Supplement to the Minutes of Meeting 110 (Part 1)

X3J3 Fortran

13 to 18 November 1988

Boston, Massachusetts

X3J3/229

CONTENTS

Pre-Meeting Distibution for the 110th X3J3 Meeting

Cambridge, Massachusetts November 13-18, 1988

110-JCA-1 (1)

MEMO TO: X3J3

FROM: Jeanne Adams Jeanne Adams

DATE: September 9, 1988

It is with much regret that I plan to withdraw my name from consideration by WG5 as chair of the Paris meeting. I have always enjoyed this work over the past decade, remembering many pleasant occasions. I made this decision after considerable thought over the past few weeks. Given the controversy over the draft standard and the proposals brought forth at the Jackson meeting, I feel that my situation would be ambiguous if I served as chair.

There are many reasons for this, an important one being the direction that X3J3 posed for its delegation to keep the technical development charge within X3J3 itself. I feel that under these circumstances I would be unable to be impartial, since my charge from you is unequivocal. My role as chair of X3J3 must be my first responsibility.

At the meeting, I will make a statement to this effect.

I am still in support of the full language for Fortran (ABMSW) as modified by the simplifications and deletions called for by the public comment. However, each of us on X3J3 has his or her own favorite plan. The work done on this document by the five members of X3J3 is technical work in the nature of what we all do in preparing for meetings or summarizing our points of view. It reflects the work of the past 10 years on X3J3, and is a matter of record. My support for the full language model of S8 does not cause me a conflict of interest. X3J3 has decided to prohibit the presentation of this plan to WG5. That decision I regret as being precipitous and not well-advised, since this is a public international meeting.

I am hoping that at the WG5 meeting member countries and plan authors will be able to come to terms quickly with the plan that will be acceptable to both X3J3 and WG5. That would be the best of all possible worlds. I plan to work very hard at the Paris meeting to achieve this compromise. When I return from Paris, I will send you my impressions of the meeting in an informal note. Both Andy Johnson and Jeanne Martin will have formal reports for you.

(2)

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
Scientific Computing Division
P. O. Box 3000 • Boulder, Colorado • 80307
Telephone: (303) 497-1275 • FTS: 320-1275 • Telex: 45 694

110-JCA-2

August 23, 1988

L. J. Gallagher
National Bureau of Standards
Building 225, Room A156
Gaithersburg, Md 20899

Dear Len;

I have appointed Kevin Harris of Digital Equipment as Liaison to X3T2 from X3J3, Fortran. Would you place his name on your list of liasion contacts and mail him any relevant material on the work of X3T2?

Kevin Harris
ZKO 2-3/N30
Digital Equipment Corp.
110 Spit Brook Road
Nashua, NH 03062

Sincerely yours,

Jeanne Adams, Advanced Methods Group
Chair, X3J3

cc. X3J3 Distribution
Kevin Harris

3

(3)

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
Scientific Computing Division/Advanced Methods Section
P. O. Box 3000 • Boulder, Colorado • 80307
Telephone: (303) 497-1275 • FAX: 497-1137 Telex: 989764

110-JCA-3

August 24, 1988

L. G. J. Ter Haar
Expl. & Prod. Laboratory
Kon Shell
Volmerlaan 6
N1-2288 Gd Rijswijk
Netherlands

Dear Leo;

Thank you for your letter about Fortran. I will distribute it in the pre-meeting for the November meeting. I will also distribute your paper on simplification of precision. There will be three different simplification plans to be presented as tutorials in November.

Brian Smith is heading the group on J3 that is looking into parallelism. You might ask him to keep your name on any mailing list that he has.

Will I see you in Paris at the WG5 meeting?

Regards,

Jeanne Adams, Advanced Methods Group
Chair, X3J3

cc. Brian Smith

5

Koninklijke/Shell Exploratie en Produktie Laboratorium

Shell Research B.V.

110-JCA-3

(3)

Ms. Jeanne C. Adams
National Center for Atmospheric Research
P.O. Box 3000
Boulder, Colorado 80307
U.S.A.

Uw/Your ref.:

Onze/Our ref.: LRG/2

Rijswijk Z-H, July 29, 1988
Postadres: Postbus 60, 2280 AB Rijswijk Z-H
Telefoon (070) 11... ...../113911

Dear Jeanne,

Recently I received the minutes of the 108th X3J3 meeting. My primary reaction was one of disenchantment, my secondary was far more positive as I believe that the standardisation efforts are now going the right direction.

Let me elaborate on both. I was disappointed to read that the concept of deprecation might be deleted from the language. Although it may be true that some American comments indicate strong opposition towards deprecation, I have always had the impression that ISO WG5 strongly supports the concept. Even if we never succeed in abolishment of the old Fortran concepts, I think it is about time that we have at least a mechanism to pinpoint the bad eggs in the basket. Moreover I am strongly convinced that the deprecation concept will work eventually despite even the strongest opposition. In order to keep Fortran alive, it must be a living language. For that reason I also support strongly the name change from FORTRAN to Fortran.

On the positive side I rate the remainder of the revision plan for various reasons except for perhaps two items. I am not in support of adding a DO WHILE. The present proposal is in my opinion far stronger with the DO EXIT concept. From the discussions around FORTRAN 77 I still remember the fuzz about the multiple branch construct. Devotees of Pascal even refused the ELSE IF THEN until they discovered the real strength of this statement. DO EXIT and ELSE IF THEN, although not basic closed control structures do support good programming practices and are admirable flexible constructs.

(3)

110-JCA-3

The second item on which at least I have my doubts is adding INCLUDE to the language. I propagated among many colleages "Fortran 8X Explained". It is remarkable how many are enthousiastic about the MODULE concept exactly for the reasons they want the INCLUDE, i.e. for defining global variables. A textual INCLUDE is dirty, but sometimes unavoidable, programming practice, to say the least. Unless we make INCLUDE as safe as MODULE, I think we should not adopt it. I rather would give up (for the time being) the MODULE as a means for data abstraction than allowing bad programming practices through the backdoor.

I was pleasingly surprised to read that efforts have started on developing constructs for paralellism and multi-tasking. As a matter of fact I would be highly interested to receive information on the progress of the Parallel Computing Forum. If possible I would even be interested to contribute. May I hear from you how I could keep up to date with the developments?

On the subject of simplification of generalised precision I made a proposal which I would like to be included in the premeeting distribution of the forthcoming WG5 meeting in Paris. If required I would be very willing to formally introduce this proposal at the meeting. As you may see it is a plea for introduction of "KINDeable" intrinsics, which in one form or another has already turned up in several proposals before. I cannot think of a better name (would TYPEable be appropriate?). It is not a worked out proposal, but is rather meant as food for thought. I realise that implementation of this proposal will mean a lot of editorial work. If however there is sufficient support in X3J3 for this proposal, I think it is well worth the trouble.

The main reasons why this proposal deserves a close look are:

- it is easier to understand than the current generalised precision in F8X
- it supports better programming practices
- it is an extendable feature with opportunities for future Fortrans
- the method is directly applicable to Japanese, Chinese or other character types.

(3)

110-JCA-3

In order to make sure that in the compromise not a facility is lost that is absolutely required for writing portable numerical software I made an alternative proposal that (I think) essentially provides the same facility, namely an absolute control of the accuracy of scientific calculations. In my view current practices with REAL* or the proposed HIGH PRECISION nowhere come near the mark. After much thought I even prefer the latter, far simpler proposal.

As from the 1st of August I have retired from Shell. I will however keep my activities in Fortran standardisation. Would you be so kind to have my address changed as from that date? I wish you and your colleages in X3J3 all success in the forthcoming meetings.

Best regards,

Leo ter Haar
Koninklijke/Shell Exploratie
en Produktie Laboratorium
P.O.Box 60
2280 AB RIJSWIJK
The Netherlands

from 1st August:
Ds van de Boschlaan 36
2286 PM Rijswijk
The Netherlands

110-JCA-3
(3)

To: WG5, X3J3
From : Leo ter Haar
Re : Simplification Generalised Precision
Date : 25 July 1988

1. Introduction

1.1 Background

Many critiques of Fortran 8X have concentrated on the concept of generalised precision. This is strange as it provides a facility for developing numerical software that is yet unsurpassed in any other language. Why then meets such a powerful feature so much opposition? If it is not the feature itself, it must be the syntaxis. I believe that the method chosen for generalised precision, namely that of parametrisation, is basically wrong. This leads to descriptions in the standard that are difficult to understand, and if fully understood still leaves doubts.

For instance in the TYPE description on the type NODE (page 5-5) I am still not sure whether the following is standard conforming:

```
TYPE NODE (PRECISION, EXPONENT_RANGE, M)
  REAL (PRECISION =PRECISION, EXPONENT_RANGE =
  EXPONENT_RANGE)                                  :: DOT
  CHARACTER (M)                                    :: DASH
END TYPE NODE
```

And if so why is then the following incorrect:

```
TYPE NODE (M1, M2, M)
  REAL (PRECISION = M1, EXPONENT_RANGE= M2)  :: DOT
  CHARACTER (M)                              :: DASH
END TYPE NODE
```

Furthermore PRECISION and EXPONENT_RANGE only define part of the characteristics of the generalised precision type. The EXPONENT LETTER statement is(???) still needed for definition of constants of a certain type.

All of the above would be so much easier to understand if we had REALs (and COMPLEX) of another KIND. The above NODE definition would then read:

```
TYPE NODE (M1, M2)
  REAL (KIND=M1) :: DOT
  CHARACTER (KIND=M2) :: DASH
END TYPE NODE
```

The problem with KIND is however that it is almost impossible to describe the full characteristics of a generalised precision type with one single parameter.
The standard says quite rightly (introduction of chapter 4) :
" A data type is characterised by a set of values, a means to denote the values, and a set of operations that can manipulate

and interpret the values." In other words, the standard defines for (default) intrinsic types:

- the type definition, i.e. the ways and means to define such types (e.g. the implicit typing rule for reals)
- data object declarations
- source representations of constants of such type
- edit representations of values of such a type
- meaning and interpretation of intrinsic operators for such type
- coercion rules

The internal representation is necessarily hardware dependent.

What is really needed is a mechanism whereby a Fortran user can define types of real (complex), integer, character, logical, that can efficiently be implemented by processors using native internal representation. The solution may be found in the KIND statement (see below), which is worked out for the REAL type only (would FLOAT be more appropriate?)

1.2 Some thoughts about generic portable software

One would think that generic intrinsic functions would have the accuracy of the argument. Unfortunately this is not always true in current implementations of Fortran 77. I believe this situation must be corrected in Fortran 8X.

On the other hand it should be possible to write generic software with user defined accuracy. (of course this is possible by adding a dummy argument).

1.3 Basic framework for the proposal

- The KIND statement is introduced to define intrinsic types whose characteristics differ from the default intrinsic types.
- The type-spec for intrinsic types has an optional KIND-selector whose value must have been defined before in a KIND statement.
- ??? KIND = 0 is equivalent with default type ???
- Each intrinsic type has its own set of KIND-specifiers.
- Intrinsic types of a defined KIND must not be EQUIVALENCEd. They have no defined storage characteristics.
- Examples in the text of the document should only refer to default types unless in the context of the description of the KIND specifiers for that type.

1.4 Advantages of KIND

In the first place it supports good programming practices. The KIND specifier may only be used if it is previously defined. This provides an easy check for the compiler.
It is easy to understand both for implementor and for the Fortran user. The feature can easily be described in separate paragraphs of the standard. Especially if we keep the examples in the

standard confined to default types unless in the context of the description of non default types, the average reader of the document may easily skip the difficult parts.
The greatest advantage of the proposal is its extendability. Suggestions for extensions are:

- CHARACTER KIND with KIND specifiers like LEN =, LANGUAGE = 'katakana', LANGUAGE = 'swedish' etcetera
- LOGICAL KIND with specifier PACKED = 'BIT' , 'BYTE' Would one need a separate BIT type?
- ENUMERABLE specifier
- If we introduce a POINTER intrinsic, the KIND mechanism would leave the way open for later extensional pointer types

1.5 About the LEN parameter

As a consequence of the proposal the LEN parameter of the CHARACTER type may have to be deleted from the language. This is not a great loss for various reasons. The most important reason is that the introduction of the LEN parameter introduces an endless series of standard conforming ways of writing the CHARACTER statement, e.g.

```
CHARACTER*20      MESSAGE
CHARACTER*(20)    MESSAGE
CHARACTER*(20),   MESSAGE
CHARACTER(LEN=20) MESSAGE
CHARACTER(20)     MESSAGE   ! etcetera
```

Especially the last example is no improvement on Fortran 77 whatsoever.

1.6 Discussion of some alternatives

C implementation is not extendable for more than double precision
Common practice REAL*4 etcetera does not work for generic software

2. Proposal 1 KINDable intrinsics

2.1 The KIND statement

Default intrinsic types have characteristics that are defined by the language. Some of the intrinsic types may be be parameterised by means of a KIND parameter. The KIND statement provides a means of declaring non default characteristics to these intrinsics. The syntax of the KIND statement is :

KIND def-type (kind-spec) (kind-param-spec-list)

```
def-type        is REAL (or FLOAT?)
                or CHARACTER
                or LOGICAL
                or INTEGER
```

```
kind-spec        is positive integer-contant-expr (or integer-
                 constant?)

kind-param-spec  is float-param-spec
                 or character-param-spec
                 or logical-param-spec
                 or integer-param-spec

The type declaration for non default types is
                    REAL ([KIND=]kind-spec)
                 or COMPLEX ([KIND=] kind-spec)
                 or ................
```

Intrinsics of a non default KIND may only be EQUIVALENCEd with types of the same kind.

2.2 Generalised precision

```
The param-specs for the FLOAT kind are given by:
    real-param-spec  is  PRECISION = prec-expr
                     or  EXPONENT-RANGE = exp-expr
                     or  EXPONENT-LETTER = letter (other than
                         D,E, or H)

    prec-expr        is  integer-constant-expr (positive)
                     or  EFFECTIVE_PRECISION (dummy argument)

    exp-expr         is  integer-constant-expr (positive)
                     or  EFFECTIVE_EXPONENT_RANGE (dummy
                         argument)
```

constraints :

There may only be one KIND statement using dummy argument (which must be present). This implies generic Maximum allowed value for the value of precision is processor dependent but at least ???? Ditto exponent range If generic precision is used the interface must be present in the calling program unit.

3. Alternative proposal generalised precision

Background

This proposal is based on the minimum that is required to achieve the following functionality:

- control in Fortran on the decimal(sic) accuracy of scientific calculations.
  No such funny things as meaningless byte accuracy!!!
- the possibility to write portable numeric (i.e. generic) software.

In this proposal it is supposed that exponent range does not play a critical role in scientific calculations other than by means of

the EFFECTIVE_EXPONENT_RANGE enquiry function. Furthermore one must have an unequivocal means of defining accuracy of numeric constants.

Proposal

- The REAL and COMPLEX types may be parameterised by means of the PREC specifier:
  prec-spec is [PREC =] positive integer-constant-expr
  or [PREC =] *
- The maximum allowed value of the integer-constant-expr is processor dependent (but at least ???)
- The * parameter may only appear in subroutines or functions which must be declared GENERIC. (GENERIC FUNCTION name etc)
- For generic subprograms the calling program must contain the GENERIC declaration (or the whole interface if you like)
- The GENERIC declaration has the form GENERIC subprogram-name (note the similarity with EXTERNAL)
- !!! In simple assignments of the form:
  variable = numeric-constant
  or constant-name = numeric-constant
  the constant assumes the accurace of the left hand side. Otherwise the accuracy is either default real (without D exponent) or double precision.
  This last rule gives a sufficient mechanism to guarantee required precision and obviates the need for the ugly EXPONENT_LETTER statement.
- Parameterised REAL and COMPLEX must not be EQUIVALENCEd

4. Proposal 3, Accuracy of intrinsic functions

Background

For numerical software it is killing not to know the accuracy of intrinsic functions.

Proposal

Provide a mechanism in Fortran to make sure that the accuracy of intrinsic functions (SIN, COS etc) is exactly the same as the accuracy of the argument.

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
Scientific Computing Division/Advanced Methods Section
P. O. Box 3000 • Boulder, Colorado • 80307
Telephone: (303) 497-1275 • FAX: 303-497-1137 TELEX:989764

(4)

August 29, 1988

110-JCA-4

Cathy Kachurik
CBEMA, X3 Secretariat
311 First street, N. W. Suite 500
Washington, DC 20001-2178

Dear Cathy;

X3J3 has instructed me to request another membership clarification from the SMC.

During the past several years, there has been a member (Leonard Moss) and his alternate (Sylvia Sund) from the Stanford Linear Accelerator Lab (SLAC). Last month, Sylvia Sund applied for membership as the representative for SHARE, the IBM User's Group.

The result of this action is that there are two members from the same organization, SLAC, even though one is representing the SHARE User's Group. Would you ask the SMC if Ms. Sund is eligible for membership privileges? In the meantime, Ms. Sund is accepted as a member with full voting privileges.

In recent months, there have been numerous membership applications; I appreciate your reviewing these questions for X3J3.

Regards,

Jeanne Adams, Advanced Methods Group
Chair, X3J3

110-JCA-5
p 1 of 4
(5)

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
Scientific Computing Division/Advanced Methods Section
P. O. Box 3000 • Boulder, Colorado • 80307
Telephone: (303) 497-1275 • FAX: 497-1137 Telex: 989764

September 6, 1988

Prof. John Rice
Dept. of Computer Science
Purdue University
West Lafayette, IN 47907

Dear John;

I welcome you to address X3J3 at the November meeting. Comments from persons and organizations concerned about Fortran are encouraged to make their views known in the current controversy.

The meeting will take place at the Royal Sonesta Hotel in Cambridge, November 13-18. Notice that we begin Sunday morning at 10 am. The host is Michael Berry of Thinking Machines. I can schedule your talk on any morning. I plan to mail a similar letter to Brian Ford. You may wish to speak on the same day or different days. I will be producing the final agenda two weeks before the meeting.

My email addrress is jeanne@scdpyr.ucar.edu. Do you have an email address, in case you need to communicate with me on arrangements?

I will place your correspondence in the pre-meeting distribution, which I will send to the distributor next week, before I leave for the Paris meeting of Working Group 5. I look forward to seeing you again at the meeting of X3J3 in November.

Regards,

Jeanne Adams, Advanced Methods Group
Chair, X3J3

cc. Michael Berry, Thinking Machines

IFIP

110 - JCA - 5
1 - 2 of 4
(5)

INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING

Date : September 1, 1988

Address reply to :

Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907
317-494-6003

Dr. Jeanne Adams
Chairman, X3J3
Scientific Computing Division
NCAR
P.O. Box 3000
Boulder, Colorado 80307

Dear Jeanne:

The enclosed letter to the members of X3J3 underscores the deep concern that the IFIP WG2.5 members have that a new Fortran standard be adopted soon. We believe that our concern is shared by the user community in general although most users are not as aware of the situation or issues involved as our members are. This letter is also being sent to the X3 committee along with a request that it also take steps to promote prompt adoption of a new standard.

Brian Ford and I wish to attend the next meeting of X3J3 to press for a resolution of the deadlock. We do not have a list of specific constructs or features that we advocate. The members of IFIP WG2.5 appreciate that there are substantial differences in technical evaluations on some points. However, we suspect that a deadlock such as this can also be partially due to other factors, e.g., unwillingness to abandon long held positions and "lose face", petty commercial advantages of a transient nature, or just plain stubbornness. Brian and I hope that our appeal as concerned outsiders will motivate X3J3 members to reevaluate their positions and move quickly toward a new Fortran standard.

For your information, Brian Ford is the head of NAG, Ltd. and a professor at Oxford University. NAG (Numerical Algorithms Group) has been producing numerical software libraries and related products since the early 1970's and is the leading European company in this field. I am a professor at Purdue University and head of the Computer Sciences Department. I have been active in most aspects of numerical software since the 1960's, for example, I founded the ACM Transactions on Mathematical Software in 1975 and am still Editor-in-Chief.

Please let me know soon if Brian and I will be addressing X3J3 at its Boston meeting so we can make travel plans. Thank you for your consideration of this request.

Sincerely,

John R. Rice
Vice-Chairman
IFIP WG2.5

JRR:pp

cc: Brian Ford
Lloyd Fosdick, Chairman of IFIP WG2.5
Mladen Vouk, Secretary of IFIP WG2.5
Richard Gibson, Chairman of X3

18

IFIP

110-JCA-5
p. 3 of 4
(5)

INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING

Date : September 1, 1988

Address reply to :

Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907
317-494-6003

Dr. Richard Gibson
AT&T
5A 211
Rt 202 & 206N
Bedminster, New Jersey 07921

Dear Dr. Gibson:

I enclose a letter to the members of X3J3 from the membership of IFIP Working Group 2.5 (Numerical Software). It expresses the deep concern that IFIP WG2.5 feels about the current deadlock on the X3J3 committee. The second enclosed letter to Jeanne Adams provides further information. The membership of IFIP WG2.5 feels strongly that the need for hard decisions should not become an excuse for indefinite delay.

I am writing you to request that you and the X3 committee, as a whole, take active steps to promote the prompt adoption of a new Fortran standard. The standard is far too important to be further delayed by personal and corporate stubbornness or by other secondary issues.

If there are further steps that are appropriate for our membership to take to pursue further this end, please advise me of them. Thank you for your consideration of this request.

Sincerely,

John R. Rice

John R. Rice
Vice-Chairman
IFIP WG2.5

JRR:pp

cc: Brian Ford
Lloyd Fosdick, Chairman of IFIP WG2.5
Mladen Vouk, Secretary of IFIP WG2.5
Jeanne Adams, Chairman of X3J3

19

IFIP

110-JCA-5
P 4 g 4
(5)

INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING

Date : September 1, 1988

Address reply to :

Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907
317-494-6003

Dear X3J3 Member:

The membership of IFIP WG2.5 (Numerical Software has just received reports of the deadlock in efforts to produce a new Fortran standard. We are alarmed and appalled. We believe that a failure of X3J3 to produce a standards proposal soon would be an abdication of its responsibilities to the scientific community. Our group represents many divergent interests and needs, yet we all join in strongly asking you to meet the needs of the scientific and engineering community promptly.

We realize that there are serious differences on some technical issues, that there are uncertainties in the relative merits of different approaches to some problems, and that the choices introduce substantial, but different, costs for users, software developers, and common system providers. Yet we believe these are all secondary issues and hard choices should not impede the production of the new Fortran standard.

The criteria for the new standard are simple: (1) Upward compatibility must be maintained, (2) Fortran 77 is increasingly outdated and must be enriched by the inclusion of a number of new constructs and features, and (3) it must form a coherent and useable whole. There are multitudes of potential standards that meet these criteria. We believe that the user's needs take priority over those of compiler writers, software developers, or hardware manufacturers.

Further excessive delay in this standard in unconscionable and unacceptable. We offer to send a delegation to your next meeting both to underscore the seriousness with which we view this deadlock and to offer more specific help in resolving the issues.

Sincerely,

John R. Rice

John R. Rice
Vice Chairman
IFIP WG2.5
Working Group on Numerical Software

JRR:pp

20

(6)

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
Scientific Computing Division/Advanced Methods Section
P. O. Box 3000 • Boulder, Colorado • 80307
Telephone: (303) 497-1275 • FAX: 497-1137 Telex: 989704

110 - JCA - 6

September 7, 1988

W. van Snyder
Mail Stop 301-490
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109

Dear Dr. van Snyder;

I will place your August 31, 1988 comments in the pre-meeting distribution for the November meeting, along with the letters you sent to Dr. Wagener in 1987. Your comments have already been distributed.

In the current controversy over the direction X3J3 should take, many of the detailed criticisms and suggestions have been set aside for later processing when the broader issues have been resolved.

It is helpful to hear from Fortran users, as we work toward the resolution of our difficulties.

Regards,

Jeanne Adams, Advanced Methods Group
Chair, X3J3

21

110-JCA-6 (6)

Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109

August 31, 1988

Dr. Jeanne Adams, Chair X3J3
Scientific Computing Division
National Center for Atmospheric Research
Box 3000
Boulder CO 80307

Dear Dr. Adams:

I have recently seen a summary of the positions of four factions within X3J3 regarding features that might be removed from or added to S8. Two of the features I have previously argued were redundant, RANGE and IDENTIFY/ALIAS, are apparently scheduled for deletion. But my arguments were that these features could be implemented more regularly by a single more powerful, more easily described mechanism. I argued that RANGE and IDENTIFY, and the associated intrinsic functions, could be subsumed into a single mechanism, the *accessor*. I haven't argued that they should simply be removed, since the facility they provide is useful.

I agree that pointers or some equivalent mechanism should be defined. I think pointer dereferencing and structure element selection can use the same syntax. I discussed this in my letter to Wagener of 27 September 1987, of which I sent you a copy. I enclose another copy.

I couldn't find really serious fault with the precision mechanism. But I believe the same mechanism should be used for all relevant types -- REALs, INTEGERs and CHARACTERs. I don't like the Japanese proposal for character *kinds*. I think the concatenation and automatic sectioning present in character variables should be extended to all types. See the arguments on page 2 of my public comment, *Critique of 8X*. I enclose another copy.

I find the sentiment to use I/O syntax for array constructors interesting. We (with Krogh) proposed exactly this overloading in response to Wilson's original array proposal. Those who advocate I/O syntax might want to dig up our old proposal. If it can't be found we can (maybe) provide one. The essential extension was to use the "//" operator to distinguish between concatenating lists to make one column, or concatenating columns to make arrays, etc.

I'm almost neutral on vector-valued subscripts. They're nice, but not if they add too much baggage.

Most compilers provide some kind of support to access the individual bits of an integer. The use of MIL-Std functions is common and a reasonable choice for standardization. The MIL-Std bit functions, together with something like PACKED LOGICAL, would completely subsume the BIT data type removed in the Halifax compromise. But I think most uses of bits are really bit fields, that could be described more clearly, and implemented more efficiently, by a bit-field descriptor in the defined-type area. I described this on page 4 of my formal critique of 8X, and in an addendum to my letter to Wagener, entitled *Packed Structures*.

I think the only possible time to add significant blanks is when the new source form is added. I find it interesting that you are in favor of making blanks significant (presumably only in the new source form), while the other factions favor removing the new source form.

Internal procedures are an extension provided by most vendors, and they should be standardized instead of prohibited. It is my understanding that the controversy surrounds the use of internal procedures as actual arguments. The reasons I have so far heard against allowing internal procedures to be actual arguments, or to be nested, are erroneous. I discussed this in my formal critique of 8X, on page 6.

I think most of the arguments against deprecation are specious. Go ahead and deprecate silly features, so long as a reasonably efficient substitute exists or is proposed. They may stay deprecated forever (rather than eventually disappearing), but that's O.K too. If the compilers diagnose them, they will eventually fall into disuse. It doesn't matter if that takes 10 years or 50.

INCLUDE is another common feature that should be standardized instead of prohibited.

Defined operators are potentially very useful, especially if an INLINE attribute can be attached to the defining subprogram. I don't think it's unreasonable to restrict them to the operators already defined. That is, allow the user to define a new action of an extant operator in certain contexts, but don't let him define operators using new symbols, or names of his choice in infix (or distfix) positions.

I'm neutral on array argument association, user elementals and multi-statement lines.

I like the new source form, but only if blanks (and ends of lines) are made significant in it.

I think structures are indispensable. I think they should have parameters. I think they should be extended to include packed structures, and to allow accessor subprograms (not their addresses) to be structure elements. I discussed these in the above-mentioned letter to Wagener.

I can do without module procedures, so long as one can still put interface blocks into modules. (But see the comments in my formal critique of 8X regarding the INLINE attribute for external subprograms). Modules should be kept.

The new syntax for DATA doesn't seem to add much baggage, and is much more flexible than the old. I'd vote to keep it.

Keyword and optional arguments would be useful, but I've figured out ways to get along without them, and still have roughly the same functionality.

Any reasonable definition of equivalence of structures (I assume this means type equivalence, not storage equivalence) is acceptable.

Keeping or discarding obsolescence is moot, since there are no obsolescent features.

(6)

I'm not sure what entity-oriented declarations are.

Interface blocks should be kept.

If MODULES are kept, and one believes MODULES are a viable substitute for COMMON, one shouldn't need to put structures in COMMON. Don't introduce new baggage that isn't necessary. If MODULES go, but structures stay, by all means allow structures to be put into COMMON.

Stream I/O and varying strings would be nice to have if they don't require too much baggage.

The summary didn't mention:

- EXIT should be allowed to apply to any structure, not just loops. A BLOCK structure, having no other purpose than as the target of an EXIT should be introduced. Absent this ability, try computing the predicate "X is not a member of the set S" without using GOTOs or extraneous logical variables, where S is represented by an array.

- The CASE statement should be extended to REAL ranges. This ISN'T the same thing as allowing REAL inductors on DO loops, or REAL subscripts. It's an efficient alternative to ELSE IF and ARITHMETIC IF that doesn't require re-evaluation of part of the predicate.

- ARRAY SECTION DESCRIPTOR should be a new type, with values that are triplets (as described for array section "constants").

- Dynamic allocation should be extended to structures, as part of a POINTER facility or equivalent.

- Needless restrictions should be removed. Ideally, every constraint should be formally justified or removed. Tradition isn't an adequate justification. Neither is compatibility, since removing a restriction can't invalidate an existing program that observed it.

Sincerely,

Van

W. Van Snyder
Mail Stop 301-490

110-JCA-6 (6)

Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109

August 4, 1987

Jerrold L.Wagener
AMOCO Production Company
Tulsa, OK 74102

Dear Mr. Wagener:

As a consequence of the size of X3J3/S8.104, I have only now noticed the CASE statement is not usable for REAL *case-expr*. This might be repaired by the following small changes:

On page 8-3, replace lines 24-26 by
{

R812	*case-expr*	is *scalar-numeric-expr* or *scalar-char-expr*

}
Replace lines 30-36 by
{

R814	*case-value-range*	is *case-value* [*rel* * [*rel case-value*]] or * *rel case-value*
R814.5	*rel*	is < or .LT. or <= or .LE.
R815	*case-value*	is *scalar-numeric-const-expr* or *scalar-char-const-expr*

}
OR
{

R814	*case-value-range*	is *case-value* [*rel-op* * [*rel-op case-value*]] or * *rel-op case-value*

Constraint: If *case-value-range* is of the form *case-value rel-op * rel-op case-value*, each instance of *rel-op* must be one of <, .LT., <= or .LE., or each instance of *rel-op* must be one of >, .GT., >= or .GE.

R815	*case-value*	is *scalar-numeric-const-expr* or *scalar-char-const-expr*

}
The constraint on R814 could instead be expressed by more syntax rules.

Allowing a LOGICAL *case-expr* provides no functionality or performance benefit as compared to an IF statement, so I have not included that possibility.

Replace from line 39 on page 8-3 to line 11 on page 8-4 by
{
8.1.3.2 Execution of a CASE Construct. The execution of the SELECT CASE statement causes the case expression to be evaluated. The resulting value is called the case discriminant and must match exactly one of the selectors of one of the CASE statements of the construct. If the case selector is a case value range list, the case discriminant matches the selector if it matches any of the

case value ranges in the list. A case discriminant with value *c* is defined to match a case value range in the following circumstances:

(1) If the case value range contains a single value *cv*, *c* matches the case value range if and only if *c* .EQ. *cv*.

(2) If the case value range is of the form $cv_1\ rel_1$ * $rel_2\ cv_2$, *c* matches the case value range if and only if $cv_1\ rel_1\ c$ .AND. $c\ rel_2\ cv_2$ is true.

(3) If the case value range is of the form $cv_1\ rel$ *, *c* matches the case value range if and only if $cv_1\ rel\ c$ is true.

(4) If the case value range is of the form * $rel\ cv_2$, *c* matches the case value range if and only if $c\ rel\ cv_2$ is true.

(5) If *c* matches no other case selector and a CASE DEFAULT selector is present, *c* matches the CASE DEFAULT selector.

(6) If *c* matches no other case selector and no CASE DEFAULT selector is present, an error is signalled and program execution terminates.

}

On page 8-4, line 16, replace "index" by "discriminant".

On page 8-4 replace lines 20-30 by

{

8.1.3.3 Examples of CASE Constructs. INTEGER and REAL signum functions:

```
INTEGER FUNCTION SIGNUM (N)     INTEGER FUNCTION SIGNUM (X)
INTEGER N                       REAL X
SELECT CASE (N)                 SELECT CASE (X)
CASE (* < 0)                    CASE (* < 0.0)
        SIGNUM = -1                     SIGNUM = -1
CASE (0)                        CASE (* == 0.0)
        SIGNUM = 0                      SIGNUM = 0
CASE (0 < *)                    CASE (* > 0.0)
        SIGNUM = 1                      SIGNUM = 1
END CASE                        END CASE
END FUNCTION SIGNUM             END FUNCTION SIGNUM
```

}

Delete lines 1-20 on page 8-5.

I don't know if there are other examples using the CASE construct and the colon notation for a range. If so, they would need to be changed.

This change would allow the CASE construct completely to subsume the functionality of the arithmetic IF with no loss of efficiency.

Sincerely,

W. Van Snyder
Mail Stop 301-490

110-JCA-6 (6)

Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109

September 24, 1987

Jerrold L.Wagener
AMOCO Production Company
4502 East 41st Street
P. O. Box 3385
Tulsa, OK 74102

Dear Dr. Wagener:

The issue on which my concern is and has been focused is data abstraction. When your letter arrived, I had thought it impossible fully to support data abstraction with any notation other than component(structure), and this is so given only the weak mechanisms for declaration of subprograms proposed for Fortran 8X. But I discussed the issue with my colleague, Fred Krogh, and we found the several notations (even S%C) to be equally amenable to data abstraction, given simple extensions of declarative mechanisms. None of the notations would be a serious mistake.

I have no arguments other than those Page sent you in 87100ART0073 (he had more than I) to distinguish between structure(component) and component(structure) notations. The most surprising result of the discussion with Krogh was that S%C notation (perhaps with a prettier character than %) allows the clearest expression of complicated references, but doesn't inhibit abstraction. In any case, all the notations must be carefully defined to avoid inconsistencies, especially between references to statically and dynamically allocated objects.

I had thought that since the desire to transform between structure components and procedures, and between array elements and procedures, required the common ability to represent a concept by a procedure, the syntaxes for array element reference, component reference, and procedure reference had to be the same. I had also thought it impossible transparently to change the representation of an abstraction between an array of structures and a collection of dynamically allocated structures using any notation. But solutions of these problems, described below, convinced me of the possibility of abstraction using any notation.

Transformations of the second kind mentioned above arise, for example, when one needs to change between an array of objects of type T and a linked list of objects of type T. But it seems that if one uses an array one is required always to mention its name in references, and if one uses pointers to anonymous objects there is no name to mention. Whatever referential syntax is chosen, data abstraction seems impossible. But by declaring J to be a pointer to T, and declaring that S *casts* its argument in the role of pointer to T, one could change from a representation requiring S(J)%C to one that would require P%C, and still write S(J)%C[1]. Similarly one could change from a representation requiring P%C to one that would require S(J)%C by changing the declaration of P

[1]The declarations of S and J may seem redundant, but they allow the referential syntax not to change when the representations change.

from "pointer to T" to "subscript of S" and still write P%C. Without these declarations, one would still need manually to change between C(S(J)) and C(P), even when using functional notation. That is, functional notation has no advantage in this case over S%C for purposes of data abstraction.

Type casting has the added benefit of allowing a natural notation for references using pointers to unions of several types. Suppose one has two types, say T_1 and T_2, and two objects, say S_1 and S_2, that are arrays of objects of types T_1 and T_2 respectively, and one wants to use the same subscript to access the arrays. Then $S_1(J)\%C$ and $S_2(J)\%C$ have unambiguous meaning. Suppose that one must change the representations of S_1 and S_2 to collections of allocated objects. The first step is to define S_1 and S_2 to be casts of the appropriate types. But beyond that, one cannot simply change the declaration of J to "pointer to T_1," because in $S_2(J)\%C$, $S_2(J)$ would have incorrect type correspondence. Similarly one cannot simply change the declaration of J to "pointer to T_2." One must change the declaration of J to "pointer to union of T_1 and T_2," and the presence of S_1 or S_2 selects the type to which J points.

The property of Fortran 8X that ultimately prevents reasonable data abstraction is not referential notation, but that *one cannot simply change the declaration of the representation or a concept from a structure component or array element to a subprogram: Fortran doesn't allow subprogram invocations to appear in value-receiving contexts.*

To repair this defect, I believe it is important to introduce "accessor" subprograms into Fortran. That is, subprograms that can be invoked in both value-providing and value-receiving contexts (including use as actual arguments and in I/O lists). If this is done, it is important ***NOT*** to implement "left-hand functions" in the sense usually described in programming language textbooks: subprograms that are invoked before the value is calculated, and produce the address at which the calculated value is to be stored. It is important that the value be calculated before the value-receiver is invoked, and that the value be passed to the value-receiver as an actual argument. Otherwise the value-receiver is unable to make decisions about the position of the value in a data structure. Imagine trying to do something as simple as keeping a list in order by using left-hand functions that calculate the address of an object before the value is known!

Uniform referential syntax seems essential to allow the representation of concepts to be changed between structure components and accessors. But we discovered the syntax issue is irrelevant in this case also if it is possible to declare that a member of a structure might be an accessor. Then the notations C(S), S(C) or S%C equally well indicate component selection or accessor invocation, depending *only* on the declarations of S and C.

In Fortran 8X as it is today, structure declarations can be viewed as a small subset of module declarations -- they both bundle related objects together. Allowing components to be procedures, and extending component (storage or procedure) declarations to allow accessibility attributes, makes structure declarations a more substantial subset of Module declarations. It would be more thrifty to remove structure declarations, and allow variables to be instances of modules.

If one is to allow several instances of a module, one must define "instance of a module." Each instantiation of a module allocates storage space for all of

the (public and private) variables declared directly in the module, or in directly contained modules, but not for global objects such as common blocks or procedures, nor for objects declared inside procedures -- the latter would be allocated only once (per recursive invocation). If modules could be dynamically instantiated by an allocator, this would provide some of the functionality of object oriented languages (inheritance and dynamic binding would still not be provided).

I am no longer convinced that uniform syntax is best. The uniformity conferred by functional notation comes at the price of unnecessarily strenuous gymnastics to refer to such complicated things as arrays of pointers to arrays of structures having components that are arrays. S%C notation allows more readable and writable references, and allows some references not possible in function notation. Since uniform notation is not necessary for data abstraction, I prefer notation(s) that are easiest to read and write.

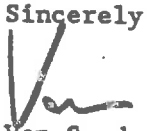
The % is ugly (and misleading -- in some fonts it looks a lot like +). But I no longer believe functional notation is the answer. I tend toward something like S^C, which I find more aesthetic than S%C, but this is an issue of such narrow content that it probably won't convince the committee to change anything. Either S%C or S^C has reasonable generalizations allowing pointers to structures, pointers to anonymous arrays (that are not structure elements), arrays of pointers to arrays of structures having components that are arrays of structures ... And to accessor subprograms, which for me is the real issue.

Although I submitted a proposal to amend S8.99 to incorporate accessor subprograms, I never had substantial hope they would be implemented into Fortran 8X. But since I have been shown how data abstraction can be accomplished without uniform syntax, I no longer believe nonuniform syntax is an impediment to future extensions that support more powerful data abstraction.

I know I've reversed my position. I hope my agitation for uniform syntax wasn't the only reason you and Walt and Rex took it up again.

If you would like more comments please feel free to call or write. I can be reached at 818/354-6271.

Sincerely,



W. Van Snyder
Mail Stop 301-490

P.S. I must repeat that adequate power to express abstraction would allow removing much of the specialized baggage of Fortran 8X, e.g. RANGE, ALIAS, ...

Enclosures

cc: Jeanne Adams, Tom Lahey

Let STRU be a derived type having a component C, S a statically declared object of type STRU, P a pointer to objects of type STRU, I, J and K subscripts, and (X) a (possibly absent) list of arguments. In the table below we abbreviate Scalar to "Sca", Array to "Arr" and Accessor to "Acc". A question mark means we couldn't find any reasonable notation.

Pointer	Component Structure		Notations: S^C style	C(S) style	S(C) style
No	Sca	Sca	S^C, S'C, S.C	C(S)	S(C)
No	Sca	Arr	S^C(J)	C(S)(J)	S(C(J))
No	Sca	All	S or S^ or S^STRU	STRU(S) or S	S(STRU) or S
No	Sca	Acc	S^C(X)	C(S)(X)	S(C(X))
No	Arr	Sca	S(K)^C	C(S(K))	S(K)(C)
No	Arr	Arr	S(K)^C(J)	C(S(K))(J)	S(K)(C(J))
No	Arr	All	S(K) or S(K)^ or S(K)^STRU	STRU(S(K)) or S(K)	S(K)(STRU) or S(K)
No	Arr	Acc	S(K)^C(X)	C(S(K))(X)	S(K)(C(X))
Sca	Sca	Sca	P^C	C(P)	P(C)
Sca	Sca	Arr	P^C(J)	C(P)(J)	P(C(J))
Sca	Sca	All	P^ or P^STRU	STRU(P)	P(STRU)
Sca	Sca	Acc	P^C(X)	C(P)(X)	P(C)(X)
Sca	Arr	Sca	P^(K)^C	C(P(K))	P(K)(C)
Sca	Arr	Arr	P^(K)^C(J)	C(P(K))(J)	P(K)(C(J))
Sca	Arr	All	P^(K) or P^(K)^ or P^(K)^STRU	STRU(P(K)) or	P(K)(STRU) or
Sca	Arr	Acc	P^(K)^C(X)	C(P(K))(X)	P(K)(C(X))
Sca	No	Arr	P^(K)	?	?
Sca	No	Sca	P^	?	?
Arr	Sca	Sca	P(I)^C	C(P(I))	P(I)(C)
Arr	Sca	Arr	P(I)^C(J)	C(P(I))(J)	P(I)(C(J))
Arr	Sca	All	P(I)^ or P(I)^STRU	STRU(P(I)) or P(I)	P(I)(STRU) or P(I)
Arr	Sca	Acc	P(I)^C(X)	C(P(I))(X)	P(I)(C(X))
Arr	Arr	Sca	P(I)^(K)^C	C(P(I)(K))	P(I)(K)(C)
Arr	Arr	Arr	P(I)^(K)^C(J)	C(P(I)(K))(J)	P(I)(K)(C(J))
Arr	Arr	All	P(I)^(K) or P(I)^(K)^ or P(I)^(K)^STRU	STRU(P(I)(K)) or P(I)(K)	P(I)(K)(STRU) or P(I)(K)
Arr	Arr	Acc	P(I)^(K)^C(X)	C(P(I)(K))(X)	P(I)(K)(C(X))
Arr	No	Arr	P(I)^(K)	?	?
Arr	No	Sca	P(I)^	?	?

Several of the functional notations are the same in different circumstances, but this is just another kind of overloading that may be put to beneficial use when transformations of representation are necessary.

One might prefer to use commas instead of parentheses in some circumstances, such as reference to an array element component of an array element structure. But one must decide on the order of "arguments." The choices reflect the conflict between referencing from the general to the specific, and "column major" storage order. Commas might be too confusing to write or read reliably.

The above notations allow consistent reference to statically and dynamically allocated objects, except for references to whole structures or unstructured objects. When P is a pointer it seems more reasonable to interpret P alone to

be its value, and use some other syntax such as P^ or STRUCT(P) to denote the value of the object P references. But symmetry demands one write S^ to refer to the value of a statically allocated object, in which case S alone would naturally be its address. This is incompatible with Fortran 77, but it allows P=S to assign the address of a statically allocated object to a pointer[2], an operation that regularizes many algorithms.

Since we ordinarily desire S alone to stand for the value of a statically allocated object, symmetry demands that P alone stand for the thing P references, and one must use something like LOC(P) to access the value of P. By symmetry, LOC(S) would then be the address of a statically declared object S, but it would be a function instead of an accessor. One could then still write LOC(P)=LOC(S). But then if one changes P from a pointer to an integer used as a subscript, LOC(P) becomes its address instead of its value, and LOC(P)=E is prohibited. The escape from this inconsistency due to the intrinsic difference in levels of indirection between pointers and objects that are not pointers is by strengthening the declarative power: declare an object to be a "locator" (subscript, pointer or ?). Then to use the value of a locator X in a non-locating context, one must write LOC(X) no matter whether X is a pointer or subscript. Declarative solutions to these problems of inconsistent referential syntax, and the corresponding references, might be

Declaration	Object value	Address or Pointer value
TYPE (STRUCT) S	S	LOC(S)
TYPE (STRUCT) (POINTER,PARAMETER) S	S^ or STRUCT(S)	S
TYPE (STRUCT) (POINTER) P	P	LOC(P)
TYPE (STRUCT) (POINTER,VALUE) P	P^ or STRUCT(P)	P
INTEGER (INDEX(S)) P	P	LOC(P)
TYPE (STRUCT) (CAST) S	S(P)	P

in which the first two statically allocate storage. The fifth and sixth allow the transformation of reference of an object between an array of structures and a collection of dynamically allocated structures. The fifth declares P to be a subscript of S, and therefore P alone stands for S(P). The sixth has the effect of casting any pointer argument P of S in the role of "pointer to STRUCT," no matter what its declaration. It must also be possible to cast a pointer using a type name.

Continuing this argument seems to lead to trouble. If one is allowed to access an object using S or S^ or P or P^ depending on the declaration, by extension one should expect to access S C or S^C etc. depending on the declaration of S. But one needs some punctuation to separate the object from the container when the container is a scalar. A further observation is that requiring ^ prohibits a (probably very rare) transformation of representation between P^(K)^C and P(K)^C. To allow this transformation, the standard should make the circumflex (or %) optional except where needed to provide a boundary between two names. Both of the above could then be written P(K)C. This is almost the same as P(K)(C), the S(C) style of notation in both of these cases. But then, does P mean P or P^ or P^^ or ...?

---

[2]By this interpretation S=P would mean "change the address of S to the value of P, which is clearly nonsense.

ACCESSOR SUBPROGRAMS

Accessor subprograms are necessary for data abstraction. But there are several related issues that must be examined. To assist the discussion, we first present a concrete example, and then discuss the variations resulting from different choices of declarative issues.

Suppose one needs some stacks. To preserve the possibility that we might want to change the representation between an array and a linked list, we choose to represent the stack by a structure, and provide its facilities by components or accessors, as appropriate to the concrete representation.

```
TYPE STACK_ELEMENT
   ...
END TYPE STACK_ELEMENT

TYPE STACK (SIZE) (REF PUSH_POP)
     ! The attribute (REF PUSH_POP) means references to objects of type
     ! STACK are to consist of invocation of the PUSH_POP accessor below.
     INTEGER, DATA, PRIVATE :: THIS_SIZE = SIZE
     INTEGER, DATA, LIMITED :: QUANTITY = 0 ! Number of objects in stack
     LOGICAL, DATA, LIMITED :: NOTEMPTY = .FALSE.
     LOGICAL, DATA, LIMITED :: NOTFULL = .TRUE.
     TYPE (STACK_ELEMENT) PRIVATE :: E(SIZE)
     ! In accessors declared within STACK, STACK denotes the variable of
     ! type STACK on which the accessor is to operate.  For example, when
     ! S^TOP is invoked, S is bound to STACK.

     TYPE (STACK_ELEMENT) ACCESSOR PUSH_POP ()
     WHEN FETCH ! Pop operation
          IF (STACK^NOTEMPTY) THEN
               PUSH_POP = STACK^E(STACK^QUANTITY)
               STACK^QUANTITY = STACK^QUANTITY - 1
               STACK^NOTEMPTY = STACK^QUANTITY <> 0
               STACK^NOTFULL = .TRUE.
          ELSE
               CALL UNDERFLOW(STACK) ! quits
          END IF
          RETURN
     WHEN STORE ! Push operation
          IF (STACK^NOTFULL) THEN
               STACK^QUANTITY = STACK^QUANTITY + 1
               STACK^E(STACK^QUANTITY) = PUSH_POP
               STACK^NOTFULL = STACK^QUANTITY <> STACK^THIS_SIZE
               STACK^NOTEMPTY = .TRUE.
          ELSE
               CALL OVERFLOW(STACK) ! quits
          END IF
          RETURN
     END ACCESSOR STACK

     TYPE (STACK_ELEMENT) ACCESSOR TOP ()
     WHEN FETCH
          IF (STACK^NOTEMPTY) THEN
               TOP = STACK^E(STACK^QUANTITY)
          ELSE
               CALL UNDERFLOW (STACK) ! quits
```

```
            END IF
            RETURN
        WHEN STORE
            IF (STACK^NOTEMPTY) THEN
                STACK^E(STACK^QUANTITY) = TOP
            ELSE
                CALL UNDERFLOW (STACK) ! quits
            END IF
            RETURN
        END ACCESSOR TOP

        SUBROUTINE MAKE_EMPTY (S)
            TYPE (STACK(*)) S
            S^QUANTITY = 0
            S^NOTEMPTY = .FALSE.
            S^NOTFULL = .TRUE.
            RETURN
        END SUBROUTINE MAKE_EMPTY

        SUBROUTINE UNDERFLOW (S)
            TYPE (STACK(*)) S
            ...
        SUBROUTINE OVERFLOW (S)
            ...
    END TYPE STACK

    TYPE (STACK(100)) S, T    ! Declare two stacks of 100 elements
    TYPE (STACK_ELEMENT) X, Y

    CALL MAKE_EMPTY (S)
    CALL MAKE_EMPTY (T)
    ...
!   The next three statements invoke the accessor PUSH_POP.
    IF (S^NOTFULL) S = X      ! Push X on stack S
    IF (T^NOTEMPTY) Y = T     ! Pop from stack T to Y
    S = T     ! Pop from stack T, push onto stack S
!   The next three statements invoke the accessor TOP
    S^TOP = X ! Replace the top element of S by X
    Y = T^TOP ! Replace Y by the top element of T
    S^TOP = T^TOP  ! Copy the top element of T to the top element of S
!   The next three statements are illegal because their left hand sides are
!   limited to references, not assignments
    S^NOTEMPTY = .FALSE.
    S^NOTFULL = .TRUE.
    S^QUANTITY = 0
```

The first issue is whether accessors may be declared as independent subprograms, or only as members of structured data types. The example solved the intended problem using only accessors contained in STACK. But independent accessors have utility, and impose no extra burden on implementors. Our example illustrated a third kind of accessor, PUSH_POP: it is not independent, but it is invoked by the name of an object of the containing type, not by its own name. The declaration that PUSH_POP is invoked when objects of type STACK are referenced was denoted by a distinguished syntax in the STACK declaration, viz. (REF PUSH_POP). It might instead have been accomplished by giving PUSH_POP the

same name as the type, STACK, or by using a distinguished syntax in the declaration of PUSH_POP, e.g. (REF STACK).

The second issue concerns access to S when S^C is invoked. There are at least four ways this may be done. Our example used reference to the containing type, that is, references to STACK from within PUSH_POP and TOP. A second method is to declare an instance argument, perhaps with a distinguishing syntax. A third method is to provide an intrinsic accessor, say SELF(), that accesses S. Fourth, one might use a distinguished syntax, say *, to denote SELF(). Using the second method, the declaration of PUSH_POP might become

```
TYPE (STACK_ELEMENT) (USING Z) ACCESSOR TOP ()
```

where USING Z means Z is bound to S when S^TOP is invoked. Using the other methods, the declaration would be the same. If the second, third or fourth methods were used, references within PUSH_POP to STACK would change to Z, SELF() and *, respectively.

The third issue concerns access to the value provided to the value-receiving branch of the accessor. In the example, we access the received value using the name of the accessor. This is symmetric to the way functions specify return values. A second method might be to declare a received-value argument, perhaps with a distinguished syntax. A third might be to provide an intrinsic function, say RECEIVE(). Fourth, one might use a distinguished syntax, say *, to access the received value. To declare a value-receiving argument, one might extend the declaration of PUSH_POP to

```
TYPE (STACK_ELEMENT) (RECEIVE R) ACCESSOR PUSH_POP ()
```

The third and fourth methods would not require change to the accessor declaration. In our example, references to PUSH_POP in the WHEN STORE branch of PUSH_POP would be replaced by R, RECEIVE() and *, respectively. Using the same distinguished syntax, say *, to refer both to the object on which the accessor is to operate and the received value would be ambiguous if the received value is a structure having a component of the same name as the containing type.

The fourth issue concerns whether a component of a structure is invisible, visible only for reading, visible only for writing, or visible for both. The *private* and *public* accessibility attributes provide the first and last, respectively. A *limited* accessibility attribute would allow access only for reading a storage component, without resort to the subterfuge of defining a private component and a function to access it; an accessor with only a FETCH branch is equivalent to a function. It probably doesn't make sense to declare a write-only storage component, but an accessor with only a STORE branch makes sense. For example, we might have provided a read-only POP accessor and a write-only PUSH accessor in STACK, and allowed S=T to denote copying an entire stack.

The fifth issue concerns the definition of whole-structure assignment of objects of a structured type when some of the components are accessors. Since the entire state of the abstraction represented by the whole structure should be represented by the stored values, it should be enough simply to copy them. One might think it necessary to copy all storage components, and copy from the fetch entry to the store entry of each accessor for which both are defined. But this might put the internal data structures into an inconsistent state, and if an accessor has arguments other than the invoking context, there is no way to provide their values. If copying the storage components is not the correct action, the programmer can provide an assignment subroutine.

Sixth, since references to members of a type from within accessors that are members of the type will be common, it may be desirable to allow an abbreviation. For example, in PUSH_POP it would have been more terse to reference ^E(^QUANTITY) than to reference STACK^E(STACK^QUANTITY). The ^ punctuation is necessary to allow one to reference an object outside the type that has the same name as an object inside the type, by naming it without punctuation[3].

Seventh, when functions or subroutines are members of a type[4] they could be invoked (to operate on an object of the type) using either structural or functional notation. That is, if F is a function that is a member of the type of S, one might write either F(S) or S^F. We prefer functional notation because it preserves the connotation (not enforced by the language) that functions have no side effects (while accessors might), and the tradition that functions are never assigned a value.

Even allowed accessors, complete abstraction would not be possible in Fortran 8X. Consider a program that uses arrays. If the problems it is to solve become large, the memory capacity of the machines on which it is to run may be insufficient. We might choose to simulate virtual memory to solve the problem. As we did in the STACK example above, we might define a type VIRTUAL_ARRAY, and define an accessor that is invoked when objects of type VIRTUAL_ARRAY are referenced. For example:

```
TYPE VIRTUAL_ARRAY (REF ELEMENT, LIMITED)
     ! The LIMITED attribute means whole-structure access is not defined.
     REAL (USING Z) ACCESSOR ELEMENT(I,J)
     WHEN FETCH
     !    Make sure the I,J element of Z is in memory, then return the
     !    I,J element.
     WHEN STORE
     !    Make sure the I,J element of Z is in memory, then store into
     !    the I,J element.
     END ACCESSOR ELEMENT
...
END TYPE VIRTUAL_ARRAY
```

But if X had originally been an array, one could have written X(I:J,K) to reference a section of a column. Since Fortran 8X has no objects of type ARRAY-SECTION-DESCRIPTOR, one could not simply replace the type of X by a structured type that provides accessors because the accessors couldn't receive arguments such as I:J.

Finally, when subprograms are small the cost to call them may well be more expensive than the body. Since subprograms that are members of types will usually be smaller than independent subprograms, this source of inefficiency will only be exacerbated. Awareness of this expense influences programmers not to use subprograms as abstraction tools. If subprograms were allowed an INLINE attribute, the use of small subprograms would be no less efficient than explicit inline programming.

---

[3]One might also allow this by a declaration such as IMPORT X, REF Y to indicate an object X outside the type is to be visible inside the type as Y.

[4]The reason one should allow functions and subroutines to be members of a type is to allow them access to private components.

Many applications of bit data are related to packed structures. The difficulty of definition of bit data type could be avoided, and much of the functionality provided, by allowing a restrictive definition of packed data: an INTEGER datum might occupy less than an entire storage unit. There should be at least two mechanisms to declare this. The first is to declare that a structure, or part of a structure, is PACKED, and to declare each of the components to be a subrange of the INTEGER type[5]. For example, a symbol table object in a compiler might be declared

```
TYPE SYMBOL_TABLE
     PACKED
          CLASS(0:31)     ! Kind of symbol table object
          REF(0:1)        ! 1 means object referenced
          DEF(0:1)        ! 1 means object assigned a value
          TYPE(0:15)      ! Declared type
          FIELDS(0:15) (0:4)  ! An array indexed by (0:4)
          ...
     END PACKED
     ...
END TYPE SYMBOL_TABLE
```

Intrinsic functions are necessary to provide the minimum and maximum values of a component. MIN and MAX would be reasonable (although symmetry with intrinsics proposed for Fortran 8X might demand TINY and HUGE).

The compiler is free to arrange the objects to occupy as little storage as possible, or not to pack them at all (if the vendor is too cheap or lazy to implement packing). One hopes the compiler uses a concrete representation of the value of a component C in the range 0..MAX(C)-MIN(C), even if MIN(C) is not zero.

This mechanism is independent of the size of a storage unit, or the radix of integer representation.

The second declaration specifies exactly which bits of which storage unit are occupied by each component. We start with an example:

```
TYPE SYMBOL_TABLE () (PACKED)
     CLASS(0:31)     (0,*:26)
     REF(0:1)        (0,25:25)
     DEF(0:1)        (0,24:24)
     TYPE(0:15)      (0,23:20)
     FIELDS(0:15)    (0,3:0)    (0:4) ! An array indexed by (0:4)
     ...
END TYPE SYMBOL TABLE
```

The first parameter list provides the range of the component, the second specifies the storage allocation, and the optional third allows a component to be an array. The first parameter of the storage allocation declares its word posi-

---

[5]If enumerated types were implemented, the components could be allowed to be either a subrange of INTEGER or any other enumerated type.

position in the structure, and the second the bits it occupies[6]. The second parameter is optional; if absent, the component occupies all of the specified word. Allowing storage allocation specification is clearly an opportunity to get in trouble by assigning overlapping fields, but that is sometimes what is desired. Unsafe as it might be, this capability is needed because in some applications, for example in telemetry processing, the organization of components into words might be dictated by external equipment.

It is necessary in some applications to declare that objects of a type occupy a certain array. To support this the TYPE declaration header above might be expanded to "TYPE SYMBOL_TABLE () (PACKED, IN(ST))" to indicate that objects of type SYMBOL_TABLE occupy the array ST. Then a reference SYMBOL_TABLE(J)^DEF means that a template described by the type declaration is to be applied to ST beginning at ST(J). Since several structures might occupy ST, it is necessary to *cast* J into the role of "locator" for SYMBOL_TABLE. This is different from declaring an array of SYMBOL_TABLE elements, because several different types, perhaps of different sizes, might be declared to occupy the same array. This is also an opportunity for a programmer to create erroneous code, but again is sometimes necessary. For example, in telemetry processing one might read a sequence of integers into an array, and then interpret them to be a sequence of packed records, not necessarily all the same size.

We also allow components to be arrays. For components that are parts of words, array elements occupy the declared field and adjacent higher order fields of the same size. That is, SYMBOL_TABLE(J)^FIELDS(0) accesses bits 3:0 of ST(J), SYMBOL_TABLE(J)^FIELDS(1) accesses bits 7:4 of ST(J), etc. If a component is a full word, then subscripting has the usual interpretation.

Additional intrinsic functions are necessary to provide the number of words occupied by a type, the smallest word index of any component in a type, the word index of a component, and the high and low bit indices of a component.

An intrinsic type INDIRECT(IN(array)), where the IN clause is optional, allows field selection to be specified by data. INDIRECT is a packed type containing components denoting a word index, high bit index, low bit index and range. Let these fields be W, H, L and R respectively. Suppose X is a variable of type INDIRECT(IN(ST)). Then a reference of the form J^X denotes reference to the field dynamically represented by X. That is, Y=J^X means "Y = (bits X^H:X^L of ST(J+X^W)) + X^R^MIN," while J^X=Y means "(bits X^H:X^L of ST(J+X^W)) = Y-X^R^MIN." If the type of X is not IN an array, then J must be a pointer, and the reference denotes bits X^H:X^L of the word X^W words from the one denoted by J (J might not be a word-granularity address). We also allow X to be an array, and allow indirect references to be interpreted as selection of an element of a component that is an array. That is, when X is a scalar, J^X(K) denotes bits K*(H-L+1)+H:K*(H-L+1)+L of ST(J+W), provided X denotes a component that occupies part of a word. To provide initial data for objects of type INDIRECT we allow data statements of the form

---

[6]The low order bit is bit zero, and the high order bit is bit *. Using * to denote the high order bit might allow the compiler to use a more efficient access if the word size is larger than necessary to contain the field (for example when the target machine has no bit field instructions, DIVIDE and MOD would be necessary for unpacking. If the compiler knows the field extends to the left end of the word, it can unpack the field with a DIVIDE alone.)

DATA X /SYMBOL_TABLE^CLASS/

or

DATA X /SYMBOL_TABLE^CLASS(constant_expression)/ ! field array component

in which X and the types in the data part (viz. SYMBOL_TABLE) must be IN the same array, or in no array. We also provide an intrinsic generic function, FIELD, of type INDIRECT, that provides the entire description of a component, so that assignments such as

X=FIELD(SYMBOL_TABLE^CLASS)

or

X=FIELD(SYMBOL_TABLE^CLASS(K))

are possible. The type of the value of FIELD is INDIRECT, and IN the same array (if any), as the argument. The type of FIELD must be compatible with its context. In the present example, the types of X and FIELD() must be the same. No operations, other than assignment and indirect field selection, are defined on objects of type INDIRECT.

(6)

Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109

September 25, 1987

Jerrold L.Wagener
AMOCO Production Company
4502 East 41st Street
P. O. Box 3385
Tulsa, OK 74102

Dear Dr. Wagener:

After finishing yesterday's letter I thought of an example that might help you choose between C(S) and S(C) notations, that Rex hadn't put in his memo. As I mention in my other letter, I've changed my views, and think S%C notation (with a prettier character than %) may be best when the whole picture is considered.

Suppose one has a program that uses complex numbers, and discovers that the most frequent operations are multiplication and ABS. Because complex multiplication and examination of the modulus are much more efficient using polar representation, we wish to change the representation. Thus we could define

```
TYPE P_COMPLEX
     REAL, LIMITED :: ABS, PHASE ! Read-only components
     REAL FUNCTION REAL(Z)
          TYPE (P_COMPLEX) Z
          REAL = ABS(Z)*COS(PHASE(Z))
          RETURN
     END FUNCTION REAL
     REAL FUNCTION AIMAG(Z)
     ! Similar to REAL
     END FUNCTION AIMAG
     TYPE (P_COMPLEX) FUNCTION CMPLX(R,I)
          REAL R,I
          ! ABS and PHASE are writable here because CMPLX is
          ! defined inside P_COMPLEX.
          ABS(CMPLX) = SQRT(R*R+I*I)
          PHASE(CMPLX) = ATAN2(R,I)
          RETURN
     END FUNCTION CMPLX
     TYPE (P_COMPLEX) FUNCTION MULT(Z1,Z2) OPERATOR "*"
     ! etc.
END TYPE P_COMPLEX
```

If one then changed the declaration of a complex variable, say T, from COMPLEX T to TYPE (P_COMPLEX) T, and S(C) notation were used, one would need to change ABS(T) to T(ABS) everywhere. If C(S) notation were used, no other changes would be needed.

On the other hand, if an INLINE attribute of functions were allowed, one could change the declaration to

(6)

```
TYPE P_COMPLEX
     REAL MODULUS, PHASE
     REAL (INLINE) FUNCTION ABS(Z)
          TYPE (P_COMPLEX) Z
          ABS=MODULUS of Z ! choose your favorite notation
     END FUNCTION ABS
     ! The rest is about the same.
END TYPE P_COMPLEX
```

ABS(T) would still be ABS(T) in S(C) notation, and efficiency wouldn't suffer (given a competent optimizer). It's just a little more work for the programmer and the optimizer to do a good job, but data abstraction is still possible.

Another idea I didn't put into my other letter is that the standard might allow two referential notations, that is both S%C and C(S). It would be redundant to allow S%C and S(C), since the primary attraction of S(C) over C(S) is that it selects from the general to the specific (depending on your point of view), but so does S%C.

Sincerely,

W. Van Snyder
Mail Stop 301-490

(7)

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
Scientific Computing Division/Advanced Methods Section
P. O. Box 3000 • Boulder, Colorado • 80307
Telephone: (303) 497-1275 • FAX: 497-1137 Telex: 989764

110-JCA-7
p.1 of 2

September 7, 1988

Barry Vickers
Martin Marietta
PO Box 2003
Oak Ridge, TN 37831-7001

Dear Mr. Vickers;

I have notifed the vice-chair, Jerrold Wagener, of Alan Hirsch's assignment as liaison to X3J3 from X3H2.

The current liaison to X3H2 from X3J3 is Miles Ellis, whom you reject because of his European address. I wish to take this matter up at the next meeting of X3J3 in November. There may be some way that the mailings can be handled within the US.

Regards,

Jeanne Adams

Jeanne Adams, Advanced Methods Group
Chair, X3J3

110-JCA-7 (7)
P. 2 of 2

Accredited Standards Committee
X3, INFORMATION PROCESSING SYSTEMS

Doc. No.: X3H2-88-297

Date: 10 August 1988
Reply to: Barry Vickers
Martin Marietta
P.O. Box 2003
Oak Ridge, TN
37831-7001
(615) 574-7657

Ms. Jeanne C. Adams, X3J3 Chair
National Center For Atmospheric Research
Scientific Computing Division
P.O. Box 3000
Boulder, CO 80307

Subject: X3H2 and X3J3 Liaison

Dear Ms. Adams:

Procedures for the X3H2 Technical Committee on Database require coordination of activities with related standards bodies like X3J3, FORTRAN. Periodically X3H2 verifies the designation of all liaison representatives.

The current X3H2 liaison to X3J3 is: Alan R. Hirsch
AMOCO Corporation
MC 1008
P.O. Box 87703
Chicago, IL 60680-0703
(312) 856-7041

Our records reflect that X3J3 does not currently have a coordinating liaison to X3H2. You recently submitted the name of an X3J3 member to serve as liaison to X3H2, however, we respectfully rejected that designation because the international address of the individual would have placed an unreasonable burden in mailing expenses on the X3H2 membership. X3H2 requests that X3J3 designate a coordinating liaison to X3H2 in the immediate future.

As X3J3 coordinating liaison to X3H2, this person will receive X3H2 mailings. Please enter the name of our X3H2 coordinating liaison on the X3J3 mailing list, and please advise X3H2 when X3J3 designates a coordinating liaison to X3H2.

Regards,

Barry D. Vickers

Barry D. Vickers, Corresponding Secretary
X3H2 Database

110 JCA-8
P. 1 of 1

(8)

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
Scientific Computing Division/Advanced Methods Section
P. O. Box 3000 • Boulder, Colorado • 80307
Telephone: (303) 497-1275 • FAX: 303-497-1137 TELEX:989761

September 9, 1988

Cathy Kachurik
CBEMA, X3 Secretariat
311 First street, N. W. Suite 500
Washington, DC 20001-2178

Dear Cathy;

X3J3 has instructed me to suggest to you some changes that would be helpful in attaining the maximum public participation in a public review process.

The exclusive rights that Global Engineering has in selling copies of a draft standard has a serious negative impact on the standards process. Many potential reviewers cannot afford the cost of a copy. While the government and ISO may produce their own copies, our private industry may not. There is no journal that would be allowed to reproduce the document without heavy cost.

X3J3 recommends that drafts should be freely available from anywhere that commits to the cost of reproduction. This however does not exclude the sale by a company such as Global. It would however allow a broader distribution of a draft standard and result in a public review that represents a broader base of commenters which is the goal of the standards making process.

Would you consider this recommendation, and if possible, bring this matter to the attention of other Technical Committees?

Regards,

Jeanne Adams

Jeanne Adams, Advanced Methods Group
Chair, X3J3

cc. William Rinehuls, Chair, SPARC

110-LJM-01

S L A C M E M O R A N D U M

August 24, 1988

(9)

To: Interested FORTRAN users

From: L. Moss

Subject: Trip Report on 109th X3J3 Meeting, 8-12 Aug 1988

Note: This is a personal report of these meetings and in no sense does it constitute an official record.

SUMMARY

X3J3 met in Jackson, Wyoming from 8 through 12 Aug 1988.

At the previous meeting in May, a number of the major concerns expressed in the public comment were identified and discussed, but the committee failed to agree to a package of changes to S8 to respond to these concerns. Many members felt that putting together such a package in full committee was bound to fail, since the result would lack consistency and have no clear focus. At the end of that meeting, it was agreed that a number of individuals and small groups within X3J3 would prepare packages according to their own sets of criteria and present them to the full committee at the August meeting.

Nine such proposals were presented to the committee on Monday. Straw votes indicated no clear consensus on any of the plans, so the main order of business for the rest of the week was to try to consolidate the plans to a smaller number. This was accomplished not in full committee, but in small, ad hoc meetings between the proponents of different plans. Each day, full committee straw votes were taken on the current set of plans in order to provide some feedback to the small groups.

By the end of the week, this process had reduced the number of plans to three. A number of changes to S8/104 were contained in all three plans:

- Delete:
  - RANGE/SET RANGE
  - IDENTIFY/ALIAS
  - Allocatable dummy arguments and function results
  - Module procedures
  - Concept of deprecation
  - Internal procedures
  - Elemental calls of user procedures

(9)

  - Free source form and semicolon statement separators (but not the other new features of the fixed source form)
  - Derived types or structures with parameters
  - New form of the DATA statement
- Add:
  - DO WHILE
  - INCLUDE
  - Possibly, pointers (if this can be done in a timely fashion)
- Modify:
  - Replace syntax for array constructors with an implied-do-style syntax
  - Reduce set of intrinsic functions allowed in constant expressions

Note that, although some of the plans in the pre-meeting distribution involved some form of subsets or multiple languages, all of the surviving plans are for a single, non-subsetted language.

The major differences from S8/104 of each of the three surviving plans may be summarized as follows.

Plan P: Philips, et al.

- Delete:
  - User-defined operators and user-overloaded intrinsic operators
  - Overloaded user procedures
  - User-defined assignment
  - Keyword and optional arguments
  - Concept of obsolescence
  - All mandatory use of interface blocks
  - ELSEWHERE
  - Possibly, DO (n) TIMES
- Add:
  - MIL-STD-1753 bit intrinsics
  - Required AUTOMATIC keyword for automatic arrays
  - Short integers, as a separate, non-parameterized type (i.e., SHORT INTEGER rather than INTEGER(KIND=2) or the like)
  - Vector-valued subscripts
  - Possibly:
    - Bit data type (if this can be done in a timely fashion)
    - Stream I/O
    - Support for multibyte character sets
- Modify:
  - Replace generalized precision with two new REAL types:
    - 1 with guaranteed 14 digits precision
    - 1 with maximum precision available from processor
  - Simplify rules for array passing:
    - Array sections and expressions may only be passed to assumed-shape dummies
    - Whole arrays and array elements may only be passed to explicit-shape dummies

(9)

- Move construct names to end of initial statement of construct (i.e., don't make them look like alphanumeric labels)
- Change structure qualification symbol from "%" to "."
- Merge DOTPRODUCT and MATMUL
- Relax rules for type equivalence: require identical declarations rather than import from the same module ("name equivalence")
- Possibly:
  - -- Replace derived types with VAX structures
  - -- Allow structured objects in COMMON

Plan R: Reid

- Delete:
  - User-defined operators (but not user-overloaded intrinsic operators)
  - ELSEWHERE
- Add:
  - MIL-STD-1753 bit intrinsics
  - Required AUTOMATIC keyword for automatic arrays
- Modify:
  - Replace generalized precision with a parameterized form of precision with a single parameter ("KIND") and no assumed precision (i.e., no "KIND=*")
  - Adopt user-defined generics from Plan W, below.
  - Adopt array passing rules from plan P, above.
  - Move construct names to end of initial statement of construct
  - Merge DOTPRODUCT and MATMUL
  - Possibly:
    - -- Allow structures in COMMON
    - -- Relax type equivalence rules to name equivalence

Plan W: Weaver, et al.

- Delete:
  - User-defined operators and user-overloaded intrinsic operators
  - User-defined assignment
  - Keyword and optional arguments
  - Concept of obsolescence
  - MODULE/USE
  - Entity-oriented declarations
  - DO (n) TIMES
  - Construct names
- Add:
  - Bit data type
  - Short integers as a separate type
  - Vector-valued subscripts
  - NCHARACTER
  - Varying strings (CHARACTER, NCHARACTER and BIT)
  - Symbolic logical operators
  - Conversion function for each type

(9)

- Modify:
  - User-defined generics: replace procedure overloading mechanism with extension to interface blocks in which the user provides an explicit mapping from generic to specific names
  - Replace derived types with VAX structures
  - Allow structures in COMMON
  - Simplify generalized precision: delete assumed precision (i.e., "REAL(*,*)") and parameter keywords ("PRECISION=" and "EXPONENT_RANGE=")

The proponents of these three plans will continue the process of developing their plans in greater detail as well as trying to find further consolidations with the other plans. It is hoped that a final plan can be chosen at the Boston meeting in November.

PRESENTATION OF PLANS

[All X3J3 working documents are assigned numbers of the form, "mmm-aaa-n", where:

mmm is the meeting number (the Aug 1988 meeting was number 109).

aaa are the initials of the author.

n is a small number to distinguish different documents from a single author at one meeting.

The results of straw votes (SV) are, unless otherwise noted, given as: (yes-no-undecided), with an asterisk next to my vote; formal votes (FV) are (yes-no-present), but are usually recorded simply as (yes-no).

[At this stage in X3J3's deliberations, any change to the draft requires a two-thirds majority of those voting AND a simple majority of the entire membership. The latter requirement translates into a minimum of 23 votes with the committee's current membership list. A number of "members-only" straw votes (MSV) were taken at this meeting in order to give a better indication of whether a given proposal might later be formally approved. These votes are recorded here as (yes-no-undecided).]

The following plans were each allotted 30 minutes on Monday for a presentation and discussion. After each discussion, a straw vote was taken on whether the plan should be adopted as a base for further work. For each plan, I will give only a very brief list of the major features ("major" according to my personal biases, of course). For a more complete description, as well as for an explanation of the philosophy, design criteria, etc., which went into the plan, please see the referenced documents. I apologize in advance to the authors of the plans for any mistakes, omissions, or oversimplifications.

Except as noted, essentially all the plans include (or at least could live with) the following:

- Delete:
  - Concept of Deprecation
  - IDENTIFY/ALIAS -- except plan VI (Barber)
  - RANGE/SET RANGE -- except plan VI (Barber), which contains a simplified, block-oriented form of range)
- Add:
  - Bit functions
  - INCLUDE -- except plan III (Reid)
  - DO WHILE -- except plans IV (Smith, et al.) and IX (Weaver, et al.)

Plan I: Ivor Philips, et al. (Ref: 109-IRP-1)

- Based on S8/104
- Delete:
  - New form of type declarations
  - User-defined operators and user-overloaded intrinsic operators
  - User-defined assignment
  - Overloaded user procedures
  - Keyword and optional arguments
  - New form of DATA statement
  - Dependent compilation (in the sense of imposing an order on compilation)
  - Concept of obsolescence (as well as deprecation)
  - Construct names
  - Derived type parameters
  - Elemental calls of user procedures
- Simplify:
  - Array language:
    - -- No array-valued functions
    - -- Only array sections passed to assumed-shape dummies
    - -- No array sections passed to explicit-shape dummies
    - -- Restrictions on allocatable arrays
    - -- Require AUTOMATIC keyword for automatic arrays
  - Precision: Add new floating point data types (both REAL and COMPLEX):
    - -- 1 with guaranteed 14 digits precision
    - -- 1 with maximum precision available from processor
  - No procedures in MODULEs
  - No mandatory use of interface blocks
  - Type matching rules for derived types: Do not require definition in a MODULE
  - Restrict set of intrinsic functions allowed in constant expressions
- Defer until next standard:
  - BIT data type
  - Pointers
  - Support for multibyte character sets

(9)

SV (22-10*-7).

Plan II: H. Wada, et al. (Ref: 109-HW-1)

- Based on S8/104
- Delete:
  - User-defined operators and user-overloaded intrinsic operators
  - User-defined assignment
  - Overloaded user procedures
  - MODULE/USE
- Simplify:
  - Replace derived types with VAX structures
  - Precision: find some solution which does not involve REAL(*,*)
- Add:
  - Support for multibyte character sets
  - Vector-valued subscripts

SV (10-16*-14).

Plan III: John Reid (Ref: 109-JKR-4)

- Based on S8/104
- Delete:
  - Internal and MODULE procedures
  - Derived type parameters
  - Elemental calls of user procedures
  - Overloaded user procedures
  - New form of DATA statement
  - Free source form (but keep new features of fixed source form)
  - ELSEWHERE
- Simplify:
  - Precision
    - -- Single KIND= parameter
    - -- No assumed precision (i.e., no "KIND=*")
    - -- No intrinsic function to map precision/range into KIND
  - MODULE/USE: No renames in USE statement, etc.
  - Argument association for arrays:
    - -- No allocatable dummy arguments or function results
    - -- Simpler rules for assumed-shape dummies, etc.
- Add pointers
- Modify array constructor syntax: Make it look like an implied-do
- Omit INCLUDE

SV (21*-2-17).

(9)

Plan IV: B. Smith, et al. (Ref: 109-ABMSW-1 to -13)

This is a three layer model:

- Outer or "full" layer, containing most of S8/104, except:
  - Delete:
    - -- Concept of deprecation
    - -- IDENTIFY/ALIAS
    - -- RANGE/SET RANGE
    - -- Internal procedures
  - Simplify precision: parameterize with a single ("KIND=") parameter, no assumed precision (i.e., no "KIND=*"), and add an intrinsic function to map precisions and exponent ranges into "kinds".
  - Add:
    - -- Pointers
    - -- Significant blanks in free source form
    - -- MIL-STD-1753 bit functions, but with different names
  - Omit DO WHILE
- Intermediate or "core" layer,
  - omitting features of full layer requiring heap storage or explicit interfaces, namely:
    - -- Pointers
    - -- ALLOCATE/DEALLOCATE
    - -- Assumed-shape dummy arguments
    - -- Elemental calls of user procedures
    - -- Parametrized data structures
    - -- Private types and MODULE entities
    - -- Interface blocks
    - -- Module procedures
    - -- USE ONLY and USE renaming
    - -- User-defined operators and user-overloaded intrinsic operators
    - -- User-defined assignment
    - -- Overloaded user procedures
    - -- Keyword or optional arguments, except for intrinsic procedures
    - -- INTENT attribute
    - -- Several intrinsic functions associated with some of these features
  - and making various other simplifications:
    - -- Omitting entity-oriented type declarations and the new form of the DATA statement
    - -- Omitting the intrinsic function mapping precision/range to "kinds"
- Inner or "base" layer, identical to Fortran 77.

SV (11-18*-10).

(9)

Plan V: E. A. Johnson and R. Swift (Ref: 109-EAJ-1)

- Based on S8/104
- Delete:
  - User-defined operators and user-overloaded intrinsic operators
  - User-defined assignment
  - Module procedures
  - USE statement
- Simplify:
  - MODULEs: textually INCLUDE modules in referencing routines, separately compile modules to instantiate data. Also, inside a MODULE, IMPLICIT NONE and SAVE would always be in effect.
  - Precision: add new intrinsic REAL types, or parametrize REAL, with standard-defined minimum precisions
  - Rules for optional and keyword arguments
- Add:
  - Small INTEGERs and LOGICALs
  - In addition to new exponent letters associated with new REAL types, also add a "generalized" exponent letter, which assumes the precision of other operands in an expression

SV (9*-12-22).

Plan VI: G. Barber (Ref: 109-GJB-1)

- Based on Fortran 77
- Add:
  - Array language, including:
    -- Vector-valued subscripts
    -- Array IDENTIFY (no scalar IDENTIFY)
    -- Block-oriented RANGE facility
  - Simplified interface blocks
  - Variant forms of intrinsic types (i.e., either "*n" forms or "SHORT INTEGER", etc.)
  - NAMELIST I/O
  - MIL-STD-1753 features (i.e., INCLUDE, DO WHILE, ENDDO, IMPLICIT NONE, and bit functions)
  - CYCLE and EXIT
  - Various new lexical features (long names, etc.)
  - Hex, octal, and binary constants

SV (18*-9-16).

Plan VII: M. Ellis (Ref: 109-TMRE-2)

This is a two language model.

- Fortran 88: consisting of Fortran 77 plus MIL-STD-1753.
- Fortran 90: new language, not completely compatible with Fortran 77 or Fortran 88, and based on S8/104:
  - Delete:

(9)

  - -- Fixed source form
  - -- All the obsolescent features
  - -- Computed GOTO
  - -- ENTRY
  - -- DIMENSION
  - -- H edit descriptor
  - -- Numeric labels
- Clean up syntax by:
  - -- Adding significant blanks
  - -- Turning all keywords into reserved words, including user-defined keywords such as derived type names
  - -- Taking advantage of these two changes to simplify some of the new syntax (e.g., declarations for objects of derived type)
- Add:
  - -- Interface to allow calling of Fortran 77 and Fortran 88 subprograms (e.g., F77 and/or F88 attributes on the EXTERNAL statement)
  - -- BIT data type
  - -- Pointers
  - -- Support for multibyte character sets

SV (7-26*-9) --- WITHDRAWN.

Plan VIII: A. Marusak (Ref: 109-ALM-2 and -3)

This was not a complete plan, but rather a set of guidelines for constructing a plan along with a few examples of how the guidelines would apply to some features. Combining these examples, resulting plans might look something like this:

- Based on S8/104
- Delete:
  - All mandatory dependent compilation (here "independent compilation" is defined in a stronger sense than in plan I: it should be possible to directly type in all code rather than INCLUDEing or USEing it)
  - Derived type parameters
- Simplify generalized precision: either drop it entirely (adding DOUBLE COMPLEX), or find some way to get rid of assumed precision (i.e., "REAL(*,*)")
- Add:
  - BIT data type
  - Pointers
  - Alphanumeric labels
  - ELEMENTAL keyword for user-defined elemental procedures
  - Any combination of new, old, and user-defined data types in COMMON and EQUIVALENCE

No straw vote was taken since the author felt that this was not a complete proposal but only a set of guidelines.

Plan IX: R. Weaver, et al. (109-RWW-1 and -2)

- Based on Fortran 77
- Add:
  - Varying length CHARACTER
  - Two new string types (both fixed and varying):
    - BIT
    - NCHARACTER
  - Symbolic BIT operators (&, ++, --, and /, for "and", "or", "xor", and "not", respectively)
  - Alternate symbolic relational operators from S8/104 (">", "==", etc.)
  - Simplified version of generalized precision from S8/104 (no REAL(*,*) and no PRECISION and EXPONENT_RANGE keywords)
  - DOUBLE PRECISION COMPLEX
  - New DO construct, including EXIT and CYCLE from innermost loop, but excluding construct names (NB: DO WHILE omitted)
  - CASE construct
  - VAX structures (with some minor variations)
  - NAMELIST I/O
  - Some intrinsic functions allowed in constant expressions
  - Array and structure assignment
  - Array and structure named constants (i.e., defined via a new form of the PARAMETER statement)
  - Conformance statement, similar to that in S8/104, less deprecated features, etc.
  - New features of fixed source form from S8/104
  - IMPLICIT NONE
  - Some additional intrinsic functions from S8/104
- Optional features
  - Recursion
  - Dynamic allocation
  - Most of array language, with simplified rules for assumed-shape dummy arguments and possibly non-contiguous actual arguments.
- Other possible additions
  - Alphanumeric labels
  - Additional forms of INTEGER
  - Stream I/O
- Defer until next standard: Pointers

Other plans

A couple of other plans were mentioned in the agenda, or included in the pre-meeting distribution

- Michael Berry: No document or presentation prepared.

- Richard Hendrickson: No document or presentation prepared.

- "Comments on the Public Review and Future Directions of FORTRAN from the Canadian Standards Association" (item 33 in the pre-

(9)

meeting distribution): This was not really a plan, but a list of Canadian positions on individual features, as well as on the plan presented at the 108th meeting by the Technical Change Review committee.

- Lawrie Schonfelder's plans, 109-JLS-1 and -2: Lawrie withdrew these before the meeting.

Straw votes on initial plans

Following the presentation of all the above plans and the withdrawal of plans VII and VIII, the straw votes were repeated on the remaining plans, with members only voting:

- Plan I: MSV (22-7*-7).
- Plan II: MSV (6-17*-10).
- Plan III: MSV (20*-5-15).
- Plan IV: MSV (11-20*-7).
- Plan V: MSV (12*-8-17).
- Plan VI: MSV (11*-12-8).
- Plan IX: MSV (16-12*-6).

PLAN CONSOLIDATION, ROUND 1

Although several of these plans appeared fairly popular none had the necessary majority of the full membership, nor did there appear to be a clear winner among the leading contenders. Therefore, the principals involved in each plan were asked to meet in a few small groups on Tuesday and attempt to consolidate the remaining 7 plans down to a more manageable number. Members unaffiliated with any plan were encouraged to attend these ad hoc meetings in order to provide additional input, but any compromises were to be left to the principals themselves in order to retain some consistency and focus in the resulting plans.

On Wednesday, the field had been reduced to 4 plans:

- John Reid and the group represented by Ivor Philips had moved closer together, though they still had some significant differences.
- Andy Johnson had also worked with Philips and Reid, and incorporated some of his ideas into each of their plans. He felt that he could probably accept either of the resulting plans, and so withdrew his plan from separate consideration.

(9)

- The group represented by Dick Weaver met with Hideo Wada and Graham Barber, and agreed to a small set of changes to plan IX.

- The group represented by Brian Smith met and agreed to a few changes to plan IV based on the discussions and straw votes on Monday.

The changes to each of the plans were briefly presented.

Plan IX' (Ref: 109-RWW-5)

- Make all optional features (array language, dynamic allocation, and recursion) mandatory
- Add allocatable arrays with global scope
- Add array and structure constructors
- Add vector-valued subscripts
- Add array-valued functions
- Add and modify some array intrinsics
- Possibly, add block forms of IDENTIFY and/or RANGE
- Add user-defined generics, with mapping to external names provided explicitly by user via extension to interface blocks
- Add DO WHILE and DO forever
- Add INTENT statement
- Allow all data types to be intermixed in COMMON
- Possibly, make some additional changes to REAL precision
- Possibly, add INTEGER precision and/or unsigned integers

Plan I' (Ref: 109-IRP-3)

- Keep entity-oriented declarations
- Disallow derived types in COMMON (unless a good solution can be found)
- Keep construct names, but move to the end of the initial statement of a construct (i.e., don't make them look like alphanumeric labels)
- Keep array-valued and structure-valued functions
- Delete internal procedures and CONTAINS
- Delete semicolons as statement separators
- Delete elemental calls of user procedures and ELEMENTAL keyword
- Add octal and hex constants
- Add vector-valued subscripts
- Add pointers -- if this can be done in a simple, efficient, and timely fashion
- Fix array passing rules: array expressions, like array sections, may only be passed to assumed-shape arrays.
- Some possibilities for replacing MODULE/USE:
  - Replace with GLOBAL attribute or statement
  - Modify according to plan V (i.e., MODULE/INCLUDE)
  - Other possibilities which preserve independent compilation (in the sense of not imposing a compilation order)
- Adopt implied-do array constructor syntax from plan III

(9)

- Merge MATMUL and DOTPRODUCT, as in plan III
- Possibly, allow REAL types to be parameterized via CHARACTER constants

Plan III' (Ref: 109-JKR-7)

- Adopt rules for association of arrays from plan I' (array sections and expressions <==> assumed-shape arrays)
- Delete semicolons as statement separators
- Add bit, octal, and hex constants
- Require AUTOMATIC keyword for automatic arrays
- Move construct names to end of initial statement of constructs
- Restrict intrinsic functions allowed in constant expressions
- MODULE data always SAVEd
- Recommend NCHARACTER as collateral international standard

A number of disagreements with Plan I' remain:

- Mild disagreements:
  - Plan I': Add INCLUDE
  - Plan I': Add short integers
  - Requirements for derived type equivalence:
    - -- This plan: imported from same MODULE via USE
    - -- Plan I': identical declarations (i.e., same type name; same component names, types, type-parameters, and shapes; same component order)
  - Plan I': Change "%" to "." for structure qualifier
  - Plan I': Delete concept of obsolescence
  - Plan I': Add vector-valued subscripts
  - This plan: Add user-defined generics via extensions to interface block from Plan IX'
- Serious disagreements:
  - Simplification to generalized precision:
    - -- This plan: Parameterized precision as in plan IV
    - -- Plan I': no change, i.e., new intrinsic types
  - This plan: Retain overloaded intrinsic operators and assignment
  - This plan: Retain MODULE/USE but without module procedures
  - This plan: Retain keyword and optional arguments
  - This plan: In general, prefers to parametrize similar types rather than having separate names; for example, this approach would be preferred if short integers or multibyte characters were to be added
  - This plan: make use of interface block for array-valued functions instead of extending syntax for EXTERNAL, etc.

(9)

Plan IV' (Ref: 109-BTS-4)

- Delete base and core subsets
- Delete square brackets from array constructor syntax
- Delete user-defined operators (but keep intrinsic operator overloading, including user overload to the intrinsic "dot" operators)
- Require ELEMENTAL attribute on dummies for elemental calls of user procedures
- Change host association to USE association for module procedures
- Delete allocatable dummies
- Integrate all types in COMMON and EQUIVALENCE
- Other possible changes:
  - Array association rules from Plan I'
  - Allow statement label in EXIT and CYCLE
  - Remove construct names and/or add alphanumeric labels
  - Add DO WHILE and/or delete DO (n) TIMES
  - Remove elemental calls of user procedures
  - Restore MIL-STD-1753 names for bit intrinsics
  - Remove ELSEWHERE
  - Adopt implied-do array constructor syntax from plan III
  - Remove NAMELIST
  - Remove allocatable function results
  - Change syntax of entity-oriented declarations
  - Adopt user-defined generics via extension to interface blocks, as in Plan IX'

Straw votes on first-round consolidated plans

- Members only
  - Plan IX': MSV (15-12-9*).
  - Plan I': MSV (17-13*-4).
  - Plan III': MSV (14*-10-9).
  - Plan IV': MSV (12*-15-7).
- Everyone
  - Plan IX': SV (24-13-9*).
  - Plan I': SV (19-13*-10).
  - Plan III': SV (14*-10-16).
  - Plan IV': SV (14*-20-10).

GOALS OF FORTRAN 8X

A number of members and public commenters have remarked at various times that Fortran 8x did not seem to them to have any clear focus, or that the development of 8x did not appear to have followed a "top-down" approach, starting with an agreed upon set of goals for the new language. Time was therefore allotted for a discussion of the goals of Fortran 8x.

(9)

John Reid pointed out that, in fact, a set of goals had been formally adopted by the committee in 1983 as part of S6 (a copy of the relevant pages from S6 was placed on the table as item 92). A number of members said they had never seen this document, and asked why no provision was made to supply such information to new members. Others questioned whether the goals expressed in 1983 were still adequate 5 years later.

A group of interested members, led by Kevin Harris, was asked to prepare a new list of goals for further discussion. A first draft of such a list was placed on the table as 109-KH-1, item 106. This is a fairly complete list of general goals, such as portability, safety, etc., with a very detailed breakdown of how these general principles apply to a number of different aspects of the complete standard defining and implementing cycle. It rightly points out that different goals very often are in direct conflict. As part of the discussion of this document, Kevin asked for several straw votes to try to get a sense of how the committee as a whole assigns weights to these different goals when they conflict. For each vote, the question is something like, "When A and B conflict, would you usually consider A more important than B?"

- Is semantic precision more important than timeliness? SV (20*-4).
- Is power more important than safety? SV (17-4).
- Is performance more important than understandability? SV (18*-3).
- Is power more important than portability? SV (6-13).
- Is power more important than timeliness? SV (6-17*).

Kurt Hirchert had placed an alternative statement of goals in the pre-meeting distribution as 109-KWH-1. This was discussed by a subgroup, and an amended version was placed on the table as 109-KWH-1a, item 118.

This subject will probably be given more full committee time at the November meeting.

PLAN CONSOLIDATION, ROUND 2

Members were asked to study the four round 1 plans overnight and try to prepare brief summaries of their objections for discussions on Thursday. At the same time, some of the plan principals met to seek further consolidations. Although a number of other possible areas of future compromise were identified, relatively few actual changes were made at this time. (One exception that I noted, since it changed my vote on one of the plans, was that Plan I' accepted MODULE/USE without module procedures when it became clear that this could be implemented

(9)

without the type of dependent compilation that the plan I' backers objected to.)

After presentations and discussions of the four plans, but before any straw votes were taken, there was a discussion of what "adopted as a base for further work" meant. A straw vote was requested on the statement: "The compromise plan should be the final choice of features": SV (29-8*-5). A formal vote was requested on the same statement. Several proponents expressed the fear that, after adopting a compromise plan, we would continue to make major changes as we did after Scranton. Some opponents pointed out that many of the plans were quite sketchy at this point, so that it was not always clear whether or not a given feature was included in a particular plan. Moreover, many possible changes had occurred over a short period of time and many members wanted more time to study any resulting compromise plan before irrevocably committing themselves to it. FV (14-17*) -- FAILS.

Another round of straw votes was taken on the slightly modified, round 2 plans:

- Members only
  - Plan IX'': MSV (11-15-4*).
  - Plan I'': MSV (15*-13-1).
  - Plan III'': MSV (13*-10-8).
  - Plan IV'': MSV (13*-17-2).
- Everyone
  - Plan IX'': SV (18-17-4*).
  - Plan I'': SV (19*-17-5).
  - Plan III'': SV (16*-13-12).
  - Plan IV'': SV (15*-23-4).

Another set of straw votes was requested in which people were asked to vote for their one favorite plan:
MSV (IX'':7 - I'':10 - III'':7 - IV'':9*).
SV (IX'':14 - I'':9 - III'':8 - IV'':10*).

I expressed some concern that none of the plans would ultimately obtain the necessary support within the committee, and therefore asked for the following straw vote: "If the committee fails to achieve consensus on any of these plans, then a minimal plan, such as that presented by Graham Barber, should be pursued": MSV (6*-16-8).

(9)

PLAN CONSOLIDATION, ROUND 3

On Friday, a further consolidation had been achieved: Brian Smith split off from the other proponents of Plan IV'', and merged his plan with that of John Reid. I will label the resulting plan III''', since the only change from Plan III'' was that INCLUDE was added. Since the other backers of Plan IV'' were willing to withdraw their plan from further consideration, the committee was down to three plans, and another round of straw votes was taken:

- Members only
  - Plan IX'': MSV (13-17-4*).
  - Plan I'': MSV (19*-12-3).
  - Plan III''': MSV (15*-14-6).
- Everyone
  - Plan IX'': SV (22-19-4*).
  - Plan I'': SV (21*-19-5).
  - Plan III''': SV (18*-18-9).

Most (though not all) members felt that considerable progress had been made by the end of the week: the initial field of nine different plans had been reduced to 3, and creative new compromises had been found for some crucial issues. There was some hope of reaching agreement on a plan at the Boston meeting in November.

INSTRUCTIONS TO THE U.S. DELEGATION TO THE WG5 MEETING

A couple of resolutions were proposed and discussed containing instructions for the U.S. delegation to the WG5 meeting in Paris next month.

The first had to do with whether or not plan IV'' (the single layer version of the plan originally presented in a three layer version by Smith, et al.) should be presented to WG5. Since this plan had been withdrawn from further active consideration by X3J3, many members felt it was simply a waste of time to present it in Paris. Motion: "X3J3 instructs the U.S. delegation not to present the ABMSW plan to WG5.": FV (11-9*) -- PASSES.

The second concerned the possibility that WG5 might decide to rescind its delegation to X3J3 of the job of preparing a new, international Fortran standard. The proponents were concerned about the danger of ending up with two standards, while the opponents were more afraid of ending up with no standard at all. Motion: "X3J3 instructs the U.S. delegation to work to prevent an international split of ownership of the Fortran standard.": FV (12-6*) -- PASSES.

(9)

OTHER TECHNICAL WORK

Several major changes which were included in virtually all the plans, as well as some minor technical fixups, were worked on at this meeting.

109-ADT-1 ES Edit Descriptor
This was a proposal to add a format descriptor for scientific notation (i.e., like E or D, but with one significant digit before the decimal point), analagous to the EN descriptor for engineering notation. Note that the P (scale factor) descriptor can be used to get this effect, but because it is "sticky" and also applies to F descriptors, is much more awkward to use. Initial straw vote: SV (8*-3-11). Several people indicated they would prefer a solution such as splitting the P descriptor into two separate descriptors, one that applied when the output had no exponent field and another when there was an exponent field: SV (1-5-17*). The proposal was withdrawn to go back to subgroup.

109-JHM-1 CARRIAGE= Specifier
This was a proposal to add a specifier to the OPEN statement to allow the programmer to indicate whether column one is to be interpreted as carriage control. An initial straw vote was taken whether to provide such functionality: SV (31*-1-0). The proposed values for this specifier were "FORTRAN" or "NONE". Several people objected to these values, noting that DEC has a similar specifier (named CARRIAGECONTROL=) for which the "NONE" value has a different meaning. A straw vote was taken on using the values "FORTRAN" and "NONE": SV (9*-12-6). There was also some discussion about the name of the specifier: "COLUMN1" and "CC" were also suggested. The proposal was sent back to subgroup for further work.

109-PLS-1 Messages for IOSTAT values
A couple of public comments suggested providing some way for the programmer to get the text of the error message that would have been issued for an I/O error if an IOSTAT= (or ERR=) specifier had not been coded. This paper outlined several possible ways of responding to this request. After some discussion, the fifth possibility -- "do nothing" -- was straw voted: SV (18*-3-8).

109-RCA-2 Overlapping CASE
The CASE construct in S8 does not allow case ranges corresponding to different blocks of code to overlap; however, this restriction is stated in the text rather than as a constraint, so processors are not required to check it. Proposal 1 in this paper would make the restriction into a constraint. Initial straw vote: SV (20-0-6*). I pointed out that this restriction had been intentionally left out of the constraints by the subgroup, since we did not have a good feel for how difficult it was to check. Several implementers indicated it was not difficult at all, so a formal vote was taken: FV (24*-0) -- PASSES.

The current text in S8 describing the CASE construct explicitly permits several case ranges associated with a single block of code to overlap. Proposal 2 would prohibit such overlaps. Initial straw vote: SV (12-3-11*). One possible situation where such overlaps might naturally occur would be if some of the ranges were defined by means of PARAMETERs. After further discussion, this proposal was sent back to subgroup.

109-RCA-3 B, O, and Z edit descriptors.
Proposal 1 would add 3 new edit descriptors to produce output in binary (B), octal (O), or hexadecimal (Z), whereas proposal 2 would add a single new "radix" descriptor which would be "sticky", like the P descriptor, and would determine the radix to be used with the I descriptor. Straw votes were taken on each proposal. Proposal 1: SV (22*-1-5). Proposal 2: SV (5-17*-6). Proposal 2 was withdrawn, and proposal 1 was taken back to subgroup.

109-ABMSW-10 Delete Concept of Deprecated Features
Since none of the plans included the concept of deprecation, and since this proposal (part of the original Smith, et al. plan) included the actual text necessary to implement such a change to S8, a formal vote was taken: FV (23*-1) -- PASSES.

EDITORIAL WORK

A number of minor editorial proposals were passed at this meeting. The details will be available in the formal minutes of the meeting.

A number of editorial, and a few technical, changes have been made to S8/104 (the version of the document distributed for public review). Up until now, these changes have been preserved in a separate standing document, S16. It is becoming increasingly difficult to accurately write proposals against the "virtual" document obtained by merging S16 changes into S8/104. Accordingly, a new document was created by carrying out this merge and was distributed before this meeting as S8/108. A formal motion to adopt this as the new base document was discussed and voted: FV (6-25*) -- FAILS. Several objections were mentioned in the discussion:

- One appendix was missing entirely, and another seemed to have reverted to an earlier version.
- Several members said they had not had time to compare the new document carefully with the old.
- There was some confusion as to what changes were included due in part to a discrepancy between the meeting number on the cover (108) and that on the bottom of each page (109).

(9)

It was agreed that an attempt to adopt a new base document should be made at the next meeting.

In the meantime it was suggested that proposals for the next meeting should use line numbers from S8/108, as distributed: FV (19-13*) -- PASSES.

OTHER BUSINESS

Public Review Forum

Three individuals who had sent in public review comments were given agenda time at this meeting to present their views. They were:

- Tom Lahey, Lahey Computer Systems, Inc.
- Prof. Geoffrey Hunter, Theoretical Chemistry Department, Oxford University.
- Dr. Henry Todd, Department of Computer Science, Brigham Young University.

Fortran 77 Reaffirmation

Due to a new ruling from ANSI, all standards must be either reaffirmed or withdrawn by their tenth anniversary, regardless of where they are in the revision cycle. As a result, X3 was forced to take emergency action to reaffirm Fortran 77. Reaffirmation involves a 2 month public review which will take place from 26 August to 25 October, 1988. Please note that this is simply a pro forma public review, since the revision process (i.e., the Fortran 8x effort) will continue independently of the Fortran 77 reaffirmation.

Parallel Computing Forum

Brian Smith reported that the Parallel Computing Forum has met twice since the last X3J3 meeting in May. The PCF is preparing a set of suggested extensions to Fortran 77 to support parallel computing, and plans to release a document for public comment sometime this month. A draft of this document was available on the table (item 96).

IRDS Letter Ballot

X3J3 is a coordinating liaison committee for dpANS IRDS Services Interface. The committee developing that standard, X3H4, has reached Milestone 8, which involves a 30-day letter ballot of the coordinating liason committees. Accordingly, Jeanne Adams is instigating such a ballot, for the period 26 August to 26 September 1988. Ballots will be mailed out shortly to any members who did not take one in Jackson. NB: This is a required letter ballot, which counts towards the two-out-of-three membership requirement.

(9)

ADMINISTRATIVE BUSINESS

Membership
At the beginning of this meeting there were 44 members, giving a quorum of 15 (=1+INT(Members/3)), and a majority of the membership of 23 (=1+INT(Members/2)).

Minutes of 108th Meeting
Motion to approve the minutes of meeting 108, as amended by 109-JKR-2 and 109-JKR-5 (containing late scribe notes and a few other minor changes): Passed by unanimous consent.

Future Meetings

1988 WG5 Meeting: 19-23 Sep 88, Paris, France (host: C. Bourstin, AFNOR).

110th: 13-18 November 1988, Cambridge, Mass. (host: Michael Berry, Thinking Machines Corporation). The meeting hotel is the Royal Sonesta, 5 Cambridge Parkway, Cambridge, MA 02142, (617) 491-3600. The room rate will be $68 for those entitled to the GSA rate and $110 otherwise (mention "ANSI Fortran Standards Committee" to get these rates). In addition, if you plan to use the GSA rate, you MUST inform the host ASAP. The registration fee will be $70.

Note that this will be a six day meeting, starting at 10:00 AM on Sunday, 13 November.

111th: 12-17 February 1989, SLAC, Calif. (hosts: Len Moss and Sunnie Sund). The meeting hotel will be the Palo Alto Holiday Inn. The single/double room rates are: $82/$92 (GSA: $51.50/$63.50).

112th: 7-12 May 89, Long Island, NY (hosts: Bruce Martin and Paul Libassi). The GSA rate of $103 (!) will be extended to all attendees; an alternate, less expensive hotel will also be available.

1989 WG5 Meeting: 10-14 July 89, Ispra, Italy (host: Aurelio Pollicini)

113th: 17-21 July 89, Vienna, Austria (host: Gerhard Schmitt, Technical University of Vienna).

114th: 5-10 November 89, Dallas, Texas (host: Presley Smith, CONVEX Computer Corporation).

Next Distribution

The closing date for the next pre-meeting distribution is 28 September 1988. To get an item into the distribution it should be received before this date by:

Neldon Marshall
EG&G Idaho Inc.
P.O. Box 1625
Idaho Falls, ID 83415
(208-526-9342)

(10)

110-JTM-1

TO: X3J3

FROM: Jeanne Martin

SUBJECT: Proposal to Add Pointers and Delete IDENTIFY/ALIAS

REFERENCES: 108-JLS-1(44)
S8.108
109-ABMSW-3(57)
109-JTM-3(85)

1 Changes to ABMSW-3 Suggested in Jackson Hole:

109-JTM-3(85) has been incorporated in this proposal. It makes the syntax for the renaming of module objects consistent with the proposed pointer assignment statement. The other changes listed below were suggested during the discussion of 109-ABMSW-3(57) in Jackson Hole or mentioned to me by individual X3J3 members.

From Rich Ragan: Make sure that the restrictions against overlapping actual arguments apply equally to pointer targets. [Page 12-13, lines 13-15, state that this is the case and proceed with an example, lines 16-21. To make sure that it is clear that this restriction also applies to pointer targets, I have added a second example; see item 94.]

From Bob Allison: Retain knowledge of one-time allocations for optimization purposes. These could be treated in the same way as static objects. [I felt this would significantly increase the possibilities for optimization, so I have added an attribute, DEFERRED, that specifies an object that is allocated once and subsequently is treated just as a static entity is treated. It is neither a target (unless so declared) nor a pointer. Aside from the fact that its size and location are determined at runtime, it is just like a static, compiletime object. The DEFERRED attribute is described in item 36; an example program using it is in item 113 (C.5.4). There are numerous other changes for this addition throughout the proposal.]

From Len Moss and Paul Sinclair Prevent indirectly recursive pointers. [I thought this was covered by Page 4-6, Line 12. However, I have added a sentence to make it explicit. See item 5.]

From Larry Rolison I prefer pointers not strongly typed. [I felt this was a minority opinion and did not make any changes to the proposal which is definitely for strongly-typed pointers.]

From Ivor Philips Remove allocatable dummy arguments and function results. [This can be done without severely damaging the functionality being proposed for pointers, however I understand that Ivor is preparing such a proposal. The items in this proposal that would have to be changed if Ivor's proposal is accepted are 24, 25, 32, 55, 56(last paragraph), 62, 69, 83, 84, 86, 88, 90, 91, 104, and 115.]

From Walt Brainerd, Carl Burch, and Brian Smith [A number of wording changes were suggested that improve the proposal. These have been incorporated.]

2 Introduction to the Proposal:

A large number of the public review comments suggested that F8X should include a pointer facility. A straw vote at the 108th meeting was (21-6-9) in favor of adding pointers. Another straw vote was (23-5-7) in favor of removing IDENTIFY/ALIAS. This proposal accomplishes both of those tasks.

108-JLS-1 makes use of the IDENTIFY statement for pointer assignment. This proposal removes the IDENTIFY statement, so a new pointer assignment mechanism is introduced. It is a pointer assignment statement that uses the symbols => (as in the USE statement). This proposal also removes the ALLOCATABLE attribute because it has exactly the same semantics as the POINTER attribute (See page 5-8 as revised). This proposal adds the DEFERRED attribute to specify an object that is allocated once at runtime, and is subsequently treated in the same fashion as a static, wholly compiletime-specified object. Otherwise this proposal is identical in intent to 108-JLS-1.

3 General Description:

This proposal introduces three new attributes, POINTER, TARGET, and DEFERRED. A pointer must be declared with the POINTER attribute,

along with the type and rank of the allowable target. Any static object that is to be permitted as a pointer target must be declared with the TARGET attribute. An object that is allocated once at runtime and subsequently treated as a static object must be declared with the DEFERRED attribute. The roles are designed to aid the processor to do as much of the usual optimization as possible. A pointer is always permitted as a pointer target, but unless declared as such a static object (including a deferred object) is not permitted as a target.

An obvious model for interpreting declarations of pointers is that such declarations create for each name a descriptor. Such a descriptor includes all the data necessary to fully describe and locate in memory, an object, and all sub-objects, of the type, type-parameters and rank specified. The descriptor is created empty; it does not contain values describing how to access any actual memory space. These descriptor values will be filled in when the pointer is associated with actual target space.

An object with the DEFERRED attribute must not be associated by pointer assignment; it can only be associated by allocation and can only be associated once within its scoping unit. A dummy argument must not be declared with the DEFERRED attribute.

A pointer may be associated with an object by allocation or pointer assignment. If a pointer is included in an ALLOCATE statement, space to hold an object of the relevant type and specified shape is allocated and is associated with the pointer; any previously existing association of the pointer with an object is broken. If a pointer appears in a pointer assignment statement with a permitted target, the pointer becomes associated with the space refered to by the target; any previous association of the pointer with an object is broken.

One or more components of a derived type may be defined to have the POINTER attribute, in which case any object of this derived type will have one or more pointers for those respective components. A derived type may contain a pointer component whose target object is of the derived type being defined. This allows the construction of lists and trees, etc.

A pointer becomes disassociated from its target object if it appears in a DEALLOCATE statement, or the program unit in which it is declared becomes inactive and the pointer is not saved. A pointer becomes disassociated from any target if it is assigned a pointer target that is itself currently disassociated.

In all expression contexts a pointer is dereferenced and the current target is used. Similarly, when a pointer appears on the left of an intrinsic

(1c)

assignment the pointer is dereferenced and the right-hand side value is assigned into the space currently associated with the pointer. All the normal conformance rules apply in both expression and assignment contexts. When a pointer appears in an input statement, the pointer is dereferenced and the input value is read into the space currently associated with the pointer. On the other hand, a pointer assignment between two pointers will make both reference the same target.

NOTE: Dereferencing can only be applied to whole scalar objects. If a structure containing a pointer component appears in a dereferencing context, the pointer component is not dereferenced. This implies that by default, assignment of a derived type with a pointer component is interpreted as component-by-component assignment for the nonpointer components and pointer assignment for the pointer components.

An undereferencable pointer may not appear in an I/O list.

NOTE: These dereferencing rules are slightly over-restrictive but they are safe and could be relaxed by an easy 9X extension. They do not allow arrays of pointers, which could be produced by selecting a pointer component from an array of structures, to be treated as a whole array. They do not allow a structure with a pointer component to appear in an I/O list.

4 Specific Text for S8.109

The following are the edits necessary to implement this proposal. They are written against S8.109, August 1988, as distributed prior to the August meeting.

1. Page 2-3, line 5, delete; after line 8, add

   or *pointer-assignment-stmt*

2. Page 2-7, line 37-38, replace "However, ... arrays." with

   The extents of a deferred array are determined when the array is allocated and do not vary. However, for dummy argument arrays, automatic arrays, and target arrays, the extents may vary during execution.

3. Page 2-8, after line 1, add

   2.4.8 Pointer. A **pointer** is an object descriptor that is dereferenced when it appears in an expression. Any data object may have

(10)

the POINTER attribute (5.1.2.7). Such an object is empty or disassociated and must not be referenced or defined until it becomes associated with a target object as a result of executing a pointer assignment statement (7.5.2) or an ALLOCATE statement (6.2.2). Once associated, a pointer may appear as a primary in an expression anywhere a variable with the same type, type parameters, and shape may appear.

4. Page 2-9, line 4, replace "alias" with "pointer".

5. Page 4-6, line 12, before the sentence "Ultimately .....type." add

   A component may be a pointer to an object of intrinsic type, to an object of a previously defined derived type, or to an object of the type being defined.

6. Page 4-6, line 13, before the period, add "or pointers"

7. Page 4-6, move constraint on lines 37-38 to follow line 22.

8. Page 4-6, line 31-32, replace Rule 419 by

   R419 *component-def-stmt* is *type-spec* [,*component-attr-spec-list* ::] □
   □ *component-decl-list*

9. Page 4-6, following line 34 add

   R419.1 *component-attr-spec* is POINTER
   or ARRAY (*component-array-spec*)

   Constraint: No *component-attr-spec* may appear more than once in a given *component-def-stmt.*

   Constraint: A *type-spec* in a *component-def-stmt* may include the *type-name* of its containing *derived-type-def* only if the POINTER attribute is specified for that component.

   R419.2 *component-array-spec* is *explicit-shape-spec-list*
   or *deferred-shape-spec-list*

10. Page 4-6, move constraints on lines 35-36 to follow line 42.

(10)

11. Page 4-6, line 39, replace "*explicit-shape-spec-list*" with "*component-array-spec*"

12. Page 4-7, lines 1-2, replace "attribute .... specified" with "attribute, the POINTER attribute, or both are specified"

13. Page 4-8, line 8, add

    A derived type may have a component that is a pointer. For example,

```
TYPE REFERENCE
    INTEGER                          :: VOLUME,YEAR,PAGE
    CHARACTER(LEN=50)                :: TITLE
    CHARACTER,ARRAY(:),POINTER       :: ABSTRACT
END TYPE REFERENCE
```

    Any object of type REFERENCE will have the four fixed sized components VOLUME, YEAR, PAGE and TITLE, plus a pointer to an array of characters holding ABSTRACT. The size of this target array will be determined by the length of the abstract. The space for the target may be allocated (6.2.2) or the pointer component may be associated with a target in a pointer assignment statement (7.5.2).

    A pointer component of a derived type may have as its target an object of the type of which it is a component. For example,

```
TYPE NODE
    INTEGER                  :: VALUE
    TYPE(NODE), POINTER      :: NEXT_NODE
END TYPE
```

    A type such as this may be used to construct linked lists of objects of type NODE.

14. Page 4-9, after line 34, add,

    Where a component in the derived type is a pointer, the corresponding constructor expressions must evaluate to an object that would be an allowable target for such a pointer in a pointer assignment statement. For example, if the variable TEXT were declared (5.1) to be

(10)

CHARACTER, ARRAY(1:400), TARGET :: TEXT

and BIBLIO were declared

TYPE(REFERENCE) :: BIBLIO

the statement

```
BIBLIO=REFERENCE(1,1987,1, "This is the title of the referenced &
                                & paper", TEXT)
```

is valid and it identifies the ABSTRACT component of the object BIBLIO with the target object TEXT.

A constant expression cannot be constructed for a derived type containing a pointer component, since a constant value is not an allowable target in a pointer assignment statement.

15. Page 5-1, delete lines 31-32

16. Page 5-1, after line 33, add

or DEFERRED

17. Page 5-1, after line 35, add

or POINTER

18. Page 5-1, after line 37, add

or TARGET

19. Page 5-1, lines 47-48, replace with

Constraint: An object must not have more than one of the attributes: POINTER, DATA, PARAMETER

Constraint: An object must not have both the TARGET attribute and the PARAMETER attribute.

Constraint: An object must not have both the DEFERRED attribute and the DATA attribute.

20. Page 5-2, line 1, replace "ALIAS" with "POINTER or DEFERRED"

21. Page 5-2, line 3, replace "an ALIAS or ALLOCATABLE" with "a POINTER or DEFERRED"

22. Page 5-2, line 5, delete "ALIAS,"

23. Page 5-6, lines 21-22, replace "an alias object, an allocatable array," with "a pointer, a deferred object,"

24. Page 5-6, line 35, replace "an allocatable array" with "a pointer"

25. Page 5-7, line 5, replace "allocatable dummy arguments" with "dummy pointers"

26. Page 5-7, line 23, replace "an allocatable array and an alias array" with "a pointer to an array and a deferred array"

27. Page 5-7, line 28-29, replace with

```
REAL, DEFERRED :: C(:)          ! deferred array
REAL, POINTER :: D(: , :)       ! pointer to an array
```

28. Page 5-8, replace lines 16-24 with

5.1.2.4.3 Deferred-Shape Array. A **deferred-shape array** is a deferred array or a pointer to an array. An object declared with a *deferred-shape-spec-list* is allocatable. If it has the DEFERRED attribute it is a deferred array. If it has the POINTER attribute it may be used as a pointer or a pointer target.

A **deferred array** is a named array whose type, type parameters, name, and rank are specified in a type declaration statement, but

(10)

whose bounds, and hence shape, are determined when it is allocated by execution of an ALLOCATE statement (6.2.2). It must not be allocated more than once within its scoping unit.

A **pointer to an array** is a named array whose type, type parameters, name, and rank are specified in a type declaration statement, but whose bounds, and hence shape, are determined when it is associated with space by execution of a pointer assignment statement (7.5.2) or when space is allocated for the array target by execution of an ALLOCATE statement (6.2.2). It may be associated more than once within its scoping unit.

29. Page 5-8, replace lines 27-30 with

    The size, bounds, and shape of an unallocated deferred array or an unassociated or unallocated pointer to an array are undefined. No reference may be made to any part of such an array, nor may any part of it be defined. The upper and lower bounds of each dimension are those specified in the ALLOCATE statement or the pointer assignment statement when the array is associated with space.

30. Page 5-8, line 31, replace "allocated" with "associated".

31. Page 5-8, delete lines 33-36

32. Page 5-8, replace lines 37-39 with

    A pointer dummy argument may be associated only with a pointer actual argument. An actual argument that is a pointer may be associated with a nonpointer dummy argument. An array-valued function may declare its result to be a pointer to an array.

33. Page 5-9, line 27 and line 32, replace "allocation status" with "association status" (twice)

34. Page 5-9, line 35, replace "an automatic data object, or an alias" with "or an automatic data object"

35. Page 5-9, lines 40-47, delete

36. Page 5-10, before line 1, add

    5.1.2.7 POINTER Attribute. The **POINTER attribute** specifies that only the type, type parameters, rank, and name of the objects

(10)

declared in the statement are specified. The object called a pointer is empty, or disassociated. It must not be referenced or defined unless, as a result of executing a pointer assignment statement (7.5.2) or an ALLOCATE statement (6.2.2), it becomes pointer associated with a target object that may be referenced or defined. If the pointer is to have an array as target object, the pointer must be declared with a *deferred-shape-spec-list.* Examples of POINTER attribute specifications are

```
TYPE(NODE), POINTER          :: CURRENT, TAIL
REAL, ARRAY(:,:), POINTER    :: IN, OUT, SWAP
```

5.1.2.8 TARGET Attribute. The **TARGET attribute** specifies that an object declared in a declaration containing this attribute may appear as the target object in a pointer assignment statement (7.5.2), that associates a pointer with a target. Any object specified to have the POINTER attribute automatically acquires the TARGET attribute as well and does not require its explicit specification. Examples of TARGET attribute specifications are

```
TYPE(NODE), TARGET                 :: HEAD
REAL, ARRAY(1000,1000), TARGET :: A, B
```

5.1.2.9 DEFERRED Attribute. The **DEFERRED attribute** specifies that only the type, type parameters, rank, and name of the object declared in the statement are specified. The object called a deferred object is disassociated. It must not be referenced or defined unless, as a result of executing an ALLOCATE statement (6.2.2), it becomes associated. It may be associated only once within its scoping unit. It cannot be associated by the execution of a pointer assignment statement (7.5.2). If the deferred object is an array, it must be declared with a *deffered-shape-spec-list.* A deferred object may be specified to have the TARGET attribute. Examples of DEFERRED attribute specifications are

```
REAL, ARRAY(:,:), DEFERRED    :: WORK
CHARACTER(:), DEFERRED        :: TITLE
```

37. Page 5-10, line 22, replace "alias" with "pointer, target, deferred"

(10)

38. Page 5-11, line 24, delete "an alias name,"

39. Page 5-13, lines 5-7, delete "an alias object,", replace "an allocatable array" with "a deferred object, a pointer"

40. Page 5-14, lines 4-5, replace "allocatable arrays, alias objects" with "deferred objects, pointers"

41. Page 5-17, lines 18-19, replace "an alias object, or a deferred-shape array" with "a deferred object, or a pointer"

42. Page 5-18, line 6, Page 5-19, line 17,

    replace "an alias object, an allocatable array" with "a deferred object, a pointer"

43. Page 6-1, line 3, after "is defined." add

    "A reference to a pointer is permitted only if the pointer is associated with a target object that is defined"

44. Page 6-1, line 22, replace "alias variables (5.1.2.7), allocatable arrays (5.1.2.4.3)" with "deferred objects, pointers"

45. Page 6-3, lines 18-19, replace "allocatable arrays" with "deferred objects and pointer targets"

46. Page 6-3, line 20, delete "*array-*"

47. Page 6-3, line 25, replace rule R612 by

    R612 *allocation* **is** *name* [(*explicit-shape-spec-list*)]

48. Page 6-3, line 26, replace with

    Constraint: *name* must be the name of a pointer or deferred object.

49. Page 6-3, line 27 and line 31, delete "*array-*" (twice)

50. Page 6-3, line 29, replace "other array" by "other object"

51. Page 6-3, lines 35-37, delete "At the time ... allocatable array."

52. Page 6-3, line 45, replace "array" with "object"

(10)

53. Page 6-3, lines 46-47, delete "Allocating a currently ...ALLOCATE statement."

54. Page 6-3, line 48, replace "arrays" with "objects"

55. Page 6-4, line 1, line 3, and line 4 replace "array" with "object" (3 times)

56. Page 6-4, following line 2, add

    The pointer target may be referred to by way of the associated pointer. Additional pointer names may become associated with the pointer target or a part of the pointer target by pointer assignment. It is not an error to allocate a currently allocated pointer. In this case a new pointer target is created as required by the attributes of the pointer and any array bounds specified in the ALLOCATE statement. The pointer is then associated with this new target. Any previous association with a target is broken. If the previous target had been created by allocation it becomes inaccessible unless it can still be referred to by other pointer names that are currently associated with it.

    [X3J3 Note: This is an essential property of pointer allocation. It is necessary to allow lists to be created by allocating a new node by way of a working pointer. This is then attached to the list by pointer assignment as the target for the next % node in the current node of the list. This process is iterated usually in a conditional exit loop, as in the example in the Appendix C additions in this proposal.]

    At the beginning of execution of a function whose result is a pointer, the result pointer is disassociated. Before such a function returns it must associate a target with this pointer.

57. Page 6-4, after line 5, add

    An object with the DEFERRED attribute must not be allocated more than once within its scoping unit. A DEALLOCATE statement causes such an object to be disassociated. A DEALLOCATE statement causes a pointer to be disassociated from its current target.

58. Page 6-4, line 6, replace "*array-name*" with "*allocation-name*"

59. Page 6-4, line 10, replace with

    Each *allocation-name* must be the name of an object with either the POINTER or DEFERRED attribute

(10)

60. Page 6-4, line 17, replace "array" with "object"

61. Page 6-4, line 19, replace "array" with "object"

62. Page 6-4, line 21, replace "An allocatable" with " A pointer"

63. Page 6-4, lines 22 and 23, replace "array" with "object" (twice)

64. Page 6-4, lines 25-26, replace with

    Such allocated objects retain their association status at the execution of the RETURN or END statement.

65. Page 6-4, line 33, replace "*array*" by "*object*"

66. Page 6-4, line 38, after "array" add, "or a pointer target".

67. Page 6-7, line 21 through page 6-10, line 10, delete all

68. Page 6-10, Table 6.2, delete column 3 (Alias Array), change heading of column 5 to "Array Target", replace lines 32-33 with

pointer-assignment-stmt	No	No	Yes	No	No

69. Page 7-6, after line 27, add after title

    If a pointer is referenced as a primary in an expression, the associated target object is referenced. The type, type parameters, and shape of the primary are those of the current target. If the pointer is not currently associated with a target it may appear as a primary only as the actual argument of a procedure whose corresponding dummy argument is declared to be a pointer.

70. Page 7-7, line 39 and Page 7-9, line 20, replace "IDENTIFY" with "a pointer assignment"

71. Page 7-9, line 35, between the sentences, add

    If the variable is a pointer, it must be associated with a target object that is defined.

72. Page 7-19, after line 4, add

    If the variable is a pointer it must be currently associated with a definable target object whose type, type-parameters and shape are

conformant with the result of evaluating the expression. The result of the expression evaluation is assigned to the currently associated pointer target.

If the variable is of a derived type containing a pointer component, the expression must evaluate to a value of this type. Each of the values of the nonpointer components is assigned to the corresponding component variable, and each pointer component is associated with the corresponding pointer component variable.

73. Page 7-21, after line 3, add the following section and adjust the following section and rule numbers

7.5.2 Pointer Assignment Statement

R723 *pointer-assignment-stmt* is *pointer-name* => *target*

R724 *target* is *variable*

Constraint: The *pointer-name* must have the POINTER attribute. The target object must have one of the attributes TARGET or POINTER or it must be a sub-object of an object with one of these attributes.

Constraint: The *target* must be of the same type, type parameters, and rank as the pointer.

A pointer assignment statement associates a *pointer-name* with a target object. If *target* is itself a pointer then *pointer-name* is associated with the same object as *target*. If *target* is a pointer that is not currently associated, then *pointer-name* also becomes disassociated.

Any association with a target object the pointer may have had previously is broken.

In addition to pointer assignment, a pointer becomes associated with a target object by allocation of the *pointer-name*.

A pointer may not be referenced or defined unless it is associated with a target that may be referenced or defined.

The following are examples of pointer assignment statements.

```
PNTR_TO_CELL => FIRST_CELL
SIMPLE_NAME => STRUCTURE % SUBSTRUCT % COMPONENT
ROW => MAT2D(N, :)
WINDOW => MAT2D(I-1:I+1,J-1:J+1)
ROW => MAT2D(K,5:5+K)
EVERY_OTHER => VECTOR(1:N:2)
PATTERN => STRUCTURE_A(1:N) % ARRAY_B(1:M)
```

74. Page 9-5, lines 43-44, replace "an allocatable array not currently allocated, an alias object not currently alias associated" with "a pointer or deferred object not currently associated"

75. Page 9-13, after line 11, add

    If an input item is a pointer it must be currently associated with a definable target object. If an input item is a deferred object it must be currently allocated.

76. Page 9-13, after line 20, add

    If a derived type contains a pointer component, an object of this type may not appear as an input item, nor as the result of the expression evaluation in an input/output list.

77. Page 9-16, after line 16, add

    If the input item is a pointer, data are transferred from the file into the currently associated target object. If the input item is a deferred object, data are transferred from the file into the allocated object.

78. Page 11-2, after line 16, add a new paragraph

    If a procedure gains access to a pointer by host association the association of the pointer with a target that is current at the time the procedure is invoked remains current within the procedure. This pointer association may be changed within the procedure by allocation, deallocation, or assignment. When execution of the procedure completes, the pointer association that was current remains current, except where the associated target was declared within the procedure and is not saved. In this case the completion of the procedure causes the pointer association status of the host associated pointer to become undefined. Such a pointer may not be used in any way until its association status is re-established by deallocation, allocation, or assignment.

(10)

96. Page 13-3, lines 40-41, change "an allocatable array that has been allocated, an alias array that is alias associated" to "a deferred-shape array that has been allocated or associated"

97. Page 13-5, after line 4, add

    13.7.8 Association Status Inquiry Functions. The function ASSOCIATED with a single argument returns true if its argument is currently associated, and false if it is currently disassociated. The two-argument form is used only for pointers. It compares the arguments. If they refer to the same object the result is true; otherwise it is false. Two pointers are the same if they are associated with the same target.

98. Page 13-7, line 38, delete

99. Page 13-8, after line 26, add

    13.19.15 Association Status Inquiry Function

    ASSOCIATED (VIRTUAL_OBJECT, TARGET) association status or comparison

100. Page 13-12, lines 34-40, delete

101. Page 13-13, after line 38, add and renumber

    13.12.13 ASSOCIATED (VIRTUAL_OBJECT, TARGET)

(10)

Optional Argument.	TARGET
Description.	Returns the association status of its virtual object argument or indicates the virtual object is associated with the target.
Kind.	Inquiry function
Arguments.	
VIRTUAL_OBJECT	must be a pointer or deferred object, may be of any type
TARGET (optional)	must be a permitted pointer target
Result Type.	The result is of type Logical
Result Value.	
Case(i):	If TARGET is absent the result is true if VIRTUAL_OBJECT is currently associated and false if it is not.
Case(ii):	If TARGET is present, the VIRTUAL_OBJECT must be a pointer and the result is true if the pointer is currently associated with TARGET and false if it is not.
Example.	ASSOCIATED(CURRENT, HEAD) is true if CURRENT points to the target HEAD

102. Page 13-19, lines 13-15, change "It must not be an allocatable array that is not allocated or an alias array that is not alias associated" to "It must not be a pointer or deferred object that is not associated." Make the same change in the following places:

Page 13-20, lines 24-25
Page 13-20, lines 38-40
Page 13-21, lines 14-16

(10)

Page 13-22, lines 19-20
Page 13-25, lines 10-12

103. Page 14-1, line 15, delete "as an IDENTIFY subscript,""

104. Page 14-2, lines 10 and 14 change "allocatable" to "pointers" (twice)

105. Page 14-3, lines 7-8, delete

106. Page 14-3, line 30. between "association" and "or" add ", pointer association". Delete the second "by".

107. Page 14-3, line 32, change "four" to "three"

108. Page 14-3, line 33, delete "alias association,"

109. Page 14-3, lines 35-36, delete "Alias association ... unit."

110. Page 14-4, lines 18-37, delete

111. Page 14-5, Table 14.2, lines 1-33, delete second column, delete lines 23-26, change title to "Summary Comparison of Use and Host Association"

112. Page 14-5, after line 33, add new section and renumber

14.7.2 Pointer Association. Pointer association between a pointer and a permitted target allows the target to be referred to by way of the pointer. A pointer may be associated with different targets or no target at different times during execution of a program.

Pointer association is established by allocation (6.2.2) or pointer assignment (7.5.2). Pointer association is broken and a pointer disassociated from any target by deallocation (6.2.3) or assignment to an already disassociated pointer.

The pointer association status of a pointer becomes undefined if the associated target ceases to exist (12.4.1.1) (11.2.2).

A pointer that is currently associated with a definable target is a variable and it becomes defined or undefined according to the same rules as for a variable (14.8).

(10)

113. Add the following section notes

C.4.4 Pointers This standard introduces pointers as names that can dynamically change their association with a target object. In a sense, a normal variable is a name with a fixed association with a specific object. A normal variable name refers to the same storage space throughout the lifetime of a variable. A pointer name may refer to different storage space, or even no storage space, at different times. A variable can be considered to be a descriptor for space to hold values of the appropriate type, type parameters and array rank such that the values stored in the descriptor are fixed when the variable is created by its declaration. A pointer can also be considered to be a descriptor but one whose values may be changed dynamically so as to describe different pieces of storage. When a pointer is declared, space to hold the descriptor is created, but not the space described, whereas for a variable, both are created.

A derived type may have one or more components that are defined to be pointers. It may have a component that is a pointer to an object of the same type as that being defined. This "recursive" data definition allows dynamic data structures such as linked lists, graphs and trees to be constructed. For example

```
TYPE CELL                          ! define a "recursive" type
    INTEGER                  :: val
    TYPE(CELL), POINTER :: next_cell
END TYPE CELL

TYPE(CELL), TARGET       :: head
TYPE(CELL), POINTER      :: current, temp ! declare pointers
INTEGER                  :: ioem, k

head%val=0
current => head                    ! current points to head of list
DO
    READ(*,*,iostat = ioem)k       ! read next value if any
    IF(ioem.NE.0)EXIT
    ALLOCATE(temp)                 ! create new cell each iteration
    temp%val = k                   ! assign value to cell
    current%next_cell => temp      ! attach new cell to list
```

```
    current => temp                    ! current points to new end of list
END DO
```

A list is now constructed and the last linked cell contains a disassociated pointer. A loop can be used to "walk through" the list.

```
current => head
DO
    WRITE(*,*) current%val
    IF(.NOT.ASSOCIATED(current%nextcell)) EXIT
    current => current%nextcell
END DO
```

C.5.2 The POINTER Attribute The pointer attribute is specified if a pointer is declared. The type, type parameters, and rank that must be specified at the same time determine the characteristics of the target objects that can be associated with the pointers declared in the statement. An obvious model for interpreting declarations of pointers is that such declarations create for each name a descriptor. Such a descriptor includes all the data necessary to describe fully and locate in memory an object and all subobjects of the type, type parameters, and rank specified. The descriptor is created empty; it does not contain values describing how to access any actual memory space. These descriptor values will be filled in when the pointer is associated with actual target space.

The following example illustrates the use of pointers in an iterative algorithm.

```
PROGRAM DYNAM_ITER
    REAL,ARRAY(:,:),POINTER :: A, B, SWAP! Declare pointers
    ...
    read (*,*) N, M
    ALLOCATE (A(N,M), B(N,M))              ! Allocate pointers
    read values into A
    ...
ITER:DO
    ...
    ! Apply transformation of values in A to produce values in B
    ...
```

```
    IF (converged) EXIT ITER
    ! Swap A and B
    SWAP => A; A => B; B => SWAP
END DO ITER
...
END
```

C.5.3 The TARGET Attribute. The TARGET attribute is specified for any object that may, during the execution of the program, become associated with a pointer. This attribute is defined entirely for optimization purposes. It allows the processor to assume that all objects not explicitly declared as targets may be referred to only by way of their original declared name. In particular, it means that implicitly-declared objects may not be used as pointer targets. This will allow a processor to perform optimizations that otherwise would not be possible in the presence of certain pointers.

The following example illustrates the use of the TARGET attribute in an iterative algorithm.

```
PROGRAM ITER
    REAL,ARRAY(1000,1000),TARGET    :: A,B
    REAL,ARRAY(:,:),POINTER         :: IN,OUT,SWAP
    ...
    read values into A
    ...
    IN => A                 ! Associate IN with target A
    OUT => B                ! Associate OUT with target B
    ...
ITER:DO
    ...
    ! Apply transformation of IN values to produce OUT
    ...
    IF (converged) EXIT ITER
    ! Swap IN and OUT
    SWAP => IN; IN => OUT; OUT => SWAP
END DO ITER
...
END
```

C.5.4 The DEFERRED Attribute. The DEFERRED attribute is specified for an object whose shape is not declared, but is specified when the object is allocated. A deferred object must not be allocated more than once in its scoping unit. This attribute is defined entirely for optimization purposes. It allows the processor to treat a deferred object in much the same manner as a fully declared object. A deferred object must not be a pointer target unless it also has the TARGET attribute.

The following example illustrates the use of the DEFERRED attribute.

```
PROGRAM TAILORED
    REAL, ARRAY(:,:),DEFERRED        :: WORK
    CHARACTER(:), DEFERRED           :: TITLE
    INTEGER M, N, L
    ...
    read values into N, M, L
    ALLOCATE (WORK(N,M), TITLE(L))
    ...
    ! calculations using WORK and output using TITLE
    ...
END
```

C.6.4 Pointer Allocation and Association The effect of ALLOCATE and DEALLOCATE (when applied to pointers) and pointer assignment is that they are interpreted as changing the values in the descriptor that is the pointer. An ALLOCATE is assumed to create space for a suitable object and to "assign" to the pointer the values necessary to describe that space. A DEALLOCATE breaks the association of the pointer with the space. Depending on the implementation, it could be seen as setting a flag in the pointer that indicates whether the values in the descriptor are valid, or it could clear the descriptor values to some (say zero) value indicative of the pointer not currently pointing to anything. A pointer assignment copies the values necessary to describe the space occupied by the target into the descriptor that is the pointer. Descriptors are copied, values of objects are not. IF PA and PB are both pointers and PB is currently associated with an object C, then

```
PA => PB
```

results in PA also being associated with C.

The standard is defined so that such associations are direct and independent. A subsequent statement

PB => D

or ALLOCATE(PB)

or DEALLOCATE(PB)

has no effect on the association of PA with C, only with the association of PB.

The basic principle is that ALLOCATE, DEALLOCATE and pointer assignment primarily affect the pointer rather than the target. ALLOCATE creates a new target but other than breaking its connection with the specified pointer it has no effect on the old target. Neither DEALLOCATE nor pointer assignment have any effect on targets. A given piece of memory that was allocated and associated with a pointer will become inaccessible to a program if the pointer is deallocated and no other pointer was associated with this piece of memory. Such pieces of memory may be reused by the processor if this is expedient. However, whether such inaccessible memory is in fact reused is entirely processor dependent.

C.7.3 Pointers in Expressions A pointer is basically considered to be like any other variable when it is used as a primary in an expression. If a pointer is used as an operand to an operator that expects a value the pointer will automatically deliver the value contained in the space currently described by the pointer, i.e. the value of the target object currently associated with the pointer. In value-demanding expression contexts pointers are dereferenced.

C.7.4 Pointers on the Left Side of an Assignment A pointer that appears on the left of an intrinsic assignment statement also is dereferenced and is taken to be referring to the space that is its current target. The assignment statement is, therefore, the normal copy of the value of the right-hand expression into this target space. All the normal rules of intrinsic assignment hold; the type, type parameters, and array shape of the expression result and the pointer target must agree.

Note that if the object on the left of an intrinsic assignment is of a derived type which contains a pointer component, the assignment will copy the "value" of the corresponding pointer component in the expression. That is, the values of the descriptor will be copied. For intrinsic assignment of derived types, non-pointer components are assigned and pointer components are pointer assigned. Dereferencing is applied only to entire scalar objects, not selectively to pointer subobjects.

For example, if a type such as

```
TYPE CELL
    INTEGER :: val
    type(CELL), POINTER :: next_cell
ENDTYPE
```

and objects such as

```
type(CELL), TARGET  :: head
type(CELL), POINTER :: current
```

exist, a linked list has been created attached to HEAD and the pointer CURRENT allocated to associate space, statements such as

```
current = head
current = current%next_cell
```

cause the contents of the CELLs referenced on the right to be copied to the CELL referred to by CURRENT. In particular, the left-hand side of the second statement causes the pointer component in the CELL, CURRENT, to be selected. This pointer is dereferenced since it is in an expression context to produce the target's integer value and a pointer to a CELL that is contained in the target's CURRENT%NEXTCELL component. The right-hand side causes the pointer CURRENT to be dereferenced to produce its present target, space to hold a cell (an integer and a cell pointer). The integer value on the right is copied to the integer space on the left and the pointer components are pointer assigned (the descriptor on the right is copied into the space for a descriptor on the left). When a statement such as

```
current => current%nextcell
```

is executed, the descriptor value in CURRENT%NEXTCELL is copied to the descriptor named CURRENT. In this case CURRENT is made to point at a different target.

In the intrinsic assignment statement, the space associated with the current pointer does not change but the values stored in that space do. In the pointer assignment statement, the current pointer is made to associate with different space. Using the intrinsic assignment causes a linked list of CELLS to be moved up through the current "window"; the pointer assignment causes the current pointer to be moved down through the list.

C.9.11 Pointers in an Input/Output List. Data transfers always involve the movement of values between a file and internal space. A pointer as such cannot be read or written. A pointer may, therefore, appear as an item in an input/output list if it is currently associated with a target that can receive a value (input) or can deliver a value (output). A derived type object with one or more pointer components must not appear as an item in an input/output list because the value of a pointer component is the descriptor for a location in memory. As such, this has no processor-independent representation external to the processor.

C.11.3 Pointers in Modules. A pointer from a module program unit may be accessible in a procedure via use association. Such pointers have a lifetime that is greater than targets that are declared in the procedure, unless such targets are saved. Therefore, if such a pointer is associated with a local target, there is the possibility that when the procedure completes execution, the target will cease to exist leaving the pointer "dangling". This standard considers such pointers to be in an undefined state. They are neither associated nor disassociated. They must not be used again in the program until their status has been reestablished. There is no requirement on a processor to be able to detect when a pointer target ceases to exist.

114. Page C-16, line 49, change "ALLOCATABLE" to "POINTER"

115. Add more section notes:

C.12.4 Dummy Arguments as Pointers. If a dummy argument is declared to be a pointer it may be matched only by an actual argument

(10)

that also is a pointer, and the target object characteristics of both arguments must agree. A model for such an association is that descriptor values of the actual pointer are copied to the dummy pointer. If the actual pointer has an associated target, this target becomes accessible via the dummy pointer. If the dummy pointer becomes associated with a different target during execution of the procedure, this target will be accessible via the actual pointer after the procedure completes execution. If the dummy pointer becomes associated with a local target that ceases to exist when the procedure completes, the actual pointer will be left dangling in an undefined state. Such dangling pointers must not be used.

C.13.1 The ASSOCIATED Function. The ASSOCIATED intrinsic function can be used to test whether a deferred object has been associated or whether a pointer is associated with a target. The one-argument form is used for this purpose. In the two-argument form, the ASSOCIATED function tests whether the pointer first argument is associated with the space that is referred to by the second argument. The values shared in the two descriptors are compared. In most cases, it will be used to test if two pointers are associated with the same target.

116. Page C-22, line 23, change the title to "Automatic and Deferred-Shape Arrays"

117. Page C-22, line 25, change "allocatable" to "deferred-shape"

118. Page C-22, line 26, change "ALLOCATE and DEALLOCATE" to "ALLOCATE, DEALLOCATE, and pointer assignment"

119. Page C-22, line 29, change "ALLOCATABLE" to "POINTER"

120. Page C-23, line 14, delete

121. Appendix H (The Glossary) Delete definitions for **alias, alias association, and parent of an alias.** Add the following:

pointer (5.1.2.7). A descriptor for an object of the declared type, type parameters, and rank. A pointer is empty until it becomes associated with a target object by the execution of an ALLOCATE statement (6.2.2) or a pointer assignment statement (7.5.2). Once associated, a pointer may appear as a primary in an expression anywhere a variable with the same type, type parameters, and shape may appear.

(10)

target (5.1.2.7). An object that may be accessed by a pointer. Any dynamic object is a permitted target. A static object may be a target only if it is declared with a TARGET attribute (5.1.2.8).

122. Page H-1, line 13, replace "A named array" with "A named, deferred-shape array"

123. Page H-1, line 29, change "whether allocatable, whether an alias" to "whether a pointer or a target, whether deferred"

124. Page H-2, line 7, change definition to "An allocatable array or a pointer to an array."

125. Page H-4, lines 27-29, replace "An assumed-size array or an explicit-shape array" with "An explicit-shape array, an assumed-size array, or a pointer to a sequence array (which may be allocated)"

5 Proposal

That the above pointer facility be added to Fortran 8X by amending S8.108 as indicated.

95

(11)

To: X3J3

From: Lloyd Campbell

Subject: Suggested Edits to S8.104 (and S8.108)

Notes: Edits followed by a "(pc no.)" were instigated by that public comment number. S8.108 page and line numbers are in parentheses at the end of each item.

1. Page 1-3, line 17: Change "are also" to "also are". (as in S8.108 at p. 1-3/17)

2. Page 2-4, line 4: Add "(4.4.1)" after "definition". (pc 340.26)(2-3/32)

3. Page 5-2, line 20: Add "(7.1.6.3)" after "specification-expr". (pc 340.48) (5-2/18)

4. Page 5-10, line 4+: Add new section:

   5.1.2.8 ALLOCATABLE Attribute. The ALLOCATABLE attribute specifies that the objects declared in the statement are allocatable arrays. Such arrays must be deferred-shape arrays whose shape is determined when space is allocated for each array by the execution of an ALLOCATE statement (6.2.2).
   (pc 338.60) (5-9/47+)

5. Page 7-2, line 9+: Add "Constraint: A defined-unary-op must not contain more than 31 letters and must not be the same as any intrinsic-operator or logical-literal-constant."
   (should repeat constraint from p. 3-4 lines 8-9)
   (repeat 3-4/6,7 at 7-2/6+)

6. Page 7-19, line 30: Change "7.11" to "7.9". (7-19/49)

7. Page 8-6, lines 45-46: "including, if necwssary ... conversion (Table 7.11)" should be in small font and "7.11" should be "7.9". (pc 338.65) (8-7/18,19)

8. Page 9-6, line 6: Add "The file must be an external file." (pc 350.28.17) (9-6/8)

9. Page 9-15, line 23+: add "If no format or namelist-group-name is specified, unformatted data transfer is established." (pc 350.28.18, redundant but nice) (9-15/7+)

10. Page 9-21: Move lines 39-44 to page 9-22 line 4+ and renumber section to 9.6.3. (move 9-21/32-36 to 9-21/45+)
    Page 9-21, line 45: Change "9.6.1.21" to "9.6.2". (pc 350.28.20) (9-21/37)

11. Page 10-4, line 12: Change "such" to "a". (pc 380.39, to include both kinds of reversion) (10-4/10)

12. Page 11-3, line 4: Add sentence "The accessed entities have the same attributes as in the module." (pc 350.28.24) (11-3/4)

13. Page 12-4, line 5: Insert sentence "If an external procedure name or a dummy procedure name is used as an actual argument, it must appear in an EXTERNAL statement or must be declared to be a procedure by an interface block in the scoping unit." (pc 350.28.25) (F77 rule modified by interface block exception) (12-3/48+)

(11)

14. Page 13-20, line 34: Change "X * Y" to "DBLE(X) * DBLE(Y)". (pc 350.28.33) (13-20/19)

15. Page C-23, line 15: Change title "Variance from the Mean" to "Sum of Squared Residuals". (isn't really the variance) (C-28/8)

16. Page F-8, line 14: Delete "array-". (pc 380.69) (F-8/9)

17. Page F-2, line 4: Change letter "I" to digit "1". (pc 382.64) (F-1/44)

18. Page F-27, line 20: Change "defined" to "undefined". (pc 380.75) (F-26/30)

19. Page 2-6, lines 43-44: Change "Statements" to "statements" four times and change "Type Declarations" to "type declarations". (pc 167.21) (2-5/43,44)

20. Page 3-1, line 16: Change "delimited character" to "character constant". (pc 167.28) (3-1/16)

21. Page 3-2, line 29: Change "These" to "Lexical tokens". (pc 167.34) (3-2/32)

22. Page 4-2, line 38: After "types" add "(7.2.1). (pc 167.45) (4-2/33)

23. Page 4-6, line 26: After "type-spec" add "(5.1)". (pc 167.64) (4-6/33)

24. Page 4-7, line 20+: Add "An example of declaring a variable LINE_SEGMENT to be of type LINE is:". (pc 167.67) (4-7/37+)

25. Page 5-10, line 27: Add "allocatable," after "optional,". (pc 167.110) (5.10/22)

26. Page 12-9, line 33: Make "RECURSIVE" bold. (to get it in index) (pc 63.9) (12-9/36)

(12)

110-CDB-1
September 24,1988

From : Carl Burch

To : X3J3

Subj : A Language-based Design for Portable Data Files

This is the second draft of this proposal. The first was on the table in Jackson and many of you were asked to comment on it then. Your comments (and those you passed it to) have been considered and this draft reflects them, particularly with regard to the language bindings and the handling of translation failures.

The more comments, the better.

Carl

(12)

A Language-based Design for Portable Data Files

Carl Burch

Hewlett-Packard Company
19447 Pruneridge Avenue
Cupertino, CA 95014

ABSTRACT

Currently data files to be accessed remotely from dissimilar systems must be transformed to the local language processors' file format and data representation; a process that has changed little since punch cards were the main form of portable data files. A proposal is presented for languages to use the data typing information available to the runtime library to make these data transformations before the data is transferred to the file or the user's variables. By specifying exactly one binary format for each basic data type (integer, real, logical, etc) and a file format that is portable between record-oriented and stream-oriented file systems, we can establish file formats that will be usable on practically all current systems.

This is a rough draft of a proposal for portable data file access. Earlier drafts have been reviewed by members of several ANSI language committees. It is provided as a request for your comments.

Background

At the May 1988 meeting of the ANSI standards committee for Fortran[1] (X3J3), Mr. John Swanson (President of Swanson Analysis Systems, Inc.) was kind enough to address X3J3 on his views of the draft Fortran 8x standard. One of his concerns was that there is no provision in the draft standard for portable access to data files, particularly in the case of networks of dissimilar machines that offer relatively transparent access to remote files (e.g., NFS (Sun), RFA (HP), RFS(AT&T)). While his primary concern was for binary data files, I have addressed character (called herein "formatted" to disambiguate them from binary files containing only character data) files as well, since they also can suffer portability problems.

It should be noted that the problem of porting data files between systems existed even when the primary mode of transport was magnetic tape. Today, the same concerns exist with the type of non-transparent copying of remote files that is more common between machines running widely dissimilar operating systems (e.g., FTP (ARPA), NFT (HP)).

Objectives

The objectives of this proposal are :

- Provide for transparent access to data files by both remote and local copies of the same source program, for systems similar enough (e.g. UNIX[2]-derived) to have network file access by naming convention (as opposed to a file transfer command or program).

[1] The X3J3 committee voted in 1984 that Fortran had passed into the language as a proper noun and no longer was required to be in all capitals when used generically. FORTRAN 77 specifically is spelled exactly that way in the X3.9-1978 Standard.

[2] UNIX is a registered trademark of AT&T.

- Allow copying of data files written on remote machines to the local machine for input to the same source program compiled locally, for file systems without transparent remote file access.
- Allow access to data files by different programs (possibly written in different languages) using a common file format known to each of the programs, which may also be used on dissimilar remote machines.

Discussion

I see an opportunity for languages that incorporate an I/O library (Fortran, Pascal, COBOL, etc) and (more tentatively) those with standard I/O modules (Ada, and to some extent C) to use the data typing information available to the runtime library to make these data transformations before the data is transferred to the file or the user's variables. By specifying exactly one binary format for each basic data type (integer, real, logical, etc) and a file format that is portable between record-oriented and stream-oriented file systems, we can establish file formats that will be usable on practically all current systems.

Limitations

We will have to specify the capability to read and write files in units of eight-bit bytes. This alone is usually enough to preclude this sort of proposal being included in major language standards. On the other hand, it is near-universal enough to make a collateral standard an attractive alternative. I am interested in hearing opinions on whether my assumption is correct, that this proposal cannot be a more general standard - after all, Ada specifies the use of ASCII for a character set.

We will avoid requiring full record lengths for direct-access files to not be rounded up (and padded) to match some multiple of bytes (usually the machine's word length). There remains an option to later define file structures covering file systems that do not read and write in multiples of bytes.

Related Standards

Sun Microsystems' XDR (External Data Representation) addresses the Remote Procedure Call (RPC) interface between cooperating processes, with a notation that it may also be used with a stream file model. It's drawbacks for data file migration are that it requires explicit coding in both the sender and receiver programs and it is defined only in C (i.e., I haven't found any other language binding).

Apollo Computer's NDR (Network Data Representation?), part of their Network Computing System (NCS), is similar except that it uses two or three intermediate formats for each data type. Each machine translates its native data to the nearest of the standard types (for transmission) and from each of the standard types (on receipt). I have failed to find any provision for data file use of NDR. It is a kind of programming language as well, using a compiler to generate code from a C-like description of the data to be passed.

The ISO X.409 standard is generally similar to NDR, but uses the self-describing ASN.1 data format. The ASN.1 data format hopes to provide more portability across dissimilar systems and more general send/receive code. It is fairly verbose, approaching the data expansion of ASCII.

None of the above approaches to standardizing data item representations address file formats to support remote file access on dissimilar file systems, especially not direct (random) access.

Language Bindings and Implementation Notes

Fortran

The OPEN statement will require a new specifier (keyword) that tells the runtime library that the file being opened is to be maintained in the portable format. The INQUIRE statement tends to include all the OPEN's specifiers, so it will also need the same specifier to inquire about the format of a (presumably open) file. While I do not have a strong preference for the syntax to be used, my favorite (so far) is to overload the FMT= specifier of the READ or WRITE. In that case, the argument must be a character expression that evaluates to either

'STANDARD' or 'NATIVE'. Another alternative would be to make the argument a logical expression and call it something like STANDARD=<logexp>.

The IOSTAT= and ERR= specifiers will be used for translation error reporting.

For Unformatted Direct files, the RECL= specifier must be interpreted as being in bytes (when FMT='STANDARD'), not processor-defined words as allowed by the FORTRAN 77 Standard. This is so that the same source program will be interpreted the same on varying implementations.

Note that in the discussion below, only Fortran seems to have much use for the concept of direct access to a formatted file. An issue here is whether Direct Formatted files should also be padded to a multiple of four bytes?

Fortran also seems to be the only language to allow Sequential access to DIRECT-organized files as a common extension to the official ANSI/ISO standard. Is this a requirement for this standard?

The requirement to translate data items on the basis of their declared data type implies that lying to the compiler about the data to be stored will cause nonportable results. The major cause of this sort of practice was the lack of a character data type in Fortran 66, forcing programmers to use Hollerith strings and store characters in numeric variables. This will work only when the declared data type happens to be implemented on the local system the same way as the standard required - i.e., there is no transformation required before the data is written or after it is read. A program that stores eight characters in a DOUBLE PRECISION variable on a machine that uses 64-bit IEEE will be able to read and write such a variable successfully - but the HP1000/VAX/IBM 370 that tries to access the data therein will hash those characters into the local floating-point format. This will severely limit the portability of data files from Fortran 66 programs like Spice that use this trick. Similarly, programs that use EQUIVALENCEs to an array to construct a Pascal record (or C struct) will be disappointed in the results of writing out the record as a whole (homogeneous) array and then reading it on a different machine.

COBOL/RPG

A new keyword STANDARD is added that parallels the EXTERNAL and GLOBAL keywords in the File Description (FD).

COBOL and RPG are typically implemented with no awareness on the part of the I/O library of individual fields of a record. This is not possible with the design that each data item must be translated to the standard equivalent of its processor-dependent value - each item of fundamental type must be handled individually. On the other hand, the editing that is performed in the MOVE statement can be extended to the translations necessary for the portable data file format.

The FILE STATUS clause will be used for translation error reporting.

This proposal makes no provision for COBOL's indexed files. RELATIVE files are discussed below as direct access files.

Pascal

A new keyword STANDARD will be introduced that precedes the FILE keyword. Similarly, a new type "standard_text" will correspond to "text".

Pascal also is typically implemented with no awareness on the part of the I/O library of individual fields of a binary file record. The new keyword will switch in library code to expand the records into their components of fundamental type (See "Structures" below). "Text" records are written one item at a time, however, which makes the concept less alien.

One problem peculiar to Pascal is that none of the Pascal standards include a means of handling I/O errors. Perhaps this standard could provide a minimal facility like the C library's "errno" to allow the reporting of translation errors.

(12)

C

The binding to C will be via the fopen(3) library call. The fopen(3) call's "type" argument will be extended to add a new letter 's' to make the format designation and a 'd' to specify that the file will be accessed directly. The 'b' value defined in the X3J11 draft suffices to define the other bit of information needed to specify the format.

The C library's "errno" values defined in <errno.h> will be extended to include ETRANS for the reporting of translation errors.

There are several significant problems with using this standard in C, the most glaring being that it is designed to support the record-oriented I/O model of Fortran/COBOL/Pascal on both record- and stream-oriented underlying file systems, not C's stream-oriented model on record-oriented systems. The best rationale I can give for C's inclusion would be that it allows access to the files written by programs in other languages, for programs willing to do their I/O in logical records, not seeking around in a file with all bytes being created equal.

Extending the file open routine will allow the library to do the translations necessary, but what about maintaining the record structure of the file? Providing the record structure is as necessary for communicating with programs in another language as it is to solve C's classic problem of garbage left before the end-of-file on record-oriented file systems. We can provide a new library call to mark the end of record (which will write the length word both before and after the user's data for sequential binary records, pad direct records, etc), but will anyone use it? An earlier draft of this paper stipulated an interface at the open(2) level as well, but making read(2) and write(2) be library calls (to support the translations necessary) did too much violence to C for there to be any hope of making it work.

Whether this model will work with C's style of charging the user with designing and maintaining the file formats is hard to anticipate. There is no technical reason why it won't, but programming style is as difficult to revise as any other human behavior. One of the main strengths of this proposal with regard to the other languages is that very little revision is required in coding style or rewriting of existing programs - just add the keyword and recompile. This is not true of C, and I wonder if the idea is an acceptable fit to C at all. As a minimum, the standard will serve to allow formatted files to be accessed across systems that normally use different CR/LF sequences to break lines.

Ada

Ada's standard I/O packages will be replaced by a STD_IO package with the same interfaces, but reading and producing files in the standard formats. Translation errors will raise a new exception defined by the STD_IO package.

File and Record Formats

The file formats we need are all designed to support the record-oriented file access model on byte-stream file systems - if we provide the means of detecting the end of line (EOL) and end of file (EOF), stream access on record-oriented systems is relatively easy. We need logically separate representations for files that will be accessed (all the combinations of) sequentially and directly; and that contain binary and formatted (in this case, ASCII) data.

Formatted Files

All data is represented in ISO 646 (eight-bit ASCII) characters.

Sequential Formatted

Logical records are separated by newline characters (ASCII 10 decimal) on byte stream-oriented file systems; variable record-length files are used on record-oriented systems. Note that this assumes transparent access to remote files will be possible only among dissimilar systems that have byte-stream file systems - a logical conclusion considering the varying naming conventions for dissimilar file systems. Current remote file access systems have to be cued by the name that the file requested is remote, hence they only work if the naming conventions are compatible.

Direct Formatted

Fixed record lengths are required, and must be specified in bytes. Logical records are not separated by any delimiter on stream file systems. Records written that do not fill the record to the specified record length are padded on the end with blanks (ASCII 32 decimal).

Binary Files

Data items are represented as below in "Binary Data Item Representation", and are not separated or padded internally (e.g., for alignment).

Sequential Binary

Each logical record is preceded and followed by a 32-bit positive integer which holds the number of data bytes in that record (not to include the size of the length words themselves). The trailing length word is necessary to efficiently support the BACKSPACE operation. On record-oriented file systems, logical records may cross "physical" record boundaries with no extra bytes being inserted. The data is read until the number of bytes in the record is exhausted or the request is satisfied.

End of file is indicated by a 32-bit length word with all bits set (two's complement -1). The file system EOF is not used, avoiding problems with trailing garbage on record-oriented file systems.

Direct Binary

Logical records are not separated by any delimiter on stream file systems. The record length specified (explicitly in Fortran, otherwise implied by the record description) is rounded up to be a multiple of four bytes (e.g, a system given a record length of five, would create a file with 8-byte records). Records written that do not fill the record to the specified record length are padded on the end with ASCII nulls (0 decimal).

Binary Data Item Representation

The proposals below represent my idea of maximal use of industry standard data types. A more "standards" approach would be to employ (a subset of) the ASN.1 self-describing data formats and just accept the performance overhead in order to maximize portability. As the ASN.1 standard defines multiple encoding rules for differing implementations, we would have to choose one for each class of data type. The concept of defining only one data format for each basic type is crucial in order to determine what size integer to write for literal constant values and to deal with varying definitions of purposely vague declarations like REAL or DOUBLE PRECISION. Either data item representation (or others) could be used with the file formats above.

All data items are written into files with the high-order bits first (i.e., not byte-swapped). Note that the requirement for conversions on some systems implies that abuses like putting Hollerith character strings in logicals, etc. will not be supported. See the section on Fortran above.

Integers

Thirty-two bit two's complement format.

Reals

Sixty-four bit IEEE 754 format. This format has an exponent range of ±308 (more on the small end if denormalized numbers are supported), and over 16 decimal digits of precision.

Another alternative is to adopt all three basic IEEE 754 data formats. Since in most languages real constants define their type by the letter starting the exponent, the main problem (of the two cited above) with using the IEEE formats for the real type is that most language standards define any number of rather amorphous real data types - except three. Fortran has REAL and DOUBLE PRECISION, Pascal only real, C float/double (ANSI C adds long double), Ada a parameterized scheme that works off the user's precision specification. The latter maps well into the IEEE versions, as both the user's data definition and the IEEE standard include precision and exponent range requirements. The other languages at most specify an ordering of their type's precision, not the values in any rigorous sense. This cavalier treatment in the language standards has been a particular pain to those doing mathematical software in Fortran,

particularly with respect to supercomputers that define REAL to be the same 64-bit size that most other machines use for DOUBLE PRECISION. Even these machines tend to have only two real formats supported in hardware. If I had to choose two formats only, the choice would not please supercomputer users since I would have to observe that far more machines and users expect 32 and 64-bit reals than 128.

As the goal here is to support portably the main thrust of computing, I propose to stay with the one data type, one format rule. Giving up the 128-bit reals is the supercomputer's contribution to the cause of portability, a counterpoint to the minicomputer's dismay at losing part of the performance advantage of 32-bit reals.

Complex

Two sixty-four bit IEEE format reals, real part followed by imaginary.

Logical/Boolean

One byte, zero if false and all ones (Hexadecimal FF)if true.

Packed Decimal

As far as I know, there is no official standard for Packed Decimal data. If COBOL or RPG are supported, the de facto standard for BCD will be needed. This includes four-bit digits with the values 0-9 (decimal) and the values for the sign digit (Hexadecimal D for negative and hex C for positive). The sign digit is the low-order (rightmost) digit. Data items with an even number of digits specified will be represented in files as one digit longer, adding a zero digit to the left so that the value (with sign) will evenly fill a multiple of eight-bit bytes.

Character

One byte per ISO 646 character, left to right, with apologies to our Japanese and Chinese friends. Perhaps when there is a firm ISO standard for large character set representation a multi-byte type can be added.

Arrays

Arrays are supported, as if the same number of items were presented one at a time. Whether column-major or row-major order is used for multiple-dimensioned arrays is a likely source for theological debate. If other languages are not interested in the idea, column-major order clearly would be adopted for Fortran.

Pointers

Pointers clearly must be barred from being written to any external file.

Structures

Structures (e.g., Pascal RECORDS, C structs, etc.) are a thorny problem. If there were no features like variant records in Pascal ("unions" in C), they could be "exploded" into their component parts down to items of the intrinsic data types and then converted and packed into the output record (reverse on input). Variant records present an ambiguity to the compiler - which variant is the real data type? Even when a tag field is present (in Pascal), is it valid to determine the variant? The alternatives I see are to allow them only if no variants are defined or to preclude structures altogether (requiring the user to expand them to their items of native type). As barring structures altogether would severely hamper languages like C where most of the code involves structures or languages like COBOL and Pascal where I/O itself is defined in terms of structures, I suggest that only variant records not be allowed. "Not be allowed" in this case means that the user will have to explicitly select out the fields (of intrinsic type or non-variant aggregates) one at a time on the I/O list, not just put the name of the record by itself on the I/O list.

Enumerations

Enumerated types are represented as a 32-bit non-negative integer, with the values starting with zero assigned to the enumeration values in the order of their declaration, increasing by one for each value.

Sets Sets are represented one bit per possible member, left justified in the fewest number of bytes that will hold the number of possible members. Any trailing bits left undefined will be written

12

as zero. The order of the bits is left to right in their cardinal order (increasing comparison order for integer or character sets, order of declaration for enumerated types). "Infinite" sets (such as SET OF INTEGER) are barred. Pascal SET OF CHAR (or equivalent) must be allowed.

Translation Failures

What if there is no plausible translation from the native data type to the type specified to be written on the file? This can occur when a real is written that requires an exponent outside the range of the IEEE 754 64-bit type (i.e. ±308). Another example is an EBCDIC machine trying to translate a character that does not exist in ASCII. These cases will be reported to the user as described in each languages' notes above. In particular, exponent overflows and underflows on reals will not be rolled to infinities or zero, respectively. The general rule is that the translation must be exact except for (possibly) the low order bits of a real's mantissa, or an error is reported. While the above addresses output, note that translation failures can also occur on input.

Acknowledgments

Walter Underwood (HP Software Development Environments) and Jason Zions (HP Colorado Networks Division) both provided descriptions of the remote file access systems that were the basis of the Related Standards section. The other HP representatives on the various ANSI language committees (Sue Meloy, Pat Mayekawa, and Julia Rodriguez) and many other members of HP's compiler laboratory were instrumental in helping my understanding of their languages' problems and approach to file I/O. Matt Yamamoto was particularly helpful in understanding the (considerable) problems of C doing I/O on record-oriented systems.

Last but most importantly, the management of HP's compiler labs were farsighted enough to understand that helping our customers solve their problems (even with using other vendors' equipment) is the surest way to keep our profit-sharing checks up.

MEMORANDUM

September 15, 1988

(13)

To: X3J3

From: M. Metcalf

Ref.: 109-LWC-2

Subject: **Editorial Assignment – Public Comments 93-319**

INTRODUCTION

The editorial comments on the following page are marked

1. with the appropriate comment number from 109-LWC-2 (a single digit), or
2. with a comment number from the next section (e.g. M2), or
3. a reassignment (group code).

The page and line numbers are based on S8.109.

NEW RESPONSES

- M1 B(1:5) is an array section of five elements. It is thus a variable according to R601.
- M2 Please refer to P. 7-5, lines 45-47.
- M3 Your proposed changes would conflict with the intended meaning of the document.
- M4 The DATA attribute is defined in R510.

MISCELLANEOUS PROPOSALS

1. P. 5-11, l. 32. After "program" add ", other than the main program".
2. P. 13-1, l. 23. At beginning of line add "**13.3 Positional and Keyword Arguments.**", and renumber subsequent sections.

(13)

93.11	M1
93.18	4
133.23	1
141.3	1
141.4	6
141.5	1
141.6	1
144.27	PROC
158.6	2
158.9	2
158.10	2
158.11	2
158.13	NO REPLY NECESSARY
158.15	2
158.19	1
158.22	1
158.26	8
158.28	2
158.31	8
158.32	1
158.33	8
158.38	1
158.59	2
158.60	2
158.62	6
158.68	1
158.69	1
158.71	2
158.72	1
189.7	8
189.11	2
189.12	2
190.6	GEN
190.11	3
190.12	8
191.5	6
191.8	2
191.9	M2
191.11	8
191.12	8
198.22	2
198.26	3
198.28	5
198.30	1
198.35	2
198.42	2
198.44	3
198.46	2
198.52	7
198.55	M3
198.86	1
198.88	2
198.92	1
198.93	1
198.104	2
199.9	1
203.1	1
204.4	2
206.2	2
206.4	7
216.12	1
216.15	2
226.4	2
230.15	1
230.25	3
230.41	C10
230.42	3
234.14	2
234.15	PROC
235.29	2
235.47	DAT
235.54	1
235.57	DAT
235.64	6
235.95	2
235.96	2
235.108	2
235.136	2
251.14	1
254.5	9
257.5	8
257.6	8
257.7	8
259.14	6
263.26	2
263.27	2
263.28	2
263.29	2
263.30	10
265.29	M4
265.36	2
265.55	2
319.1	5
319.12	5
319.17	1

PROPOSALS BASED ON THIS SET OF COMMENTS

1. *133.23*

P. 8-12, l. 10-11. Move the sentence beginning "When an input/output statement" to follow line 12.

2. *141.6*

P. 13-17, l. 11. Write the matrix as

A A A
B B B
C C C

l. 12. Write the matrix as

C C C
A A A
B B B

l. 13. Write the matrix as

C B A
A C B
B A C

P. 13-42, l. 40. Write the matrix as

2 2 2
3 3 3
4 4 4

3. *158.38*

P. 8-11, l. 19. Replace *"select-stmt"* by *"select-case-stmt"*.

4. *190.11*

P. 2-8, l. 3-4. Add hyphen between the words "association dependent".

P. 3-2, l. 19 and 22-23. Add hyphen between words "lower case".

5. *198.28*

P. vi, l. 12. Change "Vacant" to "E. Andrew Johnson".

6. *198.44*

P. 4-5, l. 17. Change "indicated" to "given".

7. *198.92*

P. C-16, l. 20. Add "! THE EMPTY SET" at end of line.

8. *203.1*

P. 5-7, l. 8. Change "TRANSFER" to "MOVE".

9. *230.15*

P. 4-10, l. 20. The sentence beginning "The type" should start a new paragraph.

10. *230.25*

P. 6-8, l. 24. Change first occurence of *"bound"* to *"bounds"*.

P. 8-6, l. 6+. After R824 add the line:
R825 *end-do-stmt* is END DO

P. 8-6, l. 33+. After R831 add the line:
R832 *do-term-shared-stmt* is *action-stmt*
and renumber following rules.

11. *230.42*

P. 7-9, l. 23. After "**specification expression**" add "(R504)".

12. *235.54*

P. 5-7, l. 40−. Add "An **automatic array** is an explicit-shape array **that is not a dummy** argument but whose bounds are dummy arguments to the procedure."

13. *263.26*

A-1, l. 31. Change "They" to "Obsolescent features".

14. *319.17*

P. 5-14, l. 12. Before "must" add "(4.5)".

END OF PROPOSALS

M. Metcalf

MEMORANDUM

September 17, 1988

(14)

To: X3J3

From: M. Metcalf

Subject: MIL-STD 1753 Bit Intrinsics and nondecimal constants

---

INTRODUCTION

The strong sentiment in favour of adding these intrinsics to S8, regardless of what form that document will finally assume, resulted in my being asked to resubmit this proposal. It now incorporates all the improvements and suggestions contained in ABMSW-4, and additional ones sent me by Brian Smith. The one exception is that I have retained the original names, in order to conform to exisisting practice. This will enable a far easier migration of existing code to the new standard, even though the names themselves are less than ideal.

PROPOSAL 1

Add the intrinsic procedures defined in MIL−STD 1753 to Section 13, with extensions to handle arrays. Existing sections require appropriate renumbering.

On p. iii, line 51 and p. iv, line 1, replace "functions and a comprehensive ... functions." *with:*

functions, a comprehensive set of numerical environmental inquiry functions, and a set of procedures for manipulation of bits in nonnegative integer data.

On page 13−1, line 16, change the title "Elemental Intrinsic Function Arguments and Results" *to* "Elemental Intrinsic Procedures" *and put this title on a separate line. Insert the title* "13.3.1 Elemental Intrinsic Function Arguments and Results.". *On p. 13−1, after line 25, add the new section 13.3.2:*

13.3.2 Elemental Intrinsic Subroutine Arguments. If a generic name is used to reference an elemental intrinsic subroutine, either all actual arguments must be scalar, or all output arguments must be arrays of the same shape and the remaining arguments must be conformable to them. In case the output arguments are arrays, the values of the elements of the results are the same as would be obtained if the subroutine with scalar arguments were applied separately to corresponding elements of each argument.

On p. 13−2, before the section "Derived-Type Inquiry Function", *add the following section and renumber the subsequent sections:*

13.4.5 Bit Manipulation and Inquiry Procedures. The bit manipulation procedures consist of a set of ten functions and one subroutine. Logical operations on bits are provided by the functions IOR, IAND, NOT, and IEOR; shift operations are provided by the functions ISHFT and ISHFTC; bit subfields may be referenced by the function IBITS and by the subroutine MVBITS (13.8.3); single-bit processing is provided by the functions BTEST, IBSET, and IBCLR. These procedures were originally defined by MIL-STD 1753 for scalar arguments, and are extended in this standard to accept array arguments and to return array-valued results.

///

For the purposes of these procedures, a bit is defined to be a binary digit w located at position k of a nonnegative integer scalar object based on a model nonnegative integer defined by

$$j = \sum_{k=0}^{s-1} w_k \times 2^k$$

and for which w_k may have the value 0 or 1. An example of a model number compatible with the examples used in 13.6.1 would have $s = 32$, thereby defining a 32-bit integer.

An inquiry function BIT_SIZE is available to determine the parameter s of the model. The value of the argument to this function need not be defined. It is not necessary for a processor to evaluate the argument of this function if the value of the function can be determined otherwise.

Effectively, this model defines an integer object to consist of s bits in an ordered sequence numbered from right to left from 0 to $s-1$. This model is valid only in the context of the use of such an object as the argument or result of one of the bit manipulation procedures. In all other contexts, the model defined for an integer in 13.6.1 applies. In particular, whereas the models are identical for $w_{s-1} = 0$, they do not correspond for $w_{s-1} = 1$, and the interpretation of bits in such objects is processor dependent.

On p. 13−5, add after line 14 (Section 13.8.2) the new section:

13.8.3 Bit Copy Subroutine. The subroutine MVBITS copies a bit field from a specified position in one integer object to a specified position in another.

On p. 13−7, add after line 2 the new section:

13.9.7 Bit Manipulation and Inquiry Functions.

BIT_SIZE (I)	Number of bits in the model
BTEST (I, POS)	Bit testing
IAND (I, J)	Logical AND
IBCLR (I, POS)	Clear bit
IBITS (I, POS, LEN)	Bit extraction
IBSET (I, POS)	Set bit
IEOR (I, J)	Exclusive OR
IOR (I, J)	Inclusive OR
ISHFT (I, SHIFT)	Logical shift
ISHFTC (I, SHIFT, SIZE)	Circular shift
Optional SIZE	
NOT (I)	Logical complement

On p. 13−8, add before line 28:

MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)	Copies bits from one object to another

On p.13−14, add the following sections after line 24 and renumber sections:

13.12.15 BIT_SIZE (I)

Description. Returns the number of bits s defined by the model.

(14)

Kind. Inquiry function.

Argument. I must be of type integer.

Result Type. The result is of type integer.

Result Value. The result has the value of the number of bits *s* in the model integer defined for bit manipulation contexts in 13.4.5.

Example. BIT_SIZE (1) has the value 32 if *s* in the model is 32.

13.12.16 BTEST (I, POS)

Description. Tests a bit of an integer value.

Kind. Elemental function.

Arguments.

I must be of type integer.

POS must be of type integer. It must be nonnegative and be less than BIT_SIZE (I).

Result Type. The result is of type logical.

Result Value. The result has the value .TRUE. if bit POS of I has the value 1, and has the value .FALSE. if bit POS of I has the value 0.

Example. BTEST (8, 3) has the value .TRUE.

On p.13−27, add the following sections after line 2 and renumber sections:

13.12.45 IAND (I, J)

Description. Performs a logical AND.

Kind. Elemental function.

Arguments.

I must be of type integer.

J must be of type integer.

Result Type. The result is of type integer.

Result Value. The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IAND (I, J)
1	1	1
1	0	0
0	1	0

(14)

0 0 0

Example. IAND (1, 3) has the value 1.

13.12.46 IBCLR (I, POS)

Description. Clears one bit to zero.

Kind. Elemental function.

Arguments.

I must be of type integer.

POS must be of type integer. It must be nonnegative and less than BIT_SIZE (I).

Result Type. The result is of type integer.

Result Value. The result has the value of the sequence of bits of I, except that bit POS of I is set to zero.

Example. IBCLR (14, 1) has the value 12.

13.12.47 IBITS (I, POS, LEN)

Description. Extracts a sequence of bits.

Kind. Elemental function.

Arguments.

I must be of type integer.

POS must be of type integer. It must be nonnegative and POS + LEN must be less than BIT_SIZE (I).

LEN must be of type integer and positive.

Result Type. The result is of type integer.

Result Value. The result has the value of the sequence of LEN bits in I beginning at bit POS right-adjusted and with all other bits zero.

Example. IBITS (14, 1, 3) has the value 7.

13.12.23 IBSET (I, POS)

Description. Sets one bit to one.

Kind. Elemental function.

Arguments.

I must be of type integer.

(14)

POS must be of type integer. It must be nonnegative and less than BIT_SIZE (I).

Result Type. The result is of type integer.

Result Value. The result has the value of the sequence of bits of I, except that bit POS of I is set to one.

Example. IBSET (12, 1) has the value 14.

On p.13−27, add the following section after line 14 and renumber sections:

13.12.46 IEOR (I, J)

Description. Performs an exclusive OR.

Kind. Elemental function.

Arguments.

I must be of type integer.

J must be of type integer.

Result Type. The result is of type integer.

Result Value. The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IEOR (I, J)
1	1	0
1	0	1
0	1	1
0	0	0

Example. IEOR (1, 3) has the value 2.

On p.13−28, add the following sections after line 7 and renumber sections:

13.12.48 IOR (I, J)

Description. Performs an inclusive OR.

Kind. Elemental function.

Arguments.

I must be of type integer.

J must be of type integer.

Result Type. The result is of type integer.

(14)

Result Value. The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IOR (I, J)
1	1	1
1	0	1
0	1	1
0	0	0

Example. IOR (1, 3) has the value 3.

13.12.49 ISHFT (I, SHIFT)

Description. Performs a logical shift.

Kind. Elemental function.

Arguments.

I must be of type integer.

SHIFT must be of type integer. The absolute value of SHIFT must be less than BIT_SIZE (I).

Result Type. The result is of type integer.

Result Value. The result has the value obtained by shifting the bits of I by SHIFT positions. If SHIFT is positive, the shift is to the left, if SHIFT is negative, the shift is to the right, and if SHIFT is zero, no shift is performed. Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are shifted in from the opposite end.

Example. ISHFT (3, 1) has the value 6.

13.12.50 ISHFTC (I, SHIFT, SIZE)

Optional Argument. SIZE

Description. Performs a circular shift of the rightmost bits.

Kind. Elemental function.

Arguments.

I must be of type integer.

SHIFT must be of type integer. The absolute value of SHIFT must be less than or equal to SIZE.

SIZE (optional) must be of type integer. The value of SIZE must be positive and must not exceed BIT_SIZE (I). If SIZE is absent, it is as if it were present with the value of BIT_SIZE (I).

Result Type. The results is of type integer.

Result Value. The result has the value obtained by shifting the SIZE rightmost bits of I circularly by SHIFT positions. If SHIFT is positive, the shift is to the left, if SHIFT is negative, the shift is to the

right, and if SHIFT is zero, no shift is performed. No bits are lost. The unshifted bits are unaltered. (14)

Example. ISHFTC (3, 2, 3) has the result 5.

On p.13−35, add the following sections after line 42 and renumber sections:

13.12.67 MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

Description. Copies a sequence of bits from one data object to another.

Kind. Elemental subroutine.

Arguments.

FROM must be of type integer. It must be conformable with TO.

FROMPOS must be of type integer. It must be conformable with TO. If TO is a scalar, FROMPOS+LEN must be less than BIT_SIZE (FROM), and otherwise MAXVAL (FROMPOS+LEN) must be less than BIT_SIZE (FROM).

LEN must be of type integer and positive. It must be conformable with TO.

TO must be a variable of type integer, and may be the same variable as FROM. It may be scalar or array-valued. TO is set by copying one or more sequences of bits of length LEN, starting at positions FROMPOS of FROM, to positions TOPOS of TO, element-by-element. No other bits of TO are altered.

TOPOS must be of type integer. It must be nonnegative. It must be conformable with TO. If TO is a scalar, TOPOS+LEN must be less than BIT_SIZE (TO), and otherwise MAXVAL (TOPOS+LEN) must be less than BIT_SIZE (TO).

Example. If TO has the initial value 6, the value of the result TO after the statement CALL MVBITS (7, 2, 2, TO, 0) is 5.

On p.13−36, add the following sections after line 20 and renumber sections:

13.12.69 NOT (I)

Description. Performs a logical complement.

Kind. Elemental function.

Argument. I must of type integer.

Result Type. The result is of type integer.

Result Value. The result has the value obtained by complementing I bit-by-bit according to the following truth table:

I	NOT (I)
1	0
0	1

7/9 117

Example. If I is represented by the string of binary digits 01010101, NOT (I) has the binary value 10101010.

END OF PROPOSAL 1

BIT CONSTANTS

It has been suggested that bit constants be added to S8, in addition to the B, O, and Z edit descriptors of 109.RCA-3. This proposal attempts to do that. The basic problem is to define a constant that is not an integral part of a type definition in the way that other constants are. Here they are associated with integer entities without developing new types, as we follow the MIL-STD in allowing them only in specification statements.

PROPOSAL 2

1. *P. 3-1, l. 20, add:* An exception is their use in a nondecimal literal constant (R404).

2. *P. 3-3, l. 8+, add:*

   or *non-dec-literal-constant*

3. *P. 4-3, l. 6+, add:*

   In type declaration statements (5.1) in which the *value-spec* (R510) is specified, and in DATA statements (5.2.6) and PARAMETER statements (5.2.7), further forms of unsigned nondecimal literal constants may be associated with integer scalar entities.

   R404 *non-dec-literal-constant* is *binary-constant*
   or *octal-constant*
   or *hex-constant*

   Constraint: A nondecimal literal constant may appear only in a type declaration statement, a DATA statement, or a PARAMETER statement.

   R405 *binary-constant* is B'*digit*[*digit*]...'

   Constraint: *digit* may have only the values 0 or 1.

   R406 *octal-constant* is O'*digit*[*digit*]...'

   Constraint: *digit* may have only the values 0 through 7.

   R407 *hex-constant* is Z'*hex-digit*[*hex-digit*]... '

   R408 *hex-digit* is *digit*
   or A
   or B
   or C
   or D
   or E

(14)

or F

In these constants, the binary, octal, and hexadecimal digits are interpreted according to their respective number systems.

and renumber subsequent rules.

4. *P. 5-6, l. 7, before the period add:* ", except that a nondecimal literal constant may be associated only with an integer object".

P. 5-6, l. 15, before the period add: ", except that a nondecimal literal constant may be associated only with an integer object".

5. *P. 5-13, l. 27, add:*

If a constant is a nondecimal literal constant the corresponding object must be of type integer.

P. 5-14, l. 24, add:

However, if the constant is a nondecimal literal constant the corresponding object must be of type integer.

P. 5-15, l. 6, add:

However, if the constant is a nondecimal literal constant the corresponding object must be of type integer.

END OF PROPOSAL 2

M. Metcalf

(15)

To: X3J3 110-RCA-1
From: Bob Allison
Subject: DO WHILE re-write
Date: September 9, 1988

This proposal is based on public review comments. The Control construct and I/O subgroup voted to forward it to the full committee. The BNF has deliberately been chosen so as to conform to MIL-STD 1753.

The proposal has been rewritten from (109-RCA-1) with different explanatory text: the BNF is unchanged (except the line numbers for S8.109 are in parentheses).

PROPOSAL

Add after page 8-5, line 30 (48):

or [,] WHILE (*scalar-logical-expr*)

Add after page 8-7, line 11 (29):

If *loop-control* takes the form [,] WHILE (*scalar-logical expr*), the result is as if no *loop-control* existed and the following were added as the next statement

IF (.NOT. *scalar-logical-expr*) EXIT

(16)

To: X3J3 110-RCA-2
From: Bob Allison
Subject: Reduction of intrinsic functions in constant expressions
Date: September 9, 1988

This proposal is based on the compromise plans. The plans by JKR, IRP, and RWW have agreed in principle to accept some form of simplification in this area.

The proposals are at the concept stage, so no specific text is provided.

Proposal 1 is tied to Inquiry functions, the most likely intrinsic functions to appear where specification statements require constant expressions.

Some believe that there are some intrinsic functions other than inquiry functions which are interesting. Proposal 2 attempts to come up with some simple rule which covers most useful cases. This proposal happens to contain all the functions in Proposal 1, plus a few more.

PROPOSAL 1

Only allow Inquiry Functions whose arguments do not depend on other objects. I.e., the SHAPE of an array with constant bounds is allowed, but not the shape of an assumed-size array.

PRESENT and ALLOCATED never meet this criteria.

Inquiry functions which meet this criteria:

LEN	TINY
DIGITS	LBOUND
EPSILON	SHAPE
HUGE	SIZE
MAXEXPONENT	UBOUND
MINEXPONENT	EFFECTIVE_EXPONENT_RANGE
RADIX	EFFECTIVE_PRECISION

(16)

PROPOSAL 2

Only allow intrinsic functions with INTEGER results whose arguments do not depend on other objects.

Intrinsic functions which meet this criteria:

<Inquiry functions in Proposal 1>

ABS	INDEX
INT	EXPONENT
MAX	COUNT
MIN	MAXVAL
MOD	MINVAL
NINT	PRODUCT
ICHAR	SUM

<Specific names of intrinsic functions above>

INT is not very useful in the list above since it may only take the other functions listed above or an integer constant as an argument and still be a constant expression (and the other functions return integers), but it keeps the rule simple.

(17)

RESOLUTIONS PASSED AT THE PARIS WG5 MEETING

[NB: Although this text is believed to be correct in every respect, it is a unofficial record of the resolutions, produced in order to allow quick distribution. David Muxworthy]

P1 LETTER CONCERNING INTERNATIONAL FORTRAN STANDARD

That WG5 requests SC22 to ask the US member body that X3J3 be reminded that X3J3 had been given the responsibility to develop the international standard for Fortran as well as the American national standard.

Passed: Individual 35 yes - 0 no - 2 abstain; Country 9 yes - 0 no - 0 abst

P2 REVISION OF DP1539

That WG5 agrees, based upon the ISO member bodies comments as documented in ISO/IEC JTC1/SC22 N464 and ISO/IEC JTC1/SC22 N495, and upon the X3J3 straw votes documented in X3J3/221 and X3J3/224, that DP1539 be revised in the following way:

a) in accordance with X3J3/S16 (S16 is a list of editorial changes)

b) as per the text in ISO/IEC JTC1/SC22/WG5 N302 with regard to the following features

1	remove the concept of deprecation	(US)
2	remove RANGE/SET RANGE	(Ca,D,NL,UK,US)
3	remove ALIAS/IDENTIFY	(Ca,D,NL,UK,US)
4	remove specified REAL/COMPLEX precision (REAL(*,*))	(D,J,NL,US)
5	remove internal procedures	(US)
6	remove square brackets for array constructors	(D)
7	add pointers (and associated facilities)	(Ca,F,D,NL,UK,US)
8	add MIL-STD bit intrinsic functions (but with original MIL-STD names restored)	(A,Ca,F,D,NL,UK,U
9	add significant blanks to free form source	(Ca,F,D,NL,UK,US)
10	change host association to use association in module procedures and remove host association	(US)
11	add parameterization (KIND=) to INTEGER	(UK)
12	add parameterization (KIND=) to REAL/COMPLEX	(D,J,NL)
13	add parameterization (KIND=) to CHARACTER so as to allow multiple character set support	(Ca,Ch,F,J,NL)
14	add the INCLUDE statement	(US)

c) text to be developed

1	remove user-defined elemental functions	(US)
2	remove the new form of the DATA statement	(US)
3	change interface blocks to that described in ISO/IEC JTC1/SC22 WG5 N316	(US)
4	change array constructor syntax to use I/O syntax	(US)
5	remove parameter to derived types	(US)
6	add stream I/O intrinsic procedures	(D,UK)

7 add binary, octal and hexadecimal constants and edit descriptors (Ca,NL,UK)
8 add parameterized LOGICAL (KIND=) (A,Ca,F,D,NL,UK,U

The codes alongside each point denote the member bodies which mentioned point in their comment. The abbreviations used are: A-Austria, Ca-Canada, Ch-China, F-France, D-Germany, J-Japan, NL-Netherlands, UK-United Kingdom, US-United States.

Passed: Individual: 30 - 2 - 5; Country: 8 - 0 - 1.

P3 WG5 AND X3J3 COOPERATION

That WG5 urges X3J3 to accept the plan passed as resolution P2 as the draft proposed standard for Fortran 8X.

Passed: Individual: 32 - 2 - 3; Country: 8 - 0 - 1.

P4 NAME OF LANGUAGE

That WG5 records its intent that Fortran 8X will be called Fortran 88, based on the 1988 date of passing resolution P2.

Passed: Individual: 30 - 0 - 7; Country: 7 - 0 - 2.

P5 A REVISED FORTRAN STANDARD IS NEEDED NOW!

That WG5 believes timely release of a revised Fortran standard to be cr and therefore establishes the following procedure and milestones:

September 23,	1988	WG5 adopts plan for revision of DP1539, accordin resolution P2; Convenor arranges for preparation revised text.
October 21,	1988	Draft text for revised DP1539 distributed to X3J
(November 13-18,	1988	X3J3 meeting.)
December 5,	1988	Draft, with possible editorial changes and corre of technical errors which might be recommended b X3J3, distributed by Convenor to WG5 for letter ballot authorizing the Convenor to forward the d to SC22.
January 20,	1989	End of WG5 letter ballot.
(February 17,	1989	End of X3J3 February 1989 meeting.)

If WG5 approves the draft, the Convenor forwards it to SC22, with possible editorial changes and correction of technical errors which might be recommended by X3J3 and as a result of WG5 ballot comments, after the Febru 1989 X3J3 meeting for further processing by SC22. The Convenor will arrang with SC22 the date of forwarding the draft so that the SC22 review period w be completed before the July 1989 WG5 meeting.

Passed: Individual: 24 - 4 - 9; Country: 6 - 0 - 3.

P6 WG5 REPRESENTATION AT X3J3 MEETING

That WG5 commission Gerhard SCHMITT (or an alternative to be named by the Convenor) to attend the next X3J3 meeting (November, 1988) for the purpose of helping communicate the WG5 position to X3J3.

Passed: Individual: 36 - 0 - 1; Country: 9 - 0 - 0

P7 VARYING CHARACTER MODULE

That WG5 requests the German member body to prepare a proposal for a Fortran module for varying character and the WG5 Convenor subsequently to request SC22 to split the work item to allow standardization of the module.

Passed: Individual: 33 - 1 - 3; Country: 9 - 0 - 0

P8 WG5 DELEGATION AT SC22/AG MEETING

That WG5 commission Gerhard SCHMITT or Brian MEEK as alternate to represent WG5 Convenor at the SC22/AG meeting October 17-19, 1988.

Passed: Unanimously

P9 WG5 CONSULTATION

That WG5 urges all its member bodies to ensure, at the time of public comment on a draft proposed standard, the widest possible distribution of the document within their respective countries, and to obtain reasoned technical comment, both positive and negative, from the largest possible number of Fortran users.

Passed: Unanimously

P10 VALIDATION

That WG5 requests the British member body to investigate the possibility of preparing a validation suite for Fortran 88 processors.

Passed: Individual: 31 - 0 - 6; Country: 8 - 0 - 1

P11 TESTING EXAMPLES

That WG5 requests members of the "Alvey Software Engineering Portable Package Framework/Fortran 8X Tools" Project to test the sample programs and program fragments contained in the revised DP1539 to be prepared in October 1988 and to report any suggested changes to the WG5 Convenor by November 21, 1988.

Passed: Individual: 30 - 2 - 5; Country: 8 - 0 - 1

P12 APPRECIATION OF X3J3 WORK

That WG5 expresses its appreciation of the work of the X3J3 committee in preparing the draft proposed standard (DP1539) for balloting in SC22.

Passed: Unanimously

P13 VOTE OF THANKS

That WG5 would like to express its appreciation to the Convenor (Jeanne MARTIN), the Chairman (Bert BUCKLEY), the Host (Christian MAS), the Organizer (Claude BOURSTIN), to AFNOR and its staff and to those organizations who provided further support and who have contributed to the success of the meeting.

Passed: Unanimously

---------- End of WG5 Paris Resolutions ----------

(18)

To: X3J3

From: N.H. Marshall

Subject: Plea to Retain Simple Internal Procedures

As a user, I feel that we should retain the Internal Procedure as a simple expansion of the statement function facility. If often happens that one needs to repeat the same basic functionality two or more times within a single program unit (and never need it outside of the program unit). Currently one's only recourse is to duplicate the coding several times, use external procedures, or to create spaghetti code by using Go to's. These options are not always desirable. If the desired functionality can be expressed in 5 to 10 Fortran statements, one may hesitate to duplicate that much coding three or four times. On the other hand, one may be reluctant to create an external procedure which is so short and called by a single program unit. In this day and age, it is never desirable to create spaghetti coding.

Simple Internal Procedures fill a basic need of Fortran programmers.

Personally, I would prefer to see Internal Procedures flagged with the keyword INTERNAL rather that use a CONTAINS statement. For example:

```
INTERNAL SUBROUTINE COLOR(BLUE)
INTERNAL INTEGER FUNCTION FCN(X)
```

(19)

110(*)LRR-1

To: X3J3
From: Larry Rolison
Subject: An alternative to the Schonfelder/Martin pointer proposal
Date: 13 September 1988

Since the topic of pointers seems to be on the ascendancy again and I strongly disagree with the concepts and syntax of the Schonfelder/Martin proposals, I'm unearthing my pointer paper from meeting 106 and once again offering it as an alternative. It has been somewhat refurbished by adding more justifications, rationalizations, and examples but the essential ideas have been maintained. This paper is a thought-piece. If the ideas are accepted, the text changes will be provided later.

I have repeatedly said that I contend the general FORTRAN [sic] population wants basically nothing more than an address to play with and a simple dynamic storage scheme to accompany pointers. My main goal is to present a model that is simple and straightforward. It has been said many times that simplicity is one of the hallmarks of FORTRAN and likely one of the main reasons for its success. I think that the concept of pointers is a fairly simple idea. They've been around for a long time and anyone other than a novice programmer has a pretty good feel for what they are and how they're used. Such an alleged simple idea, then, should be expressed in a language in simple syntax and semantics.

I have also repeatedly said that Fortran should not repeat the mistake of Pascal by implementing only strongly typed pointers. Remember just a few years ago when Pascal was the darling of the industry because it was so very safe? Remember how it enforced "correct" and "good" programming by only providing strongly typed pointers so a programmer could not shoot her/himself in the foot even if s/he wanted to? Remember what a great systems programming language it was supposed to be? And then C crawled out from under its rock and into the "darling of the industry" spotlight because real programmers found out very quickly that real systems programming could not be done with Pascal, and there was little C offering them all the freedom (and more) that they had been hungering for. Well, real programmers (you know, the full-contact variety) also use FORTRAN [sic] and in the same token will require untyped pointers to solve real problems. If the standard does not come out with untyped pointers, vendors will immediately extend it to implement them due to user demand so we may as well standardize them now.

I contend that the employment of a particular pointer mechanism (strongly typed vs. untyped) should be enforced by the programming shop, not the language. FORTRAN is a permissible and powerful language because its users require it to be so. I will try to appease both camps in this paper by suggesting a method for implementing both strongly typed and untyped pointers. I am convinced that the Fortran user community requires both forms of "pointers" to be incorporated into the language.

So, OK, let's get to specifics. The first thing needed is a pointer data type. In the previous version of this paper, I cynically said that dabbling in recursive data structures and considering the semantics of combining ALIAS and ALLOCATABLE might be an interesting intellectual exercise for some of us but that the great unwashed are crying for pointers so let's just give them to them. I'm happy to now report (a year later) that the committee in general seems to have accepted the contention that the introduction of a pointer data type is the proper course of action.

If we're going to invent a new data type, then the first thing to do is turn to Section 4 of S8 to find out what a data type is. To quote the Book: "A data type has (1) a name, (2) a set of valid values, (3) a means to denote such values (constants), and (4) a set of operations to

manipulate the values." Let's take these one at a time.

(1) a name

This one's easy. The type specifier for the pointer type is the keyword POINTER.

(2) a set of valid values

The set of values for the pointer type consists of the addresses capable of being computed by the processor.

This is the point where the strongly-typed/untyped camps part. The strong typing contingent restricts the valid values for a particular pointer to only those addresses for objects of the type to which the pointer is bound. The untyped pointer contingent allows a pointer to be able to contain any address the processor would otherwise be capable of computing. While I'm personally more inclined to the laissez faire brand pointer, I see no reason why a language (in particular, Fortran) should not be able to accommodate both preferences. To do so, a syntax must exist that allows one to simply declare a variable to be a pointer and a syntax must exist to bind a pointer to something.

To declare an untyped pointer (that may point at an arbitrary data object), you write:

```
POINTER  arb_ptr
```

I refer to this style of pointer in this proposal as being unbound.

To declare a strongly typed pointer, I propose the following syntax:

```
POINTER(type-info)
```

I refer to this style of pointer as a bound pointer because it is bound to a particular set of attributes. A bound pointer is implemented as an address to a data object as well as an implementation-dependent descriptor, if needed, to validate the pointer's use. (I believe a descriptor must exist only if the pointer is accessible externally to the program unit in which it is declared, but this is more a question of implementation I think.)

In the 106 version of this paper, I had "type-name" in place of "type-info". I restricted a pointer to being bound to a derived type definition. At that time, I thought allowing type declaration info in the parens to be too awkward (even though in my heart I wanted a pointer to be able to be bound to a set of attributes describing a simple scalar). I have rethought my position. I now intend "type-info" to mean a type declaration.

The following example shows how to declare a pointer that may only point at objects of a given derived type:

```
TYPE fruit
  INTEGER color
  INTEGER size
END TYPE
. . .
POINTER(TYPE(fruit)) :: fruit_ptr
```

The declaration of the pointer FRUIT_PTR states that it may only be used to locate structures of type FRUIT. The pointer is defined to be bound to the type FRUIT. If 8x retains type parameters, I would prefer the binding to ignore them.

The following example shows how to declare a pointer that is bound to a specified set of attributes:

```
POINTER(REAL,ARRAY(:,:))  ::  agg_ptr
```

The type declaration contained in the parens must follow the same basic rules as a normal type declaration statement.

The declaration of the pointer AGG_PTR states that it may only be used to locate data objects that are real, two-dimensioned arrays. I have not nailed down this form entirely. If this proposal is accepted, I invite discussion of how restrictive (or generous) we want this form to be. For instance, should we also allow array bounds to be specified, thereby only allowing the pointer to locate arrays with the same bounds? Should we allow storage attributes such as ALLOCATABLE or Ivor's new AUTOMATIC so the pointer may only point at an allocatable or an automatic item? I believe this form allows us as much freedom as we wish to grant it.

There are two additional points I wish to emphasize:

* Pointers are not inexorably linked to "allocatableness".

  No declaration for a data object of type FRUIT is shown above. The omission was by intent. A bound pointer declaration need not know anything about the actual data object that the pointer may be used to locate and, in particular, the object need not be allocatable. I am purposely separating pointers from "allocatableness" because I maintain they are two separate topics. They may interact with each other but they do not depend on each other. I will come back to this later.

* Pointers may only point at data objects and subobjects.

  Let's immediately dispense with notions like a pointer may point at a procedure. A pointer is a data address pure and simple.

(3) a means to denote such values (constants)

Schonfelder/Martin avoids the issue of a pointer constant by using the function ASSOCIATED to determine whether or not a pointer is associated with an object. I oppose this method because I feel it is inconsistent with the rest of the data types in the language. By this I mean that the language has no similar test to determine whether or not a numeric data item, a character string, etc. is currently defined. I don't think we should invent this idea for one particular data type.

I prefer POINTER to be consistent with other data types in that if no value has ever been assigned to it, it is undefined. (Like other data types, it may reach the undefined state by other means also.) The user may choose a value (such as 0 or ' ') to indicate an integer or character string is defined but does not contain an "interesting" value. In the same way, I propose the pointer constant NULLPTR. We can likely argue for hours on the spelling. PL/I uses NULL, Pascal uses nil, etc. I suggest NULLPTR because it mnemonically contains both ideas of what it is: a null value and of type pointer. If a future committee wants to use the keyword NULL for some general purpose, it will be available to them.

Since NULLPTR is a constant but has the same form as an identifier, we probably should allow it to also be a variable name since Fortran has no reserved keywords. This allows the pathological case:

```
REAL     nullptr
POINTER  abe
. . .
nullptr = 3.22
abe = nullptr
```

but since Fortran is already context sensitive a compiler can indeed determine that the real value 3.22 is not being propagated to the

pointer ABE. This is analogous to declaring an intrinsic function to be a type other than is language-defined type: it is not sufficient to remove its intrinsic and generic properties.

An alternative (which I favor because I think it will reduce confusion) would be to define NULLPTR in the same manner that the standard defines numeric and character constants. That is, since it is a constant like any numeric or character constant, it may not be declared (i.e., because one may not write "INTEGER 2", one may not write "INTEGER NULLPTR"). This would eliminate the above code confusion. This rule would not invalidate any existing standard-conforming program because the name NULLPTR is longer than 6 characters.

The null constant is the same for both bound and unbound pointers. There is no need to distinguish between a null-valued unbound pointer and a null-valued bound pointer.

(4) a set of operations to manipulate the values

I want to make several points in this section.

* The only operations permitted on a pointer are the tests for equality and inequality.

  A pointer may not (let me repeat that: may NOT) be an operand of an arithmetic expression. We have argued this before ad nauseum. Many people decry pointers as being inherently nonportable but it mostly comes down to the fact that they are nonportable when one performs arithmetic on them to cleverly step through a character string or some such nonsense.

  A processor must be capable of generating an address for every valid data object and subobject. Since it can do so, it follows that a pointer value can be constructed for every object and subobject (and, yes, it may require the pointer to be a software simulated pointer if the object to which it is pointing is not on a machine addressing boundary but it is a pointer nonetheless). Thus,

  (1) since all Fortran data objects are portable,

  (2) since a processor must be capable of addressing all objects and subobjects, and

  (3) since a pointer is an address,

  the pointer facility *is* portable. Note that the *value* of the pointer is not portable but we don't care about that. The code *is* portable.

* Both pointer operands of a comparison operator must be bound to the same type if at least one of them is bound to a type. An unbound pointer may only be compared to another unbound pointer. The constant NULLPTR may be compared to either a bound or unbound pointer.

  These restrictive rules are a safety net for the strong typing camp. A compiler writer could certainly relax the rules to allow the comparisons that I prohibit but I believe it would defeat the purpose of bound pointers.

* A pointer bound to a type may be assigned only to a pointer bound to the same type. An unbound pointer may only be assigned to another unbound pointer. The constant NULLPTR may be assigned to either a bound or unbound pointer.

  As stated above, this is a safety net to those who want to exercise control over the use of pointers.

The following points are not strictly part of the definition of a data type but are needed to clarify my definition of the pointer data type.

* A pointer may not be an I/O list item nor may it be a subobject of an I/O list item.

  In case I have inadvertently left any loopholes, my intent is to bar I/O of pointers completely. This restriction will likely be the proverbial straw that broke the camel's back for diehard bit-twiddlers (after already having had arithmetic operations on pointers taken away from them) but we really must close the door on this issue to maintain portability of a pointer facility.

  The complaint may arise that someone wishes to read a record from a file into a structure and link the structure into a linked list. They want the file record and the linked list structure to have the same declaration. Too bad. The programmer will simply have to declare the file record to be a substructure of the linked list node (the linked list node contains the additional pointer member). The file record can then be written from or read into the substructure.

* A pointer may be a component of a structure.

  Actually this is a moot point. Since in my model a pointer is a data type in full standing with all other data types, this is a lot like saying a real variable may be a structure component. I have no difference (I think) with Schonfelder/Martin on this topic. I would allow either type of pointer to be a structure component. I would also allow a bound pointer to be bound to the same structure definition as the structure in which the pointer is contained (to be able to set up linked lists) as well as being bound to any other structure definition. For example, the following declaration is valid:

```
TYPE vehicle
  CHARACTER*8                :: license_number
  POINTER(TYPE(vehicle))     :: next_vehicle
  POINTER(TYPE(owner))       :: owner_rec_ptr
  POINTER                    :: addl_info_ptr
END TYPE
```

* Arrays of pointers are allowed.

  Again, this is a moot point in my model because it's like saying arrays of objects of type real are allowed. However, in my model, I don't know that you can do anything really tricky with them. Given the declaration:

```
POINTER, ARRAY(10) :: p, q
```

  it would certainly be possible to write the array assignment statements:

```
p = NULLPTR
q = p
```

  I'm getting a little ahead of myself here but since I restrict a pointer to be a scalar when it is being used to reference an object, the following form is not currently valid in my model:

```
p->something = q->something_else
```

  I say "currently" because I think the idea might be too strange for Fortran (consider the logical progression from whole array pointer qualification to section references and trying to explain <u>that</u>) but if the details and sensible semantics can be worked out, I could be convinced otherwise. (A reviewer commented that this capability seemed to be a natural extension of array syntax.)

* A pointer has no storage sequence.

  I originally had this restriction in the 106 version in order to prevent access to a pointer value via EQUIVALENCE. I still want to say that a pointer (either format) has no storage sequence so access via EQUIVALENCE can be prohibited but if the compromise works out a method where items with no storage sequence can be named in a common block, I could go along with pointer being allowed in common blocks. It still makes me uncomfortable, however, because it opens the door to underhanded access to pointers.

* If a pointer is passed as an argument, the interface must be explicit.

  Since pointers do not exist in FORTRAN 77, I think we have to say this, don't we? At any rate, this is an attempt to prevent the passing of a pointer to, say, an integer and thereby opening up a world of wonderful things that could be done with (to) it.

  Argument matching for pointers follows the same rules as for assignment.

I hope it is becoming clear by now that my goal is to prevent access to a pointer in all cases except where it is used as a pointer. I believe it is the only hope we have of producing a portable pointer facility.

OK, I've defined a pointer data type. What can you do with it? The most obvious is, of course, to define a variable that may be used to locate another data object. I propose the syntax:

```
pointer -> object
```

The pointer qualifier variable must be scalar. It may be a simple scalar pointer variable, an element of an array of pointers, or a scalar structure component.

I chose the symbol "->" to indicate pointer qualification because it's intuitive, mnemonic, already used in some other languages for the same purpose (so it will already be familiar to some programmers), and (unlike symbols like the "up arrow" that Pascal uses) the individual characters are commonly found on keyboards and printers.

Note that I call the symbol a pointer qualifier. It denotes qualification in the same manner that a percent sign is used to qualify a structure component and a subscript list qualifies an array name. Since it's a qualifier, not an operator, it may be used on the left-hand side of an equal sign and may be "stacked" as in:

```
ME->HAND%FINGER->YOU%FACE
```

I oppose the Schonfelder/Martin syntax for the following reasons:

* I contend that the qualifier is not what is important when referencing a data object but rather the name of the object itself. We do not reference an array element by somehow setting up a declaration associating a subscript list with an array element then referencing the array element only using the (qualifying) subscript list (with no array name present). We don't reference a structure component by only specifying the (qualifying) structure name. So I can not understand why anyone would want to reference a data object by only naming its (qualifying) pointer.

  My syntax requires the object name as well as the pointer name to appear when referencing a pointer qualified item because both names are important.

(19)

* The Schonfelder/Martin syntax has a single form for both a pointer qualified reference to a structure component and a "normal" reference to a structure component. That is,

  P%MEMBER

  means either

  (a) the pointer P is locating the component MEMBER of some unidentified structure (that you can only find by hunting for P's declaration), or

  (b) a "normal" structure component reference.

  This syntax does a major disservice to program readability and maintainability. I am a firm believer in giving a programmer as much help as possible by exposing as much in the syntax as possible. If I could change FORTRAN such that array references could be distinguished syntactically from function references, I would do that also. Oh, sure, no syntax solves all the problems but I think we should provide as much help as we can.

* Because the Schonfelder/Martin syntax does not differentiate between a pointer and the object to which it points, a special pointer assignment statement or symbol had to be invented. I oppose both the use of IDENTIFY and "=>". I contend the arrow is not even pointing in the right direction because assignment is a movement or transference from right to left in the normal context. The intent of my syntax is to eliminate the confusion that plagued those that apparently first encountered pointers while learning a language like Pascal. I believe my syntax prevents people from becoming confused over whether it is the pointer that is being referenced or the object to which it points. Pascal-styled syntax has never made sense to me (having come from a PL/I background and a vendor systems language that has PL/I-like pointer syntax). For example, in the simple assignment statement

  P^ = Q^

  (see what I mean about character availability?) there is no indication in the syntax of the assignment statement itself what data is being moved. To me, the important thing to know when reading such a line is what data is being moved, not its locator. Thus, in a statement like

  P->TARGET = Q->SOURCE

  it is obvious in the syntax itself what data is being moved where. As I said before, I am a firm believer in <u>syntax</u> being obvious in sympathy for those that need to maintain software written by others.

  Once a reader knows that P and Q are pointers, my syntax makes it obvious to the reader that

  P = Q

  is a <u>pointer</u> assignment where

  P->TARGET = Q->SOURCE

  is a <u>data</u> movement using pointers.

Next, we need to determine just what a pointer can point at. The first thing we need to do is limit what can be accessed via a pointer. Since one of the reasons for FORTRAN's success is its optimizability (and therefore its execution speed), I want to maintain these characteristics. And since free-swinging pointers are the death of

optimization, I support the Schonfelder/Martin introduction of the TARGET attribute. (I would prefer something like PTRTARGET, or better yet POINTER_TARGET, but I can live with TARGET.) I propose that the TARGET attribute be required for any object that may be accessed via a pointer. (Structure components inherit it if applied to the structure.) I oppose the Schonfelder/Martin notion that all dynamic objects are/may be accessed via pointers by default. (I am unclear as to whether "dynamic" includes automatic. Their paper seems to be inconsistent in defining "dynamic".) Our optimization people say an automatic loop control variable in such an environment could be a disaster. Even though it can't be changed in the loop, it would have to be aliased to all other automatic items in the program unit because it could be changed outside the loop.

Recall that I said storage for a pointer-qualified item need not be created by an ALLOCATE statement. That assertion should have sparked two questions:

(1) If the pointer's target was not named in an ALLOCATE statement, how do you obtain a value for the pointer?

(2) If the pointer's target was named in an ALLOCATE statement, how do you create (and delete) the storage and how do you obtain a value for the pointer?

Let's take them in order. To generate an address for an object that was not named in an ALLOCATE statement, I propose the intrinsic function LOC. Again, we can argue about the spelling. (My heart lies with ADDR but LOC already exists in a couple of existing implementations so in the spirit of standardizing common practice....) I think there is no question on the requirement for an intrinsic function that returns an address. Of course, to make it complete, we need the usual rules that the argument must be defined, etc. I propose that the function accept arguments of any storage type.

Usage of the LOC function result is required to follow the same rules as for pointer variables. That is, if the result is assigned to a bound pointer, the argument to LOC must be an object of the type to which the assignment target is bound. However, I propose to allow a window of vulnerability here that may annoy the strong typing camp: the LOC of any data object may be assigned to an unbound pointer. I believe that this "out" is necessary occasionally and that prohibitions of its use should be legislated by the programming shop, not the language.

We can argue about whether the syntax

```
LOC(object)->another_object
```

should be permitted or not. I think it is ugly, largely unneeded, and should be prohibited.

Note that if LOC is used to obtain the address of a data object, the object must have the TARGET attribute.

Let's move on to dynamic storage. A major use of pointers is in manipulating linked lists. To create such a list, one generally makes use of a dynamic storage scheme to create (and possibly delete) storage for nodes in the list. Fortran 8x already has an ALLOCATABLE attribute and ALLOCATE/DEALLOCATE statements so I propose we generalize them. I claim that not only do allocatable arrays and pointers live quite well together but there are good reasons for allocatable arrays to remain in the language exactly as they are today.

I propose that the ALLOCATABLE attribute be used for any data object declaration to mean that the declaration is only providing a template for laying out the data object in storage and that storage will be allocated at execution time. I propose that the current ALLOCATE and DEALLOCATE statements become special ("shorthand") cases of my generalized statement form. To whit:

```
R610 allocate-stmt   is ALLOCATE(allocatable-object-name []
                        [] [, STAT=stat-variable]) SET(pointer)
                     or ALLOCATE(allocation-list []
                        [] [, STAT=stat-variable])

R613 deallocate-stmt is  DEALLOCATE []
                         [] (pointer->allocatable-object-name []
                         [] [, STAT=stat-variable])
                      or DEALLOCATE(name-list []
                         [] [, STAT=stat-variable])
```

My choice of the metaterm "allocatable-object-name" may not be 100% correct but take it at face value for the purposes of this paper. The metaterm "pointer" in the SET option and in the DEALLOCATE statement is defined to be the name of a scalar pointer. It must be definable at the time the ALLOCATE statement is executed and defined at the time the DEALLOCATE statement is executed.

First, I'll describe my generalized forms and how one uses them then I'll (attempt to) justify why the current form of the ALLOCATE statement should also be retained.

In my generalized scheme, basically the same rules exist for declaring and allocating an arbitrary object as for the current allocatable array scheme. That is, only a data object, not a subobject, may have the ALLOCATABLE attribute. It must also have the TARGET attribute. Let me illustrate by example. Suppose we want to build a linked list of structures. Using the current derived-type declaration and my generalized ALLOCATE:

```
TYPE person
  CHARACTER*24              :: last_name
  CHARACTER*24, ARRAY(10)   :: child_name
  POINTER(TYPE(person))     :: next
END TYPE

TYPE(person), ALLOCATABLE, TARGET :: employee
POINTER(TYPE(person)) :: new, delete

ALLOCATE(employee) SET(new)
```

My 106 version of this paper had a more complex example demonstrating the use of type parameters so that each structure in the list could contain components of different sizes. If 8x retains structure type parameters, my scheme will still work.

To delete a node in the linked list, one would first unlink it (code is left to the reader) then free the space:

```
DEALLOCATE(delete->employee)
```

The DEALLOCATE statement both deallocates the storage and sets the pointer DELETE to NULLPTR (to avoid the danger of dangling pointers).

The example makes use of a structure. I expect that typically structures will be the main interest of use with pointers but I see no reason why we should prohibit scalars and arrays from being allocated.

As with the LOC function, if the pointer in the SET option is bound, the object being allocated must be of the type to which the pointer is bound. If the pointer in the SET option is unbound, the object being allocated may be any object declared to be ALLOCATABLE.

Now that I have defined bound and unbound pointers and their general usage, I need to tighten up the definition of an unbound pointer somewhat. Unbound pointers are not completely and utterly unbound. I

find I must impose some restrictions in the interest of maintaining efficient code generation and execution speed. The main restriction I wish to impose is this: The processor may assume that the object being qualified by an unbound pointer is allocated on a "natural machine boundary". Let me illustrate by example:

```
INTEGER, TARGET   ::  i
CHARACTER*1, TARGET  ::  c(4)
POINTER  ::  p
...
p = LOC(c(2))
```

Assume that the processor is running on your favorite word-oriented machine. Further assume that the processor likes to allocate a character array (such as the array C) on a word boundary for easy access. And finally, assume that the processor supports some kind of (software simulated?) pointer that has a word address (the machine's "natural" address) and a bit offset within the word so that obtaining the LOC of an item not on a word boundary is possible.

Given the above code sequence, my restriction says that because pointer P is pointing at the second character of C (assume it's in the second byte of the word), if the user then writes the following statement:

```
j = p->i
```

the processor is free to assume that the integer I starts on a word boundary. This is because the "natural" addressing boundary for an integer on a word-oriented machine is a word boundary. In practical terms, this means that the compiler can ignore the bit offset portion of the pointer when accessing an item that naturally falls on a word boundary. What this means for the user in the above example is that data beginning with the first bit of C(1) will be transferred to J, not data beginning with the first bit of C(2) as the programmer might think.

So why do I want this restriction? Consider the code that would have to be generated to access a numeric or logical item if unbound pointers were completely free-ranging. The processor would have to generate a test on the bit offset portion to determine whether or not the pointer is pointing at the beginning of a word. I doubt users want this kind of execution degradation for every pointer-qualified access to a numeric or logical item.

Lest anyone accuse me of parochialism due to my coincidental employment by a manufacturer of two word-oriented mainframes, let me hasten to say that all vendors that do not have true bit-addressable machines (where no boundaries are favored) will eventually hit this same problem when a full-scale bit data type is implemented in Fortran. That is, byte-addressable machines today may happily access a numeric item on any byte boundary but what if Fortran had a bit data type and an unbound pointer was set to point at the third bit of a byte?

Although my example only specifically mentions word-oriented machines, the same problem may exist on a nominally byte-addressable machine that prefers 2, 4, or 8 byte boundaries for numeric items.

The C compiler group in our department tells me C has a cast operator that may be used to "translate" a character pointer to an integer pointer. In actuality, all it does is act as a flag to a programmer because the compiler just ignores the bit offset after the character pointer is cast to an integer pointer. The "wrong" data is then accessed (or accessible) just like I mentioned above. This may be OK in C's "the user is supposed to know what they're doing" environment, but it's not satisfactory in Fortran's world. I propose the introduction of an intrinsic function called BOUNDARY that accepts two arguments: an unbound pointer and a scalar data item (may be a variable or constant). If the pointer is pointing at the same boundary that the data item is allocated on, the function returns TRUE; otherwise, it returns FALSE.

The usual sensible rules apply: the pointer must be defined, the data item must be in the allocated state, etc. My intention is just to provide some guidance to the programmer that using an unbound pointer may not be providing them the information they assumed it was. What they do with the guidance is up to the programmer.

Using the code segment from above, the programmer could write:

```
IF (BOUNDARY(p,i))
  THEN j = p->i    ! Safe assignment.
  ELSE ...         ! Other action.
```

The result of BOUNDARY is processor-dependent. Notice that I'm purposely avoiding saying that once an unbound pointer is set to point at a numeric item, it may only be used to point at numeric items because then the pointer would not be unbound. I believe users need the power to get at arbitrary storage without sacrificing execution speed. "Normal" pointer-qualified references should be as efficient as accessing an item of the same data type that is not pointer-qualified where possible.

In the section where I talked about the operations allowed on a pointer, I said that an unbound pointer can be assigned to another unbound pointer. The restriction I have just described is not circumvented by assignment. Given the code segment supplied above, suppose another unbound pointer Q had been declared. If P is assigned to Q (after the LOC of C(2) was assigned to P) and Q is then used to reference I, the bit offset portion of the pointer is still ignored. In other words, you can not "force" an integer to begin on other than a "natural" boundary via assignment of pointers.

As promised, I will provide a justification for also retaining the current form of the ALLOCATE (and DEALLOCATE) statement. Schonfelder/Martin has the concept that ALLOCATABLE is no longer needed because ALLOCATABLE has exactly the same semantics as POINTER. In my model this is not true. ALLOCATABLE can be used in conjunction with pointers but may also stand on its own. Let me explain. It is conceivable that a user is only interested in a single instance of a data object in dynamic storage as a method of managing temporary data objects. I believe that the reason allocatable arrays were invented was to manage temporary storage, and in particular large amounts of temporary storage, without having to call a subprogram to create automatic storage. The current scheme works because such temporary storage management problems only need a single instance of the storage, manipulate it, then discard it. They don't need a collection (list) of such instances to exist simultaneously. If the need was sufficiently strong to have caused the invention of allocatable arrays, then the need must still be in existence (even with pointers). If a single instance is all the user requires, then why force them to use a pointer artificially (by eliminating the current ALLOCATE form)?

Although the single-instance allocation scheme was invented for arrays, it is a sufficiently powerful scheme that I have generalized array-allocation-list in the ALLOCATE BNF to include any allocatable data object. This would provide the power of having the processor manage the pointer if a user only needed one instance of an arbitrary data object. This generalized form is, I think, a useful expansion of the current allocatable array scheme and provides a nifty shorthand to my generalized ALLOCATE.. Of course, the user may not mix methods; that is, if the object is allocated with a SET option, it must always be accessed using a pointer. This also means that if the SET option is not used and the object is not an argument to LOC, it need not have (indeed <u>should not</u> have) the TARGET attribute.

For the user that wants to play with dynamic data objects, they would now have three choices: the current single instance scheme, the generalized ALLOCATE scheme (the item is then accessed via a pointer), or using true automatic storage (via Ivor et al.'s AUTOMATIC attribute).

(19)

Aside: The following is possible and permissible:

```
REAL, ALLOCATABLE, TARGET, ARRAY(:,:) :: a1
POINTER :: p1
...
ALLOCATE(a1(10,10))  ! No SET option; only one instance
...
p1 = LOC(a1)
```

but I'm not sure why anyone would go to the trouble.

In the 106 version of this paper, I said that I could live without the scalar IDENTIFY but that I thought the array IDENTIFY should remain in the language. I wanted to keep it around because arrays of pointers are conceptually too difficult to manipulate (people just don't think that way). I said I liked the ability of the array IDENTIFY to compute skewed sections, for example. Although the array IDENTIFY is sufficiently powerful and general enough to keep in the language and would live quite well with (my proposed) pointers, I now feel less strongly about retaining IDENTIFY. I find I need more information about the direction of 8x to make a decision.

This has certainly been a lot of material to digest in one proposal, so let me summarize:

* A new data type is proposed. It is called POINTER.

* The declaration form is POINTER [(type-info)]. If type-info is absent, the pointer being declared is an unbound pointer (and may point at any data object or subobject). If type-info is present, the pointer being declared is bound to the type specified and may only point at objects of that type. Type-info must follow the form rules for the attribute portion of a type declaration statement (with some restrictions, of course).

* The null pointer value is denoted by the language-defined constant NULLPTR.

* The symbol that represents pointer qualification is "->". Pointer qualifiers may be stacked. If an object is accessed via a pointer, the pointer name, the pointer qualification symbol, and the name of the object being qualified by the pointer must appear.

* The only operations permitted on pointer values are tests for equality and inequality. A bound pointer may only be compared to a pointer bound to the same type. An unbound pointer may only be compared to another unbound pointer. NULLPTR may be compared to either kind of pointer.

* A bound pointer may only be assigned to a pointer bound to the same type. An unbound pointer may only be assigned to another unbound pointer. NULLPTR may be assigned to either kind of pointer.

* A pointer may not be an I/O list item, directly or indirectly.

* If a pointer is passed as an argument, the interface must be explicit. The argument matching rules are the same as for assignment.

* A pointer has no storage sequence. I want to prohibit pointers from appearing in EQUIVALENCE statements but might begrudgingly allow them in common blocks.

* A LOC intrinsic function exists to compute the address of a data object. The result LOC must obey the same rules as for pointer

variables.

* In order for an object to be accessed via a pointer, it must have the TARGET attribute.

* The ALLOCATABLE attribute is extended to apply to any data object (but not subobjects).

* The ALLOCATE statement is extended to set a pointer variable and to allocate space for any object declared to be allocatable. If no pointer is provided, the rules remain the same as they are now but are extended to all allocatable objects.

* The DEALLOCATE statement is extended to free dynamic space located via a pointer and to set the pointer to NULLPTR. If no pointer is provided, the rules remain the same as they are now but are extended to all allocatable objects.

* The scalar IDENTIFY seems to have almost no usefulness given this implementation of pointers and could be abandoned. The array IDENTIFY still has some usefulness and could remain in the language.

* Disagreements with the Schonfelder/Martin model:

  - Providing only strongly-type pointers is insufficient.

  - I believe pointers should not be tied to "allocatableness". Single-level allocatable objects as in the current S8 model are sufficiently useful to retain.

  - The Schonfelder/Martin definition of a pointer is the combination of an address, a descriptor, and the qualified item's storage space. I define a bound pointer to be an address and optionally a descriptor. I define an unbound pointer to be simply an address.

  - The ASSOCIATED function in place of a null-valued pointer constant is inconsistent with other Fortran data types in that it is the only one that checks for an undefined state.

  - I contend the Schonfelder/Martin model that a dynamic object is both a pointer and a data object is confusing. I also contend that not being able to distinguish between a reference to a pointer and a reference to the qualified object is a disservice to maintenance (especially when used with a structure component).

I have tried to cover the major points of a pointer data type; I recognize work remains. My objective has been to be reasonably thorough while remaining in overview mode.

I imagine that most, if not all, of the ideas in this proposal are not original in that they have likely been covered by the committee prior to my arrival. I apologize for any toes I may have danced upon.

Appendix A: Alternatives Rejected

Pointer bound to collection of types

Our 1100 series systems programming language has a mechanism to bind a pointer to a collection of types. For example, since our 1100 machines are not hardware paging machines, the UCS (new-generation) compilers run in a software virtual paging environment. We segregate the dictionary (symbol table) information into one "area" and the text entries that represent the executable code into another "area". We have several kinds of virtual entries in the form of 8x-like structures that may be allocated in the dictionary area. Each entry may be thought of as an individual "type". It is sufficient in most cases to pass and use a pointer that points to any kind of dictionary entry. Each entry is identified by a field in the entry itself.

Although we have found this ability to be very useful, I have not included it in this proposal because I think just getting a cohesive pointer facility implemented at this point would be a minor miracle, let alone trying to add more functionality. If this proposal, or another like it, advances in the committee, I would be happy to discuss this type-grouping idea in more detail.

Appendix B: Other points

1. Is LOC a good choice for a name?

In the body of the proposal, I said that we should probably pick LOC as the intrinsic name because the name already exists in some implementations and performs basically the same operation. One of my reviewers pointed out that the "basically" may be a problem. Apparently customer programs exist that assign the LOC result to integer variables and other such nasties. Since the 8x definition of the function result may be different than the result currently being returned by a vendor's software and keeping an eye directed toward conversion costs, we perhaps should use another name. I'd like to fall back on ADDR if LOC would cause too many problems.

2. Storage allocation of structures

While working through this pointer proposal, a related problem with 8x's statement that a structure has no storage sequence reared its ugly head. In Appendix A, I mentioned that our compiler has a number of dictionary entries that are declared as having different "types" but that we access them via a single pointer. Each dictionary entry has a tag field that identifies the entry. Moving from one entry to another via a pointer depends on the fact that the tag field is always in the same relative location within the entry's storage. Thus, we can have a declaration that covers just enough of the entry to include the tag field (a header portion), interrogate the tag field, and operate on the entry accordingly.

I can not believe that we are the only programmers in existence that wish to do something like this. I am not making an argument to bring back variant types, although this might be the place to do that. I would rather resurrect an idea we discussed some time ago (Mt. Kisco maybe?): Can we develop an attribute to control the storage ordering of members of a structure? I think we only need to control the ordering; there should be no need to talk about storage allocation. I think we can have an attribute that tells the compiler to order the members exactly as declared and yet continue to state that a structure has no storage sequence (presumably to avoid EQUIVALENCE). If a user then declares a multitude of derived-type objects each having a header portion declared in exactly the same way, can we not count on a compiler to allocate storage exactly the same way?

In case I'm not making myself clear, here's what I think we need:

```
TYPE header
  INTEGER  tag
  INTEGER  other_information
END TYPE

TYPE entry_1
  TYPE(header)       header_info
  CHARACTER(LEN=20)  name
  ...
END TYPE

TYPE entry_2
  TYPE(header)       header_info
  INTEGER            number
  ...
END TYPE

TYPE(header), ORDERED   :: generic_header
TYPE(entry_1), ORDERED  :: node_type_1
TYPE(entry_2), ORDERED  :: node_type_2
POINTER                 :: gp
```

```
IF (gp->generic_header%tag .EQ. 31) THEN
  gp->node_type_1%name = its_name
ELSE
  gp->node_type_2%number = its_number
END IF
```

I have chosen the attribute ORDERED and applied it to the declaration of the structure. I would think we might want to allow such an attribute (statement?) to occur in a type definition, a la PRIVATE, to allow a user to ensure that all objects declared with the derived type are indeed ordered.

If any of you are involved with the PL/I committee, or have associates that are, you might find it interesting to discuss with them the lengths that committee has gone to in order to ensure that structures match in storage allocation. They needed to solve the same problem and did it in a more restrictive manner. In order to appease the anti-storage-association contingent of X3J3, I'm trying to be as general as possible.

Appendix C: Code examples

The following coded examples are one-to-one translations from those in 109-ABMSW-3 (109-57) to my syntax.

```
TYPE cell           ! Define a recursive type
  INTEGER                              :: val
  POINTER(TYPE(cell))                  :: next_cell
END TYPE cell

TYPE(cell), TARGET                     :: head
TYPE(cell), TARGET, ALLOCATABLE        :: node
! Declare pointers
POINTER(TYPE(cell))                    :: current, temp
INTEGER                                :: ioem, k

head%val = 0
current = LOC(head)

DO
  READ(*,*,iostat=ioem) k    ! Read next value if any
  IF (ioem .NE. 0) EXIT
  ALLOCATE(node) SET(temp)       ! Create new cell each iteration
  temp->node%val = k             ! Assign value to cell
  current->node%next_cell = temp
  current = temp
END DO

current->%next_cell = NULLPTR
```

The loop to "walk through" the list may be written:

```
current = LOC(head)

DO
  WRITE(*,*) current->node%val
  IF (current->node%next_cell .EQ. NULLPTR) EXIT
  current = current->node%next_cell
END DO
```

(19)

```
PROGRAM dynam_iter
REAL,ARRAY(:,:),TARGET,ALLOCATABLE  :: a, b
! Declare pointers
POINTER(REAL,ARRAY(:,:))            :: a_ptr, b_ptr, swap
...
READ(*,*) n, m
! Allocate arrays
ALLOCATE(a(n,m)) SET(a_ptr)
ALLOCATE(b(n,m)) SET(b_ptr)
! Read values into A
...
iter: DO
  ...
  ! Apply transformations of values in A to produce values in B
  ...
  IF (converged) EXIT iter
  ! Swap A and B
  swap = a_ptr; a_ptr = b_ptr; b_ptr = swap
  END DO iter
...
END
```

```
PROGRAM iter
REAL,ARRAY(1000,1000),TARGET        :: a, b
! Declare pointers
POINTER(REAL,ARRAY(:,:))            :: in, out, swap
...
! Read values into A
...
in = LOC(a)   ! Associate IN with target A
out = LOC(b)  ! Associate OUT with target B
...
iter: DO
  ...
  ! Apply transformations of values in A to produce values in B
  ...
  IF (converged) EXIT iter
  ! Swap IN and OUT
  swap = in; in = out; out = swap
  END DO iter
...
END
```

(20)

To: X3J3

From: N.H. Marshall

Subject: Mailing Address

Now that I am collecting the pre-meeting distribution items, several of you have tried to send me material via Federal Express, or some other such means. You have discovered, somewhat to your consternation, that I do not have a street address listed in the meeting minutes. What follows is my complete address, including a street address.

Neldon H. Marshall
EG&G Idaho, MS 2408
P.O. Box 1625
1580 Sawtelle St.
Idaho Falls, Idaho 83415

(21)

To: X3J3

110-JKR-1

From: John Reid

Subject: Guidelines for scribes

Date: 25th September 1988

Once again, I would like to thank the scribes for their support at the last meeting.

I have prepared guidelines in the form of a specimen set of notes. Please note that everything ascribed to me or the motions is genuine guidance and is not just 'space-filling'.

Specimen scribe notes

Discussion leader: Reid

Scribe: Another

Reference:	107-18 (JKR-1). Meeting minutes.
Reid:	Begin with a brief overview of the topic by summarizing the presentation. Omit this if if the title already does it.
Straw Vote:	The scribe notes must be posted by air mail to the secretary (J. K. Reid, Blg 8.9, Harwell Laboratory, Oxon OX11 0RA, England) in the week following the meeting. (30-0-0)
Jones:	What format do you want?
Ans:	Copy these notes as closely as possible. If you do not have bold, use underlining. Refer to people by their last names (first names make it hard for readers who are not committee members). Please send a top copy, printed on one side of the paper, with adequate margins (line length at most 7 inches), avoid a grey tone (worn ribbon), and do not fold it.
Smith:	How accurate do the scribe notes need to be?
Ans:	They are not intended to be a verbatim record of what is said, but should record the major points made. This is helpful to absent committee members, people who are not committee members but like to follow the progress of the committee, and to committee members wishing to remind themselves of the issues in a year or two's time. If you do not hear something that you judge to be important, ask the chairman for it to be repeated, or ask the speaker privately afterwards to explain. It is not very useful to include something like 'comment inaudible'. Also do not scribe detailed editorial changes that will be recorded in the marked up version of the proposal that will appear in the second supplement to the minutes.
Motion:	If a proposal in a paper is amended, the discussion leader must provide an amended copy of the paper to the librarian (Marshall) before the end of the meeting (Reid, Adams).
Formal Vote:	**27-0. Passed.**

(22)

To: X3J3

110-JKR-2

From: John Reid

Subject: Using i/o syntax for array constructors

Date: 25th September 1988

1 Introduction

This is a formal proposal for replacing the syntax of array constructors by that of i/o lists. I found that I needed to make a small change in Section 9, but it is my belief that it represents an editorial improvement in any case.

2 Proposal

Make the following changes to S8.108:-

1. Page 4-9, lines 41-42. Delete sentence 'The sequence .... constructors.'.
2. Page 4-9, lines 43-44. Change '*array-constructor-value*' to '*output-item*', twice.
3. Page 4-9, line 46 to page 4-10, line 17. Replace 'R423 ... the first.' by 'Each array expression in the *output-item-list* is treated as a sequence of values in array element order (6.2.4.2).'
4. Page 4-10, lines 19-20. Delete sentence 'The *scalar .... constructor.*'.
5. Page 4-10, line 21. Change '*array-constructor-value*' to '*output-item*'.
6. Page 4-10, line 29. Change '2[4.5]' to '4.5, 4.5'.
7. Page 9-12, line 44. Change '*io*' to '*input*'.
8. Page 9-13, line 1. Change '*io*' to '*output*'.
9. Page 9-13, lines 2-4. Replace 'R916 ... *output-item*' by
   R916 *input-implied-do* is (*input-item-list, io-implied-do-control*)
   R917 *output-implied-do* is (*output-item-list, io-implied-do-control*)
10. Page 9-13, lines 8-9. Delete constraint.
11. Page 13-40, line 5. Replace '1:6' by '1,2,3,4,5,6', twice.

(23)

110-JKR-3

To: X3J3

From: John Reid

Subject: The WG5 plan

Date: 25th September 1988

1. Introduction

The plans of Weaver, Philips, Reid/Smith, and Brainerd *et al.* were presented to the ISO/WG5 meeting in Paris by Dick Weaver, Ivor Philips, Andy Johnson, and Lawrie Schonfelder, respectively. It was decided quite quickly that neither the Weaver plan nor the Philips plan were suitable. They were seen as too large a departure from the draft and likely to result in many no votes in a second ISO ballot. The authors of the remaining plans meet to discuss how a compromise plan might be constructed that met the objectives of both plans and was likely to be acceptable to WG5. This left several decisions open, so straw votes of WG5 were taken before a final plan was proposed. This was modified slightly by WG5 and was adopted on the final day with a vote of 30-2-5 by individuals (Dick Weaver and Ivor Philips voting no) and 8-0-1 by countries (USA abstaining).

WG5 also adopted a resolution expressing its belief that the timely revision of the Fortran standard is critical and adopting a set of milestones leading to the completion of a second ISO ballot before the next WG5 meeting (10-14 July 1989). This was passed with a vote of 24-4-9 by individuals (Weaver, Philips, Johnson, and Warren (IBM, Canada) voting no) and 6-0-3 by countries (Japan, Sweden, and USA abstaining).

This paper is an attempt to explain the plan informally. I (and WG5) very much hope that X3J3 will accept this plan. If it does not, then ISO will proceed on its own which is certainly not an outcome for which I planned when joining X3J3 or accepting the post of Secretary.

The aim of my plan (see 109-37, JKR-4) was to reduce the size of Fortran 8x without needing a massive editorial effort and without losing the essentials of the language improvements in S8. The design objectives for the revision were discussed about ten years ago and are laid out in S6 (May 1983). I summarize these in section 2 and firmly believe that we should stay as close as possible to them. This certainly is the view of WG5.

Each major change has a separate section. The first seven are those that were accepted individually at Urbana but rejected as a package. I have tried to order the rest, with those making the biggest change at the top. I conclude by summarizing the changes from the Urbana package and considering whether the objective of reducing the language complexity has been achieved.

2. S6 design objectives

An agreed statement on design objectives for Fortran 8x is contained in S6 and was reproduced as document 109-92 (JKR-6). The 'core' language was defined as consisting of Fortran 8x less its decremental features (now just the obsolescent features) and was intended to be a complete and consistent language conforming to the following criteria:-

(i) **General purpose.** 'The core must especially strengthen Fortran's capability for general purpose scientific programming applications.'

(ii) **Portable.** 'A principal goal of the core is (program and people) portability (that is, after all, the reason for standardization).'

(iii) **Safe.** 'Preferred features for inclusion in the core are those
(a) which are least likely to be (inadvertently) mis-used,
(b) for which unexpected side-effects don't occur,
(c) for which errors in use are most easily detected, and
(d) which maximize program readability.'

(iv) **Efficient.** 'Features that preclude either compilation or execution efficiency with conventional contemporary computing technology should not be included in the core.'

(v) **Consise and consistent.** 'The core should be a small language, easy to learn and use effectively.' 'All syntactic and semantic elements of the core should follow regular and consistent patterns.'

(vi) **Contemporary.** 'The core should be characterized by language features that are broadly accepted as currently the best means of achieving the desired functionality.'

(vii) **Upward compatibility.** 'The core should maintain a high degree of compatibility with Fortran 77.'

3. Remove RANGE

Removing RANGE makes a big reduction in the size of the language and the complexity presented to the user. Not only will the number of lines removed be substantial, but every time the size, shape, or bounds of an array are mentioned, we will know what is meant without having to think about whether it is the declared or effective ones that are involved.

4. Add pointers

The plan proposes the addition of the pointer facility explained by Jeanne Martin at the Jackson meeting (see 109-57, ABMSW-3). They are typed and ranked and may not point to static objects that have not been declared with the attribute TARGET.

5. Remove IDENTIFY

Removing IDENTIFY makes a big reduction in the size of the language. It removes a new form of association that the new users will find hard to understand.

6. Simplify generalized precision

Brian Smith's plan (109-61, ABMSW-7) with FLOAT_KIND spelt KIND was thought by WG5 to be a very acceptable way to simplify generalized precision.

7. Add bit intrinsics

The plan involves the addition of the Mil-Std bit intrinsics, as in 109-58 (ABMSW-4) but with the original names restored.

(23)

8. Remove the concept of deprecation

X3J3 has already vote formally to remove the concept of deprecation. The proposal adopted is 109-64 (ABMSW-10).

9. Add INCLUDE

The plan involves the addition of the INCLUDE statement, as in 109-60 (ABMSW-6).

10. Add parameterized INTEGER, CHARACTER, and LOGICAL

Given the acceptance of the KIND solution to the generalized precision problem, WG5 saw it as natural to use the same solution for the demand for short integers, long characters, and bits. They welcomed the consistency of having a KIND parameter for all intrinsic types and regarded this as a reduction of complexity.

11. Remove host association

My suggestion at Jackson was to remove internal and module procedures completely, but I realized there that removing internal procedures was sufficient if supported by the replacement of host association in modules by use association. In a straw vote, 13 members said that the plan would be acceptable with module procedures retained and 15 said that the plan would be acceptable with module procedures removed. At Paris, there was some reluctance to accept the deletion of internal procedures and a far greater resistance to the deletion of module procedures. The modularization and name-hiding advantages of module procedures and the associated safety gains were seen as very important. The differences between use and host association were minor and hard to remember, so the change will represent a worthwhile reduction in complexity.

12. Remove derived-type parameters

WG5 was reluctant to see the removal of derived-type parameters because it leads to an inconsistency with the intrinsic types. However, it was accepted that derived types without parameters would have been far more common, and that the text describing derived-type parameters is quite complicated and involves the need to define another kind of expression. Also the implicit intrinsic functions are easily overlooked.

13. Remove elemental calls of user procedures

Most procedures would not have been called elementally, so the compiler writer would probably have implemented elemental calls as a sequence of scalar calls. It will be more efficient to demand that the user provide versions for the ranks actually wanted.

14. Add intrinsic procedures for stream i/o

The plan involves adding intrinsic procedures such as GET_CHAR and PUT_CHAR to provide the primitive facilities for implementation of stream i/o in a module.

(23)

15. Change array constructor syntax

The plan involves the removal of the use of square brackets and includes my suggestion to use the syntax of output lists for array constructors. Users are very familiar with this syntax. Here are some examples:

```
(/ 1.2, 3.4, 5.8 /)
(/ (I,I=1,N) /)
(/ (1.0,I=1,50) /)
```

16. Remove new form of DATA statement

The new form of DATA statement offers no functionality that is not available in the old form.

17. Add significant blanks to the new source form

The plan involves adding significant blanks as in 109-59 (ABMSW-5).

18. Add binary, octal, and hex constants and edit descriptors

The plan involves adding binary, octal, and hex constants and edit descriptors.

19. Overloading of user procedure names

WG5 saw overloading as an integral part of a derived-data facility and were not willing to see it removed. For example, the fuction SIN is needed as part of a module for interval arithmetic and as part of a module for extended precision arithmetic. WG5 liked Dick Weaver's idea for a generic interface block and adopted it as part of its plan.

Note that under the rule in 14.1.1 of S8, external procedures must have distinct names; since the plan removes internal procedures, only module procedures may be overloaded without the use of a generic block.

20. Conclusions

The primary objective of the plan that I presented at Jackson was to provided a simplification over the plan that was formed at the Urbana meeting and rejected there. The Urbana plan contained the items in Sections 3 to 9. Sections 11, 12, 13, 15, and 16 each represent a worthwhile reduction in complexity. Section 10 represents an increase in language size in direct response to demands, but the uniformity of the treatment of all intrinsic types represents a complexity reduction over other possible ways to provide the functionality. Section 14 represents a small increase in complexity and Sections 17 to 19 represent very small increases. Overall, I believe that the WG5 plan meets the objectives of my plan, while it certainly addresses the international comment better.

In this paper, I will look at how the concept of storage association has evolved, including both the standard provisions and the nonstandard expectations. I will then look at alternatives for extending storage association concepts to the features in Fortran 8x and make a recommendation on the approach to follow. Finally, I will make specific suggestions on how to describe this approach in the standard.

This paper does not include specific text changes to be made to the draft, but I hope that it is sufficiently specific technically that it could be used as the basis for writing such text. I have tried to allow fo expected changes in the standard (e.g., the addition of pointers), but changes may be necessary to accommodate the particular wording used to add these features.

The 1966 Standard

The 1966 standard had a storage model based on the idea that all machine representations were constructed out of a common storage unit (i.e., the computer word). A storage unit could be used in one of seven ways:

1. to store an integer value (type INTEGER)
2. to store a true/false value (type LOGICAL)
3. to store a "normal" floating point value (type REAL or the real or imaginary part of type COMPLEX)
4. to store the first half of an extended floating point value (type DOUBLE PRECISION)
5. to store the second half of an extended floating point value (type DOUBLE PRECISION)
6. to store a code address (with the ASSIGN statement)
7. to store a group of *g* characters, where *g* is a processor-dependent value (Hollerith data)

The number and relative placement of all aggregations of storage units was specified by the standard. Different variables and arrays could be forced to use the same storage units. If one variable was used to define a storage unit in one particular way, then if a second variable referenced the storage unit in the same way, it would be defined to have the same value. If the second variable referenced to storage unit in a different way, then its value would be undefined, since no relation was assumed between the different ways of interpreting a storage unit.

In this form, the storage association rules had two big advantages:

1. The rules were complete. A user could specify any storage sharing pattern he wanted or needed. It was even possible to create a pool of storage that could be allocated to different uses on a dynamic basis.

2. The rules were processor independent. A valid storage sharing pattern on one machine would be valid on another.

These rules also had their disadvantages:

1. The size relations specified are not always appropriate:

   a. Giving the LOGICAL type the same amount of storage as the INTEGER type is a waste of storage. (Fortunately, most programs had only a handful of scalar flags, so the wastage was minimal.)

   b. One would like the standard COMPLEX type to be sufficiently precise for most computations that might use it. Limiting its components to the size of the standard INTEGER sometimes prevents this.

   c. On some machines based on the IEEE floating point standard, extended is the fastest as well as the most precise representation, so one would like it to correspond to one of the FORTRAN types, but this isn't possible (without serious storage wastage) because extended is neither half nor twice the size of any of the other representations.

2. The rules treat all storage units as alike and thus do not allow for alignment constraints. For example, on many machines there is a performance penalty if double precision items are not stored on even (rather than odd) word boundaries. The standard provided no way for the processor to handle this, so it became the responsibility of the programmer on those machines, reducing portability.

3. The specified relative placement of storage units in aggregates is not always most appropriate. For example, some processors would be able to generate better code if arrays of type complex could be represented by parallel arrays for the real and imaginary part rather than interleaved values. Other processors might perform better if memory constraints such as page boundaries or bank conflicts could modify the addressing formulas used for arrays.

The Nonstandard Standard

In addition to the properties provided by the standard, a number of additional properties were assumed by many programs:

1. **Different uses of a storage unit actually used the same memory.** I know of no processor that violates this assumption, but it should be noted that the rules in the standard only allow such sharing without requiring it. One can imagine tagged

machine architectures where it might be more convenient no to share memory for incompatible uses.

2. **"Undefined" means only a processor-dependent value.** Many programs assume that it is safe to look at storage units in a way different from the way they were defined as long as you don't mind the fact that the value you see would be processor-dependent. This is ignores the fact that there may be representations that are invalid for this kind of use and that machine interrupts may result.

3. **Assignment and binary input/output are representation-preserving operations.** Typical of this assumption would be defining a REAL variable, writing out an INTEGER variable equivalenced to it, reading the INTEGER back later, and expecting that the value of the REAL variables would then be defined as it was originally. This ignores the possibility that the processor may transform the representation to some canonical representation appropriate to the type being used in the transfer (e.g. normalizing what is apparently a floating point number).

4. **The values in the INTEGER type are in 1-1 correspondence with the possible representations in a storage unit, so INTEGERs can be used for comparing and manipulating representations.** This ignores things like 1's complement machines (2 representations for zero) and machines using decimal arithmetic for INTEGERs.

5. **There are specific relations between the representations used in the different uses of a storage unit:**

   a. **The LOGICAL interpretation of a storage unit is .TRUE. if and only if the INTEGER interpretation is negative.** (Alternatively, the LOGICAL interpretation of a storage unit is .TRUE. if and only if the INTEGER interpretation is nonzero.)

   b. **The INTEGER interpretation of a storage unit has the same sign as the REAL interpretation of that storage unit.**

   c. **INTEGER and REAL zero have the same representation.**

   d. **The first storage unit in a DOUBLE PRECISION value can be interpreted as a REAL value that is approximately equal to it.**

   e. **If storage units contains Hollerith data, comparison as INTEGERs will give the same results as character comparison.** (Alternatively, it gives the same results provided you take into account the interpretation of the sign bit.)

   f. **"Simple" Hollerith values never have the same representation as "small" INTEGERs.**

None of these assumptions is likely to be safe enough across the full range of machines on which Fortran is implemented to be adopted into the standard, but all are true often enough that people get upset when features deny them the possibility of taking advantage of their favorite nonstandard assumption.

The 1978 Standard

The development of the 1978 FORTRAN standard provided a new complication — the CHARACTER data type. As with LOGICAL, the use of an entire storage unit would be a waste if storage, and, unlike LOGICAL, people tend to use enough characters that the wastage would be significant. Two possibilities would have preserved the idea of a single underlying storage unit:

1. The length in the CHARACTER type could have been interpreted as a minimum capacity rather than an exact capacity. Thus, each CHARACTER entity could have been allocated in storage units, based on the processor-dependent constant *g* originally defined for Hollerith. The presence of a processor-dependent constant would have complicated portability, this approach would not have provided a meaningful EQUIVALENCE between a CHARACTER array and a CHARACTER string declared to hold the same number of characters, and the wastage (especially for CHARACTER*1 and on byte-addressable machines) would still have been significant.

2. The storage unit for a character could have been made the new basis for a complete mapping like the one in the 1966 standard (in effect, converting the FORTRAN model from word addressing to byte addressing). Again, there would have been problems with the processor-dependent nature of *g*, and the alignment issue previously mentioned for DOUBLE PRECISION would likely now apply to all of the "numeric" data types.

To avoid these kinds of problems, X3J3 chose to recognize two different kinds of storage units — numeric and character. Each data entity consisted entirely of one or the other and complete mapping was retained among objects composed of the same kind of storage unit, but no mapping was allowed among objects composed of different kinds of storage units. This solution provided storage efficiency, portability, and nominal upwards compatibility from the 1966 standard but was unpopular with much of the FORTRAN community for a number of reasons:

1. It was no longer possible to have aggregations containing all types of information. Splitting collections of shared data in half to separate the character data from the noncharacter data was a nuisance.

2. It was no longer possible to manage a single pool of available storage and allocate it any possible usage.

3. The language no longer provided even the syntactic framework for applying some of the nonstandard assumptions.

The 198x Drafts

Fortran 8x introduces several additional complications to the storage association model. First, it introduces new types whose underlying storage units are likely to be different from those already in the language. In particular, people want the BIT type to be storage efficient. Second, it introduces parameterized types whose underlying representation (and thus storage units) vary with the parameter value. Thus, we can't portably determine how many different kinds of storage units there need to be. Third, we have introduced attributes such as ALLOCATABLE that alter the representation of the objects, thus introducing still more kinds of storage units. A number of approaches has been considered:

1. **Ignore storage association altogether and allow only that which is necessary to be upwards compatible with the 1978 standard.** Public comment has made it clear that this "solution" is not acceptable.

2. **Have many kinds of storage units and handle them as in the 1978 standard — "separate universes".** I think the displeasure already expressed about the 1978 standard on this point makes it clear that this "solution" would also not be acceptable.

3. **Build a new complete mapping based on a new basic storage unit (presumably the storage unit for the BIT type).** As noted before in the context of the 1978 standard, this causes portability problems because of the processor-dependent constants involved, and introduces the problem of expressing alignment constraints. In addition, there is the problem that the model in the 1978 standard may actually be the right one for some architectures. There are a number of hardware and software reasons why the bit-addressable store, the character-addressable store, and the word-addressable store might not be identical. If possible, we need the preserve the possibility of efficient implementations on such machines. Again, I believe we have an unacceptable "solution".

I believe an acceptable solution lies somewhere between these latter two options. Start with the idea of different storage units for different types, but recognize that these may be composed of a smaller, more basic storage units that would allow different kinds of storage units to placed in the same storage sequence. If a machine actually has multiple kinds of memory, a Fortran storage sequence may actually have to be implemented as a memory sequence in each kind of memory with a position in the storage sequence implemented as a set of position (one for each kind of memory). (In fact, a good test of the portability of storage association features is to ask whether the described behavior applies to both a monolithic memory implementation and a multiple memory implementation.)

I am suggesting the basic tactic that when two data objects have the same placement constraints (are composed of the same kind of storage units), then the order specified by the programmer should be honored, but when the data objects have different placement constraints (are composed of unlike storage units), then the processor should be free to reorder the objects (e.g. to improve efficiency by grouping objects with the same or similar placement constraints). Thus, for example, a derived data type containing only character data could be expected to portably overlay a character string, while a derived data type containing an integer and a character string could not.

I have somewhat reluctantly decided to avoid the concept of a numerical storage units in the description that follows. Instead, I will be talking about type specific storage units which are the same size (equivalent) and may thus coincide in storage sequences. It turned out to be easier to talk about different kinds of storage sequences than different ways of using one kind of storage sequence.

The Proposed Approach

Variables in Fortran are composed of storage units which contain the component values that make up the value of the variable. There are different kinds of values for storing different types of values. Unless otherwise noted in the following material, there is a different kind of storage unit for each combination of type and type parameters. In addition, there is an additional storage unit for each variable that may be composed of different storage units over the lifetime of the program unit in which the variable was declared (i.e., variables with the ALLOCATABLE, ALIAS, or "pointer" attribute). Unless otherwise noted in the following material, there are different kinds of such locater storage units for each combination of type, type parameters, and rank. {• We might prefer to distinguish based on the storage map for an element of the given type and type parameter rather than on the type and type parameters themselves. This would allow storage associated pointers under slightly more liberal circumstances. •}

When a variable is defined, its component storage units are defined. Conversely, when all of the component storage units of a variable are defined, the variable is defined. Locater storage units are always defined and indicate whether and which ordinary storage units the variable is composed of.

A scalar variable or element of an array variable always is a nonempty sequence of storage units. Thus, there is a first storage unit, there is a last storage unit (not necessarily different), each storage unit except the last has a successor in the sequence, and each storage unit except the first is the successor of a storage unit in the sequence. Unless covered by one of the following cases, that storage unit is an ordinary storage of kind determined by the type and type parameters of the variable.

1. If the type is COMPLEX, then the sequence consists of two storage units of kind corresponding to the type REAL with the same type parameters. The first storage unit contains the value of the real part; the second contains the value of the imaginary part.

2. If the type is CHARACTER and the length parameter *n* is greater than 1, then the sequence consists of *n* storage units of kind corresponding to the type CHARACTER and length parameter 1. If the length parameter is zero, then the storage unit is the same as for a zero-sized array of type CHARACTER and length parameter 1 (see below). {• If we a add a second parameter to CHARACTER to support Kanji, etc., then we should note that its value remains the same in both these cases. •}

3. If the type and type parameters indicate double precision real, then the sequence consists of two storage, the first of a kind specific to holding the first half of a double precision real value, the second of a kind specific to holding the second half of a double precision real value. Fortran provides no means for these halves to be separately defined or referenced, but they may become undefined separately due to storage mapping.

4. If the type is a derived type and the kind of storage units in the storage sequences of its components are all compatible (see below), then the storage sequence consists of the storage units of those components in the order they are declared. (I.e., the first storage unit of the first component is the first storage unit of the overall sequence, the first storage unit of component *i* is the successor in the overall sequence of the last storage unit of component *i*-1, and the last storage unit of the last component is the last storage unit of the overall sequence.) Note that if a component has attributes that give it a locater storage unit, it is this storage unit (and not the storage units located by it) that applies to this rule and the following. {• If we restore variant types or add union types, then we will get a storage map rather than a storage sequence, and change from storage sequence to storage map will have to be propagated to various points in this discussion. •}

5. If the type is a derived type and the kind of storage units in the storage sequences of its components are not all compatible, then the storage sequence consists of a single storage unit whose kind is determined not by the type and type parameters, but by the storage sequence of the components in the order they are declared. Note that this storage unit is different from the storage sequence which determines its kind (and thus differs from the previous case).

For arrays which are contiguous {• definition omitted here, but essentially equivalent to arrays which can be sequence associated in the current draft •}, its storage units also form a storage sequence in which the first storage unit of the first element in array element order is the first storage unit of the overall sequence, the first storage unit of element *i* in array element order is the successor in the overall sequence of the last storage unit of element *i*-1 in array element order, and the last storage unit of the last element in array element order is the last storage unit of the overall sequence. If the array is zero-sized (and thus has no elements), then the storage sequence consists of a special "null" storage unit of kind determined by the kind of the first storage unit elements of arrays of that type. If, in a storage sequence, the successor of such a "null" storage unit is a storage unit ("null" or not) of a compatible kind, the "null" storage unit is removed from the sequence.

{• This may need some cleaning up to make clear that "null" storage units are no interchangeable with non-"null" storage units, even when their kinds are compatible. •}

Certain kinds of storage units are compatible and can be used interchangeably in compatible storage maps. In particular, the following kinds of storage units are compatible:

1. the kind corresponding to the [default] INTEGER type
2. the kind corresponding to the [default] LOGICAL type
3. the kind corresponding to the default REAL type
4. the kind corresponding to the first half of a double precision value
5. the kind corresponding to the second half of a double precision value

The variables declared to in a COMMON block form a storage sequence, with the first storage unit of the first variable being the first storage unit of the overall sequence, etc. As with derived types, if the variable has an attribute that gives it a locater storage unit, it is the locater storage unit that is part of the overall storage sequence, and not the storage units that it locates.

A storage map is similar to a storage sequence, except that there may be multiple "first" storage units (all synchronized — see below), there may be multiple "last" storage units, and a storage unit may have multiple successors.

Synchronization is the process which converts multiple storage sequences into a storage map (or multiple storage maps into a combined storage map).

Two storage elements if

1. they are each a first storage unit in the same COMMON block (presumably declared in different scoping units);
2. they are the first storage units of variables that are EQUIVALENCEd;
3. they are the first storage units of an actual argument and dummy argument that are storage associated (synchronization lasts only as long as the argument association); {• replacing current sequence association •}
4. they are the successors of synchronized compatible storage units; or
5. they are compatible and have a common successor.

If two storage units are synchronized and one is the successor to a third storage storage unit, then the other is also a successor to that storage unit.

Two storage units which have a common successor must be compatible. {• I.e., EQUIVALENCE statements that would cause incompatible storage units to have common successors are not permitted. •}

A storage unit is said to follow a second storage unit if it is the successor of that second storage unit or of a third storage unit which follows the second storage unit.

Two storage units are disjoint if one of them follows the other.

When a storage unit is defined,

1. any storage unit of the same kind which is synchronized with it is defined with the same value, and

2. any storage unit in the same storage map which is not disjoint from it is undefined. {• This is the biggie that handles equivalencing of unlike data, etc. Its implications can be extensive •}

The first storage unit of the first variable declared in a COMMON block must not be the successor to any other storage unit. {• I.e., EQUIVALENCE can't extend before the beginning of a COMMON block. •}

Storage units from two different COMMON blocks must not be in the same storage map. {• I.e., you can't equivalence variables in two different COMMON blocks. •}

A partial storage map is that part of the storage map that is defined in a single scoping unit. Two partial storage maps are said to be compatible if synchronizing the first storage units of two partial storage maps results in a storage map in which each storage unit from one scoping unit is synchronized with a compatible storage unit from the other scoping unit. All partial storage maps of a named COMMON block must be compatible. {• I.e., a named COMMON block must have the same length in each program in which it appears. •}

Any two storage units being synchronized by an EQUIVALENCE statement must be in different storage maps if they are removed from that EQUIVALENCE. {• Prevents redundant or inconsistent EQUIVALENCEs •}

A locater storage unit must be disjoint from all incompatible storage units in the same storage map. {• You can't EQUIVALENCE a nonpointer onto a pointer. •}

{• Note that the result variables corresponding to different entry points in a function subprogram can once again be storage associated (i.e, synchronized). •}

{• We may (or may not) need some additional work on pointers to protect optimization (depending on exactly which optimization protecting attributes we include in the standard). •}

{• We may wish to require a processor to be able to be able to detect if a partial storage map contains storage units which are neither disjoint nor synchronized and compatible. •}

Ω