X3H5
Parallel Extensions
for Fortran
April 2, 1993

Document Number
X3H5/93-SD2-Revision A

Technical Committee X3H5/93-SD2-Revision A

for Parallel Computation-

Parallel Fortran-Standard

TC X3H5/92-SD2-Revision A

1.0 Introduction

This standard defines parallel language extensions for Fortran. All of the extensions are designed to feel Fortran-like to the programmer to be consistent with the X3H5 Language Independent Model for Parallel Computation (X3H5/93-SD1-Revision A).

Wherever possible, the X3H5 extensions are described in terms of those entities which are imported via a MODULE (TYPE definitions, FUNCTIONS, and SUBROUTINES). There is no presumption that this is, in fact, how they shall be implemented.

Where the gain in functionality is sufficiently meritorious, the extensions are additions to the syntax definition of Fortran. When the X3H5 module is not used, a conformant implementation need not accept these syntax extensions.

1.1 Conceptual Model of Fortran Program Execution

A parallel program written using the ANSI X3H5 Fortran Language (ANSI X3H5 FL), begins execution in the Fortran main program as it would for an ordinary Fortran program. The initial process as defined in the ANSI X3H5 language Independent Model, begins execution of the main program. Execution proceeds as it would for a serial program until a parallel construct is encountered. A parallel construct is defined by PARALLEL and END PARALLEL statements. A worksharing construct is defined by PDO and END PDO or PSECTION and END PSECTION statements.

The following statement combinations define both a parallel construct and a worksharing construct: PARALLEL PDO and END PARALLEL PDO; PARALLEL SECTION and END PARALLEL SECTION.

Implicit synchronizations occur at: PARALLEL, END PARALLEL, PARALLEL PDO, END PARALLEL PDO, PARALLEL SECTIONS and END PARALLEL SECTIONS.

A group construct is defined by PGROUP and END PGROUP statements.

1.2 Pseudo Code Form of the Conceptual Model

The following is a pseudo code skeleton of a parallel program that uses the constructs described herein.

```
program main
  ! only the initial process is active here
  ! serial execution occurs here
parallel
  ! each team member performs the same actions
  pdo i=1,n,1          ! beginning of worksharing construct
```

```
        ! iterative work is distributed among team members
          ...
      end pdo               ! end of worksharing construct
         ...
      group                 ! beginning of group construct
        ! replicated code here executed by all team members
          ...
          psection          ! beginning of worksharing construct
          ...
          psection
          ...
          end psection      ! end of worksharing construct
      end  group            ! end of group construct
      parallel do j=1,m,1   ! nested parallelism
         ...
      end parallel do       ! end of nested parallelism
    end parallel            ! end of parallel construct
    ! serial program execution
    ! possibly more parallel constructs
     ...
    end
```

2.0 Standard Compliance

This standard describes all **standard conforming** programs. A program is standard conforming if it uses only those forms and relationships described in this standard and if that program has an interpretation according to this standard. A program unit is standard conforming if it can be included in a program in a manner that allows the program to be standard conforming.

A standard conforming implementation executes a standard conforming program in a manner that fulfills the interpretations prescribed by this standard. A standard conforming implementation may allow additional forms and relationships provided that such additions do not conflict with the standard forms and relationships. In order to avoid name space pollution, all standard conforming programs must contain a USE X3H5 statement. A standard conforming processor may ignore all X3H5 constructs when USE X3H5 is omitted.

3.0 Terminology and Basic Concepts

The first time a word or phrase with a special or restricted meaning is used in this document, it is boldfaced and defined. An example of this convention is the word, **Fortran**, (any dialect of ISO/IEC 1539:1991 (E) Fortran 90). All definitions are repeated in the glossary.

In describing the form of statements or constructs, or in explaining examples, the following metalanguage conventions and symbols are used. These are similar to those defined by Fortran 90 (S8 Version 118, X3.198-1991 American National Standard Fortran 90, ISO/IEC 1539:1991) on pages 3-5.

1. The courier type font, such as `ABCDEFGHIJKLMNOP`, are characters from the Fortran character set and are to be written as shown, except as otherwise noted.

2. A construct is referenced by capitalizing the first letter of the words that make up the construct name (e.g., the Parallel Do construct).

3. A statement is referenced by capitalizing all of the letters that make up the statement key words (e.g., the PARALLEL PDO statement).

4. Entities written in lower case italics, such as *name*, indicate general entities for which specific entities must be substituted in actual statements.

   Once a given *name* is used in a syntactic specification to represent an entity, all subsequent occurrences of that *name* represent the same entity, until that *name* is used in a subsequent syntactic specification to represent a different entity.

5. The entity *name-list* indicates a comma separated list of *name*. The entity *name-list* will not be further defined, but *name* will be.

6. Square brackets (i.e., "[ ]") are used to indicate optional items.

7. Ellipses (i.e., "...") are used to indicate that only an abbreviated form of a statement has been used, and that any form is allowed.

8. Blanks are used to improve readability, but unless otherwise noted, have no significance.

9. The entity *statements* indicates zero or more statements.

10. The entity *int-exp* represents an integer expression.

References to sections in this document consist of section number and section title (e.g., "2. Terminology and Basic Concepts").

4.0 Control Structures

4.1 Parallel Region Construct

The Parallel Region construct and associated grouping and worksharing constructs are all block structured constructs. All of the constructs follow the Fortran rules for block structured constructs.

4.1.1 Syntax for Parallel Regions Construct

A *parallel-region-construct* is:

```
[name:]     PARALLEL [(parallel-option)]
                  data-sharing-spec
                  parallel-body
            END PARALLEL [name]

where
            parallel-option is MAX PARALLEL = int-expr |
                               ORDERED                 |
                               MAX PARALLEL = int-expr, ORDERED |
                               ORDERED, MAX PARALLEL = int-expr
            parallel-body is   statements                |
                               parallel-construct
            parallel-construct is parallel-region-construct |
                               pdo-construct             |
                               psections-construct       |
                               group-construct           |
                               parallel-pdo-construct    |
                               parallel-psections-construct |
                               single-process-construct

Contstraint: If the parallel-construct has a name prefix, then the it must have
the same name as a suffix.
```

4.1.2 Interpretation

The Parallel Region construct is used to specify parallel execution of a block of code. The process that executes the PARALLEL statement becomes the base process. The processes that enter the Parallel Region construct are those on the team.

If the MAX PARALLEL qualifier is not specified on the PARALLEL statement, then the number of processes on this team is limited only by the maximum number of processes available to the program. (See the intrinsic function NPSAVL)

If the MAX PARALLEL qualifier is specified on the PARALLEL statement, then the number of processes on this team is limited by the iexp1.

All code inside a Parallel Region that is not enclosed by a worksharing construct shall be redundantly executed by all of the processes on the team.

If one or more processes execute a statement that causes a transfer of control out of the block defined by the parallel region, then the program is not standard conforming. Worksharing constructs are used to identify work that is to be spread among all of the processes on the team that encounter the worksharing construct.

4.2 Work Sharing Constructs

Worksharing constructs define units of work that shall bedistributed among the team within a parallel region. Work sharing constructs may be coded outside of the lexical scope of a parallel region. However, if parallel performance is to be achieved, a worksharing construct should be encountered within a parallel region construct. Inside a worksharing construct, no new parallelism shal begin unless a parallel construct is encountered to signal the formation of a new team. Unless it is enclosed in an intervening parallel construct, the innermost of two nested worksharing constructs shall be executed solely by the process that encounters it, even if idle team members are available.

4.2.1 PDO Construct

PDO is an iterative worksharing construct as described in the LIM.

4.2.1.1 Syntax for the PDO Construct

```
[name:]     PDO [(parallel-options)]
                    parallel-body
            END PDO [name]
```

4.2.1.2 Coding Rules

4.2.1.3 Interpretation

If the MAX PARALLEL qualifier is not specified on a PDO or PSECTIONS statement, then the number of processes on this team that may enter the worksharing construct is limited only by the number of processes on the team. (See the intrinsic function NPSTM (what is the new name for NPSTM?))

If the MAX PARALLEL qualifier is specified on PDO or PSECTIONS statement, then the number of processes on this team that may enter the worksharing construct is limited by the iexp2.

A Pdo construct may be executed by a single process. A process executes multiple units of parallel work from a Pdo construct as specified by the the Language Independent Model for Parallel Computation. For example it must:

1. for each unit of parallel work to be executed:
   a. assign the appropriate value to its index variable
   b. execute the iterative portion
   c. if EXTEND is specified, execute the statements up to the END .* EXTEND statement

2. make all shared objects updated by this process within the Pdo and the group block available to all processes

3. wait for all processes that participated in executing the Pdo to complete step 2)

The value of the loop index of a Parallel Do construct is undefined outside the scope of the Parallel Do construct. The value of a loop index contained within a parallel construct is undefined outside the scope of the enclosing parallel construct. The value of the index of an implied DO contained within a parallel construct is undefined outside the scope of the enclosing parallel construct.

4.2.1.4 PARALLEL PDO

The PARALLEL PDO construct is a combined parallel construct and worksharing construct and has the same meaning as

PARALLEL
PDO

4.2.1.4 Parallel PDO Syntax

The syntax for the PARALLEL PDO is:

```
[name:]     PARALLEL PDO iter-specification parallel-option-list
                  data-sharing-spec
                  parallel-body
            END PARALLEL PDO [name]
```

4.2.1.6 Examples

<u>Example</u>

```
      SUBROUTINE EX48 (A,B,C,N)
      REAL A(N),B(N),C(N)
      PARALLEL PDO I=1,N-1
        NEW T
        T = A(I)*B(I)
        C(I+1) = T * (T-1.0)
      END PARALLEL PDO
      END
```

Example

```
      SUBROUTINE EX49 (A,B,C,N)
      REAL A(N),B(N),C(N)
      PARALLEL
        NEW T
        PDO I=1,N-1
           T = A(I)*B(I)
           C(I+1) = T * (T-1.0)
        END PDO
      END PARALLEL
      END
```

Example ? shows the Parallel Region equivalent form of the Parallel Do construct shown in Example ?. Examples ? and ? compute the same results and exhibit the same amount of parallelism.

Example 50

```
      SUBROUTINE EX50 (ZA,ZB,ZC,ZD,N)
      REAL ZA(N),ZB(N),ZC(N),ZD(N)
      PARALLEL SECTIONS
        NEW T
        SECTION
           DO 10 I=1,N
              T = ZFUNC(ZA(I))
              ZC(I) = T * T
10         END DO
        SECTION
           DO 20 I=1,N
              T = ZFUNC(ZB(I)-ZA(I))
              ZD(I) = T * T
20         END DO
      END PARALLEL SECTIONS
      END
```

Example 51

```
      SUBROUTINE EX51 (ZA,ZB,ZC,ZD,N)
      REAL ZA(N),ZB(N),ZC(N),ZD(N)
      PARALLEL
        NEW T
        PSECTIONS
        SECTION
           DO 10 I=1,N
             T = ZFUNC(ZA(I))
             ZC(I) = T * T
10         END DO
        SECTION
           NEW T
           DO 20 I=1,N
             T = ZFUNC(ZB(I)-ZA(I))
             ZD(I) = T * T
20         END DO
        END PSECTIONS
      END PARALLEL
      END
```

Example 51 shows the Parallel Region equivalent form of the Parallel Sections construct shown in Example 50. Examples 50 and 51 compute the same results and exhibit the same amount of parallelism.

Example 52

```
      SUBROUTINE EX52 (A)
      REAL A(*)
```

```
                   GETLOCK B
                   GUARDS B(SUM)
                   UNLOCK(B)
                   SUM=0.0
                   PARALLEL
                     NEW SUML
                     SUML = 0.0
                     GROUP
                     PDO          I=1,N
                        SUML = SUML + A(I)
                     END PDO
                        CRITICAL SECTION (B)
                          SUM = SUM + SUML
                        END CRITICAL SECTION (B)
                     END GROUP
                   END PARALLEL
                   END
```

Example 52 shows a typical method for computing a reduction on a machine with a relatively small number of processes. All of the processes initialize their new copy of SUML to zero, then sum up the elements of A that correspond to the iterations assigned to each process, then, without waiting for the other processes on the team, update the global SUM from their local sum (SUML). All of the processes on the team wait at the END GROUP statement before continuing.

```
Example 53         SUBROUTINE EX53 (A,B,C,D,N,M)
                   REAL A(N),B(N),C(N),D(N)
                   PARALLEL
                     PDO I=1,N
                        A(I) = B(I) * C(I)
                     END PDO
                     PDO I=1,M
                        D(I) = A(I) - C(I)
                     END PDO
                   END PARALLEL
                   END
```

Example 53 shows a typical method for reducing fork/join overhead by placing two adjacent parallel loops inside a single Parallel Region. Because GROUP is not coded, the team members wait at the end of the first Pdo construct for all of the work to be complete, and then begin working on the second Pdo construct.

```
Example 54         SUBROUTINE EX54 (A,C,N,M)
                   REAL A(N,0:M),C(N,M)
                   PARALLEL
                     DO 10 J=1,M
                        PDO I=1,N
                          A(I,J) = C(I,J)/A(I,J-1)
                        END PDO
             10      END DO
                   END PARALLEL
                   END
```

Example 54 shows a typical method for greatly reducing fork/join overhead by floating the Parallel Region outside of a serial loop.

4.2.2 PSECTION Construct

Psection is a non-iterative worksharing construct as described in the LIM.

4.2.2.1 Syntax for the PSECTION Construct

```
[name:]     PSECTION
                   sections
            END PSECTIONS [name]
```

where

sections is [*sections section]*

section is SECTION [name] [WAIT (name-list)]

parallel-region

4.2.2.2 Coding Rules for the PSECTION Construct

The Parallel Sections construct is a block structured construct. The SECTION statements mark the beginning of each block. The end of each block is delimited by either another SECTION statement or the END PARALLEL SECTIONS statement. The Parallel Sections construct follows all of the rules of Fortran block structured constructs.

The identifier used for a *section-name* is a seventh class of local names in the sense of Fortran page 18-2. This means that

> A *section-name* must be unique within a program unit (ISO/IEC 1539:1991 Section 2.2)
>
> *Section-names* share the single name space already shared by array, variable, constant, statement function, intrinsic function, and dummy procedure names

In a standard conforming program the WAIT clause shall only reference the *section-name* of a lexically preceding SECTION statement of the same Parallel Sections construct.

4.2.2.1 Interpretation

The Parallel Sections construct is used to specify parallel execution of the identified sections of code. Each section of code identified in a Parallel Sections construct is interpreted as a unit of work.

In a standard conforming program the sections of code shall be data independent, except where appropriate synchronization mechanisms are used.

A *section-name* is a label with no programmer-visible storage association.

A Psections construct may be executed by one or more processes. A process executes multiple units of parallel work from a Psections construct by performing this sequence:

1. for each unit of parallel work to be executed:
   a. if a WAIT clause is coded for this section, then wait until the sections indicated by the WAIT clause have completed execution
   b. execute the corresponding section of code

2. if the EXTEND qualifier is specified, execute the statements up to the END EXTEND statement

3. make all shared objects updated by this process within the Psections construct available to all processes

4. wait for all processes that participated in executing the Psections construct to arrive at step 2)

If the MAX PARALLEL qualifier is not specified on a PDO or PSECTIONS statement, then the number of processes on this team that may enter the worksharing construct is limited only by the number of processes on the team. (See the intrinsic function NPSTM (what is the new name for NPSTM?))

If the MAX PARALLEL qualifier is specified on PDO or PSECTIONS statement, then the number of processes on this team that may enter the worksharing construct is limited by the iexp2.

If one or more processes executes a statement that causes a transfer of control out of the blocks defined by the Parallel Sections construct, then the program is not standard conforming. <Do we need our CYCLE and EXIT words here?>

The WAIT clause specifies a partial ordering among the sections of code. All sections whose names are listed as *section-names* in the WAIT clause of a section must complete before that section can begin. The WAIT clause does not require use of the ORDERED qualifier.

The GUARDS clause shall only be specified on the SECTION statement if the WAIT clause is specified. The GUARDS clause explicitly identifies the names of objects that shall be made consistent for the process executing the waiting section.

The GUARDS clause explicitly identifies the objects that must be made consistent and removes a requirement for an implementation to make any other objects consistent at the point it is specified.

If the ORDERED qualifier is not specified, then, except for the partial ordering specified by WAIT clauses, the sections of code must be execution order independent. The implementation may assign the processes to sections of code in any order allowed by the partial ordering specified by the WAIT clauses.

If the ORDERED qualifier is specified, then synchronization mechanisms may be used that require some portion of an earlier (in lexical order) section to complete execution before some portion of a later section begins execution. While use of the ORDERED qualifier in a Parallel Sections construct that does not contain synchronization is standard conforming, it may incur a performance penalty on some implementations.

If the MAX PARALLEL qualifier is not specified, then the number of processes on this team is limited only by the number of Sections defined or the maximum number of processes available to the program. If the MAX PARALLEL qualifier is specified, then the number of processes on this team must be greater than zero and less than or equal to *int-exp*. Any lexically contained do loop index variables are treated as newly scoped objects for the parallel section. They inherit the same type as the objects of the same name outside of the parallel section. They have the automatic storage class and have no storage associations thru equivalence classes or common blocks.
There is an implicit synchronization at the end of a Parallel Sections construct.

4.2.3 PARALLEL PSECTIONS Construct

The PARALLEL PSECTIONS construct is a combination of the PARALLEL and PSECTIONS constructs.

4.2.3.1 Syntax

```
[name:]     PARALLEL PSECTIONS [parallel-options]
                    data-sharing-spec
                    sections
            END PARALLEL PSECTIONS [name]
```

4.2.4 PDONE

The PDONE statement shall be used to indicate early completion of work within a worksharing construct.

4.2.4.1 Explicit Syntax

PDONE

4.2.4.2 Coding Rules

The PDONE statement is an executable statement.

The PDONE statement shall occur lexically nested within a worksharing construct.

4.2.4.3 Interpretation

Coded directly inside of a worksharing construct, the PDONE statement is used to indicate that no more units of work need to be distributed. Any units of work that have been distributed shall be completed. A standard conforming implementation may complete all of the work specified by the worksharing construct even though a PDONE statement is encountered.

4.2.4.4 Examples

```
Subroutine EX58(x,y)
Double precision x(100),y(100)
parallel do i=1,100
  if (y(i) .eq. 0.0D0) then
     print*,i
     pdone
     cycle
   endif
   x(i)=1.0/y(i)
end parallel do
return
end
```

In example 58, a process that finds a 0 in Y will print the index and indicate that no more iterations need to be done. The other processes will complete execution of any iterations the have begun. The CYCLE statement must be specified if the iteration setting PDONE is to skip the rest of its current iteration.

4.3 GROUP Construct

The Group construct is a grouping construct. By default there is a barrier at the end of the Group construct. The barrier is removed by coding the NOWAIT option for the Group construct.

4.3.1 Syntax

```
[name:]     GROUP [(group-option)]
                   parallel-body
            END GROUP [name]
```

where

group-option is NOWAIT

4.3.2 Coding Rules

The Pdo, Psections, and Group constructs may be coded outside of the lexical scope of a parallel region. In addition, PDO and PSECTION may be coded outside of the lexical scope of an associated Group.

4.3.3 Examples

Example 52

```
SUBROUTINE EX52 (A)
REAL A(*)
GETLOCK B
GUARDS B(SUM)
UNLOCK(B)
SUM=0.0
PARALLEL
  NEW SUML
  SUML = 0.0
  GROUP
  PDO          I=1,N
     SUML = SUML + A(I)
  END PDO
     CRITICAL SECTION (B)
       SUM = SUM + SUML
     END CRITICAL SECTION (B)
  END GROUP
END PARALLEL
END
```

Example 52 shows a typical method for computing a reduction on a machine with a relatively small number of processes. All of the processes initialize their new copy of SUML to zero, then sum up the elements of A that correspond to the iterations assigned to each process, then, without waiting for the other processes on the team, update the global SUM from their local sum (SUML). All of the processes on the team wait at the END GROUP statement before continuing.

Example 53

```
SUBROUTINE EX53 (A,B,C,D,N,M)
REAL A(N),B(N),C(N),D(N)
PARALLEL
  PDO I=1,N
     A(I) = B(I) * C(I)
  END PDO
  PDO I=1,M
     D(I) = A(I) - C(I)
  END PDO
END PARALLEL
END
```

Example 53 shows a typical method for reducing fork/join overhead by placing two adjacent parallel loops inside a single Parallel Region. Because GROUP is not coded, the team members wait at the end of the first Pdo construct for all of the work to be complete, and then begin working on the second Pdo construct.

Example 54
```
      SUBROUTINE EX54 (A,C,N,M)
      REAL A(N,0:M),C(N,M)
      PARALLEL
        DO 10 J=1,M
           PDO I=1,N
             A(I,J) = C(I,J)/A(I,J-1)
           END PDO
10      END DO
      END PARALLEL
      END
```

Example 54 shows a typical method for greatly reducing fork/join overhead by floating the Parallel Region outside of a serial loop.

4.4 Single Process Section

When executing inside a Parallel Region construct, it is often convenient to use a single process to update objects that are shared among the team. The Single Process construct is a worksharing construct with exactly one unit of work.

4.4.1 Explicit Syntax

Statement Forms
```
SINGLE PROCESS

     END SINGLE PROCESS
```

Structured As
```
SINGLE PROCESS
    statements
END SINGLE PROCESS
```

4.4.2 Explicit Syntax

The Single Process construct follows all of the rules of Fortran block structured constructs.

4.4.3. Interpretation

A block of code surrounded by a Single Process construct is executed by exactly one process of a team per encounter.

Example 55
```
      SUBROUTINE EX55 (A,B,N)
      REAL A(N),B(N)
      PARALLEL
        PDO I=1,N
           A(I) = 1.0 / A(I)
        END PDO
        SINGLE PROCESS
           IF ( A(1) .GT. 1.0 ) A(1) = 1.0
        END SINGLE PROCESS
```

```
  PDO I=1,N
     B(I) = B(I) / A(1)
  END PDO
END PARALLEL
END
```

Example 56

```
SUBROUTINE EX56 (A,B,N)
REAL A(N),B(N)
PARALLEL
  PDO I=1,N
     A(I) = 1.0 / A(I)
  END PDO
  PSECTIONS
  SECTION
     IF ( A(1) .GT. 1.0 ) A(1) = 1.0
  END PSECTIONS
  PDO I=1,N
     B(I) = B(I) / A(1)
  END PDO
END PARALLEL
END
```

Example 56 illustrates the equivalence between a worksharing construct with a single unit of work and a Single Process construct demonstrated in Example 55. Examples 55 and 56 produce the same results and exhibit the same degree of parallelism.

Example 57

```
SUBROUTINE EX57 (A,AMAX,N)
REAL A(0:N)
AMAX = 0.0
PARALLEL
  NEW ALMAX
  BEGIN GROUP
  PDO I=1,N
     IF ( ABS(A(I)) .GT. ABS(ALMAX) ) ALMAX = A(I)
  END PDO
     CRITICAL SECTION
       IF ( ABS(ALMAX) .GT. ABS(AMAX) ) AMAX = ALMAX
     END CRITICAL SECTION
  END GROUP
  SINGLE PROCESS
     ALMAX = A(1)+A(N)
     IF ( AMAX .LT. ALMAX ) AMAX = 1.0 + AMAX
  END SINGLE PROCESS
  PDO I=1,N
     A(I) = ABS( A(I) / AMAX )
  END PDO
END PARALLEL
END
```

In Example 57, after the maximum absolute value of an array is computed by the first Pdo construct, a single process performs some manipulation of the maximum value prior to its use in the final Pdo construct. Because AMAX is a shared variable being updated within a Parallel Region construct, but outside of a worksharing construct, some synchronization mechanism must be employed to ensure that only one process performs the update.

4.5 Inquiry Functions

The following intrinsic functions shall be provided:

4.5.1 Maximum peformance improvement at this time

```
DOUBLE PRECISION FUNCTION PERFMAX()
```

Returns an implementation dependent run-time measurement that indicates the maximum improvement in performance the program could reasonabley expect to achieve as described in the ANSI X3H5 LIM.

4.5.2 Team size

```
INTEGER FUNCTION NPTEAM()
```

Returns the number of processes (active and blocked) on the team for the current parallel construct.

4.5.3 Looking for work

```
INTEGER FUNCTION NPLOOK()
```

Returns the number of processes that are currently looking for work as defined in the ANSI X3H5 LIM.

4.5.4 Blocked processes

```
INTEGER FUNCTION NPBLOCK()
```

Returns the number of processes that are currently blocked as defined in the ANSI X3H5 LIM.

4.5.5 Active processes

```
INTEGER FUNCTION NPACTIVE()
```

Returns the number of processes that are currently active as defined in the ANSI X3H5 LIM.

5.0 Data Environments

This section describes the ***data environments*** of ***processes*** in a parallel *Fortran 90* program.

5.1 Terminology

5.1.1 The model terminology mapped to Fortran

5.1.1.1 Object

An ***object*** as described by the ***the model*** is a *Fortran data object*[1] *(constant, variable or subobject),* or a *Fortran common block*[2].

Composite objects are variables that are *Fortran arrays* and *Fortran structures (or derived data types)* ; and *Fortran common blocks*.

5.1.1.2 Read/Modify

An ***object*** or a *subobject of the object* is ***read*** as described by the ***the model*** when it is *referenced*[3] as described by *Fortran 90*.

An ***object*** or a *subobject of the object* is ***modified*** as described by ***the model*** when it is used in a way that causes it to *become defined* as described by Fortran 90[4]. A *Fortran constant* cannot be modified[5].

5.1.1.3 Data environment

---

[1]Fortran data object Section 2.4.3.1, page 13, line 39 of Fortran 90. A Fortran structure is a variable. Fortran structure Section 5.1.1.7, page 43, line 24 of Fortran 90.

[2]Fortran common block Section 5.5.2, page 58, line 18 of Fortran 90.

[3]referenced Section 2.5.5, lines 20-26; and Section 6, page 61 lines 3,4.

[4]defines Section 14.7.5, page 250, lines 4-10.

[5]*Fortran constant* Section 6, page 61, line 37, 38.

A ***data environment*** as described by the ***the model*** is a collection of ***objects*** as defined in section 5.1.1.1. (***Data enviroment*** as used in this document is distiguished from *data environment* as used in Fortran 90[6] by the inclusion of common blocks.)

5.1.1.4 Private/Shared

An object that has a P/S attribute of ***private*** for a parallel construct shall be part of only one team member's ***data environment***. (Note that *Fortran 90* uses the adjective private for access attributes also. This is distinct from P/S attributes.)

An object that has a P/S attribute of ***shared*** for a parallel construct shall be part of all team members' ***data environments*** for that parallel construct.

5.1.2 Fortran terminology extended for the model:

5.1.2.1 Scoping Unit

A **scoping unit** in the binding is a *Fortran scoping unit*[7] augmented to include a ***parallel construct***.

5.1.2.2 Instance of a subprogram

An **instance of a subprogram** is restricted to a single process as defined in section ??? of ***model*** document. The application of this statement modifies the *Fortran 90* definition in the following way: :h5.

(NOTE - ??? was to be added to model document as of 3/93 meeting, but haven't seen latest copy to get correct reference.)

An **instance of a subprogram** in the binding is defined with respect to a ***process***. When a function or subroutine defined by a subprogram is invoked, an instance of that subprogram is created **for the invoking process**. **Multiple instances of a subprogram may be active concurrently. A process's instance of a subprogram is independent of all other processes' instances of the subprogram.**

Each instance has an independent sequence of execution and an independent set of dummy arguments and local nonsaved data objects. If an internal procedure or statement function contained in the subprogram is invoked directly from an instance of the subprogram or from an internal procedure or statement function that has access to the entities of that instance, the created

[6]Section 2.4, Data Concepts, page 13, line 2.

[7]Section 2.2, page 9, lines 44-49 and Section 14, page 241, lines 3,4.

instance of the internal procedure or statement function also has access to the entities of that instance of the host subprogram.

All other **data** *entities are shared by all instances of the subprogram* **within a process**. *For example, the value of a saved data object appearing in one instance may have been defined in a previous instance* **within the process** *or by initialization in a DATA statement or type declaration statement*.[8]

The definition of the **save attribute** is restricted to a single process as defined in section ??? of ***model*** document. The application of this statement modifies the *Fortran 90* definition in the following way: (NOTE - ??? was to be added to model document as of 3/93 meeting, but haven't seen latest copy to get correct reference.)

Objects declared with the SAVE attribute in the scoping unit of a subprogram are shared by all instances **in a process** *of the subprogram*.[9]

Items that receive the SAVE attribute implicitly shall be shared by all instances **in a process** of the subprogram.[10]

5.1.3 New terminology for the binding

5.1.3.1 Iterative Control Variables

Iterative control variables are defined to include *do-variables*, used in *loop control*[11], *implied do control*[12], and parallel loop control.[13]

5.1.3.3 Hidden

Hidden in this binding is used to clarify that a *private access attribute* is being referenced rather than a *private P/S attribute*.

---

[8] Section 12.5.2.4, Instances of a 5.1.2.3 Save Attribute.

[9] Section 5.1.2.5, SAVE attribute, page 47, lines 37-38.

[10] Section 5.1, page 41, lines 9-12. Section 5.2.9, page 52, lines 1-3.

[11] Section 8.1.4.1.1, page 100, line 37.

[12] Section 9.4.2 (Data transfer input/output list), page 123, line 27.

[13] Section 4.5 (Construction of array values), page 37, line 40.

5.2 Allowable Parallel Access Attribute

All Fortran ***objects***, except *common* and *objects in common*, have an ***APA attribute*** of ***default private, explicitly shared***. ***Objects*** that are declared ***default private*** may be ***explicitly shared*** for a parallel construct if they are *host associated*[14] with a *scoping unit*[15] containing the parallel construct.

Common blocks and the ***objects*** contained in the *common block* have the same ***APA attribute***.

Modules and the ***objects*** defined by the *module* have the same ***APA attribute***.

The ***APA attribute*** of a common block or module is defined by the *instance attribute* specified in a Fortran program. If the *instance attribute* is *single* then the common block or module has an APA attribute of ***always shared***. Neither *common blocks* nor the ***objects*** contained in the *common blocks* shall be made ***private***. Similarly, neither *modules* nor the ***objects*** contained in the *module* shall be made ***private***.

If the *instance attribute* is *parallel* then the common block or module has an APA attribute of ***default private, explicitly shared***. ***Objects*** that are declared ***default private*** may be ***explicitly shared*** for a parallel construct if they are *host associated*[16] with a *scoping unit*[17] containing the parallel construct.

Objects declared within program units declared in modules follow the same rules as other program units.

5.2.1 Definition of Instance Attribute

An instance attribute for global data objects is defined. The instance attribute specifies whether there shall be a single instance of the global object for the entire parallel program or if there may be multiple parallel instances of the global object.

An instance attribute may only be specified for the following global entities: - common blocks - module program units.

The instance attribute shall be the same for all references to the global object throughout the program.

---

[14] Section 12.1.2.2.1, page 163, 164, lines 33-39, 1-33.

[15] Section 2.2, page 9, line 45-49.

[16] Section 12.1.2.2.1, page 163, 164, lines 33-39, 1-33.

[17] Section 2.2, page 9, lines 45-49.

All objects specified in a module program unit shall have the same instance attribute.

The default instance attribute for COMMON blocks shall be single.

Blank common shall only have an instance attribute of single.

The default instance attribute for modules shall be single.

A global object with an instance attribute of single shall have an APA attriute of "always shared".

A global object with an instance attribute of parallel shall have an APA attribue of "default private, explicitly shared".

A common block with a parallel instance attribute may have the save attribute. If it has the save attribut, it shall have the same lifetime as its data environment.

A common block with the parallel instance attribute may be initialized by a block data program. This shall occur once per process.

5.2.1.1 Instance Statement Syntax

```
 INSTANCE ( single or parallel)
   or
 INSTANCE ( single or parallel) list_of_common_block_names
   or
 INSTANCE ( single or parallel) module_name
```

An instance statement shall appear in the specification statements of a program unit.

If an INSTANCE statement occurs in a program unit without any names specified, then it shall define the instance attribute for all global objects in that program unit.

If an INSTANCE statement occurs in a module program unit, it shall specify only the name of the containing module program unit.

If an INSTANCE statement occurs in a main, subroutine or function, or block data program unit, it shall specify only names of common blocks defined within the program unit.

5.3 Private/Shared Attribute

When a parallel construct is encountered all ***objects*** that are ***read or modified*** within it shall have their ***P/S attribute*** determined as follows:

- All **iterative control variables** contained within the parallel construct shall have a ***P/S attribute*** of ***private*** with respect to the parallel construct.

- All ***objects*** that are *host associated* with a containing *scoping unit* shall have a ***P/S attribute*** of ***shared*** with respect to the parallel construct.

- All *common blocks* and ***objects*** contained in *common blocks* shall have a ***P/S attribute*** of ***shared*** with respect to the parallel construct.

- All ***objects*** that are declared within the *scope* of the parallel construct shall have a ***P/S attribute*** of ***private*** with respect to the parallel construct.

- All other ***objects*** shall have a ***P/S attribute*** of ***private*** with respect to the parallel construct.

All *Fortran 90 subobjects* of an ***object*** shall have the same ***P/S attribute*** as their containing ***object***.

5.3.1 References through Pointers

The P/S attribute of a pointer object will be used to determine synchronization requirements when the value of the pointer is referenced or modified. (Examples of modification include - allocate, deallocate, and pointer assignment.)

The P/S attribute of the target of a pointer shall be used to determine synchronization requirments when the value of the target is referenced or modified thru the pointer in addition to the pointer's sycnhronization requirements in determining the validity of the address.

A program shall not *assign* the value of a ***private*** *pointer* to a ***shared*** *pointer* if the *target of the pointer* is ***private*** and if the *target of the pointer* may be ***inaccessible*** when *referenced* with the ***shared*** *pointer*.

These rules are given as interpretations of the statement in the model document, Section 5.4 Basic Mechanics - paragraph discussion early departure of team members: "A team member shall not read or modify an object which is private to another member of the team."

5.4 Basic Mechanics

All ***objects*** in a parallel Fortran program shall be part of a ***data environment***.

5.4.1 Types of Data Environments

5.4.1.1 Initial Data Environment

The ***initial data environment*** for a parallel Fortran program shall begin with a ***new data environment***. In addition, the ***initial data environment*** contains all *common blocks and modules* for the Fortran program. During program execution, the ***initial data environment*** may contain additional ***objects*** that come into *scope* during program execution. ***Objects*** that come into *scope* during execution of parallel constructs shall not be part of the ***initial data environment*** unless the initial process is participating in the execution of the parallel construct as a base process and it encounters the scoping unit.

5.4.1.2 New Data Environment

A ***new data environment*** shall consist of ***objects*** with the *save attribute* (also referred to in *Fortran 90* as *saved objects*).[18] The ***objects*** that are *initially defined*[19] as described in *Fortran 90* shall have their initial values defined.

5.4.1.3 Looking for Work Data Environment

A ***looking for work data environment*** shall consist of ***objects*** with the *saved attribute* with the appropriate *association status, allocation status, definition status and value*[20] maintained from earlier participation in the execution of a parallel construct.

5.4.2 Data Environments upon encountering a parallel construct

When a parallel construct is encountered, the ***objects*** that are ***read*** or ***modified*** within it shall have their ***P/S attributes*** determined as specified in **section 5.3 Private/Shared Attribute**.

If the ***object*** is ***private*** or ***not available*** it shall not be part of the ***data environment*** of any member of the new team formed to execute the parallel construct.

If an ***object*** is classified as ***shared*** but another **instance of the object** is declared lexically within the parallel construct, then new ***private*** instances of the ***object*** shall be used by all team members. The base process shall not use the ***shared*** instance of the ***object*** if it participates in the execution of the parallel construct. (A ***shared object*** shall not be made ***private***.)

Only ***objects*** that are in *scope* at the time the parallel construct is ***encountered*** shall be ***shared*** for the parallel construct.

All other objects shall only be shared for a parallel construct if they are accessible and visible at the parallel construct.

---

[18] Section 5.1.2.5, SAVE attribute, page 47, lines 31-33.

[19] Section 14.7.3, Variables that are initially defined, page 249, lines 35-39.

[20] Section 5.1.2.5, SAVE Attribute, page 47, lines 31-33.

5.4.3 Object creation

Objects may be created when *program units* or **scoping units** are entered or when the ***objects*** are explicitly *allocated*.

When an ***object*** is created it is added to the ***data environment*** of the creating process. (Note that *Fortran 90* initialized *data objects* have the *save attribute* implied.[21] Since all *saved objects* are part of a ***new data environment***, all initialization of *data objects* has occurred.)

All ***objects*** shall have a ***P/S attribute*** determined when a parallel construct is encountered.

Objects with the *allocatable attribute* may be allocated prior to ***encountering*** a parallel construct for which their ***P/S attribute*** will be ***shared***. If an *allocatable* ***object*** is ***shared*** for a parallel construct and is to be allocated during the execution of a parallel construct, the program shall ensure the allocation is done with appropriate ***synchronization***.

5.4.4 Destroying Objects

Objects are ***destroyed*** as follows:

- *Data objects* without the *saved attribute* are destroyed when they exit the **scoping unit** for which they were created.
- *Data objects* with the *saved attribute* are destroyed when the ***data environment*** which they belong to is ***destroyed***.
- *Allocatable objects* are destroyed when they are *deallocated*.[22]
- Some *allocated* objects are destroyed when their scope is exited.[23]

5.4.5 Exiting parallel constructs

All ***objects*** without the *saved attribute* that were created for a **scoping unit** are destroyed upon exiting the **scoping unit**. If the **scoping unit** is contained within the parallel construct, then these **objects** shall not exist in the ***data environments*** of the processes exiting the parallel construct.

All ***objects without the saved attribute that were created for the scoping unit that is the parallel constructs are destroyed.***

---

[21] Section 5.2.9, page 52, lines 1-3.

[22] Section 6.3.3.1, Deallocation of allocatable arrays, page 69, lines 2-15.

[23] Section 6.3.3.1, Deallocation of allocatable arrays, page 69, lines 2-15.

An implementation may destroy objects with the saved attribute in a data environment only if all ***objects:ehp3 with the saved attribute for that data environment*** are destroyed. (If an object with a P/S attribute of ***private*** whose lifetime is longer than that of this parallel construct is destroyed, then all such objects shall be destroyed.)

5.4.6 Early Departures of Team Members

5.5 Binding Considerations

5.5.1 APA and P/S Attributes with Fortran Scoping Rules

Fortran 90 defines the following *scopes* for names: *global entities, local entities, statement entities*.[24] **The binding** provides the following ***APA attributes*** for these *scopes of named entities*:

- *global entities*
- ***always shared***
- ***default private, explicitly shared***
- *local entities*
- ***default private, explicitly shared***
- *statement entities*
- ***default private, explicitly shared***

The binding does not provide an option for the ***APA attributes*** of ***always private***.[25]

The binding does not provide an option for the ***APA attributes*** of ***default shared, explicitly private***.[26]

5.5.2 Data Environments and Lifetime of Fortran Objects

---

[24] Section 14, Scope, association and definition., page 241.

[25] Rationale - In order to facilitate the use of nested parallel constructs at any point in the parallel program. An implementation may map some ***objects*** to ***process private*** storage when those ***objects*** cannot be ***read*** or ***modified*** by other processes in a standard-conforming program. (Note: *Statement entities* will appear to be ***always private*** because in current binding there are no parallel constructs within a statement for which they could be explicitly shared.)

[26] Rationale - In order to restrict the "accidental sharing" of ***objects*** among parallel constructs. Programs shall explicitly identify ***objects*** to be shared at parallel constructs or shall explicitly identify ***objects*** to be always shared.

All *entities* that are *associated* shall have the same ***P/S attributes*** for a given parallel construct. *Association* may be by *name, argument, use, pointer or storage*.[27]

Lifetime of an ***object*** is tied to the lifetime of the ***data environment*** it belongs to. An ***object*** shall not exist before or after the ***data environment*** it belongs to.

Saved objects shall exist for the lifetime of a ***data environment***. *Saved objects* shall only be ***accessible*** by a process if the *saved object* is in *scope*.

Objects without the *saved attribute* may exist only when they are in *scope*. ***Objects*** without the *saved attribute* shall only be ***accessed*** when they are in *scope*.

An *allocatable* ***object*** shall only be ***accessed*** when its status is *allocated*.

An object with the *private (hidden) access* attribute within a given scope shall not be ***accessible***.

5.5.3 New Instances of Objects for Parallel Constructs

Objects declared within the **scope** of a parallel construct shall have a ***P/S attribute of private*** for that parallel construct.

The binding allows the following specifications within a parallel constructs:

5.5.3.1 Syntax

```
      data-sharing-spec is new-stmt |
                          use-stmt  |
                          type-declaration-stmt  |
                          specification-stmt     |
                          parameter-stmt         |
                          format-stmt            |
                          pointer-stmt
                           [data-sharing-spec]

      new-stmt is NEW variable-list
```

```
Constraint: specification-stmt shall not contain an access-stmt, common-stmt,
data-stmt, optional-stmt, equivalence-stmt, derived-type-stmt, or save-stmt.
```

5.5.3.2 Interpretation

The binding allows ***objects*** with the following *attributes* to be declared lexically within the **scope** of a parallel construct:

- *type*

---

[27] Section 14.6, Association, page 245-247.

- *dimension*
- *allocatable*
- *pointer*
- *target*

The following ***objects*** shall not be allowed to be specified lexically within the **scope** of a parallel construct:

- the declaration of an assumed size array, dummy argument common block, function or function entry point
- character type with an assumed length
- equivalence associated with any object that is shared for this parallel construct
- have the saved attribute
- be data initialized

The dimensionality of adjustable arrays inherited is that defined at the procedure entry for the corresponding adjustable array declarator.

5.5.3.3 New Statement

The NEW statement is defined to allow new instances of common blocks and modules with the parallel instance attribute to be created within a parallel construct.

5.5.3.3.1 NEW Statement Syntax

NEW *external_name_list*

where *external_name_list* - /<common_name >/ or <module_name>

Constraint: only common block names and module names that have the parallel instance attribute shall be specified on the NEW statement. A common block or module with an instance attribute of single shall not be specified on the NEW statement.

5.5.3.4 Iterative Control Variables

All **iterative control variables** defined by and within the parallel construct shall have a ***P/S attribute*** of ***private*** for the parallel construct and shall be exist only for the *scope* of the parallel construct. This shall occur even if the **iterative control variables** are not declared within the scope of the parallel construct. The *values* of the **iterative control variables** shall be *undefined* upon exit from the parallel construct. Only the *type attributes* of the **iterative control variables** shall apply within the **scope of a parallel construct**.

5.5.4 Alternative APA Attributes for Always Shared

Common blocks and the objects in common blocks that have an instance attribute of single shall have a ***P/S attribute*** of ***shared*** for all parallel constructs. *Modules and the objects in modules* that have an instance attribute of single shall have a ***P/S attribute*** of ***shared*** for all parallel constructs.

5.5.4 External Data Objects and Multiple Processes

Fortran 90 global named entities allow ***objects*** to be shared across *scoping units*. **The binding** provides the instance attribute as a mechanism of providing *global*; ***default private, explicitly shared objects***.

Additional rules with respect to new language features:

5.5.5.1 Common and Modules

A common block or module shall have a storage sequence whenever such a storage sequence would be required by *Fortran 90* for a common block regardless of its instance attribute.

Within a process, all program units access the same named common block and modules. The instance attribute of parallel provides a means of associating entities in different program units among a team of processes. It allows different teams of processes to have different storage associations for common blocks and modules There may be multiple common blocks or modules of the same name if they have the parallel instance attribute specified in a parallel program.)

When a parallel construct is encountered, three possibilities exist for common blocks and modules:

- shared - the common or module is lexically visible in the scoping unit containing the parallel construct and has an instance attribute of single or parallel.

All team members that participate in the execution of the parallel construct share access to the same common block/module that is lexically visible. Any modifications to that common block or module by any team member are retained and accessible after the parallel construct is exited.

- explicitly private - the common or module is specified on the NEW statement within the parallel construct and has an instance attribute of parallel

All team members that participate in the execution of the parallel construct access their own distinct storage sequence for the common block or module. The storage sequences for the common block or modules are not accessible outside of the scoping unit of the parallel construct.

- implicitly private - the common or module is not lexically visible in the scoping unit containing the parallel construct and is not specified within the parallel construct and has an instance attribute of parallel.

If the common block or module is referenced by a process executing the parallel construct, then the process references its private copy of the common block or module.

5.6 Objects and Synchronization

Between synchronization points, ***objects*** shall be ***read*** and ***modified*** as follows:

- ***read***

An ***object*** is ***read*** if it is *referenced as described by Fortran 90*[28]

- ***modified***

An ***object*** is ***modified*** if it an action occurs that causes it to *become defined*[29] or *become undefined* as described by *Fortran 90*[30]

Fortran 90 subobjects (array-element, array-section, structure-component, or substring)[31] are ***objects*** in ***the model*** and may be ***read*** and ***modified*** independently of other *subobjects* by different ***processes***. :efn.

In parallel programs, it is the users responsibility to protect ***shared objects*** in common with the proper synchronization if they are ***read and modified*** by multiple processes.

5.7 Examples

```
      Subroutine EXD01(A,B,C,N)
      Real A(n),B(n),C(n)
      parallel do i=1,n
         Real t
         t=a(i)*b(i)
         c(i+1)=t* (t-1.0)
      end parallel do
      end
```

In EXD01, the variable I has a P/S attribute of private for the parallel construct because it is the iterative control variable for the parallel do. The variable T has a P/S attribute of private

---

[28] Section 6, Use of Data Objects, page 61, lines 3-7.

[29] Section 14.7.5, Events that cause variables to become defined, page 250, 251, lines 3-42, 1-10.

[30] Section 14.7.6, Events that cause variables to become undefined, page 251, 252, lines 11-45, 1-33.

[31] Section 6, Use of Data objects, page 61, lines 16-19.

for the parallel do because it is declared within the parallel construct. The arrays A,B,C, and D are shared objects for the parallel construct. The variables I and T are undefined upon exit from the parallel do.

```
Subroutine EXD02(B)
Real, Dimension(100) :: B,C
parallel do i=1,100
   call subx1(b(i))
   call subx2(c(i))
end parallel do
print*, (c(i),i=1,100)
end

subroutine subx1(x)
real, save:: a
a=x
return
entry subx2(x)
x=a
end
```

In EXD02, the SAVE attribute ensures that the value of A defined by SUBX1 will be available for entry SUBX2 to use within any iteration of the parallel do construct. Thus, the effect of this example is to copy B to C and print the result. If the SAVE attribute was not specified, the results are undefined; (Note that if the parallel do was a serial do and the save attribute was not specified the results are also undefined.)

```
Subroutine EXD03()
Real, Dimension(100) :: B
common /abc/ b
call subx1(100)
print*, (b(i),i=1,100)
end

subroutine subx1(icnt)
parallel do i=1,icnt
   call work(i)
end parallel do
return
end

subroutine work(i)
Real, Dimension(100) :: B
common /abc/ b
b(i)=i
return
end
```

In EXD03, there is only one copy of the common block /abc/, that all processes share access to. The modifications made to the array elements, or subobjects, of b are data independent. No explicit synchronization is required.

```
subroutine EXD04(in,A)
real, dimension(in,in):: A
real, dimension(:,:), allocatable:: B,E
allocate B(in,in)
```

```
parallel do i=1,in
 real, dimension(:), allocatable:: C
 Allocate C(in)
 C(:)=0
 parallel do j=1,in
   c(j)=c(j)+A(i,j)
   if (fn(c(j)).neq.0) then
       Critical section
         if (.not.allocated(E)) then
            allocate E(in,in)
         endif
       end critical section
       E(i,j)=C(j)
   endif
 end parallel do
 B(i,:)=C(i)
 deallocate C
end do
A(:,:)=B(icnt:1:-1,:)
return
end
```

In EXD04, the allocateable array B is shared for both parallel constructs and the allocateable array C is private for the parallel do i loop but shared for the parallel do j loop. The allocateable array E is shared, but is only allocated based on a function of C(j). The user is responsible for providing the proper synchronization to ensure that only one team member allocates the shared array.

```
subroutine EXD06(in)
integer pi(in),i(in)
pointer pi
target i
allocate I
PI=>I
icnt=0
parallel
integer pj(in),j(in) ,id
pointer pj
target j
critical section
 icnt=icnt+1
 id=icnt
end critical section
 if(id .eq.1) then
  PJ=>I
 else
  allocate J
  PJ=>J
 endif
 pdo i=1,100
  ...

 end pdo
 if(id .gt.1) then
  deallocate j
 endif
end parallel
```

In EXD06, references with pointer PI in the parallel do loop will be appropriately synchronized among all processes executing the parallel construct. In this example, the user wants to use the allocated array I for the first process, and only allocate additional private arrays if additional processes execute part of the parallel construct. References with pointer PJ will be to objects with P/S attributes of shared or proivate; an implementation must ensure that the proper synchronization is done for the shared target.

```
subroutine exd1()
common/abc/a(100),b(100)
common/def/d(100),e(100)
common/ghi/g(100),h(100)
instance parallel /def/,/ghi/
parallel do i=1,100
  new/def/
 ...
end parallel do
```

```
subroutine exd2()
common/abc/a(100),b(100)
common/def/d(100),e(100)
common/ghi/g(100),h(100)
instance parallel /def/,/ghi/
instance single /abc/
parallel do i=1,100
  new/def/d(100),e(100)
 ...
end parallel do
```

In both examples exd1 and exd2, common /abc/ is always shared. There is only one copy for the entire program. All processes share the same copy. Common /def/ and /ghi/ are default private, explicitly shared. Since /def/ is specified within the parallel do construct, each team member participating in the execution of an iteration of the parallel do will have its own copy. The variables in common /def/ may be referenced without synchronization. Since /ghi/is visible at the parallel point it will be shared among all team members participating in the execution of the parallel construct.

```
module data
dimension a(100),b(100)
real a,b
private b
public a
end data
```

Module data will be an always shared global object. All team members of all teams will reference same A and B. Both A and B have an instance attribute of single and therefore have APA attributes of always shared. The Fortran access attribute of private (or hidden) does not affect the APA attribute.

```
module exd7
```

```
instance parallel
dimension a(100),b(100)
real a,b
end exd7
```

Module exd7 will be default private, explicitly shared. If the program unit containing a parallel construct has a use of exd7 then a and b will be shared for team members of that parallel construct. If not, then each team member will have private copies of the module exd7 created.

```
module data
instance parallel
dimension a(100),b(100)
real a,b
common/abc/a,b
subroutine x()
instance parallel
common/abc/a,b
dimension y(10)
...
end
end data
```

The common block /abc/ has a parallel instance attribute.

The reference to /abc/ within subroutine x must specify the same instance attribute for /abc/ as the containing module. The rules stated that objects defined within program units within modules would have their instance attribute determined based on the program unit rules. The object y is a local object to subroutine x - it does not have an instance attribute.

```
Example 33        LOGICAL FUNCTION EX33 (A,IZERO,N)
                  REAL A(N)
                    PARALLEL PDO I=1,N
                       IF ( A(N) .EQ. 0.0 ) THEN
                         CRITICAL SECTION
                            IZERO = I
                         END CRITICAL SECTION
                         EX33 = .TRUE.
                         PDONE
                       ENDIF
                    END PARALLEL PDO
                    EX33 = .FALSE.
                  END
```

Example 33 demonstrates how to carry the value of a new object out of a parallel construct. The loop index of the Parallel Do is new by default, so the loop index value is undefined outside of the scope of the Parallel Do. The Critical Section ensures that updating the global variable IZERO is performed by one process at a time. Note that this code does not ensure that the smallest index of a zero element of A is returned. Also, multiple processes may set IZERO.

Example 41

```
SUBROUTINE EX41 (B)
  REAL B(100)
  PARALLEL PDO I=1,100
    CALL SUB(B(I))
  END PARALLEL PDO
  END

  SUBROUTINE SUB (X)
  INSTANCE PARALLEL
  COMMON /BLOCKA/ A
  A = X
  CALL SQUARE
  X = A
  END

  SUBROUTINE SQUARE
  INSTANCE PARALLEL
  COMMON /BLOCKA/ A
  A = A*A
  END
```

Example 42

```
SUBROUTINE EX42 (B)
INSTANCE PARALLEL
COMMON /BLOCKA/ A
REAL B(100)
PARALLEL PDO I=1,100
  NEW /BLOCKA/
  CALL SUB(B(I))
END PARALLEL PDO
END

SUBROUTINE SUB (X)
INSTANCE PARALLEL
COMMON /BLOCKA/ A
A = X
CALL SQUARE
END

SUBROUTINE SQUARE
INSTANCE PARALLEL
COMMON /BLOCKA/ A
A = A*A
END
```

Example 42 and Example 41 provide the same results. Both ensure that within the parallel construct, team members have their own copies of common blocka for communication among program units within a process. Example 41 uses an implict private copy of blocka for the parallel construct.

Example 42 specifes an explicit private copy of blocka for the parallel construct.

Example 43:

```
C This example is NON-STANDARD CONFORMING C
      INSTANCE PARALLEL /NC/
      COMMON /NC/ A(100)
      ...
```

```
y_calls:          PARALLEL PDO I=1,100
                  ...
                  CALL Y
            10    END DO y_calls
                  ...
                  RETURN
                  END

                  SUBROUTINE Y
                  ...
                  PARALLEL PDO J=1,100
                    ...
                    CALL Z
                  END DO
                    ...
                    RETURN
                    END

                  SUBROUTINE Z
                  INSTANCE PARALLEL /NC/
                  COMMON /NC/ A(100)
                   ...
                  RETURN
                  END
```

In this example, the scommon block, NC, is shared for the parallel y_calls loop in the main program. However, NC, is implicitly private at the parallel do loop subroutine Y and is referenced within that parallel construct in subroutine Z.

Possible modifications to make it standard conforming include:

1. Specify /NC/ on a NEW statement in the parallel y_calls loop in the main program.

2. Include the COMMON statement defining /NC/ in subroutine Y. Then /NC/ will be shared for all parallel constructs.

3. Include the COMMON statement defining /NC/ in subroutine Y and specify /NC/ on a NEW statement for the parallel do loop.

```
Example 45        SUBROUTINE EX45 (B)
                  REAL B(100), C(100)
                  PARALLEL PDO I=1,100
                    CALL SUB1(B(I))
                    CALL SUB2(C(I))
                  END PARALLEL PDO
                  PRINT *, (C(I), I = 1, 100)
                  END

                  SUBROUTINE SUB1 (X)
                  INSTANCE PARALLEL
                  COMMON /BLOCKA/ A
```

```
                        SAVE /BLOCKA/
                        A= X
                        END

                        SUBROUTINE SUB2 (X)
                        INSTANCE PARALLEL
                        COMMON /BLOCKA/ A
                        SAVE /BLOCKA/
                        X = A
                        END
```

In Example 45, the SAVE statement ensures that the value of A defined SUB1 will be available for SUB2 to use within any iteration of the Parallel Do contruct. Thus, the effect of SC6 is to copy B to C and print the result. If the SAVE statement is not coded, the results are undefined. Note that without the SAVE statement, the serial form of this program would not conform to Fortran section 15.9.4.

```
      Example 46        SUBROUTINE EX46 (B)
                        REAL B(100), C(100)
                        INSTANCE PARALLEL /BLOCKA/
                        COMMON /BLOCKA/ A
                        PARALLEL PDO I=1,100
                          NEW /BLOCKA/
                          CALL SUB1(B(I))
                          CALL SUB2(C(I))
                        END PARALLEL PDO
                        PRINT *, (C(I), I = 1, 100)
                        END

                        SUBROUTINE SUB1 (X)
                        INSTANCE PARALLEL /BLOCKA/
                        COMMON /BLOCKA/ A
                        A = X
                        END

                        SUBROUTINE SUB2 (X)
                        INSTANCE PARALLEL /BLOCKA/
                        COMMON /BLOCKA/ A
                        X = A
                        END
```

Example 46 demonstrates an alternative to coding the SAVE statement. It is sufficient to declare /blocka/ in the calling program and code a NEW statement for /BLOCKA/ inside the parallel construct. Examples 45 and 46 both compute the same result.

```
      Example 39        SUBROUTINE EX39 (B,C,N)
                        REAL B(N),C(N)
                        PARALLEL PDO I=1,N
                          REAL A
                          A=B(I)+C(I)
                          CALL EX39A(A,B,I)
                        END PARALLEL PDO
                        END

                        SUBROUTINE EX39A (AA,BB,N)
```

```
      REAL BB(N),BX
      DATA BX/1.0/
      BX= AA * (AA-4.0)/BX
      PARALLEL PDO J=1,N
        BB(J) = BB(J)*BX
      END PARALLEL PDO
      END
```

In Example 39, the variable BX has a data sharing attribute of newfor the parallel do insubroutine EX39, but a shared data sharing attribute for the Parallel Doin subroutine EX39A.The DATA statement initializing BX applies on aper process basis. Thefirst time a process calls subroutine EX39A, the value of BX for thatprocess is guaranteed tobe that specified by the DATA statement. Subsequent calls of subroutineEX39A by the sameprocess use the value of BX from the end of the previous call to BX bythe same process.

```
Example ??        PROGRAM MAIN
            COMMON/COM1/CA(100)
            INTEGER LA,MS,ND
            DATA /ND,1/
            ...
            SAVE /COM1/,MS
            ...
            PARALLEL PDO I=1,100
              NEW LA,MS,ND
               ...
               CALL Y
               ...
            END Parallel DO
            ...
            END

            BLOCK DATA X
            COMMON/COM1/CA(100)
            INSTANCE PARALLEL /SCOM1/
            COMMON/SCOM1/ SC(100)
            DATA /CA,100*0.0/,/SC,100*0.0/
            END

            SUBROUTINE Y
            COMMON/COM1/CA(100)
            COMMON/COM2/CB(100)
            INSTANCE PARALLEL /SCOM1/,/SCOM2/
            COMMON/SCOM1/ SC(100)
            COMMON/SCOM2/ SD(100)
            INTEGER IS(100),JA(100),KD
            DATA KD/0/
            SAVE /COM1/,/SCOM1/,IS
            ...
            END

EXPLANATION
      COMMONs:
            COM1 is single_copy_external, static storage
            COM2 is single_copy_external, dynamic storage

            SCOM1 is parallel_external, static storage
            SCOM2 is parallel_external, automatic storage
```

```
Local variables:
      IS,MS is construct_local, static storage
      JA,LA is construct_local, automatic storage
      KD,ND is construct_local, data initialized static storage

NEW variables:
      LA’ is construct_local, automatic storage
      MS’ is construct_local, ?? (auto or static)
      ND’ is construct_local, ??
```

6.0 Input/Output

Each Fortran unit number is shared among all processes of a parallel program. An implementation shall provide synchronization among all processes accessing a specified unit.

When a unit number is connected to a file (for example through the use of an open statement), then all processes are able to access that file by using the same unit number. The unit shall not be explicitly connected to a file by an OPEN statement if it is currently connected to a file by a previous OPEN statement.

The effect of executing a data transfer input/output statement shall be as if the operations were performed in the order specified on page 125, lines 17-26 in the Fortran 90 standard. If multiple processes are executing the program, then the order of operations shall be augmented as follows:

Insert the following step between steps 2 and 3:
(2.5) Obtain an implementation lock associated with the unit

Insert the following step between steps 7 and 8:
(7.5) Free the implementation lock obtained for the unit

The result shall be that once a process obtains the lock for a given unit, the data transfer of the input/output list specified for the I/O statement will be completed prior to another process transferring data to or from the same unit.

The implementation lock obtained for the unit shall control the synchronization of the file pointer to the unit among all processes. The I/O statements shall not be synchronization points for program data objects. A program shall use the explicit or implicit synchronization points defined by the model for program data objects.

If the user wishes to cause I/O statements executed by distinct, simultaneously-executing processes to be applied to a unit number in a particular order, explicit, user-coded synchronization shall be used.

A program shall control synchronization of concurrent I/O to multiple units if required.

When a READ statement detects an end-of-file for a unit, all subsequent reads issued by other processes to that unit number - prior to a file repositioning statement (REWIND, BACKSPACE, CLOSE followed by OPEN, direct-access READ, direct access WRITE) will also detect end-of-file.

6.1 Multiple End-of-File Records

For cases where multiple end-of-file records can be detected on a unit after executing a single open (example, unlableled tapes with multiple files in many implementations) it is necessary to provide an additional I/O statement to skip past the current end-of-file record. Implementatins that allow only a single end-of-file per file may implement this statement as a CONTINUE statement.

6.1.1 Explicit Syntax

SKIP PAST EOF just-like-backspace-both forms

6.2 Examples

6.2.1

```
        subroutine exio1()
        dimension a(100)
        parallel sections 10 i=1,n
           section /a/
             read (*,7) n
               ...
           section /b/
               ...
           section /c/ wait(a)
             if (n.gt.100) print*,'error', n
             read (*,7) a(1:n)
               ...
        end parallel sections
```

In example EXIO1, section c waits for section a to complete so that it knows the number of elements of A to read. The user must program the required synchronization to ensure that the read of the n value occurs before trying to read n elements of A.

6.2.2

```
        subroutine exio2()
        dimension a(100)
        parallel sections 10 i=1,n
           section /a/
             i=6
             write(*,6) f1(i)
               ...
           section /b/
             i=8
             write(*,8) f1(i)
               ...
        end parallel sections
        return
        end
        function f1(i)
        ...
        read (*,i+1) ...
        ...
        return
        end
```

In example EXIO2, the user is responsible for ensuring that there is no synchronization required between I/O to units; or for providing the necessary synchronization. The example as written is correct since the process executing section A will write to units 6 and 7; while the process executing section B will write to units 8 and 9. However, if function f1 tried to read from unit 8 when i=6 and to read from unit 6 when i=8 there would be a chance of deadlock. To prevent the deadlock, the user would have to use explicit synchronization to ensure that only one process was executing the write statements in sections a and b.

7.0 Synchronization

Implicit synchronization occurs at the following statements:

```
PARALLEL
END PARALLEL
END PARALLEL PDO
END PDO (WAIT)
PARALLEL SECTIONS
END PARALLEL SECTIONS
END PSECTIONS (WAIT)
END PGROUP
```

and after the execution of the statement that terminates a "labeled" PDO or PARALLEL PDO.

7.1 Explicit Synchronization

(The following is material suggested by Bruce Leasure on March 7,1993)

The X3H5 module defines new types to support explicit synchronization. As a group, these types are referred to as control types. These types have no public fields. Use of objects of these types is restricted by the Fortran 90 typing mechanism. The control types defined are

```
  TYPE ( LATCH )        for latch
  TYPE ( LOCK )         for lock
  TYPE ( EVENT )        for event
  TYPE ( ORDINAL )      for sequence
```

```
***** Aside to X3J3 *****
```

In the next revision of Fortran, consider extending R502 to make these types base types. Two possibilites seem plausible: make each of these types a base type (such as INTEGER is now), or make them all different KINDs of the same base type.

7.1.1 Extensions Shared by Many Synchronization Methods

7.1.1.1 Representing States

The X3H5 module defines defines 7 symbolic INTEGER constants to represent the states of objects of TYPE (LATCH), TYPE (LOCK), and TYPE (EVENT). An implementation shall assign unique values to each of the symbolic constants representing a state of a single type. An implementation should assign unique values to each of these constants. The symbolic constants representing states are

for TYPE (LATCH):

STATE_UNINITIALIZED	for state uninitialized
STATE_UNLATCHED	for state unlatched

STATE_LATCHED for state latched

for TYPE (LOCK):
STATE_UNINITIALIZED for state uninitialized
STATE_UNLOCKED for state unlocked
STATE_LOCKED for state locked

for TYPE (EVENT):
STATE_UNINITIALIZED for state uninitialized
STATE_CLEAR for state clear
STATE_SET for state set

7.1.1.2 Testing for Uninitialized State

The X3H5 module defines the unary operator .UNINITIALIZED. where the single argument is an object of a control type and the result type is LOGICAL. The operator returns ".TRUE." if the corresponding object is uninitialized, otherwise the operator returns ".FALSE.". When applied to an array argument, the operator is elemental.

An implementation may always return ".FALSE." as the result of this operator, if the implementation does not detect an error when any operation except initialize is performed on an object of a control type that has state "uninitialized".

7.1.1.3 SYNCHRONIZE Statement

7.1.1.3.1 Proposed X3H5 Extended Syntax Rule

X701 sync-stmt is SYNCHRONIZE(sync-param-list) [guards-spec]

X702 sync-param is [CONTROL=] sync-object
or [OPERATION=] sync-operation
or [POSITION=] ordinal-position
or [STATUS=] sync-status

X703 sync-operation is scalar-character-expression

X704 sync-object is scalar-latch-variable
or scalar-lock-variable
or scalar-event-variable
or scalar-ordinal-variable

X705 ordinal-position is scalar-integer-expression

X706 sync-status is scalar-integer-variable

CONSTRAINT: Exactly one sync-object shall be specified in each sync-param-list.

CONSTRAINT: Exactly one sync-operation shall be specified in each sync-param-list.

CONSTRAINT: More than one ordinal-position shall not be specified in any sync-param-list.

CONSTRAINT: An ordinal-position shall be specified only if sync-object is of TYPE (ORDINAL).

CONSTRAINT: More than one sync-status shall not be specified in any sync-param-list.

If the sync-status variable is coded, the variable is assigned the integer corresponding to the final state of the sync-object after the execution of the sync-operation. The sync-status variable may be undefined when execution of the SYNCHRONIZE statement begins.

A SYNCHRONIZE statement shall not be executed if sync-object has a state of "uninitialized".

7.1.1.3.2 Consistency Rules for the SYNCHRONIZE Statement

If the sync-stmt specifies a guards-spec, the implementation shall make the objects in the guarded-obj-list consistent as a part of the execution of the sync-stmt.

If the sync-stmt specifies a sync-obj with a GUARDS attribute then the implementation shall make the objects in the guarded-obj-list from that attribute consistent as a part of the execution of the sync-stmt.

If the sync-stmt has no guards-spec and has a sync-obj with no GUARDS attribute, the implementation shall make all shared objects, used or defined as a result of the execution of "block", consistent as a part of the execution of the sync-stmt.

7.1.1.4 Representing Synchronization Operations

The X3H5 module defines defines symbolic CHARACTER constants to represent the operations on objects of TYPE (LOCK), TYPE (EVENT) and TYPE (ORDINAL) that act as explicit synchronization points. An implementation shall assign unique values to each of the symbolic constants representing a operations on a single type. An implementation should assign unique values to each of the operations.

TYPE (LOCK)

OP_CONDITIONAL_SET	for operation conditional set
OP_SET_WITH_WAIT	for operation set with wait
OP_CLEAR	for operation clear

TYPE (EVENT)

OP_SET	for operation set
OP_CLEAR	for operation clear
OP_WAIT	for operation wait

TYPE (ORDINAL)

OP_WAIT_THEN_POST_VALUE	for operation post a value with wait
OP_WAIT_VALUE	for operation wait for a value

7.1.1.5 Use of Control Types and Assignment

The X3H5 module defines the assignment operator to represent the initialize operation, the destroy operation, and the query operation.

7.1.2 Limiting Synchronization Overhead

A new attribute is defined that only has meaning for the synchronization types defined in the X3H5 module. R503 is extended to accomplish this.

7.1.2.1 Proposed X3H5 Extended Syntax Rule

```
R503   attr-spec        is PARAMETER
                  or access-spec
                  or ALLOCATABLE
                  or DIMENSION ( array-spec )
                  or EXTERNAL
            NEW   or guards-spec
                  or INTENT ( intent-spec )
                  or INTRINSIC
                  or OPTIONAL
                  or POINTER
                  or SAVE
                  or TARGET

X707   guards-spec            is GUARDS ( guarded-obj-list )

X708   guarded-obj            is variable-name
                  or array-element
                  or array-section
                  or substring

CONSTRAINT: each subscript, substring, or section-subscript in a
       guards-spec must be an integer initialization expression
       (see Fortran 7.1.6.1)
```

7.1.2.2 GUARDS Attribute

The GUARDS attribute specifies that the entities whose names are declared on this statement control the consistency of the objects in the guarded-obj-list.

The GUARDS attribute may only be used with an object of a control type.

The GUARDS attribute reduces the default list of objects that the implementation must make consistent at a SYNCHRONIZE statement with an associated object of a control type from all shared object to only those shared objects listed in the GUARDS attribute of the associated object.

7.1.3 Critical Sections

7.1.3.1 Proposed X3H5 Extended Syntax Rule

```
X709   critical-block         is critical-stmt
                              block
                          end-critical-stmt

X710   critical-stmt          is CRITICAL SECTION [ ( scalar-latch-variable ) ]
[ guards-spec ]

X711   end-critical-stmt      is END CRITICAL SECTION [ ( scalar-latch-variable
) ]

CONSTRAINT: If the end-critical-section-stmt specifies a
        scalar-latch-variable, the corresponding
        critical-section-stmt shall specify the same
        scalar-latch-variable.
```

7.1.3.2 Consistency Rules for CRITICAL SECTION

If the *critical-stmt* specifies a *guards-spec*, the implementation shall make the objects in the *guarded-obj-list* consistent at entry and exit to the *critical-block.*

If the *critical-stmt* specifies a *scalar-latch-variable* with a GUARDS attribute then the implementation shall make the objects in the *guarded-obj-list* from that attribute consistent at entry and exit to the *critical-block.*

If the *critical-stmt* has no *guards-spec* and no *scalar-latch-variable*, the implementation shall make all shared objects, used or defined as a result of the execution of "block", consistent at entry and exit to the *critical-block.*

If the *critical-stmt* has no *guards-spec* and has a *scalar-latch-variable* with no GUARDS attribute, the implementation shall make all shared objects, used or defined as a result of the execution of block, consistent at entry and exit to the critical-block.

7.1.3.3 Operations on Objects of TYPE (LATCH)

The initialize operation is performed on an object of TYPE (LATCH) by assignment of the value STATE_UNLATCHED to the object.

The *enter_critical_section* operation is performed on an object of TYPE (LATCH) by executing a CRITICAL SECTION statement referencing the latch.

The *exit_critical_section* operation is performed on an object of TYPE (LATCH) by executing an END CRITICAL SECTION statement that corresponds to a CRITICAL SECTION statement referencing the latch.

The destroy operation is performed on an object of TYPE (LATCH) by assignment of the value STATE_UNINITIALIZED.

The query operation is performed on an object of TYPE (LATCH) by assignment of the object to a variable of type INTEGER.

7.1.3.4 Default Latch

If a *critical-stmt* does not specify a *scalar-latch-variable,* the *critical-stmt* behaves as if the *critical-stmt* referenced a unique, initialized, *scalar-latch-variable* that is shared with every process. This *scalar-latch-variable* does not have a GUARDS attribute.

7.1.4 Locks

The initialize operation is performed on an object of TYPE (LOCK) by assignment of the value STATE_UNLOCKED to the object.

The conditional set operation is performed on an object of TYPE (LOCK) by executing a SYNCHRONIZE statement specifying the object as *sync-object* and a *sync-operation* of OP_CONDITIONAL_SET. The program should use either the query operation or a sync-status variable to determine if the lock was obtained.

The set with wait operation is performed on an object of TYPE (LOCK) by executing a SYNCHRONIZE statement specifying the object as sync-object and a sync-operation of OP_SET_WITH_WAIT.

The clear operation is performed on an object of TYPE (LOCK) by executing a SYNCHRONIZE statement specifying the object as sync-object and a sync-operation of OP_CLEAR.

The destroy operation is performed on an object of TYPE (LOCK) by assignment of the value STATE_UNINITIALIZED.

The query operation is performed on an object of TYPE (LOCK) by assignment of the object to a variable of type INTEGER.

7.1.5 Events

The initialize operation is performed on an object of TYPE (EVENT) by assignment of the value STATE_CLEAR to the object.

The set operation is performed on an object of TYPE (EVENT) by executing a SYNCHRONIZE statement specifying the object as sync-object and a sync-operation of OP_SET.

The clear operation is performed on an object of TYPE (EVENT) by executing a SYNCHRONIZE statement specifying the object as sync-object and a sync-operation of OP_CLEAR.

The wait operation is performed on an object of TYPE (EVENT) by executing a SYNCHRONIZE statement specifying the object as sync-object and a sync-operation of OP_WAIT.

The destroy operation is performed on an object of TYPE (EVENT) by assignment of the value STATE_UNINITIALIZED. No more processes.

7.1.6 Sequences

The initialize operation is performed on an object of TYPE (ORDINAL) by assignment of either an *scalar-integer-exp* or a one-dimensional INTEGER array with 2 elements to the object. When a *scalar-integer-exp* is used, the arithmetic sequence begins at the value of *scalar-integer-exp* and has a stride of 1. When a one-dimensional INTEGER array with 2 elements is used, the arithmetic sequence begins at the value of the first element of the array, and has a stride of the second element of the array. A program shall not use a stride of zero. The implementation shall detect a zero stride as an error. The post a value with wait operation is performed on an object of TYPE (ORDINAL) by executing a SYNCHRONIZE statement specifying the object as *sync-object*, a *sync-operation* of OP_WAIT_THEN_POST_VALUE, and an *ordinal-position* of the value of the arithmetic sequence to post.

The clear operation is performed on an object of TYPE (ORDINAL) by executing a SYNCHRONIZE statement specifying the object as sync-object, a sync-operation of OP_WAIT_VALUE, and an ordinal-position of the value of the arithmetic sequence to wait for.

The destroy operation is performed on an object of TYPE (ORDINAL) by assignment of the value STATE_UNINITIALIZED.

(The following is material put in during the march 1-3, 1993 meeting.)

7.2 Explicit Synchronization

Derived Types are defined in the X3H5 module for each of the synchronization objects specified by the model.

Relationship between model synchronizer types and Fortran synchronizer types:

model synchronizer type	derived type name
lock	Type (lock)
latch	Type (latch)
event	Type (event)
sequence	Type (ordinal)

A new attribute, the "guards" attribute for synchronizers is defined only for use with these derived types. This attribute associates one or more objects with the synchronizer:

```
GUARDS (guarded-list) sync-object
or
GUARDS :: sync-guards-list
where guarded is variable-name,
            array-name,
            array-element,
            array-section,
            module-name, or
            /common-block-name/ and
sync-guards-list is sync-object (guarded-list) [, sync-guards-list]
```

7.2.1 Critical Sections

Critical sections provide an easy to use method of allowing only one process at a time to execute the enclosed portion of code. Only one process is allowed within all critical sections that share a Lock. Critical sections are a structured use of lock synchronization. The structured approach is much more reliable than using the equivalent unstructured synchronization. Critical section synchronization can be used anywhere in the program. Most uses of critical sections preserve execution order independence so use within a worksharing construct without the ORDERED qualifier is standard conforming.

7.2.1.1 Explicit Syntax

Statement Forms

```
[label:]         CRITICAL SECTION [(lock)] [GUARDS(object-name-list)]
      END CRITICAL SECTION [(lock)] [label]
```

Structured As

```
  [label:] CRITICAL SECTION ...
                 statements
      END CRITICAL SECTION ...[label]
```

Where

lock is a variable name or array element of type lock

object-name is a data object

7.2.1.2 Coding Rules

The Critical Section construct is a block structured construct. The Critical Section construct follows all of the rules of Fortran block structured constructs. <* so we mean EXIT and Cycle WORK?*>

If the lock is coded on the END CRITICAL SECTION statement, it must match the corresponding lock on the CRITICAL SECTION statement.

7.2.1.3 Interpretation

A program that executes a CRITICAL SECTION statement with a lock that has a value of undefined is not standard conforming.

Entering a Critical Section construct, is equivalent to executing a GET_LOCK statement on the specified lock with an identical GUARDS clause. Leaving the critical section, by executing the END CRITICAL SECTION statement or executing a PDONE statement, is equivalent to executing an UNLOCK statement on the lock controlling the section with an identical GUARDS clause, and then resuming execution at the appropriate statement outside the block.

An unnamed Critical Section (one without a lock specified) is functionally equivalent to a Critical Section that specifies a lock that is

a) shared among all teams

b) initialized at program start-up to "unlocked"

c) is only referenced by that Critical Section construct

These rules cause lexically distinct unnamed Critical Sections to function independently. Any single unnamed Critical Section controls all processes, allowing at most one process within the Critical Section at any point in time.

7.2.1.4 Examples

```
Example 12           SUBROUTINE EX12 (A,B,SUM)
                  REAL B(0:100)
                  Lock A
```

```
      PARALLEL PDO I=1,10
        NEW T
        CRITICAL SECTION (A)
           T   = B(I) * B(I-1)
           SUM = SUM + T
        END CRITICAL SECTION (A)
      END PARALLEL PDO
      END
```

In Example 12, the lock A is used to control access to all shared objects and limit access to the enclosed block of code. The implementation must ensure that the shared object SUM is consistent upon entry and exit to the Critical Section construct, and that the shared array B is consistent upon entry to the Critical Section construct. (Note that B may be changed by a process that is not visible and that A and SUM must be initialized outside of the EX12 subroutine.)

Example 13

```
      SUBROUTINE EX13 (A,B,SUM)
      REAL B(0:100)
      Lock A
      GUARDS A(SUM)
      PARALLEL PDO I=1,10
        NEW T
        CRITICAL SECTION (A)
           T   = B(I) * B(I-1)
           SUM = SUM + T
        END CRITICAL SECTION
      END PARALLEL PDO
      END
```

In Example 13, the lock A is used to control access to the variable SUM. Because of the GUARDS statement, the implementation need only ensure that the shared variable SUM is consistent upon entry and exit to the Critical Section construct. This differs from the previous example in that shared array, B, is not required to be consistent during the critical section.

Example 14

```
      SUBROUTINE EX14 (A,B,SUM)
      REAL B(0:100)
      Lock A
      PARALLEL PDO I=1,10
        NEW T
        CRITICAL SECTION (A) GUARDS(SUM)
           T   = B(I) * B(I-1)
           SUM = SUM + T
        END CRITICAL SECTION (A)
      END PARALLEL PDO
      END
```

In Example 14, the lock A is used to control access to the variable SUM. Because of the GUARDS clause on the CRITICAL SECTION statement, the implementation shall ensure that the shared variable SUM is consistent upon entry and exit to the Critical Section construct. Example 14 is identical in functionality to Example 13.

Example 15

```
      SUBROUTINE EX15 (A,B,MAXA,GMAXA,N)
      REAL A(N), B(N), MAXA
      Lock GMAXA
      GUARDS GMAXA(MAXA)
```

```
         PARALLEL SECTIONS
         NEW AM
         SECTION
           AM = A(1)
           DO 10 I=2,N
              IF(AM.LT.A(I))AM=A(I)
10         CONTINUE
           CRITICAL SECTION (GMAXA)
              IF(MAXA.LT.AM) MAXA=AM
           END CRITICAL SECTION
         SECTION
           CRITICAL SECTION (GMAXA)
              AM=MAXA
           END CRITICAL SECTION
           DO 20 I=1,N
              B(I)=B(I)/AM
20         CONTINUE
         END PARALLEL SECTIONS
         END
```

In Example 15, the lock GMAXA is used to control access to the variable MAXA. The scaling of array B by the maximum element of the array A is performed in a nondeterministic fashion, depending upon the number of processes available, the assignment of the sections to processes, and the relative execution speed of the processes. In particular, the scaling may be done with the value of MAXA that was available upon invocation of this routine, or it may be done with the value of MAXA that will be returned to the calling program. This is an example of a program that is non-deterministic but standard conforming.

Example 16

```
               SUBROUTINE EX16 (A,B,MAXA,GMAXA,N)
         REAL A(N), B(N), MAXA
         Lock GMAXA
         GUARDS GMAXA(MAXA)
         PARALLEL SECTIONS
         NEW AM
         SECTION
           CRITICAL SECTION (GMAXA)
              AM=MAXA
           END CRITICAL SECTION (GMAXA)
           DO 10 I=2,N
              IF(AM.LT.A(I)) THEN
              CRITICAL SECTION (GMAXA)
                IF(MAXA.LT.A(I)) MAXA=A(I)
                AM=MAXA
              END CRITICAL SECTION
              ENDIF
10         CONTINUE
         SECTION
           DO 20 I=1,N
              CRITICAL SECTION (GMAXA)
                B(I)=B(I)/MAXA
              END CRITICAL SECTION
20         CONTINUE
         END PARALLEL SECTIONS
         END
```

In Example 16, the lock GMAXA is used to control access to the variable MAXA. The scaling of array B by MAXA is performed in a non-deterministic fashion because the scaling does not wait for the computation of MAXA to be complete. The value of MAXA used at any point in the scaling process depends upon the number of processes available, the assignment of the sections to processes, and the relative execution speed of the processes. In particular, the scaling of an individual element of B may be done with the value of MAXA that was available upon invocation of this routine, or it may be done with the value of MAXA that will be returned to the calling program, or with some intermediate value. All elements of B need not be scaled with the same value. While non-deterministic, this program is standard conforming.

```
  Example 17           SUBROUTINE EX17 (B,SUM)
                 REAL B(0:100)
                 SUM = 0.0
                 PARALLEL PDO I=1,10
                   NEW T
                   CRITICAL SECTION  GUARDS(SUM)
                      T   = B(I) * B(I-1)
                      SUM = SUM + T
                   END CRITICAL SECTION
                 END PARALLEL PDO
                 END
```

In Example 17, an unnamed Critical Section construct is used to control access to the shared variable SUM. Behavior is as if all processes used the same lock variable to control the access, even if the processes that called this routine happened to be on distinct teams, and SUM was a new object at those higher levels of parallelism (think of nested parallelism).

```
  Example 18           SUBROUTINE EX18 (B,SUM,PROD)
                 REAL B(100)
                 PARALLEL SECTIONS
                 NEW T
                 SECTION
                   T = 0.0
                   DO 10 I=1,10
      10              T = T + B(I)
                   CRITICAL SECTION  GUARDS(SUM)
                      SUM = T
                   END CRITICAL SECTION
                 SECTION
                   T = 1.0
                   DO 20 I=1,10
      20              T = T * B(I)
                   CRITICAL SECTION  GUARDS(PROD)
                      PROD = T
                   END CRITICAL SECTION
                 END PARALLEL SECTIONS
                 END
```

In Example 18, unnamed Critical Sections are used to control access to distinct shared variables SUM and PROD. Each lexical occurrence of an unnamed Critical Section construct operates independently, so one process can be executing inside the first Critical Section and another process can be executing inside the second Critical Section.

```
  Example 19           SUBROUTINE EX19 (A,B,MAXA,N)
            C
            C >>> NOT STANDARD CONFORMING <<<
            C
```

```
      REAL A(N), B(N), MAXA
      PARALLEL SECTIONS
      NEW AM
      SECTION
        AM = A(1)
        DO 10 I=2,N
           IF(AM.LT.A(I))AM=A(I)
10      CONTINUE
        CRITICAL SECTION GUARDS(MAXA)
           IF(MAXA.LT.AM) MAXA=AM
        END CRITICAL SECTION
      SECTION
        CRITICAL SECTION GUARDS(MAXA)
           AM=MAXA
        END CRITICAL SECTION
        DO 20 I=1,N
           B(I)=B(I)/AM
20      CONTINUE
      END PARALLEL SECTIONS
      END
```

In Example 19, two unnamed Critical Section constructs are used in an attempt to control access to the variable MAXA. But, because each unnamed Critical Section construct has its own unique lock variable, this program is not standard conforming because it allows one process to be reading the value of a shared variable while another process is updating it.

Example 20

```
      SUBROUTINE EX20 (B,SUM)
      REAL B(0:100)
      Lock A
      UNLOCK(A)
      PARALLEL PDO I=1,10
        NEW T
        T = B(I) * B(I-1)
        CRITICAL SECTION (A)
           SUM = SUM + T
        END CRITICAL SECTION (A)
      END PARALLEL PDO
      END
```

In Example 20, the lock A is used to control access to all shared objects and limit access to the enclosed block of code, but a good implementation can remove the shared array B from the list of controlled objects because the lock A is new to the team created by the Parallel Do construct. (Note that B may be not changed by a process that is not visible because the visibility of the lock A does not extend outside of this program unit.) It is important for an implementation to reduce the amount of code within a Critical Section to a minimum. This can easily be done if only updated objects or read objects are listed in the GUARDS clause or applicable GUARDS statement. The programmer should also make an effort to code small Critical Sections, but the easy optimizations should be done by an implementation.

Example 21

```
      SUBROUTINE EX21 (A,B,SUM)
      REAL B(0:100)
      Lock A
      PARALLEL PDO I=1,10
        CRITICAL SECTION (A) GUARDS(SUM)
           SUM = SUM + B(I) * B(I-1)
        END CRITICAL SECTION
```

```
                    END PARALLEL PDO
                    END
```

In Example 21, the a good implementation would move the multiplication of elements of B out of the Critical Section.

7.2.2 Event Synchronization

Event synchronization is most often used to signify when something has occurred, especially in those cases where more than one process is interested in the occurrence.

Event synchronization provides operations to indicate that an event has not occurred (CLEAR), to indicate that an event has occurred (POST), and to ensure that an event has occurred (WAIT).

Event synchronization may be used anywhere in the program. Care shall be taken to

1. preserve execution order independence if used within a worksharing construct without the ORDERED qualifier.

2. ensure that the synchronization pattern described does not require more than one process for correct execution.

7.2.2.1 Explicit Syntax

Statement Forms

POST (*event*) [GUARDS(*object-name-list*)]

WAIT (*event*) [GUARDS(*object-name-list*)]

CLEAR (*event*) [GUARDS(*object-name-list*)]

Where

event is a variable or array element of type event

object-name is a variable name, an array name, an array element, or a common block name enclosed in /'s

7.2.2.2 Coding Rules

POST, WAIT and CLEAR are executable statements.

7.2.2.3 Interpretation

An event may assume one of two values: "cleared" or "posted".

When a CLEAR statement is executed,

a) the appropriate shared variables are made consistent

b) event is set to "cleared", no matter what its value was previously.

When a POST statement is executed,

a) the appropriate shared variables are made consistent

b) the value of event is set to "posted", no matter what its value was previously.

When a WAIT statement is executed,

a) the appropriate shared variables are made consistent

b) the value of event is tested to see if it is "posted" if it is not, the process retry's this step at a later time,

The initial value of an event is undefined. It becomes defined only upon the execution of a CLEAR or POST statement. A program that executes a WAIT statement on an event with an undefined value is not standard conforming.

7.2.2.4 Examples

Example 22

```
      SUBROUTINE EX22 (B,E)
REAL B(100),C
EVENT E(100)
PARALLEL PDO I=1,97
  IF (I .LT. 4) THEN
     POST E(I)
  ELSE
     CLEAR E(I)
  ENDIF
END PARALLEL PDO
PARALLEL PDO (ORDERED) I=4,100
  NEW C
  C = SIN(B(I))
  WAIT E(I-3)
  B(I) = B(I) + B(I-3)*C
  POST E(I)
END PARALLEL PDO
END
```

Example 22 computes a recurrence to solve for B. Each computed value of B is used in the computation of the value of B three iterations later of the loop. The code above permits the SIN calculations to be done completely in parallel, while the computation of B is synchronized.

7.2.2.5 Intrinsic Functions for Events

LOGICAL FUNCTION POSTED(event)

This intrinsic function returns a logical value that is .TRUE. if the event is "posted" and otherwise it returns .FALSE..

```
  Example 23            SUBROUTINE EX23 (C,D)
            C
            C  >>> NOT STANDARD CONFORMING <<<
            C
                  REAL C,D
                  EVENT A, B
                  CLEAR A
                  CLEAR B
                  PARALLEL SECTIONS (ORDERED)
                  SECTION
                    WAIT A
                    C = C + 1
                    POST B
                  SECTION
                    POST A
                    WAIT B
                    D = C + 2
                  END PARALLEL SECTIONS
                  END
```

If Example 23 is executed by a single process, it will deadlock because that process will be assigned to the first section and immediately go into a permanent wait. Example 23 is not standard conforming.

Deadlock avoidance is the responsibility of the programmer. Here are some hints that can help in avoiding deadlock. (A standard conforming program need not follow these hints.)

(1) Do not use event synchronization in unordered parallel loops or unordered parallel sections.

(2) In Parallel Do and Pdo constructs with the ORDERED qualifier, make sure that POST statement is executed for an iteration earlier in the serial order than the iteration containing the corresponding WAIT statement.

In Parallel Sections and Psections constructs with the ORDERED qualifier, make sure that the section containing the POST statement occurs lexically before the section containing the corresponding WAIT statement.

7.2.3 Sequences: Ordinal Synchronization

Ordinal synchronization is used to communicate between iterations of a loop, or to communicate between distinct loops. Any series of events that can be numbered can be synchronized with ordinal synchronization.
Ordinal synchronization describes an arithmetic sequence. It provides operations to define an arithmetic sequence (SET), indicate that computation for a particular element of the sequence is

complete (POST), and to ensure that the computation for a particular element of the sequence completes (WAIT).

Ordinal synchronization may be used anywhere in the program. If a Parallel Do or Pdo construct is used to create the arithmetic sequence being synchronized, then the ORDERED qualifier is required. Care shall be taken to

1. preserve execution order independence if used within a worksharing construct without the ORDERED qualifier.

2. ensure that the synchronization pattern described does not require more than one process for correct execution.

Most uses of a single ordinal synchronizer do not describe a synchronization pattern that requires more than one process for correct execution.

7.2.3.1 Explicit Syntax

Statement Forms

```
POST (seq, iexp1) [GUARDS(object-name-list)]

WAIT (seq, iexp2) [GUARDS(object-name-list)]

SET (seq [, iexp3[, iexp4]]) [GUARDS(object-name-list)]
```

Where

seq is a variable or array element of type ordinal

iexp1, *iexp2* and *iexp3* are integer expressions

iexp4 is an integer expression not equal to zero

object-name is a variable name, an array name, an array element, or a common block name enclosed in /'s

7.2.3.2 Coding Rules

POST, WAIT and SET are executable statements.

7.2.3.3 Interpretation

All integer expressions are evaluated just once, before any of the statement specific actions are performed.

When a SET statement is executed,

1. the appropriate shared objects are made consistent as specified by the Language Independent Model, *X3H5 Language Independent Model*.

2. iexp3 is the initial value of seq. If iexp3 is not coded, an initial value of 0 is assumed. iexp4 is the increment between elements of the sequence. If iexp4 is not coded, an increment of 1 is assumed.

When a POST statement is executed,

1. the appropriate shared objects shall be made consistent as specified by the Language Independent Model, *X3H5 Language Independent Model*.

2. the value of seq is compared with iexp1 - increment. If seq is less than, and increment >0, or if seq is greater than, and increment <0 then the process repeats step 1) at a later time

3. if the value of seq is equal to iexp1 - increment then set the value of seq to be iexp1.

When a WAIT statement is executed

1. the appropriate shared objects shall be made consistent as specified by the Language Independent Model, *X3H5 Language Independent Model*.

2. the value of seq is compared with iexp2 If seq is less than, and increment >0, or if seq is greater than, and increment <0 then the process repeats step 1) at a later time

The initial value of an object of type ordinal is undefined. It becomes defined only by execution of a SET statement.

A program that executes a POST or WAIT with a seq that has an undefined value is not standard conforming.

Anything that can be done with ordinal synchronization, can also be done with an array of type event, but the reverse is not true. In the cases where ordinal synchronization can be used, it permits a significant storage savings.

7.2.3.4 Examples

Example 24

```
      SUBROUTINE EX24 (B,SUM)
REAL B(100),SUM(100)
ORDINAL A
GUARDS A(SUM)
SUM(1) = 0.0
SET A
PARALLEL PDO (ORDERED) I=2,10
  NEW T
  T = B(I) * B(I-1)
```

```
    WAIT (A,I-1)
    SUM(I) = SUM(I-1) + T
    POST (A,I)
  END PARALLEL PDO
  END
```

Example 25

```
      SUBROUTINE EX25 (B,SUM)
  REAL B(100),SUM(100)
  EVENT AA(100)
  GUARDS AA(SUM)
  POST(AA(1))
  PARALLEL PDO I=2,100
    CLEAR AA(I)
  END DO
  SUM(1) = 0.0
  PARALLEL PDO (ORDERED) I=2,10
    NEW T
    T = B(I) * B(I-1)
    WAIT AA(I-1)
    SUM(I) = SUM(I-1) + T
    POST AA(I)
  END PARALLEL PDO
  END
```

To illustrate, consider Examples 24 and 25 which perform exactly the same computation using ordinal and event synchronization. However, Ordinal synchronization is not general enough to code every program that can be built with events with equivalent efficiency.

Example 26

```
      SUBROUTINE EX26 (B,C,N)
  REAL B(N),C(N)
  PARAMETER (MAXN=1000)
  EVENT E(MAXN)
  PARALLEL PDO 10 I=1,N
    IF (I .lt. 4) THEN
       POST E(I)
    ELSE
       CLEAR E(I)
    ENDIF
10 CONTINUE
  PARALLEL PDO (ORDERED) 20 I=4,N
    C(I) = FUNC(B(I))
    WAIT E(I-3) GUARDS(B(I-3))
    B(I) = B(I) + B(I-3)*C(I)
    POST E(I) GUARDS (B(I))
20 CONTINUE
  END
```

Consider Example 26 where the user function FUNC may have widely varying execution times

Example 27

```
      SUBROUTINE EX27 (B,C,N)
  REAL B(N),C(N)
  ORDINAL E
  GUARDS E(B)
  SET (E,3)
  PARALLEL PDO (ORDERED) 20 I=4,N
    C(I) = FUNC(B(I))
    WAIT (E, I-3))
    B(I) = B(I) + B(I-3)*C(I)
```

```
            POST (E,I)
  20      CONTINUE
          END
```

and the obvious transcription to ordinal synchronization provided by Example 27. Examples 26 and 27 both compute the same result as long as the value of N is less than MAXN. Both examples are standard conforming. Example 26 allows 3 processes to execute totally independently, but uses more storage, and must know the maximum value of N. Example 27 requires that all of the POST statements be completed in serial iteration order (recall that posting a ordinal synchronizer has an implied wait for the previous value in the sequence to be posted), thus providing more synchronization than is absolutely necessary to compute the result. Example 27 does not require as much storage for synchronizers.

Example 28

```
          SUBROUTINE EX28 (A,B,C,N1,N2,N3)
          REAL A(*),B(*),C(*)
          ORDINAL D
          GUARDS D(C)
          SET (D,N1,N3)
          PARALLEL SECTIONS
          SECTION
            DO 10 I=N1,N2,N3
               C(I) = MAX(A(I),A(I-N3))
               POST(D,I)
  10        CONTINUE
          SECTION
            DO 20 I=N1,N2,N3
               WAIT(D,I)
               B(I) = B(I)/C(I)
  20        CONTINUE
          END PARALLEL SECTIONS
          END
```

Example 28 demonstrates use of Ordinal synchronization to perform pipeline style synchronization. In this case, the result of one DO loop is piped into another DO loop operating on the same index set. In Example 28, the first loop computes the maximum element of A encountered so far, and stores this local maximum in C. The second loop scales the array B based upon the local maximum.

Example 29

```
          SUBROUTINE EX29 (B)
          REAL B(100)
          ORDINAL A
          SET (A,2)
          PARALLEL PDO (ORDERED) I=1,99
            NEW T
            T = B(I+1)
            POST (A,I+1)
            B(I) = T
            END PARALLEL PDO
            B(100) = 0.0
          END
```

Example 29 demonstrates the use of ordinal synchronization utilizing the implied wait function that is built-in to the POST statement. This subroutine shifts the array B to the left, throwing away B(1). There is no need to wait, because when the POST statement is executed, the implied wait insures that the previous iteration has already been posted.

7.2.3.5 Intrinsic Functions for Ordinals

INTEGER FUNCTION INT(seq)

This intrinsic function, which is already defined for other Fortran data types, is extended to return the integer value of the current position in the arithmetic sequence described by seq, which is of type ordinal.

Avoiding Deadlock

As with event synchronization, deadlock is a possibility with ordinals.

```
Example 30           SUBROUTINE EX30 (B,C)
          C
          C  >>> NOT STANDARD CONFORMING <<<
          C
                REAL B(100), C
                ORDINAL A
                SET (A, -99)
                PARALLEL PDO (ORDERED) 10 I = 1,99
                  WAIT (A,-(I+1))
                  B(I) = B(I+1) + C
                  POST (A,-I)
          10    CONTINUE
                END
```

In Example 30, the program will deadlock with any number of processors less than 99, because the iterations are handed in order from first to last. If there are only 98 processors, they will all wait for the last iteration to execute its POST statement. This program unit is not standard conforming because it requires at least 99 processes to avoid deadlock. To be standard conforming, a program unit must be capable of completing execution with any number of processes.

7.2.4 Unstructured synchronization - Locks

Unstructured control of LOCKs should not be used if some other LOCK synchronization mechanism is more appropriate (try critical sections or ordinal synchronization). Unstructured control of LOCKs is prone to many, hard to find, programming errors.

Unstructured control of LOCKs can be used anywhere within the program. Care should be taken to preserve execution order independence if used within a worksharing construct without the ORDERED qualifier. Care should be taken to ensure that the synchronization pattern described does not require more than one process for correct execution.

7.2.4.1 Explicit Syntax

Statement Forms

```
GET_LOCK (lock) [GUARDS(object-name-list)]
UNLOCK (lock) [GUARDS(object-name-list)]
```

Where

lock is a variable or array element of type lock

object-name is a data object>

7.2.4.2 Coding Rules

The GET_LOCK and UNLOCK statements are executable statements.
<GET_LOCK and UNLOCK are subroutines defined in the X3H5 module. >

7.2.4.3 Interpretation

A lock may assume one of two values: "locked" and "unlocked". Execution of UNLOCK causes the value of the specified LOCK to become "unlocked", no matter what the value was previously. When UNLOCK is executed, these actions take place:

U1) the appropriate shared objects are made consistent

U2) if the current value of the lock is "locked", the value is changed to "unlocked". GET_LOCK has the following effect:

L1) appropriate shared objects are made consistent

L2) if the current value of the specified LOCK is "unlocked" then

L2a) the value is changed to "locked"

L2b) execution continues with the next statement

L3) if the value of the specified LOCK is "locked", the process retries step L2) at a later time.

Step L2) and L2a) above are executed as a single atomic operation.

The initial value of a LOCK is undefined. It becomes defined only at the execution of UNLOCK.

A program that executes GET_LOCK on lock with an undefined value is not standard conforming.

If a GUARDS clause is specified then for the duration of the synchronization statement, the names listed shall be used to augment the set of objects guarded by that synchronizer if the synchronizer was specified in a <sync-list> of the GUARDS statement. The merged set of guarded objects shall be made consistent when the synchronization statement is encountered. By explicitly identifying names of objects that shall be made consistent, the GUARDS clause and GUARDS statement remove a requirement for the implementation to make any other objects consistent when the synchronization statement is encountered.

7.2.4.4 Examples

Example 7

```
              REAL FUNCTION SUM(A,B)
          REAL B(0:100)
          LOCK A
sumproduct:             PARALLEL PDO I=1,10
              NEW T
              GET_LOCK (A)
              T = B(I) * B(I-1)
              SUM = SUM + T
              UNLOCK (A)
            END PARALLEL PDO sumproduct
            END
```

In Example 7, the Lock A is used to control access to the variable SUM. The implementation must ensure that all necessary shared objects, SUM and B are consistent at the GET_LOCK statement and the UNLOCK statement. Because of the possibility that another process executing some other parallel construct might change elements of the array B, both elements of B would have to be read from shared memory on every iteration of the loop unless the implementation could determine that those elements of B would not change while this parallel construct was executing.

Example 8

```
              SUBROUTINE EX8 (A,B,SUM)
          REAL B(0:100)
          LOCK A
          GUARDS A(SUM)
          PARALLEL PDO I=2,10
            NEW T
            GET_LOCK (A)
            T = B(I) * B(I-1)
            SUM = SUM + T
            UNLOCK (A)
          END PARALLEL PDO
          END
```

In Example 8, the variable A is used as a lock to control access to the variable SUM. Because of the GUARDS statement, the implementation need only ensure that the shared variable SUM is consistent at the GET_LOCK statement and at the UNLOCK statement. No action is required with respect to array B because B is not changed during this operation.

7.2.4.5 Intrinsic Functions for Locks

LOGICAL FUNCTION TRY_LOCK(lock)

The value of an object of type lock may be determined using the intrinsic function TRY_LOCK. TRY_LOCK accepts a single argument of type lock, returning a result of type logical. If the value of the <u>lock</u> is locked, the result is .TRUE., otherwise it is .FALSE..

```
Example 9       SUBROUTINE EX9 (NAME,A)
                CHARACTER*(*) NAME
                CHARACTER*10 PG
                LOCK A
                IF ( TRY_LOCK (A) ) THEN
                  PG = "LOCKED"
                ELSE
                  PG = "UNLOCKED"
                ENDIF
                PRINT *,"Lock ",NAME," was ",PG
                END
```

In Example 9, the subprogram prints the current value of the lock A. The intrinsic TRY_LOCK is used to obtain the current value of the lock without modifying it.

<u>LOGICAL FUNCTION GET_LOCK(lock)</u>

This intrinsic function locks the lock if possible, but does not wait if it is already locked. GET_LOCK accepts a single argument of type lock, returning a result of type logical. The GET_LOCK intrinsic attempts to lock the <u>lock</u>. If the GET_LOCK intrinsic is successful in locking the <u>lock</u>, then the GET_LOCK intrinsic returns .TRUE.. If the <u>lock</u> is already locked, then the GET_LOCK intrinsic returns .FALSE.. The GET_LOCK intrinsic works exactly like the GET_LOCK statement, except that the GET_LOCK intrinsic does not wait if the lock is already locked.

```
Example 10      SUBROUTINE EX10 (A)
                Lock A
      5         IF (.NOT. GET_LOCK(A)) THEN
                  CALL USEFUL
                  GO TO 5
                ENDIF
                CALL UPDATE
                UNLOCK (A)
                END
```

In Example 10, the subprogram does some useful work rather than waiting for the lock to change values.

```
Example 11      SUBROUTINE EX11 (A)
                Lock A
      5            IF (TRY_LOCK (A)) THEN
                  CALL USEFUL
                  GO TO 5
                ELSE
                  GET_LOCK(A)
                  CALL UPDATE
                  UNLOCK (A)
                ENDIF
```

```
END
```

Notice the subtle difference between Examples 10 and 11. The TRY_LOCK intrinsic does not actually lock the lock, so it is possible for another process to lock the lock A in between the test performed with the TRY_LOCK intrinsic and the lock performed by the GET_LOCK statement.

8.0 Nondeterministic Programs

In parallel programming, there are situations in which the same program when run twice may not produce the same results. Such a program is **nondeterministic**. The X3H5 Fortran standard allows some standard conforming programs to be nondeterministic. In such cases, it is the programmer's responsibility to ensure that nondeterministic behavior is acceptable to the functioning of the program.

If a program is nondeterministic, an implementation is free to choose between the possible nondeterministic results. An implementation may always produce the same value for a nondeterministic result, or an implementation may be nondeterministic, and produce different results from one run to the next.

A.0 X3H5 Directive Binding

A.1 Directives - Introduction

The use of directives to provide information to a compiler is an established practice. The ability to parallelize programs with directives has been demonstrated to be useful on a number of parallel systems. Given an appropriate set of directives, an advantage of this approach has been that the directives may be treated as comments and the program will still run correctly. This has allowed programs that are parallelized with such directives to be run serially on a computer that may not understand those directives by treating them as comments.

This is understood to be particularly important to some code developers who must support both parallel and serial targets with a single source code. This is viewed by the committee to be an interim problem, given that there may be some time before compilers on serial systems handle the parallel statements defined herein in an appropriate serial manner.

The system of directives described in this appendix is imperative -- they are not advisory. The directives assert specific behavior for the parallel program or for the implementation.

Directive syntax and structure are specified in this appendix. Because of a basic one to one association between the directives and corresponding language statements, the specification for the directives will not replicate specifications given in this document for those associated language statements. Interpretations and coding rules are provided only when they are in addition to those provided for the corresponding language statement.

Examples in this appendix have been derived from those in the body of this document when useful for illustrating some aspect of the directive binding. Corresponding example numbers have been used to facilitate comparison between language and directive bindings, although this does not result in a sequential numbering of the examples in this appendix.

A.1.1 Role of the Directive Binding

This directive binding is specified for the Fortran-77 language only and is provided as a conversion aid. It will not be specified or extended to use additional features of the Fortran-90 language. To aid as an interim conversion aid, this set of directives has been designed to be easily replaced, either manually or mechanically, by their corresponding language statements.

The directive binding has a direct correspondence to statements in the language binding and these directives instruct the implementation just as if the corresponding language statement were present. When they are coded, they result in exactly the same interpretation being taken by the implementation as if it encountered the corresponding language statements.

A.1.2 Single Process Execution Requirement for Compliant Programs

The X3H5 LIM requires that a compliant parallel program be written so that it may be executed with an arbitrary number or processes. Notably, the program must be executable by a single process. A key implication of this rule is that when a compliant program is being executed by a single process, the process shall never encounter a barrier that would cause it to be blocked.

Equivalent Serial Execution:

A compliant parallel program using this binding can be written so that it has an "equivalent serial execution". A program has an "equivalent serial execution", if that program is written so that the semantic features introduced by the parallel directives are rendered superfluous by the construction of the code. Serial execution of such a program, achieved by ignoring directives, will produce a result that is one of the possible results from the parallel execution of that parallel program.

There are two features of a X3H5 parallel directives to be discussed when considering the serial interpretation of a X3H5 compliant program:

A) Implicit and explicit synchronization points, and
B) The introduction of scoping at parallel constructs.

Following this discussion, the X3H5 intrinsic functions will be examined in the context of serial execution.

Coding to provide an equivalent serial execution is not a requirement when using the X3H5 directive binding, but ignores the primary advantage for use of directives. Unless otherwise noted the examples in this appendix are coded so that they have an equivalent serial execution.

A.1.3 Synchronization and Serial Execution

A parallel program is similar to a traffic grid - synchronization is the system of traffic lights that keep multiple processes from "running into each other". When those streets are used by a single vehicle, it is free to ignore all of the lights without worry of a collision at an intersection.

The single process execution requirement guarantees that a "serial process" may ignore the synchronization points (implicit or explicit) in a compliant parallel program without hazard. Those synchronization points can never block that single process. Because there is a single process executing the program, there is not need to communicate values of shared objects at synchronization points.

A.1.3.2 Scoping at Parallel Constructs and Serial Execution

The addition of a scope at the level of the parallel construct allows the mapping associated with a construct private object to change at the construct boundary. The definition/reference pattern for that object will determine whether change in storage association is significant to the semantics of the program when the construct is ignored.

Naming private objects for a parallel construct uniquely from any objects used outside the scope of that construct is sufficient to ensure an equivalent serial execution. Uniquely naming the objects used within a parallel construct nullifies the effect of the new scope -- allowing the directives to be safely ignored.

Alternate Intrinsic Functions

Because the synchronization points in a serial execution will be ignored, the values of synchronizers between synchronization points are meaningless. The intrinsic inquiry functions that relate to binary states are specified to return fixed values that allow the serial process to proceed undeterred.

Although the directive binding supports the INT function for ORDINALs, this function is not supported under serial execution. This is because ORDINAL synchronizers do not have a binary state and a suitable version of the INT function for serial use cannot be constructed. A program using the X3H5 directive binding that is to be interpreted serially can not use the INT function.

A.1.4 Terminology

A program using this directive binding has an "equivalent serial execution" if coded in a fashion that ensures the result of its serial interpretation will be one of the results of the parallel execution of the program.

A "directive sentinel" is the special pattern of characters that appears beginning in column 1, and indicates that the line is to be interpreted as an X3H5 parallel directive. The X3H5 directive sentinel is 'C$PAR'.

A.1.5 Directives - General Usage Requirements in Parallel Programs

This set of directives is intended to be easily replaced, either manually or mechanically, by their corresponding language statements. Because of this, they may only be coded at statement boundaries.

A.1.5.1 Continued Directives

Unlike X3H5 parallel statements which may be continued by the conventional Fortran continuation mechanism, there is no mechanism in Fortran for comments of which directives are a special case. In the case of a long directive in a construct, the optional clauses may be combined with a "directive sentinel", to form an additional directive. Such a directive must immediately follow the base directive. The specifications of individual directives that may require continuation in this manner contain specific instructions.

A.1.6 Parallel Intrinsic Functions

A program utilizing the X3H5 directive binding uses the same set of intrinsic functions as in the case of the language binding. These functions are specified in the main portion of this document.

A.1.6.1 Parallel Intrinsic Behavior for Equivalent Serial Execution

When a program with these parallel directives is to be executed serially, it is linked with an alternate library. In this library, fixed values are returned by intrinsic to reflect the values that are appropriate for a serial execution on a single processor computing system. The behavior of these functions is defined in the appropriate sections of this appendix, paralleling the corresponding sections in the body of this standard.

A.1.6.2 Functionality Not Supported Under Serial Interpretation

When the SET and POST directives for ORDINALs are ignored, a value to be returned by the INT function cannot be reconciled in a way that reflects the state of the sequence. Therefore, the INT function for ORDINAL data types can not be coded in a program that is to be interpreted serially.

A.2 Syntax Rules

A.2.1 Parallel Do Construct

A.2.1.1 Syntax

Directive Forms for Component Directives:

```
C$PAR    PARALLEL PDO [(option_list)]
```

```
C$PAR    END [PARALLEL] DO
```

Structured As:

```
 C$PAR   PARALLEL PDO [(option_list)]
[C$PAR   NEW obj_list]
         >> Fortran do-loop <<
[C$PAR   END PARALLEL PDO]
```

A.2.1.2 Coding Rules

No executable statements may appear between the PARALLEL PDO directive and the beginning of the do-loop.

The coding of the END PARALLEL PDO directive is optional. If the END PARALLEL PDO directive is coded, no executable statements may appear between the last statement of the do-loop and the END PARALLEL PDO directive.

A.2.1.3 Examples

```
Example 1
               SUBROUTINE EX1 (A,B,C,E,T,N)
               REAL A(N),B(N),C(N+1),E(N),T

       C$PAR   PARALLEL PDO
               DO 10 I=1,N
                 E(I) = A(I)*B(I)
                 C(I+1) = E(I) * (T-1.0)
        10     CONTINUE
               END
```

```
Example 2
               SUBROUTINE EX2 (A,B,C,E,T,N)
               REAL A(N),B(N),C(N+1),E(N),T

       C$PAR   PARALLEL PDO
               DO I=1,N
                 E(I) = A(I)*B(I)
                 C(I+1) = E(I) * (T-1.0)
               END DO
 C$PAR   END PARALLEL PDO
               END
```

A.2.2 Parallel Sections Construct

A.2.2.1 Syntax

Directive Forms for Component Directives:

```
C$PAR     PARALLEL SECTIONS [(qual_list)]

C$PAR     SECTION [/sec_nm/] [WAIT (sec_nm_list)] [GUARDS (obj_nm_list)]

C$PAR     END [PARALLEL] SECTIONS
```

Structured As:

```
 C$PAR    PARALLEL SECTIONS [(option_list)]]
[C$PAR    NEW obj_list]
 C$PAR    SECTION ...
          >> statements <<
    [ ... zero or more additional section blocks ]
 C$PAR    END PARALLEL SECTIONS
```

A.2.2.2 Interpretation

A "section block" is composed of a SECTION directive followed by some number of executable Fortran statements. The end of a section block is signalled by the next SECTION or END PARALLEL SECTIONS directive.

The WAIT and GUARDS clauses may appear as separate directives immediately following the corresponding SECTION directive. This is achieved by coding a line with the directive sentinel and the particular clause. Multiple instances of the WAIT and GUARDS clauses associated with a particular SECTION directive are additive, having the same effect as if they had appeared in a single clause for that section block.

A.2.2.3 Examples

Example 3

```
      SUBROUTINE EX3 (A,B,C,D,E,F,N)
      REAL A(N),B(N),C(N),D(N),E(N),F(N)

C$PAR PARALLEL SECTIONS
C$PAR SECTION
        DO 10 I=1,N
           A(I) = B(I) * C(I)
   10   CONTINUE
C$PAR SECTION
        DO 20 J=1,M
           D(I) = F(J) / E(I)
   20   CONTINUE
C$PAR END PARALLEL SECTIONS
      END
```

Example 4

```
      SUBROUTINE EX4 (A,B,C,D,E,F,N)
      REAL A(N),B(N),C(N),D(N),E(N),F(N)

C$PAR PARALLEL SECTIONS
C$PAR SECTION
C$PAR   PARALLEL PDO
        DO I=1,N
           A(I) = B(I) * C(I)
        END DO
C$PAR SECTION
C$PAR   PARALLEL PDO
        DO J=1,M
           D(I) = F(J) / E(I)
        END DO
C$PAR END PARALLEL SECTIONS
      END
```

Example 5

```
      SUBROUTINE EX5 (Z,ZA,ZB,ZC,ZD,ZE)
      REAL Z(5)

C$PAR PARALLEL SECTIONS (ORDERED)
C$PAR SECTION /A/
        ZA = ZFUNC(Z(1))
C$PAR SECTION /B/
        ZB = 2*ZFUNC(Z(2))
C$PAR SECTION /C/ WAIT (A)
        ZC = ZA * ZA + ZFUNC(Z(3))
C$PAR SECTION /D/ WAIT (A,B)
        ZD = ZB - ZA + ZFUNC(Z(4))
C$PAR SECTION /E/ WAIT (C,B)
        ZE = ZC - ZB + ZFUNC(Z(5))
C$PAR END PARALLEL SECTIONS
      END
```

Example 6

```
      SUBROUTINE EX6
      REAL Z(10)
C$PAR SCOMMON /Z/
   COMMON /Z/ ZB,ZD,ZE,ZTOT

C$PAR PARALLEL SECTIONS
C$PAR SECTION /A/
        ZA = ZFUNC(Z(1))
```

```
        C$PAR  SECTION /BC/
                 ZB = ZFUNC(Z(2))
                 ZC = ZFUNC(Z(3))
        C$PAR  SECTION /D/ WAIT (A)
                 ZD = ZFUNC(ZA)
        C$PAR  SECTION /E/ WAIT (A,BC) GUARDS (ZA,ZC)
                 ZE = ZJOIN(ZA,ZC))
        C$PAR  END PARALLEL SECTIONS
     ZTOT = ZJOINs(ZE,ZD,ZB)
               END
```

```
Example 6A
               SUBROUTINE EX6A
               REAL Z(10)
  C$PAR  SCOMMON /Z/
     COMMON /Z/ ZB,ZD,ZE,ZTOT

        C$PAR  PARALLEL SECTIONS
        C$PAR  SECTION /A/
                 ZA = ZFUNC(Z(1))
        C$PAR  SECTION /BC/
                 ZB = ZFUNC(Z(2))
                 ZC = ZFUNC(Z(3))
        C$PAR  SECTION /D/ WAIT (A)
                 ZD = ZFUNC(ZA)
        C$PAR  SECTION /E/
  C$PAR    WAIT (A,BC)
  C$PAR    GUARDS (ZA,ZC)
                 ZE = ZJOIN(ZA,ZC))
        C$PAR  END PARALLEL SECTIONS
     ZTOT = ZJOINs(ZE,ZD,ZB)
               END
```

This example derived from example 6 illustrates how a long SECTION directive may be "continued" by decomposing it into components.

A.2.3 Synchronization Declarations

A.2.3.1 Syntax

Directive Forms

```
C$PAR    GATE declarator_list
C$PAR    EVENT declarator_list
C$PAR    ORDINAL declarator_list

C$PAR    GUARDS guards_list
```

Directive Forms

```
C$PAR    IMPLICIT sync_type
```

Structured As

```
C$PAR   IMPLICIT sync_type
        IMPLICIT fort_type >"just-list-an-implicit-range"_list<
```

where

sync_type is one of GATE, EVENT or ORDINAL

A.2.3.2 Coding Rules

Variables identified in a GATE or EVENT declaration directive shall be Fortran variables that occupy exactly one numeric storage location. Variables identified in an ORDINAL declaration shall be Fortran variables that occupy exactly two numeric storage locations. An X3H5 compliant compiler shall verify the storage requirements and flag noncompliance as an error.

The GATE, EVENT and ORDINAL directives are specifications, and may be coded anywhere a Fortran specification statement may be coded.

The IMPLICIT directive must appear immediately preceding the Fortran IMPLICIT statement to which it applies. The "IMPLICIT directive/IMPLICIT statement" pairs may be coded anywhere a Fortran IMPLICIT statement may be coded.

A.2.4 Unstructured Locking Synchronization

A.2.4.1 Syntax

Directive Forms

```
C$PAR     GETLOCK (gate) [GUARDS (obj_nm_list)]
```

```
C$PAR     UNLOCK (gate) [GUARDS (obj_nm_list)]
```

The GUARDS clause may appear as separate directive immediately following the corresponding GETLOCK or UNLOCK directive. This is achieved by coding a line with the directive sentinel and the particular GUARDS clause. Multiple instances of the GUARDS clauses associated with a particular GETLOCK or UNLOCK directive are additive, having the same effect as if they had appeared in a single clause.

A.2.4.2 Examples

```
Example 7
                SUBROUTINE EX7 (A,B)
                REAL B(0:100)
    C$PAR       GATE A
                INTEGER AA

    C$PAR       PARALLEL PDO
    C$PAR       NEW T
    C$PAR       DO I=1,10
                  T = B(I) * B(I-1)
    C$PAR         LOCK (A)
                  SUM = SUM + T
    C$PAR         UNLOCK (A)
                END DO
    C$PAR       END PARALLEL PDO
```

```
            END
```

Example 8

```
            SUBROUTINE EX8 (A,B,SUM)
            REAL B(0:100)
C$PAR       GATE A

C$PAR       GUARDS A(SUM)
C$PAR       UNLOCK (A)
            SUM = 0.0

C$PAR       PARALLEL PDO
C$PAR       NEW T
            DO I=1,10
              T = B(I) * B(I-1)
C$PAR         GETLOCK (A)
              SUM = SUM + T
C$PAR         UNLOCK (A)
            END DO
C$PAR       END PARALLEL PDO
            END
```

Note that variable A defaults to type REAL, having one numeric storage unit as required.

Example 9

```
            SUBROUTINE EX9 (NAME,A)
            CHARACTER*(*) NAME
            CHARACTER*10 PG
C$PAR       GATE A

            IF ( LOCKED (A) ) THEN
              PG = "LOCKED"
            ELSE
              PG = "UNLOCKED"
            ENDIF
            PRINT *,"GATE ",name," is ",PG
            END
```

Example 10

```
            SUBROUTINE EX10 (A)
C$PAR       GATE A

          5 IF (.NOT. LOCK(A)) THEN
              CALL USEFUL
              GO TO 5
            ENDIF
            CALL UPDATE
C$PAR       UNLOCK (A)
            END
```

Example 11

```
            SUBROUTINE EX11 (A)
C$PAR       GATE A

          5 IF (LOCKED (A)) THEN
              CALL USEFUL
              GO TO 5
            ELSE
C$PAR         GETLOCK(A)
              CALL UPDATE
C$PAR         UNLOCK (A)
            ENDIF
            END
```

A.2.4.2.1 Function Values for GATEs in Serial Execution

The X3H5 directive binding uses the same intrinsic functions as specified for the X3H5 Fortran language. These functions are specified in the body of this standard.

A program containing these functions that is to be executed serially should be bound to a set of corresponding intrinsic that always return a value that indicates that the synchronizer is "open".

function name	value returned
LOCKED(gate_name)	.FALSE.
LOCK(gate_name)	.TRUE.

A.2.5 Critical Sections

A.2.5.1 Syntax

Directive Forms

```
C$PAR     CRITICAL SECTION [(gate)] [GUARDS (obj_nm_list)]

C$PAR     END CRITICAL SECTION [(gate)]
```

Structured As

```
C$PAR   CRITICAL SECTION ...
             >statements<
C$PAR   END CRITICAL SECTION ...
```

The GUARDS clause may appear as separate directive immediately following the corresponding CRITICAL SECTION directive. This is achieved by coding a line with the directive sentinel and the particular GUARDS clause. Multiple instances of the GUARDS clauses associated with a particular CRITICAL SECTION directive are additive, having the same effect as if they had appeared in a single clause.

A.2.5.1 Examples

```
Example 12
                SUBROUTINE EX12 (A,B,SUM)
                REAL B(0:100)
    C$PAR       GATE A

    C$PAR       UNLOCK(A)
    C$PAR       PARALLEL PDO
    C$PAR       NEW T
                DO I=1,10
                  T = B(I) * B(I-1)
    C$PAR         CRITICAL SECTION (A)
                     SUM = SUM + T
    C$PAR         END CRITICAL SECTION (A)
                END DO
    C$PAR       END PARALLEL PDO
                END
```

Example 13

```
            SUBROUTINE EX13 (A,B,SUM)
            REAL B(0:100)
    C$PAR   GATE A

    C$PAR   GUARDS A(SUM)
    C$PAR   UNLOCK(A)
            SUM = 0.0
    C$PAR   PARALLEL PDO
    C$PAR   NEW T
            DO I=1,10
    C$PAR     CRITICAL SECTION (A)
                 T = B(I) * B(I-1)
                 SUM = SUM + T
    C$PAR     END CRITICAL SECTION
            END DO
    C$PAR   END PARALLEL PDO
            END
```

Example 14

```
            SUBROUTINE EX14 (A,B,SUM)
            REAL B(0:100)
    C$PAR   GATE A

    C$PAR   UNLOCK(A)
            SUM = 0.0

    C$PAR   PARALLEL PDO
    C$PAR   NEW T
            DO I=1,10
              T = B(I) * B(I-1)
    C$PAR     CRITICAL SECTION (A) GUARDS(SUM)
                 SUM = SUM + T
    C$PAR     END CRITICAL SECTION
            END DO
    C$PAR   END PARALLEL PDO
            END
```

Example 15

```
            SUBROUTINE EX15 (A,B,MAXA,GMAXA,N)
            REAL A(N), B(N), MAXA
    C$PAR   GATE GMAXA
    C$PAR   GUARDS GMAXA(MAXA)

    C$PAR   PARALLEL SECTIONS
    C$PAR   NEW AM
    C$PAR   SECTION
              AM = A(1)
              DO 10 I=2,N
                 IF(AM.LT.A(I))AM=A(I)
       10     CONTINUE
    C$PAR     CRITICAL SECTION (GMAXA)
                 IF(MAXA.LT.AM) MAXA=AM
    C$PAR     END CRITICAL SECTION (GMAXA)
    C$PAR   SECTION
    C$PAR     CRITICAL SECTION (GMAXA)
                 AM=MAXA
    C$PAR     END CRITICAL SECTION (GMAXA)
              DO 20 I=1,N
                 B(I)=B(I)/AM
       20     CONTINUE
    C$PAR   END PARALLEL SECTIONS
```

```
      END
```

Example 16

```
      SUBROUTINE EX16 (A,B,MAXA,GMAXA,N)
      REAL A(N), B(N), MAXA
C$PAR GATE GMAXA
C$PAR GUARDS GMAXA(MAXA)

C$PAR PARALLEL SECTIONS
C$PAR NEW AM
C$PAR SECTION
C$PAR   CRITICAL SECTION (GMAXA)
           AM=MAXA
C$PAR   END CRITICAL SECTION (GMAXA)
        DO 10 I=2,N
           IF(AM.LT.A(I)) THEN
C$PAR         CRITICAL SECTION (GMAXA)
                 IF(MAXA.LT.A(I)) MAXA=A(I)
                 AM=MAXA
C$PAR         END CRITICAL SECTION (GMAXA)
           ENDIF
   10   CONTINUE
C$PAR SECTION
        DO 20 I=1,N
C$PAR      CRITICAL SECTION (GMAXA)
              B(I)=B(I)/MAXA
C$PAR      END CRITICAL SECTION (GMAXA)
   20   CONTINUE
C$PAR END PARALLEL SECTIONS
      END
```

Example 17

```
      SUBROUTINE EX17 (B,SUM)
      REAL B(0:100)

      SUM = 0.0
C$PAR PARALLEL PDO
C$PAR NEW T
      DO I=1,10
        T = B(I) * B(I-1)
C$PAR   CRITICAL SECTION  GUARDS(SUM)
           SUM = SUM + T
C$PAR   END CRITICAL SECTION
      END DO
      END
```

Example 18

```
      SUBROUTINE EX18 (B,SUM,PROD)
      REAL B(100)

C$PAR PARALLEL SECTIONS
C$PAR NEW T
C$PAR SECTION
        T = 0.0
        DO 10 I=1,10
   10     T = T + B(I)
C$PAR   CRITICAL SECTION  GUARDS(SUM)
          SUM = T
C$PAR   END CRITICAL SECTION
C$PAR SECTION
        T = 1.0
        DO 20 I=1,10
   20     T = T * B(I)
```

```
    C$PAR      CRITICAL SECTION  GUARDS(PROD)
                 PROD = T
    C$PAR      END CRITICAL SECTION
    C$PAR    END PARALLEL SECTIONS
             END
```

```
Example 20
             SUBROUTINE EX20 (B,SUM)
             REAL B(0:100)
    C$PAR    GATE A

    C$PAR    UNLOCK(A)
    C$PAR    PARALLEL PDO
    C$PAR    NEW T
             DO I=1,10
               T = B(I) * B(I-1)
    C$PAR      CRITICAL SECTION (A)
                    SUM  = SUM + T
    C$PAR      END CRITICAL SECTION (A)
             END DO
             END
```

```
Example 21
             SUBROUTINE EX21 (A,B,SUM)
             REAL B(0:100)
    C$PAR    GATE A

    C$PAR    UNLOCK(A)
    C$PAR    PARALLEL PDO
             DO I=1,10
    C$PAR      CRITICAL SECTION (A) GUARDS(SUM)
                 SUM = SUM + B(I) * B(I-1)
    C$PAR      END CRITICAL SECTION (A)
             END DO
             END
```

A.2.6 Event Synchronization

A.2.6.1 Syntax

Directive Forms

```
C$PAR     POST (event) [GUARDS (obj_nm_list)]

C$PAR     WAIT (event) [GUARDS (obj_nm_list)]

C$PAR     CLEAR (event) [GUARDS (obj_nm_list)]
```

The GUARDS clause may appear as separate directive immediately following the corresponding POST, WAIT, CLEAR directive. This is achieved by coding a line with the directive sentinel and the particular GUARDS clause. Multiple instances of the GUARDS clauses associated with a particular POST, WAIT, CLEAR directive are additive, having the same effect as if they had appeared in a single clause.

```
Example 22
             SUBROUTINE EX22 (B,E)
             REAL B(100),C
    C$PAR    EVENT E(100)
```

```
C$PAR    PARALLEL PDO
         DO I=1,97
           IF (I .lt. 4) THEN
C$PAR         POST (E(I))
           ELSE
C$PAR         CLEAR (E(I))
           ENDIF
         END DO

C$PAR    PARALLEL PDO (ORDERED)
C$PAR    NEW C
         DO I=4,100
           C = SIN(B(I))
C$PAR      WAIT (E(I-3))
           B(I) = B(I) + B(I-3)*C
C$PAR      POST (E(I))
         END DO

         END
```

A.2.6.1.1 Function Values for Events in Serial Execution

A program containing these functions that is to be executed serially should be bound to a set of corresponding intrinsic that always return a value that indicates that the synchronizer is "open".

function name	value returned
POSTED(event_name)	.TRUE.

A.2.7 Ordinal (Sequence) Synchronization

A.2.7.1 Syntax

Directive Forms

```
C$PAR   POST (seq, iexp1) [GUARDS (obj_nm_list)]

C$PAR   WAIT (seq, iexp2) [GUARDS (obj_nm_list)]

C$PAR   CLEAR (seq[, iexp3[, iexp4]]) [GUARDS (obj_nm_list)]
```

The GUARDS clause may appear as separate directive immediately following the corresponding POST, WAIT, CLEAR directive. This is achieved by coding a line with the directive sentinel and the particular GUARDS clause. Multiple instances of the GUARDS clauses associated with a particular POST, WAIT, CLEAR directive are additive, having the same effect as if they had appeared in a single clause.

```
  Example 24
        SUBROUTINE EX24 (B,SUM)
        REAL B(100),SUM(100)
C$PAR   ORDINAL A
C$PAR   GUARDS A(SUM)

        SUM(1) = 0.0
C$PAR   SET (A)
C$PAR   PARALLEL PDO (ORDERED)
```

```
C$PAR   NEW T
        DO I=2,10
          T = B(I) * B(I-1)
C$PAR     WAIT (A,I-1)
          SUM(I) = SUM(I-1) + T
C$PAR     POST (A,I)
        END DO
        END
```

Example 25

```
        SUBROUTINE EX25 (B,SUM)
        REAL B(100),SUM(100)
C$PAR   EVENT AA(100)
C$PAR   GUARDS AA(SUM)

C$PAR   POST(AA(1))
C$PAR   PARALLEL PDO
        DO I=2,100
C$PAR     CLEAR (AA(I))
        END DO
        SUM(1) = 0.0
C$PAR   PARALLEL PDO (ORDERED)
C$PAR   NEW T
        DO I=2,10
          T = B(I) * B(I-1)
C$PAR     WAIT (AA(I-1))
          SUM(I) = SUM(I-1) + T
C$PAR     POST (AA(I))
        END DO
        END
```

Example 26

```
        SUBROUTINE EX26 (B,C,N)
        REAL B(N),C(N)
        PARAMETER (MAXN=1000)
C$PAR   EVENT E(MAXN)

C$PAR   PARALLEL PDO
        DO 10 I=1,N
          IF (I .lt. 4) THEN
C$PAR       POST (E(I))
          ELSE
C$PAR       CLEAR (E(I))
          ENDIF
   10   CONTINUE
C$PAR   PARALLEL PDO (ORDERED)
        DO 20 I=4,N
          C(I) = FUNC(B(I))
C$PAR     WAIT (E(I-3)) GUARDS(B(I-3))
          B(I) = B(I) + B(I-3)*C(I)
C$PAR     POST (E(I)) GUARDS (B(I))
   20   CONTINUE
        END
```

Example 27

```
        SUBROUTINE EX27 (B,C,N)
        REAL B(N),C(N)
C$PAR   ORDINAL E
C$PAR   GUARDS E(B)

C$PAR   SET (E,3)
C$PAR   PARALLEL PDO (ORDERED)
        DO 20 I=4,N
```

```
                C(I) = FUNC(B(I))
C$PAR           WAIT (E, I-3))
                B(I) = B(I) + B(I-3)*C(I)
C$PAR           POST (E,I)
   20         CONTINUE
              END
```

Example 28

```
              SUBROUTINE EX28 (A,B,C,N1,N2,N3)
              REAL A(*),B(*),C(*)
C$PAR         ORDINAL D
C$PAR         GUARDS D(C)

C$PAR         SET (D,N1,N3)
C$PAR         PARALLEL SECTIONS
C$PAR         SECTION
                DO 10 I=N1,N2,N3
                  C(I) = MAX(A(I),A(I-N3))
C$PAR             POST(D,I)
   10           CONTINUE
C$PAR         SECTION
                DO 20 I=N1,N2,N3
C$PAR             WAIT(D,I)
                  B(I) = B(I)/C(I)
   20           CONTINUE
C$PAR         END PARALLEL SECTIONS
              END
```

Example 29

```
              SUBROUTINE EX29 (B)
              REAL B(100)
C$PAR         ORDINAL A

C$PAR         SET (A,2)
C$PAR         PARALLEL PDO (ORDERED)
C$PAR         NEW T
              DO I=1,99
                T = B(I+1)
C$PAR           POST (A,I+1)
                B(I) = T
              END DO
              B(100) = 0.0
              END
```

A.2.7.1.1 Function Values for Counters in Serial Execution

The X3H5 intrinsic function INT(ordnl_var) will not produce a correct result under serial interpretation. If one expects to run a directive based parallel program serially, this function should not be used.

A.3 Data Sharing

A.3.1 Data Sharing Directives

A.3.1.1 Syntax

Directive Forms

```
C$PAR     NEW obj_nm_list
```

A.3.1.2 Rules

The NEW directive may only appear within a PARALLEL, PARALLEL PDO or PARALLEL SECTIONS construct. It appear with other NEW directives after the PARALLEL directive and the first executable statement.

A.3.2 Partially Shared Common Blocks

A.3.2.1 Syntax

Directive Forms

```
C$PAR     SCOMMON sname_list
```

Structured As

```
C$PAR     SCOMMON /COMM1/
     COMMON /COMM1/ A(99), B(99,73), X, Y, ZZ
```

A.3.2.2 Rules

The SCOMMON directive shall be located immediately before common block that is to be interpreted as an SCOMMON block.

COMMONs and SCOMMONs occupy the same name space, therefore if a COMMON block is associated with an SCOMMON directive anywhere in a parallel program, it shall have an associated SCOMMON directive everywhere that it occurs.

```
Example 40
                  SUBROUTINE EX40 (B)
          C$PAR  SCOMMON /BLOCKA/
                  COMMON /BLOCKA/ A(100)
                  REAL B(100)

          C$PAR  PARALLEL PDO
                  DO I=1,100
                    A(I) = I * I
                    B(I) = A(I) + B(I)
                  END DO
                  END

Example 41
                  SUBROUTINE EX41 (B)
      REAL B(100)

   C$PAR  PARALLEL PDO
      DO I=1,100
        CALL SUB(B(I))
      END DO
      END
```

```
     SUBROUTINE SUB(X)
  C$PAR  SCOMMON /BLOCKA/
               COMMON /BLOCKA/ A
     A=X
     CALL SQUARE
     X=A
     END

     SUBROUTINE SQUARE
        C$PAR  SCOMMON /BLOCKA/
               COMMON /BLOCKA/ A
     A=A*A
     END
```

Example 41A

```
               SUBROUTINE EX41A (B)
        C$PAR  SCOMMON /BLOCKA/
               COMMON /BLOCKA/ A
               REAL B(100)

        C$PAR  PARALLEL PDO
        C$PAR  NEW /BLOCKA/
               DO I=1,100
                 CALL SUB(B(I))
               END DO
               END

               SUBROUTINE SUB (X)
        C$PAR  SCOMMON /BLOCKA/
               COMMON /BLOCKA/ A
               A = X
               CALL SQUARE
               END

               SUBROUTINE SQUARE
        C$PAR  SCOMMON /BLOCKA/
               COMMON /BLOCKA/ A
               A = A*A
               END
```

Example 45

```
               SUBROUTINE EX45 (B)
               REAL B(100), C(100)

        C$PAR  PARALLEL PDO
               DO I=1,100
                 CALL SUB1(B(I))
       CALL SUB2(C(I))
               END DO
               PRINT *, (C(I), I = 1, 100)
               END

               SUBROUTINE SUB1 (X)
        C$PAR  SCOMMON /BLOCKA/
               COMMON /BLOCKA/ A
               SAVE /BLOCKA/
               A = X
               END

               SUBROUTINE SUB2 (X)
        C$PAR  SCOMMON /BLOCKA/
               COMMON /BLOCKA/ A
               SAVE /BLOCKA/
```

```
              X = A
              END
```

Example 46

```
              SUBROUTINE EX46 (B)
              REAL B(100), C(100)
       C$PAR  SCOMMON /BLOCKA/
              COMMON /BLOCKA/ A

       C$PAR  PARALLEL PDO
       C$PAR  NEW /BLOCKA/
              DO I=1,100
                CALL SUB1(B(I))
                CALL SUB2(C(I))
              END DO
              PRINT *, (C(I), I = 1, 100)
              END

              SUBROUTINE SUB1 (X)
       C$PAR  SCOMMON /BLOCKA/
              COMMON /BLOCKA/ A
              A = X
              END

              SUBROUTINE SUB2 (X)
       C$PAR  SCOMMON /BLOCKA/
              COMMON /BLOCKA/ A
              X = A
              END
```

Example 39

```
      SUBROUTINE EX39(B,C,N)
              REAL B(N),C(N)

C$PAR  PARALLEL PDO
C$PAR  NEW A
      PARALLEL PDO I=1,N
        A=B(I)+C(I)
        CALL EX39A(A,B,I)
      END DO
      END

      SUBROUTINE EX39A(AA,BB,N)
      REAL BB(N),BX
      DATA BX/1.0/

      BX=AA*(AA-4.0)/BX
C$PAR  PARALLEL PDO
      DO J=1,N
        BB(J)=BB(J)*BX
      END DO
      END
```

A.4 Parallel Region Construct

A.4.1 Syntax

Directive Forms - Parallel Region parallel construct component directives

```
C$PAR    PARALLEL [(roption_list)]

C$PAR    END PARALLEL
```

Structured As

```
C$PAR     PARALLEL [(roption_list)]
    [C$PAR     NEW obj_list]
   >> Statements <<
C$PAR     END PARALLEL
```

Directive Forms - Pdo worksharing construct component directives

```
C$PAR     PDO [(poption_list]

C$PAR     END PDO
```

Structured As:

```
C$PAR     PDO ...
   >> legal do loop <<
    [C$PAR     END PDO]
```

Directive Forms - Psections worksharing construct component directives

```
C$PAR     PSECTIONS [(poption_list)]

C$PAR     SECTION [/sec_nm/] [wait (sec_nm_list)] [GUARDS(obj_nm_list)]

C$PAR     END PSECTIONS
```

Structured As:

```
C$PAR     PSECTIONS ...
C$PAR     SECTION
   >> statements <<
    [     ... zero or more section blocks ]
C$PAR     END PSECTIONS
```

Directive Forms - Grouping construct component directives

```
C$PAR     GROUP [(poption_list)]

   C$PAR END GROUP
```

Structured As:

```
C$PAR     GROUP [(goption_list)]
   >> statements <<  ! replicated code for wsc 1
   >> worksharing construct 1 <<
   >> statements <<  ! replicated code for wsc 1
    [     ... zero or more redundant-code/worksharing blocks ]
C$PAR     END GROUP
```

The WAIT and GUARDS clauses may appear as separate directives immediately following the corresponding SECTION directive. This is achieved by coding a line with the directive sentinel and the particular clause. Multiple instances of the WAIT and GUARDS clauses associated with a particular SECTION directive are additive, having the same effect as if they had appeared in a single clause for that section block.

Example 48

```
              SUBROUTINE EX48 (A,B,C,N)
              REAL A(N),B(N),C(N)

      C$PAR  PARALLEL PDO
      C$PAR  NEW T
              DO I=1,N
                T = A(I)*B(I)
                C(I+1) = T * (T-1.0)
              END DO
              END
```

Example 49

```
              SUBROUTINE EX49 (A,B,C,N)
              REAL A(N),B(N),C(N)

      C$PAR  PARALLEL
      C$PAR  NEW T
      C$PAR    PDO
                DO I=1,N
                   T = A(I)*B(I)
                   C(I+1) = T * (T-1.0)
                END DO
      C$PAR  END PARALLEL
              END
```

Example 50

```
     SUBROUTINE EX50 (ZA,ZB,ZC,ZD,N)
              REAL ZA(N),ZB(N),ZC(N),ZD(N)

      C$PAR  PARALLEL SECTIONS
      C$PAR  NEW T
      C$PAR  SECTION /DS5A/
                DO 10 I=1,N
                  T = ZFUNC(ZA(I))
                  ZC(I) = T * T
        10      CONTINUE
      C$PAR  SECTION /DS5B/
                DO 20 I=1,N
                  T = ZFUNC(ZB(I)-ZA(I))
                  ZD(I) = T * T
        20      CONTINUE
      C$PAR  END PARALLEL SECTIONS
              END
```

Example 51

```
     SUBROUTINE EX51 (ZA,ZB,ZC,ZD,N)
              REAL ZA(N),ZB(N),ZC(N),ZD(N)

      C$PAR  PARALLEL
      C$PAR  NEW T
      C$PAR    PSECTIONS
      C$PAR    SECTION /DS5A/
                   DO 10 I=1,N
                     T = ZFUNC(ZA(I))
                     ZC(I) = T * T
        10         CONTINUE
      C$PAR    SECTION /DS5B/
                   DO 20 I=1,N
                     T = ZFUNC(ZB(I)-ZA(I))
                     ZD(I) = T * T
        20         CONTINUE
      C$PAR    END PSECTIONS
```

```
      C$PAR  END PARALLEL
             END
```

```
Example 52
     SUBROUTINE EX52 (A)
             REAL A(*)
             GATE B
             GUARDS B(SUM)

             UNLOCK(B)
             SUM=0.0
      C$PAR  PARALLEL
      C$PAR  NEW SUML
               SUML = 0.0
      C$PAR    PDO
               DO I=1,N
                  SUML = SUML + A(I)
               END DO
      C$PAR    CRITICAL SECTION (B)
                 SUM = SUM + SUML
      C$PAR    END CRITICAL SECTION (B)
      C$PAR  END PARALLEL
             END
```

All team members initialize SUML and execute the Critical Section construct regardless of whether they participated in the execution of the Pdo construct.

```
Example 52A
     SUBROUTINE EX52A (A)
             REAL A(*)
             GATE B
             GUARDS B(SUM)

             UNLOCK(B)
             SUM=0.0
      C$PAR  PARALLEL
      C$PAR  NEW SUML

  C$PAR    GROUP
                 SUML = 0.0
      C$PAR      PDO
                 DO I=1,N (NOWAIT)
                    SUML = SUML + A(I)
                 END DO
      C$PAR      CRITICAL SECTION (B)
                   SUM = SUM + SUML
      C$PAR      END CRITICAL SECTION (B)
      C$PAR    END GROUP

      C$PAR  END PARALLEL
             END
```

In this example, derived from EX52, team members to not enter the Group construct once all work in the Pdo construct has been assigned. Use of the Group construct helps prevent unnecessary executions of the Critical Section construct. Typical of Group construct usage, this example shows a pattern of private object initialization, worksharing construct execution, and reduction into a shared variable.

```
Example 53
     SUBROUTINE EX53 (A,B,C,D,N,M)
```

```
      REAL A(N),B(N),C(N),D(N)
C$PAR  PARALLEL
C$PAR    PDO
         DO I=1,N
            A(I) = B(I) * C(I)
         END DO
C$PAR    PDO
         DO I=1,M
            D(I) = A(I) - C(I)
         END DO
C$PAR  END PARALLEL
       END
```

```
Example 54
     SUBROUTINE EX54 (A,C,N,M)
      REAL A(N,0:M),C(N,M)
C$PAR  PARALLEL
         DO 10 J=1,M
C$PAR       PDO
            DO I=1,N
              A(I,J) = C(I,J)/A(I,J-1)
            END DO
   10    CONTINUE
C$PAR  END PARALLEL
       END
```

A.4.2 Single Process Sections

A.4.2.1 Syntax

Directive Forms

```
C$PAR     SINGLE PROCESS

C$PAR     END SINGLE PROCESS
```

Structured as

```
  C$PAR   SINGLE PROCESS
>> Statements <<
  C$PAR   END SINGLE PROCESS
```

```
Example 55
     SUBROUTINE EX55 (A,B,N)
      REAL A(N),B(N)
C$PAR  PARALLEL
C$PAR    PDO
         DO I=1,N
            A(I) = 1.0 / A(I)
         END DO
C$PAR    SINGLE PROCESS
            IF ( A(1) .GT. 1.0 ) A(1) = 1.0
C$PAR    END SINGLE PROCESS
C$PAR    PDO
         DO I=1,N
            B(I) = B(I) / A(1)
         END DO
```

```
     C$PAR  END PARALLEL
            END
```

Example 56

```
SUBROUTINE EX56 (A,B,N)
            REAL A(N),B(N)

     C$PAR  PARALLEL
     C$PAR    PDO
              DO I=1,N
                 A(I) = 1.0 / A(I)
              END DO
       C$PAR  PSECTIONS
       C$PAR  SECTION
                 IF ( A(1) .GT. 1.0 ) A(1) = 1.0
       C$PAR  END PSECTIONS
       C$PAR  PDO
              DO I=1,N
                 B(I) = B(I) / A(1)
              END DO
     C$PAR  END PARALLEL
            END
```

Example 57

```
SUBROUTINE EX57 (A,AMAX,N)
            REAL A(0:N)

            AMAX = 0.0
     C$PAR  PARALLEL
     C$PAR  NEW ALMAX

C$PAR    GROUP
     C$PAR      PDO (NOWAIT))
                DO I=1,N
                  IF ( ABS(A(I)) .GT. ABS(ALMAX) ) ALMAX = A(I)
                END DO
     C$PAR      CRITICAL SECTION
                  IF ( ABS(ALMAX) .GT. ABS(AMAX) ) AMAX = ALMAX
     C$PAR      END CRITICAL SECTION
     C$PAR    END GROUP

     C$PAR    SINGLE PROCESS
                ALMAX = A(1)+A(N)
                IF ( AMAX .LT. ALMAX ) AMAX = 1.0 + AMAX
     C$PAR    END SINGLE PROCESS

     C$PAR    PDO
              DO I=1,N
                A(I) = ABS( A(I) / AMAX )
              END DO

     C$PAR  END PARALLEL
            END
```

A.5 Exits from Parallel Constructs

A.5.1 Syntax

Directive Forms

```
C$PAR     PDONE

  Example 3
        SUBROUTINE EX3 (A,N,*)
        REAL A(N)
        LOGICAL FOUND

        FOUND=.FALSE.
     C$PAR  PARALLEL PDO
        DO I=1,N
          IF ( A(I) .EQ. 0.0 ) THEN
     C$PAR      PDONE
            FOUND=.TRUE.
          ENDIF
        END DO

        IF ( .NOT. FOUND ) THEN
          PRINT*,'ALL ELEMENTS ARE NON-ZERO'
          RETURN 0
        ELSE
          PRINT*,'ERROR: THERE IS A ZERO ELEMENT IN A'
        ENDIF
        END
```

Note that because the PDONE directive/statement is not preemptive, it may be coded anywhere in the conditional above with the same effect.

A.6 Extended Intrinsic

A.6.1 Parallel Intrinsic Functions

The X3H5 directive binding uses the same intrinsic functions as specified for the X3H5 Fortran language. These functions are specified in the body of this standard.

A.6.2 Definition of Serial Execution Library

Intrinsic	Value Returned
INTEGER FUNCTION NPRCFG()	1
INTEGER FUNCTION MPRTOT()	1
INTEGER FUNCTION NPRAVL()	0
INTEGER FUNCTION NPRUSE()	1
INTEGER FUNCTION NPSCFG()	1
INTEGER FUNCTION MPSTOT()	1
INTEGER FUNCTION NPSAVL()	0
INTEGER FUNCTION NPSUSE()	1
INTEGER FUNCTION NPSTM()	1
SUBROUTINE SPRTOT(integer-expr)	none, routine has no effect
SUBROUTINE SPSTOT(integer-expr)	none, routine has no effect

A *parallel-region-construct* is:

```
[name:]     PARALLEL [(parallel-option)]
            data-sharing-spec
            parallel-body
      END PARALLEL [name]
```

```
where
      parallel-option is MAX PARALLEL = int-expr |
                         ORDERED                 |
                         MAX PARALLEL = int-expr, ORDERED |
                         ORDERED, MAX PARALLEL = int-expr
      parallel-body is   statements               |
                         parallel-construct
      parallel-construct is parallel-region-construct |
                               pdo-construct             |
                               psections-construct       |
                               group-construct           |
                               parallel-pdo-construct    |
                               parallel-psections-construct |
                               single-process-construct
```

Contstraint: If the parallel-construct has a name prefix, then the it must have the same name as a suffix.

```
      data-sharing-spec is new-stmt |
                           use-stmt  |
                           type-declaration-stmt  |
                           specification-stmt     |
                           parameter-stmt         |
                           format-stmt            |
                           pointer-stmt
                            [data-sharing-spec]

      new-stmt is NEW variable-list
```

Constraint: specification-stmt shall not contain an access-stmt, common-stmt, data-stmt, optional-stmt, equivalence-stmt, derived-type-stmt, or save-stmt.

```
[name:]     PDO [(parallel-options)]
            parallel-body
      END PDO [name]
```

```
[name:]     PSECTION
            sections
      END PSECTIONS [name]
```

where

sections is [*sections section]*

section is SECTION [name] [WAIT (name-list)]

parallel-region

```
[name:]     PARALLEL PDO iter-specification parallel-option-list
            data-sharing-spec
            parallel-body
      END PARALLEL PDO [name]
```

[name:] PARALLEL PSECTIONS [*parallel-options]*
data-sharing-spec
sections
END PARALLEL PSECTIONS [name]

[name:] GROUP [(*group-option*)]
parallel-body
END GROUP [name]

where

group-option is NOWAIT

R503 *attr-spec* is PARAMETER
or *access-spec*
or ALLOCATABLE
or DIMENSION (array-spec)
or EXTERNAL
NEW or *guards-spec*
or INTENT (intent-spec)
or INTRINSIC
or OPTIONAL
or POINTER
or SAVE
or TARGET

X707 *guards-spec* is GUARDS (*guarded-obj-list*)

X708 *guarded-obj* is variable-name
or array-element
or array-section
or substring

CONSTRAINT: each subscript, substring, or section-subscript in a *guards-spec* must be an integer initialization expression (see Fortran 7.1.6.1)

X709 *critical-block* is *critical-stmt*
block
end-critical-stmt

X710 *critical-stmt* is CRITICAL SECTION [(*scalar-latch-variable*)] [*guards-spec*]

X711 *end-critical-stmt* is END CRITICAL SECTION [(*scalar-latch-variable*)]

CONSTRAINT: If the *end-critical-section-stmt* specifies a *scalar-latch-variable*, the corresponding *critical-section-stmt* shall specify the same *scalar-latch-variable*.

GUARDS (*guarded-list*) *sync-object*
or
GUARDS :: *sync-guards-list*
where *guarded* is variable-name,
array-name,
array-element,
array-section,

```
              module-name, or
              /common-block-name/ and
sync-guards-list is sync-object (guarded-list) [, sync-guards-list]
```

C.0 Lex/Yacc Syntax Rules (Informative)

The following is a simple Yacc grammar for recognizing X3H5 extensions for Fortran. This is an informative exercise to help keep the X3H5 grammar consistent and parsable by a simple parser.

It also might be a useful starting point for building real grammar rules for X3H5 Fortran extensions.

```
%{
#include <stdio.h>

%}

%union {
  char string[33];
}


%token PARALLEL MAX_PARALLEL WAIT GUARDS ORDERED NAME VARIABLE
%token SECTION BLOCK PARALLEL_PSECTIONS PSECTIONS PARALLEL_PDO
INTEGER
%token PDO INT_EXPR TYPE_STMTS END_PARALLEL END END_PDO END_PSECTIONS
%token CODE_BLOCK DO_VARIABLE PARALLEL_PDO END_PARALLEL_PDO
%token PARALLEL_SECTIONS END_PARALLEL_SECTIONS GROUP NOWAIT
%token PARALLEL_SPECIFICATION_PART CONTINUE

%type <string> NAME
%type <string> name
%type <string> INTEGER
%%
pgm             : blocks
                ;

blocks          : /* empty */
                | blocks block
                ;


block           : unnamed_p_block
                | named_p_block
                | code_block
                ;
unnamed_p_block : parallel_block
                | parallel_pdo
                | parallel_sections
                | pdo_block
```

```
                    | psection_block
                    | group_construct
                    ;
/* ---------------------------------------------------------------------- */
/*                                                                         */
/* Constraint: An unnamed_p_block shall not contain an exit, return,      */
/*               stop, or entry-statement.                                 */
/*                                                                         */
/* ---------------------------------------------------------------------- */

named_p_block    : name ':' unnamed_p_block name
                      {
                      if(strcmp($1,$4))
                          {
            printf("The starting and ending names of a block are different\n");
            printf("They are %s, %s\n",$1,$4);
                          }
                      }
                    ;
/* ---------------------------------------------------------------------- */
/*                                                                         */
/* Constraint: The name coded at the beginning of a named_p_block shall be   */
/*               the same as the name coded at the end of the named_p_block.   */
/*                                                                         */
/* ---------------------------------------------------------------------- */

parallel_block  : PARALLEL ptoption  blocks END_PARALLEL
                    ;

ptoption          : /* empty */
                    | poption
                    | parallel_specification_part
                    | poption parallel_specification_part
                    ;

parallel_specification_part : PARALLEL_SPECIFICATION_PART ;



/* ---------------------------------------------------------------------- */
/*                                                                         */
/* See ISO/IEC 1539:1991 (E) page 304 for Fortran 90 specifications.          */
/*                                                                         */
/* parallel_specification_part     : use_part decl_part                    */
/*                                   ;                                     */
```

```
/* use_part                  :  */ /* empty */
/*                          | use-stmt use_part                    */
/*                          ;
/* decl_part                 :  */ /* empty */
/*                          | declaration_construct decl_part        */
/*                          ;                                    */
/*                                                               */
/*                                                               */
/* Constraint: specification-stmt must not contain an access-stmt,          */
/*           allocatable-stmt(check with data section), common-stmt(check   */
/*           with data section), data-stmt, intent-stmt, optional-stmt,     */
/*           pointer-stmt (check with data section) or save-stmt.           */
/*           The decl_part shall not contain the entry_stmt, or             */
/*           stmt_function_stmt.                                           */
/* -------------------------------------------------------------------- */


poptions        : /* empty */
                | poption
                ;
poption         : '(' popt ')'
                ;
popt            : MAX_PARALLEL '=' INT_EXPR
                | ORDERED
                | ORDERED MAX_PARALLEL '=' INT_EXPR
                ;
psection_block  : PSECTIONS poptions
                    sections
                  END_PSECTIONS
                ;

sections        : section
                | sections section
                ;

section         : SECTION section_name wait_list guards_list
                  block
                ;
section_name    : /* empty */
                | '/' name '/'
                ;
wait_list       : /* empty */
                | WAIT '(' wlist ')'
                ;
wlist           : /* empty */
```

```
                   | wlist name
                   ;
guards_list      : /* empty */
                   | GUARDS '(' glist ')'
                   ;
glist            : /* empty */
                   | glist name
                   ;

pdo_block        : PDO iter_spec poptions blocks END_PDO
                   | PDO INTEGER iter_spec poptions blocks
                     INTEGER CONTINUE
                     {
                     if(strcmp($2,$6))
                        {
             printf("The starting and ending labels of a pdo block are different\n");
             printf("They are %s, %s\n",$2,$6);
                        }
                     }
                   ;

iter_spec        : do_variable '=' INT_EXPR ',' INT_EXPR ',' INT_EXPR
                   | do_variable '=' INT_EXPR ',' INT_EXPR
                   ;
/* ---------------------------------------------------------------------- */
/*                                                                        */
/* Constraint: The pdo-variable must be a named scalar variable of type   */
/*              integer and cannot be an element of a common block.       */
/*                                                                        */
/* ---------------------------------------------------------------------- */


group_construct : GROUP goption
                        blocks
                     END GROUP
                   ;
goption          : /* empty */
                   | NOWAIT
                   ;

/* ---------------------now provide for combined constructs------------ */

parallel_pdo     : PARALLEL_PDO iter_spec ptoption blocks END_PARALLEL_PDO
                   ;
parallel_sections : PARALLEL_SECTIONS ptoption
```

```
                    sections
                  END_PARALLEL_SECTIONS
                ;

/* -------------------here we provide stubs for various productions from the */
/*                   native language (Fortran 90)        -----------------*/
code_block      : CODE_BLOCK  ;
do_variable     : DO_VARIABLE ;
name            : NAME
                { strcpy($$,$1); }
;
%%
#include "lex.yy.c"
main()
{
if ( yyparse() )
   { printf("error in line number: %d\n", line);
     printf("Errors in this code\n");}
else
   printf("YIPPEE no errors\n");
}
```

Dummy Lexical Analizer for X3H5 Fortran

```
%{
int line;
%}
name     [a-zA-Z]+[a-zA-Z0-9_]*
integer [1-9][0-9]*
endline [\n]
blank   [ \t]+
%p 10000
%o 10000
%a 19000
%%
{endline}           line ++       ;
{blank}                           ;
PARALLEL                 {return (PARALLEL);}
CONTINUE                 {return (CONTINUE);}
END" "PARALLEL           {return (END_PARALLEL);}
PARALLEL_PDO             {return (PARALLEL_PDO);}
END_PARALLEL_PDO         {return (END_PARALLEL_PDO);}
PARALLEL_SECTIONS        {return (PARALLEL_SECTIONS);}
END_PARALLEL_SECTIONS    {return (END_PARALLEL_SECTIONS);}
PDO                      {return (PDO);}
WAIT                     {return (WAIT);}
GUARDS                   {return (GUARDS);}
ORDERED                  {return (ORDERED);}
MAX_PARALLEL             {return (MAX_PARALLEL);}
SECTION                  {return (SECTION);}
PSECTIONS                {return (PSECTIONS);}
BLOCK                    {return (BLOCK);}
END                      {return (END); }
GROUP                    {return (GROUP);}
NOWAIT                   {return (NOWAIT);}
CODE{blank}BLOCK         {return (CODE_BLOCK);}
END{blank}PDO            { /* printf("Found pdo\n");   */
                         return (END_PDO);}
END{blank}PSECTIONS      {return (END_PSECTIONS);}
INT_EXPR                 {return (INT_EXPR);}
VARIABLE                 {return (VARIABLE);}
DO_VARIABLE              {return (DO_VARIABLE);}
{integer}                {strcpy(yylval.string,yytext);
                         printf("Found integer %s\n",yytext);
                         return (INTEGER);}
INTEGER                  {return (INTEGER);}
PSPEC_PART               {return (PARALLEL_SPECIFICATION_PART);}
```

```
{name}                    {strcpy(yylval.string,yytext);
                           return (NAME);}
.                         {   /* printf("Lex got %c\n",yytext[0]); */
                           return (yytext[0]);}
%%
```

Index

B.0 Syntax Rules (Informative)