-prefix* may appear in a *function-stmt* or *subroutine-stmt* (as defined in the Fortran standard) in the same place that the keywords `RECURSIVE`, `PURE`, and `ELEMENTAL` may appear. This is specified by an extension of rule R1219 for *prefix-spec* in the Fortran standard. Rules R1217 for *function-stmt*, R1218 for *prefix*, and R1222 for *subroutine-stmt* are not changed, but are restated here for reference.

H601	*function-stmt*	**is**	[*prefix*] `FUNCTION` *function-name* ([*dummy-arg-name-list*]) [`RESULT` (*result-name*)]
H602	*subroutine-stmt*	**is**	[*prefix*] `SUBROUTINE` *subroutine-name* [([*dummy-arg-list*])]
H603	*prefix*	**is**	*prefix-spec* [*prefix-spec*] ...
H604	*prefix-spec*	**is**	*type-spec*
		or	`RECURSIVE`
		or	`PURE`
		or	`ELEMENTAL`
		or	*extrinsic-prefix*

Constraint: Within any HPF *external-subprogram*, every *internal-subprogram* must be of the same extrinsic kind as its host and any *internal-subprogram* whose extrinsic kind is not given explicitly is assumed to be of that extrinsic kind.

The definition of *characteristics of a procedure* as given in F95:12.2 is extended to include the procedure's extrinsic kind.

6.2.2 Program, Module, and Block Data Statements

An *extrinsic-prefix* may also appear at the beginning of a *program-stmt*, *module-stmt*, or *block-data-stmt*. The following syntax definition extends the Fortran 95 syntax rules R1102 for *program-stmt*, R1105 for *module-stmt*, and R1111 for *block-data-stmt*.

H605	*program-stmt*	**is**	[*extrinsic-prefix*] `PROGRAM` *program-name*
H606	*module-stmt*	**is**	[*extrinsic-prefix*] `MODULE` *module-name*
H607	*block-data-stmt*	**is**	[*extrinsic-prefix*] `BLOCK DATA` [*block-data-name*]

Constraint: Every *module-subprogram* of any HPF *module* must be of the same extrinsic kind as its host, and any *module-subprogram* whose extrinsic kind is not given explicitly is assumed to be of that extrinsic kind.

Constraint: Every *internal-subprogram* of any HPF *main-program* or *module-subprogram* must be of the same extrinsic kind as its host, and any *internal-subprogram* whose extrinsic kind is not given explicitly is assumed to be of that extrinsic kind.

6.2.3 The EXTRINSIC Prefix

H608	*extrinsic-prefix*	**is**	`EXTRINSIC (` *extrinsic-spec* `)`
H609	*extrinsic-spec*	**is**	*extrinsic-spec-arg-list*
		or	*extrinsic-kind-keyword*
H610	*extrinsic-spec-arg*	**is**	*language*
		or	*model*
		or	*external-name*
H611	*language*	**is**	[`LANGUAGE =`] *scalar-char-initialization-expr*
H612	*model*	**is**	[`MODEL =`] *scalar-char-initialization-expr*
H613	*external-name*	**is**	[`EXTERNAL_NAME =`] *scalar-char-initialization-expr*

Constraint: In an *extrinsic-spec-arg-list*, at least one of *language*, *model*, or *external-name* must be specified and none may be specified more than once.

Constraint: If *language* is specified without `LANGUAGE=`, *language* must be the first item in the *extrinsic-spec-arg-list*. If *model* is specified without `MODEL=`, *language* without `LANGUAGE=` must be the first item and *model* must be the second item in the *extrinsic-spec-arg-list*. If *external-name* is specified without `EXTERNAL_NAME=`, *language* without `LANGUAGE=` must be the first item and *model* without `MODEL=` must be the second item in the *extrinsic-spec-arg-list*.

Constraint: The forms with `LANGUAGE=`, `MODEL=`, and `EXTERNAL_NAME=` may appear in any order except as prohibited above.

Note that these rules for *extrinsic-spec-arg-list* are as if `EXTRINSIC` were a procedure with an explicit interface with a *dummy-arg-list* of `LANGUAGE, MODEL, EXTERNAL_NAME`, each of which were `OPTIONAL`.

Constraint: In *language*, values of *scalar-char-initialization-expr* may be:

- `'HPF'`, referring to the HPF language; if a *model* is not explicitly specified, the *model* is implied to be `'GLOBAL'`;
- `'FORTRAN'`, referring to the ANSI/ISO standard Fortran language; if a *model* is not explicitly specified, the *model* is implied to be `'SERIAL'`;
- `'F77'`, referring to the former ANSI/ISO standard FORTRAN 77 language; if a *model* is not explicitly specified, the *model* is implied to be `'SERIAL'`;
- `'C'`, referring to the ANSI standard C programming language; if a *model* is not explicitly specified, the *model* is implied to be `'SERIAL'`; or
- an implementation-dependent value with an implementation-dependent implied *model*.

Note that, for most implementations, `'C'` will only be allowed for *function-stmt*s and *subroutine-stmt*s occurring in an *interface-body*.

Constraint: If language is not specified it is the same as that of the host scoping unit.

Constraint: In *model*, values of *scalar-char-initialization-expr* may be:

- `'GLOBAL'`, referring to the global model,
- `'LOCAL'`, referring to the local model,
- `'SERIAL'`, referring to the serial model, or
- an implementation-dependent value.

Constraint: If *model* is not specified or implied by the specification of a language, it is the same as that of the host scoping unit.

Constraint: All *language*s and *model*s whose names begin with the three letters `HPF` are reserved for present or future definition by this specification and its successors.

Constraint: In *external-name*, the value of *scalar-char-initialization-expr* is a character string whose use is determined by the extrinsic kind. For example, an extrinsic kind may use the *external-name* to specify the name by which the procedure would be known if it were referenced by a C procedure. In such an implementation, a user would expect the compiler to perform any transformations of that name that the C compiler would perform. If *external-name* is not specified, its value is implementation-dependent.

H614 *extrinsic-kind-keyword*	**is**	`HPF`
	or	`HPF_LOCAL`
	or	`HPF_SERIAL`

Constraint: `EXTRINSIC(HPF)` is equivalent to `EXTRINSIC('HPF','GLOBAL')`. In the absence of an *extrinsic-prefix* an HPF compiler interprets a compilation unit as if it were of extrinsic kind `HPF`. Thus, for an HPF compiler, specifying `EXTRINSIC(HPF)` or `EXTRINSIC('HPF','GLOBAL')` is redundant. Such explicit specification may, however, be required for use with a compiler that supports multiple extrinsic kinds.

Constraint: `EXTRINSIC(HPF_LOCAL)` is equivalent to `EXTRINSIC('HPF','LOCAL')`. A *main-program* whose extrinsic kind is `HPF_LOCAL` behaves as if it were a subroutine of extrinsic kind `HPF_LOCAL` that is called with no arguments from a main program of extrinsic kind `HPF` whose executable part consists solely of that call.

Constraint: `EXTRINSIC(HPF_SERIAL)` is equivalent to `EXTRINSIC('HPF','SERIAL')`. A *main-program* whose extrinsic kind is `HPF_SERIAL` behaves as if it were a subroutine of extrinsic kind `HPF_SERIAL` that is called with no arguments from a main program of extrinsic kind `HPF` whose executable part consists solely of that call.

Constraint: All *extrinsic-kind-keyword*s whose names begin with the three letters `HPF` are reserved for present or future definition by this specification and its successors.

Advice to implementors.

Other *language*s or *model*s may be defined and provided by compiler vendors. Although not part of this HPF specification, they are expected to conform to the rules and spirit of HPF extrinsic kinds.

An implementation may place certain restrictions on the programmer; moreover, each extrinsic kind may call for a different set of restrictions.

For example, an implementation on a parallel processor may find it convenient to replicate scalar arguments so as to provide a copy on every processor. This is permitted so long as this process is invisible to the caller. One way to achieve this is to place a restriction on the programmer: on return from the subprogram, all the copies of this scalar argument must have the same value. This implies that if the dummy argument has `INTENT(OUT)`, then all copies must have been updated consistently by the time of subprogram return.

(*End of advice to implementors.*)

6.3 Calling HPF Extrinsic Subprograms

A call to an extrinsic procedure behaves, as observed by a calling program coded in HPF, exactly as if the subprogram were coded in HPF. If a function or subroutine called from a program unit of an HPF extrinsic kind does not have an explicit interface visible in the caller, it is assumed to have the same extrinsic kind as the caller.

In order to call a subprogram of an extrinsic kind other than that of the caller, that subprogram must have an explicit interface visible in the caller, and the subprogram is expected to behave, as observed by the caller, roughly as if it had been written as code of the same extrinsic kind as the caller. Some of the responsibility for meeting this requirement

		Extrinsic kind of the used module		
		HPF	HPF_SERIAL	HPF_LOCAL
Extrinsic kind of the using program unit	HPF	**T P D**	**T P**	**T P**
	HPF_SERIAL	**T**	**T P D**	**T**
	HPF_LOCAL	**T**	**T**	**T P D**

T = derived type definitions
P = procedures and procedure interfaces
D = data objects

Table 6.1: Entities that a using program unit is entitled to access from a module, according to the HPF extrinsic kind of each.

may rest on the compiler and some on the programmer. This interface defines the "HPF view" of the extrinsic procedure.

A called procedure that is written in a model or language other than HPF, whether or not it uses the local procedure execution model, should be declared `EXTRINSIC` within an HPF program that calls it. The `EXTRINSIC` prefix declares what sort of interface should be used when calling indicated subprograms. If there is no extrinsic specification, then the users must assume full responsibility for correctness of the implementation-dependent interface.

A *function-stmt* or *subroutine-stmt* that appears within an *interface-block* within a program unit of an HPF extrinsic kind may have an extrinsic prefix mentioning any extrinsic kind supported by the language implementation. If no *extrinsic-prefix* appears in such a *function-stmt* or *subroutine-stmt*, then it is assumed to be of the same HPF extrinsic kind as the program unit in which the interface block appears.

The procedure characteristics defined by an *interface-body* must be consistent with the procedure's definition.

The definition and rules for a procedure with an extrinsic interface lies outside the scope of HPF. However, explicit interfaces to such procedures must conform to HPF. Note that any particular HPF implementation is free to support any selection of extrinsic kinds, or none at all except for `HPF` itself, which clearly must be supported by an HPF implementation.

6.3.1 Access to Types, Procedures, and Data

In general, program units of a given extrinsic kind may use names of types, procedures, or data of another program unit of the same extrinsic kind, subject to the scoping rules of Fortran.

Use of names of types, procedures, or data of another program unit of a different extrinsic kind are subject to additional restrictions summarized in Table 6.1 and described below.

Note that, if a module X of one HPF extrinsic kind is used by a program unit Y of another HPF extrinsic kind, then only names of items in X that Y is entitled to use or invoke may be use associated; that is, either X must make private all items that Y is not entitled to use, or the `USE` statement in Y must have an `ONLY` option that lists only names of items it is entitled to use.

6.3.1.1 Types

Derived type definitions without explicitly mapped components may be thought of as "extrinsic kind neutral"; a program unit of any HPF extrinsic kind may use derived type definitions from a module of any HPF extrinsic kind. Note that an Approved Extension permits the mapping of components of derived types.

6.3.1.2 Procedures

An HPF global program or procedure may call other HPF procedures that are global, local, or serial.

An HPF local program or procedure may call only other HPF local procedures and not HPF global or serial procedures.

An HPF serial program or procedure may call only other HPF serial procedures and not HPF global or local procedures.

6.3.1.3 Data

A named `COMMON` block in any program unit of an HPF kind will be associated with the `COMMON` block, if any, of that same name in every other program unit of that same extrinsic kind; similarly for unnamed `COMMON`. (Such `COMMON` storage behaves like other declared data objects within program units of that extrinsic kind; in particular, for `HPF_LOCAL` code there will be one copy of the `COMMON` block on each processor.)

It is not permitted for any given `COMMON` block name to be used in program units of different HPF kinds within a single program; similarly, it is not permitted for unnamed `COMMON` to be used in program units of different HPF kinds within a single program.

6.3.2 The Effect of a Call

A call to an extrinsic procedure must be semantically equivalent to a call of an ordinary HPF procedure that does not remap its arguments. Thus a call to an extrinsic procedure must behave *as if* the following actions occur. The HPF technical term *as if* means that the described actions should appear to a user as if they occurred, in the order specified; an implementation may carry out any actions in any order that provide the correct user-visible effects.

1. All actions of the caller preceding the subprogram invocation should be completed before any action of the subprogram is executed; and all actions of the subprogram should be completed before any action of the caller following the subprogram invocation is executed.

2. Each actual argument is remapped, if necessary, according to the directives (explicit or implicit) in the declared interface for the extrinsic procedure. Thus, HPF mapping directives appearing in the interface are binding—the compiler must obey these directives in calling local extrinsic procedures. As in the case of non-extrinsic subprograms, actual arguments may be mapped in any way; if necessary, they are copied automatically to correctly mapped temporaries before invocation of—and copied back to the actual arguments after return from—the extrinsic procedure. Scalar dummy

arguments and scalar function results behave as if they are replicated on each processor. These mappings may, optionally, be explicit in the interface, but any other explicit mapping is not HPF conforming.

3. IN, OUT, and INOUT intent restrictions should be observed.

4. No HPF variable is modified unless it could be modified by an HPF procedure with the same explicit interface. Note that even though HPF_LOCAL and HPF_SERIAL routines are not permitted to access and modify HPF global data, other kinds of extrinsic routines may do so to the extent that an HPF procedure could.

5. When a procedure returns and the caller resumes execution, all objects accessible to the caller after the call are mapped exactly as they were before the call. In particular, the original distribution of arguments is restored, if necessary.

6. Exactly the same set of processors is visible to the HPF environment before and after the subprogram call.

Advice to implementors.

To ensure that all actions that logically precede the call are completed, multiple processors may need to be synchronized before the call is made.

If a variable accessible to the called routine has a replicated representation, then all copies may need to be updated prior to the call to contain the correct current value according to the sequential semantics of the source program.

Replicated variables, if updated in the procedure, must be updated consistently. More precisely, if a variable accessible to a procedure has a replicated representation and is updated by (one or more copies of) the procedure, then all copies of the replicated variable must have identical values when the last processor returns from the local procedure.

An implementation might check, before returning from the local subprogram, to make sure that replicated variables have been updated consistently by the subprogram. Note, however, that there is no requirement for an implementation to do so; it is merely an implementation tradeoff between speed and, for instance, debuggability.

Note that, as with a global HPF subprogram, actual arguments may be copied or remapped in any way, so long as the effect is undone on return from the subprogram.

To ensure that all actions of the procedure logically complete before execution in the caller is resumed, multiple processors may need to be synchronized after the call.

(*End of advice to implementors.*)

6.4 Examples of Extrinsic Procedures

Consider:

```
PROGRAM DUMPLING
  INTERFACE
    EXTRINSIC('HPF','LOCAL') SUBROUTINE GNOCCHI(P, L, X)
```

```
          INTERFACE
            SUBROUTINE P(Q)
              REAL Q
            END SUBROUTINE P
            EXTRINSIC('COBOL','LOCAL') SUBROUTINE L(R)
              REAL R(:,:)
            END SUBROUTINE L
          END INTERFACE
          REAL X(:)
        END SUBROUTINE GNOCCHI
        EXTRINSIC('HPF','LOCAL') SUBROUTINE POTSTICKER(Q)
          REAL Q
        END SUBROUTINE POTSTICKER
        EXTRINSIC('COBOL','LOCAL') SUBROUTINE LEBERKNOEDEL(R)
          REAL R(:,:)
        END SUBROUTINE LEBERKNOEDEL
      END INTERFACE
      ...
      CALL GNOCCHI(POTSTICKER, LEBERKNOEDEL, (/ 1.2, 3.4, 5.6 /) )
      ...
    END PROGRAM DUMPLING
```

The main program, DUMPLING, when compiled by an HPF compiler, is implicitly of extrinsic kind HPF. Interfaces are declared to three external subroutines GNOCCHI, POTSTICKER, and LEBERKNOEDEL. The first two are of extrinsic kind HPF_LOCAL and the third is of an extrinsic kind specified by the language COBOL and the local model. Now, GNOCCHI accepts two dummy procedure arguments and so interfaces must be declared for those. Because no *extrinsic-prefix* is given for dummy argument P, its extrinsic kind is that of its host scoping unit, the declaration of subroutine GNOCCHI, which has extrinsic kind HPF_LOCAL. The declaration of the corresponding actual argument POTSTICKER needs to have an explicit *extrinsic-prefix* because its host scoping unit is program DUMPLING, of extrinsic kind HPF.

Here are some more examples. In the first example, note that the declaration of the explicit size of BAGEL as 100 refers to its *global* size and not its local size:

```
    INTERFACE
      EXTRINSIC('HPF','LOCAL') FUNCTION BAGEL(X)
        REAL BAGEL(100)
        REAL X(:)
          !HPF$ DISTRIBUTE (CYCLIC) :: BAGEL, X
      END FUNCTION
    END INTERFACE
```

In the next example, note that the ALIGN statement asserts that X, Y, and Z all have the same shape:

```
    INTERFACE OPERATOR (+)
      EXTRINSIC('C','LOCAL') FUNCTION LATKES(X, Y) RESULT(Z)
        REAL, DIMENSION(:,:), INTENT(IN) :: X
        REAL, DIMENSION(:,:), INTENT(IN) :: Y
        REAL, DIMENSION(SIZE(X,1), SIZE(X,2)) :: Z
           !HPF$ ALIGN WITH X :: Y, Z
           !HPF$ DISTRIBUTE (BLOCK, BLOCK) X
      END FUNCTION
    END INTERFACE
```

In the interface block in this final example, two external procedures, one of them extrinsic and one not, are associated with the same generic procedure name, which returns a scalar of the same type as its array argument:

```
  INTERFACE KNISH
    FUNCTION RKNISH(X)                        !normal HPF interface
      REAL X(:), RKNISH
    END RKNISH
    EXTRINSIC('SISAL') FUNCTION CKNISH(X)   !extrinsic interface
      COMPLEX X(:), CKNISH
    END CKNISH
  END INTERFACE
```

Section 7

Intrinsic and Library Procedures

HPF includes Fortran's intrinsic procedures. It also adds new intrinsic procedures in two categories: system inquiry intrinsic functions and computational intrinsic functions.

In addition to the new intrinsic functions, HPF defines a library module, `HPF_LIBRARY`, that must be provided by vendors of any full HPF implementation.

This description of HPF intrinsic and library procedures follows the form and conventions of the Fortran standard. The material of Sections F95:13.1, F95:13.2, F95:13.3, F95:13.5.7, F95:13.8.1, F95:13.8.2, F95:13.9, and F95:13.10 is applicable to the HPF intrinsic and library procedures and to their descriptions in this section of the HPF document.

7.1 Notation

In the examples of this section, `T` and `F` are used to denote the logical values true and false.

7.2 System Inquiry Intrinsic Functions

In a multi-processor implementation, the processors may be arranged in an implementation-dependent multi-dimensional processor array. The system inquiry functions return values related to this underlying machine and processor configuration, including the size and shape of the underlying processor array. `NUMBER_OF_PROCESSORS` returns the total number of processors available to the program or the number of processors available to the program along a specified dimension of the processor array. `PROCESSORS_SHAPE` returns the shape of the processor array.

The Fortran definition of restricted expression is extended to permit references to the HPF system inquiry intrinsic functions. In particular, at the end of the numbered list in Section 7.1.6.2 of the Fortran standard, add:

(13) A reference to one of the system inquiry functions `NUMBER_OF_PROCESSORS` or `PROCESSORS_SHAPE`, where any argument is a restricted expression.

A variable that appears in a restricted expression in an HPF directive in the scoping unit of a module or main program must be an implied-`DO` variable or an argument in a reference to an array inquiry function, bit inquiry function, character inquiry function, kind inquiry function, or numeric inquiry function.

The values returned by the system inquiry intrinsic functions remain constant for the duration of one program execution. Thus, `NUMBER_OF_PROCESSORS` and `PROCESSORS_SHAPE` have values that are restricted expressions and may be used wherever any other Fortran restricted expression may be used. In particular, NUMBER_OF_PROCESSORS may be used in a specification expression.

The values of system inquiry functions may not occur in initialization expressions, because they may not be assumed to be constants. In particular, HPF programs may be compiled to run on machines whose configurations are not known at compile time.

Note that the system inquiry functions query the physical machine, and have nothing to do with any PROCESSORS directive that may occur. If an HPF program is running on a physical partition of a larger machine, then it is the smaller partition that actually executes the HPF program whose parameters are returned by the system inquiry functions.

Some machines may not have a "natural" shape to return as the value of the function PROCESSORS_SHAPE, for example, a machine with a tree topology. In these cases, the implementation must provide some reasonable, consistent description of the machine, such as an rank-one array of size NUMBER_OF_PROCESSORS(). The compiler will also have to arrange to map between this description and the underlying hardware processor identification mechanism.

Advice to users. SIZE(PROCESSORS_SHAPE()) returns the rank of the processor array. References to system inquiry functions may occur in array declarations and in HPF directives, as in:

```
      INTEGER, DIMENSION(SIZE(PROCESSORS_SHAPE())) :: PSHAPE
!HPF$ TEMPLATE T(100, 3*NUMBER_OF_PROCESSORS())
```

(*End of advice to users.*)

7.3 Computational Intrinsic Functions

HPF adds one new computational intrinsic function, ILEN, which computes the number of bits needed to store an integer value.

7.4 Library Procedures

The mapping inquiry subroutines and computational functions described in this section are available in the HPF library module, HPF_LIBRARY. Use of these procedures must be accompanied by an appropriate USE statement in each scoping unit in which they are used. They are not intrinsic.

7.4.1 Mapping Inquiry Subroutines

HPF provides data mapping directives that are advisory in nature. The mapping inquiry subroutines allow the program to determine the actual mapping of an array at run time. It may be especially important to know the exact mapping when an EXTRINSIC subprogram is invoked. For these reasons, HPF includes mapping inquiry subroutines which describe how an array is actually mapped onto a machine. To keep the number of routines small, the inquiry procedures are structured as subroutines with optional INTENT (OUT) arguments.

7.4.2 Bit Manipulation Functions

The HPF library includes three elemental bit-manipulation functions. `LEADZ` computes the number of leading zero bits in an integer's representation. `POPCNT` counts the number of one bits in an integer. `POPPAR` computes the parity of an integer.

7.4.3 Array Reduction Functions

HPF adds additional array reduction functions that operate in the same manner as the Fortran `SUM` and `ANY` intrinsic functions. The new reduction functions are `IALL`, `IANY`, `IPARITY`, and `PARITY`, which correspond to the commutative, associative binary operations `IAND`, `IOR`, `IEOR`, and `.NEQV.` respectively.

In the specifications of these functions, the terms "`XXX` reduction" are used, where `XXX` is one of the binary operators above. These are defined by means of an example. The `IAND` reduction of all the elements of `array` for which the corresponding element of `mask` is true is the scalar integer computed in `result` by

```
result = IAND_IDENTITY_ELEMENT
DO i_1 = LBOUND(array,1), UBOUND(array,1)
  ...
    DO i_n = LBOUND(array,n), UBOUND(array,n)
      IF ( mask(i_1,i_2,...,i_n) ) &
        result = IAND( result, array(i_1,i_2,...,i_n) )
    END DO
  ...
END DO
```

Here, n is the rank of `array` and `IAND_IDENTITY_ELEMENT` is the integer which has all bits equal to one. (The interpretation of an integer as a sequence of bits is given in F95:13.5.7.) The other three reductions are similarly defined. The identity elements for `IOR` and `IEOR` are zero. The identity element for `.NEQV.` is `.FALSE.`

7.4.4 Array Combining Scatter Functions

The `XXX_SCATTER` functions are generalized array reduction functions in which an arbitrary subset of the elements of an array can be combined to produce an element of the result; the subset corresponding to the result's elements are nonoverlapping. Each of the eleven reduction operation in the language corresponds to one of the scatter functions, while `COPY_SCATTER` supports overwriting an existing value with any one of the values in the corresponding subset. The way that elements of the source array are associated with the elements of the result is described in this section; the method of combining their values is described in the specifications of the individual functions in Section 7.7.

These functions have the general form

```
XXX_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
```

except in the special cases noted below. The allowed values of `XXX` are `ALL`, `ANY`, `COPY`, `COUNT`, `IALL`, `IANY`, `IPARITY`, `MAXVAL`, `MINVAL`, `PARITY`, `PRODUCT`, and `SUM`. `ARRAY`, `MASK`, and all the `INDX` arrays are conformable. The `INDX` arrays are integer, and the number of `INDX` arguments must equal the rank of `BASE`. The argument `MASK` is logical, and it is optional. Except for `COUNT_SCATTER`, `ARRAY` and `BASE` are arrays of the same type. For `COUNT_SCATTER`, `ARRAY` is of type logical and `BASE` is of type integer. (For `ALL_SCATTER`, `ANY_SCATTER`, `COUNT_SCATTER`,and `PARITY_SCATTER`, the `ARRAY` argument must be logical. These functions do not have an optional `MASK` argument. To conform with the conventions of the Fortran standard, the required `ARRAY` argument to these functions is called `MASK` in their specifications in Section 7.7.) In all cases the result array is an array with the same type, kind type parameter, and shape as `BASE`.

For every element a in `ARRAY` there is a corresponding element in each of the `INDX` arrays, since they all have the same shape as `ARRAY`. For each $j = 1, 2, \ldots, n$, where n is the rank of `BASE`, let s_j be the value of the element of `INDXj` that corresponds to element a in `ARRAY`. These indices determine the element of the result that is affected by element a of `ARRAY`. For each of the indices s_j, let the corresponding index for `BASE` be given by $b_j = s_j +$ `LBOUND(BASE, j) - 1`.

The integers b_j, $j = 1, \ldots, n$, form a subscript selecting an element of `BASE`: `BASE(`$b_1, b_2, \ldots, b_n$`)`. Because `BASE` and the result are conformable, for each element of `BASE` there is a corresponding element of the result.

Thus the `INDX` arrays establish a mapping from all the elements of `ARRAY` onto selected elements of the result and `BASE`. Viewed in the other direction, this mapping associates with each element b of `BASE` a set S of elements from `ARRAY`.

If S is empty, then the element of the result corresponding to the element b of `BASE` has the same value as b.

If S is non-empty, then the elements of S will be combined with element b to produce an element of the result. The detailed specifications of the scatter functions describe the particular means of combining these values. As an example, for `SUM_SCATTER`, if the elements of S are $a_1, \ldots, a_m$, then the element of the result corresponding to the element b of `BASE` is the result of evaluating `SUM((/`$a_1, a_2, \ldots, a_m, b$`/))`.

Note that the elements of the `INDX` arrays must be non-negative, and that `INDXj` may not exceed `SIZE(BASE, j)`. The result computed is not affected by the declared upper or lower bounds on indices of `BASE`; it depends only on the shape of `BASE`.

Note that, since a scalar is conformable with any array, a scalar may be used in place of an `INDX` array, in which case one hyperplane of the result is selected. See the example below.

If the optional, final `MASK` argument is present, then only the elements of `ARRAY` in positions for which `MASK` is true participate in the operation. All other elements of `ARRAY` and of the `INDX` arrays are ignored and cannot have any influence on any element of the result.

For example, if

$$\texttt{A}\text{ is the array }\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}; \qquad \texttt{B}\text{ is the array }\begin{bmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \\ -7 & -8 & -9 \end{bmatrix};$$

$$\texttt{I1}\text{ is the array }\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 1 \\ 3 & 2 & 1 \end{bmatrix}; \qquad \texttt{I2}\text{ is the array }\begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

then

SUM_SCATTER(A, B, I1, I2) is $\begin{bmatrix} 14 & 6 & 0 \\ 8 & -5 & -6 \\ 0 & -8 & -9 \end{bmatrix}$;

SUM_SCATTER(A, B, 2, I2) is $\begin{bmatrix} -1 & -2 & -3 \\ 30 & 3 & -3 \\ -7 & -8 & -9 \end{bmatrix}$;

SUM_SCATTER(A, B, I1, 2) is $\begin{bmatrix} -1 & 24 & -3 \\ -4 & 7 & -6 \\ -7 & -1 & -9 \end{bmatrix}$;

SUM_SCATTER(A, B, 2, 2) is $\begin{bmatrix} -1 & -2 & -3 \\ -4 & 40 & -6 \\ -7 & -8 & -9 \end{bmatrix}$.

If A is the array $\begin{bmatrix} 10 & 20 & 30 & 40 & -10 \end{bmatrix}$, B is the array $\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$, and IND is the array $\begin{bmatrix} 3 & 2 & 2 & 1 & 1 \end{bmatrix}$, then SUM_SCATTER(A, B, IND, MASK=(A .GT. 0)) is $\begin{bmatrix} 41 & 52 & 13 & 4 \end{bmatrix}$.

7.4.5 Array Prefix and Suffix Functions

In a scan of a vector, each element of the result is a function of the elements of the vector that precede it (for a prefix scan) or that follow it (for a suffix scan). These functions provide scan operations on arrays and subarrays. The functions have the general form

```
  XXX_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
  XXX_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
```

except in the special cases noted below. The allowed values of XXX are ALL, ANY, COPY, COUNT, IALL, IANY, IPARITY, MAXVAL, MINVAL, PARITY, PRODUCT, and SUM.

When comments below apply to both prefix and suffix forms of the routines, we will refer to them as YYYFIX functions.

The arguments DIM, MASK, SEGMENT, and EXCLUSIVE are optional. The COPY_YYYFIX functions do not have MASK or EXCLUSIVE arguments. The ALL_YYYFIX, ANY_YYYFIX, COUNT_-YYYFIX, and PARITY_YYYFIX functions do not have MASK arguments. Their ARRAY argument must be of type logical; it is denoted MASK in their specifications in Section 7.7.

The arguments MASK and SEGMENT must be of type logical. SEGMENT must have the same shape as ARRAY. MASK must be conformable with ARRAY. EXCLUSIVE is a logical scalar. DIM is a scalar integer between one and the rank of ARRAY.

Result Value. The result has the same shape as ARRAY, and, with the exception of COUNT_YYYFIX, the same type and kind type parameter as ARRAY. (The result of COUNT_YYYFIX is default integer.)

In every case, every element of the result is determined by the values of certain selected elements of ARRAY in a way that is specific to the particular function and is described in its specification. The optional arguments affect the selection of elements of ARRAY for each element of the result; the selected elements of ARRAY are said to contribute to the result element. This section describes fully which elements of ARRAY contribute to a given element of the result.

If no elements of ARRAY are selected for a given element of the result, that result element is set to a default value that is specific to the particular function and is described in its specification.

For any given element r of the result, let a be the corresponding element of ARRAY. Every element of ARRAY contributes to r unless disqualified by one of the following rules.

1. If the function is XXX_PREFIX, no element that follows a in the array element ordering of ARRAY contributes to r. If the function is XXX_SUFFIX, no element that precedes a in the array element ordering of ARRAY contributes to r.
2. If the DIM argument is provided, an element z of ARRAY does not contribute to r unless all its indices, excepting only the index for dimension DIM, are the same as the corresponding indices of a. (It follows that if the DIM argument is omitted, then ARRAY, MASK, and SEGMENT are processed in array element order, as if temporarily regarded as rank-one arrays. If the DIM argument is present, then a family of completely independent scan operations are carried out along the selected dimension of ARRAY.)
3. If the MASK argument is provided, an element z of ARRAY contributes to r only if the element of MASK corresponding to z is true. (It follows that array elements corresponding to positions where the MASK is false do not contribute anywhere to the result. However, the result is nevertheless defined at all positions, even positions where the MASK is false.)
4. If the SEGMENT argument is provided, an element z of ARRAY does not contribute if there is some intermediate element w of ARRAY, possibly z itself, with all of the following properties:
   (a) If the function is XXX_PREFIX, w does not precede z but does precede a in the array element ordering; if the function is XXX_SUFFIX, w does not follow z but does follow a in the array element ordering;
   (b) If the DIM argument is present, all the indices of w, excepting only the index for dimension DIM, are the same as the corresponding indices of a; and
   (c) The element of SEGMENT corresponding to w does not have the same value as the element of SEGMENT corresponding to a. (In other words, z can contribute only if there is an unbroken string of SEGMENT values, all alike, extending from z through a.)
5. If the EXCLUSIVE argument is provided and is true, then a itself does not contribute to r.

These general rules lead to the following important cases:

Case (i): If ARRAY has rank one, element i of the result of XXX_PREFIX(ARRAY) is determined by the first i elements of ARRAY; element SIZE(ARRAY) $- i + 1$ of the result of XXX_SUFFIX(ARRAY) is determined by the last i elements of ARRAY.

Case (ii): If ARRAY has rank greater than one, then each element of the result of XXX_PREFIX(ARRAY) has a value determined by the corresponding element a of the ARRAY and all elements of ARRAY that precede a in array element order. For `XXX_SUFFIX`, a is determined by the elements of `ARRAY` that correspond to or follow a in array element order.

Case (iii): Each element of the result of `XXX_PREFIX(ARRAY,MASK=MASK)` is determined by selected elements of `ARRAY`, namely the corresponding element a of the `ARRAY` and all elements of `ARRAY` that precede a in array element order, but an element of `ARRAY` may contribute to the result only if the corresponding element of `MASK` is true. If this restriction results in selecting no array elements to contribute to some element of the result, then that element of the result is set to the default value for the given function.

Case (iv): Each element of the result of `XXX_PREFIX(ARRAY,DIM=DIM)` is determined by selected elements of `ARRAY`, namely the corresponding element a of the `ARRAY` and all elements of `ARRAY` that precede a along dimension `DIM`; for example, in `SUM_PREFIX(A(1:N,1:N), DIM=2)`, result element (i_1, i_2) could be computed as `SUM(A(`i_1`,1 :` i_2`))`. More generally, in `SUM_PREFIX(ARRAY, DIM)`, result element $i_1, i_2, \ldots, i_{DIM}, \ldots, i_n$ could be computed as `SUM(ARRAY(` $i_1, i_2, \ldots, :i_{DIM}, \ldots, i_n$ `))` . (Note the colon before i_{DIM} in that last expression.)

Case (v): If `ARRAY` has rank one, then element i of the result of `XXX_PREFIX(ARRAY, EXCLUSIVE=.TRUE.)` is determined by the first $i-1$ elements of `ARRAY`.

Case (vi): The options may be used in any combination.

Advice to users. A new segment begins at every *transition* from false to true or true to false; thus a segment is indicated by a maximal contiguous subsequence of like logical values:

```
(/T,T,T,F,T,F,F,F,T,F,F,T/)
  ----- - - ----- - --- -    seven segments
```

(*End of advice to users.*)

Rationale.

One existing library delimits the segments by indicating the *start* of each segment. Another delimits the segments by indicating the *stop* of each segment. Each method has its advantages. There is also the question of whether this convention should change when performing a suffix rather than a prefix. HPF adopts the symmetric representation above. The main advantages of this representation are:

(A) It is symmetrical, in that the same segment specifier may be meaningfully used for prefix and suffix without changing its interpretation (start versus stop).

(B) The start-bit or stop-bit representation is easily converted to this form by using `PARITY_PREFIX` or `PARITY_SUFFIX`. These might be standard idioms for a compiler to recognize:

```
SUM_PREFIX(FOO,SEGMENT=PARITY_PREFIX(START_BITS))
SUM_PREFIX(FOO,SEGMENT=PARITY_SUFFIX(STOP_BITS))
SUM_SUFFIX(FOO,SEGMENT=PARITY_SUFFIX(START_BITS))
SUM_SUFFIX(FOO,SEGMENT=PARITY_PREFIX(STOP_BITS))
```

(*End of rationale.*)

Examples. The examples below illustrate all possible combinations of optional arguments for `SUM_PREFIX`. The default value for `SUM_YYYFIX` is zero.

Case (i): `SUM_PREFIX((/1,3,5,7/))` is $\begin{bmatrix} 1 & 4 & 9 & 16 \end{bmatrix}$.

Case (ii): If `B` is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$,

then `SUM_PREFIX(B)` is the array $\begin{bmatrix} 1 & 14 & 30 \\ 5 & 19 & 36 \\ 12 & 27 & 45 \end{bmatrix}$.

Case (iii): If `A` is the array $\begin{bmatrix} 3 & 5 & \text{-2} & \text{-1} & 7 & 4 & 8 \end{bmatrix}$,

then `SUM_PREFIX(A, MASK = A .LT. 6)` is $\begin{bmatrix} 3 & 8 & 6 & 5 & 5 & 9 & 9 \end{bmatrix}$.

Case (iv): If `B` is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, then `SUM_PREFIX(B, DIM=1)` is the array $\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 12 & 15 & 18 \end{bmatrix}$ and `SUM_PREFIX(B, DIM=2)` is the array $\begin{bmatrix} 1 & 3 & 6 \\ 4 & 9 & 15 \\ 7 & 15 & 24 \end{bmatrix}$.

Case (v): `SUM_PREFIX((/1,3,5,7/), EXCLUSIVE=.TRUE.)` is $\begin{bmatrix} 0 & 1 & 4 & 9 \end{bmatrix}$.

Case (vi): If `B` is the array $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{bmatrix}$, `M` is the array $\begin{bmatrix} T & T & T & T & T \\ F & F & T & T & T \\ T & F & T & F & F \end{bmatrix}$, and `S` is the array $\begin{bmatrix} T & T & F & F & F \\ F & T & T & F & F \\ T & T & T & T & T \end{bmatrix}$, then:

`SUM_PREFIX(B, DIM=2, MASK=M, SEGMENT=S, EXCLUSIVE=.TRUE.)` is
$\begin{bmatrix} 0 & 1 & 0 & 3 & 7 \\ 0 & 0 & 0 & 0 & 9 \\ 0 & 11 & 11 & 24 & 24 \end{bmatrix}$.

`SUM_PREFIX(B, DIM=2, MASK=M, SEGMENT=S, EXCLUSIVE=.FALSE.)` is
$\begin{bmatrix} 1 & 3 & 3 & 7 & 12 \\ 0 & 0 & 8 & 9 & 19 \\ 11 & 11 & 24 & 24 & 24 \end{bmatrix}$.

`SUM_PREFIX(B, DIM=2, MASK=M, EXCLUSIVE=.TRUE.)` is $\begin{bmatrix} 0 & 1 & 3 & 6 & 10 \\ 0 & 0 & 0 & 8 & 17 \\ 0 & 11 & 11 & 24 & 24 \end{bmatrix}$.

SUM_PREFIX(B, DIM=2, MASK=M, EXCLUSIVE=.FALSE.) is $\begin{bmatrix} 1 & 3 & 6 & 10 & 15 \\ 0 & 0 & 8 & 17 & 27 \\ 11 & 11 & 24 & 24 & 24 \end{bmatrix}$.

SUM_PREFIX(B, DIM=2, SEGMENT=S, EXCLUSIVE=.TRUE.) is $\begin{bmatrix} 0 & 1 & 0 & 3 & 7 \\ 0 & 0 & 7 & 0 & 9 \\ 0 & 11 & 23 & 36 & 50 \end{bmatrix}$.

SUM_PREFIX(B, DIM=2, SEGMENT=S, EXCLUSIVE=.FALSE.) is
$$\begin{bmatrix} 1 & 3 & 3 & 7 & 12 \\ 6 & 7 & 15 & 9 & 19 \\ 11 & 23 & 36 & 50 & 65 \end{bmatrix}.$$

SUM_PREFIX(B, DIM=2, EXCLUSIVE=.TRUE.) is $\begin{bmatrix} 0 & 1 & 3 & 6 & 10 \\ 0 & 6 & 13 & 21 & 30 \\ 0 & 11 & 23 & 36 & 50 \end{bmatrix}$.

SUM_PREFIX(B, DIM=2, EXCLUSIVE=.FALSE.) is $\begin{bmatrix} 1 & 3 & 6 & 10 & 15 \\ 6 & 13 & 21 & 30 & 40 \\ 11 & 23 & 36 & 50 & 65 \end{bmatrix}$.

SUM_PREFIX(B, MASK=M, SEGMENT=S, EXCLUSIVE=.TRUE.) is $\begin{bmatrix} 0 & 11 & 0 & 0 & 0 \\ 0 & 13 & 0 & 4 & 5 \\ 0 & 13 & 8 & 0 & 0 \end{bmatrix}$.

SUM_PREFIX(B, MASK=M, SEGMENT=S, EXCLUSIVE=.FALSE.) is
$$\begin{bmatrix} 1 & 13 & 3 & 4 & 5 \\ 0 & 13 & 8 & 13 & 15 \\ 11 & 13 & 21 & 0 & 0 \end{bmatrix}.$$

SUM_PREFIX(B, MASK=M, EXCLUSIVE=.TRUE.) is $\begin{bmatrix} 0 & 12 & 14 & 38 & 51 \\ 1 & 14 & 17 & 42 & 56 \\ 1 & 14 & 25 & 51 & 66 \end{bmatrix}$.

SUM_PREFIX(B, MASK=M, EXCLUSIVE=.FALSE.) is $\begin{bmatrix} 1 & 14 & 17 & 42 & 56 \\ 1 & 14 & 25 & 51 & 66 \\ 12 & 14 & 38 & 51 & 66 \end{bmatrix}$.

SUM_PREFIX(B, SEGMENT=S, EXCLUSIVE=.TRUE.) is $\begin{bmatrix} 0 & 11 & 0 & 0 & 0 \\ 0 & 13 & 0 & 4 & 5 \\ 0 & 20 & 8 & 0 & 0 \end{bmatrix}$.

SUM_PREFIX(B, SEGMENT=S, EXCLUSIVE=.FALSE.) is $\begin{bmatrix} 1 & 13 & 3 & 4 & 5 \\ 6 & 20 & 8 & 13 & 15 \\ 11 & 32 & 21 & 14 & 15 \end{bmatrix}$.

SUM_PREFIX(B, EXCLUSIVE=.TRUE.) is $\begin{bmatrix} 0 & 18 & 39 & 63 & 90 \\ 1 & 20 & 42 & 67 & 95 \\ 7 & 27 & 50 & 76 & 105 \end{bmatrix}$.

SUM_PREFIX(B, EXCLUSIVE=.FALSE.) is $\begin{bmatrix} 1 & 20 & 42 & 67 & 95 \\ 7 & 27 & 50 & 76 & 105 \\ 18 & 39 & 63 & 90 & 120 \end{bmatrix}$.

7.4.6 Array Sorting Functions

HPF includes procedures for sorting multidimensional arrays. The `SORT_UP` and `SORT_DOWN` functions return sorted arrays; the `GRADE_UP` and `GRADE_DOWN` functions return sorting permutations. An array can be sorted along a given axis, or the whole array may be viewed as a sequence in array element order. The grade functions use stable sorts, allowing for convenient sorting of structures by major and minor keys.

7.5 Generic Intrinsic and Library Procedures

For all of the intrinsic and library procedures, the arguments shown are the names that must be used for keywords when using the keyword form for actual arguments. Many of the argument keywords have names that are indicative of their usage, as is the case in Fortran. See Section F95:13.11.

7.5.1 System Inquiry Intrinsic Functions

`NUMBER_OF_PROCESSORS(DIM)` Optional `DIM`	The number of executing processors
`PROCESSORS_SHAPE()`	The shape of the executing processor array

7.5.2 Mapping Inquiry Subroutines

```
HPF_ALIGNMENT(ALIGNEE, LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, &
          NCOPIES)
     Optional LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, NCOPIES
HPF_DISTRIBUTION(DISTRIBUTEE, AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, &
          PROCESSORS_SHAPE)
     Optional AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE
HPF_TEMPLATE(ALIGNEE, TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO, &
          NUMBER_ALIGNED)
     Optional TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO,
          NUMBER_ALIGNED
```

7.5.3 Bit Manipulation Functions

`ILEN(I)`	Bit length (intrinsic)
`LEADZ(I)`	Leading zeros
`POPCNT(I)`	Number of one bits
`POPPAR(I)`	Parity

7.5.4 Array Reduction Functions

IALL(ARRAY, DIM, MASK)	Bitwise logical AND reduction
Optional DIM, MASK	
IANY(ARRAY, DIM, MASK)	Bitwise logical OR reduction
Optional DIM, MASK	
IPARITY(ARRAY, DIM, MASK)	Bitwise logical EOR reduction
Optional DIM, MASK	
PARITY(MASK, DIM)	Logical EOR reduction
Optional DIM	

7.5.5 Array Combining Scatter Functions

ALL_SCATTER(MASK, BASE, INDX1 ..., INDXn)
ANY_SCATTER(MASK, BASE, INDX1, ..., INDXn)
COPY_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
    Optional MASK
COUNT_SCATTER(MASK, BASE, INDX1, ..., INDXn)
IALL_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
    Optional MASK
IANY_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
    Optional MASK
IPARITY_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
    Optional MASK
MAXVAL_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
    Optional MASK
MINVAL_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
    Optional MASK
PARITY_SCATTER(MASK, BASE, INDX1, ..., INDXn)
PRODUCT_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
    Optional MASK
SUM_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
    Optional MASK

7.5.6 Array Prefix and Suffix Functions

ALL_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
    Optional DIM, SEGMENT, EXCLUSIVE
ALL_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
    Optional DIM, SEGMENT, EXCLUSIVE
ANY_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
    Optional DIM, SEGMENT, EXCLUSIVE
ANY_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
    Optional DIM, SEGMENT, EXCLUSIVE

```
COPY_PREFIX(ARRAY, DIM, SEGMENT)
     Optional DIM, SEGMENT
COPY_SUFFIX(ARRAY, DIM, SEGMENT)
     Optional DIM, SEGMENT
COUNT_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
     Optional DIM, SEGMENT, EXCLUSIVE
COUNT_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
     Optional DIM, SEGMENT, EXCLUSIVE
IALL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
IALL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
IANY_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
IANY_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
IPARITY_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
IPARITY_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
MAXVAL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
MAXVAL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
MINVAL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
MINVAL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
PARITY_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
     Optional DIM, SEGMENT, EXCLUSIVE
PARITY_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
     Optional DIM, SEGMENT, EXCLUSIVE
PRODUCT_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
PRODUCT_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
SUM_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
SUM_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
     Optional DIM, MASK, SEGMENT, EXCLUSIVE
```

7.5.7 Array Sort Functions

`GRADE_DOWN(ARRAY,DIM)` Optional `DIM`	Permutation that sorts into descending order
`GRADE_UP(ARRAY,DIM)` Optional `DIM`	Permutation that sorts into ascending order
`SORT_DOWN(ARRAY,DIM)` Optional `DIM`	Sort into descending order
`SORT_UP(ARRAY,DIM)` Optional `DIM`	Sort into ascending order

7.6 Specifications of Intrinsic Procedures

ILEN(I)

Description. Returns one less than the length, in bits, of the two's-complement representation of an integer.

Class. Elemental function.

Argument. `I` must be of type integer.

Result Type and Type Parameter. Same as `I`.

Result Value. If `I` is nonnegative, `ILEN(I)` has the value $\lceil \log_2(\texttt{I+1}) \rceil$; if `I` is negative, `ILEN(I)` has the value $\lceil \log_2(\texttt{-I}) \rceil$.

Examples. `ILEN(4) = 3`. `ILEN(-4) = 2`. `2**ILEN(N-1)` rounds `N` up to a power of 2 (for `N` > 0), whereas `2**(ILEN(N)-1)` rounds `N` down to a power of 2. Compare with `LEADZ`.

The value returned is one *less* than the length of the two's-complement representation of `I`, as the following explains. The shortest two's-complement representation of 4 is 0100. The leading zero is the required sign bit. In 3-bit two's complement, 100 represents -4.

NUMBER_OF_PROCESSORS(DIM)

Optional Argument. `DIM`

Description. Returns the total number of processors available to the program or the number of processors available to the program along a specified dimension of the processor array.

Class. System inquiry function.

Arguments.

`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$ where n is the rank of the processor array.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to the extent of dimension `DIM` of the implementation-dependent hardware processor array or, if `DIM` is absent, the total number of elements of the implementation-dependent hardware processor array. The result is always greater than zero.

Examples. For a computer with 8192 processors arranged in a 128 by 64 rectangular grid, the value of `NUMBER_OF_PROCESSORS()` is 8192; the value of `NUMBER_OF_PROCES-SORS(DIM=1)` is 128; and the value of `NUMBER_OF_PROCESSORS(DIM=2)` is 64. For a single-processor workstation, the value of `NUMBER_OF_PROCESSORS()` is 1; since the rank of a scalar processor array is zero, no `DIM` argument may be used.

PROCESSORS_SHAPE()

Description. Returns the shape of the implementation-dependent processor array.

Class. System inquiry function.

Arguments. None

Result Type, Type Parameter, and Shape. The result is a default integer array of rank one whose size is equal to the rank of the implementation-dependent processor array.

Result Value. The value of the result is the shape of the implementation-dependent processor array.

Example. In a computer with 2048 processors arranged in a hypercube, the value of `PROCESSORS_SHAPE()` is [2,2,2,2,2,2,2,2,2,2,2]. In a computer with 8192 processors arranged in a 128 by 64 rectangular grid, the value of `PROCESSORS_SHAPE()` is [128,64]. For a single processor workstation, the value of `PROCESSORS_SHAPE()` is [] (the size-zero array of rank one).

7.7 Specifications of Library Procedures

ALL_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented logical AND scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`.

`SEGMENT` (optional)	must be of type logical and must have the same shape as `MASK`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `MASK`.

Result Value. Element r of the result has the value `ALL((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `MASK` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `ALL_PREFIX( (/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/) )` is $\left[\begin{array}{ccccc} \mathrm{T} & \mathrm{F} & \mathrm{F} & \mathrm{T} & \mathrm{T} \end{array}\right]$.

ALL_SCATTER(MASK,BASE,INDX1, ..., INDXn)

Description. Scatters elements of `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. An element of the result is true if and only if the corresponding element of `BASE` and all elements of `MASK` scattered to that position are true.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`BASE`	must be of type logical with the same kind type parameter as `MASK`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `MASK`. The number of `INDX` arguments must be equal to the rank of `BASE`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `ALL( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `MASK` associated with b as described in Section 7.4.4.

Example. `ALL_SCATTER( (/T, T, T, F/), (/T, T, T/), (/1, 1, 2, 2/) )` is $\left[\begin{array}{ccc} \mathrm{T} & \mathrm{F} & \mathrm{T} \end{array}\right]$.

ALL_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented logical AND scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `MASK`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `MASK`.

Result Value. Element r of the result has the value `ALL((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `MASK` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `ALL_SUFFIX( (/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} \mathrm{F} & \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{T} \end{bmatrix}$.

ANY_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented logical OR scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `MASK`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `MASK`.

Result Value. Element r of the result has the value `ANY((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `MASK` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `ANY_PREFIX( (/F,T,F,F,F/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{F} \end{bmatrix}$.

ANY_SCATTER(MASK,BASE,INDX1, ..., INDXn)

Description. Scatters elements of MASK to positions of the result indicated by index arrays INDX1, ..., INDXn. An element of the result is true if and only if the corresponding element of BASE or any element of MASK scattered to that position is true.

Class. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
BASE	must be of type logical with the same kind type parameter as MASK. It must not be scalar.
INDX1,...,INDXn	must be of type integer and conformable with MASK. The number of INDX arguments must be equal to the rank of BASE.

Result Type, Type Parameter, and Shape. Same as BASE.

Result Value. The element of the result corresponding to the element b of BASE has the value ANY((/$a_1, a_2, ..., a_m, b$/)), where $(a_1, \ldots, a_m)$ are the elements of MASK associated with b as described in Section 7.4.4.

Example. ANY_SCATTER((/T, F, F, F/), (/F, F, T/), (/1, 1, 2, 2/)) is $\begin{bmatrix} \text{T} & \text{F} & \text{T} \end{bmatrix}$.

ANY_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. DIM, SEGMENT, EXCLUSIVE

Description. Computes a reverse, segmented logical OR scan along dimension DIM of MASK.

Class. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of MASK.
SEGMENT (optional)	must be of type logical and must have the same shape as MASK.
EXCLUSIVE (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as MASK.

Result Value. Element r of the result has the value ANY((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of MASK selected to contribute to r by the rules stated in Section 7.4.5.

Example. ANY_SUFFIX((/F,T,F,F,F/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} \text{T} & \text{T} & \text{F} & \text{F} & \text{F} \end{bmatrix}$.

COPY_PREFIX(ARRAY, DIM, SEGMENT)

Optional Arguments. `DIM`, `SEGMENT`

Description. Computes a segmented copy scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	may be of any type. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value a_1 where $(a_1, \ldots, a_m)$ is the set, in array element order, of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5. Note that this set is never empty.

Example. `COPY_PREFIX( (/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 1 & 1 & 1 & 4 & 4 \end{bmatrix}$.

COPY_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

Optional Argument. `MASK`

Description. Scatters elements of `ARRAY` selected by `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. Each element of the result is equal to one of the elements of `ARRAY` scattered to that position or, if there is none, to the corresponding element of `BASE`.

Class. Transformational function.

Arguments.

`ARRAY`	may be of any type. It must not be scalar.
`BASE`	must be of the same type and kind type parameter as `ARRAY`.
`INDX1,...,INDXn`	must be of type integer and conformable with `ARRAY`. The number of`INDX` arguments must be equal to the rank of `BASE`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. Let S be the set of elements of `ARRAY` associated with element b of `BASE` as described in Secion 7.4.4.

If S is empty, then the element of the result corresponding to the element b of `BASE` has the same value as b.

If S is non-empty, then the element of the result corresponding to the element b of `BASE` is the result of choosing one element from S. HPF does not specify how the choice is to be made; the mechanism is implementation dependent.

Example. `COPY_SCATTER((/1, 2, 3, 4/), (/7, 8, 9/), (/1, 1, 2, 2/))` is `[`x`, `y`, 9]`, where x is a member of the set $\{1,2\}$ and y is a member of the set $\{3,4\}$.

COPY_SUFFIX(ARRAY, DIM, SEGMENT)

Optional Arguments. `DIM`, `SEGMENT`

Description. Computes a reverse, segmented copy scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	may be of any type. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value a_m where $(a_1, \ldots, a_m)$ is the set, in array element order, of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5. Note that this set is never empty.

Example. `COPY_SUFFIX( (/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 3 & 3 & 3 & 5 & 5 \end{bmatrix}$.

COUNT_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented `COUNT` scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.

`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `MASK`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. The result is of type default integer and of the same shape as `MASK`.

Result Value. Element r of the result has the value `COUNT((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `MASK` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `COUNT_PREFIX( (/F,T,T,T,T/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 0 & 1 & 2 & 1 & 2 \end{bmatrix}$.

COUNT_SCATTER(MASK,BASE,INDX1, ..., INDXn)

Description. Scatters elements of `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. Each element of the result is the sum of the corresponding element of `BASE` and the number of true elements of `MASK` scattered to that position.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`BASE`	must be of type integer. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `MASK`. The number of `INDX` arguments must be equal to the rank of `BASE`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value b `+ COUNT( (/`$a_1, a_2, ..., a_m$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `MASK` associated with b as described in Section 7.4.4.

Example. `COUNT_SCATTER((/T, T, T, F/),(/1, -1, 0/),(/1, 1, 2, 2/))` is $\begin{bmatrix} \texttt{3} & \texttt{0} & \texttt{0} \end{bmatrix}$.

COUNT_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented `COUNT` scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `MASK`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. The result is of type default integer and of the same shape as `MASK`.

Result Value. Element r of the result has the value `COUNT((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `MASK` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `COUNT_SUFFIX( (/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 2 & 1 & 1 & 2 & 1 \end{bmatrix}$.

GRADE_DOWN(ARRAY,DIM)

Optional Argument. `DIM`

Description. Produces a permutation of the indices of an array, expressed as one-based coordinates, and sorted by descending array element values.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or character. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result is of type default integer. If `DIM` is present, the result has the same shape as `ARRAY`. If `DIM` is absent, the result has shape `(/ SIZE(SHAPE(ARRAY)), SIZE(ARRAY) /)`.

Result Value.

Case (i): The result of

```
S = GRADE_DOWN(ARRAY)
      + SPREAD(LBOUND(ARRAY),DIM=2, NCOPIES=SIZE(ARRAY))-1
```

is such that if one computes the rank-one array `B` of size `SIZE(ARRAY)` by

`FORALL (K=1:SIZE(B)) B(K)=ARRAY(S(1,K),S(2,K),...,S(N,K))`
where `N` has the value `SIZE(SHAPE(ARRAY))`, then `B` is sorted in descending order; moreover, all of the columns of `S` are distinct, that is, if $j \neq m$ then `ALL(S(:,`j`) .EQ. S(:,`m`))` will be false. The sort is stable; if $j \leq m$ and `B(`j`)` $=$ `B(`m`)`, then `ARRAY(S(1,`j`),S(2,`j`),...,S(`n,j`))` precedes `ARRAY(S(1,`m`),S(2,`m`),...,S(`n,m`))` in the array element ordering of `ARRAY`. The collating sequence for an array of type `CHARACTER` is that used by the Fortran intrinsic functions, namely ASCII.

Case (ii): The result of
`R = GRADE_DOWN(ARRAY, DIM=K) + LBOUND(ARRAY, DIM=K) -1`
has the property that if one computes the array
`B(`$i_1, i_2, \ldots, i_k, \ldots, i_n$`) =`
`ARRAY(`$i_1, i_2, \ldots,$ `R(`$i_1, i_2, \ldots, i_k, \ldots, i_n$`)`$, \ldots, i_n$ `)`
then for all $i_1, i_2, \ldots,$ (omit i_k)$, \ldots, i_n$, the vector `B(`$i_1, i_2, \ldots, :, \ldots, i_n$`)` is sorted in descending order; moreover, `R(`$i_1, i_2, \ldots, :, \ldots, i_n$`)` is a permutation of all the integers between 1 and `SIZE(ARRAY, DIM=K)`, inclusive. The sort is stable; that is, if $j \leq m$ and
`B(`$i_1, i_2, \ldots, j, \ldots, i_n$`)` $=$ `B(`$i_1, i_2, \ldots, m, \ldots, i_n$`)`, then
`R(`$i_1, i_2, \ldots, j, \ldots, i_n$`)` $\leq$ `R(`$i_1, i_2, \ldots, m, \ldots, i_n$`)`. The collating sequence for an array of type `CHARACTER` is that used by the Fortran intrinsic functions, namely ASCII.

Examples.

Case (i): `GRADE_DOWN( (/30, 20, 30, 40, -10/) )` is a rank two array of shape $\begin{bmatrix} 1 & 5 \end{bmatrix}$ with the value $\begin{bmatrix} 4 & 1 & 3 & 2 & 5 \end{bmatrix}$. (To produce a rank-one result, the optional `DIM = 1` argument must be used.)

If `A` is the array $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$,

then `GRADE_DOWN(A)` has the value $\begin{bmatrix} 1 & 2 & 2 & 3 & 3 & 1 & 2 & 1 & 3 \\ 2 & 2 & 1 & 3 & 2 & 3 & 3 & 1 & 1 \end{bmatrix}$.

Case (ii): If `A` is the array $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$,

then `GRADE_DOWN(A, DIM = 1)` has the value $\begin{bmatrix} 2 & 1 & 3 \\ 1 & 2 & 1 \\ 3 & 3 & 2 \end{bmatrix}$,

and `GRADE_DOWN(A, DIM = 2)` has the value $\begin{bmatrix} 2 & 3 & 1 \\ 2 & 1 & 3 \\ 3 & 2 & 1 \end{bmatrix}$.

GRADE_UP(ARRAY,DIM)

Optional Argument. `DIM`

Description. Produces a permutation of the indices of an array, expressed as one-based coordinates, and sorted by ascending array element values.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or character. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result is of type default integer. If `DIM` is present, the result has the same shape as `ARRAY`. If `DIM` is absent, the result has shape `(/ SIZE(SHAPE(ARRAY)), SIZE(ARRAY) /)`.

Result Value.

Case (i): The result of

```
S = GRADE_UP(ARRAY)
      + SPREAD(LBOUND(ARRAY),DIM=2, NCOPIES=SIZE(ARRAY))-1
```

is such that if one computes the rank-one array `B` of size `SIZE(ARRAY)` by `FORALL (K=1:SIZE(B)) B(K)=ARRAY(S(1,K),S(2,K),...,S(N,K))` where `N` has the value `SIZE(SHAPE(ARRAY))`, then `B` is sorted in ascending order; moreover, all of the columns of `S` are distinct, that is, if $j \neq m$ then `ALL(S(:,`j`) .EQ. S(:,`m`))` will be false. The sort is stable; if $j \leq m$ and `B(`j`) = B(`m`)`, then `ARRAY(S(`$1, j$`),S(`$2, j$`),...,S(`n, j`))` precedes `ARRAY(S(`$1, m$`),S(`$2, m$`),...,S(`n, m`))` in the array element ordering of `ARRAY`. The collating sequence for an array of type `CHARACTER` is that used by the Fortran intrinsic functions, namely ASCII.

Case (ii): The result of

`R = GRADE_UP(ARRAY, DIM=K) + LBOUND(ARRAY, DIM=K) - 1`

has the property that if one computes the array

`B(`$i_1, i_2, \ldots, i_k, \ldots, i_n$`) =`
      `ARRAY(`$i_1, i_2, \ldots,$ `R(`$i_1, i_2, \ldots, i_k, \ldots, i_n$`)`$, \ldots, i_n$ `)`

then for all $i_1, i_2, \ldots,$ (omit i_k)$, \ldots, i_n$, the vector `B(`$i_1, i_2, \ldots, :, \ldots, i_n$`)` is sorted in ascending order; moreover, `R(`$i_1, i_2, \ldots, :, \ldots, i_n$`)` is a permutation of all the integers between 1 and `SIZE(ARRAY, DIM=K)`, inclusive. The sort is stable; that is, if $j \leq m$ and `B(`$i_1, i_2, \ldots, j, \ldots, i_n$`)` $=$ `B(`$i_1, i_2, \ldots, m, \ldots, i_n$`)`, then `R(`$i_1, i_2, \ldots, j, \ldots, i_n$`)` $\leq$ `R(`$i_1, i_2, \ldots, m, \ldots, i_n$`)`. The collating sequence for an array of type `CHARACTER` is that used by the Fortran intrinsic functions, namely ASCII.

Examples.

Case (i): `GRADE_UP( (/30, 20, 30, 40, -10/) )` is a rank two array of shape $\begin{bmatrix} 1 & 5 \end{bmatrix}$ with the value $\begin{bmatrix} 5 & 2 & 1 & 3 & 4 \end{bmatrix}$. (To produce a rank-one result, the optional `DIM = 1` argument must be used.)

If `A` is the array $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$,

then `GRADE_UP(A)` has the value $\begin{bmatrix} 1 & 3 & 3 & 1 & 2 & 2 & 3 & 2 & 1 \\ 1 & 1 & 2 & 3 & 3 & 1 & 3 & 2 & 2 \end{bmatrix}$.

Case (ii): If `A` is the array $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$,

then `GRADE_UP(A, DIM = 1)` has the value $\begin{bmatrix} 1 & 3 & 1 \\ 3 & 2 & 2 \\ 2 & 1 & 3 \end{bmatrix}$,

and `GRADE_UP(A, DIM = 2)` has the value $\begin{bmatrix} 1 & 3 & 2 \\ 3 & 1 & 2 \\ 1 & 2 & 3 \end{bmatrix}$.

HPF_ALIGNMENT(ALIGNEE, LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, NCOPIES)

Optional Arguments. `LB`, `UB`, `STRIDE`, `AXIS_MAP`, `IDENTITY_MAP`, `NCOPIES`

Description. Returns information regarding the correspondence of a variable and the *align-target* (array or template) to which it is ultimately aligned.

Class. Mapping inquiry subroutine.

Arguments.

`ALIGNEE`	may be of any type. It may be scalar or array valued. It must not be an assumed-size array. If it is a member of an aggregate variable group, then it must be an aggregate cover of the group. (See Section 3.8 for the definitions of "aggregate variable group" and "aggregate cover.") It must not be a pointer that is disassociated or an allocatable array that is not allocated. It is an `INTENT (IN)` argument. If `ALIGNEE` is a pointer, information about the alignment of its target is returned. The target must not be an assumed-size dummy argument or a section of an assumed-size dummy argument.
`LB` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of `ALIGNEE`. It is an `INTENT (OUT)` argument. The first element of the i th axis of `ALIGNEE` is ultimately aligned to the `LB`(i) th

	align-target element along the axis of the *align-target* associated with the ith axis of `ALIGNEE`. If the ith axis of `ALIGNEE` is a collapsed axis, `LB`(i) is implementation dependent.
`UB` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of `ALIGNEE`. It is an `INTENT (OUT)` argument. The last element of the ith axis of `ALIGNEE` is ultimately aligned to the `UB`(i)th *align-target* element along the axis of the *align-target* associated with the ith axis of `ALIGNEE`. If the ith axis of `ALIGNEE` is a collapsed axis, `UB`(i) is implementation dependent.
`STRIDE` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of `ALIGNEE`. It is an `INTENT (OUT)` argument. The ith element of `STRIDE` is set to the stride used in aligning the elements of `ALIGNEE` along its ith axis. If the ith axis of `ALIGNEE` is a collapsed axis, `STRIDE`(i) is zero.
`AXIS_MAP` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of `ALIGNEE`. It is an `INTENT (OUT)` argument. The ith element of `AXIS_MAP` is set to the *align-target* axis associated with the ith axis of `ALIGNEE`. If the ith axis of `ALIGNEE` is a collapsed axis, `AXIS_MAP(i)` is 0.
`IDENTITY_MAP` (optional)	must be scalar and of type default logical. It is an `INTENT (OUT)` argument. It is set to true if the ultimate *align-target* associated with `ALIGNEE` has a shape identical to `ALIGNEE`, the axes are mapped using the identity permutation, and the strides are all positive (and therefore equal to 1, because of the shape constraint); otherwise it is set to false. If a variable has not appeared as an *alignee* in an `ALIGN` or `REALIGN` directive, and does not have the `INHERIT` attribute, then `IDENTITY_MAP` must be true; it can be true in other circumstances as well.
`NCOPIES` (optional)	must be scalar and of type default integer. It is an `INTENT (OUT)` argument. It is set to the number of copies of `ALIGNEE` that are ultimately aligned to *align-target*. For a non-replicated variable, it is set to one.

Examples. If `ALIGNEE` is scalar, then no elements of `LB`, `UB`, `STRIDE`, or `AXIS_MAP` are set.

Given the declarations

```
      REAL PI = 3.1415927
      DIMENSION A(10,10),B(20,30),C(20,40,10),D(40)
!HPF$ TEMPLATE T(40,20)
```

```
!HPF$ ALIGN A(I,:) WITH T(1+3*I,2:20:2)
!HPF$ ALIGN C(I,*,J) WITH T(J,21-I)
!HPF$ ALIGN D(I) WITH T(I,4)
!HPF$ PROCESSORS PROCS(4,2), SCALARPROC
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO PROCS
!HPF$ DISTRIBUTE B(CYCLIC,BLOCK) ONTO PROCS
!HPF$ DISTRIBUTE ONTO SCALARPROC :: PI
```

assuming that the actual mappings are as the directives specify, the results of calling HPF_ALIGNMENT are:

	A	B	C	D
LB	[4, 2]	[1, 1]	[20, N/A, 1]	[1]
UB	[31, 20]	[20, 30]	[1, N/A, 10]	[40]
STRIDE	[3, 2]	[1, 1]	[-1, 0, 1]	[1]
AXIS_MAP	[1, 2]	[1, 2]	[2, 0, 1]	[1]
IDENTITY_MAP	false	true	false	false
NCOPIES	1	1	1	1

where "N/A" denotes a implementation-dependent result. To illustrate the use of NCOPIES, consider:

```
      LOGICAL BOZO(20,20),RONALD_MCDONALD(20)
!HPF$ TEMPLATE EMMETT_KELLY(100,100)
!HPF$ ALIGN RONALD_MCDONALD(I) WITH BOZO(I,*)
!HPF$ ALIGN BOZO(J,K) WITH EMMETT_KELLY(J,5*K)
```

CALL HPF_ALIGNMENT(RONALD_MCDONALD, NCOPIES = NC) sets NC to 20. Now consider:

```
      LOGICAL BOZO(20,20),RONALD_MCDONALD(20)
!HPF$ TEMPLATE WILLIE_WHISTLE(100)
!HPF$ ALIGN RONALD_MCDONALD(I) WITH BOZO(I,*)
!HPF$ ALIGN BOZO(J,*) WITH WILLIE_WHISTLE(5*J)
```

CALL HPF_ALIGNMENT(RONALD_MCDONALD, NCOPIES = NC) sets NC to one.

HPF_DISTRIBUTION(DISTRIBUTEE, AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE)

Optional Arguments. AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE

Description. The HPF_DISTRIBUTION subroutine returns information regarding the distribution of the ultimate *align-target* associated with a variable.

Class. Mapping inquiry subroutine.

Arguments.

DISTRIBUTEE	may be of any type. It may be scalar or array valued. It must not be an assumed-size array. If it is a member of an aggregate variable group, then it must be an aggregate

cover of the group. (See Section 3.8 for the definitions of "aggregate variable group" and "aggregate cover.") It must not be a pointer that is disassociated or an allocatable array that is not allocated. It is an `INTENT (IN)` argument.

If `DISTRIBUTEE` is a pointer, information about the distribution of its target is returned. The target must not be an assumed-size dummy argument or a section of an assumed-size dummy argument.

`AXIS_TYPE` (optional) must be a rank one array of type default character. It may be of any length, although it must be of length at least 9 in order to contain the complete value. Its elements are set to the values below as if by a character intrinsic assignment statement. Its size must be at least equal to the rank of the *align-target* to which `DISTRIBUTEE` is ultimately aligned; this is the value returned by `HPF_TEMPLATE` in `TEMPLATE_RANK`). It is an `INTENT (OUT)` argument. Its ith element contains information on the distribution of the ith axis of that *align-target*. The following values are defined by HPF (implementations may define other values):

`'BLOCK'` The axis is distributed `BLOCK`. The corresponding element of `AXIS_INFO` contains the block size.

`'COLLAPSED'` The axis is collapsed (distributed with the "*" specification). The value of the corresponding element of `AXIS_INFO` is implementation dependent.

`'CYCLIC'` The axis is distributed `CYCLIC`. The corresponding element of `AXIS_INFO` contains the block size.

`AXIS_INFO` (optional) must be a rank one array of type default integer, and size at least equal to the rank of the *align-target* to which `DISTRIBUTEE` is ultimately aligned (which is returned by `HPF_TEMPLATE` in `TEMPLATE_RANK`). It is an `INTENT (OUT)` argument. The ith element of `AXIS_INFO` contains the block size in the block or cyclic distribution of the ith axis of the ultimate *align-target* of `DISTRIBUTEE`; if that axis is a collapsed axis, then the value is implementation dependent.

`PROCESSORS_RANK` (optional) must be scalar and of type default integer. It is set to the rank of the processor arrangement onto which `DISTRIBUTEE` is distributed. It is an `INTENT (OUT)` argument.

`PROCESSORS_SHAPE` (optional) must be a rank one array of type default integer and of size at least equal to the value, m, returned in `PROCESSORS_RANK`. It is an `INTENT (OUT)` argument. Its first m

elements are set to the shape of the processor arrangement onto which DISTRIBUTEE is mapped. (It may be necessary to call HPF_DISTRIBUTION twice, the first time to obtain the value of PROCESSORS_RANK in order to allocate PROCESSORS_SHAPE.)

Example. Given the declarations in the example illustrating HPF_ALIGNMENT, and assuming that the actual mappings are as the directives specify, the results of HPF_DISTRIBUTION are:

	A	B	PI
AXIS_TYPE	['BLOCK', 'BLOCK']	['CYCLIC', 'BLOCK']	[ ]
AXIS_INFO	[10, 10]	[1, 15]	[ ]
PROCESSORS_SHAPE	[4, 2]	[2, 2]	[ ]
PROCESSORS_RANK	2	2	0

HPF_TEMPLATE(ALIGNEE, TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO, NUMBER_ALIGNED)

Optional Arguments. LB, UB, AXIS_TYPE, AXIS_INFO, NUMBER_ALIGNED, TEMPLATE_RANK

Description. The HPF_TEMPLATE subroutine returns information regarding the ultimate *align-target* associated with a variable; HPF_TEMPLATE returns information concerning the variable from the point of view of its ultimate *align-target*, while HPF_ALIGNMENT returns information from the variable's point of view.

Class. Mapping inquiry subroutine.

Arguments.

ALIGNEE	may be of any type. It may be scalar or array valued. It must not be an assumed-size array. If it is a member of an aggregate variable group, then it must be an aggregate cover of the group. (See Section 3.8 for the definitions of "aggregate variable group" and "aggregate cover.") It must not be a pointer that is disassociated or an allocatable array that is not allocated. It is an INTENT (IN) argument. If ALIGNEE is a pointer, information about the alignment of its target is returned. The target must not be an assumed-size dummy argument or a section of an assumed-size dummy argument.
TEMPLATE_RANK (optional)	must be scalar and of type default integer. It is an INTENT (OUT) argument. It is set to the rank of the ultimate *align-target*. This can be different from the rank of the ALIGNEE, due to collapsing and replicating.

`LB` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of the *align-target* to which `ALIGNEE` is ultimately aligned; this is the value returned in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. The i$^{\text{th}}$ element of `LB` contains the declared *align-target* lower bound for the i$^{\text{th}}$ template axis.
`UB` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of the *align-target* to which `ALIGNEE` is ultimately aligned; this is the value returned in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. The i$^{\text{th}}$ element of `UB` contains the declared *align-target* upper bound for the i$^{\text{th}}$ template axis.
`AXIS_TYPE` (optional)	must be a rank one array of type default character. It may be of any length, although it must be of length at least 10 in order to contain the complete value. Its elements are set to the values below as if by a character intrinsic assignment statement. Its size must be at least equal to the rank of the *align-target* to which `ALIGNEE` is ultimately aligned; this is the value returned in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. The i$^{\text{th}}$ element of `AXIS_TYPE` contains information about the i$^{\text{th}}$ axis of the *align-target*. The following values are defined by HPF (implementations may define other values):

`'NORMAL'` The *align-target* axis has an axis of `ALIGNEE` aligned to it. For elements of `AXIS_TYPE` assigned this value, the corresponding element of `AXIS_INFO` is set to the number of the axis of `ALIGNEE` aligned to this *align-target* axis.

`'REPLICATED'` `ALIGNEE` is replicated along this *align-target* axis. For elements of `AXIS_TYPE` assigned this value, the corresponding element of `AXIS_INFO` is set to the number of copies of `ALIGNEE` along this *align-target* axis.

`'SINGLE'` `ALIGNEE` is aligned with one coordinate of the *align-target* axis. For elements of `AXIS_TYPE` assigned this value, the corresponding element of `AXIS_INFO` is set to the *align-target* coordinate to which `ALIGNEE` is aligned.

`AXIS_INFO` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of the *align-target* to which `ALIGNEE` is ultimately aligned; this is the value returned in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. See the description of `AXIS_TYPE` above.
`NUMBER_ALIGNED` (optional)	must be scalar and of type default integer. It is an `INTENT (OUT)` argument. It is set to the total number

of variables aligned to the ultimate *align-target*. This is the number of variables that are moved if the *align-target* is redistributed.

Example. Given the declarations in the example illustrating HPF_ALIGNMENT, and assuming that the actual mappings are as the directives specify, the results of HPF_TEMPLATE are:

	A	C	D
LB	[1, 1]	[1, 1]	[1, 1]
UB	[40, 20]	[40, 20]	[40, 20]
AXIS_TYPE	['NORMAL', 'NORMAL']	['NORMAL', 'NORMAL']	['NORMAL', 'SINGLE']
AXIS_INFO	[1, 2]	[3, 1]	[1, 4]
NUMBER_ALIGNED	3	3	3
TEMPLATE_RANK	2	2	2

IALL(ARRAY, DIM, MASK)

Optional Arguments. DIM, MASK

Description. Computes a bitwise logical AND reduction along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of type integer with the same kind type parameter as ARRAY. It is scalar if DIM is absent or if ARRAY has rank one; otherwise, the result is an array of rank $n - 1$ and shape $(d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of IALL(ARRAY) is the IAND reduction of all the elements of ARRAY. If ARRAY has size zero, the result is equal to a implementation-dependent integer value x with the property that IAND(I, x) = I for all integers I of the same kind type parameter as ARRAY. See Section 7.4.3.

Case (ii): The result of `IALL(ARRAY, MASK=MASK)` is the `IAND` reduction of all the elements of `ARRAY` corresponding to the true elements of `MASK`; if `MASK` contains no true elements, the result is equal to a implementation-dependent integer value x (of the same kind type parameter as `ARRAY`) with the property that `IAND(I, x) = I` for all integers `I`.

Case (iii): If `ARRAY` has rank one, `IALL(ARRAY, DIM [,MASK])` has a value equal to that of `IALL(ARRAY [,MASK])`. Otherwise, the value of element $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of `IALL(ARRAY, DIM [,MASK])` is equal to `IALL(ARRAY(`$s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$`)` `[,MASK = MASK(`$s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$`)])`

Examples.

Case (i): The value of `IALL((/7, 6, 3, 2/))` is 2.

Case (ii): The value of `IALL(C, MASK = BTEST(C,0))` is the `IAND` reduction of the odd elements of `C`.

Case (iii): If `B` is the array $\begin{bmatrix} 2 & 3 & 5 \\ 3 & 7 & 7 \end{bmatrix}$, then `IALL(B, DIM = 1)` is $\begin{bmatrix} 2 & 3 & 5 \end{bmatrix}$ and `IALL(B, DIM = 2)` is $\begin{bmatrix} 0 & 3 \end{bmatrix}$.

IALL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented bitwise logical AND scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `IALL((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `IALL_PREFIX( (/1,3,2,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 1 & 1 & 0 & 4 & 4 \end{bmatrix}$.

IALL_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

Optional Argument. `MASK`

Description. Scatters elements of `ARRAY` selected by `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. The j$^{\text{th}}$ bit of an element of the result is 1 if and only if the j$^{\text{th}}$ bits of the corresponding element of `BASE` and of the elements of `ARRAY` scattered to that position are all equal to 1.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`BASE`	must be of type integer with the same kind type parameter as `ARRAY`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `ARRAY`. The number of `INDX` arguments must be equal to the rank of `BASE`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `IALL( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `ARRAY` associated with b as described in Section 7.4.4.

Example. `IALL_SCATTER((/1, 2, 3, 6/), (/1, 3, 7/), (/1, 1, 2, 2/))` is $\begin{bmatrix} 0 & 2 & 7 \end{bmatrix}$.

IALL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented bitwise logical AND scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `IALL((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `IALL_SUFFIX( (/1,3,2,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 0 & 2 & 2 & 4 & 5 \end{bmatrix}$.

IANY(ARRAY, DIM, MASK)

Optional Arguments. `DIM`, `MASK`

Description. Computes a bitwise logical OR reduction along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. The result is of type integer with the same kind type parameter as `ARRAY`. It is scalar if `DIM` is absent or if `ARRAY` has rank one; otherwise, the result is an array of rank $n-1$ and shape $(d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of `ARRAY`.

Result Value.

Case (i): The result of `IANY(ARRAY)` is the `IOR` reduction of all the elements of `ARRAY`. If `ARRAY` has size zero, the result has the value zero. See Section 7.4.3.

Case (ii): The result of `IANY(ARRAY, MASK=MASK)` is the `IOR` reduction of all the elements of `ARRAY` corresponding to the true elements of `MASK`; if `MASK` contains no true elements, the result is zero.

Case (iii): If `ARRAY` has rank one, `IANY(ARRAY, DIM [,MASK])` has a value equal to that of `IANY(ARRAY [,MASK])`. Otherwise, the value of element $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of `IANY(ARRAY, DIM [,MASK])` is equal to `IANY(ARRAY(`$s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$`)` `[,MASK = MASK(`$s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$`)])`

Examples.

Case (i): The value of `IANY((/9, 8, 3, 2/))` is 11.

Case (ii): The value of `IANY(C, MASK = BTEST(C,0))` is the `IOR` reduction of the odd elements of `C`.

Case (iii): If `B` is the array $\begin{bmatrix} 2\ 3\ 5 \\ 0\ 4\ 2 \end{bmatrix}$, then `IANY(B, DIM = 1)` is $\begin{bmatrix} 2 & 7 & 7 \end{bmatrix}$ and `IANY(B, DIM = 2)` is $\begin{bmatrix} 7 & 6 \end{bmatrix}$.

IANY_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented bitwise logical OR scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `IANY((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `IANY_PREFIX( (/1,2,3,2,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 1 & 3 & 3 & 2 & 7 \end{bmatrix}$.

IANY_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

Optional Argument. `MASK`

Description. Scatters elements of `ARRAY` selected by `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. The j$^{\text{th}}$ bit of an element of the result is 1 if and only if the j$^{\text{th}}$ bit of the corresponding element of `BASE` or of any of the elements of `ARRAY` scattered to that position is equal to 1.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`BASE`	must be of type integer with the same kind type parameter as `ARRAY`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `ARRAY`. The number of `INDX` arguments must be equal to the rank of `BASE`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `IANY( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `ARRAY` associated with b as described in Section 7.4.4.

Example. `IANY_SCATTER((/1, 2, 3, 6/), (/1, 3, 7/), (/1, 1, 2, 2/))` is $\begin{bmatrix} 3 & 7 & 7 \end{bmatrix}$.

IANY_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented bitwise logical OR scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `IANY((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `IANY_SUFFIX( (/4,2,3,2,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 7 & 3 & 3 & 7 & 5 \end{bmatrix}$.

IPARITY(ARRAY, DIM, MASK)

Optional Arguments. DIM, MASK

Description. Computes a bitwise logical exclusive OR reduction along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of type integer with the same kind type parameter as ARRAY. It is scalar if DIM is absent or if ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and shape $(d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of IPARITY(ARRAY) is the IEOR reduction of all the elements of ARRAY. If ARRAY has size zero, the result has the value zero. See Section 7.4.3.

Case (ii): The result of IPARITY(ARRAY, MASK=MASK) is the IEOR reduction of all the elements of ARRAY corresponding to the true elements of MASK; if MASK contains no true elements, the result is zero.

Case (iii): If ARRAY has rank one, IPARITY(ARRAY, DIM [,MASK]) has a value equal to that of IPARITY(ARRAY [,MASK]). Otherwise, the value of element $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of IPARITY(ARRAY, DIM [,MASK]) is equal to IPARITY(ARRAY$(s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n)$ [,MASK = MASK$(s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n)$])

Examples.

Case (i): The value of IPARITY((/13, 8, 3, 2/)) is 4.

Case (ii): The value of IPARITY(C, MASK = BTEST(C,0)) is the IEOR reduction of the odd elements of C.

Case (iii): If B is the array $\begin{bmatrix} 2 & 3 & 7 \\ 0 & 4 & 2 \end{bmatrix}$, then IPARITY(B, DIM = 1) is $\begin{bmatrix} 2 & 7 & 5 \end{bmatrix}$ and IPARITY(B, DIM = 2) is $\begin{bmatrix} 6 & 6 \end{bmatrix}$.

IPARITY_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented bitwise logical exclusive OR scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `IPARITY((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `IPARITY_PREFIX( (/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 1 & 3 & 0 & 4 & 1 \end{bmatrix}$.

IPARITY_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

Optional Argument. `MASK`

Description. Scatters elements of `ARRAY` selected by `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. The j$^{\text{th}}$ bit of an element of the result is 1 if and only if there are an odd number of ones among the j$^{\text{th}}$ bits of the corresponding element of `BASE` and the elements of `ARRAY` scattered to that position.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`BASE`	must be of type integer with the same kind type parameter as `ARRAY`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `ARRAY`. The number of `INDX` arguments must be equal to the rank of `BASE`.

`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `IPARITY( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `ARRAY` associated with b as described in Section 7.4.4.

Example. `IPARITY_SCATTER((/1,2,3,6/), (/1,3,7/), (/1,1,2,2/))` is $\begin{bmatrix} 2 & 6 & 7 \end{bmatrix}$.

IPARITY_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented bitwise logical exclusive OR scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `IPARITY((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `IPARITY_SUFFIX( (/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 0 & 1 & 3 & 1 & 5 \end{bmatrix}$.

LEADZ(I)

Description. Return the number of leading zeros in an integer.

Class. Elemental function.

Argument. `I` must be of type integer.

Result Type and Type Parameter. Same as `I`.

Result Value. The result is a count of the number of leading 0-bits in the integer `I`. The model for the interpretation of an integer as a sequence of bits is in Section F95:13.5.7 `LEADZ(0)` is `BIT_SIZE(I)`. For nonzero `I`, if the leftmost one bit of `I` occurs in position $k-1$ (where the rightmost bit is bit 0) then `LEADZ(I)` is `BIT_SIZE(I) - k`.

Examples. `LEADZ(3)` has the value `BIT_SIZE(3) - 2`. For scalar `I`, `LEADZ(I) .EQ. MINVAL( (/ (J, J=0, BIT_SIZE(I) ) /), MASK=M )` where `M =(/ (BTEST(I,J), J=BIT_SIZE(I)-1, 0, -1), .TRUE. /)`. A given integer `I` may produce different results from `LEADZ(I)`, depending on the number of bits in the representation of the integer (`BIT_SIZE(I)`). That is because `LEADZ` counts bits from the most significant bit. Compare with `ILEN`.

MAXVAL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented `MAXVAL` scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer or real. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `MAXVAL((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `MAXVAL_PREFIX( (/3,4,-5,2,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 3 & 4 & 4 & 2 & 5 \end{bmatrix}$.

MAXVAL_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

Optional Argument. `MASK`

Description. Scatters elements of `ARRAY` selected by `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. Each element of the result is assigned the maximum value of the corresponding element of `BASE` and the elements of `ARRAY` scattered to that position.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer or real. It must not be scalar.
`BASE`	must be of the same type and kind type parameter as `ARRAY`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `ARRAY`. The number of `INDX` arguments must be equal to the rank of `BASE`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `MAXVAL( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `ARRAY` associated with b as described in Section 7.4.4.

Example. `MAXVAL_SCATTER((/1, 2, 3, 1/), (/4, -5, 7/), (/1, 1, 2, 2/))` is $\begin{bmatrix} 4 & 3 & 7 \end{bmatrix}$.

MAXVAL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented `MAXVAL` scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer or real. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `MAXVAL((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `MAXVAL_SUFFIX( (/3,4,-5,2,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 4 & 4 & \text{-5} & 5 & 5 \end{bmatrix}$.

MINVAL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented `MINVAL` scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer or real. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `MINVAL((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `MINVAL_PREFIX( (/1,2,-3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 1 & 1 & \text{-3} & 4 & 4 \end{bmatrix}$.

MINVAL_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

Optional Argument. `MASK`

Description. Scatters elements of `ARRAY` selected by `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. Each element of the result is assigned the minimum value of the corresponding element of `BASE` and the elements of `ARRAY` scattered to that position.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer or real. It must not be scalar.
`BASE`	must be of the same type and kind type parameter as `ARRAY`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `ARRAY`. The number of `INDX` arguments must be equal to the rank of `BASE`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `MINVAL( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `ARRAY` associated with b as described in Section 7.4.4.

Example. `MINVAL_SCATTER((/ 1,-2,-3,6 /), (/ 4,3,7 /), (/ 1,1,2,2 /))` is $\begin{bmatrix} \texttt{-2} & \texttt{-3} & \texttt{7} \end{bmatrix}$.

MINVAL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented `MINVAL` scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer or real. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `MINVAL((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `MINVAL_SUFFIX( (/1,2,-3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} -3 & -3 & -3 & 4 & 5 \end{bmatrix}$.

PARITY(MASK, DIM)

Optional Argument. `DIM`

Description. Determine whether an odd number of values are true in `MASK` along dimension `DIM`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result is of type logical with the same kind type parameter as `MASK`. It is scalar if `DIM` is absent or if `MASK` has rank one; otherwise, the result is an array of rank $n-1$ and shape $(d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of `MASK`.

Result Value.

Case (i): The result of `PARITY(MASK)` is the `.NEQV.` reduction of all the elements of `MASK`. If `MASK` has size zero, the result has the value false. See Section 7.4.3.

Case (ii): If `MASK` has rank one, `PARITY(MASK, DIM)` has a value equal to that of `PARITY(MASK)`. Otherwise, the value of element $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of `PARITY(MASK, DIM)` is equal to `PARITY(MASK(`$s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$`))`

Examples.

Case (i): The value of `PARITY((/T, T, T, F/))` is true.

Case (ii): If `B` is the array $\begin{bmatrix} \text{T} & \text{T} & \text{F} \\ \text{T} & \text{T} & \text{T} \end{bmatrix}$, then `PARITY(B, DIM = 1)` is $\begin{bmatrix} \text{F} & \text{F} & \text{T} \end{bmatrix}$ and `PARITY(B, DIM = 2)` is $\begin{bmatrix} \text{F} & \text{T} \end{bmatrix}$.

PARITY_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented logical exclusive OR scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `MASK`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `MASK`.

Result Value. Element r of the result has the value `PARITY((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `MASK` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `PARITY_PREFIX( (/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{T} & \mathrm{F} \end{bmatrix}$.

PARITY_SCATTER(MASK,BASE,INDX1, ..., INDXn)

Description. Scatters elements of `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. An element of the result is true if and only if the number of true values among the corresponding element of `BASE` and the elements of `MASK` scattered to that position is odd.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`BASE`	must be of type logical with the same kind type parameter as `MASK`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `MASK`. The number of `INDX` arguments must be equal to the rank of `BASE`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `PARITY( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `MASK` associated with b as described in Section 7.4.4.

Example. `PARITY_SCATTER((/ T,T,T,T /), (/ T,F,F /), (/ 1,1,1,2 /))` is $\begin{bmatrix} \mathtt{F} & \mathtt{T} & \mathtt{F} \end{bmatrix}$.

PARITY_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented logical exclusive OR scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `MASK`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `MASK`.

Result Value. Element r of the result has the value `PARITY((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `MASK` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `PARITY_SUFFIX( (/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{T} \end{bmatrix}$.

POPCNT(I)

Description. Return the number of one bits in an integer.

Class. Elemental function.

Argument. `I` must be of type integer.

Result Type and Type Parameter. Same as `I`.

Result Value. `POPCNT(I)` is the number of one bits in the binary representation of the integer `I`. The model for the interpretation of an integer as a sequence of bits is in Section F95:13.5.7

Example. `POPCNT(I) = COUNT((/ (BTEST(I,J), J=0, BIT_SIZE(I)-1) /))`, for scalar `I`.

POPPAR(I)

Description. Return the parity of an integer.

Class. Elemental function.

Argument. `I` must be of type integer.

Result Type and Type Parameter. Same as `I`.

Result Value. `POPPAR(I)` is 1 if there are an odd number of one bits in `I` and zero if there are an even number. The model for the interpretation of an integer as a sequence of bits is in Section F95:13.5.7

Example. For scalar `I`, `POPPAR(I) = MERGE(1,0,BTEST(POPCNT(I),0))`.

PRODUCT_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented `PRODUCT` scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or complex. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `PRODUCT((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `PRODUCT_PREFIX( (/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 1 & 2 & 6 & 4 & 20 \end{bmatrix}$.

PRODUCT_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

Optional Argument. `MASK`

Description. Scatters elements of `ARRAY` selected by `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. Each element of the result is equal to the product of the corresponding element of `BASE` and the elements of `ARRAY` scattered to that position.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or complex. It must not be scalar.
`BASE`	must be of the same type and kind type parameter as `ARRAY`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `ARRAY`. The number of `INDX` arguments must be equal to the rank of `BASE`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `PRODUCT( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `ARRAY` associated with b as described in Section 7.4.4.

Example. `PRODUCT_SCATTER((/ 1,2,3,1 /), (/ 4,-5,7 /), (/ 1,1,2,2 /))` is $\begin{bmatrix} 8 & -15 & 7 \end{bmatrix}$.

PRODUCT_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented `PRODUCT` scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or complex. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `PRODUCT((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `PRODUCT_SUFFIX( (/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\left[\begin{array}{ccccc} 6 & 6 & 3 & 20 & 5 \end{array}\right]$.

SORT_DOWN(ARRAY,DIM)

Optional Argument. `DIM`

Description. Sort by descending value.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or character. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result has the same shape, type, and type parameter as `ARRAY`.

Result Value.

Case (i): The result of `SORT_DOWN(ARRAY)`, when `ARRAY` is one-dimensional, is a vector of the same shape as `ARRAY`, containing the same elements (with the same number of instances) but sorted in descending element order. The collating sequence for an array of type `CHARACTER` is that used by the Fortran intrinsic functions, namely ASCII.

Case (ii): The result of `SORT_DOWN(ARRAY)` for a multi-dimensional `ARRAY` is the result that would be obtained by reshaping `ARRAY` to a rank-one array `V` using array element order, sorting that rank-one array in descending order, as in Case(i), and finally restoring the result to the original shape. That is, it gives the same result as `RESHAPE( SORT_DOWN(V), SHAPE = SHAPE(ARRAY) )`, where `V = RESHAPE( ARRAY, SHAPE = (/ M /)` and `M = SIZE(ARRAY)`.

Case (iii): The result of `SORT_DOWN(ARRAY, DIM=k)` contains the same elements as `A`, but each one-dimensional array section of the form `ARRAY(`$i_1, i_2, \ldots, i_{k-1}, :, i_{k+1}, \ldots, i_n$`)`, where n is the rank of `ARRAY`, has been sorted in descending element order, as in Case(i) above.

Examples.

Case (i): `SORT_DOWN( (/30, 20, 30, 40, -10/) )` has the value $\begin{bmatrix} 40 & 30 & 30 & 20 & -10 \end{bmatrix}$.

Case (ii): If `A` is the array $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$,

then `SORT_DOWN(A)` has the value $\begin{bmatrix} 9 & 4 & 2 \\ 5 & 2 & 1 \\ 4 & 2 & 1 \end{bmatrix}$,

Case (iii): If `A` is the array $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$,

then `SORT_DOWN(A, DIM = 1)` has the value $\begin{bmatrix} 4 & 9 & 4 \\ 1 & 5 & 2 \\ 1 & 2 & 2 \end{bmatrix}$.

SORT_UP(ARRAY,DIM)

Optional Argument. `DIM`

Description. Sort by ascending value.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or character. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result has the same shape, type, and type parameter as `ARRAY`.

Result Value.

Case (i): The result of `SORT_UP(ARRAY)`, when `ARRAY` is one-dimensional, is a vector of the same shape as `ARRAY`, containing the same elements (with the same number of instances) but sorted in ascending element order. The collating sequence for an array of type `CHARACTER` is that used by the Fortran intrinsic functions, namely ASCII.

Case (ii): The result of `SORT_UP(ARRAY)` for a multi-dimensional `ARRAY` is the result that would be obtained by reshaping `ARRAY` to a rank-one array `V` using array element order, sorting that rank-one array in ascending order, as in Case(i), and finally restoring the result to the original shape. That is, it gives the same result as `RESHAPE( SORT_UP(V), SHAPE = SHAPE(ARRAY) )`, where `V = RESHAPE( ARRAY, SHAPE = (/ M /)` and `M = SIZE(ARRAY)`.

Case (iii): The result of `SORT_UP(ARRAY, DIM=k)` contains the same elements as `A`, but each one-dimensional array section of the form `ARRAY(`$i_1, i_2, \ldots, i_{k-1}, :, i_{k+1}, \ldots, i_n$`)`, where n is the rank of `ARRAY`, has been sorted in ascending element order, as in Case(i) above.

Examples.

Case (i): `SORT_UP( (/30, 20, 30, 40, -10/) )` has the value $\begin{bmatrix} \text{-10} & 20 & 30 & 30 & 40 \end{bmatrix}$.

Case (ii): If `A` is the array $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$,

then `SORT_UP(A)` has the value $\begin{bmatrix} 1 & 2 & 4 \\ 1 & 2 & 5 \\ 2 & 4 & 9 \end{bmatrix}$,

Case (iii): If `A` is the array $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$,

then `SORT_UP(A, DIM = 1)` has the value $\begin{bmatrix} 1 & 2 & 2 \\ 1 & 5 & 2 \\ 4 & 9 & 4 \end{bmatrix}$.

SUM_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented `SUM` scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or complex. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.

`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `SUM((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `SUM_PREFIX( (/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 1 & 3 & 6 & 4 & 9 \end{bmatrix}$.

SUM_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

Optional Argument. `MASK`

Description. Scatters elements of `ARRAY` selected by `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. Each element of the result is equal to the sum of the corresponding element of `BASE` and the elements of `ARRAY` scattered to that position.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or complex. It must not be scalar.
`BASE`	must be of the same type and kind type parameter as `ARRAY`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `ARRAY`. The number of `INDX` arguments must be equal to the rank of `BASE`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `SUM( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `ARRAY` associated with b as described in Section 7.4.4.

Example. `SUM_SCATTER((/1, 2, 3, 1/), (/4, -5, 7/), (/1, 1, 2, 2/))` is $\begin{bmatrix} \texttt{7} & \texttt{-1} & \texttt{7} \end{bmatrix}$.

SUM_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented `SUM` scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or complex. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `SUM((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 7.4.5.

Example. `SUM_SUFFIX( (/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 6 & 5 & 3 & 9 & 5 \end{bmatrix}$.

Part III

HPF Approved Extensions

This major section describes the syntax and semantics of features of approved extensions to High Performance Fortran. In most cases, these features build on concepts found in HPF itself; it may therefore be necessary to refer back to Parts I and II for background information.

Section 8

Approved Extensions for Data Mapping

This section describes a set of data mapping features that extend the capabilities provided by the base set as described in Section 3. These extensions can be divided into two categories.

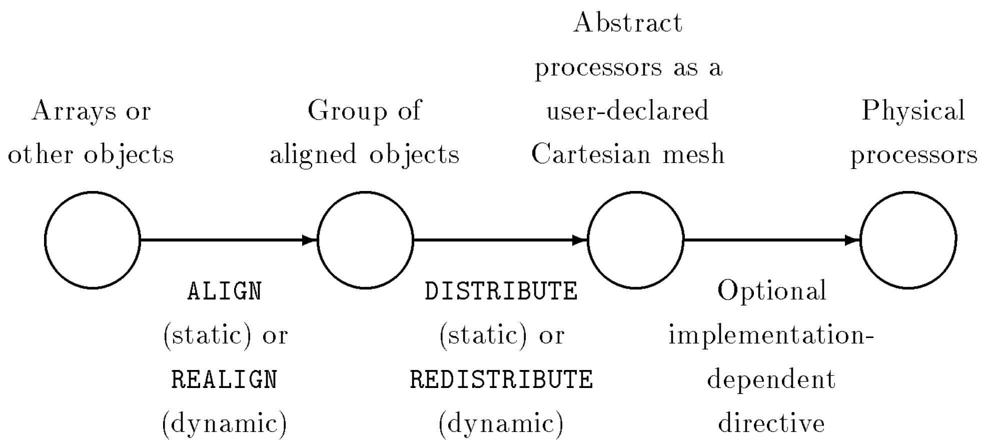
The first set of extensions provides the user greater control over the mapping of the data. These include directives for dynamic remapping of data, which allow the user to redistribute and realign at run time data that has been declared `DYNAMIC`. The `ONTO` clause used in the `DISTRIBUTE` directive is extended to allow direct distribution to subsets of processors. Explicit mapping of pointers and components of derived types are also introduced. Two new distributions are included: the `GEN_BLOCK` distribution, which generalizes the block distribution, and the `INDIRECT` distribution, which allows the mapping of individual array elements to be specified through a mapping array.

The programmer can use the second set of extensions to provide the compiler with information useful for generating efficient code. This category includes the `RANGE` directive, which allows the user to specify the range of distributions that a dynamically distributed array, a pointer, or a dummy argument may have. The `SHADOW` directive allows the user to specify the amount of additional space required on a processor to accommodate non-local elements in a nearest-neighbor computation.

Since this section deals with extensions, we repeat some of the sections of Sections 3 and 4, providing new rules and extending old ones where necessary. In particular, subsections 8.13, 8.14 and 8.15 extend the corresponding subsections in Section 3 based on the approved extensions described here.

8.1 Extended Model

The fundamental model for allocation of data to abstract processors still remains a two-level mapping as described in Section 3. However, it is extended to allow the dynamic remapping of the data objects as illustrated by the following diagram:


Arrays or other objects
Group of aligned objects
Abstract processors as a user-declared Cartesian mesh
Physical processors
ALIGN (static) or REALIGN (dynamic)
DISTRIBUTE (static) or REDISTRIBUTE (dynamic)
Optional implementation-dependent directive

Thus, objects can be remapped at execution time using the executable directives `REALIGN` and `REDISTRIBUTE`. Any object that is the root of an alignment tree (i.e., is not explicitly aligned to another object) can be explicitly redistributed. Redistributing such an object causes all objects ultimately aligned with it also to be redistributed so as to maintain the alignment relationships.

Any object that is not a root of an alignment tree can be explicitly realigned but not explicitly redistributed. Such a realignment does not change the mapping of any other object. Note that such remapping of data may require communication among the processors.

By analogy with the Fortran `ALLOCATABLE` attribute, HPF includes the `DYNAMIC` attribute. It is not permitted to `REALIGN` an array that has not been declared `DYNAMIC`. Similarly, it is not permitted to `REDISTRIBUTE` an array or template that has not been declared `DYNAMIC`.

Saved local variables, variables in common, and variables accessed by use association must not be implicitly remapped (e.g., by having variable distribution formats or being aligned with entities having variable distribution formats). Of these three categories of variables, only variables accessed by use association may have the `DYNAMIC` attribute.

As in Section 3.1, an object is considered to be *explicitly mapped* if it appears in an HPF mapping directive within the scoping unit in which it is declared; otherwise it is *implicitly mapped*. The definition of a mapping directive in Section 3.1 is extended as follows: A ⇑ mapping directive is an `ALIGN`, `DISTRIBUTE`, `INHERIT`, `DYNAMIC`, `RANGE`, or `SHADOW` directive, or any directive that confers an alignment, a distribution, or the `INHERIT`, `DYNAMIC`, `RANGE`, or `SHADOW` attributes.

8.2 Syntax of Attributed Forms of Extended Data Mapping Directives

Like other mapping directives, the executable directives `REALIGN` and `REDISTRIBUTE` also come in two forms (statement form and attribute form) but may not be combined with other attributes in a single directive. The `RANGE` and `SHADOW` attributes may be combined with other attributes in a single directive.

H801	*combined-attribute-extended*	**is**	`ALIGN` *align-attribute-stuff*
		or	`DISTRIBUTE` *dist-attribute-stuff*
		or	`INHERIT`
		or	`TEMPLATE`
		or	`PROCESSORS`
		or	`DIMENSION (` *explicit-shape-spec-list* `)`
		or	`DYNAMIC`
		or	`RANGE` *range-attr-stuff*
		or	`SHADOW` *shadow-attr-stuff*
		or	`SUBSET`

Constraint: The `SUBSET` attribute may be applied only to a processors arrangement.

The `SUBSET` attribute is discussed in Section 9; the rest are discussed below.

8.3 The REDISTRIBUTE Directive

The `REDISTRIBUTE` directive is similar to the `DISTRIBUTE` directive but is considered executable. An object or template may be redistributed at any time, provided it has been declared `DYNAMIC` (see Section 8.5). Any other objects currently ultimately aligned with an array (or template) when it is redistributed are also remapped to reflect the new distribution, in such a way as to preserve alignment relationships (see Section 3.4). (This can require a lot of computational and communication effort at run time; the programmer must take care when using this feature.) ⇑

The `DISTRIBUTE` directive may appear only in the *specification-part* of a scoping unit. The `REDISTRIBUTE` directive may appear only in the *execution-part* of a scoping unit. The principal difference between `DISTRIBUTE` and `REDISTRIBUTE` is that `DISTRIBUTE` must contain only a *specification-expr* as the argument to a distribution format such as `BLOCK` or `CYCLIC`, whereas in `REDISTRIBUTE` such an argument may be any integer expression. Another difference is that `DISTRIBUTE` is an attribute, and so can be combined with other attributes as part of a *combined-directive*, whereas `REDISTRIBUTE` is not an attribute (although a `REDISTRIBUTE` statement may be written in the style of attributed syntax, using "`::`" punctuation).

The syntax of the `REDISTRIBUTE` directive is:

H802	*redistribute-directive*	**is**	`REDISTRIBUTE` *distributee dist-directive-stuff*
		or	`REDISTRIBUTE` *dist-attribute-stuff* `::` *distributee-list*

Constraint: A *distributee* that appears in a `REDISTRIBUTE` directive must have the `DYNAMIC` attribute (see Section 8.5).

Constraint: A *distributee* in a `REDISTRIBUTE` directive may not appear as an *alignee* in an `ALIGN` or `REALIGN` directive.

Constraint: Neither the *dist-format-clause* nor the *dist-target* in a `REDISTRIBUTE` directive may begin with "`*`".

Note that, although an object may not have both the `INHERIT` attribute and the `DISTRIBUTE` attribute, any object—whether or not it has the `INHERIT` attribute—may appear as a *distributee* in a `REDISTRIBUTE` directive, provided that it has the `DYNAMIC` attribute and that it does not appear as an *alignee* in a `ALIGN` or `REALIGN` directive.

If a range directive (see Section 8.11) has been used to restrict the set of distribution formats allowed for a *distributee*, then the new mapping must match one of the formats specified in the range directive.

The statement form of a `REDISTRIBUTE` directive may be considered an abbreviation for an attributed form that happens to mention only one *distributee*; for example,

```
!HPF$ REDISTRIBUTE distributee ( dist-format-list ) ONTO dist-target
```

is equivalent to

```
!HPF$ REDISTRIBUTE ( dist-format-list ) ONTO dist-target :: distributee
```

8.4 The REALIGN Directive

The `REALIGN` directive is similar to the `ALIGN` directive but is considered executable. An array (or template) may be realigned at any time, provided it has been declared `DYNAMIC` (see Section 8.5). Unlike redistribution (also in Section 8.5), realigning a data object does not cause any other object to be remapped. (However, realignment of even a single object, if it is large, can require a lot of computational and communication effort at run time; the programmer must take care when using this feature.)

The `ALIGN` directive may appear only in the *specification-part* of a scoping unit. The `REALIGN` directive is similar but may appear only in the *execution-part* of a scoping unit. The principal difference between `ALIGN` and `REALIGN` is that `ALIGN` must contain only a *specification-expr* as a *subscript* or in a *subscript-triplet*, whereas in `REALIGN` an expression as a *subscript* or in a *subscript-triplet* need not be a *specification-expr*. Another difference is that `ALIGN` is an attribute, and so can be combined with other attributes as part of a *combined-directive*, whereas `REALIGN` is not an attribute (although a `REALIGN` statement may be written in the style of attributed syntax, using "`::`" punctuation).

The syntax of `REALIGN` is as follows:

H803	*realign-directive*	**is**	`REALIGN` *alignee align-directive-stuff*
		or	`REALIGN` *align-attribute-stuff* `::` *alignee-list*

Constraint: Any *alignee* that appears in a `REALIGN` directive must have the `DYNAMIC` attribute (see Section 8.5).

Constraint: If the *align-target* specified in the *align-with-clause* has the `DYNAMIC` attribute, then each *alignee* must also have the `DYNAMIC` attribute.

Constraint: An *alignee* in a `REALIGN` directive may not appear as a *distributee* in a `DISTRIBUTE` or `REDISTRIBUTE` directive.

Note that, although an object may not have both the `INHERIT` attribute and the `ALIGN` attribute, any object—whether or not it has the `INHERIT` attribute—may appear as an *alignee* in a `REALIGN` directive, provided it has the `DYNAMIC` attribute and that it does not appear as a *distributee* in a `DISTRIBUTE` or `REDISTRIBUTE` directive.

If a range directive (see Section 8.11) has been used to restrict the set of distribution formats allowed for an *alignee*, then the new mapping must match one of the formats specified in the range directive.

8.5 The DYNAMIC Directive

The `DYNAMIC` attribute specifies that an object may be dynamically realigned or redistributed.

H804	*dynamic-directive*	**is**	`DYNAMIC` *alignee-or-distributee-list*
H805	*alignee-or-distributee*	**is**	*alignee*
		or	*distributee*

Constraint: An object in `COMMON` may not be declared `DYNAMIC` and may not be aligned to an object (or template) that is `DYNAMIC`. (To get this kind of effect, modules must be used instead of `COMMON` blocks.)

Constraint: A component of a derived type may have the `DYNAMIC` attribute only if it also has the `POINTER` attribute. (See Section 8.9 for further discussion.)

Constraint: An object with the `SAVE` attribute may not be declared `DYNAMIC` and may not be aligned to an object (or template) that is `DYNAMIC`.

A `REALIGN` directive may not be applied to an *alignee* that does not have the `DYNAMIC` attribute. A `REDISTRIBUTE` directive may not be applied to a *distributee* that does not have the `DYNAMIC` attribute.

A `DYNAMIC` directive may be combined with other directives, with the attributes stated in any order, consistent with the Fortran attribute syntax.

Examples:

```
!HPF$ DYNAMIC A,B,C,D,E
!HPF$ DYNAMIC:: A,B,C,D,E
!HPF$ DYNAMIC, ALIGN WITH SNEEZY:: X,Y,Z
!HPF$ ALIGN WITH SNEEZY, DYNAMIC:: X,Y,Z
!HPF$ DYNAMIC, DISTRIBUTE(BLOCK, BLOCK) :: X,Y
!HPF$ DISTRIBUTE(BLOCK, BLOCK), DYNAMIC :: X,Y
```

The first two examples mean exactly the same thing. The next two examples mean exactly the same second thing. The last two examples mean exactly the same third thing.

The three directives

```
!HPF$ TEMPLATE A(64,64),B(64,64),C(64,64),D(64,64)
!HPF$ DISTRIBUTE(BLOCK, BLOCK) ONTO P:: A,B,C,D
!HPF$ DYNAMIC A,B,C,D
```

may be combined into a single directive as follows:

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) ONTO P,   &
!HPF$   DIMENSION(64,64),DYNAMIC :: A,B,C,D
```

An ALLOCATABLE object may also be given the DYNAMIC attribute. If an ALLOCATE statement is immediately followed by REDISTRIBUTE and/or REALIGN directives, the meaning in principle is that the array is first created with the statically declared mapping, if any, then immediately remapped. In practice there is an obvious optimization: create the array in the processors to which it is about to be remapped, in a single step. HPF implementors are strongly encouraged to implement this optimization and HPF programmers are encouraged to rely upon it. Here is an example:

```
      REAL,ALLOCATABLE(:,:) :: TINKER, EVERS
!HPF$ DYNAMIC :: TINKER, EVERS
      REAL, ALLOCATABLE :: CHANCE(:)
!HPF$ DISTRIBUTE(BLOCK),DYNAMIC :: CHANCE
      ...
      READ 6,M,N
      ALLOCATE(TINKER(N*M,N*M))
!HPF$ REDISTRIBUTE TINKER(CYCLIC, BLOCK)
      ALLOCATE(EVERS(N,N))
!HPF$ REALIGN EVERS(:,:) WITH TINKER(M::M,1::M)
      ALLOCATE(CHANCE(10000))
!HPF$ REDISTRIBUTE CHANCE(CYCLIC)
```

While CHANCE is by default always allocated with a BLOCK distribution, it should be possible for a compiler to notice that it will immediately be remapped to a CYCLIC distribution. Similar remarks apply to TINKER and EVERS. (Note that EVERS is mapped in a thinly-spread-out manner onto TINKER; adjacent elements of EVERS are mapped to elements of TINKER separated by a stride M. This thinly-spread-out mapping is put in the lower left corner of TINKER, because EVERS(1,1) is mapped to TINKER(M,1).)

⇑ In Section 5.1, a list is given of operations that, if performed in a do loop, cause the iterations of the loop to interfere with each other, and thereby prevent the loop from being characterized as INDEPENDENT. To that list must be added:

- Any REALIGN or REDISTRIBUTE directive performed in the loop interferes with any access to or any other remapping of the same data.

  Rationale. REALIGN and REDISTRIBUTE may change the processor storing a particular array element, which interferes with any assignment or use of that element. Similarly, multiple remapping operations may cause the same element to be stored in multiple locations. (*End of rationale.*)

8.6 Remapping and Subprogram Interfaces

If the dummy argument of any subprogram has the DYNAMIC attribute, then an explicit interface is required for the subprogram (see subsection 8.14). The rules on the interaction of the REALIGN and REDISTRIBUTE directives with a subprogram argument interface are:

1. A dummy argument may be declared DYNAMIC. However, it is subject to the general restrictions concerning the use of the name of an array to stand for its associated template.

The effect of any redistribution of the dummy after the procedure returns to the caller is dependent on the attribute of the actual argument. If the actual argument associated with the dummy has also been declared `DYNAMIC`, then any explicit remapping of the dummy is visible in the caller after the procedure returns. If a range directive (see Section 8.11) has been used to restrict the set of distribution formats allowed for the actual argument, then the new mapping must match one of the formats specified in the range directive.

A dummy argument whose associated actual argument has the `DYNAMIC` attribute may be used in `REALIGN` and `REDISTRIBUTE` as an *alignee* or *distributee* if and only if the associated actual argument is a whole array, not an array section.

If the actual argument associated with the dummy has not been declared `DYNAMIC` then the original mapping of the actual has to be restored on return. When the subprogram returns and the caller resumes execution, all objects accessible to the caller after the call that are not declared `DYNAMIC` are mapped exactly as they were before the call.

2. If an array or any section thereof is accessible by two or more paths, it is not HPF-conforming to remap it through any of those paths. For example, if an array is passed as an actual argument, it is forbidden to realign that array, or to redistribute an array or template to which it was aligned at the time of the call, until the subprogram has returned from the call. This prevents nasty aliasing problems. An example:

```
      MODULE FOO
      REAL A(10,10)
!HPF$ DYNAMIC ::  A
      END

      PROGRAM MAIN
      USE FOO
      CALL SUB(A(1:5,3:9))
      END

      SUBROUTINE SUB(B)
      USE FOO
      REAL B(:,:)
!HPF$ DYNAMIC ::  B
      ...
!HPF$ REDISTRIBUTE A            !Nonconforming
      ...
      END
```

Situations such as this are forbidden, for the same reasons that an assignment to `A` at the statement marked “Nonconforming” would also be forbidden. In general, in *any* situation where assignment to a variable would be nonconforming by reason of aliasing, remapping of that variable by an explicit `REALIGN` or `REDISTRIBUTE` directive is also forbidden.

Note that it is permitted to remap a host-associated or use-associated variable in a subprogram if it has been declared `DYNAMIC` and is accessible only through a single path. Such remappings stay in effect even after the subprogram has returned to its caller.

8.7 Mapping to Processor Subsets

This extension allows objects to be directly distributed to processor subsets by allowing a processor subset to be specified where a processor could be named, e.g., in a `DISTRIBUTE` directive. The specified subset must be a proper subset of the named processor arrangement.

The syntax of the extended *dist-target* is as follows:

H806	*extended-dist-target*	**is**	*processors-name* [(*section-subscript-list*)]
		or	* *processors-name* [(*section-subscript-list*)]
		or	*

Constraint: The *section-subscripts* in the *section-subscript-list* may not be *vector-subscripts* and are restricted to be either *subscripts* or *subscript-triplets*.

Constraint: In the *section-subscript-list*, the number of *section-subscripts* must equal the rank of the *processor-name*.

Constraint: Within a `DISTRIBUTE` directive, each *section-subscript* must be a *specification-expr*.

Constraint: Within a `DISTRIBUTE` or a `REDISTRIBUTE` directive, if both a *dist-format-list* and a *dist-target* appear, the number of elements of the *dist-format-list* that are not "*" must equal the number of *subscript-triplets* in the named processor arrangement.

Constraint: Within a `DISTRIBUTE` or a `REDISTRIBUTE` directive, if a *dist-target* appears but not a *dist-format-list*, the rank of each *distributee* must equal the number of *subscript-triplets* in the named processor arrangement.

```
!Example 1
!HPF$ PROCESSORS P(10)
      REAL A(100)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P(2:5)


!Example 2
!HPF$ PROCESSORS Q(10,10)
      REAL A(100,100)
!HPF$ DISTRIBUTE B(BLOCK,BLOCK) ONTO Q(5:10,5:10)
```

In Example 1, the array A is distributed by block across the processors P(2) to P(5) while in the second example, the array B is distributed across the lower right quadrant of the processor array Q.

Advice to users. This extension is most useful in conjunction with the tasking construct, see Section 9.4, which allows multiple independent phases of a computation to execute simultaneously on different subsets of processors. A similar situation arises when the code uses multiple data structures which can be computed in parallel where the computation on each individual object also exhibits parallelism, e.g., the multiple blocks in a multi-block grid used in some fluid dynamics calculation. Here, the individual blocks have to be distributed over subsets of processors to exploit both levels of parallelism. (*End of advice to users.*)

8.8 Pointers

8.8.1 Mapped Pointers

As an approved extension to HPF, pointers and targets can be explicitly mapped. Formally, this implies that the constraints that a *distributee* and an *alignee* may not have the POINTER or TARGET attribute as stated in Sections 3.3 and 3.4 respectively, have to be removed. ⇑

As in the case of an allocatable object, the mapping specification for a pointer does not take effect immediately but plays a role when the pointer becomes pointer associated with a target either through allocation or through pointer assignment.

When a pointer with an explicit mapping is used in an ALLOCATE statement, the data is allocated with the specified mapping.

For example:

```
      REAL, POINTER, DIMENSION(:) :: A, B
!HPF$ ALIGN B(I) WITH A(I)
!HPF$ DISTRIBUTE A(BLOCK)
      ...
      ALLOCATE(A(100))
      ALLOCATE(B(50))
      ...
      ALLOCATE(B(200))                ! Nonconforming
```

Pointer A is declared to have a BLOCK distribution while pointer B is declared to be identically aligned with A. When A is allocated, it is created with a block distribution. When B is allocated, it is aligned with the first 50 elements of A. Note that the allocation statements may not occur in the opposite order, since an object may be aligned to another only if it has already been created or allocated. Also, the second allocation for B is nonconforming, since a larger object, B here, cannot be aligned with a smaller object, A in this case.

A pointer P with an explicit mapping can be pointer associated with a target T through a pointer assignment statement under the following conditions:

1. The mapping of T is a specialization of the mapping of P (in particular, T must be a whole array); and

2. If P is explicitly aligned, its ultimate align target has a fully-specified non-transcriptive distribution; and

3. P and T are either both DYNAMIC or neither is.

Here are some examples:

```
      REAL, POINTER, DIMENSION(:,:) :: P
!HPF$ DISTRIBUTE P(BLOCK,BLOCK)
```

```
      REAL, TARGET, DIMENSION (100, 100) ::  B, C, D
!HPF$ DISTRIBUTE B(BLOCK, BLOCK)
!HPF$ DISTRIBUTE C(BLOCK, CYCLIC)
      ...
      P => B               ! Conforming
      P => B(1:50, 1:50)   ! Nonconforming: target must be a whole array.
      P => C               ! Nonconforming: the distribution in the
                           ! second dimension does not match
      P => D               ! Nonconforming: D is not explicitly mapped
      ...
```

The intuitive reason that the pointer assignment P => B(1:50, 1:50) above is nonconforming is similar to the reason that the example on page 53 (illustrating the difference between INHERIT A and DISTRIBUTE A * ONTO *) is nonconforming: Suppose for instance that the array B is distributed over a 2×2 processor arrangement. Then the section B(1:50, 1:50) would live entirely on processor (1, 1). This mapping is not correctly described by a (BLOCK, BLOCK) distribution for P.

The following pointer assignment is valid even though no processor arrangement is specified for the pointer; in this case, the mapping of B is a specialization of the mapping of P:

```
      REAL, POINTER, DIMENSION(:) :: P
      REAL, TARGET, DIMENSION(100) ::  B
!HPF$ PROCESSORS PROC(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE P(BLOCK)
!HPF$ DISTRIBUTE (BLOCK) ONTO PROC :: B
      ...
      P => B               ! Conforming
      ...
```

```
      REAL, POINTER, DIMENSION(:) :: P
!HPF$ DISTRIBUTE * :: P
      REAL, TARGET, DIMENSION(100) ::  B, C
!HPF$ DISTRIBUTE B(BLOCK), C(CYCLIC)
      ...
      P => B               ! Conforming
      P => C               ! Conforming
      P => C(1:50)         ! Nonconforming: target must be a whole array
      ...
```

Here, the * is used to indicate a transcriptive distribution for the pointer P and thus it can be pointer associated with both targets B and C distributed by BLOCK and CYCLIC respectively. However, it still cannot be used to point to an array section such as C(1:50). To do that, the pointer must have the INHERIT attribute:

```
      REAL, POINTER, DIMENSION(:) :: P
!HPF$ INHERIT :: P
      REAL, TARGET, DIMENSION(100) ::  B, C
!HPF$ DISTRIBUTE B(BLOCK), C(CYCLIC)
```

```
        ...
        P => B              ! Conforming
        P => C              ! Conforming
        P => C(1:50)        ! Conforming
        ...
```

To allow pointers to have transcriptive distributions, we have to change the constraint for *dist-format-clause* as specified in Section 3.3, to read as follows: ⇑

Constraint: If either the *dist-format-clause* or the *dist-target* in a `DISTRIBUTE` directive begins with "`*`" then every *distributee* must be a dummy argument, *except if the distributee has the `POINTER` attribute.*

The constraint for *align-spec* as specified in Section 3.4, should be changed to read as follows: ⇑

Constraint: If the *align-spec* in an `ALIGN` directive begins with "`*`" then every *alignee* must be a dummy argument, *except if the* alignee *has the `POINTER` attribute.*

The constraint for *inheritee* as specified in Section 4.4.2, should be changed to read as follows: ⇑

Constraint: An *inheritee* must be a dummy argument, *except if the* alignee *has the `POINTER` attribute.*

When pointers with such transcriptive mappings are used in an `ALLOCATE` statement, the compiler may choose any arbitrary mapping for the allocated data. A range declaration (see Section 8.11) can be used to restrict the set of distribution formats.

If a pointer has the `DYNAMIC` attribute, then any target associated with the pointer (which must therefore also have the `DYNAMIC` attribute) may be remapped using a `REALIGN` or `REDISTRIBUTE` statement under the following restriction:

A pointer may be used in `REALIGN` and `REDISTRIBUTE` as an *alignee*, *align-target*, or *distributee* if and only if it is currently associated with a whole array, not an array section.

Note that when an object is remapped, the new mapping is visible through any pointer that may be associated with the object.

8.8.2 Pointers and Subprograms

If a pointer dummy argument is not explicitly mapped, then the actual argument must also not be explicitly mapped.

If a pointer dummy argument has an explicit mapping, then the actual argument must follow the rules for pointer assignment as stated above, with one exception: If the actual argument has the `DYNAMIC` attribute, it is not necessary that the corresponding dummy argument have the `DYNAMIC` attribute. That is, item 3 on page 151 is weakened to

3. If a pointer dummy argument has the `DYNAMIC` attribute, then the corresponding actual argument must also have the `DYNAMIC` attribute.

A range declaration (see Section 8.11) can be used to restrict the set of distribution formats of the actual.

A pointer dummy argument may have the `DYNAMIC` attribute. In this case, the actual argument must also have the `DYNAMIC` attribute. The target associated with the dummy argument may be redistributed under the restrictions stated in the last subsection. Following Fortran rules, if the actual is also visible (through host- or use-association), the target may be redistributed only through the dummy argument. If the dummy argument is redistributed, then the actual argument has the new mapping on return from the procedure. In such a case, the new mapping must match the range restrictions (if any) of the actual.

8.8.3 Restrictions on Pointers and Targets

If, on invocation of a procedure P: (a) a dummy argument has the `TARGET` attribute, and (b) the corresponding actual argument has the `TARGET` attribute and is not an array section with a vector subscript (and therefore is an object A or a section of an array A), then the program is HPF-conforming only if:

1. No remapping of the actual argument occurs during the call; or

2. the remainder of program execution would be unaffected if

   (a) each pointer associated with any portion of A before the call were to acquire undefined pointer association status on entry to P and, if not reassigned during execution of P, were to be restored on exit to the pointer association status it had before entry.

   (b) each pointer associated with any portion of the dummy argument or with any portion of A during execution of P were to acquire undefined pointer association status on exit from P; and

Advice to users. One way of ensuring that no remapping occurs is to give the dummy argument the `INHERIT` attribute. (*End of advice to users.*)

Rationale. These restrictions are made in order to support the following part of the Fortran standard (F95:12.4.1.1) in the face of implicit remapping across the subprogram interface:

> If the dummy argument does not have the `TARGET` or `POINTER` attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument on invocation of the procedure.
>
> If the dummy argument has the `TARGET` attribute and the corresponding actual argument has the `TARGET` attribute but is not an array section with a vector subscript:
>
> 1. Any pointers associated with the actual argument become associated with the corresponding dummy argument on invocation of the procedure.
> 2. When execution of the procedure completes, any pointers associated with the dummy argument remain associated with the actual argument.

> If the dummy argument has the TARGET attribute and the corresponding actual argument does not have the TARGET attribute or is an array section with a vector subscript, any pointers associated with the dummy argument become undefined when execution of the procedure completes.

(*End of rationale.*)

Here is an example that illustrates the restrictions of this section:

```
      INTEGER, TARGET, DIMENSION (10) :: ACT
      INTEGER, POINTER, DIMENSON (:) :: POINTS_TO_ACT, POINTS_TO_DUM
!HPF$ DISTRIBUTE ACT(BLOCK)

      POINTS_TO_ACT => ACT
      CALL F(ACT)
      POINTS_TO_DUM(1) = 1             ! ILLEGAL

      CONTAINS
        SUBROUTINE F(DUM)
          INTEGER, TARGET, DIMENSION(10) :: DUM
        !HPF$ DISTRIBUTE DUM(CYCLIC)

          POINTS_TO_DUM => DUM
          POINTS_TO_ACT(1) = 1         ! ILLEGAL
        END SUBROUTINE
      END
```

The assignment to POINTS_TO_DUM(1) is illegal because it violates item 2b; the assignment to POINTS_TO_ACT(1) is illegal because it violates item 2a.

8.9 Mapping of Derived Type Components

An ALIGN, DISTRIBUTE, or DYNAMIC directive may appear within a *derived-type-def* wherever a *component-def-stmt* may appear. Every *alignee* or *distributee* within such a directive must be the name of a component defined within that *derived-type-def.* To allow mapping of the structure components, the rules have to be extended as follows:

H807	*distributee-extended*	**is**	*object-name*
		or	*template-name*
		or	*component-name*
		or	*structure-component*

A derived type is said to be an *explicitly mapped type* if any of its components is explicitly mapped or if any of its components is of an explicitly mapped type.

Constraint: A component of a derived type may be explicitly distributed only if the type of the component is not an explicitly mapped type.

Constraint: An object of a derived type may be explicitly distributed only if the derived type is not an explicitly mapped type.

Constraint: A *distributee* in a `DISTRIBUTE` directive may not be a *structure-component*.

Constraint: A *distributee* in a `DISTRIBUTE` directive which occurs in a *derived-type-def* must be the *component-name* of a component of the derived type.

Constraint: A *component-name* may occur as a *distributee* in a `DISTRIBUTE` directive occuring within the derived type definition only.

Constraint: A *distributee* that is a *structure-component* may occur only in a `REDISTRIBUTE` directive and every *part-ref* except the rightmost must be scalar (rank zero). The rightmost *part-name* in the *structure-component* must have the `DYNAMIC` attribute.

H808	*alignee-extended*	**is**	*object-name*
		or	*component-name*
		or	*structure-component*

Constraint: A component of a derived type may be explicitly aligned only if the type of the component is not an explicitly mapped type.

Constraint: An object of a derived type may be explicitly aligned only if the derived type is not an explicitly mapped type.

Constraint: An *alignee* in an `ALIGN` directive may not be a *structure-component*.

Constraint: An *alignee* in an `ALIGN` directive that occurs in a *derived-type-def* must be the *component-name* of a component of the derived type.

Constraint: A *component-name* may occur as an *alignee* only in an `ALIGN` directive occuring within the derived type definition.

Constraint: An *alignee* that is a *structure-component* may occur only in a `REALIGN` directive and every *part-ref* except the rightmost must be scalar (rank zero). The rightmost *part-name* in the *structure-component* must have the `DYNAMIC` attribute.

H809	*align-target-extended*	**is**	*object-name*
		or	*template-name*
		or	*component-name*
		or	*structure-component*

Constraint: A *component-name* may appear as an align target only in an `ALIGN` directive occuring within the derived type definition that defines that component.

Constraint: In an *align-target* that is a *structure-component*, every *part-ref* except the rightmost must be scalar (rank zero).

The above constraints imply that components of derived type can be mapped within the derived type definition itself such that when any objects of that type are created the components will be created with the specified mapping.

Consider the following example:

```
      TYPE DT
        REAL C(100)
!HPF$   DISTRIBUTE C(BLOCK) ONTO P
      END TYPE DT

      TYPE (DT) :: S1
      TYPE (DT) :: S2(100)
```

a derived type with one component, array `C`, which is specified to be distributed block. Therefore the scalar variable `S1` of derived type `DT` has a structure component `S1%C` that is distributed block onto the processor arrangement `P`. Similarly, the component `C` of each of the elements of the array `S2` will also be distributed block onto the processor arrangement `P`.

An align directive inside a derived type definition may align a component of the derived type with another component of the same derived type or with another object. A structure component can be used as a target to align other objects including components of derived types.

Example:

```
!HPF$ TEMPLATE T(100)
!HPF$ DISTRIBUTE T(CYLIC)

      TYPE DT
        REAL, DIMENSION(100) :: A, B, C
!HPF$   ALIGN WITH A :: B
!HPF$   DISTRIBUTE (BLOCK) :: A
!HPF$   ALIGN WITH T :: C
      END TYPE DT
```

Here variables of derived type `DT` will be created such the component `B` is aligned with `A`, which is itself distributed block, and such that the component `C` is aligned with a template `T` that is external to the derived type definition.

Note that if a derived type component is given a partial mapping, it is up to the compiler to choose the rest of the mapping of that component. However, it is expected that the compiler will choose the same mapping for this component of all variables of such a derived type. For example, consider a modification of the above code in which the distribution of the component `A` is omitted. `B` and `A` are specified to be aligned but no distribution is given for `A`. In such a situation, it is expected that all variables of the derived type `DT` will be created such that the component `A` (and in turn the component `B`) have the same distribution.

The constraints for the mapping of derived type components allow the mapping of structure variables at only one level. Consider for example the following code in which a derived type contains a components that is itself a derived type:

```
      TYPE SIMPLE
        REAL S(100)
!HPF$   DISTRIBUTE S(BLOCK)
      END TYPE SIMPLE
```

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, *) :: HAIRY_TEMPLATE(47,73)

      TYPE COMPLICATED
        INTEGER SIZE
        REAL RV(100,100), KV(100,100), QV(47,73)
! Arrays RV, KV, and QV may be mapped
!HPF$   DISTRIBUTE (BLOCK, BLOCK):: RV, KV
!HPF$   ALIGN WITH HAIRY_TEMPLATE :: QV
        TYPE(SIMPLE) SV(100)
! The following directive is not valid because SIMPLE
!  is an explicitly mapped type.
!HPF$   DISTRIBUTE SV(BLOCK)
      END TYPE COMPLICATED

      TYPE(COMPLICATED) LOTSOF(20)

! The following directive is not valid because COMPLICATED
! is an explicitly mapped type.
!HPF$ DISTRIBUTE LOTSOF(BLOCK)
```

Here, a component of the derived type SIMPLE has been mapped; thus objects of this type, e.g., SV in type COMPLICATED, cannot be distributed. The array LOTSOF cannot be distributed for the same reason.

Structure components having the POINTER attribute can be remapped using the REALIGN or REDISTRIBUTE directive if they have been declared DYNAMIC. For example, the following code fragment can be used to allocate and map multiple blocks (called SUBGRID here) of a multi-block grid:

```
!HPF$ PROCESSORS P( number_of_processors() )

      TYPE SUBGRID
        INTEGER SIZE
        INTEGER LO, HI          ! target subset of processors
        REAL, POINTER BL(:)
!HPF$   DYNAMIC BL
      END TYPE SUBGRID

      TYPE (SUBGRID), ALLOCATABLE :: GRID(:)

      READ (*,*) SUBGRID_COUNT
      ALLOCATE GRID(SUBGRID_COUNT)
      DO I = 1, SUBGRID_COUNT
        READ(*,*) GRID(I)%SIZE
      END DO

! Compute processor subsets for each subgrid, setting
! the LO and HI values
      CALL FIGURE_THE_PROCS ( GRID, number_of_processors())
```

```
! Allocate each subgrid and distribute to the computed processors subset
      DO I = 1, SUBGRID_COUNT
        ALLOCATE( GRID(I)%BL( GRID(I)%SIZE ) )
!HPF$   REDISTRIBUTE GRID(I)%BL(BLOCK) ONTO P( GRID(I)%LO : GRID(I)%HI )
     END DO
```

Rationale. Components of derived types can be remapped only if they have the POINTER attribute in addition to the DYNAMIC attribute. This restriction has been placed to disallow mappings which cannot be directly specified using HPF directives. Consider, for instance, the following code fragment:

```
!HPF$ PROCESSORS P(4)

      TYPE DT
        REAL C(100)
!HPF$   DISTRIBUTE C(BLOCK) ONTO P
!HPF$   DYNAMIC C                    ! Nonconforming
      END TYPE DT

      TYPE (DT) :: S(10)
       ...
      J = 3
       ...
!HPF$ REDISTRIBUTE S(J)%C(CYCLIC) ONTO P

        ... S(:)%C(2) ...
```

Here the component C of derived type DT has been declared DYNAMIC. Thus, the array variable S consists of 10 elements each of which is a structure with a component C initially distributed block. The REDISTRIBUTE directive remaps the structure component C of the Jth element of S so that it is distributed cyclic. Consider now the mapping of the data object referred to by the expression S(:)%C(2) which picks out the second element from each of the ten structures that make up the array variable S. After the redistribution of one of the elements of S (element 3 in this case), each element of the object will reside on processor P(1) except for the third element, which will reside on processor P(2). Such a distribution cannot be specified directly using HPF directives.

The Fortran standard disallows such expressions for components with the POINTER attribute. In particular, if a *part-name* in a data reference has the POINTER attribute then each *part-ref* to its left must be scalar (F95:6.1.2). Thus, we avoid the above situation by

- disallowing the remapping of components that do not have the POINTER attribute, and
- relying on the Fortran standard to disallow expressions such as the above for components with the POINTER attribute.

(*End of rationale.*)

8.10 New Distribution Formats

This section describes two new distribution formats. The syntax is extended as follows:

H810	*extended-dist-format*	**is**	`BLOCK [ (` *int-expr* `) ]`
		or	`CYCLIC [ (` *int-expr* `) ]`
		or	`GEN_BLOCK (` *int-array* `)`
		or	`INDIRECT (` *int-array* `)`
		or	`*`

Constraint: An *int-array* appearing in a *extended-dist-format* of a `DISTRIBUTE` directive or `REDISTRIBUTE` directive must be an integer array of rank 1.

Constraint: An *int-array* appearing in a *extended-dist-format* of a `DISTRIBUTE` directive must be a *restricted-expr*.

Constraint: The size of any *int-array* appearing with a `GEN_BLOCK` distribution must be equal to the extent of the corresponding dimension of the target processor arrangement.

Constraint: The size of any *int-array* appearing with an `INDIRECT` distribution must be equal to the extent of the corresponding dimension of the *distributee* to which the distribution is to be applied.

The "generalized" block distribution, `GEN_BLOCK`, allows contiguous segments of an array, of possibly unequal sizes, to be mapped onto processors. The sizes of the segments are specified by values of a user-defined integer mapping array, one value per target processor of the mapping. That is, the *ith* element of the mapping array specifies the size of the block to be stored on the *ith* processor of the target processor arrangement. Thus, the values of the mapping arrays are restricted to be non-negative numbers and their sum must be greater than or equal to the extent of the corresponding dimension the array being distributed.

The mapping array has to be a restricted expression when used in the `DISTRIBUTE` directive, but can be an array variable in a `REDISTRIBUTE` directive. In the latter case, changing the value of the map array after the directive has been executed will not change the mapping of the distributed array.

Let l and u be the lower and upper bounds of the dimension of the *distributee*, *MAP* be the mapping array and let *BS(i):BE(i)* be the resultant elements mapped to the *ith* processor in the corresponding dimension of the target processor arrangements. Then,

$$\begin{aligned} BS(1) &= l, \\ BE(i) &= \min(BS(i) + MAP(i) - 1, u), \\ BS(i) &= BE(i-1) + 1. \end{aligned}$$

Example:

```
      PARAMETER (S = /2,25,20,0,8,65/)
!HPF$ PROCESSORS P(6)
      REAL A(100), B(200), new(6)
!HPF$ DISTRIBUTE A( GEN_BLOCK( S) ) ONTO P
```

```
!HPF$ DYNAMIC  B
          ...
        new = ...
!HPF$ REDISTRIBUTE ( B( GEN_BLOCK(new) )
```

Given the above specification, array elements A(1:2) are mapped on P(1), A(3:27) are mapped on P(2), A(28:47) are mapped on P(3), no elements are mapped on P(4), A(48:55) are mapped on P(5), and A(56:100) are mapped on P(6). The array *B* is distributed based on the array *new* whose values are computed at runtime.

Advice to implementors. Accessing elements of an array distributed using the generalized block distribution may require accessing the values of the mapping array at runtime. However, since the size of such an array is equal to that of the processor arrangement, it can in most cases be replicated over all processors.

For dynamic arrays, an independent copy of the mapping array will have to be maintained internally so that a change in the values of the mapping array does not affect the access of the distributed array. (*End of advice to implementors.*)

There are many scientific applications in which the structure of the underlying domain is such that it does not map directly onto Fortran data structures. For example, in many CFD applications an unstructured mesh (consisting of triangles in 2D or tetrahedra in 3D) is used to represent the underlying domain. The nodes of such a mesh are generally represented by a one-dimensional array while another is used to represent their interconnections. Mapping such arrays using the structured distribution mechanisms, BLOCK and CYCLIC, results in mappings in which unrelated elements are mapped onto the same processor. This in turn leads to massive amounts of unnecessary communication. What is required is a mechanism to map a related set of arbitrary array elements onto the same processor. The INDIRECT distribution provides such a mechanism.

The INDIRECT distribution allows a many-to-one mapping of elements of a dimension of a data array to a dimension of the target processor arrangement. An integer array is used to specify the target processor of each individual element of the array dimension being distributed. That is, the *ith* element of the mapping array provides the processor number onto which the *ith* array element is to be mapped. Since the mapping array maps array elements onto processor elements, the extent of the mapping array must match the extent of the dimension of the array it is distributing. Also, the values of the mapping array must lie between the lower and upper bound of the target dimension of the processor arrangement.

The mapping array has to be a restricted expression when used in the DISTRIBUTE directive, but can be an array variable in a REDISTRIBUTE directive. In the latter case, changing the value of the mapping array after the directive has been executed will not change the mapping of the distributed array.

Example:

```
!HPF$ PROCESSORS P(4)
         REAL A(100), B(50)
         INTEGER map1(100), map2(50)
         PARAMETER (map1 = /1,3,4,3,3,2,1,4, ..../)
!HPF$ DYNAMIC B
!HPF$ DISTRIBUTE A( INDIRECT(map1) ) ONTO P
```

```
!HPF$ DISTRIBUTE map2(BLOCK) ONTO P

       map2 = ...
!HPF$ DISTRIBUTE B( INDIRECT(map2) ) ONTO P
       ....
```

Here, the array A is distributed statically using the constant array map1. Thus:
A(1) is mapped onto P(1),
A(2) is mapped onto P(3),
A(3) is mapped onto P(4),
A(4) is mapped onto P(3),
A(5) is mapped onto P(3),
A(6) is mapped onto P(2),
A(7) is mapped onto P(1),
A(5) is mapped onto P(4), and so on.
The array B is declared dynamic and is redistributed using the mapping array map2.

Advice to implementors. In general, the INDIRECT distribution is going to be used in the REDISTRIBUTE directive with an array variable as the map array. Also, since the size of the mapping array must be the same as the array being distributed, it will itself be distributed most likely using the BLOCK distribution. This raises several issues. To correctly implement this distribution, the runtime system should maintain a (distributed) copy of the mapping array so that if the program modifies the mapping array, the distribution does not change. Using an array variable as a mapping array implies that the location of each element of the array will not be known until runtime. Thus, a communication maybe required to figure out the location of a specific array element. (*End of advice to implementors.*)

8.11 The RANGE Directive

The RANGE attribute is used to restrict the possible distribution formats for an object or template that has the DYNAMIC attribute or a transcriptive distribution format (including pointers).

H811	*range-directive*	**is**	RANGE *ranger range-attr-stuff*
H812	*ranger*	**is**	*object-name*
		or	*template-name*
H813	*range-attr-stuff*	**is**	*range-distribution-list*
H814	*range-distribution*	**is**	(*range-attr-list*)
H815	*range-attr*	**is**	*range-dist-format*
		or	ALL
H816	*range-dist-format*	**is**	BLOCK [( )]
		or	CYCLIC [( )]
		or	GEN_BLOCK
		or	INDIRECT
		or	*

Constraint: At least one of the following must be true:

- The *ranger* has the `DYNAMIC` attribute.
- The *ranger* has the `INHERIT` attribute.
- The *ranger* is specified with a *dist-format-clause* of `*` in a `DISTRIBUTE` or combined directive.

Constraint: The length of each *range-attr-list* must be equal to the rank of the *ranger*.

Constraint: The *ranger* must not appear as an alignee in an `ALIGN` or `REALIGN` directive.

Since the length of each *range-attr-list* is the same as the rank of the *ranger*, each *range-attr*, R, in each *range-distribution* corresponds positionally to a dimension D of the ranger. This dimension D in turn either corresponds (though not necessarily positionally) to an axis A of the template with which the ranger is ultimately aligned, or corresponds to no axis in that template.

With this notation, a `RANGE` attribute on a *ranger* is equivalent to the following restriction:

For at least one *range-distribution* in the *range-distribution-list*, every *range-attr*, R, must either

- be compatible with the distribution format of the corresponding axis A, if such a corresponding axis exists, or
- be either `*` or `ALL`, if no such corresponding axis exists.

This compatibility must be maintained by any `DISTRIBUTE` or `REDISTRIBUTE` directive in which the *ranger* appears as a *distributee*, or if the ranger has the `POINTER` attribute and is transcriptively distributed, for any target with which the *ranger* becomes associated.

A distribution format of

1. `BLOCK` is compatible with a *range-dist-format* of `BLOCK`, `BLOCK()` or `CYCLIC()`;
2. `BLOCK(n)` is compatible with a *range-dist-format* of `BLOCK()`, or `CYCLIC()`;
3. `CYCLIC` is compatible with a *range-dist-format* of `CYCLIC` or `CYCLIC()`;
4. `CYCLIC(n)` is compatible with a *range-dist-format* of `CYCLIC()`;
5. `GEN_BLOCK(a)` is compatible with a *range-dist-format* of `GEN_BLOCK`;
6. `INDIRECT(a)` is compatible with a *range-dist-format* of `INDIRECT`;
7. `*` is compatible with a *range-dist-format* of `*`.

All distribution formats are compatible with a *range-dist-format* of `ALL`.

Note that the possibility of a `RANGE` directive of the form

`!HPF$ RANGE` *range-attr-stuff-list* `::` *ranger-list*

is covered by syntax rule H301 for a *combined-directive* using *combined-attribute-extended* as defined in rule H801.

Examples:

```
!HPF$    DISTRIBUTE T(BLOCK)
!HPF$    ALIGN A(I,J) WITH T(I)

         CALL SUB(A)
            ....

         SUBROUTINE SUB(X)
!HPF$    INHERIT X
!HPF$    RANGE X (BLOCK, *), (CYCLIC, *)
```

Since the ultimate align target of X, the inherited template T in this case, does not have a second dimension, only a * or ALL can be used in the second dimension of each range-distribution for X.

```
         REAL A(100, 100, 100)
!HPF$    DISTRIBUTE A(BLOCK, *, CYCLIC)

         CALL SUB( A(:,:,1) )          ! Conforming
         CALL SUB( A(:,1,:) )          ! Nonconforming
         CALL SUB( A(1,:,:) )          ! Nonconforming
            ....

         SUBROUTINE SUB(X)
         REAL X(:, :)
!HPF$    INHERIT X
!HPF$    RANGE X (BLOCK, *)
```

Given the range directive in the subroutine SUB, only the first call to SUB is conforming. However, all three calls can be made conforming if the range directive above is replaced by the following directive:

```
!HPF$    RANGE (BLOCK, *), (BLOCK, CYCLIC), (*, CYCLIC) :: X
```

8.12 The SHADOW Directive

In compiling nearest-neighbor code—for example, in discretizing partial differential equations or implementing convolutions—a standard technique is to allocate storage on each processor for the local array section so as to include additional space for the elements that have to be moved in from neighboring processors. This additional storage is referred to as "shadow edges." There are conceptually two shadow edges for each array dimension: one at the low end of the local array section and the other at the high end.

In a single routine, the compiler can tell which arrays require shadow edges and allocate this additional space accordingly. However, since the width of the shadow area is dependent on the size of the computational stencil being used, an array may require different shadow widths in different routines. Thus, without interprocedural analysis, an array argument may need to be copied into a space with the appropriate shadow width on each procedure call. A similar data motion would be required to copy the data back to its original location on exit from the subroutine. This unnecessary data motion can be avoided by allowing the user to specify the required shadow width when the array is declared.

The syntax for declaring shadow widths is as follows:

H817	*shadow-directive*	**is**	SHADOW *shadow-target shadow-attr-stuff*
H818	*shadow-target*	**is**	*object-name*
		or	*component-name*
H819	*shadow-attr-stuff*	**is**	(*shadow-spec-list*)
H820	*shadow-spec*	**is**	*width*
		or	*low-width* : *high-width*
H821	*width*	**is**	*int-expr*
H822	*low-width*	**is**	*int-expr*
H823	*high-width*	**is**	*int-expr*

Constraint: The *int-expr* representing a *width*, *low-width*, or *high-width* must be a constant *specification-expr* with value greater than or equal to 0.

A *shadow-spec* of *width* is equivalent to a *shadow-spec* of *width*:*width*. Thus, the directives

```
!HPF$    DISTRIBUTE (BLOCK) :: A
!HPF$    SHADOW (w) :: A
```

specify that the array A is distributed BLOCK and is to have a shadow width of w on both sides. If A is a dummy argument, this gives the compiler enough information to inhibit unnecessary data motion at procedure calls.

Alternatively, different shadow widths can be specified for the low end and high end of a dimension. For example:

```
         REAL, DIMENSION (1000) :: A
!HPF$    DISTRIBUTE(BLOCK), SHADOW(1:2) ::  A
         ....
         FORALL (i = 2, 998)
            A(i) = 0.25 * (A(i) + A(i-1) + A(i+1) + A(i+2))
         END FORALL
```

specifies that only one non-local element is needed at the lower end while two are needed at the high end.

8.13 Equivalence and Partial Order on the Set of Mappings

Section 4.5 has to be changed to accommodate the new distributions, the SHADOW attribute, and mapping of components of derived types, all introduced as approved extensions. The relevant text now reads as follows; additions are in **bold-face** type. ⇑

First, we define a notion of equivalence for *dist-format* specifications:

1. Using the notation ≡ for the phrase "is equivalent to",

`BLOCK`	$\equiv$	`BLOCK`	
`CYCLIC`	$\equiv$	`CYCLIC`	
`*`	$\equiv$	`*`	
`BLOCK(`*n*`)`	$\equiv$	`BLOCK(`*m*`)`	iff m and n have the same value
`CYCLIC(`*n*`)`	$\equiv$	`CYCLIC(`*m*`)`	iff m and n have the same value
`CYCLIC`	$\equiv$	`CYCLIC(1)`	
`GEN_BLOCK(`*v*`)`	$\equiv$	`GEN_BLOCK(`*w*`)`	**iff the values of corresponding elements of *v* and *w* are equal**
`INDIRECT(`*v*`)`	$\equiv$	`INDIRECT(`*w*`)`	**iff the values of corresponding elements of *v* and *w* are equal**

2. Other than this, no two lexically distinct *dist-format* specifications are equivalent.

This is an equivalence relation in the usual mathematical sense.

Next we define a notion of equivalence for `SHADOW` attributes (see Section 8.12 for the syntax):

1. **The *shadow-spec* expressions w_1 and w_2 are equivalent iff they have the same value.**
2. **The *shadow-spec* w is equivalent to the *shadow-spec* w:w.**
3. **The *shadow-spec* l_1:h_1 is equivalent to the *shadow-spec* l_2:h_2 iff l_1 is equivalent to l_2 and h_1 is equivalent to h_2.**
4. **Other than this, no two lexically distinct *shadow-spec* specifications are equivalent.**

We then say that two `SHADOW` attributes are equivalent iff the *shadow-spec-list* of one is elementwise equivalent to the *shadow-spec-list* of the other.

Now we define the partial order on mappings: Let `S` ("special") and `G` ("general") be two data objects.

The mapping of `S` is a *specialization* of the mapping of `G` if and only if either

1. `G` has the `INHERIT` attribute, or
2. `S` does not have the `INHERIT` attribute, and the following constraints all hold:
   (a) `S` is a named object **or structure component**, and
   (b) The shapes of the ultimate align targets of `S` and `G` are the same, and
   (c) Corresponding dimensions of `S` and `G` are mapped to corresponding dimensions of their respective ultimate align targets, and corresponding elements of `S` and `G` are aligned with corresponding elements of their respective ultimate align targets, and
   (d) Either
      i. The ultimate align targets of both `S` and `G` are not explicitly distributed, or

ii. The ultimate align targets of both S and G are explicitly distributed. In this case, the distribution directive specified for the ultimate align target of G must satisfy one of the following conditions:

A. It has no *dist-onto-clause*, or

B. It has a *dist-onto-clause* of "ONTO *", or

C. It has a *dist-onto-clause* specifying a processor arrangement having the same shape as that explicitly specified in a distribution directive for the ultimate align target of S.

and must also satisfy one of the following conditions:

A. It has no *dist-format-clause*, or

B. It has a *dist-format-clause* of "*", or

C. Each *dist-format* is equivalent (in the sense defined above) to the *dist-format* in the corresponding position of the *dist-format-clause* in an explicit distribution directive for the ultimate align target of S.

(e) **Either S and G both have no SHADOW attribute or they have equivalent SHADOW attributes.**

8.14 Conditions for Omitting Explicit Interfaces

The requirements in Section 4.6 are extended as follows to account for the possible presence of the DYNAMIC attribute; the addition is in **bold-face** type: ⇑

An explicit interface is required *except* when all of the following conditions hold:

1. Fortran does not require one, *and*
2. No dummy argument is distributed transcriptively or with the INHERIT attribute, *and*
3. **No dummy argument has the DYNAMIC attribute, *and***
4. For each pair of corresponding actual and dummy arguments, either:
   (a) They are both implicitly mapped, or
   (b) They are both explicitly mapped and
      i. The mapping of the actual argument is a specialization of the mapping of the dummy argument, and
      ii. If the ultimate align targets of the actual and dummy arguments are both explicitly distributed, then the *dist-onto-clause* of each must specify processor arrangements with the same shape.

   and
5. For each pair of corresponding actual and dummy arguments, either:
   (a) Both are sequential, or
   (b) Both are nonsequential.

8.15 Characteristics of Procedures

The `SHADOW` and `DYNAMIC` attributes, if present, are HPF-characteristics of dummy arguments and procedure return values. To be precise, the definitions in Section 4.7 are rewritten ⇑ as follows; additions are in **bold-face** type:

- A processor arrangement has one HPF-characteristic: its shape.
- A template has up to three HPF-characteristics:
  1. its shape;
  2. its distribution, if explicitly stated;
  3. the HPF-characteristic (i.e., the shape) of the processor arrangement onto which it is distributed, if explicitly stated.
- A dummy data object has the following HPF-characteristics:
  1. its alignment, if explicitly stated, as well as all HPF-characteristics of its align target;
  2. its distribution, if explicitly stated, as well as the HPF-characteristic (i.e., the shape) of the processor arrangement onto which it is distributed, if explicitly stated;
  3. **its `SHADOW` attribute, if explicitly stated.**
  4. **its `DYNAMIC` attribute, if explicitly stated.**
- A function result has the same HPF-characteristics as a dummy data object. Specifically, it has the following HPF-characteristics:
  1. its alignment, if explicitly stated, as well as all HPF-characteristics of its align target;
  2. its distribution, if explicitly stated, as well as the HPF-characteristic (i.e., the shape) of the processor arrangement onto which it is distributed, if explicitly stated;
  3. **its `SHADOW` attribute, if explicitly stated.**
  4. **its `DYNAMIC` attribute, if explicitly stated.**

Section 9

Approved Extensions for Data and Task Parallelism

Modern parallel machines achieve their best performance if operations are performed by many processors with each processor accessing its own data. As such, the highest-performing programs will be those for which the computation partitioning and data mapping work in synergy. Three approved extensions provide the means to exploit this symmetry:

1. The `ON` directive partitions computations among the processors of a parallel machine (much as the `DISTRIBUTE` directive partitions the data among the processors).

2. The `RESIDENT` directive asserts that certain data accesses do not require interprocessor data movement for their implementation.

3. The `TASK_REGION` construct provides the means to create independent coarse-grain tasks, each of which can itself execute a data-parallel (or nested task-parallel) computation.

All three constructs are related to the concept of *active processors*, introduced in Section 9.1 below. By assigning computations to processors, the `ON` directive (Section 9.2) defines the active processors. The `RESIDENT` directive (Section 9.3) uses this set and the information given by mapping directives in its assertions of locality. Finally, the `TASK_REGION` construct (Section 9.4) builds its tasks from active processor sets.

9.1 Active Processor Sets

Active processors are an extension of the idea of processors and processors arrangements as used in HPF 2.0. HPF 2.0 assumes that a (static) set of processors exists, and that the program uses these processors to store data (e.g., through the `DISTRIBUTE` directive) and perform computations (e.g., by execution of `FORALL` statements). Finer divisions of the processor set are seldom mentioned, although they do have uses (e.g., mapping onto processor subsets as in an approved extension, Section 8.7, or in explaining the performance of computations on subarrays). Features such as task parallelism, however, require considering a more dynamic set of processors. In particular, to answer the question "What processor(s) is (are) currently executing?" it is important to define these features.

Simply put, an active processor is one that executes an HPF statement (or group of statements). Active processors perform all operations required to execute the statement(s) *except* (perhaps) for the initial access of data and writing of results. Some operations require certain processors to be active, as described below, but for the most part any processor can be active in the execution of any statement. An HPF program begins execution with all processors active. As described in Section 9.2, the ON directive restricts the active processor set for the duration of execution of statements in its scope. Consider this simple example (which has a reasonably intuitive meaning):

```
!HPF$ ON HOME( Z(INDX) )
        X(INDX-1) = X(INDX-1) + Y(INDX) * Z(INDX+1)
```

Let X, Y, and Z have the same distribution, which does not replicate data. Following the ON directive, the statement would be executed as follows:

1. The processor owning Z(INDX) is identified as the active processor. On different executions of this ON block, this may be a different processor.

2. The values of X(INDX-1), Y(INDX), and Z(INDX+1) are made available to the active processor. Because of the identical distributions, Y(INDX) is already stored there. Depending on the data distribution and the hardware running the program, retrieving the others might correspond to the active processor loading registers from memory, or it might mean one or two other processors sending messages to the active processor.

3. The active processor performs an addition and a multiplication, using the values sent in the last step.

4. The result is stored to X(INDX-1), which may be on another processor. Again, this may require synchronization or other cross-processor operations.

There are considerable subtleties of this scheme when one of the statements involved is a function or subroutine call. Section 9.2.4 deals with these cases. Advice on the implementation of the ON directive is given in Section 9.2.2 below.

A few additional terms are useful in conjunction with the concept of active processors. If all processors in a set are active, then the set is called an *active processor set*. The set of all active processors is sometimes called *the* active processor set. This set is dynamic, and if a statement is executed repeatedly the active processor set may be different each time. In general, an HPF construct can only restrict the active set, not enlarge it. However, if the original active set is partitioned into several independent sets, all partitions may execute simultaneously. This is exactly how the TASK_REGION construct (described in Section 9.4) works.

The *universal* processor set is the set of all processors available to the HPF program. It is precisely the set of processors that is active when execution of the main program begins.

A processor that is not in the active set is called *inactive*. (Note that a processor may be inactive with respect to one statement, but active with respect to another. This is common in TASK_REGION constructs.)

It is sometimes necessary to query properties of the active processor set; this is accomplished by the approved extension intrinsics ACTIVE_NUM_PROCS and ACTIVE_PROCS_SHAPE described in Section 12.1.

The data mapped to a processor is said to be *resident* on it. A replicated object is resident on all of the processors that have a copy of it.

Rationale. It may seem odd at first to concentrate on shrinking the active processor set. However, HPF's design assumes that all processors are available at the beginning of execution. For example, implementing `DISTRIBUTE` requires information about the number of processors (in order to determine block sizes, for example) and their identity (in order to allocate the memory and perform data motion). Therefore, the execution model uses a static set of processors that can be subdivided and reunited dynamically. (*End of rationale.*)

9.1.1 The SUBSET Directive

This subsection explains the interaction of active and inactive processor sets with explicitly mapped data. The rule of thumb is that allocating memory must be done locally; that is, if a processor stores part of an array, then that processor must be active when the array is created. Implications of this rule include:

- Local objects must be stored on a set of active processors, either when their subprogram is invoked or when they are allocated.
- Dummy arguments are always mapped to a set of active processors. Section 9.2.4 explains the mechanism that ensures this.
- Global objects (i.e., objects in `COMMON` or `MODULE`s or objects accessed via host association) may be explicitly mapped to inactive processors. However, those processors must have been active when the globals were allocated, whether at program initialization (when all processors were active), or at on entry to another subprogram, or on execution of an `ALLOCATE` statement.

It should be clear from the treatment of local and global objects that declarations may need to refer to two classes of processors arrangements. The first, used mainly for the declaration of global data, consists of arrangements of the universal processor set. These are known as *universal* processors arrangements. Since they always represent the same processors, these serve as a fixed frame of reference, allowing consistent declarations. A *processors-directive* (Rule H329) defines a universal processors arrangement by default. To accommodate active processors, two slight changes to the rules in Section 3.6 need to be made: ⇑

- An HPF compiler is required to accept any *universal* processors arrangement that is scalar, or whose size (i.e., product of the arrangement's dimensions) is equal to the size of the universal processor set.
- If two *universal* processors arrangements have the same shape, then corresponding elements of the two arrangements are understood to refer to the same abstract processor.

In both cases, the only change is the addition of the word "universal."

Restricted processors arrangements represent only processors that, at the time the arrangement is declared, are active. They are used for mapping local objects and dummy arguments. To declare a subset processors arrangement, one can use the `SUBSET` option of *combined-attribute-extended* (H801), defined on page 145. One can also use the statement form of the `SUBSET` attribute:

H901 *subset-directive* **is** `SUBSET` *processors-name*

Examples of the two forms are

```
!HPF$ PROCESSORS, SUBSET :: P(NP/4,4)
!HPF$ PROCESSORS Q(ACTIVE_NUM_PROCS())
!HPF$ SUBSET Q
```

As for universal arrangements, there are some modified rules for the use of subset processors arrangements:

- An HPF compiler is required to accept any *subset* processors arrangement that is scalar, or whose size is equal to the number of active processors, i.e., the number that would be returned by the call `ACTIVE_NUM_PROCS()`.
- If two *subset* processors arrangements are declared with the same shape and the active processor set has not changed between their declarations, then corresponding elements of the two arrangements are understood to refer to the same abstract processor.

It is important to note that a scalar subset processors arrangement is considered to represent a processor that is active at the time the arrangement is created.

Note that it is permitted for a subset processors arrangements to have fewer than `NUMBER_OF_PROCESSORS()` elements; this reflects the way that the active processor set can shrink. Also note that there is an added condition before two subset processors arrangements are considered identical; this reflects the dynamic nature of the active processor set. Finally, note that a local, subset processors arrangement will be an arrangement of the set of active processors until such time as the active processor set is further restricted by an `ON` directive.

9.1.2 Mapping Local Objects and Dummy Arguments

For explicitly mapped local objects without the `SAVE` attribute, the declarations must map all elements of the object onto active processors. This requirement gives rise to several cases:

- If the local object is mapped via a `DISTRIBUTE` directive, then it must be distributed onto a set of active processors. One way, but not the only way, is to use a local, subset processor arrangement as the *dist-target*. If there is no explicit `ONTO` clause, the implementation is free to choose any arrangement of active processors as the *dist-target*.
- A local, universal processors arrangement of size one is always identified with an active processor, and may occur as the *dist-target* for a local object.
- If the local object is mapped by an `ALIGN` directive, then the corresponding elements of the ultimate align target must be distributed exclusively onto active processors. This certainly occurs if the whole of the ultimate align target is distributed onto active processors. It also occurs if the local object is aligned to a section of a target that is distributed onto both active and inactive processors, provided the section that is "hit" by the aligned object is mapped only to active processors. If the align replicates the alignee over one or more axes of the align target, then the distribution of the align target must ensure that all copies of the alignee are mapped to active processors.

In any of these cases, the active processor set is determined at the time that the DISTRIBUTE or ALIGN becomes instantiated. That is, the mapping directives for ALLOCATABLE variables are instantiated when the variable is allocated; other objects have their mapping instantiated when they are declared.

The declaration of subset processors arrangements does not cause processors to become active or inactive; only the execution of ON directives does that. In particular, if a program contains no ON directives or constructs that modify a program's active processor set, then all processors are always active and all DISTRIBUTE directives can use universal arrangements.

Explicitly mapped global objects must have consistent mappings wherever they appear. This will usually (for COMMON and USE associated objects) be accomplished by distribution onto universal processors arrangements. Notice that the interpretation of an implicit (i.e., missing) ONTO clause differs for local and global objects; globals may be distributed onto all processors, while locals must use only active processors. Also note that, since universal processors arrangements are the default for the PROCESSORS directive, no modification to the mapping of global objects is needed when active processors are introduced.

Dummy arguments must be explicitly mapped in the same way as local objects, using the rules above. As Section 9.2.4 explains, the effect of this is that dummy arguments are always stored on the active processor set. Other data objects, particularly objects local to the subprogram, can therefore be aligned to the dummy arguments and allocated on the active processor set.

Objects with the SAVE attribute must be mapped consistently whenever they come into scope. They are not subject to the restriction of mapping to active processors; where mapping is concerned, they conform to the same rules as global objects.

9.1.3 Other Restrictions on Active Processors

In addition to the mapping of locals and dummy arguments, several other constructs are restricted when the active processor set does not match the universal processor set. In general, the intent of these restrictions is to ensure that all processors that are needed for an operation are active when it is performed. In particular, allocating or freeing memory mapped to a processor requires the cooperation of that processor.

For a REDISTRIBUTE directive, the active processor set must include:

- All processors that stored any element of the *distributee* before the REDISTRIBUTE was encountered, and

- The processors that will store any element of the *distributee* after the REDISTRIBUTE is performed.

This implies that all elements of the redistributed object reside on active processors, both before and after the REDISTRIBUTE operation. Effectively, this means that all data movement for the REDISTRIBUTE will be among active processors. In addition, the processors that owned the *distributee* (or anything aligned to it) beforehand can free the memory, and processors that now own the *distributee* can allocate memory for it.

Similarly, for a REALIGN directive, the set of active processors must include all processors that stored elements of the *alignee* before the REALIGN and all processors that will store *alignee* elements after the REALIGN.

For an ALLOCATE statement that creates an explicitly mapped object, the set of active processors must include the processors used by the mapping directive for the allocated object. The allocated object's ultimate align target may fall into one of two classes:

- Distributed with no explicit ONTO clause. (This case includes ultimate align targets with no DISTRIBUTE directive at all.) In this case, the compiler must choose a set of active processors that the object will be stored on.

- Distributed ONTO a section of a processors arrangement. In this case, the specified section must be an arrangement of an active processor set.

For example:

```
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
!HPF$ ON (P(1:4))
      CALL OF_THE_WILD()
      ...

      SUBROUTINE OF_THE_WILD()
      INTEGER, ALLOCATABLE, DIMENSION(:) :: A, B, C, D, E, F
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS()), ONE_P
!HPF$ PROCESSORS, SUBSET :: Q(ACTIVE_NUM_PROCS())
!HPF$ DISTRIBUTE (BLOCK) :: A, E
!HPF$ DISTRIBUTE (BLOCK) ONTO P(1:4) :: B
!HPF$ DISTRIBUTE (*) ONTO ONE_P :: C
!HPF$ DISTRIBUTE (BLOCK) ONTO Q :: D, F

      ALLOCATE (A(100))   ! No explicit ONTO; block size is probably 25
      ALLOCATE (B(100))   ! Block size IS 25
      ALLOCATE (C(100))   ! On one active processor
      ALLOCATE (D(100))   ! On Q(1:4); block size 25
!HPF$ ON HOME(B(1:50)) BEGIN
      ALLOCATE (E(100))   ! No ONTO; E is allocated on Q(1:2)
      ALLOCATE (F(100))   ! Nonconforming since Q(3:4) are inactive
!HPF$ END ON
```

For a DEALLOCATE statement that destroys an explicitly mapped object, the active processor set must include all processors that own any element of that object. Again, there are two cases for the deallocated object's ultimate align target:

- Distributed onto a section of a processors arrangement. In this case, the processors that store part of the object must be active when it is deallocated. One way to guarantee this is to ensure that any ON block enclosing the DEALLOCATE statement also encloses the corresponding ALLOCATE.

- Distributed with no explicit ONTO clause. (This case includes ultimate align targets with no DISTRIBUTE directive at all.) In this case, the active processor set must include all the processors that were active when the object was allocated in order to guarantee that the processors that store part of the object are active when it is deallocated. Again, this is ensured if all ON blocks that enclose the deallocation also enclose the allocation operation.

An example may be helpful:

```
      REAL, ALLOCATABLE :: X(:), Y(:)
!HPF$ PROCESSORS P(8)
!HPF$ DISTRIBUTE X(BLOCK) ONTO P(1:4)
!HPF$ DISTRIBUTE Y(CYCLIC)

!HPF$ ON ( P(1:6) )
!HPF$     ON ( P(1:5) )
              ALLOCATE( X(1000), Y(1000)
!HPF$         ON ( P(1:3) )
                  ! Point 1
!HPF$         END ON
              ! Point 2
!HPF$     END ON
         ! Point 3
!HPF$ END ON
      ...
!HPF$ ON ( P(1:4) )
          ! Point 4
!HPF$ END ON
      ! Point 5
```

At point 1, neither X nor Y can be deallocated, since some of the processors that store their elements might not be active. If the innermost directive were

```
!HPF$       ON ( P(1:4) )
```

then X could be safely deallocated because of its explicit ONTO clause; it would still be incorrect to deallocate Y. At points 2 and 3, both X and Y can safely be deallocated. In general, if the deallocation occurs at the same level of ON nesting or at an outer level and the flow of control has not left the outer ON construct, then the deallocation is safe. At point 4 it is correct to deallocate X because its ONTO clause matches the enclosing ON. It is not, however, correct to deallocate Y, since some processors (e.g., P(5)) that were active at the ALLOCATE statement are not active at point 4. This illustrates the care that must be exercised if a DEALLOCATE statement is controlled by an ON clause. One can avoid potential problems by performing the deallocation outside of any ON construct in the same procedure, as at point 5.

It is possible that only of subset of the processors active at allocation time and named in the ONTO clause actually store part of the object:

```
!HPF$ DISTRIBUTE A(BLOCK(10)) ONTO P(1:4)
      INTEGER, ALLOCATABLE :: A(:)
      ALLOCATE A(10)
!HPF$ ON (P(1))
      DEALLOCATE(A)     ! Correct, because only P(1) owns any part of A
```

9.2 The ON Directive

The purpose of the ON directive is to allow the programmer to control the distribution of computations among the processors of a parallel machine. In a sense, this is the computational analog of the DISTRIBUTE and ALIGN directives for data. The ON directive does this by specifying the active processor set for a statement or set of statements. This temporarily shrinks the active processor set.

If the computations in two `ON` block executions are not related (for example, if the `ON` block executions are two iterations of an `INDEPENDENT` loop), their `ON` directives give the compiler clear instructions for exploiting this potential parallelism.

9.2.1 Syntax of the ON Directive

There are two flavors of the `ON` directive: a single-statement form and a multi-statement form. The syntax for these directives is

H902	*on-directive*	**is**	`ON` *on-stuff*
H903	*on-stuff*	**is**	*home* [, *resident-clause*] [, *new-clause*]
H904	*on-construct*	**is**	*directive-origin block-on-directive* *block* *directive-origin end-on-directive*
H905	*block-on-directive*	**is**	`ON` *on-stuff* `BEGIN`
H906	*end-on-directive*	**is**	`END ON`
H907	*home*	**is**	`HOME (` *variable* `)`
		or	`HOME (` *template-elmt* `)`
		or	`(` *processors-elmt* `)`
H908	*template-elmt*	**is**	*template-name* [(*section-subscript-list*)]
H909	*processors-elmt*	**is**	*processors-name* [(*section-subscript-list*)]

The nonterminal *resident-clause* will be defined in Section 9.3. For the present, it suffices to say that this is a form of the `RESIDENT` directive mentioned in the introduction.
The *home* is often called the `HOME` clause, even in cases where the keyword `HOME` is not used. Note that *variable* is a Fortran syntax term that means (roughly) "a reference, including an array element, array section, or derived type field"; *variable* does not include template or processor elements because they are defined only in directives. Note also that *block* is a Fortran syntax term for "a series of statements treated as a group"—for example, the body of a `DO` construct.

The *on-directive* is a kind of *executable-directive* (see rule H205). This means that an *on-directive* can appear wherever an executable statement can.

An *on-construct* is a Fortran *executable-construct*. This syntax implies that such constructs can be nested, and if so they will be properly nested.

Rationale. Note the use of parentheses in the last option of the *home* rule (involving *processors-elmt*). This prevents the following ambiguity:

```
      INTEGER X(4)         ! X(I) will be on processor I
!HPF$ PROCESSORS HOME(4)
!HPF$ DISTRIBUTE X(BLOCK)
      X = (/ 4,3,2,1 /)
```

```
    !HPF$ ON HOME(X(2))
              X(2) = X(1)
```

If the parentheses were not required, where should the computation be done?

1. Processor `HOME(2)` (i.e., the owner of `X(2)`)?
2. Processor `HOME(3)` (i.e., use the value of `X(2)`, before the assignment)?
3. Processor `HOME(4)` (i.e., use the value of `X(2)`, after the assignment)?

The definition of `ON` clearly indicates that interpretation 1 is correct. One can get the effect of interpretation 2 by the directive

```
    !HPF$ ON(HOME(X(2)))
```

There is no way to get the effect of interpretation 3. Introducing reserved keywords into Fortran was suggested as a better solution to this problem, but was seen as too large a change to the underlying language. (*End of rationale.*)

9.2.2 Semantics of the ON Directive

The `ON` directive restricts the active processor set for a computation to those processors named in its *home*. The computation controlled is either the following Fortran statement (for a *on-directive* or the contained *block* for a *block-on-directive*. We refer to the controlled computation as the `ON`-block.

That is, it advises the compiler to use the named processor(s) to perform the `ON` block. Like the mapping directives `ALIGN` and `DISTRIBUTE`, this is advice rather than an absolute commandment; the compiler may override an `ON` directive. Also like `ALIGN` and `DISTRIBUTE`, the `ON` directive may affect the efficiency of computation, but not the final results.

Advice to implementors. If the compiler may override the user's advice in an `ON` directive, then the compiler should also offer the user an option to force all directives to be obeyed. Because dummy arguments and local objects are required to be mapped onto active processors, an HPF compiler that fails to heed the programmer's advice with respect to the active processor set may also be required to ignore some of the programmer's advice concerning data mapping. (*End of advice to implementors.*)

The single-statement `ON` directive sets the active processor set for the first non-comment statement that follows it. It is said to apply to that statement. If the statement is a compound statement (e.g., a `DO` loop or an `IF-THEN-ELSE` construct), then the `ON` directive also applies to all statements nested therein. Similarly, the `ON` construct applies the initial `ON` clause to—i.e., sets the active processor set for—all statements up to the matching `END ON` directive.

The evaluation of any function referred to in the home expression is not affected by the `ON` directive; these functions are called on all processors active when control reached the directive. Thus,

```
  !HPF$ ON HOME( P(1: (ACTIVE_NUM_PROCS() - 1)) ) ...
```

is a reasonable way to idle one active processor, and is not paradoxically self-referential.

The HOME clause can name a program object, a template, or a processors arrangement. For each of these possibilities, it can specify a single element or multiple elements. This is translated into the processor(s) executing the ON block as follows:

- If the HOME clause names a program object, then every processor owning any part of that object should execute the ON block. For example, if A is an explicitly mapped array, then

  ```
  !HPF$ ON HOME ( A(2:4) )
  ```

  tells the compiler to perform the statement on the processors owning A(2), A(3), and A(4). If A were distributed BLOCK, this might be one processor; if it were distributed CYCLIC, it would be three processors (assuming that many processors were available). Extra copies of elements created by a SHADOW directive (H817) are not taken into consideration by the HOME clause.

- If the HOME clause names a template element or section, then every processor owning any element of the template element or section should execute the ON block. The example above applies here as well, if A is a template rather than an array.

- If the HOME clause names a processors arrangement, then the processor(s) referenced there should execute the ON block. For example, if P is a processors arrangement, then

  ```
  !HPF$ ON ( P(2:4) )
  ```

  will execute the following statement on the three processors P(2), P(3), and P(4).

In every case, the ON directive specifies the processor(s) that should perform a computation. More formally, it sets the active processors for the statements governed by the ON directive, as described in Section 9.1. That section also describes how some statements (notably ALLOCATE and dynamic remapping directives) require that particular processors be included in the active set. If one of these constructs occurs in the ON block and the active processor set does not contain all the required processors, then the program is not standard-conforming.

Note that the ON directive only specifies how *computation* is partitioned among processors; it does not indicate processors that may be involved in data transfer. Also, the ON clause by itself does not guarantee that its body can be executed in parallel with any other operation. However, placing the computation can have a significant effect on data locality. As later examples will show, the combination of ON and INDEPENDENT can also provide control over the load balance of parallel computations.

Advice to implementors. If the HPF program is compiled into Single-Program-Multiple-Data (SPMD) code, then the ON clause can always be implemented (albeit inefficiently) by having all processors compare their processor *id* to an *id* (or list of *ids*) generated from the HOME clause. (Similar naive implementations can be constructed in other paradigms as well.) If the ON clause will be executed repeatedly, for example in a DO loop, it is worthwhile to invert this process. That is, instead of all processors executing all the HOME clause tests, the compiler should determine the range of loop iterations that will test true on the given processor. (See the "Advice to implementors" in Section 9.2.3 for more details.) For example, consider the following complex case:

```
        DO I = 1, N
            !HPF$ ON HOME( A(MY_FCN(I)) ) BEGIN
                ...
            !HPF$ END ON
        END DO
```

Here, the generated code can perform an "inspector" (i.e., a skeleton loop that only evaluates the HOME clause of each iteration) to produce a list of iterations assigned to each processor. This list can be produced in parallel, since MY_FCN must be side-effect free (at least, the programmer cannot rely on any side effects). However, distributing the computation of *home* to all processors may require unstructured communications patterns, possibly negating the advantage of parallelism. In general, more advanced compilers will be able to efficiently invert more complex HOME clauses. It is recommended that the abilities (and limitations) of a particular compiler be documented clearly for users.

Note that processors "screened out" by the naive implementation may still be required to participate in data transfer. If the underlying architecture allows one-sided communication (e.g., shared memory or GET/PUT), this is not a problem. On message-passing machines, a request-reply protocol may be used. This requires the inactive processors to enter a wait loop until the ON block completes, or requires the runtime system to handle requests asynchronously. Again, it is recommended that the documentation tell programmers which cases are likely to be efficient and which inefficient on a particular system. (*End of advice to implementors.*)

Advice to users. The form of the *home* in an ON directive can be arbitrarily complex. This is a two-edged sword; it can express very complicated computation partitioning, but the implementation of these partitions may not be efficient. More concretely, it may express a perfectly load-balanced computation, but force the compiler to serialize the computation to implement the HOME clauses. Although the amount of overhead for an ON clause will vary based on the HPF code, the compiler, and the hardware, one can expect that compilers will generate very good code based solely on array mappings or a named processors arrangement, and progressively worse code as the complexity of the *home* increases. A rough measure of the complexity of an ON directive is the amount of run-time data used to compute it; for example, a constant offset is fairly simple, while a permutation array is very complex. See Section 9.2.3 below for more concrete examples of this phenomenon.

It should also be noted that the ON clause does not change the semantics of a program, in the same sense that DISTRIBUTE does not change semantics. In particular, an ON clause *by itself* does not change sequential code into parallel code, because the code in the ON block can still interact with code outside the ON block. (To put it another way, ON does not spawn processes.) (*End of advice to users.*)

It is legal to nest ON directives, if the set of active processors named by the inner ON directive is included in the set of active processors from the outer directive. The syntax of *on-construct* automatically ensures that it is properly nested inside other compound statements, and that compound statements properly nest inside of it. As with other Fortran compound statements, transfer of control to the interior of an *on-construct* from outside the block is prohibited: an *on-construct* may be entered only by executing the (executable) ON directive.

Transfers within a block may occur. However, HPF *also* prohibits transfers of control from the interior of an *on-construct* to outside the *on-construct*, except by "falling through" the `END ON` directive. Note that this is stricter than in ordinary Fortran. If `ON` clauses are nested, then the innermost *home* effectively controls execution of the statement(s). A programmer can think of this as successively restricting the set of processors at each level of `ON` nesting; clearly, the last restriction must be the strongest. Alternately, the programmer can think of this as a fork-join approach to nested parallelism.

Rationale. The restrictions about control flow into and out of an `ON` block essentially make it a single-entry single-exit region, thus simplifying the semantics considerably. (*End of rationale.*)

If an `ON` directive includes a `NEW` clause, the meaning is the same as a `NEW` clause in an `INDEPENDENT` directive. The operation of the program would be identical if the `NEW` variables were allocated anew, and distributed onto the active processors, on every entry to the `ON` directive's scope, and deallocated on exit from the `ON` block. That is, the `NEW` variables are undefined on entry (i.e., assigned before use in the `ON` block) and undefined on exit (i.e., not used after the `ON` block, unless first reassigned). In addition, `NEW` variables cannot be remapped in the `ON` clause's scope, whether by `REALIGN`, `REDISTRIBUTE`, or by argument association (at subroutine calls). If a variable appears in a `NEW` clause but does not meet these conditions, then the program is not HPF-conforming. `NEW` variables are not considered by any nested `RESIDENT` directives, as detailed in Section 9.3.

The `NEW` variables are implictly reallocated and remapped onto the active processors on entry to the `ON` block. For this reason, there are restrictions on their explicit mappings.

- An `ON` block `NEW` variable may not occur as an alignee.
- An `ON` block `NEW` variable may occur as a distributee only if there is no `ONTO` clause.

```
!HPF$ DISTRIBUTE X(BLOCK, *)
!HPF$ DISTRIBUTE Y ONTO P              ! Nonconforming due to ONTO clause
!HPF$ ALIGN WITH X :: Z                ! Nonconforming; ALIGN forbidden
!HPF$ ON (P(1:4), NEW(X, Y, Z), BEGIN
!HPF$ END ON
```

Rationale. `NEW` clauses provide a simple way to create temporary variables. This ability is particularly important when `RESIDENT` directives come into play, as will be clear below. (*End of rationale.*)

Advice to implementors. Because they are not used outside of the `ON` blocks, `NEW` variables need not be kept consistent before and after `ON` clauses. Therefore, no communication outside of the active processor set, determined by the `ON` directive, is required to implement them. Scalar `NEW` variables should be replicated over the active processor set, or allocated in memory areas shared by the active processor set. Note that memory must be dynamically allocated if there is a possibility that multiple instances of the `ON` block could be active concurrently. This is similar to the requirements for implementing `NEW` variables in `INDEPENDENT` loops. (*End of advice to implementors.*)

9.2.3 Examples of ON Directives

The following are valid examples of ON directives. Most of them illustrate idioms that programmers might want to use, rather than contrived situations. For simplicity, the first several examples assume the following array declarations:

```
REAL A(N), B(N), C(N), D(N)
!HPF$ DISTRIBUTE A(BLOCK), B(BLOCK), C(BLOCK), D(BLOCK)
```

One of the most commonly requested capabilities for HPF is to control how loop iterations were assigned to processors. (Historically, the ON clause first appeared to perform exactly this role in the Kali FORALL construct.) This can be done by the ON directive, as shown in the following examples:

```
!HPF$ INDEPENDENT
DO I = 2, N-1
  !HPF$ ON HOME(A(I))
  A(I) = (B(I) + B(I-1) + B(I+1))/3
END DO

!HPF$ INDEPENDENT
DO J = 2, N-1
  !HPF$ ON HOME(A(J+1)) BEGIN
   A(J) = B(J+1) + C(J+1) + D(J+1)
  !HPF$ END ON
END DO
```

The ON directive in the I loop sets the active processor for each iteration of the loop to be the processor that stores A(I). In other words, it advises the compiler to have each processor run over its own section of the A array (and therefore B as well). The references to B(I-1) and B(I+1) must be fetched from off-processor for the first and last iterations on each processor (except for the boundary processors); note that those processors are not mentioned in the HOME clause. The ON directive in the J loop similarly sets the active set for each iteration, but advises the compiler to shift computations. As a result, each processor does a vector sum of its own sections of B, C, and D, stores the first element of the result on the processor to its left, and stores the rest of the result (shifted by one) in A. It is worth noting that the directives would still be valid (and minimize nonresident data accesses) if the arrays were distributed CYCLIC, although the number of nonresident references would be much higher.

Advice to implementors. It is highly recommended that compilers concentrate on optimizing DO loops with a single ON clause including the entire loop body. Schematically, the code will be:

```
DO i = lb, ub, stride
  !HPF$ ON HOME(array(f(i))) BEGIN
     body
  !HPF$ END ON
END DO
```

where array has some data mapping. Assume the mapping gives processor p the elements myset(p). (In a BLOCK distribution, for example, myset(p) is a contiguous range of integers.) Then the generated code on processor p should be

```
DO $i \in [lb : ub : stride] \cap f^{-1}(myset(p))$
    body
END DO
```

(This schematic does not show where communication or synchronization must be placed; that must be derived from analysis of the body.) Moreover, f is likely to be the identity function or a linear function with integer coefficients, both of which can be inverted easily. Given this, techniques for iterating through the set can be found in several recent conferences. (*End of advice to implementors.*)

Advice to users. One can expect the I loop above to generate efficient code for the computation partitioning. In effect, the compiler will arrange for each processor to iterate over its own section of array A. The J loop is slightly more complex, since the compiler must find the inverse of the HOME clause's subscripting function. That is, the compiler must solve K=J+1 for J, where K ranges over the resident elements of A. Of course, in this case J=K-1; in general, linear functions can be inverted by the compiler. (It should be pointed out, however, that complex combinations of ALIGN and DISTRIBUTE may make the description of K unwieldy, and this may add overhead to the inversion process.) (*End of advice to users.*)

Sometimes it is advantageous to "split" an iteration between processors. The following case shows one example of this:

```
!HPF$ INDEPENDENT
DO I = 2, N-1
  !HPF$ ON HOME(A(I))
  A(I) = (B(I) + B(I-1) + B(I+1))/3
  !HPF$ ON HOME C(I+1)
  C(I+1) = A(I) * D(I+1)
END DO
```

Here, the active processor sets for the two statements in the loop body are different. Due to the first ON clause, the reference to A(I) is resident in the first statement. The second ON clause makes A(I) nonresident (for some values of I) there. This maximizes the data locality in both statements, but does require data movement between the two.

Advice to implementors. If there are several non-nested ON clauses in a loop, then the schematic above needs to be generalized. In essence, the iteration range for each individual ON clause must be generated. A processor will then iterate over the union of these ranges. Statements guarded by an ON directive must now be guarded by an explicit test. In summary, the code for

```
DO $i$ = $lb$, $ub$, $stride$
    !HPF$ ON HOME($array_1$($f_1$($i$)))
    $stmt_1$
    !HPF$ ON HOME($array_2$($f_2$($i$)))
```

```
        $stmt_2$
    END DO
```

on processor p becomes

```
$set_1 = [lb : ub : stride] \cap {f_1}^{-1}(myset_1(p))$
$set_2 = [lb : ub : stride] \cap {f_2}^{-1}(myset_2(p))$
DO $i \in set_1 \cup set_2$
    IF ($i \in set_1$) THEN
        stmt1
    END IF
    IF ($i \in set_2$) THEN
        stmt2
    END IF
END DO
```

where $myset_1(p)$ is the resident set for $array_1$, and $myset_2(p)$ is the resident set for $array_2$. (Again, synchronization and communication must be handled by other means.) Code transformations such as loop distribution and loop peeling can be used to eliminate the tests in many cases. They will be particularly profitable if there are data dependences between the ON blocks. (*End of advice to implementors.*)

Advice to users. Splitting an iteration like this is likely to require either additional tests at runtime or additional analysis by the compiler. Even if the compiler can generate low-overhead scheduling for the individual ON clauses, combining them is not necessarily low-overhead. The locality benefits must be rather substantial for this to pay off, but there are cases where multiple ON clauses are valuable. (All these statements are particularly true if one ON block uses data computed in another one.) (*End of advice to users.*)

Because ON clauses nest naturally, they can be useful for expressing parallelism along different dimensions. Consider the following examples:

```
REAL X(M,M)
!HPF$ DISTRIBUTE X(BLOCK,BLOCK)

!HPF$ INDEPENDENT, NEW(I)
DO J = 1, M
  !HPF$ ON HOME(X(:,J)) BEGIN
    DO I = 2, M
      !HPF$ ON HOME(X(I,J))
      X(I,J) = (X(I-1,J) + X(I,J)) / 2
    END DO
  !HPF$ END ON
END DO
```

The active processor set for each iteration of the J loop is a column of the (presumably universal) processors arrangement. The I loop further subdivides the computation, giving each processor responsibility for computing the elements it owns. Many compilers would have chosen this computation partitioning automatically for such a simple example. However, the compiler might have attempted to fully parallelize the outer loop, executing each inner loop sequentially on one processor. (This might be attractive on a machine with very fast communications.) By inserting the ON clauses, the user has advised against this strategy, thus trading additional locality for restricted parallelism. Notice that the ON directive neither requires nor implies the INDEPENDENT assertion. In both nests, each iteration of the I loop depends on the preceding iteration, but the ON directive can still partition the computation among processors. The ON directive does not automatically make a loop parallel.

Advice to implementors. "Dimension-based" nesting, as above, will probably be a common case. The HOME clauses can be inverted at each level, treating indices from outer loops as run-time invariants. (*End of advice to implementors.*)

Advice to users. Nested ON directives will tend to have efficient implementations if their HOME clauses refer to different dimensions of the processors arrangements, as in the above example. This minimizes the interaction between the levels of the loops, simplifying the implementation. (*End of advice to users.*)

Consider the following variation on the above example:

```
!HPF$ DISTRIBUTE Y(BLOCK,*)

!HPF$ INDEPENDENT, NEW(I)
DO J = 1, M
  !HPF$ ON HOME(Y(:,J)) BEGIN
    DO I = 2, M
      !HPF$ ON HOME(Y(I,J))
      Y(I,J) = (Y(I-1,J) + Y(I,J)) / 2
    END DO
  !HPF$ END ON
END DO
```

Note that the ON clauses have not changed, except for the name of the array. The interpretation is similar to the above, except that the outer ON directive assigns each iteration of the J loop to all of the processors. The inner ON directive again implements a simple owner-computes rule. The programmer has directed the compiler to distribute a serial computation across all the processors. There are a few scenarios where this is more efficient than parallelizing the outer loop:

1. Parallelizing the outer loop will generate many nonresident references, since only a part of each column is on any processor. If nonresident references are very expensive (or if M is relatively small), this overhead may outweigh any gain from parallel execution.

2. The compiler may take advantage of the INDEPENDENT directive to avoid inserting any synchronization. This allows a natural pipelined execution. A processor will execute

its part of the I loop for one value of J, then immediately go on to the next J iteration. Thus, the first processor will start on J=2 while the second receives the data it needs (from processor one) for J=1. (A similar pipeline would develop in the X example above.)

Clearly, the suitability of these ON clauses will depend on the underlying parallel architecture.

Advice to users. This example points out how ON may improve software engineering. While the "value" of HOME(X(I)) will change if X's mapping changes, its intent will usually stay the same - run the loop "aligned with" the array X. Moreover, the form of the clauses is portable, and they simplify experimenting with alternative computation partitioning. Both qualities are similar to the advantages of DISTRIBUTE and ALIGN over low-level data layout mechanisms. (*End of advice to users.*)

ON directives are particularly useful when the compiler cannot accurately estimate data locality, for example when the computation uses indirection arrays. Consider three variations of the same loop:

```
REAL X(N), Y(N)
INTEGER IX1(M), IX2(M)
!HPF$ DISTRIBUTE X(BLOCK), Y(BLOCK)
!HPF$ DISTRIBUTE IX(BLOCK), IY(BLOCK)

!HPF$ INDEPENDENT
DO I = 1, N
  !HPF$ ON HOME( X(I) )
  X(I) = Y(IX(I)) - Y(IY(I))
END DO

!HPF$ INDEPENDENT
DO J = 1, N
  !HPF$ ON HOME( IX(J) )
  X(J) = Y(IX(J)) - Y(IY(J))
END DO

!HPF$ INDEPENDENT
DO K = 1, N
  !HPF$ ON HOME( X(IX(K)) )
  X(K) = Y(IX(K)) - Y(IY(K))
END DO
```

In the I loop, each processor runs over its section of the X array. (That is, the active processor for iteration I is the owner of X(I).) Only the reference X(I) is guaranteed to be resident. (If $M \neq N$, then IX and IY have a different block size than X, and thus a different mapping.) However, if it is *usually* the case that X(I), Y(IX(I)), and Y(IY(I)) are located on the same processor, then this choice of active processors may be the best available. (If X(I) and one of the other references are *always* on the same processor, then the programmer should add the RESIDENT clause as explained in Section 9.3.) In the next loop, iteration J's active processor is the owner of IX(J). Because IY has the same distribution as IX, reference IY(J) is always resident as well as IX(J). This is the most common array reference class in the loop, so it minimizes the number of nonresident data references in the absence of any special properties of IX and IY. It may not evenly balance the load among processors; for example, if /(N=M/2 /) then half the processors will be idle. As before, if the values in IX or IY ensure that one of the Y references is always resident, a RESIDENT assertion should be added. In the K loop, only reference Y(IX(K)) is guaranteed to be resident (because Y and X have the same distribution). However, the values stored in IX and IY may ensure that Y(IY(K)) and X(K) are always resident. Even if the three REAL values are not always, but merely "usually" on the same processor, this may be a good computation partitioning for both locality and parallelism. However, these advantages must be weighed against the cost of computing this partitioning. Since the HOME clause depends on a (presumably large) array of runtime values, substantial time may be required to determine which iterations are assigned to each processor. It should be clear from this discussion that there is no magic solution for handling complex computation partitionings; the best answer is usually a combination of application knowledge, careful data structure design (including ordering of the elements), and efficient compilation methodology and runtime support.

Advice to implementors. The K loop is the situation that the inspector strategy described above was designed for. If there is an outer loop around any of these examples, and that loop does not modify the distribution of X or the values of IX, then a record of each processor's iterations can be saved for reuse. The cost is at worst linear in the sizes of the arrays. (*End of advice to implementors.*)

Advice to users. It is unlikely that any current production compiler will generate low-overhead code for K loop. The difference from previous examples is that the HOME clause is not a function that can be easily inverted by the compiler. Some compilers may choose to execute every iteration on all processors, testing the HOME clause at run-time; others may pre-compute a list of iterations for every processor. Of course, the cost of computing the list will be substantial.

In practice, one would make all the arrays the same size to avoid some of the alignment problems above; the example was written this way for pedagogical reasons, not as an example of good data structure design. (*End of advice to users.*)

9.2.4 ON Directives Applied to Subprogram Invocations

The key rule about ON directives when applied to subprogram invocations is that the invocation does not change the active processor set. In effect, the callee inherits the caller's active processors. Thus,

```
!HPF$ PROCESSORS P(10)
!HPF$ DISTRIBUTE X(BLOCK) ONTO P

!HPF$ ON ( P(1:3) )
      CALL PERSON_TO_PERSON()
!HPF$ ON ( P(4:7) )
      CALL COLLECT( X )
```

calls PERSON_TO_PERSON on three processors, while it calls COLLECT on four. The actual argument to COLLECT does not reside completely on the active set of processors. This is allowed, with appropriate declarations of the corresponding dummy argument as explained below.

The above rule has interesting implications for data distributions within the called routine. In particular, dummy arguments must be declared under the same restrictions as local objects, thus ensuring that the `dummy` is always stored on the active processor set. This does not imply that the corresponding `actual` argument is local, however. Consider the possibilities for how a dummy can be explicitly mapped:

- **Prescriptive mapping:** If the actual is not mapped on the active processor set, it will be remapped. This is exactly analogous to remapping a `BLOCK`-distributed array to `CYCLIC` via a prescriptive mapping.
- **Descriptive mapping:** The user is asserting that the actual is already mapped onto the set of active processors. If the assertion is true, then the dummy is already stored locally; if not, then the compiler inserts a remapping operation (and reports a warning, following the recommendations in Section 4).
- **Transcriptive mapping:** In this case, a new restriction must be made to allow efficient access to the dummy argument. *If a dummy is transcriptively mapped, then the actual argument must be resident on the active processor set at the time of the invocation.* This may be checked at run-time.

In summary, a dummy argument is always mapped to the set of active processors, although the actual argument need not be (except in the case of transcriptive mappings).

Rationale. The treatment of dummy arguments as local objects is consistent with all previous Fortran (and FORTRAN) standards. Moreover, it has the advantage that it reflects the usual expectations and wishes of programmers. Dummy arguments are not expected to create great inefficiencies in Fortran programs; ensuring that they are always stored locally tends to reinforce that expectation. Also, programmers are used to "pass by reference" behavior, in which arguments are not copied; the restrictions on data mapping to active processor sets allow this implementation when the data is not remapped on subprogram call. One case of this deserves special mention—transcriptive mappings. If the programmer wants to keep the data in place (the usual expectation of `INHERIT` and related features) and control which processors execute the computation (the meaning of `ON`), then the basic principles of active sets (set forth in Section 9.1) imply that the data must be resident before the call is made. When remapping occurs due to explicit directives, then surely the user expects a communication cost to accompany the remapping.

It should also be noted that these rules do not invalidate any HPF programs written without using the `ON` directive. In those programs, the active processor set never changes (at least, not from the view of the language). Therefore, subset and universal processors arrangements can be used interchangeably, and the restriction on use of transcriptive mappings is obeyed automatically. (*End of rationale.*)

Advice to implementors. These restrictions imply that accesses to dummy arguments never require one-sided communication if the argument is explicitly mapped and the `ON` clause is used. Of course, accesses to global data may still run into serious complications. If the compiler itself partitions the computation, it is not restricted by the `ON` directive rules. (*End of advice to implementors.*)

Advice to users. The idea to remember in calling subprograms from an ON block is this: make sure that the actual arguments are stored on the active set. If the subroutine interface uses transcriptive ("take anything") mappings, then this is a requirement. If the subroutine uses any other type of mapping, then having resident actual arguments *may* avoid the expense of remapping data. (Of course, it does not by itself guarentee that remapping doesn't occur—a prescriptive interface can force a BLOCK-to-CYCLIC redistribution—but it does ensure that the remapping is between active processors. This allows simpler and more efficient collective communications operations to be generated in the runtime system.) (*End of advice to users.*)

Let us return to the previous example:

```
!HPF$ PROCESSORS P(10)
!HPF$ DISTRIBUTE X(BLOCK) ONTO P

!HPF$ ON ( P(4:7) )
      CALL COLLECT( X )
```

If COLLECT were declared as

```
      SUBROUTINE COLLECT( A )
!HPF$ DISTRIBUTE A(CYCLIC)
```

then the call will be executed as follows:

1. X will be remapped from BLOCK on 10 processors (i.e., all of P) to CYCLIC on 4 processors (i.e., P(4:7)). This will be a many-to-many exchange pattern.
2. COLLECT will be called on processors P(4), P(5), P(6), and P(7). Accesses to A within the subroutine will be satisfied from the redistributed array on those processors.
3. A will be remapped back to the distribution of X. This is the inverse of step 1.

Note that the distribution of A is onto 4 processors (the active processor set inside the call), not onto the universal processor set. If the interface is

```
      SUBROUTINE COLLECT( A )
!HPF$ DISTRIBUTE A(BLOCK)
```

then the process would be the same, except that there would be a remapping from BLOCK on 10 processors to BLOCK on 4 processors. That is, the block size would increase by 2.5 times (with related shuffling of data) and then revert to the original. Again, it is important to note that the distribution of A is onto the active processor set rather than onto all of P.

The similar examples

```
      REAL X(100,100), Y(100,100)
!HPF$ PROCESSORS P(4), Q(2,2)
!HPF$ DISTRIBUTE X(BLOCK,*) ONTO P
!HPF$ DISTRIBUTE Y(BLOCK,BLOCK) ONTO Q

      INTERFACE
        SUBROUTINE A_CAB( B )
```

```
          REAL B(:)
!HPF$     DISTRIBUTE B *(BLOCK)
        END INTERFACE

!HPF$ ON ( P(4:7) )
        CALL A_CAB( X( 1:100, 1 )
!HPF$ ON HOME( X(1:100,1) )
        CALL A_CAB( X(1:100,100) )
!HPF$ ON HOME( Y(1:100,1) )
        CALL A_CAB( Y(1:100,1) )
!HPF$ ON HOME( Y(99,1:100) )
        CALL A_CAB( Y(99,1:100) )
```

can be explained as follows. Calling `A_CAB(1:100,1)` on `P(4:7)` will produce a remapping from 10 processors to 4, as in the example above. (The compiler would be expected to produce a warning in this case, as explained in Section 4.) Calling `A_CAB(X(1:100,100))` on `HOME(X(1:100,1))` produces no such remapping (or warning), because the active processor set does not change; therefore, the descriptive mapping correctly asserts that the data is already on the right processors. The last two examples, calling `A_CAB(Y(1:100,1))` and `A_CAB( Y(99,1:100) )` on the homes of their arguments, are also accomplished without remapping. In both cases, the actual arguments are mapped `BLOCK`-wise onto a subset of the processors (a column of `Q` in the first case, a row of `Q` in the second). Some compilers may not be able to generate code for these more complex examples, however.

Two examples of transcriptive mapping are also useful:

```
! Assume
! PROCESSORS P(4)
! is declared in a module
        REAL X(100)
!HPF$ DISTRIBUTE X(CYCLIC(5)) ONTO P

        INTERFACE
          SUBROUTINE FOR_HELP( C )
          REAL C(:)
!HPF$     INHERIT C
        END INTERFACE

!HPF$ ON HOME( X(11:20) )
        CALL FOR_HELP( X(11:20) )
!HPF$ ON ( P(1) )
        CALL FOR_HELP( X(51:60) )     ! Nonconforming
```

The first example is valid—the actual argument is (trivially) distributed on the active processor set. The second example is invalid—for example, element `X(51)` is stored on `P(3)`, which is not in the active processor set for the call. The second example would be valid if the `ON` directive specified `P(3:4)` or `HOME(X(11:20))`, both of which map to the same processor set.

Calls to `EXTRINSIC` subprograms also deserve mention. The “standard” HPF 2.0 description of calling an `EXTRINSIC` (Section 6) says in part:

⇑

A call to an extrinsic procedure must be semantically equivalent to a call of an ordinary HPF procedure. Thus a call to an extrinsic procedure must behave *as if* the following actions occur...

1. Exactly the same set of processors are visible to the HPF environment before and after the subprogram call.

This constraint is changed to read

- Exactly the same set of *active* processors are available to the HPF environment before and after the subprogram call.
- Exactly the same *universal* processor set is visible to the HPF environment before and after the subprogram call.

The intent is the same as in the original language design. Processors where data is stored can neither appear not disappear; nor may the set of processors executing the program change without notice to the program. Similarly, some extrinsic kinds specify "all processors must be synchronized" or "execution of a local procedure on each processor"; such language is understood to mean "all *active* processors must be synchronized" or "execution of a local procedure on each *active* processor."

Rationale. This gives the combination of `EXTRINSIC` procedures and `ON` directives a fork-join model of parallelism, which seems to be both natural and semantically clean. (*End of rationale.*)

If a procedure uses alternate return, then the target of the return must be have the same active processor set as the `CALL` statement. In effect, this means that labels passed as arguments must refer to statements in the same `ON` block as the `CALL` statement.

Rationale. This constraint is similar to the prohibition against jumping out of an `ON` block, and has the same justification. (*End of rationale.*)

Explicit use of `CALL`s in `ON` directives is often associated with task parallelism. Several examples can be found in Section 9.4. The following example illustrates how processors can be used for a one-dimensional domain decomposition algorithm:

```
!HPF$ PROCESSORS PROCS(NP)
!HPF$ DISTRIBUTE X(BLOCK) ONTO PROCS

! Compute ILO(IP) = lower bound on PROCS(IP)
! Compute IHI(IP) = upper bound on PROCS(IP)
DONE = .FALSE.
DO WHILE (.NOT. DONE)
  !HPF$ INDEPENDENT
  DO IP = 1, NP
    !HPF$ ON (PROCS(IP))
    CALL SOLVE_SUBDOMAIN( IP, X(ILO(IP):IHI(IP)) )
  END DO
  !HPF$ ON HOME(X) BEGIN
```

```
            CALL SOLVE_BOUNDARIES( X, ILO(1:NP), IHI(1:NP) )
            DONE = CONVERGENCE_TEST( X, ILO(1:NP), IHI(1:NP) )
          !HPF$ END ON
        END DO
```

The algorithm divides the entire computational domain (array X) into NP subdomains, one for each processor. The INDEPENDENT IP loop performs a computation on each subdomain's interior. The ON directive tells the compiler which processors to use in executing these (conceptually) parallel operations. This can increase data locality substantially, particularly if the compiler could not otherwise analyze the data access patterns in SOLVE_SUBDOMAIN. The subroutine SOLVE_SUBDOMAIN can use a transcriptive or descriptive mapping for its array argument, placing it on a single processor. In the next phase, the processors collaborate to update the boundaries of the subdomains and test for convergence. Subroutines SOLVE_BOUNDARIES and CONVERGENCE_TEST may well have their own loops similar to the IP loop, with similar RESIDENT clauses. Note that only the lower and upper bounds of each subdomain is recorded; this allows different processors to process different-sized subdomains. However, each subdomain must "fit" into one processor's section of the X array.

Advice to implementors. The IP loop above is likely to be a common idiom among programmers doing block-structured codes. In general, it can be implemented by inverting the HOME clause as was done above. In the one-to-one case shown here (probably very popular with programmers), it can be implemented by assigning the processor id to the loop index variable and testing the range of the loop (once). (*End of advice to implementors.*)

Rationale. Some compilers will propagate the ON information from the caller to the callee at compile time, and some at run time. Repeating the ON clause in the caller and callee will tend to give the compiler better information, resulting in better generated code. (*End of rationale.*)

9.3 The RESIDENT Clause, Directive, and Construct

The purpose of the RESIDENT clause is to promise that data accessed by a computation are mapped to active processors. That is, RESIDENT asserts that certain references (or all references) in its scope are stored on the active processor set. The compiler can use this information to avoid generating communication or to simplify array address calculations. Note that whether a given data element is resident depends on two facts:

- Where the data is stored (i.e., DISTRIBUTE and ALIGN attributes for the object)
- Where the computation is executed (i.e., its active set, as specified by an ON directive)

For these reasons, the RESIDENT clause is added to the ON directive, which is usually the earliest point in the program where the needed facts might be available. The RESIDENT clause can also appear as a stand-alone directive; this is useful when the locality information is not true for an entire ON region. Note that changing the ON directive may invalidate some RESIDENT clauses, or may make more RESIDENT clauses true.

H910	*resident-clause*	**is**	RESIDENT *resident-stuff*
H911	*resident-stuff*	**is**	[(*res-object-list*)]

H912	*resident-directive*	**is**	`RESIDENT` *resident-stuff*
H913	*resident-construct*	**is**	*directive-origin block-resident-directive* *block* *directive-origin end-resident-directive*
H914	*block-resident-directive*	**is**	`RESIDENT` *resident-stuff* `BEGIN`
H915	*end-resident-directive*	**is**	`END RESIDENT`
H916	*res-object*	**is**	*object*

A *resident-directive* is a kind of *executable-directive*. Similarly, a *resident-construct* is a kind of *executable-construct*.

Any top-level objects in the `RESIDENT` clause must be explicitly mapped. Similarly, the `RESIDENT` clause must appear at a point in the program with a declared active processor set (i.e., inside an `ON` block). Otherwise, the assertion (see below) is a statement about how the compiler works, not about the program.

> *Advice to implementors.* `RESIDENT` removes the need for inactive processors to participate in communication into/out of an `ON` clause. (*End of advice to implementors.*)

The `RESIDENT` directive is an assertion to the compiler that certain object references made within the `ON` are stored on the active processors if the computation is performed by the specified active processor set. The scope of the assertion is the next Fortran statement if the *resident-directive* form is used and the enclosed *block* of code if the *resident-construct* form is used. If `RESIDENT` appears as a clause in an `ON` directive, then the `ON` and `RESIDENT` apply to the same statements.

`RESIDENT(`*var*`)` means the *lexical expression var*, when encountered in the execution of statements in the scope of the `RESIDENT` directive, accesses only data resident on the set of active processors. (That is, the set of processors named by the innermost available `ON` directive.) If *var* is accessed by the statement (e.g., it appears on the right-hand side of an assignment statement, or in the evaluation of a conditional expression), then at least one copy of the object and any subobject of the object must be mapped to the active processor set. If *var* is assigned to by the statement (e.g., it appears on the left hand side of an assignment statement, or in the variable list of a `READ` statement), or in any other context that may cause its value to change) then all copies of the variable and all subobjects of the variable must reside in the active processor set. The application of `RESIDENT` to `CALL` statements and function invocations introduces some complexity into this interpretation; these issues will be dealt with in Section 9.3.2.

Note that `RESIDENT` is always an assertion relative to the surrounding `ON` directive. Therefore, if the compiler does not implement the `ON` directive then it must be careful in interpreting `RESIDENT`. Similarly, if the compiler overrules the programmer-specified `ALIGN` and `DISTRIBUTE` directives, then it may not rely on the `RESIDENT` clause in general.

Finally, `NEW` variables are not considered by any nested `RESIDENT` directives, as detailed below.

> *Rationale.* The different treatment of variable reads and writes is due to the implementation requirements. If a variable's value is read (but not written), then it can be taken from any consistent copy. Therefore, `RESIDENT` only asserts that one of those

copies is available. Conversely, all copies of a replicated variable must be consistent, so RESIDENT asserts that all copies are available if it is updated.

The RESIDENT assertion is always relative to the declared data mappings and ON clauses because both pieces of information are necessary to determine the locality of data references. Data mapping determines where the data is stored, while ON clauses determine where they are used; in essence they determine the endpoints of a data path. RESIDENT itself says that the path length is very short; obviously, one cannot measure a path without knowing both endpoints. (*End of rationale.*)

Consider the following:

```
!HPF$ ON HOME(Z(I)), RESIDENT(X,Y,RECORD(I))
      X(I) = Y(I+1) + RECORD(I)%FIELD1 + RECORD(I+1)%FIELD2
```

The following facts are asserted by the directive:

- Z(I) would be local if it appeared, due to its use in the HOME directive.
- All copies of X(I) are stored on processors that also store a copy of Z(I), due to the RESIDENT clause. This may be true because X and Z have the same mapping, or because Z is replicated over a set of processors that contains the set of processors that store X(I).
- At least one copy of Y(I+1) is on the same processor as Z(I), due to the RESIDENT clause. This may be true because Y is replicated on all processors, because Z(I) and Y(I+1) are the only elements of their arrays that are mapped to the same processor, or because the directive

  ```
  !HPF$ ALIGN Y(J) WITH Z(J-1)
  ```

  appears elsewhere in the program. (Other situations also make the RESIDENT assertion true.)
- At least one copy of all subobjects of RECORD(I) is mapped on the same processor as Z(I). In particular, the reference RECORD(I)%FIELD1 (i.e., a subobject consisting of one component) can be accessed locally. The situations in which this is true are similar to those for X(I). No information is available in this example regarding RECORD(I+1)%FIELD2.

If there is no *res-object-list*, then *all* references to *all* variables referenced during execution of the RESIDENT directive's body except those declared NEW in a surrounding ON directive are local in the sense described above. That is, for every usage of any variable's value, at least one copy of the variable will be mapped to the ON processor set. Likewise, for every operation that assigns to a variable, all copies of that variable are mapped to the ON processor set. References and assignments to NEW variables are always considered resident. If there are no function or subroutine invocations, this is syntactic sugar for listing all variable references within the directive's scope. (See Section 9.3.2 for a discussion of RESIDENT clauses applied to subprogram calls.) It might well have been named the ALL_RESIDENT clause; the present form, however, does not add yet another keyword to the directive sublanguage.

Note that if the active set includes more than one processor, then RESIDENT only asserts that the variables are stored on one of the processors. For example, if a statement is executed on a section of the processors arrangement, then communication within that section may be needed for some variables in the RESIDENT clause. Communication with processors outside of the section will not be needed for those variables, however.

Rationale. The alternative to this interpretation would be that any variable named in the RESIDENT clause would be local to all processors, i.e., replicated. While that certainly allows more extensive optimizations, it is a less common case. In addition, it does not seem to capture the intent of ON directives applied to CALL statements or compound statements. For example,

```
!HPF$ PROCESSORS PROCS(MP,MP)
!HPF$ DISTRIBUTE X(BLOCK,BLOCK) ONTO PROCS
!HPF$ ON HOME(PROCS(1,1:MP)), RESIDENT(X(K,1:N) )
      CALL FOO( X(K,1:N) )
```

would presumably call FOO on a row of the processors arrangement, passing elements of X in place. This is what the current definition does; if RESIDENT meant "resident on every processor", the call would force X to be replicated. (*End of rationale.*)

It is not correct to assert that an unmapped object, including of necessity any sequential object, is mapped exclusively to the active processors, unless the programmer knows that all of the processors are active. Thus, when a proper subset of processors is active, no such object can occur in a *res-object-list* or in the scope of a RESIDENT directive with no *res-object-list*.

The RESIDENT directive is similar to the INDEPENDENT directive, in that if it is correct it does not change the meaning of the program. If the RESIDENT clause is incorrect, the program is not standard-conforming (and is thus undefined). Like the INDEPENDENT directive, the compiler may use the information in the RESIDENT clause, or ignore it if it is insufficient for the compiler's purposes. If the compiler can detect that the RESIDENT clause is incorrect (i.e., that a RESIDENT variable is definitely nonlocal), it is justified in producing a warning. Unlike the INDEPENDENT directive, however, the truth of the RESIDENT clause depends on the mapping of computations (specified with the ON clause) and the mapping of data (specified with DISTRIBUTE and ALIGN clauses); if the compiler overrides either of these, then it may not be able to use information in the RESIDENT directive.

Rationale. Knowing that a reference is local is valuable information for the optimizer. It is in keeping with the spirit of HPF to phrase this as an assertion of fact, which the compiler can use as it pleases. Expressing it as advice to the compiler seems to have disadvantages. Some possible ways this advice could be phrased, and the counter-arguments, are

- "Don't generate communication for this reference" has great potential for changing the meaning of the program. Some programmers want this capability, but it violates the "correct directives should not change the meaning of a program" principle of HPF. Also, once communication is "turned off" for a reference, it's not clear how to turn it back on.

- "Generate communication for this reference" is not a useful directive, since the compiler has to do this anyway.
- "Generate communication for this reference, and place it here" is useful, since it can override the default placement by the compiler. It still has potential for changing program meaning. It also has the potential to create programs as complex as message-passing, as programmers try to move communication out of loops.

(*End of rationale.*)

9.3.1 Examples of RESIDENT Clauses

As in Section 9.2.3, our aim here is to suggest idioms that may be generally useful to programmers. We begin by expanding on two earlier examples.

RESIDENT is most useful in cases where the compiler cannot detect access patterns. Often this arises due to the use of indirection, as in the following examples:

```
REAL X(N), Y(N)
INTEGER IX1(M), IX2(M)
!HPF$ PROCESSORS P(NP)
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: X, Y
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: IX, IY

!HPF$ INDEPENDENT
DO I = 1, N
  !HPF$ ON HOME( X(I) ), RESIDENT( Y(IX(I)) )
  X(I) = Y(IX(I)) - Y(IY(I))
END DO

!HPF$ INDEPENDENT
DO J = 1, N
  !HPF$ ON HOME( IX(J) ), RESIDENT( Y )
  X(J) = Y(IX(J)) - Y(IY(J))
END DO

!HPF$ INDEPENDENT
DO K = 1, N
  !HPF$ ON HOME( X(IX(K)) ), RESIDENT( X(K) )
  X(K) = Y(IX(K)) - Y(IY(K))
END DO
```

As we saw in Section 9.2.3, X(I) is always local in the I loop and IX(I) and IY(I) rarely are. The RESIDENT directive above ensures that Y(IX(I)) is local as well. This would most likely be due to some property of the algorithm that generated IX (for example, if IX(I)=I for all I). Note that it is possible for an expression (e.g., Y(IX(I))) to be local even though one of its subexpressions (IX(I)) is not.

The directive gives no information about Y(IY(I)); it might have only one nonlocal value, or all its values might be nonlocal. (We assume that if there were no nonlocal values, then the RESIDENT clause would include Y(IY(I)) as well.) If there are many local elements referenced by this expression, and they can easily be separated from the local elements, then it may be worthwhile to restructure the loop to make this clear to the compiler. For example, suppose that we knew that only the "first" and "last" X elements on each processor were nonlocal. The loop could then be split thus:

```
!HPF$ INDEPENDENT, NEW(LOCALI)
DO I = 1, N
  !HPF$ ON HOME( X(I) ), RESIDENT( Y(IX(I)), Y(IY(I)) ) BEGIN
    LOCALI = MOD(I,N/NP)
    IF (LOCALI\=1 .AND. LOCALI\=0) THEN
      X(I) = Y(IX(I)) - Y(IY(I))
    END IF
  !HPF$ END ON
END DO
!HPF$ INDEPENDENT, NEW(LOCALI)
DO I = 1, NP
  !HPF$ ON (P(I)), RESIDENT( X(LOCALI), Y(IX(LOCALI)) ) BEGIN
    LOCALI = (I-1)*N/NP
    X(LOCALI) = Y(IX(LOCALI)) - Y(IY(LOCALI))
    LOCALI = I*N/NP
    X(LOCALI) = Y(IX(LOCALI)) - Y(IY(LOCALI))
  !HPF$ END ON
END DO
```

The first loop (inefficiently) processes the local elements of `Y(IY(I))`, while the second (more efficiently) handles the rest. On most machines, it would pay to rewrite both loops to avoid the division operations, for example by creating a logical mask *a priori*.

In the `J` loop, the `RESIDENT` clause asserts that all accessed elements of `Y` are local. In this case, that is equivalent to the assertion

```
!HPF$ RESIDENT( Y(IX(J)), Y(IY(J)) )
```

Although the original `RESIDENT` clause only referred to the lexical expression `Y`, the compiler can infer that the subexpressions are also local. This is because it is impossible for a subobject to be on a different processor than the "parent" object is. This observation can often shorten `RESIDENT` clauses substantially.

In the `K` loop, the following references are local:

- `Y(IX(K))`, because `Y` has the same distribution as `X` and `X(IX(K))` is local (due to the `ON` clause).
- `X(K)`, because of the `RESIDENT` clause.

Note that a reference may be local even if it does not appear explicitly in a `RESIDENT` clause. One mark of a good compiler will be that it aggressively identifies these elements.

Because it is an assertion of act, the compiler can draw many inferences from a single `RESIDENT` clause. For example, consider the following case:

```
!HPF$ ALIGN Y(I) WITH X(I)
!HPF$ ALIGN Z(J) WITH X(J+1)
```

```
!HPF$ ON HOME( X(K) ), RESIDENT( X(INDX(K)) )
        X(K) = X(INDX(K)) + Y(INDX(K)) + Z(INDX(K))
```

The compiler is justified in making the following assumptions in compiling the assignment statement (assuming it honors both the ALIGN directives and the `ON` directive):

- `X(K)` requires no communication (because of the `HOME` clause)
- `X(INDX(K))` requires no communication (because of the `RESIDENT` clause)
- `Y(INDX(K))` requires no communication (because `Y` has the same mapping as `X`, and `INDX(K)` clearly cannot change values between its use in the two references `X(INDX(K))` and `Y(INDX(K))`)

The compiler cannot make any assumption about `INDX(K)` or `Z(INDX(K))` from the above code. There is no indication how `INDX` is mapped relative to `X`, so the `ON` directive gives no guidance. Note that the fact that an expression (here, `X(INDX(K))`) is local does not imply that its subexpressions (here, `INDX(K)`) are also local. Similarly, `Z`'s mapping does not determine if `Z(INDX(K))` would be local; it indicates that `Z(INDX(K)-1)` is local, but that isn't a great help. If the compiler has additional information (for example, `X` is distributed by `BLOCK` and `INDX(K)` is not near a block boundary), it might be able to make additional deductions.

Advice to implementors. One mark of a good compiler will be that it aggressively propagates `RESIDENT` assertions. This is likely to significantly reduce communication costs. Note the cases under "Advice to users" below. (*End of advice to implementors.*)

Advice to users. One can expect compilers to differ in how aggressive they are in drawing these deductions. Higher-quality compilers will be able to identify more references as local, and use this information to eliminate data movement. All compilers should recognize that if an element of one array is local, then the same element of any other arrays with the same static mapping (i.e., arrays aligned together, or with the same `DISTRIBUTE` pattern and array size) will also be local. That is, any compiler should recognize `Y(INDX(K))` in the above example as local. Dynamically changing array mappings (i.e., `REALIGN` and `REDISTRIBUTE`) will tend to limit such information and information propagation. Also, assignments that might change subexpressions (for example, an assignment to `K` or any element of `INDX` in the above example) will force the compiler to be conservative in its deductions. (*End of advice to users.*)

9.3.2 RESIDENT Directives Applied to Procedure Reference

If a `RESIDENT` directive applies to procedure reference, then the assertion is more subtle.

- If a *res-object-list* appears in the `RESIDENT` directive, then no assertion is made about behavior within the called procedure. For example, consider the statements:

```
    !HPF$ RESIDENT( A(I), B )
            A(I) = F( A(I), B(LO:HI) )
```

The directive declares all variable refences in the statement (including the actual arguments) to be local to the current ON processor set. However, the execution of F itself could access elements of arrays named A and B stored on arbitrary processors.

Rationale. Propagating assertions about the behavior of lexical entities is difficult to define consistently and usefully. For example, consider the following function called from the code fragment above:

```
REAL FUNCTION F( X, Y )
USE MODULE_DEFINING_A
REAL X, Y(:), B(I)
!HPF$ INHERIT Y
!HPF$ ALIGN B(:) WITH Y(:)
INTEGER I

Z = 0.0
DO I = 1, SIZE(Y)
    Z = Z + A(I)*X + B(I)*Y(I)
END DO
F = Z
END FUNCTION
```

Assume A is defined as a distributed, global array in module MODULE_DEFINING_A. What should the RESIDENT clause mean regarding operations in F? The expression A(I) in the RESIDENT directive might reasonably mean references only to the array A that is visible in the caller, or it might mean references to any array named A. Note that the A in the caller may be local, the same global array as the A in F (if the caller used MODULE_DEFINING_A), or a different global array (if the caller used a different module). Perhaps a limiting case is array B. The array B in function F is local, and thus different from the caller; however, because of the restrictions on ON clauses it is certain that the local B will be mapped to the ON processor set. Thus, the RESIDENT assertion is trivially true. To further confuse matters, RESIDENT variables might seem to apply to dummy arguments that become associated with those variables. Unfortunately, this implies that the lexical expression B in the caller refers to the lexical expression Y in F, which stretches the definition of "lexical" beyond the breaking point. For all these reasons, it was decided to limit the meaning of named variables in RESIDENT clauses to the lexical scope of the directive. (*End of rationale.*)

- If the RESIDENT directive does not contain a *res-object-list*, then the directive asserts that all references in the caller *and the called procedures* are local as defined above. For example, consider the statements:

```
!HPF$ RESIDENT
A(I) = F( A(I), B(LO:HI) )
```

The directive declares all variable refences in the statement (including the actual arguments) to be local to the current ON processor set, and that F itself does not reference or update any nonlocal variables.

Rationale. The `RESIDENT` assertion is always true for data local to the called procedure. This is true because the called procedure must use a declarative `ON` clause, which in turn limits the set of processors that can store any local explicitly mapped variables. The above definition extends this assertion to all global explicitly mapped data, producing a very powerful directive. This is similar to the meaning of `INDEPENDENT`, in that it also makes an assertion about variable accesses in any called procedure in the loop. An alternative semantics for `RESIDENT` would have been to avoid propagating the assertion interprocedurally (i.e., treat both the variable-list version and the no-list version the same). However, this would not provide enough information to optimize code on certain machines. In particular, it would have made task parallelism quite difficult on message-passing machines. (*End of rationale.*)

Advice to implementors. `RESIDENT` without a variable list guarantees that no one-sided communication outside of the `ON` processor set will be generated by the callee. Such a procedure can be called only on the "active" processors, unless the runtime system has additional constraints (for example, if the runtime system requires all processors to participate in collective communications).

The other forms of `RESIDENT` provide information that could be propagated interprocedurally. For example, if the actual argument to a subprogram is asserted to be `RESIDENT` and is passed transcriptively, then anything that is aligned to it in the callee will also be `RESIDENT`. If the information is not propagated, the only result will be less optimization. (*End of advice to implementors.*)

Advice to users. Although the `RESIDENT` assertion applies interprocedurally, it is by no means certain that all compilers will make use of this information. In particular, separate compilation limits the propagation that can take place. It is therefore good practice to include a `RESIDENT` clause both in the caller's `ON` directive and in the callee. (This assumes that the assertion is true, of course!) This ensures that the compiler has the `RESIDENT` information available when it is compiling both ends of the procedure call. This is especially useful for `RESIDENT` clauses without a variable list; knowing that all data accessed is local allows many optimizations that are not otherwise possible. (*End of advice to users.*)

Locality information is particularly critical interprocedurally. Here, the `RESIDENT` directive without a *res-object-list* can be used to good advantage. Consider the following extension of the block-structured example from Section 9.2.4:

```
!HPF$ PROCESSORS PROCS(NP)
!HPF$ DISTRIBUTE X(BLOCK) ONTO PROCS

! Compute ILO(IP) = lower bound on PROCS(IP)
! Compute IHI(IP) = upper bound on PROCS(IP)
DONE = .FALSE.
DO WHILE (.NOT. DONE)
  !HPF$ INDEPENDENT
  DO IP = 1, NP
    !HPF$ ON (PROCS(IP)), RESIDENT
```

```
          CALL SOLVE_SUBDOMAIN( IP, X(ILO(IP):IHI(IP)) )
        END DO
        !HPF$ ON HOME(X) BEGIN
          CALL SOLVE_BOUNDARIES( X, ILO(1:NP), IHI(1:NP) )
          !HPF$ RESIDENT
          DONE = CONVERGENCE_TEST( X, ILO(1:NP), IHI(1:NP) )
        !HPF$ END ON
      END DO
```

Recall that the INDEPENDENT IP loop performs a computation on each subdomain's interior, where a subdomain is mapped to a particular processor. The first RESIDENT clause additionally informs the compiler that no subdomain uses data from another processor. Without this information, the compiler would have to assume a worst-case scenario in which each subdomain performed its updates based on non-local read-only data. Any nonlocal data could not be written by another processor without violating the tt INDEPENDENT directive; however, if the data were not updated (for example, a large lookup table) it could be stored remotely. Particularly on nonshared-memory machines, access to this remote data would be difficult. The RESIDENT clause ensures that this possibility need not be considered. All data required by SOLVE_SUBDOMAIN is stored locally. The second RESIDENT clause asserts that all data for CONVERGENCE_TEST is stored on the same processors that store X. The same cannot be said for SOLVE_BOUNDARIES, which does not fall in the scope of the RESIDENT directive. For example, there might be a processors arrangement other than PROCS with necessary data. Accessing this data might well cause a bottleneck in the computation as described above.

Again, note the usefulness of RESIDENT clauses in giving the compiler information. Few compilers would be able to unravel nontrivial assignments to ILO and IHI, and no current compiler would even attempt to understand the comments in the above code fragment. End of advice to programmers.

9.4 The TASK_REGION Construct

Task parallelism is expressed in HPF by mapping data objects onto subsets of processors and adding assertions that allow concurrent execution of different code blocks on different processor subsets. A data object is mapped to a processor subset by distribution onto a subsection of a processors arrangement. Execution on a subset of processors is specified by using an ON directive. This section introduces a TASK_REGION directive that allows the user to specify that disjoint processor subsets can execute blocks of code concurrently.

A TASK_REGION directive is used to assert that a block of code satisfies the following set of constraints. All lexically outermost ON blocks inside a task region must have a RESIDENT attribute implying that all data accessed inside them is mapped to the corresponding active processor subset. Further, the code inside two such ON blocks must not have interfering I/O. Under these constraints, two such ON blocks can safely execute concurrently if they execute on disjoint processor subsets.

9.4.1 Syntax of the TASK_REGION Construct

A task region is a single entry region delimited by two structured comments:

H917	*task-region-construct*	**is**	*directive-origin block-task-region-directive*
			block
			directive-origin end-task-region-directive
H918	*block-task-region-directive*	**is**	`TASK_REGION`
H919	*end-task-region-directive*	**is**	`END TASK_REGION`

A *task-region-construct* is a kind of *executable-construct.*

There must not be a transfer of control from outside the *task-region-construct* to inside the *task-region-construct.* Transfer of control out of the *task-region-construct* is allowed provided that the transfer does not originate inside an `ON` block. (The reason for this will be apparent later.)

9.4.2 Semantics of the TASK_REGION Construct

We will refer to a block of code enclosed by a `TASK_REGION ... END TASK_REGION` pair as a *task region.* The `TASK_REGION` directives are a way for the programmer to assert that a section of code satisfies a set of conditions. The compiler is expected to use these assertions to generate task-parallel code.

A task region can contain blocks of code that are directed to execute `ON` processor subsets. All other code executes on a subset that contains all active processors. Every `ON` block at the outermost nesting level (i.e., not inside another `ON` block or another task region) inside a task region is defined as a *lexical task.* Every execution instance of a lexical task is defined as an *execution task* and will also be referred to as just *task* when the distinction is clear from the context. An execution task is associated with a set of *active processors* discussed earlier in this chapter.

The following restrictions must hold inside a task region:

- Every `ON` block corresponding to a lexical task must have the `RESIDENT` attribute. This means that, for reading a variable inside an execution task, the corresponding active processors must own at least one copy of the variable, and for writing, they must own all copies of that variable.

- An I/O operation inside an execution task may interfere with an I/O operation inside another execution task if and only if the two tasks execute on identical subsets of processors. Note that two execution tasks can be instances of the same or different lexical tasks. In general, two I/O operations interfere if they access the same file or unit. The conditions for interference of I/O operations are detailed in Section 5.1 in the context of the `INDEPENDENT` directive.

9.4.3 Execution Model and Usage

A task region does not introduce a fundamentally new execution model. However, the assertions implicit in a task region imply that only the specified active processors of an execution task need to participate in its execution, and that other processors can skip its execution. A processor executing a task region participates in the execution of all tasks executing on a processor subset that it belongs to, and does not participate in the execution of tasks executing on processor subsets that it does not belong to. Code outside lexical tasks is executed as normal data parallel code by all active processors of the task region. The access restrictions for a task region guarantee that the results obtained by this execution paradigm will be consistent with pure data parallel execution of a task region.

A task region presents a simple yet powerful model to write integrated task and data parallel programs. We illustrate three basic computation structures in which task parallelism can be effectively exploited with this model.

1. *Parallel sections:* A task region can be used to divide the available processors into disjoint sets for performing independent computations, simulating what is often referred to as *parallel sections*. This form of task parallelism is relatively straightforward and useful in many application scenarios, an example being multiblock applications. The task region simply contains a sequence of `RESIDENT ON` blocks on disjoint processor subsets. Note that the division of processors among subsets can be dynamic, that is, it can be in terms of other variables computed during execution.

2. *Nested parallelism:* Task regions can be nested, and in particular, a subroutine call made from an execution task can further subdivide the active processors using another task region directive. This allows the exploitation of nested parallelism. An example is the implementation of dynamic tree structured divide and conquer computations. As a specific example, *quicksort* can be implemented by recursively partitioning the input array of keys around a pivot, and assigning proportionate number of processors to the two new arrays obtained as a result of partitioning.

3. *Data parallel pipelines:* Task regions can be used to implement pipelined data parallel computations. We will illustrate this with a 2 dimensional fast Fourier transform (2D FFT) computation. The first stage of a 2D FFT reads a two dimensional matrix and performs a 1 dimensional FFT on each row of the matrix. The second stage performs a 1 dimensional FFT on each column of the matrix and generates the final output. In a pipelined data parallel implementation of this form of 2D FFT, the two stages are mapped on to disjoint subsets of processors. Task and data parallel code for a 2D FFT, along with a brief description, is included in Section 9.4.5.

9.4.4 Implementation

A task region is simply an assertion about a code block and the exploitation of task parallelism is, at least partially, dependent on the compilation scheme. While the specifics of how task parallelism is exploited will be strongly dependent on the parallel system architecture, the compiler, and the underlying communication model, we will point out some important considerations and illustrate task parallel code generation with an example. We primarily address distributed memory machines using a message passing communication and synchronization model, but will point out some of the important issues relating to shared memory implementations.

9.4.4.1 Localized computation and communication

It is of central importance that computation and communication inside an executing task should not involve any processors other than those directed to execute the task in the relevant `ON` clause.

On entry to a lexical task, the compiler has to insert checks so that the inactive processors jump to the code following the task. Since an execution task cannot access data outside of the active processor set, no communication needs to be generated between the relevant active processors and other processors. In a message passing model, a communication generation algorithm that only generates necessary messages will naturally achieve the desired results. However, some communication schemes can involve generation of empty messages between processors that do not communicate and it is important to ensure that empty messages are not generated between active processors of an executing task and other processors.

A communication model that uses barriers for synchronization (in shared or distributed memory machines) must ensure that all barriers inside an executing task are subset barriers that only span the active processors. An implementation may also need to include a subset barrier, on entry to and on exit from, an executing task for consistency of data accesses inside and outside an executing task. In general, the compilation framework has to ensure the consistency of data accesses inside and and outside an executing task and this can be done in the context of virtually any synchronization scheme in a shared or distributed memory environment.

9.4.4.2 Replicated computations

All computations exclusively involving replicated variables should be replicated on all executing processors. A simple alternative is that one processor performs the computation and broadcasts the results to all processors. While such replication is generally profitable in HPF anyway, it has additional importance in a task region since the communication generated by a broadcast can cause additional synchronization that may interfere with task parallelism.

9.4.4.3 Implications for I/O

In some parallel system implementations, I/O is performed through a single processor of the system. Task parallelism in the presence of I/O assumes that all processors can perform I/O independently and this paradigm has to be supported, although it is not necessary that each processor be able to physically perform all I/O operations independently. One simple solution is to have a single designated I/O processor that performs all I/O but is not considered an executing processor and hence does not have any execution related dependences.

9.4.4.4 SPMD or MIMD code generation

Another issue for the compiler is whether or not the same code image should execute on all processors. Since different processor groups may need different variables, a naive SPMD implementation is likely to be wasteful of memory since it must allocate all variables on all processors. This can be addressed by dynamic memory allocation, but at the cost of added complexity. Using different code images for different processor subsets is another solution that also leads to significant added complexity.

9.4.5 Example: 2-D FFT

This section shows the use of task parallelism to build a pipelined data-parallel 2-dimensional FFT and illustrates the compilation of task parallelism by showing SPMD code generated from the HPF program.

The basic sequential 2DFFT code is as follows:

```
      REAL, DIMENSION(n,n) :: a1, a2

      DO WHILE(.true.)
          READ (unit = 1, end = 100) a1
          CALL rowffts(a1)
          a2 = a1
          CALL colffts(a2)
          WRITE (unit = 2) a2
          CYCLE
100       CONTINUE
          EXIT
      END DO
```

To write a pipelined task and data parallel 2D FFT in HPF, the code is slightly modified and several HPF directives are added. First, variables `a1` and `a2` are distributed onto disjoint subsets of processors, and then a task region is used to create two lexical tasks to perform `rowffts` and `colffts` on different subsets of processors. The assignment `a2 = a1` in the task region specifies the transfer of data between the tasks. A new variable `done1` is introduced to store the termination condition. The modified code is as follows:

```
        REAL, DIMENSION(n,n) :: a1,a2
        LOGICAL done1
!HPF$   PROCESSORS procs(8)

!HPF$   DISTRIBUTE a1(block,*) ONTO procs(1:4)
!HPF$   DISTRIBUTE a2(*,block) ONTO procs(5:8)

!HPF$   TEMPLATE, DIMENSION(4), DISTRIBUTE(BLOCK) ONTO procs(1:4) :: td1
!HPF$   ALIGN WITH td1(*) :: done1

!HPF$   TASK_REGION
        done1 = .false.
        DO WHILE (.true.)
!HPF$       ON (procs(1:4)) BEGIN, RESIDENT
              READ (unit = iu,end=100) a1
              CALL rowffts(a1)
              GOTO 101
    100       done1 = .true.
    101       CONTINUE
!HPF$       END ON

            IF (done1) EXIT
            a2 = a1

!HPF$       ON (procs(5:8)) BEGIN, RESIDENT
               CALL colffts(a2)
               WRITE(unit = ou) a2
```

```
!HPF$       END ON
        END DO
!HPF$   END TASK_REGION
```

Finally, we show simplified SPMD code generated for each processor. We assume a message passing model where sends are asynchronous and nonblocking and receives block until the data is available. We use a simple memory model where variable declarations are identical across all processors even though some variables will be referenced only on subsets of the processors. A shadow variable `done1_copy` is created by the compiler to transfer information from processor subset 1 to processor subset 2 about termination of processing. The code is as follows:

```
        REAL DIMENSION(n/4,n) :: a1
        REAL DIMENSION(n,n/4) :: a2
        LOGICAL done1

C       Following are compiler generated variables
        LOGICAL done1_copy
        LOGICAL inset1, inset2
C
C       Following magic compiler function call is to set the variables
C       inset1 and  inset2 to .true. for subset 1 and subset 2 processors
C       respectively, and .false. otherwise.
C
        CALL initialize_tasksets(inset1,inset2)

C       Code for processor subset 1
        IF (inset1)
           done1 = .false.
           DO WHILE (.true.)

C       Read is left unchanged as the code depends on the I/O model
              READ (unit = 1,end=100) a1

              CALL rowffts(a1)
              GOTO 101
    100       done1 = .true.
    101       CONTINUE
              _send(done1,procs(5:8))
              IF (done1) EXIT
              _send(a1,proces(5:8))
           END DO
         END IF

C       Code for processor subset 2
        IF (inset2)
           DO WHILE(.true.)
              _receive(done1_copy,procs(1:4))
```

```
            IF (local_done1) EXIT
            _receive(a2,procs(1:4))
            CALL colffts(a2)

C       Write is left unchanged as the code depends on the I/O model.
            WRITE (unit = 2) a2
         END DO
      END IF
```

_send and _receive are communication calls to transfer variables between subsets of processors. Program execution until the end of input is as follows. Subset 1 processors repeatedly read input, compute rowffts, and send the computed output as well as done1 flag, which normally has the value .false., to subset 2 processors. The subset 2 processors receive the flag and the data set, compute colffts and write the results to the output. When the end of input is reached, subset 1 processor set the value flag done1 to .true., send it and terminate execution. Subset 2 processors receive the flag, recognize that the end of input has been reached, and terminate execution.

Section 10

Approved Extension for Asynchronous I/O

This section defines a mechanism for performing Asynchronous I/O from an HPF or Fortran program. These are presented as changes to the Fortran 95 proposed draft standard, X3J3/96-007r1. This extension is a subset of the proposed X3J3 Asynchronous I/O extension, paper X3J3/96-158r2. Briefly, this extension allows direct unformatted data transfers to be performed asynchronously with program execution. The `WAIT` statement can be used to wait for the data transfers to complete. The `INQUIRE` statement can be used to determine if the data transfers are complete.

To section 9.3.4, rule R905 *connect-spec*, add

or `ASYNCHRONOUS`

Add a new section after 9.3.4.10, entitled "**ASYNCHRONOUS specifier in the OPEN statement**", containing the following paragraphs:

> 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` statement is used to wait for or inquire as to the status of asynchronous input/output operations.

To section 9.4.1, rule R912 *io-control-spec*, add

or `ID =` *scalar-default-int-variable*
or `ASYNCHRONOUS`

and also add the constraints

Constraint: If either an `ASYNCHRONOUS` or an `ID=` specifier is present, then both shall be present.

Constraint: If an `ASYNCHRONOUS` specifier is present, the `REC=` specifier shall appear, a *format* shall not appear, and a *namelist-group-name* shall not appear.

Constraint: If an `ASYNCHRONOUS` specifier is present, then no function reference may appear in an expression anywhere in the data transfer statement.

At the end of section 9.4.1, add the following paragraphs:

The addition of the `ID=` specifier results in the initiation of an asynchronous data transfer. Execution of the data transfer statement shall be eventually followed by execution of a `WAIT` statement specifying the same `ID` value that was returned to the `ID` variable in a data transfer statement. This `WAIT` statement is called the *matching* `WAIT` statement. Note that asynchronous data transfer shall be direct and unformatted.

The matching `WAIT` statement shall be executed in the same instance of the same subprogram in which the asynchronous data transfer statement was executed.

Advice to implementors. The above restriction is to prevent the compiler from performing code motion optimizations across `WAIT` statements. Any operations involving variables listed in asynchronous input/output lists must be performed after the matching `WAIT` statement is executed. (*End of advice to implementors.*)

No `ASYNCHRONOUS` specifier nor any `ID=` specifier shall be specified if the `io-unit` was not opened with the `ASYNCHRONOUS` specifier.

In section 9.4.1, in the fourth and fifth paragraphs after the constraints, change both instances of "`IOSTAT=` or a `SIZE=`" to "`IOSTAT=`, `SIZE=`, or an `ID=`".

Insert the following text at the end of section 9.4.3 before the final paragraph:

For an asynchronous data transfer, errors may occur either during execution of the data transfer statement or during subsequent data transfer. If these errors occur during the data transfer statement and do not result in termination of the program, then they will be detectable using `ERR=` and `IOSTAT=` specifiers in the data transfer statement. If these error conditions occur during subsequent data transfer and do not result in termination of the program, then they will be detectable using `ERR=` and `IOSTAT=` specifiers in the matching `WAIT` statement.

In the paragraph at the end of section 9.4.3, change the first occurrence of "execution" to read "execution or subsequent data transfer."

To section 9.4.4, add the following paragraphs:

For asynchronous data transfers steps 1–8 correspond to both the asynchronous data transfer statement and the matching `WAIT` statement. Steps 4–7 may occur asynchronously with program execution. If an implementation does not support asynchronous data transfers then steps 1–8 may be performed by the asynchronous data transfer statement. The matching `WAIT` statement shall still be executed, the only effect being to return status information.

Any variable that appears as an *input-item* or *output-item* in an asynchronous data transfer statement, or that is associated with such a variable, shall not be referenced, become defined, or become undefined until the execution of the matching `WAIT` statement. However, it is allowed for a pointer to become associated with such a variable.

Multiple outstanding asynchronous data transfer operations (`READ` or `WRITE`) are allowed; however, no two `WRITE` operations may use the same unit and record number without an intervening `WAIT`.

Advice to users. HPF permits left-to-right definition of the I/O list on a `READ`, whether or not it is asynchronous. This means that a statement such as

```
        READ(10,ID=IDNUM,REC=10) I,A(I)
```

is conforming and has the same input behavior as a synchronous `READ`. (*End of advice to users.*)

In section 9, change "and `INQUIRE` statements" to ", `INQUIRE`, and `WAIT` statements". In section 9.6.1.14, add the following sentence as the last sentence of the paragraph:

If there are outstanding data transfer operations for the specified unit, the value assigned to the `NEXTREC=` specifier is computed as if all the outstanding data transfers had already completed, in the order in which they were issued.

To section 9.6.1, rule R924 *inquire-spec*, add

or `ID =` *scalar-default-int-variable*
or `PENDING =` *scalar-default-logical-variable*

and also add the constraints

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

Constraint: If either an `ID=` specifier or a `PENDING=` specifier is present, then both shall be present.

Add a new section after 9.6.1.22, entitled "**ID= and PENDING= specifiers in the INQUIRE statement**", containing the following paragraph:

If an `ID=` specifier is present in an `INQUIRE` statement, then 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.

10.1 The WAIT Statement

H1001 *wait-stmt*	**is**	`WAIT (` *wait-spec-list* `)`
H1002 *wait-spec*	**is**	`UNIT =` *io-unit*
	or	`ID =` *scalar-default-int-expr*
	or	`ERR =` *label*
	or	`IOSTAT =` *label*

Constraint: A *wait-spec-list* shall contain exactly one `UNIT=` specifier, exactly one `ID=` specifier, and at most one of each of the other specifiers.

The `WAIT` statement terminates an asynchronous data transfer. The `IOSTAT=` and `ERR=` specifiers are optional and are described in sections 9.4.1.4 and 9.4.1.5, respectively.

Advice to implementors. Implementors may choose to implement any or all asynchronous I/O synchronously. This essentially means using the `ID=` clause on the data transfer statement to record the results of the transfer, then supplying the results to the matching `WAIT` statement. (*End of advice to implementors.*)

Section 11

Approved Extensions for HPF Extrinsics

This section builds on Section 6 by defining specific interfaces for HPF with different models of parallelism, `LOCAL` and `SERIAL` (Section 11.1), and different languages, HPF (Section 11.3), C (Section 11.4), Fortran (Section 11.5), and Fortran 77 (Section 11.6). Library routines useful in the extrinsic models are defined in Section 11.7. These are defined with Fortran bindings. An implementation may chose to define similar routines for use with C. The intent of the HPF extrinsic mechanism is to generalize. These definitions may serve as a model for other interfaces. Some additional extrinsic interfaces are given after Annex E.

In HPF 1.1, specific extrinsic types `HPF`, `HPF_LOCAL`, and `HPF_SERIAL` were defined. There was also a short discussion of `F90_LOCAL`. In this more general setting, the `HPF` keywords are retained for compatibility. As defined in Section 6, a model of `HPF` refers to the global language, `HPF_LOCAL` is the same as `LANGUAGE='HPF',MODEL='LOCAL'`, and `HPF_SERIAL` is the same as `LANGUAGE='HPF',MODEL='SERIAL'`. The `F90_LOCAL` extrinsic is now addressed by the `LANGUAGE='FORTRAN',MODEL='SERIAL'` extrinsic kind.

From the caller's standpoint, an invocation of an extrinsic procedure from a "global" HPF program has the same semantics as an invocation of a regular procedure. The callee may see a different picture. The following sections describe sets of conventions for coding callees in the various extrinsic options. The set of extrinsic options supported by a particular HPF compiler is implementation dependent. They are not limited to those described in this chapter. Furthermore, the language processor used to compile the actual extrinsic subprogram need not be an HPF compiler. More specifically, it need not actually be a compiler for the language noted in the `LANGUAGE` specification, as long as the executing extrinsic code conforms to the conventions defined for the language. We define these interfaces to promote portability and interoperability, but a given implementation and the programmer are free to create other combinations of models and languages.

11.1 Alternative Extrinsic Models: LOCAL and SERIAL

A global HPF program may be thought of as a set of processors cooperating in a loosely synchronous fashion on a single logical thread of program control. Section 6 defines two additional models that may be invoked from global HPF: `LOCAL`, where the model is single-processor "node" code, in which all active processors participate, but with only the data that is mapped to a given physical processor directly accessible, and `SERIAL`, where the model is a single-processor operating alone, with all necessary data aggregated by the caller before the serial subprogram is invoked. As examples of use, the `LOCAL` model is useful for programs that drop down into code with explicit message passing for data communications, while the `SERIAL` model may be needed for calls to system libraries or specialized I/O routines.

Both the `LOCAL` and the `SERIAL` models can be invoked from a global HPF program, but in general these models may not be mixed. Calling global HPF from local or serial procedures is not allowed. Furthermore, calling a serial procedure from a local one is not allowed. Section 6.3.1 gives more detail about how various models can interact with each other.

Some additional restrictions are placed on all local and serial subprograms invoked from global HPF:

- The subprogram directly invoked by global HPF must not be recursive.
- The subprogram directly invoked by global HPF must not use any alternate return mechanism.

The behavior of I/O statements in local and serial subprograms is implementation dependent.

11.1.1 The LOCAL Model

An extrinsic procedure can be defined as explicit SPMD code by specifying the local procedure code that is to execute on each processor. In this section, we describe the contract between the caller and an `EXTRINSIC(MODEL='LOCAL')` callee. It is important not to confuse the extrinsic procedure, which is conceptually a single procedural entity called from the HPF program, with the individual local procedures that are executed on each node, one apiece. An *invocation* of an extrinsic procedure results in a separate invocation of a local procedure on each processor. The *execution* of an extrinsic procedure consists of the concurrent execution of a local procedure on each executing processor. Each local procedure may terminate at any time by executing a `RETURN` statement. However, the extrinsic procedure as a whole terminates only after every local procedure has terminated; in effect, the processors are synchronized before return to a global HPF caller.

With the exception of returning from a local procedure to the global caller that initiated local execution, there is no implicit synchronization required of the locally executing processors. Variables declared in a local procedure are held in local storage, private to each processor. To access data outside the processor requires either preparatory communication to copy data into the processor before running the local code, or explicit communication operations between the separately executing copies of the local procedure. Individual implementations may provide implementation-dependent means for communicating, for example, through a message-passing library or a shared-memory mechanism. Such communication mechanisms are beyond the scope of this specification. Note, however, that many useful portable algorithms that require only independence of control structure can take advantage of local routines, without requiring a communication facility.

The `LOCAL` model assumes only that nonsequential array axes are mapped independently to axes of a rectangular processor grid, each array axis to at most one processor axis (no “skew” distributions) and no two array axes to the same processor axis. This restriction suffices to ensure that each physical processor contains a subset of array elements that can be locally arranged in a rectangular configuration. (Of course, to compute the global indices of an element given its local indices, or vice versa, may be quite a tangled computation—but it will be possible.) In the case of cyclic distributions, multiple sections of the array may be mapped to the local processors.

It is recommended that if, in any given implementation, an extrinsic type does not obey the conventions described in this section, then its model name or keyword should not contain the word `LOCAL`.

11.1.1.1 Conventions for Calling LOCAL Subprograms

The default mapping of scalar dummy arguments, of scalar function results, and of sequential arrays is such that the argument is replicated on each physical processor. These mappings may, optionally, be explicit in the interface (except in the case of sequential arrays, which may not be explicitly mapped), but any other explicit mapping is not HPF conforming. Dummy arguments may not be of explicitly mapped derived types or have explicitly mapped structure components.

As in the case of non-extrinsic subprograms, actual arguments may be mapped in any way; if necessary, they are copied automatically to correctly mapped temporaries before invocation of and after return from the extrinsic procedure.

It should be noted that the conventions for calling local subprograms apply only at the interface between the `GLOBAL` and `LOCAL` models. The conventions do not propagate to further subprograms called from within the `LOCAL` model.

11.1.1.2 LOCAL Calling Sequence

Execution of an extrinsic local procedure must be performed as if the actions detailed below occur prior to the invocation of the local procedure on each processor (see Section 6.3.2 for a related list of conditions and for the meaning of *as if*). Any actions thus required are enforced by the compiler of the calling routine, and are not the responsibility of the programmer, nor do they impact the local procedure.

1. The processors are synchronized. In other words, all actions that logically precede the call are completed.

2. Each actual argument is remapped, if necessary, according to the directives (explicit or implicit) in the declared interface for the extrinsic procedure. Thus, HPF mapping directives appearing in the interface are binding—the compiler must obey these directives in calling local extrinsic procedures. (The reason for this rule is that data mapping is explicitly visible in local routines). Actual arguments corresponding to sequential arrays and scalar dummy arguments are replicated (by broadcasting, for example) in all processors. Scalars of derived types with explicitly mapped structure components or of an explicitly mapped derived type cannot be passed from global HPF to an extrinsic local procedure.

3. If a variable accessible to the called routine has a replicated representation, then all copies are updated prior to the call to contain the correct current value according to the sequential semantics of the source program.

After these actions have occurred, the local procedure is invoked on each processor. The information available to the local invocation is described below in Section 11.1.1.3.

When control is transferred back to the caller at the conclusion of the extrinsic local procedure, execution must resume as if the following actions have already been performed.

1. All processors are synchronized after the call. In other words, global computation procedes as if the execution of every copy of the local routine is completed before execution in the caller is resumed.

2. The original distribution of arguments (and of the result of an extrinsic function) is restored, if necessary.

   Advice to implementors. An implementation might check, before returning from the local subprogram, to make sure that replicated variables have been updated consistently by the subprogram. However, there is certainly no requirement—perhaps not even any encouragement—to do so. This is the responsibilty of the local subprogram, and any checks in the caller are a tradeoff between speed and, for instance, debuggability. (*End of advice to implementors.*)

11.1.1.3 Information Available to the Local Procedure

The local procedure invoked on each processor is passed a *local argument* for each *global argument* passed by the caller to the (global) extrinsic procedure interface. Each global argument is an HPF array or scalar. The corresponding local argument is the part of the global array stored locally, or a local copy of a scalar argument or sequential array replicated across processors. Note that if the HPF array is replicated, each local procedure receives a copy of the entire actual. An array actual argument passed by an HPF caller is called a *global array*; the subgrid of that global array passed to one copy of a local routine (because it resides in that processor) is called a *local array.*

If the extrinsic procedure is a function, then the local procedure is also a function. Only scalar-valued extrinsic functions are allowed. All local functions must return the same value.

If a global HPF program calls local subprogram `A` with an actual array argument `X`, and `A` receives a portion of array `X` as dummy argument `P`, then `A` may call another local subprogram `B` and pass `P` or a section of `P` as an actual argument to `B`.

The run-time interface must provide enough information that each local function can discover for each local argument the mapping of the corresponding global argument, translate global indices to local indices, and vice-versa. A specific set of procedures that provide this information is listed in the HPF Local Library Section 11.7.1. The manner in which this information is made available to the local routine depends on the implementation and the programming language used for the local routine.

11.1.2 The SERIAL Model

This section defines a set of conventions for writing code in which an instance of a subprogram executes on only one processor (of which there may be more than one).

If a program unit has extrinsic model `SERIAL`, an HPF compiler should assume that the subprogram is coded to be executed on a single processor. From the point of view of a global HPF caller, the `SERIAL` procedure behaves the same as an identically coded HPF procedure would. Differences might only arise in implementation-specific behavior (such as the performance).

There is currently no manner in which to specify which processor is to execute an `HPF_SERIAL` procedure.

11.1.2.1 SERIAL Calling Sequence

Prior to invocation of a `SERIAL` procedure from global HPF, the behavior of the program will be as if the following actions take place:

1. The processors are synchronized. All actions that logically precede the call are completed.
2. All actual arguments are remapped to the processor that will actually execute the `SERIAL` procedure. Each argument will appear to the `SERIAL` procedure as a sequential argument.

The behavior of the `SERIAL` procedure will be as if it was executed on only one processor. After the instance of the `SERIAL` procedure invoked from global HPF has completed, the behavior will be as if the following happens:

1. All processors are synchronized after the call.
2. The original mappings of actual arguments are restored.

11.2 Extrinsic Language Bindings

The previous section lays out the rules and considerations for execution models defined for HPF extrinsics. The HPF extrinsic interface is also used to tell the compiler what the language conventions of a called subprogram are. Four language bindings are defined here: `HPF, Fortran, F77`, and `C`. A given implementation may support additional interfaces or allow a user to construct custom interfaces. Taken together, these sections define the special extrinsics `HPF_LOCAL` and `HPF_SERIAL`.

The key feature of the language interface is an extended set of attributes for dummy arguments in explicit extrinsic interfaces, which can give the programmer control over aspects of argument passing between procedures of different extrinsic types. This mechanism is used extensively in the interfaces to `C` and `Fortran 77`, but it is defined in this more general context because it can also apply to other language interfaces.

11.2.1 Control of Arguments

The special data attributes for dummy arguments in routines of certain extrinsic types are `MAP_TO`, `LAYOUT`, and `PASS_BY`. These may only appear in data types statements declaring dummy arguments within explicit interfaces to procedures of appropriate extrinsic types. In particular, all of these attributes have been defined for extrinsic interfaces of type `LANGUAGE = 'C'` (Section 11.4), and the latter two have been defined for extrinsic interfaces of type `LANGUAGE = 'F77'` (Section 11.6).

The purpose of this language extension is to increase the flexibility of the `EXTRINSIC` interface mechanism to facilitate argument passing between procedures written in different programming languages. These three attributes provide a convenient way to pass data between nearly equivalent data type representations and array layouts, as well as to allow for different data passing conventions and options. It should be noted, however, that these mechanisms are by no means expected to address the problems of full equivalence between data types and implementations of different languages.

In particular, the `MAP_TO` attribute is designed to provide a standard, portable mechanism for passing arguments between data types (in different languages) that have substantial overlap but not necessarily identical ranges of values or an identical machine representation. The programmer, not the language implementer, retains the responsibility for determining whether or not any actual argument's value will be adequately represented in the new data type, or whether that value may be altered in successive operations involved in conversion first to the new language, in operations within the extrinsic procedure, and then potentially in conversion back to the original language. The `LAYOUT` attribute is used in cases when the ordering of array elements within one or more processors may need to be changed when passing them as arguments between procedures of different languages. Finally, the `PASS_BY` attribute is designed to offer more detailed control of passing mechanisms for arguments to allow for differences between language implementations, to choose between distinct passing options offered in the non-HPF language, and to enable passing implementation-specific data structures when it is desired to convey HPF mapping information along with data values to non-HPF procedures.

These attributes are specified by an extension of the syntax rule R503 for *attr-spec* in the Fortran standard. Rule R501 for *type-declaration-stmt* is not changed except to refer to the extended *attr-spec*. The first two constraints below are repeated without change from the Fortran standard for clarity, since they apply generally to all attributes. The remaining constraints in the Fortran standard following rules R501 through R506 are specific to attributes already defined in the standard and will also be assumed but not repeated here.

These changes to Fortran syntax are made in anticipation of such extensions being considered for addition to the standard language in the next revision, as language interoperability is an area of active interest to the full Fortran community.

H1101 *type-declaration-stmt-extended*	**is**	*type-spec* [[, *attr-spec-extended*] ... ::] *entity-decl-list*
H1102 *attr-spec-extended*	**is**	`PARAMETER`
	or	*access-spec*
	or	`ALLOCATABLE`
	or	`DIMENSION (` *array-spec* `)`
	or	`EXTERNAL`
	or	`INTENT (` *intent-spec* `)`
	or	`INTRINSIC`
	or	`OPTIONAL`
	or	`POINTER`
	or	`SAVE`
	or	`TARGET`
	or	`MAP_TO (` *map-to-spec* `)`
	or	`LAYOUT (` *layout-spec* `)`
	or	`PASS_BY (` *pass-by-spec* `)`
H1103 *map-to-spec*	**is**	*scalar-char-initialization-expr*
H1104 *layout-spec*	**is**	*scalar-char-initialization-expr*
H1105 *pass-by-spec*	**is**	*scalar-char-initialization-expr*

Constraint: The same *attr-spec-extended* shall not appear more than once in a given *type-declaration-stmt*.

Constraint: An entity shall not be explicitly given any attribute more than once in a scoping unit.

Constraint: The attributes `MAP_TO`, `LAYOUT`, and `PASS_BY` may be specified only for dummy arguments within a scoping unit of an extrinsic type for which these attributes have been explicitly defined.

The definition of *characteristics of a dummy data object* as given in F95:12.2.1.1 and extended in Section 8.15 is further extended to include the dummy data object's `MAP_TO`, `LAYOUT`, and `PASS_BY` attributes.

In the `MAP_TO` attribute, values of *map-to-spec* are intended to describe how the data type of the named actual argument is mapped to the data type of the dummy argument in the extrinsic procedure. An example is given in Section 11.4.2.1

For a given extrinsic type that allows the `MAP_TO` attribute, the set of permitted values for the *map-to-spec* will be specified. If the range of permitted values of the type and mapped-to type differ, and the value of the actual argument or some subobject of the actual argument is not within the permitted range of the mapped-to type, the value of the associated dummy argument or subobject becomes undefined. Conversely, if the value of the dummy argument or some subobject of the dummy is not within the permitted range of values of the associated actual argument, and the associated actual argument is a variable, the value of the associated actual argument or subobject of the actual becomes undefined.

If there is a mismatch between the precision, representation method, range of permitted values, or storage sequence between the type of the dummy argument and the permitted mapped-to type of the dummy argument, the compiler shall ensure that, for the duration of the reference to the extrinsic, the dummy argument is represented in a manner that is compatible with the expectations of the callee for an object of the permitted mapped-to type. Upon return from the procedure, the compiler shall ensure that the value of an actual argument that is a variable is restored to the specified type and kind.

Advice to users. This rule was created to ensure the portability of interoperability. However, it should be noted that for large objects, a significant overhead may be incurred if there is a mismatch between the representation method used for the data type versus the representation method used for the permitted mapped-to type. (*End of advice to users.*)

In the `LAYOUT` attribute, any permitted values of *layout-spec* for a given extrinsic interface are intended to describe how the data layout of the named actual argument is mapped to the data layout of the dummy argument in the extrinsic procedure. An example is given in Section 11.6.3. If no `LAYOUT` attribute is specified for a dummy array argument, the data layout shall be the same as if it were being passed to an HPF procedure of the same model, unless another default layout is defined for the given extrinsic type.

In the `PASS_BY` attribute, any permitted values of *pass-by-spec* for a given extrinsic interface indicate a choice of mechanism used to associate the named actual argument with the dummy argument in the extrinsic procedure. Examples are given in Sections 11.6.3

and 11.4.2.1. If no PASS_BY attribute is specified, the argument association mechanism is implementation dependent, based on the compiler's knowlege of the extrinsic language processor.

11.3 HPF Bindings

HPF is the default language assumption. It requires no MAP_TO, LAYOUT, or PASS_BY attributes in explicit interface definitions. All required subprogram binding information can be accomplished via the standard Fortran explicit interface.

The rules stated in section 14.7 of the Fortran standard will apply to variables defined in Fortran-based SERIAL and LOCAL scoping units. In particular, if the definition status, association status, or allocation status of a variable is defined upon execution of a RETURN statement or an END statement in a Fortran subprogram, such a variable in an SERIAL or LOCAL subprogram will be defined upon execution of a RETURN statement or an END statement.

Any I/O performed within an extrinsic subprogram of a different model, and the correspondence between file names and unit numbers used in global HPF and those used in local or serial subprogram code will be implementation defined.

11.3.1 Additional Special Considerations for HPF_LOCAL

There are some considerations about what HPF features may be used in writing a local, per-processor procedure.

Local program units can use all HPF constructs except for REDISTRIBUTE and REALIGN. Moreover, DISTRIBUTE, ALIGN, and INHERIT directives may be applied only to dummy arguments; that is, every *alignee* and *distributee* must be a dummy argument, and every *align-target* must be a template or a dummy argument. Mapping directives in local HPF program units are understood to have global meaning, as if they had appeared in global HPF code, applying to the global array of which a portion is passed on each processor. (The principal use of such mapping directives is in an HPF_LOCAL module that is used by a global HPF module.)

HPF_ALIGNMENT, HPF_TEMPLATE, and HPF_DISTRIBUTION, the distribution query library subroutines, may be applied to non-sequential local arrays. Their outcome is the same as for a global array that happens to have all its elements on a single node.

As introduced in Section 6.3.1, a local HPF program unit may not access global HPF data other than data that is accessible, either directly or indirectly, via the actual arguments. In particular, a local HPF program unit does not have access to global HPF COMMON blocks; COMMON blocks appearing in local HPF program units are not identified with global HPF COMMON blocks. The same name may not be used to identify a COMMON block both within a local HPF program unit and an HPF program unit in the same executable program.

Like local variables in local subprograms, COMMON blocks in local subprograms contain local data, held in local storage on each processor. This storage is only accessible locally, and it will in general contain data that is different on each processor. Indeed, the size of a local COMMON block can be different on each processor.

According to the Fortran specification, a COMMON block that goes out of scope is not preserved, unless it has the SAVE attribute. This is true of local COMMON blocks, which should be given the SAVE attribute if they are intended to convey information between calls to the local subprogram.

Scalars of an explicitly mapped derived type cannot be passed to an HPF_LOCAL subprogram.

The attributes (type, kind, rank, optional, intent) of the dummy arguments must match the attributes of the corresponding dummy arguments in the explicit interface. A dummy argument of an EXTRINSIC('HPF','LOCAL') routine may not be a procedure name.

A dummy argument of an EXTRINSIC('HPF','LOCAL') routine may not have the POINTER attribute.

A nonsequential dummy array argument of an EXTRINSIC('HPF','LOCAL') routine must have assumed shape. Note that, in general, the shape of a dummy array argument differs from the shape of the corresponding actual argument, unless there is a single executing processor.

Explicit mapping directives for dummy arguments may appear in a local procedure. Such directives are understood as applying to the global array whose local sections are passed as actual arguments or results on each processor. If such directives appear, corresponding mapping directives must be visible to every global HPF caller. This may be done by providing an interface block in the caller, or by placing the local procedure in a module of extrinsic kind HPF_LOCAL that is then used by the global HPF program unit that calls the local procedure.

An EXTRINSIC('HPF','LOCAL') routine may not be invoked, either directly or indirectly, in the body of a FORALL construct or in the body of an INDEPENDENT loop.

A local procedure may have several ENTRY points. A global HPF caller must contain a separate extrinsic interface for each entry point that can be invoked from the HPF program.

11.3.2 Argument Association

If a dummy argument of an EXTRINSIC('HPF','LOCAL') routine is a scalar, then the corresponding dummy argument of the local procedure must be a scalar of the same type and type parameters. Only scalars of intrinsic types, or of derived types that are not explicitly mapped, may be passed from a global to an HPF_LOCAL routine. When the extrinsic procedure is invoked, the local procedure is passed an argument that consists of a local copy of the scalar. This copy will be a valid HPF scalar.

If a dummy argument of an EXTRINSIC('HPF','LOCAL') routine is an array, then the corresponding dummy argument in the specification of the local procedure must be an array of the same rank, type, and type parameters.

If the array is sequential in the extrinsic interface, the corresponding actual argument will be passed by replicatation on all processors, just as scalar arguments are passed. Each local dummy argument will be associated with a full copy of the actual array argument. The dummy argument in the extrinsic interface and the corresponding dummy argument in the specification of the local procedure may be declared with the same explicit shape. All sequential dummy arguments passed by replication to an EXTRINSIC('HPF','LOCAL') procedure must either be INTENT(IN) arguments or should be updated consistently across processors.

If the dummy argument is a nonsequential array, then, when the extrinsic procedure is invoked, the local dummy argument is associated with the local array that consists of the subgrid of the global array that is stored locally. This local array will be a valid HPF array.

If an EXTRINSIC('HPF','LOCAL') routine is a function, then the local procedure is a function that returns a scalar of the same type and type parameters as the HPF extrinsic function. The value returned by each local invocation must be the same.

Each physical processor has at most one copy of each HPF variable.

Consider the following extrinsic interface:

```
INTERFACE
  EXTRINSIC('HPF','LOCAL') FUNCTION MATZOH(X, Y) RESULT(Z)
    REAL, DIMENSION(:,:) :: X
    REAL, DIMENSION(:) :: Y
    REAL Z
    !HPF$ ALIGN WITH X(:,*) :: Y(:)
      ! note that this asserts that size(Y) = size(X,1)
    !HPF$ DISTRIBUTE X(BLOCK, CYCLIC)
  END FUNCTION
END INTERFACE
```

The corresponding local HPF procedure is specified as follows.

```
EXTRINSIC('HPF','LOCAL') FUNCTION MATZOH(XX, YY) RESULT(ZZ)
  REAL, DIMENSION(:,:) :: XX
  REAL, DIMENSION(5:) :: YY ! assumed shape with lower bound of 5
  REAL ZZ
  NX1 = SIZE(XX, 1)
  LX1 = LBOUND(XX, 1)
  UX1 = UBOUND(XX, 1)
  NX2 = SIZE(XX, 2)
  LX2 = LBOUND(XX, 2)
  UX2 = UBOUND(XX, 2)
  NY  = SIZE(YY, 1)
  LY  = LBOUND(YY, 1)
  UY  = UBOUND(YY, 1)
  ...
END FUNCTION
```

Assume that the function is invoked with an actual (global) array X of shape 3×3 and an actual vector Y of length 3 on a 4-processor machine, using a 2×2 processor arrangement (assuming one abstract processor per physical processor).

Then each local invocation of the function MATZOH receives the following actual arguments:

Processor (1,1)		Processor (1,2)
X(1,1)	X(1,3)	X(1,2)
X(2,1)	X(2,3)	X(2,2)
Y(1)		Y(1)
Y(2)		Y(2)
Processor (2,1)		Processor (2,2)
X(3,1)	X(3,3)	X(3,2)
Y(3)		Y(3)

Here are the values to which each processor would set NX1, LX1, UX1, NX2, LX2, UX2, NY, LY, and UY:

```
Processor (1,1)                    Processor (1,2)
NX1 = 2   LX1 = 1   UX1 = 2        NX1 = 2   LX1 = 1   UX1 = 2
NX2 = 2   LX2 = 1   UX2 = 2        NX2 = 1   LX2 = 1   UX2 = 1
 NY = 2    LY = 5    UY = 6         NY = 2    LY = 5    UY = 6

Processor (2,1)                    Processor (2,2)
NX1 = 1   LX1 = 1   UX1 = 1        NX1 = 1   LX1 = 1   UX1 = 1
NX2 = 2   LX2 = 1   UX2 = 2        NX2 = 1   LX2 = 1   UX2 = 1
 NY = 1    LY = 5    UY = 5         NY = 1    LY = 5    UY = 5
```

An actual argument to an extrinsic procedure may be a pointer. Since the corresponding dummy argument may not have the `POINTER` attribute, the dummy argument becomes associated with the target of the HPF global pointer. In no way may a local pointer become pointer associated with a global HPF target. Therefore, an actual argument may not be of a derived-type containing a pointer component.

Rationale. It is expected that global pointer variables will have a different representation from that of local pointer variables, at least on distributed memory machines, because of the need to carry additional information for global addressing. This restriction could be lifted in the future. (*End of rationale.*)

Other inquiry intrinsics, such as `ALLOCATED` or `PRESENT`, should also behave as expected. Note that when a global array is passed to a local routine, some processors may receive an empty set of elements.

11.3.3 Special Considerations for HPF_SERIAL

There are restrictions that apply to an `HPF_SERIAL` subprogram.

No *specification-directive*, *realign-directive*, or *redistribute-directive* is permitted to be appear in an `HPF_SERIAL` subprogram or interface body.

Rationale. An HPF mapping directive would likely be meaningless in an `HPF_SERIAL` subprogram. Note, however, the *independent-directive* may appear in an `HPF_SERIAL` subprogram, since it may provide meaningful information to a compiler about a `DO` loop or a `FORALL` statement or construct. (*End of rationale.*)

Any dummy data objects and any function result variables of an `HPF_SERIAL` procedure will be considered to be sequential.

An `HPF_SERIAL` subprogram must not contain a definition of a common block that has the same name as a common block defined in an HPF or `HPF_LOCAL` program unit. In addition, an `HPF_SERIAL` subprogram must not contain a definition of the blank common block if an HPF or `HPF_LOCAL` program unit has a definition of the blank common block.

A dummy argument or function result variable of an `HPF_SERIAL` procedure that is referenced in global HPF must not have the `POINTER` attribute. A subobject of a dummy argument or function result of an `HPF_SERIAL` procedure that is referenced in global HPF, must not have the `POINTER` attribute.

A dummy argument of an `HPF_SERIAL` procedure that is referenced in global HPF and any subobject of such a dummy argument must not have the `TARGET` attribute.

A dummy procedure argument of an `HPF_SERIAL` procedure must be an `HPF_SERIAL` procedure.

```
      PROGRAM MY_TEST
        INTERFACE
          EXTRINSIC('HPF','SERIAL') SUBROUTINE GRAPH_DISPLAY(DATA)
            INTEGER, INTENT(IN) :: DATA(:, :)
          END SUBROUTINE GRAPH_DISPLAY
        END INTERFACE

        INTEGER, PARAMETER :: X_SIZE = 1024, Y_SIZE = 1024

        INTEGER DATA_ARRAY(X_SIZE, Y_SIZE)
!HPF$   DISTRIBUTE DATA_ARRAY(BLOCK, BLOCK)

!  Compute DATA_ARRAY
        ...
        CALL DISPLAY_DATA(DATA_ARRAY)
      END PROGRAM MY_TEST

! The definition of a graphical display subroutine.
! In some implementation-dependent fashion,
! this will plot a graph of the data in DATA.

      EXTRINSIC('HPF','SERIAL') SUBROUTINE GRAPH_DISPLAY(DATA)
        INTEGER, INTENT(IN) :: DATA(:, :)
        INTEGER :: X_IDX, Y_IDX

        DO Y_IDX = LBOUND(DATA, 2), UBOUND(DATA, 2)
          DO X_IDX = LBOUND(DATA, 1), UBOUND(DATA, 1)
          ...
          END DO
        END DO
      END SUBROUTINE GRAPH_DISPLAY
```

11.4 C Language Bindings

A common problem faced by Fortran users is the need to call procedures written in other languages, particularly those written in C or ones that have interfaces that can be described by C prototypes. Although many Fortran implementations provide methods that solve this problem, these solutions are rarely portable.

This section defines a method of specifying interfaces to procedures defined in C that removes most of the common obstacles to interoperability, while retaining portability.

11.4.1 Specification of Interfaces to Procedures Defined in C

If a user wishes to specify that a procedure is defined by a C procedure, this is specified with an *extrinsic-spec-arg* of `LANGUAGE = 'C'`, or an *extrinsic-kind-keyword* of `C`, as specified in Section 6.

For C subprograms for which `EXTRINSIC (LANGUAGE = 'C')` has been specified, the constraints associated with the syntax for *attr-spec-extended* (H1102) are extended as follows:

Constraint: A `LANGUAGE = 'C'` function shall have a scalar result of type integer, real or double precision.

Constraint: A dummy argument of a `LANGUAGE = 'C'` procedure shall not be an assumed-shape array, shall not have the `POINTER` attribute, shall not have the `TARGET` attribute, nor shall it have a subobject that has the `POINTER` attribute.

Constraint: The bounds of a dummy argument shall not be specified by specification expressions that are not constant specification expressions, nor shall the character length parameter of a dummy argument of such a procedure be specified by a specification expression that is not a constant specification expression.

Constraint: A *dummy-arg-list* of a `LANGUAGE = 'C'` subroutine shall not have a *dummy-arg* that is `*` or a dummy procedure.

The value of the *scalar-char-initialization-expr* in the `EXTERNAL_NAME` specifier gives the name of the procedure as defined in C. This value need not be the same as the procedure name specified by the *function-stmt* or *subroutine-stmt.* If `EXTERNAL_NAME` is not specified, it is as if it were specified with a value that is the same as the procedure name in lower case letters.

> *Advice to users.* Note that the `EXTERNAL_NAME` specifier does not necessarily specify the name by which a binder knows the procedure. It specifies the name by which the procedure would be known if it were referenced by a C program, and the HPF compiler is required to perform any transformations of that name that the C compiler would perform.
>
> The `EXTERNAL_NAME` specifier also allows the user to specify a name that might not be permitted by an HPF compiler, such as a name beginning with an underscore, or as a way of enforcing the distinction between upper and lower case characters in the name. (*End of advice to users.*)

The *extrinsic-spec-arg* of `LANGUAGE = 'C'` helps a compiler identify a procedure that is defined in C so that it can take appropriate steps to ensure that the procedure is invoked in the manner required by the C compiler.

> *Advice to implementors.* A vendor may feel compelled to provide support for more than one C compiler, if different C compilers available for a system provide different procedure calling conventions or different data type sizes. For instance, a vendor's compiler may provide support for a value of `GNU_C` in the `LANGUAGE=` specifier, or it may provide support through the use of compiler switches. (*End of advice to implementors.*)

11.4.2 Specification of Data Type Mappings for C

The extrinsic dummy argument feature, consisting of the `MAP_TO`, `LAYOUT`, and `PASS_BY` attributes, is the principal feature that facilitates referencing procedures defined in C from within Fortran programs. Together, these attributes allow the user to specify conversions required to associate the actual arguments specified in the procedure reference with the formal arguments defined by the referenced procedure. In particular, the MAP_TO attribute indicates the type of the C data to which the HPF data shall be converted by the compiler; the PASS_BY attribute indicates whether a C pointer to the dummy argument needs to be passed; the LAYOUT attribute indicates for an array whether the array element order needs to be changed from Fortran's array element ordering to C's.

For C, the constraints associated with *attr-spec-extended*, *map-to-spec*, *layout-spec*, and *pass-by-spec* (H1102–H1105) are further extended as follows.

Constraint: The MAP_TO attribute shall be specified for all dummy arguments and function result variables of a LANGUAGE = 'C' explicit interface.

Constraint: The *map-to-spec* associated with a dummy argument shall be compatible with the type of the dummy argument. (See below for compatibility rules.)

Constraint: A LAYOUT attribute shall only be specified for a dummy argument that is an array.

Constraint: A LAYOUT attribute shall not be specified for an assumed-size array.

If the compiler is capable of representing letters in both upper and lower case, the value specified for a *map-to-spec*, *layout-spec* or *pass-by-spec* is without regard to case. Any blanks specified for a *map-to-spec*, *layout-spec* or *pass-by-spec* shall be ignored by the compiler for the purposes of determining its value.

An implementation shall provide a module, ISO_C, that shall define a derived type, C_VOID_POINTER. The components of the C_VOID_POINTER type shall be private.

> *Advice to users.* The C_VOID_POINTER type provides a method of using void * pointers in a program, but does not give the user any way of manipulating such a pointer in the Fortran part of the program, since I/O cannot be performed on an object with private components outside the module that defines the type, neither can the components or structure constructor of such a structure be used outside of the module that defines the type. (*End of advice to users.*)

The values permitted for a *map-to-spec* for LANGUAGE = 'C' are 'INT', 'LONG', 'SHORT', 'SIGNED_CHAR', 'FLOAT', 'DOUBLE', 'LONG_DOUBLE', 'CHAR', 'CHAR_PTR', 'VOID_PTR', or a comma-separated list, delimited by parentheses, of any of these values. The HPF types with which these are compatible are shown in the table below.

A *map-to-spec* that is a parenthesized list of values is compatible with a dummy argument of derived type if each value in the list is compatible with the corresponding component of the derived type.

When the PASS_BY attribute is used, the values permitted for a *pass-by-spec* are 'VAL', '*', or '**'. If no PASS_BY attribute is specified, then PASS_BY ('VAL') is assumed. If a *pass-by-spec* of VAL is specified, the dummy argument shall not have the INTENT(OUT) or INTENT(INOUT) attribute specified. If a value of '*' or '**' is specified for the *pass-by-spec*, an associated actual argument shall be a variable.

The value of the *map-to-spec* specified for a dummy argument in the interface body of a procedure for which a LANGUAGE= specifier whose value is C appears shall be such that at least one of the permitted mapped-to types is the same as the C data type of the corresponding formal argument in the C definition of the procedure (or a type that is compatible with one of the permitted mapped-to types). The C data type of a function in the C definition of a procedure shall be one of the permitted mapped-to types (or a type that is equivalent to the permitted mapped-to types) specified for the function result variable in the interface body of a function with the `LANGUAGE=` specifier whose value is `C`. If a subroutine has been specified with a `LANGUAGE=` specifier whose value is `C`, the C definition of the procedure shall be specified with a data type of `void`.

The permitted mapped-to types for scalar dummy arguments of intrinsic type or of the derived type `C_VOID_POINTER`, are shown in the following table.

`MAP_TO`	**Compatible**	**C Type if** `PASS_BY`		
	With	`'VAL'`	`'*'`	`'**'`
`'INT'`	`INTEGER`	`int`	`int*`	`int**`
`'LONG'`	`INTEGER`	`long`	`long*`	`long**`
`'SHORT'`	`INTEGER`	`short`	`short*`	`short**`
`'SIGNED_CHAR'`	`INTEGER`	`signed char`	`signed char*`	`signed char**`
`'FLOAT'`	`REAL`	`float`	`float*`	`float**`
`'DOUBLE'`	`REAL`	`double`	`double*`	`double**`
`'LONG_DOUBLE'`	`REAL`	`double`	`double*`	`double**`
`'CHAR'`	`CHARACTER(1)`	`char`	`char*`	`char**`
`'CHAR_PTR'`	`CHARACTER`	`char*`	`char**`	`char***`
`'VOID_PTR'`	`C_VOID_POINTER`	`void*`	`void**`	`void***`

The permitted mapped-to types of an array are the same as the permitted mapped-to types of a scalar variable of that type followed by a left bracket (`[`), followed by the extent of the corresponding dimension of the dummy argument, followed by a right bracket (`]`), for each dimension of the array. If no value is specified for the `LAYOUT` attribute, the corresponding dimensions of the dummy argument are determined from right to left; if the value `C_ARRAY` is specified for the `LAYOUT` attribute, the corresponding dimensions of the dummy argument are determined from left to right.

The value permitted for a `LANGUAGE = 'C'` *layout-spec* is `C_ARRAY`.

The permitted mapped-to types of a scalar variable of derived type are the structures whose corresponding members are of one of the permitted mapped-to types of the components of the derived type.

If there is a mismatch between the precision, representation method, range of permitted values or storage sequence between the type of the dummy argument and the permitted mapped-to type of the dummy argument, the compiler shall ensure that, for the duration of the reference to a procedure defined with a `LANGUAGE=` specifier whose value is `C`, the dummy argument is represented in a manner that is compatible with the expectations of the C processor for an object of the permitted mapped-to type. Upon return from the procedure, the compiler shall ensure that the value of an actual argument that is a variable is restored to the specified type and kind.

If the range of permitted values of the type and mapped-to type differ and the value of the actual argument or some subobject of the actual argument is not within the permitted range of the mapped-to type, the value of the associated dummy argument or subobject becomes undefined. Conversely, if the value of the dummy argument or some subobject of the dummy is not within the permitted range of values of the associated dummy argument, and the associated actual argument is a variable, the value of the associated actual argument or subobject of the actual becomes undefined.

Advice to users. These rules were created to ensure the portability of interoperability. However, it should be noted that for large objects, a significant overhead may be incurred if there is a mismatch between the representation method used for the data type versus the representation method used for the permitted mapped-to type. (*End of advice to users.*)

Advice to users. In some cases, this may cause the value of the actual argument to change without the value being modified by the procedure referenced. For example,

```
PROGRAM P
  INTERFACE
    EXTRINSIC(LANGUAGE='C') SUBROUTINE C_SUB(R,I)
      REAL(KIND(1.0D0)), MAP_TO('FLOAT'), PASS_BY('*') :: R
      INTEGER, MAP_TO('INT'), PASS_BY('*') :: I
    END SUBROUTINE C_SUB
  END INTERFACE
  REAL(KIND(0.0D0)) RR

  RR = 1.0D0 + 1.0D-10
  I = 123456789
  PRINT *, RR
  CALL C_SUB(RR, I)
  PRINT *, RR
END PROGRAM P

void c_sub(float *r, int *i)
{
}
```

might print

```
1.00000000010000000
1.00000000000000000
```

although the value of `*r` is not modified in `c_sub`. Similarly, the value of `I` might become undefined after the reference to `c_sub`, although `*i` is not modified.

Although it is good practice to avoid specifying a mapped-to type of `float` for a dummy argument of any type other than default real, or a mapped-to type of `double` for a dummy argument of any type other than double precision real, selecting an appropriate dummy argument type for objects requiring a mapped-to type `int` or `long` might not be so simple. (*End of advice to users.*)

If no *layout-spec* is specified for a dummy array argument, the array element order shall be the same as that specified by Fortran. If the value of *layout-spec* specified is `C_ARRAY`, the array element order of the array shall be transposed for the duration of the reference to the procedure.

11.4.2.1 Examples of Data Type Mappings

Some examples should help to clarify what sorts of C procedure definitions would be permitted given an interface body in a Fortran program. For example, the following interface body

```
INTERFACE
  EXTRINSIC('C') SUBROUTINE C_SUB(I, R, DARR, STRUCT)
    INTEGER, MAP_TO('INT') :: I
    REAL, MAP_TO('FLOAT'), PASS_BY('*') :: R
    REAL(KIND(1.0D0)), MAP_TO('DOUBLE') :: DARR(10)
    TYPE DT
      SEQUENCE
      INTEGER :: I, J
    END TYPE DT
    TYPE(DT), MAP_TO('(INT, LONG)'), PASS_BY('*') :: STRUCT
  END SUBROUTINE C_SUB
END INTERFACE
```

could correspond to a C procedure that has the prototype

```
void c_sub(int i, float r*, double darr[10], struct {int i, long j} *)
```

In the following example of the LAYOUT attribute,

```
PROGRAM P
  INTERFACE
    EXTRINSIC('C') SUBROUTINE C_SUB(A, B)
      INTEGER, MAP_TO('INT') :: A(2,2)
      INTEGER, MAP_TO('INT'), LAYOUT('C_ARRAY') :: B(2,2)
    END SUBROUTINE C_SUB
  END INTERFACE

  INTEGER :: AA(2,2), BB(2,2)
  CALL C_SUB(AA, BB)
END PROGRAM P

void c_sub(int a[2][2], b[2][2])
```

the correspondence between elements of AA and a, and elements of BB and b is

AA(1,1)	a[0][0]	BB(1,1)	b[0][0]
AA(2,1)	a[0][1]	BB(2,1)	b[1][0]
AA(1,2)	a[1][0]	BB(1,2)	b[0][1]
AA(2,2)	a[1][1]	BB(2,2)	b[1][1]

11.5 Fortran Language Bindings

When the language specified in an extrinsic definition is Fortran the rules are basically the same as those for HPF because HPF is based on the Fortran standard. There are a few issues to consider in this case:

- Only Fortran constructs should be used. Features such as asynchronous I/O or the HPF library may not be supported.
- It is recommended that Fortran language processors to be used for this purpose be extended to support the `HPF_LOCAL` distribution query routines and the associated `HPF_LOCAL_LIBRARY`
- Assuming the intent is to compile the extrinsic routines with a Fortran processor, these routines should be in separate files rather than incorporated into files with HPF source code.
- The programmer should expect any HPF directives may be ignored.

11.6 Fortran 77 Language Bindings

For language interface purposes, Fortran 77 is still essentially a subset of ANSI/ISO standard Fortran, so most considerations relating to HPF calling Fortran also apply to HPF calling Fortran 77 extrinsic procedures. However, two chief differences between Fortran and Fortran 77 complicate the specification of any `EXTRINSIC(LANGUAGE='F77')`, interface from HPF, especially for the local model:

- Arguments are usually passed differently. Fortran 77 implementations typically pass arguments between subprograms by address (reference). That is, no other information about the actual argument is passed — for example, data type, size, distribution, etc. In contrast, HPF implementations often pass by variables by descriptor in order to make such information available to the subprogram.
- Very different information about how array elements are to be assigned to specific memory locations is available to Fortran 77 and HPF programmers.

  In Fortran 77, the declaration of an array prescribes exactly the mapping of array elements to the linear sequence of storage locations. In HPF, the mapping of array elements to different processors may be controlled (e.g., with `DISTRIBUTION` and `ALIGN` directives) and queried (e.g., with `HPF_ALIGNMENT`, `HPF_DISTRIBUTION`, and `HPF_TEMPLATE`) but there is absolutely no information about how array elements on a given processor are organized within local, serial memory. Even in Fortran 90, assumed shape dummy arrays, for example, do not have to follow the same storage and sequence association rules as Fortran 77 arrays do.

  Indeed, different HPF compilers may organize the data locally in different manners — perhaps including border cells for "stencil" optimizations, or global padding to ensure equal-size subgrids on all processors. Certainly, different HPF compilers are not *bound* to organize local data in any particular manner, and some may choose imaginative orderings in such cases as SMP's, for example.

11.6.1 Special Considerations for F77_LOCAL

The `EXTRINSIC(F77_LOCAL)` interface extends the `HPF_LOCAL` and `FORTRAN_LOCAL` extrinsic interfaces to meet the needs of Fortran 77 programmers.

This `EXTRINSIC` type uses the syntax for calling extrinsic subprograms described above. It can be described more precisely as an `EXTRINSIC(LANGUAGE='F77',MODEL='LOCAL')` interface. The basic conventions for transferring control between global and local routines described previously in Section 11.1 also apply.

However, the differences in argument passing and data distribution between these two languages, as well as the different possible motivations for using such an interface, can be better addressed by allowing additional options for passing data and distribution information. These options are provided with the help of `LAYOUT` and `PASS_BY` attributes.

11.6.2 Argument Passing to F77_LOCAL Procedures

A typical Fortran 77 implementation passes arguments by reference, usually by passing the base address of the location of the first data element, and such arguments may also be assumed to be sequence associated. These facts make it most practical for the default method of passing a distributed data structure from `HPF` to an `F77_LOCAL` procedure by passing the base address of that section of local memory that has been allocated to it. To allow for sequence association of actual and dummy arguments, data should also be reordered or compressed or both, if necessary, on all processors. This is the safest method of passing distributed data to an `EXTRINSIC(F77_LOCAL)` procedure, and hence it should be the default one. However, it tends to have the greatest performance costs.

A second argument passing option is to pass distributed array data "as is" from a global HPF procedure to the local F77 ones, not guaranteeing sequence association of the dummy arguments in order to avoid unwanted local data motion that might be required to compress or reorder the elements of an array local to a processor. In other words, it should be possible to do no more data motion than if the same argument were being passed to another HPF procedure. The guarantee of a sequence associated dummy argument is sacrificed for the possible gains in performance available because the local components of the actual argument are not reordered or compressed. The local programmer must be able to use the implementation-dependent ordering created by the global HPF program.

A third option that can be useful to permit `HPF_LOCAL`-style local programming from an `EXTRINSIC(F77_LOCAL)` procedure call is to pass an array via a descriptor or handle, as is typically done in HPF implementations or for Fortran 90 assumed shape arrays. The local procedure may not access elements of this dummy argument directly but may only pass it on to special utility routines, perhaps to obtain local or global distribution information.

The following attributes suffice to support the above three alternate form of passing data to an `EXTRINSIC(F77_LOCAL)x` procedure:

- `LAYOUT('F77_ARRAY')` indicates that the rectangular configuration should be FORTRAN 77 sequence associated in local, serial memory.

  For example, many compilers add border elements for "stencil" optimizations or pad array allocations on particular processors so that all processors allocate equal amounts of memory for each array. Local reordering eliminates such padding and provides FORTRAN 77 sequence association for actual data values.

  Any local reordering is in addition to any global remapping that may be dictated by `DISTRIBUTION` or `ALIGN` directives in the `INTERFACE` block.

  If no `LAYOUT` attribute is specified, then `LAYOUT('F77_ARRAY')` is assumed.

- `LAYOUT('HPF_ARRAY')` indicates that an array argument is passed just as it would be to a global HPF procedure, with no local reordering of the data.

This option is desirable when the programmer decides that the overhead of local reordering should be eliminated or that certain characteristics of the global HPF compiler's ordering (border cells, equal-size allocations among processors, etc.) should be preserved at the local level. It forces the local programmer to access the local data in whatever implementation-dependent style the global HPF compiler employs.

Furthermore, each argument in the `INTERFACE` block can also have its `PASS_BY` attribute specified to indicate whether the data is passed by reference, for Fortran 77-style access, or via a special handle, perhaps a descriptor used for HPF variable passing, that permits the global HPF caller to pass special mapping information for use within the local Fortran 77 procedure.

- `PASS_BY('*')` indicates that the local procedure should be able to access the dummy argument locally as an F77-style variable, passed by reference.
- `PASS_BY('HPF_HANDLE')` indicates that the local procedure should receive a reference to a global descriptor that can be used with special inquiry routines to obtain useful distribution information.

Thus, the default dummy argument attributes are `LAYOUT('F77_ARRAY')`, a guarantee of sequence association, and `PASS_BY('*')`, an indication that data is being passed via a pointer to its location.

Advice to implementors.

In addition to providing argument passing and data reordering options, a good `EXTRINSIC(F77_LOCAL)` implementation should address the problem of declaring arbitrary sized local subgrids and accessing their elements without being able to describe them as assumed-shape arrays, as in HPF. Dealing with the local results of global data distributions within each local procedure initiated by an extrinsic procedure call can also be difficult without Fortran 90 array inquiry functions and the inquiry subroutines in the HPF Library. Special inquiry routines, callable globally or locally, such as the proposed library of Fortran 77 function interfaces in Annex G are recommended as supplements to the `EXTRINSIC(F77_LOCAL)` procedure interface in order to permit more flexible and efficient use of a broad range of possible global HPF data distributions.

(*End of advice to implementors.*)

11.6.3 F77_LOCAL Programming Examples

11.6.3.1 LAYOUT('F77_ARRAY') and PASS_BY('*')

This example illustrates F77_LOCAL programming using the default `LAYOUT('F77_ARRAY')` and `PASS_BY('*')` attributes, and the use of inquiry routines from the local level using the `LAYOUT('HPF_ARRAY')` attribute.

- HPF caller

```
          PROGRAM EXAMPLE
```

```
! Declare the data array and a verification copy
      INTEGER, PARAMETER :: NX = 100, NY = 100
      REAL, DIMENSION(NX,NY) :: X, Y
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X, Y

! The global sum will be computed
! by forming partial sums on the processors
      REAL PARTIAL_SUM(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE PARTIAL_SUM(BLOCK)

! Local subgrid parameters are declared per processor
! for a rank-two array
      INTEGER, DIMENSION(NUMBER_OF_PROCESSORS(),2) ::
     & LB, UB, NUMBER
!HPF$ DISTRIBUTE(BLOCK,*) :: LB, UB, NUMBER

! Define interfaces
      INTERFACE

        EXTRINSIC(F77_LOCAL) SUBROUTINE LOCAL1
     &   ( LB1, UB1, LB2, UB2, X , X_DESC )
        INTEGER, DIMENSION(:) :: LB1, UB1, LB2, UB2
        REAL,DIMENSION(:,:),LAYOUT('HPF_ARRAY') :: X
        REAL,DIMENSION(:,:),LAYOUT('HPF_ARRAY'),     &
               PASS_BY('HPF_HANDLE')           :: X_DESC

!HPF$   DISTRIBUTE(BLOCK) :: LB1, UB1, LB2, UB2
!HPF$   DISTRIBUTE(BLOCK,BLOCK) :: X, X_DESC
        END

        EXTRINSIC(F77_LOCAL) SUBROUTINE LOCAL2(N,X,R)
        INTEGER N(:)
        REAL X(:,:), R(:)
! Defaults:
!       LAYOUT('F77_ARRAY')     sequential, column-major storage
!       PASS_BY('*')            pass by reference (local address)
!HPF$   DISTRIBUTE N(BLOCK)
!HPF$   DISTRIBUTE X(BLOCK,BLOCK)
!HPF$   DISTRIBUTE R(BLOCK)
        END

      END INTERFACE

! Determine result using only global HPF

      ! Initialize values
      FORALL (I=1:NX,J=1:NY) X(I,J) = I + (J-1) * NX
      ! Determine and report global sum
      PRINT *, 'Global HPF result: ',SUM(X)

! Determine result using local subroutines

      ! Initialize values ( assume stride = 1 )
      CALL HPF_SUBGRID_INFO( Y, IERR, LB=lb, UB=UB )
      IF (IERR.NE.0) STOP 'ERROR!'
      CALL LOCAL1( LB(:,1), UB(:,1), LB(:,2), UB(:,2), Y , Y )

      ! DETERMINE AND REPORT GLOBAL SUM
      NUMBER = UB - LB + 1
      CALL LOCAL2 ( NUMBER(:,1) * NUMBER(:,2) , Y , PARTIAL_SUM )
      PRINT *, 'F77_LOCAL result #1 : ',SUM(PARTIAL_SUM)

      END
```

- FORTRAN 77 callee

```
      SUBROUTINE LOCAL1( LB1, UB1, LB2, UB2, X , DESCRX )

      REAL X ( LB1 : UB1 , LB2 : UB2 )
      INTEGER DESCRX ( * )

! Get the global extent of the first axis
! This is an HPF_LOCAL type of inquiry routine with an 'F77_' prefix
      CALL F77_GLOBAL_SIZE ( NX , DESCRX , 1 )

! Initialize elements of the array
      DO J = LB2, UB2
        DO I = LB2, UB2
          X(I,J) = I + (J-1) * NX
        END DO
      END DO

      END


      SUBROUTINE LOCAL2(N,X,R)

! Here, the correspondence to the global indices is not important
! Only the total size of the subgrid is passed in
      REAL X(N)

      R = 0.
      DO I = 1, N
        R = R + X(I)
      END DO
      END
```

11.6.3.2 LAYOUT('HPF_ARRAY') and PASS_BY('HPF_HANDLE')

This example performs only the initialization of the above example. It illustrates use of the `LAYOUT('F77_ARRAY')` attribute to pass an HPF distributed array without remapping, as well as use of `PASS_BY('HPF_HANDLE')` to pass an HPF-style descriptor or handle for use in the F77_LOCAL subgrid inquiry function. It also illustrates the addressing of data in terms of "embedding arrays."

- HPF caller

```
      PROGRAM EXAMPLE

      INTEGER, PARAMETER :: NX = 100, NY = 100
      REAL, DIMENSION(NX,NY) :: Y
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: Y

! Local subgrid parameters are declared per processor
! for a rank-two array
      INTEGER, DIMENSION(NUMBER_OF_PROCESSORS(),2) ::
     & LB, UB, LB_EMBED, UB_EMBED
!HPF$ DISTRIBUTE(BLOCK,*) :: LB, UB, LB_EMBED, UB_EMBED


! Define interfaces

      INTERFACE

        EXTRINSIC(F77_LOCAL) SUBROUTINE LOCAL1(
     &   LB1, UB1, LB_EMBED1, UB_EMBED1,
     &   LB2, UB2, LB_EMBED2, UB_EMBED2, X, X_DESC )
        INTEGER, DIMENSION(:) ::
     &   LB1, UB1, LB_EMBED1, UB_EMBED1,
     &   LB2, UB2, LB_EMBED2, UB_EMBED2
 !             By default, X is passed by reference
        REAL, DIMENSION(:,:), LAYOUT('HPF_ARRAY') :: X
 !             X_DESC is passed by its descriptor or 'handle'
        REAL, DIMENSION(:,:), LAYOUT('HPF_ARRAY'),
     &                    PASS_BY('HPF_HANDLE') :: X_DESC
!HPF$ DISTRIBUTE(BLOCK) :: LB1, UB1, LB_EMBED1, UB_EMBED1
!HPF$ DISTRIBUTE(BLOCK) :: LB2, UB2, LB_EMBED2, UB_EMBED2
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X
        END

      END INTERFACE
```

```
! Initialize values
! ( Assume stride = 1 and no axis permutation )

      CALL HPF_SUBGRID_INFO( Y, IERR,
     & LB=LB, LB_EMBED=LB_EMBED,
     & UB=UB, UB_EMBED=UB_EMBED)
      IF (IERR.NE.0) STOP 'ERROR!'

      CALL LOCAL1(
     & LB(:,1), UB(:,1), LB_EMBED(:,1), UB_EMBED(:,1),
     & LB(:,2), UB(:,2), LB_EMBED(:,2), UB_EMBED(:,2), Y, Y )

      END
```

- Fortran 77 callee

```
      SUBROUTINE LOCAL1(
     & LB1, UB1, LB_EMBED1, UB_EMBED1,
     & LB2, UB2, LB_EMBED2, UB_EMBED2, X, X_DESC )

! The subgrid has been passed in its 'embedded' form
      REAL X ( LB_EMBED1 : UB_EMBED1 , LB_EMBED2 : UB_EMBED2 )

! This argument is used only as input to inquiry functions
      INTEGER X_DESC

! Get the global extent of the first axis
! This is an HPF_LOCAL type of inquiry routine with an 'F77_' prefix
      CALL F77_GLOBAL_SIZE(NX,X_DESC,1)

! Otherwise, initialize elements of the array
! Loop only over actual array elements
      DO J = LB2, UB2
        DO I = LB2, UB2
          X(I,J) = I + (J-1) * NX
        END DO
      END DO

      END
```

11.7 The Extrinsic Library

Following are Fortran bindings for routines useful in intrinsic subprograms.

11.7.1 HPF Local Routine Library

Local HPF procedures can use any HPF intrinsic or library procedure.

Advice to implementors. The arguments to such procedures will be local arrays. Depending on the implementation, the actual code for the intrinsic and library routines used by local HPF procedures may or may not be the same code used when called from global HPF code. (*End of advice to implementors.*)

In addition, local library procedures GLOBAL_ALIGNMENT, GLOBAL_DISTRIBUTION, and GLOBAL_TEMPLATE are provided to query the global mapping of an actual argument to an extrinsic function. Other local library procedures are provided to query the size, shape, and array bounds of an actual argument. These library procedures take as input the name of a dummy argument and return information on the corresponding global HPF actual argument. They may be invoked only by a local procedure that was directly invoked by global HPF code. If module facilities are available, they reside in a module called HPF_LOCAL_LIBRARY; a local routine that calls them should include the statement

```
      USE HPF_LOCAL_LIBRARY
```

or some functionally appropriate variant thereof.

The HPF local routine library identifies each physical processor by an integer in the range 0 to $n - 1$, where n is the value returned by the global HPF_LIBRARY function NUMBER_OF_PROCESSORS. Processor identifiers are returned by ABSTRACT_TO_PHYSICAL, which establishes the one-to-one correspondence between the abstract processors of an HPF processors arrangement and the physical processors. Also, the local library function MY_PROCESSOR returns the identifier of the calling processor.

In all cases, when an argument of one of the procedures of the local HPF library is required to be a local dummy argument associated with a global HPF actual argument, such association is not considered to be transitive. That is, the local dummy argument must be a dummy argument of a procedure which was referenced from global HPF, not from another local subprogram.

11.7.1.1 Accessing Dummy Arguments by Blocks

The mapping of a global HPF array to the physical processors places one or more *blocks*, which are groups of elements with consecutive indices, on each processor. The number of blocks mapped to a processor is the product of the number of blocks of consecutive indices in each dimension that are mapped to it. For example, a rank-one array X with a CYCLIC(4) distribution will have blocks containing four elements, except for a possible last block having $1 + \texttt{SIZE(X)} \bmod 4$ elements. On the other hand, if X is first aligned to a template or an array having a CYCLIC(4) distribution, and a non-unit stride is employed (as is !HPF$ ALIGN X(I) WITH T(3*I)), then its blocks may have fewer than four elements. In this case, when the align stride is three and the template has a block-cyclic distribution with four template elements per block, the blocks of X have either one or two elements each. If the align stride were five, then all blocks of X would have exactly one element, as template blocks to which no array element is aligned are not counted in the reckoning of numbers of blocks.

The portion of a global array argument associated with a dummy argument in an HPF_LOCAL subprogram may be accessed in a block-by-block fashion. Three of the local library routines, LOCAL_BLKCNT, LOCAL_LINDEX, and LOCAL_UINDEX, allow easy access to the local storage of a particular block. Their use for this purpose is illustrated by the following example, in which the local data are initialized one block at a time:

```
EXTRINSIC(HPF_LOCAL) SUBROUTINE NEWKI_DONT_HEBLOCK(X)
REAL X(:,:,:)
INTEGER BL(3)
INTEGER, ALLOCATABLE LIND1(:), LIND2(:), LIND3(:)
INTEGER, ALLOCATABLE UIND1(:), UIND2(:), UIND3(:)

BL = LOCAL_BLKCNT(X)

ALLOCATE LIND1(BL(1))
ALLOCATE LIND2(BL(2))
ALLOCATE LIND3(BL(3))

ALLOCATE UIND1(BL(1))
ALLOCATE UIND2(BL(2))
ALLOCATE UIND3(BL(3))

LIND1 = LOCAL_LINDEX(X, DIM = 1)
UIND1 = LOCAL_UINDEX(X, DIM = 1)

LIND2 = LOCAL_LINDEX(X, DIM = 2)
UIND2 = LOCAL_UINDEX(X, DIM = 2)

LIND3 = LOCAL_LINDEX(X, DIM = 3)
UIND3 = LOCAL_UINDEX(X, DIM = 3)

DO IB1 = 1, BL(1)
  DO IB2 = 1, BL(2)
    DO IB3 = 1, BL(3)
      FORALL (I1 = LIND1(IB1) : UIND1(IB1),  &
              I2 = LIND2(IB2) : UIND2(IB2),  &
              I3 = LIND3(IB3) : UIND3(IB3) ) &
                X(I1, I2, I3) = IB1 + 10*IB2 + 100*IB3
    ENDDO
  ENDDO
ENDDO
END SUBROUTINE NEWKI_DONT_HEBLOCK
```

GLOBAL_ALIGNMENT(ARRAY, ...)

This has the same interface and behavior as the HPF inquiry subroutine `HPF_ALIGNMENT`, but it returns information about the *global* HPF array actual argument associated with the local dummy argument `ARRAY`, rather than returning information about the local array.

GLOBAL_DISTRIBUTION(ARRAY, ...)

This has the same interface and behavior as the HPF inquiry subroutine `HPF_DISTRIBUTION`, but it returns information about the *global* HPF array actual argument associated with the local dummy argument `ARRAY`, rather than returning information about the local array.

GLOBAL_TEMPLATE(ARRAY, ...)

This has the same interface and behavior as the HPF inquiry subroutine `HPF_TEMPLATE`, but it returns information about the *global* HPF array actual argument associated with the local dummy argument `ARRAY`, rather than returning information about the local array.

GLOBAL_SHAPE(SOURCE)

Description. Returns the shape of the global HPF actual argument associated with an array or scalar dummy argument of an HPF_LOCAL procedure.

Class. Inquiry function.

Argument.

`SOURCE`	may be of any type. It may be array valued or a scalar. It must be a dummy argument of an HPF_LOCAL procedure which is argument associated with a global HPF actual argument.

Result Type, Type Parameter and Shape. The result is a default integer array of rank one whose size is equal to the rank of `SOURCE`.

Result Value. The value of the result is the shape of the global actual argument associated with the actual argument associated with `SOURCE`.

Examples. Assuming `A` is declared by the statement

```
INTEGER A(3:100, 200)
```

and is argument associated with `B`, the value of `GLOBAL_SHAPE(B)` is $\left[\ 98\ 200\ \right]$. If `B` is argument associated with the section, `A(5:10, 10)`, the value of `GLOBAL_SHAPE(B)` is $\left[\ 6\ \right]$.

GLOBAL_SIZE(ARRAY, DIM)

Optional argument. `DIM`

Description. Returns the extent along a specified dimension of the global HPF actual array argument associated with a dummy array argument of an HPF_LOCAL procedure.

Class. Inquiry function.

Argument.

`ARRAY`	may be of any type. It must not be a scalar. It must be a dummy argument of an HPF_LOCAL procedure which is argument associated with a global HPF actual argument.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \le$ `DIM` $\le n$, where n is the rank of `ARRAY`.

Result Type, Type Parameter and Shape. Default integer scalar.

Result Value. The result has a value equal to the extent of dimension `DIM` of the actual argument associated with the actual argument associated with `ARRAY` or, if `DIM` is absent, the total number of elements in the actual argument associated with the actual argument associated with `ARRAY`.

Examples. Assuming `A` is declared by the statement

```
  INTEGER A(3:10, 10)
```

and is argument associated with `B`, the value of `GLOBAL_SIZE(B, 1)` is 8. If `B` is argument associated with the section, `A(5:10, 2:4)`, the value of `GLOBAL_SIZE(B)` is 18.

ABSTRACT_TO_PHYSICAL(ARRAY, INDEX, PROC)

Description. Returns processor identification for the physical processor associated with a specified abstract processor relative to a global actual argument array.

Class. Subroutine.

Arguments.

`ARRAY`	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an `INTENT(IN)` argument.
`INDEX`	must be a rank-1 integer array containing the coordinates of an abstract processor in the processors arrangement onto which the global HPF array is mapped. It is an `INTENT(IN)` argument. The size of `INDEX` must equal the rank of the processors arrangement. The value of the i^{th} element must be in the range 1 to e_i, where e_i is the extent of the i^{th} dimension of the processors arrangement.
`PROC`	must be scalar and of type integer. It is an `INTENT(OUT)` argument. It receives the identifying value for the physical processor associated with the abstract processor specified by `INDEX`.

PHYSICAL_TO_ABSTRACT(ARRAY, PROC, INDEX)

Description. Returns coordinates for an abstract processor, relative to a global actual argument array, corresponding to a specified physical processor.

Class. Subroutine.

Arguments.

`ARRAY`	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an `INTENT(IN)` argument.

`PROC`	must be scalar and of type default integer. It is an `INTENT(IN)` argument. It contains an identifying value for a physical processor.
`INDEX`	must be a rank-1 integer array. It is an `INTENT(OUT)` argument. The size of `INDEX` must equal the rank of the processor arrangement onto which the global HPF array is mapped. `INDEX` receives the coordinates within this processors arrangement of the abstract processor associated with the physical processor specified by `PROC`. The value of the i^{th} element will be in the range 1 to e_i, where e_i is the extent of the i^{th} dimension of the processors arrangement.

This procedure can be used only on systems where there is a one-to-one correspondence between abstract processors and physical processors. On systems where this correspondence is one-to-many an equivalent, system-dependent procedure should be provided.

LOCAL_TO_GLOBAL(ARRAY, L_INDEX, G_INDEX)

Description. Converts a set of local coordinates within a local dummy array to an equivalent set of global coordinates within the associated global HPF actual argument array.

Class. Subroutine.

Arguments.

`ARRAY`	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an `INTENT(IN)` argument.
`L_INDEX`	must be a rank-1 integer array whose size is equal to the rank of `ARRAY`. It is an `INTENT(IN)` argument. It contains the coordinates of an element within the local dummy array `ARRAY`. The value of the i^{th} element must be in the range 1 to e_i, where e_i is the extent of the i^{th} dimension of `ARRAY`.
`G_INDEX`	must be a rank-1 integer array whose size is equal to the rank of `ARRAY`. It is an `INTENT(OUT)` argument. It receives the coordinates within the global HPF array actual argument of the element identified within the local array by `L_INDEX`. The value of the i^{th} element will be in the range 1 to e_i, where e_i is the extent of the i^{th} dimension of the global HPF actual argument array associated with `ARRAY`.

GLOBAL_TO_LOCAL(ARRAY, G_INDEX, L_INDEX, LOCAL, NCOPIES, PROCS)

Optional arguments. `L_INDEX`, `LOCAL`, `NCOPIES`, `PROCS`

Description. Converts a set of global coordinates within a global HPF actual argument array to an equivalent set of local coordinates within the associated local dummy array.

Class. Subroutine.

Arguments.

ARRAY	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an INTENT(IN) argument.
G_INDEX	must be a rank-1 integer array whose size is equal to the rank of ARRAY. It is an INTENT(IN) argument. It contains the coordinates of an element within the global HPF array actual argument associated with the local dummy array ARRAY. The value of the i^{th} element must be in the range 1 to e_i, where e_i is the extent of the i^{th} dimension of the global HPF actual argument array associated with ARRAY.
L_INDEX (optional)	must be a rank-1 integer array whose size is equal to the rank of ARRAY. It is an INTENT(OUT) argument. It receives the coordinates within a local dummy array of the element identified within the global actual argument array by G_INDEX. (These coordinates are identical on any processor that holds a copy of the identified global array element.) The value of the i^{th} element will be in the range 1 to e_i, where e_i is the extent of the i^{th} dimension of ARRAY.
LOCAL (optional)	must be scalar and of type LOGICAL. It is an INTENT(OUT) argument. It is set to .TRUE. if the local array contains a copy of the global array element and to .FALSE. otherwise.
NCOPIES (optional)	must be scalar and of type integer. It is an INTENT(OUT) argument. It is set to the number of processors that hold a copy of the identified element of the global actual array.
PROCS (optional)	must be a rank-1 integer array whose size is at least the number of processors that hold copies of the identified element of the global actual array. The identifying numbers of those processors are placed in PROCS. The order in which they appear is implementation dependent.

MY_PROCESSOR()

Description. Returns the identifying number of the calling physical processor.

Class. Pure function.

Result Type, Type Parameter, and Shape. The result is scalar and of type default integer.

Result Value. Returns the identifying number of the physical processor from which the call is made. This value is in the range $0 \leq$ `MY_PROCESSOR` $\leq n - 1$ where n is the value returned by `NUMBER_OF_PROCESSORS`.

LOCAL_BLKCNT(ARRAY, DIM, PROC)

Optional arguments. `DIM`, `PROC`.

Description. Returns the number of blocks of elements in each dimension, or of a specific dimension of the array on a given processor.

Class. Pure function.

Arguments.

`ARRAY`	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.
`PROC` (optional)	must be scalar and of type integer. It must be a valid processor number.

Result Type, Type Parameter, and Shape. The result is of type default integer. It is scalar if `DIM` is present; otherwise the result is an array of rank one and size n, where n is the rank of `ARRAY`.

Result Value.

Case (i): The value of `LOCAL_BLKCNT(ARRAY, DIM, PROC)` is the number of blocks of the ultimate align target of `ARRAY` in dimension `DIM` that are mapped to processor `PROC` and which have at least one element of `ARRAY` aligned to them.

Case (ii): `LOCAL_BLKCNT(ARRAY, DIM)` returns the same value as `LOCAL_BLKCNT(ARRAY, DIM, PROC=MY_PROCESSOR())`.

Case (iii): `LOCAL_BLKCNT(ARRAY)` has a value whose i^{th} component is equal to `LOCAL_BLKCNT(ARRAY,` i`)`, for $i = 1, \ldots, n$, where n is the rank of `ARRAY`.

Examples. Given the declarations

```
      REAL A(20,20), B(10)
!HPF$ TEMPLATE T(100,100)
!HPF$ ALIGN B(J) WITH A(*,J)
!HPF$ ALIGN A(I,J) WITH T(3*I, 2*J)
```

```
!HPF$   PROCESSORS PR(5,5)
!HPF$   DISTRIBUTE T(CYCLIC(3), CYCLIC(3)) ONTO PR
!HPF$   CALL LOCAL_COMPUTE(A, B)
        ...
        ...
        ...
        EXTRINSIC(HPF_LOCAL) SUBROUTINE LOCAL_COMPUTE(X, Y)
        USE HPF_LOCAL_LIBRARY
        REAL X(:,:), Y(:)
        INTEGER NBY(1), NBX(2)
        NBX = LOCAL_BLKCNT(X)
        NBY = LOCAL_BLKCNT(Y)
```

the values returned on the physical processor corresponding to PR(2,4) in NBX is $\begin{bmatrix} 4 & 3 \end{bmatrix}$ and in NBY is $\begin{bmatrix} 1 \end{bmatrix}$.

LOCAL_LINDEX(ARRAY, DIM, PROC)

Optional argument. PROC.

Description. Returns the lowest local index of all blocks of an array dummy argument in a given dimension on a processor.

Class. Pure function.

Arguments.

ARRAY	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument.
DIM	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.
PROC (optional)	must be scalar and of type integer. It must be a valid processor number.

Result Type, Type Parameter, and Shape. The result is a rank-one array of type default integer and size b, where b is the value returned by LOCAL_BLKCNT(ARRAY, DIM [, PROC])

Result Value.

Case (i): The value of LOCAL_LINDEX(ARRAY, DIM, PROC) has a value whose i^{th} component is the local index of the first element of the i^{th} block in dimension DIM of ARRAY on processor PROC. The value of the i^{th} element will be in the range 1 to e_i, where e_i is the extent of the i^{th} dimension of ARRAY.

Case (ii): LOCAL_LINDEX(ARRAY, DIM) returns the same value as LOCAL_LINDEX(ARRAY, DIM, PROC=MY_PROCESSOR()).

Examples. With the same declarations as in the example under LOCAL_BLKCNT, on the physical processor corresponding to PR(2,4) the value returned by LOCAL_LINDEX(X, DIM=1) is $\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$;
the value of LOCAL_LINDEX(X, DIM=2) is $\begin{bmatrix} 1 & 3 & 4 \end{bmatrix}$.

LOCAL_UINDEX(ARRAY, DIM, PROC)

Optional argument. PROC.

Description. Returns the highest local index of all blocks of an array dummy argument in a given dimension on a processor.

Class. Pure function.

Arguments.

ARRAY	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument.
DIM	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.
PROC (optional)	must be scalar and of type integer. It must be a valid processor number.

Result Type, Type Parameter, and Shape. The result is a rank-one array of type default integer and size b, where b is the value returned by LOCAL_BLKCNT(ARRAY, DIM [, PROC])

Result Value.

Case (i): The value of LOCAL_UINDEX(ARRAY, DIM, PROC) has a value whose i^{th} component is the local index of the last element of the i^{th} block in dimension DIM of ARRAY on processor PROC. The value of the i^{th} element will be in the range 1 to e_i, where e_i is the extent of the i^{th} dimension of ARRAY.

Case (ii): LOCAL_UINDEX(ARRAY, DIM) returns the same value as LOCAL_UINDEX(ARRAY, DIM, PROC=MY_PROCESSOR()).

Examples. With the same declarations as in the example under LOCAL_BLKCNT, on the physical processor corresponding to PR(2,4) the value returned by LOCAL_UINDEX(X, DIM=1) is $\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$;
the value of LOCAL_UINDEX(X, DIM=2) is $\begin{bmatrix} 2 & 3 & 4 \end{bmatrix}$.

11.7.2 Library Access from Serial Extrinsics

A SERIAL subprogram may contain references to any HPF_LIBRARY procedure or HPF intrinsic function, except HPF_ALIGNMENT, HPF_DISTRIBUTION or HPF_TEMPLATE. Within a SERIAL scope the HPF_LOCAL_LIBRARY module must not be used.

References to the intrinsic functions NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE will return the same value as if the function reference appeared in global HPF.

Section 12

Approved Extensions to the HPF Intrinsic and Library Procedures

This chapter describes intrinsic and library routines that have been approved as extensions to HPF Version 2.0.

The extended intrinsic procedures include a transpose function that generalizes the Fortran TRANSPOSE intrinsic function. Certain algorithms require access to multidimensional arrays along different axes. In modern machines, it will usually be best to make the array axis along which an inner loop runs the first axis, so that in local memory the elements will be contiguous. A generalized transpose is required to do this data rearrangement, which is not simply a data remapping. In many cases, the result of the transpose will be assigned to a variable whose first axis is distributed with a *dist-format* of (*).

For this sort of operation, FORALL is adequate when the rank and the particular set of axes to be exchanged are known; for example:

```
FORALL(I1 = 1:SIZE(ARRAY,1))
   FORALL(I2 = 1:SIZE(ARRAY,2))
      FORALL(I3 = 1:SIZE(ARRAY,3))
         RESULT(I3,I1,I2) = ARRAY(I1,I2,I3)
      ENDFORALL
   ENDFORALL
ENDFORALL
```

If, however, the relation between input and result axes is to be variable, FORALL is an inconvenient idiom. Thus we have generalized the TRANSPOSE intrinsic function, allowing as arguments an input array (which is to be transposed) of any nonzero rank, and an integer rank-one array (giving the axis permutation) whose size is the rank of the first input array.

The default value for the order argument makes this an extension of the existing Fortran one-argument TRANSPOSE function.

Two new intrinsic inquiry functions, ACTIVE_NUM_PROCS and ACTIVE_PROCS_SHAPE are useful for determining the size and the shape of the processor subset executing the program, as modified by ON constructs.

The extended library consists of mapping inquiry subroutines. Extended versions of HPF_ALIGNMENT and HPF_TEMPLATE allow an additional, optional, DYNAMIC output argument. This allows a program to determine whether an object, or its align ultimate target, has the DYNAMIC attribute. There is a new version of HPF_DISTRIBUTION, and two new mapping inquiry subroutines that are especially useful for determining mappings produced by the general block and indirect distribution forms.

12.1 Specifications of Extended Intrinsic Procedures

ACTIVE_NUM_PROCS(DIM)

Optional Argument. DIM

Description. Returns the total number of processors currently executing the program or the number of processors currently executing the program along a specified dimension of the processor array, as determined by the innermost ON block.

Class. Processors inquiry function.

Arguments.

DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$ where n is the rank of the processor array.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to the extent of dimension DIM of the processor array determined by the innermost containing ON block or, if DIM is absent, the total number of elements of this processor array. The result is always greater than zero. Outside of any ON block, the result is the same as that returned by NUMBER_OF_PROCESSORS().

Examples. The program fragment

```
INTEGER X(16, 3)
!hpf$ TEMPLATE T(16, 8)
!hpf$ PROCESSORS PROCS(4, 4)
!hpf$ ALIGN X(I, J) WITH T(I, 3*J-1)
!hpf$ DISTRIBUTE T(CYCLIC(2), BLOCK) ONTO PROCS
!hpf$ ON     (PROCS(:,:)) BEGIN
!hpf$ ON HOME(X(2:12:10, :)) BEGIN
PRINT *, ACTIVE_NUM_PROCS()
PRINT *, ACTIVE_NUM_PROCS(DIM=1)
PRINT *, ACTIVE_NUM_PROCS(DIM=2)
!hpf$ END ON
!hpf$ END ON
```

prints 6, 2 and 3 regardless of the size or shape of the hardware processor array on which the program is running,

ACTIVE_PROCS_SHAPE()

Description. Returns the shape of the currently active processor array, as determined by the innermost `ON` block.

Class. Processors inquiry function.

Arguments. None

Result Type, Type Parameter, and Shape. The result is a default integer array of rank one whose size is equal to the rank of the processor array determined by the innermost containing `ON` block.

Result Value. The value of the result is the shape of the processor array determined by the innermost containing `ON` block. Outside of any `ON` block, the result is the same as that returned by `PROCESSORS_SHAPE()`.

Examples. The program fragment

```
INTEGER X(16, 3)
!hpf$ TEMPLATE T(16, 8)
!hpf$ PROCESSORS PROCS(4, 4)
!hpf$ ALIGN X(I, J) WITH T(I, 3*J-1)
!hpf$ DISTRIBUTE T(CYCLIC(2), BLOCK) ONTO PROCS
!hpf$ ON     (PROCS(:,:)) BEGIN
    PRINT *, ACTIVE_PROCS_SHAPE()
    !hpf$ ON HOME(X(2:12:10, :)) BEGIN
    PRINT *, ACTIVE_PROCS_SHAPE()
    !hpf$ END ON
!hpf$ END ON
```

prints $4, 4$ and $2, 3$ regardless of the size or shape of the hardware processor array on which the program is running,

TRANSPOSE(ARRAY,ORDER)

Optional Argument. `ORDER`

Description. Permute the axes (a generalized transpose) of an array.

Class. Transformational function.

Arguments.

`ARRAY`	may be of any type, and must be array valued.
`ORDER` (optional)	must be of type integer, rank one, and of size equal to the rank of `ARRAY`. Its elements must be a permutation of $(1, 2, \ldots, n)$, where n is `RANK(ARRAY)`.

Result Type, Type Parameters, and Shape. The result is an array of the same rank, type, and type parameters as ARRAY. Its shape satisfies the relation RS(ORDER) == AS, where RS is the shape of the result and AS is SHAPE(ARRAY). If ORDER is absent, it defaults to $(n, n-1, \ldots, 1)$, where n is RANK(ARRAY).

Result value. Element $(j_1, j_2, \ldots, j_n)$ of the result is ARRAY$(j_{order(1)}, j_{order(2)}, \ldots, j_{order(n)})$.

Examples. For an array of rank two, TRANSPOSE(ARRAY) is the usual matrix transpose.

If ARRAY has shape $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ and ARRAY(1,:,:) is $\begin{bmatrix} 111 & 112 & 113 \\ 121 & 122 & 123 \end{bmatrix}$ and ORDER is $\begin{bmatrix} 3\ 1\ 2 \end{bmatrix}$ then the shape of the result is $\begin{bmatrix} 2\ 3\ 1 \end{bmatrix}$; if R is the result then R(:,:,1) is equal to ARRAY(1,:,:). The rule is that axis i of ARRAY becomes axis ORDER(i) of the result.

12.2 Specifications of Extended Library Procedures

HPF_ALIGNMENT(ALIGNEE, LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, DYNAMIC, NCOPIES)

Optional Arguments. LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, DYNAMIC, NCOPIES

Description. Returns information regarding the correspondence of a variable and the *align-target* (array or template) to which it is ultimately aligned.

Class. Mapping inquiry subroutine.

Arguments.

ALIGNEE — may be of any type. It may be scalar or array valued. It must not be an assumed-size array. It must not be a pointer that is disassociated or an allocatable array that is not allocated. It is an INTENT (IN) argument.

If ALIGNEE has the pointer attribute, information about the alignment of its target is returned. The target must not be an assumed-size dummy argument or a section of an assumed-size dummy argument.

LB (optional) — must be of type default integer and of rank one. Its size must be at least equal to the rank of ALIGNEE. It is an INTENT (OUT) argument. The first element of the ith axis of ALIGNEE is ultimately aligned to the LB(i)th *align-target* element along the axis of the *align-target* associated with the ith axis of ALIGNEE. If the ith axis of ALIGNEE is a collapsed axis, LB(i) is implementation dependent.

`UB` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of `ALIGNEE`. It is an `INTENT (OUT)` argument. The last element of the i$^{\text{th}}$ axis of `ALIGNEE` is ultimately aligned to the `UB`(i)$^{\text{th}}$ *align-target* element along the axis of the *align-target* associated with the i$^{\text{th}}$ axis of `ALIGNEE`. If the i$^{\text{th}}$ axis of `ALIGNEE` is a collapsed axis, `UB`(i) is implementation dependent.
`STRIDE` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of `ALIGNEE`. It is an `INTENT (OUT)` argument. The i$^{\text{th}}$ element of `STRIDE` is set to the stride used in aligning the elements of `ALIGNEE` along its i$^{\text{th}}$ axis. If the i$^{\text{th}}$ axis of `ALIGNEE` is a collapsed axis, `STRIDE`(i) is zero.
`AXIS_MAP` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of `ALIGNEE`. It is an `INTENT (OUT)` argument. The i$^{\text{th}}$ element of `AXIS_MAP` is set to the *align-target* axis associated with the i$^{\text{th}}$ axis of `ALIGNEE`. If the i$^{\text{th}}$ axis of `ALIGNEE` is a collapsed axis, `AXIS_MAP(i)` is 0.
`IDENTITY_MAP` (optional)	must be scalar and of type default logical. It is an `INTENT (OUT)` argument. It is set to true if the ultimate *align-target* associated with `ALIGNEE` has a shape identical to `ALIGNEE`, the axes are mapped using the identity permutation, and the strides are all positive (and therefore equal to 1, because of the shape constraint); otherwise it is set to false. If a variable has not appeared as an *alignee* in an `ALIGN` or `REALIGN` directive, and does not have the `INHERIT` attribute, then `IDENTITY_MAP` must be true; it can be true in other circumstances as well.
`DYNAMIC` (optional)	must be scalar and of type default logical. It is an `INTENT (OUT)` argument. It is set to true if `ALIGNEE` has the `DYNAMIC` attribute; otherwise it is set to false.
`NCOPIES` (optional)	must be scalar and of type default integer. It is an `INTENT (OUT)` argument. It is set to the number of copies of `ALIGNEE` that are ultimately aligned to *align-target*. For a non-replicated variable, it is set to one.

If `ALIGNEE` is scalar, then no elements of `LB`, `UB`, `STRIDE`, or `AXIS_MAP` are set.

Examples.

```
      REAL PI = 3.1415927
      DIMENSION A(10,10),B(20,30),C(20,40,10),D(40)
!HPF$ TEMPLATE T(40,20)
!HPF$ DYNAMIC A
```

```
!HPF$ ALIGN A(I,:) WITH T(1+3*I,2:20:2)
!HPF$ ALIGN C(I,*,J) WITH T(J,21-I)
!HPF$ ALIGN D(I) WITH T(I,4)
!HPF$ PROCESSORS PROCS(4,2), SCALARPROC
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO PROCS
!HPF$ DISTRIBUTE B(CYCLIC,BLOCK) ONTO PROCS
!HPF$ DISTRIBUTE ONTO SCALARPROC :: PI
```

assuming that the actual mappings are as the directives specify, the results of calling HPF_ALIGNMENT are:

	A	B	C	D
LB	[4, 2]	[1, 1]	[20, N/A, 1]	[1]
UB	[31, 20]	[20, 30]	[1, N/A, 10]	[40]
STRIDE	[3, 2]	[1, 1]	[-1, 0, 1]	[1]
AXIS_MAP	[1, 2]	[1, 2]	[2, 0, 1]	[1]
IDENTITY_MAP	false	true	false	false
DYNAMIC	true	false	false	false
NCOPIES	1	1	1	1

where "N/A" denotes a implementation-dependent result. To illustrate the use of NCOPIES, consider:

```
      LOGICAL BOZO(20,20),RONALD_MCDONALD(20)
!HPF$ TEMPLATE EMMETT_KELLY(100,100)
!HPF$ ALIGN RONALD_MCDONALD(I) WITH BOZO(I,*)
!HPF$ ALIGN BOZO(J,K) WITH EMMETT_KELLY(J,5*K)
```

Then CALL HPF_ALIGNMENT(RONALD_MCDONALD, NCOPIES = NC) sets NC to 20. Now consider:

```
      LOGICAL BOZO(20,20),RONALD_MCDONALD(20)
!HPF$ TEMPLATE WILLIE_WHISTLE(100)
!HPF$ ALIGN RONALD_MCDONALD(I) WITH BOZO(I,*)
!HPF$ ALIGN BOZO(J,*) WITH WILLIE_WHISTLE(5*J)
```

Then CALL HPF_ALIGNMENT(RONALD_MCDONALD, NCOPIES = NC) sets NC to one.

HPF_DISTRIBUTION(DISTRIBUTEE, AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE, PLB, PUB, PSTRIDE, LOW_SHADOW, HIGH_SHADOW)

Optional Arguments. AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE, PLB, PUB, PSTRIDE, LOW_SHADOW, HIGH_SHADOW.

Description. The HPF_DISTRIBUTION subroutine returns information regarding the distribution of the ultimate *align-target* associated with a variable.

Class. Mapping inquiry subroutine.

Arguments.

`DISTRIBUTEE`	may be of any type. It may be scalar or array valued. It must not be sequential. It must not be a pointer that is disassociated or an allocatable array that is not allocated. It is an `INTENT (IN)` argument.
`AXIS_TYPE` (optional)	must be a rank one array of type default character. It may be of any length, although it must be of length at least 9 in order to contain the complete value. Its elements are set to the values below as if by a character intrinsic assignment statement. Its size must be at least equal to the rank of the *align-target* to which `DISTRIBUTEE` is ultimately aligned; this is the value returned by `HPF_TEMPLATE` in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. Its i$^{\text{th}}$ element contains information on the distribution of the i$^{\text{th}}$ axis of that *align-target*. The following values are defined by HPF (implementations may define other values):

`'BLOCK'` The axis is distributed `BLOCK`. The corresponding element of `AXIS_INFO` contains the block size.

`'GEN_BLOCK'` The axis is distributed `BLOCK(array)`. The value of the corresponding element of `AXIS_INFO` is implementation dependent.

`'COLLAPSED'` The axis is collapsed (distributed with the "`*`" specification). The value of the corresponding element of `AXIS_INFO` is implementation dependent.

`'CYCLIC'` The axis is distributed `CYCLIC`. The corresponding element of `AXIS_INFO` contains the block size.

`'INDIRECT'` The axis is distributed `INDIRECT(map-array)`. The value of the corresponding element of `AXIS_INFO` is implementation dependent.

`AXIS_INFO` (optional)	must be a rank one array of type default integer, and size at least equal to the rank of the *align-target* to which `DISTRIBUTEE` is ultimately aligned (which is returned by `HPF_TEMPLATE` in `TEMPLATE_RANK`). It is an `INTENT (OUT)` argument. The i$^{\text{th}}$ element of `AXIS_INFO` contains the block size in the block or cyclic distribution of the i$^{\text{th}}$ axis of the ultimate *align-target* of `DISTRIBUTEE`; if that axis is a collapsed axis, then the value is implementation dependent.
`PROCESSORS_RANK` (optional)	must be scalar and of type default integer. It is set to the rank of the processor arrangement onto which `DISTRIBUTEE` is distributed. It is an `INTENT (OUT)` argument.

`PROCESSORS_SHAPE` (optional) must be a rank one array of type default integer and of size at least equal to the value, m, returned in `PROCESSORS_RANK`. It is an `INTENT (OUT)` argument. Its first m elements are set to the shape of the processor arrangement onto which `DISTRIBUTEE` is mapped. (It may be necessary to call `HPF_DISTRIBUTION` twice, the first time to obtain the value of `PROCESSORS_RANK` in order to allocate `PROCESSORS_SHAPE`.)

`PLB` (optional) must be a rank one array of type default integer and of size at least equal to the rank of the ultimate *align-target* of `DISTRIBUTEE`. It is an `INTENT (OUT)` argument. The ith element is set to the smallest processor index `ONTO` which the ith axis of the ultimate *align-target* of `DISTRIBUTEE` is mapped; if that axis is collapsed, then the corresponding element of `PLB` is implementation dependent. The value returned is in the range 1 to e_i where e_i is the extent of processor arrangement axis onto which the selected axis of the ultimate *align-target* of `DISTRIBUTEE` is mapped.

`PUB` (optional) must be a rank one array of type default integer and of size at least equal to the rank of the ultimate *align-target* of `DISTRIBUTEE`. It is an `INTENT (OUT)` argument. The ith element is set to the largest processor index `ONTO` which the ith axis of the ultimate *align-target* of `DISTRIBUTEE` is mapped; if that axis is collapsed, then the corresponding element of `PUB` is implementation dependent. The value returned is in the range 1 to e_i where e_i is the extent of processor arrangement axis onto which the selected axis of the ultimate *align-target* of `DISTRIBUTEE` is mapped.

`PSTRIDE` (optional) must be a rank one array of type default integer and of size at least equal to the rank of `DISTRIBUTEE`. It is an `INTENT (OUT)` argument. The ith element is set to the interprocessor stride in the `ONTO` clause with which the ith axis of `DISTRIBUTEE` is mapped; if that axis is collapsed, then the corresponding element of `PSTRIDE` is set to zero.

`LOW_SHADOW` (optional) must be a rank one array of type default integer, and size at least equal to the rank of the *align-target* to which `DISTRIBUTEE` is ultimately aligned (which is returned by `HPF_TEMPLATE` in `TEMPLATE_RANK`). It is an `INTENT (OUT)` argument. The ith element of `LOW_SHADOW` contains the low-side shadow width in the block or cyclic distribution of the ith axis of the ultimate *align-target* of `DISTRIBUTEE`; if that axis is a collapsed axis, then the value is implementation dependent.

`HIGH_SHADOW` (optional)	must be a rank one array of type default integer, and size at least equal to the rank of the *align-target* to which `DISTRIBUTEE` is ultimately aligned (which is returned by `HPF_TEMPLATE` in `TEMPLATE_RANK`). It is an `INTENT (OUT)` argument. The i$^{\text{th}}$ element of `HIGH_SHADOW` contains the high-side shadow width in the block or cyclic distribution of the i$^{\text{th}}$ axis of the ultimate *align-target* of `DISTRIBUTEE`; if that axis is a collapsed axis, then the value is implementation dependent.

Example. Given the declarations in the example illustrating `HPF_ALIGNMENT` and assuming that the actual mappings are as the directives specify, the results of `HPF_DISTRIBUTION` are:

	A	B	PI
AXIS_TYPE	['BLOCK', 'BLOCK']	['CYCLIC', 'BLOCK']	[ ]
AXIS_INFO	[10, 10]	[1, 15]	[ ]
PROCESSORS_SHAPE	[4, 2]	[2, 2]	[ ]
PROCESSORS_RANK	2	2	0
PLB	[4, 1]	[2, 1]	[ ]
PUB	[1, 2]	[3, 2]	[ ]
PSTRIDE	[-1,1]	[1, 1]	[ ]

HPF_TEMPLATE(ALIGNEE, TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO, NUMBER_ALIGNED, DYNAMIC)

Optional Arguments. `LB`, `UB`, `AXIS_TYPE`, `AXIS_INFO`, `NUMBER_ALIGNED`, `TEMPLATE_RANK`, `DYNAMIC`

Description. The `HPF_TEMPLATE` subroutine returns information regarding the ultimate *align-target* associated with a variable; `HPF_TEMPLATE` returns information concerning the variable from the point of view of its ultimate *align-target*, while `HPF_ALIGNMENT` returns information from the variable's point of view.

Class. Mapping inquiry subroutine.

Arguments.

`ALIGNEE`	may be of any type. It may be scalar or array valued. It must not be an assumed-size array. It must not be a pointer that is disassociated or an allocatable array that is not allocated. It is an `INTENT (IN)` argument. If `ALIGNEE` has the pointer attribute, information about the alignment of its target is returned. The target must not be an assumed-size dummy argument or a section of an assumed-size dummy argument.
`TEMPLATE_RANK` (optional)	must be scalar and of type default integer. It is an `INTENT (OUT)` argument. It is set to the rank of the ultimate *align-target*. This can be different from the rank of the `ALIGNEE`, due to collapsing and replicating.

`LB` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of the *align-target* to which `ALIGNEE` is ultimately aligned; this is the value returned in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. The i$^{\text{th}}$ element of `LB` contains the declared *align-target* lower bound for the i$^{\text{th}}$ template axis.
`UB` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of the *align-target* to which `ALIGNEE` is ultimately aligned; this is the value returned in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. The i$^{\text{th}}$ element of `UB` contains the declared *align-target* upper bound for the i$^{\text{th}}$ template axis.
`AXIS_TYPE` (optional)	must be a rank one array of type default character. It may be of any length, although it must be of length at least 10 in order to contain the complete value. Its elements are set to the values below as if by a character intrinsic assignment statement. Its size must be at least equal to the rank of the *align-target* to which `ALIGNEE` is ultimately aligned; this is the value returned in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. The i$^{\text{th}}$ element of `AXIS_TYPE` contains information about the i$^{\text{th}}$ axis of the *align-target*. The following values are defined by HPF (implementations may define other values):

`'NORMAL'` The *align-target* axis has an axis of `ALIGNEE` aligned to it. For elements of `AXIS_TYPE` assigned this value, the corresponding element of `AXIS_INFO` is set to the number of the axis of `ALIGNEE` aligned to this *align-target* axis.

`'REPLICATED'` `ALIGNEE` is replicated along this *align-target* axis. For elements of `AXIS_TYPE` assigned this value, the corresponding element of `AXIS_INFO` is set to the number of copies of `ALIGNEE` along this *align-target* axis.

`'SINGLE'` `ALIGNEE` is aligned with one coordinate of the *align-target* axis. For elements of `AXIS_TYPE` assigned this value, the corresponding element of `AXIS_INFO` is set to the *align-target* coordinate to which `ALIGNEE` is aligned.

`AXIS_INFO` (optional)	must be of type default integer and of rank one. Its size must be at least equal to the rank of the *align-target* to which `ALIGNEE` is ultimately aligned; this is the value returned in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. See the description of `AXIS_TYPE` above.
`NUMBER_ALIGNED` (optional)	must be scalar and of type default integer. It is an `INTENT (OUT)` argument. It is set to the total number

	of variables aligned to the ultimate *align-target*. This is the number of variables that are moved if the *align-target* is redistributed.
`DYNAMIC` (optional)	must be scalar and of type default logical. It is an `INTENT (OUT)` argument. It is set to true if the *align-target* has the `DYNAMIC` attribute, and to false otherwise.

Example. Given the declarations in the example illustrating `HPF_ALIGNMENT`, and assuming that the actual mappings are as the directives specify, the results of `HPF_TEMPLATE` are:

	A	C	D
`LB`	[1, 1]	[1, 1]	[1, 1]
`UB`	[40, 20]	[40, 20]	[40, 20]
`AXIS_TYPE`	`['NORMAL', 'NORMAL']`	`['NORMAL', 'NORMAL']`	`['NORMAL', 'SINGLE']`
`AXIS_INFO`	[1, 2]	[3, 1]	[1, 4]
`NUMBER_ALIGNED`	3	3	3
`TEMPLATE_RANK`	2	2	2
`DYNAMIC`	`false`	`false`	`false`

HPF_MAP_ARRAY(ARRAY, TEMPLATE_DIM, MAP_ARRAY)

Description. Returns the map array used in the indirect distribution of axis `TEMPLATE_DIM` of the ultimate *align-target* associated with `ARRAY`.

Class. Mapping inquiry subroutine.

Arguments.

`ARRAY`	may be of any type. It must not be scalar. It must not be sequential. It must not be a pointer that is disassociated or an allocatable array that is not allocated. It is an `INTENT(IN)` argument.
`TEMPLATE_DIM`	must be scalar and of type default integer. Its value must be between one and the rank of the ultimate *align-target* of `ARRAY`. It is an `INTENT(IN)` argument.
`MAP_ARRAY`	must be of type default integer and of rank one. Its size must be no smaller than the extent of the `PROCESSORS_DIM`th axis of the processors arrangement onto which is distributed the ultimate *align-target* associated with `ARRAY`. It is an `INTENT(OUT)` argument. The i^{th} element of `MAP_ARRAY` is set to the processor index to which the i^{th} element of the ultimate *align-target* of `ARRAY` along axis `TEMPLATE_DIM` is mapped. If axis `TEMPLATE_DIM` of the ultimate *align-target* of `ARRAY` is collapsed, then all elements of the result have the value one.

Example. Given the declarations

```
      DIMENSION A(2)
!HPF$ TEMPLATE T(4,8)
!HPF$ ALIGN A(I,*) WITH T(2*I,5)
!HPF$ PROCESSORS PROCS(2,2)
!HPF$ DISTRIBUTE T(INDIRECT( (/1,2,2,1/) ), BLOCK( (/3,5/) )) ONTO PROCS
```

assuming that the actual mappings are as the directives specify, Then after calling `HPF_MAP_ARRAY(A,TEMPLATE_DIM=1, MAP_ARRAY=M)`, `M` has the value $\begin{bmatrix} 1 & 2 & 2 & 1 \end{bmatrix}$. After calling `HPF_MAP_ARRAY(A,TEMPLATE_DIM=2, MAP_ARRAY=M)`, `M` has the value $\begin{bmatrix} 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 \end{bmatrix}$.

HPF_NUMBER_MAPPED(ARRAY, PROCESSORS_DIM, NUMBER_MAPPED)

Description. Returns the number of elements of the ultimate *align-target* of `ARRAY` mapped to the each element of axis `PROCESSORS_DIM` of the processors arrangement onto which the ultimate *align-target* of `ARRAY` is distributed.

Class. Mapping inquiry subroutine.

Arguments.

`ARRAY`	may be of any type. It must not be scalar. It must not be sequential. It must not be a pointer that is disassociated or an allocatable array that is not allocated.
`PROCESSORS_DIM`	must be scalar and of type default integer. Its value must be between one and the rank of the processors arrangement onto which the ultimate *align-target* of `ARRAY` is distributed.
`NUMBER_MAPPED`	must be of type default integer and of rank one. Its size must be no smaller than the extent of axis `PROCESSORS_DIM` of the processors arrangement onto which the ultimate *align-target* of `ARRAY` is distributed. The i^{th} element of `NUMBER_MAPPED` is set to the number of elements of an axis of the ultimate *align-target* of `ARRAY` that are mapped to the i^{th} processor of axis `PROCESSORS_DIM` of the processors arrangement onto which the ultimate *align-target* of `ARRAY` is distributed. If axis `PROCESSORS_DIM` of the processors arrangement onto which the ultimate *align-target* of `ARRAY` is distributed is associated with a `BLOCK` distributed axis, then `MAP_ARRAY` is set to the array of block sizes used to distribute that axis.

Example. Given the declarations

```
      DIMENSION A(2,40)
!HPF$ TEMPLATE T(4,8,4,16)
!HPF$ ALIGN A(I,*) WITH T(2*I, 5, *, *)
!HPF$ PROCESSORS PROCS(2,2,3)
!HPF$ DISTRIBUTE T(INDIRECT((/2,2,1,2/)),  BLOCK((/3,5/)), *, BLOCK) &
!HPF$      ONTO PROCS
```

assuming that the actual mappings are as the directives specify, after calling `HPF_NUMBER_MAPPED(A,PROCESSORS_DIM=1, NUMBER_MAPPED = M)` `M` has the value $\begin{bmatrix} 1 & 3 \end{bmatrix}$; after calling `HPF_NUMBER_MAPPED(A,PROCESSORS_DIM=2, NUMBER_MAPPED = M)` `M` has the value $\begin{bmatrix} 3 & 5 \end{bmatrix}$; after calling `HPF_NUMBER_MAPPED(A,PROCESSORS_DIM=3, NUMBER_MAPPED = M)` `M` has the value $\begin{bmatrix} 6 & 6 & 4 \end{bmatrix}$.

Part IV

Annexes

This major section organizes descriptions of the syntax and semantics of features of the High Performance Fortran language, version 2.0 (described n Parts I and II) and its approved extensions (described in Part III) for reference use. It is not a part of the HPF language specification proper.

Annex A

Syntax Rules

This Appendix collects the formal syntax definitions of this High Performance Fortran Language Specification.

A.2 Notation and Syntax

A.2.2 Syntax of Directives

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	*inherit-directive*
		or	*template-directive*
		or	*combined-directive*
		or	*sequence-directive*
H205	*executable-directive*	**is**	*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 free form (F95:3.3.1.1) or fixed form (F95:3.3.2.1) comment lines, depending on the source form of the surrounding Fortran source form in that program unit. (F95:3.3)

H206	*specification-directive-extended*	**is**	*processors-directive*
		or	*subset-directive*
		or	*align-directive*
		or	*distribute-directive*
		or	*inherit-directive*
		or	*template-directive*
		or	*combined-directive*
		or	*sequence-directive*
		or	*dynamic-directive*
		or	*range-directive*
		or	*shadow-directive*
H207	*executable-directive-extended*	**is**	*independent-directive*
		or	*realign-directive*
		or	*redistribute-directive*
		or	*on-directive*
		or	*resident-directive*
H208	*executable-construct-extended*	**is**	*action-stmt*
		or	*case-construct*
		or	*do-construct*
		or	*if-construct*
		or	*where-construct*
		or	*on-construct*
		or	*resident-construct*
		or	*task-region-construct*

A.3 Data Mapping

A.3.2 Syntax of Data Alignment and Distribution Directives

H301	*combined-directive*	**is**	*combined-attribute-list* `::` *combined-decl-list*
H302	*combined-attribute*	**is**	`ALIGN` *align-attribute-stuff*
		or	`DISTRIBUTE` *dist-attribute-stuff*
		or	`INHERIT`
		or	`TEMPLATE`
		or	`PROCESSORS`
		or	`DIMENSION (` *explicit-shape-spec-list* `)`
H303	*combined-decl*	**is**	*hpf-entity* [`(` *explicit-shape-spec-list* `)`]
		or	*object-name*
H304	*hpf-entity*	**is**	*processors-name*
		or	*template-name*

Constraint: The same kind of *combined-attribute* must not appear more than once in a given *combined-directive*.

Constraint: If the `DIMENSION` attribute appears in a *combined-directive*, any entity to which it applies must be declared with the HPF `TEMPLATE` or `PROCESSORS` type specifier.

A.3.3 The DISTRIBUTE Directive

H305	*distribute-directive*	**is**	`DISTRIBUTE` *distributee dist-directive-stuff*
H306	*dist-directive-stuff*	**is**	*dist-format-clause* [*dist-onto-clause*]
H307	*dist-attribute-stuff*	**is**	*dist-directive-stuff*
		or	*dist-onto-clause*
H308	*distributee*	**is**	*object-name*
		or	*template-name*
H309	*dist-format-clause*	**is**	(*dist-format-list*)
		or	`*` (*dist-format-list*)
		or	`*`
H310	*dist-format*	**is**	`BLOCK` [(*scalar-int-expr*)]
		or	`CYCLIC` [(*scalar-int-expr*)]
		or	`*`
H311	*dist-onto-clause*	**is**	`ONTO` *dist-target*
H312	*dist-target*	**is**	*processors-name*
		or	`*` *processors-name*
		or	`*`

Constraint: An *object-name* mentioned as a *distributee* must be a simple name and not a subobject designator or a *component-name*.

Constraint: An *object-name* mentioned as a *distributee* may not appear as an *alignee*.

Constraint: An *object-name* mentioned as a *distributee* may not have the `POINTER` attribute.

Constraint: An *object-name* mentioned as a *distributee* may not have the `TARGET` attribute.

Constraint: If the *distributee* is scalar, the *dist-format-list* (and its surrounding parentheses) must not appear. In this case, the statement form of the directive is allowed only if a *dist-format-clause* of "`*`" is present.

Constraint: If a *dist-format-list* is specified, its length must equal the rank of each *distributee* to which it applies.

Constraint: If both a *dist-format-list* and a *dist-target* appear, the number of elements of the *dist-format-list* that are not "`*`" must equal the rank of the specified processor arrangement.

Constraint: If a *dist-target* appears but not a *dist-format-list*, the rank of each *distributee* must equal the rank of the specified processor arrangement.

Constraint: If either the *dist-format-clause* or the *dist-target* in a `DISTRIBUTE` directive begins with "`*`" then every *distributee* must be a dummy argument.

Constraint: Any *scalar-int-expr* appearing in a *dist-format* of a `DISTRIBUTE` directive must be a *specification-expr*.

A.3.4 The ALIGN Directive

H313	*align-directive*	**is**	`ALIGN` *alignee align-directive-stuff*
H314	*align-directive-stuff*	**is**	(*align-source-list*) *align-with-clause*
H315	*align-attribute-stuff*	**is**	[(*align-source-list*)] *align-with-clause*
H316	*alignee*	**is**	*object-name*
H317	*align-source*	**is**	:
		or	*
		or	*align-dummy*
H318	*align-dummy*	**is**	*scalar-int-variable*

Constraint: An *object-name* mentioned as an *alignee* must be a simple name and not a subobject designator or a *component-name*.

Constraint: An *object-name* mentioned as an *alignee* may not appear as a *distributee*.

Constraint: An *object-name* mentioned as an *alignee* may not have the `POINTER` attribute.

Constraint: An *object-name* mentioned as an *alignee* may not have the `TARGET` attribute.

Constraint: If the *alignee* is scalar, the *align-source-list* (and its surrounding parentheses) must not appear. In this case the statement form of the directive is not allowed.

Constraint: If the *align-source-list* is present, its length must equal the rank of each *alignee* to which it applies.

Constraint: An *align-dummy* must be a named variable.

Constraint: An object may not have both the `INHERIT` attribute and the `ALIGN` attribute.

H319	*align-with-clause*	**is**	`WITH` *align-spec*
H320	*align-spec*	**is**	*align-target* [(*align-subscript-list*)]
		or	* *align-target* [(*align-subscript-list*)]
H321	*align-target*	**is**	*object-name*
		or	*template-name*
H322	*align-subscript*	**is**	*int-expr*
		or	*align-subscript-use*
		or	*subscript-triplet*
		or	*
H323	*align-subscript-use*	**is**	[[*int-level-two-expr*] *add-op*] *align-add-operand*
		or	*align-subscript-use add-op int-add-operand*
H324	*align-add-operand*	**is**	[*int-add-operand* *] *align-primary*
		or	*align-add-operand* * *int-mult-operand*
H325	*align-primary*	**is**	*align-dummy*
		or	(*align-subscript-use*)

H326 *int-add-operand* **is** *add-operand*

H327 *int-mult-operand* **is** *mult-operand*

H328 *int-level-two-expr* **is** *level*-2-*expr*

Constraint: An *object-name* mentioned as an *align-target* must be a simple name and not a subobject designator or a *component-name*.

Constraint: An *align-target* may not have the `OPTIONAL` attribute.

Constraint: If the *align-spec* in an `ALIGN` directive begins with "`*`" then every *alignee* must be a dummy argument.

Constraint: In an *align-directive* any *int-expr*, *int-level-two-expr*, *int-add-operand* or *int-mult-operand* must be a specification expression.

Constraint: Any *subscript* or *stride* in a *subscript-triplet* that is an *align-subscript* in an *align-directive* must be a specification expression.

Constraint: Each *align-dummy* may appear at most once in an *align-subscript-list*.

Constraint: An *align-subscript-use* expression may contain at most one occurrence of an *align-dummy*.

Constraint: A *scalar-int-variable* that is used as an *align-dummy* may not appear anywhere in the *align-spec* except where explicitly permitted to appear by virtue of the grammar shown above. Paraphrased, one may construct an *align-subscript-use* only by starting with an *align-dummy* and then doing additive and multiplicative things to it with integer specification expressions that contain no *align-dummy*.

Constraint: A *subscript* within an *align-subscript* may not contain occurrences of any *align-dummy*.

Constraint: An *int-add-operand*, *int-mult-operand*, or *int-level-two-expr* must be of type integer.

A.3.6 The PROCESSORS Directive

H329 *processors-directive* **is** `PROCESSORS` *processors-decl-list*

H330 *processors-decl* **is** *processors-name*
[(*explicit-shape-spec-list*)]

A.3.7 The TEMPLATE Directive

H331 *template-directive* **is** `TEMPLATE` *template-decl-list*

H332 *template-decl* **is** *template-name* [(*explicit-shape-spec-list*)]

A.3.8 Storage and Sequence Association

H333	*sequence-directive*	**is**	`SEQUENCE` [[`::`] *association-name-list*]
		or	`NO SEQUENCE` [[`::`] *association-name-list*]
H334	*association-name*	**is**	*object-name*
		or	`/` [*common-block-name*] `/`

Constraint: An object name or `COMMON` block name may appear at most once in a *sequence-directive* within any scoping unit.

Constraint: Only one sequence directive with no *association-name-list* is permitted in the same scoping unit.

A.4 Data Mapping in Subprogram Interfaces

A.4.4 Alignment

H401	*inherit-directive*	**is**	`INHERIT` *inheritee-list*
H402	*inheritee*	**is**	*object-name*

Constraint: An *inheritee* must be a dummy argument.

Constraint: An *inheritee* must not be an *alignee*.

Constraint: An *inheritee* must not be a *distributee*.

A.5 INDEPENDENT and Related Directives

A.5.1 The INDEPENDENT Directive

H501	*independent-directive*	**is**	`INDEPENDENT` [, *new-clause*] [, *reduction-clause*]
H502	*new-clause*	**is**	`NEW (` *variable-name-list* `)`
H503	*reduction-clause*	**is**	`REDUCTION (` *reduction-variable-list* `)`
H504	*reduction-variable*	**is**	*array-variable-name*
		or	*scalar-variable-name*
		or	*structure-component*

Constraint: The first non-comment line following an *independent-directive* must be a *do-stmt*, *forall-stmt*, or a *forall-construct*.

Constraint: If the first non-comment line following an *independent-directive* is a *do-stmt*, then that statement must contain a *loop-control* option containing a *do-variable*.

Constraint: If either the `NEW` clause or the `REDUCTION` clause is present, then the first non-comment line following the directive must be a *do-stmt*.

Constraint: A *variable* named in the `NEW` or the `REDUCTION` clause and any component or element thereof must not:

- Be a dummy argument;
- Have the `SAVE` or `TARGET` attribute;
- Occur in a `COMMON` block;
- Be storage associated with another object as a result of appearing in an `EQUIVALENCE` statement;
- Be use associated;
- Be host associated; or
- Be accessed in another scoping unit via host association.

Constraint: A variable that occurs as a *reduction-variable* may not appear in a *new-clause* in the same *independent-directive*, nor may it appear in either a *new-clause* or a *reduction-clause* in the range (i.e., the lexical body) of the following *do-stmt*, *forall-stmt*, or *forall-construct* to which the *independent-directive* applies.

Constraint: A *structure-component* in a *reduction-variable* may not contain a *subscript-section-list*.

Constraint: A variable that occurs as a *reduction-var* must be of intrinsic type. It may not be of type `CHARACTER`.

H505	*reduction-stmt*	**is**	*variable* = *variable mult-op mult-operand*
		or	*variable* = *add-operand* * *variable*
		or	*variable* = *variable add-op add-operand*
		or	*variable* = *level*-2-*expr* + *variable*
		or	*variable* = *variable and-op and-operand*
		or	*variable* = *and-operand and-op variable*
		or	*variable* = *variable or-op or-operand*
		or	*variable* = *or-operand or-op variable*
		or	*variable* = *variable equiv-op equiv-operand*
		or	*variable* = *equiv-operand equiv-op variable*
		or	*variable* = *reduction-function* (*variable* , *expr*)
		or	*variable* = *reduction-function* (*expr* , *variable*)
H506	*reduction-function*	**is**	`MAX`
		or	`MIN`
		or	`IAND`
		or	`IOR`
		or	`IEOR`

Constraint: The two occurances of *variable* in a *reduction-stmt* must be textually identical.

A.6 Extrinsic Program Units

A.6.2 Declaration of Extrinsic Program Units

H601	*function-stmt*	**is**	[*prefix*] `FUNCTION` *function-name* ([*dummy-arg-name-list*]) [`RESULT` (*result-name*)]

H602 *subroutine-stmt* **is** [*prefix*] `SUBROUTINE` *subroutine-name*
  [([*dummy-arg-list*])]

H603 *prefix* **is** *prefix-spec* [*prefix-spec*] ...

H604 *prefix-spec* **is** *type-spec*
  or `RECURSIVE`
  or `PURE`
  or `ELEMENTAL`
  or *extrinsic-prefix*

Constraint: Within any HPF *external-subprogram*, every *internal-subprogram* must be of the same extrinsic kind as its host and any *internal-subprogram* whose extrinsic kind is not given explicitly is assumed to be of that extrinsic kind.

H605 *program-stmt* **is** [*extrinsic-prefix*] `PROGRAM` *program-name*

H606 *module-stmt* **is** [*extrinsic-prefix*] `MODULE` *module-name*

H607 *block-data-stmt* **is** [*extrinsic-prefix*] `BLOCK DATA`
  [*block-data-name*]

Constraint: Every *module-subprogram* of any HPF *module* must be of the same extrinsic kind as its host, and any *module-subprogram* whose extrinsic kind is not given explicitly is assumed to be of that extrinsic kind.

Constraint: Every *internal-subprogram* of any HPF *main-program* or *module-subprogram* must be of the same extrinsic kind as its host, and any *internal-subprogram* whose extrinsic kind is not given explicitly is assumed to be of that extrinsic kind.

H608 *extrinsic-prefix* **is** `EXTRINSIC (` *extrinsic-spec* `)`

H609 *extrinsic-spec* **is** *extrinsic-spec-arg-list*
  or *extrinsic-kind-keyword*

H610 *extrinsic-spec-arg* **is** *language*
  or *model*
  or *external-name*

H611 *language* **is** [`LANGUAGE =`]
  scalar-char-initialization-expr

H612 *model* **is** [`MODEL =`]
  scalar-char-initialization-expr

H613 *external-name* **is** [`EXTERNAL_NAME =`]
  scalar-char-initialization-expr

Constraint: In an *extrinsic-spec-arg-list*, at least one of *language*, *model*, or *external-name* must be specified and none may be specified more than once.

Constraint: If *language* is specified without `LANGUAGE=`, *language* must be the first item in the *extrinsic-spec-arg-list*. If *model* is specified without `MODEL=`, *language* without `LANGUAGE=` must be the first item and *model* must be the second item in the *extrinsic-spec-arg-list*. If *external-name* is specified without `EXTERNAL_NAME=`, *language* without `LANGUAGE=` must be the first item and *model* without `MODEL=` must be the second item in the *extrinsic-spec-arg-list*.

Constraint: The forms with `LANGUAGE=`, `MODEL=`, and `EXTERNAL_NAME=` may appear in any order except as prohibited above.

Note that these rules for *extrinsic-spec-arg-list* are as if `EXTRINSIC` were a procedure with an explicit interface with a *dummy-arg-list* of `LANGUAGE, MODEL, EXTERNAL_NAME`, each of which were `OPTIONAL`.

Constraint: In *language*, values of *scalar-char-initialization-expr* may be:

- `'HPF'`, referring to the HPF language; if a *model* is not explicitly specified, the *model* is implied to be `'GLOBAL'`;
- `'FORTRAN'`, referring to the ANSI/ISO standard Fortran language; if a *model* is not explicitly specified, the *model* is implied to be `'SERIAL'`;
- `'F77'`, referring to the former ANSI/ISO standard FORTRAN 77 language; if a *model* is not explicitly specified, the *model* is implied to be `'SERIAL'`;
- `'C'`, referring to the ANSI standard C programming language; if a *model* is not explicitly specified, the *model* is implied to be `'SERIAL'`; or
- an implementation-dependent value with an implementation-dependent implied *model*.

Note that, for most implementations, `'C'` will only be allowed for *function-stmt*s and *subroutine-stmt*s occurring in an *interface-body*.

Constraint: If language is not specified it is the same as that of the host scoping unit.

Constraint: In *model*, values of *scalar-char-initialization-expr* may be:

- `'GLOBAL'`, referring to the global model,
- `'LOCAL'`, referring to the local model,
- `'SERIAL'`, referring to the serial model, or
- an implementation-dependent value.

Constraint: If *model* is not specified or implied by the specification of a language, it is the same as that of the host scoping unit.

Constraint: All *language*s and *model*s whose names begin with the three letters `HPF` are reserved for present or future definition by this specification and its successors.

Constraint: In *external-name*, the value of *scalar-char-initialization-expr* is a character string whose use is determined by the extrinsic kind. For example, an extrinsic kind may use the *external-name* to specify the name by which the procedure would be known if it were referenced by a C procedure. In such an implementation, a user would expect the compiler to perform any transformations of that name that the C compiler would perform. If *external-name* is not specified, its value is implementation-dependent.

H614 *extrinsic-kind-keyword* **is** `HPF`
or `HPF_LOCAL`
or `HPF_SERIAL`

Constraint: `EXTRINSIC(HPF)` is equivalent to `EXTRINSIC('HPF','GLOBAL')`. In the absence of an *extrinsic-prefix* an HPF compiler interprets a compilation unit as if it were of extrinsic kind `HPF`. Thus, for an HPF compiler, specifying `EXTRINSIC(HPF)` or `EXTRINSIC('HPF','GLOBAL')` is redundant. Such explicit specification may, however, be required for use with a compiler that supports multiple extrinsic kinds.

Constraint: `EXTRINSIC(HPF_LOCAL)` is equivalent to `EXTRINSIC('HPF','LOCAL')`. A *main-program* whose extrinsic kind is `HPF_LOCAL` behaves as if it were a subroutine of extrinsic kind `HPF_LOCAL` that is called with no arguments from a main program of extrinsic kind `HPF` whose executable part consists solely of that call.

Constraint: `EXTRINSIC(HPF_SERIAL)` is equivalent to `EXTRINSIC('HPF','SERIAL')`. A *main-program* whose extrinsic kind is `HPF_SERIAL` behaves as if it were a subroutine of extrinsic kind `HPF_SERIAL` that is called with no arguments from a main program of extrinsic kind `HPF` whose executable part consists solely of that call.

Constraint: All *extrinsic-kind-keyword*s whose names begin with the three letters `HPF` are reserved for present or future definition by this specification and its successors.

A.8 Approved Extensions for Data Mapping

A.8.2 Syntax of Attributed Forms of Extended Data Mapping Directives

H801 *combined-attribute-extended* **is** `ALIGN` *align-attribute-stuff*
or `DISTRIBUTE` *dist-attribute-stuff*
or `INHERIT`
or `TEMPLATE`
or `PROCESSORS`
or `DIMENSION (` *explicit-shape-spec-list* `)`
or `DYNAMIC`
or `RANGE` *range-attr-stuff*
or `SHADOW` *shadow-attr-stuff*
or `SUBSET`

Constraint: The `SUBSET` attribute may be applied only to a processors arrangement.

A.8.3 The REDISTRIBUTE Directive

H802 *redistribute-directive* **is** `REDISTRIBUTE` *distributee dist-directive-stuff*
or `REDISTRIBUTE` *dist-attribute-stuff* `::` *distributee-list*

Constraint: A *distributee* that appears in a `REDISTRIBUTE` directive must have the `DYNAMIC` attribute (see Section 8.5).

Constraint: A *distributee* in a `REDISTRIBUTE` directive may not appear as an *alignee* in an `ALIGN` or `REALIGN` directive.

Constraint: Neither the *dist-format-clause* nor the *dist-target* in a `REDISTRIBUTE` directive may begin with "*".

A.8.4 The REALIGN Directive

H803	*realign-directive*	**is**	`REALIGN` *alignee align-directive-stuff*
		or	`REALIGN` *align-attribute-stuff* `::` *alignee-list*

Constraint: Any *alignee* that appears in a `REALIGN` directive must have the `DYNAMIC` attribute (see Section 8.5).

Constraint: If the *align-target* specified in the *align-with-clause* has the `DYNAMIC` attribute, then each *alignee* must also have the `DYNAMIC` attribute.

Constraint: An *alignee* in a `REALIGN` directive may not appear as a *distributee* in a `DISTRIBUTE` or `REDISTRIBUTE` directive.

A.8.5 The DYNAMIC Directive

H804	*dynamic-directive*	**is**	`DYNAMIC` *alignee-or-distributee-list*
H805	*alignee-or-distributee*	**is**	*alignee*
		or	*distributee*

Constraint: An object in `COMMON` may not be declared `DYNAMIC` and may not be aligned to an object (or template) that is `DYNAMIC`. (To get this kind of effect, modules must be used instead of `COMMON` blocks.)

Constraint: A component of a derived type may have the `DYNAMIC` attribute only if it also has the `POINTER` attribute. (See Section 8.9 for further discussion.)

Constraint: An object with the `SAVE` attribute may not be declared `DYNAMIC` and may not be aligned to an object (or template) that is `DYNAMIC`.

A.8.7 Mapping to Processor Subsets

H806	*extended-dist-target*	**is**	*processors-name* [(*section-subscript-list*)]
		or	* *processors-name* [(*section-subscript-list*)]
		or	*

Constraint: The *section-subscripts* in the *section-subscript-list* may not be *vector-subscripts* and are restricted to be either *subscripts* or *subscript-triplets*.

Constraint: In the *section-subscript-list*, the number of *section-subscripts* must equal the rank of the *processor-name*.

Constraint: Within a `DISTRIBUTE` directive, each *section-subscript* must be a *specification-expr.*

Constraint: Within a `DISTRIBUTE` or a `REDISTRIBUTE` directive, if both a *dist-format-list* and a *dist-target* appear, the number of elements of the *dist-format-list* that are not "`*`" must equal the number of *subscript-triplet*s in the named processor arrangement.

Constraint: Within a `DISTRIBUTE` or a `REDISTRIBUTE` directive, if a *dist-target* appears but not a *dist-format-list*, the rank of each *distributee* must equal the number of *subscript-triplet*s in the named processor arrangement.

Constraint: If either the *dist-format-clause* or the *dist-target* in a `DISTRIBUTE` directive begins with "`*`" then every *distributee* must be a dummy argument, *except if the distributee has the `POINTER` attribute.*

Constraint: If the *align-spec* in an `ALIGN` directive begins with "`*`" then every *alignee* must be a dummy argument, *except if the* alignee *has the `POINTER` attribute.*

Constraint: An *inheritee* must be a dummy argument, *except if the* alignee *has the `POINTER` attribute.*

A.8.9 Mapping of Derived Type Components

H807 *distributee-extended* **is** *object-name*
or *template-name*
or *component-name*
or *structure-component*

Constraint: A component of a derived type may be explicitly distributed only if the type of the component is not an explicitly mapped type.

Constraint: An object of a derived type may be explicitly distributed only if the derived type is not an explicitly mapped type.

Constraint: A *distributee* in a `DISTRIBUTE` directive may not be a *structure-component.*

Constraint: A *distributee* in a `DISTRIBUTE` directive which occurs in a *derived-type-def* must be the *component-name* of a component of the derived type.

Constraint: A *component-name* may occur as a *distributee* in a `DISTRIBUTE` directive occuring within the derived type definition only.

Constraint: A *distributee* that is a *structure-component* may occur only in a `REDISTRIBUTE` directive and every *part-ref* except the rightmost must be scalar (rank zero). The rightmost *part-name* in the *structure-component* must have the `DYNAMIC` attribute.

H808 *alignee-extended* **is** *object-name*
or *component-name*
or *structure-component*

Constraint: A component of a derived type may be explicitly aligned only if the type of the component is not an explicitly mapped type.

Constraint: An object of a derived type may be explicitly aligned only if the derived type is not an explicitly mapped type.

Constraint: An *alignee* in an `ALIGN` directive may not be a *structure-component*.

Constraint: An *alignee* in an `ALIGN` directive that occurs in a *derived-type-def* must be the *component-name* of a component of the derived type.

Constraint: A *component-name* may occur as an *alignee* only in an `ALIGN` directive occuring within the derived type definition.

Constraint: An *alignee* that is a *structure-component* may occur only in a `REALIGN` directive and every *part-ref* except the rightmost must be scalar (rank zero). The rightmost *part-name* in the *structure-component* must have the `DYNAMIC` attribute.

H809	*align-target-extended*	**is**	*object-name*
		or	*template-name*
		or	*component-name*
		or	*structure-component*

Constraint: A *component-name* may appear as an align target only in an `ALIGN` directive occuring within the derived type definition that defines that component.

Constraint: In an *align-target* that is a *structure-component*, every *part-ref* except the rightmost must be scalar (rank zero).

A.8.10 New Distribution Formats

H810	*extended-dist-format*	**is**	`BLOCK [ (` *int-expr* `) ]`
		or	`CYCLIC [ (` *int-expr* `) ]`
		or	`GEN_BLOCK (` *int-array* `)`
		or	`INDIRECT (` *int-array* `)`
		or	`*`

Constraint: An *int-array* appearing in a *extended-dist-format* of a `DISTRIBUTE` directive or `REDISTRIBUTE` directive must be an integer array of rank 1.

Constraint: An *int-array* appearing in a *extended-dist-format* of a `DISTRIBUTE` directive must be a *restricted-expr*.

Constraint: The size of any *int-array* appearing with a `GEN_BLOCK` distribution must be equal to the extent of the corresponding dimension of the target processor arrangement.

Constraint: The size of any *int-array* appearing with an `INDIRECT` distribution must be equal to the extent of the corresponding dimension of the *distributee* to which the distribution is to be applied.

A.8.11 The RANGE Directive

H811	*range-directive*	**is**	RANGE *ranger* *range-attr-stuff*
H812	*ranger*	**is**	*object-name*
		or	*template-name*
H813	*range-attr-stuff*	**is**	*range-distribution-list*
H814	*range-distribution*	**is**	(*range-attr-list*)
H815	*range-attr*	**is**	*range-dist-format*
		or	ALL
H816	*range-dist-format*	**is**	BLOCK [( )]
		or	CYCLIC [( )]
		or	GEN_BLOCK
		or	INDIRECT
		or	*

Constraint: At least one of the following must be true:

- The *ranger* has the DYNAMIC attribute.
- The *ranger* has the INHERIT attribute.
- The *ranger* is specified with a *dist-format-clause* of * in a DISTRIBUTE or combined directive.

Constraint: The length of each *range-attr-list* must be equal to the rank of the *ranger*.

Constraint: The *ranger* must not appear as an alignee in an ALIGN or REALIGN directive.

A.8.12 The SHADOW Directive

H817	*shadow-directive*	**is**	SHADOW *shadow-target* *shadow-attr-stuff*
H818	*shadow-target*	**is**	*object-name*
		or	*component-name*
H819	*shadow-attr-stuff*	**is**	(*shadow-spec-list*)
H820	*shadow-spec*	**is**	*width*
		or	*low-width* : *high-width*
H821	*width*	**is**	*int-expr*
H822	*low-width*	**is**	*int-expr*
H823	*high-width*	**is**	*int-expr*

Constraint: The *int-expr* representing a *width*, *low-width*, or *high-width* must be a constant *specification-expr* with value greater than or equal to 0.

A.9 Approved Extensions for Data and Task Parallelism

A.9.1 Active Processor Sets

H901	*subset-directive*	**is**	SUBSET *processors-name*

A.9.2 The ON Directive

H902	*on-directive*	**is**	ON *on-stuff*
H903	*on-stuff*	**is**	*home* [, *resident-clause*] [, *new-clause*]
H904	*on-construct*	**is**	*directive-origin block-on-directive* *block* *directive-origin end-on-directive*
H905	*block-on-directive*	**is**	ON *on-stuff* BEGIN
H906	*end-on-directive*	**is**	END ON
H907	*home*	**is**	HOME (*variable*)
		or	HOME (*template-elmt*)
		or	(*processors-elmt*)
H908	*template-elmt*	**is**	*template-name* [(*section-subscript-list*)]
H909	*processors-elmt*	**is**	*processors-name* [(*section-subscript-list*)]

A.9.3 The RESIDENT Clause, Directive, and Construct

H910	*resident-clause*	**is**	RESIDENT *resident-stuff*
H911	*resident-stuff*	**is**	[(*res-object-list*)]
H912	*resident-directive*	**is**	RESIDENT *resident-stuff*
H913	*resident-construct*	**is**	*directive-origin block-resident-directive* *block* *directive-origin end-resident-directive*
H914	*block-resident-directive*	**is**	RESIDENT *resident-stuff* BEGIN
H915	*end-resident-directive*	**is**	END RESIDENT
H916	*res-object*	**is**	*object*

A.9.4 The TASK_REGION Construct

H917	*task-region-construct*	**is**	*directive-origin block-task-region-directive* *block* *directive-origin end-task-region-directive*
H918	*block-task-region-directive*	**is**	TASK_REGION
H919	*end-task-region-directive*	**is**	END TASK_REGION

A.10 Approved Extension for Asynchronous I/O

	or	`ASYNCHRONOUS`
	or	`ID =` *scalar-default-int-variable*
	or	`ASYNCHRONOUS`

Constraint: If either an `ASYNCHRONOUS` or an `ID=` specifier is present, then both shall be present.

Constraint: If an `ASYNCHRONOUS` specifier is present, the `REC=` specifier shall appear, a *format* shall not appear, and a *namelist-group-name* shall not appear.

Constraint: If an `ASYNCHRONOUS` specifier is present, then no function reference may appear in an expression anywhere in the data transfer statement.

	or	`ID =` *scalar-default-int-variable*
	or	`PENDING =` *scalar-default-logical-variable*

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

Constraint: If either an `ID=` specifier or a `PENDING=` specifier is present, then both shall be present.

A.10.1 The WAIT Statement

H1001 *wait-stmt*	**is**	`WAIT (` *wait-spec-list* `)`
H1002 *wait-spec*	**is**	`UNIT =` *io-unit*
	or	`ID =` *scalar-default-int-expr*
	or	`ERR =` *label*
	or	`IOSTAT =` *label*

Constraint: A *wait-spec-list* shall contain exactly one `UNIT=` specifier, exactly one `ID=` specifier, and at most one of each of the other specifiers.

A.11 Approved Extensions for HPF Extrinsics

A.11.2 Extrinsic Language Bindings

H1101 *type-declaration-stmt-extended*	**is**	*type-spec* [[, *attr-spec-extended*] ... ::] *entity-decl-list*

H1102 *attr-spec-extended*	**is**	`PARAMETER`
	or	*access-spec*
	or	`ALLOCATABLE`
	or	`DIMENSION (` *array-spec* `)`
	or	`EXTERNAL`
	or	`INTENT (` *intent-spec* `)`
	or	`INTRINSIC`
	or	`OPTIONAL`
	or	`POINTER`
	or	`SAVE`
	or	`TARGET`
	or	`MAP_TO (` *map-to-spec* `)`
	or	`LAYOUT (` *layout-spec* `)`
	or	`PASS_BY (` *pass-by-spec* `)`
H1103 *map-to-spec*	**is**	*scalar-char-initialization-expr*
H1104 *layout-spec*	**is**	*scalar-char-initialization-expr*
H1105 *pass-by-spec*	**is**	*scalar-char-initialization-expr*

Constraint: The same *attr-spec-extended* shall not appear more than once in a given *type-declaration-stmt*.

Constraint: An entity shall not be explicitly given any attribute more than once in a scoping unit.

Constraint: The attributes `MAP_TO`, `LAYOUT`, and `PASS_BY` may be specified only for dummy arguments within a scoping unit of an extrinsic type for which these attributes have been explicitly defined.

Annex B

Syntax Cross-reference

This Appendix cross-references symbols used in the formal syntax rules. Rule identifiers beginning with "H" refer to syntax rules of this High Performance Fortran Language Specification; the full rule may be found in Appendix A. Rule identifiers beginning with "R" refer to syntax rules of the Fortran Language Standard ("Fortran 95").

B.1 Nonterminal Symbols That Are Defined

Symbol	**Defined**	**Referenced**
action-stmt	R216	H208
add-op	R710	H323 H505
add-operand	R706	H326 H505
align-add-operand	H324	H323 H324
align-attribute-stuff	H315	H302 H801 H803
align-directive	H313	H204 H206
align-directive-stuff	H314	H313 H803
align-dummy	H318	H317 H325
align-primary	H325	H324
align-source	H317	H314 H315
align-spec	H320	H319
align-subscript	H322	H320
align-subscript-use	H323	H322 H323 H325
align-target	H321	H320
align-target-extended	H809	
align-with-clause	H319	H314 H315
alignee	H316	H313 H803 H805
alignee-extended	H808	
alignee-or-distributee	H805	H804
allocate-object	R625	
allocate-stmt	R622	
and-op	R720	H505
and-operand	R715	H505
array-constructor	R432	
array-spec	R513	H1102
assignment-stmt	R735	

association-name	H334	H333				
attr-spec	R503					
attr-spec-extended	H1102	H1101				
block	R801	H904	H913	H917		
block-data-stmt	H607					
block-on-directive	H905	H904				
block-resident-directive	H914	H913				
block-task-region-directive	H918	H917				
call-stmt	R1211					
case-construct	R808	H208				
combined-attribute	H302	H301				
combined-attribute-extended	H801					
combined-decl	H303	H301				
combined-directive	H301	H204	H206			
data-stmt	R532					
deallocate-stmt	R631					
directive-origin	H202	H201	H904	H913	H917	
dist-attribute-stuff	H307	H302	H801	H802		
dist-directive-stuff	H306	H305	H307	H802		
dist-format	H310	H309				
dist-format-clause	H309	H306				
dist-onto-clause	H311	H306	H307			
dist-target	H312	H311				
distribute-directive	H305	H204	H206			
distributee	H308	H305	H802	H805		
distributee-extended	H807					
do-construct	R816	H208				
dummy-arg	R1223	H602				
dynamic-directive	H804	H206				
end-function-stmt	R1220					
end-on-directive	H906	H904				
end-resident-directive	H915	H913				
end-subroutine-stmt	R1224					
end-task-region-directive	H919	H917				
entity-decl	R504	H1101				
equiv-op	R722	H505				
equiv-operand	R717	H505				
executable-construct	R215					
executable-construct-extended	H208					
executable-directive	H205	H203				
executable-directive-extended	H207					
execution-part	R208					
explicit-shape-spec	R514	H302	H303	H330	H332	H801
expr	R723	H505				
extended-dist-format	H810					
extended-dist-target	H806					
external-name	H613	H610				
extrinsic-kind-keyword	H614	H609				

extrinsic-prefix	H608	H604 H605 H606 H607
extrinsic-spec	H609	H608
extrinsic-spec-arg	H610	H609
function-reference	R1210	
function-stmt	H601	
function-subprogram	R1216	
high-width	H823	H820
home	H907	H903
hpf-directive	H203	H201
hpf-directive-line	H201	
hpf-entity	H304	H303
if-construct	R802	H208
independent-directive	H501	H205 H207
inherit-directive	H401	H204 H206
inheritee	H402	H401
input-item	R914	
int-add-operand	H326	H323 H324
int-expr	R728	H310 H322 H810 H821 H822 H823
int-level-two-expr	H328	H323
int-mult-operand	H327	H324
int-variable	R607	H318
interface-body	R1205	
internal-subprogram-part	R210	
io-unit	R901	H1002
kind-selector	R506	
label	R313	H1002
language	H611	H610
layout-spec	H1104	H1102
level-2-expr	R707	H328 H505
low-width	H822	H820
map-to-spec	H1103	H1102
mask-expr	R743	
model	H612	H610
module-stmt	H606	
mult-op	R709	H505
mult-operand	R705	H327 H505
namelist-stmt	R544	
new-clause	H502	H501 H903
nullify-stmt	R629	
on-construct	H904	H208
on-directive	H902	H207
on-stuff	H903	H902 H905
or-op	R721	H505
or-operand	R716	H505
output-item	R915	
pass-by-spec	H1105	H1102
pointer-assignment-stmt	R736	
pointer-object	R630	

prefix	H603	H601	H602		
prefix-spec	H604	H603			
processors-decl	H330	H329			
processors-directive	H329	H204	H206		
processors-elmt	H909	H907			
program-stmt	H605				
range-attr	H815	H814			
range-attr-stuff	H813	H801	H811		
range-directive	H811	H206			
range-dist-format	H816	H815			
range-distribution	H814	H813			
ranger	H812	H811			
read-stmt	R909				
realign-directive	H803	H207			
redistribute-directive	H802	H207			
reduction-clause	H503	H501			
reduction-function	H506	H505			
reduction-stmt	H505				
reduction-variable	H504	H503			
res-object	H916	H911			
resident-clause	H910	H903			
resident-construct	H913	H208			
resident-directive	H912	H207			
resident-stuff	H911	H910	H912	H914	
section-subscript	R618	H806	H908	H909	
sequence-directive	H333	H204	H206		
shadow-attr-stuff	H819	H801	H817		
shadow-directive	H817	H206			
shadow-spec	H820	H819			
shadow-target	H818	H817			
specification-directive	H204	H203			
specification-directive-extended	H206				
specification-expr	R734				
specification-part	R204				
stat-variable	R623				
stop-stmt	R840				
stride	R620				
structure-component	R614	H504	H807	H808	H809
subroutine-stmt	H602				
subscript	R617				
subscript-triplet	R619	H322			
subset-directive	H901	H206			
target	R737				
task-region-construct	H917	H208			
template-decl	H332	H331			
template-directive	H331	H204	H206		
template-elmt	H908	H907			
type-declaration-stmt	R501				

type-declaration-stmt-extended	H1101	
type-spec	R502	H604 H1101
variable	R601	H505 H907
wait-spec	H1002	H1001
wait-stmt	H1001	
where-construct	R739	H208
where-stmt	R738	
width	H821	H820
write-stmt	R910	

B.2 Nonterminal Symbols That Are Not Defined

Symbol	**Referenced**
access-spec	H1102
array-variable-name	H504
block-data-name	H607
char-initialization-expr	H611 H612 H613 H1103 H1104 H1105
common-block-name	H334
component-name	H807 H808 H809 H818
default-int-expr	H1002
dummy-arg-name	H601
function-name	H601
int-array	H810
intent-spec	H1102
module-name	H606
object	H916
object-name	H303 H308 H316 H321 H334 H402
	H807 H808 H809 H812 H818
processors-name	H304 H312 H330 H806 H901 H909
program-name	H605
result-name	H601
subroutine-name	H602
template-name	H304 H308 H321 H332 H807 H809
	H812 H908
variable-name	H502 H504

B.3 Terminal Symbols

Symbol	**Referenced**
`!HPF$`	H202
(	H302 H303 H309 H310 H314 H315
	H320 H325 H330 H332 H502 H503
	H505 H601 H602 H608 H801 H806
	H810 H814 H816 H819 H907 H908
	H909 H911 H1001 H1102
)	H302 H303 H309 H310 H314 H315

	H320 H325 H330 H332 H502 H503 H505 H601 H602 H608 H801 H806 H810 H814 H816 H819 H907 H908 H909 H911 H1001 H1102
`*`	H309 H310 H312 H317 H320 H322 H324 H505 H806 H810 H816
`*HPF$`	H202
`+`	H505
`,`	H501 H505 H903 H1101
`/`	H334
`:`	H317 H820
`::`	H301 H333 H802 H803 H1101
`=`	H505 H611 H612 H613 H1002
`ALIGN`	H302 H313 H801
`ALL`	H815
`ALLOCATABLE`	H1102
`BEGIN`	H905 H914
`BLOCK`	H310 H607 H810 H816
`CHPF$`	H202
`CYCLIC`	H310 H810 H816
`DATA`	H607
`DIMENSION`	H302 H801 H1102
`DISTRIBUTE`	H302 H305 H801
`DYNAMIC`	H801 H804
`ELEMENTAL`	H604
`END`	H906 H915 H919
`ERR`	H1002
`EXTERNAL`	H1102
`EXTERNAL_NAME`	H613
`EXTRINSIC`	H608
`FUNCTION`	H601
`GEN_BLOCK`	H810 H816
`HOME`	H907
`HPF`	H614
`HPF_LOCAL`	H614
`HPF_SERIAL`	H614
`IAND`	H506
`ID`	H1002
`IEOR`	H506
`INDEPENDENT`	H501
`INDIRECT`	H810 H816
`INHERIT`	H302 H401 H801
`INTENT`	H1102
`INTRINSIC`	H1102
`IOR`	H506
`IOSTAT`	H1002
`LANGUAGE`	H611
`LAYOUT`	H1102

MAP_TO	H1102
MAX	H506
MIN	H506
MODEL	H612
MODULE	H606
NEW	H502
NO	H333
ON	H902 H905 H906
ONTO	H311
OPTIONAL	H1102
PARAMETER	H1102
PASS_BY	H1102
POINTER	H1102
PROCESSORS	H302 H329 H801
PROGRAM	H605
PURE	H604
RANGE	H801 H811
REALIGN	H803
RECURSIVE	H604
REDISTRIBUTE	H802
REDUCTION	H503
RESIDENT	H910 H912 H914 H915
RESULT	H601
SAVE	H1102
SEQUENCE	H333
SHADOW	H801 H817
SUBROUTINE	H602
SUBSET	H801 H901
TARGET	H1102
TASK_REGION	H918 H919
TEMPLATE	H302 H331 H801
UNIT	H1002
WAIT	H1001
WITH	H319

Annex C

HPF 1.1 Subset

As part of the definition of the previous version of the High Performance Fortran language, HPF 1.1, a subset language was formally defined, based on the Fortran 77 language. The goal was to permit more rapid implementations of a useful subset of HPF that did not require full implementation of the new ANSI/ISO standard Fortran ("Fortran 90").

No subset language is defined as part of the current version, HPF 2.0. This Annex is included in the HPF 2.0 language document as a convenient summary of the HPF 1.1 Subset, which has served as a minimum requirement for HPF implementations.

C.1 Fortran 90 Features in the HPF 1.1 Subset

The features of the HPF 1.1 subset languages are listed below. For reference, the section numbers from the Fortran 90 standard are given along with the related syntax rule numbers:

- All FORTRAN 77 standard conforming features, except for storage and sequence association.
- The Fortran 90 definitions of MIL-STD-1753 features:
  - `DO WHILE` statement (8.1.4.1.1 / R821)
  - `END DO` statement (8.1.4.1.1 / R825)
  - `IMPLICIT NONE` statement (5.3 / R540)
  - `INCLUDE` line (3.4)
  - scalar bit manipulation intrinsic procedures: `IOR`, `IAND`, `NOT`, `IEOR`, `ISHFT`, `ISHFTC`, `BTEST`, `IBSET`, `IBCLR`, `IBITS`, `MVBITS` (13.13)
  - binary, octal and hexadecimal constants for use in `DATA` statements (4.3.1.1 / R407 and 5.2.9 / R533)
- Arithmetic and logical array features:
  - array sections (6.2.2.3 / R618–621)
    - subscript triplet notation (6.2.2.3.1)
    - vector-valued subscripts (6.2.2.3.2)
  - array constructors limited to one level of implied `DO` (4.5 / R431)

- arithmetic and logical operations on whole arrays and array sections (2.4.3, 2.4.5, and 7.1)
- array assignment (2.4.5, 7.5, 7.5.1.4, and 7.5.1.5)
- masked array assignment (7.5.3)
  * `WHERE` statement (7.5.3 / R738)
  * block `WHERE . . . ELSEWHERE` construct (7.5.3 / R739)
- array-valued external functions (12.5.2.2)
- automatic arrays (5.1.2.4.1)
- `ALLOCATABLE` arrays and the `ALLOCATE` and `DEALLOCATE` statements (5.1.2.4.3, 6.3.1 / R622, and 6.3.3 / R631)
- assumed-shape arrays (5.1.2.4.2 / R516)

• Intrinsic procedures:

The list of intrinsic functions and subroutines below is a combination of (a) routines which are entirely new to Fortran and (b) routines that have always been part of Fortran, but have been extended here to new argument and result types. The new or extended definitions of these routines are part of the subset. If a FORTRAN 77 routine is not included in this list, then only the original FORTRAN 77 definition is part of the subset.

For all of the intrinsics that have an optional argument `DIM`, only actual argument expressions for `DIM` that are initialization expressions are part of the subset. The intrinsics with this constraint are marked with †in the list below.

- the argument presence inquiry function: `PRESENT` (13.10.1)
- all the numeric elemental functions: `ABS`, `AIMAG`, `AINT`, `ANINT`, `CEILING`, `CMPLX`, `CONJG`, `DBLE`, `DIM`, `DPROD`, `FLOOR`, `INT`, `MAX`, `MIN`, `MOD`, `MODULO`, `NINT`, `REAL`, `SIGN` (13.10.2)
- all mathematical elemental functions: `ACOS`, `ASIN`, `ATAN`, `ATAN2`, `COS`, `COSH`, `EXP`, `LOG`, `LOG10`, `SIN`, `SINH`, `SQRT`, `TAN`, `TANH` (13.10.3)
- all the bit manipulation elemental functions : `BTEST`, `IAND`, `IBCLR`, `IBITS`, `IBSET`, `IEOR`, `IOR`, `ISHFT`, `ISHFTC`, `NOT` (13.10.10)
- all the vector and matrix multiply functions: `DOT_PRODUCT`, `MATMUL` (13.10.13)
- all the array reduction functions: `ALL`†, `ANY`†, `COUNT`†, `MAXVAL`†, `MINVAL`†, `PRODUCT`†, `SUM`†(13.10.14)
- all the array inquiry functions: `ALLOCATED`, `LBOUND`†, `SHAPE`, `SIZE`†, `UBOUND`†(13.10.15)
- all the array construction functions: `MERGE`, `PACK`, `SPREAD`†, `UNPACK` (13.10.16)
- the array reshape function: `RESHAPE` (13.10.17)
- all the array manipulation functions: `CSHIFT`†, `EOSHIFT`†, `TRANSPOSE` (13.10.18)
- all array location functions: `MAXLOC`†, `MINLOC`†(13.10.19)
- all intrinsic subroutines: `DATE_AND_TIME`, `MVBITS`, `RANDOM_NUMBER`, `RANDOM_SEED`, `SYSTEM_CLOCK` (3.11)

- Declarations:
  - Type declaration statements, with all forms of *type-spec* except *kind-selector* and `TYPE`(type-name), and all forms of *attr-spec* except *access-spec*, `TARGET`, and `POINTER`. (5.1 / R501-503, R510)
  - attribute specification statements: `ALLOCATABLE`, `INTENT`, `OPTIONAL`, `PARAMETER`, `SAVE` (5.2)
- Procedure features:
  - `INTERFACE` blocks with no *generic-spec* or *module-procedure-stmt* (12.3.2.1)
  - optional arguments (5.2.2)
  - keyword argument passing (12.4.1 /R1212)
- Syntax improvements:
  - long (31 character) names (3.2.2)
  - lower case letters (3.1.7)
  - use of "_" in names (3.1.3)
  - "!" initiated comments, both full line and trailing (3.3.2.1)

C.2 HPF 1.1 Directives and Language Extensions in the HPF 1.1 Subset

The following HPF 1.1 directives and language extensions to Fortran 90 were included in the HPF 1.1 Subset:

- The basic data distribution and alignment directives: `ALIGN`, `DISTRIBUTE`, `PROCESSORS`. and `TEMPLATE`.
- The *forall-statement* (but not the *forall-construct*).
- The `INDEPENDENT` directive.
- The `SEQUENCE` and `NO SEQUENCE` directives.
- The system inquiry intrinsic functions `NUMBER_OF_PROCESSORS` and `PROCESSORS_SHAPE`.
- The computational intrinsic functions `ILEN`, and the HPF extended Fortran intrinsics `MAXLOC` and `MINLOC`, with the restriction that any actual argument expression corresponding to an optional `DIM` argument must be an initialization expression.

For a discussion of the rationale by which features were chosen for the HPF 1.1 Subset, please consult HPF Language Specification Version 1.1.

Annex D

Previous HPFF Acknowledgments

The current HPF 2.0 document would not have been possible without the contributions of the previous series of HPFF meetings. Following are the acknowlegements for those efforts.

D.1 HPFF Acknowledgments

Technical development for HPF 1.0 was carried out by subgroups, and was reviewed by the full committee. Many people served in positions of responsibility:

- Ken Kennedy, Convener and Meeting Chair;
- Charles Koelbel, Executive Director and Head of the FORALL Subgroup;
- Mary Zosel, Head of the Fortran 90 and Storage Association Subgroup;
- Guy Steele, Head of the Data Distribution Subgroup;
- Rob Schreiber, Head of the Intrinsics Subgroup;
- Bob Knighten, Head of the Parallel I/O Subgroup;
- Marc Snir, Head of the Extrinsics Subgroup;
- Joel Williamson and Marina Chen, Heads of the Subroutine Interface Subgroup; and
- David Loveman, Editor.

Geoffrey Fox convened the first HPFF meeting with Ken Kennedy and later led a group to develop benchmarks for HPF. Clemens-August Thole organized a group in Europe and was instrumental in making this an international effort. Charles Koelbel produced detailed meeting minutes that were invaluable to subgroup heads in preparing successive revisions to the draft proposal. Guy Steele developed LaTeX macros for a variety of tasks, including formatting BNF grammar, Fortran code and pseudocode, and commentary material; the document would have been much less aesthetically pleasing without his efforts.

Many companies, universities, and other entities supported their employees' attendance at the HPFF meetings, both directly and indirectly. The following organizations were represented at two or more meetings by the following individuals (not including those present at the first HPFF meeting in January of 1992, for which there is no accurate attendee list):

Alliant Computer Systems Corporation	David Reese

Amoco Production Company	Jerrold Wagener, Rex Page
Applied Parallel Research	John Levesque, Rony Sawdayi, Gene Wagenbreth
Archipel	Jean-Laurent Philippe
CONVEX Computer Corporation	Joel Williamson
Cornell Theory Center	David Presberg
Cray Research, Inc.	Tom MacDonald, Andy Meltzer
Digital Equipment Corporation	David Loveman
Fujitsu America	Siamak Hassanzadeh, Ken Muira
Fujitsu Laboratories	Hidetoshi Iwashita
GMD-I1.T, Sankt Augustin	Clemens-August Thole
Hewlett Packard	Maureen Hoffert, Tin-Fook Ngai, Richard Schooler
IBM	Alan Adamson, Randy Scarborough, Marc Snir, Kate Stewart
Institute for Computer Applications in Science & Engineering	Piyush Mehrotra
Intel Supercomputer Systems Division	Bob Knighten
Lahey Computer	Lev Dyadkin, Richard Fuhler, Thomas Lahey, Matt Snyder
Lawrence Livermore National Laboratory	Mary Zosel
Los Alamos National Laboratory	Ralph Brickner, Margaret Simmons
Louisiana State University	J. Ramanujam
MasPar Computer Corporation	Richard Swift
Meiko, Inc.	James Cownie
nCUBE, Inc.	Barry Keane, Venkata Konda
Ohio State University	P. Sadayappan
Oregon Graduate Institute of Science and Technology	Robert Babb II
The Portland Group, Inc.	Vince Schuster
Research Institute for Advanced Computer Science	Robert Schreiber
Rice University	Ken Kennedy, Charles Koelbel
Schlumberger	Peter Highnam
Shell	Don Heller
State University of New York at Buffalo	Min-You Wu
SunPro and Sun Microsystems	Prakash Narayan, Douglas Walls
Syracuse University	Alok Choudhary, Tom Haupt
TNO-TU Delft	Edwin Paalvast, Henk Sips
Thinking Machines Corporation	Jim Bailey, Richard Shapiro, Guy Steele
United Technologies Corporation	Richard Shapiro
University of Stuttgart	Uwe Geuder, Bernhard Woerner, Roland Zink
University of Southampton	John Merlin
University of Vienna	Barbara Chapman, Hans Zima
Yale University	Marina Chen, Aloke Majumdar

Many people contributed sections to the final language specification and HPF Journal of Development, including Alok Choudhary, Geoffrey Fox, Tom Haupt, Maureen Hoffert, Ken Kennedy, Robert Knighten, Charles Koelbel, David Loveman, Piyush Mehrotra, John Merlin, Tin-Fook Ngai, Rex Page, Sanjay Ranka, Robert Schreiber, Richard Shapiro, Marc Snir, Matt Snyder, Guy Steele, Richard Swift, Min-You Wu, and Mary Zosel. Many others contributed shorter passages and examples and corrected errors.

Because public input was encouraged on electronic mailing lists, it is impossible to identify all who contributed to discussions; the entire mailing list was over 500 names long. Following are some of the active participants in the HPFF process not mentioned above:

N. Arunasalam	Werner Assmann	Marc Baber
Babak Bagheri	Vasanth Bala	Jason Behm
Peter Belmont	Mike Bernhardt	Keith Bierman
Christian Bishof	John Bolstad	William Camp
Duane Carbon	Richard Carpenter	Brice Cassenti
Doreen Cheng	Mark Christon	Fabien Coelho
Robert Corbett	Bill Crutchfield	J. C. Diaz
James Demmel	Alan Egolf	Bo Einarsson
Pablo Elustondo	Robert Ferrell	Rhys Francis
Hans-Hermann Frese	Steve Goldhaber	Brent Gorda
Rick Gorton	Robert Halstead	Reinhard von Hanxleden
Hiroki Honda	Carol Hoover	Steven Huss-Lederman
Ken Jacobsen	Elaine Jacobson	Behm Jason
Alan Karp	Ronan Keryell	Anthony Kimball
Ross Knippe	Bruce Knobe	David Kotz
Ed Krall	Tom Lake	Peter Lawrence
Bryan Lawver	Bruce Leasure	Stewart Levin
David Levine	Theodore Lewis	Woody Lichtenstein
Ruth Lovely	Doug MacDonald	Raymond Man
Stephen Mark	Philippe Marquet	Jeanne Martin
Oliver McBryan	Charlie McDowell	Michael Metcalf
Charles Mosher	Len Moss	Lenore Mullin
Yoichi Muraoka	Bernie Murray	Vicki Newton
Dale Nielsen	Kayutov Nikolay	Steve O'Neale
Jeff Painter	Cherri Pancake	Harvey Richardson
Bob Riley	Kevin Robert	Ron Schmucker
J.L. Schonfelder	Doug Scofield	David Serafini
G.M. Sigut	Anthony Skjellum	Niraj Srivastava
Paul St.Pierre	Nick Stanford	Mia Stephens
Jaspal Subhlok	Xiaobai Sun	Hanna Szoke
Bernard Tourancheau	Anna Tsao	Alex Vasilevsky
Stephen Vavasis	Arthur Veen	Brian Wake
Ji Wang	Karen Warren	D.C.B. Watson
Matthijs van Waveren	Robert Weaver	Fred Webb
Stephen Whitley	Michael Wolfe	Fujio Yamamoto
Marco Zagha		

The following organizations made the language draft available by anonymous FTP access and/or mail servers: AT&T Bell Laboratories, Cornell Theory Center, GMD-I1.T (Sankt Augustin), Oak Ridge National Laboratory, Rice University, Syracuse University, and Thinking Machines Corporation. These outlets were instrumental in distributing the document.

The High Performance Fortran Forum also received a great deal of volunteer effort in nontechnical areas. Theresa Chatman and Ann Redelfs were responsible for most of the meeting planning and organization, including the first HPFF meeting, which drew over 125 people. Shaun Bonton, Rachele Harless, Rhonda Perales, Seryu Patel, and Daniel Swint helped with many logistical details. Danny Powell spent a great deal of time handling the financial details of the project. Without these people, it is unlikely that HPF would have been completed.

HPFF operated on a very tight budget (in reality, it had no budget when the first meeting was announced). The first meeting in Houston was entirely financed from the conferences budget of the Center for Research on Parallel Computation, an NSF Science and Technology Center. DARPA and NSF have supported research at various institutions that have made a significant contribution towards the development of High Performance Fortran. Their sponsored projects at Rice, Syracuse, and Yale Universities were particularly influential in the HPFF process. Support for several European participants was provided by ESPRIT through projects P6643 (PPPE) and P6516 (PREPARE).

D.2 HPFF94 Acknowledgments

The HPF 1.1 version of the document was prepared during the HPFF94 series of meetings. A number of people shared technical responsibilities for the activities of the HPFF94 meetings:

- Ken Kennedy, Convener and Meeting Chair;
- Mary Zosel, Executive Director and head of CCI Group 2;
- Richard Shapiro, Head of CCI Group 1;
- Ian Foster, Head of Tasking Subgroup;
- Alok Choudhary, Head of Parallel I/O Subgroup;
- Chuck Koelbel, Head of Irregular Distributions Subgroup;
- Rob Schreiber, Head of Implementation Subgroup;
- Joel Saltz, Head of Benchmarks Subgroup;
- David Loveman, Editor, assisted by Chuck Koelbel, Rob Schreiber, Guy Steele, and Mary Zosel, section editors.

Attendence at the HPFF94 meetings included the following people from organizations that were represented two or more times.

Don Heller	Ames Laboratory
Jerrold Wagener	Amoco Production Company
John Levesque	Applied Parallel Research
Ian Foster	Argonne National Laboratory
Terry Pratt	CESDIS/NASA Goddard
Jim Cowie	Cooperating Systems
Andy Meltzer, Jon Steidel	Cray Research, Inc.
David Loveman	Digital Equipment Corporation
Bruce Olsen	Hewlett Packard
E. Nunohiro, Satoshi Itoh	Hitachi
Henry Zongaro	IBM
Piyush Mehrotra	Institute for Computer Applications in Science & Engineering
Bob Knighten, Roy Touzeau	Intel SSD
Mary Zosel, Bor Chan, Karen Warren	Lawrence Livermore National Laboratory
Ralph Brickner	Los Alamos National Laboratory
J. Ramanujam	Louisiana State University

Paula Vaughan, Donna Reese	Mississippi State University and NSF ERC
Shoichi Sakon, Yoshiki Seo	NEC
P. Sadayappan, Chua-Huang Huang	Ohio State University
Andrew Johnson	OSF Research Institute
Chip Rodden, Jeff Vanderlip	Pacific Sierra Research
Larry Meadows, Doug Miles	The Portland Group, Inc.
Robert Schreiber	Research Institute for Advanced Computer Science
Ken Kennedy, Charles Koelbel	Rice University
Ira Baxter	Schlumberger
Alok Choudhary	Syracuse University
Guy Steele	Thinking Machines Corporation, Sun Microsystems
Richard Shapiro	Thinking Machines Corp., Silicon Graphics Inc.
Scott Baden, Val Donaldson	University of California, San Diego
Robert Babb	University of Denver
Joel Saltz, Paul Havlak	University of Maryland
Nicole Nemer-Preece	University of Missouri-Rolla
Hans Zima, Siegfried Benkner, Thomas Fahringer	University of Vienna

An important activity of HPFF94 was the processing of the many items submitted for comment and interpretation which led to the HPF 1.1 update of the language document. A special acknowlegement goes to Henry Zongaro, IBM, for many thoughtful questions exposing dark corners of language design that were previously overlooked, and to Guy Steele, Thinking Machines/Sun Microsystems for his analysis of, and solutions for some of the thornier issues discussed. And general thanks to the people who submitted comments and interpretation requests, including:

David Loveman, Michael Hennecke, James Cownie, Adam Marshall, Stephen Ehrlich, Mary Zosel, Matt Snyder, Larry Meadows, Dick Hendrickson, Dave Watson, John Merlin, Vasanth Bala, Paul.Wesson, Denis.Hugli, Stanly Steinberg, Henk Sips, Henry Zongaro, Eiji_Nunohiro, Jens Bloch Helmers, Rob Schreiber, David B. Serafini, and Allan Knies.

Other special mention goes to Chuck Koelbel at Rice University for continued maintenance of the HPFF mailing lists, to Donna Reese and staff at Mississippi State University for establishing and maintaining a WWW home-page for HPFF, and to the University of Maryland for establishing a benchmark FTP site.

Theresa Chatman and staff at Rice University were responsible for meeting planning and organization and Danny Powell continued to handle financial details of the project.

HPFF94 received direct support for research and administrative activities from grants from ARPA, DOE, and NSF.

Bibliography

[1] American National Standards Institute, Inc., 1430 Broadway, New York, NY. *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, approved April 3, 1978.

[2] American National Standards Institute, Inc., 1430 Broadway, New York, NY. *American National Standard for Information Systems Programming Language FORTRAN*, S8 (X3.9-198x) Revision of X3.9-1978, Draft S8, Version 104, April 1987.

[3] High Performance Fortran Forum. *High Performance Fortran Language Specification* Scientific Programming, 2,1, 1993. Also published as: CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1992 (revised May. 1993). Also published as: Fortran Forum, 12,4, Dec. 1993 and 13,2, June 1994.

[4] High Performance Fortran Forum. *High Performance Fortran Language Specification, version 1.0* CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1992 (revised May. 1993).

[5] US Department of Defense. *Military Standard, MIL-STD-1753: FORTRAN, DoD Supplement to American National Standard X3.9-1978*, November 9, 1978.

Annex E

Policy and Mechanism for Recognized Extrinsic Interfaces

HPF defines certain extrinsics such as `HPF_LOCAL`, and `HPF_SERIAL` as interfaces that HPFF believes are useful to the HPF community. But there are many more such extrinsic interfaces beyond those maintained by HPFF. HPFF has a adopted a policy of formally recognizing certain extrinsic interface definitions, where the interface, and its addition to the HPF document is considered to be a service to the HPF community. Examples are language bindings to HPF or library packages.

E.1 Extrinsic Policy

To be considered for HPFF recognition, a proposed extrinsic must demonstrate the following things. It should be noted, however, that meeting these criteria does not guarentee acceptance of a proposed interface by HPFF.

- conformance to HPF rules for calling extrinsics,
- significant new functionality,
- existing practice such as users, implementations, etc.,
- institutional backing with evidence of ongoing support,
- coherent documentation,
- non-proprietary interface definition, and
- copyright goes to HPFF for interface, with permission to use (royalty free).

If a proposed extrinsic is accepted by HPFF, then:

- HPFF will recogize the interface and reference it in documentation, but HPFF does not assume responsibility for the extrinsic or its interface.
- The sponsor of the extrinsic must continue to conform to the HPF interface rules for extrinsics. The interface HPFF approves must not change without HPFF approval.
- The sponsor must assume responsibility for any CCI requests concernting the extrinsic.

A list of recognized extrinsic interfaces will be included in HPF documentation, with the following guidelines:

- There should be a single page introduction to the extrinsic which contains:
  - the name of the extrinsic,
  - a brief abstract of functionality,
  - a brief and informal description of the interface,
  - information about platform and system availability, and
  - reference and contacts for formal documentation, continued responsibility, and additional information (e.g. compiler availability).
- There should be about two pages with short examples of usage.
- A short paper with the formal definition of the interface and an informal description of the functionality of the extrinsic.

E.2 Extrinsic Interface Mechanism

The HPF www-home page will have instructions for submission of an extrinsic interface. For HPFF consideration, the sponsor prepares a proposal that includes:

- a statement of what significant new functionality is provided,
- a description of existing practice,
- a statement of institutional backing with evidence of ongoing support,
- a copy of the complete documentation or a reference to an online version of the documentation,
- a draft of the text (described above) that would be included in the HPFF documentation, and
- a statement justifying the claim that the interface follows HPF conventions for calling extrinsics.

If the proposed extrinsic interface is approved by HPFF, the sponsor then submits:

- a formal statement for HPFF records that the interface definition is non-proprietary and that the copyright of the interface belongs HPFF,
- the formal contact for CCI and continued maintenance of the interface, and
- a copy of the interface documentation formatted for HPFF use, including a copy in the current document and web mark-up languages.

Annex F

HPF_CRAFT

HPF_CRAFT is a hybrid language, combining an SPMD execution model with high performing HPF features. The model combines the multi-threaded execution of HPF_LOCAL and the HPF syntax. The goal of HPF_CRAFT is to attain the potential performance of an SPMD programming model with access to HPF features and a well-defined extrinsic interface to HPF.

F.1 Introduction

HPF_CRAFT is a hybrid language, combining an SPMD execution model with high performing and portable HPF features. The model combines the multi-threaded execution of HPF_LOCAL and the HPF syntax and features. The goal of HPF_CRAFT is to attain the potential performance of an SPMD programming model with access to HPF features and a well-defined extrinsic interface to HPF. It is built on top of the HPF_LOCAL extrinsic environment.

SPMD features and a multi-threaded model allow the user to take advantage of the performance and opportunity for low level access of a more general purpose programming model. Including HPF data distribution features gives the programmer access to high performing aspects of both models, but with the added responsibility of working with a more low-level execution model. HPF_CRAFT is best suited for platforms that support one way communication features, but is consistent with HPF and easily targeted for platforms that have HPF and can support SPMD programming styles.

The HPF features included in HPF_CRAFT are a subset of the full HPF language chosen for their performance and their broad portability and ease of use. HPF_CRAFT contains additional features to support SPMD programming styles. There are some differences from HPF, however. For example, I/O causes differences; in HPF_CRAFT different processors are allowed to read from different files at the same time, in HPF the processors must all read from the same file. The differences in the models are principally caused by the multi-threaded execution model and the introduction of HPF_LOCAL data rules.

HPF_CRAFT allows for the notion of *private data*. Data defaults to a mapping in which data items are allocated so that each processor has a unique copy. The values of the individual data items and the flow of control may vary from processor to processor within HPF_CRAFT. This behavior is consistent with the behavior of HPF_LOCAL. In HPF_CRAFT a processor may be individually named and code executed based upon which processor it is executing on. HPF_CRAFT also allows for the notion of *private loops*. A private loop is executed in entirety by each processor.

The rules governing the interface to HPF_CRAFT subprograms are similar to those for the HPF_LOCAL interface. Dummy arguments use a hybrid of the interfaces between HPF and itself and that of HPF and HPF_LOCAL. Explicitly mapped dummy arguments behave just as they do in HPF, while default (private) dummy arguments use the HPF_LOCAL calling convention.

HPF_CRAFT will be initially made available on Cray MPP systems and may also be available on Cray vector architectures. Future versions of HPF_CRAFT are possible on other vendor's architectures as well.

HPF_CRAFT is being implemented for Cray Research by The Portland Group, Inc. For Cray systems, HPF_CRAFT may be obtained through the Cray Research Inc. Orderdesk,

Cray Research Inc.
orderdsk@cray.com
(612) 683-5907

Additional formal documentation, requests, and suggestions can be made to

The Portland Group
9150 SW Pioneer Ct., Suite H
Wilsonville, OR 97070
(503) 682-2806
trs@pgroup.com

F.2 Examples of Use

HPF_CRAFT is intended for use in circumstances where greater control and performance are desired for MIMD style architectures. Since data may be declared to be private, local control is made more available and since processor information is available message passing and direct memory access programming styles can be seamlessly integrated with explicitly mapped data.

The following examples show some of the capabilities of HPF_CRAFT that are different from those of HPF. Others, such as integrated message passing and synchronization primitives are not shown. Much of HPF can also be used within HPF_CRAFT.

Example 1 illustrates the difference between the default distribution for data and the distribution of mapped data.

```
!  Example 1

      INTEGER PRIVATE_A(100, 20), PRIVATE_B(12, 256), PRIVATE_C
      INTEGER MAPPED_A(100, 20), MAPPED_B(12, 256), MAPPED_C
!HPF$  DISTRIBUTE MAPPED_A(BLOCK, BLOCK), MAPPED_B(BLOCK, *), MAPPED_C
```

In the above example, given 8 processors, there would be 8 * 100 * 20 (or 16,000) elements in the array `PRIVATE_A`. Each processor contains an entire array named `PRIVATE_A`. The elements of `PRIVATE_A` on processor 1 cannot be referenced using implicit syntax by any other processor. There are only 100 * 20 (or 2000) elements of array MAPPED_A, however, and these elements are distributed about the machine in a (BLOCK, BLOCK) fashion.

The difference between the PRIVATE_A declaration in HPF_CRAFT and that in HPF is the most instructive. In HPF_CRAFT each processor contains one copy of the array, and the values of the elements of the array may vary from processor to processor. HPF implementations are permitted to make one copy of the array per processor the default, but the values of these copies must remain coherent across all processors. In HPF there is no way to write a conforming program in which different processors have different values for the same array.

Example 2 shows the usefulness of the ON clause for the INDEPENDENT loop as well as giving an example of how private data may be used.

```
!  Example 2

      PRIVATE_C = 0
!HPF$ INDEPENDENT (I, J) ON MAPPED_B(I, J)
      DO J=1,256
        DO I=1,12
          MAPPED_B(I, J) = MAPPED_B(I, J) + 5
          PRIVATE_C = PRIVATE_C + MAPPED_B(I, J)
        ENDDO
      ENDDO
```

In this example, each iteration is executed on the processor containing the data that is mapped to it. The user was allowed to specify this.

In addition, the private variable PRIVATE_C is used to compute a total for each processor. At the end of execution of the loop, the values of PRIVATE_C may be different on each processor depending upon the values in the elements of the array on each processor. This data may be used as is, or it can be quickly summed using a barrier or an ATOMIC UPDATE.

Example 3 shows the final total value being combined into the variable MAPPED_C whose value is available to all processors.

```
!  Example 3

      MAPPED_C = 0
!HPF$ ATOMIC UPDATE
      MAPPED_C = MAPPED_C + PRIVATE_C
```

Example 4 shows how the language allows private data to vary from processor to processor.

```
!  Example 4

      IF (MY_PE() .EQ. 5) THEN
        PRIVATE_C = some-big-expression
      ENDIF
```

In this example, PRIVATE_C on processor 5 will have the result of *some-big-expression*. Each processor can do distinctly different work and communicate through mapped data.

The code fragment in Example 5 is from an application and shows a few features of the language.

```
!  Example 5

!HPF$ GEOMETRY G(*, CYCLIC)
      REAL FX(100,100), FY(100,100), FZ(100,100)
!HPF$ DISTRIBUTE (G) ::  FX,FY,FZ
      REAL FXP(100,16,100), FYP(100,16,100)
!HPF$ DISTRIBUTE FXP(*,*, BLOCK) FYP(*,*, BLOCK)
      INTEGER CELL, ATOM, MAP(1000), NACELL(1000)

!HPF$ INDEPENDENT (CELL) ON FX(1,CELL)
      DO CELL=1,100
         JCELL0 = 16*(CELL-1)
         DO NABOR = 1, 13
            JCELL = MAP(JCELL0+NABOR)
            DO ATOM=1, NACELL(CELL)
               FX(ATOM, CELL) = FX(ATOM, CELL) + FXP(ATOM, NABOR, JCELL)
               FY(ATOM, CELL) = FY(ATOM, CELL) + FYP(ATOM, NABOR, JCELL)
            ENDDO
         ENDDO
      ENDDO
```

The GEOMETRY directive allows the user to generically specify a mapping and use it to apply to many arrays (they need not have the same extents.)

Example 5 has a single INDEPENDENT loop which is the outer loop. It executes 100 iterations total. Within this loop the private value of JCELL0 is set for each processor (ensuring that it is a local computation everywhere.) Nested inside the INDEPENDENT loop is a private loop; this loop executes 13 times *per* processor. Inside this loop JCELL is computed locally on each processor, minimizing unnecessary communication. Finally the innermost loop is also private.

F.3 External Interface

This section describes the behavior when an HPF_CRAFT routine is called from HPF.

The calling convention and argument passing rules for HPF_CRAFT are a hybrid of those for HPF calling HPF_LOCAL and HPF calling HPF. Explicit interfaces are required. Where dummy arguments are private (default) storage, the HPF calling HPF_LOCAL conventions are used. Where dummy arguments are explicitly mapped, the calling convention matches HPF calling HPF.

There are a number of constraints on HPF_CRAFT routines that are called from HPF. The following is a list of restrictions placed on HPF_CRAFT routines called from HPF:

- Recursive HPF_CRAFT routines cannot be called from HPF.

- HPF_CRAFT routines called from HPF may only enter the routine at a single place (no alternate entries).
- An HPF_CRAFT supprogram may not be invoked directly or indirectly from within the body of a `FORALL` construct or within the body of an `INDEPENDENT DO` loop that is inside an HPF program.
- The attributes (type, kind, rank, optional, intent) of the dummy arguments in a supprogram called by HPF must match the attributes of the corresponding dummy arguments in the explicit interface.
- A dummy argument of an HPF_CRAFT supprogram called by HPF
  - must not be a procedure name.
  - must not have the `POINTER` attribute.
  - must not be sequential, unless it is also `PE_PRIVATE`.
  - must have assumed shape even when it is explicit shape in the interface.
  - if scalar, it must be mapped so that each processor has a copy of the argument.
- The default mapping of scalar dummy arguments and of scalar function results when an HPF program calls an HPF_CRAFT routine is that it is replicated on each processor.

If a dummy argument of an `EXTRINSIC('HPF_CRAFT')` routine interface block is an array and the dummy argument of the HPF_CRAFT supprogram has the default private mapping, then the corresponding dummy argument in the specification of the HPF_CRAFT procedure must be an array of the same rank, type, and type parameters. When the extrinsic procedure is invoked, the dummy argument is associated with the local array that consists of the subgrid of the global array that is stored locally.

If the dummy argument of the HPF_CRAFT supprogram is explicitly mapped, it must have the same mapping as the dummy argument of the `EXTRINSIC('HPF_CRAFT')` supprogram. Note that this restriction does not require actual and dummy arguments to match and is no more stringent than saying that mappings of dummy arguments in interface blocks must match those in the actual routine.

F.4 Execution Model

HPF_CRAFT is built upon the fundamental execution model of HPF_LOCAL, augmented with data mapping and work distribution features from HPF. It is also augmented with explicit low-level control features, many taken from Cray Research's CRAFT language.

In HPF_CRAFT there is a single task on each processor and all tasks begin executing in parallel, with data defaulting to a private distribution, the same default distribution used in HPF_LOCAL. Each processor gets a copy of the data storage unless specified otherwise by the user. Consequently I/O works identically to I/O in HPF_LOCAL and message passing libraries are easily integrated.

Simply stated, the execution model is that of HPF_LOCAL.

To provide correct behavior when explicitly mapped data is involved, this model defines implicit barrier points at which the execution model requires that all processors must stop and wait for the execution of all other processors before continuing. These barriers add additional semantics to the HPF_LOCAL behavior. An implementation may remove any of these barriers that are deemed unnecessary, but *every* processor must participate in the barriers at each one of these points.

The points where there are implicit barriers are conceptually after those instances in which the processors in the HPF_CRAFT program are executing cooperatively, as if in an HPF program (e.g., after an `INDEPENDENT` loop). An HPF_CRAFT program treats operations on explicitly mapped objects as if they were operations in an HPF program and it treates operations on private data as if they were executed within the HPF_LOCAL framework. It is occasionally useful for an advanced programmer to indicate to the compilation system where barriers are not needed; HPF_CRAFT has syntax to allow this capability.

F.5 HPF_CRAFT Functional Summary

HPF_CRAFT contains a number of features not available in HPF, and restricts the usage of many of the features currently available. The following is a concise list of the differences.

- `INDEPENDENT` has been extended to better support an `ON` clause.
- There are new rules defining the interaction of explicitly mapped and private data.
- Parallel inquiry intrinsics `IN_PARALLEL()` and `IN_INDEPENDENT()` have been added.
- Serial regions (`MASTER / END MASTER`) have been added.
- Explicit synchronization primitives are provided.
- The `ATOMIC UPDATE`, `SYMMETRIC`, and `GEOMETRY` directives have been added.
- Many other compiler information directives have been added to assist the compiler in producing good quality code.

F.5.1 Data Mapping Features

Data mapping features provided are those that have been found useful most often. When data is explicitly mapped, only one copy of the data storage is created unless the explicit mapping directs otherwise. The value of explicitly mapped replicated data items must be consistent between processors as is the case in HPF. Storage and sequence association for explicitly mapped arrays is not guaranteed in HPF_CRAFT. For private data, storage and sequence association follows the Fortran 90 rules.

A new directive is included for completeness: `PE_PRIVATE`, which specifies that the data should conform to the default behavior. The values of private varaibles may vary on different processors.

F.5.2 Subprogram Interfaces

The behavior and requirements of an HPF_CRAFT program at subprogram interfaces may be divided into three cases. Each case is also available using some combination of HPF and HPF_LOCAL. For dummy arguments that are explicitly mapped, the behavior is identical to that of HPF. All processors must cooperate in a subprogram invocation that remaps or explicitly maps data. In other words, if an explicit interface is required (by the HPF rules) or the subprogram declares explicitly mapped data, the subprogram must be called on all processors. Processors need not cooperate if there are only reads to non-local data. The INHERIT attribute may only be applied to explicitly mapped data.

Data that has the default private mapping (case two) the behavior of an HPF_CRAFT subprogram at subprogram interfaces is identical to that of HPF_LOCAL. Data is passed individually on every processor and the processors need not interact in any way.

When a subprogram is passed actual arguments that are a combination of both explicitly mapped data and private data, the explicitly mapped data follows the HPF rules and the private data follows the HPF_LOCAL rules.

In case three, the user has the option of passing data with explicitly mapped actual arguments to dummy arguments that are not explicitly mapped (i.e., private.) The mapping rules for this data are identical to the mapping rules when HPF calls an HPF_LOCAL subprogram. The data remains "in-place." All HPF arrays are logically carved up into pieces; the HPF_CRAFT procedure executing on a particular physical processor sees an array containing just those elements of the global array that are mapped to that physical processor. There is implicit barrier synchronization after an INDEPENDENT loop. Transfer of control into or out of an INDEPENDENT loop is prohibited.

Finally, it is undefined behavior when an actual argument is private and the dummy argument is explicitly mapped. A definition could be supplied for this interaction, but it is the same solution that one might propose for a calling sequence when HPF_LOCAL subprograms call HPF subprograms.

F.5.3 The INDEPENDENT Directive

The INDEPENDENT directive is part of HPF_CRAFT with the same semantics as in HPF. However, within INDEPENDENT loops the values of private data may vary from processor to processor. INDEPENDENT applied to FORALL has identical syntax and semantics as in HPF.

An HPF independent loop optionally may have a NEW clause. The NEW clause is not required by HPF_CRAFT for default (not explicitly mapped) data. In HPF_CRAFT data defaults to private so values may differ from processor to processor.

Private data has slightly different behavior than data specified in the NEW clause. The value of a private datum on each processor can be used beyond a single iteration of the loop. Private data may be used to compute local sums, for example. The values of data items named in a NEW clause may not be used beyond a single iteration. The NEW clause asserts that the INDEPENDENT directive would be valid if new objects were created for the variables named in the clause for each iteration of the loop. The semantics of the NEW clause are identical in HPF_CRAFT and HPF.

The semantics of an INDEPENDENT applied to loops containing private data references changes with respect to the private data. The change can be summarized to say that instead of indicating that iterations have no dependencies upon one-another, with respect to the private data, iterations on different processors have no dependencies upon one-another.

F.5.4 The ON Clause

In addition to the version of INDEPENDENT available from HPF, a new version of INDEPENDENT is included that incorporates the ON clause. There are a number of differences between the versions of INDEPENDENT with and without the ON clause.

The new version of the INDEPENDENT directive may be applied to the first of a group of tightly nested loops and may apply to more than one of them. This more easily facilitates the use of the ON clause. The current INDEPENDENT directive applies only to a single loop nest. The INDEPENDENT directive is extended so that multiple loop nests can be named. The general syntax for these new independent loops is as follows:

```
!HPF$ INDEPENDENT ($I_1$,$I_2$,...,$I_n$) ON array-name($h_1(I_1)$,$h_2(I_2)$,...,$h_n(I_n)$)
      DO $I_1 = L_1, U_1, S_1$
            DO $I_2 = L_2, U_2, S_2$
              ...
                DO $I_n = L_n, U_n, S_n$
                  ...
                END DO
              ...
            END DO
      END DO
```

The syntax and semantics of INDEPENDENT with the ON clause are different from its syntax and semantics without the ON clause. With the ON clause the directive states that there are no cross-processor dependencies, but there may be dependencies between iterations on a processor. There is an implicit barrier synchronization after an INDEPENDENT loop. Transfer of control into or out of an INDEPENDENT loop is prohibited.

The iteration space of an INDEPENDENT nest must be rectangular. That is, the lower loop bound, the upper loop bound, and the step expression for each loop indicated by the INDEPENDENT induction list must be invariant with regard to the INDEPENDENT nest. Each index expression of *array-name* in the ON clause (the functions h_i above,) must be one of the following two forms:

[*a* * loop_control_variable +] *b*
[*a* * loop_control_variable -] *b*

where *a* and *b* must be integer values; they can be expressions, constants, or variables. The values of *a* and *b* must be invariant with regard to the INDEPENDENT loop nest. For example, specifying A(I,J,K) is valid. Specifying A(3,I+J,K) is not valid. Specifying A(I,I,K) is not valid because I appears twice. Division is prohibited in any index expression of the ON clause.

F.5.5 Array Syntax

Array syntax is treated identically in HPF_CRAFT as in HPF for explicitly mapped objects. For private objects the behavior is identical to that of HPF_LOCAL. When private objects and explicitly mapped objects are combined the rules are as follows:

$$\mathit{result} = \mathit{rhs}_1 \ \mathrm{op}_1 \ \mathit{rhs}_2 \ \mathrm{op}_2 \ \ldots \ \mathrm{op}_m \ \mathit{rhs}_n$$

- If *result* is explicitly mapped and all *rhs* arrays are explicitly mapped, the work is distributed as in HPF.
- If *result* is private and all *rhs* arrays are private the computation is done on all processors as an HPF_LOCAL program would do it.
- If *result* is private and all *rhs* arrays are explicitly mapped, the work is distributed as in HPF and the values of the results are broadcast to the *result* on each processor.

- If *result* is explicitly mapped and *not* all *rhs* arrays are explicitly mapped, the results of the operation are undefined, unless all corresponding elements of all private *rhs* arrays have the same values.
- If *result* is private and some, but not all *rhs* arrays are explicitly mapped, the value is computed on each processor and saved to the local *result*.

All processors must participate in any array syntax statement in which the value of an explicitly mapped array is modified, and there is implicit barrier synchronization after the statement executes.

F.5.6 Treatment of FORALL and WHERE Statements

The `FORALL` and `WHERE` statements are treated exactly as in HPF when data is explicitly mapped. When private data is modified, the statement is executed separately on each processor. Finally, when data in a `FORALL` or `WHERE` are mixed, the rules for array syntax apply. If any explicitly mapped data item is modified in a *forall-stmt* or *where-stmt* then arrays in the *forall-header* or *where-header* must be explicitly mapped. In a `FORALL` construct, if any explicitly mapped array is modified, all modified arrays must be explicitly mapped. There is an implicit barrier synchronization after `FORALL` and `WHERE` statements if any arrays in the *forall-header* or *where-header* are explicitly mapped.

F.5.7 Synchronization Primitives

A number of synchronization primitives are provided. These primitives include:

Barriers (test, set, wait)
Locks (test, set, clear)
Critical Sections
Events (test, set, wait, clear)

Barriers provides an explicit mechanism for a task to indicate its arrival at a program point and to wait there until all other tasks arrive. A task may test and optionally wait at an explicit barrier point. In the following example, a barrier is used to make sure that *block3* is not entered by any task until all tasks have completed execution of *block1*.

```
block1
CALL SET_BARRIER()
block2
CALL WAIT_BARRIER()
block3
```

The following example performs a similar function as above. However, while waiting for all tasks to arrive at the barrier, the early tasks perform work within a loop.

```
block1
CALL SET_BARRIER()
DO WHILE (.NOT. TEST_BARRIER())
      block2
END DO
block3
```

Locks are used to prevent the simultaneous access of data by multiple tasks.

The `SET_LOCK(`*lock*`)` intrinsic sets the mapped integer variable *lock* atomically. If the lock is already set, the task that called `SET_LOCK` is suspended until the lock is cleared by another task and then sets it. Individual locks may be tested or cleared using *result* = `TEST_LOCK(`*lock*`)` and `CLEAR_LOCK(`*lock*`)` respectively.

A *critical section* protects access to a section of code rather than to a data object. The `CRITICAL` directive marks the beginning of a code region in which only one task can enter at a time. The `END CRITICAL` directive marks the end of the critical section. Transfer of control into or out of a critical section is prohibited.

```
!HPF$ CRITICAL
      GLOBAL_SUM = GLOBAL_SUM + LOCAL_SUM
!HPF$ END CRITICAL
```

Events are typically used to record the state of a program's execution and to communicate that state to another task. Because they do not set locks, as do the lock routines described earlier, they cannot easily be used to enforce serial access of data. They are suited to work such as signalling other tasks when a certain value has been located in a search procedure. There are four routines needed to perform the event functions, and each requires a mapped argument.

The `SET_EVENT(`*event*`)` routine sets or *posts* an event; it declares that an action has been accomplished or a certain point in the program has been reached. A task can post an event at any time, whether the state of the event is cleared or already posted. The `CLEAR_EVENT(`*event*`)` routine clears an event, the `WAIT_EVENT(`*event*`)` routine waits until a particualr event is posted, and the *result* = `TEST_EVENT(`*event*`)` function returns a logical value indicating whether a particular event has been posted.

F.5.8 Barrier Removal

You can explicitly remove an implicit barrier after any `INDEPENDENT` loop, or after any array syntax statement that modifies explicitly mapped arrays, by using the `NO BARRIER` directive.

```
!HPF$ NO BARRIER
```

F.5.9 Serial Regions

It is often useful to enter a region where only one task is executing. This is particularly useful for certain types of I/O. To facilitate this, two directives are provided. In addition, one may optionally attach a `COPY` clause to the `END MASTER` directive which specifies the private data items whose values should be broadcast to all processors. The syntax of this directive is:

```
!HPF$ MASTER
      sequential region
      ...
!HPF$ END MASTER [, COPY( var_1 [, var_2, ..., var_n ])]
```

where *var* is `SYMMETRIC` private data to be copied to the same named private data on other processors.

If a routine is called within a serial region, the routine executes serially; there is no way to get back to parallel execution within the routine. All explicitly mapped data is accessible from within routines called in a serial region, but a routine called from within a serial region cannot allocate explicitly mapped data or remap data. All processors must participate in the invocation of the serial region. Transfer of control into or out of a serial region is not permitted.

F.5.10 Libraries

The HPF Local Routine Library is available in HPF_CRAFT. The HPF_LOCAL extrinsic environment contains a number of libraries that are useful for local SPMD programming and a number of libraries that allow the user to determine global (rather than local) state information. These library procedures take as input the name of a dummy argument and return information on the corresponding global HPF actual argument. They may only be invoked by an HPF_CRAFT procedure that was directly invoked by global HPF code. They may be called only for private data. The libraries reside in a module called `HPF_LOCAL_LIBRARY`.

The HPF Library is available to HPF_CRAFT when called with data that is explicitly mapped and all processors are participating in the call. In addition, as in HPF_LOCAL, the entire HPF Library is available for use with private data. Mixing private and explicitly mapped data in calls to the HPF library produces undefined behavior.

F.5.11 Parallel Inquiry Intrinsics

These intrinsic functions are provided as an extension to HPF. They return a logical value that provides information to the programmer about the state of execution in a program.

```
      IN_PARALLEL()
      IN_INDEPENDENT()
```

F.5.12 Task Identity

`MY_PE()` may be used to return the local processor number. The physical processors are identified by an integer in the range of 0 to *n-1* where *n* is the value returned by the global HPF_LIBRARY function `NUMBER_OF_PROCESSORS`. Processor identifiers are returned by `ABSTRACT_TO_PHYSICAL`, which establishes the one-to-one correspondence between the abstract processors of an HPF processors arrangement and the physical processors. Also, the local library function `MY_PROCESSOR` returns the identifier of the task executing the call.

F.5.13 Parallelism Specification Directives

These directives allow a user to assert that a routine will only be called from within a parallel region, a serial region, or from within both regions. Without these directives an implementation might be required to generate two versions of code for each routine, depending upon implementation strategies. The directives simply make the generated code size smaller and remove a test.

```
!HPF$ PARALLEL_ONLY
!HPF$ SERIAL_ONLY
```

```
!HPF$ PARALLEL_AND_SERIAL
```

The default is PARALLEL_ONLY.

F.5.14 The SYMMETRIC Directive

SYMMETRIC variables are private data that are guaranteed to be at the same storage location on every processor. The feature is beneficial to implementations that provide one-way communication functionality. One task can either *get* or *put* data into another task's symmetric data location, without involving the other task. There is an implicit barrier synchronization after SYMMETRIC data is allocated.

```
      REAL PRIV1(100), PRIV2
!HPF$ SYMMETRIC PRIV1, PRIV2
```

F.5.15 The RESIDENT Directive

The RESIDENT directive can be specified at the loop level and at the routine level. It is an assertion that the references to particular variables in the routine (or loop) are only references to data that are local to the task making the assertion. In the following loop, all references to arrays A, B, and C are local to the task executing each iteration.

```
      REAL A(100), B(100), C(100)
      INTEGER IX(100)
!HPF$ DISTRIBUTE A(BLOCK), B(BLOCK), C(BLOCK)
!HPF$ RESIDENT A

      ...
!HPF$ INDEPENDENT (I) ON B(I) RESIDENT(C)
      DO I = 1, 100
            A(IX(I)) = B(I) + C(IX(I))
      END DO
```

F.5.16 The ATOMIC UPDATE Directive

In HPF_CRAFT, the ATOMIC UPDATE directive tells the compiler that a particular data item or the elements of a particular array for a specified operation must be updated atomically. This can be used within loops or in array syntax and applies to both the elements of an array with an assignment of a permutation and the elements of an array within a loop.

In the following example, all references to R(IX(I)) occur atomically, thus eliminating the possibility that different iterations might try to modify the same element concurently.

```
      REAL R(200), S(1000)
      INTEGER IX(1000)
!HPF$ DISTRIBUTE R(BLOCK), S(BLOCK), IX(BLOCK)

!HPF$ INDEPENDENT (I) ON S(I)
      DO I = 1, 1000
!HPF$       ATOMIC UPDATE
```

```
          R(IX(I)) = R(IX(I)) + S(I)
      END DO
```

F.5.17 The GEOMETRY Directive

The `GEOMETRY` directive is simliar to a `typedef` in C, only it is for data mapping. It allows the user to conveniently change the mappings of many arrays at the same time. It is similar in many ways to the `TEMPLATE` directive, but since it is bound to no particular extent it is sometimes easier to apply.

`!HPF$ GEOMETRY` *geom*(*d_1* `[`, *d_2*, ..., *d_n*`]`)
`!HPF$ DISTRIBUTE (` *geom* `) [::]` *var_1*`[,` *var_2*`, ... ,` *var_m*`]`

Where d_i indicates one of the allowable distribution formats.

```
!HPF$ GEOMETRY GBB(BLOCK, CYCLIC)
      REAL A(300,300), B(400,400)
!HPF$ DISTRIBUTE (GBB) ::  A, B
!     if GBB  changes then both A  and B  change
```

Annex G

The FORTRAN 77 Local Library

The HPF standard now describes an `EXTRINSIC(LANGUAGE='F77',MODEL='LOCAL')` interface, or `EXTRINSIC(F77_LOCAL)` to use the keyword identification (see Section 11.6 for its description), similar in characteristics to the `EXTRINSIC(LANGUAGE='HPF',MODEL='LOCAL')` and `EXTRINSIC(LANGUAGE='FORTRAN',MODEL='LOCAL')` interfaces. This section describes a set of library routines to make it easier to make use of the `F77_LOCAL` interface when passing distributed array data. These library routines can facilitate, for example, a portable blend of global data parallel code with preexisting FORTRAN 77-based code using explicit message passing calls for interprocessor communication. The FORTRAN 77 Local Library interface described in this section was originally developed as part of Thinking Machines TMHPF and is now supported by Sun Microsystems Inc. For suggestions, requests, or corrections concerning this interface, please contact

Sun Microsystems Inc.
High Performance Computing
M/S UCHLO5-104
5 Omni Way
Chelmsford, MA 01824
f77-local-library@sun.com

G.1 Introduction

The basic constraints for the local model (Section 11.1) together with the `F77_LOCAL`-specific argument passing options (Section 11.6) define the nature of the `F77_LOCAL` interface: how control is to be transferred from a global HPF procedure to a set of local procedures described by an `EXTRINSIC(F77_LOCAL)` procedure interface and how data can be passed between these two types of procedures: by reference or by descriptor, and with or without temporary local reordering of data to satisfy FORTRAN 77 provisions for sequential, contiguous storage of array data in Fortran array element order. These alternative methods of argument passing can be obtained by use of the two special-purpose attributes for extrinsic dummy arguments defined for `LANGUAGE='F77'` routines: `LAYOUT('F77_ARRAY')` (the default) vs. `LAYOUT('HPF_ARRAY')`, and `PASS_BY('*')` (the default) vs. `PASS_BY('HPF_HANDLE')`. However, to take advantage of the option allowing one to pass global HPF array "handles" to local FORTRAN 77 procedures and then obtain information locally about how the local portion of a given parallel array is actually distributed requires special inquiry routines comparable to the HPF Local Library of functions. Since this library is not only described as a module, but uses many features such as array-valued functions and optional arguments not available in FORTRAN 77 code, it is recommended that a modified FORTRAN 77 interface to this library be provided in the manner described below. Furthermore, there is the problem of describing local portions of parallel arrays in the FORTRAN 77 code used in each local routine called from a global HPF one. Since assumed-shape syntax may not be used, explicit shape arrays are required. But it is common for global distribution of arbitrary sized arrays to result in local portions of arrays that do not have constant shapes on all processors, and the actual extents in each processor cannot necessarily be predicted in advance. In order to allow programmers to obtain axis extent information at run time from the HPF global caller, a special HPF-callable subgrid inquiry subroutine is provided. A FORTRAN 77 callable version of the same routine is also described below, for flexibility in programming.

G.2 Summary

- One HPF-callable subgrid inquiry subroutine

  `HPF_SUBGRID_INFO`

- A set of FORTRAN 77-callable inquiry subroutines

  `F77_SUBGRID_INFO`
  `F77_GLOBAL_ALIGNMENT`
  `F77_GLOBAL_DISTRIBUTION`
  `F77_GLOBAL_TEMPLATE`
  `F77_ABSTRACT_TO_PHYSICAL`
  `F77_PHYSICAL_TO_ABSTRACT`
  `F77_LOCAL_TO_GLOBAL`
  `F77_GLOBAL_TO_LOCAL`
  `F77_LOCAL_BLKCNT`
  `F77_LOCAL_LINDEX`
  `F77_LOCAL_UINDEX`
  `F77_GLOBAL_SHAPE`
  `F77_GLOBAL_SIZE`
  `F77_SHAPE`
  `F77_SIZE`
  `F77_MY_PROCESSOR`

G.3 Global HPF Subgrid Inquiry Routine

The `F77_LOCAL` library interface includes only one global HPF subroutine, `HPF_SUBGRID_INFO`, whose implementation should be added as an extension to the standard HPF Library module. Its purpose is to provide per-processor information about the local subgrids of distributed arrays. This information is often critical when passing such arrays to local procedures written in FORTRAN 77, where array argument shapes must be stated explicitly in the local procedure (except in the last dimension; there are "assumed size" but no "assumed shape" arrays) , but may be expressed in terms of arguments passed at run time ("adjustable shape arrays"). Thus the subgrid parameters obtained from this subgrid inquiry routine can be passed as arguments to the local routines and used there to describe the extents of the locally visible portions of global HPF arrays, as the example in Section G.5 will demonstrates.

HPF_SUBGRID_INFO (ARRAY, IERR, DIM, LB, UB, STRIDE, LB_EMBED, UB_EMBED, AXIS_MAP)

Description. Gives local information about local subgrid allocation onto each processor of a distributed array; callable from a global HPF routine.

Class. Inquiry subroutine.

Arguments.

`ARRAY`	is a nonsequential array of any type, size, shape, or mapping. It is an `INTENT (IN)` argument.
`IERR`	is a scalar integer of default kind. It is an `INTENT (OUT)` argument. Its return value is zero upon successful return and nonzero otherwise. Errors result if local subgrids cannot be expressed as array sections of `ARRAY`. If any of the optional arguments `LB_EMBED`, `UB_EMBED`, or `AXIS_MAP` is present, then a nonzero value is also returned if the compiler does not organize the local data in serial memory by sequence associating a larger "embedding" array (see Section G.3.1 below for more explanation).
`DIM` (optional)	is a scalar integer of default kind. It is an `INTENT (IN)` argument. `DIM` indicates the axis along which return values are desired. If `DIM` is not present, values are returned for all axes.
`LB` (optional)	is an `INTENT (OUT)`, default integer array. If this argument is present, and if the value returned in `IERR` is zero, the values returned in array `LB` are the lower bounds in global coordinates of each processor's subgrid, along one (if `DIM` is present) or each dimension of `ARRAY`.
`UB` (optional)	is an `INTENT (OUT)`, default integer array. If this argument is present, and if the value returned in `IERR` is zero, the values returned in array `UB` are the upper bounds in global coordinates of each processor's subgrid, along one (if `DIM` is present) or each dimension of `ARRAY`.
`STRIDE` (optional)	is an `INTENT (OUT)`, default integer array. If this argument is present, and if the value returned in `IERR` is zero, the values returned in array `STRIDE` are the strides in local memory between elements of each processor's subgrid, along one (if `DIM` is present) or each dimension of `ARRAY`.

LB_EMBED (optional) is an INTENT (OUT), default integer array. If this argument is present, and if the value returned in IERR is zero, the values returned in array LB_EMBED are the lower bounds in global coordinates of the actual global array elements allocated on each processor, possibly a superset of the user-visible subgrid, along one (if DIM is present) or each dimension of ARRAY.

UB_EMBED (optional) is an INTENT (OUT), default integer array. If this argument is present, and if the value returned in IERR is zero, the values returned in array UB_EMBED are the upper bounds in global coordinates of the actual global array elements allocated on each processor, possibly a superset of the user-visible subgrid, along one (if DIM is present) or each dimension of ARRAY.

AXIS_MAP (optional) is a rank 2, INTENT (OUT), default integer array. If this argument is present, its shape must be at least [n,r], where n is the number of processors and r is the rank of ARRAY.

If the value returned in IERR is zero, the values returned in AXIS_MAP(i,1:r) represent the numbers of the axes of the subgrid on processor i from fastest varying to slowest varying, and form a permutation of the sequence 1,2,...,r.

For the last six arguments, LB, UB, STRIDE, LB_EMBED, UB_EMBED, and AXIS_MAP, each array has a first axis of extent at least n, where n is the number of processors, and the first n indices of that axis of each array must be distributed (perhaps via an explicit CYCLIC or BLOCK distribution) one index per processor. If a second dimension is needed, it should be a collapsed axis of extent at least equal to the rank of ARRAY.

If HPF_SUBGRID_INFO is called, and the elements of ARRAY that are local to any particular processor are not representable as an array section of the global user array, then a nonzero value is returned for IERR. Otherwise, if any of the optional arguments LB, UB, or STRIDE is present, then the lower bounds, upper bounds, or strides, respectively, that describe the local array sections are returned in terms of one-based, global coordinates.

G.3.1 Subgrid Inquiries Involving Embedding Arrays

In the common case in which the elements of each local subgrid of the global array argument are distributed across processors, with no overlap, and allocated in local memory like a local FORTRAN 77 array, as a contiguous sequence of elements in Fortran array element order, these three last optional arguments would not be required.

However, some implementations may choose less common layouts in local memory, that involve "embedding" these elements in a larger array section of equal rank that *is* sequence-associated in serial memory. For example, alignment of axes of arrays in different orders may result in a permuting embedding of the subgrid. Or axes of subgrids map be padded with ghost cells, either for stencil optimizations or to achieve same-size subgrids on all nodes.

In variations such as these, we may still view the subgrid as being "embedded" in a sequence associated array which may be accessible in F77_LOCAL operations, if the permutation of axes, shape of any embedding array, and offsets into that array can be obtained at runtime. The last three arguments of HPF_SUBGRID_INFO are provided to allow programmers to obtain this information when it is appropriate, with the help of the IERR flag to signal when this is not the case.

In this mapping, local memory has been allocated for a larger array section, with coordinates (LB_EMBED : UB_EMBED : STRIDE). The coordinates of the *actual* computational elements are limited to the subset (LB : UB : STRIDE). The sequence association is generalized to an arbitrary mapping of axes. Here, AXIS_MAP numbers the axes from fastest varying to slowest varying. If LB_EMBED, UB_EMBED, or AXIS_MAP is specified in a call to HPF_SUBGRID_INFO but ARRAY does not satisfy the assumptions of this mapping model, then a nonzero value is returned for IERR.

G.4 Local FORTRAN 77 Inquiry Routines

Here the F77-callable inquiry subroutines are described briefly. These provide essentially the same capability as the combination of the HPF intrinsic array inquiry functions such as SHAPE and SIZE, together with the HPF LOCAL LIBRARY inquiry routines. The subroutine F77_SUBGRID_INFO serves as a local counterpart to the globally callable subroutine HPF_SUBGRID_INFO described above. In all of the following:

- ARRAY is a dummy argument passed in from a global HPF caller using the LAYOUT ('HPF_ARRAY') attribute and declared within the FORTRAN 77 local subroutine as a scalar integer variable. It is an INTENT (IN) argument.
- DIM is a scalar integer of default kind. It is an INTENT (IN) argument. This argument specifies a particular axis of the global array associated with ARRAY or, if DIM = -1, inquiry is for all axes.
- An "inquiry result" is an INTENT (OUT) argument. If DIM = -1, it is a rank-one array of size equal to at least the rank of the global array associated with ARRAY, returning information associated with all axes. If DIM is positive, the "inquiry result" is a scalar, returning information only for the axis indicated by DIM.
- The arguments are defined in the same way as for the corresponding HPF or HPF_LOCAL routines unless otherwise noted. See the description of HPF_SUBGRID_INFO above and Section 11.7.1 for full specifications of the similarly-named HPF_LOCAL_LIBRARY procedures.

F77_SUBGRID_INFO (ARRAY, IERR1, IERR2, DIM, LB, UB, STRIDE, LB_EMBED, UB_EMBED, AXIS_MAP)

Description. This is a FORTRAN 77-callable version of the HPF subroutine HPF_SUBGRID_INFO.

Arguments.

IERR1	is a scalar integer of default kind. It is an INTENT (OUT) argument. Its return value is zero if LB, UB, and STRIDE were determined successfully and nonzero otherwise.

`IERR2`	is a scalar integer of default kind. It is an `INTENT (OUT)` argument. Its return value is zero if `LB_EMBED` and `UB_EMBED` were determined successfully and nonzero otherwise.
`LB, UB, STRIDE, LB_EMBED, UB_EMBED, AXIS_MAP`	are "inquiry results" of default integer type. They are the lower and upper bounds and strides of the array sections describing the local data (in terms of global indices), the lower and upper bounds of the embedding arrays (again, in terms of global indices), and the axes of the embedding arrays to which the axes of `ARRAY` are mapped.

F77_GLOBAL_ALIGNMENT (ALIGNEE, LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, DYNAMIC, NCOPIES)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` subroutine `GLOBAL_ALIGNMENT`. All but the first are `INTENT (OUT)` arguments whose return values are as specified by the corresponding HPF routine.

Arguments.

`ALIGNEE`	is a dummy argument passed in from global HPF. It is an `INTENT (IN)` argument.
`LB, UB, STRIDE, AXIS_MAP`	are integer arrays of rank one. Their size must be at least equal to the rank of the global HPF array associated with `ALIGNEE`.
`IDENTITY_MAP, DYNAMIC`	are scalar logicals.
`NCOPIES`	is a scalar integer of default kind.

F77_GLOBAL_DISTRIBUTION (DISTRIBUTEE, AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` subroutine `GLOBAL_DISTRIBUTION`. All but the first are `INTENT (OUT)` arguments whose return values are as specified by the corresponding HPF routine.

Arguments.

`DISTRIBUTEE`	is a dummy argument passed in from global HPF. It is an `INTENT (IN)` argument.
`AXIS_TYPE`	is a `CHARACTER*9` array of rank one. Its size must be at least equal to the rank of the global HPF array associated with `DISTRIBUTEE`.
`AXIS_INFO`	is a default integer array of rank one. Its size must be at least equal to the rank of the global HPF array associated with `DISTRIBUTEE`.
`PROCESSORS_RANK`	is a scalar of default integer type.

`PROCESSORS_SHAPE`	is an integer array of rank one. Its size must be at least equal to the value returned by `PROCESSORS_RANK`.

F77_GLOBAL_TEMPLATE (ALIGNEE, TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO, NUMBER_ALIGNED, DYNAMIC)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` subroutine `GLOBAL_TEMPLATE`. All but the first are `INTENT (OUT)` arguments whose return values are as specified by the corresponding HPF routine.

Arguments.

`ALIGNEE`	is a dummy argument passed in from global HPF. It is an `INTENT (IN)` argument.
`TEMPLATE_RANK`	is a scalar integer of default kind.
`LB, UB, AXIS_INFO`	are integer arrays of rank one. Their size must be at least equal to the rank of the align-target to which the global HPF array associated with `ALIGNEE` is ultimately aligned.
`AXIS_TYPE`	is a `CHARACTER*10` array of rank one. Its size must be at least equal to the rank of the align-target to which the global HPF array associated with `ALIGNEE` is ultimately aligned.
`NUMBER_ALIGNED`	is a scalar integer of default kind.
`DYNAMIC`	is a scalar logical.

F77_ABSTRACT_TO_PHYSICAL(ARRAY, INDEX, PROC)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` subroutine `ABSTRACT_TO_PHYSICAL`.

Arguments.

`INDEX`	is a rank-one, `INTENT (IN)`, integer array.
`PROC`	is a scalar, `INTENT (OUT)`, integer.

F77_PHYSICAL_TO_ABSTRACT(ARRAY, PROC, INDEX)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` subroutine `PHYSICAL_TO_ABSTRACT`.

Arguments.

`PROC`	is a scalar, `INTENT (IN)`, integer.
`INDEX`	is a rank-one, `INTENT (OUT)`, integer array.

F77_LOCAL_TO_GLOBAL(ARRAY, L_INDEX, G_INDEX)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` subroutine `LOCAL_TO_GLOBAL`.

Arguments.

`L_INDEX`	is a rank-one, `INTENT (IN)`, integer array.
`G_INDEX`	is a rank-one, `INTENT (OUT)`, integer array.

F77_GLOBAL_TO_LOCAL(ARRAY, G_INDEX, L_INDEX, LOCAL, NCOPIES, PROCS)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` subroutine `GLOBAL_TO_LOCAL`.

Arguments.

`G_INDEX`	is a rank-one, `INTENT (IN)`, integer array.
`L_INDEX`	is a rank-one, `INTENT (OUT)`, integer array.
`LOCAL`	is a scalar, `INTENT (OUT)`, logical.
`NCOPIES`	is a scalar, `INTENT (OUT)`, integer.
`PROCS`	is a rank-one, integer array whose size is at least the number of processors that hold copies of the identified element.

F77_LOCAL_BLKCNT(L_BLKCNT, ARRAY, DIM, PROC)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` function `LOCAL_BLKCNT`.

Arguments.

`L_BLKCNT`	is an “inquiry result” of type integer.
`PROC`	is a scalar integer of default kind. It must be a valid processor number or, if `PROC = -1`, the value returned by `F77_MY_PROCESSOR()` is implied.

F77_LOCAL_LINDEX(L_LINDEX, ARRAY, DIM, PROC)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` function `LOCAL_LINDEX`.

Arguments.

`L_LINDEX`	is a rank-one, integer array of size equal to at least the value returned by `F77_LOCAL_BLKCNT`.
`DIM`	may not be -1.
`PROC`	is a scalar integer of default kind. It must be a valid processor number or, if `PROC = -1`, the value returned by `F77_MY_PROCESSOR()` is implied.

F77_LOCAL_UINDEX(L_UINDEX, ARRAY, DIM, PROC)

Description. This is a FORTRAN 77-callable version of the HPF_LOCAL function LOCAL_UINDEX.

Arguments.

L_UINDEX	is a rank-one, integer array of size equal to at least the value returned by F77_LOCAL_BLKCNT.
DIM	may not be -1.
PROC	is a scalar integer of default kind. It must be a valid processor number or, if PROC = -1, the value returned by F77_MY_PROCESSOR() is implied.

F77_GLOBAL_SHAPE(SHAPE, ARRAY)

Description. This is a FORTRAN 77-callable version of the HPF_LOCAL function GLOBAL_SHAPE.

Arguments.

SHAPE	is a rank-one, integer array of size equal to at least the rank of the global array associated with ARRAY. Its return value is the shape of that global array.

F77_GLOBAL_SIZE(SIZE, ARRAY, DIM)

Description. This is a FORTRAN 77-callable version of the HPF_LOCAL function GLOBAL_SIZE.

Arguments.

SIZE	is a scalar integer equal to the extent of axis DIM of the global array associated with ARRAY or, if DIM = -1, the total number of elements in that global array.

F77_SHAPE(SHAPE, ARRAY)

Description. This is a FORTRAN 77-callable version of the HPF intrinsic SHAPE, as it would behave as called from HPF_LOCAL.

Arguments.

SHAPE	is a rank-one, integer array of size equal to at least the rank of the subgrid associated with ARRAY. Its return value is the shape of that subgrid.

F77_SIZE(SIZE, ARRAY, DIM)

Description. This is a FORTRAN 77-callable version of the HPF intrinsic `SIZE`, as it would behave as called from `HPF_LOCAL`.

Arguments.

`SIZE`	is a scalar integer equal to the extent of axis `DIM` of the subgrid associated with `ARRAY` or, if `DIM = -1`, the total number of elements in that subgrid.

F77_MY_PROCESSOR(MY_PROC)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` function `MY_PROCESSOR`.

Arguments.

`MY_PROC`	is a scalar, `INTENT (OUT)`, integer. Its value is the identifying number of the physical processor from which this call is made.

G.5 Programming Example Using HPF_SUBGRID_INFO

G.5.1 HPF Caller

```
      PROGRAM EXAMPLE
! Declare the data array and a verification copy
      INTEGER, PARAMETER :: NX = 100, NY = 100
      REAL, DIMENSION(NX,NY) :: X, Y
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X, Y
! The global sum will be computed
! by forming partial sums on the processors
      REAL PARTIAL_SUM(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE PARTIAL_SUM(BLOCK)
! Local subgrid parameters are declared per processor
! for a rank-two array
      INTEGER, DIMENSION(NUMBER_OF_PROCESSORS(),2) ::
     & LB, UB, NUMBER
!HPF$ DISTRIBUTE(BLOCK,*) :: LB, UB, NUMBER
! Define interfaces
      INTERFACE
        EXTRINSIC(F77_LOCAL) SUBROUTINE LOCAL1
     &   ( LB1, UB1, LB2, UB2, NX, X )
! Arrays LB1, UB1, LB2, UB2, and X are passed by default
! as LAYOUT('F77_ARRAY') and PASS_BY('*')
        INTEGER, DIMENSION(:) :: LB1, UB1, LB2, UB2
        INTEGER NX
        REAL X(:,:)
!HPF$ DISTRIBUTE(BLOCK) :: LB1, UB1, LB2, UB2
```

```
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X
        END
        EXTRINSIC(F77_LOCAL) SUBROUTINE LOCAL2(N,X,R)
! Arrays N, X, and R are passed by default
! as LAYOUT('F77_ARRAY') and PASS_BY('*')
        INTEGER N(:)
        REAL X(:,:), R(:)
!HPF$ DISTRIBUTE N(BLOCK)
!HPF$ DISTRIBUTE X(BLOCK,BLOCK)
!HPF$ DISTRIBUTE R(BLOCK)
        END
      END INTERFACE

! Determine result using only global HPF
      ! Initialize values
      FORALL (I=1:NX,J=1:NY) X(I,J) = I + (J-1) * NX
      ! Determine and report global sum
      PRINT *, 'GLOBAL HPF RESULT: ',SUM(X)
! Determine result using local subroutines
      ! Initialize values ( assume stride = 1 )
      CALL HPF_SUBGRID_INFO( Y, IERR, LB=LB, UB=UB )
      IF (IERR.NE.0) STOP 'ERROR!'
      CALL LOCAL1( LB(:,1), UB(:,1), LB(:,2), UB(:,2), NX, Y )
      ! Determine and report global sum
      NUMBER = UB - LB + 1
      CALL LOCAL2 ( NUMBER(:,1) * NUMBER(:,2) , Y , PARTIAL_SUM )
      PRINT *, 'F77_LOCAL RESULT #1 : ',SUM(PARTIAL_SUM)
      END
```

G.5.2 FORTRAN 77 Callee

```
      SUBROUTINE LOCAL1( LB1, UB1, LB2, UB2, NX, X )
! The global actual arguments passed to LB1, UB1, LB2, and UB2
! have only one element apiece and so can be treated as scalars
! in the local Fortran 77 procedures
      INTEGER LB1, UB1, LB2, UB2
! NX contains the global extent of the first dimension
! of the global array associated with local array X
      INTEGER NX
! Note that X may have no local elements.
      REAL X ( LB1 : UB1 , LB2 : UB2 )
! Initialize the elements of the array, if any
      DO J = LB2, UB2
        DO I = LB2, UB2
          X(I,J) = I + (J-1) * NX
        END DO
      END DO
      END
      SUBROUTINE LOCAL2(N,X,R)
! Here, the rank of the original array is unimportant
! Only the total number of local elements is needed
!     INTEGER N
      REAL X(N), R
! If N is zero, local array X has no elements, but R
! still computes the correct local sum
      R = 0.
      DO I = 1, N
        R = R + X(I)
      END DO
      END
```

G.6 Programming Example Using F77-Callable Inquiry Subroutines

This example performs only the initialization of the above example. It illustrates use of the F77-callable inquiry routines on descriptors passed from HPF, as well as the addressing of uncompressed local subgrid data in terms of "embedding arrays."

G.6.1 HPF Caller

```
      PROGRAM EXAMPLE
      INTEGER, PARAMETER :: NX = 100, NY = 100
      REAL, DIMENSION(NX,NY) :: X
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X
! Local subgrid parameters are declared per processor
! for a rank-two array
      INTEGER, DIMENSION(NUMBER_OF_PROCESSORS(),2) ::
     & LB, UB, LB_EMBED, UB_EMBED
!HPF$ DISTRIBUTE(BLOCK,*) :: LB, UB, LB_EMBED, UB_EMBED
! Define interfaces
      INTERFACE
        EXTRINSIC(F77_LOCAL) SUBROUTINE LOCAL1(
     &   LB1, UB1, LB_EMBED1, UB_EMBED1,
     &   LB2, UB2, LB_EMBED2, UB_EMBED2, X, X_DESC )
        INTEGER, DIMENSION(:) ::
     &   LB1, UB1, LB_EMBED1, UB_EMBED1,
     &   LB2, UB2, LB_EMBED2, UB_EMBED2
! X is passed twice, both times without local reordering.
! First, it is passed by reference for accessing array elements.
        REAL, DIMENSION(:,:), LAYOUT('HPF_ARRAY'),
     &          PASS_BY('*')                  :: X
! It is also passed by descriptor for use in F77 LOCAL
! LIBRARY subroutines only.
        REAL, DIMENSION(:,:), LAYOUT('HPF_ARRAY'),
     &          PASS_BY('HPF_HANDLE')         :: X_DESC
!HPF$ DISTRIBUTE(BLOCK) :: LB1, UB1, LB_EMBED1, UB_EMBED1
```

```
!HPF$ DISTRIBUTE(BLOCK) :: LB2, UB2, LB_EMBED2, UB_EMBED2
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X
          END
        END INTERFACE
! Initialize values
! ( Assume stride = 1 and no axis permutation )
        CALL HPF_SUBGRID_INFO( X, IERR,
       & LB=LB, LB_EMBED=LB_EMBED,
       & UB=UB, UB_EMBED=UB_EMBED)
        IF (IERR.NE.0) STOP 'ERROR!'
        CALL LOCAL1(
       & LB(:,1), UB(:,1), LB_EMBED(:,1), UB_EMBED(:,1),
       & LB(:,2), UB(:,2), LB_EMBED(:,2), UB_EMBED(:,2), X, X )
        END
```

G.6.2 FORTRAN 77 Callee

```
        SUBROUTINE LOCAL1(
       & LB1, UB1, LB_EMBED1, UB_EMBED1,
       & LB2, UB2, LB_EMBED2, UB_EMBED2, X, X_DESC )
        INTEGER LB1, UB1, LB_EMBED1, UB_EMBED1
        INTEGER LB2, UB2, LB_EMBED2, UB_EMBED2
! The subgrid has been passed in its 'embedded' form
        REAL X ( LB_EMBED1 : UB_EMBED1 , LB_EMBED2 : UB_EMBED2 )
! Locally X_DESC is declared as an INTEGER
        INTEGER X_DESC
! Get the global extent of the first axis
! This is an HPF_LOCAL type of inquiry routine with an
! 'F77_' prefix
        CALL F77_GLOBAL_SIZE(NX,X,1)
! Otherwise, initialize elements of the array
! Loop only over actual array elements
        DO J = LB2, UB2
          DO I = LB2, UB2
            X(I,J) = I + (J-1) * NX
          END DO
        END DO
        END
```