Nugget or Fool's Gold? —The Modulo Operator

by Winfried Gerum

Winfried Gerum

Usually M textbooks and manuals do not devote too much time to the operators. There seems to be no need to explain the basic arithmatic operators "+", "-", "*", "/" as all but children take them for granted. Other operators are perceived as self-explanatory as well, as they are known from other programming languages, e.g., exponentiation "**", comparisons ">", "<", and the logical operators "!", "&".

The modulo operator # and its definition is not so straightforward because there is no universally accepted symbol to denote it. Young children are not introduced to this operator. Teachers of calculus and other higher mathematics rarely feel the need to explain such easy concepts as basic operators.

The standard says this about the modulo operator:

(1)
```
# produces the value of the
left operand modulo the
right argument. It is
defined for nonzero values
of the right operand as
follows:
A#B= A - (B*floor(A/B))
where  floor(x) =  the
largest  integer '>x.
```

Since this definition seems clumsy, few take the trouble to look at the details. But it is worth it to take a few minutes to do so: First, because it is important to avoid confusion with a modulo operation in other programming languages and, second, to make the best use of the M modulo operator.

Looking at various assemblers, it is quite common that there is an integer division operation that gives the result of the division as an integer in one register and the remainder in another register. So it seems quite natural that a high-level language makes an operation available resulting in a remainder. Coming from this side it seems to be a given that

(2)
```
(A "DIV" B * B + (A "MOD" B)'
) = A
```

As long as both operands are positive integers, there is consensus among various programming languages as to how that modulo operator should function. If B is zero, or if A or B are negative, there is no longer a consensus. Division is not defined with a divisor of zero. The modulo operator just inherits this error condition from the related division. Some programming languages prefer to return zero as a result of `A "MOD" 0`.

With negative values of the operands, there are two lines of thinking. One is to implement the mathematical concept of residue classes. The other one is to have the "remainder" of a division, with the idea of computing the numeric values regardless of the signs and fixing the signs after the computation. Some programming languages are mute on the definition. They call the behavior implementation-dependent. This gives implementors a choice of pursuing their own philosophy or passing the philosophy of the underlying processor.

It is of little value to leave this to an implementor's whims. So in this case it is advantageous if a language standard gives positive guidance. But M is more: It gives the "right" definition by providing the mathematical concept of residue classes. Only ALGOL (a superb but almost forgotten language) has the same definition for its "MOD" operator.

Some time ago, the MUMPS Development Committee (MDC) got a request *to fix the modulo problem.* In M, there is no modulo problem, perhaps except that some people are not aware of the consequences of the definition of the modulo operator in the language. Therefore, I include some examples of what can be done with it later in the column.

The people unfamiliar with a "modulo" operator might think that this operator is of little importance in everyday life. But cyclical things are very common in our lives indeed. Time is organized into cycles of various length. Data are stored in blocks of 512 bytes or a multiple. Bits of data are grouped into bytes. Goods are stored in packages of certain sizes. Modulo operator will be very handy in dealing with cyclical items.

In M, there is no modulo problem, perhaps except that some people are not aware of the consequences of the definition of the modulo operator in the language.

The definition of modulo implies that all the following formulas and identities hold for positive as well as negative numbers. But be careful: (2) above is **not** an identity valid for arbitrary numbers. It holds just for positive numbers, because the symmetries of "Integer Division \" are different from those of modulo!

These are examples of modulo computations in M:

(3a) `A=+15, B=+7 => A#B = 1`

(3b) `A=-15, B=+7 => A#B = 6`

(3c) `A=+15, B=-7 => A#B = -6`

(3d) `A=-15, B=-7 => A#B = -1`

The definition of modulo gives the following identities:

(A, B are real; M, N are integers)

(4) `0#M = 0                    (zero)`

(5) `(A#M) = -(-A#-M)   (symmetry)`

(6) `(A#M) = ((A+(N*M))#M)`
`(shift invariance)`

(7) `(A#M + (B#M))#M =  (A+B)#M`

(8) `(A#(M*N)#N) = (A#N)`

Because these identities hold for any real numbers, this definition of modulo is mathematically far superior to the definition used in other programming languages.

Formula (5) is of little practical importance. It just says that (3a) vs. (3d) and (3b) vs. (3c) are symmetric. (3a) and (3b) show what the modulo does. Some people take only (3a) seriously and take (3b) just for a mathematical curiosity. It is not.

The modulo operator gives the difference between a value (a number) and cyclical boundaries. The amazing thing is that the lower **and** the higher boundaries can be probed: `$H#7` gives the days since this past Thursday (and 0 if `$H` indicates Thursday). Likewise `-$H#7` gives the number of days until next Thursday (and 0 if `$H` indicates Thursday). Every M programmer knows that 1840/12/31 was a Thursday, because `$H` started there and obviously `0#7` is 0. Before that Thursday comes Wednesday. The day 1840/12/30 has a `+$H` value of -1. Take that modulo 7 and you get 6, correctly indicating Wednesday.

Often you see the modulo operator used in conjunction with the division \ to extract bits from an integer.

```
  I\64#2
```

gives Bit 6 of Integer I. It is used in conjunction with \ to split a number into its integer and its fraction (the numbers should not be negative!).

```
; does not work if X<0
W "Integer part ",X\1," fractio
n part ",X#1
; valid for any X
W "Integer part ",X\1," fractio
n part ",+("."_$P(+X,".",2))
```

Frequently # is used after a `$DATA` value:

```
$DATA(X)#2 or $DATA(X)#5 or $DA
TA(X)#10
```

to determine whether `$DATA(X)` is 1 or 11. A nice curiosity is

```
W "$D(X) is ",$P("zero^one^ten
^eleven","^",$D(X)#4+1)
```

To compute cyclical functions, again, it is very convenient to reduce all possible argument values by a simple # operation to values within one cycle:

```
  SIN(X)=SIN(X#(2*PI))
```

Before making the modulo operation, you are allowed to add or subtract any integer multiple of the modulus for the same result:

```
  (A+(N*M))#M = (A#M)
```

E.g.,

```
(SIZE+1023)#1024 is equivalent
to (SIZE-1)#1024
```

likewise

```
  (H+672411)#7 is equivalent to
  (H+5)#7
```

Random number algorithms seem to be of little concern to M programmers, as the language provides a `$RANDOM` function. The problem is that you do not have control over the `$RANDOM` algorithm. Test runs cannot be reproduced identically. Therefore, occasionally you have to code your own random function. The most popular family of random algorithms is called the linear congruential method and it uses the modulo operator:

(9) $R_{n+1} = (R_n * A + B) \# C$

With A, B, and C integers and relative prime, it yields random integers between 0 and (C-1). It is not trivial to select good values for A, B, and C. For details see Donald E. Knuth's *The Art of Computer Programming*.[1] The good old TI-59 had the following values: A=24298, B=99991, C=199017.

Typical problems of a cyclical nature are:

(1) Compute day of the week from $H format date.
(2) When was the last Sunday?
(3) When will the next Sunday be?
(4) Convert seconds to hours, minutes, seconds.
(5) How many seconds are there to the next full hour?
(6) A vending machine accepts quarters only. Calculate the vending price.
(7) Compute the number of crates needed to package items for several orders.

Solutions to the above problems in M are as follows:

Problem 1:

```
; Compute day of the week from
+$H format day:
WRITE H," is a ",$$DOW(H)
```

The formula looks very simple (see box). Note that values of H are not restricted to positive integers. The concept of $H format extends naturally to negative numbers. This function serves these negative values as well.

Trying to port this code to other programming languages with a different definition of modulo is not straightforward at all!

Problem 2:

```
; Compute next Sunday
Write "Last Sunday was on $H="
,$$LAST(+$H,3)
```

Problem 3:

```
; Compute next Sunday
Write "next Sunday is $H=",
$$NEXT(+$H,3)
```

Problem 4:

```
Write S," is ",$$SEC(S)
```

Problem 5:

```
S SEC=-$P($H,",",2)#3600
IF SEC WRITE "The bell rings in'
",SEC
ELSE WRITE "The bell rings now"
```

Problem 6:

```
S VPRICE=-RAWPRICE#.25+RAWPRICE
```

Problem 7:

```
Let ORDER be an array of
orders with order numbers 1
through N. The number of
items ordered for order#i
are stored under ORDER(i).
The shipments are packaged in
crates holding SIZE items.

Then $$PACK0($NAME(ORDER),
SIZE) or $$PACK1($NA(ORDER,
SIZE)) gives the number of
crates needed to ship these
items. Note that this is the
same problem as computing the
number of blocks needed to
store a number of files.
```

The modulo operator isn't fool's gold at all. As you have seen, its definition has been done with great care to make it another nugget in M Technology.

Winfried Gerum's column appears regularly in *M Computing*. He is president of Winner Software GmbH in Roettenbach, Germany.

Endnote

1. D.E. Knuth, *The Art of Computer Programming*, 2:3 (Reading, MA: Addison-Wesley Press, 1968).

```
MOD  ;demonstrations of the modulo operator
     Q
     ;Extract Bit N from integer I (1=LSB)
EBIT(I,N) Q I\(2**(N-1))#2
     ;Set Bit N in integer I (1=LSB)
SBIT(I,N) Q:$$EBIT(I,N) I Q I+(2**(N-1))
     ;Clear Bit N in integer I (1=LSB)
CBIT(I,N) Q:'$$EBIT(I,N) I Q I-(2**(N-1))
     ;floor function: largest integer '>X
FLOOR(X)  Q X-(X#1)
     ;ceiling function: smallest integer '<X
CEIL(X)   Q X+(-X#1)
     ;Compute day of the week
     ;H=Date in +$H-format (may be zero or negative)
     ;Returns name of the appropriate day of the week
DOW(H) QUIT $PIECE("Thurs^Fri^Satur^Sun^Mon^Tues^Wednes","^",H#7+1)_"day"
     ;Compute last . . .day
     ;H=Date in +$H-format (may be zero or negative)
     ;IDOW= DayOfWeek 0=Thursday, 1=Friday, . . . 6=Wednesday
     ;if DayOfWeek of H is IDOW, return H
LAST0(H,IDOW) Q H-(H-IDOW#7) ;or IDOW-H#-7+H without brackets
     ;Compute last . . . day
     ;H=Date in +$H-format (may be zero or negative)
     ;IDOW= DayOfWeek 0=Thursday, 1=Friday, . . . 6=Wednesday
     ;if DayOfWeek of H is IDOW, return H-7
LAST(H,IDOW) Q $$NEXT0(H,IDOW)-7
     ;Compute next . . . day
     ;H=Date in +$H-format (may be zero or negative)
     ;IDOW= DayOfWeek 0=Thursday, 1=Friday, . . . 6=Wednesday
     ;if DayOfWeek of H is IDOW, return H
```

```
NEXTO(H,IDOW) Q H+(-H+IDOW#7) ;or IDOW-H#7+H without brackets
     ;Compute next . . . day
     ;H=Date in +$H-format (may be zero or negative)
     ;IDOW= DayOfWeek 0=Thursday, 1=Friday, . . . 6=Wednesday
     ;if DayOfWeek of H is IDOW, return H+7
NEXT(H,IDOW) Q $$LASTO(H,IDOW)+7
     ;
SEC(S)     Q S\3600_" hrs "_(S#3600\60)_" min "_(S#60)_" sec"
     ;pack imtms
PACKO(ARRAY,PAKSIZE) N SIZE,TOTAL,X
     Q:'$G(PAKSIZE) ""
     S TOTAL=0,X=""
     F  S X=$O(@ARRAY@(X)) Q:X=""  S SIZE=@ARRAY@(X) D
     .S TOTAL=-SIZE#PAKSIZE+SIZE+TOTAL
     Q TOTAL\PAKSIZE
PACK1(ARRAY,PAKSIZE) N SIZE,TOTAL,X
     Q:'$G(PAKSIZE) ""
     S TOTAL=0,X=""
     F  S X=$O(@ARRAY@(X)) Q:X=""  S SIZE=@ARRAY@(X)/PAKSIZE D
     .S TOTAL=SIZE\1+(SIZE#1>0)+TOTAL
     Q TOTAL
```

Figure 1. Selected demonstrations of the modulo operator.

Correction

The February *M Computing* inadvertently omitted some characters in figures 1 and 2 of the Tips 'n' Tricks column. Here are the corrected and complete figures.

```
     ;Input params: IN=input dev, OUT=output dev, must be OPEN
CMPR(IN,OUT) NEW C,I,IO,OFS,X,Y
     SET IO=$IO
     KILL ^UTILITY($JOB)
     SET ^($JOB,0)=0 FOR C=0:1:255 SET ^(C)=C
     SET X=-1 FOR  DO  QUIT:X=-1
     .  USE IN SET Y=$JOB IF $DATA(^UTILITY(Y))
     .  IF X<0 READ *X IF $$EndOfFile() QUIT
     .  FOR I=0:1 QUIT:'$DATA(^(Y,X))  SET OFS=^(X),Y=X  READ *X QUIT:I>31  IF $$EndOfFile SET X=-1 QUIT
     .  IF X>-1,I<32,C<32766,'$DATA(^(X)) SET C=C+1,^(X)=C
     .  USE OUT WRITE *OFS\256+($TEST*128),*OFS#256
     .  IF  WRITE *X SET X=-2
     USE IO KILL ^UTILITY($JOB)
     QUIT
```

Figure 1. The compression algorithm.

```
     ;Input params: IN=input dev, OUT=output dev, must be OPEN
DCMPR(IN,OUT) NEW C,IO,OFS,X,Y,Z
     SET IO=$IO
     KILL ^UTILITY($JOB)
     SET ^($JOB,0)=0 FOR C=0:1:255 SET ^(C)=$CHAR(C)
     FOR  DO  QUIT:X<0
     .  USE IN READ *X IF $$EndOfFile() SET X=-1 QUIT
     .  READ *Y IF X>127 SET X=X-128 READ *Z
     .  USE OUT SET OFS=X*256+Y WRITE ^(OFS)
     .  IF  WRITE *Z SET C=C+1,^(C)=^(OFS)_$CHAR(Z)
     USE IO KILL ^UTILITY($JOB)
     QUIT
```

Figure 2. The decompression algorithm.