A Review and Analysis of Fortran 8x

Edited by

Brian T. Smith

Mathematics and Computer Science Division

October 1987

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Research, U.S. Department of Energy, under contract W-31-109-Eng-38.
# CONTENTS

ACKNOWLEDGMENTS ........................................................................................................................ vi

DISCLAIMER .......................................................................................................................................... vi

ABSTRACT ............................................................................................................................................. vii

1. AN OVERVIEW OF THE PROPOSED NEW FORTRAN 8x STANDARD ....................................... 1
  1.1. Introduction ............................................................................................................................. 1
  1.2. Compatibility Between Fortran 77 and Fortran 8x .............................................................. 1
  1.3. Major New Features ............................................................................................................. 2
    1.3.1. Arrays and Dynamic Allocation .............................................................................. 2
    1.3.2. Enhanced Floating-Point Features ........................................................................... 2
    1.3.3. User-defined Data Types .......................................................................................... 3
    1.3.4. Modules and Procedure Interfaces ............................................................................. 3
  1.4. Additional Features ................................................................................................................ 4
    1.4.1. Source Form ............................................................................................................. 4
    1.4.2. Intrinsic Subroutines ............................................................................................ 5
    1.4.3. Control Constructs .................................................................................................... 5
    1.4.4. Call Mechanisms ...................................................................................................... 6
    1.4.5. Input/Output .............................................................................................................. 6
    1.4.6. Internal Procedures ................................................................................................... .. 7
    1.4.7. Miscellaneous Features ............................................................................................ 7
  1.5. The Language Evolution Mechanism ................................................................................... 8
  1.6. Major Controversies in Fortran 8x ....................................................................................... 8
  1.7. Process of Defining a New Standard .................................................................................... .. 9

2. ARRAY PROCESSING ................................................................................................................... .10
  2.1. Introduction ........................................................................................................ ................. 10
  2.2. Major Features ..................................................................................................................... 10
    2.2.1. Whole Arrays and Sections .............................................................................. 10
    2.2.2. Range .......................................................................................................................... 12
  2.3. Dynamic Memory Allocation ............................................................................................. 12
  2.4. Conditional Array Assignment .......................................................................................... 13
  2.5. Array Mapping Capability ................................................................................................. 14
  2.6. Intrinsic Functions .............................................................................................................. 15
  2.7. Optimization Issues ............................................................................................................. 16
### 3. ENHANCED FLOATING-POINT FEATURES

3.1. Introduction

3.2. Generalized Precision Facility

3.3. Environmental Inquiry Intrinsic Functions

3.4. Floating-Point Manipulation Functions

3.5. Integration into Fortran 8x Features

3.6. Purpose of These Features

### 4. USER-DEFINED DATA TYPES, PROCEDURE INTERFACES, AND MODULES

4.1. Introduction

4.2. User-defined Data Types

4.3. Procedure Interfaces

4.4. Modules

### 5. OMISSIONS AND CONTROVERSIES IN FORTRAN 8X

5.1. Introduction

5.2. Bit Data Type and Bit Manipulation Facilities

5.3. Pointers

5.4. Significant Blanks

5.5. Other Features Moved to Appendix F by the Compromise Proposal

5.5.1. Variant Structures

5.5.2. Structure Arrays of Arrays Treated As Higher-Order Arrays

5.5.3. Vector-Valued Subscripts

5.5.4. FORALL Construct

5.5.5. Extensions to Internal Procedures

5.5.6. Condition Handling

5.6. Totally Omitted Features

5.6.1. Macros

5.6.2. Variable-Length Character Strings

5.6.3. Specialized Loop Constructs

5.6.4. Asynchronous Input/Output

5.6.5. Multitasking

5.7. Some Compilation Dependencies in Fortran 8x

5.7.1. Data Structures

5.7.2. Explicit Interfaces

5.8. Recurring Themes in the Second Ballot Responses

### 6. A BRIEF COMPARISON OF FORTRAN 8X AND Ada

6.1. Introduction

6.2. Intrinsic and Derived Data Types
ACKNOWLEDGMENTS

This document is a collection of articles prepared by members of the Language Working Group (LWG) of the Department of Energy. We thank our member institutions for their support over the past nine years, permitting us to follow and assess the development of the draft Fortran standard prepared by the ANSI Standards Subcommittee X3J3 on Fortran. We also thank our parent organization, the Scientific Computing Information Exchange (SCIE) Council, for their oversight and encouragement of this work. In addition, we thank Dr. Paul C. Messina, our past chairman, for his initiation, support, and encouragement of the project to prepare this document.

We also thank Dr. Gail Pieper of Argonne National Laboratory for her assistance in editing and preparing of this document.

The members of the LWG and others who participated in the preparation of this document are

Lawrence Berdahl, Lawrence Livermore National Laboratory
Bruce Curtis, Lawrence Livermore National Laboratory
Donna Fink, EG&G Idaho Inc.
Gary Blair, Lawrence Livermore National Laboratory
Jerry Ivey, EG&G Idaho Inc.
Neldon Marshall, EG&G Idaho Inc.
Jeanne T. Martin, Lawrence Livermore National Laboratory
Alex Marusak, Los Alamos National Laboratory
Lester Petrie, Oak Ridge National Laboratory
Guylaine Pollock, Sandia National Laboratories
Barbara Privet, EG&G Idaho Inc.
Brian T. Smith, Argonne National Laboratory (Chairman of LWG)
Sunnie Sund, Stanford Linear Accelerator Laboratory
Joe Wolf, Lawrence Livermore National Laboratory

DISCLAIMER

The opinions expressed in this document are those of the individual authors and members of the Language Working Group. They do not represent in any way a public comment or position on the draft proposed Fortran standard by the Language Working Group, the SCIE Council, the Department of Energy, or any of our supporting institutions.
ABSTRACT

The ANSI Standards Subcommittee X3J3 on Fortran has recently completed a draft proposed standard for the Fortran programming language. The draft proposed standard, known informally as Fortran 8x, is a revision of the current standard X3.9-1978, known informally as Fortran 77. This report is a review of Fortran 8x and consists of a series of six articles. The first article gives a general overview of Fortran 8x. The next three articles give brief discussions of the array facilities; the enhanced numeric facilities; and user-defined data types, procedure interfaces, and the new program unit called a module. The fifth article provides a brief analysis of the controversial issues discussed by X3J3 (including both the accepted and rejected facilities for Fortran 8x). The sixth article gives a brief comparison with Ada*. The report concludes with a summary giving information on how and where to express opinions of the draft proposed standard.

* Ada is a registered trademark of the US-Government-Ada Joint Program Office.
1. AN OVERVIEW OF THE PROPOSED NEW FORTRAN 8x STANDARD

Principal Authors: Donna Fink and Lester Petrie

1.1. Introduction

The ANSI Standards Subcommittee X3J3 has recently released the draft proposed Fortran standard [1] for public review. This article is the first in a series of six articles describing the major features of the proposed standard. In this article, we give an overview of the proposed standard; our objective is to inform Fortran users, who have not been following this work, of the proposed standard so that they can participate in the public review. Later articles explore some of the significant proposals in more depth, as well as discuss some of the possible features that the subcommittee considered but did not include.

This overview article comprises five main sections:
- a high-level comparison of Fortran 77 and Fortran 8x;
- major new features, including highlights of the first three articles;
- additional features;
- major controversies, as discussed in the fourth article; and
- the process of defining a new standard.

The topics of the subsequent articles are
- array features,
- user-defined types and modules,
- enhanced floating-point features,
- major controversies of Fortran 8x, and
- a comparison of Ada* and Fortran 8x.

This article and the five supporting articles were prepared by the DOE Language Working Group (DOE-LWG). This group contains representatives from many of the national laboratories and is interested in informing their user community of programming language issues. These representatives can provide access to a current copy of the Fortran 8x Draft Proposed Standard and will assist readers in providing comments to the standard. A further description of Fortran 8x appears in [3].

1.2. Compatibility Between Fortran 77 and Fortran 8x

Fortran 8x is upwards compatible with Fortran 77; that is, a standard-conforming Fortran 77 program will be a standard-conforming Fortran 8x program also. Such a program should generally compile and execute under a Fortran 8x compiler in the same manner as under a Fortran 77 compiler. A possible exception is if a function name is used that is the same as one of the new intrinsic functions and whose name was not defined in an EXTERNAL statement.

The ability to compile old programs with new compilers without change also does not necessarily extend to old programs that were modified by adding some of the new features to them. Some of the new

* Ada is a registered trademark of the US-Government-Ada Joint Program Office.
features, such as the new case construct, may be added to an old program with only very local effects. Other new features, such as modules or array operations, may require changes to subprograms other than the subprogram in which the feature is added, particularly if the interface to the subprogram is changed. In some instances, these new features require a form of dependent compilation; that is, the compiler will be able to correctly compile a program unit only with knowledge of interfaces to other routines or of public entities in specified modules. However, this requirement for dependent compilation still permits separate compilation.

A major difference between Fortran 77 and 8x is that Fortran 8x allows a new source form. Fortran 8x does not allow the mixing of old and new source forms within a program unit (subroutine). If a routine is modified using the new source form, the entire routine will have to be converted to the new source form. Type statements and DATA statements have been enhanced considerably, but the old forms are still valid.

1.3. Major New Features

Major new features added to the proposed standard include array operations and dynamic allocation of memory; generalized precision, numerical inquiry functions, and floating-point manipulation functions; user-defined data types; and modules and interface blocks. These features will be covered individually in subsequent articles, but a brief description of each is included here to give a more complete overview.

1.3.1. Arrays and Dynamic Allocation

One of the most significant features of the new standard is its array processing capabilities. Fortran 8x allows whole arrays, or portions thereof, to be treated as primary data objects. Operations, such as add and multiply, and intrinsic functions, such as sine and square root, manipulate array objects on an element-by-element (elemental) basis. A set of transformational operations, such as matrix multiplication and dot product, is provided to operate on a non-elemental basis. Facilities are also provided to define and redefine the index range of arrays during program execution, and to associate various elements of one array with those of another via subscript mapping. Conditional or masked assignment of array elements according to the evaluation of a logical array expression is another key mechanism for manipulating array objects. The resulting code is intended to be more concise, easier to read and maintain, and less error prone, while at the same time simplifying the compiler's job of optimization (notably vectorization).

Another important advance in the proposed standard is the dynamic allocation of data. Scientific computing often requires arrays that grow and shrink, or even appear and disappear. Since memory limitations can prevent all the data from being present at the same time, there must be a way to create and destroy blocks of storage. Fortran 8x provides a portable means to manage memory dynamically.

1.3.2. Enhanced Floating-Point Features

To provide a more portable specification of the floating-point types in Fortran 8x, a generalized precision facility has been added. Floating-point data objects can be specified with a decimal precision and decimal exponent range. The processor is required to provide at least the decimal precision and decimal exponent range specified; the result is processor dependent if the processor is unable to supply a data type with the requested properties. The specified precision and exponent range are optional; if neither is specified, a processor-dependent default real or complex data type is used. In general, precision data objects can appear anywhere the Fortran 77 real, complex, or double precision data objects can appear, except in storage association contexts (for example, in COMMON and EQUIVALENCE statements).
The generalized precision facility addresses current problems in the portability of floating-point software in Fortran. By changing the value of a parameter and recompiling all affected program units, the data type of all floating-point objects including constants can be changed from one precision to another, for example, from single precision to double precision. The generalized precision facility can be used to permit programs to run in double precision on short wordlength machines (e.g., IBM 360/370) and in single precision on long wordlength machines (e.g., CRAY machines) without any changes to the source. Complex double precision is also available, if double precision can be supplied using a precision specification.

Another major facility to assist in floating-point computation is a collection of environmental inquiry and floating-point manipulation intrinsic functions. These functions permit a program to obtain values for each real data type such as the largest and smallest floating-point values, the radix of floating-point values, and a number that is small relative to one. Additional intrinsic functions return the effective decimal precision and effective decimal exponent range provided by the processor of data objects specified by the generalized precision facility above. Finally, other intrinsic functions scale a real data object by a power of the radix, return the exponent of a real data object or its mantissa, and permit floating-point numbers to be formed from an exponent and mantissa without rounding errors.

Similar environmental inquiry functions have also been defined for the integer data type. In addition, the environmental and other intrinsic functions can be used in integer expressions to specify the type parameters for real, character, and parameterized user-defined types.

1.3.3. User-defined Data Types

Fortran has five intrinsic data types: integer, real, complex, logical, and character. Fortran types are characterized by

- a name,
- a set of values,
- a means to represent those values, and
- a set of operators.

Fortran 8x allows users to define new types that have the intrinsic types, or other user-defined types, as components. Means are provided to define and name a type, to declare objects to be of that type, to specify values for those objects, and to define operations that can be performed on them. A type definition is used to define and name a type. A structure constructor is used to specify its values. An operator may be defined by a function with an operator clause. Assignment of an object of a user-defined type may be defined by a subroutine with an assignment clause. A later article contains an example of a user-defined type.

1.3.4. Modules and Procedure Interfaces

A module is a new program unit for Fortran. It may contain data definitions, type definitions, procedure definitions, and procedure interface descriptions. Information in a module may be declared public or private. A USE statement permits access to the public entities in a module but not the private ones. By distinguishing between public and private entities, the module/use mechanism permits private information to be protected from misuse.

In Fortran 77, a common block provides global data definitions, but the common block must be defined in every program unit that accesses the global data. In Fortran 8x, the global data in modules are
defined only once and accessed wherever needed by a USE statement. A USE statement with an ONLY option will access only certain named entities in the module. A renaming option is provided in the case where a global data name would conflict with a local data name. These features provide more control over the information available to a routine, which contributes to safety as well as readability.

In addition to data, modules may contain procedures and procedure interface descriptions. A procedure interface is the name of the routine; its kind (subroutine or function); its arguments (names, types, and attributes); and whether the arguments have values on entry, values assigned within the routine, or both. A procedure definition contains this information as part of the definition.

A procedure interface description is a new syntactic mechanism in Fortran 8x that allows this information to be made available even when the body of the procedure is not available. The procedure itself may be in a proprietary library of routines, or it may be defined by means other than Fortran (assembler, C, etc.). When a processor has this information available, it can check that subroutine calls and function invocations are correct. This feature contributes to safety, readability, and optimization.

A module is also a good way to encapsulate a user-defined type, since it can contain the type definition and the procedures that define the operations for that type. In fact, it is the only standard-conforming way to make the type available in distinct external program units. Each of these features is useful by itself; put together, they form a language extension mechanism that is sometimes referred to as data abstraction.

1.4. Additional Features

Other features, not described in the above section Major New Features, are included in Fortran 8x. These features include improved source input facilities, intrinsic subroutines, enhanced and new control constructs, enhanced call mechanisms, input/output statement additions, and internal procedures. Other miscellaneous features include recursion, IMPLICIT NONE, and alternative forms for the relational operators. This section provides brief descriptions and examples.

1.4.1. Source Form

Thirty-one character, alphanumeric names with embedded underscores are allowed for variable, procedure, and file names (e.g., TIME_OF_DAY).

Fortran statements may be free-form. No longer must labels be in columns 1-5, continuation in column 6, and statements in columns 7-72. Rather than a C in column 1 for a comment, an explanation point (!) anywhere on a line (except within a character string) starts a comment, which continues to the end of the line, for example,

\[
X = \left(-B + \sqrt{D}\right) / \left(2A\right) \quad \text{COMPUTE ROOT}
\]

Outside of a comment, when the rightmost nonblank character on a line is an ampersand (&), the next line is a continuation line, for example,

\[
Y = X + \left(-B + \sqrt{D}\right) / \quad \&
\]

\[
(2A)
\]

Multiple statements may be placed on one line and are delimited by a semicolon (;), for example,
Free-form and fixed-form source text must not be mixed in the same program unit.

Programs, functions, subroutines, block data subprograms, and modules may have END statements, specifying the name of the program or program unit that is terminated (for example, END FUNCTION A, for a function with name A).

1.4.2. Intrinsic Subroutines

A small collection of intrinsic subroutines has been added to Fortran 8x. These include subroutines to obtain information about the system real-time clock, to determine the time and date, and generate random numbers.

1.4.3. Control Constructs

A CASE statement has been added. The form is

\[
\text{SELECT CASE (key)} \\
\text{CASE (option_1)} \\
\text{...} \\
\text{CASE (option_2)} \\
\text{...} \\
\text{CASE DEFAULT} \\
\text{END SELECT}
\]

where \( \text{key} \) is the expression to evaluate to determine which case to select, \( \text{option}_1, \text{option}_2, ... \) are the selectors to match against, and optional DEFAULT satisfies the case where a selector was not matched. The expression \( \text{key} \) must match one of the selectors, or there must be a DEFAULT selector. The expression \( \text{key} \) and the values \( \text{option}_1, \text{option}_2, ... \) must be of the same type and can be of type integer, character, or logical.

The option may be of the form \( \text{low:high} \), indicating a range. The expression \( \text{key} \) will match the selector if \( \text{low} \leq \text{key} \leq \text{high} \). The values \( \text{low} \) and \( \text{high} \) indicate the minimum and maximum values of the range; either \( \text{low} \) or \( \text{high} \) is optional, but not both.

New forms of the DO loop construct are allowed. A DO construct begins with a DO statement and ends with an END DO statement. The EXIT statement terminates the DO loop. The CYCLE statement starts the next iteration of the DO loop. (Example 1 illustrates these features.) In addition, a DO loop can be performed \( N \) times (DO (N TIMES) or appear as in Fortran 77 except without a statement label (DO K=1,5). An alphanumeric construct name may be used to name a DO loop for the purposes of identifying which END DO matches the DO statement and to provide a means to EXIT to a particular level of DO loop (as shown in Example 2).
Example 1

\begin{verbatim}
DO
  IF (condition_1) EXIT
  IF (condition_2) CYCLE
  END DO
\end{verbatim}

Example 2

\begin{verbatim}
DO M TIMES
  DO K = 1, 5
    L = K
    N = N1 + 1
    IF( FCN(L,N) ) EXIT LOOP1
  END DO LOOP2
END DO LOOP1
\end{verbatim}

1.4.4. Call Mechanisms

Actual argument lists for subprograms may have keyword arguments. Also, dummy arguments of subprograms may be specified as optional, and default values may be provided for missing actual arguments. Keyword arguments are placed after positional arguments and may appear in any order. The keywords are the names of the dummy arguments either in the definition of the subprogram or in the interface for the subprogram. The inquiry intrinsic function PRESENT(dummy argument) returns a value of true if the argument is provided.

\begin{verbatim}
SUBROUTINE SUBR_NAME( KEY1, KEY2, KEY3, KEY4 )
  OPTIONAL KEY3
  ...
  CALL SUBR_NAME(A, B, KEY3=C, KEY4=D)
  CALL SUBR_NAME(A, B, KEY4=D, KEY3=C)
  CALL SUBR_NAME(A, B, KEY4=D)
\end{verbatim}

1.4.5. Input/Output

Namelist input/output is provided (as was previously implemented by many compilers as an extension to Fortran 77). The namelist specifier (NML=) can be used in the READ, WRITE, or PRINT statements. For example,

\begin{verbatim}
NAMELIST /NLIST/ A,B,C
READ(5, NML=NLIST)
PRCMPT = has been added as an optional specifier to the READ statement. For example:
READ(5,*,PRCMPT=' SIZE:') SIZE
\end{verbatim}

will display "SIZE:" on the input unit.

The ASIS option is available (along with REWIND and APPEND) for the POSITION= input/output specifier on OPEN and INQUIRE statements. For OPEN, ASIS leaves the file position unchanged if the
file exists and already is connected, for example,

\[
\text{OPEN}(10, \text{FILE}='employee', \text{ACTION}='READ', \text{POSITION}='ASIS')
\]

When an input/output statement terminates, the variable specified in the VALUES= specifier will be the count of the number of values successfully read or written (including null values).

1.4.6. Internal Procedures

By using internal procedures the user can hide subroutines and functions from other program units. The internal procedures have access to the variables within their containing procedure. The CONTAINS statement indicates that the procedures that follow are internal.

```
SUBROUTINE A

... data definitions for A ...
... executable code for A ...

CONTAINS

SUBROUTINE A2

... data definitions for A2 ...
... executable code for A2 ...

END SUBROUTINE A2

FUNCTION A3(X)

... data definitions for A3 ...
... executable code for A3 ...

END FUNCTION A3

END SUBROUTINE A
```

1.4.7. Miscellaneous Features

New relational operator symbols may be used in place of the existing symbols as shown below:

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Alternative Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>.EQ.</td>
<td>==</td>
</tr>
<tr>
<td>.NE.</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>.LT.</td>
<td>&lt;</td>
</tr>
<tr>
<td>.LE.</td>
<td>&lt;=</td>
</tr>
<tr>
<td>.GT.</td>
<td>&gt;</td>
</tr>
<tr>
<td>.GE.</td>
<td>&gt;=</td>
</tr>
</tbody>
</table>
Recursive subroutines and functions can be specified, for example,

```
RECURSIVE SUBROUTINE QUICK_SORT
```

IMPLICIT NONE is a valid statement which requires the user to declare all data objects within that procedure.

1.5. The Language Evolution Mechanism

Fortran 8x includes a mechanism for informing users of the subcommittee's view of the direction of evolution of the Fortran language. Language features may be classified in one of three categories: deleted, obsolescent, and deprecated. Deleted features are features in the current standard that are considered to have been redundant and are largely unused. Such features are not included in successive standards. This category is empty in Fortran 8x. Obsolescent features are those redundant features for which better methods already exist in the current standard but which are still used frequently. The use of these features presumably will decrease with time, and if the use of these features has become insignificant by the time the next revision of the standard is completed, those features might be deleted. Deprecated features are those that have become redundant by the inclusion of certain new features of Fortran 8x. These features may at some future time become obsolescent.

1.6. Major Controversies in Fortran 8x

The proposed standard includes a large number of new features. There has been considerable debate among those who have followed the development of this proposal as to whether the new language is too large and whether it is too different from present practice. In an effort to achieve an internal compromise on the size of the language, the subcommittee removed a number of proposed new features. Because much work had already been done on these features and many members of the subcommittee think that they should be in the new standard, these features are included in Appendix F of the draft proposed standard [1].

The most controversial omissions from the proposed Fortran 8x are facilities for (a) bit data types and bit manipulation; (b) pointers; and (c) significant blanks. Bit manipulation facilities (shifting, masking, etc.) have been "standard" extensions in Fortran compilers since the development of Fortran. One form of bit data and bit manipulation is included in Appendix F. Although there is increasing interest in including pointer facilities, such facilities are in neither the proposed standard nor Appendix F. Fortran 77 uses blanks as insignificant "fillers"; but there is strong sentiment to make blanks significant in Fortran 8x, as is the case in other modern, high-level languages.

Other omissions include (d) condition handling; (e) variant structures; (f) vector-valued subscripts; (g) nested internal procedures; and (h) array element assignment (the FORALL construct).

Major concerns with the existing proposed standard include (i) the complexity and size of the language; (j) the incompatibility of old and new source forms (in the same procedure); (k) the demise of independent compilation; (l) the potential overhead of the array processing facilities; (m) the potential overhead of user-specified numeric precision; and (n) the possible elimination of existing features in a future standard (cf. Section 5 of this publication).
1.7. Process of Defining a New Standard

For those readers who would like to know the procedure for defining a new standard, the following is a brief description of the history of this proposal. When the standard for Fortran 77 was approved in 1978, the subcommittee X3J3 began work on the next revision. The size of the subcommittee has varied between about 30 and 40 members. No more than one member is allowed from any organization. The public comment for Fortran 77 and user surveys were used to determine what was needed. About four years were spent defining what the architecture of the language should be and what new features would be included. Another three to four years were spent defining the new features and producing a draft proposed standard. During this time there was a continual evolution of the list of new features.

After the draft standard is produced, it is released for public comment. The subcommittee must respond to all public comments in some form. This may range from a simple acknowledgment of receipt for a comment that says the draft is generally good, to a detailed explanation of how an inconsistency or contradiction in the draft pointed out in the comment was corrected. If there is a consensus in the public comments that some feature needs to be in or out of the language, the subcommittee will most likely make the requested change. If the response changes the proposed standard enough, another public review may be required.

At the international level, the International Standards Organization has delegated the responsibility for generating the draft Fortran standard to ANSI and X3J3. Once the draft proposed standard is completed by X3J3, it is sent to the appropriate international committee for consideration as a draft standard. In the case with Fortran 8x, the public review at the international level standard will take place at the same time. After the public reviews are completed, the draft standard goes to the parent committees at both the national and international levels for final approval as a new standard.

Public comment on the proposed draft standard [1] has always been an important part of the standardization process; but because of the many new features proposed, it will be even more influential with this standard. Public comments will be accepted by X3 from October 23, 1987 to February 23th, 1988 and should be sent to

Public Comment for Dpans
Fortran Revision
X3 Secretariat
Attn: Gwendy Phillips
Computer and Business Equipment
Manufacturers Association
Suite 300
311 First Street, N. W.
Washington, DC 20001-2178

A copy of any comments should also be sent to

Public Comment for Dpans
Fortran Revision
Board of Standards Review
American National Standards Institute
1430 Broadway
New York, NY 10018
2. ARRAY PROCESSING

Principal Author: Bruce Curtis

2.1. Introduction

The ANSI Standards Subcommittee X3J3 for Fortran has recently released the draft proposed standard for public review. This is the second in a series of articles describing the major features of the proposed standard. In this article we focus on the language constructs that relate to array processing, which many consider as one of the most significant advantages of Fortran 8x. The goal is to treat whole arrays and certain pieces of arrays as primary data objects. Primitive operations such as add and multiply, subroutine and function calls, intrinsic functions, input/output, and control constructs all can involve whole arrays and subarrays. The resulting code is intended to be more concise, easier to read and maintain, and less error prone, while at the same time simplifying the compiler's job of optimization (notably vectorization).

Another important advance in the proposed standard is the dynamic allocation of data. Scientific computing often requires arrays that grow and shrink, or even appear and disappear. Memory limitations can prevent all the data from being present at the same time, so there must be a way to create and release blocks of storage. Fortran 8x provides a portable means to manage memory dynamically.

Much of the array manipulation capability in the proposed standard is similar to and in some cases is derived from the DOE Language Working Group technical report [2].

2.2. Major Features

2.2.1. Whole Arrays and Sections

Operations can be applied to arrays, as long as the arrays conform. Conforming arrays have the same rank (number of dimensions), and the same extent (number of elements) in each dimension. The rank and extents of an array determine its shape, which is defined as the rank-one array consisting of the extents of the array. A portion of an array known as an array section can also be used as an operand. An array section is expressed by restricting the extent in one or more dimensions, as in A(1, 2:N:2) which is the vector [A(1,2), A(1,4), A(1,6), ..., A(1,N)]. Thus an array section is a rectangular subarray. The general form for an array section subscript is lb:ub:inc, where lb is the lower bound, ub is the upper bound, and inc is the increment between elements. This is called triplet notation. All but the first colon are optional; A(:,:) means the elements of A from the effective lower bound (see RANGE below) to the effective upper bound in increments of one in both dimensions, that is, the whole array A.

Here is a simple example:
REAL A(101,3), B(101,3), C(101,3)

\[
A = 0.5*\text{SQRT}(A) + B*C
\]

DO 20 I = 1, 101
  DO 10 J = 1, 3
    \(A(I,J) = 0.5*\text{SQRT}(A(I,J)) + B(I,J)*C(I,J)\)
  10 CONTINUE
20 CONTINUE

The multiplication \(B*C\) is not the usual matrix multiplication, but results in an array whose elements are the product of the corresponding elements of \(B\) and \(C\). Matrix multiplication is achieved by calling the intrinsic function \text{MATMUL}. \text{SQRT}(A) is an array whose elements are the square roots of the corresponding elements of \(A\). The numeric operations \(+, -, *, /, **, =\), logical and relational operations (.AND., .GE., etc.) and many of the intrinsic functions behave in this element-by-element fashion and are called \text{elemental} operations. Other intrinsic functions, such as \text{MATMUL} and \text{DOTPRODUCT}, are not elemental and are called \text{transformational}.

It is important to note that the preceding two programs do not have equivalent semantics in the following sense — the elements of \(A\) are not defined in the same order, although they produce the same values. That is, Fortran 77 requires the nested loops to be evaluated for each loop variable \(I\) and \(J\) in the order \(I = 1, 2, ..., 101\) and \(J = 1, 2, 3\) for each \(I\). However, the array formulation in Fortran 8x allows the elements of \(A\) to be defined in any order. The processor is permitted to select an order that is most suitable for the machine architecture. This freedom to optimize the code makes the semantics slightly different from the obvious code with loops but potentially results in more efficient code being generated.

A recurrence such as

\[
\text{DO 10 } I = 2, N \\
A(I) = A(I-1)*B(I) + C(I) \\
10 \text{ CONTINUE}
\]

cannot be expressed in array syntax. One might expect that the Fortran 8x statement

\[
A(2:N) = A(1:N-1)*B + C
\]

has the same effect, but Fortran 8x states that the right-hand side of an assignment statement is evaluated fully before any assignment takes place, so there is no recurrence. The loop must be written as a DO loop in order to get recurrence.

Array-valued constants and expressions can be written by enclosing in square brackets a sequence of scalar values separated by commas:

\[
\text{REAL } A(3) \\
A = [0.0, \pi, 2.0*\pi]
\]
This is interpreted as a rank-one array and is called an *array constructor*. The intrinsic function RESHAPE can be used to transform an array constructor into any allowable shape. For example,

```
REAL B(2, 2)
B = RESHAPE([2, 2], [1.0, 0.0, 0.0, 1.0])
```

assigns a 2x2 identity matrix to B. The first argument to RESHAPE is the "mold," a description of the desired shape (2x2). The second argument is an array of values to be transformed to the required shape. RESHAPE is not restricted to operate on array constructors; it may be given array names or expressions as arguments.

### 2.2.2. Range

It is often the case that the *declared* array bounds are inappropriate as bounds for whole array operations. For instance, the bounds to be used may change with each iteration of an outer loop. This occurs so frequently in scientific computing that Fortran 8x provides a means to assert the effective bounds for arrays in a straightforward manner. There is a declaration to group a set of arrays that will have the same effective range:

```
REAL VGFUN(0:150, 63), VGPARL(0.150, 63), DRES(0:150, 63)
RANGE /KMAXNK/ VGFUN, VGPARL, DRES
```

Here VGFUN, VGPARL, and DRES are put in a group called KMAXNK.

Then, an executable statement sets the effective range for all arrays in the group:

```
SET RANGE (0:KMAX-1, NK) /KMAXNK/
```

Now all unsubscripted references to VGFUN, VGPARL, and DRES have bounds 0 to KMAX-1 in the first dimension, and 1 to NK in the second dimension, according to the values of KMAX and NK at the time the SET RANGE statement is executed. The effect of the SET RANGE statement is global for arrays accessed from a module (modules are described in another article) and otherwise is local, and lasts until another SET RANGE statement with the same group name is executed. Note that an array must be defined in a module in order for a set range on it to have a global effect. Even if the array is in a common block, the common block must be in a module for the effect to be global.

### 2.3. Dynamic Memory Allocation

Memory management has always been a major concern in scientific software and has been a substantial problem in code portability since Fortran 77 does not provide for it. Fortran 8x fills this gap by allowing the dynamic allocation and deallocation of arrays. In order for an array to be created, it must be declared with the ALLOCATABLE attribute:

```
REAL, ALLOCATABLE, ARRAY(:,:) :: A
```

which says A is a rank-two array with real elements and is allocated space at run time.
Then, the ALLOCATE statement creates storage:

\[
\text{ALLOCATE ( \(A(101, 2\times N-1)\) )}
\]

In this example, A is created with 101 rows and 2N-1 columns, using the value of N at the time the ALLOCATE statement is executed.

The DEALLOCATE statement releases storage:

\[
\text{DEALLOCATE ( \(A\) )}
\]

Both the ALLOCATE and DEALLOCATE statements allow a list of arrays to be allocated and deallocated at once. Both statements also have an option that provides an indication of errors:

\[
\text{ALLOCATE ( \(X(10, 10), Y(10, 20), \text{STAT = IERR}\) )}
\]

IERR is an integer variable that the ALLOCATE statement sets to zero if no error occurred in the allocation, or to a positive integer if an error occurred.

It should be noted that an allocatable array that has been allocated must be deallocated before being reallocated. This means that in order to change the size of an allocated array and preserve its values, the user must copy the array into an array temporary, deallocate the original, reallocate it, copy it back from the temporary array, and deallocate the temporary array.

### 2.4. Conditional Array Assignment

A conditional computation facility is essential for array processing programs. Fortran 8x provides a simple mechanism for conditionally masking the evaluation of array assignment statements, namely, the WHERE construct.

<table>
<thead>
<tr>
<th>Fortran 8x Segment</th>
<th>Fortran 77 Segment Providing the Same Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL A(30,20), B(30,20)</td>
<td>REAL A(30,20), B(30,20)</td>
</tr>
<tr>
<td>WHERE (A .NE. 0.0) B = B / A ELSEWHERE B = 0.0 ENDWHERE</td>
<td>DO 99 I = 1, 30 DO 98 J = 1, 20 IF (A(I,J) .NE. 0.0) THEN B(I,J) = B(I,J) / A(I,J) ELSE B(I,J) = 0.0 ENDIF 98 CONTINUE 99 CONTINUE</td>
</tr>
</tbody>
</table>

In this example, the expression \(A .NE. 0.0\) serves as a mask controlling the execution of the two array assignment statements. For the nonzero elements of A, the statement \(B = B/A\) is executed; for the zero elements of A, \(B = 0.0\) is executed.
WHERE constructs cannot be nested. The only allowed statements inside a WHERE construct are array assignment statements that conform to the WHERE conditional. A single array assignment statement can be masked by a WHERE statement:

```fortran
REAL TEMP(N)
WHERE( TEMP .LT. 0.0 ) TEMP = 0.0
```

2.5. Array Mapping Capability

A rectangular subarray can be expressed with the array section described earlier. Other array subpieces can be expressed with the IDENTIFY statement. In the IDENTIFY statement, a second array is associated with the desired elements of the main array via a subscript mapping. As a simple example, suppose the subportion of interest is the diagonal of array A. One must declare another array B of rank 1, with the ALIAS attribute:

```fortran
REAL, ARRAY(N), ALIAS :: B
```

Then, after executing the statement:

```fortran
IDENTIFY (B(I) = A(I, I), I=1:N)
```

B is associated with the diagonal of A. Changing the diagonal elements of A will change the values of B and vice versa.

There are restrictions on the subscript mappings allowed. The IDENTIFY statement is of the form

```fortran
IDENTIFY (B(I1, I2, ..., In) = A(E1, E2, ..., Em), I1=L1:U1, ..., In = Ln:Un)
```

where

1) Each of the $I_k$ is an integer variable (essentially a place-holder with meaning only in the IDENTIFY statement; variables with the same name outside the IDENTIFY statement are not affected). The same integer variable must not be used for more than one $I_k$.
2) Each of the $E_k$ is an expression containing at most one of the $I_k$, and is linear in $I_k$. So $E_3$ could be $L_2-4$, or $3*1_1+1$, but could not be $1_1+I_2$, or $I_4**2$.
3) Each of the $I_k$ must appear in at least one $E_k$.
4) The mapping must not be many-to-one; that is, two or more elements of the left-hand-side array must not be mapped to the same datum.
5) The $L_k$ and $U_k$ are, respectively, lower and upper bounds for the $I_k$.

The IDENTIFY statement has local effect. The association between the alias array (B in the preceding example) and the parent array (A in the example) is dissolved when the subprogram returns or another IDENTIFY statement with the alias array is executed or the parent array is deallocated.

IDENTIFY statements allow the naming of some nonrectangular subarrays so that they can be used in array operations. Without IDENTIFY, these cases would not be expressible in array syntax; they would have to be written with subscripted syntax as in Fortran 77. There are other uses of IDENTIFY — a scalar
IDENTIFY (essentially a dynamic equivalence of scalars), and structure component aliasing — not described here.

It is worth noting that IDENTIFY has its drawbacks. The unrestrained use of the IDENTIFY statement can detract from the readability, maintainability, and efficiency of codes. The constraints on the IDENTIFY statement, including the restriction on the mapping described above, were applied by X3J3 to minimize the problems at the expense of some functionality. Hence, there is a compromise between the usefulness and potency of the IDENTIFY capability and its readability, safety, and efficiency. Users should determine whether this compromise is successful.

2.6. Intrinsic Functions

Many intrinsic functions are defined for operations on array objects. These functions are described as being elemental functions, which apply to scalar arguments or portions of array objects on an element-by-element basis; transformational functions which apply to whole array objects; and inquiry functions, which return information about array attributes. The familiar Fortran 77 numerical, mathematical, and character functions are all included in the Fortran 8x standard and operate on array objects in an elemental manner.

The transformational functions operate on whole arrays or allow a mask argument to be defined that selects the elements of the arrays upon which the functions operate. These functions are as follows:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY/ALL</td>
<td>true if any/all elements are true</td>
</tr>
<tr>
<td>COUNT</td>
<td>number of elements in the array</td>
</tr>
<tr>
<td>MAXVAL/MINVAL</td>
<td>maximum/minimum value in the array</td>
</tr>
<tr>
<td>PACK/UNPACK</td>
<td>gather/scatter elements from/to another array under mask</td>
</tr>
<tr>
<td>SPREAD</td>
<td>replicates arrays by adding a dimension</td>
</tr>
<tr>
<td>CSHIFT/EOSHIFT</td>
<td>circular/end-off shifts</td>
</tr>
<tr>
<td>MAXLOC/MINLOC</td>
<td>location of the maximum/minimum value in the array</td>
</tr>
<tr>
<td>SUM/PRODUCT</td>
<td>sum/product of array elements</td>
</tr>
<tr>
<td>MERGE</td>
<td>merge under mask</td>
</tr>
<tr>
<td>TRANSPOSE</td>
<td>transpose an array of rank two</td>
</tr>
<tr>
<td>RESHAPE</td>
<td>reshape an array</td>
</tr>
</tbody>
</table>

The array inquiry functions are as follows:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATED</td>
<td>array allocation status</td>
</tr>
<tr>
<td>DLBOUND/ELBOUND</td>
<td>declared/effective lower dimension bounds</td>
</tr>
<tr>
<td>DUBOUND/EUBOUND</td>
<td>declared/effective upper dimension bounds</td>
</tr>
<tr>
<td>DSHAPE/ESHAPE</td>
<td>declared/effective shape of array or scalar</td>
</tr>
<tr>
<td>DSIZE/ESIZE</td>
<td>declared/effective number of elements in an array</td>
</tr>
</tbody>
</table>
In addition, users can define their own functions to operate on array objects.

2.7. Optimization Issues

One motivation for array syntax is that it makes the compiler’s job of optimization easier. Code written in array syntax is vectorizable by the definition of the array syntax. However, modern vectorizing compilers are good enough to vectorize practically any DO loop that can be written in the Fortran 8x array syntax.

The IDENTIFY statement complicates the task of analyzing DO loops for vector dependencies. The run time mapping of arrays on top of arrays makes the compile time analysis of loops containing those arrays difficult. Of course, the intent is that DO loops will be replaced by array syntax, thereby making dependency analysis unnecessary, but in fact many loops cannot be replaced by the proposed array syntax. An important class of loops not expressible in array syntax is exemplified by the Hilbert matrix:

```fortran
REAL H(10,10)
DO 20 J = 1, 10
  DO 10 I = 1, 10
    H(I,J) = 1.0 / (I+J-1)
  10 CONTINUE
20 CONTINUE
```

DO loop indices are used as part of the calculation as well as subscripts for arrays. Appendix F of Fortran 8x contains a construct that would handle this particular class (the FORALL statement) and may or may not become part of the next standard. Even if the FORALL statement were included, DO loops would remain a workhorse in scientific codes.

Passing array sections or arrays aliased via the IDENTIFY statement as arguments to subroutines and functions has consequences on execution speed. Access to all array arguments is slowed, since the compiler must generate code that allows for the generalized array objects.

Array syntax indeed makes code more concise, easier to read and maintain, and less error prone. However, it should not be expected to speed up programs where a high-quality vectorizing compiler is available.
3. ENHANCED FLOATING-POINT FEATURES

Principal Author: Brian T. Smith

3.1. Introduction

The ANSI Standards Subcommittee X3J3 for Fortran has prepared a draft Fortran standard for public review. This article is the third in a series of six articles describing the major features of the proposed draft standard. In this article we give a general introduction of the new facilities for computation with the basic floating-point data types. When these features are used in programs incorporating either Fortran 77 features or the new Fortran 8x features such as array processing, the combination provides improved and powerful techniques for numerical computation.

3.2. Generalized Precision Facility

A generalized precision facility has been added to Fortran to provide a more portable specification of the floating-point types. Floating-point data objects (real or complex, or floating-point components of user-defined type objects) can be specified with a decimal precision and decimal exponent range.

Decimal precision and decimal exponent range provide a user-friendly specification and measure of the number of decimal digits and the valid range of numeric values used by the processor in numerical computation. The definitions of decimal precision and decimal exponent range for the processor's real data types are based on a signed magnitude model for the real values that can be represented, manipulated, and produced by the intrinsic numeric operations. These definitions are given precisely by the intrinsic inquiry functions EFFECTIVE_PRECISION and EFFECTIVE_EXPONENT_RANGE and are known as the effective type parameter values. For example, the effective precisions and exponent ranges for single and double precision on CRAY and IBM 360/370 routines are (14, 2465), (28, 2465) and (6, 75), (15, 75), respectively.

The processor is required to provide a floating-point data type that has at least the decimal precision and decimal exponent range specified. The specified precision or exponent range or both are optional; if neither is specified, a machine-dependent default real or complex data type is used. For compatibility with Fortran 77, the supplied data types in these cases are the usual single-precision real and complex data types.

Some examples of the various forms of the declaration are

\[
\begin{align*}
&\text{REAL } A, B, \ldots \\
&\text{REAL}(p, r) \quad A, B, \ldots \\
&\text{REAL} (\text{EXPONENT\_RANGE} = r) \quad A, B, \ldots \\
&\text{COMPLEX} \quad A, B, \ldots \\
&\text{COMPLEX}(p, r) \quad A, B, \ldots \\
&\text{COMPLEX} (\text{PRECISION} = p) \quad A, B, \ldots
\end{align*}
\]

where \( p \) is the specified decimal precision and \( r \) is the specified decimal exponent range. An integer constant, a somewhat restricted integer expression, or an asterisk can be supplied for \( p \) and \( r \).

Particular examples of declarations of generalized precision data types are

\[
\begin{align*}
&\text{RFAL}(10) \quad A
\end{align*}
\]
From these declarations, A has a decimal precision of at least 10 digits and a processor-dependent default exponent range. X is complex and of default precision and exponent range. B has a decimal precision larger than that of the processor-selected precision for A (usually twice that of A).

Argument association between real (or complex) data objects is valid only if the types are the same and the declared type parameters are the same except when the dummy argument is specified with type parameters that are asterisks. Only a dummy argument of a subprogram can be declared with asterisks (as DUMMY_ARGUMENT above), and it is specified to match the precision and exponent range of the corresponding actual argument.

The expressions permitted for p and r are essentially constant expressions in the sense that their values must be determined at compile time. That is, the operands in such expressions can be references to intrinsic functions, whose arguments are constants, or can be references to inquiry functions, whose results do not depend on the values of their arguments (as is the case for the inquiry function EFFECTIVE_PRECISION above).

A specification of asterisk for p or r specifies that the effective type parameter values of the dummy and actual argument will match. This rule implies that the referenced subprogram must accept floating-point actual arguments of more than one precision and/or exponent range. However, the use of asterisks is restricted so that the processor can, in effect, determine at compile time a small number of effective values for each asterisk. In particular, the declared type parameters of the floating-point actual arguments for a reference to such a subprogram are required to be identical.

The rules for argument matching, including the restricted use of asterisks, have been designed so that argument matching be standard conforming, portable, and efficient. The requirement that the declared type parameters (except when the dummy is an asterisk) be the same avoids a problem in the portability of code, namely, that an actual argument declared with REAL(6) matches the effective type parameters of a dummy argument declared REAL(10) on a CRAY computer but does not match on IBM 360/370 machines. The use of an asterisk provides a portable specification in this case.

However, the use of asterisk type parameters for floating-point types seems to require an implementation that selects the precision of its operations at run time. To avoid the need for such an implementation or likelihood that it be used except on machines that support it efficiently, the use of asterisks has been severely limited. The effect of the limitation is to permit the programming of generic subprograms whose actual arguments all have the same declared type parameters (such as linear equation solvers) with little or no run time cost. This use of the asterisk is, however, controversial.

Mixtures of generalized precision data objects are permitted in expressions. The mixing of operands with different type parameters requires conversion of the operands to the type determined by the maximum of the declared precisions and exponent ranges of the operands, and then the operation is performed. Constants can be specified explicitly to have a declared precision.

Generalized precision data objects can appear anywhere the Fortran 77 real, complex, or double precision data objects can appear, except in storage association contexts (for example, in COMMON and EQUIVALENCE statements). Storage association contexts are excluded because storage association has been deprecated. It is believed that Fortran 8x has better alternative mechanisms to accomplish the same effect.
3.3. Environmental Inquiry Intrinsic Functions

Another major facility to assist in floating-point computation is a collection of environmental inquiry and floating-point manipulation intrinsic functions. The results of inquiry functions do not depend on the values of their arguments but only on the type specification for the argument. Examples of these functions include a function to obtain certain machine-dependent values for each real data type such as the largest and smallest floating-point values, the radix of floating-point values, and a number that is small relative to one. See the table at the end of this section for a complete list of these functions.

The environmental inquiry functions permit the formulation of programs involving floating-point computation that are more portable than can be written in Fortran 77. For example, frequently in iterative computations, one needs to formulate a portable termination of the iteration. By using the intrinsic function EPSILON, the correction from one iterate to the next can be compared with

\[ \text{ABS}(\text{iterate}) \times \text{EPSILON}(\text{iterate}) \]

If the correction is smaller, the iteration can be terminated, whereas if the correction is not smaller, the iteration should be continued. With this kind of formulation, machine-dependent constants can be avoided.

In addition, other intrinsic functions return the effective decimal precision and effective decimal exponent range provided by the processor for data objects specified by the generalized precision facility above.

The complete list of environmental inquiry functions is given below:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIGITS</td>
<td>number of significant digits in the model</td>
</tr>
<tr>
<td>EFFECTIVE_EXPONENT_RANGE</td>
<td>effective decimal exponent range</td>
</tr>
<tr>
<td>EFFECTIVE_PRECISION</td>
<td>effective decimal precision</td>
</tr>
<tr>
<td>EPSILON</td>
<td>a number that is almost negligible compared to one</td>
</tr>
<tr>
<td>HUGE/TINY</td>
<td>largest/smallest number in the model</td>
</tr>
<tr>
<td>MAXEXPONENT/MINEXPONENT</td>
<td>maximum/minimum exponent in the model</td>
</tr>
<tr>
<td>RADIX</td>
<td>base of the model</td>
</tr>
</tbody>
</table>

3.4. Floating-Point Manipulation Functions

Additional intrinsic functions return the exponent of a real data object or its mantissa, and permit floating-point numbers to be formed from an exponent and mantissa without rounding errors. A function to scale a real data object by a power of the radix is also provided.

The floating-point manipulation functions also permit portable formulations of algorithms that require exact reduction or scaling. Typically, such formulations are needed to avoid overflow, underflow, and rounding errors in critical parts of numerical computations.

The complete list of floating-point manipulation functions is given below:
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPONENT</td>
<td>exponent part of a model number</td>
</tr>
<tr>
<td>FRACTION</td>
<td>fractional part of a number</td>
</tr>
<tr>
<td>NEAREST</td>
<td>nearest different processor number in a given direction</td>
</tr>
<tr>
<td>RRSPACING</td>
<td>reciprocal of the relative spacing of model numbers near a given number</td>
</tr>
<tr>
<td>SCALE</td>
<td>multiplication of a real number by the base of the model to a given integer power</td>
</tr>
<tr>
<td>SETEXPONENT</td>
<td>set exponent part of a number</td>
</tr>
<tr>
<td>SPACING</td>
<td>absolute spacing of model numbers near a given number</td>
</tr>
</tbody>
</table>

3.5. Integration into Fortran 8x Features

Other than in storage association context, both the specified precision data objects and additional intrinsic functions have been integrated into the new and old features of Fortran. For example, arrays of specified precision objects can be defined, returned as function results, allocated, specified in modules, or used in user-defined types.

The environmental inquiry functions RADIX, DIGITS, and HUGE have also been defined for the integer data type. In addition, the environmental and other intrinsic functions can be used in integer expressions to specify the type parameters for real, character, and parameterized user-defined types.

3.6. Purpose of These Features

The generalized precision facility addresses current problems in the portability of floating-point software in Fortran. For example, the generalized precision facility can be used to permit programs to run in double precision on short wordlength machines (e.g., IBM 360/370) and in single precision on long wordlength machines (e.g., CRAY machines) without any changes to the source. As another example, complex double precision is available, using a form of specification that is uniform with the specification of the real data type. The data type specifications for machines with more than two real data types also use this same form of specification for both real and complex data objects.

As a third example, users can organize their programs so that precisions can be readily changed (as is available with the "autodouble" capability on some machines). This can be accomplished by specifying a global integer parameter, say PREC, that is used to specify the precision of all floating-point data objects. By changing the value of this parameter PREC and recompiling all affected program units, the data type of all floating-point objects including constants can be changed from one precision to another precision.
4. USER-DEFINED DATA TYPES, PROCEDURE INTERFACES, AND MODULES

Principal Author: Jeanne T. Martin

4.1. Introduction

The ANSI Standards Subcommittee X3J3 for Fortran has recently released the draft proposed standard for public review. This article is the fourth in a series of articles describing the major features of the proposed standard. In this article we focus on user-defined data types, procedure interfaces, and modules. These three facilities provide a powerful language extension mechanism that permits the development of program libraries and packages of procedures in a wide variety of application areas.

4.2. User-defined Data Types

The Fortran language provides five intrinsic data types: integer, real, complex, logical, and character. Operations are defined for these types. Some operations, such as concatenation (/), are defined for a single data type (in this case the character data type), whereas others, such as addition (+), are defined for several of the types (integer, real, complex). When an operator is defined for more than one data type, it is said to be overloaded. Each data type has a set of values associated with it. A means is provided to represent constant values for each of the intrinsic types.

Thus, in Fortran, a data type has four characteristics:

- a name,
- a set of values,
- a means to represent the values, and
- a set of operations.

Since there are only five intrinsic data types, Fortran 8x allows users to extend the language by defining new data types. These types can be tailored to a particular application. There are so many possible applications that it is difficult to list examples that are universally meaningful, but for rational arithmetic it is easy to see the need for overloading existing arithmetic operators; for vector or matrix algebra there would also be a need for new operators to specify dot or cross products.

The Fortran 8x type definition facility can be used to define “pure” types, or it can be used as a structuring mechanism to keep associated data together. For example, physical units (such as grams or centimeters), represented as character data, might be associated with values of type real. Once such a structure is defined, it is convenient to specify operations that can be applied to it. Since the only operations permitted on objects of user-defined type are those that are specifically defined by the user, these types are of necessity “strongly typed.” This has the advantage of preventing such errors as adding “inches” to “hours.”

The Fortran 8x facilities for type definition allow users to create types that have the same four characteristics that the intrinsic types have. A user-defined type is constructed from intrinsic types or other user-defined types. A type definition can supply the name of the user-defined type and specify its components. The values of the new data type consist of the set of all combinations of the values of the individual components. Values of the type may be constructed by means of a derived-type constructor. When a new type is defined, a derived-type constructor is implicitly defined as well. It allows a value to be con-
constructed from a sequence of values, one for each component of the user-defined type since each component either is specified to be of intrinsic type or ultimately resolves into components that are specified to be of intrinsic type.

The mechanism for defining a new operator or assignment is the operator procedure, or assignment procedure. This is either a function that has been extended to permit the specification of an infix operator or a subroutine with an assignment clause. The operator may be an intrinsic operator that is being overloaded to have meaning for objects of user-defined types, or it may be a totally new operator. New operators have the form .abc. where abc is any sequence of letters that does not specify an intrinsic operator.

Some of the intrinsic types have parameters such as the length parameter for the character data type or the precision and exponent range parameters for real and complex types. User-defined types may have parameters as well. These parameters must be of type integer. The parameter names implicitly define inquiry functions, so that the value of a parameter for a given object is accessible.

These facilities can perhaps best be illustrated by a rather simple, universally understood example: Roman numerals. To extend the language to accommodate Roman numbers, a user would define a type and some operations such as +, -, *, /, .EQ., .NE., or .LE. and perhaps some conversion functions between integer and roman. For the type definition, a representation would be chosen from among the intrinsic types. The type might be parameterized:

```
TYPE ROMAN (RCM_SZ)
  INTEGER :: VALUE
  INTEGER :: LEN_ROM
  CHARACTER (RCM_SZ) :: AS_SEEN
END TYPE ROMAN
```

It is possible to set the individual components of a structured object by using their qualified names. For example, if FOUR is of type roman, one could write:

```
FOUR%VALUE = 4
FOUR%AS_SEEN = 'IV'
FOUR%LEN_ROM = 2
```

Or a structure constructor could be used to accomplish the same setting:

```
FOUR = ROMAN(30) (4, 2, 'IV')
```

In the latter case, the components must be matched by type with the components in the type declaration in the same order.

A global parameter would probably be defined for the length of roman numbers:

```
PARAMETER (MAX_LEN = 30)
```

Also needed would be an assignment coercion procedure to convert an integer object to an object of type
This subroutine would be invoked whenever the pattern

\[
\text{object of type(roman)} = \text{integer expression}
\]

was encountered in the program. So if \( \text{YEAR} \) was declared to be of type roman,

\[
\text{TYPE(\text{roman}(\text{MAX}_\text{LEN}))} \quad \text{YEAR}
\]

and the user wrote

\[
\text{YEAR} = 1987
\]

\( \text{YEAR} \) would have the value
A coercion to type roman for objects of type character would also be useful but messier to define since the character object would have to be validated as a legitimate roman number.

To add two objects of type roman, the user would want to overload the + operator, as follows:

```fortran
TYPE(ROMAN(MAX_LEN)) FUNCTION PLUS_ROMAN (R1, R2) OPERATOR (+)
    TYPE(ROMAN(MAX_LEN)), INTENT(IN) :: R1, R2
    PLUS_ROMAN = R1%VALUE + R2%VALUE
END FUNCTION PLUS_ROMAN
```

The single statement in this function would invoke the previously defined subroutine, since the pattern is

object of type(roman) = integer expression

If DATE_OF_STND were declared to be of type roman, one could write

```fortran
DATE_OF_STND = YEAR + ROMAN(MAX_LEN)(2,2,'II')
```

The other arithmetic operators could be overloaded similarly. The user might also want to overload the comparison operators:

```fortran
LOGICAL FUNCTION EQ_ROMAN (R1, R2) OPERATOR (.EQ.)
    TYPE(ROMAN(MAX_LEN)), INTENT(IN) :: R1, R2
    IF (R1%VALUE .EQ. R2%VALUE) THEN
        EQ_ROMAN = .TRUE.
    ELSE
        EQ_ROMAN = .FALSE.
    END IF
END FUNCTION EQ_ROMAN
```

It is always possible to find the value of ROM_SZ for an object of type roman by using the implicitly defined inquiry function by that name. For example, ROM_SZ(YEAR) would return 30 since MAX_LEN is 30. It might be useful to find the actual length of a character string representation of a roman number. For this the LEN function could be overloaded, as follows:

```fortran
INTEGER FUNCTION LEN (R1)
    TYPE(ROMAN(MAX_LEN)), INTENT(IN) :: R1
    LEN = R1%LEN_ROM
END FUNCTION LEN
```

Then if YEAR retained its previous value (Arabic 1978), LEN(YEAR) would return 10, the actual length of the roman numeral character string.

Suppose a unary operator ROM. was desired that would convert a real variable into a roman number. The user could define that operator with an OPERATOR function as follows:

```fortran
TYPE(ROMAN(MAX_LEN)) FUNCTION REAL_TO_R (A) OPERATOR (.ROM.)
    REAL, INTENT(IN) :: A
    REAL_TO_R = INT(ABS(A)) ! assignment defined by I_TO_R
END FUNCTION REAL_TO_R
```
Then one could write .ROM. 2.0005E3 and get MM (or 2000 in Arabic numerals). This is not very useful, but it illustrates the way in which new operators are defined.

A more useful facility would be an ordinary subroutine to print roman numbers:

```fortran
SUBROUTINE PRINT_ROMAN (R)
  TYPE(RCMAN), INTENT(IN) :: R
  PRINT '"" // R%ASSEEN // '""'
END SUBROUTINE PRINT_ROMAN
```

There are many possibilities. For example, arithmetic operations between objects of type roman and objects of type character might be defined and the character object validated prior to the operation being performed.

This rather trivial example illustrates the functionality. The facilities provided by Fortran 8x are very flexible in order to accommodate many useful extensions. These facilities allow the language to be extended to build new types on top of the existing types. A type may be named and defined. The values of the type may be represented. Objects may be declared to be of the new type. Operations can be defined to manipulate these objects.

4.3. Procedure Interfaces

Fortran has two kinds of procedures: functions and subroutines. Each of these has an external interface that consists of the name of the procedure; its kind, in the case of a function the type of its result; and the number, names, types, and attributes of its arguments. This information must be known to invoke the procedure correctly.

Fortran 8x has formalized a mechanism that allows this information to be made available to the invoking program unit. A processor can use this information to ensure that subroutine calls and function invocations are correct. Some features of Fortran 8x require that external interfaces be made explicit in a program unit by the presence of interface description blocks.

For libraries of mathematical software such as IMSL, NAG, or MSLIB, or for routines described by some means other than Fortran (assembler, C, etc.), procedure interfaces can contribute to safety, readability, and optimization. The interfaces for a couple of MSLIB routines would be specified as follows:

```fortran
INTERFACE
  SUBROUTINE ZPOLR (A, NDEG, Z, IER)
    REAL, INTENT(IN), ARRAY (NDEG+1) :: A
    INTEGER, INTENT(IN) :: NDEG
    COMPLEX, INTENT(OUT), ARRAY (NDEG) :: Z
    INTEGER, INTENT(OUT) :: IER
  END SUBROUTINE ZPOLR

  SUBROUTINE VDCPS (N, NPF, IPF, IEXP, IPW)
    INTEGER, INTENT(IN) :: N
    INTEGER, INTENT(OUT) :: NPF
    INTEGER, INTENT(OUT), ARRAY(13) :: IPF, IEXP, IPW
  END SUBROUTINE VDCPS
END INTERFACE
```

4.4. Modules

Fortran 77 has four kinds of program units: a main program, a subroutine, a function, and a block data subprogram. Fortran 8x adds a fifth: a module. The module has many uses, some of which are listed...
below:

- It provides a global data base of entities that can be accessed by name.
- Procedures that are logically associated and share data can be collected in a module. Information that is relevant only to these procedures can be hidden from other program units. This promotes safety and more highly structured programs.
- A module containing procedure interfaces can be constructed to provide correct usage of proprietary libraries of mathematical software, routines described by means other than Fortran, or other user-defined external routines.
- User-defined data types and the procedures that define the set of operations permitted for the types can be encapsulated in modules. For a user-defined data type to be global, it must be defined in a module.

Fortran 8x provides a better way of handling global data than the common blocks of Fortran 77. It will be possible to gradually upgrade an existing Fortran 77 program by moving a common block into a module. A routine that uses the module need not define the common block — in fact, it must not. When all routines that access data in the common block have been modified to use the module, the common declaration can be dropped from the module.

Information in a module can be declared to be PRIVATE. When a USE statement naming the module appears in another program unit, it can make accessible to the program unit only the public entities of the module. There is also an ONLY option with the USE statement that permits a user to further restrict the amount of information that is accessible. With greater control over the accessibility of information, there is less chance of inadvertent misuse.

There is another advantage to this information-hiding capability. When a module is used to encapsulate a data type and its operations, it is possible to change the underlying representation of the type or any of the procedures that define the operations for the type. This can be done without changing any of the program units that use the module, provided the external interfaces remain unchanged.

A module for the roman number type would look something like the following:

```fortran
MODULE ROMAN_NUMERALS
  TYPE ROMAN (ROM_SZ)
    PRIVATE
    INTEGER :: VALUE
    INTEGER :: LEN_ROM
    CHARACTER (ROM_SZ) :: AS_SEEN
  END TYPE ROMAN
  PARAMETER (MAXLEN = 30)
  SUBROUTINE I_TO_R (R, I) ASSIGNMENT
    I overloads intrinsic assignment (=)
    ...
  TYPE(ROMAN(MAX_LEN)) FUNCTION PLUS_ROMAN (R1, R2) OPERATOR (+)
    I overloads intrinsic operator +
    ...
  LOGICAL FUNCTION EQ_ROMAN (R1, R2) OPERATOR (.EQ.)
    I overloads intrinsic operator .EQ.
    ...
  INTEGER FUNCTION LEN (R1)
    I overloads intrinsic operator LEN
    ...
END MODULE ROMAN_NUMERALS
```
TYPE(ROMAN(MAX_LEN)) FUNCTION REAL_TO_R (A) OPERATOR (.RCM.)
 1 defines new unary operator LEN
...
SUBROUTINE PRINT_ROMAN (R)
 1 defines new subroutine
...
END MODULE ROMAN_NUMERALS

To use the user-defined type roman and its operations in another program unit, the following statement would be required:

USE ROMAN_NUMERALS

Because the type definition is declared to be PRIVATE, the names of its components would not be available to a program unit that is using the module. Likewise, the structure constructor would not be available. A routine could be added to the module to construct values in this case.

With modules, interfaces are more formally described, access to information can be better controlled, and procedures that are logically associated can be physically grouped together. In the case where a module entity has the same name as a local entity, the module entity can be renamed via an option available in the USE statement. These characteristics of modules were designed to promote the creation of reusable software.
5. OMISSIONS AND CONTROVERSIES IN FORTRAN 8X

Principal Author: Alex Marusak

5.1. Introduction

The ANSI Standards Subcommittee X3J3 for Fortran has recently released the draft proposed standard for public review. This article is the fifth in a series of articles describing the major features of the proposed standard. In this article we focus on the omissions and controversies in Fortran 8x. The omissions include both features described in the public review document in an appendix and major features considered and rejected by X3J3 over the past nine years.

The first X3J3 letter ballot to release Fortran 8x for public review was held in early 1986. The ballot failed. A recurring theme in the "no" votes was that Fortran 8x was too big. Following the failure of this first letter ballot, a Compromise Proposal was assembled and passed. The Compromise Proposal took some features that were in Fortran 8x at the time and moved them to a new Appendix F. These features were considered to have less support than others, or were more controversial, or could more easily be removed, or could significantly shorten Fortran 8x. As is implied in the name, the Compromise Proposal was designed to make everyone give up a little, but to make no one give up everything.

Appendix F is attached to Fortran 8x for the public review. It is the intent that items in Appendix F either will move back to the standard or will disappear entirely.

Other features that one might expect to find in Fortran 8x are missing entirely; these are discussed briefly. Some "gotchas" (or, depending on one's point of view, useful features) are discussed. Finally, some recurring reasons for "no" votes on the second, successful ballot are given.

5.2. Bit Data Type and Bit Manipulation Facilities

Most major scientific Fortran compilers have always included extensions for manipulating bits, usually as bit strings. Typically these extensions provided shifting and masking; octal and hexadecimal representations; and the ability to mix bit, logical, and integer data object expressions (and in some cases real data objects as well).

At one point X3J3 had voted to include a bit facility in Fortran 8x. This was not a bit string facility; only single bits could be defined. With the new array facilities, however, one could do some "string-like" manipulations.

Following the failure of the first letter ballot, the bit facility, in response to not a single "no" vote, was moved to Appendix F as part of the Compromise Proposal. None of the "no" votes in the first ballot asked for removal of the bit facility; all but one of the "no" votes in the second ballot asked for a bit facility to be added.

Sentiment remains strong for including a bit facility. At the latest International Standards Organization (ISO) meeting on Fortran in Liverpool, England, in August 1987, ISO members voted (22 yes, 3 no, 10 undecided) to include a bit facility in Fortran 8x.
5.3. Pointers

Scientific programmers in Fortran have for years used array indexing as an inelegant and nonstandard form of pointers. They would obtain the address of an array (LOC, LOCF, .LOC., etc.) and use this as an offset index in an array reference to map anywhere in their data (or code) space. By equivalencing arrays of different types and by ignoring Fortran rules of definition/undefinition, they could contrive a "typeless" pointer to boot.

X3J3 has attempted to provide, not an explicit pointer facility, but other high-level constructs to provide similar capabilities. The ALLOCATE/DEALLOCATE statements allow one to dynamically manage memory. The IDENTIFY mechanism allows one to map arrays on top of other arrays, and to alias scalars (including objects of user-defined types).

At the moment Fortran 8x has no facility to link lists. The variant structure facility, which was once included in Fortran 8x, also fell victim to the Compromise Proposal, and is now in Appendix F. IDENTIFY is strongly typed; one cannot, for instance, map an INTEGER array onto a LOGICAL array.

Sentiment remains strong for including pointer facilities in Fortran 8x. In August 1987 at Liverpool, England, ISO members voted (23 yes, 3 no, 10 undecided) to include pointer facilities in Fortran 8x.

X3J3 members are currently considering methods of providing more high-level pointer-like constructs, especially for linked lists. At the moment, there is no movement afoot to introduce an explicit POINTER facility.

5.4. Significant Blanks

Fortran has always treated blanks as insignificant, except in FORMATs and in character strings. Many X3J3 members feel strongly that Fortran should reverse this policy and treat blanks as significant characters, as do other high-level languages.

Fortran 8x has introduced a free-source form, in addition to the traditional fixed-source form. For some period of time, Fortran 8x treated blanks as significant in the free-source form. For reasons of upwards compatibility, blanks would have continued to be insignificant in the old fixed-source form. The Compromise Proposal removed significant blanks from free source form to a brief paragraph in Appendix F.

Few X3J3 members believe that the concept of significant blanks in computer languages is a bad idea. Opponents of significant blanks in Fortran generally believe that the added complexity of having both significant and insignificant blanks, depending on the source form, is more trouble than benefit. Advocates believe that significant blanks will result in cleaner constructs, easier parsing, and better error detection, and hence in safer code. Having significant blanks in Fortran would allow programmers to write tools that work on more than one language. A logical time to introduce significant blanks, if ever, is with the introduction of free source form. Fewer people are without opinions on the subject; at the ISO meeting on Fortran in Liverpool, England, in August 1987, members voted (26 yes, 8 no, 1 undecided) in favor of having significant blanks in free-source form.

5.5. Other Features Moved to Appendix F by the Compromise Proposal

5.5.1. Variant Structures

These structures allow a programmer to define alternative sequences of data variables. At most one sequence is defined at any instant, allowing memory savings by overlapping the variants. No particular
storage sequence is implied. Variant structures are a dynamic form of EQUIVALENCE; with EQUIVALENCE being a deprecated feature and variant structures being in Appendix F, there is no approved method of overlaying memory in Fortran 8x.

5.5.2. Structure Arrays of Arrays Treated As Higher-Order Arrays

At the moment, there is no way of directly referencing an array of arrays in a Fortran expression. One may, however, map a higher dimensioned array onto an array of arrays, through the IDENTIFY statement. Some X3J3 members consider this added step unnecessary and cumbersome. Others consider it the only logical way to address an array of arrays.

5.5.3. Vector-Valued Subscripts

Some X3J3 members believe that scientific array processing requires indirect, or vector-valued, subscripting (gather/scatter operations). This is supported by the hardware of some computer manufacturers as well. The principal objection to vectorized subscripts is the possibility of confusion in many-to-one stores.

5.5.4. FORALL Construct

The FORALL construct allows array operations mixing array element values and array element indices in expressions, a common mathematical construct. No order of evaluation is implied.

5.5.5. Extensions to Internal Procedures

Having introduced internal procedures, Fortran 8x at one time allowed any level of nesting. The Compromise Proposal, in an attempt to simplify the language, reduced nesting to one level. Furthermore, internal procedure names may not be actual arguments associated with dummy procedure names.

5.5.6. Condition Handling

A nicely structured condition-handling construct had been developed but fell victim to the Compromise Proposal, in the interests of simplification. Fortran 8x contains no facilities for responding to errors, to real-time interrupts, or to other unusual conditions.

5.6. Totally Omitted Features

All of the following features were considered by X3J3. Some were actually included in earlier drafts of the proposed standard. However, none of these are in Fortran 8x or in Appendix F.

5.6.1. Macros

There is no macro facility for manipulating source code in Fortran 8x. The MODULE construct, it is argued, obviates the need for much of the common usage of macros in Fortran. A proposal to include INCLUDE in Fortran 8x failed; this would have allowed straight text insertion, with no alteration through arguments.
5.6.2. Variable-Length Character Strings

There are no variable-length character strings in Fortran 8x. Character strings are fixed length, as they are in Fortran 77.

5.6.3. Specialized Loop Constructs

The DO-WHILE and REPEAT...UNTIL constructs are not a part of Fortran 8x. A majority of X3J3 believe that these loops are best provided with existing constructs, rather than with additional syntax.

5.6.4. Asynchronous Input/Output

There is no BUFFER IN/BUFFER OUT, nor is there an equivalent construct, to allow user control of asynchronous input/output.

5.6.5. Multitasking

Fortran 8x makes no provisions for multitasking. X3J3 believes that the state of the art in multitasking is too fluid to be locked into standard constructs at this time.

5.7. Some Compilation Dependencies in Fortran 8x

5.7.1. Data Structures

Consider the following two quotations from Fortran 8x:

- Section 11.3.3.3 Data Structures (page 11-4):
  A derived type may be defined in a module and accessed in a number of program units. This is the only way to access the same type definition in more than one program unit...

- Section 12.4.1.1 Arguments Associated with Dummy Data Objects (page 12-5):
  If a dummy argument is a dummy data object, the associated actual argument must be an expression of the same type or a data object of the same type...

Programmers may be surprised to learn that they cannot write a library using user-defined derived types unless they also define a MODULE to carry the user-defined type definition. Users of that library will have to USE the module in each procedure that references a library procedure with user-defined type arguments.

5.7.2. Explicit Interfaces

Consider the following quotation from Fortran 8x:

Section 12.3.1.1 Explicit Interface (page 12-2):
A procedure must have an explicit interface if any of the following is true: ... (2) The procedure has: ... (d) A dummy argument that is an assumed-shape or allocatable array ...
Some programmers may be surprised to discover that, if they wish to use the new whole-array notation in arguments, they must also now define an explicit interface for every procedure having assumed-shape or allocatable array dummy arguments. As whole-array notation becomes widely used and as array references become assumed-shape references, virtually every procedure (including those in libraries) that has array arguments will require an explicit interface.

Some X3J3 members strongly dislike these dependencies. Others think that these dependencies are a natural outgrowth of new Fortran 8x features, and as such are desirable.

5.8. Recurring Themes in the Second Ballot Responses

The recurring themes in the documentation accompanying the second X3J3 ballot on Fortran 8x were as follows:

- All but one of the “no” votes requested a bit facility. Many of the “yes” votes made the same request.
- Many “no” votes expressed the fear that the proposed new language is too big: (a) to read and understand easily; (b) to implement on small computers; and (c) to build compilers that will generate efficient code.
- Several “no” votes believe that X3J3 has strayed far from its charter — that instead of standardizing existing practice, X3J3 is designing a new language and calling it Fortran.
- The “yes” votes stress overwhelmingly that Fortran 77 is long due for an overhaul.
- The “yes” votes stress that public review is needed NOW. This is more important than getting every desirable feature into Fortran 8x. If a missing feature is important enough, it can be added in response to the public comments.
- The “no” votes stress that following the public review, significant changes will require not merely a majority vote (as before public review), but a two-thirds vote for approval.
6. A BRIEF COMPARISON OF FORTRAN 8x AND Ada*

Principal Author: Guylaine Pollock

6.1. Introduction

The ANSI Standards Subcommittee X3J3 for Fortran has recently released the draft proposed standard for public review. This article is the sixth in a series of articles concerning the proposed standard. In this article we compare Fortran 8x and Ada [4]. The two languages are compared on the topics of intrinsic and derived data types, data object declarations and specifications, array manipulation, program units, generics, input/output, environmental inquiries, and other miscellaneous features.

The reader should bear in mind that Ada and Fortran 8x are different programming languages with somewhat different goals and purposes. But, as both are general-purpose programming languages, there is overlap in their functionality. This article is provided to indicate where the languages differ and where they are similar. The concluding section gives some insight into how the different purposes motivate and determine the development of different programming languages.

6.2. Intrinsic and Derived Data Types

In comparing the intrinsic and derived data types between Fortran 8x and the strongly typed Ada programming language, it is evident that Ada provides more extensive typing facilities. This can be seen by examining the various type classes found in each of these languages.

6.2.1. Scalar Data Types

Both languages provide scalar data types which include integer, real, logical (or Boolean), and character types. Ada provides five predefined types that denote integer values: integer, long_integer, short_integer, natural, and positive; and three predefined types for real values: short_float, float, and long_float. In contrast, Fortran 8x provides a single integer type and two real types: real and double precision. However, Fortran 8x does provide a complex data type which is not defined in Ada. On the other hand, Ada has additional scalar data types (called enumerated data types) which are user defined, whereas Fortran 8x does not.

Facilities are provided in both languages to alter these intrinsic types, with Ada providing greater range in the types that may be altered. In particular, Fortran 8x does not provide facilities to define subtypes from the intrinsic data types, whereas Ada does. This facility allows the Ada programmer to specify constraints on the particular values from a base type and to define these values as a subtype. For example, in Ada one defines a subtype such as

```ada
subtype GRID_VALUES is INTEGER range -10 .. 10;
```

to limit the specific values that an object of this type may be assigned to the integer values ranging from -10 through 10. Subtypes may be used in Ada to constrain any type.

---

* Ada is a registered trademark of the US-Government-Ada Joint Program Office.
Although Fortran 8x does not allow one to constrain or alter specifications for integer data types, it does provide increased control, compared with Fortran 77, in specifying real data types. In Fortran 8x, one writes

\[
\text{REAL (PRECISION, EXPONENT\_RANGE) :: A}
\]

where PRECISION indicates the desired minimum precision and EXPONENT\_RANGE indicates the required minimum effective decimal exponent range. Ada, however, allows the programmer to specify either relative or absolute accuracy for real data types. For example, one may specify relative accuracy as

\[
\text{type PRECISE\_FLOAT is digits D range L .. U}
\]

where D indicates the desired minimum precision and L .. U indicates the desired lower and upper bounds for the range of values the type may represent. The specifications L .. U, therefore, can be used to specify minimum and maximum exponent ranges. Note that in Ada a specification of digits and/or range in certain type statements yields a distinct type; objects of different types then cannot be freely mixed in expressions, as the language is strongly typed. Fortran 8x, on the other hand, does not treat the specifications of the precision and/or exponent ranges as separate type declarations but treats such specifications as minimum requirements for precision and exponent range attributes of the supplied real (or complex) type.

Furthermore, Ada allows the user to define fixed-point real data types which specify absolute accuracy. An example of a fixed-point type in Ada is

\[
\text{type AMPS is delta 0.015 range 0.0 .. 15.0;}
\]

Fortran 8x does not have facilities for any fixed-point type but does provide an intrinsic COMPLEX type in which both precision and exponent range may be specified. Ada does not have an intrinsic COMPLEX type, but one may define such a type with available type definition facilities. Because COMPLEX is not intrinsic to the Ada programming language, the programmer must also define the necessary operations to manipulate COMPLEX data types in addition to defining the type.

The other scalar data types provided in both languages are character and logical. In Fortran 8x, these scalar types are denoted as CHARACTER and LOGICAL, whereas they are denoted as character, string, and boolean in Ada. Additional scalar types may not be defined in Fortran 8x. Programmers may define additional scalar types in Ada by enumerating an ordered set of distinct values that the type represents and are called enumerated data types — for example,

\[
\text{type WEEK\_DAY is (MON, TUES, WED, THUR, FRI);}
\]

Composite data types are available in both Fortran 8x and Ada. Composite data types may be record types and, in Ada, array types. An interesting conceptual difference between these languages is the fact that an array is considered an anonymous composite data type in Ada, whereas the array property in Fortran is an attribute of data objects, not of types. Record types must be defined by the programmer and may be heterogeneous structures. Both languages allow these user-defined types to be specified with parameters that essentially allow the creation of unconstrained types that can be constrained differently for individual objects. In Fortran 8x, type parameters pertaining to precision or exponent range information must be explicitly named PRECISION and EXPONENT\_RANGE.

In addition to allowing parameterized composite data types, variant record types may be defined in Ada that alter their composite structure based upon the parameterized values provided when an object is
elaborated. (Elaboration in Ada is discussed in the next section.) Appendix F of the proposed Fortran 8x describes variant record types; they are not part of the proposed Fortran standard but are included in an appendix for review purposes only.

Two classes of data that are available in Ada but not in Fortran 8x are access types and task types. Access types are essentially pointers. They may access objects that are created dynamically. Access types may be used to define recursive record types in that a component of the record structure may be an access type to the record structure being defined. Thus Ada also allows partial type definitions to satisfy elaboration requirements, for example,

```plaintext
type RECURSIVE_DEFINITION;
type POINTER is access RECURSIVE_DEFINITION;
type RECURSIVE_DEFINITION is
  record
    VALUE : INTEGER;
    SELF  : POINTER;
  end record;
```

A class of private and limited private types is available in Ada. These types enable the programmer to focus on the principle of information hiding, allowing implementation details and characteristics to be hidden from the user of the type. The only operations available on a private type are assignment, membership tests, selected components for the selection of any discriminant, qualification, explicit conversions, and tests for equality and inequality. Limited private types, however, have further restrictions on the implicitly available operations. Neither assignment nor the predefined comparison for equality and inequality is intrinsically available for such types.

Fortran 8x does have an accessibility statement that may appear in the scoping unit of a module or of a derived-type definition contained in a module. This statement may identify PUBLIC and/or PRIVATE entities contained within the module. This differs from the concept of private in Ada, as private entities in Fortran 8x may not be referenced outside the scoping unit in which they are defined. In Ada, private types may be exported but the inherent implementation characteristics are hidden, and available implicit operations are restricted.

6.3. Data Object Declarations and Specifications

In declaring data objects, both languages specify a type, a rank, a shape, and any other properties that might be germane to a particular type. In this regard, a number of similar attributes can be found. However, the inherent differences are more interesting to examine. Areas to peruse include the concept of elaboration in Ada, locality of definition, initialization of objects, implicit typing, and array attributes.

6.3.1. Elaboration

Special terminology is used in Ada to describe the process by which a declaration achieves its effect. This process, called elaboration, consists of the necessary steps in creating an object according to the given specification. It includes verifying that the object satisfies all indicated constraints and initializing where specified. Entities are elaborated in the order in which they appear in the declarative part and may not be referenced until they have been elaborated. Once the begin statement at the end of a declarative section has been reached, the declared objects are elaborated. The concept of elaboration applies to compilation units as well as declarative parts and items. The process of elaboration for an object declaration, for example, consists of elaborating the specified subtype, creating the object, and assigning any initial values. Ela-
boration of a subtype consists of elaborating the specified constraint and then checking to see whether the constraint is compatible with the base type or subtype. In Fortran 8x, an object is simply said to be declared once its specification has been encountered.

6.3.2. Locality

In Fortran 8x, objects may be accessed throughout the enclosing program unit in which they are defined and, in the case of data modules, may be used by other program units with the appropriate specifications. This concept also applies to objects used in the Ada programming language. In addition, Ada programmers may use block statements (declare blocks starting with a begin statement and ending with an end statement) within the executable code to define local objects that exist for the span of statements included in the begin portion of the block. These entities cease to exist once the block has been exited.

In Fortran 8x, there are no declare block statements. However, objects in internal procedures are in a separate scoping unit within a host scoping unit. As such, internal procedures have access to the host's objects; an internal procedure's local objects are inaccessible to the host.

6.3.3. Initialization

In Fortran 8x, data objects may be initialized when they are declared. In addition, the DATA statement can be used to initialize composite structures. The capabilities provided by the DATA statement in Fortran are also available in Ada but are expressed by the statements declaring individual objects instead of by a special statement. The notation used on the Fortran's DATA statement would permit one to initialize various parts of an array, whereas in Ada the entire array must be initialized at declaration if any part is initialized.

Of course, executable statements may be coded to initialize objects in both languages. In Ada, these executable statements may be part of the elaboration of the objects. For example, when a package is elaborated in Ada, an optional sequence of statements in its package body will be executed before the elaboration is completed.

6.3.4. Typing

In declaring data objects in Fortran 8x, implicit typing is allowed. This means that the actual use of an object may specify its type attribute through default typing rules or through the typing rules specified by an IMPLICIT statement. Objects may be accessed throughout the enclosing program unit in which they are defined and, in the case of data modules, may be used by other program units with the appropriate specifications. On the other hand, the IMPLICIT NONE statement is available in Fortran 8x and requires all data objects to be typed explicitly.

Ada is a strongly typed language, and object declarations are name-type associated. Thus two objects are considered to be the same type only if the same type name is used in their declarations. This is important to the programmer because intrinsic operations are defined only for objects of the same type. This means that two object values may not be added even if the respective objects have the same base type such as integer, but are declared as “new” subtypes or user-derived types. These strong typing facilities help to support and enforce data abstraction and in particular permit the compiler to enforce the typing rules. Although typing facilities have been added to Fortran, the language does not enforce strong typing.
rules even among objects with different base types.

6.3.5. Array Attributes

Fortran 8x provides an extensive array facility including operations on arrays and declarations of arrays. The concept of assumed-size arrays, in which an array dummy argument assumes the characteristics of the actual argument that is passed even if the rank is different from that of the dummy argument, is not allowed in Ada because it violates the concept of strong typing. The concept of assumed shape may be implemented in Ada through additional parameter passing. The concept of deferred shape is partially supported in Ada in that unconstrained array types may be specified. The concept of array aliasing in Fortran 8x, however, is not supported in Ada and is not possible because Ada has strong typing and the array property is a type property.

6.4. Array Manipulation

Although Ada provides more extensive typing facilities, Fortran 8x focuses more on array manipulation and correspondingly offers a few more facilities for creating and processing arrays. The Fortran array facilities have been covered in an earlier article but are addressed again briefly in this article to contrast the differences in array functionality between Ada and Fortran 8x.

Array attribute specifications in Fortran 8x identify the set of operations that may be performed on a particular array object. The attributes that influence possible operations include ALLOCATABLE and RANGE. Use of these attributes allows execution of the corresponding ALLOCATE and DEALLOCATE statements and SET RANGE statement.

Use of the ALLOCATABLE attribute indicates that the array memory space may be dynamically allocated with the ALLOCATE statement, and the DEALLOCATE statement can be used to reclaim array space that is no longer needed. An ALLOCATABLE array is a deferred-shape array. The bounds of such an array are determined when the space for the array is actually allocated.

6.4.1. Dynamic Allocation

Dynamic allocation of arrays is also possible in the Ada programming language, but the procedure is different. An allocator generates an object dynamically. An access type object is always used to reference a dynamically created object. A simple example of dynamically allocating an array in Ada is

```ada
type ARRAY_OBJECT is array(INTEGER range <> , INTEGER range <> ) of FLOAT;
type ARRAY_POINTER is access ARRAY_OBJECT;

ARRAY_POINTER := new ARRAY_OBJECT(10,15);
ARRAY_POINTER := null;
```

In Fortran 8x, one writes

```fortran
REAL, ALLOCATABLE, ARRAY (:,:) :: ARRAY_OBJECT

ALLOCATE (ARRAY_OBJECT(10,15))
DEALLOCATE (ARRAY_OBJECT)
```
6.4.2. Range

Fortran 8x allows the programmer to dynamically specify range dimensions for array objects either selectively or in named groups. To do so, the programmer must initially specify the object with the RANGE attribute. The SET RANGE statement is used to constrain the array. Ada allows one to specify unconstrained array types and correspondingly unconstrained objects. The objects may be constrained when they are elaborated, or the range constraint may be specified with an expression that is evaluated at run time.

6.4.3. Aliasing

Both Fortran 8x and the Ada programming languages allow the use of aliasing. In Fortran, one may alias scalar variable names, array elements, scalar structure components, substrings, or array names. In Ada, one may alias objects, exceptions, task entries, and subprograms. An object may not be aliased in Fortran unless it has been specified with an ALIAS attribute. Once it has been so specified, the IDENTIFY statement can be used to associate the entity with a parent object.

Two forms of the IDENTIFY statement exist, and the objects used within the statement determine which form is being used. The first, scalar alias, is utilized with scalar objects and simply provides a dynamic name-aliasing mechanism. This dynamic name-aliasing capability is provided by the aliasing feature in Ada, where its use is specified by a renames clause. This functionality allows the user to resolve name conflicts and introduce synonyms for entities.

The second form, dynamic array alias, provides the programmer with a dynamic remapping capability. If it is not used with care, it can produce surprising results. This statement allows the programmer to generate a one-to-one mapping of elements from one array to another in a nonsequential ordering. Thus, it provides even greater variability in selecting array elements upon which to operate than is provided by array sectioning references. Ada does not provide such functionality.

6.4.4. Array References

Fortran 8x allows the programmer to reference sections of an array through the use of subscript triplets. A subscript triplet is in the form \( L:U:S \), where \( L \) indicates the lower bound on the index value, \( U \) identifies the upper bound on the index value, and \( S \) represents the stride or difference between two index values in stepping from the lower bound to the upper bound.

Ada does not allow one to specify an array section quite so elegantly. However, one may specify an array section with aggregates that correspond conceptually to array sections in Fortran 8x. One does this by assigning the values of an array that is created to store the array section from an aggregate. The aggregate is formed by selecting the desired elements of the array to be sectioned as initialization values for assignment of the aggregate. A problem with this method is that except for the selected elements located in a sequential sequence, the elements must be specified individually for the assignment. Ada does provide an easier notation for specifying slices from one-dimensional arrays; one may represent an array slice for one-dimensional arrays by specifying a range value instead of a single index value.

In Fortran 8x, whole arrays may be referenced by their names, and operations may be applied to each element of the array by the appearance of an unsubscripted array reference. Furthermore, the order of applying an operation to each array element when a subscript is omitted from the array reference is undefined and implementation dependent.
Ada does not provide intrinsic array operations to reference whole arrays, but they can be defined by the programmer if desired by using the capability to overload operator symbols for procedure or function names. For example, one writes

```ada
type ARRAY_TYPE_1 is array . . .
type ARRAY_TYPE_2 is array . . .
function "+" (A_1 : in ARRAY_TYPE_1;
A_2 : in ARRAY_TYPE_2) return ARRAY_TYPE_1;
```

to supply the specification for the addition of two arrays. Operators may be placed between operands in Ada allowing one to reference the above function as "A_1 + A_2", a notation similar to that employed in Fortran 8x for whole array references.

Fortran 8x also provides a WHERE statement to mask array assignment when the reference is to the name of the array, rather than to a subscripted element of the array, which implies an element-by-element assignment of the array. This allows the programmer to selectively and dynamically determine the array elements to participate in an assignment. Ada does not provide a construct to mask array assignment.

6.5. Program Units

Fortran 8x provides main procedures, subroutines, functions, internal procedures, interface blocks, block data program units, and modules. The Ada programming language provides main procedures, subroutines, functions, internal procedures, packages, and tasks. Tasks are discussed briefly below in Section 6.9. Interface blocks provided in Fortran 8x allow the specification of external routines within the routine that will be referencing the external routine. Explicit passing of external routines in Ada is prohibited. Block data program units in Fortran 8x are used to specify initialization of entities within a named common block. The concept of a storage-associated common block is not supported in the Ada programming language.

In Ada, packages serve to syntactically and semantically encapsulate data objects and types and procedures to manipulate the encapsulated data entities. Packages also provide facilities to dynamically initialize encapsulated objects before elaboration of the package is completed.

Fortran 8x provides a module that can be used to encapsulate data objects. Initial values may be assigned to objects in modules, and procedures that specify operations on derived-type definitions may be encapsulated within the module. This approach permits the programmer to enforce data abstraction within the program structure. Modules in Fortran serve the same purpose as packages in the Ada programming language; however, executable statements cannot appear in Fortran modules.

Modules are accessed in Fortran 8x with a USE statement which allows one to access all objects within the module or to select a subset of the objects available in the module and to rename objects if potential name conflicts are a concern. In Ada, the corresponding concept is a package which is invoked using a combination of WITH and USE statements. However, the USE statement in Ada serves to achieve direct visibility and does not allow one to select a subset of the available objects from the package named in the USE statement. Finally, the USE statement in Ada does not allow direct visibility of homographs (different objects from different program units with the same name); that is, Ada, unlike Fortran 8x, does not allow one to rename objects by using the USE statement.
6.6. Generics

Fortran 8x does provide minimal generic facilities for various operations including attribute and environmental queries for arrays, character data, and numeric data. Also, intrinsic functions to manipulate the exponent and fractional parts of real data objects are provided. The generic parameters to these functions are objects. This approach is different from that taken in defining the Ada programming language, as all generic procedures are predefined in Fortran and the user does not have the capability to define his own generic routines.

In Ada, some generic routines are predefined within the language standard. In particular, input/output is implemented in Ada through instantiation of these predefined packages and procedures. Facilities are also provided, however, to allow the user to define his own generic routines. Generic parameters may be types and procedures in addition to objects.

Use of these generic routines is also different between the languages. In Fortran 8x, one simply uses the desired generic function while passing appropriate object structures for arguments. In Ada, however, generic routines must be instantiated and elaborated before they can be used. Instantiation is the process of specifying the actual types, objects, and procedures to be used in creating a specific instance of the generic template. Generic units must be instantiated before they can be elaborated.

6.7. Input/Output

The basic approaches to input and output in the Ada programming language and the proposed Fortran 8x standard are quite different. In Ada, input/output is expressed as operations on objects of some specific file type rather than in terms of external files. Generic packages are defined to provide input/output and the programmer must instantiate an appropriate package based on the types of objects he wishes to read or write. For example, packages that provide input/output processing for integer values and float (real) values may be defined as

```fortran
package INT_IO is new SEQUENTIAL_IO(INTEGER);
package FLOAT_IO is new SEQUENTIAL_IO(FLOAT);
```

Four generic packages are provided: Sequential_io, Direct_io, Text_io, and Low_level_io. Various operations are exported as procedures and functions from these packages. The specific operations depend on the type of input/output specified. For example, Sequential_io and Direct_io provide routines such as close, open, create, delete, read, reset, write, end_of_file, form, and is_open. Text_io provides routines to handle formatting of columns, lines, and pages.

Fortran 8x, on the other hand, manipulates records, not objects, and automatically selects the appropriate routines to handle text position based on a concise notation that specifies the desired spacing. Essentially, generic routines are also being used in Fortran for input/output operations, but the programmer is not required to explicitly create and invoke the routines.

6.8. Environmental Inquiries and Access

Both Fortran 8x and Ada provide facilities to make inquiries about the environment and various attributes of objects within the environment. Fortran provides intrinsic procedures and functions that can be used to ascertain certain aspects of the environment such as the presence or absence of an argument or the exact shape of an array argument. Most of these procedures focus on aspects of arrays or numerical objects.
Ada provides attributes and pragmas to query and alter the existing environment. Attributes are specified in the form PREFIX'ATTRIBUTE. The prefix may be an object type, task, label, entry, or program unit. The appropriate entity is determined by the attribute requested as different entities have different attributes. The attributes generally deal with object or machine characteristics. Ada provides approximately fifty predefined language attributes. A list of the attributes may be found in the Ada reference manual. Ada also provides fourteen language pragmas that can be used to alter the environment. An example of a pragma is

    pragma INLINE (program unit name);

which specifies that the given program unit must be placed inline wherever the routine is called. The pragma must appear immediately after the program unit declaration.

There is a great deal of overlap in functionality between Ada's attributes and Fortran 8x's intrinsic procedures. They differ, however, in two significant areas: (1) Ada permits inquiries about the declared attributes for precision and exponent ranges, whereas Fortran 8x does not; and (2) Ada does not define any floating-point manipulation functions and mathematical functions (square root, sinc, exponential, power function for powers of type real, etc.), whereas Fortran 8x does.

6.9. Miscellaneous

It is impossible to provide a complete analysis in this brief article of the differences and similarities between two languages designed for different purposes. It should be noted, however, that Ada provides several important facilities that have not been previously addressed and that are not available in Fortran 8x. These features include tasking, exception handling, and separate compilation.

Tasks are one of Ada's three primary program units. These units are used to specify concurrent operations that may occur, and provide a mechanism, called a rendezvous, for handling the synchronization and intercommunication among these operations. There is no comparable facility in Fortran 8x.

Exception-handling facilities in Ada allow the user to define his own exceptions in addition to those exceptions that are predefined within the language such as constraint_error, numeric_error, program_error, storage_error, and tasking_error. The programmer may write code to raise exceptions if they occur or may omit code testing for exceptions and let them occur dynamically. Code indicating how individual exceptions should be handled may be placed at the end of any block in which the programmer wishes to catch the exception. Exceptions are propagated up through nested blocks until they reach a level at which they can be handled.

Fortran 8x provides some predefined variables that can signal the status of a particular operation. One example is the use of STAT= specifier within the ALLOCATE statement and the use of ERR=, END=, and STATUS= specifiers in input/output statements. The programmer does not have facilities to define his own exceptions. It should be noted that, although an exception-handling facility does not appear in the draft proposed standard, an exception-handling facility with propagation rules similar to those of Ada and a user-specified scope is described in Appendix F for consideration during the public review of Fortran 8x.

All program units in Ada have a distinct and separate specification from their bodies. This allows the language to support separate compilation of the specifications and the bodies. The advantage in providing separate compilation is that it helps to support abstraction, information hiding, and modularity which are desirable properties in supporting good software engineering techniques. It also helps reduce compila-
tion time.

For program units such as packages that tend to encapsulate other program units, separate compilation support allows the specifications of the encapsulated units to appear within the package body independently of the bodies for the encapsulated units. Consequently, altering the body of a unit that is separate from its specification simply requires the recompilation of the body and not of the specification, unless the alteration changed the program unit's interface. If the body of an encapsulated unit is not compiled separately, then the entire encapsulating unit must be recompiled.

Program units in Fortran 8x do have a distinct specification part but do not allow separate compilation of the specification part of a program unit from its body or executable part. In Fortran, separate compilation refers to a slightly different concept, namely, the ability to compile the called program unit separately from the calling program unit. This capability still remains in Fortran 8x.

6.10. Conclusions

Before any conclusions can be drawn from this brief comparison of the proposed Fortran 8x standard and the Ada programming language, one should recall the basic goals and purpose of each language. Fortran's purpose is to provide a mechanism that facilitates development and execution of programs for scientific applications. It incorporates as many proven software engineering techniques as possible for timely production of high-quality code that is easily maintained and modified without reducing speed of execution. The Ada programming language is defined to facilitate development and execution of embedded real-time control applications and support the software engineering principles of abstraction, information hiding, modularity, localization, uniformity, completeness, and confirmability.

Fortran 8x allows the programmer greater flexibility in manipulating arrays, provides him with faster input/output facilities, and still provides support for basic software engineering principles such as data abstraction and information hiding. However, by providing the programmer with transformation and mapping facilities, Fortran 8x may still be used with a poor programming style leading to extremely difficult, if not impossible, codes to debug.

Ada has been shown to be a strongly typed language that restricts or even prohibits programming facilities provided in Fortran 8x. Overall, however, Ada provides more functionality than Fortran 8x. The question to be asked is whether this entire functionality is required for applications typically developed in Fortran 8x. Furthermore, certain aspects of Ada are lacking in appropriate support for coding various scientific applications. Most notable is the lack of intrinsic numerical routines.

If strong software engineering support is the driving factor in developing a code, the Ada programming language should be considered. If execution cycles are the primary goal, Ada may not be acceptable at this time, as a number of languages including Fortran provide greater execution speed. In particular, Ada's input/output facilities are comparatively slow. Thus, Fortran 8x may serve as an excellent compromise between the focus on execution speed found in Fortran 77 and the strong software engineering support found in Ada.
7. SUMMARY AND CONCLUSION

The collection of articles has reviewed the major aspects of the draft proposed Fortran standard, informally known as Fortran 8x. Fortran 77 programs are required to be standard-conforming programs in Fortran 8x. Features added to Fortran 77 vary from enhanced control constructs to a comprehensive array-processing facility to new mechanisms for global data and user-defined types. The articles have reviewed the controversial issues considered by X3J3, the facilities considered by the subcommittee and rejected for the reasons stated, and those rejected as too early to be candidates for standardization. Finally, a comparison with Ada is provided to expose the similarities and differences between the proposed Fortran and its potential competitor.

As with any change that affects the basic tools we use daily, there is concern that we will not be able to perform our programming tasks without a great deal of upheaval because of the new proposed standard. On the other hand, there is concern that Fortran will gradually lose its support by the majority of users unless Fortran is enhanced to meet the needs of advanced architectures and modern programming practice. If user support for Fortran diminishes, it will also create an upheaval in our programming environment because high-quality implementations of the complete language will be less prevalent. The following quote from The Prince by Niccolo Machiavelli (early 1500's) expresses this conflict eloquently:

Nothing is more difficult to carry out, nor more doubtful of success, nor more dangerous to handle, than to initiate a new order of things. For the reformer has enemies in all those who profit by the old order, and only lukewarm defenders in all those who would profit by the new order, this lukewarmness arising partly from ... the incredulity of mankind, who do not truly believe in anything new until they have had actual experience in it.

The subcommittee X3J3 has prepared a draft that it believes properly balances these conflicting goals for the time period when implementations will be prevalent. Now the subcommittee wants to know whether users agree.

The Language Working Group, by preparing and publishing these articles as widely as possible, believes that users of Fortran have enough information to assess the draft proposed standard for Fortran [1] and to determine how well it will provide a programming language in the 1990s to run applications. We encourage readers to consider the proposed standard carefully and to send comments to the Fortran standards subcommittee X3J3 on the good and bad aspects of the draft.

8. REFERENCES

Distribution for ANL-87-40

Internal:

J. M. Beumer (3)
F. Y. Fradin
H. G. Kaper
A. B. Krisciunas
F. M. Moszur
G. W. Pieper
B. T. Smith (63)

ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (5)

External:

DOE-TIC, for distribution per UC-32 (113)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
J. L. Bona, Pennsylvania State University
T. L. Brown, University of Illinois, Urbana
P. Concus, Lawrence Berkeley Laboratory
S. Gerhart, Micro Electronics and Computer Technology Corp., Austin, TX
H. B. Keller, California Institute of Technology
J. A. Nohel, University of Wisconsin, Madison
M. J. O'Donnell, University of Chicago
Jeanne C. Adams, National Center for Atmospheric Research
D. Austin, ER-DOE
Lawrence Berdahl, Lawrence Livermore National Laboratory
Gary Blair, Lawrence Livermore National Laboratory
Margaret M. Butler, National Energy Software Center
Roger Crane, David Sarnoff Research Laboratory
Bruce Curtis, Lawrence Livermore National Laboratory (10)
Ronald Detry, Sandia National Laboratories, Livermore
Donna Fink, EG&G Idaho Inc. (10)
Brian Ford, Numerical Algorithms Group Ltd.
Patrick W. Gaffney, Bergen Scientific Centre
Martin Gelbaum, Lawrence Berkeley Laboratory
Jerry Ivcy, EG&G Idaho Inc.
Rondall E. Jones, Sandia National Laboratories, Albuquerque
Neldon Marshall, EG&G Idaho Inc.
Jeanne T. Martin, Lawrence Livermore National Laboratory (10)
Alex Marusak, Los Alamos National Laboratory (10)
Loren P. Meissner, University of San Francisco
Paul C. Messina, California Institute of Technology
G. Michael, Lawrence Livermore National Laboratory
Cleve B. Moler, Dana Group
Leonard Moss, Stanford Linear Accelerator
Lester Petrie, Oak Ridge National Laboratory (10)
Guylaine Pollock, Sandia National Laboratories (10)
Barbara Privet, EG&G Idaho Inc.
Russell Rew, National Center for Atmospheric Research
Sunnie Sund, Stanford Linear Accelerator Laboratory (10)
Roy Thatcher, Fermi National Laboratory
Jerrold L. Wagener, Amoco Research, Tulsa
Joe Wolf, Lawrence Livermore National Laboratory (10)