Computer System Support for Scientific and Engineering Computation
Lecture 4 - May 12, 1988

1 Exception Handling Revisited

In the previous chapter we posed a problem concerning algorithms to determine any machine's integer format's range. The solution to this question is not as trivial as one would imagine.

Assignment : In a vanilla higher-level language like FORTRAN, program a way to discover the full range of any machine's integer format.

Discussion : There are some languages where one gets the illusion that this task can be done quite easily, but the solution may not work on all classes of machines. For example, in the C programming language, one can assign an integer variable the value of 1 and then use the left shift operator to shift the "1" bit left. One would argue that when the "1" reaches the most significant bit, the integer will suddenly turn negative. By counting the number of shifts before the integer turns negative, one can presumably derive the full range of the machine's integer format. Even though this procedure appears logical, it is not guaranteed to work on all classes of machines because C can, in principle, be implemented on a decimal machine. On a decimal machine, the left shift operator may be implemented in a different manner in software. In such a machine, integers may not be represented in binary, but may be represented in decimal instead. For decimal integers, the left shift operator can presumably be simulated by doubling the value of the integer. Under this circumstances, we may not eventually obtain the negative number that we expected.

The moral of this example is that we must be extremely careful when we design algorithms. Designing algorithms requires an abstract model of the computing environment that takes into consideration every possibility which may arise. As an example of the complexity of constructing an algorithm, consider the program listed in Appendix A which determines the range of integers on machines whose radices are either 2, 3 or 10. In order to determine the largest integer we must generate it, but we won't know that we have generated the largest integer until we have gone too far. Trying to generate an integer larger that the largest possible integer results in unexpected results which we cannot predict. In fact, it may simply abort on an integer overflow. Some computer languages have the capability of trapping an error when it

Ma
18 May

is encountered, passing control to an error handler. For instance, the "ON ERROR GOTO" and the "resume" statements in the program in Appendix A function as error handlers. Error handling such as described above is, in essence, exception handling.

Programmers generally try to avoid dealing with exceptions. When one begins to work with exceptions, there is no end to them. Those of us who are sane would rather not deal with exception handling, but some of us are willing to do the "dirty job". Even though it makes perfect sense for us to reap the benefits of these masochists, a majority of us are still unwilling to deal with exception handling as it is disgustingly non-standard. In most situations when we encounter an exception, either the machine-dependent exception handler takes over or the program aborts. Under such an environment, how can one write a portable program to discover the full range of any machine's integer format?

A solution to the problem of discovering a machine's integer format's range is to generate all consecutive integers in turn, and when the next integer generated results in an exception we know that the previous integer was an extreme integer in the range. A problem here, as discussed earlier, is that some machines simply terminate when an exception is encountered. We must, therefore, somehow keep track of the largest or the smallest integer successfully generated. A simple way to do this is to write integers as they are successfully generated onto a file. If the integers are generated in increasing order, and if the file buffer is emptied promptly, the largest integer in the file upon program termination is the largest integer possible on that particular computer. The *Paranoia* test program to test implementations of floating point arithmetic developed by W. Kahan at the University of California at Berkeley is based on the same idea. The BASIC implementation of Paranoia is littered with "ON ERROR GOTO" statements, but a Fortran implementation due to Tom Quarles, George Taylor, Daniel Feenberg and David Gay is full of "write" statements. Each write statement, whose purpose is to keep track of the progress, is appropriately called a "milestone".

Although the procedure described appears to work in theory, in practice, one may obtain rather unexpected results as the program may *not* abort where the exception occurs. The computer may have executed a few other instructions before it aborts because the compiler may overlap or rearrange these instructions, or the file buffer may not be empty when the program terminates resulting in loss of output. Consequently, one must be very careful to ensure that he is getting the correct "milestone".

2 Benchmarks

Benchmarks are designed to evaluate the performance of computer systems. Some benchmarks are used to determine the speed of a system, others to measure the bandwidth of its input-output devices or to test for the conformance of its floating point arithmetic with certain standards. One of the factors on which computer system customers based their decision is by comparing benchmark results on the various systems they have under consideration. Examples of benchmarks commonly used are Linpack, SPICE, Whetstone, Paranoia and numerous other tests.

As an example of a simple benchmark program, consider the following program which resembles one used by an Australian university to determine which system to acquire :

$$
\begin{aligned}
a &= b/c \\
d &= a/h \\
e &= d*c \\
f &= e*h \\
g &= |f-b|/c \\
&\vdots \\
\cdots\ f &\ \cdots
\end{aligned}
$$

The above code is enclosed in a loop where b and c are assigned different values for each iteration. The objective of this test is to compare the accuracy and the speed of the various systems under consideration. The correct value of g should be 0 but it is often non-zero because of rounding errors. The computational accuracy of a system under test is determined by the tininess of g; smaller values of g earn bonus points for the system. To be fair, points are also deducted in proportion to the amount of time the computations take. This way, if a system uses double precision to improve its accuracy, its computational speed will suffer and it will lose points.

The computer company which was awarded the contract was Prime. Prime's compiler realized that since a, d and e were not referenced in other parts of the program, their values did not have to be stored in memory; they were kept in registers, which resulted in a substantial reduction in execution time. Prime's computational accuracy far exceeded those of its competitors; in fact, its results were exact. The reason for the exact result is that Prime's floating point arithmetic operations are performed in internal registers which had greater precision than the precision of the variables. The values of b and c are first loaded in the registers where b/c is computed. With the value b/c still in the register, it is divided by h which is later multiplied by c and then by h. All these computations are computed in registers where rounding errors occur only at the lower few bits. However, when the result is rounded and stored in f all the noise incurred in the multiplications and divisions is rounded away and $f = b$.

It should be noted that benchmarking is very important and that we should understand it thoroughly. Computer vendors sometimes select and show only benchmarks for which their computers are well suited. Computer customers need to really understand benchmark results presented by computer vendors in order to make wise decisions. Interested readers should read critically the January 1988 issue of *PC Tech Journal* on floating point arithmetic benchmarking.

3 Exact Arithmetic

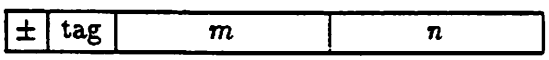
There are people, and there will always be people, who believe that we should represent numbers exactly instead of using floating point number representation. It is generally possible to represent numbers exactly, even if they involve transcendental numbers, because there are so few of them that matter. For example, we can always represent the value of π by the symbol π, which results in no rounding errors. When we encounter the symbol $\frac{\pi}{2}$ in a familiar context, like $\cos(\frac{\pi}{2})$, we can replace it by 0. If, however, we had used floating point

representation for π, we may not have obtained exactly 0 since $\frac{\pi}{2}$ was not stored exactly. In fact, there are theorems which state that if we must compute exactly then we may have to wait a very long time. The word "exactly" in this context is usually interpreted as not exactly "exactly", but as close as we wish provided we wait long enough. Since we do not know how long is long enough, we have to compute approximately in order to obtain the solutions in a reasonable length of time.

When we perform computations involving fractions on some calculators, we sometimes get the impression that the computations are exact when in fact they really are not. Internally, when the integers for the fractions are too large, the fractions are approximated by some other number representation, like floating point representation. Before the solution is displayed, it is tested by a continued fraction technique to determine if it can be approximated by a pretty simple fraction. If the computed solution can be approximated by a fraction, the fraction is displayed giving us the illusion that the computation is exact when, in fact, the "exactness" is simply cosmetic. Such a scheme is not desirable because if we need to compute $c = a - b$ where a, a value very close to b, is approximated to b, then we would obtain 0 instead of a very small number.

3.1 Floating Slash Number Representation

A way to represent numbers exactly by fractions is called *floating slash number representation.* This number representation scheme has been proposed by Matula and Kornerup in *Proceedings of IEEE Symposia on Computer Arithmetic*; these Proceedings also contain many other schemes of varying merits. Interested readers should refer to the proceedings for further details. The floating slash representation for $\pm \frac{m}{n}$ is as follow :



In this representation, there is a tag field which points to the boundary between integers m and n. It is possible to represent a wide range of numbers using the ratio of two integers if the integers can get big enough. Floating slash representation, however, has some very disconcerting properties. The numbers which are representable by the floating slash scheme are incredibly variable, that is, two adjacent representable numbers can be separated by a distance which is several orders of magnitude from that of the next adjacent representable number. Ironically, the more bits we use to represent the integers, the wilder the variation becomes. Although the average separation between two representable adjacent numbers is very satisfactory, the extreme behavior is very extreme indeed. Because of the radical variations in distances between representable numbers, the floating slash scheme is not widely appreciated by error analysts.

There are applications in which floating slash and other rational arithmetic schemes are appropriate and adequate. In problems such as solving small systems of linear equations by the Gaussian elimination method and linear programming by simplex methods, one often starts with fairly small integers and the solutions can normally be represented by simple fractions. In general, however, these arithmetic schemes are not adequate. Another problem with these arithmetic schemes is that, in order to represent numbers in fractions, one does not generally know beforehand the number of bits needed to represent the integers in the worst case. Ideally, the scheme used should be able to practically represent a very wide range of numbers, but this is incompatible with a prior commitment to a fixed word size

for the representation.

3.2 Symbol Manipulators

In spite of the deficiencies of using floating slash or other rational arithmetic schemes, they are used in symbol manipulating environments like MACSYMA, MAPLE and REDUCE. Some symbol manipulators can solve certain problems exactly, sometimes even in closed forms. Closed form exact solutions are not as attractive as they sound. As an example, consider evaluating the indefinite integral

$$\int \frac{dx}{x^{16}-1}$$

which can be performed in closed form. The closed form solution returned by the symbol manipulators, however, are not always desirable. It may be an awkward expression with unreasonable coefficients. This situation is especially so when x^{16} is replaced by x^{32} or x^{64}. Nonetheless, they are adequate in a wide number of applications.

Symbol manipulators are accomplished most naturally by arbitrary width multi-word integer arithmetic. Multi-word integer arithmetic can be implemented quite easily using the linked list structure where integers are linked together to denote the multi-word integer. When we perform a certain operation like multiplication, we operate on the list of integers, and produce another, possibly longer, list of integers. The LISP environment is especially well suited for handling multi-word integers using a linked list structure. Alas, the time for a computation can grow enormously with the increase in the sizes of integers used in the multi-word scheme. The sizes of integers used in the fraction can be reduced if we reduce the numerator and denominator to lowest terms, that is, to divide both the numerator and denominator by their greatest common divisor using the greatest common divisor (g.c.d.) process.

3.3 Arithmetic Using Rational Numbers

As mentioned earlier, there are certain domains where representing numbers by fractions is adequate. If the solutions in these domains are representable by rational numbers where the denominators are predictable, then the computations can be performed *entirely* in integers. Under this condition, operations on the integers can be done very efficiently by a modular technique involving the *Chinese remainder algorithm.* An interesting and disconcerting point about integer arithmetic is that even though all the final results are of reasonable size, it is not unusual for the intermediate values to be enormously larger. The Chinese remainder theorem provides a way to deal with this problem. We can quite easily compute a collection of prime numbers which are slightly smaller than the word size of the machine. Whatever integer we wish to represent can then be represented by its remainders modulo the collected primes. It turns out that for all ring operations (add, subtract, multiply but not divide) on integers, we can actually operate on the residues without any carry propagation. Since there are no propagations, these operations can actually be performed in parallel. Since the magnitudes of the constituents are bounded in magnitude, operations on them are much faster than if we had operated directly on the given integers. After all necessary computations on the residues, the Chinese remainder algorithm enables us to retrieve all the answers from the various remainders. For further details on this nifty way of operating

on integers, interested readers should refer to *The Art of Computer Programming : Seminumerical Algorithms* by Knuth and *The Design and Analysis of Computer Algorithms* by Aho, Hopcroft and Ullman.

The range representable by floating point numbers is larger by far than the integers. Suppose we wish to add two integers. If these integers are in integer format and their sum is too large, then we have an overflow. If, however, these integers are in floating point format, their sum may not overflow, but we may lose the bottom few bits instead. The following is an assignment for interested readers :

Assignment : Compute the following expression where A, B, C and D are integers, represented in floating point format, which are less than half the largest representable integer :

$$\frac{M}{N} = \frac{A}{B} + \frac{C}{D}$$

Structure your computations so that there is no loss in accuracy if M and N are also representably exactly.

Hint : Use the g.c.d. algorithm.

In the computation above, it is sometimes not easy to determine if the computation has been done without any loss in accuracy. For machines which conform to the IEEE format, the solution is surprisingly simple. One simply has to check the inexact flag to determine when there is any loss.

4 Floating Point Number Representation

As we have seen, there are ways of computing arithmetic exactly if we so desire. Often, however, these methods are sufficiently complicated that we would rather use a simpler method at the expense of some inaccuracy. At present, by far the most convenient and simple way is the floating point scheme. The conventional floating point scheme represents a number x as follows :

$$x = \pm\beta^e \times [d_1 d_2 \cdots d_p] \qquad d_j \in [0, 1, \ldots, \rho]$$
$$\rho = \beta - 1$$

where β is the radix and e is the exponent. There is a point somewhere in the d_j field; if we are working in the binary domain ($\beta = 2$), then the point is called a *binary point*; if $\beta = 10$, a *decimal point.*

The encoding of floating point numbers shown above is stored in memory as shown :

$$X = \boxed{\pm} \boxed{e + \text{BIAS}} \boxed{\cdots d_j \cdots d_p}$$

For example, if BIAS = 128,

$$+1.234 \times 10^6 = 12340000$$

when stored in memory may be as follows :

where the exponent may be a binary integer and the significant field may be 5 hexadecimal digits. In this example, 8 hex digits are used : 1 for the sign, 2 for the exponent and 5 for the significand. (There are more compact encodings.)

The fields can be arranged in other permutations, but there is a very good reason for arranging the fields as they are shown. The main reason is that this arrangement enables making comparisons very quick and easy. If the fields are arranged in some other manner then the numbers may have to be subtracted in floating point format to be compared. This way of performing comparison is clearly inefficient. Using the encoding shown above, which has lexicographical ordering properties, comparison can be achieved easily and quickly. In lexicographical ordering, if we consider the bit string representing the floating point number as a signed integer, the ordering of the floating point numbers is the same as the ordering of the integers corresponding to the bit strings of the floating point numbers. In practice, this is usually how floating point numbers are compared. Floating point numbers represented in the IEEE 754 standard format have the lexicographical ordering properties. Note that the reason for the bias in the exponent is to preserve lexicographical ordering properties. Not all current machines use lexicographical ordering for their comparisons; Burrough machines have a separate sign for the exponent, so their floating point number representation is not lexicographically ordered. As a consequence, comparisons on these machines are performed the hard way.

We have not discussed where the point is located in the significand. DEC and IBM have their points to the left of d_1, but the point is between d_1 and d_2 in the IEEE 754 standard. CDC Cyber 170 series places the point immediately to the right of d_p. The difference between the placement of the point merely amounts to a change in the bias, BIAS. If the point is moved one digit to the right, BIAS is incremented by 1. Thus, by changing the bias, we can shift the position of the point. The placement of point, therefore, is just a matter of convenience.

We know that the position of the point is related to the bias. We still have not determined what the value of bias should be, or equivalently where the point should be placed. Changing the value of BIAS affects the range of representable numbers. For instance, adding 10 to BIAS results has the effect of shifting the range of numbers towards zero. So, when considering the value of BIAS, there is a question of the balance of the range of representable numbers.

Let us consider the importance of choosing the appropriate value for BIAS. In a CDC Cyber, the bias is chosen so that the product of the largest and the smallest positive number is a number significantly larger than 1.0. This situation poses problems because when we take the reciprocal of a number smaller than the largest number, we may have an underflow!

In the next chapter, we shall discuss which radix is the best and the effects of the various methods for rounding.

Appendix A

```
10  '  WHICHINT.BAS  is a BASIC program to discover which integers the
20  '  computer on which it runs can handle in its  INTEGER  format.
30   DEFINT A-Z ' ...  or  INTEGER ...  in other BASIC dialects.
40   01 = 1 :  IF (01>0 AND 01*01=01) THEN 60
50      PRINT "Something is  VERY  wrong with  1 ." :  STOP
60   02 = 01+01 ' ... Test the hypothesis that the machine is  BINARY :
70   P = 02 :  J = 02+01 ' ...  j = 2^P - 1
80   ON ERROR GOTO 220 ' ... and resume at 120
90      P = P+01 :  I = J :  J = I+I+01 :  D = (J-I) - I
100     IF D><01 THEN PRINT "FLOATING-POINT is used for INTEGERS." :  STOP
110     IF J>I THEN 90 ' ...  else now  i = 2^(P-1)-1 >= j = i+i+1 . !
120  ON ERROR GOTO 230 ' ... and resume at 140
130  J = I+01 :  IF J>I THEN 300 ' ...  else now the machine IS binary.
140  ON ERROR GOTO 240 ' ... and resume at 160
150  M = -I :  IF M<0 THEN 170 ' ...  This ought not to overflow,  but ...
160     PRINT "Negative integers malfunction!" :  STOP
170  ON ERROR GOTO 250 ' ... and resume at 200
180  J = M-01 :  IF J>=M THEN 200
190     PRINT P;" digits of Twos' complement"; :  GOTO 210
200     PRINT P;" digits of either  Sign-Magnitude or Ones' complement";
210     PRINT "  BINARY  (B = 2)." :  STOP
220               RESUME 120    ' ...   IBM PC  BASIC  requires these
230                RESUME 140   ' ...   RESUME statements to prevent
240                 RESUME 160  ' ...   subsequent  "ERRORS"  from
250                  RESUME 200 ' ...   terminating the program.
300  03 = 02+01 ' ... Test the hypothesis that the machine is  TERNARY :
310  P = 02 :  J = 03+01 ' ...  j = (3^P - 1)/2
320  ON ERROR GOTO 490 ' ... and resume at 350
330     P = P+01 :  I = J :  J = I+I+I+1
340     IF J>I THEN 330 ' ...  else now  i = (3^(P-1)-1)/2 >= j = 3i+1 . !
350  ON ERROR GOTO 500 ' ... and resume at 370
360  J = I+01 : IF J>I THEN 410 ' ...  else now i = 111...111 is maximal.
370  ON ERROR GOTO 240 ' ... and resume at 160
380  M = -I ' ...  This ought not to overflow,  but ...
390     IF M>=0 THEN 160 ' ... else now  m = 222...222 or TTT...TTT < 0 .
400     PRINT P;" digits of  Threes' complement  or  Balanced";: GOTO 480
410  ON ERROR GOTO 510 ' ... and resume at 600
420  J = I+I :  IF J<=I THEN 600 ' ... else  j = 222...222 > 0 .
430  ON ERROR GOTO 240 ' ... and resume at 160
440  M = -J :  IF M>= 0 THEN 160 ' ... else now  m = -222...222 < 0 .
450  ON ERROR GOTO 510 ' ... and resume at 600
460  J = J+1 :  IF J>I THEN 600 ' ...  else now  222...222  is maximal.
470     PRINT P;" digits and a sign for  Sign-Magnitude";
480     PRINT "  TERNARY  (B = 3) ." :  STOP
490               RESUME 350
500                RESUME 370
```

```
510                 RESUME 600
600  N = 03*03 :  T = N+1 ' ... Check that the machine really is  DECIMAL :
610  P = 01 :  J = 03+01 ' ...  j = 5*10^P - 1
620  ON ERROR GOTO 810 ' ... and resume at 650
630     P = P+1 :  I = J :  J = T*I+N
640     IF J>I THEN 630 ' ... else now i = 499...999 >= j = 10*i+9 . !
650  ON ERROR GOTO 820 ' ... and resume at 670
660  K = I+01 :  IF K>I THEN 700 ' ... else now  i  is maximal.
670  ON ERROR GOTO 240 ' ... and resume at 160
680  M = -I-01 :  IF M>=0 THEN 160 ' ... else now  m = -500...000 < 0 .
690     PRINT P;" digits of  Tens' complement"; :  GOTO 770
700  ON ERROR GOTO 800 ' ... and stop
710  K = I+K :  IF K<=I THEN 800 ' ... else  k = 999...999 > 499...999 .
720  ON ERROR GOTO 830 ' ... and resume at 740
730  J = K+01 :  IF J>K THEN 800 ' ... else  k  is maximal.
740  ON ERROR GOTO 240 ' ... and resume at 160
750  M = -K :  IF M>=0 THEN 160
760     PRINT P;" digits and a sign for  Sign-Magnitude";
770     PRINT "  DECIMAL  (B = 10) ." :  STOP
800  PRINT "This program can't tell what happens to integers > ";I :  STOP
810            RESUME 650
820             RESUME 670
830              RESUME 740 :  END
```

FLOATING-POINT RANGE / PRECISION TRADEOFF FOR RADICES $\beta = 2^k$, $k = 1, 2, 3, \ldots$.

WHICH RADIX IS BEST?

Name	β	k	Who?
BINARY	2	1	IEEE 754, DEC VAX, CDC, CRAY, ...
QUATERNARY	4	2	--- no more ---
OCTAL	8	3	Burroughs B65xx
HEXADECIMAL	16	4	IBM 370, Amdahl, ...

Floating-point word:

$X =$ | $\pm$ (1 bit) | exponent + Bias ($\leftarrow \ell$ bits $\rightarrow$) | $d \mid d_2 \mid d_3 \mid \cdots \mid d_{p-1} \mid d_p \mid d_p$ ($\leftarrow P$ sig. digits $= P \cdot k$ bits $\rightarrow$) |

value $x = \pm \beta^{\text{exponent}} \times [d_1 d_2 d_3 \cdots d_{p-2} d_{p-1} d_p]$

where $0 \leq \text{exponent} + \text{Bias} \leq 2^\ell - 1$

$0 \leq [d_1 d_2 d_3 \cdots d_{p-2} d_{p-1} d_p] \leq \beta^p - 1$

and $\beta = 2^k$ for some fixed k

Total wordsize $w = 1 + \ell + P \cdot k$ bits

Let $\rho = \beta - 1$ so $[00\ldots00] \leq [d_1 d_2 \cdots d_{p-1} d_p] \leq [\rho\rho\cdots\rho]$

Normally X is NORMALIZED: $d_1 \geq 1$ unless $x = 0$.

$$X = \boxed{\pm}\ \boxed{\text{exponent + Bias}}\ \boxed{d_1 . d_2 \cdots d_{p-1} d_p}$$

$$x = \pm\ \beta^{\text{exponent}} \times [d_1 . d_2 \cdots d_{p-1} d_p] \qquad \text{Normalized} \neq 0,$$

$$\beta = 2^k \qquad p = \beta - 1 \qquad d_1 \geq 1.$$

$$0 \leq \text{exponent+Bias} \leq 2^\ell - 1$$

RANGE:

$$\frac{\text{Max. } x}{\text{Min. } x>0} = \frac{\beta^{2^\ell - 1 - \text{Bias}} \times [pp \cdots pp]}{\beta^{0 - \text{Bias}} \cdot [10 \cdots 00]}$$

$$= \frac{\beta^{2^\ell - 1} \times (\beta^p - 1)}{\beta^{p-1}} \overset{\circ}{=} \beta^{2^\ell} = 2^{k \cdot 2^\ell}$$

WORST-CASE PRECISION:

$$\underset{x>0}{\text{Max.}} \frac{(\text{Successor of } x) - x}{x} = \frac{[100\cdots001] - [100\cdots000]}{[100\cdots000]}$$

$$= 1/\beta^{p-1} = 2^{k\cdot(p-1)}$$

What BINARY format has the same RANGE and WORST-CASE PRECISION?

Say ℓ' exponent bits, where $2^{1 \cdot 2^{\ell'}} = 2^{k \cdot 2^\ell}$

p' significant bits, where $2^{1\cdot(p'-1)} = 2^{k\cdot(p-1)}$

i.e. $\ell' = \ell + \log_2 k$, $\quad p' = 1 + k\cdot(p-1)$

For "same" RANGE & WORST-CASE PRECISION

	$\beta = 2^k$	$\beta' = 2$
Exponent field #bits	l	$l' = l + \log_2 k$
Sig. dig. field #bits	pk	$p' = 1 + k \cdot (p-1)$
Total wordsize	$w = 1 + l + pk$	$w' = 1 + l' + p'$

Hence $w - w' = l - l' + pk - p'$

$= -\log_2 k + k - 1$

≥ 0 for all $k \geq 1$.

Name	k	lost bits $w - w' = -\log_2 k + k - 1$
BINARY	1	0 $\quad -1$ for Hidden Bit!
QUATERNARY	2	0
OCTAL	3	$2 - \log_2 3 = 0.415$
HEX.	4	1

WITHOUT HIDDEN BIT (Goldberg's variation),

BINARY matches QUATERNARY's RANGE/PRECISION.

WITH HIDDEN BIT,

BINARY beats QUATERNARY by 1 bit

OCTAL by 1.415 bits

HEX by 2 bits.

Another view is WOBBLING PRECISION

```
10   '  WHICHINT.BAS  is a BASIC program to discover which integers the
20   '  computer on which it runs can handle in its  INTEGER  format.
30   DEFINT A-Z ' ... or  INTEGER ...  in other BASIC dialects.
40   O1 = 1 :  IF (O1>0 AND O1*O1=O1) THEN 60
50      PRINT "Something is  VERY  wrong with  1 ." :  STOP
60   O2 = O1+O1 ' ... Test the hypothesis that the machine is  BINARY :
70   P = O2 :  J = O2+O1 ' ...  j = 2^P - 1
80   ON ERROR GOTO 220 ' ... and resume at 120
90      P = P+O1 :  I = J :  J = I+I+O1 :  D = (J-I) - I
100     IF D><O1 THEN PRINT "FLOATING-POINT is used for INTEGERS." :  STOP
110     IF J>I THEN 90 ' ...  else now  i = 2^(P-1)-1 >= j = i+i+1 . !
120  ON ERROR GOTO 230 ' ... and resume at 140
130  J = I+O1 :  IF J>I THEN 300 ' ...  else now the machine IS binary.
140  ON ERROR GOTO 240 ' ... and resume at 160
150  M = -I :  IF M<0 THEN 170 ' ...  This ought not to overflow,  but ...
160     PRINT "Negative integers malfunction!" :  STOP
170  ON ERROR GOTO 250 ' ... and resume at 200
180  J = M-O1 :  IF J>=M THEN 200
190     PRINT P;" digits of Twos' complement"; :  GOTO 210
200     PRINT P;" digits of either  Sign-Magnitude or Ones' complement";
210     PRINT "  BINARY  (B = 2)." :  STOP
220               RESUME 120     ' ...   IBM PC  BASIC  requires these
230                RESUME 140    ' ...   RESUME statements to prevent
240                 RESUME 160   ' ...   subsequent  "ERRORS"  from
250                  RESUME 200  ' ...   terminating the program.
300  O3 = O2+O1 ' ... Test the hypothesis that the machine is  TERNARY :
310  P = O2 :  J = O3+O1 ' ...  j = (3^P - 1)/2
320  ON ERROR GOTO 490 ' ... and resume at 350
330     P = P+O1 :  I = J :  J = I+I+I+1
340     IF J>I THEN 330 ' ...  else now  i = (3^(P-1)-1)/2 >= j = 3i+1 . !
350  ON ERROR GOTO 500 ' ... and resume at 370
360  J = I+O1 :  IF J>I THEN 410 ' ...  else now  i = 111...111  is maximal.
370  ON ERROR GOTO 240 ' ... and resume at 160
380  M = -I ' ...  This ought not to overflow,  but ...
390     IF M>=0 THEN 160 ' ... else now  m = 222...222 or TTT...TTT < 0 .
400     PRINT P;" digits of  Threes' complement  or  Balanced";: GOTO 480
410  ON ERROR GOTO 510 ' ... and resume at 600
420  J = I+I :  IF J<=I THEN 600 ' ... else  j = 222...222 > 0 .
430  ON ERROR GOTO 240 ' ... and resume at 160
440  M = -J :  IF M>= 0 THEN 160 ' ... else now  m = -222...222 < 0 .
450  ON ERROR GOTO 510 ' ... and resume at 600
460  J = J+1 :  IF J>I THEN 600 ' ...  else now  222...222  is maximal.
470     PRINT P;" digits and a sign for  Sign-Magnitude";
480     PRINT "  TERNARY  (B = 3) ." :  STOP
490               RESUME 350
500                RESUME 370
510                 RESUME 600
600  N = O3*O3 :  T = N+1 ' ... Check that the machine really is  DECIMAL :
610  P = O1 :  J = O3+O1 ' ...  j = 5*10^P - 1
620  ON ERROR GOTO 810 ' ... and resume at 650
630     P = P+1 :  I = J :  J = T*I+N
640     IF J>I THEN 630 ' ... else now i = 499...999 >= j = 10*i+9 . !
650  ON ERROR GOTO 820 ' ... and resume at 670
660  K = I+O1 :  IF K>I THEN 700 ' ... else now  i  is maximal.
670  ON ERROR GOTO 240 ' ... and resume at 160
680  M = -I-O1 :  IF M>=0 THEN 160 ' ... else now  m = -500...000 < 0 .
690     PRINT P;" digits of  Tens' complement"; :  GOTO 770
700  ON ERROR GOTO 800 ' ... and stop
710  K = I+K :  IF K<=I THEN 800 ' ... else  k = 999...999 > 499...999 .
720  ON ERROR GOTO 830 ' ... and resume at 740
730  J = K+O1 :  IF J>K THEN 800 ' ... else  k  is maximal.
740  ON ERROR GOTO 240 ' ... and resume at 160
750  M = -K :  IF M>=0 THEN 160
760     PRINT P;" digits and a sign for  Sign-Magnitude";
770     PRINT "  DECIMAL  (B = 10) ." :  STOP
800  PRINT "This program can't tell what happens to integers > ";I :  STOP
810               RESUME 650
820                RESUME 670
830                 RESUME 740 :  END
```