Bell Laboratories

Computing Science Technical Report #37

Basic Utilities for Portable FORTRAN Libraries

P. A. Fox, A. D. Hall and N. L. Schryer

-8816-C (1-73)

A Dynamic Storage Allocator for Portable FORTRAN Libraries

by

A. D. Hall and N. L. Schryer
Bell Laboratories
Murray Hill, N. J.

ABSTRACT

This paper describes a package of simple portable FORTRAN subprograms for dynamic allocation and de-allocation of scratch space. Available space is managed on a last-in first-out basis in a manner similar to the stack discipline implicit in ALGOL 60 implementations. Use of this package often leads to more clearly structured programs, cleaner calling sequences, improved memory utilization, and better error detection.

A Dynamic Storage Allocator for Portable FORTRAN Libraries

by

A. D. Hall and N. L. Schryer
Bell Laboratories
Murray Hill, N. J.

1. Introduction

One of the biggest problems in FORTRAN programming is deciding how to provide scratch space for intermediate calculations in a subprogram. There are two commonly used techniques. The first attempts to keep calling sequences clean by providing scratch arrays local to each subprogram. The second attempts to avoid the wastefulness of local arrays by requiring the invoker of a subprogram to provide scratch space through an argument.

Neither of these solutions is entirely satisfactory. Since arrays local to a FORTRAN subprogram are fixed in size at the time of compilation, they are usually made big enough to accommodate the "largest" problem a subprogram may be called upon to solve. This not only places an upper limit on the size of a problem that can be handled but results in wasted storage when small problems are solved. When several such subprograms are used within a single program, the waste is compounded.

By requiring scratch space to be passed to a subprogram through one or more additional arguments, local arrays can be avoided, but the calling sequence becomes cumbersome and error prone, and the problem of storage allocation is simply left to the invoker. Worse yet, the subprogram has no way of verifying that enough scratch space has been provided.

This paper describes a package of simple portable FORTRAN subprograms for dynamic allocation of scratch space. Use of this package often leads to more clearly structured programs, cleaner calling sequences, improved memory utilization, and better error detection. Listings of these subprograms are included in the Appendix.

2. General Philosophy

The basic allocation mechanism provided is a stack similar to that which underlies ALGOL 60 [1] implementations. Unlike ALGOL 60, however, the allocation and de-allocation of space on the stack must be carried out through the use of explicit subprogram calls. Because of the nature of a stack, allocations and de-allocations must also be carried out on a last-in first-out basis. This approach not only keeps the programming simple, but it reduces the overhead to a minimum.

In order to make the stack invisible to most users of library programs, the package is self-initializing and contains a default stack size equivalent to 1000 "words" (FORTRAN INTEGER variables). If desired, larger amounts of stack space can be allocated for a particular run.

3. Allocation and De-allocation

The stack resides in the labeled COMMON region CSTAK. Any subroutine that uses space allocated in the stack must include the following declarations:

```
      COMMON /CSTAK/DSTAK(500)
C
      DOUBLE PRECISION DSTAK
```

These ensure that the length and type of the stack are properly and consistently declared in all subprograms; including those which use the allocator and are loaded from libraries. Failure to use these declarations could lead to unexpected difficulties during loading (or link-editing). If needed, a larger stack can be declared in the MAIN program (see Sec. 4).

To provide INTEGER, REAL and COMPLEX aliases for the stack the following declarations may be included:

```
      INTEGER ISTAK(1000)
      REAL RSTAK(1000)
      COMPLEX CMSTAK(500)
C
      EQUIVALENCE (DSTAK(1),ISTAK(1))
      EQUIVALENCE (DSTAK(1),RSTAK(1))
      EQUIVALENCE (DSTAK(1),CMSTAK(1))
```

If any of these is not wanted, its declaration and EQUIVALENCE to DSTAK may be left out.

We now present two basic subprograms, IALLOC and DALLOC, for allocating and de-allocating stack space, respectively. The allocation subprogram is

```
INTEGER FUNCTION IALLOC(NITEMS,ISIZE)
```

where NITEMS is the number of items of size ISIZE words to be allocated. The most commonly used values of ISIZE are as follows:

Item Type	ISIZE
INTEGER	1
REAL	1
DOUBLE PRECISION	2
COMPLEX	2

The statement

```
I = IALLOC(N,1)
```

returns an index I so that the locations

```
ISTAK(I) , ... , ISTAK(I+N-1)
```

form the storage allocated for N INTEGER items. Alternatively, the locations

```
RSTAK(I) , ... , RSTAK(I+N-1)
```

can be thought of as the space allocated for N REAL items. Note that the space allocated is not initialized to any particular value.

The statement

```
I = IALLOC(N,2)
```

returns an index I so that the locations

```
DSTAK(I) , ... , DSTAK(I+N-1)
```

form the space allocated for N DOUBLE PRECISION items. Similarly, the locations

```
CMSTAK(I) , ... , CMSTAK(I+N-1)
```

can be thought of as the space allocated for N COMPLEX items.

It is important to note that ISIZE is not a switch that determines what type of allocation is desired. Rather, it

determines the alignment and number of words for each item. Thus, if a FORTRAN compiler supports DOUBLE PRECISION COMPLEX declarations, allocations for this data type can be accomplished in the following way:

```
      COMMON /CSTAK/DSTAK(500)
C
      DOUBLE PRECISION DSTAK
      DOUBLE PRECISION COMPLEX DCSTAK(250)
C
      EQUIVALENCE (DSTAK(1),DCSTAK(1))
        .
        .
      I = IALLOC(N,4)
        .
        .
```

and the allocated space would be the locations

```
DCSTAK(I) , ... , DCSTAK(I+N-1).
```

The de-allocation subprogram is

```
SUBROUTINE DALLOC(N)
```

which simply de-allocates the last N allocations.

As a simple example of the use of these two subprograms, consider a "little black box" subroutine LBB(A,N) which is supposed to return something in a REAL vector A of length N and requires two REAL scratch arrays of length N to do it. LBB would look roughly as follows:

```
      SUBROUTINE LBB(A,N)
C
      COMMON /CSTAK/DSTAK(500)
C
      DOUBLE PRECISION DSTAK
      REAL A(N)
      REAL RSTAK(1000)
C
      EQUIVALENCE (DSTAK(1),RSTAK(1))
C
      IB = IALLOC(N,1)
      IC = IALLOC(N,1)
         .
         .
      { code referring to RSTAK(IB+n) and RSTAK(IC+m) }
         .
         .
      CALL DALLOC(2)
C
      RETURN
      END
```

To avoid messy (and possibly non-standard) subscript calculations, it is sometimes more convenient to pass the arguments and the allocated scratch space down one more level to a subprogram which does the real work. This not only will make programs more readable and easier to code, but it will, in many cases, make them more efficient. LBB would be coded as an "executive" routine calling on a "work-horse" routine, as follows:

```
      SUBROUTINE LBB(A,N)
C
      COMMON /CSTAK/DSTAK(500)
C
      DOUBLE PRECISION DSTAK
      REAL A(N)
      REAL RSTAK(1000)
C
      EQUIVALENCE (DSTAK(1),RSTAK(1))
C
      IB = IALLOC(N,1)
      IC = IALLOC(N,1)
C
      CALL LBB1(A,RSTAK(IB),RSTAK(IC),N)
C
      CALL DALLOC(2)
      RETURN
      END
```

4. Initialization

As previously mentioned, the subprograms in the allocation package are all self-initializing so that a user with small requirements need not even know of their existence. However, there will be applications which require a larger stack than that provided by default. In this case, declarations for the stack and an explicit call to an initialization subprogram must be made in the MAIN program. The initialization subprogram is

```
      SUBROUTINE STINIT(NITEMS,ISIZE)
```

where NITEMS is the number of items of size ISIZE words set aside for the stack.

For example, to set up a stack with 1000 DOUBLE PRECISION items or, equivalently, 2000 INTEGER items, the following declarations and subroutine call would be used.

```
      COMMON /CSTAK/DSTAK(1000)
C
      DOUBLE PRECISION DSTAK
         .
         .
      CALL STINIT(1000,2)
```

If desired, the CALL STINIT above could have been replaced by:

```
      CALL STINIT(2000,1)
```

It should be noted that the first four words of the stack are reserved for use by the allocator and that each allocation has an associated space overhead of at least 2, but no more than ISIZE+1 words. When estimating the length of the stack required, these overheads should be taken into account. To determine the exact stack length required, one can use the SRECAP subprogram (see Sec. 5).

5. Miscellaneous Subprograms

By design, it is considered a fatal error to attempt to allocate more space than is actually available. The error could have been made recoverable (in the sense of [2]) but it was felt that this would unnecessarily complicate both implementation and use. For those situations when it is desirable to know how much stack remains so that it may all be allocated, the subprogram

```
      INTEGER FUNCTION NIRALL(ISIZE)
```

can be used. NIRALL returns the number of items of size ISIZE remaining to be allocated in a single invocation of IALLOC. (Recall from Section 4 that there are 2 or more words of space overhead associated with each allocation. If the stack is effectively full, NIRALL will return 0). The statements

```
      NLEFT = NIRALL(1)
      I = IALLOC(NLEFT,1)
```

allocate all remaining space as a single block of INTEGER or REAL items.

In some applications it may be necessary to change the size of the most recent allocation. This can be accomplished with the subprogram

```
      INTEGER FUNCTION MTSTAK(NITEMS)
```

which will reset the length of the last allocation to NITEMS items and, in a manner similar to IALLOC, return the index of the first item of that allocation. If the last allocation is truncated, only the first NITEMS are preserved. If the last allocation is extended, existing information is preserved but the added space is not initialized.

As an example of the use of NIRALL and MTSTAK, the following program fragment reads an indeterminate number of positive REALs into the stack. For convenience, we assume that a negative data item marks the end of the data.

```
         .
         .
C
C  FIND OUT HOW MUCH STACK SPACE IS LEFT
C  AND ALLOCATE IT ALL.
C
      NLEFT = NIRALL(1)
      IF (NLEFT .EQ. 0) GO TO error
      I = IALLOC(NLEFT,1)
C
C  INITIALIZE COUNT OF ITEMS READ SO FAR.
C
      NITEMS = 0
C
C  READ AN ITEM INTO THE STACK AND TEST FOR END-OF-DATA.
C
 10   IF (NITEMS .EQ. NLEFT) GO TO error
      READ (6,100) RSTAK(I)
 100  FORMAT ( F10.6 )
C
      IF (RSTAK(I) .LT. 0) GO TO 20
C
      NITEMS = NITEMS + 1
      I = I + 1
      GO TO 10
C
C  HERE WHEN ALL DATA READ. TRUNCATE THE ALLOCATION
C
 20   IF (NITEMS .EQ. 0) GO TO error
      I = MTSTAK(NITEMS)
C
C  NOW THE ITEMS ARE IN LOCATIONS
C  RSTAK(I) , ... , RSTAK(I+NITEMS-1)
C
         .
         .
```

The subprogram

```
SUBROUTINE SRECAP(IUNIT)
```

will write on logical unit IUNIT a summary of the status of the stack, namely the number of outstanding allocations, the current active length, the maximum length used and the maximum length allowed. Typically, this subroutine would be called at the end of a run to obtain five lines of output like the following

```
STACK STATISTICS...

OUTSTANDING ALLOCATIONS        0
CURRENT ACTIVE LENGTH          4
MAXIMUM LENGTH USED         1825
MAXIMUM LENGTH ALLOWED      9000
```

which says that the same or a similar run could be made with only 1825 INTEGER words in the stack and that since the number of outstanding allocations is 0, all allocations have been successfully de-allocated. This information provides a check on the balancing of allocations and de-allocations.

6. Portability Considerations

In the implementation of the allocator, every attempt has been made to ensure portability [3,4]. Nevertheless, it has been necessary to make two assumptions about FORTRAN that are valid for most production systems.

First it is assumed that there is no subscript range checking. Second it is assumed that variables local to a subprogram which are initialized by DATA statements retain their values from one subprogram invocation to the next.

Also, in order to adhere to a strict interpretation of the FORTRAN Standard [5, Sec 10.2.5], it is necessary in the MAIN program to declare the COMMON region CSTAK and to call the subroutine STINIT. These precautions will ensure that data stored in the stack will not be lost when using overlays (segments) or when running under FORTRAN systems in which COMMON is dynamically allocated.

7. Implementation Notes

Each allocation consists of three parts: padding, allocated space, and control information. The padding takes from 0 to ISIZE-1 words and is present to provide the proper alignment for the allocated space which occupies NITEMS*ISIZE words. The control information takes two words the first of which contains ISIZE. The second word contains the index (in ISTAK) of the second word of the control information associated with the previous allocation. If there is no previous allocation, this contains 4.

The first four locations in ISTAK contain the following data:

ISTAK(1) = the number of outstanding allocations

ISTAK(2) = the current active length of the stack (second control word of last allocation)

ISTAK(3) = the maximum value of ISTAK(2) achieved so far during the run

ISTAK(4) = the maximum possible length of the stack.

All lengths are in words (FORTRAN INTEGER variables) and the default value of ISTAK(4) is 1000.

The consistency of these data and the control information associated with the last allocation is checked on every call to allocator. If an inconsistency is found, the Error Handler [2] is called to deliver an appropriate message and terminate the run.

References

1. "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM, vol. 6 (1963), p. 1.

2. A. D. Hall and N. L. Schryer, A Centralized Error Handling Facility for Portable FORTRAN Libraries, this report.

3. B. G. Ryder, The PFORT Verifier, Software Practice and Experience, Vol. 4, No. 4, (October-December 1974), pp. 359-377.

4. A. D. Hall and B. G. Ryder, The PFORT Verifier, Computing Science Technical Report #12, Bell Laboratories, Murray Hill, N. J., 07974 (March 1975).

5. USA Standard FORTRAN, USA Standards Institute, New York, N. Y., 1966.

Appendix

```
      INTEGER FUNCTION IALLOC(NITEMS,ISIZE)
C
C  ALLOCATES AN ARRAY OF LENGTH NITEMS*ISIZE OUT
C  OF THE INTEGER ARRAY ISTAK. ON RETURN, THE ARRAY WILL OCCUPY ...
C
C       ISTAK((1+(IALLOC-1)*ISIZE),...,ISTAK((IALLOC-1+NITEMS)*ISIZE))
C
C  ERROR STATES -
C
C    1 - ONE OF (LNOW,LUSED,LMAX) HAS BEEN OVERWRITTEN.
C    2 - NITEMS.LE.0.
C    3 - ISIZE.LE.0.
C    4 - STACK OVERFLOW.
C
C  THE ALLOCATOR RESERVES THE FIRST FOUR INTEGER WORDS OF THE STACK
C  FOR ITS OWN INTERNAL BOOK-KEEPING. THE USE OF THESE FOUR WORDS IS
C  DESCRIBED BELOW
C
C    ISTAK(1) - THE NUMBER OF CURRENT OUTSTANDING ALLOCATIONS.
C    ISTAK(2) - THE CURRENT ACTIVE LENGTH OF THE STACK IN INTEGER WORDS.
C    ISTAK(3) - THE MAXIMUM VALUE OF ISTAK(2) ACHIEVED SO FAR DURING
C               THE RUN.
C    ISTAK(4) - THE MAXIMUM LENGTH THE STACK CAN HAVE, IN INTEGER WORDS.
C
       COMMON /CSTAK/DSTAK
C
       DOUBLE PRECISION DSTAK(500)
       REAL RSTAK(1000)
       INTEGER ISTAK(1000)
       LOGICAL INIT
C
       EQUIVALENCE (DSTAK(1),RSTAK(1),ISTAK(1))
       EQUIVALENCE (ISTAK(1),LOUT)
       EQUIVALENCE (ISTAK(2),LNOW)
       EQUIVALENCE (ISTAK(3),LUSED)
       EQUIVALENCE (ISTAK(4),LMAX)
C
       DATA INIT/.TRUE./
C
       IF (INIT) CALL S0TAK0(INIT,1000)
C
       IF (LNOW.LT.4.OR.LNOW.GT.LUSED.OR.LUSED.GT.LMAX) CALL SETERR
      1   (54HIALLOC - ONE OF (LNOW,LUSED,LMAX) HAS BEEN OVERWRITTEN,
      2    54,1,2)
C
       IF (NITEMS.LE.0) CALL SETERR(20HIALLOC - NITEMS.LE.0,20,2,2)
       IF (ISIZE.LE.0)  CALL SETERR(19HIALLOC - ISIZE.LE.0, 19,3,2)
C
       IALLOC = (LNOW-1)/ISIZE+2
       I = (IALLOC-1+NITEMS)*ISIZE+2
C
```

```
C  STACK OVERFLOW IS AN UNRECOVERABLE ERROR.
C
      IF (I.GT.LMAX) CALL SETERR(23HIALLOC - STACK OVERFLOW,23,4,2)
C
C  ISTAK(I-1) CONTAINS THE ITEM SIZE FOR THE FRAME.
C  ISTAK(I  ) CONTAINS A POINTER TO THE END OF THE PREVIOUS
C             ALLOCATION.
C
      ISTAK(I-1) = ISIZE
      ISTAK(I  ) = LNOW
      LOUT = LOUT+1
      LNOW = I
      LUSED = MAX0(LUSED,LNOW)
C
      RETURN
C
      END



      SUBROUTINE DALLOC(NUMBER)
C
C  DE-ALLOCATES THE LAST (NUMBER) ALLOCATIONS MADE IN THE STACK
C  BY IALLOC.
C
C  ERROR STATES -
C
C    1 - NUMBER.LT.0.
C    2 - ONE OF (LNOW,LUSED,LMAX) HAS BEEN OVERWRITTEN.
C    3 - CANNOT DE-ALLOCATE MORE THAN THE ENTIRE STACK.
C    4 - THE POINTER AT ISTAK(LNOW) HAS BEEN OVERWRITTEN.
C
      COMMON /CSTAK/DSTAK
C
      DOUBLE PRECISION DSTAK(500)
      REAL RSTAK(1000)
      INTEGER ISTAK(1000)
      LOGICAL INIT
C
      EQUIVALENCE (DSTAK(1),RSTAK(1),ISTAK(1))
      EQUIVALENCE (ISTAK(1),LOUT)
      EQUIVALENCE (ISTAK(2),LNOW)
      EQUIVALENCE (ISTAK(3),LUSED)
      EQUIVALENCE (ISTAK(4),LMAX)
C
      DATA INIT/.TRUE./
C
      IF (INIT) CALL SOTAK0(INIT,1000)
C
      IF (NUMBER.LT.0) CALL SETERR(20HDALLOC - NUMBER.LT.0,20,1,2)
      IF (LNOW.LT.4.OR.LNOW.GT.LUSED.OR.LUSED.GT.LMAX) CALL SETERR
     1   (54HDALLOC - ONE OF (LNOW,LUSED,LMAX) HAS BEEN OVERWRITTEN,
     2    54,2,2)
```

```
C
      IN = NUMBER
 10      IF (IN.EQ.0) RETURN
C
         IF (LNOW.EQ.4) CALL SETERR
     1    (54HDALLOC - CANNOT DE-ALLOCATE MORE THAN THE ENTIRE STACK,
     2     54,3,2)
C
C     CHECK TO MAKE SURE THE BACK POINTERS ARE MONOTONE.
C
         IF (ISTAK(LNOW).LT.4.OR.ISTAK(LNOW).GE.LNOW-2) CALL SETERR
     1    (56HDALLOC - THE POINTER AT ISTAK(LNOW) HAS BEEN OVERWRITTEN,
     2     56,4,2)
C
         LOUT = LOUT-1
         LNOW = ISTAK(LNOW)
         IN = IN-1
         GO TO 10
C
      END



      SUBROUTINE STINIT(NITEMS,ISIZE)
C
C  INITIALIZES THE STACK ALLOCATOR, SETTING THE LENGTH OF THE STACK.
C
C  ERROR STATES -
C
C    1 - NITEMS.LE.0.
C    2 - ISIZE.LE.0.
C
      IF (NITEMS.LE.0) CALL SETERR(20HSTINIT - NITEMS.LE.0,20,1,2)
      IF (ISIZE.LE.0)  CALL SETERR(19HSTINIT - ISIZE.LE.0, 19,2,2)
C
      CALL S0TAK0(.FALSE.,NITEMS*ISIZE)
C
      RETURN
C
      END
```

```
      INTEGER FUNCTION NIRALL(ISIZE)
C
C  RETURNS THE NUMBER OF ITEMS OF SIZE ISIZE THAT REMAIN
C  TO BE ALLOCATED IN ONE REQUEST.
C
C  ERROR STATES -
C
C    1 - ONE OF (LNOW,LUSED,LMAX) HAS BEEN OVERWRITTEN.
C    2 - ISIZE.LT.1.
C
      COMMON /CSTAK/DSTAK
C
      DOUBLE PRECISION DSTAK(500)
      REAL RSTAK(1000)
      INTEGER ISTAK(1000)
      LOGICAL INIT
C
      EQUIVALENCE (DSTAK(1),RSTAK(1),ISTAK(1))
      EQUIVALENCE (ISTAK(1),LOUT)
      EQUIVALENCE (ISTAK(2),LNOW)
      EQUIVALENCE (ISTAK(3),LUSED)
      EQUIVALENCE (ISTAK(4),LMAX)
C
      DATA INIT/.TRUE./
C
      IF (INIT) CALL S0TAK0(INIT,1000)
C
      IF (LNOW.LT.4.OR.LNOW.GT.LUSED.OR.LUSED.GT.LMAX) CALL SETERR
     1   (54HNIRALL - ONE OF (LNOW,LUSED,LMAX) HAS BEEN OVERWRITTEN,
     2    54,1,2)
C
      IF (ISIZE.LT.1) CALL SETERR
     1   (19HNIRALL - ISIZE.LT.1,19,2,2)
C
      NIRALL = MAX0( (LMAX-2)/ISIZE-(LNOW-1)/ISIZE-1, 0 )
C
      RETURN
C
      END



      INTEGER FUNCTION MTSTAK(NITEMS)
C
C  CHANGES THE LENGTH OF THE FRAME AT THE TOP OF THE STACK
C  TO NITEMS.
C
C  ERROR STATES -
C
C    1 - LNOW HAS BEEN OVERWRITTEN.
C    2 - ISTAK(LNOWO-1) HAS BEEN OVERWRITTEN.
C
      COMMON /CSTAK/DSTAK
```

```
C
      DOUBLE PRECISION DSTAK(500)
      REAL RSTAK(1000)
      INTEGER ISTAK(1000)
C
      EQUIVALENCE (DSTAK(1),RSTAK(1),ISTAK(1))
      EQUIVALENCE (ISTAK(1),LOUT)
      EQUIVALENCE (ISTAK(2),LNOW)
      EQUIVALENCE (ISTAK(3),LUSED)
      EQUIVALENCE (ISTAK(4),LMAX)
C
      LNOWO = LNOW
      CALL DALLOC(1)
C
      IF (LNOWO.LT.7) CALL SETERR
     1   (34HMTSTAK - LNOW HAS BEEN OVERWRITTEN,34,1,2)
C
      ISIZE = ISTAK(LNOWO-1)
C
      IF (ISIZE.LT.1) CALL SETERR
     1   (44HMTSTAK - ISTAK(LNOWO-1) HAS BEEN OVERWRITTEN,44,2,2)
C
      MTSTAK = IALLOC(NITEMS,ISIZE)
C
      RETURN
C
      END



      SUBROUTINE SRECAP(IWUNIT)
C
C  WRITES LOUT, LNOW, LUSED AND LMAX ON LOGICAL UNIT IWUNIT.
C
      COMMON /CSTAK/DSTAK
C
      DOUBLE PRECISION DSTAK(500)
      REAL RSTAK(1000)
      INTEGER ISTAK(1000)
      INTEGER ISTATS(4)
      LOGICAL INIT
C
      EQUIVALENCE (DSTAK(1),RSTAK(1),ISTAK(1))
      EQUIVALENCE (ISTAK(1),ISTATS(1))
      EQUIVALENCE (ISTAK(1),LOUT)
      EQUIVALENCE (ISTAK(2),LNOW)
      EQUIVALENCE (ISTAK(3),LUSED)
      EQUIVALENCE (ISTAK(4),LMAX)
C
      DATA INIT/.TRUE./
C
      IF (INIT) CALL S0TAK0(INIT,1000)
C
```

```
      WRITE(IWUNIT,9000) ISTATS
C
 9000 FORMAT(20H0STACK STATISTICS...//
     1       24H OUTSTANDING ALLOCATIONS,I8/
     1       24H CURRENT ACTIVE LENGTH  ,I8/
     3       24H MAXIMUM LENGTH USED    ,I8/
     4       24H MAXIMUM LENGTH ALLOWED ,I8)
C
      RETURN
C
      END



      SUBROUTINE S0TAK0(LARG,LENGTH)
C
C  INITIALIZES THE STACK TO LENGTH INTEGER WORDS
C
      COMMON /CSTAK/DSTAK
C
      DOUBLE PRECISION DSTAK(500)
      REAL RSTAK(1000)
      INTEGER ISTAK(1000)
      LOGICAL LARG,INIT
C
      EQUIVALENCE (DSTAK(1),RSTAK(1),ISTAK(1))
      EQUIVALENCE (ISTAK(1),LOUT)
      EQUIVALENCE (ISTAK(2),LNOW)
      EQUIVALENCE (ISTAK(3),LUSED)
      EQUIVALENCE (ISTAK(4),LMAX)
C
      DATA INIT/.FALSE./
C
      IF (.NOT.LARG) GO TO 10
C
C  HERE IF NOT FROM STINIT
C
      LARG = .FALSE.
      IF (INIT) RETURN
C
C  HERE TO INITIALIZE
C
 10   INIT = .TRUE.
      LOUT = 0
      LNOW = 4
      LUSED = 4
      LMAX = MAX0(LENGTH,6)
C
      RETURN
C
      END
```

A Centralized Error Handling Facility for Portable FORTRAN Libraries

by

A. D. Hall and N. L. Schryer
Bell Laboratories
Murray Hill, New Jersey

ABSTRACT

Although it is obvious and widely recognized that library procedures intended for general use ought to check for incorrect arguments and other error conditions, most FORTRAN subprograms avoid such testing because the required code is annoyingly awkward and bulky. This paper presents a portable centralized error handling facility which permits a subprogram to detect an error, print an appropriate message, and terminate execution all in a single statement.

The facility also allows a subprogram to indicate an error condition and then return control to its caller if the caller so desires. The caller may either retrieve the error number and choose to print the saved message and terminate execution, or leave the error state and proceed by an alternative route, or return control to its caller if the latter is willing.

The use of the facility leads to clean, simple code for the detection and handling of error conditions, and provides a uniform mechanism for passing error information from one subprogram to another. The need for additional arguments in calling sequences is avoided.

A Centralized Error Handling Facility for Portable FORTRAN Libraries

by
A. D. Hall and N. L. Schryer
Bell Laboratories
Murray Hill, New Jersey

1. Introduction

Although it is obvious and widely recognized that library procedures intended for general use ought to check for incorrect arguments and other error conditions, most programs avoid such testing because the required code is annoyingly awkward or bulky. For example, to ensure that the dimension, N, of an array is positive and issue an appropriate diagnostic if not, one would have to write

```
      ...
      IF (N .GE. 1) GO TO 10
      WRITE(6,9000)
 9000 FORMAT(19H N IS LESS THAN ONE)
      CALL FDUMP
      STOP
C
 10   CONTINUE
           ...
```

where FDUMP is a locally provided subroutine intended to produce a dump (hopefully symbolic, see Section 6).

Another drawback to the above, non-centralized, error handling method is that it terminates execution unconditionally even though the calling subprogram might want to recover control and either correct the error or proceed by an alternative route.

This paper describes an Error Handler designed around the idea that most errors detectable in library subprograms are fatal while only a few are recoverable. We consider an error fatal if it prevents a program or subprogram from obtaining computationally useful results. For example, a non-positive dimension for an adjustably dimensioned array would be considered a fatal error. In this case, a single subroutine call will suffice to print an error message and terminate the run with a dump.

We consider an error recoverable only if a calling subprogram might use that fact to obtain computationally useful results. For example, a singular matrix in a triangularization subprogram might be considered a

recoverable error. In this case, the called subprogram sets an error state with a single subroutine call and provides code for return of control to the caller. If the caller has set the recovery mode on, control will be returned and the caller may test for the presence of an error and take appropriate action. If the caller has set the recovery mode off, the error will be treated as if it were fatal and the caller need not provide any code whatsoever for handling the error.

Listings of the subprograms comprising the Error Handler are given in the Appendix.

2. Error Setting

This section describes the use of the error-setting subroutine, SETERR.

The following statement:

```
CALL SETERR(MESSG, NMESSG, NERR, IOPT)
```

sets an error state with the Hollerith message MESSG of length NMESSG characters ($1 \leq$ NMESSG ≤ 72), and error number NERR. The parameter IOPT determines what action is to be taken by SETERR, as follows:

If IOPT = 1, the error is recoverable; but if not in recovery mode, SETERR simply prints the message and stops. If in recovery mode, SETERR records the error number and message, sets the error state, and returns.

If IOPT = 2, the error is fatal; SETERR prints the message, produces a dump and stops.

Messages are written on the standard output unit used for error messges [1].

The following simple dot product function illustrates the use of SETERR for a fatal error.

```
      REAL FUNCTION DOT(A, B, N)
      REAL A(N), B(N)
C
      IF (N .LT. 1) CALL SETERR(12HDOT - N.LT.1, 12, 1, 2)
C
      DOT = 0.0
      DO 10 I = 1, N
 10     DOT = DOT + A(I)*B(I)
C
      RETURN
      END
```

The above call to SETERR indicates that "N.LT.1" is the first error condition (NERR = 1) in "DOT" and that it is a fatal (IOPT = 2) error. In case of such an error, the following line would be output

```
ERROR     1 IN DOT - N.LT.1
```

followed by a dump and termination of the run. We adopt the convention that all error messages begin with the name of the package or subprogram which produced the error.

An example of the use of SETERR for recoverable errors is given at the end of Section 4.

3. Error Recovery

Recovery from an error is permitted only when the value of IOPT is 1 in the call to SETERR. In this case, a recovery switch internal to the error handling routines is tested to see if recovery is permitted. If the value of the recovery switch is 1, recovery is permitted, and SETERR records the error number and message, sets the error state, and returns control to the caller. If the value of the recovery switch is 2, the error message is printed and the run terminated. Execution begins with the recovery switch set to 2, so that all errors are considered fatal.

Two subroutines, ENTSRC and RETSRC, are provided for manipulating the recovery switch. ENTSRC is used to retrieve the old value of the recovery switch and simultaneously set a new value. RETSRC is used to restore the recovery switch to its previous value. ENTSRC, RETSRC and SETERR are so designed that a program can never be in the error state unless the value of the recovery switch is 1.

When IRNEW is 1 or 2, the statement

```
CALL ENTSRC(IROLD, IRNEW)
```

sets the recovery switch to IRNEW and returns the previous value in IROLD. If IRNEW is 0, the recovery switch is left unchanged and the current value returned in IROLD. It is considered a fatal error to call ENTSRC with a value of IRNEW other than 0, 1, or 2.

The statement

```
CALL RETSRC(IROLD)
```

restores the recovery switch to IROLD. It is considered a fatal error if IROLD has a value other than 1 or 2.

Typically, calls of ENTSRC and RETSRC are used to surround sections of program where it is desired to force a particular setting of the recovery switch and then restore it to its previous value. For example, to turn recovery mode on upon entry to a subprogram and restore it to its previous value upon return, the following coding is used:

```
      SUBROUTINE WORKER
C
      CALL ENTSRC(IROLD,1)
      ...
      ...
      CALL RETSRC(IROLD)
      RETURN
C
      END
```

Recovery from recoverable errors is a two-sided process, with the called subprogram setting an error state (via SETERR) and the caller then testing for, and responding to, that error state. If a subprogram explicitly puts itself in recovery mode (via ENTSRC), it should test for the occurrence of an error after any call to a subprogram that might set a recoverable error state. If an error has occurred, the subprogram may choose to either (1) force printing of the saved message and terminate execution, (2) leave the error state and proceed by an alternative route, or (3) return control to its caller if the latter is willing.

Two errors in a row, with no explicit recovery statement between, is regarded as a fatal error since it means that the user has failed to recover properly from an error condition. Similarly, it is a fatal error to call ENTSRC while the program is in the error state, or to call RETSRC with IROLD = 2 while the program is in the error state. In each of these three cases, two error messages are printed and execution stops.

Every call of a subprogram that has recoverable error conditions and could be executed in recovery mode should be followed by a test for an occurrence of an error. The error number may be retrieved by writing

```
NERR2 = NERROR(NERR1)
```

which sets both NERR2 and NERR1 to the current value of the error number. If the error number is non-zero, it means that an error has occurred and that corrective action must be taken. When done, the error state may be left by writing

```
CALL ERROFF
```

A typical use of ENTSRC, RETSRC and NERROR in a subprogram takes the following form:

```
      SUBROUTINE WORKER
      ...
C
C     SET RECOVERY MODE ON.
C
      CALL ENTSRC(IROLD, 1)
      ...
      CALL SUBPRG
      IF (NERROR(NERR) .EQ. 0) GO TO 10
C
C     ERROR IN SUBPRG. TAKE ONE OF THE THREE
C     CORRECTIVE ACTIONS PRESENTED BELOW:
C
C     (1) TERMINATE EXECUTION
C     (2) LEAVE THE ERROR STATE AND PROCEED BY
C         ALTERNATIVE ROUTE
C     (3) RETURN AN ERRCR TO CALLER
C
      ...
C
C     NO ERROR. PROCEED.
C
 10   ...
C
C     RESTORE PREVIOUS RECOVERY MODE AND RETURN
C
 20   CALL RETSRC(IROLD)
      RETURN
C
      END
```

Possible ways of coding the three alternatives outlined in the above program are given below.

(1) Terminating execution

```
C
C     ERROR IN SUBPRG. TERMINATE EXECUTION
C     BY FORCING A DOUBLE ERROR.
C
      CALL SETERR(22HWORKER - SUBPRG FAILED, 22, 1, 2)
```

(2) Proceeding by an alternative route

```
C
C     ERROR IN SUBPRG. TRY SOMETHING ELSE.
C
      CALL ERROFF
      ...
      GO TO 20
```

(3) Returning an error to the caller

```
C
C     ERROR IN SUBPRG. RETURN NEW ERROR TO CALLER.
C
      CALL ERROFF
      CALL SETERR(22HWORKER - SUBPRG FAILED, 22, 1, 1)
      GO TO 20
```

To avoid having to deal with recoverable errors at all, the subroutine WORKER illustrated above could be coded as follows:

```
      SUBROUTINE WORKER
      ...
C
C     SET RECOVERY MODE OFF SC THAT ERRORS IN
C     ANY SUBPROGRAMS ARE IMMEDIATELY FATAL.
C
      CALL ENTSRC(IROLD, 2)
      ...
      CALL SUBPRG
      ...
C
C     RESTORE PREVIOUS RECOVERY MODE AND RETURN.
C
      CALL RETSRC(IROLD)
      RETURN
C
      END
```

4. Returning in the Error State

For NERROR to work reliably, it is important that the error number, if non-zero, be due to an error in the subprogram invoked in the preceeding statement. This can only be guaranteed in the current framework if we are certain tnat the program is not in the error state when that subprogram is invoked. Otherwise, the subprogram could succeed but NERROR would report an error. To ensure that this does not happen, we adopt the convention that any subprogram which can return one or more recoverable errors

and might be invoked while in the error state must begin with a call to ENTSRC, even if only to check this condition. A corresponding call to RETSRC is necessary prior to returning only if the recovery switch is altered by the call to ENTSRC (IRNEW ≠ 0).

A simple subprogram which returns recoverable errors might be coded roughly as follows:

```
      SUBROUTINE WORKER
      ...
C
C     CHECK FOR EXISTING ERROR STATE
C
      CALL ENTSRC(IDUMMY, 0)
      ...
C
C     IF ZERO PIVOT, SET ERROR AND RETURN.
C
      IF (X(I,J) .NE. 0.) GO TO 10
        CALL SETERR(19HWORKER - ZERO PIVOT, 19, 1, 1)
        RETURN
C
 10   CONTINUE
      ...
```

5. Debugging and Testing

When testing subprograms which can return a recoverable error, it is often desirable to print the error number and associated message, and the subroutine EPRINT is provided for this purpose. For example, if the statements

```
...
CALL LBB
CALL EPRINT
IF(NERROR(IERR) .EQ. 0) GO TO 10
...
```

are executed in recovery mode, and if a recoverable error occurs in LBB, then EPRINT will print the corresponding message.

6. Implementation Notes

In the implementation of the Error Handler, every attempt has been made to ensure portability [2,3]. However, it has been necessary to make an assumption about FORTRAN systems which is almost universally valid, but not guaranteed by the FORTRAN Standard [4]. Specifically, it

has been assumed that local variables initialized by DATA statements retain their values from one subprogram invocation to the next. (This assumption is likely to be violated in the presence of overlays.)

SETERR also calls a subroutine FDUMP to produce a dump. Ideally, this dump ought to be symbolic in the sense of [5,6,7,8], and not simply a sheaf of octal or hexadecimal constants. More specifically, for each active subprogram a symbolic dump should list the names and values of all its variables, the name of its caller, and the location in the caller from which it was called. Since it is impossible to write a portable FORTRAN subprogram to print symbolic dumps in a an arbitrary FORTRAN environment, such a subprogram has not been included. Instead, a dummy version of FDUMP has been supplied which may be replaced by a locally supplied subroutine.

References

1. P. A. Fox, A. D. Hall and N. L. Schryer, Machine Constants for Portable FORTRAN Libraries, this report.

2. B. G. Ryder, The PFORT Verifier, Software Practice and Experience, Vol. 4, No. 4, (October-December 1974), pp. 359-377.

3. A. D. Hall and B. G. Ryder, The PFORT Verifier, Computing Science Technical Report #12, Bell Laboratories, Murray Hill, N. J., 07974 (March 1975).

4. American National Standard FORTRAN, American National Standards Institute, New York, N. Y., 1966.

5. W. S. Brown, An Operating Environment for Dynamic Recursive Computer Programming Systems, Comm. ACM, Vol. 8, (1965), pp. 371-377.

6. R. Bayer, D. Gries, M. Paul and H. R. Wiehle, The ALCOR Illinois 7090/7094 Post Mortem Dump, Comm. ACM, Vol. 10, (1967), pp. 804-808.

7. A. D. Hall, FORTREX: GECOS III Extended FORTRAN, Bell Laboratories Computer Center, unpublished (November 1968).

8. A. D. Hall, FDS: A FORTRAN Debugging System - Overview and Installer's Guide, Computing Science Technical Report #29, Bell Laboratories, Murray Hill, N. J., 07974 (May 1975).

```
      SUBROUTINE SETERR(MESSG,NMESSG,NERR,IOPT)
C
C  SETERR SETS LERROR = NERR, OPTIONALLY PRINTS THE MESSAGE AND
C  DUMPS ACCORDING TO THE FOLLOWING RULES...
C
C    IF IOPT = 1 AND RECOVERING      - JUST REMEMBER THE ERROR.
C    IF IOPT = 1 AND NOT RECOVERING  - PRINT AND STOP.
C    IF IOPT = 2                     - PRINT, DUMP AND STOP.
C
C  INPUT
C
C    MESSG  - THE ERROR MESSAGE.
C    NMESSG - THE LENGTH OF THE MESSAGE, IN CHARACTERS.
C    NERR   - THE ERROR NUMBER. MUST HAVE NERR NON-ZERO.
C    IOPT   - THE OPTION. MUST HAVE IOPT=1 OR 2.
C
C  ERROR STATES -
C
C    1 - MESSAGE LENGTH NOT POSITIVE.
C    2 - CANNOT HAVE NERR=0.
C    3 - AN UNRECOVERED ERROR FOLLOWED BY ANOTHER ERROR.
C    4 - BAD VALUE FOR IOPT.
C
C  ONLY THE FIRST 72 CHARACTERS OF THE MESSAGE ARE PRINTED.
C
C  THE ERROR HANDLER CALLS A SUBROUTINE NAMED FDUMP TO PRODUCE A
C  SYMBOLIC DUMP. TO COMPLETE THE PACKAGE, A DUMMY VERSION OF FDUMP
C  IS SUPPLIED, BUT IT SHOULD BE REPLACED BY A LOCALLY WRITTEN
C  VERSION WHICH AT LEAST GIVES A TRACE-BACK.
C
      INTEGER MESSG(1)
C
C  THE UNIT FOR ERROR MESSAGES.
C
      IWUNIT=I1MACH(4)
C
      IF (NMESSG.GE.1) GO TO 10
C
C  A MESSAGE OF NON-POSITIVE LENGTH IS FATAL.
C
        WRITE(IWUNIT,9000)
9000    FORMAT(52H1ERROR    1 IN SETERR - MESSAGE LENGTH NOT POSITIVE.)
        GO TO 60
C
C  NW IS THE NUMBER OF WORDS THE MESSAGE OCCUPIES.
C
 10   NW=(MIN0(NMESSG,72)-1)/I1MACH(6)+1
C
      IF (NERR.NE.0) GO TO 20
```

```
C
C  CANNOT TURN THE ERROR STATE OFF USING SETERR.
C
      WRITE(IWUNIT,9001)
 9001 FORMAT(42H1ERROR    2 IN SETERR - CANNOT HAVE NERR=0//
     1       34H THE CURRENT ERROR MESSAGE FOLLOWS///)
      CALL E9RINT(MESSG,NW,NERR,.TRUE.)
      ITEMP=I8SAVE(1,1,.TRUE.)
      GO TO 50
C
C  SET LERROR AND TEST FOR A PREVIOUS UNRECOVERED ERROR.
C
 20   IF (I8SAVE(1,NERR,.TRUE.).EQ.0) GO TO 30
C
      WRITE(IWUNIT,9002)
 9002 FORMAT(23H1ERROR    3 IN SETERR -,
     1       48H AN UNRECOVERED ERROR FOLLOWED BY ANOTHER ERROR.//
     2       48H THE PREVIOUS AND CURRENT ERROR MESSAGES FOLLOW.///)
      CALL EPRINT
      CALL E9RINT(MESSG,NW,NERR,.TRUE.)
      GO TO 50
C
C  SAVE THIS MESSAGE IN CASE IT IS NOT RECOVERED FROM PROPERLY.
C
 30   CALL E9RINT(MESSG,NW,NERR,.TRUE.)
C
      IF (IOPT.EQ.1 .OR. IOPT.EQ.2) GO TO 40
C
C  MUST HAVE IOPT = 1 OR 2.
C
      WRITE(IWUNIT,9003)
 9003 FORMAT(42H1ERROR    4 IN SETERR - BAD VALUE FOR IOPT//
     1       34H THE CURRENT ERROR MESSAGE FOLLOWS///)
      GO TO 50
C
C  TEST FOR RECOVERY.
C
 40   IF (IOPT.EQ.2) GO TO 50
C
      IF (I8SAVE(2,0,.FALSE.).EQ.1) RETURN
C
      CALL EPRINT
      STOP
C
 50   CALL EPRINT
 60   CALL FDUMP
      STOP
C
      END
```

```
      SUBROUTINE ENTSRC(IROLD,IRNEW)
C
C  THIS ROUTINE RETURNS IROLD = LRECOV AND SETS LRECOV = IRNEW.
C
C  IF THERE IS AN ACTIVE ERROR STATE, THE MESSAGE IS PRINTED
C  AND EXECUTION STOPS.
C
C  IRNEW = 0 LEAVES LRECOV UNCHANGED, WHILE
C  IRNEW = 1 GIVES RECOVERY AND
C  IRNEW = 2 TURNS RECOVERY OFF.
C
C  ERROR STATES -
C
C    1 - ILLEGAL VALUE OF IRNEW.
C    2 - CALLED WHILE IN AN ERROR STATE.
C
      IF (IRNEW.LT.0 .OR. IRNEW.GT.2)
     1   CALL SETERR(31HENTSRC - ILLEGAL VALUE OF IRNEW,31,1,2)
C
      IROLD=I8SAVE(2,IRNEW,IRNEW.NE.0)
C
      IF (I8SAVE(1,0,.FALSE.) .NE. 0) CALL SETERR
     1   (39HENTSRC - CALLED WHILE IN AN ERROR STATE,39,2,2)
C
      RETURN
C
      END



      SUBROUTINE RETSRC(IROLD)
C
C  THIS ROUTINE SETS LRECOV = IRCLD.
C
C  IF THE CURRENT ERROR BECOMES UNRECOVERABLE,
C  THE MESSAGE IS PRINTED AND EXECUTION STOPS.
C
C  ERROR STATES -
C
C    1 - ILLEGAL VALUE OF IROLD.
C
      IF (IROLD.LT.1 .OR. IROLD.GT.2)
     1  CALL SETERR(31HRETSRC - ILLEGAL VALUE OF IROLD,31,1,2)
C
      ITEMP=I8SAVE(2,IROLD,.TRUE.)
C
      IF (IROLD.EQ.1 .OR. I8SAVE(1,0,.FALSE.).EQ.0) RETURN
C
        CALL EPRINT
        STOP
C
      END
```

```
      INTEGER FUNCTION NERROR(NERR)
C
C  RETURNS NERROR = NERR = THE VALUE OF THE ERROR FLAG LERROR.
C
      NERROR=I8SAVE(1,0,.FALSE.)
      NERR=NERROR
      RETURN
C
      END



      SUBROUTINE ERROFF
C
C  TURNS OFF THE ERROR STATE OFF BY SETTING LERROR=0.
C
      I=I8SAVE(1,0,.TRUE.)
      RETURN
C
      END



      SUBROUTINE EPRINT
C
C  THIS SUBROUTINE PRINTS THE LAST ERROR MESSAGE, IF ANY.
C
      INTEGER MESSG(1)
C
      DATA MESSG(1)/1H /
C
      CALL E9RINT(MESSG,1,1,.FALSE.)
      RETURN
C
      END
```

```
      SUBROUTINE E9RINT(MESSG,NW,NERR,SAVE)
C
C  THIS ROUTINE STORES THE CURRENT ERROR MESSAGE OR PRINTS THE OLD
C  ONE, IF ANY, DEPENDING ON WHETHER OR NOT SAVE = .TRUE. .
C
      INTEGER MESSG(NW)
      LOGICAL SAVE
C
C  MESSGP STORES AT LEAST THE FIRST 72 CHARACTERS OF THE PREVIOUS
C  MESSAGE. ITS LENGTH IS MACHINE DEPENDENT AND MUST BE AT LEAST
C
C       1 + 71/(THE NUMBER OF CHARACTERS STORED PER INTEGER WORD).
C
      INTEGER MESSGP(36),FMT(14),CCPLUS
C
C  START WITH NO PREVIOUS MESSAGE.
C
      DATA MESSGP(1)/1H1/, NWP/0/, NERRP/0/
C
C  SET UP THE FORMAT FOR PRINTING THE ERROR MESSAGE.
C  THE FORMAT IS SIMPLY (A1,14X,72AXX) WHERE XX=I1MACH(6) IS THE
C  NUMBER OF CHARACTERS STORED PER INTEGER WORD.
C
      DATA CCPLUS  / 1H+ /
C
      DATA FMT( 1) / 1H( /
      DATA FMT( 2) / 1HA /
      DATA FMT( 3) / 1H1 /
      DATA FMT( 4) / 1H, /
      DATA FMT( 5) / 1H1 /
      DATA FMT( 6) / 1H4 /
      DATA FMT( 7) / 1HX /
      DATA FMT( 8) / 1H, /
      DATA FMT( 9) / 1H7 /
      DATA FMT(10) / 1H2 /
      DATA FMT(11) / 1HA /
      DATA FMT(12) / 1HX /
      DATA FMT(13) / 1HX /
      DATA FMT(14) / 1H) /
C
      IF (.NOT.SAVE) GO TO 20
C
C  SAVE THE MESSAGE.
C
        NWP=NW
        NERRP=NERR
        DO 10 I=1,NW
 10     MESSGP(I)=MESSG(I)
C
        RETURN
C
 20   IF (I8SAVE(1,0,.FALSE.).EQ.0) GO TO 30
C
C  PRINT THE MESSAGE.
```

```
C
        IWUNIT=I1MACH(4)
        WRITE(IWUNIT,9000) NERRP
 9000   FORMAT(7H ERROR ,I4,4H IN )
C
        CALL S88FMT(2,I1MACH(6),FMT(12))
        WRITE(IWUNIT,FMT) CCPLUS,(MESSGP(I),I=1,NWP)
C
 30   RETURN
C
      END



      SUBROUTINE S88FMT(N,W,IFMT)
C
C  S88FMT  REPLACES IFMT(1), ... , IFMT(N) WITH
C  THE CHARACTERS CORRESPONDING TO THE N LEAST
C  SIGNIFICANT DIGITS OF W.
C
      INTEGER N,W,IFMT(N)
C
      INTEGER NT,WT,DIGITS(10)
C
      DATA DIGITS( 1) / 1H0 /
      DATA DIGITS( 2) / 1H1 /
      DATA DIGITS( 3) / 1H2 /
      DATA DIGITS( 4) / 1H3 /
      DATA DIGITS( 5) / 1H4 /
      DATA DIGITS( 6) / 1H5 /
      DATA DIGITS( 7) / 1H6 /
      DATA DIGITS( 8) / 1H7 /
      DATA DIGITS( 9) / 1H8 /
      DATA DIGITS(10) / 1H9 /
C
      NT = N
      WT = W
C
 10   IF (NT .LE. 0) RETURN
        IDIGIT = MOD( WT, 10 )
        IFMT(NT) = DIGITS(IDIGIT+1)
        WT = WT/10
        NT = NT - 1
        GO TO 10
C
      END
```

```
      INTEGER FUNCTION I8SAVE(ISW,IVALUE,SET)
C
C  IF (ISW = 1) I8SAVE RETURNS THE CURRENT ERROR NUMBER AND
C               SETS IT TO IVALUE IF SET = .TRUE. .
C
C  IF (ISW = 2) I8SAVE RETURNS THE CURRENT RECOVERY SWITCH
C               AND SETS IT TO IVALUE IF SET = .TRUE. .
C
      LOGICAL SET
C
      INTEGER IPARAM(2)
C
      EQUIVALENCE (IPARAM(1),LERROR)
      EQUIVALENCE (IPARAM(1),LRECOV)
C
C  START EXECUTION ERROR FREE AND WITH RECOVERY TURNED OFF.
C
      DATA LERROR/0/ , LRECOV/2/
C
      I8SAVE=IPARAM(ISW)
      IF (SET) IPARAM(ISW)=IVALUE
C
      RETURN
C
      END



      SUBROUTINE FDUMP
C
C  FDUMP IS INTENDED TO BE REPLACED BY A LOCALLY WRITTEN
C  VERSION WHICH PRODUCES A SYMBOLIC DUMP.  FAILING THIS,
C  IT SHOULD BE REPLACED BY A VERSION WHICH PRINTS  THE
C  SUBPROGRAM NESTING LIST.
C
      RETURN
C
      END
```

Machine Constants for Portable FORTRAN Libraries

Phyllis A. Fox

A.D. Hall

N.L. Schryer

Bell Laboratories,
Murray Hill, New Jersey 07974

ABSTRACT

One of the essential principles of programming for portability is the isolation of the machine dependent aspects of a program through the use of parameters or primitive subprograms. When FORTRAN is the vehicle for portability, it is usually more convenient to use primitive subprograms since the language does not provide even a rudimentary parameter substitution facility.

This paper describes a set of three FORTRAN function subprograms which provide a machine independent way of obtaining a number of important machine or operating system dependent constants. Among these constants are the "standard" I/O unit numbers, the word size in bits and characters, and the relative spacing of floating-point numbers. The choice of available constants was dictated primarily by experience, convenience and simplicity, with completeness and minimality being secondary considerations.

Machine Constants for Portable FORTRAN Libraries

Phyllis A. Fox

A.D. Hall

N.L. Schryer

Bell Laboratories,
Murray Hill, New Jersey 07974

1. Introduction

One of the essential principles of programming for portability is the isolation of the machine dependent aspects of a program in parameters or primitive subprograms. In this way the alterations that need to be made when moving a substantial program library from one environment to another are isolated, minimized and easily documented. When FORTRAN is the vehicle for portability, it is usually more convenient to use primitive subprograms since the language does not provide even a rudimentary parameter substitution facility.

This paper describes a set of three FORTRAN function subprograms which can be invoked to determine one of a number of basic machine or operating system dependent constants. These functions are

I1MACH	which delivers integer constants,
R1MACH	which delivers single-precision floating-point (real) constants, and
D1MACH	which delivers double-precision floating-point constants.

By requiring that all machine-dependent constants be incorporated into programs through references to these functions, the effort required to move a library to a new environment is minimized; only DATA statements in the three function subprograms need to be changed. A listing of the functions is included in the Appendix. Included in the comments are specific constants for the Honeywell 6000 Series, the IBM 360 and 370 Series, the SEL Systems 85/86, the UNIVAC 1100 Series, the DEC PDP 10 (KA and KI processors), the DEC PDP 11, and the CDC 6000 and 7000 Series.

The functions have a single integer argument indicating the particular constant desired. For example, I1MACH(2) is the logical unit number of the "standard" output unit, so the statements

```
IWUNIT = I1MACH(2)
WRITE (IWUNIT, 9003) ...
```

will write output (using Format statement 9003) on the standard output unit. As another example R1MACH(1) is the smallest positive single-precision number. If a program wishes to test how small a quantity, x, is becoming, perhaps to avoid underflow, it can test x against some reasonable multiple of R1MACH(1).

If the integer argument to R1MACH or D1MACH is out of range, the Error Handler [1] is called to deliver an appropriate message and terminate the run. In I1MACH, the message is output directly to avoid the possibility of a recursive call from the Error Handler.

2. The Constants

The constants cover four basic areas:

1) Logical unit numbers
2) Word size
3) Integer variables
4) Floating-point variables

These areas are discussed in the following subsections:

2.1. Logical Unit numbers

The FORTRAN run-time environment in many operating systems has several logical unit numbers which have preassigned (or default) associations with particular "devices". For example, logical unit 5 is frequently the standard input unit and logical unit 6 is frequently the standard output unit. Often there are also logical unit numbers assigned for punched output and error messages.

For any particular system, the logical unit numbers are available as follows:

I1MACH(1) = the standard input unit

I1MACH(2) = the standard output unit

I1MACH(3) = the standard punch unit

I1MACH(4) = the standard error message unit

2.2. Word Size

A **word** is defined to be that machine storage element allocated to an INTEGER or REAL variable. We define its size in both bits and characters as follows:

I1MACH(5) = the number of bits per word

I1MACH(6) = the number of characters per word

These allow subprograms which perform character or bit manipulation to be suitably parameterized.

2.3. Integer Variables

We assume that the permissible values of INTEGER variables are represented in the s-digit, base-a form:

$$\pm (x_{s-1}\, a^{s-1} + x_{s-2}\, a^{s-2} + \cdots + x_1\, a + x_0)$$

where $0 \leqslant x_i < a$ for $i = 0, \ldots, s-1$. We then have,

I1MACH(7) = a

I1MACH(8) = s

and the largest integer is,

I1MACH(9) = $a^s - 1$

Although I1MACH(9) can be computed from I1MACH(7) and I1MACH(8), it is provided because a straightforward evaluation of the formula may cause overflow. Note also that I1MACH(8) need not be directly related to I1MACH(5). For instance, on the CDC 6000 Series, integers have 48 bits of magnitude and 1 sign bit, yet there are 60 bits in a word.

2.4. Floating-point Variables

We assume that floating-point numbers are represented in the t-digit, base b form:

$$\pm b^e \left(\frac{x_1}{b} + \frac{x_2}{b^2} + \cdots + \frac{x_t}{b^t}\right)$$

where $0 \leqslant x_i < b$ for $i = 1, \ldots, t$, $0 < x_1$ and $e_{\min} \leqslant e \leqslant e_{\max}$. For a particular machine, we choose values for the parameters, t, $e_{\min}$, and $e_{\max}$, such that all numbers expressible in this form are representable by the hardware and usable from FORTRAN. Note that the formula is symmetrical under negation but not reciprocation. On some machines a small portion of the range of permissible numbers may be excluded.

For both single and double precision we have,

$$\text{I1MACH(10)} = b$$

In order to accommodate machines such as the CDC 6000 Series which put the b-point on the right, we must concede the possibility that the magnitude of $e_{\min}$ may be substantially smaller than $e_{\max}$. Thus, for single-precision floating-point we have:

$$\text{I1MACH(11)} = t$$

$$\text{I1MACH(12)} = e_{\min}$$

$$\text{I1MACH(13)} = e_{\max}$$

For double precision, b remains the same, but t, $e_{\min}$, and $e_{\max}$ are replaced by T, $E_{\min}$, and $E_{\max}$, as follows:

$$\text{I1MACH(14)} = T > t$$

$$\text{I1MACH(15)} = E_{\min} \leqslant e_{\min}$$

$$\text{I1MACH(16)} = E_{\max} \geqslant e_{\max}$$

3. Derived Quantities

We now describe a number of derived floating-point quantities which frequently are used in mathematical software. Although redundant, in the sense that they can be computed from previously given quantities, they are provided for efficiency and convenience. It is recommended that the derived quantities be used whenever possible in case the defining equations have to be generalized to accommodate some future machine architecture.

The smallest positive single and double precision magnitudes are given by:

$$\text{R1MACH(1)} = b^{e_{\min} - 1}$$

$$\text{D1MACH(1)} = b^{E_{\min} - 1}$$

The largest single and double precision magnitudes are given by:

$$\text{R1MACH}(2) = b^{e_{\max}}(1 - b^{-t})$$

$$\text{D1MACH}(2) = b^{E_{\max}}(1 - b^{-T})$$

The smallest relative spacings between adjacent single-precision or double-precision values are given by:

$$\text{R1MACH}(3) = b^{-t}$$

$$\text{D1MACH}(3) = b^{-T}$$

The largest relative spacings between adjacent single-precision or double-precision values are given by:

$$\text{R1MACH}(4) = b^{(1-t)}$$

$$\text{D1MACH}(4) = b^{(1-T)}$$

The largest relative spacing is the smallest value of ϵ that can safely be used in tests for relative error of the form $|(x-y)/x| \leqslant \epsilon$. It is also the smallest positive value of δ for which $1+\delta$ is not equal to 1 and is known as Wilkinson's error constant [2].

The logarithm of the base b is given by:

$$\text{R1MACH}(5) = \log_{10} b$$

$$\text{D1MACH}(5) = \log_{10} b$$

4. Decimal Input-Output

In some applications, particularly input-output, it is often useful to know the basic relationships between the internal representation of numbers and an external decimal representation. Some of the simpler relationships are summarized below. More detail can be found in [3].

For output, one usually wants to know how much space to allow for the decimal representation of an internal number. In the case of integers, the number s' of decimal places that are needed is given by

$$s' = \left\lceil s \log_{10} a \right\rceil ,$$

where a and s are defined in Sec. 2.3, and where $\lceil x \rceil$ denotes the smallest integer not less than x.

For single-precision floating-point, the situation is slightly more complex. If the external representation is of the form $m10^{e'}$ with $10^{-1} \leqslant m < 10$, then the minimum and maximum values of e' are:

$$e'_{\min} = \left\lfloor (e_{\min} - 1)\log_{10} b \right\rfloor + 1$$

$$e'_{\max} = \left\lfloor e_{\max} \log_{10} b \right\rfloor ,$$

where b, $e_{\min}$ and $e_{\max}$ are defined in Sec. 2.4. Here, $\lfloor x \rfloor$ denotes the largest integer not exceeding x.

The number of decimal places required for the decimal exponent is therefore

$$\left\lceil \log_{10} (\max (|e'_{\min}|, |e'_{\max}|)) \right\rceil$$

To determine the number of decimal places to allow for m, we observe that integers in the range 0 to $b^t - 1$ can be represented exactly in single-precision floating-point. If these are to be represented exactly on output, then the number t' of decimal places required is

$$t' = \left\lfloor t \log_{10} b \right\rfloor ,$$

where t and b are defined in Sec. 2.4. Relations similar to those given above hold for double-precision.

It should be noted that a decimal floating-point system carrying t' significant digits has a smallest relative spacing which is less than or equal to the smallest relative spacing of our assumed internal representation.

For input, one usually wants to know the approximate ranges of decimal numbers which can be represented in the machine. For instance, all integers of s'' decimal digits, where

$$s'' = \left\lfloor s \log_{10} a \right\rfloor$$

can be represented internally. Of course, it is possible that some larger integers can be represented, but a more complicated test would be needed.

All single-precision floating-point numbers of the form $m10^{e''}$ where $10^{-1} \leqslant m < 10$ and

$$\left\lfloor (e_{\min} - 1) \log_{10} b \right\rfloor + 1 \leqslant e'' \leqslant \left\lfloor e_{\max} \log_{10} b \right\rfloor$$

can be approximated in the machine. Similar relations hold for double-precision.

5. Programming Hints

In some cases, particularly inner loops, it may be desirable to avoid repeated calls to the functions described above. The obvious technique is to retrieve the needed values before entering the loop, but there are cases where substantial overhead may be incurred, even by this technique. One way to eliminate repeated calls to these functions in a portable (but non-standard) way is to use a carefully constructed "first-time" switch.

For example, to retrieve R1MACH(4) on first entry to a subprogram, the following coding can be used:

```
      REAL MCHEPS
      ...
      DATA MCHEPS / 0.0 /
      ...
      IF (MCHEPS .EQ. 0.0) MCHEPS = R1MACH(4)
```

If more than one value is to be obtained in this way, the following coding will suffice:

```
      REAL SMALL, LARGE, MCHEPS
      ...
      DATA SMALL, LARGE, MCHEPS / 3*0.0 /
      ...
      IF (SMALL .NE. 0.0) GO TO 10
      SMALL  = R1MACH(1)
      LARGE  = R1MACH(2)
      MCHEPS = R1MACH(4)
   10 CONTINUE
```

To ensure portability it is essential that *all values* obtained in this way be initialized in a DATA statement. If not, some operating systems (notably Burroughs) will not preserve the values from one subroutine call to the next.

References

[1] Hall, A.D., and Schryer, N.L., "A Centralized Error Handling Facility for Portable FORTRAN Libraries", March 3, 1975.

[2] Wilkinson, J.H., **Rounding Errors in Algebraic Processes,** Prentice-Hall, Inc., 1963.

[3] Matula, D.H., "In-and-Out Conversions", *Comm. ACM 11* (Nov. 1968), 47.

```
C
C     I1MACH(13) = EMAX, THE LARGEST EXPONENT E.
C
C   DOUBLE-PRECISION
C
C     I1MACH(14) = T, THE NUMBER OF BASE-B DIGITS.
C
C     I1MACH(15) = EMIN, THE SMALLEST EXPONENT E.
C
C     I1MACH(16) = EMAX, THE LARGEST EXPONENT E.
C
C   TO ALTER THIS FUNCTION FOR A PARTICULAR ENVIRONMENT,
C   THE DATA STATEMENTS FOR THE HONEYWELL 6000 SERIES
C   SHOULD BE TURNED INTO COMMENTS BY ADDING A C IN COLUMN 1.
C   THE DESIRED SET OF DATA STATEMENTS SHOULD BE ACTIVATED BY
C   REMOVING THE C FROM COLUMN 1.  ALSO, THE VALUES OF
C   I1MACH(1) - I1MACH(4) SHOULD BE CHECKED FOR CONSISTENCY
C   WITH THE LOCAL OPERATING SYSTEM.
C
      INTEGER IMACH(16),OUTPUT
C
      EQUIVALENCE (IMACH(4),OUTPUT)
C
C     MACHINE CONSTANTS FOR THE HONEYWELL 6000 SERIES.
C
      DATA IMACH( 1) /   5/
      DATA IMACH( 2) /   6/
      DATA IMACH( 3) /  43/
      DATA IMACH( 4) /   6/
      DATA IMACH( 5) /  36/
      DATA IMACH( 6) /   6/
      DATA IMACH( 7) /   2/
      DATA IMACH( 8) /  35/
      DATA IMACH( 9) /O377777777777/
      DATA IMACH(10) /   2/
      DATA IMACH(11) /  27/
      DATA IMACH(12) /-127/
      DATA IMACH(13) / 127/
      DATA IMACH(14) /  63/
      DATA IMACH(15) /-127/
      DATA IMACH(16) / 127/
C
C     MACHINE CONSTANTS FOR THE IBM 360 AND 370 SERIES,
C     AND THE SEL SYSTEMS 85/86.
C
C     DATA IMACH( 1) /   5/
C     DATA IMACH( 2) /   6/
C     DATA IMACH( 3) /   7/
C     DATA IMACH( 4) /   6/
C     DATA IMACH( 5) /  32/
C     DATA IMACH( 6) /   4/
C     DATA IMACH( 7) /   2/
C     DATA IMACH( 8) /  31/
```

```
C      DATA IMACH( 9) /Z7FFFFFFF/
C      DATA IMACH(10) /  16/
C      DATA IMACH(11) /   6/
C      DATA IMACH(12) / -64/
C      DATA IMACH(13) /  63/
C      DATA IMACH(14) /  14/
C      DATA IMACH(15) / -64/
C      DATA IMACH(16) /  63/
C
C     MACHINE CONSTANTS FOR THE CDC 6000 AND 7000 SERIES.
C
C      DATA IMACH( 1) /   5/
C      DATA IMACH( 2) /   6/
C      DATA IMACH( 3) /   7/
C      DATA IMACH( 4) /   6/
C      DATA IMACH( 5) /  60/
C      DATA IMACH( 6) /  10/
C      DATA IMACH( 7) /   2/
C      DATA IMACH( 8) /  48/
C      DATA IMACH( 9) /00007777777777777777/
C      DATA IMACH(10) /   2/
C      DATA IMACH(11) /  48/
C      DATA IMACH(12) / -974/
C      DATA IMACH(13) / 1070/
C      DATA IMACH(14) /  96/
C      DATA IMACH(15) / -974/
C      DATA IMACH(16) / 1070/
C
C     MACHINE CONSTANTS FOR THE PDP-10 (KA PROCESSOR).
C
C      DATA IMACH( 1) /   5/
C      DATA IMACH( 2) /   6/
C      DATA IMACH( 3) /   5/
C      DATA IMACH( 4) /   6/
C      DATA IMACH( 5) /  36/
C      DATA IMACH( 6) /   5/
C      DATA IMACH( 7) /   2/
C      DATA IMACH( 8) /  35/
C      DATA IMACH( 9) / "377777777777/
C      DATA IMACH(10) /   2/
C      DATA IMACH(11) /  27/
C      DATA IMACH(12) /-128/
C      DATA IMACH(13) / 127/
C      DATA IMACH(14) /  54/
C      DATA IMACH(15) /-128/
C      DATA IMACH(16) / 127/
C
C     MACHINE CONSTANTS FOR THE PDP-10 (KI PROCESSOR).
C
C      DATA IMACH( 1) /   5/
C      DATA IMACH( 2) /   6/
C      DATA IMACH( 3) /   5/
C      DATA IMACH( 4) /   6/
```

```
C      DATA IMACH( 5) /   36/
C      DATA IMACH( 6) /    5/
C      DATA IMACH( 7) /    2/
C      DATA IMACH( 8) /   35/
C      DATA IMACH( 9) / "377777777777/
C      DATA IMACH(10) /    2/
C      DATA IMACH(11) /   27/
C      DATA IMACH(12) /-128/
C      DATA IMACH(13) /  127/
C      DATA IMACH(14) /   62/
C      DATA IMACH(15) /-128/
C      DATA IMACH(16) /  127/
C
C      MACHINE CONSTANTS FOR THE PDP-11.
C
C      DATA IMACH( 1) /    5/
C      DATA IMACH( 2) /    6/
C      DATA IMACH( 3) /    5/
C      DATA IMACH( 4) /    6/
C      DATA IMACH( 5) /   32/
C      DATA IMACH( 6) /    4/
C      DATA IMACH( 7) /    2/
C      DATA IMACH( 8) /   31/
C      DATA IMACH( 9) / 2147483647/
C      DATA IMACH(10) /    2/
C      DATA IMACH(11) /   24/
C      DATA IMACH(12) /-127/
C      DATA IMACH(13) /  127/
C      DATA IMACH(14) /   56/
C      DATA IMACH(15) /-127/
C      DATA IMACH(16) /  127/
C
C      MACHINE CONSTANTS FOR THE UNIVAC 1100 SERIES.
C
C      DATA IMACH( 1) /    5/
C      DATA IMACH( 2) /    6/
C      DATA IMACH( 3) /    7/
C      DATA IMACH( 4) /    6/
C      DATA IMACH( 5) /   36/
C      DATA IMACH( 6) /    6/
C      DATA IMACH( 7) /    2/
C      DATA IMACH( 8) /   35/
C      DATA IMACH( 9) /0377777777777/
C      DATA IMACH(10) /    2/
C      DATA IMACH(11) /   27/
C      DATA IMACH(12) /-128/
C      DATA IMACH(13) /  127/
C      DATA IMACH(14) /   61/
C      DATA IMACH(15) /-1024/
C      DATA IMACH(16) / 1023/
C
       IF (I .LT. 1 .OR. I .GT. 16) GO TO 10
C
```

```
      I1MACH=IMACH(I)
      RETURN
C
 10   WRITE(OUTPUT,9000)
 9000 FORMAT(39H1ERROR    1 IN I1MACH - I OUT OF BOUNDS)
C
      CALL FDUMP
C
      STOP
C
      END
```

The single-precision floating-point constant
FORTRAN function subprogram R1MACH

```
      REAL FUNCTION R1MACH(I)
C
C  SINGLE-PRECISION MACHINE CONSTANTS
C
C  R1MACH( 1) = B**(EMIN-1), THE SMALLEST POSITIVE MAGNITUDE.
C
C  R1MACH( 2) = B**EMAX*(1 - B**(-T)), THE LARGEST MAGNITUDE.
C
C  R1MACH( 3) = B**(-T), THE SMALLEST RELATIVE SPACING.
C
C  R1MACH( 4) = B**(1-T), THE LARGEST RELATIVE SPACING.
C
C  R1MACH( 5) = LOG10(B)
C
C  TO ALTER THIS FUNCTION FOR A PARTICULAR ENVIRONMENT,
C  THE DATA STATEMENTS FOR THE HONEYWELL 6000 SERIES
C  SHOULD BE TURNED INTO COMMENTS BY ADDING A C IN COLUMN 1.
C  THE DESIRED SET OF DATA STATEMENTS SHOULD BE ACTIVATED BY
C  REMOVING THE C FROM COLUMN 1.
C
C  WHERE POSSIBLE, OCTAL OR HEXADECIMAL CONSTANTS HAVE BEEN USED
C  TO SPECIFY THE CONSTANTS EXACTLY.
C
      REAL RMACH(5)
C
C     MACHINE CONSTANTS FOR THE HONEYWELL 6000 SERIES.
C
      DATA RMACH( 1) / O402400000000/
      DATA RMACH( 2) / O376777777777/
      DATA RMACH( 3) / O714400000000/
      DATA RMACH( 4) / O716400000000/
      DATA RMACH( 5) / O776464202324/
C
C     MACHINE CONSTANTS FOR THE IBM 360 AND 370 SERIES,
C     AND THE SEL STSTEMS 85/86.
C
C     DATA RMACH( 1) / Z00100000/
C     DATA RMACH( 2) / Z7FFFFFFF/
C     DATA RMACH( 3) / Z3B100000/
C     DATA RMACH( 4) / Z3C100000/
C     DATA RMACH( 5) / Z41134413/
C
C     MACHINE CONSTANTS FOR THE CDC 6000 AND 7000 SERIES.
C
C     DATA RMACH( 1) / 00014000000000000000B/
C     DATA RMACH( 2) / 37767777777777777777B/
C     DATA RMACH( 3) / 16404000000000000000B/
C     DATA RMACH( 4) / 16414000000000000000B/
C     DATA RMACH( 5) / 17164642023241175720B/
C
```

```
C      MACHINE CONSTANTS FOR THE PDP-10 (KA OR KI PROCESSOR).
C
C      DATA RMACH( 1) / "000400000000/
C      DATA RMACH( 2) / "377777777777/
C      DATA RMACH( 3) / "146400000000/
C      DATA RMACH( 4) / "147400000000/
C      DATA RMACH( 5) / "177464202324/
C
C      MACHINE CONSTANTS FOR THE PDP-11.
C
C      DATA RMACH( 1) / 0.29387358771E-38/
C      DATA RMACH( 2) / 0.17014117331E+39/
C      DATA RMACH( 3) / 0.59604644775E-07/
C      DATA RMACH( 4) / 0.11920928955E-06/
C      DATA RMACH( 5) / 0.30102999566    /
C
C      MACHINE CONSTANTS FOR THE UNIVAC 1100 SERIES.
C
C      DATA RMACH( 1) / O0000400000000/
C      DATA RMACH( 2) / O0377777777777/
C      DATA RMACH( 3) / O0146400000000/
C      DATA RMACH( 4) / O0147400000000/
C      DATA RMACH( 5) / O0177464202324/
C
       IF (I .LT. 1  .OR.  I .GT. 5)
     1    CALL SETERR(24HR1MACH - I OUT OF BOUNDS,24,1,2)
C
       R1MACH = RMACH(I)
       RETURN
C
       END
```

The double-precision floating-point constant
FORTRAN function subprogram D1MACH

```
      DOUBLE PRECISION FUNCTION D1MACH(I)
C
C  DOUBLE-PRECISION MACHINE CONSTANTS
C
C  D1MACH( 1) = B**(EMIN-1), THE SMALLEST POSITIVE MAGNITUDE.
C
C  D1MACH( 2) = B**EMAX*(1 - B**(-T)), THE LARGEST MAGNITUDE.
C
C  D1MACH( 3) = B**(-T), THE SMALLEST RELATIVE SPACING.
C
C  D1MACH( 4) = B**(1-T), THE LARGEST RELATIVE SPACING.
C               THE LARGEST RELATIVE SPACING
C
C  D1MACH( 5) = LOG10(B)
C
C  TO ALTER THIS FUNCTION FOR A PARTICULAR ENVIRONMENT,
C  THE DATA STATEMENTS FOR THE HONEYWELL 6000 SERIES
C  SHOULD BE TURNED INTO COMMENTS BY ADDING A C IN COLUMN 1.
C  THE DESIRED SET OF DATA STATEMENTS SHOULD BE ACTIVATED BY
C  REMOVING THE C FROM COLUMN 1.
C
C  WHERE POSSIBLE, OCTAL OR HEXADECIMAL CONSTANTS HAVE BEEEN USED
C  TO SPECIFY THE CONSTANTS EXACTLY WHICH HAS IN SOME CASES
C  REQUIRED THE USE OF EQUIVALENT INTEGER ARRAYS.
C
      INTEGER SMALL(2)
      INTEGER LARGE(2)
      INTEGER RIGHT(2)
      INTEGER DIVER(2)
      INTEGER LOG10(2)
C
      DOUBLE PRECISION DMACH(5)
C
      EQUIVALENCE (DMACH(1),SMALL(1))
      EQUIVALENCE (DMACH(2),LARGE(1))
      EQUIVALENCE (DMACH(3),RIGHT(1))
      EQUIVALENCE (DMACH(4),DIVER(1))
      EQUIVALENCE (DMACH(5),LOG10(1))
C
C     MACHINE CONSTANTS FOR THE HONEYWELL 6000 SERIES.
C
      DATA SMALL(1),SMALL(2) / O402400000000,O000000000000/
      DATA LARGE(1),LARGE(2) / O376777777777,O777777777777/
      DATA RIGHT(1),RIGHT(2) / O604400000000,O000000000000/
      DATA DIVER(1),DIVER(2) / O606400000000,O000000000000/
      DATA LOG10(1),LOG10(2) / O776464202324,O117571775714/
C
C     MACHINE CONSTANTS FOR THE IBM 360 AND 370 SERIES,
C     AND THE SEL SYSTEMS 85/86.
C
```

```
C      DATA SMALL(1),SMALL(2) / Z00100000,Z00000000/
C      DATA LARGE(1),LARGE(2) / Z7FFFFFFF,ZFFFFFFFF/
C      DATA RIGHT(1),RIGHT(2) / Z33100000,Z00000000/
C      DATA DIVER(1),DIVER(2) / Z34100000,Z00000000/
C      DATA LOG10(1),LOG10(2) / Z41134413,Z509F79FF/
C
C      MACHINE CONSTANTS FOR THE CDC 6000 AND 7000 SERIES.
C
C      DATA SMALL(1) / 00014000000000000000B/
C      DATA SMALL(2) / 00000000000000000000B/
C
C      DATA LARGE(1) / 37767777777777777777B/
C      DATA LARGE(2) / 37167777777777777777B/
C
C      DATA RIGHT(1) / 15604000000000000000B/
C      DATA RIGHT(2) / 00000000000000000000B/
C
C      DATA DIVER(1) / 15614000000000000000B/
C      DATA DIVER(2) / 00000000000000000000B/
C
C      DATA LOG10(1) / 17164642023241175717B/
C      DATA LOG10(2) / 16367571421742254654B/
C
C      MACHINE CONSTANTS FOR THE PDP-10 (KA PROCESSOR).
C
C      DATA SMALL(1),SMALL(2) / "000400000000,"000000000000/
C      DATA LARGE(1),LARGE(2) / "377777777777,"344777777777/
C      DATA RIGHT(1),RIGHT(2) / "113400000000,"000000000000/
C      DATA DIVER(1),DIVER(2) / "114400000000,"000000000000/
C      DATA LOG10(1),LOG10(2) / "177464202324,"144117571776/
C
C      MACHINE CONSTANTS FOR THE PDP-10 (KI PROCESSOR).
C
C      DATA SMALL(1),SMALL(2) / "000400000000,"000000000000/
C      DATA LARGE(1),LARGE(2) / "377777777777,"377777777777/
C      DATA RIGHT(1),RIGHT(2) / "103400000000,"000000000000/
C      DATA DIVER(1),DIVER(2) / "104400000000,"000000000000/
C      DATA LOG10(1),LOG10(2) / "177464202324,"476747767461/
C
C      MACHINE CONSTANTS FOR THE PDP-11.
C
C      DATA DMACH(1) / 0.2938735877055719  D-38/
C      DATA DMACH(2) / 0.170141183460469229D+39/
C      DATA DMACH(3) / 0.138777878078144568D-16/
C      DATA DMACH(4) / 0.277555756156289135D-16/
C      DATA DMACH(5) / 0.301029995663981195D+00/
C
C      MACHINE CONSTANTS FOR THE UNIVAC 1100 SERIES.
C
C      DATA SMALL(1),SMALL(2) / O0000040000000,O0000000000000/
C      DATA LARGE(1),LARGE(2) / O0377777777777,O0777777777777/
C      DATA RIGHT(1),RIGHT(2) / O0170540000000,O0000000000000/
C      DATA DIVER(1),DIVER(2) / O0170640000000,O0000000000000/
```

```
C       DATA LOG10(1),LOG10(2) / 0177746420232,04117571775 72/
C
        IF (I .LT. 1  .OR.  I .GT. 5)
     1    CALL SETERR(24HD1MACH - I OUT OF BOUNDS,24,1,2)
C
        D1MACH = DMACH(I)
        RETURN
C
        END
```