Computing Science Technical Report #47

THE PORT MATHEMATICAL SUBROUTINE LIBRARY

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

May 1, 1977

Part 1: Description

Part 2: Utility program listings:

Machine constants
Error handling
Stack allocation

September 1976
Revised May 1977

The PORT Mathematical Subroutine Library

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

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The development at Bell Laboratories of PORT, a library of portable Fortran programs for numerical computation, is discussed.

Portability is achieved by careful language specification, together with the key technique of specifying computer classes by means of pre-defined machine constants.

The library is built around an automatic error-handling facility and a dynamic storage allocation scheme, both of which are implemented portably. These, together with the modular structure of the library, lead to simplified calling sequences and ease of use.

March 22, 1979

The PORT Mathematical Subroutine Library

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

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

We have celebrated the 25th anniversary of computer program libraries by producing another.

The library is called PORT. Our interest in developing it, and particularly our resolve to create a portable library, can best be motivated by sketching a brief history of mathematical subroutine libraries. A look at the tremendous effort that went into the early machine-dependent libraries, each to be ultimately thrown on the scrap heap along with its passé computer, and a glance at more recent developments involving either maintenance or generation of several machine-dependent versions may indicate our drive to construct a single portable library. But first a word on the historical setting.

It was indeed in 1951 that Maurice V. Wilkes, David J. Wheeler, and Stanley Gill, all of the University of Cambridge, published their book: **The Preparation of Programs for an Electronic Digital Computer**, subtitled *With special reference to the EDSAC and the use of a library of subroutines*, [Wilkes *(1951)]*. Their library was in machine language, but in that era the need for moving the library to another computer, and the trauma involved, had not yet been experienced. The very thought of having a library was new.

Since that time many libraries have been developed. Rice (1971a, Chapter 1) gives some historical notes including the remark that the *Communications of the Association for Computing Machinery* published 73 algorithms during 1960-1961. Also in 1961 IBM made the decision to enter the software field [Battiste (1971), page 121], and the SSP library was launched. Initially IBM provided SSP free with the the hardware, but by 1971 IBM was charging (around $100/month) for the new SL-Math library [IBM (1971)] developed in Germany for the 360/370/1130/1800 IBM computers.

Other libraries dating from the '60's, and we mention only a few, include the Monsanto Company's subroutine library started in 1967, [Dickinson *et al.* (1971)], and the Boeing library [Newbery (1971)], developed about the same time. The Harwell Atomic Energy Research Establishment, in England, also in 1967, was converting their subroutine library, developed in 1963 for the IBM 7030 (STRETCH) computer, to an IBM 360 version, [Hopper (1973)]. In December of 1967 the Sandia Mathematical Library Project [Jones and Bailey (1976)] was initiated, and doubtless many other library projects were underway, at that time, across the computer world.

By the early '70's an appreciation of the magnitude of the effort required to establish libraries was beginning to be felt. CDC, rather than developing a mathematical library from scratch, purchased the high-quality Boeing library. Also in 1970 a commercial library became available when the International Mathematical and Statistical Libraries (IMSL) was incorporated [Johnson (1971)], and produced Library 1 in Fortran for the IBM/360-370 series. Another approach to avoiding duplication of effort was taken by the NATS (National Activity to Test Software) group, [Boyle *et al.* (1972)], which was established to produce quality special-purpose software packages. The first of these, the eigenvalue-eigenvector package, EISPACK [Smith *et al.* (1976)], is already in a second release, the package for special functions, FUNPACK, has appeared, [Cody (1975)], and MINPACK(minimization package) and LINPACK(linear equations package) are being developed.

The libraries discussed above are written in Fortran; in other countries the language problem (Algol vs. Fortran) adds to the difficulties. In England, the Numerical Algorithms Group (NAG), [Ford and Sayers (1976)], has for some time been furnishing a large library in either Algol or Fortran to a variety of types of computers across the universities of that country.

Throughout the development of these software packages, the effort involved in terms of manpower, time, and money, not only to develop the packages, but to adapt them to particular computers, has been a source of wonder, and the desire to have the software portable has been growing. Even the early (1967) Monsanto library emphasized machine independence [Dickinson *et al.* (1971, page 143)], and Rice (1971b) has emphasized the importance of making quality software transportable. In fact he makes the rather striking observation that, "It has been estimated that 60%-90% of all research and development work is a duplication of previous work, and it is easy to believe that this applies to mathematical software with perhaps an even higher percentage than 90." The development of mathematical subroutine libraries is certainly a case in point.

At Bell Labs, in 1967, Gentleman and Traub [1968] proposed the development of a machine-independent library using a Fortran-Algol interface to alleviate the language problem. Three sets of programs were implemented in that effort: DESUB (differential equation solution), NSEVB (eigenvalues and eigenvectors of nonsymmetric matrices), and MIDAS (linear equation solution). The work described in the present paper is an independent effort, initiated in 1973. PORT does not include any of the earlier programs, but the experience gained in that work has been useful.

Preview

In the next section we discuss portability: the definition of the term, the several factors inhibiting portability, the solutions taken by others in some of the libraries we have discussed, and the procedures we have applied toward portability. In the following section of the paper we describe in more detail the structure of the PORT library — in particular the simplified calling sequences, the error monitoring and error handling, and the dynamic storage allocation — relating each topic to the corresponding approaches used in other libraries. Finally, we give an overview of the contents of PORT, that is, the subprograms included in the current edition. To keep the size of the paper down somewhat, we defer discussion of other aspects of the library to another time. Relevant topics not covered here include algorithm selection, implementation, testing, refereeing, documentation, certification, and distribution, as well as installation procedures and maintenance.

Part 2 of this report contains the listings for the basic PORT utilities, and includes a brief summary of their use.

2. Portability

Definitions of 'portability' are rampant; the view is so clouded that we might do well to adopt an entirely new term, possibly that used by Waite (1970) in another connection: 'mobility.' The problem is that there is a continuum of degrees of portability, from 'completely' to 'not at all'. Clearly an unportable Fortran program could be made even less so by recoding it in assembler language. W. S. Brown (1970), recognizing this matter of degree, offered the definition: "A program or programming system is called 'portable' if the effort required to move it into a new environment is much less than the effort that would be required to reprogram it for the new environment." The environment, of course, includes the computer, the compiler, the operating system and the particular computer hardware/software configuration.

Aird *et al.* (1975) define four distinct concepts to span the high end of the spectrum: (1) portable, (2) converter portable, (3) processor portable, and (4) transportable. They call a program "*portable*, across a set of computer-compiler environments if, without any modification, it can be compiled and executed, according to defined performance criteria, for every member of the set." The second and third definitions cover the cases where a 'master version' of the library is established and a processor is provided that can particularize the program to a particular member of the set of computer-compiler environments. The term, 'converter portable,' is distinguished from 'processor portable' in indicating that the unprocessed version runs, without alteration, in at least one environment. Finally they term a program 'transportable' if enough information is available to guide a required particularization, perhaps even by hand. (Note that this is an imprecise paraphrase of their very careful definitions.)

The IFIP Working Group 2.5 (on Numerical Software) has proposed, in a working paper draft, the following definitions [Ford and Smith (1975)]:

> *Portable*
>
> A program is portable over a given range of machines and compilers if *without any alteration*, it can compile and run to satisfy specified performance criteria on that range.
>
> *Transportable*
>
> In transferring a program between members of a given range of machines and compilers, some changes may be necessary to the base version before it satisfies specified performance criteria on each of the machines and compilers. The program is *transportable* if
>
> (1) the changes lend themselves to mechanical implementation by a processor.
>
> (2) ideally the changes are limited in number, extent and complexity.

In PORT we have so far taken the view that the library subprograms must be *portable* in the IFIP sense, with the exception that the three machine-constant defining functions must be particularized to the host computer once, at installation. Section 2.2 of the paper discusses the details of the approach.

What gets in the way of portability?

A routine that runs well on one computer may run badly or not at all on another. Its failure may be due to language and compiler differences, to differing word structures, to variations in arithmetic (both static number representation and dynamic performance), or to differences in operating systems. If a library of mathematical subroutines aspires to portability, each of these problems must be solved.

2.1. Overview of various libraries and their approach to portability

There is general agreement among the developers of mathematical libraries on one major point: *only one source should be maintained.* But there is a wide divergence of opinion as to what the source should look like, and particularly how much coded information it should contain. Some would urge that the master source should work in at least one environment without change, that is, be converter portable in terms of the definition cited above. The developers of the IMSL library take this view; the library is based on a source called the basis deck, which exists as an executable deck in one environment, but which contains control information, in the form of keyed comment cards, permitting a converter program to generate a deck for another computer-compiler environment, or to generate a double-precision version of the program. [Aird *et al.* (1975)]. The NATS project in its initial work with EISPACK used a similar approach [Boyle and Dritz (1974)].

The NAG library is based on a master library file system consisting of card-image files and three main utility programs which operate on the files [Hague and Ford (1976)]. The programs include editors, selectors, and extractor-comparison programs, all written in Fortran with some assembly code routines. The programming took about 18 man-months to complete. A second version to be written in Algol 68 is under study.

Krogh (1974,1976) has developed a method, called the 'specializer language', for maintaining a composite source. The approach, which is rather like the IMSL approach, allows code to be specified for different machine environments and for different base precisions.

A few library projects are beginning to take the view that the master tape or composite source should be a more abstract vehicle. The NATS II system, in fact, is based on the concept of the 'abstract form' of a program combined with control programs [Boyle and Dritz (1974)]. One of these, the 'recognizer' can map Fortran programs written for a variety of computer systems into the abstract form. Then to generate particularized programs from the abstract form a 'formatter' program is used.

Most of the composite or master sources are based on a system of flagged comment cards, control cards and other types of record-based selection clues. Some use has been made of macro capabilities to generate particularized versions, and Boyle and Dritz (1974) have proposed actually parsing the Fortran input and storing the symbol table and parse tree as the master or composite tape.

In the various schemes for maintaining a master source proposed to date, it has generally been true that either the source, or the programs which generate particularized programs from the coded source, assume the existence of a finite set of specific machines; each program in the master source contains information on the attributes of the given computer environments. If this approach is used, the introduction of a new computer-compiler configuration into the scheme presents a major problem — in essence, portability has been defined only with respect to the initial set of computers.

A better way of approaching the problem, in our view, is to formulate an idealized, but robust, model of a computer from the standpoint of numerical computation. The model should be simple and yet apply to most existing (and many future) computers, and should be consonant with an ANSI Fortran computing environment. Then, given the parameters defining the model, portable software can be written from a truly machine-independent, but *model-dependent* orientation. To particularize the model for any given target site, appropriate values for the parameters can be set in a simple way. The model exemplified in PORT is described in some detail below; further details on floating-point arithmetic in the model are given by Brown (1977).

2.2. PORT and portability

The techniques used in the PORT library to make it easily portable are, then, the following:

(1) programs are written in a subset of ANSI Fortran

(2) the target environment is specified in terms of machine-dependent parameters

We have evidence of success to the extent that the PORT tape has been compiled and is in use on three IBM 360/370 computers, a UNIVAC 1100, two Honeywell 6000 series, a CDC Cyber '72 system, a Harris S220, and a PDP 11.

Language

The programming language used for PORT subprograms is restricted to the particular, portable subset of ANSI Fortran known as PFORT, described by Ryder (1974). Programs submitted to PORT are always sent through the PFORT Verifier program, described in that reference, to guarantee their adherence to this language requirement.

There are two non-ANSI Fortran usages made in PORT; both are valid for the usual production system. First, it is assumed that there is no runtime subscript range checking. Second, for some of the subprograms implementing the error handling and stack allocation it is assumed that a variable (local to a subprogram) that is initialized by a DATA statement and then changed within the subprogram, retains its most recently assigned value. If overlays are used by a programmer, care must be taken to avoid overlaying these few routines. These issues and others related to portability are discussed in more detail in Appendix A.

Although the ANSI Fortran Standard makes the assumption that LOGICAL, INTEGER, and REAL data are allocated one "storage unit" and that DOUBLE PRECISION and COMPLEX data are allocated two "storage units", the assumption is frequently violated in minicomputer Fortran systems. To allow for use of the library on these smaller computer systems, we have been careful, in formulating the dynamic storage scheme, to make it independent of the amounts of storage allocated to the different data types.

Specification of machine-dependent quantities

Very early on in the development of mathematical subroutine libraries, the importance of isolating the machine-dependent parameters and constants was recognized. Newbery (1971, page 155) notes that the Boeing library provided a single program whose function was to store these values in one place. EISPACK uses only two machine-dependent constants: RADIX, the base of the machine floating-point representation, and MACHEP (machine epsilon), the relative precision of the floating-point arithmetic. However, for more general libraries, it is convenient to have a number of important machine and operating system dependent constants available. Redish and Ward (1971), Aird *et al.* (1974), Krogh and Singletary (1976), and others have presented lists of machine-dependent constants and parameters. The IFIP Working Group 2.5 is studying the matter in some detail [Ford (1976)], and presumably will propose a standard set.

Once a set of values is decided upon, there still remains the question of getting them into the running programs. Four principal mechanisms come to mind:

(1) Dynamically sample the host computer using subroutines for discovering the base of the arithmetic, the number of digits and so on, an approach discussed by Malcolm (1972), Gentleman and Marovich (1974), and George (1975).

(2) Flag or mark the machine-dependent quantities in the master source tape, enabling the correct values to be generated when a particular machine-dependent version of the library is created. This approach is used, for example, in IMSL, [Aird *et al.* (1975)].

(3) Use a language (possibly a Fortran preprocessor) that allows global definitions of variables, and build a portable scheme based on that language's capabilities.

(4) Develop library subprograms which can be particularized for each target computer, and then called, during run time, to obtain desired machine-dependent values. Redish and Ward (1971) have proposed using this method. Their discussion is excellent. Also Ford and Sayers (1976) note that the NAG II system uses a similar approach, permitting machine-dependent numbers to be evaluated by procedure calls to a sub-library, called the 'constants and utilities library.'

Each of these mechanisms has its problems. The original version of (1) failed on computers, such as the Honeywell 6000 or the ICL 4130, for which the floating-point registers contain more digits than a word in storage. Later versions have had troubles stemming from the fact that assumptions have to be made on computer hardware or compiler design. The techniques used in (2) have been discussed above in Section 2.2. They have the advantage that the generated version can be tailored to be especially efficient for a given computer-compiler environment (including alters to get around known bugs), but a change of compiler, or a new computer require extensive changes in the master source and the programs that control it. These requirements, together with the fact that updates and corrections have to be generated in machine-dependent form before being sent out, make a sizeable staff of maintenance people necessary. Finally, the use of method (3) above, which may be the way of the future, means giving up (or extending) Fortran, and that is hard to do right now.

In PORT we use the fourth approach: three Fortran function subprograms are provided which can be invoked to determine basic machine or operating system dependent constants. When the library is moved to a new environment, only the DATA statements in these three subprograms need to be changed. Values are provided, in the library, for the Burroughs 5700/6700/7700, the CDC 6000/7000 Series, the Data General Eclipse S/200, the DEC PDP 10 (KA and KI processors), the DEC PDP 11, the Harris S220, the Honeywell 6000 Series, the IBM 360/370 Series, the SEL Systems 85/86, the UNIVAC 1100 Series, and the XEROX SIGMA 5/7/9; others can be added.

We have found this approach advantageous since the source code for the library, except for the three subprograms, remains unchanged from one environment to another. A potential drawback to the approach is the difficulty of writing portable programs for some areas of numerical computation. The field of special functions is the most trying: recursive algorithms are quite portable, but not always adequate, and rational function approximations are machine-dependent. We are looking into several techniques, including program generators and other tactics, and we have detected some promising directions.

But to return to our mechanism for specifying machine-dependent quantities, the three 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.

The function names stem from the PORT convention that subprograms which will not be called by the casual user are given names with a digit as the second character to help avoid name conflicts. 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(2) is the largest positive single-precision number, so if a program wishes to test, a priori, whether the product $x \times y$ will overflow,(where x, $y > 1$), it can include the test

IF (Y .GE. R1MACH(2)/X) GO TO *overflow*

(The ultra-precise reader may note that the subsequent multiplication might still overflow by as much as two round-off units, so the test should be shaded to be safe.)

If the integer argument to R1MACH or D1MACH is out of range, the error handling facility used in PORT, which is discussed in the next section, 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.

The constants provided in the function subprograms cover logical unit numbers, and certain properties of integer, floating-point, and character-string quantities. Care has been taken to distinguish the space configuration used by integers from that used by single-precision (REAL) quantities; for some computers this distinction is required since the concept of a computer 'word' for both types is not valid. (For instance, on the CDC 6000 Series, both integers and reals are stored in 60-bit words, yet integers have 48 bits of magnitude and 1 sign bit.) The model of a computer we represent is based entirely on Fortran *types*, and the values we provide completely specify the model. In fact, some redundancy has been included for purposes discussed a little later.

The following are specified:

Logical unit numbers

the standard input unit
the standard output unit
the standard punch unit
the standard error message unit

Integer and character storage

the number of bits per INTEGER storage unit
the number of characters per INTEGER storage unit

Integer variables

Let the values for integer variables be written in the s-digit, base-a form:

$$\pm(x_{s-1}a^{s-1}+x_{s-2}a^{s-2}+\cdots+x_1a+x_0)$$

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

Then we specify,

the base, a
the maximum number of digits, s
the largest integer, a^s-1.

Although the quantity, a^s-1, can easily be computed from s, and the base, a, it is provided because a naive evaluation of the formula would cause overflow on most machines. (Storage of integers as magnitude and sign or in a complement notation is not specified since PORT subprograms must be independent of the storage mode.)

Floating-point variables

If floating-point numbers are written 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 $e_{\min} \leqslant e \leqslant e_{\max}$, then 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. Also, for 2's complement machines care must be taken in assigning the values; see page 10.

Then we specify,

the base, b. for both single and double precision
the number, t, of base-b digits

In order to accommodate machines (such as the CDC 6000 Series) with 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 specify,

the minimum exponent, $e_{\min}$
the maximum exponent, $e_{\max}$

For double precision, b remains the same, but t, $e_{\min}$, and $e_{\max}$ are replaced by T, $E_{\min}$, and $E_{\max}$. Normally, we have $E_{\min} \leqslant e_{\min}$ and $E_{\max} \geqslant e_{\max}$, and $T > t$. However, in machines such as the CDC 6000 Series or the PDP-10 KA Processor, where double precision is implemented by software simulation, small double-precision floating-point numbers carry only t base-b significant digits. In such cases, we take $E_{\min}$ to be the exponent of the smallest number with T base-b significant digits, and it may be that, $E_{\min} > e_{\min}$.

The 16 values given above are all integers and are obtained by invoking the function I1MACH with the appropriate argument. The floating-point single-precision and double-precision quantities provided by the functions R1MACH and D1MACH can be derived from the given integer quantities, but are provided for efficiency and convenience.

The single-precision floating-point quantities provided in R1MACH are,

the smallest positive magnitude, $b^{e_{\min}-1}$
the largest magnitude, $b^{e_{\max}}(1-b^{-t})$
the smallest relative spacing between values, b^{-t}
the largest relative spacing between values, b^{1-t}
the logarithm of the base b, $\log_{10} b$

The relative spacing is $|\ (y-x)/x\ |$, when x and y are successive floating-point numbers.

Equivalent values for the double-precision floating-point quantities are provided by D1MACH, with $e_{\min}$, $e_{\max}$, and t replaced by $E_{\min}$, $E_{\max}$, and T.

A note on 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 Matula (1968).

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

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

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

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

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

$$e'_{\max} = \lfloor e_{\max} \log_{10} b \rfloor .$$

Here, $\lfloor x \rfloor$ denotes the largest integer not exceeding x.

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

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

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

$$t' = \left\lceil t \log_{10} b \right\rceil .$$

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, the actual range may be larger, but a more complicated test would be needed.

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

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

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

Programming using the machine-constant functions

In most cases it is desirable to avoid repeated calls in a single subprogram to the functions described above. The obvious technique is to retrieve the needed values at the outset, but there are cases where substantial overhead may be incurred, even by this technique. One way to eliminate multiple calls is to use a carefully constructed 'first-time' switch.

For example, to retrieve I1MACH(9), the largest integer, on first entry to a subprogram, the following coding can be used:

```
DATA IMAX / 0 /
. . .
IF (IMAX .EQ. 0) IMAX = I1MACH(9)
```

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.

Use of definition redundancy to check installation

Aside from convenience in programming, the redundancy in the definitions of the machine constants allows a particular installation of the PORT library to be checked for consistency. After the library has been compiled, a special checking subprogram is called to verify that the integer and floating-point constants satisfy the following conditions:

(1) $t \leqslant T$

(2) $E_{\max} \geqslant e_{\max}$

(3) $E_{\min} \leqslant e_{\min}$

(4) the largest integer agrees with $a^s - 1$

(5) the floating-point constants agree with those computed from the integer constants

(6) the largest and smallest floating-point values are closed under negation, i.e.: $-(-x) = x$

If a discrepancy is found, a warning message and the values of the quantities involved are printed. Condition (3) may fail on some machines, even though the specifications are correct: in this case the warning massage should be ignored. For 2's complement machines, if $e_{\min}$ has been set too small, condition (6) will fail, since the negative of the smallest number will underflow. Note that some of the integer definitions must be used by the checking routine to determine the output unit for the error messages and the correct formats for printing.

3. The PORT Library

The proof of the porting, one might say, is in the using. Libraries, to be effective, must take into account the motivations of the users. A user is grateful if the programs are easy to use and well documented: protection against errors is particularly appreciated.

To gain user acceptance, PORT provides: (1) simplified calling sequences, (2) careful error-handling, (3) dynamic storage allocation, and (4) brief but complete documentation, all implemented in a portable way. The first three of these four topics are discussed below.

3.1. Calling sequences and modular structure

PORT is structured like an onion. The programs most visible, on the outer layer of the library, are the simplest. The calls to these top-level routines need few parameters and are documented in brief (typically one-page) reference sheets. The top-level routines, in turn, can set default values and call lower level routines containing more parameters. A routine at the second level often is documented and available to the more sophisticated user, who may wish, for example, to influence the details of the step-size monitoring in differential equation solution. Then a second level subprogram may call on a third, perhaps undocumented, 20-parameter, subprogram.

At the innermost level, the picture simplifies again to a more primitive state. PORT includes small subprograms for complex double-precision arithmetic, and for the trigonometric functions that are not ANSI Fortran; also the library probably will incorporate some version of the basic linear algebra modules (inner products, norms, etc.) proposed by Hanson, Krogh, and Lawson (1973), (see Lawson (1975)). Further routines include those for initializing a vector, for moving arrays and/or changing their type, for determining if a vector is (strictly) monotone, for finding the ceiling or floor of a floating-point quantity, and for doing internal sorting. All the subprograms are of course implemented portably and can be called directly by the user, or by other routines in the library.

Calling sequences in PORT are simplified in another sense, by not including parameters for error indication or for scratch storage. The centralized error-handling procedure of the library eliminates the need for error flags, while its dynamic storage allocation capability eliminates scratch storage arrays in the calls to outer level subprograms.

3.2. Centralized error handling

In most libraries, a program which can reach an error state includes in its calling sequence a parameter to indicate, on return from the subprogram, whether an error has occurred. The user is responsible for testing the error flag and taking the appropriate action, but, as we all know, the user frequently assumes that the program "will work this time." A safer, but more extreme approach, is to eliminate the error flag, and, if an error occurs, simply print an error message from within the subprogram and terminate the run.

The picture is really more complicated, because the proper treatment of an error may depend on the degree of severity of the error, the sophistication of the user, and other matters not known to the subprogram in which the error is detected. A recent paper by Goodenough (1975) discusses the general matter of 'exception handling' in a very thorough manner. Various approaches to error-handling have been taken in currently available mathematical libraries:

The severity of the error is taken into account in the EISPACK package which uses the sign of the error flag to distinguish 'path-terminating' (fatal) errors, from those which indicate that some of the computation can be salvaged, even though trouble has occurred [Smith *et al.* (1976)]. Some libraries [IBM (1971), IMSL (1975)] define in greater detail the degree of severity of an error; from 'warning', through 'an error for which the subroutine has taken a default action', to 'dangerous but non-terminating error', and finally 'terminating error'.

For the unsophisticated user, the safest action in all cases is to print a message and stop. The experienced user, on the other hand, usually wants to control the error-handling to some extent, and various techniques for doing so are provided in the different libraries.

The NAG (Mk 2, 1973) library, in the calling sequence to the error routine, provides a parameter, IFAIL, which can be set by the calling program to control the action: if IFAIL is input as 0 ('hard fail'), an error message is printed and execution terminated; if the input value is set to 1 ('soft fail'), the error routine assigns the current error number to IFAIL, now used as the output parameter, and returns to continue execution.

FUNPACK [Cody (1975)] provides the experienced user with a mechanism for accessing error patterns in great detail. Within the library, tables are kept of the frequency of occurrence of each error; the user can monitor a particular error, and allow or suppress the printing of error messages. Continuation or termination of the run, at any point, can also be controlled.

The SANDIA library [Jones and Bailey (1976)] uses a technique which is, to some extent, similar to the one used in PORT and described below. In their library a set of error routines is provided that allow the user to override the default message printing and termination. One routine is used to set the flags controlling the printing and/or termination, and another routine can be used to access the current setting of the flags. The principle error routine, ERRCHK, does the actual error handling as specified by the current flag settings. A fourth routine is provided for one-time-only printing of an error message. The four routines communicate with each other via a COMMON block, and are implemented in a machine-independent version.

Error handling in PORT

In PORT, only two types of error can occur: 'fatal' and 'recoverable', and two types of user are catered to. For the unwary user either type of error causes an error message to be printed and the run to be terminated: in the case of a fatal error, a call is made to a dump routine. (The dump routine itself is a local option: at Bell Labs Murray Hill a symbolic dump is provided, which lists the names of the variables and their values when the dump was called, and prints out the list of active subprograms, [Hall (1975)].) For the user who wishes to recover from an error and to gain control over the error-handling process, a 'recovery mode' is provided. At any point in a run the user can enter the recovery mode, and, while in this mode, can

- determine whether an error has occurred, and, if so, obtain the error number
- print any current error message
- turn off the error state
- leave the recovery mode

Only recoverable errors can be controlled by the user; the fatal errors represent unrecoverable situations or user blunders such as setting an input parameter to an impossible value.

When an error is detected in a PORT subprogram, a call is made to the error-handling routine, SETERR. (Of course a user may also call SETERR in a main program, or user-written subprogram.) The calling sequence is:

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

where

MESSG	=	a Hollerith message
NMESSG	=	the number of characters in MESSG
NERR	=	the error number
IOPT	=	1, to specify that the error is recoverable
	=	2, to specify that the error is fatal

Unless recovery mode is in effect, SETERR prints an error message and terminates execution.

For the casual user of PORT, the possibility of regaining control after an error will probably not be of interest. The message printed out if an error occurs will usually indicate where a change or correction need be made. For the user who does wish to recover from certain errors and continue the computation, there are, as noted above, subprograms permitting this flexibility. Part 2 of this report provides further discussion of the programs in the error-handling package.

3.3. Dynamic storage allocation using a stack

Dynamic storage allocation in programming systems (as opposed to memory management in operating systems) is most often found in list processing applications such as LISP. Fortran, which has no mechanism for recursion, is not a natural setting for dynamic storage management, although some work has been done. Barron (1975) reports the work of Ayers (1963) on implementing a set of stack-handling routines in Fortran. Jensen (1974) describes some routines for dynamically providing portions of a storage pool to an application program. He has developed a 'modular storage management system' which uses data structures in a hierarchy of records, groups of records, and groups of groups, so that the scheme is more structured than is a simple stack. Other libraries of mathematical subprograms do contain sets of programs to implement dynamic storage-handling, but the only library we happen to know, besides PORT, which is completely built around a dynamic storage scheme is STATLIB [Bresford, Relles *et al.* (1969)].

The PORT library has integrated a dynamic storage allocator into the basic library structure. We consider this method for providing scratch space greatly superior to other methods; the historical approach of compiling workspace directly into individual subprograms is clearly inefficient, and the other general method of passing names of scratch arrays puts a considerable naming and dimensioning burden on the user. We have found that use of dynamic storage allocation in PORT leads to more clearly structured programs, cleaner calling sequences, improved memory utilization, and better error detection. The allocator is implemented as a package of simple portable Fortran subprograms which manipulate a dynamic storage stack.

In general, the casual PORT user need not be concerned about the operation, or even the existence of the dynamic storage stack; the fact that the PORT subprograms are using the stack is invisible. However, for strict conformance with the ANSI Standard, and particularly when overlays are being used, a declaration of the stack in the main program should be included, (cf. the discussion of nonstandard usages in Appendix A.)

Below we discuss the capabilities included in PORT's storage-allocation package, and give examples of its use. Appendix B discusses the implementation of the storage stack, and Part 2 of the report contains the subprograms for it.

The stack: allocation and de-allocation

Allocation and de-allocation of space on the stack is carried out through the use of explicit subprogram calls in the subprograms of the PORT library. By the nature of a stack, allocations and de-allocations are carried out on a last-in first-out basis. In order to make the stack invisible to most users of library programs, the package is self-initializing and contains a default stack size which will hold approximately 500 DOUBLE PRECISION data items. If desired, larger amounts of stack space can be reserved for a particular run.

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)
      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, most Fortran environments permit a larger stack to be declared in the main program (see below), without adjusting these other declarations to match.

To provide LOGICAL, INTEGER, REAL and COMPLEX aliases for the stack, the following declarations appear in many PORT subprograms.

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

The dimensions are chosen for the ANSI standard situation with LOGICAL, REAL, and INTEGER variables taking half the space of DOUBLE PRECISION or COMPLEX. If the relative lengths are nonstandard (see Section 2.2), there is one stack-management subprogram that must be modified. (See Part 2 of the report.)

PORT contains two basic subprograms, ISTKGT and ISTKRL, for getting and releasing stack space, respectively. The function for getting stack space is

INTEGER FUNCTION ISTKGT(NITEMS,ITYPE)

where NITEMS is the number of items of type ITYPE to be allocated. The values of ITYPE are as follows:

ITYPE	Item Type
1	LOGICAL
2	INTEGER
3	REAL
4	DOUBLE PRECISION
5	COMPLEX

For example, the statement

I = ISTKGT(N,2)

returns an index I so that the locations

ISTAK(I) , ... , ISTAK(I+N−1)

form the space allocated for N INTEGER items. Similarly, the statement

I = ISTKGT(N,3)

returns an index I so that the locations

RSTAK(I) , ... , RSTAK(I+N−1)

form the space allocated for N REAL items. Further, the statement

I = ISTKGT(N,4)

returns an index I so that the locations

DSTAK(I) , ... , DSTAK(I+N−1)

form the space allocated for N DOUBLE PRECISION items. Space may be obtained for LOGICAL or COMPLEX items in a similar fashion. Note that the space allocated is not initialized to any particular value.

Since no assumption is made about the relative amounts of storage allocated by the Fortran system to the various data types, it is important that allocations not be divided into sub-allocations for data of different types. Instead, ISTKGT should be invoked separately to obtain space for each of the different types being used.

The subroutine for releasing space is

```
SUBROUTINE ISTKRL(K)
```

which simply releases the space obtained by the last K ISTKGT invocations.

As a simple example of the use of these two subprograms, consider a 'little black box' subroutine LBB(A,N) which returns something in a REAL vector A of length N and requires two scratch arrays to do so: an INTEGER array of length 2N and a REAL array of length N. LBB would look roughly as follows:

```
      SUBROUTINE LBB(A,N)
C
      COMMON /CSTAK/DSTAK(500)
C
      DOUBLE PRECISION DSTAK
      INTEGER ISTAK(1000)
      REAL A(1)
      REAL RSTAK(1000)
C
      EQUIVALENCE (DSTAK(1),ISTAK(1))
      EQUIVALENCE (DSTAK(1),RSTAK(1))
C
      II = ISTKGT(2*N,2)
      IR = ISTKGT(N,3)
        .
        .
      { code referring to RSTAK(IR+n) and ISTAK(II+m)
        probably ending with code to store the stuff
        from the real scratch storage into array A }
        .
        .
      CALL ISTKRL(2)
C
      RETURN
      END
```

To avoid messy (and possibly non-standard) subscript calculations, it is often 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 makes programs more readable and easier to code, but often more efficient too. Thus the above can be coded as a 'shell,' LBB, calling on a 'workhorse' subprogram, L1BB, as follows:

```
      SUBROUTINE LBB(A,N)
C
      COMMON /CSTAK/DSTAK(500)
C
      DOUBLE PRECISION DSTAK
      INTEGER ISTAK(1000)
      REAL A(1)
      REAL RSTAK(1000)
C
      EQUIVALENCE (DSTAK(1),ISTAK(1))
      EQUIVALENCE (DSTAK(1),RSTAK(1))
C
      II = ISTKGT(2*N,2)
      IR = ISTKGT(N,3)
C
      CALL L1BB(A,ISTAK(II),RSTAK(IR),N)
C
      CALL ISTKRL(2)
      RETURN
      END
```

Initializing the stack size

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 ISTKIN(NITEMS,ITYPE)
```

where NITEMS is the number of items of type ITYPE set aside for the stack.

For example, to set up a larger stack with 1000 DOUBLE PRECISION items, the following declarations and subroutine call would be put in the main program.

```
      COMMON /CSTAK/DSTAK(1000)
      DOUBLE PRECISION DSTAK
        .
        .
      CALL ISTKIN(1000,4)
```

(Since the library programs are compiled with a stack of 500 DOUBLE PRECISION items, this is a non-standard usage, but one supported by most Fortran environments — see Appendix A.)

Stack status: query and modification

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, but it was felt that this would unnecessarily complicate both implementation and use. For those situations when it is desirable to query how much stack remains, the function

```
INTEGER FUNCTION ISTKQU(ITYPE)
```

can be used. ISTKQU returns the number of items of type ITYPE remaining to be allocated in a *single* invocation of ISTKGT. (As noted in Appendix B, there is a small amount of space overhead associated with each allocation. If the stack is effectively full, ISTKQU will return 0). The statements

```
NLEFT = ISTKQU(3)
INDEX = ISTKGT(NLEFT,3)
```

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

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

```
INTEGER FUNCTION ISTKMD(NITEMS)
```

which will modify the length of the last allocation to NITEMS items and, in a manner similar to ISTKGT, return the index of the first item of that allocation. If the last allocation is truncated, only the first NITEMS items 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 ISTKQU and ISTKMD, 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 = ISTKQU(3)
      I = ISTKGT(NLEFT,3)
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 (IRUNIT,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.
C  CHECK THAT AT LEAST ONE ITEM WAS READ,
C  AND, IF SO, TRUNCATE THE ALLOCATION.
C
 20   IF(NITEMS .EQ. 0) GO TO elswhere
      I = ISTKMD(NITEMS)
C
C  NOW THE ITEMS ARE IN LOCATIONS
C  RSTAK(I) , ... , RSTAK(I+NITEMS-1)
C
       .
       .
```

The function

```
INTEGER FUNCTION ISTKST(N)
```

allows one to obtain certain statistics on the storage allocator. All quantities are measured in terms of INTEGER items. Because there is no fixed relation assumed about the relative sizes of the various data types, the values returned should only be used for observing the status of the stack. The values returned by ISTKST are determined by the argument N as follows:

N	Statistic Returned
1	Number of outstanding allocations
2	Current active length
3	Maximum active length achieved
4	Maximum active length permitted

To determine the exact number of INTEGER items required for the stack, one might include the following statements at the end of the main program.

```
      IUSED = ISTKST(3)
      WRITE(IWUNIT,100) IUSED
  100 FORMAT(1X,13HSTACK USED = ,I6)
```

More detail on the Fortran implementation of the stack is given in Appendix B.

3.4. Mathematical programs in PORT

The programs in the PORT library are grouped into twelve chapter areas: Approximation, Computer Arithmetic, Differential Equations, Linear Algebra and Eigensystems, Mathematical Programming, Optimization, Probability and Statistics, Quadrature, Roots, Special Functions, Transforms, and Utility. Some of the routines, such as the Jenkins-Traub (1972,1975) polynomial root finder, Singleton's (1969) Mixed Radix Fast Fourier Transform, and three programs from EISPACK have been adapted from versions appearing in the open literature. All of these have been revised to fit PORT's portability, error-handling, and dynamic storage requirements. The bulk of the routines have been developed at Bell Labs. These include Blue's (1975) quadrature routine, QUAD, which can integrate 'noisy' integrands or integrands with a singularity, and Schryer's (1975) differential equation solver, ODES, which is built around an efficient and robust extrapolation algorithm. Warner and Eldredge (1976) have provided a program, BURAM, for finding the best uniform rational approximation on a mesh using the differential correction algorithm. This last group of programs are included in the second edition of PORT which is now out. Also included now are programs for finding the roots of a set of nonlinear equations [Blue (1976)], and a random number generator, implemented to be portable and to provide the same random deviates on any computer with at least 16 bits [Gross 1976]. (The latter uses Marsaglia's mixed congruential-Tausworthe shift method [Marsaglia and Ananthanarayanan (1973)].) An extensive spline approximation, interpolation and integration package, has been added to the library, and there are many new utility routines, including sorting and various vector operations.

PORT now contains 546 subprograms. The library is not as large as that sounds: separate single-precision and double-precision versions of the subprograms (when appropriate) are each counted. There are 125 documented programs — one piece of documentation applies to a given single/double-precision pair. Many of the lower level routines which are not documented in the current edition will be written up for future editions. The tape contains some 40,000 lines of Fortran, including comment cards.

4. Summary

The PORT library project has been under way for three years; the library is now installed on various classes of computers and users have found it to provide a solid foundation for program development. The emphasis, throughout the development of the library, has been put on portability, structure, and ease of use. Above, we have surveyed some of the techniques used to achieve these ends. In summary, we make the library portable by using a subset of ANSI Fortran, and by using function calls to obtain machine-dependent quantities. We have structured the library around a centralized error-handling procedure, and we have included dynamic storage allocation, implemented in a portable manner. These approaches, together with simple brief user reference sheets, have helped us achieve user acceptance at the various installation sites.

Acknowledgments

Members of both the Computing Science Research Center and the Computing Technology Center at Bell Laboratories have contributed in substantial ways to the development of the PORT library. The authors wish to express their appreciation for this support. In particular, W. Stanley Brown is to be thanked for his many valuable suggestions on the direction the library should take.

Appendix A

Nonstandard Fortran in PORT

PORT makes two assumptions on its host Fortran environment that are outside of the ANSI standard. These are that there is no runtime subscript range checking, and that variables initialized in DATA statements retain their most recently assigned values. These transgressions and their effect on the library are the subject of this appendix.

The assumption on subscript ranges allows dummy arrays in subroutines to be given a last subscript dimension of 1 under the assumption that a larger value can actually be used. The extension of the assumption to cover arrays in COMMON means that the stack can be initialized in the main program to a size larger than its default size of 500 double-precision locations, even though the library has been compiled using the default size. Similarly, if the relative lengths of different data types are nonstandard (see Section 2.2), the library need not be recompiled to reflect the actual ratios.

The second assumption, that a variable which is initialized by a DATA statement in a subprogram, and then changed within the subprogram, keeps the latest value from one invocation of the subprogram to the next, is used in PORT's error-handling and stack allocation packages. In the error handler, a DATA variable of this type is used to hold the error number of the last error that has occurred, another is used for the current mode (recovery or nonrecovery), and another to store the error message pertaining to the last error, if any. In the implementation of the storage stack, such a DATA variable is used as a flag to specify whether the stack has been intitialized, either by the user, or, in the usual fashion, by the stack subprograms.

These various slippages from standard Fortran usage could cause trouble if overlays are being used. When a user is running a large program in a smaller space using overlays, care should be taken to keep the error-handling and stack-allocation subprograms in the main memory link in order that the values of the DATA variables be retained. This is probably a good idea anyhow, from the standpoint of efficiency, and, for the same reason, the three small machine-constant functions should also be kept resident.

Finally, if a user has had to enlarge the storage stack from its default value, or if overlays are being used, the stack should be declared within the main program and the call to the stack initializing routine made in the main program. This will serve to keep the stack in the main link.

Appendix B

PORT storage stack — implementation notes

Each allocation on PORT's dynamic storage stack consists of four parts: *initial padding, allocated space, final padding* and *control information.* The amount of space allocated to the initial padding is less than the space occupied by one item of the type being allocated. The final padding is less than the space occupied by an integer. The padding simply accounts for the differences in the relative positions of items of different type in the COMMON block CSTAK. The control information takes two integers the first of which contains ITYPE, the type of the allocation. 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 the space reserved for internal bookkeeping, currently 10 integer locations.

In these 10 locations in the stack, information is stored about the number of allocations currently outstanding, the current active length of the stack, the maximum length achieved so far during the run, etc. Also stored here are 5 integers giving the amount of space allocated to each of the different data types. Since these numbers are used solely for computing subscripts, the unit of measurement is arbitrary and may be words, bytes, bits or whatever is convenient. By default, the ANSI standard 'storage unit' is used. For mini-computer Fortran systems which do not allocate storage as prescribed by the Fortran Standard, the subprogram (I0TK00) that initializes these 5 locations should be modified as appropriate. This subprogram and the others in the stack allocation package are given in Part 2 of the report.

Fig. 1 shows a schematic of the dynamic storage stack with two allocations outstanding, the first for 5 integers, and the second for 3 double-precision values. The cross-hatching after the first allocation represents the padding needed to align the double-precision quantities on the proper boundary.

At each call to the allocator, the consistency of the stack and the control information stored in it is checked. If an inconsistency is found, SETERR is called to deliver an appropriate message and terminate the run.

Nonstandard Fortran usages in the implementation of the stack are discussed in Appendix A above.

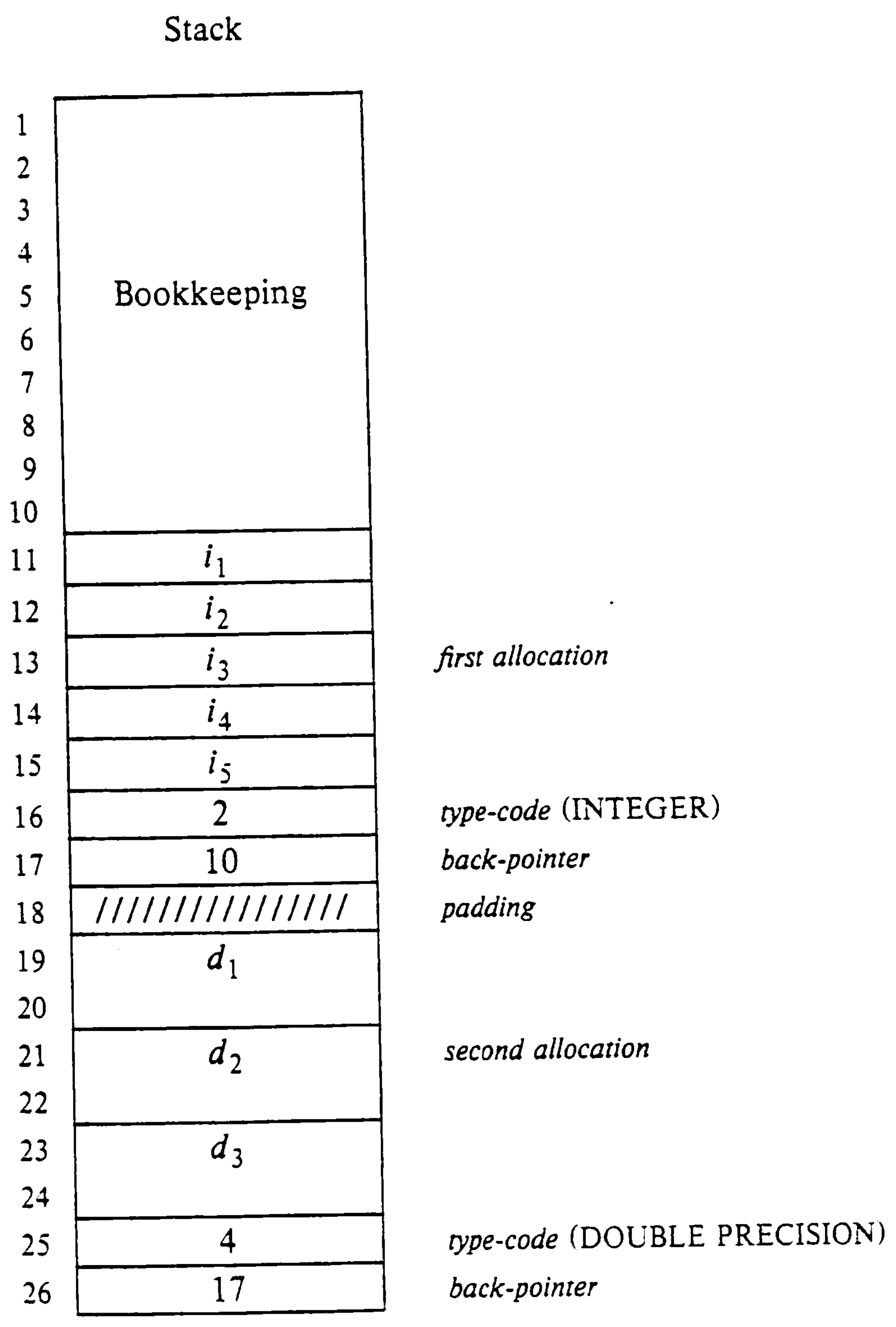
Stack
1
2
3
4
5
6
7
8
9
10
Bookkeeping
11 i_1
12 i_2
13 i_3 *first allocation*
14 i_4
15 i_5
16 2 *type-code* (INTEGER)
17 10 *back-pointer*
18 //////////////// *padding*
19 d_1
20
21 d_2 *second allocation*
22
23 d_3
24
25 4 *type-code* (DOUBLE PRECISION)
26 17 *back-pointer*

Figure 1. Dynamic Storage Stack

References

Aird, T. J., Battiste, E. L., Bosten, N. E., Darilek, H. L., and Gregory, W. C. (1974). Name standardization and value specification for machine dependent constants, *SIGNUM Newsletter* 9, Number 4, 11-14.

Aird, T. J., Battiste, E. L., and Gregory, W. C. (1975). Portability of mathematical software coded in FORTRAN, (private communication).

Ayers, J. A. (1963). Recursive programming in Fortran II, *CACM* 6, 667-668.

Barron, D. W. (1975). *Recursive Techniques in Programming, 2nd Ed.* American Elsevier, N.Y., 42-45.

Battiste, E. L. (1971). The production of mathematical software for a mass audience, in Rice (1971a), 121-130.

Blue, J. L. (1975). Automatic numerical quadrature — DQUAD, Computing Science Technical Report Number 25, Bell Laboratories, Murray Hill, N. J.

Blue, J. L. (1976). Solving systems of nonlinear equations, Computing Science Technical Report Number 50, Bell Laboratories, Murray Hill, N. J.

Boyle, J. M., Cody, W. J., Cowell, W. R., Garbow, B. S., Ikebe, Y., Moler, C. B., and Smith, B. T. (1972). NATS, a collaborative effort to certify and disseminate mathematical software. *Proc. 1972 ACM Annual Conf.*, Vol. II, Assoc. for Computing Mach., New York, 630-635.

Boyle, J. M., and Dritz, K. D. (1974). An automated programming system to facilitate the development of quality mathematical software. *Information Processing 74 (Proc. IFIP 74)*, 3, North-Holland, Amsterdam, 542-546.

Bresford, W. M., Relles, D. A., *et al.* (1969). STATLIB, Bell Laboratories, Holmdel, N. J.

Brown, W. S. (1970). Software portability, in Buxton, J. N., and Randell, B.(Eds.), *Report of the 1969 NATO Conference on Software Engineering Techniques. NATO Science Committee*, 80-84.

Brown, W. S. (1977), A realistic model of floating-point computation, in Rice (1977), pp. -

Cody, W. J. (1975). The FUNPACK package of special function subroutines. *ACM Trans. Math. Software* 1, 13-25.

Dickinson, A. W., Herbert, V. P., Pauls, A. C., and Rosen, E. M. (1971). The development and maintenance of a technical subprogram library, in Rice (1971a), 141-151.

Ford, B. (1976). Machine characteristics and their parameterisation in numerical software, (private communication).

Ford, B., and Sayers, D. (1976). Developing a single numerical algorithms library for different machine ranges, *ACM Trans. Math. Software* 2, 115-131.

Ford, B., and Smith, B. T. (1975). Transportable mathematical software: a substitute for portable mathematical software, IFIP Working Group 2.5 (on Numerical Software), position paper.

Gentlemen, W. M., and Marovich, S. B. (1974). More on algorithms that reveal properties of floating-point arithmetic units, *CACM* 17, 276-277.

Gentleman, W. M., and Traub, J. F. (1968). The Bell Laboratories numerical mathematics program library project, *Proc. 1968 ACM 23rd National Conference,* 485-490.

George, J. E. (1975). Algorithms to reveal the representation of characters, integers, and floating-point numbers, *ACM Trans. Math. Software* 1, 210-216.

Goodenough, J. B. (1975). Exception handling: issues and a proposed notation, *CACM* 18, 683-696.

Gross, A. (1976). Portable random number generation. Internal Bell Laboratories report.

Hague, S. J., and Ford, B. (1976). Portability – prediction, and correction, *Software Practice and Experience* 6, 61-69.

Hall, A. D., (1975). FDS: a Fortran debugging system, Computing Science Technical Report Number 29, Bell Laboratories, Murray Hill, N. J.

Hanson, R. J., Krogh, F. T., and Lawson, C. L. (1973). A proposal for standard linear algebra subprograms, Technical Memorandum 33-660, Jet Propulsion Laboratory, California Institute of Technology.

Hopper, M. J. (1973). *Harwell Subroutine Library, A Catalogue of Subroutines,* Theoretical Physics Division, U.K.A.E.A. Research Group, Atomic Energy Research Establishment, HARWELL, England.

IBM, *IBM System/360 and System/370 Subroutine Library - Mathematics User's Guide,* First Edition, November 1971, IBM Germany .

IFIP Working Group on Numerical Program Libraries, SIGNUM Newsletter: Vol. 7, Number 3, October 1972, pp.10-11; Vol. 9, Number 3 July 1974, pp.3-4; Vol. 9, Number 4, October 1974, pp.3-4; Vol. 10, Number 2/3, November 1975, pp.16,25.

IMSL, International Mathematical and Statistical Libraries, Inc., Library 2, Edition 5, 1975. Sixth Floor, GNB Bldg., 7500 Bellaire Blvd., Houston, Texas 77036.

Jenkins, M. A., and Traub, J. F. (1972). Zeros of a complex polynomial, Algorithm 419, *CACM* 15, 97-99.

Jenkins, M. A., and Traub, J. F. (1975), Zeros of a real polynomial, Algorithm 493, *ACM Trans. Math. Software,* 1, 178-189.

Jensen, P. S. (1974). Storage management of numerical processes, presented at *Software II,* (abstract) 265.

Johnson, O. G. (1971). IMSL's ideas on subroutine library problems, *SIGNUM Newsletter* 6, Number 3, 10-12.

Jones, R. E., and Bailey, C. B. (1976). Brief instructions for using MATHLIB (Version 6.0), Sandia Laboratories, Albuquerque, NM 87115.

Krogh, F. T. (1974). A language to simplify maintenance of software which has many versions, Computing Memorandum No. 360, Jet Propulsion Laboratory, April 18, 1974.

Krogh, F. T., and Singletary, S. A. (1976). Specializer User's Guide, (draft, private communication).

Lawson, C. L. (1975). Current status of the SIGNUM Basic Linear Algebra project, Computing Memorandum No. 378, Jet Propulsion Laboratory, California Institute of Technology.

Malcolm, M. A. (1972). Algorithms to reveal properties of floating-point arithmetic. *CACM* 15, 949-951.

Marsaglia, G., and Ananthanarayanan, K. (1973). Random number generator package - 'Super-Duper,' School of Computer Science, McGill University.

Matula, D. H. (1968). In-and-out conversions, *Comm. ACM* 11, 47-50.

NAG Reference Manual, Mark 2 (1973). NAG Central Office, Oxford University Computing Laboratory, 13 Banbury Road, Oxford, OX2 6NN, England.

Newbery, A. C. R. (1971). The Boeing library and handbook of mathematical routines, in Rice (1971a), 153-169.

Redish, K. A., and Ward, W. (1971). Environment enquiries for numerical analysis, *SIGNUM Newsletter* 6, Number 1, 10-15.

Rice, J. R. (1971a). *Mathematical Software,* Academic Press, N. Y.

Rice, J. R. (1971b). The distribution and sources of mathematical software, in Rice (1971a), 13-25.

Rice, J. R. (Ed.) (1977). (Need to have title for Wisconsin Software Book)

Ryder, B. G. (1974). The PFORT verifier, *Software Practice and Experience* 4, 359-377.

Schryer, N. L. (1975). A user's guide to DODES, a double-precision ordinary differential equation solver, Computing Science Technical Report Number 33, Bell Laboratories, Murray Hill, N. J.

Singleton, R. C. (1969). An algorithm for computing the mixed radix Fast Fourier Transform. *IEEE Trans. on Audio and Electroacoustics,* AU-17, 93-103.

Smith, B. T., Boyle, J. M., Dongerra, J. J., Garbow, B. S., Ikebe, Y., Klema, V. C., and Moler, C. B. (1976). *Matrix Eigensystem Routines — EISPACK Guide, 2nd Edition,* Springer-Verlag, New York.

Software II (1974). Informal proceedings of a conference held at Purdue University, May 29-31.

Waite, W. M. (1970). Building a mobile programming system, *The Computer Journal* 13, 28-31.

Warner, D. D., and Eldredge, B. D. (1976). An implementation of the differential correction algorithm, Computing Science Technical Report Number 48, Bell Laboratories, Murray Hill, N.J.

Wilkes, M. V., Wheeler, D. J., and Gill, S. (1951). *The Preparation of Programs for an Electronic Digital Computer,* Addison-Wesley, Reading, Mass.

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

Clarification of FORTRAN Standards - Initial Progress, *CACM* 12, (1969), 289-294.

Clarification of FORTRAN Standards - Second Report, *CACM* 14, (1971), 628-642.

Program Listings:

Machine Constants

Error Handling

Stack Allocation

PORT Utilities: Description and Listings

Machine constants
Error handling
Stack allocation

Machine-dependent constants

The first package contains three Fortran function subprograms which can be invoked to determine basic machine or operating system dependent constants. Values are provided in commented DATA statements for the Burroughs 5700/6700/7700, the CDC 6000/7000 Series, the Data General Eclipse, DEC PDP 10 (KA and KI processors), the DEC PDP 11, the Harris S220, the Honeywell 6000 Series, the IBM 360/370 Series, the SEL Systems 85/86, the UNIVAC 1100 Series, the XEROX SIGMA 5/7/9; others can be added. When the library is moved to a new environment, only the appropriate DATA statements in these three subprograms need to be activated by removing the C's from column 1.

The three 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.

The functions have a single integer argument indicating the particular constant desired. For example, R1MACH(2) is the largest single-precision floating-point number on the host machine, so the statement

XMAX = R1MACH(2)

sets XMAX to this largest number.

The machine-dependent values provided are defined in the initial comment sections of the listings below. Details of constant specification and usage are described in Part 1 of the report.

```
      INTEGER FUNCTION I1MACH(I)
C
C  I/O UNIT NUMBERS.
C
C    I1MACH( 1) = THE STANDARD INPUT UNIT.
C
C    I1MACH( 2) = THE STANDARD OUTPUT UNIT.
C
C    I1MACH( 3) = THE STANDARD PUNCH UNIT.
C
C    I1MACH( 4) = THE STANDARD ERROR MESSAGE UNIT.
C
C  WORDS.
```

```
C
C   I1MACH( 5) = THE NUMBER OF BITS PER INTEGER STORAGE UNIT.
C
C   I1MACH( 6) = THE NUMBER OF CHARACTERS PER INTEGER STORAGE UNIT.
C
C  INTEGERS.
C
C   ASSUME INTEGERS ARE REPRESENTED IN THE S-DIGIT, BASE-A FORM
C
C              SIGN ( X(S-1)*A**(S-1) + ... + X(1)*A + X(0) )
C
C              WHERE 0 .LE. X(I) .LT. A FOR I=0,...,S-1.
C
C   I1MACH( 7) = A, THE BASE.
C
C   I1MACH( 8) = S, THE NUMBER OF BASE-A DIGITS.
C
C   I1MACH( 9) = A**S - 1, THE LARGEST MAGNITUDE.
C
C  FLOATING-POINT NUMBERS.
C
C   ASSUME FLOATING-POINT NUMBERS ARE REPRESENTED IN THE T-DIGIT,
C   BASE-B FORM
C
C              SIGN (B**E)*( (X(1)/B) + ... + (X(T)/B**T) )
C
C              WHERE 0 .LE. X(I) .LT. B FOR I=1,...,T,
C              0 .LT. X(1), AND EMIN .LE. E .LE. EMAX.
C
C   I1MACH(10) = B, THE BASE.
C
C  SINGLE-PRECISION
C
C   I1MACH(11) = T, THE NUMBER OF BASE-B DIGITS.
C
C   I1MACH(12) = EMIN, THE SMALLEST EXPONENT E.
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 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 BURROUGHS 1700 SYSTEM.
C
C     DATA IMACH( 1) /    7 /
C     DATA IMACH( 2) /    2 /
C     DATA IMACH( 3) /    2 /
C     DATA IMACH( 4) /    2 /
C     DATA IMACH( 5) /   36 /
C     DATA IMACH( 6) /    4 /
C     DATA IMACH( 7) /    2 /
C     DATA IMACH( 8) /   33 /
C     DATA IMACH( 9) / Z1FFFFFFFF /
C     DATA IMACH(10) /    2 /
C     DATA IMACH(11) /   24 /
C     DATA IMACH(12) / -256 /
C     DATA IMACH(13) /  255 /
C     DATA IMACH(14) /   60 /
C     DATA IMACH(15) / -256 /
C     DATA IMACH(16) /  255 /
C
C     MACHINE CONSTANTS FOR THE BURROUGHS 5700 SYSTEM.
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) /   48 /
C     DATA IMACH( 6) /    6 /
C     DATA IMACH( 7) /    2 /
C     DATA IMACH( 8) /   39 /
C     DATA IMACH( 9) / O0007777777777777 /
C     DATA IMACH(10) /    8 /
C     DATA IMACH(11) /   13 /
C     DATA IMACH(12) /  -50 /
C     DATA IMACH(13) /   76 /
C     DATA IMACH(14) /   26 /
C     DATA IMACH(15) /  -50 /
C     DATA IMACH(16) /   76 /
C
C     MACHINE CONSTANTS FOR THE BURROUGHS 6700/7700 SYSTEMS.
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) /   48 /
C     DATA IMACH( 6) /    6 /
C     DATA IMACH( 7) /    2 /
C     DATA IMACH( 8) /   39 /
C     DATA IMACH( 9) / 00007777777777777 /
C     DATA IMACH(10) /    8 /
C     DATA IMACH(11) /   13 /
C     DATA IMACH(12) /  -50 /
C     DATA IMACH(13) /   76 /
C     DATA IMACH(14) /   26 /
C     DATA IMACH(15) / -32754 /
C     DATA IMACH(16) /  32780 /
C
C     MACHINE CONSTANTS FOR THE CDC 6000/7000 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) / 00007777777777777777B /
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) /  -927 /
C     DATA IMACH(16) /  1070 /
C
C     MACHINE CONSTANTS FOR THE CRAY 1
C
C     DATA IMACH( 1) /   100 /
C     DATA IMACH( 2) /   101 /
C     DATA IMACH( 3) /   102 /
C     DATA IMACH( 4) /   101 /
C     DATA IMACH( 5) /    64 /
C     DATA IMACH( 6) /     8 /
C     DATA IMACH( 7) /     2 /
C     DATA IMACH( 8) /    63 /
C     DATA IMACH( 9) /  777777777777777777777B /
C     DATA IMACH(10) /     2 /
C     DATA IMACH(11) /    47 /
C     DATA IMACH(12) / -8192 /
C     DATA IMACH(13) /  8190 /
C     DATA IMACH(14) /    95 /
C     DATA IMACH(15) / -8192 /
```

```
C     DATA IMACH(16) /  8190 /
C
C     MACHINE CONSTANTS FOR THE DATA GENERAL ECLIPSE S/200
C
C     DATA IMACH( 1) /    11 /
C     DATA IMACH( 2) /    12 /
C     DATA IMACH( 3) /     8 /
C     DATA IMACH( 4) /    10 /
C     DATA IMACH( 5) /    16 /
C     DATA IMACH( 6) /     2 /
C     DATA IMACH( 7) /     2 /
C     DATA IMACH( 8) /    15 /
C     DATA IMACH( 9) /32767 /
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 HARRIS SLASH 6 AND SLASH 7
C
C     DATA IMACH( 1) /       5 /
C     DATA IMACH( 2) /       6 /
C     DATA IMACH( 3) /       0 /
C     DATA IMACH( 4) /       6 /
C     DATA IMACH( 5) /      24 /
C     DATA IMACH( 6) /       3 /
C     DATA IMACH( 7) /       2 /
C     DATA IMACH( 8) /      23 /
C     DATA IMACH( 9) / 8388607 /
C     DATA IMACH(10) /       2 /
C     DATA IMACH(11) /      23 /
C     DATA IMACH(12) /    -127 /
C     DATA IMACH(13) /     127 /
C     DATA IMACH(14) /      38 /
C     DATA IMACH(15) /    -127 /
C     DATA IMACH(16) /     127 /
C
C     MACHINE CONSTANTS FOR THE HONEYWELL 600/6000 SERIES.
C
C     DATA IMACH( 1) /     5 /
C     DATA IMACH( 2) /     6 /
C     DATA IMACH( 3) /    43 /
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) / O377777777777 /
C     DATA IMACH(10) /    2 /
C     DATA IMACH(11) /   27 /
C     DATA IMACH(12) / -127 /
C     DATA IMACH(13) /  127 /
C     DATA IMACH(14) /   63 /
C     DATA IMACH(15) / -127 /
C     DATA IMACH(16) /  127 /
C
C     MACHINE CONSTANTS FOR THE IBM 360/370 SERIES,
C     THE XEROX SIGMA 5/7/9 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 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) / -101 /
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 PDP-11 FORTRAN'S SUPPORTING
C     32-BIT INTEGER ARITHMETIC.
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 PDP-11 FORTRAN'S SUPPORTING
C     16-BIT INTEGER ARITHMETIC.
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) /    16 /
C     DATA IMACH( 6) /     2 /
C     DATA IMACH( 7) /     2 /
C     DATA IMACH( 8) /    15 /
C     DATA IMACH( 9) / 32767 /
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     NOTE THAT THE PUNCH UNIT, I1MACH(3), HAS BEEN SET TO 7
C     WHICH IS APPROPRIATE FOR THE UNIVAC-FOR SYSTEM.
C     IF YOU HAVE THE UNIVAC-FTN SYSTEM, SET IT TO 1.
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) / O377777777777 /
C     DATA IMACH(10) /    2 /
C     DATA IMACH(11) /   27 /
C     DATA IMACH(12) / -128 /
C     DATA IMACH(13) /  127 /
C     DATA IMACH(14) /   60 /
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
      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 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 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
      REAL RMACH(5)
C
      EQUIVALENCE (RMACH(1),SMALL(1))
      EQUIVALENCE (RMACH(2),LARGE(1))
      EQUIVALENCE (RMACH(3),RIGHT(1))
      EQUIVALENCE (RMACH(4),DIVER(1))
      EQUIVALENCE (RMACH(5),LOG10(1))
C
C     MACHINE CONSTANTS FOR THE BURROUGHS 1700 SYSTEM.
C
C     DATA RMACH(1) / Z400800000 /
C     DATA RMACH(2) / Z5FFFFFFFF /
C     DATA RMACH(3) / Z4E9800000 /
C     DATA RMACH(4) / Z4EA800000 /
C     DATA RMACH(5) / Z500E730E8 /
C
C     MACHINE CONSTANTS FOR THE BURROUGHS 5700/6700/7700 SYSTEMS.
C
C     DATA RMACH(1) / O1771000000000000 /
C     DATA RMACH(2) / O0777777777777777 /
C     DATA RMACH(3) / O1311000000000000 /
C     DATA RMACH(4) / O1301000000000000 /
```

```
C     DATA RMACH(5) / 01157163034761675 /
C
C     MACHINE CONSTANTS FOR THE CDC 6000/7000 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 CRAY 1
C
C     DATA RMACH(1) / 200004000000000000000B /
C     DATA RMACH(2) / 577767777777777777776B /
C     DATA RMACH(3) / 377224000000000000000B /
C     DATA RMACH(4) / 377234000000000000000B /
C     DATA RMACH(5) / 377774642023241175720B /
C
C     MACHINE CONSTANTS FOR THE DATA GENERAL ECLIPSE S/200
C
C     NOTE - IT MAY BE APPROPRIATE TO INCLUDE THE FOLLOWING CARD -
C     STATIC RMACH(5)
C
C     DATA SMALL/20K,0/,LARGE/77777K,177777K/
C     DATA RIGHT/35420K,0/,DIVER/36020K,0/
C     DATA LOG10/40423K,42023K/
C
C     MACHINE CONSTANTS FOR THE HARRIS SLASH 6 AND SLASH 7
C
C     DATA SMALL(1),SMALL(2) / '20000000, '00000201 /
C     DATA LARGE(1),LARGE(2) / '37777777, '00000177 /
C     DATA RIGHT(1),RIGHT(2) / '20000000, '00000352 /
C     DATA DIVER(1),DIVER(2) / '20000000, '00000353 /
C     DATA LOG10(1),LOG10(2) / '23210115, '00000377 /
C
C     MACHINE CONSTANTS FOR THE HONEYWELL 600/6000 SERIES.
C
C     DATA RMACH(1) / O402400000000 /
C     DATA RMACH(2) / O376777777777 /
C     DATA RMACH(3) / O714400000000 /
C     DATA RMACH(4) / O716400000000 /
C     DATA RMACH(5) / O776464202324 /
C
C     MACHINE CONSTANTS FOR THE IBM 360/370 SERIES,
C     THE XEROX SIGMA 5/7/9 AND THE SEL SYSTEMS 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 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 PDP-11 FORTRAN'S SUPPORTING
C     32-BIT INTEGERS (EXPRESSED IN INTEGER AND OCTAL).
C
C     DATA SMALL(1) /    8388608 /
C     DATA LARGE(1) / 2147483647 /
C     DATA RIGHT(1) /  880803840 /
C     DATA DIVER(1) /  889192448 /
C     DATA LOG10(1) / 1067065499 /
C
C     DATA RMACH(1) / 000040000000 /
C     DATA RMACH(2) / 017777777777 /
C     DATA RMACH(3) / 006440000000 /
C     DATA RMACH(4) / 006500000000 /
C     DATA RMACH(5) / 007746420233 /
C
C     MACHINE CONSTANTS FOR PDP-11 FORTRAN'S SUPPORTING
C     16-BIT INTEGERS  (EXPRESSED IN INTEGER AND OCTAL).
C
C     DATA SMALL(1),SMALL(2) /   128,      0 /
C     DATA LARGE(1),LARGE(2) / 32767,     -1 /
C     DATA RIGHT(1),RIGHT(2) / 13440,      0 /
C     DATA DIVER(1),DIVER(2) / 13568,      0 /
C     DATA LOG10(1),LOG10(2) / 16282,   8347 /
C
C     DATA SMALL(1),SMALL(2) / 0000200, 0000000 /
C     DATA LARGE(1),LARGE(2) / 0077777, 0177777 /
C     DATA RIGHT(1),RIGHT(2) / 0032200, 0000000 /
C     DATA DIVER(1),DIVER(2) / 0032400, 0000000 /
C     DATA LOG10(1),LOG10(2) / 0037632, 0020233 /
C
C     MACHINE CONSTANTS FOR THE UNIVAC 1100 SERIES.
C
C     DATA RMACH(1) / 0000400000000 /
C     DATA RMACH(2) / 0377777777777 /
C     DATA RMACH(3) / 0146400000000 /
C     DATA RMACH(4) / 0147400000000 /
C     DATA RMACH(5) / 0177464202324 /
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



      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
C  D1MACH( 5) = LOG10(B)
C
C  TO ALTER THIS FUNCTION FOR A PARTICULAR ENVIRONMENT,
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 WHICH HAS IN SOME CASES
C  REQUIRED THE USE OF EQUIVALENT INTEGER ARRAYS.
C
      INTEGER SMALL(4)
      INTEGER LARGE(4)
      INTEGER RIGHT(4)
      INTEGER DIVER(4)
      INTEGER LOG10(4)
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 BURROUGHS 1700 SYSTEM.
C
C     DATA SMALL(1) / ZC00800000 /
C     DATA SMALL(2) / Z000000000 /
C
```

```
C     DATA LARGE(1) / ZDFFFFFFFF /
C     DATA LARGE(2) / ZFFFFFFFFF /
C
C     DATA RIGHT(1) / ZCC5800000 /
C     DATA RIGHT(2) / Z000000000 /
C
C     DATA DIVER(1) / ZCC6800000 /
C     DATA DIVER(2) / Z000000000 /
C
C     DATA LOG10(1) / ZD00E730E7 /
C     DATA LOG10(2) / ZC77800DC0 /
C
C     MACHINE CONSTANTS FOR THE BURROUGHS 5700 SYSTEM.
C
C     DATA SMALL(1) / O1771000000000000 /
C     DATA SMALL(2) / O0000000000000000 /
C
C     DATA LARGE(1) / O0777777777777777 /
C     DATA LARGE(2) / O0007777777777777 /
C
C     DATA RIGHT(1) / O1461000000000000 /
C     DATA RIGHT(2) / O0000000000000000 /
C
C     DATA DIVER(1) / O1451000000000000 /
C     DATA DIVER(2) / O0000000000000000 /
C
C     DATA LOG10(1) / O1157163034761674 /
C     DATA LOG10(2) / O0006677466732724 /
C
C     MACHINE CONSTANTS FOR THE BURROUGHS 6700/7700 SYSTEMS.
C
C     DATA SMALL(1) / O1771000000000000 /
C     DATA SMALL(2) / O7770000000000000 /
C
C     DATA LARGE(1) / O0777777777777777 /
C     DATA LARGE(2) / O7777777777777777 /
C
C     DATA RIGHT(1) / O1461000000000000 /
C     DATA RIGHT(2) / O0000000000000000 /
C
C     DATA DIVER(1) / O1451000000000000 /
C     DATA DIVER(2) / O0000000000000000 /
C
C     DATA LOG10(1) / O1157163034761674 /
C     DATA LOG10(2) / O0006677466732724 /
C
C     MACHINE CONSTANTS FOR THE CDC 6000/7000 SERIES.
C
C     DATA SMALL(1) / 00604000000000000000B /
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) / 15000000000000000000B /
C
C     DATA DIVER(1) / 15614000000000000000B /
C     DATA DIVER(2) / 15010000000000000000B /
C
C     DATA LOG10(1) / 17164642023241175717B /
C     DATA LOG10(2) / 16367571421742254654B /
C
C     MACHINE CONSTANTS FOR THE CRAY 1
C
C     DATA SMALL(1) / 200004000000000000000B /
C     DATA SMALL(2) / 00000000000000000000B /
C
C     DATA LARGE(1) / 577767777777777777777B /
C     DATA LARGE(2) / 000007777777777777776B /
C
C     DATA RIGHT(1) / 376424000000000000000B /
C     DATA RIGHT(2) / 000000000000000000000B /
C
C     DATA DIVER(1) / 376434000000000000000B /
C     DATA DIVER(2) / 000000000000000000000B /
C
C     DATA LOG10(1) / 377774642023241175717B /
C     DATA LOG10(2) / 000007571421742254654B /
C
C     MACHINE CONSTANTS FOR THE DATA GENERAL ECLIPSE S/200
C
C     NOTE - IT MAY BE APPROPRIATE TO INCLUDE THE FOLLOWING CARD -
C     STATIC DMACH(5)
C
C     DATA SMALL/20K,3*0/,LARGE/77777K,3*177777K/
C     DATA RIGHT/31420K,3*0/,DIVER/32020K,3*0/
C     DATA LOG10/40423K,42023K,50237K,74776K/
C
C     MACHINE CONSTANTS FOR THE HARRIS SLASH 6 AND SLASH 7
C
C     DATA SMALL(1),SMALL(2) / '20000000, '00000201 /
C     DATA LARGE(1),LARGE(2) / '37777777, '37777577 /
C     DATA RIGHT(1),RIGHT(2) / '20000000, '00000333 /
C     DATA DIVER(1),DIVER(2) / '20000000, '00000334 /
C     DATA LOG10(1),LOG10(2) / '23210115, '10237777 /
C
C     MACHINE CONSTANTS FOR THE HONEYWELL 600/6000 SERIES.
C
C     DATA SMALL(1),SMALL(2) / O402400000000, O000000000000 /
```

```
C     DATA LARGE(1),LARGE(2) / 0376777777777, 0777777777777 /
C     DATA RIGHT(1),RIGHT(2) / 0604400000000, 0000000000000 /
C     DATA DIVER(1),DIVER(2) / 0606400000000, 0000000000000 /
C     DATA LOG10(1),LOG10(2) / 0776464202324, 0117571775714 /
C
C     MACHINE CONSTANTS FOR THE IBM 360/370 SERIES,
C     THE XEROX SIGMA 5/7/9 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 PDP-10 (KA PROCESSOR).
C
C     DATA SMALL(1),SMALL(2) / "033400000000, "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 PDP-11 FORTRAN'S SUPPORTING
C     32-BIT INTEGERS (EXPRESSED IN INTEGER AND OCTAL).
C
C     DATA SMALL(1),SMALL(2) /    8388608,           0 /
C     DATA LARGE(1),LARGE(2) / 2147483647,          -1 /
C     DATA RIGHT(1),RIGHT(2) /  612368384,           0 /
C     DATA DIVER(1),DIVER(2) /  620756992,           0 /
C     DATA LOG10(1),LOG10(2) / 1067065498, -2063872008 /
C
C     DATA SMALL(1),SMALL(2) / O00040000000, O00000000000 /
C     DATA LARGE(1),LARGE(2) / O17777777777, O37777777777 /
C     DATA RIGHT(1),RIGHT(2) / O04440000000, O00000000000 /
C     DATA DIVER(1),DIVER(2) / O04500000000, O00000000000 /
C     DATA LOG10(1),LOG10(2) / O07746420232, O20476747770 /
C
C     MACHINE CONSTANTS FOR PDP-11 FORTRAN'S SUPPORTING
C     16-BIT INTEGERS (EXPRESSED IN INTEGER AND OCTAL).
C
C     DATA SMALL(1),SMALL(2) /    128,      0 /
C     DATA SMALL(3),SMALL(4) /      0,      0 /
```

```
C
C     DATA LARGE(1),LARGE(2) /  32767,      -1 /
C     DATA LARGE(3),LARGE(4) /     -1,      -1 /
C
C     DATA RIGHT(1),RIGHT(2) /   9344,       0 /
C     DATA RIGHT(3),RIGHT(4) /      0,       0 /
C
C     DATA DIVER(1),DIVER(2) /   9472,       0 /
C     DATA DIVER(3),DIVER(4) /      0,       0 /
C
C     DATA LOG10(1),LOG10(2) /  16282,    8346 /
C     DATA LOG10(3),LOG10(4) / -31493,  -12296 /
C
C     DATA SMALL(1),SMALL(2) / 0000200, 0000000 /
C     DATA SMALL(3),SMALL(4) / 0000000, 0000000 /
C
C     DATA LARGE(1),LARGE(2) / 0077777, 0177777 /
C     DATA LARGE(3),LARGE(4) / 0177777, 0177777 /
C
C     DATA RIGHT(1),RIGHT(2) / 0022200, 0000000 /
C     DATA RIGHT(3),RIGHT(4) / 0000000, 0000000 /
C
C     DATA DIVER(1),DIVER(2) / 0022400, 0000000 /
C     DATA DIVER(3),DIVER(4) / 0000000, 0000000 /
C
C     DATA LOG10(1),LOG10(2) / 0037632, 0020232 /
C     DATA LOG10(3),LOG10(4) / 0102373, 0147770 /
C
C     MACHINE CONSTANTS FOR THE UNIVAC 1100 SERIES.
C
C     DATA SMALL(1),SMALL(2) / 0000040000000, 0000000000000 /
C     DATA LARGE(1),LARGE(2) / 0377777777777, 0777777777777 /
C     DATA RIGHT(1),RIGHT(2) / 0170540000000, 0000000000000 /
C     DATA DIVER(1),DIVER(2) / 0170640000000, 0000000000000 /
C     DATA LOG10(1),LOG10(2) / 0177746420232, 0411757177572 /
C
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
```

Automatic error-handling

The second package provides a basic mechanism for dealing with the occurrence of errors.

In the PORT library, calls to the general subroutines in the library do not include, in their calling sequences flags for error indication. Instead, when a called subroutine detects an error it calls the principal error-handling routine, SETERR.

The package allows for two types of error, 'fatal', and 'recoverable,' and a parameter in the call to SETERR must be set to specify the type. Fatal errors cause an error message to be printed, the run terminated, and a call made to a dump routine. (A dummy dump routine, FDUMP, is provided here.) For recoverable errors, unless the user has specifically requested to enter the recovery mode, similar events occur, an error message is printed and the run terminated. Thus the process is fail safe for unwary users.

When the recovery mode is in effect, any call to SETERR given within a subprogram which has detected a recoverable error, has the effect only of storing the fact that that an error has occurred; the run is not terminated. The user, upon return from the subprogram is responsible for testing for the occurrence of an error. If an error has occurred, the user must turn off the error state, because additional errors might arise and the occurrence of a recoverable error while in the error state constitutes an unrecoverable error, terminating the run.

Finally, since a called subprogram, say SUBA, may, in turn, call a lower-level subprogram containing recoverable errors, SUBA must check for the occurrence of errors in the lower-level routine and reinterpret them in the context of SUBA, which the user knows about. This means that SUBA must enter the recovery mode, (saving the mode previously in effect), make the call to the lower-level subprogram, then, upon return from the lower-level routine, check for errors, and, before returning to the user, restore the previous recovery mode.

An error which has caused an invocation of SETERR has an associated number, message, and type (fatal or recoverable), and the effect of the error depends on whether the recovery mode is in effect or not. The various capabilities offered in the subprograms of the package are summarized below:

To signal that an error has occurred:

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

where MESSG and NMESSG are, respectively a Hollerith message and the number of characters in the message, and NERR is the error number. IOPT is used to specify the type of error: IOPT = 1 for a recoverable error, and = 2 for a fatal error.

To save the recovery (or nonrecovery) mode currently in effect, and enter a new one:

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

which saves the current mode in IRNEW and sets the new one to IRNEW .

To avoid having multiple errors outstanding, it is a fatal error to call SETERR or ENTSRC if the error state is on, meaning that an error has occurred but not been recovered from.

To restore the recovery (or nonrecovery) mode which was previously saved in IROLD:

```
CALL RETSRC(IROLD)
```

RETSRC not only restores the previous mode, but also acts as a 'safety' exit gate: Since multiple errors are illegal, RETSRC checks out the situation and allows return to the calling program only if (1) an error is not outstanding, or (2) the restored mode is recovery, so that the calling program is responsible for error checking.

To test if an error has occurred, and if its number was, say, 4, a statement such as the following is used:

```
IF ( NERROR(NERR) .EQ. 4 ) GO TO 50
```

The value of the function, NERROR, and the value of the argument, NERR, are both set to the current value of the error number by NERROR. (The double assignment may be useful and comes free since Fortran prohibits functions with no arguments.) If the error number is non-zero, it means that an error has occurred and that corrective action must be taken.

To turn off the error state:

```
CALL ERROFF
```

In summary the user subprograms are:

SETERR	-	turns on the error state, and saves a message and an error number
ENTSRC	-	at entry, sets recovery (or nonrecovery) mode, provided no error state exists
RETSRC	-	before returning, checks error situation and, if ok, restores prior recovery (or nonrecovery) mode
NERROR	-	returns the error number
ERROFF	-	turns off the error state
EPRINT	-	prints the error message

These, in turn, call on the lower-level subprograms:

E9RINT -	stores or prints error message, depending on switch setting
S88FMT -	sets up FORMAT array for printing
I8SAVE -	returns error number or recovery (or nonrecovery) mode, depending on one switch, and resets or does not reset the corresponding value depending on another
FDUMP -	a dummy routine to be replaced, if possible, by a locally written symbolic dump routine

```
      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  IF THE ERROR IS FATAL, PRINT, DUMP, AND STOP
C
```

```
 40   IF (IOPT.EQ.2) GO TO 50
C
C  HERE THE ERROR IS RECOVERABLE
C
C  IF THE RECOVERY MODE IS IN EFFECT, OK, JUST RETURN
C
      IF (I8SAVE(2,0,.FALSE.).EQ.1) RETURN
C
C  OTHERWISE PRINT AND STOP
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
```