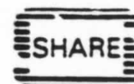
SESSION REPORT



1

SHARE NO.	SESSION NO.	SESSION TITLE	ATTENDANCE
	D732	FORTRAN Project Swap Meet	27-28 Aug 1987

PROJECT	SESSION CHAIRMAN	INST. CODE
	Sunnie Sund	

SLAC, BIN 96, Box 4349, Stanford, CA 94305, (415) 854-3300

SESSION CHAIRMAN'S COMPANY, ADDRESS, and PHONE NUMBER

HIGH-LEVEL FORTRAN

Frederick W. Nagle

U.S. Department of Commerce
NOAA/NESDIS Systems Design and Applications Branch
1225 West Dayton Street
Madison, Wisconsin 53706

Installation Code: WP

FORTRAN

D732

The Schwerdtfeger Library
1225 W. Dayton Street
Madison, WI 53706

High-Level Fortran (HLF) is a pre-compiler (not a compiler and not an interpreter) which enhances the Fortran language in that it implements two types of variables, VECTOR and MATRIX, not available in conventional Fortran-77. Lengthy matrix expressions can be evaluated with the aid of the STACK (see below) which permits partial results to be computed, set aside, and later retrieved to complete an expression.

HLF accomplishes its purpose by inspecting the submitted high-level source code, and modifying it where necessary to make it acceptable to an existing Fortran compiler. Those Fortran statements which are already acceptable to a standard compiler are left unchanged.

The following is a brief summary of salient HLF capabilities which go beyond those of conventional Fortran:

1. The variable types VECTOR*4, VECTOR*8, MATRIX*4, and MATRIX*8 are available and may appear in arithmetic expressions as such. Vector- and matrix-valued functions, returning vectors and matrices through their own names in arithmetic expressions, are also allowed. Notation exists for the dot and cross products of two vectors. A row, column, or main diagonal of a matrix may be directly referenced as a sub-matrix. Matrix concatenation is allowed, i.e., AABB = AA // BB, where AABB is the matrix AA augmented columnwise by BB.

2. The dimensions of a matrix are not fixed by a DIMENSION or MATRIX statement. The dimensionality of a matrix may be changed dynamically as the program executes. The dimensions of a matrix-valued function can also vary dynamically.

The Schwerdtfeger Library
1225 W. Dayton Street
Madison, WI 53706

3. Masking and extraction operations are available within the language. It is not necessary to invoke subroutines for this purpose, e.g., A <AND> B denotes the 'logical AND' of A and B.

4. Sexadecimal (hex) constants may appear in arithmetic statements, i.e., they need not be defined in DATA statements.

5. The swap or exchange operator using the double equality sign allows two variables of any type, including vectors and matrices, or rows and columns of matrices, to exchange values, e.g., A == B.

6. Somewhat curiously, any Fortran SUBROUTINE written in an existing Fortran language, which returns a vector or matrix as its output, may be invoked as a HLF matrix- or vector-valued FUNCTION, subject to very mild restrictions.

7. Source code may be submitted in either UPPER or lower case.

8. On-line comments are allowed, i.e., a line of Fortran source code may end with a semi-colon (;) or exclamation mark (!), after which the remainder of that line may be used for comments.

As American National Standard Language Fortran-8x evolves over the next several years, efforts will be made to maintain HLF in a status wherein it is at least not incompatible with the ANSI standard, i.e., a program written in the new Fortran-8x will be accepted by HLF.

The rules governing the choice of symbols for constants and variables are essentially those with which the reader is acquainted. In the absence of a user-provided IMPLICIT statement, HLF assumes that variables beginning with the letters A-H and O-Z are REAL*4, the remainder being INTEGER*4.

If the user wishes to depart from the foregoing implicit name rule, he can provide his own IMPLICIT statement, or provide explicit type statements. The permissible variable types in HLF are:

1. CHARACTER
2. LOGICAL*1
3. LOGICAL*4
4. INTEGER*2
5. INTEGER*4
6. REAL*4 (or REAL)
7. REAL*8 (or DOUBLE PRECISION)
8. COMPLEX
9. VECTOR*4 (or VECTOR)
10. VECTOR*8
11. MATRIX*4 (or MATRIX)
12. MATRIX*8

VECTORS:

Let us first discuss VECTORs in HLF. The term VECTOR is used herein to mean only three-dimensional vectors in Cartesian space. The term is not used to denote a row, column, or eigen-vector of a matrix. Three-dimensional space is not considered a special case of n-dimensional space because the vector (cross) product of two vectors is defined only in three-dimensional space, and unlike the scalar product of two vectors, it is not generalized to higher dimensions.

Just as a REAL in Fortran occupies a single word of memory, and a COMPLEX number two words, so a VECTOR in HLF occupies three memory words containing, respectively, the x-, y-, and z-components of that vector. A vector variable, if dimensioned, signifies an array of vectors. The permissible operations on a vector are addition (+), subtraction (-), scalar (dot) multiplication (*), and vector (cross) multiplication (**) or (<x>).

Let us consider a few examples of vector arithmetic in HLF. In the following, vector variables will be capitalized for clarity in reading the examples, as is permitted but not required in HLF.

Firstly, vectors, like other variables in Fortran, may be typed either by the implicit name rule, or by specific typing statements.

```
      Implicit Vector*8 (V), double precision (d)
      Vector*4 UA,UB
      Vector*8 (f)VUNIT8
      UA = UB + VUNIT8(VA ** VB) - VC
```

The typing statements indicate that any variable beginning with the letter V is a VECTOR*8, i.e., a vector each of whose three components is a REAL*8 quantity. Moreover, the vectors UA and UB are VECTOR*4, i.e. their components are REAL*4 values, and finally the vector-valued FUNCTION VUNIT8 returns a VECTOR*8 through its own name appearing in an expression. As any Fortran programmer will instantly discern, the replacement statement first computes the cross product VA x VB, normalizes (unitizes) this vector to one having a length of unity, adds to it the vector UB, and subtracts the vector VC, storing the result of this arithmetic into the vector UA.

Note that vector-valued functions must specifically appear in a VECTOR type statement, preceded by the symbol (f). The reason for this requirement is discussed below under MATRICES.

In the case of the foregoing example, the output of the HLF processor would include the following:

```
*
      REAL UA(3),UB(3)
C     VECTOR*8 (F)VUNIT8
      DOUBLE PRECISION V$81(3),V$82(3),V$83(3),V$84(3)
      DOUBLE PRECISION  VA(3),VB(3),VC(3)
*           ***
C     UA = UB + VUNIT8(VA ** VB) - VC
      CALL V$CROS(VA, -8,VB, -8,V$81, -8)
      CALL VUNIT8(V$81,V$82)
      V$83(1) = UB(1)+V$82(1)
      V$83(2) = UB(2)+V$82(2)
      V$83(3) = UB(3)+V$82(3)
      V$84(1) = V$83(1)-VC(1)
      V$84(2) = V$83(2)-VC(2)
      V$84(3) = V$83(3)-VC(3)
      UA(1) = V$84(1)
      UA(2) = V$84(2)
      UA(3) = V$84(3)
```

It may be seen that the HLF processor initially creates the temporary three word holding arrays V$81,V$82, V$83, and V$84 to contain intermediate vector results. More would have been defined had they been needed. These same arrays would later be used in evaluating another vector expression. The submitted statement then is commented out, and is replaced by the code which follows. The subroutine V$CROS computes the cross product of two vectors, storing the resulting vector in V$81. This is then normalized to V$82 by VUNIT8. Note that the vector-valued function VUNIT8 is changed to a subroutine call. The remaining steps of the computation are performed in-line, one component at a time, until the final result is stored into the vector UA.

In most Fortrans, mixed mode arithmetic normally produces a result having the mode of the "higher" operand, i.e., the sum or product of a REAL*4 and REAL*8 is REAL*8. Likewise, in the case of HLF vector arithmetic, mixed mode arithmetic involving VECTOR*4 and VECTOR*8 leads to a VECTOR*8 result. Incompatible types are flagged as errors by the pre-compiler, i.e., an attempt to add a REAL to a VECTOR will produce a diagnostic. However, a vector may be multiplied or divided by a scalar, as in VTERRA below.

A vector on the left side of a replacement statement may be set equal to a REAL or INTEGER expression on the right, e.g.,

```
UA = 10. + tempo
```

will set all three components of UA to the current value of the expression on the right. This feature is most commonly used in setting a vector to zero.

Any vector may be expressed in terms of its three components by means of the vector-valued function VEC4 or VEC8, e.g.,

```
UA = VEC4( 1., 2., -.3)
```

A single component of a vector is itself a REAL*4 or REAL*8 value, and may be used as such, e.g.,

```
UA(3) = 100.
UB = VEC4( UB(1), UB(2), UB(3))
```

The second statement above is frivolous, since it sets each component of UB to its current value.

An expression involving vectors or vector functions may be used as an argument to a subprogram:

```
rough = tough + glitch( VA*VB, VA**VB)
```

In this case, the first argument to the function 'glitch' is the scalar (dot) product VA*VB, and the second argument is the vector (cross) product VAxVB.

We conclude our discussion of vectors by displaying a practical and much-used vector-valued function which computes the point on the ground viewed by a scanning satellite when an initial, final, and current position are known in vector form (VSAT1, VSAT2, and VSAT, respectively), as well as the left-to-right scan angle.

```
C
      Vector Function VTERRA*4 (VSAT1,VSAT2,VSAT,scan)
c
c     Capitalized values are vectors;  the remainder are scalars.
c     To compute the point on the ground which a scanning satellite
c     views when its position is VSAT, given VSAT1 and VSAT2 as
c     its positions at the start and finish of an image.
C     The scan angle is negative to the left of orbit.
C     All vectors are in the celestial coordinate system.
C
*     The arcsin calculated by statement 10 is the zenith angle of
*     the satellite from the ground point to be determined.
*
c     If this routine is called by an assembler or conventional
c     Fortran routine, the call is...
C
c     CALL VTERRA(VSAT1, VSAT2, VSAT, scan,      VGRND)
c
      Implicit Vector*4 (u-v)
      Vector*4 (f)VUNIT4 *
      data init/0/ c
      arcsin(x) = 57.29578*atan2( x, sqrt(1.-x*x))
c
      if(init .eq. 0) then
      cd1 = sqrt(vsat1(1)**2 + vsat1(2)**2 + vsat1(3)**2)
      cd2 = sqrt(vsat2(1)**2 + vsat2(2)**2 + vsat2(3)**2)
      cd = .5*(cd1 + cd2)  ;  mean distance, earth center to satellite
      UVORB = VUNIT4(VSAT1**VSAT2)   ;  unit vector orbital plane
      ratio = cd/6371.   ;  ratio of cen dist to earth radius
      init = 1
      end if
c
c     Subsequent entries execute only the following...
10    gamma = arcsin( ratio*sine(scan)) - scan
      VTERRA = 6371.*(VSAT*cosine(gamma)/cd - UVORB*sine(gamma))
      return
      end
```

When modified by HLF, this routine looks like:

```
C     HLF SUPPLEMENTARY PRINT-OUT ********************
C     VECTOR FUNCTION VTERRA*4 (VSAT1,VSAT2,VSAT,SCAN)
      SUBROUTINE VTERRA(VSAT1,VSAT2,VSAT,SCAN,V$ERRA)
C     TYPING STATEMENTS ...
C     IMPLICIT VECTOR*4 (U-V)
      INTEGER J$TACK,N$A1,N$A2,N$1S,N$2S,M$NTAG,M$BASE,N$ADDR
C     REAL DECLARATIONS ...
      REAL V$ERRA(3)
C     VECTOR*4 (F)VUNIT4
      DOUBLE PRECISION D$RET
      REAL V$41(3),V$42(3),V$43(3),V$44(3),V$45(3)
      REAL VSAT1(3),VSAT2(3),UVORB(3),VSAT(3)
C     DIMENSIONS ...
```

```
C     COMMON DECLARATIONS ...
      COMMON/R$ES/ J$TACK(   100)
C     DATA STATEMENTS ...
      DATA INIT/0/
      ARCSIN(X) = 57.29578*ATAN2( X, SQRT(1.-X*X))
C
      IF(J$TACK(6).LT.     100) J$TACK(6) =      100
      M$NTAG = J$TACK(3)
      M$BASE = J$TACK(4)
      J$TACK(3) = J$TACK(1)
      J$TACK(4) = J$TACK(2) C
      IF(INIT.EQ.0)THEN
      CD1 = SQRT(VSAT1(1)**2 + VSAT1(2)**2 + VSAT1(3)**2)
      CD2 = SQRT(VSAT2(1)**2 + VSAT2(2)**2 + VSAT2(3)**2)
      CD = .5*(CD1 + CD2)
C
C     UVORB = VUNIT4(VSAT1**VSAT2)
      CALL V$CROS(VSAT1, -4,VSAT2, -4,V$41, -4)
      CALL VUNIT4(V$41,V$42)
      UVORB(1) = V$42(1)
      UVORB(2) = V$42(2)
      UVORB(3) = V$42(3)
      RATIO = CD/6371.
      INIT = 1
      END IF
10    GAMMA = ARCSIN( RATIO*SINE(SCAN)) - SCAN
C
C     VTERRA = 6371.*(VSAT*COSINE(GAMMA)/CD - UVORB*SINE(GAMMA))
      R$41=COSINE(GAMMA)
      R$42=SINE(GAMMA)
      V$41(1)=VSAT(1)*R$41
      V$41(2)=VSAT(2)*R$41
      V$41(3)=VSAT(3)*R$41
      V$42(1)=V$41(1)/CD
      V$42(2)=V$41(2)/CD
      V$42(3)=V$41(3)/CD
      V$43(1)=UVORB(1)*R$42
      V$43(2)=UVORB(2)*R$42
      V$43(3)=UVORB(3)*R$42
      V$44(1) = V$42(1)-V$43(1)
      V$44(2) = V$42(2)-V$43(2)
      V$44(3) = V$42(3)-V$43(3)
      V$45(1)=V$44(1)*6371.
      V$45(2)=V$44(2)*6371.
      V$45(3)=V$44(3)*6371.
      V$ERRA(1) = V$45(1)
      V$ERRA(2) = V$45(2)
      V$ERRA(3) = V$45(3)
C
      J$TACK(1) = J$TACK(3)
      J$TACK(2) = J$TACK(4)
      J$TACK(3) = M$NTAG
      J$TACK(4) = M$BASE
C
```

```
RETURN
END
```

Note that HLF comments out the VECTOR FUNCTION statement, and replaces it with a SUBROUTINE header card. The resulting vector value of the function is returned to the calling routine as a subroutine argument whose name is derived from that of the function, with $ replacing the second character.

References to the array J$TACK are stack control statements, not actually needed in this application, but discussed below under MATRICES.

MATRICES:

Perhaps of greater importance even than vector arithmetic is the ability of HLF to deal with matrix expressions. A subroutine P$ROC (see below) contained in the link library accompanying the HLF package performs most of the matrix arithmetic, and in addition manages the stack (see below). Matrix variables will be capitalized in the following examples, though of course capitalization is not required.

Unlike VECTOR variables in HLF, MATRIX variables cannot be defined by the implicit name rule because the latter conveys no dimensioning information. Matrices can be typed only by a specific MATRIX*4 or MATRIX*8 type statement, which contains the actual dimensions, and perhaps also the "working" dimensions which the matrix is presumed to possess at any instant during execution.

A matrix in HLF is always presumed to be rectangular, i.e., it has two subscripts. A MATRIX type statement having only a single subscript (n) means (n,1):

```
MATRIX*4 GLITCH (10)        means        MATRIX*4 GLITCH(10,1)
```

If a MATRIX type statement contains more than two subscripts, the effect is to define an ARRAY of matrices, i.e.,

```
MATRIX*8 GUMBO(3,4, 10)
```

defines an array of ten matrices, each of which has three rows and four columns. A MATRIX type statement may contain not more than four subscripts, two intrinsic subscripts stating the shape of the matrices, and not more than two extrinsic subscripts defining the array containing these matrices.

Like vector-valued functions, matrix-valued functions appear in a type statement preceded by the symbol (f). The reason for this should be clear. Most Fortrans typically interpret a symbol followed by a left parenthesis as either a dimensioned variable or as a function, and distinguish between these two by noting whether the symbol is or is not dimensioned. In the case of HLF, however, a symbol may be BOTH dimensioned and also be an external function. To resolve this ambiguity, which does not occur in conventional Fortrans, the symbol (F) is required in the MATRIX*n or VECTOR*n card. The reader might suppose that (F) would not be needed in the case of a vector-valued function, since a single vector is not formally dimensioned. However, the occurrence of a symbol like VXYZ(J) is ambiguous, since it is

unclear if it refers to a vector function with argument J, or to the J-th component of a local vector, which is in fact a real number. The presence or absence of the symbol (F) resolves this question.

The permissible operations involving matrices are addition (+), subtraction (-), multiplication of two matrices (*), and concatenation (//). These operations are accomplished by the subroutine P$ROC contained in the link-library accompanying the HLF system. A matrix may also be multiplied or divided by a scalar, the effect being to multiply or divide each element by that scalar.

The rules of mixed mode arithmetic with matrices are analogous to those for VECTORs and REALs, i.e., the sum, difference, product, or concatenation of a MATRIX*4 and a MATRIX*8 is MATRIX*8. An exception is multiplication or division by a scalar, in which case the resulting matrix has the same length attribute as the given matrix, regardless of the length attribute of the scalar.

The dimensionality, or shape, of a matrix or matrix-valued function may fluctuate during execution. For this reason, the programmer, in addition to the "true" dimensions of a matrix, can also optionally provide "working" dimensions, expressing the shape of the matrix at any instant. These are provided by using a semi-colon (;) within the type statement, followed by two integer variables whose value is defined during execution. A matrix-valued function has only working dimensions, because no space is allocated to contain its result, which is sent to the stack when it has been computed.

```
      MATRIX*4 GUMBO(10,10; ii,jj), XX(6,9)
*
      ii = 6
      jj = 9
      GUMBO = XX
*
```

The matrix XX has the inflexible dimensions (6,9), whereas GUMBO has (10,10) as its maximum size, but its instantaneous dimensionality is (ii,jj). The programmer is thus permitted, indeed required, to define the integers ii and jj in order to correctly shape the receiving matrix GUMBO. The 54 matrix elements of XX are stored into the first 54 elements of GUMBO. It is the true dimensions (preceding the semi-colon, if any) which allocate storage space. The working dimensions, if given, govern the dimensionality only during execution.

The operation of concatenation is somewhat like concatenation for CHARACTER variables. A matrix of dimensions (m,n) may be concatenated with one of dimensions (m,k), the result being a matrix with dimensions (m, k+n), e.g.,

```
      MATRIX*4 GLITCH(10,20; 10,ii), HITCH(10,20; 10,jj),
      1  NITCH(10,20; 10,kk)
*          ***
      ii = 4
      jj = 7
      kk = ii + jj
      NITCH = GLITCH // HITCH
```

where NITCH is set equal to the four columns of GLITCH followed by the seven columns of HITCH. Concatenation is allowable only if the matrices involved have the same number of rows. It is not necessary that the two matrices have equal length attributes, so that if HITCH in this example were MATRIX*8, the operation would still be allowable. Either or both of the operands may be a matrix-valued function.

As with vectors, an expression involving matrices or matrix-valued functions may be used as a subprogram argument. Consider the following trivial program to compute the inverse of the transpose of the product of two matrices.

```
     Program Triv
     Matrix MRES(30,30; jj,jj)
     Matrix*8 (f)MINVER(jj,jj), (f)MTRAN8(jj,jj),
     1  MA(24,5; jj,5), MB(5,24; 5,jj) * *    Assume that MA and MB have been
defined.
     jj = 20   ;   assigns value to jj in the type statements
     MRES = MINVER( MTRAN8( MA*MB, jj,jj), jj)
     stop
     end
```

The 20x20 matrix product MA*MB is first transposed by the transposition function MTRAN8, and this result is in turn inverted by the inversion function MINVER. This example further shows that a matrix-valued function may have its shape determined by working dimensions to which actual values are assigned during execution. The result returned by a matrix-valued function is actually sent to the stack, from which it is retrieved for later inclusion in a final result.

Let us next look at the foregoing example as it would be interpreted and modified by HLF. The output would be:

```
C     HLF SUPPLEMENTARY PRINT-OUT ********************
      PROGRAM TRIV
C     TYPING STATEMENTS ...
      INTEGER J$TACK,N$A1,N$A2,N$1S,N$2S,M$NTAG,M$BASE,N$ADDR
C     REAL DECLARATIONS ...
C     MATRIX MRES(30,30; JJ,JJ)
      REAL MRES(30,30)
C     MATRIX*8 (F)MINVER(JJ,JJ), (F)MTRAN8(JJ,JJ),
C     1  MA(24,5; JJ,5), MB(24,5; JJ,5)
      DOUBLE PRECISION MA(24,5),MB(24,5)
      DOUBLE PRECISION D$RET
C     DIMENSIONS ...
C     COMMON DECLARATIONS ...
      COMMON/R$ES/ J$TACK(   100)
C     DATA STATEMENTS ...
      JJ = 20
C
C     MRES = MINVER( MTRAN8( MA*MB, JJ,JJ), JJ)
C
```

```
      IF(J$TACK(6).LT.     100) J$TACK(6) =      100
      M$NTAG = J$TACK(3)
      M$BASE = J$TACK(4)
      J$TACK(3) = J$TACK(1)
      J$TACK(4) = J$TACK(2)
C
      J$TACK(1) = J$TACK(3)
      CALL P$ROC(J$TACK,MA,JJ,5,12,8,MB,5,JJ,12,8,92,'M$81',12,D$RET)
      CALL P$ROC(J$TACK,N$A1,JJ,JJ,12,0,0,0,0,0,0,64,'M$82',12,D$RET)
      I$41 = N$ADDR('M$81')
      CALL MTRAN8(J$TACK(I$41),JJ,JJ,J$TACK(N$A1))
      CALL P$ROC(J$TACK,N$A1,JJ,JJ,12,0,0,0,0,0,0,64,'M$83',12,D$RET)
      I$42 = N$ADDR('M$82')
      CALL MINVER(J$TACK(I$42),JJ,J$TACK(N$A1))
      CALL P$ROC(J$TACK,MRES,JJ,JJ,11,4,0,0,0,0,'M$83',126,0,0,D$RET)
      STOP
      END
```

Those HLF statements which are patently illegal in standard Fortran are commented out. DIMENSION or REAL type statements are used to generate the needed storage, and certain stack-control statements involving J$TACK are evident. The first call to P$ROC performs the multiplication of MA and MB with JJ as a dimension. The resulting product is stored in the stack, and a tag or "claim check" M$81 is assigned to this product identifying it for later use by the calling routine when this product is to be retrieved. The second call to P$ROC merely allocates space to contain the result of the transposition function MTRAN8, and the tag M$82 is assigned to this anticipated result. The integer variable N$A1 returned by P$ROC is the location within the stack where the transpose is to be placed. The function N$ADDR returns the stack location I$41 where MA*MB was stored. The transposition function MTRAN8, called as a subroutine, is then invoked with the stack location J$TACK(I$41) as input, and the location J$TACK(N$A1) where the result is to be sent. The third P$ROC call allocates space where the inversion routine MINVER may place its result, and the function N$ADDR returns the location within the stack I$42 corresponding to the tag of the transpose M$82. We next invoke MINVER to compute the inverse, where J$TACK(I$42) marks the location of the input to be inverted, and J$TACK(N$A1) the location of the result. This inverse has the claim check M$83, and the final call to P$ROC moves this result to its final destination MRES.

As the reader may have inferred, the "claim check" serves a purpose analogous to the claim check which a concert-goer receives when he checks his coat in the cloak-room at a concert. He has no idea where the attendant has actually placed his coat, but he expects that when he presents the claim check to the attendant after the concert, the attendant will use the identifier on the claim check to find the coat. Similarly, the user's program has no knowledge where within the stack an intermediate result has been placed, but P$ROC uses the "claim check" as a means to find the desired result, along with other information such as the number of rows or columns, length attribute, etc., in order to combine it with other data to complete an arithmetic expression.

We are now in a position to discuss the stack more fully.

THE STACK:

In the course of performing matrix arithmetic, it follows that a number of intermediate results will be generated for which temporary storage must be provided. Since these intermediate matrix results are not explicitly named by the user, there will of course be no dimension or common statements written by the user to allocate storage for them, for the programmer sees no need to define them formally.

For example, in the following matrix replacement statement, where MINVER is a matrix-valued inversion function

A = B*C + D*E + MINVER(GG,jj)

it is necessary first to compute the inverse of GG, which must then be added to the products B*C and D*E. Each of these partial results must be set aside somewhere while other intermediate results are generated, and finally the various intermediate results must be combined obtain the final result A.

In order to provide storage for intermediate results, HLF utilizes the so-called STACK - an array whose size is declared by the programmer at the outset, but whose management thereafter need not be the concern of the user. At the end of execution, the user can ask that the largest utilized amount of the stack be displayed, so that he can adjust the size upward or downward if it is inadequate or grossly excessive. The fifth stack word J$TACK(5) contains this information expressed in bytes, or the user may simply CALL MFSTAK(0).

Prior to each operation requiring stack space, a HLF program will ascertain whether the anticipated operation will result in an overflow of the stack, and if so, a warning message is displayed before the operation occurs, so that the user will be informed of the probable reason for failure of his task. He can then recompile his job, requesting a larger stack size.

The subroutine P$ROC performs most matrix operations on behalf of the user's program, and also manages the stack. When P$ROC has computed an intermediate result, it records its attributes (numbers of rows and columns, word length, location where stored, etc.), and returns to the calling program the tag or 'claim check' whereby the user's program can request the quantity at a later stage.

Clearly, the stack must be carefully managed. In evaluating a matrix expression, a HLF program may invoke an external function which itself may or may not use the stack, and this function in turn may call yet another which may or may not use it, etc. Hence, P$ROC must insure that these in-depth stack usages do not overlap or conflict. The logical structure used to maintain integrity of stack usage is reminiscent of the Stack Pointer and Base Pointer used in governing the stack segment on a Personal Computer equipped with an 8088 chip (push-pop logic), wherein the top of the stack used at one program level becomes the bottom location available to a lower-level routine. Statements in the HLF output listing referring to J$TACK are stack control statements needed to avoid stack-usage conflicts. The logical similarity

between HLF stack management, and management of the stack segment of a PC, is possibly significant, and is discussed below.

Sub-Matrices:

A row, column, or main diagonal of a matrix may be referenced directly as a sub-matrix without recourse to an external subprogram. The programmer uses a defined value for the desired row (column), and a dollar sign ($) for the column (row). A dollar sign in both subscripts references the main diagonal:

```
Matrix AA(10,10), X(10), MSUB(5,2), (f)MSUBI4(5,2)
X = AA(4,$) + AA($,6)
```

This statement defines X as the sum of the fourth row and the sixth column of AA.
*

```
MSUB = MSUBI4(AA,10,10,  2,10,2, 5,10,5)
```

This statement defines a sub-matrix MSUB by means of the matrix-valued function MSUBI4, extracting rows 2,4,6,8,10, and columns 5,10 of AA. The last six arguments of MSUBI4 are DO-loop indices which select the rows and columns to be included in the sub-matrix.

A row or column may be PASSED to a subprogram as an argument, but in general a row or column cannot be DEFINED as a subprogram argument. Unlike the elements of a matrix column, the elements of a row in general do not occupy sequential memory locations, but are separated in memory by a distance depending on the number of rows. Hence,

```
call jello(tom,dick,harry,  AA(7,$) )
```

will correctly pass to 'jello' the scalars 'tom,dick,harry', and the seventh row of AA as a sub-matrix. The seventh row is first copied into consecutive locations within the stack, and it is from this stack location that it is then passed to 'jello'. However, no provision currently exists to return a row or main diagonal defined by 'jello' backward through the stack, and thence to its location in discontiguous locations in AA. Thus, if the subroutine 'hello' DEFINES a ten-word array X which we wish inserted into the main diagonal of AA, we must

```
call hello( tom,dick,harry,   X)
AA($,$) = X
```

By analogy, the same restriction applies to a matrix column, although this restriction is removable in principle, since the elements of a column occupy contiguous memory.

A matrix on the left of an equality may be set equal to a matrix or matrix expression on the right having the same number of rows, but having more or fewer columns. The effect is to copy the number of columns of the left or right matrix, whichever is FEWER. For example,

```
Matrix*4 SHORT(10,10), LONG(10,20)
...
SHORT = LONG
```

will set SHORT equal to the first ten columns of LONG. On the other hand,

```
LONG = SHORT
```

will set the first ten columns of LONG equal SHORT, leaving unchanged columns 11-20 of LONG.

As with vectors, a matrix or sub-matrix on the left side of a replacement statement may be set equal to a scalar expression on the right. The effect is to set every element of the matrix to the value of the scalar expression. For instance,

```
AA = 0.
AA($,$) = 1.
```

will first set the entire matrix AA to zero, and will then create 1's down the main diagonal. (The matrix-valued function MIDEN4 can also be used to create the identity matrix.)

We conclude this section with a comparison of standard Fortran and HLF, both used to code four matrix equations in one version of a Kalman filter. The four matrix equations in algebraic notation are

$$
\begin{aligned}
P' &= TPT' + Q \\
G &= P'H'(HP'H' + R)^{-1} \\
x_k &= Tx_{k-1} + G(z - HTx_{k-1}) \\
P^k &= P'^{k-1}GHP'
\end{aligned}
$$

where the prime indicates transposition. We shall not attempt here to define the meaning and shapes of all the matrices involved, but simply to encode these matrix operations using standard and High-Level Fortrans. First, in standard Fortran, with the assumption that the programmer has available a subroutine MATMUL which multiplies two matrices, another MATSUM which computes the sum or difference of two matrices, and MINVER which inverts a matrix, the coding for these equations might look like:

```
C     FIRST MATRIX EQUATION
      CALL MATMUL(T,P,XX1, 10,10,10)
      CALL MATMUL(XX1,TT,XX2, 10,10,10)
      CALL MATSUM(XX2,Q,PT, 1., 10, 10)
C
C     SECOND MATRIX EQUATION
      CALL MATMUL(H,PT,XX1, 5,10,10)
      CALL MATMUL(XX1,HT,XX2, 5,10,5)
      CALL MATSUM(XX2,R,XX1, 1., 5,5)
      CALL MINVER(XX1,5,XX2)
C     THIS IS THE MATRIX INVERSE IN THE SECOND MATRIX EQUATION
      CALL MATMUL(HT,XX2,XX1,10,5,5)
      CALL MATMUL(PT,XX1,G, 10,10,5)
C
```

```
C     THIRD MATRIX EQUATION
      CALL MATMUL(H,T,XX1,5,10,10)
      CALL MATMUL(XX1,X,XX2, 5,10,1)
      CALL MATSUM(Z,XX2,XX1, -1., 5,1)
      CALL MATMUL(G,XX1,XX2, 10,5,1)
      CALL MATMUL(T,X,XX1, 10,10,1)
      CALL MATSUM(XX1,XX2,X, 1., 10, 1)
C
C     FOURTH MATRIX EQUATION
      CALL MATMUL(G,H,XX1, 10,5,10)
      CALL MATMUL(XX1,PT,XX2, 10,10,10)
      CALL MATSUM(PT, XX2, P, -1., 10,10)
```

In contrast, the same computational steps encoded in HLF would have an appearance almost identical to the above matrix equations, i.e.,

```
      PP = T*P*TT + Q
      G = PP*HT*MINVER(H*PP*HT + R, 5)
      XP = T*X + G*(Z - H*T*X)
      P = PP - G*H*PP
```

Quite aside from the fact that the HLF coding is far more succinct and self-documenting than the standard code, a salient advantage of the HLF code is that the dimensions of the matrices involved can be changed during execution, so that on the next pass the matrices P and T, for example, could be 8x8 rather than 10x10, etc. In standard code, it is somewhat awkward to change the dimensionality of locally-defined arrays during execution. Speed of execution on the two versions is competitive.

VECTOR AND MATRIX FUNCTION-TYPE SUBPROGRAMS

Just as ordinary Fortrans allow function subprograms which return REAL, INTEGER, or LOGICAL values, so it is possible in HLF to define function subprograms which return VECTOR or MATRIX values as well.

Matrix Functions:

To DEFINE a MATRIX function subprogram, the following general format is used:

```
      MATRIX FUNCTION MGUMBO*8 (arg1, arg2, ...M,N,... )
      MATRIX*8 MGUMBO(M,N)
      ...
      DO 100 I = 1,M
      DO 100 J = 1,N
      ...
100   MGUMBO(I,J) = xxx
      ...
      or
      MGUMBO = some matrix-valued expression
      ...
      RETURN
      END
```

The function name must appear in a MATRIX type statement which contains the dimensions of the result. These dimensions may be formal arguments defined by the input list, PARAMETERs, or literal constants.

Within the CALLING program, the function name must appear in a MATRIX type statement, preceded by (F)

```
MATRIX*8 MRESLT(10,10), MA(10,10), (F)MGUMBO(10,10)
...
MRESLT = MA + MGUMBO( args... )
...
```

The MATRIX*8 statement as shown serves to inform the user's program that the functional value returned by MGUMBO will be a MATRIX*8 10x10 result, but no actual space within the user's area will be allocated for it. (The functional value will be returned to the stack instead.)

***

Vector Functions:

Within the VECTOR FUNCTION subprogram, the following general arrangement is used. Note that the function name does NOT appear in a type statement, since the type is implied by the functional header statement.

```
VECTOR FUNCTION VGUMBO*4 ( args... )
...
VGUMBO(1) = aaa
VGUMBO(2) = bbb
VGUMBO(3) = ccc
...
or
...
VGUMBO = (some vector expression)
...
RETURN
END
```

Within the CALLING program, the function name must appear within a VECTOR type statement, preceded by (F), e.g.

```
VECTOR*4 VAL,VA,VB, ..., (F)VGUMBO, ...
...
...
VAL = VA + VB + VGUMBO (args )
```

***

The user may discover that HLF actually restructures a VECTOR or MATRIX function subprogram into a SUBROUTINE. HLF alters the FUNCTION header card from

```
MATRIX (or VECTOR) FUNCTION GUMBO*n (a,b,c...)
```

to

```
SUBROUTINE GUMBO(a,b,c...,  G$MBO)
```

creating an output argument whose name is that of the given function, but with $ as the second character.

As a matter of fact, it is an interesting and possibly confusing circumstance that any existing SUBROUTINE, written in some conventional Fortran, and possessing the property that it returns its result (vector or matrix) as the LAST formal parameter in its argument list, can be invoked as a FUNCTION by a HLF program.

Caution:

In the case of both VECTOR and MATRIX function subprograms, as with other Fortrans, it is necessary that the function name appear at least once on the left side of an equality, or in any event, the separate elements must be defined in some manner.

Statement-type functions involving vector or matrix variables are not permitted in HLF. A vector- or matrix-valued function must be an external sub-program.

MISCELLANEOUS FEATURES:

HLF affords the user a few other minor capabilities. One of these is the swap or exchange operation within a single statement, denoted by the double equality sign ==, e.g.,

```
A = = B
```

will cause the variables A and B to exchange values. The two variables involved may be integers, reals, vectors, or matrices, or rows and columns of matrices, but must be of the same type and word length, e.g., both REAL*4, MATRIX*8, etc. For instance, to exchange two rows of a matrix, one could use

```
MAT(I,$) == MAT(J,$).
```

COMPLEX variables cannot be swapped in this manner, nor can ordinary scalar arrays. Arrays must be typed as VECTOR or MATRIX in order to be 'swappable'. The restriction on swapping complex variables is to be removed.

***

HLF will also accept the .BUT. connector in lieu of the .AND. connector within a Fortran test, e.g.,

```
IF(RELHUM.GT.85. .BUT. ALBEDO.LT.0.5) GO TO 50
```

The two conjunctions BUT and AND mean the same in the English language, the choice depending on the degree of similarity or antithesis in the mind of

the speaker. In this example, a high relative humidity may suggest cloudiness, but a low albedo tends to contradict this assumption.

HLF simply replaces .BUT. with .AND.

MASKING AND SHIFTING OPERATIONS:

HLF allows the user to perform certain Boolean or masking operations on 32-bit (4-byte) quantities (INTEGER*4, REAL*4, or LOGICAL*4). The operations allowed are 'logical OR', 'logical AND', and 'exclusive OR', denoted respectively by the operators

<OR> <AND> <XOR>

The result of such an operation is an INTEGER*4, regardless of the type of the operands involved. In the hierarchy of operations, <OR> and <XOR> have the same rank as the arithmetic sum or difference operators (+) and (-), whereas the <AND> operator has the same rank as the arithmetic operator for multiplication (*). For example,

J = A <OR> B <AND> C

would be construed as A <OR> (B <AND> C), i.e., the 'logical-AND' of B and C would first be computed, and this result would then be logically OR'ed with A. The final result would be an integer, and the value of J would then depend on the type of variable that J might happen to be.

The truth table for the three types of operations are

	OR	XOR	AND
A	0011	0011	0011
B	0101	0101	0101
RESULT	0111	0110	0001

Shifting:

HLF possesses a rotating shift operation of the form

VAL <ROT> nn

which causes any 32-bit quantity VAL to be circularly shifted by 'nn' bits. The rotation is to the left if 'nn' is positive, and to the right otherwise. The result is an integer, regardless of mode of VAL. The quantity 'nn' is an integer, and must be enclosed in parentheses if it is an arithmetic expression, even a unary negated value:

VAL <ROT> (8*j+12) or VAL <ROT> (-12)

In a rotating shift, bits shifted off one end of a 32-bit word reappear at the opposite end, and no bits are lost. A negative rotation is equivalent to a 32's-complement positive rotation, e.g.,

VAL <ROT> (-8) = VAL <ROT> 24

Another shift operator <SHIFT> allows an end-off shift in either direction, positive left and negative right. In this case, any 32-bit quantity may be logically shifted, wherein bits shifted off the end of a word are lost, and zeroes are introduced at the opposite end. The same restrictions on the shift count pertaining to <ROT> also apply to <SHIFT>. The sign bit is included in the shift along with the 31 magnitude bits. For example,

JSHIFT = 0AA000000 Z <SHIFT> (-8) yields 00AA0000.

SEXADECIMAL CONSTANTS:

A sexadecimal (hex) integer may be used directly within in-line source code in HLF, i.e., it is not necessary to define such a value within a DATA statement, provided the desired value is an INTEGER*4. A sexadecimal integer in source code is denoted by a symbol whose first digit is a DECIMAL digit 0-9, and which ENDS with the letter 'Z'. If the value begins with a sexadecimal digit A-F, then the programmer MUST prefix a lead non-significant zero, e.g., to code the hex constant ABC one must use '0ABCZ'.

As an example, to extract the three rightmost sexadecimal digits from the real quantity GUMBO, one could code

RIGHT3 = GUMBO <AND> 0FFFZ

Note that if fewer than eight digits are expressed, the resulting constant is presumed to be right justified and zero-filled. Given the possibility of a mandatory lead zero, the longest permissible hex constant would contain nine digits, e.g., 0ABCDEF12Z.

WARNING:

Failure to include the mandatory lead zero when it is required will produce an undiagnosed error, because the resulting symbol is often indistinguishable from a valid Fortran variable name, e.g., ABCZ, when 0ABCZ is intended.

IMPLEMENTATION OF HLF ON PERSONAL COMPUTERS:

The logical structure of a Stack Segment on a personal computer driven by a 8088 or 8086 micro-chip, though never used as a model in the design of HLF, is nonetheless almost identical to the logic used in managing the HLF Stack, as described. A means must be available whereby partial matrix results created at one program level remain undisturbed by routines at a lower level which also use stack storage. HLF, to date, has never been installed on a PC, and if it were to be installed immediately, the basic logic would be the same as that used in main frame systems which do not possess a Stack Segment. However, the architecture of a 8088-driven PC naturally lends itself to a version of HLF properly designed to take advantage of it.

CONCLUSION:

High-Level Fortran provides a source language upward compatible with existing Fortran languages which at the source level complements the advent of array-processing computers already available at the hardware level. Vector and matrix arithmetic is available to the programmer with the same formalism long familiar to Fortran programmers, thus minimizing learning time. Vector- and matrix- valued functions fully implement these two new variable types, and allow the user to code a vector or matrix expression far more succinctly and with far greater self-documentability than has heretofore been offered in conventional Fortrans. Finally, the stack is managed in a manner similar to the stack segment of a Personal Computer, which therefore readily lends itself to the implementation of HLF.

Acknowledgements:

The author is indebted to Dr. C. M. Hayden and Mr. L. R. Herman for helpful comments resulting from their reading of this paper; to Mr. H. M. Woolf and Mr. H. B. Howell for aid in digitally transcribing the manuscript to a suitable word processor; and to Ms. Laura Beckett for the tedious preparation of the manuscript. The matrix inversion function MINVER is used by permission of its author Mr. R. J. Purser.