IEEE 754 Error Handling and Programming Languages

Nick Maclaren

March 2000

IEEE 754-1985 introduced order to a particularly chaotic area and, in general, it has been very successful, though experts disagree about which of its properties are merits. Hindsight shows several areas where some unfortunate decisions were taken, but few of the problems were predicted at the time.

Many of the objections made to IEEE 754 are on the grounds of efficiency. There are quite a lot of features that are rarely important, but are expensive to implement; in modern fixed cycle-count logic, this can add an extra cycle per operation. This document does **not** address such matters, and is solely about the aspects that affect the robustness of code and error detection.

Exception and error handling is one area where the design has had most undesirable consequences. The main problem with this aspect is **not** so much that IEEE 754 got it wrong, but that it has been interpreted in harmful and unexpected ways, and used to justify poor and even incorrect language design and implementation. It is not clear whether anyone predicted this – I did not and have not spoken to anyone who did.

Furthermore, this describes programming as done by ordinary programmers, especially those who are concerned with performance, and does not address the problem of whether they are using the best approach. For example, it ignores almost all aspects of numerical accuracy analysis and concentrates on whether major errors will be detected.

[The term IEEE 754 will be used here, though others may prefer IEC 559 or even IEEE 854; the differences between these do not affect this document significantly.]

Traditional Exception Handling

Traditional exception handling was based on signalling; that is, an exception raised a signal which was trapped by the run-time system, which issued a diagnostic for each failure and stopped after a threshold had been reached. There was a religious war over whether underflow was an error, which has still not resolved itself, even though one side has won. The other anomalous exception was integer overflow, including conversions, which was sometimes trapped and sometimes ignored (see Appendix B.)

Experience with this model is that it enabled even average programmers to locate most numeric errors that were severe enough to trigger overflow or divide by zero. Furthermore, relatively few programs would perform numerically invalid operations with apparently correct output and no error indication at all. No diagnostics did not mean that the output was meaningful, of course.

1960s and 1970s experience with signalling on underflow was that underflow often (perhaps 30% of the time) hid serious errors, which usually caused a later overflow or division by zero. However, the people who claimed that underflow was almost always harmless have won out, and few systems support its trapping any more. Even NAG (Numerical

Algorithms Group) code no longer protects itself against underflow being trapped, though it used to.

Signalling has been almost entirely replaced by the model of not trapping errors, but by setting a flag and returning an indicative value (i.e. an infinity or a quiet NaN). As an example of how far this has gone, the Hitachi S-3600 used the traditional model and neither overflow nor division by zero could be masked off. Out of an active programmer population of perhaps 20–30 in 1998, two users complained that their code **relied** on such errors being ignored during the early stages of an iterative process; they did not know what they were assuming, in terms of the Fortran standard, of course.

Trapping

IEEE 754 describes an optional trapping mode that can be enabled and disabled locally for each exception, with the implication is that the user can request this for program debugging and similar purposes. IEEE 754 also defines signalling NaNs, but they are often (erroneously) regarded as being useful solely for detecting uninitialised data, and are rarely supported by programming languages.

The change that finally killed the use of trapping was multi-threaded and out-of-order execution. The conflict between precise exception handling and wide-ranging optimisation became too severe for any compromise to work. This was predicted before 1980, and it has been one of the main reasons that vendors have converted to the exceptional value and exception flag model of IEEE 754.

The main other reasons for the lack of trapping are poor design and implementation of the hardware, compilers, language run-time systems and operating systems, and have little to do with IEEE 754. It is fair to place much of the blame onto Intel, X3J11 and SC22/WG15 (i.e. C), which together were responsible for standardising some severely broken behaviour. Unfortunately, their dominance is such that there is little chance of changing direction now.

IEEE 754 specifies trapping as an optional feature, required to be off by default, which has been interpreted by C99 and Java to mean that IEEE 754 deprecates trapping as a method of diagnosing errors. That being so, they have chosen to support neither trapping nor signalling NaNs at all, even when the hardware and operating system would permit them to do so. In practice, enabling traps often causes the language run-time system to crash or the program to behave in an unpredictable fashion.

Types of Trapping

The most ridiculous aspect is that none of the problems apply to implementing trap-diagnose-and-terminate (as in LIA-1), which was the mode that 1960s and 1970s experience showed was both essential and adequate for locating most cases of numeric failure. Yet that mode has been thrown out together with the technically tricky trap-fixup-and-resume, except in the almost unimplemented LIA-1.

The authors of IEEE 754 probably assumed that readers would be familiar with trapping, and its benefits, and therefore skimped on its description. Similarly, they probably assumed that all implementations would support trap-diagnose-and-terminate as an option, anyway. However, most modern readers have never used a system that supported trapping, and many have misread IEEE 754 to imply that even this is deprecated.

Technically, the problems of even trap-fixup-and-resume are soluble without unreasonable loss of efficiency, in several ways, but all efficient solutions have to be at a higher level than that of individual floating-point operations. The language and operating system design aspects are extremely tricky, and are understood by fewer and fewer people as the "Old Guard" retire, move areas or die off. There seems to be no likelihood of any improvement, especially in C-related languages.

Exceptional Values

The IEEE 754 model of setting a flag and returning an infinity or a quiet *NaN* assumes that the user tests the status frequently (or at least appropriately.) Diagnosing what the original problem was requires the user to check all results for exceptional values, which in turn assumes that they are percolated through **all** operations, so that erroneous data can be flagged. Given these assumptions, everything should work, but unfortunately they are not very realistic.

An obvious preliminary question is "What is a *NaN*, anyway?", which affects which percolation rules are appropriate, and which cases are errors; this is discussed in Appendix A. This document assumes that it is a special value which is known to be numerical but has an indeterminate result, usually due to an uninitialised datum, mathematically erroneous calculation or one where all accuracy has been lost.

Percolation

Most code will be written without explicit *NaN* tests, and IEEE 754 states that it should percolate the *NaN* value unless the programmer takes special action to clear the state. There are problems with this in several constructions, but let us consider just a few examples.

The first problem concerns conditionals. There are some languages that have expressions like "`IF a THEN b ELSE c`", which evaluate to "unknown" if **either** the predicate cannot be evaluated **or** the chosen expression is "unknown", and this feature could fairly easily be added to any purely functional language. But, for the vast majority of languages, IEEE 754 requires most relations to evaluate to false if either operand is a *NaN*. This causes serious trouble with constructions like:

```
IF (A(I) .GT. 0.0) THEN
    A(I) = B*A(I)
ELSE
    A(I) = 0.0
FI
```

or:

```
a[i] = (a[i] > 0.0 ? b*a[i] : 0.0);
```

The above code has the effect of 'losing' a *NaN* value. Obviously, the intent of IEEE 754 is that the above should be coded only if using the limiting value is always safe, and otherwise an explicit test for a *NaN* should be used. Again, experience from the 1970s in (statistical) languages with the above feature showed that this is almost impossible to write code to fail safe, even when it is attempted by an expert. I did an experiment on myself attempting to use the above construction safely (and I am a careful programmer), and found that it mattered for only about 20% of my conditionals but I made an error in about 20% of those cases.

The above code will raise the invalid exception, which is something, but an even nastier trap occurs in constructions like:

```
IF (A(I) .NE. 0.0) A(I) = SIGN(1.0,A(I))
```

or:

```
a[i] = (a[i] == 0.0 ? 0.0 : copysign(1.0,a[i]));
```

If `SIGN` is implemented as IEEE 754 `copysign`, which is both reasonable and likely (and required in C), this has the effect of 'losing' a quiet *NaN* value **without** raising any exception. In fact, IEEE 754 permits even a **signalling** *NaN* to be quietly lost by `copysign`.

This also applies to maximum and minimum. The definition "$max(1.0, NaN) = NaN$" is correct when a *NaN* is a missing value and what is wanted is the maximum non-missing value of a vector (as in one expression mode in many statistical packages) but is mathematically incorrect when it is an error state (as generally in IEEE 754).

Until Java and C99, few languages included *NaN*s, and therefore their handling was unpredictable. Those two did include them, which might have been good, but both actually **require** the above and many other common constructions to lose the *NaN* state without raising an exception (see Appendix B.) Those constructions are so common and so important that the percolated *NaN* values cannot be relied on, at least in code written by ordinary users.

Exception Flags

IEEE 754 describes a set of 'sticky' exception flags, which can be set by any dubious operation and can only be cleared by user action. This method can work reasonably well, provided that the language supports it properly, and users understand its constraints.

The ICL 1900 used a numeric error flag, with the property that returning from a function with the error flag set would cause an interrupt; this worked quite well. Unfortunately, most modern language systems that diagnose IEEE exception flags tend to do so only at the end of the program; this action is implied (but not explicitly required) by IEEE 754. Most users can be taught to use binary chop to locate a problem in their code, provided that it is repeatable. However, almost all express distaste at this, and ask how they can get a traceback at the point of failure, as in LIA-1.

There is a far more serious problem, which is why most run-time systems (especially C) do not diagnose such flags, at least by default. Almost all numerical code (including libraries) sets such flags spuriously, and relies on them not being inspected; C99 actually **encourages** such behaviour (see Appendix B.) The flags cannot be relied upon even for languages and implementations that get this correct, because most serious programs rely on at least one library written in C (e.g. MPI.)

Unfortunately, this is not simply a matter of negligence. Some basic operations, such as complex multiplication, division and absolute value are extremely hard to implement, when both efficiency and error handling are important. Almost all implementations sacrifice reliable error handling for efficiency, which is understandable given the marketing importance of benchmarks.

A reasonable compromise is to ensure that the overflow and divide flags are reliable, but to ignore the underflow and inexact ones. Unfortunately, while this compromise is well-known to numerical analysts (as are its failings), it is not clearly indicated by any standard or language specification. So most designs regard all flags as being of the same importance, and support them all equally unreliably.

Again, older languages rarely included support for IEEE 754 exception flags. C99 has added support, but got it spectacularly wrong (see Appendix B) – so much so that it is arguable whether Java does worse or better, by not including the flags in the language.

Eliminating Errors

The prime example of this is Java, which claims to have eliminated the problem of integer overflow by defining the value delivered when it occurs (i.e. wrapping in twos' complement); converting a NaN to an integer is defined to quietly deliver 0, for example! The fact that this is neither mathematically consistent nor what the user is likely to have expected means that what was a numeric error in Fortran or C has now become a logic error. The problem has been moved, not eliminated.

C99 has often taken this approach; for example, `atan2(0,0)` returns $-\pi$, -0, $+0$, or $+\pi$, depending on the signs of the zeroes, and `pow(`$\pm inf$`,0)` and `pow(`NaN`,0)` (i.e. x^0) return 1. This would not matter, but they are also **forbidden** from flagging an error in any way (which **includes** setting even the inexact flag); Java has the same specification in both cases. Another is that C99 `fmin(`x`,`NaN`)` and `fmax(`x`,`NaN`)` are defined to return x, which is the correct handling for missing values, but is wrong for error indicators; here Java has **not** made the same mistake.

Another notorious area is mixed floating-point and integer arithmetic, which is rather glossed over by IEEE 754, and made much worse by modern hardware architectures and C (see Appendix B.) If we assume that integers have no equivalent of a NaN, then any operation that goes via an integer (or equivalent) may well turn an erroneous value into a wrong answer. IEEE 754 does require that an impossible conversion will raise the invalid exception, but that does not help with errors that occur in the integer arithmetic.

C99 was originally drafted to forbid integer representations from having a value with NaN properties, but its final form permits this for signed integers. It is extremely unlikely that such an implementation will appear, but at least it is not actually forbidden for signed integers, though it is for unsigned ones.

Problems like the above mean that there are several places in which the implementation is forbidden from flagging mathematically undefined operations, and others in which NaN

values can be quietly converted back into apparently valid ones. Java is perhaps worse than C99, but both are full of such pitfalls.

All of this may appear independent of IEEE 754, but it is not. The authors have often quoted IEEE 754 (especially `copysign`) or private communications with Professor Kahan as justification for this approach. I cannot say whether such reports are correct.

Consistency

There is no doubt that IEEE 754 has increased the consistency of results between systems, but it is less clear that this has been a good thing. Traditionally, one of the standard ways that a numerically dubious program was detected was when different arithmetics, or different optimisation levels, gave different answers. Getting the same answer did not demonstrate accuracy, but getting different ones usually demonstrated inaccuracy.

Nowadays, most users develop and 'debug' their programs on IEEE 754 systems with negligible optimisation. On the other hand, most supercomputers do NOT use the strict IEEE 754 model by default, though many hide this very deeply in their documentation. For example, none of the Cray, DEC Alpha, SGI Origin or Hitachi SR2201 do, and I have not checked up on any of the others. This is usually in subtleties like not supporting denormalised numbers, or in various forms of error handling.

But, as all numerical analysts know, optimisation is as effective as arithmetic variations at causing numerical instability to show up as different answers. This is particularly common on parallel code (whether vector, shared memory or distributed), where the order of accumulation of sums and products may vary. Again, the users do not see this effect while developing and testing their code, but do as soon as they put it on a supercomputer.

A good 25% of Cambridge High Performance Computing Facility programmers have reported bugs in the system or compiler that were quite clearly due to numerical instability, and most of them claimed that it could not be the case because their program gave the same answers on many different workstations (which all used IEEE 754 and little optimisation). These were not just the naive users, but included some quite experienced ones.

It is a great pity that IEEE 754 did not include at least an optional probabilistic rounding mode. While this does not provide reliable error estimates (as Professor Kahan points out), it has two great advantages over fixed rounding schemes. One is that it can deliver a higher order (and not just a constant factor) of accuracy when accumulating some types of sum, and the other is that it forces the user to face the fact that floating-point arithmetic is not exact.

In 1975, a good 90% of programmers experienced in using floating-point were at least aware that such problems existed; today, the figure is at most 25%, and at most 5% in those under 30. Virtually everyone under 30 that I have spoken to who has encountered the problem has believed that it is **solely** a problem with optimisation-related bugs in the code, compiler or hardware. Almost none are aware that the problem can be serious on a correctly compiled, correct program on systems with the same floating-point representation. There seems little chance of improvement, especially given the influence of Java.

Composite Operations and Higher Intermediate Precision

From the point of view of accuracy and error handling, these are two facets of the same thing. The most common composite operation is, of course, fused multiplication and addition, which is widely available and can speed up linear algebra significantly. However, the advantages of such things for exponentiation with integer powers, complex arithmetic and so on are often forgotten.

With integer powering, it is an accuracy issue, especially with expressions like $(1/3)^N$; because of the pattern of the bits, this can have an accuracy of little more than $log_2(N)$. Interestingly, this is one of the calculations where probabilistic rounding does rather better, at least on average, though not as much better as might be thought if one were to naïvely assume independence of errors.

Avoiding spurious overflows in complex multiplication is extremely tricky, and the obvious code almost does so, so most vendors ignore the problem. But a trivial increase in exponent range for the intermediate calculations completely eliminates them without any loss of efficiency. Complex division is trickier, but roughly a factor of two increase in the exponent range would eliminate spurious errors in that, too.

The problem implementors have with this is the choice between using a higher precision format for calculations, which is often unavailable or inefficient anyway, and breaking the IEEE 754 rules. They usually choose to follow the standard, and generate spurious errors, for which the "solution" is often to suppress error detection!

Complex Infinities, NaNs and Zeroes

The main problem here is that adding the infinities to the real line is a mathematical closure operation of sorts, but the comparable closure for complex numbers leads to a single infinity. This conflicts very badly with the naïve specifications of complex numbers as real and imaginary parts, though it would fit better with a polar representation.

Unfortunately, that **does** mean "specification" and not "implementation", too. Fortran, C and most languages derived from them permit a program to read and write the real and imaginary parts as real numbers, thus forcing the specification to consider **all** pairs of IEEE real numbers as permissible complex numbers.

This causes minor trouble in that there are four possible complex infinities and four zeroes, which map naturally to four directions. Except, of course, there is no particular reason to believe that a complex infinity or infinitesimal does lie on one of the four directions!

A more serious problem, however, is the interaction of infinities and NaNs, such as when the real part is an infinity and the imaginary one a NaN. Which attribute should override the other comes down to the precise model of what a NaN is (see Appendix A).

C99 produced a specification of this area, but it was opposed so vehemently that it was published as an informative Annex. It seems likely that it will be standardised in due course, probably without a change of design (see Appendix B).

The Existing State

I did a limited test on over a dozen compilers on several systems in 1995, and discovered that they were all ghastly or broken; I haven't repeated it, but I don't think that it has improved. Doing a serious test on this area is very hard, as the problem is that all exception handling is undefined behaviour.

Summary of Error Diagnosis

Because of these problems, none of signals, *NaN*s or exception flags can be relied upon to flag invalid operations in Fortran, C or Java, and we have no way of checking for numerical failure. This is not just a theoretical problem, but is the case in most of the systems that I have access to. Even the traditional "last resort" practice of running the code on a different system is no longer of much use.

This situation has been of the greatest advantage to the purveyors of inferior libraries. Back in the 1970s, numeric programmers often found that their "working" code crashed or gave completely wrong answers when used with a different set of data or on a different system. This was much less common with high-quality libraries like NAG, though it still happened, of course. But, nowadays, users find that their code "works" and gives very similar answers on all the machines they try it on. In many cases, this remains the case even when critical calculations include operations like $0/0$.

Clearly, it is unfair to blame IEEE 754 for this. But it is unfortunately true that many of the changes that it introduced have led to this state of affairs.

Possible Improvements

Realistically, none of these are going to help, because the damage has been done, and far to many subsequent developments are now dependent on getting numerical error handling wrong. To maintain some sanity, only changes to areas related to IEEE 754 will be mentioned here.

- *NaN*s should be specified more precisely, at least by specifying how signalling and quiet *NaN*s are distinguished, and preferably by specifying the values that are delivered in operations defined by IEEE 754. This would make it possible to write semi-portable support libraries and test suites.
- A more complete set of the "bit munging" support operations should be specified, starting from the Appendix. These should be specified **not** to trap on signalling *NaN*s, and **not** to raise exceptions. They should be documented as **not** appropriate for direct use as numeric primitives, which is essential to prevent languages like C and Java from using them in that way.
- Some more numeric operations should be specified, starting from the Appendix and including `max` and `min`, and these should be **required** to percolate *NaN*s and raise exceptions, just like any other numeric operations.
- The existing `copysign` specification should be deprecated, as a misleading combination of the two categories. Several of the operations in the Appendix should be converted into two forms, one of each of the above types.

- An extra mandatory operation should be specified, that takes a flag mask as an argument, traps if any of the exception flags are set that correspond to that argument, and clears all of the flags. Obviously, there is no concept of a current floating-point value in this case. "Third generation" languages should be recommended to include a mode that calls this upon function entry and return, trapping at least the invalid and divide by zero exceptions, with an option to trap overflow as well.
- Recommended practice for the implementation of the standard special functions and complex arithmetic should be specified, preferring safety over convenience – i.e. **not** like C99.
- True probabilistic rounding should be included as an implemention option, which the user could select, to at least permit it to be used as a debugging option.
- The common extended format that uses 15 exponent bits and 112 mantissa bits (excluding the top bit) should be specified as the standard extended type, and made an implemention option.
- Ideally, a sign-indeterminate zero should be added to the representation, but that is is a huge change. It is easy to see how it could be done, albeit messily, but it would never be accepted.

Appendix A: Infinities, NaNs and Zeroes

This describes some of the problems with the IEEE numeric model that are referred to above. It is not complete, and is not meant to be, but may help clarify some of the above points. Note that it is referring simply to the use of IEEE 754 for a single value, and does not cover its use for interval arithmetic.

The Problem of Zero

Floating-point zero is used to stand for three types of numerical quantity: true zero, negative infinitesimals and positive infinitesimals. Calculations can produce zeroes that belong to any of the subsets defined by the 7 combinations of these, though the discontiguous subset (i.e. infinitesimal, of either sign, but definitely not zero) is relatively uncommon.

The traditional model of unsigned zeroes maps all of these into one, which is mathematically wrong, but it is not clear that the signed zero model (as used in IEEE 754) is any better. There is no problem when numbers are stored as intervals, of course, but consider the standard case when they are stored as single numbers. Which sign should $sin(x)/exp(x)$ where x is $+inf$, for example?

This confusion is compounded in IEEE 754 by the specification of how the sign bit is handled in cases such as the following:

$0 + x$ ($\neq x$ if x is -0)

$x - y$ ($\neq -(y - x)$ if $x = y$)

`copysign(1,x) = -1` ($\neq x < y$))

$\sqrt{(}-0)$

-0 is the only number acceptable to *sqrt* and the only result of it that has the sign bit set, which is a trifle bizarre. Furthermore, some of these operations might be expected to raise the inexact exception, because they can lose the sign information of a zero, but they do not.

So we end up with the situation where zeroes are signed, but the sign does not have any standard mathematical interpretation, and is often unpredictable anyway. This is clearly not satisfactory, and means that the sign bit of zero cannot be regarded as more than a vague indication of the sign of an infinitesimal.

What we really need, of course, is a way of specifying a sign-indeterminate zero **in addition** to the signed zeroes. This would have the slightly odd property that its inverse would have to be a NaN and not an infinity, but that is not really a problem. However, we are not likely to see it provided in any hardware arithmetic, given the current situation.

Infinities

IEEE 754 infinities are fairly simple, once it is realised that they often (or even usually) stand for a finite number whose sign is known, but which is too large to represent (perhaps the result of overflow), and not a transfinite number. However, they do add a little extra complexity to some calculations and, as far as exception handling is concerned, they can be regarded as a simple form of NaN.

Perhaps the nastiest problem is the way in which they can cause standard mathematical invariants to fail. This is precisely the traditional "quiet underflow to zero" problem, but for large numbers rather than small. For example, $A < 0.5 * B$ is quite consistent with $(A * C)/C > 2.0 \times B$. It is not clear whether this is soluble.

A related problem concerns the logarithm function. The logarithm of infinity (i.e. a number that has overflowed) is not necessarily very large, yet the only reasonable value to return is an infinity. This is consistent, but extremely deceptive, and few users realise the consequences. In particular, it means that an IEEE infinity is an overflow indication, but does not necessarily correspond to a large number.

What does "Not a Number" Really Mean?

A large part of the confusion over NaNs is that people have different ideas of what they are supposed to represent. If we think mathematically, and ignore such things as invalid data formats, we still get at least the following categories of NaN (plus combinations and variations):

A A missing value (i.e. unknown but valid)

B Not numeric at all (e.g. 'purple')

C Inapplicable (i.e. not a datum)

D Numerically indefinite (e.g. $\approx 0/\approx 0$)

E The result of an invalid operation

A stands for a value that is 'reasonable' (i.e. defined and finite) but is not known. It is traditional in statistical packages, and is often caused by measurement failure due to unrelated circumstances (e.g. a motorway was built over the experimental plot before harvest, or the mouse escaped.)

B is a value that is not expressible as a simple number, but is most definitely not zero, missing or invalid. It can occur as exceptional cases in basically numeric fields (e.g. the personal income of a Trappist monk, or 'most dreadful' as a historical measure of windspeed.)

C is when the value has no meaningful existence – the datum is simply absent. It is used for things like a conditional measurement when the condition is false, such as a man's age at menarch. It is not missing, because it cannot be interpolated.

D is the result of a calculation where the inaccuracy is such that the value could be anything, and all that is known is that it is a (mathematical) real number. It is often the result of division by an approximate zero and similar actions.

E is the result of a plain error, such as adding '1' to 'most dreadful', and indicates that the computation process has done something stupid or undefined. It differs from **D** in that it does not stand for an unknown number but for a definitely invalid datum, often of an unclassifiable type.

The point about the above is that each category should have different percolation rules and cases when signals should be raised. There is no point in going through the complete list here but, for example, consider multiplication by zero:

Operation	Result
0×0	0
$0 \times finite$	0
$0 \times \pm infinity$	**D** †
$0 \times$ **A**	0
$0 \times$ **B**	**E** †
$0 \times$ **C**	**C**
$0 \times$ **D**	**D**
$0 \times$ **E**	**E**

† This should signal an invalid error. The reason that $0 \times$ **B** should do this is because the 'value' might include an infinity or even not correspond to a numeric value of any type.

C, **D** and **E** are fairly similar in this calculation, but differ in others. For example, the **only** calculations on **C** that should ever raise an exception or return any other value are when the other operand is **E**. Operands **C** do not exist and should be ignored in calculations.

The term "Not a Number" implies category **B** but the IEEE 754 rules fit rather better with category **D**. While I may be mistaken, I believe that IEEE 754 uses the term *NaN* to refer to a value that is known to be numeric, but where its value could be anything.

A hierarchy of *NaN*s would be needed to support the different categories of *NaN* as described above, and multiple distinct values are needed for category **B**, with all the specification that such a use implies. It is a great pity that IEEE 754 did not close the few loose threads what would have made *NaN*s useful for category **B**, as it is of such great importance in many areas of statistics, but it did not. Some of the practical problems using IEEE 754 to implement such use include the lack of defined methods for for testing for NaN equality, and for preserving information held in the value when converting a signalling NaN to a quiet one.

IEEE 754 was clearly **not** meant to address this sort of use, which is reasonable, but some of its requirements actually prevent NaNs being used in such a fashion, which is unfortunate. For example, consider the case of $0 \times \mathbf{A}$ above: IEEE 754 requires the answer to be **A**, whereas the correct answer should be 0. Similarly, consider $\mathbf{B} + \mathbf{D}$: IEEE 754 requires this **not** to raise an exception and to deliver one of **B** or **D**, whereas the correct result is to raise an invalid exception and deliver **E**.

Given this situation, there seems no way to use NaNs portably except as a single, unqualified, 'error' value corresponding to category **D**. This is a reasonable position, provided that its consequences are understood.

Appendix B: The Influence of C and Unix

Whether we like it or not, C and Unix have been the dominating language and operating system of the 1990s, and the vast majority of new special-purpose languages, libraries and protocols have used them as models. This often applies even to systems that are not remotely C- or Unix-like, because they often take some of their language-independent concepts from them. Exception handling is particularly pernicious in this respect, and the following describes some of the problems.

Integer Overflow and Conversions

COBOL, SNOBOL etc. required integer overflow to be trapped, but most languages (including Fortran) leave it undefined; there have been several Fortran systems that checked for it properly, and several others that allowed it to be trapped but did not always generate an exception. Kernighan and Ritchie C (K&R C) relied on it being ignored, and C90 formalised this by requiring two's complement arithmetic with wrapping on overflow for unsigned integers and leaving overflow in signed arithmetic undefined. Java requires the former for all integer arithmetic.

For related reasons, most modern hardware architectures either do not support the trapping of integer overflow or make it extremely hard to enable. Almost all provide only two's complement with wrapping on overflow, with at most a temporary carry flag, and I know of none that provide infinity or NaN support, along the lines of IEEE 754 or otherwise. Even on systems where it is possible to enable trapping of integer overflow, it is usual for the compiled code or run-time system to fail horribly if it is enabled.

Conversions from floating-point to integer are equally bad, and overflow or the conversion of a NaN often gives nonsense and no error indication. The IEEE 754 support in C99 has **not** made this error, and requires an invalid exception in these cases. On the other hand, Java **defines** the conversion of a NaN to an integer to produce zero, and to raise no exception.

Mainly for (hardware) implementation convenience, almost all integer arithmetic is now two's complement with quiet wrapping on overflow, and this has become enshrined in many programming languages, including Java and (to some extent) C99. Furthermore, a horrific number of computer scientists believe that it has been **proved** to be the 'best' design for **all** purposes; there is effectly no chance of an improvement in the forseeable future.

The effect seems to be that, for the forseeable future, we shall be stuck with a situation where integer arithmetic has no useful error handling, whether via interrupts, flags or exceptional values. Even worse in this context, integer arithmetic (in all its guises) will usually have the effect of losing the IEEE 754 NaN values.

Returning Error Indications

The original design of Unix seemed to be to detect and flag all errors that the programmer might want to handle, which is reasonable. But the converse was that 'hard' errors were typically ignored completely, however serious they were, and I have reason to believe that this was a deliberate omission. For example, it was POSIX that first specified that closing a Unix file **could** return an error indication – even now, there are some important Unix operations which commonly fail in detectable ways but have no way to indicate failure to their caller.

This has been copied into C. While most of the really nasty traps are in the I/O area, some are in the numerical one. For example, the mathematical functions in <`math.h`> are **required** not to signal an exception, whatever their arguments, but to return an indicative value and **optionally** set a global variable (`errno`.) Unfortunately, functions whose specification does not mention `errno` are permitted to set it even when they succeed (and need not do so consistently, either.)

The same approach has been taken for the IEEE 754 features in C99. Functions such as `rand` and (even integer) **abs** may set any of the IEEE 754 exception flags, spuriously, and it is possible that ones like `tan` may do so as well. So the **only** way to use the exception flags is to separate expressions into assignments with at most one function call or floating-point/integer conversion in each, clear the flags immediately before each one, and test them immediately afterwards. In 'normal' code, neither `errno` nor the exception flags can be relied upon.

In fact, even the above does not work in many codes. The IEEE flags are program global, so that a program that supports signals must save them on entry to the handler and restore them on exit. As most daemons and parallel codes rely on at least some asynchronous or external signal handling, this is yet another "gotcha". This also applies to any library code that traps such signals, even if that code does not use IEEE support itself.

This has the effect that the IEEE exception flags are of little use in most programs. They are only reliable in those that are written in a specially unclear and inefficient style, and which use only libraries guaranteed to be "IEEE safe", which is not very useful in practice.

Signalling (Interrupts) and Traps

The trapping and handling of arithmetic exceptions has a long and ignoble history.

Most languages have either ignored it completely (e.g. Fortran) or defined some facilities that are too restrictive to be of much use (e.g. Ada, Modula-3). Part of the reason for this is that most hardware architectures and operating systems have provided facilities with such bizarre and incompatible restrictions that they are almost impossible to describe, let alone standardise.

There are essentially four things that can be done on an interrupt: the error can be flagged in a global variable, control can be passed to an earlier stack level, the system can write a diagnostic and stop the program, or an asynchronous thread can be started to handle the error without necessarily halting execution of the failing thread. The last aberration was once favoured by POSIX.4a, but sanity eventually prevailed, and it was made optional. For a variety of reasons, the first three have been standardised out of consideration in C.

The Intel 8086 excluded one of these possibilities, because it allowed the trapping of floating-point exceptions but did not save enough state to permit continuation. This was fixed in the 80286 – unfortunately, this deficiency had by then been made part of the C90 standard (i.e. `SIGFPE` is trappable, but a handler may not return.) More recently, this mistake has been copied to the POSIX standard, and was introduced (deliberately) on the DEC Alpha.

Jumping out to an earlier stack level has always been tricky, in almost any language, because it can leave operations half completed and conflicts with global optimisation. C90 did not want to make the `setjmp` macro into a complete barrier, so it put very loose restrictions on this area. Unfortunately, experience has shown that the restrictions are too loose to be usable, and this is unchanged in C99. To a first approximation, it is not possible to jump out of a handler in C and continue normal execution.

Writing a diagnostic and stopping is equally unreliable, largely because much of C I/O is done by macros (e.g. `putc`) that operate on the run-time system's internal variables, but expand into open code and are subject to normal optimisation. Given the amount of code movement and the asynchronicity of interrupts on modern hardware, the run-time system's internal variables are extremely likely to be inconsistent on entry to the handler. While this **can** be implemented safely, and I have done so, it is very tricky and few systems even attempt it.

I have some proposals on how this situation could be improved, but failed to get any progress in C99. We have to assume that signalling and trapping will remain unavailable or hopelessly unreliable in C and C-like languages for the forseeable future.

Not surprisingly, given the above, C99 does not support signalling NaNs at all, and use the term NaN to mean a quiet NaN throughout. It does not go quite as far as actually forbidding signalling NaNs, but the restrictions are such that I cannot imagine any implementation actually bothering to support them, or any program attempting to use them.

And, lastly, Java actually forbids the use of signalling NaNs altogether.

Complex Arithmetic

And this is where I stopped.