

X3H5
Parallel Extensions
for Fortran
April 2, 1993

Document Number
X3H5/93-SD2-Revision A

*Technical
Committee
X3H5/93-
S D 2 -
Revision A*

*for Parallel Computation-
Parallel Fortran-
Standard*

1	1.0 Introduction	8
2	1.1 Conceptual Model of Fortran Program Execution	8
3	1.2 Pseudo Code Form of the Conceptual Model	8
4	2.0 Standard Compliance	10
5	3.0 Terminology and Basic Concepts	11
6	4.0 Control Structures	12
7	4.1 Parallel Region Construct	12
8	4.1.1 Syntax for Parallel Regions Construct	12
9	4.1.2 Interpretation	12
10	4.2 Work Sharing Constructs	13
11	4.2.1 PDO Construct	13
12	4.2.1.1 Syntax for the PDO Construct	13
13	4.2.1.2 Coding Rules	13
14	4.2.1.3 Interpretation	13
15	4.2.1.4 PARALLEL PDO	14
16	4.2.1.4 Parallel PDO Syntax	14
17	4.2.1.6 Examples	14
18	4.2.2 PSECTION Construct	16
19	4.2.2.1 Syntax for the PSECTION Construct	17
20	4.2.2.2 Coding Rules for the PSECTION Construct	17
21	4.2.2.1 Interpretation	17
22	4.2.3 PARALLEL PSECTIONS Construct	19
23	4.2.3.1 Syntax	19
24	4.2.4 PDONE	19
25	4.2.4.1 Explicit Syntax	19
26	4.2.4.2 Coding Rules	19
27	4.2.4.3 Interpretation	20
28	4.2.4.4 Examples	20
29	4.3 GROUP Construct	20
30	4.3.1 Syntax	20
31	4.3.2 Coding Rules	20
32	4.3.3 Examples	21
33	4.4 Single Process Section	22
34	4.4.1 Explicit Syntax	22
35	4.4.2 Explicit Syntax	22
36	4.4.3. Interpretation	22
37	4.5 Inquiry Functions	23
38	4.5.1 Maximum performance improvement at this time	24
39	4.5.2 Team size	24
40	4.5.3 Looking for work	24

1	4.5.4 Blocked processes	24
2	4.5.5 Active processes	24
3	5.0 Data Environments	25
4	5.1 Terminology	25
5	5.1.1 The model terminology mapped to Fortran	25
6	5.1.1.1 Object	25
7	5.1.1.2 Read/Modify	25
8	5.1.1.3 Data environment	25
9	5.1.1.4 Private/Shared	26
10	5.1.2 Fortran terminology extended for the model:	26
11	5.1.2.1 Scoping Unit	26
12	5.1.2.2 Instance of a subprogram	26
13	5.1.3 New terminology for the binding	27
14	5.1.3.1 Iterative Control Variables	27
15	5.1.3.3 Hidden	27
16	5.2 Allowable Parallel Access Attribute	27
17	5.2.1 Definition of Instance Attribute	28
18	5.2.1.1 Instance Statement Syntax	29
19	5.3 Private/Shared Attribute	29
20	5.3.1 References through Pointers	30
21	5.4 Basic Mechanics	30
22	5.4.1 Types of Data Environments	30
23	5.4.1.1 Initial Data Environment	30
24	5.4.1.2 New Data Environment	31
25	5.4.1.3 Looking for Work Data Environment	31
26	5.4.2 Data Environments upon encountering a parallel construct	31
27	5.4.3 Object creation	31
28	5.4.4 Destroying Objects	32
29	5.4.5 Exiting parallel constructs	32
30	5.4.6 Early Departures of Team Members	33
31	5.5 Binding Considerations	33
32	5.5.1 APA and P/S Attributes with Fortran Scoping Rules	33
33	5.5.2 Data Environments and Lifetime of Fortran Objects	33
34	5.5.3 New Instances of Objects for Parallel Constructs	34
35	5.5.3.1 Syntax	34
36	5.5.3.2 Interpretation	34
37	5.5.3.3 New Statement	35
38	5.5.3.3.1 NEW Statement Syntax	35
39	5.5.3.4 Iterative Control Variables	35
40	5.5.4 Alternative APA Attributes for Always Shared	35
41	5.5.4 External Data Objects and Multiple Processes	36
42	5.5.5.1 Common and Modules	36
43	5.6 Objects and Synchronization	37

1	5.7 Examples	37
2	6.0 Input/Output	47
3	6.1 Multiple End-of-File Records	47
4	6.1.1 Explicit Syntax	48
5	6.2 Examples	48
6	7.0 Synchronization	50
7	7.1 Explicit Synchronization	50
8	7.1.1 Extensions Shared by Many Synchronization Methods	50
9	7.1.1.1 Representing States	50
10	7.1.1.2 Testing for Uninitialized State	51
11	7.1.1.3 SYNCHRONIZE Statement	51
12	7.1.1.3.1 Proposed X3H5 Extended Syntax Rule	51
13	7.1.1.3.2 Consistency Rules for the SYNCHRONIZE	
14	Statement	52
15	7.1.1.4 Representing Synchronization Operations	52
16	7.1.1.5 Use of Control Types and Assignment	53
17	7.1.2 Limiting Synchronization Overhead	53
18	7.1.2.1 Proposed X3H5 Extended Syntax Rule	53
19	7.1.2.2 GUARDS Attribute	53
20	7.1.3 Critical Sections	54
21	7.1.3.1 Proposed X3H5 Extended Syntax Rule	54
22	7.1.3.2 Consistency Rules for CRITICAL SECTION	54
23	7.1.3.3 Operations on Objects of TYPE (LATCH)	54
24	7.1.3.4 Default Latch	55
25	7.1.4 Locks	55
26	7.1.5 Events	56
27	7.1.6 Sequences	56
28	7.2 Explicit Synchronization	56
29	7.2.1 Critical Sections	57
30	7.2.1.1 Explicit Syntax	57
31	7.2.1.2 Coding Rules	58
32	7.2.1.3 Interpretation	58
33	7.2.1.4 Examples	58
34	7.2.2 Event Synchronization	63
35	7.2.2.1 Explicit Syntax	63
36	7.2.2.2 Coding Rules	63
37	7.2.2.3 Interpretation	63
38	7.2.2.4 Examples	64
39	7.2.2.5 Intrinsic Functions for Events	64
40	7.2.3 Sequences: Ordinal Synchronization	65
41	7.2.3.1 Explicit Syntax	66
42	7.2.3.2 Coding Rules	66

1	7.2.3.3 Interpretation	66
2	7.2.3.4 Examples	67
3	7.2.3.5 Intrinsic Functions for Ordinals	70
4	7.2.4 Unstructured synchronization - Locks	70
5	7.2.4.1 Explicit Syntax	70
6	7.2.4.2 Coding Rules	71
7	7.2.4.3 Interpretation	71
8	7.2.4.4 Examples	72
9	7.2.4.5 Intrinsic Functions for Locks	72
10	8.0 Nondeterministic Programs	75
11	A.0 X3H5 Directive Binding	76
12	A.1 Directives - Introduction	76
13	A.1.1 Role of the Directive Binding	76
14	A.1.2 Single Process Execution Requirement for Compliant Programs . . .	76
15	A.1.3 Synchronization and Serial Execution	77
16	A.1.3.2 Scoping at Parallel Constructs and Serial Execution	77
17	Alternate Intrinsic Functions	78
18	A.1.4 Terminology	78
19	A.1.5 Directives - General Usage Requirements in Parallel Programs	78
20	A.1.5.1 Continued Directive	78
21	A.1.6 Parallel Intrinsic Functions	78
22	A.1.6.1 Parallel Intrinsic Behavior for Equivalent Serial	
23	Execution	79
24	A.1.6.2 Functionality Not Supported Under Serial Interpretation . .	79
25	A.2 Syntax Rules	79
26	A.2.1 Parallel Do Construct	79
27	A.2.1.1 Syntax	79
28	A.2.1.2 Coding Rules	79
29	A.2.1.3 Examples	79
30	A.2.2 Parallel Sections Construct	80
31	A.2.2.1 Syntax	80
32	A.2.2.2 Interpretation	80
33	A.2.2.3 Examples	81
34	A.2.3 Synchronization Declarations	82
35	A.2.3.1 Syntax	82
36	A.2.3.2 Coding Rules	83
37	A.2.4 Unstructured Locking Synchronization	83
38	A.2.4.1 Syntax	83
39	A.2.4.2 Examples	83
40	A.2.4.2.1 Function Values for GATEs in Serial	
41	Execution	85
42	A.2.5 Critical Sections	85

1	A.2.5.1 Syntax	85
2	A.2.5.1 Examples	85
3	A.2.6 Event Synchronization	88
4	A.2.6.1 Syntax	88
5	A.2.6.1.1 Function Values for Events in Serial	
6	Execution	89
7	A.2.7 Ordinal (Sequence) Synchronization	89
8	A.2.7.1 Syntax	89
9	A.2.7.1.1 Function Values for Counters in Serial	
10	Execution	91
11	A.3 Data Sharing	91
12	A.3.1 Data Sharing Directives	91
13	A.3.1.1 Syntax	91
14	A.3.1.2 Rules	92
15	A.3.2 Partially Shared Common Blocks	92
16	A.3.2.1 Syntax	92
17	A.3.2.2 Rules	92
18	A.4 Parallel Region Construct	94
19	A.4.1 Syntax	94
20	A.4.2 Single Process Sections	98
21	A.4.2.1 Syntax	98
22	A.5 Exits from Parallel Constructs	99
23	A.5.1 Syntax	99
24	A.6 Extended Intrinsic	100
25	A.6.1 Parallel Intrinsic Functions	100
26	A.6.2 Definition of Serial Execution Library	100
27	B.0 Syntax Rules (Informative)	102
28	C.0 Lex/Yacc Syntax Rules (Informative)	105

1 **1.0 Introduction**

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

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

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

11 **1.1 Conceptual Model of Fortran Program Execution**

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

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

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

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

25 **1.2 Pseudo Code Form of the Conceptual Model**

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

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



```
1      ! iterative work is distributed among team members
2      ...
3  end pdo          ! end of worksharing construct
4      ...
5  group           ! beginning of group construct
6      ! replicated code here executed by all team members
7      ...
8      psection    ! beginning of worksharing construct
9      ...
10     psection
11     ...
12     end psection ! end of worksharing construct
13 end group       ! end of group construct
14 parallel do j=1,m,1 ! nested parallelism
15     ...
16     end parallel do ! end of nested parallelism
17 end parallel    ! end of parallel construct
18 ! serial program execution
19 ! possibly more parallel constructs
20     ...
21 end
```

1 **2.0 Standard Compliance**

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

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

1 **3.0 Terminology and Basic Concepts**

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

5 In describing the form of statements or constructs, or in explaining examples, the following
6 metalanguage conventions and symbols are used. These are similar to those defined by Fortran
7 90 (S8 Version 118, X3.198-1991 American National Standard Fortran 90, ISO/IEC 1539:1991)
8 on pages 3-5.

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

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

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

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

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

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

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

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

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

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

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

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

1 **4.0 Control Structures**

2 **4.1 Parallel Region Construct**

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

6 **4.1.1 Syntax for Parallel Regions Construct**

7 A *parallel-region-construct* is:

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

12 where

```
13             parallel-option is MAX PARALLEL = int-expr |  
14                                    ORDERED |  
15                                    MAX PARALLEL = int-expr, ORDERED |  
16                                    ORDERED, MAX PARALLEL = int-expr |  
17             parallel-body is statements |  
18                                    parallel-construct |  
19             parallel-construct is parallel-region-construct |  
20                                    pdo-construct |  
21                                    psections-construct |  
22                                    group-construct |  
23                                    parallel-pdo-construct |  
24                                    parallel-psections-construct |  
25                                    single-process-construct
```

26 Constraint: If the *parallel-construct* has a name prefix, then the it must have
27 the same name as a suffix.

28 **4.1.2 Interpretation**

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

32
33 If the MAX PARALLEL qualifier is not specified on the PARALLEL statement, then the number
34 of processes on this team is limited only by the maximum number of processes available to the
35 program. (See the intrinsic function NPSAVL)

36
37 If the MAX PARALLEL qualifier is specified on the PARALLEL statement, then the number
38 of processes on this team is limited by the *iexp1*.

1
2 All code inside a Parallel Region that is not enclosed by a worksharing construct shall be
3 redundantly executed by all of the processes on the team.
4

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

9 **4.2 Work Sharing Constructs**

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

18 **4.2.1 PDO Construct**

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

20 **4.2.1.1 Syntax for the PDO Construct**

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

24 **4.2.1.2 Coding Rules**

25 **4.2.1.3 Interpretation**

26 If the MAX PARALLEL qualifier is not specified on a PDO or PSECTIONS statement, then the
27 number of processes on this team that may enter the worksharing construct is limited only by the
28 number of processes on the team. (See the intrinsic function NPSTM (what is the new name for
29 NPSTM?))
30

31 If the MAX PARALLEL qualifier is specified on PDO or PSECTIONS statement, then the
32 number of processes on this team that may enter the worksharing construct is limited by the
33 iexp2.
34

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

- 4 1. for each unit of parallel work to be executed:
 - 5 a. assign the appropriate value to its index variable
 - 6 b. execute the iterative portion
 - 7 c. if EXTEND is specified, execute the statements up to the END .*
8 EXTEND statement
- 9 2. make all shared objects updated by this process within the Pdo and the group
10 block available to all processes
- 11 3. wait for all processes that participated in executing the Pdo to complete step 2)

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

17 **4.2.1.4 PARALLEL PDO**

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

20 PARALLEL
21 PDO

22 **4.2.1.4 Parallel PDO Syntax**

23 The syntax for the PARALLEL PDO is:

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

28 **4.2.1.6 Examples**

```
29 Example SUBROUTINE EX48 (A,B,C,N)  
30 REAL A(N),B(N),C(N)  
31 PARALLEL PDO I=1,N-1  
32 NEW T  
33 T = A(I)*B(I)  
34 C(I+1) = T * (T-1.0)  
35 END PARALLEL PDO  
36 END  
37
```

```

1      Example          SUBROUTINE EX49 (A,B,C,N)
2                          REAL A(N),B(N),C(N)
3                          PARALLEL
4                              NEW T
5                              PDO I=1,N-1
6                                  T = A(I)*B(I)
7                                  C(I+1) = T * (T-1.0)
8                              END PDO
9                          END PARALLEL
10                         END

```

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

```

14
15      Example 50      SUBROUTINE EX50 (ZA,ZB,ZC,ZD,N)
16                          REAL ZA(N),ZB(N),ZC(N),ZD(N)
17                          PARALLEL SECTIONS
18                              NEW T
19                              SECTION
20                                  DO 10 I=1,N
21                                      T = ZFUNC(ZA(I))
22                                      ZC(I) = T * T
23                                  END DO
24                              SECTION
25                                  DO 20 I=1,N
26                                      T = ZFUNC(ZB(I)-ZA(I))
27                                      ZD(I) = T * T
28                                  END DO
29                              END PARALLEL SECTIONS
30                              END

```

```

31
32      Example 51      SUBROUTINE EX51 (ZA,ZB,ZC,ZD,N)
33                          REAL ZA(N),ZB(N),ZC(N),ZD(N)
34                          PARALLEL
35                              NEW T
36                              PSECTIONS
37                                  SECTION
38                                      DO 10 I=1,N
39                                          T = ZFUNC(ZA(I))
40                                          ZC(I) = T * T
41                                      END DO
42                                  SECTION
43                                      NEW T
44                                      DO 20 I=1,N
45                                          T = ZFUNC(ZB(I)-ZA(I))
46                                          ZD(I) = T * T
47                                      END DO
48                                  END PSECTIONS
49                              END PARALLEL
50                              END

```

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

```

54
55      Example 52      SUBROUTINE EX52 (A)
56                          REAL A(*)

```

```

1      GETLOCK B
2      GUARDS B(SUM)
3      UNLOCK(B)
4      SUM=0.0
5      PARALLEL
6          NEW SUML
7          SUML = 0.0
8          GROUP
9              PDO          I=1,N
10             SUML = SUML + A(I)
11         END PDO
12         CRITICAL SECTION (B)
13             SUM = SUM + SUML
14         END CRITICAL SECTION (B)
15     END GROUP
16 END PARALLEL
17 END

```

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

```

23
24     Example 53      SUBROUTINE EX53 (A,B,C,D,N,M)
25                       REAL A(N),B(N),C(N),D(N)
26                       PARALLEL
27                           PDO I=1,N
28                               A(I) = B(I) * C(I)
29                           END PDO
30                           PDO I=1,M
31                               D(I) = A(I) - C(I)
32                           END PDO
33                       END PARALLEL
34                       END

```

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

```

39
40     Example 54      SUBROUTINE EX54 (A,C,N,M)
41                       REAL A(N,0:M),C(N,M)
42                       PARALLEL
43                           DO 10 J=1,M
44                               PDO I=1,N
45                                   A(I,J) = C(I,J)/A(I,J-1)
46                               END PDO
47                           10  END DO
48                       END PARALLEL
49                       END
50

```

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

1 **4.2.2 PSECTION Construct**

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

3 **4.2.2.1 Syntax for the PSECTION Construct**

```
4    [name:]       PSECTION  
5                    sections  
6                    END PSECTIONS [name]
```

7 where

```
8                    sections is [sections section]  
9            section is SECTION [name] [WAIT (name-list)]  
10                    parallel-region
```

11 **4.2.2.2 Coding Rules for the PSECTION Construct**

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

16
17 The identifier used for a *section-name* is a seventh class of local names in the sense of Fortran
18 page 18-2. This means that

19 A *section-name* must be unique within a program unit (ISO/IEC 1539:1991 Section 2.2)

20
21 *Section-names* share the single name space already shared by array, variable, constant,
22 statement function, intrinsic function, and dummy procedure names

23
24 In a standard conforming program the WAIT clause shall only reference the *section-name* of a
25 lexically preceding SECTION statement of the same Parallel Sections construct.

26
27

28 **4.2.2.1 Interpretation**

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

32
33 In a standard conforming program the sections of code shall be data independent, except where
34 appropriate synchronization mechanisms are used.

35
36 A *section-name* is a label with no programmer-visible storage association.

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

- 3 1. for each unit of parallel work to be executed:
 - 4 a. if a WAIT clause is coded for this section, then wait until the sections
5 indicated by the WAIT clause have completed execution
 - 6 b. execute the corresponding section of code
- 7 2. if the EXTEND qualifier is specified, execute the statements up to the END
8 EXTEND statement
- 9 3. make all shared objects updated by this process within the Psections construct
10 available to all processes
- 11 4. wait for all processes that participated in executing the Psections construct to
12 arrive at step 2)

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

17
18 If the MAX PARALLEL qualifier is specified on PDO or PSECTIONS statement, then the
19 number of processes on this team that may enter the worksharing construct is limited by the
20 iexp2.

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

24
25 The WAIT clause specifies a partial ordering among the sections of code. All sections whose
26 names are listed as *section-names* in the WAIT clause of a section must complete before that
27 section can begin. The WAIT clause does not require use of the ORDERED qualifier.

28
29 The GUARDS clause shall only be specified on the SECTION statement if the WAIT clause is
30 specified. The GUARDS clause explicitly identifies the names of objects that shall be made
31 consistent for the process executing the waiting section.

32
33 The GUARDS clause explicitly identifies the objects that must be made consistent and removes
34 a requirement for an implementation to make any other objects consistent at the point it is
35 specified.

36

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

5
6 If the ORDERED qualifier is specified, then synchronization mechanisms may be used that
7 require some portion of an earlier (in lexical order) section to complete execution before some
8 portion of a later section begins execution. While use of the ORDERED qualifier in a Parallel
9 Sections construct that does not contain synchronization is standard conforming, it may incur a
10 performance penalty on some implementations.

11
12 If the MAX PARALLEL qualifier is not specified, then the number of processes on this team is
13 limited only by the number of Sections defined or the maximum number of processes available
14 to the program. If the MAX PARALLEL qualifier is specified, then the number of processes on
15 this team must be greater than zero and less than or equal to *int-exp*. Any lexically contained
16 do loop index variables are treated as newly scoped objects for the parallel section. They inherit
17 the same type as the objects of the same name outside of the parallel section. They have the
18 automatic storage class and have no storage associations thru equivalence classes or common
19 blocks.

20 There is an implicit synchronization at the end of a Parallel Sections construct.

21 **4.2.3 PARALLEL PSECTIONS Construct**

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

24 **4.2.3.1 Syntax**

```
25 [name:]    PARALLEL PSECTIONS [parallel-options]  
26           data-sharing-spec  
27           sections  
28           END PARALLEL PSECTIONS [name]
```

29 **4.2.4 PDONE**

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

32 **4.2.4.1 Explicit Syntax**

```
33         PDONE
```

34 **4.2.4.2 Coding Rules**

1 The PDONE statement is an executable statement.

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

4 **4.2.4.3 Interpretation**

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

11 **4.2.4.4 Examples**

```
12 Subroutine EX58(x,y)
13 Double precision x(100),y(100)
14 parallel do i=1,100
15   if (y(i) .eq. 0.0D0) then
16     print*,i
17     pdone
18     cycle
19   endif
20   x(i)=1.0/y(i)
21 end parallel do
22 return
23 end
```

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

29 **4.3 GROUP Construct**

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

32 **4.3.1 Syntax**

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

1 where

2 *group-option* is NOWAIT

3 4.3.2 Coding Rules

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

7 4.3.3 Examples

8

```
9 Example 52 SUBROUTINE EX52 (A)
10 REAL A(*)
11 GETLOCK B
12 GUARDS B(SUM)
13 UNLOCK(B)
14 SUM=0.0
15 PARALLEL
16 NEW SUML
17 SUML = 0.0
18 GROUP
19 PDO I=1,N
20 SUML = SUML + A(I)
21 END PDO
22 CRITICAL SECTION (B)
23 SUM = SUM + SUML
24 END CRITICAL SECTION (B)
25 END GROUP
26 END PARALLEL
27 END
```

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

33

```
34 Example 53 SUBROUTINE EX53 (A,B,C,D,N,M)
35 REAL A(N),B(N),C(N),D(N)
36 PARALLEL
37 PDO I=1,N
38 A(I) = B(I) * C(I)
39 END PDO
40 PDO I=1,M
41 D(I) = A(I) - C(I)
42 END PDO
43 END PARALLEL
44 END
```

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

```

1
2      Example 54          SUBROUTINE EX54 (A,C,N,M)
3                          REAL A(N,0:M),C(N,M)
4                          PARALLEL
5                              DO 10 J=1,M
6                                  PDO I=1,N
7                                      A(I,J) = C(I,J)/A(I,J-1)
8                                          END PDO
9      10                      END DO
10                             END PARALLEL
11                             END
12

```

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

15 4.4 Single Process Section

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

20 4.4.1 Explicit Syntax

21 Statement Forms

```

22          SINGLE PROCESS
23
24          END SINGLE PROCESS
25

```

26 Structured As

```

27          SINGLE PROCESS
28              statements
29          END SINGLE PROCESS

```

30 4.4.2 Explicit Syntax

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

32 4.4.3. Interpretation

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

```

36      Example 55          SUBROUTINE EX55 (A,B,N)
37                          REAL A(N),B(N)
38                          PARALLEL
39                              PDO I=1,N
40                                  A(I) = 1.0 / A(I)
41                                          END PDO
42                          SINGLE PROCESS
43                              IF ( A(1) .GT. 1.0 ) A(1) = 1.0
44                          END SINGLE PROCESS

```

```

1          PDO I=1,N
2            B(I) = B(I) / A(1)
3          END PDO
4        END PARALLEL
5      END

6      Example 56          SUBROUTINE EX56 (A,B,N)
7                          REAL A(N),B(N)
8                          PARALLEL
9                            PDO I=1,N
10                           A(I) = 1.0 / A(I)
11                          END PDO
12                          PSECTIONS
13                          SECTION
14                            IF ( A(1) .GT. 1.0 ) A(1) = 1.0
15                          END PSECTIONS
16                          PDO I=1,N
17                            B(I) = B(I) / A(1)
18                          END PDO
19                          END PARALLEL
20      END

```

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

```

24      Example 57          SUBROUTINE EX57 (A,AMAX,N)
25                          REAL A(0:N)
26                          AMAX = 0.0
27                          PARALLEL
28                            NEW ALMAX
29                            BEGIN GROUP
30                              PDO I=1,N
31                                IF ( ABS(A(I)) .GT. ABS(ALMAX) ) ALMAX = A(I)
32                              END PDO
33                              CRITICAL SECTION
34                                IF ( ABS(ALMAX) .GT. ABS(AMAX) ) AMAX = ALMAX
35                              END CRITICAL SECTION
36                            END GROUP
37                          SINGLE PROCESS
38                            ALMAX = A(1)+A(N)
39                            IF ( AMAX .LT. ALMAX ) AMAX = 1.0 + AMAX
40                          END SINGLE PROCESS
41                          PDO I=1,N
42                            A(I) = ABS( A(I) / AMAX )
43                          END PDO
44                          END PARALLEL
45      END
46
47

```

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

1 **4.5 Inquiry Functions**

2 The following intrinsic functions shall be provided:

3 **4.5.1 Maximum performance improvement at this time**

4 DOUBLE PRECISION FUNCTION PERFMAX()

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

8 **4.5.2 Team size**

9 INTEGER FUNCTION NPTEAM()

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

12 **4.5.3 Looking for work**

13 INTEGER FUNCTION NPLOOK()

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

16 **4.5.4 Blocked processes**

17 INTEGER FUNCTION NPBLOCK()

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

20 **4.5.5 Active processes**

21 INTEGER FUNCTION NPACTIVE()

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

1 **5.0 Data Environments**

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

3 **5.1 Terminology**

4 **5.1.1 The model terminology mapped to Fortran**

5 **5.1.1.1 Object**

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

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

10 **5.1.1.2 Read/Modify**

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

13 An *object* or a *subobject of the object* is *modified* as described by *the model* when it is used in
14 a way that causes it to *become defined* as described by *Fortran 90*⁴. A *Fortran constant* cannot
15 be modified⁵.

16 **5.1.1.3 Data environment**

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

20 ²Fortran common block Section 5.5.2, page 58, line 18 of
21 Fortran 90.

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

24 ⁴defines Section 14.7.5, page 250, lines 4-10.

25 ⁵Fortran constant Section 6, page 61, line 37, 38.

1 A *data environment* as described by the *the model* is a collection of *objects* as defined in
2 section 5.1.1.1. (*Data enviroment* as used in this document is distiguished from *data environment*
3 as used in Fortran 90⁶ by the inclusion of common blocks.)

4 **5.1.1.4 Private/Shared**

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

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

10 **5.1.2 Fortran terminology extended for the model:**

11 **5.1.2.1 Scoping Unit**

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

14 **5.1.2.2 Instance of a subprogram**

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

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

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

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

29 ⁶Section 2.4, Data Concepts, page 13, line 2.

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

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

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

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

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

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

15 **5.1.3 New terminology for the binding**

16 **5.1.3.1 Iterative Control Variables**

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

19 **5.1.3.3 Hidden**

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

22 ⁸Section 12.5.2.4, Instances of a 5.1.2.3 Save Attribute.

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

24 ¹⁰Section 5.1, page 41, lines 9-12. Section 5.2.9, page 52,
25 lines 1-3.

26 ¹¹Section 8.1.4.1.1, page 100, line 37.

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

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

1 **5.2 Allowable Parallel Access Attribute**

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

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

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

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

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

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

19 **5.2.1 Definition of Instance Attribute**

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

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

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

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

28 ¹⁵Section 2.2, page 9, line 45-49.

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

30 ¹⁷Section 2.2, page 9, lines 45-49.

- 1 All objects specified in a module program unit shall have the same instance attribute.
- 2 The default instance attribute for COMMON blocks shall be single.
- 3 Blank common shall only have an instance attribute of single.
- 4 The default instance attribute for modules shall be single.
- 5 A global object with an instance attribute of single shall have an APA attribute of "always shared".
- 6 A global object with an instance attribute of parallel shall have an APA attribute of "default
7 private, explicitly shared".
- 8 A common block with a parallel instance attribute may have the save attribute. If it has the save
9 attribute, it shall have the same lifetime as its data environment.
- 10 A common block with the parallel instance attribute may be initialized by a block data program.
11 This shall occur once per process.

12 **5.2.1.1 Instance Statement Syntax**

13 INSTANCE (single or parallel)
14 or
15 INSTANCE (single or parallel) list_of_common_block_names
16 or
17 INSTANCE (single or parallel) module_name

- 18 An instance statement shall appear in the specification statements of a program unit.
- 19 If an INSTANCE statement occurs in a program unit without any names specified, then it shall
20 define the instance attribute for all global objects in that program unit.
- 21 If an INSTANCE statement occurs in a module program unit, it shall specify only the name of
22 the containing module program unit.
- 23 If an INSTANCE statement occurs in a main, subroutine or function, or block data program unit,
24 it shall specify only names of common blocks defined within the program unit.

25 **5.3 Private/Shared Attribute**

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

- 1 - All **iterative control variables** contained within the parallel construct shall have a *P/S attribute* of *private* with respect to the parallel construct.
2
- 3 - All *objects* that are *host associated* with a containing *scoping unit* shall have a *P/S attribute*
4 of *shared* with respect to the parallel construct.
- 5 - All *common blocks* and *objects* contained in *common blocks* shall have a *P/S attribute* of
6 *shared* with respect to the parallel construct.
- 7 - All *objects* that are declared within the *scope* of the parallel construct shall have a *P/S*
8 *attribute* of *private* with respect to the parallel construct.
- 9 - All other *objects* shall have a *P/S attribute* of *private* with respect to the parallel construct.

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

12 **5.3.1 References through Pointers**

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

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

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

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

25 **5.4 Basic Mechanics**

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

27 **5.4.1 Types of Data Environments**

28 **5.4.1.1 Initial Data Environment**

29 The *initial data environment* for a parallel Fortran program shall begin with a *new data*
30 *environment*. In addition, the *initial data environment* contains all *common blocks and modules*

1 for the Fortran program. During program execution, the *initial data environment* may contain
2 additional *objects* that come into *scope* during program execution. *Objects* that come into *scope*
3 during execution of parallel constructs shall not be part of the *initial data environment* unless
4 the initial process is participating in the execution of the parallel construct as a base process and
5 it encounters the scoping unit.

6 **5.4.1.2 New Data Environment**

7 A *new data environment* shall consist of *objects* with the *save attribute* (also referred to in
8 *Fortran 90* as *saved objects*).¹⁸ The *objects* that are *initially defined*¹⁹ as described in *Fortran*
9 *90* shall have their initial values defined.

10 **5.4.1.3 Looking for Work Data Environment**

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

14 **5.4.2 Data Environments upon encountering a parallel construct**

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

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

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

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

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

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

28 ¹⁹Section 14.7.3, Variables that are initially defined, page
29 249, lines 35-39.

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

1 **5.4.3 Object creation**

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

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

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

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

12 **5.4.4 Destroying Objects**

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

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

20 **5.4.5 Exiting parallel constructs**

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

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

26 ²¹Section 5.2.9, page 52, lines 1-3.

27 ²²Section 6.3.3.1, Deallocation of allocatable arrays, page 69,
28 lines 2-15.

29 ²³Section 6.3.3.1, Deallocation of allocatable arrays, page 69,
30 lines 2-15.

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

5 **5.4.6 Early Departures of Team Members**

6 **5.5 Binding Considerations**

7 **5.5.1 APA and P/S Attributes with Fortran Scoping Rules**

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

- 11 - *global entities*
- 12 - *always shared*
- 13 - *default private, explicitly shared*
- 14 - *local entities*
- 15 - *default private, explicitly shared*
- 16 - *statement entities*
- 17 - *default private, explicitly shared*

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

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

21 **5.5.2 Data Environments and Lifetime of Fortran Objects**

22 ²⁴Section 14, Scope, association and definition., page 241.

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

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

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

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

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

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

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

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

11 5.5.3 New Instances of Objects for Parallel Constructs

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

14 The binding allows the following specifications within a parallel constructs:

15 5.5.3.1 Syntax

```
16     data-sharing-spec is new-stmt |  
17                       use-stmt |  
18                       type-declaration-stmt |  
19                       specification-stmt |  
20                       parameter-stmt |  
21                       format-stmt |  
22                       pointer-stmt |  
23                       [data-sharing-spec]
```

24 new-stmt is NEW variable-list

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

27 5.5.3.2 Interpretation

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

30 - *type*

31 ²⁷Section 14.6, Association, page 245-247.

- 1 - *dimension*
- 2 - *allocatable*
- 3 - *pointer*
- 4 - *target*

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

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

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

15 **5.5.3.3 New Statement**

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

18 **5.5.3.3.1 NEW Statement Syntax**

19 NEW *external_name_list*

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

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

24 **5.5.3.4 Iterative Control Variables**

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

31 **5.5.4 Alternative APA Attributes for Always Shared**

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

4 **5.5.4 External Data Objects and Multiple Processes**

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

8 Additional rules with respect to new language features:

9 **5.5.5.1 Common and Modules**

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

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

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

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

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

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

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

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

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

3 **5.6 Objects and Synchronization**

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

5 - *read*

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

7 - *modified*

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

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

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

15 **5.7 Examples**

```
16 Subroutine EXD01(A,B,C,N)  
17 Real A(n),B(n),C(n)  
18 parallel do i=1,n  
19 Real t  
20 t=a(i)*b(i)  
21 c(i+1)=t* (t-1.0)  
22 end parallel do  
23 end  
24
```

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

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

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

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

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

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

```

4      Subroutine EXD02(B)
5      Real, Dimension(100) :: B,C
6      parallel do i=1,100
7          call subx1(b(i))
8          call subx2(c(i))
9      end parallel do
10     print*, (c(i),i=1,100)
11     end

12     subroutine subx1(x)
13     real, save:: a
14     a=x
15     return
16     entry subx2(x)
17     x=a
18     end
  
```

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

```

24     Subroutine EXD03()
25     Real, Dimension(100) :: B
26     common /abc/ b
27     call subx1(100)
28     print*, (b(i),i=1,100)
29     end

30     subroutine subx1(icnt)
31     parallel do i=1,icnt
32         call work(i)
33     end parallel do
34     return
35     end

36     subroutine work(i)
37     Real, Dimension(100) :: B
38     common /abc/ b
39     b(i)=i
40     return
41     end
  
```

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

```

45     subroutine EXD04(in,A)
46     real, dimension(in,in):: A
47     real, dimension(:,,:), allocatable:: B,E
48     allocate B(in,in)
  
```

```

1      parallel do i=1,in
2          real, dimension(:), allocatable:: C
3          Allocate C(in)
4          C(:)=0
5          parallel do j=1,in
6              c(j)=c(j)+A(i,j)
7              if (fn(c(j)).neq.0) then
8                  Critical section
9                      if (.not.allocated(E)) then
10                         allocate E(in,in)
11                         endif
12                     end critical section
13                     E(i,j)=C(j)
14                 endif
15             end parallel do
16             B(i,:)=C(i)
17             deallocate C
18         end do
19         A(:, :)=B(icnt:1:-1, :)
20         return
21     end

```

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

```

27     subroutine EXD06(in)
28     integer pi(in), i(in)
29     pointer pi
30     target i
31     allocate I
32     PI=>I
33     icnt=0
34     parallel
35     integer pj(in), j(in) ,id
36     pointer pj
37     target j
38     critical section
39         icnt=icnt+1
40         id=icnt
41     end critical section
42     if(id .eq.1) then
43         PJ=>I
44     else
45         allocate J
46         PJ=>J
47     endif
48     pdo i=1,100
49         ...
50     end pdo
51     if(id .gt.1) then
52         deallocate j
53     endif
54     end parallel

```

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

```
7      subroutine exd1()  
8      common/abc/a(100),b(100)  
9      common/def/d(100),e(100)  
10     common/ghi/g(100),h(100)  
11     instance parallel /def/,/ghi/  
12     parallel do i=1,100  
13         new/def/  
14         ...  
15     end parallel do  
  
16     subroutine exd2()  
17     common/abc/a(100),b(100)  
18     common/def/d(100),e(100)  
19     common/ghi/g(100),h(100)  
20     instance parallel /def/,/ghi/  
21     instance single /abc/  
22     parallel do i=1,100  
23         new/def/d(100),e(100)  
24         ...  
25     end parallel do
```

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

```
33     module data  
34     dimension a(100),b(100)  
35     real a,b  
36     private b  
37     public a  
38     end data
```

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

```
43     module exd7
```



```

1      instance parallel
2      dimension a(100),b(100)
3      real a,b
4      end exd7

```

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

```

8      module data
9      instance parallel
10     dimension a(100),b(100)
11     real a,b
12     common/abc/a,b
13     subroutine x()
14     instance parallel
15     common/abc/a,b
16     dimension y(10)
17     ...
18     end
19     end data

```

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

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

```

25     Example 33      LOGICAL FUNCTION EX33 (A, IZERO, N)
26                   REAL A(N)
27                   PARALLEL PDO I=1, N
28                   IF ( A(N) .EQ. 0.0 ) THEN
29                   CRITICAL SECTION
30                   IZERO = I
31                   END CRITICAL SECTION
32                   EX33 = .TRUE.
33                   PDONE
34                   ENDIF
35                   END PARALLEL PDO
36                   EX33 = .FALSE.
37                   END

```

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

```

1      Example 41      SUBROUTINE EX41 (B)
2                      REAL B(100)
3                      PARALLEL PDO I=1,100
4                        CALL SUB(B(I))
5                      END PARALLEL PDO
6                      END

7                      SUBROUTINE SUB (X)
8                      INSTANCE PARALLEL
9                      COMMON /BLOCKA/ A
10                     A = X
11                     CALL SQUARE
12                     X = A
13                     END

14                     SUBROUTINE SQUARE
15                     INSTANCE PARALLEL
16                     COMMON /BLOCKA/ A
17                     A = A*A
18                     END

```

```

19     Example 42     SUBROUTINE EX42 (B)
20                   INSTANCE PARALLEL
21                   COMMON /BLOCKA/ A
22                   REAL B(100)
23                   PARALLEL PDO I=1,100
24                     NEW /BLOCKA/
25                     CALL SUB(B(I))
26                   END PARALLEL PDO
27                   END

28                   SUBROUTINE SUB (X)
29                   INSTANCE PARALLEL
30                   COMMON /BLOCKA/ A
31                   A = X
32                   CALL SQUARE
33                   END

34                   SUBROUTINE SQUARE
35                   INSTANCE PARALLEL
36                   COMMON /BLOCKA/ A
37                   A = A*A
38                   END

```

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

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

```

44     Example 43:     C This example is NON-STANDARD CONFORMING C
45                   INSTANCE PARALLEL /NC/
46                   COMMON /NC/ A(100)
47                   ...

```

```

1      y_calls:      PARALLEL PDO I=1,100
2                    ...
3                    CALL Y
4                    10  END DO y_calls
5                    ...
6                    RETURN
7                    END

8                    SUBROUTINE Y
9                    ...
10                   PARALLEL PDO J=1,100
11                   ...
12                   CALL Z
13                   END DO
14                   ...
15                   RETURN
16                   END

17                   SUBROUTINE Z
18                   INSTANCE PARALLEL /NC/
19                   COMMON /NC/ A(100)
20                   ...
21                   RETURN
22                   END

```

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

26 Possible modifications to make it standard conforming include:

- 27 1. Specify /NC/ on a NEW statement in the parallel y_calls loop
28 in the main program.
- 29 2. Include the COMMON statement defining /NC/ in subroutine Y.
30 Then /NC/ will be shared for all parallel constructs.
- 31 3. Include the COMMON statement defining /NC/ in subroutine Y and
32 specify /NC/ on a NEW statement for the parallel do loop.

```

33      Example 45      SUBROUTINE EX45 (B)
34                    REAL B(100), C(100)
35                    PARALLEL PDO I=1,100
36                    CALL SUB1(B(I))
37                    CALL SUB2(C(I))
38                    END PARALLEL PDO
39                    PRINT *, (C(I), I = 1, 100)
40                    END

41                    SUBROUTINE SUB1 (X)
42                    INSTANCE PARALLEL
43                    COMMON /BLOCKA/ A

```

```

1      SAVE /BLOCKA/
2      A= X
3      END
4
5      SUBROUTINE SUB2 (X)
6      INSTANCE PARALLEL
7      COMMON /BLOCKA/ A
8      SAVE /BLOCKA/
9      X = A
10     END

```

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

```

15     Example 46      SUBROUTINE EX46 (B)
16                     REAL B(100), C(100)
17                     INSTANCE PARALLEL /BLOCKA/
18                     COMMON /BLOCKA/ A
19                     PARALLEL PDO I=1,100
20                       NEW /BLOCKA/
21                       CALL SUB1(B(I))
22                       CALL SUB2(C(I))
23                     END PARALLEL PDO
24                     PRINT *, (C(I), I = 1, 100)
25                     END
26
27                     SUBROUTINE SUB1 (X)
28                     INSTANCE PARALLEL /BLOCKA/
29                     COMMON /BLOCKA/ A
30                     A = X
31                     END
32
33                     SUBROUTINE SUB2 (X)
34                     INSTANCE PARALLEL /BLOCKA/
35                     COMMON /BLOCKA/ A
36                     X = A
37                     END

```

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

```

39     Example 39      SUBROUTINE EX39 (B,C,N)
40                     REAL B(N),C(N)
41                     PARALLEL PDO I=1,N
42                       REAL A
43                       A=B(I)+C(I)
44                       CALL EX39A(A,B,I)
45                     END PARALLEL PDO
46                     END
47
48                     SUBROUTINE EX39A (AA,BB,N)

```

```

1      REAL BB(N),BX
2      DATA BX/1.0/
3      BX= AA * (AA-4.0)/BX
4      PARALLEL PDO J=1,N
5          BB(J) = BB(J)*BX
6      END PARALLEL PDO
7      END

```

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

```

14      Example ??      PROGRAM MAIN
15                      COMMON/COM1/CA(100)
16                      INTEGER LA,MS,ND
17                      DATA /ND,1/
18                      ...
19                      SAVE /COM1/,MS
20                      ...
21                      PARALLEL PDO I=1,100
22                          NEW LA,MS,ND
23                          ...
24                          CALL Y
25                          ...
26                      END Parallel DO
27                      ...
28                      END

29                      BLOCK DATA X
30                      COMMON/COM1/CA(100)
31                      INSTANCE PARALLEL /SCOM1/
32                      COMMON/SCOM1/ SC(100)
33                      DATA /CA,100*0.0/,/SC,100*0.0/
34                      END

35                      SUBROUTINE Y
36                      COMMON/COM1/CA(100)
37                      COMMON/COM2/CB(100)
38                      INSTANCE PARALLEL /SCOM1/,/SCOM2/
39                      COMMON/SCOM1/ SC(100)
40                      COMMON/SCOM2/ SD(100)
41                      INTEGER IS(100),JA(100),KD
42                      DATA KD/0/
43                      SAVE /COM1/,/SCOM1/,IS
44                      ...
45                      END

```

```

46      EXPLANATION
47          COMMONs:
48          COM1 is single_copy_external, static storage
49          COM2 is single_copy_external, dynamic storage

50          SCOM1 is parallel_external, static storage
51          SCOM2 is parallel_external, automatic storage

```

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

5      NEW variables:
6          LA' is construct_local, automatic storage
7          MS' is construct_local, ?? (auto or static)
8          ND' is construct_local, ??
```

1 **6.0 Input/Output**

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

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

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

11 Insert the following step between steps 2 and 3:
12 (2.5) Obtain an implementation lock associated with the unit

13 Insert the following step between steps 7 and 8:
14 (7.5) Free the implementation lock obtained for the unit

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

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

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

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

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

30 **6.1 Multiple End-of-File Records**

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

6 **6.1.1 Explicit Syntax**

7 SKIP PAST EOF just-like-backspace-both forms

8 **6.2 Examples**

9 **6.2.1**

```
10 subroutine exio1()  
11 dimension a(100)  
12 parallel sections 10 i=1,n  
13 section /a/  
14 read (*,7) n  
15 ...  
16 section /b/  
17 ...  
18 section /c/ wait(a)  
19 if (n.gt.100) print*,'error', n  
20 read (*,7) a(1:n)  
21 ...  
22 end parallel sections
```

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

26 **6.2.2**

```
27 subroutine exio2()  
28 dimension a(100)  
29 parallel sections 10 i=1,n  
30 section /a/  
31 i=6  
32 write(*,6) f1(i)  
33 ...  
34 section /b/  
35 i=8  
36 write(*,8) f1(i)  
37 ...  
38 end parallel sections  
39 return  
40 end  
41 function f1(i)  
42 ...  
43 read (*,i+1) ...  
44 ...  
45 return  
46 end
```


1 In example EXIO2, the user is responsible for ensuring that there is no synchronization required
2 between I/O to units; or for providing the necessary synchronization. The example as written is
3 correct since the process executing section A will write to units 6 and 7; while the process
4 executing section B will write to units 8 and 9. However, if function f1 tried to read from unit
5 8 when i=6 and to read from unit 6 when i=8 there would be a chance of deadlock. To prevent
6 the deadlock, the user would have to use explicit synchronization to ensure that only one process
7 was executing the write statements in sections a and b.

1 **7.0 Synchronization**

2 Implicit synchronization occurs at the following statements:

3 PARALLEL
4 END PARALLEL
5 END PARALLEL PDO
6 END PDO (WAIT)
7 PARALLEL SECTIONS
8 END PARALLEL SECTIONS
9 END PSECTIONS (WAIT)
10 END PGROUP

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

12 **7.1 Explicit Synchronization**

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

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

17 TYPE (LATCH) for latch
18 TYPE (LOCK) for lock
19 TYPE (EVENT) for event
20 TYPE (ORDINAL) for sequence

21 ***** Aside to X3J3 *****

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

25 **7.1.1 Extensions Shared by Many Synchronization Methods**

26 **7.1.1.1 Representing States**

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

32 for TYPE (LATCH):
33 STATE_UNINITIALIZED for state uninitialized
34 STATE_UNLATCHED for state unlatched

1 STATE_LATCHED for state latched

2 for TYPE (LOCK):

3 STATE_UNINITIALIZED for state uninitialized

4 STATE_UNLOCKED for state unlocked

5 STATE_LOCKED for state locked

6 for TYPE (EVENT):

7 STATE_UNINITIALIZED for state uninitialized

8 STATE_CLEAR for state clear

9 STATE_SET for state set

10 **7.1.1.2 Testing for Uninitialized State**

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

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

18 **7.1.1.3 SYNCHRONIZE Statement**

19 **7.1.1.3.1 Proposed X3H5 Extended Syntax Rule**

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

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

25 X703 sync-operation is scalar-character-expression

26 X704 sync-object is scalar-latch-variable
 27 or scalar-lock-variable
 28 or scalar-event-variable
 29 or scalar-ordinal-variable

30 X705 ordinal-position is scalar-integer-expression

31 X706 sync-status is scalar-integer-variable

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

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

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

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

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

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

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

15 **7.1.1.3.2 Consistency Rules for the SYNCHRONIZE Statement**

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

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

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

24 **7.1.1.4 Representing Synchronization Operations**

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

30 TYPE (LOCK)

1 OP_CONDITIONAL_SET for operation conditional set
 2 OP_SET_WITH_WAIT for operation set with wait
 3 OP_CLEAR for operation clear

 4 TYPE (EVENT)
 5 OP_SET for operation set
 6 OP_CLEAR for operation clear
 7 OP_WAIT for operation wait

 8 TYPE (ORDINAL)
 9 OP_WAIT_THEN_POST_VALUE for operation post a value with wait
 10 OP_WAIT_VALUE for operation wait for a value

11 **7.1.1.5 Use of Control Types and Assignment**

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

14 **7.1.2 Limiting Synchronization Overhead**

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

17 **7.1.2.1 Proposed X3H5 Extended Syntax Rule**

18 R503 *attr-spec* is PARAMETER
 19 or *access-spec*
 20 or ALLOCATABLE
 21 or DIMENSION (*array-spec*)
 22 or EXTERNAL
 23 NEW or *guards-spec*
 24 or INTENT (*intent-spec*)
 25 or INTRINSIC
 26 or OPTIONAL
 27 or POINTER
 28 or SAVE
 29 or TARGET

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

 31 X708 *guarded-obj* is variable-name
 32 or array-element
 33 or array-section
 34 or substring

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

38 **7.1.2.2 GUARDS Attribute**

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

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

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

9 **7.1.3 Critical Sections**

10 **7.1.3.1 Proposed X3H5 Extended Syntax Rule**

11 X709 *critical-block* is *critical-stmt*
12 block
13 *end-critical-stmt*

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

16 X711 *end-critical-stmt* is END CRITICAL SECTION [(*scalar-latch-variable*
17)]

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

22 **7.1.3.2 Consistency Rules for CRITICAL SECTION**

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

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

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

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

34 **7.1.3.3 Operations on Objects of TYPE (LATCH)**

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

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

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

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

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

12 **7.1.3.4 Default Latch**

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

16 **7.1.4 Locks**

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

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

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

26
27 The clear operation is performed on an object of TYPE (LOCK) by executing a
28 SYNCHRONIZE statement specifying the object as *sync-object* and a *sync-operation* of
29 OP_CLEAR.

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

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

3 **7.1.5 Events**

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

6
7 The set operation is performed on an object of TYPE (EVENT) by executing a
8 SYNCHRONIZE statement specifying the object as sync-object and a sync-operation of OP_SET.

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

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

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

17 **7.1.6 Sequences**

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

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

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

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

1 **7.2 Explicit Synchronization**

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

4 Relationship between model synchronizer types and Fortran synchronizer types:

5 model synchronizer type	derived type name
6 lock	Type (lock)
7 latch	Type (latch)
8 event	Type (event)
9 sequence	Type (ordinal)

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

```
12   GUARDS (guarded-list) sync-object  
13   or  
14   GUARDS :: sync-guards-list  
15   where guarded is variable-name,  
16            array-name,  
17            array-element,  
18            array-section,  
19            module-name, or  
20            /common-block-name/ and  
21   sync-guards-list is sync-object (guarded-list) [, sync-guards-list]
```

22 **7.2.1 Critical Sections**

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

30 **7.2.1.1 Explicit Syntax**

31 Statement Forms

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

34 Structured As

```
35    [label:] CRITICAL SECTION ...  
36                                    statements  
37                                    END CRITICAL SECTION ...[label]
```

1
2 Where

3 *lock* is a variable name or array element of type lock

4 *object-name* is a data object

5 **7.2.1.2 Coding Rules**

6
7 The Critical Section construct is a block structured construct. The Critical Section construct
8 follows all of the rules of Fortran block structured constructs. <* so we mean EXIT and Cycle
9 WORK?*>

10
11 If the lock is coded on the END CRITICAL SECTION statement, it must match the
12 corresponding lock on the CRITICAL SECTION statement.

13
14 **7.2.1.3 Interpretation**

15
16 A program that executes a CRITICAL SECTION statement with a lock that has a value of
17 undefined is not standard conforming.

18
19 Entering a Critical Section construct, is equivalent to executing a GET_LOCK statement on the
20 specified lock with an identical GUARDS clause. Leaving the critical section, by executing the
21 END CRITICAL SECTION statement or executing a PDONE statement, is equivalent to
22 executing an UNLOCK statement on the lock controlling the section with an identical GUARDS
23 clause, and then resuming execution at the appropriate statement outside the block.

24
25 An unnamed Critical Section (one without a lock specified) is functionally equivalent to a Critical
26 Section that specifies a lock that is

- 27 a) shared among all teams
- 28
- 29 b) initialized at program start-up to "unlocked"
- 30
- 31 c) is only referenced by that Critical Section construct

32
33 These rules cause lexically distinct unnamed Critical Sections to function independently. Any
34 single unnamed Critical Section controls all processes, allowing at most one process within the
Critical Section at any point in time.

35 **7.2.1.4 Examples**

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

```

1      PARALLEL PDO I=1,10
2          NEW T
3          CRITICAL SECTION (A)
4              T = B(I) * B(I-1)
5              SUM = SUM + T
6          END CRITICAL SECTION (A)
7      END PARALLEL PDO
8      END

```

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

```

15      Example 13
16          SUBROUTINE EX13 (A,B,SUM)
17              REAL B(0:100)
18              Lock A
19              GUARDS A(SUM)
20              PARALLEL PDO I=1,10
21                  NEW T
22                  CRITICAL SECTION (A)
23                      T = B(I) * B(I-1)
24                      SUM = SUM + T
25                  END CRITICAL SECTION
26              END PARALLEL PDO
27              END

```

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

```

32      Example 14
33          SUBROUTINE EX14 (A,B,SUM)
34              REAL B(0:100)
35              Lock A
36              PARALLEL PDO I=1,10
37                  NEW T
38                  CRITICAL SECTION (A) GUARDS(SUM)
39                      T = B(I) * B(I-1)
40                      SUM = SUM + T
41                  END CRITICAL SECTION (A)
42              END PARALLEL PDO
43              END
44

```

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

```

49      Example 15
50          SUBROUTINE EX15 (A,B,MAXA,GMAXA,N)
51              REAL A(N), B(N), MAXA
52              Lock GMAXA
53              GUARDS GMAXA(MAXA)

```

```

1      PARALLEL SECTIONS
2      NEW AM
3      SECTION
4          AM = A(1)
5          DO 10 I=2,N
6              IF(AM.LT.A(I))AM=A(I)
7          CONTINUE
8          CRITICAL SECTION (GMAXA)
9              IF(MAXA.LT.AM) MAXA=AM
10         END CRITICAL SECTION
11     SECTION
12     CRITICAL SECTION (GMAXA)
13     AM=MAXA
14     END CRITICAL SECTION
15     DO 20 I=1,N
16         B(I)=B(I)/AM
17     CONTINUE
18     END PARALLEL SECTIONS
19     END
20

```

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 non-deterministic 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.

```

21
22
23
24
25
26
27
28
29     Example 16          SUBROUTINE EX16 (A,B,MAXA,GMAXA,N)
30     REAL A(N), B(N), MAXA
31     Lock GMAXA
32     GUARDS GMAXA(MAXA)
33     PARALLEL SECTIONS
34     NEW AM
35     SECTION
36         CRITICAL SECTION (GMAXA)
37         AM=MAXA
38     END CRITICAL SECTION (GMAXA)
39     DO 10 I=2,N
40         IF(AM.LT.A(I)) THEN
41             CRITICAL SECTION (GMAXA)
42                 IF(MAXA.LT.A(I)) MAXA=A(I)
43                 AM=MAXA
44             END CRITICAL SECTION
45         ENDIF
46     CONTINUE
47     SECTION
48     DO 20 I=1,N
49         CRITICAL SECTION (GMAXA)
50             B(I)=B(I)/MAXA
51         END CRITICAL SECTION
52     CONTINUE
53     END PARALLEL SECTIONS
54     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

1 wait for the computation of MAXA to be complete. The value of MAXA used at any point in
 2 the scaling process depends upon the number of processes available, the assignment of the
 3 sections to processes, and the relative execution speed of the processes. In particular, the scaling
 4 of an individual element of B may be done with the value of MAXA that was available upon
 5 invocation of this routine, or it may be done with the value of MAXA that will be returned to
 6 the calling program, or with some intermediate value. All elements of B need not be scaled with
 7 the same value. While non-deterministic, this program is standard conforming.

```

8   Example 17                SUBROUTINE EX17 (B,SUM)
9                               REAL B(0:100)
10                              SUM = 0.0
11                              PARALLEL PDO I=1,10
12                                NEW T
13                                CRITICAL SECTION GUARDS(SUM)
14                                  T = B(I) * B(I-1)
15                                  SUM = SUM + T
16                                END CRITICAL SECTION
17                              END PARALLEL PDO
18                              END
  
```

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

```

23   Example 18                SUBROUTINE EX18 (B,SUM,PROD)
24                               REAL B(100)
25                               PARALLEL SECTIONS
26                                NEW T
27                                SECTION
28                                  T = 0.0
29                                  DO 10 I=1,10
30                                    T = T + B(I)
31                                  CRITICAL SECTION GUARDS(SUM)
32                                    SUM = T
33                                  END CRITICAL SECTION
34                                SECTION
35                                  T = 1.0
36                                  DO 20 I=1,10
37                                    T = T * B(I)
38                                  CRITICAL SECTION GUARDS(PROD)
39                                    PROD = T
40                                  END CRITICAL SECTION
41                                END PARALLEL SECTIONS
42                                END
43
44
  
```

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

```

49   Example 19                SUBROUTINE EX19 (A,B,MAXA,N)
50                               C
51                               C >>> NOT STANDARD CONFORMING <<<
52                               C
53
  
```

```

1      REAL A(N), B(N), MAXA
2      PARALLEL SECTIONS
3      NEW AM
4      SECTION
5          AM = A(1)
6          DO 10 I=2,N
7              IF(AM.LT.A(I))AM=A(I)
8          CONTINUE
9      10    CRITICAL SECTION GUARDS(MAXA)
10         IF(MAXA.LT.AM) MAXA=AM
11         END CRITICAL SECTION
12     SECTION
13         CRITICAL SECTION GUARDS(MAXA)
14         AM=MAXA
15         END CRITICAL SECTION
16         DO 20 I=1,N
17             B(I)=B(I)/AM
18         CONTINUE
19     20    END PARALLEL SECTIONS
20     END

```

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

```

25
26     Example 20          SUBROUTINE EX20 (B,SUM)
27     REAL B(0:100)
28     Lock A
29     UNLOCK(A)
30     PARALLEL PDO I=1,10
31     NEW T
32     T = B(I) * B(I-1)
33     CRITICAL SECTION (A)
34     SUM = SUM + T
35     END CRITICAL SECTION (A)
36     END PARALLEL PDO
37     END
38

```

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

```

48
49     Example 21          SUBROUTINE EX21 (A,B,SUM)
50     REAL B(0:100)
51     Lock A
52     PARALLEL PDO I=1,10
53     CRITICAL SECTION (A) GUARDS(SUM)
54     SUM = SUM + B(I) * B(I-1)
55     END CRITICAL SECTION

```

1 END PARALLEL PDO
2 END
3

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

6 **7.2.2 Event Synchronization**

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

9 Event synchronization provides operations to indicate that an event has not occurred (**CLEAR**),
10 to indicate that an event has occurred (**POST**), and to ensure that an event has occurred (**WAIT**).
11

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

- 13 1. preserve execution order independence if used within a worksharing construct without the
- 14 ORDERED qualifier.
- 15 2. ensure that the synchronization pattern described does not require more than one process
- 16 for correct execution.
- 17

18 **7.2.2.1 Explicit Syntax**

19
20 Statement Forms

21 POST (*event*) [GUARDS(*object-name-list*)]
22
23 WAIT (*event*) [GUARDS(*object-name-list*)]
24
25 CLEAR (*event*) [GUARDS(*object-name-list*)]

26 Where

27 *event* is a variable or array element of type event

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

30 **7.2.2.2 Coding Rules**

31 **POST**, **WAIT** and **CLEAR** are executable statements.

32
33 **7.2.2.3 Interpretation**

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

1 When a CLEAR statement is executed,

- 2 a) the appropriate shared variables are made consistent
- 3 b) event is set to "cleared", no matter what its value was previously.

4
5 When a POST statement is executed,

- 6 a) the appropriate shared variables are made consistent
- 7
8 b) the value of event is set to "posted", no matter what its value was previously.

9
10 When a WAIT statement is executed,

- 11 a) the appropriate shared variables are made consistent
- 12
13 b) the value of event is tested to see if it is "posted" if it is not, the process retry's this step
- 14 at a later time,

15
16 The initial value of an event is undefined. It becomes defined only upon the execution of a
17 CLEAR or POST statement. A program that executes a WAIT statement on an event with an
18 undefined value is not standard conforming.

19 **7.2.2.4 Examples**

```
20 Example 22          SUBROUTINE EX22 (B,E)
21                      REAL B(100),C
22                      EVENT E(100)
23                      PARALLEL PDO I=1,97
24                        IF (I .LT. 4) THEN
25                          POST E(I)
26                        ELSE
27                          CLEAR E(I)
28                        ENDIF
29                      END PARALLEL PDO
30                      PARALLEL PDO (ORDERED) I=4,100
31                        NEW C
32                        C = SIN(B(I))
33                        WAIT E(I-3)
34                        B(I) = B(I) + B(I-3)*C
35                        POST E(I)
36                      END PARALLEL PDO
37                      END
38
```

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

42 **7.2.2.5 Intrinsic Functions for Events**

1 LOGICAL FUNCTION POSTED(event)

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

```
4 Example 23                SUBROUTINE EX23 (C,D)  
5                               C  
6                               C >>> NOT STANDARD CONFORMING <<<  
7                               C  
8                               C  
9                               REAL C,D  
10                              EVENT A, B  
11                              CLEAR A  
12                              CLEAR B  
13                              PARALLEL SECTIONS (ORDERED)  
14                              SECTION  
15                                WAIT A  
16                                C = C + 1  
17                                POST B  
18                              SECTION  
19                                POST A  
20                                WAIT B  
21                                D = C + 2  
22                              END PARALLEL SECTIONS  
23                              END  
24
```

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

28
29 Deadlock avoidance is the responsibility of the programmer. Here are some hints that can help
30 in avoiding deadlock. (A standard conforming program need not follow these hints.)

- 31
32 (1) Do not use event synchronization in unordered parallel loops or unordered parallel
33 sections.
34
35 (2) In Parallel Do and Pdo constructs with the ORDERED qualifier, make sure that POST
36 statement is executed for an iteration earlier in the serial order than the iteration
37 containing the corresponding WAIT statement.

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

41 **7.2.3 Sequences: Ordinal Synchronization**

42 Ordinal synchronization is used to communicate between iterations of a loop, or to communicate
43 between distinct loops. Any series of events that can be numbered can be synchronized with
44 ordinal synchronization.

45 Ordinal synchronization describes an arithmetic sequence. It provides operations to define an
46 arithmetic sequence (SET), indicate that computation for a particular element of the sequence is

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

3
4 Ordinal synchronization may be used anywhere in the program. If a Parallel Do or Pdo construct
5 is used to create the arithmetic sequence being synchronized, then the ORDERED qualifier is
6 required. Care shall be taken to

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

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

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

13 **7.2.3.1 Explicit Syntax**

14 Statement Forms

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

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

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

18 Where

19 *seq* is a variable or array element of type ordinal

20

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

22 *iexp4* is an integer expression not equal to zero

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

25 **7.2.3.2 Coding Rules**

26 POST, WAIT and SET are executable statements.

27 **7.2.3.3 Interpretation**

28

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

31

32 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)

```

```

1      WAIT (A,I-1)
2      SUM(I) = SUM(I-1) + T
3      POST (A,I)
4  END PARALLEL PDO
5  END
6
7

```

8 Example 25

```

9      SUBROUTINE EX25 (B,SUM)
10     REAL B(100),SUM(100)
11     EVENT AA(100)
12     GUARDS AA(SUM)
13     POST(AA(1))
14     PARALLEL PDO I=2,100
15     CLEAR AA(I)
16     END DO
17     SUM(1) = 0.0
18     PARALLEL PDO (ORDERED) I=2,10
19     NEW T
20     T = B(I) * B(I-1)
21     WAIT AA(I-1)
22     SUM(I) = SUM(I-1) + T
23     POST AA(I)
24     END PARALLEL PDO
25     END

```

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

29 Example 26

```

30     SUBROUTINE EX26 (B,C,N)
31     REAL B(N),C(N)
32     PARAMETER (MAXN=1000)
33     EVENT E(MAXN)
34     PARALLEL PDO 10 I=1,N
35     IF (I .lt. 4) THEN
36     POST E(I)
37     ELSE
38     CLEAR E(I)
39     ENDIF
40     10 CONTINUE
41     PARALLEL PDO (ORDERED) 20 I=4,N
42     C(I) = FUNC(B(I))
43     WAIT E(I-3) GUARDS(B(I-3))
44     B(I) = B(I) + B(I-3)*C(I)
45     POST E(I) GUARDS (B(I))
46     20 CONTINUE
47     END
48

```

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

50 Example 27

```

51     SUBROUTINE EX27 (B,C,N)
52     REAL B(N),C(N)
53     ORDINAL E
54     GUARDS E(B)
55     SET (E,3)
56     PARALLEL PDO (ORDERED) 20 I=4,N
57     C(I) = FUNC(B(I))
58     WAIT (E, I-3))
59     B(I) = B(I) + B(I-3)*C(I)

```

```

1          POST (E,I)
2
3      20    CONTINUE
4          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.

```

14 Example 28          SUBROUTINE EX28 (A,B,C,N1,N2,N3)
15                      REAL A(*),B(*),C(*)
16                      ORDINAL D
17                      GUARDS D(C)
18                      SET (D,N1,N3)
19                      PARALLEL SECTIONS
20                      SECTION
21                          DO 10 I=N1,N2,N3
22                              C(I) = MAX(A(I),A(I-N3))
23                              POST(D,I)
24      10    CONTINUE
25                      SECTION
26                          DO 20 I=N1,N2,N3
27                              WAIT(D,I)
28                              B(I) = B(I)/C(I)
29      20    CONTINUE
30                      END PARALLEL SECTIONS
31                      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.

```

38 Example 29          SUBROUTINE EX29 (B)
39                      REAL B(100)
40                      ORDINAL A
41                      SET (A,2)
42                      PARALLEL PDO (ORDERED) I=1,99
43                          NEW T
44                          T = B(I+1)
45                          POST (A,I+1)
46                          B(I) = T
47                      END PARALLEL PDO
48                      B(100) = 0.0
49                      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

1 away B(1). There is no need to wait, because when the POST statement is executed, the implied
2 wait insures that the previous iteration has already been posted.

3 **7.2.3.5 Intrinsic Functions for Ordinals**

4 INTEGER FUNCTION INT(seq)

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

8 Avoiding Deadlock

9
10
11 As with event synchronization, deadlock is a possibility with ordinals.

```
12  
13 Example 30                SUBROUTINE EX30 (B,C)  
14 C  
15 C >>> NOT STANDARD CONFORMING <<<  
16 C  
17     REAL B(100), C  
18     ORDINAL A  
19     SET (A, -99)  
20     PARALLEL PDO (ORDERED) 10 I = 1,99  
21         WAIT (A, -(I+1))  
22         B(I) = B(I+1) + C  
23         POST (A, -I)  
24     10 CONTINUE  
25     END
```

26
27 In Example 30, the program will deadlock with any number of processors less than 99, because
28 the iterations are handed in order from first to last. If there are only 98 processors, they will all
29 wait for the last iteration to execute its POST statement. This program unit is not standard
30 conforming because it requires at least 99 processes to avoid deadlock. To be standard
31 conforming, a program unit must be capable of completing execution with any number of
32 processes.

33 **7.2.4 Unstructured synchronization - Locks**

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

37
38 Unstructured control of LOCKs can be used anywhere within the program. Care should be taken
39 to preserve execution order independence if used within a worksharing construct without the
40 ORDERED qualifier. Care should be taken to ensure that the synchronization pattern described
41 does not require more than one process for correct execution.

42 **7.2.4.1 Explicit Syntax**

1 Statement Forms
2 GET_LOCK (*lock*) [GUARDS(*object-name-list*)]
3 UNLOCK (*lock*) [GUARDS(*object-name-list*)]

4 Where
5 *lock* is a variable or array element of type lock
6 *object-name* is a data object>

7 **7.2.4.2 Coding Rules**

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

10 **7.2.4.3 Interpretation**

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

- 14
- 15 U1) the appropriate shared objects are made consistent
- 16 U2) if the current value of the lock is "locked", the value is changed to "unlocked".
17 GET_LOCK has the following effect:
- 18
- 19 L1) appropriate shared objects are made consistent
- 20 L2) if the current value of the specified LOCK is "unlocked" then
- 21 L2a) the value is changed to "locked"
- 22 L2b) execution continues with the next statement
- 23 L3) if the value of the specified LOCK is "locked", the process retries step L2) at a later time.

24
25 Step L2) and L2a) above are executed as a single atomic operation.

26
27 The initial value of a LOCK is undefined. It becomes defined only at the execution of
28 UNLOCK.

29
30 A program that executes GET_LOCK on lock with an undefined value is not standard
31 conforming.

32
33 If a GUARDS clause is specified then for the duration of the synchronization statement, the
34 names listed shall be used to augment the set of objects guarded by that synchronizer if the

1 synchronizer was specified in a <sync-list> of the GUARDS statement. The merged set of
 2 guarded objects shall be made consistent when the synchronization statement is encountered. By
 3 explicitly identifying names of objects that shall be made consistent, the GUARDS clause and
 4 GUARDS statement remove a requirement for the implementation to make any other objects
 5 consistent when the synchronization statement is encountered.

6 **7.2.4.4 Examples**

```

7 Example 7      REAL FUNCTION SUM(A,B)
8                REAL B(0:100)
9                LOCK A
10 sumproduct:   PARALLEL PDO I=1,10
11                NEW T
12                GET_LOCK (A)
13                T = B(I) * B(I-1)
14                SUM = SUM + T
15                UNLOCK (A)
16                END PARALLEL PDO sumproduct
17                END

```

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

```

25 Example 8      SUBROUTINE EX8 (A,B,SUM)
26                REAL B(0:100)
27                LOCK A
28                GUARDS A(SUM)
29                PARALLEL PDO I=2,10
30                NEW T
31                GET_LOCK (A)
32                T = B(I) * B(I-1)
33                SUM = SUM + T
34                UNLOCK (A)
35                END PARALLEL PDO
36                END
37

```

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

42 **7.2.4.5 Intrinsic Functions for Locks**

43 LOGICAL FUNCTION TRY_LOCK(lock)
 44

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

```
4  
5 Example 9          SUBROUTINE EX9 (NAME,A)  
6     CHARACTER*(*) NAME  
7     CHARACTER*10 PG  
8     LOCK A  
9     IF ( TRY_LOCK (A) ) THEN  
10      PG = "LOCKED"  
11    ELSE  
12      PG = "UNLOCKED"  
13    ENDIF  
14    PRINT *, "Lock ",NAME," was ",PG  
15  END  
16
```

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

20 LOGICAL FUNCTION GET_LOCK(lock)

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

```
28  
29 Example 10        SUBROUTINE EX10 (A)  
30     Lock A  
31     5   IF (.NOT. GET_LOCK(A)) THEN  
32       CALL USEFUL  
33       GO TO 5  
34     ENDIF  
35     CALL UPDATE  
36     UNLOCK (A)  
37     END
```

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

```
40  
41 Example 11        SUBROUTINE EX11 (A)  
42     Lock A  
43     5   IF (TRY_LOCK (A)) THEN  
44       CALL USEFUL  
45       GO TO 5  
46     ELSE  
47       GET_LOCK(A)  
48       CALL UPDATE  
49       UNLOCK (A)  
50     ENDIF
```

1 END

2

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

1 **8.0 Nondeterministic Programs**

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

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

1 **A.0 X3H5 Directive Binding**

2 **A.1 Directives - Introduction**

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

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

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

15
16 Directive syntax and structure are specified in this appendix. Because of a basic one to one
17 association between the directives and corresponding language statements, the specification for
18 the directives will not replicate specifications given in this document for those associated
19 language statements. Interpretations and coding rules are provided only when they are in addition
20 to those provided for the corresponding language statement.

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

25 **A.1.1 Role of the Directive Binding**

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

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

34 **A.1.2 Single Process Execution Requirement for Compliant Programs**

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

5 Equivalent Serial Execution:

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

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

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

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

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

21 **A.1.3 Synchronization and Serial Execution**

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

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

30 **A.1.3.2 Scoping at Parallel Constructs and Serial Execution**

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

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

5 **Alternate Intrinsic Functions**

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

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

14 **A.1.4 Terminology**

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

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

21 **A.1.5 Directives - General Usage Requirements in Parallel Programs**

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

25 **A.1.5.1 Continued Directives**

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

32 **A.1.6 Parallel Intrinsic Functions**

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

3 **A.1.6.1 Parallel Intrinsic Behavior for Equivalent Serial Execution**

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

9 **A.1.6.2 Functionality Not Supported Under Serial Interpretation**

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

14 **A.2 Syntax Rules**

15 **A.2.1 Parallel Do Construct**

16 **A.2.1.1 Syntax**

17 Directive Forms for Component Directives:

```
18 C$PAR PARALLEL PDO [(option_list)]  
19 C$PAR END [PARALLEL] DO
```

20 Structured As:

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

25 **A.2.1.2 Coding Rules**

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

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

31

32 **A.2.1.3 Examples**

```

1      Example 1
2          SUBROUTINE EX1 (A,B,C,E,T,N)
3          REAL A(N),B(N),C(N+1),E(N),T
4
5          C$PAR    PARALLEL PDO
6          DO 10 I=1,N
7              E(I) = A(I)*B(I)
8              C(I+1) = E(I) * (T-1.0)
9          CONTINUE
10         END

```

```

10     Example 2
11         SUBROUTINE EX2 (A,B,C,E,T,N)
12         REAL A(N),B(N),C(N+1),E(N),T
13
14         C$PAR    PARALLEL PDO
15         DO I=1,N
16             E(I) = A(I)*B(I)
17             C(I+1) = E(I) * (T-1.0)
18         END DO
19     C$PAR    END PARALLEL PDO

```

20 **A.2.2 Parallel Sections Construct**

21 **A.2.2.1 Syntax**

22 Directive Forms for Component Directives:

```

23     C$PAR    PARALLEL SECTIONS [(qual_list)]
24
25     C$PAR    SECTION [/sec_nm/] [WAIT (sec_nm_list)] [GUARDS (obj_nm_list)]
26
27     C$PAR    END [PARALLEL] SECTIONS

```

26 Structured As:

```

27     C$PAR    PARALLEL SECTIONS [(option_list)]
28     [C$PAR    NEW obj_list]
29     C$PAR    SECTION ...
30             >> statements <<
31     [ ... zero or more additional section blocks ]
32     C$PAR    END PARALLEL SECTIONS

```

33 **A.2.2.2 Interpretation**

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

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

1 A.2.2.3 Examples

2 Example 3

```
3 SUBROUTINE EX3 (A,B,C,D,E,F,N)
4 REAL A(N),B(N),C(N),D(N),E(N),F(N)
5 C$PAR PARALLEL SECTIONS
6 C$PAR SECTION
7 DO 10 I=1,N
8 A(I) = B(I) * C(I)
9 10 CONTINUE
10 C$PAR SECTION
11 DO 20 J=1,M
12 D(I) = F(J) / E(I)
13 20 CONTINUE
14 C$PAR END PARALLEL SECTIONS
15 END
```

16 Example 4

```
17 SUBROUTINE EX4 (A,B,C,D,E,F,N)
18 REAL A(N),B(N),C(N),D(N),E(N),F(N)
19 C$PAR PARALLEL SECTIONS
20 C$PAR SECTION
21 C$PAR PARALLEL PDO
22 DO I=1,N
23 A(I) = B(I) * C(I)
24 END DO
25 C$PAR SECTION
26 C$PAR PARALLEL PDO
27 DO J=1,M
28 D(I) = F(J) / E(I)
29 END DO
30 C$PAR END PARALLEL SECTIONS
31 END
```

32 Example 5

```
33 SUBROUTINE EX5 (Z,ZA,ZB,ZC,ZD,ZE)
34 REAL Z(5)
35 C$PAR PARALLEL SECTIONS (ORDERED)
36 C$PAR SECTION /A/
37 ZA = ZFUNC(Z(1))
38 C$PAR SECTION /B/
39 ZB = 2*ZFUNC(Z(2))
40 C$PAR SECTION /C/ WAIT (A)
41 ZC = ZA * ZA + ZFUNC(Z(3))
42 C$PAR SECTION /D/ WAIT (A,B)
43 ZD = ZB - ZA + ZFUNC(Z(4))
44 C$PAR SECTION /E/ WAIT (C,B)
45 ZE = ZC - ZB + ZFUNC(Z(5))
46 C$PAR END PARALLEL SECTIONS
47 END
```

48 Example 6

```
49 SUBROUTINE EX6
50 REAL Z(10)
51 C$PAR SCOMMON /Z/
52 COMMON /Z/ ZB,ZD,ZE,ZTOT
53 C$PAR PARALLEL SECTIONS
54 C$PAR SECTION /A/
55 ZA = ZFUNC(Z(1))
```

```

1      C$PAR SECTION /BC/
2          ZB = ZFUNC(Z(2))
3          ZC = ZFUNC(Z(3))
4      C$PAR SECTION /D/ WAIT (A)
5          ZD = ZFUNC(ZA)
6      C$PAR SECTION /E/ WAIT (A,BC) GUARDS (ZA,ZC)
7          ZE = ZJOIN(ZA,ZC))
8      C$PAR END PARALLEL SECTIONS
9      ZTOT = ZJOINS(ZE,ZD,ZB)
10     END

```

Example 6A

```

12     SUBROUTINE EX6A
13     REAL Z(10)
14     C$PAR SCOMMON /Z/
15     COMMON /Z/ ZB,ZD,ZE,ZTOT

16     C$PAR PARALLEL SECTIONS
17     C$PAR SECTION /A/
18         ZA = ZFUNC(Z(1))
19     C$PAR SECTION /BC/
20         ZB = ZFUNC(Z(2))
21         ZC = ZFUNC(Z(3))
22     C$PAR SECTION /D/ WAIT (A)
23         ZD = ZFUNC(ZA)
24     C$PAR SECTION /E/
25     C$PAR WAIT (A,BC)
26     C$PAR GUARDS (ZA,ZC)
27         ZE = ZJOIN(ZA,ZC))
28     C$PAR END PARALLEL SECTIONS
29     ZTOT = ZJOINS(ZE,ZD,ZB)
30     END

```

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

33 A.2.3 Synchronization Declarations

34 A.2.3.1 Syntax

35 Directive Forms

```

36     C$PAR GATE declarator_list
37     C$PAR EVENT declarator_list
38     C$PAR ORDINAL declarator_list

39     C$PAR GUARDS guards_list

```

40 Directive Forms

```

41     C$PAR IMPLICIT sync_type

```

42 Structured As

```

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

```

1 where
2 sync_type is one of GATE, EVENT or ORDINAL

3 **A.2.3.2 Coding Rules**

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

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

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

13 **A.2.4 Unstructured Locking Synchronization**

14 **A.2.4.1 Syntax**

15 Directive Forms

16 C\$PAR GETLOCK (gate) [GUARDS (obj_nm_list)]

17 C\$PAR UNLOCK (gate) [GUARDS (obj_nm_list)]

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

23 **A.2.4.2 Examples**

```
24 Example 7
25 SUBROUTINE EX7 (A,B)
26 REAL B(0:100)
27 C$PAR GATE A
28 INTEGER AA
29 C$PAR PARALLEL PDO
30 C$PAR NEW T
31 C$PAR DO I=1,10
32     T = B(I) * B(I-1)
33 C$PAR LOCK (A)
34     SUM = SUM + T
35 C$PAR UNLOCK (A)
36 END DO
37 C$PAR END PARALLEL PDO
```

```

1          END
2      Example 8
3          SUBROUTINE EX8 (A,B,SUM)
4          REAL B(0:100)
5          C$PAR GATE A
6
7          C$PAR GUARDS A(SUM)
8          C$PAR UNLOCK (A)
9          SUM = 0.0
10
11         C$PAR PARALLEL PDO
12         C$PAR NEW T
13         DO I=1,10
14             T = B(I) * B(I-1)
15         C$PAR GETLOCK (A)
16         SUM = SUM + T
17         C$PAR UNLOCK (A)
18         END DO
19         C$PAR END PARALLEL PDO
20         END

```

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

```

20      Example 9
21          SUBROUTINE EX9 (NAME,A)
22          CHARACTER*(*) NAME
23          CHARACTER*10 PG
24          C$PAR GATE A
25
26          IF ( LOCKED (A) ) THEN
27              PG = "LOCKED"
28          ELSE
29              PG = "UNLOCKED"
30          ENDIF
31          PRINT *, "GATE ",name," is ",PG
32          END

```

```

32      Example 10
33          SUBROUTINE EX10 (A)
34          C$PAR GATE A
35
36          5 IF (.NOT. LOCK(A)) THEN
37              CALL USEFUL
38              GO TO 5
39          ENDIF
40          CALL UPDATE
41          C$PAR UNLOCK (A)
42          END

```

```

42      Example 11
43          SUBROUTINE EX11 (A)
44          C$PAR GATE A
45
46          5 IF (LOCKED (A)) THEN
47              CALL USEFUL
48              GO TO 5
49          ELSE
50              C$PAR GETLOCK(A)
51              CALL UPDATE
52              C$PAR UNLOCK (A)
53          ENDIF
54          END

```

1 **A.2.4.2.1 Function Values for GATEs in Serial Execution**

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

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

6	<u>function name</u>	<u>value returned</u>
7	LOCKED(gate_name)	.FALSE.
8	LOCK(gate_name)	.TRUE.

9 **A.2.5 Critical Sections**

10 **A.2.5.1 Syntax**

11 Directive Forms

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

14 Structured As

```
15 C$PAR CRITICAL SECTION ...  
16 >statements<  
17 C$PAR END CRITICAL SECTION ...
```

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

23 **A.2.5.1 Examples**

```
24 Example 12  
25 SUBROUTINE EX12 (A,B,SUM)  
26 REAL B(0:100)  
27 C$PAR GATE A  
  
28 C$PAR UNLOCK(A)  
29 C$PAR PARALLEL PDO  
30 C$PAR NEW T  
31 DO I=1,10  
32 T = B(I) * B(I-1)  
33 C$PAR CRITICAL SECTION (A)  
34 SUM = SUM + T  
35 C$PAR END CRITICAL SECTION (A)  
36 END DO  
37 C$PAR END PARALLEL PDO  
38 END
```

```

1      Example 13
2          SUBROUTINE EX13 (A,B,SUM)
3          REAL B(0:100)
4          C$PAR GATE A
5          C$PAR GUARDS A(SUM)
6          C$PAR UNLOCK(A)
7          SUM = 0.0
8          C$PAR PARALLEL PDO
9          C$PAR NEW T
10         DO I=1,10
11         C$PAR CRITICAL SECTION (A)
12             T = B(I) * B(I-1)
13             SUM = SUM + T
14         C$PAR END CRITICAL SECTION
15         END DO
16         C$PAR END PARALLEL PDO
17         END
18
19      Example 14
20          SUBROUTINE EX14 (A,B,SUM)
21          REAL B(0:100)
22          C$PAR GATE A
23          C$PAR UNLOCK(A)
24          SUM = 0.0
25          C$PAR PARALLEL PDO
26          C$PAR NEW T
27          DO I=1,10
28             T = B(I) * B(I-1)
29         C$PAR CRITICAL SECTION (A) GUARDS(SUM)
30             SUM = SUM + T
31         C$PAR END CRITICAL SECTION
32         END DO
33         C$PAR END PARALLEL PDO
34         END
35
36      Example 15
37          SUBROUTINE EX15 (A,B,MAXA,GMAXA,N)
38          REAL A(N), B(N), MAXA
39          C$PAR GATE GMAXA
40          C$PAR GUARDS GMAXA(MAXA)
41          C$PAR PARALLEL SECTIONS
42          C$PAR NEW AM
43          C$PAR SECTION
44             AM = A(1)
45             DO 10 I=2,N
46                 IF(AM.LT.A(I))AM=A(I)
47             CONTINUE
48         C$PAR CRITICAL SECTION (GMAXA)
49             IF(MAXA.LT.AM) MAXA=AM
50         C$PAR END CRITICAL SECTION (GMAXA)
51         SECTION
52         C$PAR CRITICAL SECTION (GMAXA)
53             AM=MAXA
54         END CRITICAL SECTION (GMAXA)
55         DO 20 I=1,N
56             B(I)=B(I)/AM
57         CONTINUE
58         C$PAR END PARALLEL SECTIONS

```

```

1          END
2
3      Example 16
4          SUBROUTINE EX16 (A,B,MAXA,GMAXA,N)
5          REAL A(N), B(N), MAXA
6          C$PAR GATE GMAXA
7          C$PAR GUARDS GMAXA(MAXA)
8
9          C$PAR PARALLEL SECTIONS
10         C$PAR NEW AM
11         C$PAR SECTION
12         C$PAR CRITICAL SECTION (GMAXA)
13             AM=MAXA
14         C$PAR END CRITICAL SECTION (GMAXA)
15         DO 10 I=2,N
16             IF(AM.LT.A(I)) THEN
17                 C$PAR CRITICAL SECTION (GMAXA)
18                     IF(MAXA.LT.A(I)) MAXA=A(I)
19                     AM=MAXA
20                 C$PAR END CRITICAL SECTION (GMAXA)
21             ENDIF
22         CONTINUE
23     SECTION
24     DO 20 I=1,N
25         C$PAR CRITICAL SECTION (GMAXA)
26             B(I)=B(I)/MAXA
27         C$PAR END CRITICAL SECTION (GMAXA)
28     CONTINUE
29     END PARALLEL SECTIONS
30     END
31
32     Example 17
33     SUBROUTINE EX17 (B,SUM)
34     REAL B(0:100)
35
36     SUM = 0.0
37     C$PAR PARALLEL PDO
38     C$PAR NEW T
39     DO I=1,10
40         T = B(I) * B(I-1)
41         C$PAR CRITICAL SECTION GUARDS(SUM)
42             SUM = SUM + T
43         C$PAR END CRITICAL SECTION
44     END DO
45     END
46
47     Example 18
48     SUBROUTINE EX18 (B,SUM,PROD)
49     REAL B(100)
50
51     C$PAR PARALLEL SECTIONS
52     C$PAR NEW T
53     C$PAR SECTION
54         T = 0.0
55         DO 10 I=1,10
56             T = T + B(I)
57         C$PAR CRITICAL SECTION GUARDS(SUM)
58             SUM = T
59         C$PAR END CRITICAL SECTION
60     SECTION
61         T = 1.0
62         DO 20 I=1,10
63             T = T * B(I)

```

```

1      C$PAR      CRITICAL SECTION  GUARDS(PROD)
2                PROD = T
3      C$PAR      END CRITICAL SECTION
4      C$PAR      END PARALLEL SECTIONS
5      END

6      Example 20
7                SUBROUTINE EX20 (B,SUM)
8                REAL B(0:100)
9      C$PAR      GATE A

10     C$PAR      UNLOCK(A)
11     C$PAR      PARALLEL PDO
12     C$PAR      NEW T
13     DO I=1,10
14         T = B(I) * B(I-1)
15     C$PAR      CRITICAL SECTION (A)
16         SUM = SUM + T
17     C$PAR      END CRITICAL SECTION (A)
18     END DO
19     END

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

24     C$PAR      UNLOCK(A)
25     C$PAR      PARALLEL PDO
26     DO I=1,10
27     C$PAR      CRITICAL SECTION (A) GUARDS(SUM)
28         SUM = SUM + B(I) * B(I-1)
29     C$PAR      END CRITICAL SECTION (A)
30     END DO
31     END

```

32 A.2.6 Event Synchronization

33 A.2.6.1 Syntax

34 Directive Forms

```

35     C$PAR      POST (event) [GUARDS (obj_nm_list)]
36     C$PAR      WAIT (event) [GUARDS (obj_nm_list)]
37     C$PAR      CLEAR (event) [GUARDS (obj_nm_list)]

```

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

```

43     Example 22
44     SUBROUTINE EX22 (B,E)
45     REAL B(100),C
46     C$PAR      EVENT E(100)

```



```

1      C$PAR      PARALLEL PDO
2                DO I=1,97
3                  IF (I .lt. 4) THEN
4                    C$PAR      POST (E(I))
5                  ELSE
6                    C$PAR      CLEAR (E(I))
7                  ENDIF
8                END DO

9      C$PAR      PARALLEL PDO (ORDERED)
10     C$PAR      NEW C
11     DO I=4,100
12       C = SIN(B(I))
13     C$PAR      WAIT (E(I-3))
14       B(I) = B(I) + B(I-3)*C
15     C$PAR      POST (E(I))
16   END DO

17   END

```

18 A.2.6.1.1 Function Values for Events in Serial Execution

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

21	<u>function name</u>	<u>value returned</u>
22	POSTED(event_name)	.TRUE.

23 A.2.7 Ordinal (Sequence) Synchronization

24 A.2.7.1 Syntax

25 Directive Forms

```

26     C$PAR      POST (seq, iexp1) [GUARDS (obj_nm_list)]
27
28     C$PAR      WAIT (seq, iexp2) [GUARDS (obj_nm_list)]
29
30     C$PAR      CLEAR (seq[, iexp3[, iexp4]]) [GUARDS (obj_nm_list)]

```

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

36 Example 24

```

37     SUBROUTINE EX24 (B,SUM)
38     REAL B(100),SUM(100)
39     C$PAR      ORDINAL A
40     C$PAR      GUARDS A(SUM)

41     SUM(1) = 0.0
42     C$PAR      SET (A)
43     C$PAR      PARALLEL PDO (ORDERED)

```

```

1      C$PAR   NEW T
2              DO I=2,10
3                  T = B(I) * B(I-1)
4      C$PAR   WAIT (A,I-1)
5              SUM(I) = SUM(I-1) + T
6      C$PAR   POST (A,I)
7              END DO
8      END

```

```

9      Example 25
10     SUBROUTINE EX25 (B,SUM)
11     REAL B(100),SUM(100)
12     C$PAR   EVENT AA(100)
13     C$PAR   GUARDS AA(SUM)
14
15     C$PAR   POST(AA(1))
16     C$PAR   PARALLEL PDO
17             DO I=2,100
18     C$PAR   CLEAR (AA(I))
19             END DO
20             SUM(1) = 0.0
21     C$PAR   PARALLEL PDO (ORDERED)
22     C$PAR   NEW T
23             DO I=2,10
24     C$PAR   T = B(I) * B(I-1)
25             WAIT (AA(I-1))
26     C$PAR   SUM(I) = SUM(I-1) + T
27             POST (AA(I))
28             END DO
29             END

```

```

29     Example 26
30     SUBROUTINE EX26 (B,C,N)
31     REAL B(N),C(N)
32     PARAMETER (MAXN=1000)
33     C$PAR   EVENT E(MAXN)
34
35     C$PAR   PARALLEL PDO
36             DO 10 I=1,N
37     C$PAR   IF (I .lt. 4) THEN
38             POST (E(I))
39     C$PAR   ELSE
40             CLEAR (E(I))
41             ENDDIF
42             10 CONTINUE
43     C$PAR   PARALLEL PDO (ORDERED)
44             DO 20 I=4,N
45     C$PAR   C(I) = FUNC(B(I))
46             WAIT (E(I-3)) GUARDS(B(I-3))
47     C$PAR   B(I) = B(I) + B(I-3)*C(I)
48             POST (E(I)) GUARDS (B(I))
49             20 CONTINUE
50             END

```

```

51     Example 27
52     SUBROUTINE EX27 (B,C,N)
53     REAL B(N),C(N)
54     C$PAR   ORDINAL E
55     C$PAR   GUARDS E(B)
56
57     C$PAR   SET (E,3)
58     C$PAR   PARALLEL PDO (ORDERED)
59             DO 20 I=4,N

```

```

1          C(I) = FUNC(B(I))
2      C$PAR    WAIT (E, I-3))
3          B(I) = B(I) + B(I-3)*C(I)
4      C$PAR    POST (E,I)
5          20    CONTINUE
6          END

7      Example 28
8          SUBROUTINE EX28 (A,B,C,N1,N2,N3)
9          REAL A(*),B(*),C(*)
10     C$PAR    ORDINAL D
11     C$PAR    GUARDS D(C)

12     C$PAR    SET (D,N1,N3)
13     C$PAR    PARALLEL SECTIONS
14     C$PAR    SECTION
15         DO 10 I=N1,N2,N3
16             C(I) = MAX(A(I),A(I-N3))
17     C$PAR    POST(D,I)
18         10    CONTINUE
19     C$PAR    SECTION
20         DO 20 I=N1,N2,N3
21             C$PAR    WAIT(D,I)
22                 B(I) = B(I)/C(I)
23         20    CONTINUE
24     C$PAR    END PARALLEL SECTIONS
25     END

26     Example 29
27     SUBROUTINE EX29 (B)
28     REAL B(100)
29     C$PAR    ORDINAL A

30     C$PAR    SET (A,2)
31     C$PAR    PARALLEL PDO (ORDERED)
32     C$PAR    NEW T
33         DO I=1,99
34             T = B(I+1)
35     C$PAR    POST (A,I+1)
36             B(I) = T
37         END DO
38         B(100) = 0.0
39     END

```

40 **A.2.7.1.1 Function Values for Counters in Serial Execution**

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

44 **A.3 Data Sharing**

45 **A.3.1 Data Sharing Directives**

46 **A.3.1.1 Syntax**

47 Directive Forms

1 C\$PAR NEW obj_nm_list

2 A.3.1.2 Rules

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

6 A.3.2 Partially Shared Common Blocks

7 A.3.2.1 Syntax

8 Directive Forms

9 C\$PAR SCOMMON sname_list

10 Structured As

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

13 A.3.2.2 Rules

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

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

19 Example 40

20 SUBROUTINE EX40 (B)
21 C\$PAR SCOMMON /BLOCKA/
22 COMMON /BLOCKA/ A(100)
23 REAL B(100)

24 C\$PAR PARALLEL PDO
25 DO I=1,100
26 A(I) = I * I
27 B(I) = A(I) + B(I)
28 END DO
29 END

30 Example 41

31 SUBROUTINE EX41 (B)
32 REAL B(100)

33 C\$PAR PARALLEL PDO
34 DO I=1,100
35 CALL SUB(B(I))
36 END DO
37 END

```

1      SUBROUTINE SUB(X)
2      C$PAR  SCOMMON /BLOCKA/
3              COMMON /BLOCKA/ A
4      A=X
5      CALL SQUARE
6      X=A
7      END

8      SUBROUTINE SQUARE
9      C$PAR  SCOMMON /BLOCKA/
10             COMMON /BLOCKA/ A
11      A=A*A
12      END

13     Example 41A
14             SUBROUTINE EX41A (B)
15     C$PAR  SCOMMON /BLOCKA/
16             COMMON /BLOCKA/ A
17             REAL B(100)

18             C$PAR  PARALLEL PDO
19     C$PAR  NEW /BLOCKA/
20             DO I=1,100
21                 CALL SUB(B(I))
22             END DO
23             END

24             SUBROUTINE SUB (X)
25     C$PAR  SCOMMON /BLOCKA/
26             COMMON /BLOCKA/ A
27             A = X
28             CALL SQUARE
29             END

30             SUBROUTINE SQUARE
31     C$PAR  SCOMMON /BLOCKA/
32             COMMON /BLOCKA/ A
33             A = A*A
34             END

35     Example 45
36             SUBROUTINE EX45 (B)
37             REAL B(100), C(100)

38             C$PAR  PARALLEL PDO
39             DO I=1,100
40                 CALL SUB1(B(I))
41     CALL SUB2(C(I))
42             END DO
43             PRINT *, (C(I), I = 1, 100)
44             END

45             SUBROUTINE SUB1 (X)
46     C$PAR  SCOMMON /BLOCKA/
47             COMMON /BLOCKA/ A
48             SAVE /BLOCKA/
49             A = X
50             END

51             SUBROUTINE SUB2 (X)
52     C$PAR  SCOMMON /BLOCKA/
53             COMMON /BLOCKA/ A
54             SAVE /BLOCKA/
55
56

```

```

1           X = A
2           END

3   Example 46
4           SUBROUTINE EX46 (B)
5           REAL B(100), C(100)
6           C$PAR SCOMMON /BLOCKA/
7           COMMON /BLOCKA/ A

8           C$PAR PARALLEL PDO
9           C$PAR NEW /BLOCKA/
10          DO I=1,100
11             CALL SUB1(B(I))
12             CALL SUB2(C(I))
13          END DO
14          PRINT *, (C(I), I = 1, 100)
15          END

16
17          SUBROUTINE SUB1 (X)
18          C$PAR SCOMMON /BLOCKA/
19          COMMON /BLOCKA/ A
20          A = X
21          END

22
23          SUBROUTINE SUB2 (X)
24          C$PAR SCOMMON /BLOCKA/
25          COMMON /BLOCKA/ A
26          X = A
27          END

```

```

28   Example 39
29           SUBROUTINE EX39(B,C,N)
30           REAL B(N),C(N)

31          C$PAR PARALLEL PDO
32          C$PAR NEW A
33          PARALLEL PDO I=1,N
34             A=B(I)+C(I)
35             CALL EX39A(A,B,I)
36          END DO
37          END

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

41          BX=AA*(AA-4.0)/BX
42          C$PAR PARALLEL PDO
43             DO J=1,N
44                 BB(J)=BB(J)*BX
45             END DO
46          END

```

47 **A.4 Parallel Region Construct**

48 **A.4.1 Syntax**

49 Directive Forms - Parallel Region parallel construct component directives

```

50   C$PAR   PARALLEL [(roption_list)]
51   C$PAR   END PARALLEL

```

1 **Structured As**

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

6 **Directive Forms - Pdo worksharing construct component directives**

```
7     C$PAR     PDO [(poption_list]
8     C$PAR     END PDO
```

9 **Structured As:**

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

13 **Directive Forms - Psections worksharing construct component directives**

```
14     C$PAR     PSECTIONS [(poption_list)]
15     C$PAR     SECTION [/sec_nm/] [wait (sec_nm_list)] [GUARDS(obj_nm_list)]
16     C$PAR     END PSECTIONS
```

17 **Structured As:**

```
18     C$PAR     PSECTIONS ...
19     C$PAR     SECTION
20         >> statements <<
21         [     ... zero or more section blocks ]
22     C$PAR     END PSECTIONS
```

23 **Directive Forms - Grouping construct component directives**

```
24     C$PAR     GROUP [(poption_list)]
25     C$PAR     END GROUP
```

26 **Structured As:**

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

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

```

1      Example 48
2          SUBROUTINE EX48 (A,B,C,N)
3          REAL A(N),B(N),C(N)
4
5          C$PAR PARALLEL PDO
6          C$PAR NEW T
7          DO I=1,N
8              T = A(I)*B(I)
9              C(I+1) = T * (T-1.0)
10         END DO
11         END
12
13      Example 49
14          SUBROUTINE EX49 (A,B,C,N)
15          REAL A(N),B(N),C(N)
16
17         C$PAR PARALLEL
18         C$PAR NEW T
19         C$PAR PDO
20             DO I=1,N
21                 T = A(I)*B(I)
22                 C(I+1) = T * (T-1.0)
23             END DO
24         C$PAR END PARALLEL
25         END
26
27      Example 50
28          SUBROUTINE EX50 (ZA,ZB,ZC,ZD,N)
29          REAL ZA(N),ZB(N),ZC(N),ZD(N)
30
31         C$PAR PARALLEL SECTIONS
32         C$PAR NEW T
33         C$PAR SECTION /DS5A/
34             DO 10 I=1,N
35                 T = ZFUNC(ZA(I))
36                 ZC(I) = T * T
37             10 CONTINUE
38         C$PAR SECTION /DS5B/
39             DO 20 I=1,N
40                 T = ZFUNC(ZB(I)-ZA(I))
41                 ZD(I) = T * T
42             20 CONTINUE
43         C$PAR END PARALLEL SECTIONS
44         END
45
46      Example 51
47          SUBROUTINE EX51 (ZA,ZB,ZC,ZD,N)
48          REAL ZA(N),ZB(N),ZC(N),ZD(N)
49
50         C$PAR PARALLEL
51         C$PAR NEW T
52         C$PAR PSECTIONS
53         C$PAR SECTION /DS5A/
54             DO 10 I=1,N
55                 T = ZFUNC(ZA(I))
56                 ZC(I) = T * T
57             10 CONTINUE
58         C$PAR SECTION /DS5B/
59             DO 20 I=1,N
60                 T = ZFUNC(ZB(I)-ZA(I))
61                 ZD(I) = T * T
62             20 CONTINUE
63         C$PAR END PSECTIONS

```



```

1          C$PAR  END PARALLEL
2          END
3
4  Example 52
5  SUBROUTINE EX52 (A)
6          REAL A(*)
7          GATE B
8          GUARDS B(SUM)
9
10         UNLOCK(B)
11         SUM=0.0
12         C$PAR  PARALLEL
13         C$PAR  NEW SUML
14             SUML = 0.0
15         C$PAR  PDO
16             DO I=1,N
17                 SUML = SUML + A(I)
18             END DO
19         C$PAR  CRITICAL SECTION (B)
20             SUM = SUM + SUML
21         C$PAR  END CRITICAL SECTION (B)
22         C$PAR  END PARALLEL
23         END

```

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

```

24
25  Example 52A
26  SUBROUTINE EX52A (A)
27          REAL A(*)
28          GATE B
29          GUARDS B(SUM)
30
31         UNLOCK(B)
32         SUM=0.0
33         C$PAR  PARALLEL
34         C$PAR  NEW SUML
35
36         C$PAR  GROUP
37             SUML = 0.0
38         C$PAR  PDO
39             DO I=1,N (NOWAIT)
40                 SUML = SUML + A(I)
41             END DO
42         C$PAR  CRITICAL SECTION (B)
43             SUM = SUM + SUML
44         C$PAR  END CRITICAL SECTION (B)
45         C$PAR  END GROUP
46
47         C$PAR  END PARALLEL
48         END

```

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

```

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

```

```

1           REAL A(N),B(N),C(N),D(N)
2
3   C$PAR  PARALLEL
4   C$PAR  PDO
5           DO I=1,N
6           A(I) = B(I) * C(I)
7           END DO
8   C$PAR  PDO
9           DO I=1,M
10          D(I) = A(I) - C(I)
11          END DO
12   C$PAR  END PARALLEL
13          END
14
15 Example 54
16 SUBROUTINE EX54 (A,C,N,M)
17 REAL A(N,0:M),C(N,M)
18
19 C$PAR  PARALLEL
20 DO 10 J=1,M
21 C$PAR  PDO
22 DO I=1,N
23 A(I,J) = C(I,J)/A(I,J-1)
24 END DO
25 10 CONTINUE
26 C$PAR  END PARALLEL
27          END

```

25 A.4.2 Single Process Sections

26 A.4.2.1 Syntax

27 Directive Forms

```

28 C$PAR  SINGLE PROCESS
29 C$PAR  END SINGLE PROCESS

```

30 Structured as

```

31 C$PAR  SINGLE PROCESS
32 >> Statements <<
33 C$PAR  END SINGLE PROCESS

```

```

34 Example 55
35 SUBROUTINE EX55 (A,B,N)
36 REAL A(N),B(N)
37
38 C$PAR  PARALLEL
39 C$PAR  PDO
40 DO I=1,N
41 A(I) = 1.0 / A(I)
42 END DO
43 C$PAR  SINGLE PROCESS
44 IF ( A(1) .GT. 1.0 ) A(1) = 1.0
45 C$PAR  END SINGLE PROCESS
46 C$PAR  PDO
47 DO I=1,N
48 B(I) = B(I) / A(1)
49 END DO

```

```

1          C$PAR  END PARALLEL
2          END
3
4  Example 56
5  SUBROUTINE EX56 (A,B,N)
6          REAL A(N),B(N)
7
8          C$PAR  PARALLEL
9          C$PAR  PDO
10         DO I=1,N
11            A(I) = 1.0 / A(I)
12        END DO
13        C$PAR  PSECTIONS
14        C$PAR  SECTION
15            IF ( A(1) .GT. 1.0 ) A(1) = 1.0
16        C$PAR  END PSECTIONS
17        C$PAR  PDO
18        DO I=1,N
19            B(I) = B(I) / A(1)
20        END DO
21    C$PAR  END PARALLEL
22    END
23
24 Example 57
25 SUBROUTINE EX57 (A,AMAX,N)
26     REAL A(0:N)
27
28     AMAX = 0.0
29     C$PAR  PARALLEL
30     C$PAR  NEW ALMAX
31
32     C$PAR  GROUP
33     C$PAR  PDO (NOWAIT))
34     DO I=1,N
35         IF ( ABS(A(I)) .GT. ABS(ALMAX) ) ALMAX = A(I)
36     END DO
37     C$PAR  CRITICAL SECTION
38         IF ( ABS(ALMAX) .GT. ABS(AMAX) ) AMAX = ALMAX
39     C$PAR  END CRITICAL SECTION
40     C$PAR  END GROUP
41
42     C$PAR  SINGLE PROCESS
43     ALMAX = A(1)+A(N)
44     IF ( AMAX .LT. ALMAX ) AMAX = 1.0 + AMAX
45     C$PAR  END SINGLE PROCESS
46
47     C$PAR  PDO
48     DO I=1,N
49         A(I) = ABS( A(I) / AMAX )
50     END DO
51
52     C$PAR  END PARALLEL
53     END

```

47 A.5 Exits from Parallel Constructs

48 A.5.1 Syntax

49 Directive Forms

```

1      C$PAR      PDONE
2
3      Example 3
4      SUBROUTINE EX3 (A,N,*)
5      REAL A(N)
6      LOGICAL FOUND
7
8      FOUND=.FALSE.
9      C$PAR PARALLEL PDO
10     DO I=1,N
11     IF ( A(I) .EQ. 0.0 ) THEN
12     C$PAR      PDONE
13     FOUND=.TRUE.
14     ENDDIF
15     END DO
16
17     IF ( .NOT. FOUND ) THEN
18     PRINT*, 'ALL ELEMENTS ARE NON-ZERO'
19     RETURN 0
20     ELSE
21     PRINT*, 'ERROR: THERE IS A ZERO ELEMENT IN A'
22     ENDDIF
23     END

```

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

24 A.6 Extended Intrinsic

25 A.6.1 Parallel Intrinsic Functions

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

29 A.6.2 Definition of Serial Execution Library

30	<u>Intrinsic</u>	<u>Value Returned</u>
31	INTEGER FUNCTION NPRCFG()	1
32	INTEGER FUNCTION MPRTOT()	1
33	INTEGER FUNCTION NPRAVL()	0
34	INTEGER FUNCTION NPRUSE()	1
35	INTEGER FUNCTION NPSCFG()	1
36	INTEGER FUNCTION MPSTOT()	1
37	INTEGER FUNCTION NPSAVL()	0
38	INTEGER FUNCTION NPSUSE()	1
39	INTEGER FUNCTION NPSTM()	1
40	SUBROUTINE SPRTOT(integer-expr)	none, routine has no effect
41	SUBROUTINE SPSTOT(integer-expr)	none, routine has no effect

```

1  A parallel-region-construct is:
2  [name:]    PARALLEL [(parallel-option)]
3             data-sharing-spec
4             parallel-body
5  END PARALLEL [name]
6
6  where
7  parallel-option is MAX PARALLEL = int-expr |
8                     ORDERED
9                     MAX PARALLEL = int-expr, ORDERED |
10                    ORDERED, MAX PARALLEL = int-expr
11  parallel-body is  statements |
12                    parallel-construct
13  parallel-construct is parallel-region-construct |
14                       pdo-construct
15                       psections-construct
16                       group-construct
17                       parallel-pdo-construct
18                       parallel-psections-construct |
19                       single-process-construct
20
20  Constraint: If the parallel-construct has a name prefix, then the it must have
21  the same name as a suffix.
22
22  data-sharing-spec is new-stmt |
23                     use-stmt |
24                     type-declaration-stmt |
25                     specification-stmt
26                     parameter-stmt
27                     format-stmt
28                     pointer-stmt
29                     [data-sharing-spec]
30
30  new-stmt is NEW variable-list
31
31  Constraint: specification-stmt shall not contain an access-stmt, common-stmt,
32  data-stmt, optional-stmt, equivalence-stmt, derived-type-stmt, or save-stmt.
33
33  [name:]    PDO [(parallel-options)]
34             parallel-body
35  END PDO [name]
36
36  [name:]    PSECTION
37             sections
38  END PSECTIONS [name]
39
39  where
40  sections is [sections section]
41  section is SECTION [name] [WAIT (name-list)]
42             parallel-region
43  [name:]    PARALLEL PDO iter-specification parallel-option-list
44             data-sharing-spec
45             parallel-body
46  END PARALLEL PDO [name]

```

```

1  [name:]      PARALLEL PSECTIONS [parallel-options]
2              data-sharing-spec
3              sections
4  END PARALLEL PSECTIONS [name]

5  [name:]      GROUP [(group-option)]
6              parallel-body
7  END GROUP [name]

8  where
9  group-option is NOWAIT
10 R503  attr-spec          is PARAMETER
11              or access-spec
12              or ALLOCATABLE
13              or DIMENSION ( array-spec )
14              or EXTERNAL
15              NEW          or guards-spec
16              or INTENT ( intent-spec )
17              or INTRINSIC
18              or OPTIONAL
19              or POINTER
20              or SAVE
21              or TARGET

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

23 X708  guarded-obj          is variable-name
24              or array-element
25              or array-section
26              or substring

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

30 X709  critical-block          is critical-stmt
31              block
32              end-critical-stmt

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

35 X711  end-critical-stmt      is END CRITICAL SECTION [ ( scalar-latch-variable
36 ) ]

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

41 GUARDS (guarded-list) sync-object
42 or
43 GUARDS :: sync-guards-list
44 where guarded is variable-name,
45         array-name,
46         array-element,
47         array-section,

```

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

1 C.0 Lex/Yacc Syntax Rules (Informative)

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

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

```
7  %{  
8  #include <stdio.h>
```

```
9  %}
```

```
10 %union {  
11     char string[33];  
12 }
```

```
13 %token PARALLEL MAX_PARALLEL WAIT GUARDS ORDERED NAME VARIABLE  
14 %token SECTION BLOCK PARALLEL_PSECTIONS PSECTIONS PARALLEL_PDO  
15 INTEGER  
16 %token PDO INT_EXPR TYPE_STMTS END_PARALLEL END_PDO END_PSECTIONS  
17 %token CODE_BLOCK DO_VARIABLE PARALLEL_PDO END_PARALLEL_PDO  
18 %token PARALLEL_SECTIONS END_PARALLEL_SECTIONS GROUP NOWAIT  
19 %token PARALLEL_SPECIFICATION_PART CONTINUE
```

```
20 %type <string> NAME
```

```
21 %type <string> name
```

```
22 %type <string> INTEGER
```

```
23 %%
```

```
24 pgm          : blocks
```

```
25              ;
```

```
26 blocks       : /* empty */
```

```
27              | blocks block
```

```
28              ;
```

```
29 block        : unnamed_p_block
```

```
30              | named_p_block
```

```
31              | code_block
```

```
32              ;
```

```
33 unnamed_p_block : parallel_block
```

```
34                 | parallel_pdo
```

```
35                 | parallel_sections
```

```
36                 | pdo_block
```



```

1      | psection_block
2      | group_construct
3      ;
4      /* ----- */
5      /* */
6      /* Constraint: An unnamed_p_block shall not contain an exit, return, */
7      /* stop, or entry-statement. */
8      /* */
9      /* ----- */

10     named_p_block : name ':' unnamed_p_block name
11     {
12     if(strcmp($1,$4))
13     {
14     printf("The starting and ending names of a block are different\n");
15     printf("They are %s, %s\n",$1,$4);
16     }
17     }
18     ;
19     /* ----- */
20     /* */
21     /* Constraint: The name coded at the beginning of a named_p_block shall be */
22     /* the same as the name coded at the end of the named_p_block. */
23     /* */
24     /* ----- */

25     parallel_block : PARALLEL ptoption blocks END_PARALLEL
26     ;

27     ptoption : /* empty */
28     | poption
29     | parallel_specification_part
30     | poption parallel_specification_part
31     ;

32     parallel_specification_part : PARALLEL_SPECIFICATION_PART ;
33

34     /* ----- */
35     /* */
36     /* See ISO/IEC 1539:1991 (E) page 304 for Fortran 90 specifications. */
37     /* */
38     /* parallel_specification_part : use_part decl_part */
39     /* ; */

```

```

1  /* use_part          : /* /* empty */
2  /*                  | use-stmt use_part          */
3  /*                  ;
4  /* decl_part        : /* /* empty */
5  /*                  | declaration_construct decl_part      */
6  /*                  ;
7  /*
8  /*
9  /* Constraint: specification-stmt must not contain an access-stmt, */
10 /*      allocatable-stmt(check with data section), common-stmt(check */
11 /*      with data section), data-stmt, intent-stmt, optional-stmt, */
12 /*      pointer-stmt (check with data section) or save-stmt.      */
13 /*      The decl_part shall not contain the entry_stmt, or      */
14 /*      stmt_function_stmt.
15 /* ----- */

```

```

16  poptions      : /* empty */
17              | poption
18              ;
19  poption       : '(' popt ')'
20              ;
21  popt          : MAX_PARALLEL '=' INT_EXPR
22              | ORDERED
23              | ORDERED MAX_PARALLEL '=' INT_EXPR
24              ;
25  psection_block : PSECTIONS poptions
26                sections
27                END_PSECTIONS
28                ;
29
30  sections      : section
31              | sections section
32              ;
33
34  section       : SECTION section_name wait_list guards_list
35                block
36                ;
37  section_name  : /* empty */
38                | '/' name '/'
39                ;
40  wait_list     : /* empty */
41                | WAIT '(' wlist ')'
42                ;
43  wlist         : /* empty */

```

```

1      | wlist name
2      ;
3  guards_list  : /* empty */
4      | GUARDS '(' glist ')'
5      ;
6  glist        : /* empty */
7      | glist name
8      ;
9
10 pdo_block    : PDO iter_spec poptions blocks END_PDO
11      | PDO INTEGER iter_spec poptions blocks
12      INTEGER CONTINUE
13      {
14      if(strcmp($2,$6))
15      {
16      printf("The starting and ending labels of a pdo block are different\n");
17      printf("They are %s, %s\n",$2,$6);
18      }
19      }
20      ;

21 iter_spec     : do_variable '=' INT_EXPR ',' INT_EXPR ',' INT_EXPR
22      | do_variable '=' INT_EXPR ',' INT_EXPR
23      ;
24 /* ----- */
25 /* */
26 /* Constraint: The pdo-variable must be a named scalar variable of type */
27 /* integer and cannot be an element of a common block. */
28 /* */
29 /* ----- */

30 group_construct : GROUP goption
31      blocks
32      END GROUP
33      ;
34 goption        : /* empty */
35      | NOWAIT
36      ;

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

38 parallel_pdo   : PARALLEL_PDO iter_spec ptoption blocks END_PARALLEL_PDO
39      ;
40 parallel_sections : PARALLEL_SECTIONS ptoption

```

```

1           sections
2           END_PARALLEL_SECTIONS
3           ;
4
5  /* -----here we provide stubs for various productions from the */
6  /*           native language (Fortran 90)           -----*/
7  code_block   : CODE_BLOCK ;
8  do_variable  : DO_VARIABLE ;
9  name         : NAME
10             { strcpy($$, $1); }
11 ;
12 %%
13 #include "lex.yy.c"
14 main()
15 {
16 if ( yyparse() )
17     { printf("error in line number: %d\n", line);
18       printf("Errors in this code\n");}
19 else
20     printf("YIPPEE no errors\n");
21 }

```

```

1  Dummy Lexical Analyzer for X3H5 Fortran

2  %{
3  int line;
4  %}
5  name  [a-zA-Z][a-zA-Z0-9_]*
6  integer [1-9][0-9]*
7  newline [\n]
8  blank  [\t]+
9  %p 10000
10 %o 10000
11 %a 19000
12 %%
13 {newline}      line ++      ;
14 {blank}        ;
15 PARALLEL      {return (PARALLEL);}
16 CONTINUE      {return (CONTINUE);}
17 END" PARALLEL  {return (END_PARALLEL);}
18 PARALLEL_PDO  {return (PARALLEL_PDO);}
19 END_PARALLEL_PDO {return (END_PARALLEL_PDO);}
20 PARALLEL_SECTIONS {return (PARALLEL_SECTIONS);}
21 END_PARALLEL_SECTIONS {return (END_PARALLEL_SECTIONS);}
22 PDO           {return (PDO);}
23 WAIT          {return (WAIT);}
24 GUARDS        {return (GUARDS);}
25 ORDERED       {return (ORDERED);}
26 MAX_PARALLEL  {return (MAX_PARALLEL);}
27 SECTION       {return (SECTION);}
28 PSECTIONS     {return (PSECTIONS);}
29 BLOCK         {return (BLOCK);}
30 END           {return (END); }
31 GROUP         {return (GROUP);}
32 NOWAIT        {return (NOWAIT);}
33 CODE{blank}BLOCK {return (CODE_BLOCK);}
34 END{blank}PDO   { /* printf("Found pdo\n"); */
35                return (END_PDO);}
36 END{blank}PSECTIONS {return (END_PSECTIONS);}
37 INT_EXPR       {return (INT_EXPR);}
38 VARIABLE      {return (VARIABLE);}
39 DO_VARIABLE    {return (DO_VARIABLE);}
40 {integer}      {strcpy(yylval.string,yytext);
41                printf("Found integer %s\n",yytext);
42                return (INTEGER);}
43 INTEGER        {return (INTEGER);}
44 PSPEC_PART     {return (PARALLEL_SPECIFICATION_PART);}

```

```
1 {name}          {strcpy(yylval.string,yytext);
2                return (NAME);}
3 .              { /* printf("Lex got %c\n",yytext[0]); */
4                return (yytext[0]);}
5 %%
```

Index

1
2 **A**
3 atomic operation 71
4 **C**
5 CLEAR 63
6 Critical Section 59
7 Critical sections 57
8 CLEAR 64
9 Critical Sections 57
10 **D**
11 deadlock 65, 70
12 **E**
13 Event synchronization 63
14 **G**
15 GATE synchronization 70
16 GUARDS 59
17 **I**
18 intrinsic 70
19 **M**
20 metalanguage conventions 11
21 **N**
22 nondeterministic 60, 75
23 **O**
24 ORDERED 63, 66
25 Ordinal 65
26 **P**
27 Parallel Sections 17
28 POST 64, 67
29 POST, 66
30 Ordinal 65
31 **S**
32 Sequence 69
33 Sequence synchronization 65
34 SET 66
35 standard conforming 10
36 Synchronization
37 Explicit synchronization 50
38 Implicit synchronization 50
39 Ordinal 65
40 structured 57
41 synchronization objects 57
42 unstructured 70
43 SET
44 Define ordinal 65
45 Unstructured

- 1 synchronization 70
- 2 **W**
- 3 **WAIT 64-67B.0 Syntax Rules (Informative)**