EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN – CN /90/6

April 9, 1990

Fortran 90 – A Summary

Michael Metcalf

Computing and Networks Division, CERN

To be submitted to *International Journal of Modern Physics C*

FORTRAN 90 - A SUMMARY

Michael Metcalf

CERN, Geneva, Switzerland

Abstract

The fortunes of Fortran have changed markedly over the last two years. An increasingly gloomy outlook for the new standard, following publication of the first draft, culminated in a complete impasse in the summer of 1988. The deadlock was broken as the result of firm action by the international community and X3J3's positive response to that pressure, resulting in a second draft and period of public comment. The final result, now renamed Fortran 90, is outlined.

1. Standardization

X3J3 is a body composed of about forty-five representatives of computer hardware and software vendors, users, and academia. It is accredited to ANSI, (the American National Standards Institute, the body which publishes final American standards), but reports directly to its parent committee, X3, which is responsible for actually adopting, or rejecting, the proposed draft standards presented to it. In these decisions, X3 tries to ensure that the proposals really do represent a consensus of those concerned. The work of X3J3 takes place in collaboration with a corresponding International Standards Organization (ISO) group, ISO/IEC JTC1/SC22/WG5, consisting of international experts responsible for recommending that a draft standard become an international standard as well as a national one. X3J3 maintains other close contacts with the international community by having half-a-dozen foreign members.

As part of the lengthy standardization procedure, it is necessary to gain at least a two-thirds majority of X3J3 to be able to forward a proposed draft standard to X3, before that committee can consider it. Once X3 agrees to begin consideration, a period of public comment is initiated, and X3J3 is required to take note of and act on the public comments, and to resubmit a revised draft for consideration by X3. This cycle can be repeated several times, twice so far for this revision of the standard.

Formally, X3J3 is currently at the fourteenth of the eighteen milestones which have to be passed before a new standard receives full international recognition. A ballot held in January 1987 - 29 to 7 - showed that no true consensus had been achieved in the committee, and many of the dissenting votes were those of large vendors claiming to act as representatives of their users. This failure to agree was confirmed in the final ballot to forward the first draft of the new standard to X3, held in May 1987: 26 to 9, a sufficient but not overwhelming result.

In its turn, X3 decided by a large majority to allow the first draft standard, known at that time as Fortran 8x, to go to public comment, and over 400 letters were received. Many shades of opinion were represented in these replies, but a general feeling was that the language was too complex, lacked certain existing, popular

features, and was burdened by other features which were unattractive. As a result of this rather sobering experience, a number of members of X3J3 attempted to prepare plans to respond in various ways to this public sentiment, but none succeeded quickly in securing sufficient support from the committee. Finally, in September 1988, WG5, despairing of ever seeing a standard emerge from X3J3, defined exactly which changes *it* wanted to the original draft, and set its convenor a time-table to prepare a second draft conforming to those requirements. Defiantly, it renamed the proposed language Fortran 88! This action sent shock waves through the standards world, as an international committee was essentially reprimanding a national one and setting it an ultimatum. WG5's action received support from other bodies, including X3, which sent X3J3 a message reminding it of its international as well as its national obligations. In response to this pressure, X3J3 voted 24 to 9 in November 1988 to accept the principle of WG5's plan and, at a rather extraordinary meeting, passed proposals corresponding to 18 of WG5's wishes, rejected two, and continued work on the two remaining, more difficult ones — pointers and stream I/O. Work on these was completed in February 1989 and the resulting text was forwarded again in the summer of 1989 for both American and international public comment.

Following the more positive response shown in the nearly 170 new letters received, there is a definite feeling that the work is now on the home straight, and at least some of the vendors have changed their position from outright opposition to reluctant acceptance of the inevitable. Thus, X3J3 has renamed the language Fortran 90, and the final formal stages of its standardization are nearing completion. Only the last-minute decision of X3, to make Fortran 90, within the USA, an *additional* standard to Fortran 77 rather than a *replacement*, has produced a dissonant note in these latest developments. A complete description of this version of the language is given in [1], shortly to appear.

2. Features New to Fortran

This section summarises those features which have been added to Fortran 77 to create Fortran 90.

2.1 Source form

The new source form allows free form source input, without regard to columns. Comments may be in-line, following an exclamation mark(!), and lines which are to be continued bear a trailing ampersand (&). The character set is extended to include the full ASCII set, including lower-case letters (which in Fortran syntax are interpreted as upper case.) The underscore character is accepted as part of a name which may contain up to 31 characters. Thus, one may write a snippet of code such as

```
SUBROUTINE CROSS_PRODUCT (x, y, z) ! Z = X*Y
.
.
z1 = x(2) * y(3) -          &
x(3) * y(2)
```

Blanks are significant in the new source form, and may not be embedded inside tokens, such as keywords and variable names.

2.2 *Alternative style of relational operators*

In response to user demand, the deliberate redundancy of an alternative set of relational operators is introduced. They are

<	for .LT.	>	for .GT.
< =	for .LE.	> =	for .GE.
= =	for .EQ.	/ =	for .NE.

enabling statements such as

```
IF (x < y .AND. z**2 >= radius_squared) THEN
```

to be written.

2.3 *INCLUDE facility*

The widely used INCLUDE facility has been put into the language in response to American public comment. An INCLUDE line cannot reference a file containing partial statements. The MODULE/USE facility (Section 2.17) provides, amongst other things, a safer and more powerful general solution to most uses of the INCLUDE directive.

2.4 *Specification of variables*

The existing form of type declaration, as shown by

```
REAL a(5,5), b(5,5)
```

is extended to allow all the attributes of the variables concerned to be declared in a single statement. For instance, the statement

```
REAL, DIMENSION(5,5), PARAMETER :: a = (/ (0., i = 1, 25) /), &
                                     b = (/ ( i, i = 1, 25) /)
```

declares the two objects *a* and *b* to be named constant arrays, whose values are specified by the array constructors (see Section 2.13) following the equals signs.

For those who wish to define explicitly the type of all variables, the statement

```
IMPLICIT NONE
```

turns off the usual implicit typing rules.

2.5 Parameterized data types

The dispute about the complexity of the user-specified range and precision for real data in the first draft led to a complete reworking of this area of the standard, such that all the intrinsic data types are now parameterized. The way this is done is best explained by examples.

Suppose a given processor gives access to two *kinds* of integers — a default kind, represented by four bytes, and a short kind represented by two bytes. The processor might designate the default kind by a *kind type value* of 4, and the short kind by a value of 2. This value can be associated with the *kind type parameter* of a given variable as shown:

```
INTEGER(KIND=4) :: i      ! same as default
INTEGER(KIND=2) :: j      ! half the size
```

Constants corresponding to these kinds may be written. The value 99 might also be written as 99_4; a short integer constant is 63_2. It is, of course, not very portable to use explicit kind type values, and they can be parameterized, in this example for instance as

```
INTEGER, PARAMETER :: short=2
INTEGER(short)      :: j
.
.
j = 63_short
```

A similar, optional facility is available for logical entities which, when it is available, might pack down into bits or bytes, depending on the value of the kind type parameter, and for character entities, in order to accommodate non-European languages whose characters require more than one byte for their representation, e.g. Kanji. (This facility allows better support also for European languages.)

In the case of reals, at least two kinds must be provided, corresponding to the present single- and double-precision features. The kinds can be defined in terms of a range and/or precision, allowing a reasonable measure of portability. To define an array whose elements have a range of at least $10^{\pm 99}$ and a precision of at least nine decimal digits, we might write

```
INTEGER, PARAMETER :: long = SELECTED_REAL_KIND(9, 99)
REAL(long), DIMENSION :: a(1024)
a(1) = 1.23456789_long
```

where the intrinsic function `SELECTED_REAL_KIND` returns the kind value most closely but completely corresponding to the desired precision and range. This facility is of great importance for portable numerical software.

Other intrinsic functions are available as part of this facility, in particular an enquiry function, equivalent to the one just shown, for integers, returning effectively their decimal range.

2.6 Binary, octal and hexadecimal constants

The standard allows binary, octal and hexadecimal constants to be used in DATA statements, and integers to be read and written in the corresponding formats:

```
INTEGER :: i, j
DATA j / Z'abcd'/
.
.
WRITE (unit, '(Z4)' ) i
```

2.7 The POINTER attribute

Pointers have been added to Fortran in the form of an *attribute*, (rather than as a separate data type). The specification of an object with the pointer attribute designates a descriptor of the object, rather than the object itself. No storage is reserved — it has to be allocated explicitly.

A pointer may reference any dynamic object, such as an array whose actual bounds are defined at run-time, or any static object defined with the TARGET attribute. The use of the name of the pointer is, in most contexts, such as expressions or I/O lists, interpreted as a reference to the contents of the target. This is known as *automatic dereferencing*, i.e. no explicit notation using a pointer symbol is required. This means that existing code can readily be modified to use pointers, as only the specification statements need to be changed. Simple examples are:

```
REAL, POINTER          :: arrow ! may point to a real scalar
REAL, DIMENSION(:,:), POINTER :: pfeil ! may point to a 2D array
.
.
ALLOCATE(pfeil(10, 10))      ! allocate storage to pfeil
.                            ! define parts of pfeil
.
pfeil(1,1) = pfeil(5,5) + pfeil(10,10) ! the target values are used
```

Pointers are strongly typed, that is, they may point only at objects of the same type and type parameters as the ones with which they themselves are defined. Sometimes, of course, it is necessary to change the value of a pointer. For this case, a *pointer assignment* is foreseen. An example is shown in

```

REAL, DIMENSION(:, :), POINTER :: a, b, c
.
.
ALLOCATE (a(10,10), b(10,10)) ! give actual space to a and b
.
.
! define a and b
.
.
c => a                                ! c now points to the array a
DO i = 1, 2
    IF (i == 2) c => b                ! change target of c, no copy of data
    .                                ! on first pass, c refers to a
    .                                ! on second pass to b
! operations on c
.
.
END DO
.
.

```

Pointers may be components of a structure (see Section 2.16), and of the same type as the structure. In other words, a recursive definition is allowed which enables structures representing lists, trees and graphs to be defined. An example is

```

TYPE pixel
    REAL x, y
    INTEGER colour
    TYPE (pixel), POINTER :: next
END TYPE pixel

```

Pointers become associated with a target by execution of a pointer assignment or of an `ALLOCATE` statement. They become disassociated by, in general, execution of a `NULLIFY` statement. Storage occupied by a target may be returned to the processor by execution of a `DEALLOCATE` statement; this can result in a pointer being left 'dangling', and programmers must guard against their further use until they are newly associated. An intrinsic inquiry function, `ASSOCIATED`, indicates whether an association exists between a pointer and its target, or whether a pointer is associated at all.

Pointers may be used for static objects. In order that this should not inhibit optimization, any static object which might become a pointer target must be declared with the `TARGET` attribute. The compiler then knows which assumptions it can and cannot make about the references to static objects.

2.8 The *CASE* construct

The `CASE` construct allows the execution of zero or one block of code, selected from several blocks, depending on the value of an integer, logical, or character expression. An example is

```

SELECT CASE(3*i-j)
CASE(0)           ! for 0
.                ! executable code
CASE(2,4:8)       ! for 2, 4 to 8
.                ! executable code
CASE DEFAULT     ! for all other values
.                ! executable code
END SELECT

```

The default clause is optional; overlapping ranges are not permitted (this being in contrast to the IF...THEN...ELSE).

2.9 The loop construct

A new loop construct is introduced whose syntax is combined with that of the old form of the DO-loop. In a simplified form it is

```

[name:] DO [(control)]
    block of statements
END DO [name]

```

(where square brackets indicate optional items). If the control parameter is omitted, an endless loop is implied; if present, it has the form:

$i = \text{intexp1}, \text{intexp2} [, \text{intexp3}]$

The optional name may be used in conjunction with CYCLE and EXIT statements to specify which loop in a set of nested loops is to begin a new iteration or which is to be terminated, respectively.

The DO...WHILE has also been incorporated into the language, in order to satisfy American public opinion. It is redundant, as the DO construct and the EXIT statement provide the same functionality.

2.10 Program units

An enhanced form of the call to a procedure allows *keyword* and *optional* arguments, with *intent* attributes. For instance, a subroutine beginning

```

SUBROUTINE solve (a, b, n)
OPTIONAL, INTENT(IN) :: b

```

might be called as

```

CALL solve (n = i, a = x)

```

where two arguments are specified in keyword (rather than positional) form, and the third, which may not be redefined within the scope of the subroutine, is not given in this call. The mechanism underlying this form of call requires an *interface block* containing the relevant argument information to be specified.

Procedures may be specified to be recursive, as in

```
RECURSIVE FUNCTION factorial(x)
```

The old form of the statement function is generalized to an *internal procedure* which allows more than one statement of code, permits variables to be shared with the host procedure, and contains a mechanism for extending operators and assignment for use with derived-data types. This we shall return to in Section 2.17.

2.11 Extensions to CHARACTER data type

A number of extensions to the existing CHARACTER data type, in addition to the optional kinds, permit the use of strings of zero length:

```
a = ''
```

and the assignment of overlapping substrings:

```
a(:5) = a(3:7)
```

and introduce new intrinsic functions, such as TRIM, to remove the trailing blanks in a string. Some intrinsics, including INDEX, may operate, optionally, in the reverse sense.

2.12 Input/Output

The work in the area of input/output has been mainly on extensions to support the new data types, and an increase in the number of attributes which an OPEN statement may specify, for instance to position a file or to specify the actions allowed on it.

In addition, there are new specifiers on READ and WRITE statements. The first, the ADVANCE specifier for formatted, sequential files, allows data to be read and written without being concerned with any record structure, if it has the value NO. It is, for instance, possible to read a specified number of characters from part of a record, to remain positioned at that point in the record, and to read from the remainder with a subsequent read. At present, a single read would cause the whole record to be read, regardless of the length of the input list.

The second provides a NAMELIST feature, illustrated by

```
NAMELIST/list/a, i, x
READ(unit, NML = list)
```

which would expect an input record of the form

```
&LIST X = 4.3, A = 1.E20, I = -4
```

2.13 Array processing

The introduction of array processing features is one of the most important new aspects of the language. The reasons are threefold: an array notation defined in the language simplifies the syntax; new features extend the power of the language to manipulate arrays; and the concise syntax makes the presence of array processing obvious to compilers which, especially on vector processors, are able to optimize object code better.

An array is defined to have a *shape* given by its number of dimensions, or *rank*, and the *extent* of each dimension. Two arrays are conformable if they have the same shape. The operations, assignments and intrinsic functions are extended to apply to whole arrays on an element-by-element basis, provided that when more than one array is involved they are all conformable. When one of the variables involved is a scalar rather than an array, its value is distributed as necessary. Thus we may write

```
REAL, DIMENSION(5, 20)    :: x, y
REAL, DIMENSION(-2:2, 20) :: z
.
.
z = 4.0 * y * SQRT(x)
```

In this example we may wish to include a protection against an attempt to extract a negative square root. This facility is provided by the WHERE construct:

```
WHERE (x >= 0.)
  z = 4.0 * y * SQRT(x)
ELSEWHERE
  z = 0.
END WHERE
```

which tests *x* on an element-by-element basis.

A means is provided to define *array sections*. Such sections are themselves array-valued objects, and may thus be used wherever an array may be used, in particular as an actual argument in a procedure call. Array sections are selected using a triplet notation. For an array

```
REAL, DIMENSION(-4:0, 7) :: a
```

a(-3,:) selects the whole of the second row (*a*(-3,1),... *a*(-3,7)), and *a*(0:-4:-2, 1:7:2) selects in reverse order every second element of every second column.

Just as variables may be array-valued, so may constants. It is possible to define a rank-one array-valued constant as in

```
(/1, 1, 2, 3, 5, 8/)
```

and to reshape it to any desired form:

```
REAL, DIMENSION(2, 3) :: a
a = RESHAPE( (/ (I, I = 1, 6) /), (/2, 3/) )
```

where the second argument to the intrinsic function defines the shape of the result, and the first defines an array of the first six natural numbers, using the implied-DO loop notation familiar in I/O lists.

Many vector processors have a gather/scatter capability. This can be directly accessed in Fortran 90 by use of the vector-valued subscript notation:

```

      INTEGER, DIMENSION(1024) :: i, j
      REAL, DIMENSION(1024)    :: a, b
      .
      .
! Define i, j and a
      .
      .
      b(j) = a(i)                ! permute the elements of a into b

```

2.14 Dynamic storage

Fortran 90 provides three separate mechanisms for accessing storage dynamically. The first mechanism is via the `ALLOCATE` and `DEALLOCATE` statements which, as their names imply, can be used to obtain and return the actual storage required for an array whose type, rank, name, and allocatable attribute have been declared previously:

```

      REAL, DIMENSION(:), ALLOCATABLE :: x
      .
      ALLOCATE(x(n:m)) ! n and m are integer expressions
      .
      x(j) = q
      CALL sub(x)
      .
      DEALLOCATE (x)

```

Unless an allocated array has the `SAVE` attribute, it becomes undefined whenever a `RETURN` or `END` statement is executed in the procedure in which it is declared. For good housekeeping, such arrays should be explicitly deallocated. The fact that allocation and deallocation can occur in random order implies an underlying heap storage mechanism.

The second mechanism, useful for local arrays with variable dimensions, is the *automatic* array:

```

      SUBROUTINE sub(i, j, k)
      REAL, DIMENSION(i, j, k) :: x ! bounds from dummy arguments

```

whose actual storage space is provided (on a stack) when the procedure is called.

Finally, we have the *assumed-shape* array, whose storage is defined in a calling procedure (so it is not strictly dynamic), and for which only a type, rank, and name are supplied:

```

      SUBROUTINE sub(a)
      REAL, DIMENSION(:, :, :) :: a

```

Various enquiry functions may be used to determine the actual bounds of the array:

```
DO (i = LBOUND (a, 3), UBOUND (a, 3))
  DO (j = LBOUND (a, 2), UBOUND (a, 2))
    DO (k = LBOUND (a, 1), UBOUND (a, 1))
```

where LBOUND and UBOUND give the lower and upper bounds of a specified dimension, respectively.

2.15 Intrinsic procedures

Fortran 90 defines about 100 intrinsic procedures. Many of these are intended for use in conjunction with arrays for the purposes of reduction (e.g. SUM), inquiry (e.g. RANK), construction (e.g. SPREAD), manipulation (e.g. TRANSPOSE), and location (e.g. MAXLOC). Others allow the attributes of the working environment to be determined (e.g. the smallest and largest positive real and integer values), and access to the system and real-time clocks is provided. A random number subroutine provides a portable interface to a machine-dependent sequence, and a transfer function allows the contents of a defined area of physical storage to be transferred to another area without type conversion occurring.

Of particular note is that the bit string manipulation functions and subroutine of MIL-STD 1753 have been incorporated into the language; their definitions have been extended to handle integer arrays in a completely upwards-compatible fashion.

2.16 Derived-data types

Fortran has hitherto lacked the possibility of building user-defined data types. This will be possible in Fortran 90, using a syntax illustrated by the example

```
TYPE staff_member
  CHARACTER(LEN=20) :: first_name, last_name
  INTEGER           :: id, department
END TYPE
```

which defines a structure which may be used to describe an employee in a company. An aggregate can be defined as

```
TYPE(staff_member), DIMENSION(1000) :: staff
```

defining 1000 such structures to represent the whole staff. Individual staff members may be referenced as, for example, staff(no), and a given field of a structure as staff(no)%first_name, for the first name of a particular staff member. More elaborate data types may be constructed using the ability to nest definitions as in

```
TYPE company
  CHARACTER(LEN=20)           :: name
  TYPE(staff_member), DIMENSION(1000) :: staff
END TYPE

TYPE(company), DIMENSION(20) :: companies
```

to build a structure to define companies.

Subject to the conditions that the type definition contains a SEQUENCE specifier and that its components ultimately resolve to all numeric or all character types, objects of derived-data types may be placed in COMMON blocks. The purpose of the SEQUENCE specifier is to force the compiler to lay out the components of the structure in memory as it would lay out the same sequence defined as a list in a COMMON block under the usual rules of storage association.

2.17 Data abstraction

It is possible to define a derived-data type, and operations on that data type may be defined in an internal procedure. These two features may be combined into a module which can be propagated through a whole program to provide a new level of data abstraction. As an example we may take an extension to Fortran's intrinsic CHARACTER data type whose definition is of a fixed and pre-determined length. This user-defined derived-data type defines a set of modules to provide the functionality of a variable length character type, which we shall call *string*. The module for the type definition might be

```
MODULE string_type
  TYPE string(maxlen)
    INTEGER                :: length
    CHARACTER(LEN=maxlen) :: string_data
  END TYPE String
END MODULE String_type
```

With

```
USE string_type
.
TYPE(string(60)), DIMENSION(10) :: cord
```

we define an array of 10 elements of maximum length 60. An actual element can be set by

```
cord(3) = 'ABCD'
```

but this implies a redefinition, or *extension*, of the assignment operator to define correctly both fields of the element. This can be achieved by the internal procedure

```
SUBROUTINE c_to_s_assign(s,c)
  TYPE (string)    :: s
  CHARACTER(LEN=*) :: c
  s%string_data = c
  s%length      = LEN(c)
END SUBROUTINE c_to_s_assign
```

and the interface block

```
INTERFACE ASSIGNMENT(=)      ! Extend assignment operator
  MODULE PROCEDURE c_to_s_assign
END INTERFACE
```

which can both be included in the module, together with other valid functions such as concatenation, length extraction, etc., to allow the user-defined string type to be imported into any program unit where it may be required, in a uniformly consistent fashion.

3. Backwards compatibility

The procedures under which X3J3 works require that a period of notice be given before any existing feature is removed from the language. This means, in practice, a minimum period of one revision cycle, which for Fortran means a decade or more. The need to remove features is evident: if the only action of the standards committee is to add new features, the language will become grotesquely large, with many overlapping and redundant items. The solution adopted by X3J3 is to publish as an Appendix to the standard a set of two lists showing which items have been removed or are candidates for eventual removal.

The first list contains the Deleted Features (those which in the previous standard were listed as Obsolescent Features) which have now been removed. There are no Deleted Features in Fortran 90 which contains the whole of Fortran 77.

The second list contains the Obsolescent Features, those considered to be redundant and little used, and which should be removed in the next revision (although that is not binding on a future committee). The Obsolescent Features are:

Arithmetic-IF	Real and double precision DO-variables
Shared DO termination	DO termination not on CONTINUE (or ENDDO)
Alternate RETURN	Branch to ENDIF from outside block
PAUSE	ASSIGN and assigned GOTO

4. Conclusion

This paper has set out the background to the current revision of the Fortran language standard, and given an overview of its principal new features. This standard, like any other, is a compromise between conflicting points of view, and is not satisfactory to everyone. On the other hand, the presence of a widely accepted and implemented standard provides the only practical means to write and to use portable programs, and the evolution of that standard is the only means to escape from the temptation to use vendors' extensions or private dialects and languages. It is my opinion that its final acceptance and implementation are the only way to avoid a future of anarchy and chaos in scientific programming.

Acknowledgements

I thank J. Blake, J. Bunn, T.M.R.E. Ellis, F. James, and H. Lipps for their careful reading and suggestions on drafts of this paper.

Bibliography

1. M. Metcalf and J. Reid, *Fortran 90 Explained* (Oxford University Press, Oxford and New York, 1990).