ON A PROPOSED FLOATING-POINT STANDARD

W. Kahan[1]
University of California, Berkeley

J. Palmer
INTEL Corporation, Aloha, Oregon

October 1, 1979

A standard for binary floating-point arithmetic is being proposed and there is a very real possibility that it will be adopted by many manufacturers and implemented on a wide range of computers. This development matters to all of us concerned with numerical software. One of the principal motivations for the standard is to distribute more evenly the burden of portability between hardware and software. At present, any program intended to be portable must be designed for a mythical computer that enjoys no capability not supported by every computer on which the program will be run. That mythical computer is so much grubbier than almost any real computer that a portable program will frequently be denigrated as "suboptimal" and then supplanted by another program supposedly "optimal" for the real computer in question but often inferior in critical respects like reliability. A standard — almost any reasonable standard — will surely improve the situation. A standard environment for numerical programs will promote fair comparisons and sharing of numerical codes, thereby lowering costs and prices. Furthermore, we have chosen repeatedly to enrich that environment in order that applications programs be simpler and more reliable. Thus will the onus of portability be shared among hardware manufacturers and software producers.

The standard that we will describe has been under development for some time and there have been many contributors. Its most recent history began with the publication, after a year's study, of a paper [1] in 1977, indicating INTEL's committment to a subset of the standard. This paper was the basis for the initial discussions of the IEEE microprocessor floating-point standards committee that began in late 1977. Within a few months, a much expanded proposal was brought to the committee by H. Stone, J. Coonen and one of us (W. Kahan); that proposal's latest drafts have been prepared by J. Coonen [3]. At the same time we and several other people at INTEL have been scrutinizing the proposal, attempting to balance user benefit and implementation cost. Meanwhile the IEEE "microprocessor" committee has been augmented by representatives from minicomputer and mainframe companies because they foresee a likely migration towards any such standard.

The ideas embodied in this standard have been brewing for years, and it appears that their time has finally come. The momentum behind the standard is growing and even now may be irreversible. Two chips are on sale, the AMD 9512 and the INTEL 8232, which almost conform to a subset of the proposed standard, and INTEL has indicated that its 8087, scheduled to appear in about a year, will support the standard fully. And other companies are known to be implementing it. What then is this standard?

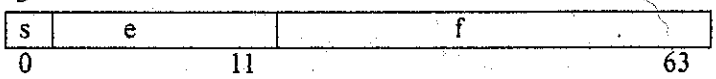
The standard is not a set of radically new and ingenious concepts but a carefully considered collection and integration of many people's good ideas that have been thought of and occasionally implemented in isolation. In this case the whole is indeed greater than the sum of its parts, for when they are all integrated the various features support each other synergetically in a system whose increased safety, capability and simplicity transcend its slightly increased implementation cost. Perhaps "simplicity" is the wrong word, because the draft standard [2] looks very complicated; but so are the blueprints for a car's automatic transmission much more complicated than what a driver has to know to use it. The document [3] describing the implementation also appears complicated because it handles all exceptions and covers several permissible levels of implementation. If a single implementation level is selected and all exceptions are delegated to software, the hardware is simple to build.

The standard specifies two basic formats, single and double precision, that are intended to be fully supported by accurate functions and conversions and to be used in the ordinary way as a program's named variables. (An optional third format, called extended, unpacked or temporary, is recommended for purposes to be discussed later.) The two formats are shown below:

single (32 bit): X_s has the form

[1]Supported in part by ONR Grant N00014–76–C–0013

double (64 bit): X_D has the form



The values of the representations are, for normalized numbers,

$X_S = (-1)^s 2^{e-127}(1.f)$, provided $0 < e < 255$

$X_D = (-1)^s 2^{e-1023}(1.f)$, provided $0 < e < 2047$.

The 1 to the left of the binary point is implicit and is not stored. The special value zero lies in the midst of the "denormalized" numbers all distinguished by $e = 0$ and interpreted thus:

$X_S = (-1)^s 2^{-126}(0.f)$, so $f = 0$ when $X_S = 0$

$X_D = (-1)^s 2^{-1022}(0.f)$, so $f = 0$ when $X_D = 0$.

These numbers are used to implement gradual underflow [4], a technique for handling underflows that will be discussed later. Exponents $e = 255$ in single or $e = 2047$ in double designate a reserved operand, a Not-a-Number (NaN), that is used primarily for diagnostic information, except that $e = 255$ and $f = 0$ (or $e = 2047$ and $f = 0$) represent $\pm\infty$; the use of ± 0 and $\pm\infty$ will be explained later.

The third format, extended, is optional but strongly recommended. It takes advantage of an unpacked format which is expected to be used by hardware or firmware to implement the basic formats described above. Whereas the single and double precision variables are to be used in the ordinary way, extended precision, intended for their support, is to be used initially to implement elementary functions (log, exp, cos, ...) and binary-decimal conversion efficiently, later to be available for use for anonymous variables and for a limited number of named variables. The extended named variables are useful to hold accumulations during the execution of a loop after which the result will normally be assigned to one of the basic formats. The extended format is somewhat machine dependent but must, if supporting double,

1. have at least 15 bits of exponent
2. have at least 64 bits of precision
3. have an explicit first bit
4. include ± 0, $\pm\infty$ and the full set of Not-a-Numbers.

In a system where only the single format is implemented, the extended would have at least 11 bits of exponent and 32 bits of precision.

If compilers used the extended format for subexpressions then over/underflows, which mostly occur during intermediate calculations, would almost disappear. Also, the extra precision would ensure that, for many calculations, rounding errors be really negligible. The idea here is that the easiest errors and exceptions to analyze and handle are those that do not occur. Provided the compiler would allow some named extended variables, mixed mode arithmetic would be encouraged. Thus, the use by default of extended precision would usually ensure that the only significant rounding error or possiblity of over/underflow would occur at the conclusion of a calculation when the calculated value in extended must be rounded and assigned to a basic format.

But we speak rather vaguely of rounding; how do we propose to do it?

The standard requires two kinds of control on rounding: its direction and precision. We mean to provide a very simple model:

> *The rounded result will be one of the representable neighbors of the infinitely precise true result, depending on the specified direction.*

The standard requires one rounding mode called "round to nearest even" (RN):

> *The rounded result is the nearest representable number to the true result; if there are two then it is the one with the least significant bit zero.*

In this mode, not only are the statistics impeccable [5] but the error and error bound are unimprovable. There have been arguments, some spurious, that round to nearest odd (RO) is preferable. There is of course little difference, but the only one visible to a user is that RN produces integers more often than does RO. Consider:

Theorem: *Suppose m and n are small integers in the sense that 3m, 5m, 3n, 5n and mn can all be stored in a binary floating-point word. Then*

$$RN(3*RN((mn)/(3n))) = m$$
$$RN(5*RN((mn)/(5n))) = m$$

and the conclusion is **not** *true with RO in place of RN.*

In addition to RN, the standard recommends the provision of

RZ: round toward zero
RP: round toward $+\infty$
RM: round toward $-\infty$

The recommendation is justified because the mechanism to provide RN is exactly that needed for the others, namely a "sticky" or indicator [6] bit, and the additional modes yield capabilities far more valuable than their trivial cost:

RZ: allows one to do the usual Fortran things with integers

RP and RM: permit implementation of both rigorous and tight bounding and Interval Arithmetic [7]. In fact, Interval Arithmetic can be implemented only a factor of about 5 slower than ordinary arithmetic. (Our implementation of Interval Arithmetic [8] is safe, fast and closed: division by intervals containing zero is allowed.)

We hope that higher level languages will soon provide facilities for controlling (testing and setting) the rounding mode for expressions and blocks, and that they will add INTERVAL as a new two-word data type with the obvious mixed mode arithmetic allowed.

The other aspect of rounding control, namely precision control, is simply a way of enforcing that the precision of results be determined not by the ostensible precision of operands but rather by the results' intended use. If the destination is single or double, then the result must be correctly rounded to single or double precision respectively; if the destination is an extended variable then precision control allows us to choose whether single, double or full (extended) precision results will be computed. Thus, a computation may be controlled so that all calculations are done to single precision, as is currently required by some languages, but the greater exponent range of extended is still used for intermediates. Although the intended default is extended precision, we hope languages will provide precision control capability as well as rounding direction control. We hope too that languages will be more flexible in allowing mixed precision arithmetic, and in treating the arguments of subroutines as numbers rather than words of predetermined widths.

In addition to rounding and precision control, the standard provides for "infinity" control. Since the system must cope somehow with division by zero and overflow, we have provided the two infinity symbols to allow a computation to proceed as far as possible. Infinity control must be provided because there are two types of infinity closure: projective and affine.

The projective closure is what you get when the real numbers are thought to lie on a circle with infinity at the top. For complex numbers, the Riemann sphere is the projective closure. There is a single point at infinity, (the signs of zero and infinity are ignored) and infinity obeys no order relationships. This mode is generally used for rational arithmetic, real or complex, such as continued fractions, and thus its main effect is to allow division by zero without risk of undetected anomalies. The projective is the default mode.

Another mode, the affine mode, exists mainly to fix up overflow. Here the signs of infinity and zero are relevant. This mode is not perfect and must be used with care. One must analyze his program to make sure the signs of zero and infinity are not misinformation inherited from rounding errors.

Obviously the system must treat the infinities in a slightly different manner in the two modes and thus the language must allow the user to specify which mode he wants. (In Interval Arithmetic the signs of zero and infinity have a different use. They signal whether an interval with endpoints zero or infinity is open or closed there [8].) Examples using the two modes will be given later.

The standard also allows automatic handling of underflow using gradual underflow. This is implemented using denormalized numbers in single and double precision and unnormalized numbers in extended. These numbers, while retaining as much information as possible, carry a warning ("not normalized") that will not disappear unless the information lost by underflow is approximately the same as that lost to rounding error. This mechanism is also not perfect, but in comparison to the usual "underflow to zero" (UZ), gradual underflow

1. works better when they both work
2. occasionally works when UZ does not
3. usually provides a warning when it does not work (with UZ all information is lost with no warning except possibly an underflow indicator that, if present, is usually ignored.)

Thus, the system provides slightly more functionality and much more safety for only a small additional implementation cost, and without obliging programmers to abandon their bad habits.

From the error analyst's point of view, over/underflow is a violation of the simple rule

(*) (computed result) = (true result)$\times$(1 $\pm$ small quantity bounded in advance)

valid for each elementary arithmetic operation: add, subtract, multiply, divide, square root. Hence, over/underflow must always set a flag, even if it does not trap. But if denormalized numbers are used, the error analyst has two more options. One, used by default, is to regard nonzero numbers whose inherited error may exceed that allowed by the rule (*) above as marked with a scarlet letter which will not fade unless the information lost to underflow turns out to have been information that would have been lost to roundoff anyway. Consequently, attempts to greatly magnify denormalized numbers, by multiplying them by huge factors or dividing them by tiny divisors are treated as invalid. This mechanism is appropriate for most "forward" error analyses, especially where an unwanted zero or infinite result would call attention to itself but an ordinary number without the scarlet letter would not. Another option is to think of denormalized numbers in connection with an error bound

(computed result) = (true result)×(1 ± small quantity bounded in advance) ± (another tiny quantity)

e.g. in single precision, with RN rounding,

(computed result) = (true result)×(1 ± 2^{-24}) ± 2^{-150}.

This interpretation is appropriate for most "backward" error analyses; usually the term 2^{-150} will be seen to disappear amidst the rounding errors. This interpretation is realized by allowing denormalized numbers to participate in all multiplications and divisions as if they had first been normalized.

A simple example will illustrate the usefulness of gradual underflow. Suppose the values A, B, C, D, E, F are regarded as exact and we calculate

$$X := (A + B\times C) / (D + E\times F).$$

If an overflow occurs the calculated result X will be infinite, zero or NaN and that fact will be obvious.

If products underflowed to zero and if D or A were small, comparable to the underflow threshold, X could easily be off by a factor of, say, four, but otherwise unexceptional. Furthermore, this error could be enormous compared with the damage that roundoff could cause. We don't do this.

Using gradual underflow ensures that the uncertainty in the result is virtually the same as roundoff uncertainty unless a warning (besides underflow) is given by an invalid operation, a division by zero or an unnormalized or zero result X.

Thus, in the absence of additional warnings gradual underflow makes the threat of underflow comparable to that of roundoff.

Another feature that is provided by the standard is a set of entities called Not-a-Numbers that may be used in many ways, the principal one being the provision of retrospective diagnosis. For instance, one can defer judgement on whether an error (such as 0/0 or using unitialized data) is significant until the program has completed, confident that he will find encoded in a Not-a-Number some debugging information such as a pointer to the first occurrence of the error.

The last exception specified by the standard is a maskable "trap on exact result". This gives one the ability to use, as conveniently as one uses floating point arithmetic, the precision of the extended format for exact computations (accounting, extended integers). In addition it provides another debugging aid. And finally, it allows certain fast but unstable algorithms to be used safely since on detecting a dangerous rounding error one can switch to a slower, more stable procedure.

The ability to handle exceptions in the high level language is a major consideration that must be given more thought than it has heretofore evidently received.

The standard provides by default for the fix-ups described above, but since all these may be in the software or since others may be desired, the language must provide for exception handling. An example of a different fix-up is:

> if the system, instead of using gradual underflow, produces an encoding of the correct result and then traps, one may wish to create a heap of these numbers for later use.

The major features of the standard have been discussed and we will now briefly outline some of its applications and advantages. One of its advantages is that it allows portability at two levels.

At the higher level, we will have portability of "programs" expressed and documented in natural language. This can only be the case if the standard is so tight that when we translate into computer language, we will all obtain essentially the same program and results. This allows us to take components of programs as ideas and transfer them to other applications confident that the assumed computational environments are the same.

The lower level of portability concerns programs written in high level languages. The standard preserves portability of codes already in existence to within an edit and recompile because the functionality of the standard equals or exceeds the capability assumed by existing portable software. Thus, existing portable software, run on a standard system, will very likely either run better if the program was correct, or yield a diagnostic if the old one was wrong.

In addition to portability, the standard provides "program robustness": codes written by programmers who are not numerical analysts have a much better chance of "working" on a standard system than on a typical system. (A working program either yields the correct answer or an explanation of why it can not.) A good example is solving a quadratic equation.

$$A*X*X - 2*B*X + C = 0$$

Almost everyone writes:

```
ROOT1 := (B + SQRT(B*B - A*C))/A
ROOT2 := (B - SQRT(B*B - A*C))/A
```

which is exceedingly vulnerable to roundoff on the typical system; but on a system with extended precision for intermediates, not only does the rounding error problem in these expressions recede substantially, but over/underflow occurs only if the correct result cannot be represented.

In addition to portability and robustness, the system admits to a simple model. This is important not only for ease of explanation and use, but also for automatic analysis of rounding errors. With this simple model, a symbolic manipulator like MACSYMA, and a set of numerical routines, it should be possible to produce a system that allows a user to simply describe his problem and the system will analyze it and choose the correct routine to solve it. Such a system would make use of extended precision, inexact result trap, diagnostic Not-a-Numbers, gradual underflow, infinity fix-ups and Interval Arithmetic to detect and diagnose incorrect numerical results and then to choose alternative procedures to recalculate those results more accurately.

Below we give examples of programs that use various features of the system and illustrate its advantages.

0. Normalizing a vector: $v_i = x_i / \|x\|$.

```
Extended variable d (optional)
d := 0
for i=1 to n do d := d + x_i*x_i
d := √d
for i=1 to n do v_i := x_i/d
```

If extended variables are availiable this program will always yield a valid answer given valid data. Without extended precision but relying on gradual underflow, the computed result will be as reliable as roundoff allows as long as *underflow* is the only exception generated. This is true even though all x_i^2 may be denormalized and yet produce a normalized sum. In that case the uncertainty is only twice the uncertainty attributed to roundoff alone.

1. Newton's divided difference formula. Consider a polynomial

$$f(x) = \sum_{j=0}^{n} a_j \prod_{k=j+1}^{n} (x - b_k)$$

and its derivative f'(x):

```
Extended variables f, f', y (optional)
f' := 0; f := a_0
for j=1 to n do begin
   y := x-b_j; f' := y*f' + f; f := y*f + a_j
   end j
New x := x - f/f'     ... for Newton's iteration to solve f(x) = 0.
```

Denormalized response to over/underflow upon multiplication avoids unnecessary loss of precision; invalid multiplication or division, if it occurs, warns of unavoidable loss of precision to underflow.

2. Scalar product. $r = b - \sum_{j=1}^{n} a_j x_j = b - \mathbf{a}^T\mathbf{x}$.

```
Extended variable sum (optional)
sum := b
for j=1 to n do sum := sum - a_j*x_j
r := sum
```

Denormalized underflow ensures that r is about as correct as roundoff allows despite underflow except possibly if *both* underflow occurred and neither r nor b is a normalized nonzero number. The extended variable sum, if available, would both preclude intermediate over/underflow and suppress roundoff to an extent that would make the uncertainty in r nearly independent of n until n is very big, because the most significant rounding error would usually occur when sum is rounded into r.

3. Continued fraction f and its derivative f':

$$f(x) = a_n + \cfrac{b_n}{x + a_{n-1} + \cfrac{b_{n-1}}{x + a_{n-2} + \cdots + \cfrac{b_1}{x + a_0}}}$$

Extended variables f, f', r, s (optional)
... On division by zero deliver ∞ except 0/0 = NaN.
.. Either Affine or Projective mode is allowed, and so is infinite x.

```
f := f' := 0
for j=1 to n do
   begin
   r := (x + a[j-1]) + f;  f := b[j]/r
   if (r is finite)
      then r := -f*(1+f')/r
      else r := b[j]*(1+s)/b[j-1]
   s := f';  f' := r    ... s = previous f'
   end j
f := a[n] + f;  if x is infinite then f' := -b[n]/x^2
```

This program is intended to give results f and f' as correct as rounding errors will allow despite the possible intrusion of over/underflow and/or infinities. In particular, at poles where f is infinite, f' will be infinite too with the correct sign. (If the test "r is finite" were taken for granted to simplify the program, then f' would be calculated as NaN whenever division by zero occurred in the loop.)

4. Rayleigh quotient. Given a symmetric matrix A and an approximate eigenpair (λ, x), so $Ax \simeq \lambda x$, compute an improvement $\sigma = x^T(A-\lambda)x / x^Tx$ and a residual $\rho = ||(A-\lambda-\sigma)x|| / ||x||$ which bounds the error in the eigenvalue extimate $\lambda+\sigma$.

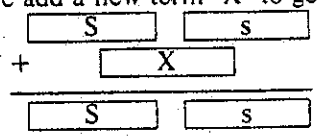
Extended variables β, ξ, δ, ϕ (optional)

```
β := ξ := δ := 0
for i=1 to n do
   begin
   φ := -λ*x[i]
   ξ := x[i]*x[i] + ξ
   for j=1 to n do φ := φ + a[ij]*x[j]
   β := β + φ*x[i]
   δ := δ + φ*φ
   end i
σ := β/ξ
ρ := sqrt(δ/ξ - (β/ξ)^2)
```

We have calculated

$$c = x^Tx = ||x||^2$$
$$f_i = ((A-\lambda)x)_i$$
$$b = x^T(A-\lambda)x$$
$$d = ||(A-\lambda)x||^2$$

5. Extended accumulation: $S = \Sigma_j x_j$ via Dekker's method [9] for extending the available precision. To an accumulation S+s we add a new term X to get an updated accumulation S+s:



Applications: summation of infinite series; numerical quadrature; integration of ordinary differential equations; running averages and standard deviations.

Assume: All variables and intermediate results are to one level of precision (all *single* or all *double*), with neither extended precision nor range. Underflows are denormalized. Rounding is to nearest.

Program:
```
if |S| < |X| then swap(X,S)
T := (X+s) + S
t := ((S−T)+X) + s
S := T+t
s := (T−S) + t
```

Only if X itself underflowed could underflow contaminate the sum, and then only if the final sum S is not a normalized number. **Do not** use this program with other kinds of rounding, nor with any other radix but **binary** unless T and t have extended precision.

6. Eigenvalue problem. Let T be the symmetric tridiagonal matrix

$$\begin{vmatrix} a_1 & b_2 & & & \\ b_2 & a_2 & b_3 & & \\ & \cdot & \cdot & \cdot & \\ & & \cdot & \cdot & \cdot \\ & & & b_n & a_n \end{vmatrix}$$

where $b_1 = 0$, but every other $b_i \neq 0$. Given any number p, determine

k = number of T's eigenvalues less than p.

The following procedure was first introduced in the mid 1950's by Boris Davison, a physicist interested in transport theory, and is used to find a few eigenvalues when n is very large. Normally T is given in terms of two arrays, one of a_i's and the other of b_i^2's. In the latter $b_1^2 = 0$ but every other $b_j^2 \neq 0$ since otherwise T would split into independent parts whose eigenvalues could be located separately faster than jointly.

> Extended variable d (optional, to avoid the nuisance of over/underflow; otherwise normalize denormalized divisors.)
> ... On division by zero deliver infinity (0/0 cannot occur). Default Projective mode is all right but Affine is better because it avoids an unnecessary setting of the *invalid operation* flag when "d ⩾ 0" is executed with infinite d. However to avoid d = −0 in the Affine mode we need the statement ...
>
> ```
> if p=0 then p := −|p|
> d := 1;  k := 0     ... k counts the number of times d < 0.
> for j=1 to n do
>    begin
>    d := (a_j−p) − b_j²/d;  if not(d⩾0) then k := k+1
>    end j
> ```

Note: If d=0 on one pass through the loop, then d = +0 so that on the next pass d = −∞ is computed correctly.

If d is not an extended variable, replacing b_j^2 by $(b_j/d)*b_j$ will diminish the nuisance caused by over/underflow, though at some cost in speed.

7. Preconditioning a quadratic equation. $A*X^2-2*B*X+C = 0$. The usual formula for the roots, $X = (B \pm \sqrt{(B^2-AC)})/A$, suffers grievous harm from roundoff whenever the two roots are either too nearly equal or too disparate in magnitude. One way to cope is with the aid of double precision, by whose aid the discriminant B^2-AC may be calculated precisely. But an equally useful accomplishment is within reach of a mere extended precision short of double precision. Here is a program which exploits the relation

$$-Discriminant = \det\begin{bmatrix} A & B \\ B & C \end{bmatrix} = \det\begin{bmatrix} 1 & 0 \\ -s & 1 \end{bmatrix}\begin{bmatrix} A & B \\ B & C \end{bmatrix}\begin{bmatrix} 1 & -s \\ 0 & 1 \end{bmatrix} \quad \text{for } \textit{every } s$$

$$= \det\begin{bmatrix} A & b \\ b & c \end{bmatrix} \quad \textit{where } b = B - sA \text{ and } c = (C - sB) - sb.$$

The program chooses s in a way calculated to exploit cancellation without roundoff error and hence without significant loss of significance.

```
Procedure Quadratic (A, B, C, root1, root2, realpart, imaginarypart):
Extended variables d, all subexpressions
Copydata: b := B; c := C
preconditionloop:
        s := (b/A rounded to 8 significant bits for single data;
              rounded to 11 significant bits for double data)
        if s is finite and s ≠ 0 then
           begin
           Reset INEXACT flag
           b1 := b − s*A; c1 := (c−s*b) − s*b1
           if not INEXACT then begin c := c1; b := b1; go to preconditionloop end
           end s≠0
d := b*b − A*c    ... to extended precision and range, so unnecessary over/underflow is avoided.
if d < 0 then begin realpart := B/A; imaginarypart := √−d/A; return
d := √d; if B < 0 then d := −d
d := d + B; root1 := C/d; root2 := d/A; return
end Quadratic
```

Each root is correct to nearly data-precision, unless it lies out of range, despite that intermediate results might over/underflow if extended range were not available.

Similar notions lead to schemes to precondition very ill-conditioned systems of linear equations in cases where backward error-analysis explains but does not excuse more conventional methods.

8. Over/underflow without heaps for products and quotients. Calculate:

$$r = \prod_{i=1}^{m}(a_i+b_i) \Big/ \prod_{j=1}^{n}(c_j+d_j)$$

exploiting exponent wrap-around by $\pm B$ whenever over/underflow is trapped, i.e.

$$\text{Overflowed}(X*Y) = (X*Y)/2^B,$$
$$\text{Underflowed}(X*Y) = (X*Y)*2^B,$$

in the absence of Extended format. The standard specifies B = 192 or 1536 for Single and Double respectively.

```
        r := 1; K := 0    ... count over/underflows
count: begin
        Upon OVERFLOW do K := K+1
        Upon UNDERFLOW do K := K−1
        for j=1 to n do r := r*(cj+dj)
        K := −K;  r := 1/r
        for i=1 to m do r := r*(ai+bi)
        ... now (desired r) = (stored r)*2^(B*K)
        if r = 0 or r is not finite then K := 0
        if |K| = 1 then begin L := K; K := 0; r := r*2^(L*B/2)*2^(L*B/2) end
        end count    ... revert to previous mode of response to over/underflow
        if K ≠ 0 then   ... r lies out of range
```

This kind of code provides a large part of the capability otherwise requiring a heap.

CONCLUSION

The standard has been described briefly. One area that is also specified but not discussed is BCD conversion. We also intend that in the future the elementary transcendental functions will be specified in the standard. And, ultimately, we hope a decimal version of this standard may be promulgated.

Floating-point arithmetic, as most are aware, has inherent pitfalls that are so varied and subtle that almost everyone would prefer not to think about them. Unfortunately that cannot be — some of us must think about them, including language people; but with a well designed and widely implemented standard, the number of those who must think about pitfalls and the required depth of their thought can be minimized. Thus, more people will be able to devote their full intellectual energy to their own problems.

We live in an imperfect world and this standard is no exception; absolute safety, if attainable, would have cost more that its worth. This standard was designed to yield, for a slightly increased implementation cost, a considerably safer system with greater capability (debugging aids, deferred judgement) and simpler explanation than typical floating-point systems.

BIBLIOGRAPHY

[1] Palmer, J. (1977) "The INTEL Standard for Floating-Point Arithmetic," Proc. COMPSAC, 107-112.

[2] Coonen, J., W. Kahan, J. Palmer, T. Pittman and D. Stevenson (1979), "A Proposed Standard for Binary Floating Point Arithmetic," This issue, pages xx-yy.

[3] Coonen, J. (1979), "Specifications for a Proposed Standard for Floating-Point Arithmetic," Draft submitted to IEEE Microprocessor Floating-Point Standards Committee, August 26.

[4] Kahan, W. (1966), "7094-II System Support for Numerical Analysis," SHARE Secretarial Distribution SSD-159, item C4537.

[5] Brent, R. (1973), "On the Precision Attainable with Various Floating-Point Number Systems,", IEEE Trans. Computers, Vol. C-22, No. 6, 601-607.

[6] Yohe, J. (1973), "Roundings in Floating-Point Arithmetic," IEEE Trans. Computers, Vol. C-22, No.6, 577-586.

[7] Moore, R.E. (1966), **Interval Analysis**, Englewood Cliffs, N.J.: Prentice-Hall.

[8] Kahan, W. (1968), "A More Complete Interval Arithmetic," Lecture Notes for a course at University of Michigan, June 17-21.

[9] Dekker, T.J. (1971), "A Floating-Point Technique for Extending the Available Precision," Numerische Mathematik, Vol. 18, 224-242.