CTRL87.DOC: DOCUMENTATION for CTRL87 WORK IN PROGRESS!

Prof. W. Kahan
Elect. Eng. & Computer Science Dept.
University of California
Berkeley CA 94720
Sept. 27, 1994

The program CTRL87.EXE, run under MS-DOS 3.3 or later, affects the execution of floating-point arithmetic by Intel numeric (co)processors (ix87s) in IBM PCs and their clones, including in particular ...

- Intel i8087 coprocessors used with i8086/88 and clones,
- Intel and AMD 287 coprocessors used with 286,
- Intel i387 and Cyrix 83D87 coprocessors used with any 386,
- Intel i486DX, and i487 with i486SX, and Cyrix clones,
- Intel Pentium (i586).

CTRL87 exposes ix87 capabilities unexploited by most PC software.

The invocation " ctrl87 ctl msk ", with suitable 3-hex-digit words in place of " ctl " and " msk ", sets as many as nine bits in the (co)processor Control-Word, thereby controlling subsequent floating-point operations' ...
Trapping upon Exceptions like Overflow, Division by Zero, ...,
Rounding to one of three selected Precisions, and
in one of four selected Directions.

The possibilities are summarized in the following lines, which show up after invocations " ctrl87 ? " or " ctrl87 " with blank parameters:

```
+-------------------------------------------------------------+
|  CTRL87 <ctl<, msk>>  sets the  ix87  Control-Word          |
|  C-W := (msk AND ctl) OR (NOT(msk) AND C-W)  from 2         |
|  3-hex-digit parameters  ctl and msk .  C-W's bits          |
|  are  OR'd  to affect floating-point thus:         C-W      |
|  TRAPS:  (default)    Disable All traps      ...  _3D      |
|   or     Disable  trap for  INVALID OP       ...  _01      |
|     and  Disable  trap for  DIV by ZERO      ...  _04      |
|     and  Disable  trap for  OVERFLOW         ...  _08      |
|     and  Disable  trap for  UNDERFLOW        ...  _10      |
|     and  Disable  trap for  INEXACT          ...  _20      |
|  PRECISION:  (default) Round to  REAL*10     ...  3__      |
|                 or else  Round to  REAL*8    ...  2__      |
|                 or else  Round to  REAL*4    ...  0__      |
|  DIRECTION:  (default) Round to  Nearest     ...  0__      |
|                 or else  Round    Down       ...  4__      |
|                 or else  Round    Up         ...  8__      |
|                 or else  Round to  Zero      ...  C__      |
|  Initial Control-Word  ctl  set by  FINIT ...  33D         |
|  Default msk = 0F00 .  Maximal effective msk = F3D         |
|  Current setting of  Control-Word  ctl     ... 0xxx        |
|  Enter  3  hex digits for new  ctl :                       |
+-------------------------------------------------------------+
```

After this display, pressing [Enter] does nothing but quit CTRL87 . Entering a 3-hex-digit ctl instead causes another prompt to show up:
" Enter 3 hex digits for msk or accept 0F00 : "
After this prompt, pressing [Enter] assigns msk := 0F00 and does what invoking " CTRL87 ctl " in the first place would have done; otherwise entering a 3-hex-digit msk does what " CTRL87 ctl msk " would have done. These effects are explained at great length below.

The effects displayed above and to be explained below are mandated for floating-point arithmetics that conform fully with IEEE Standard 754 for Floating-Point Arithmetic, though few compilers support them; and different chips (e.g., Motorola's 68881/2 and 68040) get those effects from different bit-patterns. IEEE 754 is not standard enough!

When CTRL87 may be used. *
Program CTRL87.EXE works by setting nine of the bits in a sixteen-bit Control-Word on Intel ix86/x87 chips. That must be done after the Control-Word has been initialized (by a FINIT instruction etc.) and before executing the floating-point operations to be controlled. This timing is awkward because those operations are performed in some other software from which CTRL87.EXE must be invoked, sometimes via another copy of COMMAND.COM , if that other software allows such invocations. They are allowed by some software, for instance ...

```
MAPLE V release 2   invocation:  [F4] ctrl87 ....  ... Exit
MATHEMATICA  2.2    invocation:     ! ctrl87 ....
MATLAB 3.5          invocation:     ! ctrl87 ....
```

Each such invocation of CTRL87.EXE could be countermanded by software that immediately afterward put a saved copy of the previous Control-Word back, but I have yet to find an instance of that under MS-DOS.

You can get the same effect as " ctrl87 ctl msk " from 16-bit words 0ctl and 0msk within your own programs by coding in ...

```
*** A86 Assembly Language ***     **** Microsoft or Borland  C  ****
    FSTCW  temp       ;
    MOV  AX, 0msk     ;              #include <float.h>
    AND  AX, 0F3D     ;                 ...
    FWAIT             ;              ctrl = 0x0ctl ;
    OR   temp, AX     ;              mask = 0x0msk ;
    NOT  AX           ;              temp = mask & 0x0F3D ;
    OR   AX, 0ctl     ;              temp = _control87(ctrl, temp) ;
    AND  temp, AX     ;              /*  temp  is new Control-Word */
    FLDCW  temp       ;                 ...
```

What " ctrl87 ctl msk" does. *
The hexadecimal words ctl and msk are used to (re)set any of up to 9 bits in the 16-bit Control-Word that determines the nature of ...
Trapping upon Exceptions like Overflow, Division by Zero, ...,
Rounding to one of three Precisions, in one of four Directions,
for all subsequent floating-point arithmetic operations. These options will be explained in detail below under the headings Exception: ..., Precisions, and Directions. The remaining 7 bits in the Control-Word are best left alone provided they have been initialized correctly (one way for the i8087, another for the i387, etc.) To leave them alone, CTRL87 replaces msk by (msk AND 0F3D) internally.

Through an accident of language we say " and " to describe effects achieved by a logical OR of bits for the Control-Word ; e.g., to
Disable trapping on INEXACT (0020) and UNDERFLOW (0010),
and leave other exceptions' traps unchanged, and
Round to Double-Precision (0200) and Towards Zero (0C00) ,
we use logical OR's to form corresponding hexadecimal words

```
ctl  :=  0020  +  0010  +  0200  +  0C00  =  0E30   and
msk  :=  0020  +  0010  +  0300  +  0C00  =  0F30   ,
```

and invoke " ctrl87 0E30 0F30 " to get the effect desired.

Exceptions in General. *
Designers of operating systems tend to incorporate all trap-handling into their handiwork, thereby burdening floating-point exception-handling with unnecessary and intolerable overheads. Better designs should incorporate all floating-point trap-handling into a run-time math. library, along with logarithms and cosines, which the operating system merely loads. To this end, the operating system has only to provide default handlers (in case the loaded library neglects trap-handling) and secure trap re-vectoring functions for libraries that take up that duty and later, at the end of a task, relinquish it.

Exceptions in General on the ix87. * * * * * * * * * * * * * * * * * * *
To Disable an exception's trap is to let the numeric (co)processor respond to every instance of that exception by raising its Flag and delivering the result specified as its " Default " by IEEE 754. For example, the default result for 3.0/0.0 is Infinity with the same sign as that 0.0 . The raised flag stays raised until later set down by the program, presumably after it has been sensed. The ix87 family keep their flags in a Status Register whose sensing and clearing fall outside the scope of this documentation; see manuals for your chip and for the programming environment (e.g. compiler) that concerns you.

```
+--------------------------------------------------------------+
| CAUTION:  Do not change  ( enable or disable )  exception traps |
|        in a way contrary to what is expected by your programming |
|        environment or application program,  lest unpredictable  |
|        consequences ensue.  The default value  0F00  provided for |
|        msk  when  CTRL87  is invoked simply via  " ctrl87  ctl " |
|        prevents such a change from occurring inadvertently.      |
+--------------------------------------------------------------+
```

The ix87 Control-Word has a bit for each of the five exceptions that IEEE 754 recognizes. Setting that bit to 1 disables the exception's trap; 0 enables. An explanation for every exception follows.

Exception: ix87 STACK-BLUNDER. *
An excessive obeisance to Compatibility has propagated a design flaw in Intel's 8087 to later 80287, 387, 486 and Pentium floating-point arithmetics. Our blunder practically prohibits extension into memory of the ix87's on-chip eight-register floating-point " Stack," which was originally intended to obviate any need to save and restore ix87 registers between calls-to and returns-from function subprograms.

Ideally, a function's floating-point arguments (passed by value) were to be pushed onto that stack, consumed, and replaced by the function's return-value(s) without regard for whatever was already on the stack. Actually, pushing a ninth item onto the stack's top causes " Stack-Overflow " that would force the first item out the bottom to be conveyed onto an extension of the stack in memory. Later that item would be conveyed back onto the chip when a " Stack-Underflow " were triggered by a reference to a now empty stack-register. But, because of our blunder, no simple nor fast way will ever exist to convey items from stack-bottom to memory and back, and consequently no generally usable and fast extension of the stack into memory will ever exist.

Software to extend the floating-point stack into memory should have been written and tested, but wasn't, before the 8087's design was frozen. By the time this blunder was appreciated, it was too late to add the eight more tag bits and two special load and store instructions that would have banished the threat of Stack-Over/Underflow from the concerns of early compiler writers and applications programmers. Now the FXCH (exchange registers) instruction has come to be used in ways that almost preclude ever going back to the original intention.

To compound our blunder, Stack-Over/Underflow was mixed up with the INVALID arithmetic operations to be discussed below. Both exceptions are enabled/disabled by the same bit in the Control-Word and signaled by the same flag bit in the Status-Word of the ix87, although they can be distinguished with the aid of a Stack-Fault flag added to the i387 and later chips. That mix-up necessitates the following ...

```
+-------------------------------------------------------------+
| WARNING:  Do not disable the  INVALID  exception,  by invoking |
|     " ctrl87  ctl msk "  with odd integers  ctl  and  msk ,  if |
|     floating-point Stack-Over/Underflow is possible.  There are |
|     compilers that do not preclude that possibility.  Should it |
|     occur with the trap disabled,  a result indistinguishable   |
|     from a data-dependent  INVALID  arithmetic operation could  |
|     seriously confuse subsequent attempts to debug the program. |
+-------------------------------------------------------------+
```

Stack-Over/Underflow must be avoided. That can be arranged by using the stack exclusively for evaluating expressions that are not too long. Fortunately, caches are fast enough nowadays that saving intermediate results in memory and reloading them to registers is barely tolerable.

Exception: INVALID operation. *
Signaled whenever an operation's operands lie outside its domain, this exception's default, delivered only because any other real or infinite value would most likely cause worse confusion, is NaN , which means " Not a Number ". NaN also means " Not All Numbers" ; NaN does not represent the set of all real numbers, which is an interval for which Interval Arithmetic provides the appropriate representation.

NaN must not be confused with " Undefined ". On the contrary, IEEE 754 defines NaN perfectly well even though most language standards ignore and many compilers deviate from that definition. The deviations usually afflict relational expressions, discussed below. Arithmetic operations upon NaNs other than SNaNs (see below) never signal INVALID, and always produce NaN unless replacing every NaN operand by any finite or infinite real values would produce the same finite or infinite result independent of the replacements. For example, 0*NaN must be NaN because 0*Infinity is an INVALID operation (NaN). On the other hand, for hypot(x, y) := sqrt(x*x + y*y) we deduce that hypot(Infinity, NaN) = +Infinity since hypot(Infinity, y) = +Infinity for all real y , finite or not; naive implementations of hypot may do differently.

Some familiar functions have yet to be defined for NaN . For instance max{x, y} should deliver the same result as max{y, x} but almost no implementations do that when x is NaN ; there are good reasons to define max{NaN, 5} := max{5, NaN} := 5 but many people disagree.

IEEE 754's specification for NaN endows it with a field of bits into which software can record, say, how and/or where the NaN came into existence. That information would be extremely helpful for subsequent " Retrospective Diagnosis " of malfunctioning computations, but no software exists now to employ it. The ix87 copies that field from an operand NaN to the result NaN of an arithmetic operation, or fills that field with binary 1000...000 when a new NaN is created by an untrapped INVALID operation. (Other chips may behave differently.)

The ix87 treats a NaN with any nonzero binary 0xxx...xxx in that field as an SNaN (Signaling NaN) in accordance with a requirement of IEEE 754. An SNaN may be moved (copied) without incident, but any arithmetic operation upon an SNaN is an INVALID operation that must trap or else produce a new nonsignaling NaN. (Another way to turn an SNaN into a NaN is to turn 0xxx...xxx into 1xxx...xxx with a logical OR.) Intended for, among other things, data missing from statistical collections, and for uninitialized variables, SNaNs seem preferable for such purposes to zeros or haphazard traces left in memory by a previous program. No more will be said about SNaNs here.

IEEE 754 defines all relational expressions involving NaN too. In the syntax of C , the predicate x != y is True but all others, x < y , x <= y , x == y , x >= y and x > y , are False whenever x or y or both are NaN, and then all but x != y and x == y are invalid operations too and must signal INVALID. Ideally, expressions x !< y , x !<= y , x !>= y and x !> y should be valid predicates, quietly True whenever x or y or both are NaN , but arbiters of taste and fashion for ANSI Standard C have refused to recognize those expressions. In any event, !(x < y) differs from x >= y when NaN is involved, though rude compilers " optimize " the difference away. Worse, some compilers mishandle NaNs in all relational expressions.

IEEE 754 recognizes seven invalid arithmetic operations, all NaN :
real SQRT(Negative) , 0.0/0.0 , Infinity/Infinity, 0*Infinity
Infinity - Infinity , Anything REM 0.0 , Infinity REM Anything

Certain conversions between floating-point and other formats are also invalid. However Infinity + Infinity = Infinity is valid if signs are all the same. Some language standards conflict with IEEE 754; for example, APL expects 0.0/0.0 to deliver 1.0 . Sometimes naive compile-time optimizations replace x/x by 1 (wrong if x is zero, Infinity or NaN) and x - x by 0 (wrong if x is Infinity or NaN) and 0*x and 0/x by 0 (wrong if ...), alas.

Ideally, certain other real expressions should behave the same as the invalid operations recognized by IEEE 754 ; some examples in Fortran syntax are ...
(Negative)**(Noninteger) , LOG(Negative) , ASIN(Bigger than 1) ,
SIN(Infinity) , ACOSH(Less than 1) , ..., all of them NaN .
Those expressions do behave that way if implemented well in software that exploits the transcendental functions built into the ix87 ; to this end, i387 and successors are easier to use than 8087 and 80287.

A number of real expressions are sometimes implemented as INVALID by mistake, or declared Undefined by ill-considered language standards; a few examples are ...

```
      0.0**0.0  =  Infinity**0.0  =  NaN**0.0  =  1.0 ,
      COS( 2.0**120 )  =  -0.9258790228548378673038617641...
```

More examples like these will be offered under DIVIDE by ZERO below.

Differences of opinion persist about whether certain functions should be INVALID or defined by convention at internal discontinuities; a few examples are ...

```
     1.0**Infinity  =  (-1.0)**Infinity  =  1.0  ?    ( NaN  is better.)
     ATAN2(0.0, 0.0)  =  0.0  or  +Pi  or  -Pi  ?     ( NaN  is worse.)
     ATAN2(+Infinity, +Infinity)  =  Pi/4  ?          ( NaN  is worse.)
     SIGNUM(0.0)  =  0.0 ,  or  +1.0  or  -1.0  ?     ( 0.0  is best.)
(  but CopySign(1.0, +0.0) := +1.0  and  CopySign(1.0, -0.0) := -1.0 .)
```

Between 1964 and 1970 the U.S. National Bureau of Standards changed its definition of arccot(x) from Pi/2 - arctan(x) to arctan(1/x) , thereby introducing a jump at x = 0 . This change appears to be a bad idea, but it is hard to argue with an arm of the U.S. government.

Some programmers think invoking " ctrl87 0 1 ", which enables the trap to abort upon INVALID operations, is a safe way to avoid such disputes; they are mistaken. Doing so may abort searches prematurely. For example, try to find a positive root x of an equation like
(TAN(x) - ASIN(x))/x**4 = 0.0
by using Newton's iteration or the Secant iteration starting from various first guesses between 0.1 and 0.9 . In general, a root-finder that does not know the boundary of an equation's domain must be doomed to abort, if it tests a wild guess thrown outside that domain, unless it can respond to NaN by retracting the wild guess back toward a previous guess inside the domain. Such a root-finder is built into current Hewlett-Packard calculators that solve equations like the one above far more easily than root-finders on most PC's and workstatons.

(Attempts to cope decently with all INVALID operations must run into unresolvable dilemmas sooner or later unless the computing environment provides what I call " Retrospective Diagnostics ". These exist in a rudimentary form in Sun Microsystems' operating system on SPARCs.)

Exception: DIVIDE by ZERO. *
This is a misnomer perpetrated for historical reasons. A better name for this exception is
" Infinite result obtained Exactly from Finite operands. "
An example is 3.0/0.0 , for which IEEE 754 specifies an Infinity as the default result. The sign bit of that result is, as usual for quotients, the exclusive OR of the operands' sign bits. Since 0.0 can have either sign, so can Infinity; in fact, division by zero is the only algebraic operation that reveals the sign of zero. (IEEE 754 recommends a non-algebraic function CopySign to reveal a sign without ever signaling an exception, but few compilers offer it, alas.)

Ideally, certain other real expressions should be treated just the way IEEE 754 treats divisions by zero, rather than all be misclassified as errors or " Undefined "; some examples in Fortran syntax are ...

```
 0.0**(NegativeNonInteger) = +Infinity            LOG(0.0) = -Infinity
 0.0**(NegativeEvenInteger) = +Infinity        ATANH(-1.0) = -Infinity
 ATANH(+1.0) = +Infinity       (-0.0)**(NegativeOddInteger) = -Infinity
```

The sign of Infinity may be accidental in some cases; for instance, if TANdeg(x) delivers the TAN of an angle x measured in degrees, then TANdeg(90.0 + 180*Integer) is infinite with a sign that depends upon details of the implementation. Perhaps that sign might best match the sign of the argument, but no such convention exists yet. (For x in radians, accurately implemented TAN(x) need never be infinite !)

Operations that produce an infinite result from an infinite operand or two must not signal DIVIDE by ZERO. Examples include Infinity + 3 , Infinity*Infinity , EXP(+Infinity) , LOG(+Infinity) , ... Neither should 3.0/Infinity = EXP(-Infinity) = 0.0 , ATAN(+Infinity) = Pi/2 , and similar examples be regarded as exceptional. If all goes well, infinite intermediate results will turn quietly into correct finite final results that way. If all does not go well, Infinity will turn into NaN and signal INVALID. Unlike integer division by zero, for which no integer infinity nor NaN has been provided, floating-point division by zero poses no danger provided subsequent INVALID operations, if any, are not ignored; in that case disabling the trap for DIVIDE by ZERO by invoking " ctrl87 4 4 " is quite safe.

Exception: OVERFLOW. *
This happens after an attempt to compute a finite result that would lie beyond the finite range of the floating-point format for which it is destined. The default specified in IEEE 754 is to approximate that result by an appropriately signed Infinity. Since it is approximate, OVERFLOW is also INEXACT. Often that approximation is worthless; it is almost always worthless in matrix computations, and soon turns into NaN or, worse, misleading numbers. Consequently OVERFLOW is often trapped to abort seemingly futile computation sooner rather than later.

Actually, OVERFLOW more often implies that a different computational path should be chosen than that no path leads to the desired goal. For example, if the expression x/SQRT(x*x + y*y) encounters OVERFLOW before the SQRT can be computed, it should be replaced by something like (s*x)/SQRT((s*x)*(s*x) + (s*y)*(s*y)) with a suitably chosen tiny positive Scale-Factor s . The cost of computing and applying s beforehand could be reckoned as the price paid for insurance against OVERFLOW. Is that price too high?

The biggest finite IEEE 754 Double is almost 9.0 e307 , which is so huge that OVERFLOW occurs extremely rarely if not yet rarely enough to ignore. The cost of defensive tests, branches and scaling to avert OVERFLOW seems too high a price to pay for insurance against an event that hardly ever happens. A lessened average cost will be incurred in most situations by first running without defensive scaling but with a judiciously placed test for OVERFLOW (and for severe UNDERFLOW); in the example above the test should just precede the SQRT. Only when necessary need scaling be instituted. Thus our treatment of OVERFLOW has come down to this question: how best can OVERFLOW be detected?

The ideal test for OVERFLOW tests its flag; but that flag cannot be mentioned in most programming languages for lack of a name. Next best are tests for Infinities and NaNs consequent upon OVERFLOW, but prevailing programming languages lack names for them; suitable tests have to be contrived. For example, the C predicate (z != z) is True only when z is NaN and the compiler has not " optimized " overzealously; ((1.0 e37)/(1 + fabs(z)) == 0.0) is True only when z is infinite; and (z-z != 0.0) is True only when z is NaN or infinite, the INVALID trap has been disabled, and optimization is not overzealous.

A third way to detect OVERFLOW is to enable its trap and attach a handler to it. Even if a programming language in use provides control structures for this purpose, this approach is beset by hazards. The worst is the possibility that the handler may be entered inadvertently from unanticipated places. Another hazard arises from the concurrent execution of integer and floating-point operations; by the time an OVERFLOW has been detected, data associated with it may have become inaccessible because of changes in pointers and indices. Therefore this approach works only when a copy of the data has been saved to be reprocessed by a different method than the one thwarted by OVERFLOW, and when the scope of the handler has been properly localized; note that the handler must be detached before and reattached after functions that handle their own OVERFLOWs are executed. The two costs, of saving and scoping, must be paid all the time even though OVERFLOW rarely occurs. For these reasons and more, other approaches to the OVERFLOW problem are to be preferred, but a more extensive discussion of them lies beyond the intended scope of this document.

When OVERFLOW's trap is enabled, the IEEE 754 default Infinity is not generated; instead, the results's exponent is " wrapped," which means in this case that the delivered result has an exponent too small by an amount that depends upon its format:

```
  REAL*10  in stack   ...  too small by  24576 ;  2^24576 = 1.3 E 7398
( REAL*8   in memory  ...  too small by   1536 ;   2^1536 = 2.4 E 462 )
( REAL*4   in memory  ...  too small by    192 ;    2^192 = 6.3 E 57  )
( The latter two, though required by IEEE 754, cannot be performed )
( by the ix87 without help from suitable trap-handling software.    )
```

In effect, the delivered result has been divided by a power of 2 so huge as to turn what would have overflowed into a relatively small but predictable quantity that a trap-handler can reinterpret. If there is no trap handler, computation will proceed with that smaller quantity or, in the case of FSTore instructions, without storing anything. The reason for exponent wrapping is explained after UNDERFLOW.

" Ctrl87 0 8 " enables, " ctrl 8 8 " disables the OVERFLOW trap; they are not to be invoked lightly.

Exception: UNDERFLOW. *
This happens after an attempt to approximate a nonzero result that lies closer to zero than the intended floating-point destination's "Normal" positive number nearest zero. 2.2 e-308 is that number for IEEE 754 Double. A nonzero Double result tinier than that must by default be approximated by a nearest Subnormal number, whose magnitude can run from 2.2 e-308 down to 4.9 e-324 (but with diminishing precision), or else by 0.0 when no Subnormal is nearer. IEEE 754 Single and Extended formats have different UNDERFLOW thresholds, for which see the Appendix: Representable Floating-Point Numbers.

Subnormal numbers, also called " Denormalized," allow UNDERFLOW to occur Gradually; this means that gaps between adjacent floating-point numbers do not widen suddenly as zero is passed. That is why Gradual UNDERFLOW incurs errors no worse in absolute magnitude than rounding errors among Normal numbers. No such property is enjoyed by older schemes that, lacking Subnormals, flush UNDERFLOW to zero abruptly and suffer consequently from anomalies more fundamental than afflict Gradual UNDERFLOW.

For example, the C predicates x == y and x-y == 0 are identical in the absence of OVERFLOW only if UNDERFLOW is Gradual. That is so because x-y cannot UNDERFLOW Gradually; if x-y is Subnormal then it is Exact. Without Subnormal numbers, x/y might be 0.95 and yet x-y could UNDERFLOW abruptly to 0.0 , as could happen for x and y tinier than 20 times the tiniest nonzero Normal number.

Though afflicted by fewer anomalies, Gradual UNDERFLOW is not free of them. For instance, it is possible to have x/y = 0.95 coexist with (x*z)/(y*z) = 0.5 because x*z and probably also y*z UNDERFLOWed to Subnormal numbers; without Subnormals the last quotient turns into either an ephemeral 0.0 or a persistent NaN (INVALID 0/0). Thus, UNDERFLOW cannot be ignored entirely whether Gradual or not.

UNDERFLOWs are uncommon; even if flushed to zero they rarely matter; if handled Gradually they cause harm extremely rarely. That harmful remnant have to be treated much as OVERFLOWs are, with testing and scaling, or trapping, etc.; however, the most common treatment is to ignore them and attribute whatever harm that may cause to poor taste in someone else's choice of initial data.

UNDERFLOWs resemble ants; where there is one there are quite likely many more, and they tend to come one after another. That tendency has no direct effect upon the i387 and i486, but it can severely retard computation on other implementations of IEEE 754 that have to trap to software to UNDERFLOW Gradually for lack of hardware to do it. They take extra time to Denormalize after UNDERFLOW and/or, worse, to prenormalize Subnormals before multiplication or division. Worse still is the threat of traps, whether they occur or not, to machines that cannot enable traps without disabling concurrency and pipelining; such machines are slowed also by Gradual UNDERFLOWs that do not occur!

Why should we care about such benighted machines? They pose dilemmas for developers of applications software designed to be portable (after recompilation) to those machines as well as ours. To avoid sometimes severe performance degradation by Gradual UNDERFLOW, developers will sometimes resort to simple-minded alternatives. The simplest violates IEEE 754 by flushing every UNDERFLOW to 0.0 , and computers have been sold that flush by default. (DEC ALPHA is a recent example; it has been advertised as conforming to IEEE 754 without mention of how slowly it runs with traps enabled for full conformance.) Applications designed with flushing in mind may, when run on i387s and i486s, have to enable the UNDERFLOW trap and provide a handler that flushes to zero, thereby running slower to get generally worse results! (This is what MathCAD does on PCs and on Macintoshes.) Few applications are designed with flushing in mind nowadays; since some of them might malfunction if UNDERFLOW were made Gradual instead, disabling the ix87 UNDERFLOW trap to speed them up is not always a good idea.

...... Digression on Gradual Underflow ......

To put things in perspective, here is an example of a kind that, when it appears in benchmarks, scares many people into choosing flush-to-zero rather than Gradual UNDERFLOW. To simulate the diffusion of heat through a conducting plate with edges held at fixed temperatures, a rectangular mesh is drawn on the plate and temperatures are computed only at mesh points. The finer the mesh, the more accurate is the simulation. Time is discretized too; at each time-step, temperature at every interior point is replaced by a positively weighted average of that point's temperature and those of nearest neighbors. Simulation is more accurate for smaller time-steps, which entail larger numbers of time-steps and tinier weights on neighbors; typically these weights are smaller than 1/8 , and time-steps number in the thousands.

When edge temperatures are mostly fixed at 0 , and when interior temperatures are mostly initialized to 0 , then at every time-step those nonzero temperatures next to zeros get multiplied by tiny weights as they diffuse to their neighbors. With fine meshes, large numbers of time-steps can pass before nonzero temperatures have diffused almost everywhere, and then tiny weights can get raised to large powers, so UNDERFLOWs are numerous. If UNDERFLOW is Gradual, denormalization will produce numerous subnormal numbers; they slow computation badly on a computer designed to handle subnormals slowly because the designer thought they would be rare. Flushing UNDERFLOW to zero does not slow computation on such a machine; zeros created that way may speed it up.

When this simulation figures in benchmarks that test computers' speeds, the temptation to turn slow Gradual UNDERFLOW Off and fast flush-to-zero On is more than a marketing manager can resist. Compiler vendors succumb to the same temptation; they make flush-to-zero their default. Such practices bring to mind the unfortunate accidents that occurred a century or so ago among high-pressure steam boilers whose noisy over-pressure relief valves had been tied down by attendants who wished to sleep undisturbed.

```
+----------------------------------------------------------------+
| Vast numbers of  UNDERFLOWs  usually signify that something about |
| a program or its data is strange if not wrong;  this signal should |
| not be ignored,  much less squelched by flushing  UNDERFLOW  to  0 |
+----------------------------------------------------------------+
```

What is strange about the foregoing simulation is that exactly zero temperatures occur rarely in Nature, mainly at the boundary between colder ice and warmer water. Initially increasing all temperatures by some negligible amount, say 1.0E-30 , would not alter their physical significance but it would eliminate practically all UNDERFLOWs and so render their treatment, gradual or flush-to-zero, irrelevant.

To use such atypical zero data in a benchmark is justified only if it is intended to expose how long some hardware takes to handle UNDERFLOW and subnormal numbers. Unlike many other floating-point engines, the i387 and its successors are slowed very little by subnormal numbers; we should thank Intel's engineers for that and enjoy it rather than resort to an inferior scheme which also runs slower on the ix87 .

...... End of Digression ......

When UNDERFLOW's trap is enabled, the IEEE 754 default Gradual Underflow does not occur; the results's exponent is " wrapped " instead, which means in this case that the delivered result has an exponent too big by an amount that depends upon its format:

```
  REAL*10   in stack   ...  too big by  24576 ;  2^24576 = 1.3 E 7398
( REAL*8    in memory  ...  too big by   1536 ;   2^1536 = 2.4 E 462    )
( REAL*4    in memory  ...  too big by    192 ;    2^192 = 6.3 E 57     )
( The latter two,  though required by  IEEE 754,  cannot be performed )
( by the  ix87  without help from suitable trap-handling software.     )
```

In effect, the delivered result has been multiplied by a power of 2 so huge as to turn what would have underflowed into a relatively big but predictable quantity that a trap-handler can reinterpret. If there is no trap handler, computation will proceed with that bigger quantity or, in the case of FSTore instructions, without storing anything.

Exponent wrapping provides the fastest and most accurate way to compute

```
          (a1 + b1)*(a2 + b2)*(a3 + b3)*( ... )*(aN + bN)
     Q  = ----------------------------------------------
          (c1 + d1)*(c2 + d2)*(c3 + d3)*( ... )*(cM + dM)
```

when N and M are huge and the numerator and denominator are likely to OVER/UNDERFLOW even though the value of Q would be unexceptional if it could be computed. This situation arises in certain otherwise attractive algorithms for calculating eigensystems, or Hypergeometric series, for example. What Q requires is an OVER/UNDERFLOW trap-handler that counts OVERFLOWs and UNDERFLOWs but leaves wrapped exponents unchanged during each otherwise unaltered loop that computes separately the numerator's and denominator's product of sums. The final quotient of products will have the correct significant bits but an exponent which, if wrong, can be corrected by taking counts into account. This is by far the most satisfactory way to compute Q , but for lack of suitable trap-handlers it is hardly ever exploited though it was implemented on machines as diverse as the IBM 7094 and /360, Burroughs B5500, and DEC VAX.

" Ctrl87 0 10 " enables, " ctrl 10 10 " disables the UNDERFLOW trap; they are not to be invoked lightly.

Exception: INEXACT. *
This is signaled whenever the ideal result of an arithmetic operation would not fit into its intended destination, so the result had to be altered by rounding it off to fit. The INEXACT trap is disabled and its flag ignored by almost all floating-point software. Arcane ways exist to improve the accuracy of some delicate approximate computations by exploiting this signal, but they will not be discussed here. Only exact integer computation will be considered.

The ix87 can handle integers up to 65 bits wide including sign, and converts all narrower integers to this format on the fly. In consequence, arithmetic with wide integers may go faster in floating-point than in integer registers at most 32 bits wide. Even so, an integer result can get too wide to fit exactly in floating-point, and then will be rounded off. If this rounding went unnoticed it could lead to final results that were all unaccountably multiples of 64 for lack of their last few bits. Instead, the INEXACT exception serves in lieu of an INTEGER OVERFLOW signal; it can be trapped or flagged: " Ctrl87 0 20 " enables, " ctrl 20 20 " disables INEXACT traps.

Well implemented Binary-Decimal conversion software signals INEXACT just when it is deserved, just as rational operations and square root do. However, transcendental functions like COS and X**Y may on occasion deliver exact results and yet signal INEXACT undeservedly; such a signal is very difficult to prevent.

Precisions of Rounding: *
IEEE 754 obliges only machines that compute in the Extended (long double or REAL*10) format to let programmers control the precision of rounding from a control word. This permits the ix87 to emulate the roundoff characteristics of those machines that conform to IEEE 754 but support only Single (C's float, or REAL*4) and Double (C's double, or REAL*8) but not Extended. Software developed and checked out on one of those machines can be recompiled for the ix87 and, if anomalies arouse concerns about differences in roundoff, the software can be run very nearly as if on its original host without sacrificing speed on the ix87. Conversely, software developed on the ix87 but without exlicit mention of Extended can be rerun in a way that indicates what it will do on those other machines. Precision control rounds to 24 sig. bits to emulate Single, to 53 Sig. bits to emulate Double, leaving zeros in the rest of the 64 sig. bits of the Extended format.

The emulation is imperfect. Transcendental functions are unlikely to match. Binary-Decimal conversion software should ideally be unaffected by rounding precision's setting, but ideals are not always attained. Some OVER/UNDERFLOWs that would occur on those other machines need not occur on the ix87 ; IEEE 754 allows this, perhaps unwisely, to relieve hardware implementers of details that were thought unimportant.

```
" ctrl87 300 300 "   sets precision to the default  Extended   REAL*10 ,
" ctrl87 200 300 "   sets precision to  Double Precision,      REAL*8 ,
" ctrl87   0 300 "   sets precision to  Single Precision,      REAL*4 .
```

Directions of Rounding: *
The default, reset by " ctrl87 0 C00 ", rounds every arithmetic operation to the nearest value allowed by the assigned precision of rounding. When that nearest value is ambiguous (because the exact result would be one bit wider than the precision calls for) the rounded result is the " even " one with its last bit zero. Note that rounding to the nearest 16-, 32- or 64-bit integer (FIST and FISTP) in this way takes both 1.5 and 2.5 to 2 , so the various INT, IFIX, ... conversions to integer supported by diverse languages may require something else. One of my Fortran compilers makes the following distinctions among roundings to nearest integers:

```
   IRINT, RINT, DRINT     round to nearest even as  FIST  does.
   NINT, ANINT, DNINT     round half-integers away from  0 .
   INT, AINT, DINT        truncate to integers towards  0 .
```

" Ctrl87 C00 C00 " causes subsequent arithmetic operations to be truncated, rather than rounded, to the nearest value in the direction of 0.0 . In this mode, FIST provides INT etc. This mode also resembles the way many old machines, now long gone, used to round.

" Ctrl87 400 C00 " rounds subsequent operations towards -Infinity ;
" Ctrl87 800 C00 " rounds subsequent operations towards +Infinity .
These " Directed " roundings can be used to implement Interval Arithmetic, which is a scheme that approximate every variable not by one value of unknown reliability but by two that are guaranteed to straddle the ideal value. This scheme is not so popular in the U.S.A. as it is in parts of Europe, where some people distrust computers.

Control-Word control of rounding modes allows software modules to be re-run in different rounding modes without recompilation. This cannot be done with some other computers, notably DEC ALPHA, that can set two bits in every instruction to control rounding direction at compile-time; that is a mistake. It is worsened by the designers' decision to take rounding direction from a Control-Word when the two bits are set to what would otherwise have been one of the directed roundings; had ALPHA obtained the round-to-nearest mode only from the Control-Word, their mistake could have been transformed into an advantageous feature.

All these rounding modes round to a value drawn from the set of values representable with the precision selected by rounding precision control as decribed earlier. The sets of representable values are spelled out in the Appendix that follows. The direction of rounding can also affect OVER/UNDERFLOW ; a positive quantity that would OVERFLOW to +Infinity in the default mode will turn into the format's biggest finite floating-point number when rounded towards -Infinity. And the expression " X - X " delivers +0.0 for every finite X in all rounding modes except for rounding directed towards -Infinity, for which -0.0 is delivered instead. These details are designed to make Interval Arithmetic work better.

Ideally, software that performs Binary-Decimal conversion (both ways) should respect the requested direction of rounding and the precision of the conversion's destination regardless of the Control-Word's setting of rounding precision. Algorithms that do this have been put into the public domain (Netlib) by David Gay of AT&T, but apparently few compiler writers know about it.

Appendix: Representable Floating-Point Numbers. ********************

The ix87 handles three types or Formats of floating-point numbers: Single (REAL*4), Double (REAL*8), and Extended (REAL*10). Each format has representations for NaNs, +Infinity, -Infinity, and its own set of finite real numbers all of the simple form

$$n * 2^{k+1-N}$$

with two integers n (Significand) and k (unbiased Exponent) that run throughout two intervals determined from the format thus:

N significant bits: $-2^N < n < 2^N$. Exponent: $1 - 2^K < k < 2^K$.

Format	N	K
Single:	24	7
Double:	53	10
Double-Extended:	64	14

This concise representation, unique to IEEE 754, is deceptively simple. At first sight it appears potentially ambiguous because, if n is even, dividing n by 2 (a right-shift) and then adding 1 to k makes no difference. Whenever such an ambiguity could arise it is resolved by minimizing the exponent k and thereby maximizing the magnitude of n ; this is " Normalization." IEEE 754's Normals are distinguishable from the Subnormal (Denormalized) numbers lacking or suppressed in earlier computer arithmetics; Subnormals are nonzero numbers with unnormalized significand and minimal exponent:

$$-2^{N-1} < n < 2^{N-1} \quad \text{and} \quad k = 2 - 2^K .$$

```
 Subnormal Nos. [--- Normalized Numbers ----- - - -  -  -  -  -  - ->
                |               |                               |
0-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+---+---+---+---+---+---+---+---+------
  | |   |       |               |                               |
0                K               K                               K
              2-2             3-2                             4-2
Powers of 2 :  2               2                               2
```

-+- Consecutive Positive Floating-Point Numbers -+-
****

Since the Extended format is optional in implementations of IEEE 754, most others do not offer it; it is available only on Intel's x86/x87 and Pentium, Intel's 80960 KB, and Motorola's 68040 and earlier 680x0 with 68881/2 coprocessor, and Motorola's 88110 .

Most microprocessors that support floating-point on-chip, and all that serve in prestigious workstations, support just the two REAL*4 and REAL*8 floating-point formats. In some cases the registers are all 8 bytes wide, and REAL*4 operands are converted on the fly to their REAL*8 equivalents when they are loaded into a register; in such cases, immediately rounding to REAL*4 every REAL*8 result of an operation upon such converted operands produces the same result as if the operation had been performed in the REAL*4 format all the way.

But Motorola 680x0-based Macintoshes and Intel ix86-based PCs with ix87-based (not Weitek's 1167 or 3167) floating-point behave quite differently; they perform all arithmetic operations in the Extended format, regardless of the operands' widths in memory, and round to whatever precision is called for by the setting of a control word.

Only the Extended format appears in an ix87's eight stack-registers so all numbers loaded from memory in any other format, floating-point or integer or BCD, are converted on the fly into Extended with no change in value. All arithmetic operations enjoy the Extended range and precision. Storing from a register into a narrower memory format requires rounding on the fly, and may also incur OVER/UNDERFLOW; the register's value remains unchanged if not popped off the stack. (Some compilers, based upon misconstrued ambiguities in manuals or upon ill-considered " optimization," sometimes wrongly reuse that register's value in place of what was stored from it; the correct procedure is to store and pop (FSTP) and then reload (FLD) reused values.)

The ix87 encodes its floating-point numbers in memory and registers (in ways first proposed by I.B. Goldberg in Comm. ACM (1967) 105-6) by packing three fields with integers derived from the sign, exponent and significand of a number as follows. The leading bit is the sign bit, 0 for + and 1 for - . The next K+1 bits hold a biased exponent. The last N or N-1 bits hold the significand's magnitude.

There are also three special cases necessitated by Infinities, NaNs and Subnormal numbers. Note that +0 and -0 are distinguishable and follow special rules specified by IEEE 754 even though floating-point arithmetical comparison says they are equal; there are good reasons to do this. The two zeros are distinguishable arithmetically only by either division-by-zero (producing appropriately signed infinities) or by the CopySign function in IEEE 754 / 854.

To simplify our presentation, the sign bit is assumed below to be 0 so the significand n is nonnegative.

********* Encodings of n*(2**(k+1-N)) into Binary Fields ********

Number Type	K+1 bit Exponent	Nth bit	N-1 bits of Significand
NaNs:	binary 111...111	1	binary 1xxx...xxx
SNaNs:	binary 111...111	1	nonzero binary 0xxx...xxx
Infinities:	binary 111...111	1	0
Normals:	k - 1 + 2**K	1	nonnegative n - 2**(N-1)
Subnormals:	0	0	positive n < 2**(N-1)
Zeros:	0	0	0

IEEE Single and Double have no Nth bit in their significant digit fields; it is " implicit." (The ix87's Extended has the explicit Nth bit for historical reasons; it allowed the 8087 to suppress the normalization of subnormals advantageously for certain scalar products in matrix computations, but this and other features of the 8087 were later deemed too arcane to include in IEEE 754, and have atrophied.)

The foregoing encodings are all " Lexicographically Ordered," which means that if two floating-point numbers in the same format are ordered (say x < y), then they are ordered the same way when their bits are reinterpreted as Sign-Magnitude integers. Consequently, processors do not need floating-point hardware to search, sort and window floating-point arrays quickly. (However, some processors reverse byte-order!)

Finally, as an amenity, the following table exhibits the span of each floating-point format, and its precision in " significant decimals."

****** Span and Precision of ix87 Floating-Point Formats ******

Format.	Min. Subnormal	Min. Normal	Max. Finite	Sig. Dec.
Single:	1.4 E-45	1.2 E-38	3.4 E38	6 - 9
Double:	4.9 E-324	2.2 E-308	1.8 E308	15 - 17
Extended:	3.6 E-4951	3.4 E-4932	1.2 E4932	18 - 21

The entries in the table come from the following formulas:

```
Min. Positive Subnormal:   2**(3 - 2**K - N)
Min. Positive Normal:      2**(2 - 2**K)
Max. Finite:               (1 - 1/2**N) * 2**(2**K)

Sig. Dec.,  at least:   floor( (N-1)*Log10(2) )  sig. dec.
            at most:    ceil( 1 + N*Log10(2) )   sig. dec.
```

The precision is bracketed within a range in order to characterize how accurately conversion between binary and decimal has to be implemented to conform to IEEE 754. For instance, " 6 - 9 " Sig. Dec. for Single means that, in the absence of OVER/UNDERFLOW, ...

If a decimal string with at most 6 sig. dec. is converted to Single and then converted back to the same number of sig. dec., then the final string should match the original. Also, ...

If a Single Precision floating-point number is converted to a decimal string with at least 9 sig. dec. and then converted back to Single, then the final number must match the original.

```
***************************************************************************
===========================================================================

Appendix:     How to set the  Motorola 68881/2  or  68040  Control Word
~~~~~~~~~                  on a  680x0-based  Apple Macintosh

The idea is to use the  " MacsBug "  Debugger  to alter  12 bits in the
Floating-Point Control Register  fpcr ,  thereby controlling precision,
rounding direction,  and trapping on floating-point exceptions.  ( The
Floating-Point Status Register  fpsr  can be read and set too.)

    To trap out of any program and enter the debugger,  press the
    Programmer's Interrupt  button or,  lacking that,  press
                    [Command] [Power On/Off] .

  In the debugger,  type
     tf                 ...  to display floating-point registers.
     fpcr = $XXXX       ...  to enter hex digits  XXXX  into  fpcr .
     fpsw = $YYYYYYYY   ...  to enter hex digits  YYYYYYYY  into  fpsr.
     tf                 ...  to display floating-point registers.
     g                  ...  to resume executing the trapped program.

The bits in the  fpcr  have the following effects:

  Enable traps,  rather than deliver default results:
    Branch/Set trap on unordered  - - - - - - - - - -  $80_0
    Trap to signal  NaN  operand  - - - - - - - - - -  $40_0
    Trap on  Invalid Operation  - - - - - - - - - - -  $20_0
    Trap on  Overflow - - - - - - - - - - - - - - - -  $10_0
    Trap on  Underflow  - - - - - - - - - - - - - - -  $08_0
    Trap on  Divide-by-Zero - - - - - - - - - - - - -  $04_0
    Trap on  Inexact Arithmetic  - - - - - - - - - -  $02_0
    Trap on  Inexact Decimal-Binary Conversion  - -  $01_0
  These may be combined by addition;  e.g.,  to trap on
    either  Invalid Operation  or  Overflow,  use     $30_0 .

  Set  Precision  of  Rounding   ( choose just one ) :
    Extended  ( REAL*10 )  - - - - - - - - - - - - - -  $__00
    Double    ( REAL*8 )   - - - - - - - - - - - - - -  $__80
    Single    ( REAL*4 )   - - - - - - - - - - - - - -  $__40

  Set  Direction  of  Rounding   ( choose just one ) :
    To Nearest    - - - - - - - - - - - - - - - - - -  $__00
    Toward Zero   - - - - - - - - - - - - - - - - - -  $__10
    Toward  -Infinity  - - - - - - - - - - - - - - - -  $__20
    Toward  +Infinity  - - - - - - - - - - - - - - - -  $__30

  Precision and Direction may be combined by addition.

===========================================================================
***************************************************************************
```


Bibliography. **

IEEE standards 754 and 854 for Floating-Point Arithmetic. For a readable account see the article by W. J. Cody et al. in the IEEE Magazine MICRO, Aug. 1984, pp. 84 - 100.

"What every computer scientist should know about floating-point arithmetic" D. Goldberg, pp. 5-48 in ACM Computing Surveys vol. 23 #1 (1991). Also his "Computer Arithmetic," appendix A in "Computer Architecture: A Quantitative Approach" J.L. Hennessey and D.A. Patterson (1990), Morgan Kaufmann, San Mateo CA. Surveys the basics.

"Intel Pentium Family User's Manual, Volume 3: Architecture and Programming Manual" (1994) Order no. 241430 Explains instruction set, control word, flags; gives examples.

"Programming the 80386, featuring 80386/387" John H. Crawford & Patrick P. Gelsinger (1987) Sybex, Alameda CA. Explains instruction set, control word, flags; gives examples.

"The 8087 Primer" John F. Palmer & Stephen P. Morse (1984) Wiley Press, New York NY. Mainly of historical interest now.

User's Manuals for the Motorola ...

MC 68881 and 68882 Floating-Point Coprocessors	MC68881UM/AD (1987)
MC 68040 Microprocessor	MC68040UM/AD (1989)
PowerPC 601 Microprocessor	MPC601UM/AD (1993)

Explain instruction sets, control word, flags, ...

"Apple Numerics Manual, Second Edition" (1988) Addison-Wesley, Reading, Mass. Covers Apple II and 680x0-based Macintosh floating-point; it is a pity that nothing like this has been promulgated for Intel ix87 floating-point.

"Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit" W. Kahan; ch. 7 in "The State of the Art in Numerical Analysis" ed. by M. Powell and A. Iserles (1987) Oxford. Explains how proper respect for -0 eases implementation of conformal maps of slitted domains arising in studies of flows around obstacles.

"The Effects of Underflow on Numerical Computation" J.W. Demmel, pp. 887-919 in SIAM Jl. on Scientific & Statistical Computing vol. 5 #4 (Dec. 1984). Explains advantages of gradual underflow.

"Faster Numerical Algorithms via Exception Handling" J.W. Demmel and X. Li, pp. 983-992 in IEEE Tran. on Computers vol. 43 #8 (Aug. 1994). Some computations can go rather faster if OVERFLOW is flagged than if it will be trapped.

"Database Relations with Null Values" C. Zaniolo, pp. 142-166 in Jl. Computer and System Sciences vol. 28 (1984). Tells how best to treat a NaN (he calls it "ni" for " no information ") when it turns up in a database.

```
{$R-,S-,I-,D-,T-,F-,V-,B-,N-,L+ }
{$M 1024,0,1024 }

 program ctrl87;    {  Written in  Borland's  Turbo-Pascal  }
{ CTRL87 <ctl<, msk>>  uses two 3-hex-digit parameters  ctl and msk
   to set as many as 9 bits in the  ix87  Control-Word  as follows:
          New CW  :=  (msk AND ctl) OR (NOT(msk) AND Old CW) .
   If  msk  is omitted,  0F00  is used in its place.  If both  msk
   and  ctl  are omitted,  or if either is  " ? "  or not hexadecimal
   they will be prompted from the keyboard after the display of
   DOC  below,  which explains how they affect subsequent floating-
   point arithmetic operations.  To do nothing,  [Enter]  nothing.

   To prevent mishaps,  msk  is filtered thus:  msk := msk AND 0F3D
   ================================================================
 }
 const
    n = 19 ;    {  n = current number of lines in  DOC  }
    DOC: array[1..n] of string[55] = (
       '  CTRL87 <ctl<, msk>>  sets the  ix87  Control-Word ',
       '  C-W := (msk AND ctl) OR (NOT(msk) AND C-W)  from 2',
       '  3-hex-digit parameters  ctl and msk .  C-W''s bits ',
       '  are  OR''d  to affect floating-point thus:      C-W ',
       '  TRAPS:  (default)   Disable All traps     ...  _3D ',
       '   or      Disable  trap for  INVALID OP     ...  _01 ',
       '     and  Disable  trap for  DIV by ZERO    ...  _04 ',
       '     and  Disable  trap for  OVERFLOW       ...  _08 ',
       '     and  Disable  trap for  UNDERFLOW      ...  _10 ',
       '     and  Disable  trap for  INEXACT        ...  _20 ',
       '  PRECISION:  (default) Round to  REAL*10   ...  3__ ',
       '                or else  Round to  REAL*8    ...  2__ ',
       '                or else  Round to  REAL*4    ...  0__ ',
       '  DIRECTION:  (default) Round to  Nearest   ...  0__ ',
       '                or else  Round     Down      ...  4__ ',
       '                or else  Round     Up        ...  8__ ',
       '                or else  Round to  Zero      ...  C__ ',
       '  Initial Control-Word  ctl  set by  FINIT ...  33D ',
       '  Default msk = 0F00 .  Maximal effective msk = F3D ' );
 Sctl = '  Current setting of  Control-Word  ctl    ... ' ;
 S3H =  '  Enter  3  hex digits for ' ;
 msx = $0F3D ;  { ... maximal  msk }

 var
    ctl, i, j, k, L, msk : word ;   s : string ;

  function  Wrd2Str( i : word ) :  string ;
    { ... converts word  i  to its string of  4  hex digits.}
    var  j, k : word ;  s : string[4] ;
    begin
      s := '' ;
      for  k := 0 to 3 do begin
        j := i AND $F ;
        i := i shr 4 ;
        if j > 9 then  j := j + $37
                 else  j := j + $30 ;
        s := Concat( Chr(j), s ) ;
        end ; {  k  }
      Wrd2Str := s
    end; { Wrd2Str }
```



```
  procedure  GetHex( var j, k : word;  s : string );
    begin   { converts string  s  to  4-hex-digit word  j }
       Val( ConCat('$',s), j, k ) ; { j = value of $s if  k = 0 }
       if  k > 0  then  Writeln(s, '  is not hexadecimal.') ;
       end ; { GetHex }

  begin
    inline( $9B/$D9/$3E/i/$9B ) ;  {  fstcw  i ;  old Control-Word }
    L := ParamCount ;
    if  L = 0  then  k := 1  else begin
      s := ParamStr(1) ;  { = first parameter on DOS command line }
      if  Copy(s,1,1) = '?'  then  k := 1  else  GetHex(ctl, k, s) ;
      if  k = 0  then
        if  L < 2  then  msk := $0F00
                   else  GetHex(msk, k, ParamStr(2)) ;
      end ;  { L > 0 }

    while  k > 0  do begin { Prompt for  ctl  and  msk .}
       for  j := 1 to n  do  Writeln( DOC[j] ) ;
       Writeln( Sctl, Wrd2Str( i AND msx ) ) ;
       Writeln( S3H, 'new  ctl :' ) ;
       Readln(s) ;
       if  (s = '') or (s = ' ') or (s = '  ')
              then  Exit ;  { Do nothing.}
       if Copy(s,1,1) = '?' then  k := 1  else  GetHex(ctl, k, s) ;
       if  k = 0  then  { Prompt for  msk .}
         repeat
           Writeln( S3H, ' msk  or accept  0F00 :' ) ;
           Readln(s) ;
           if  (s = '') or (s = ' ') or (s = '  ')
                                    then  msk := $0F00
                                    else  GetHex(msk, k, s) ;
          until  k = 0 ;  { Prompted for  msk .}
        L := 0 ;
       end ;  { Prompted values for  ctl  and  msk .}

    msk := msk and msx ;  { Don't change  8087 vs. 387  C-W .}
    ctl := (msk and ctl) or ((not msk) and i) ;
    inline( $9B/$D9/$2E/ctl/$9B ) ;  {  fldcw  ctl  }
    if  L = 0  then  Writeln( Sctl, Wrd2Str( ctl AND msx ) ) ;
  end.
```