Double Precision Library V1.0

For TurboForth V1.2

by Mark Wills

Table of Contents

1 Introduction

The double-precision library presented herein adds full double-precision (32-bit) functionality to TurboForth version 1.2

The library itself consists of 52 words, and occupies (at the time of writing) 2224 bytes when all the words are loaded. A subset of words can be loaded in memory-constrained situations; all the words list their dependencies so it is possible to create custom versions on a blocks disk that only loads the desired words/functionality.

The words in the library are grouped into the following sections for discussion purposes:

- Math words
- Comparison words
- Logical words
- Programming and stack management words
- Double-precision Numeric Display words

The source code is presented at the end of this document in two forms: The first form is an easy to read, commented version of the source code, which can be entered directly into TurboForth at the command line (please note: the TurboForth assembler is required).

The second form is formatted for hosting within blocks in TurboForth for easy loading from disk. The assembly language source has been converted to CODE: words as it massively reduces the number of blocks required, thus reducing compilation time. It is of course further possible, if the user desires, to produce a binary version of the library using BSAVE in order to bypass compilation delays during loading.

2 Acknowledgements

I'd like to thank the following people for their help with the development of this library:

- Bob Carmany, for ideas, suggestions, tips and testing.
- Karsten Torbenen & Sam Castledine provided help on the theory of dividing 32-bit numbers on a 16-bit processor, something I had never previously encountered.
- Fred Kaal for 32-bit multiplication and division routines. The code used in DU* and DU/ here is based on his code, with his permission.
- Lee Stewart for bug-spotting and testing the Floored Integer Division logic.

3 Notes on Stack Signatures

Standard Forth stack signatures are used in this document to document the effects on the stack while using the double-precision words. Where d: is seen preceding a stack item, it simply means that the stack item is a double (32-bit) value, which occupies two cells on the stack. For example:

```
2SWAP ( d:x d:y – d:y d:x)
```

Indicates that **2SWAP** swaps two *double* values (each occupying two cells on the stack), x, and y, on the stack. Unsigned-double values are represented using the notation ud, e.g., ud:x ud:y etc.

4 Entering Double Numbers in TurboForth

The number parser built into TurboForth has the ability to interpret double-precision numbers. To specify a double number, either in code, or at the keyboard, simply place a period before, after, or inside the number:

```
.99
9.9
99.
```

The above three examples will place a *double-precision* (i.e. a 32-bit) value on the stack, occupying two stack cells. If you examine the stack with **.S** you will see:

```
99 0
```

As can be seen, the high word (modulo 65536) occupies the top stack cell, with the low word below it.

Another example:

```
123.456
```

This yields:

```
-7616 1
```

Note that **.S** displays *signed* numbers, so -7616 is actually 57920. (1x65536)+57920=123456

Please note: The notion of using a period to represent a double number is standard in the Forth world. This author doesn't particularly agree with it, since it can cause confusion with regard to floating point values, but in order to remain standard with F83 and Forth in general, a period is used to represent a double value. The *standard convention* is to place the period at the *end* of the number. This convention will be used throughout this document.

Note that . can be mixed with other number modifiers:

```
$.BABEFACE
```

Will place the hex value BABEFACE (baby face!) on the stack (-1,161,889,074 decimal).

```
$.-100
```

Will place the hex value -100 (-256 decimal) on the stack, and so on.

As it is with single values, the number is evaluated according to the value in BASE.

5 Math Words

The following words are discussed in this section:

- **D1+ d:x -- d:x+1**
- **D1- d:x -- d:x-1**
- **D2+ d:x -- d:x+2**
- **D2- d:x -- d:x+2**
- **DNEGATE d:x -- d:-x**
- **DABS d:x -- ABS(d:x)**
- **D<< d:x s:count -- d:y**
- **D>> d:x s:count -- d:y**
- **D2/ d:x -- d:x/2**
- **D+ d:x d:y -- d:x+y**
- **D- d:x d:y -- d:x-y**
- **DU* ud:x ud:y -- ud:x*y**
- **DU/ ud:dividend ud:divisor -- ud:remainder ud:quotient**
- **D* d:x d:y -- d:x*y**
- **D/ d:dividend d:divisor -- d:remainder d:quotient**

5.1 D1+

Stack Signature: **d:x – d:x+1**

D1+ adds 1 to the double value on the stack.

Example:

```
1000000. D1+ D.          (displays 1000001)
```

The above example places 1,000,000 on the stack, adds one to it, and displays the result.

See section 9.8, page 23 for a description of the word **D.**

5.2 D1-

Stack Signature: **d:x – d:x-1**

D1- subtracts 1 from the double value on the stack.

Example:

```
1000000. D1- D.          (displays 999999)
```

The above example places 1,000,000 on the stack, subtracts one from it, and displays the result.

5.3 D2+

Stack Signature: **d:x – d:x+2**

D2+ adds 2 to the double value on the stack.

Example:

```
1000000. D2+ D.          (displays 1000002)
```

The above example places 1,000,000 on the stack, adds two, and displays the result.

5.4 D2-

Stack Signature: **d:x – d:x-2**

D2- subtracts 2 from the double value on the stack.

Example:

```
1000000. D2- D.          (displays 999998)
```

The above example places 1,000,000 on the stack, subtracts two, and displays the result.

5.5 DNEGATE

Stack Signature: **d:x – d:-x**

DNEGATE negates the double value on the stack, making it negative if it is currently positive, or positive if it is currently negative.

Examples:

```
 123456. DNEGATE D.      (displays -123456)
-123456. DNEGATE D.      (displays 123456)
```

5.6 DABS

Stack Signature: **d:x – abs(d:x)**

DABS converts the double value on the stack to its absolute value. i.e. if the value is negative, it is converted to a positive value. If the value is positive the value remains unchanged.

Examples:

```
-99. DABS D.             (displays 99)
 99. DABS D.             (displays 99)
```

5.7 D<<

Stack Signature: **ud:x s:count – d:y**

D<< logically shifts the double value x on the stack left by count bits, yielding the unsigned double value y. The value of y doubles with each shift. Zeros are shifted in from the least significant bit. Note that count is a single value, not a double.

Care should be taken when shifting signed values (or rather, values that are to be interpreted as signed values) since the sign of the value (as dictated by the most significant bit) may change as a result of the shift.

Examples:

```
 100. 5 D<< D.          (displays 3200)
-100. 5 D<< D.          (displays -3200)
```

5.8 D>>

Stack Signature: **ud:x s:count – d:y**

D>> logically shifts the double value x right by count bits, yielding the double value y. The value x halves with each shift. Zeros are shifted in from the most significant bit, meaning the sign of the value could change. Note that count is a single value, not a double.

Examples:

```
 3200. 5 D>> D.         (displays 100)
-3200. 5 D>> D.         (displays 134217628)
```

Care should be taken when shifting signed values (or rather, values that are to be interpreted as signed values) since the sign of the value (as dictated by the most significant bit) may change as a result of the shift.

Examples:

```
-3200. DABS 5 D>> DEGATE D.  (displays -100)
```

5.9 D2/

Stack Signature: **d:x – d:x/2**

D2/ arithmetically shifts the double value x one bit position to the right, hence dividing the value by 2. Since the shift is an arithmetic shift the sign of the value is maintained.

Examples:

```
 123456. D2/ D.         (displays 61728)
-123456. D2/ D.         (displays -61728)
```

5.10 D+

Stack Signature: **d:x d:y – d:x+y**

D+ performs a signed addition of the double values x and y, yielding a double result.

Examples:

```
123456. 789567. D+ D.      (displays 913023)
```

5.11 D-

Stack Signature: **d:x d:y – d:x-y**

D- performs a signed subtraction of the double values y from x, yielding a double result.

Examples:

```
123456. 789567. D- D.      (displays -666111)
```

5.12 DU*

Stack Signature: u**d:x ud:y -- d:x*y**

Performs an unsigned multiplication of the double values x and y, yielding an unsigned-double result.

Examples:

```
123456. 8. DU* D.          (displays 987648)
```

5.13 DU/

Stack Signature: u**d:dividend ud:divisor -- ud:remainder ud:quotient**

Performs an unsigned division of the double divisor into the double dividend, leaving an unsigned-double quotient and unsigned-double remainder.

Examples:

```
2468123. 99. DU/           (divides 99 into 2,468,123)
D.                         (displays the quotient: 24930)
D.                         (displays the remainder: 53)
```

5.14 D*

Stack Signature: **d:x d:y -- d:x*y**

Performs a *signed* multiplication of x and y. Because x and y are treated as signed, they only have 31 bits of magnitude. The most significant bit is interpreted as the sign bit.

Examples:

2468123. 99. D* D.	(displays 244344177)
-2468123. 99. D* D.	(displays -244344177)
2468123. -99. D* D.	(displays -244344177)
-2468123. -99. D* D.	(displays 244344177)

5.15 D/

Stack Signature: **d:dividend d:divisor – d:remainder d:quotient**

Performs a signed division, dividing the divisor into the dividend, returning a double quotient and double remainder.

Examples:

244344178. 99. D/	(divides 244344177 by 99)
D.	(displays the quotient: 2468123)
D.	(displays the remainder:1)

6 Comparison Words

The following words are discussed in this section:

- **D= d:x d:y -- true|false**
- **D0= d:x -- true|false**
- **D< d:x d:y -- true|false**
- **DU< ud:x ud:y -- true|false**
- **D> d:x d:y -- true|false**
- **DU> ud:x ud:y -- true|false**
- **DMAX d:x d:y -- d:x|d:y**
- **DMIN d:x d:y -- d:x|d:y**

6.1 D=

Stack Signature: **d:x d:y – true|false**

Returns TRUE if the double values x and y are the same, otherwise returns FALSE.

Examples:

```
123456. 123456. D= .          (displays -1 (true))
123456. 654321. D= .          (displays 0 (false))
```

6.2 D0=

Stack Signature: **d:x – true|false**

Returns TRUE if the double value x is equal to 0, otherwise returns FALSE.

Examples:

```
123456. D0= .                 (displays 0 (false))
     0. D0= .                 (displays -1 (true))
```

6.3 D<

Stack Signature: **d:x d:y – true|false**

Returns TRUE if the double value x is arithmetically less than the double value y, otherwise returns FALSE.

Examples:

```
123456. 100. D< .             (displays 0 (false))
   100. -99. D< .             (displays 0 (false))
   100. 101. D< .             (displays -1 (true))
```

6.4 DU<

Stack Signature: u**d:x ud:y – true|false**

Returns TRUE if the double value x is logically less than the double value y (an unsigned comparison), otherwise returns FALSE.

Examples:

```
123456. 100. DU< .          (displays 0 (false))
   100. -99. DU< .          (displays -1 (true))
   100. 101. DU< .          (displays -1 (true))
```

6.5 D>

Stack Signature: **d:x d:y – true|false**

Returns TRUE if the double value x is arithmetically greater than the double value y, otherwise returns FALSE.

Examples:

```
123456. 100. D> .           (displays -1 (true))
   100. -99. D> .           (displays -1 (true))
   100. 101. D> .           (displays 0 (false))
```

6.6 DU>

Stack Signature:

Stack Signature: u**d:x ud:y – true|false**

Returns TRUE if the double value x is logically greater than the double value y (an unsigned comparison), otherwise returns FALSE.

Examples:

```
123456. 100. DU> .          (displays -1 (true))
   100. -99. DU> .          (displays 0 (false))
   100. 101. DU> .          (displays 0 (false))
```

6.7 DMAX

Stack Signature: **d:x d:y – d:x|d:y**

Arithmetically compares the double values x and y and drops the lower value, retaining the higher value.

Examples:

```
10. -10. DMAX D.          (displays 10)
```

6.8 DMIN

Stack Signature: **d:x d:y – d:x|d:y**

Arithmetically compares the double values x and y and drops the higher value, retaining the lower value.

Examples:

```
10. -10. DMIN D.          (displays -10)
```

7 Logical Words

The following words are discussed in this section:

- **DNOT d:x – ~d:x**
- **2OR d:x d:y – d:(x|y)**
- **2AND d:x d:y – d:(x&y)**
- **2XOR d:x d:y – d:x XOR y**

7.1 DNOT

Stack Signature: **d:x -- ~d:x**

DNOT performs a bitwise inversion of each bit in the double (32-bit) value x. i.e. all 1 bits are set to 0, and all 0 bits are set to 1.

Examples:

```
$7EDC1852. DNOT D$.          (displays: 8123E7AD)
```

See section 10.5.12, page 48 for a description of the word **D$.**

7.2 2OR

Stack Signature: **d:x d:y – d:(x|y)**

Logically ORs the 32-bit values x and y.

Examples:

```
$7EDC1852. $994A107E. 2OR D$.      (displays FFDE187E)
```

7.3 2AND

Stack Signature: **d:x d:y – d:(x&y)**

Logically ANDs the 32-bit values x and y.

Examples:

```
$7EDC1852. $994A107E. 2AND D$.     (displays 18481052)
```

7.4 2XOR

Stack Signature: **d:x d:y -- d:(x XOR y)**

Logically Exclusive-ORs the 32-bit values x and y.

Examples:

```
$7EDC1852. $994A107E. 2XOR D$.     (displays E796082C)
```

8 Programming and Stack Management Words

The following words are discussed in this section:

- **2VARIABLE -- address**
- **2! d:x addr --**
- **2@ addr -- d:x**
- **2CONSTANT -- d:value**
- **2LITERAL d:x --**
- **2, d:x --**
- **2SWAP d:x d:y -- d:y d:x**
- **2OVER d:x d:y -- d:x d:y d:x**
- **2NIP d:x d:y -- d:y**
- **2ROT d:x d:y d:z -- d:y d:z d:x**
- **-2ROT d:x d:y d:z -- d:z d:x d:y**
- **S>D s:x -- d:x**

8.1 2VARIABLE

Stack Signature: – **address**

2VARIABLE is used to reserve space for a double-precision (32-bit) variable. It is used in conjunction with **2!** and **2@**.

Examples:

```
2VARIABLE AMOUNT
```

The above reserves 2 cells (32 bits) of memory. When AMOUNT is referenced, the address of the first cell is returned (the 'low cell' address).

8.2 2!

Stack Signature: **d:x addr –**

2! (pronounced “two store” or “double-store”) stores a double-precision value into a variable previously declared with **2VARIABLE**.

Examples:

```
2VARIABLE AMOUNT
1667488. AMOUNT 2!
```

The above stores the value 1,667,488 into the double-precision variable AMOUNT.

8.3 2@

Stack Signature: **address – d:x**

2@ (pronounced “2 fetch” or “double-fetch”) returns a double-precision value previously stored in a double-precision variable with **2!**.

Examples:

```
2VARIABLE AMOUNT
1234567. AMOUNT 2!
AMOUNT 2@ D.                    (displays 1234567)
```

The above example declares a double-precision variable called AMOUNT. The value 1,234,567 is then stored in the variable with **2!**. Finally, the value is retrieved from the variable using **2@** and displayed with **D.**.

As can be seen (and as indicated by the stack signature) AMOUNT returns its address (see **2VARIABLE)** which is used by **2@** to return the actual double value stored at that address.

8.4 2CONSTANT

Stack Signature: – **value**

2CONSTANT is used to declare and store a double-precision constant value (a value that does not change).

Example:

```
1234567. 2CONSTANT AREA
AREA D.                         (displays 1234567)
```

The above example places the double-precision value 1,234,567 on the stack then calls **2CONSTANT**. **2CONSTANT** creates a constant in the dictionary with the name AREA and stores the value 1,234,567 into it.

The constant is subsequently referenced by simply referring to its name, leaving its value on the stack rather than its address.

8.5 2LITERAL

Stack Signature: **d:x –**

2LITERAL is used within a colon definition *during compilation* to compile a value on the stack *into* to the colon definition as a literal, such that, at *run-time*, the value is pushed onto the stack. In other words, it is used to encode a value on the stack at compile time (while the colon definition is being built).Examples:

```
1234567. 2. DU*
: TEST 2LITERAL D. ;
TEST                                    (displays 2469134)
```

Here, we perform a calculation in interpret state: We take 1,234,567 and multiply it by 2. We then start a colon definition (TEST) . **2LITERAL** is then executed. **2LITERAL** is an immediate word so it executes *during the compilation* of test; however **2LITERAL** is *not* compiled *into* TEST. The behaviour of **2LITERAL** is such that it takes a double value off the stack and compiles it into the colon definition as a literal, such that when TEST is subsequently executed, the literal value is pushed to the stack.

2LITERAL is normally used in conjunction with **[** and **]** to perform a calculation in interpret mode from within a colon definition and compile the result of the calculation into the colon definition, in order to avoid having to perform the calculation repeatedly at runtime.

For example:

```
: TEST 100 0 DO 1234567. 987. DU* D. LOOP ;
```

or

```
: TEST 100 0 DO [ 1234567. 987. DU* 2LITERAL ] D. LOOP ;
```

Both of the above examples display the number 1,218,517,629 in a loop 100 times. However, performing the same calculation repeatedly is wasteful.

The second example uses [to temporarily drop into interpret mode, perform the calculation, compile it into TEST as a literal, and then resumes compilation with]. Thus the second version runs faster, since the calculation was performed *once* at *compile-time* rather than 100 times at run-time.

8.6 2,

Stack Signature: **d:x –**

2, is used to compile a double-precision value on the stack directly to memory (at the address pointed to by **HERE**) as two consecutive cells. Note, the value is *not* compiled as a literal value, it is simply compiled to memory as two consecutive cells with no **LIT** instruction preceding them.

2, is used to compile lists or arrays of double precision numbers.

Example: (see next page)

```
CREATE FIBONACCI
    0. 2,     1. 2,      1. 2,      2. 2,      3. 2,     5. 2,
    8. 2,    13. 2,     21. 2,     34. 2,     55. 2,    89. 2,
  144. 2,   233. 2,    377. 2,    610. 2,    987. 2,  1597. 2,
 2584. 2,  4181. 2,   6765. 2,  10946. 2,  17711. 2, 28657. 2,
46368. 2, 75025. 2, 121393. 2, 196418. 2, 317811. 2,

: FIB? ( n – fibonacci) 1- 4 * FIBONACCI + 2@ D. ;
27 FIB?                                          (displays 121393)
```

8.7 2SWAP

Stack Signature: **d:x d:y -- d:y d:x**

2SWAP swaps the double values x and y on the stack.

Examples:

```
12345678. 87654321. 2SWAP D. D.      (displays 12345678 87654321)
```

8.8 2OVER

Stack Signature: **d:x d:y -- d:x d:y d:x**

2OVER places a copy of the double value x (which is underneath y) at the top of the stack.

Example:

```
12345678.
87654321.
2OVER .DS                            (displays 12345678  87654321  12345678)
```

Note: The word **.DS** is used to non-destructively display the contents of the stack as double-precision numbers. See section 9.13, page 24 for more information.

8.9 2NIP

Stack Signature: **d:x d:y -- d:y**

2NIP removes the double value x from the stack, moving y down on the stack to take its place.

Examples:

```
12345678.
87654321.
2NIP .DS                             (displays 87654321)
```

8.10 2ROT

Stack Signature: **d:x d:y d:z -- d:y d:z d:x**

Rotates the top 3 double stack values to the left. The bottom of the 3 values (prior to the instruction being executed) is placed at the top of the stack.

Examples:

```
1. 2. 3.  2ROT  .DS          (displays 2 3 1)
```

8.11 -2ROT

Stack Signature: **d:x d:y d:z -- d:z d:x d:y**

Rotates the top 3 double values on the stack to the right. The top of the stack (prior to the instruction being executed) is placed at the bottom.

```
1. 2. 3. -2ROT .DS           (displays 3 1 2)
```

8.12 S>D

Stack Signature: **s:x -- d:x**

Converts a signed single value on the stack to a signed double value, by adding a 0 in the high order cell (for positive values) or a -1 for negative values.

Examples:

```
21499 S>D .S                 (displays 21499 0)
-21499 S>D .S                (displays -24199 -1)
```

This can be confirmed with **D.** :

```
24199 S>D D.                 (displays 24199)
-24199 S>D D.                (displays -24199)
```

Note: Unsigned single values can be converted to a double simply by pushing a 0 on top of the value. For example, to convert the number 41999 to an unsigned-double:

```
41999 0 D.                   (displays 41999)
```

For clarity, this could be added as a colon definition:

```
: US>UD ( us – ud) 0 ;
```

Here “US>UD” means “unsigned-single to unsigned-double”. The stack signature shows an unsigned-single value on the top of the stack before conversion, and an unsigned-double value after the conversion.

9 Double-Precision Numeric Display Words

A number of words are included to permit the display of double precision numbers. Standard Forth Pictured Numeric Output (PNO) using the words **<#** **#** and **#>** is included. Indeed, many of the other numeric display words are built using the PNO words.

The following words are discussed in this section:

- **<#** **--**
- **#** **ud:x -- ud:y**
- **#S** **ud:x -- d:0**
- **#>** **d:x -- addr len**
- **HOLD** **ascii --**
- **INS:** **--**
- **SIGN** **n --**
- **D.** **d:x --**
- **UD.** **ud:x --**
- **D.R** **d:x s:w --**
- **UD.R** **ud:x s:w --**
- **D$.** **d:x --**
- **.DS** **--**

9.1 <#

Stack signature: **--**

Begins Pictured Numeric Output (PNO) conversion. PNO is a method of formatting numbers for display on the screen, using # symbols to represent digits. PNO actually converts the number to be displayed to a string, allowing the opportunity to insert characters into the character stream as conversion progresses (for example, commas, to separate hundreds and thousands, etc.). See **#** and **#>** for more information.

<# begins the PNO conversion process, it initialises the PNO buffer (referred to as `_PNOB` in the code), the PNO buffer pointer (`_PNOBP`) and the PNO length indicator (`_PNOLEN`). Therefore, if importing the PNO words into your own system you must observe the above dependencies.

`_PNOB _PNOBP and _PNOLEN` are declared as follows:

```
CREATE _PNOB 34 CHARS ALLOT  \ pno buffer
0 VALUE _PNOLEN              \ pno length
_PNOB 33 + VALUE _PNOBP      \ pno current position pointer
```

To begin PNO conversion, an *unsigned-double* value must be on the stack (note, however, that **<#** does not affect the value on the stack, hence its stack signature is empty).

For example:

```
12345678. <#
```

At this point, the double value 12,345,678 is on the stack, and the PNO buffers and pointers etc. are initialised to commence PNO conversion. The *actual* conversion is performed using the words **#** and **#S**. Please see the following sections for more information.

9.2

Stack Signature: u**d:x – ud:y**

The word # takes a single digit from the unsigned-double number on the stack (it divides the number by the current number base, as determined by **BASE**) and places this digit into the PNO buffer for display later.

For example:

```
DECIMAL 123. <# # # # #> TYPE
```

Taking the above example one 'word' at a time:

- First, the double value 123 is placed on the stack.
- **<#** is executed, which prepares the PNO buffer to receive data.
- **#** is executed. This divides the number on the stack (123) by **BASE**, which has been set to 10 in the above example by means of **DECIMAL**. The division of 123 by 10 yields a quotient of 12 and a remainder of 3. The remainder (3) is placed in the PNO buffer in ASCII form, and the quotient (12) replaces the value on the stack.
- **#** is again executed. This divides the number on the stack (12) by **BASE**. The division of 12 by 10 yields a quotient of 1 and a remainder of 2. The remainder (2) is placed in the PNO buffer in ASCII form, and the quotient (1) replaces the value on the stack.
- **#** is again executed. This divides the number on the stack (1) by **BASE**. The division of 1 by 10 yields a quotient of 0 and a remainder of 1. The remainder (1) is placed in the PNO buffer in ASCII form, and the quotient (0) replaces the value on the stack.
- **#>** is then executed. **#>** removes the value from the stack, and replaces it with the address and length of the string that has been constructed in the PNO buffer. This address/length pair can be fed directly to **TYPE** to display the number. At this point, PNO conversion is complete.
- **TYPE** is executed, which simply uses the data left on the stack by **#>** to display the string inside the PNO buffer.

Upon completion of PNO conversion the PNO buffer contains the ASCII characters:

1 2 3

And _PNOLEN contains the length.

Since **#** performs division, taking the remainder and adding it to the PNO buffer, *the order of the conversion proceeds from least significant digit to most significant digit*. However, to make things simple for the user, the PNO buffer is populated 'backwards' meaning that the string is actually

stored 'forwards' in the PNO buffer, which is why words such as TYPE can be used to display it.

Additional features exist to add characters to the PNO string in order to format the number as desired. See **HOLD** and **INS:** for more information.

9.3 #>

Stack signature: **d:x – addr len**

#> ends pictured numeric output. The double value that has/was been converted is removed from the stack and is replaced with the address and length of the string that has been built in the PNO buffer. This address/length pair is suitable for use with **TYPE** for displaying the string in the PNO buffer.

9.4 #S

Stack Signature: u**d:x – d:0**

The word **#S** converts all *remaining* digits until the value on the stack is 0 (i.e. there are no more digits to convert).

For example:

```
1234. <# #S #> TYPE                    (displays 1234)
```

Here, **#S** performs the same division technique as **#**, looping until the value on the stack is 0.

It is possible to partially convert a number yourself manually, and then pass the remaining digits to **#S** to perform the rest of the conversion for you. This is particularly useful if you don't know the length of the number. For example:

```
123456. <# # # #S #> TYPE              (displays 123456)
```

Here, the end result is the same: The example simply displays 123456, however, the first two digits (6 and 5) were converted manually, and the remaining digits (4321) were converted by **#S** and added to the PNO buffer automatically.

The following example formats a number by separating 100's from 1000's using a comma (see **HOLD** for more information on inserting characters into the PNO output):

```
123456. <# # # # ASCII , HOLD #S #> TYPE      (displays 123,456)
```

Here, we manually processed the first three digits, then we manually inserted a comma character into the PNO stream using **HOLD** (see section 9.5). Finally, we let **#S** convert the rest of the number for us.

9.5 HOLD

Stack Signature: **ascii –**

HOLD inserts the ASCII value on the stack directly into the PNO buffer at the current PNO buffer position. Normally the word **ASCII** or **CHAR** is used to obtain the ASCII value of the character immediately following it.

Example:

```
123456. <# # #  ASCII . HOLD  #S  ASCII $ HOLD #> TYPE
```

(displays $1234.56)

In this example, we use PNO to format a double number into dollars and cents by simply adding the appropriate characters into the PNO string in the correct places. The first two digits of the value are converted manually using **#**. Then a decimal place is added using **ASCII** to obtain the ASCII value of the period character, and **HOLD** to physically insert it into the PNO buffer. **#S** is then used to convert the rest of the number to PNO format. Finally, a dollar sign is added to the PNO string.

It is, of course, possible to use this as a basis for displaying any double value in a numeric format:

```
: .$ <# # #  ASCII . HOLD  #S  ASCII $ HOLD #> TYPE ;
```

The above definition will display any double value as a correctly formatted currency value in dollars and cents:

Example:

```
9984751. .$
```
(displays $99847.51)

```
      1. .$
```
(displays $0.01)

It should be noted that it is perfectly possible to mix normal Forth code within <# … #> blocks. Indeed, all the words introduced such as **# #S HOLD** etc. are normal Forth colon definitions. Looping, and IF...THEN constructs are all allowed. For example, the following code will correctly display large numeric values, correctly formatted with commas, regardless of the size of the number:

```
: .$ ( ud:value – ) <# # # ASCII . HOLD  BEGIN 2DUP 999.  D>
WHILE # # #  ASCII , HOLD REPEAT  #S  ASCII $ HOLD #> TYPE ;
```

```
   123456. .$
```
(displays $1,234.56)

```
  1234567. .$
```
(displays $12,345,67)

```
 12345678. .$
```
(displays $123,456.78)

```
123456789. .$
```
(displays $1,234,567.89)

Using the above, it is simple to perform fast and accurate math using currency. Floating point is not required; indeed floating point could introduce inaccuracies when dealing with currency values (especially if any sort of division is performed). Since we are dealing with double precision numbers, which can range from -2,147,483,648 to 2,147,483,647 (signed) or 0 to 4,294,967,295 (unsigned) we have plenty of magnitude available for most general purpose applications.

Therefore, when dealing with currency we can simply multiply the value by 100 and represent it as a double-precision value:

$100.41	becomes	**10041.**
$987.64	becomes	**98764.**

Or, since the period can appear anywhere in the number (see section 4, Entering Double Numbers in TurboForth) one could also enter:

100.41

and

987.64

This means the standard double-precision math words such as **D+ D-** etc. can be used, maintaining perfect numeric accuracy, and .$ can be used to correctly display a value in dollars and cents.

For example:

$109.23 + $641.79 becomes **109.23 641.79 D+ .$**

Which correctly displays $751.02

9.6 INS:

Stack Signature: **–**

The word **INS:** (short for "insert") is not a standard double-precision word, it was added by this author as a convenience to make code both shorter and easier to read. **INS:** behaves the same as **HOLD** – it inserts an ASCII character into the PNO buffer; however, it removes the need for the words **ASCII** or **CHAR**, thus making the code easier to read. For example, the .$ example above can be re-written using INS: as follows:

```
: .$ ( ud:value – ) <# # # INS: .  BEGIN 2DUP .1000 D>
WHILE # # #  INS: ,  REPEAT  #S  INS: $  #> TYPE ;
```

Which (in this author's opinion) makes the code much more readable. Internally, **INS:** simply uses **ASCII** and **HOLD**.

9.7 SIGN

Stack Signature: **n --**

SIGN is used to display the sign of a number. It simply adds a minus sign into the PNO buffer at the current PNO position *if* the number is negative (i.e. if its most significant bit is on). If the number is positive (MSB=0), **SIGN** performs no action.

Note that SIGN takes a *single* signed value as its input. When using <# … #> to display a signed-double value, one will normally copy the high-order word of the double value, and store it somewhere (e.g. in a variable or on the return stack) to feed to SIGN later.

For example:

```
: TEST  TUCK DABS <# #S ROT SIGN #> TYPE SPACE ;
-12345678. TEST                 (displays -12345678)
 12345678. TEST                 (displays 12345678)
```

The above code first TUCKs the high order cell of the double value (which contains the sign of the number in the most significant bit) and stores it on the data stack, underneath the double value. The double value is then forced positive using **DABS**. Next, all digits of the number are added to the PNO buffer via **#S**, leaving the address and length of the resultant string on the stack (with the saved high-order value underneath it). The previously saved high-order cell is then ROTated into position, and fed to **SIGN**. If the high-order cell is negative, **SIGN** will insert a minus symbol (-) into the PNO buffer, otherwise it will take no action.

The .$ example code shown in sections 9.5 and 9.6 is flawed, since it fails to deal with the possibility that a number could be negative.

For example:

$199.99 - $748.56 = $-548.57

However, if you try this using the current definition of .$ as follows:

```
199.99 748.56 D- .$
```

Will give $42,949,124.39, which is not what we expect. This is the result of using the word #S, which expects to operate on *unsigned*-double numbers.

The correct way to deal with this is to store the high-order cell, so that we can feed it to **SIGN** later on, make the number positive, display it as a *positive* number, and *then* add the sign.

The best way to convert a negative value to positive (for our purposes) is to use **DABS** (see section 5.6, page 4), since **DABS** will take no action if the number is already positive.

Thus, .$ can be re-defined as follows:

```
: .$ ( d:value - ) DUP >R DABS <# # # INS: .  BEGIN 2DUP
  999. D>     WHILE # # #  INS: ,  REPEAT  #S  R> SIGN  INS:
  $ #> TYPE ;
```

Now, when we try:

```
199.99  748.56 D- .$
```

We get the correct output of $-548.57

It's important to understand what happened here. When **<#** was called, the number on the stack was negative. The high-order cell, containing the sign, was saved to the return stack, and the number forced positive with **DABS** (hence -54857 becomes 54857). When it is time to display the sign, it is retrieved from the return stack, and fed to **SIGN**.

Caution: *When storing data on the return stack, it is crucial that the return stack be restored before the colon definition exits. Additionally, the return stack should not be used to hold data inside of a loop. An alternative to the return stack would be to use the word* **TUCK** *to store a copy of the high-order cell underneath the double number, as in:*

```
: TEST TUCK DABS <# #S ROT SIGN #> TYPE SPACE ;
```

9.8 D.

Stack Signature: d:x –

D. is used to display a signed double-precision value from the top of the stack. Like its single precision counterpart **.** a single space is displayed at the end of the number.

9.9 UD.

Stack Signature: u**d:x –**

UD. is used to display an unsigned double-precision value from the top of the stack. A single space is displayed at the end of the number.

9.10 D.R

Stack Signature: **d:x s:width –**

D.R is used to display a signed double-precision value right justified to 'width' characters. The number is padded on the left with space characters.

Example:

```
-123. 6 D.R                (displays **-123)
```

Where each * represents a space character.

If the number to be displayed is too large to fit within the specified width, then 'width-1' characters of the number are displayed, with a leading # symbol to indicate that characters are missing from the displayed number. Thus, where a width is too small, the displayed value still fits within the field length (to preserve screen layout in applications) and a visible warning (in the form of the # symbol) is given to warn the user that digits remain hidden. This approach is borrowed from spread-sheet applications where a leading # indicates insufficient field width.

If the specified field width is <=0 then it is replaced with 1.

9.11 UD.R

Stack Signature: u**d:x s:width –**

UD.R is used to display an unsigned double-precision value right justified to 'width' characters.

Example:

```
123. 6 UD.R                (displays ***123)
```

Where each * represents a space character.

If the number to be displayed is too large to fit within the specified width, then 'width-1' characters of the number are displayed, with a leading # symbol to indicate that characters are missing from the displayed number. See D.R (section 9.10).

9.12 D$.

Stack Signature: **d:x –**

D$. displays a double-precision value as an *unsigned* hexadecimal value.

9.13 .DS

Stack signature: –

.DS displays the contents of the stack, interpreting the contents of the stack (by default) as signed double values. .DS is non-destructive; i.e. the contents of the stack do not change.

.DS also displays "<TOP" to show which item is the top stack item.

Examples:

```
123456. 99. 41. 102938475. .DS
```

(displays 123456 99 41 102938475 <TOP)

Note that the actual routine to call to physically display the values is vectored via **_D.**. Thus the display mode can be changed by ticking the appropriate display word (e.g. D. D$. etc.) and storing the address in **_D.**.

For example, **.DS** can be modified to display the values as hex values by default by 'ticking' **D$.** and storing its execution address in **_D.** as follows:

```
' D$. TO _D.
123456. 99. 41. 102938475. .DS
```

(displays 1E240 63 29 622B76B <--TOP)

10 Source Code

The following sections list the source code that comprises the double-precision library. The source code is presented here for general interest and understanding. It also provides a source base if the reader wishes to make modifications or add extra functionality.

Note that it is not really practical to store the *assembler* source code in block format – it would require approximately 60 blocks as written. It *is* possible to use the TurboForth assembler to write assembly code in a horizontal format, similar to high level Forth, so that it will load faster, but it makes the assembler code very difficult to read.

TurboForth contains a word in its standard dictionary called **CODE:** which permits the definition of an assembly language word using numeric values only. This uses vastly less space in a block than its assembly source equivalent in horizontal format. Additionally, it compiles faster because TurboForth does not need to translate assembly language mnemonics into machine code op-codes, as with the assembler source code.

CODE: versions of all assembly language coded words are presented here. The **CODE:** versions of the assembly language words were produced with a small utility called ASM>CODE. See section 12, page 51 for more information.

Finally, a version suitable for inclusion on disk blocks is presented in section 11 (see page 49).

10.1 Math Words

10.1.1 D1+

```
ASM: D1+ ( d:x -- d:x+1)
\ adds 1 to the double number on the stack
\ Dependencies: None
    2 sp ()  inc,           \ increment low word
    oc if,                  \ carry set?
      sp **  inc,           \ if yes increment high word
    endif,
;ASM
```

Code Version

```
CODE: D1+ 05A4 0002 1701 0594 ;CODE
```

10.1.2 D1-

```
ASM: D1- ( d:x -- d:x-1)
\ subtracts 1 from the double number on the stack
\ Dependencies: None
    2 sp ()  dec,           \ decrement low word
    nc if,                  \ carry set?
      sp **  dec,           \ decrement high word if not
    endif,
;ASM
```

Code Version

```
CODE: D1- 0624 0002 1801 0614 ;CODE
```

10.1.3 D2+

```
ASM: d2+ ( d:x -- d:x+2)
\ adds 2 to the double number on the stack
\ dependencies: none
    2 sp ()  inct,          \ add 2 to low word
    oc if,                  \ carry?
      sp **  inc,           \ if yes then increment high word
    endif,
;ASM
```

Code Version

```
CODE: D2+ 05E4 0002 1701 0594 ;CODE
```

10.1.4 D2-

```
ASM: D2- ( d:x -- d:x+2)
\ subtracts 2 from the double number on the stack
\ Dependencies: None
    2 sp ()  dect,          \ subtract 2 from low word
    nc if,                  \ no carry?
      sp **  dec,           \ then decrement high word
    endif,
;ASM
```

Code Version

```
CODE: D2- 0664 0002 1801 0614 ;CODE
```

10.1.5 DNEGATE

```
ASM: DNEGATE ( d:x -- d:-x)
\ negates the double number on the stack if it is non 0
\ Dependencies: None
    *sp  r0  mov,           \ get high word
    2 sp ()  r1 mov,        \ get low word
    r0 r1  a,               \ check for zero
    ne if,                  \ if non zero then do negate...
        *sp inv,            \ invert high word
        2 sp ()  neg,       \ negate low word
        oc if,
          *sp inc,          \ if carry then increase high word
        endif,
    endif,
;ASM
```

Code Version

```
CODE: DNEGATE C014 C064 0002 A040 1305 0554 0524 0002 1701 0594 ;CODE
```

10.1.6 DABS

```
ASM: DABS ( d:x -- ABS(d:x))
\ returns the absolute value of the double number on the stack
\ Dependencies: None
    sp **  r0  mov,          \ test high word
    lt if,                   \ if <0 then do a negate
        sp **  inv,          \ invert high word
        2 sp ()  neg,        \ negate low word
        oc if,               \ if carry...
            sp **  inc,      \ ...increase high word
        endif,
    endif,
;ASM
```

Code Version

```
CODE: DABS C014 1101 1005 0554 0524 0002 1701 0594 ;CODE
```

10.1.7 D<<

```
ASM: D<< ( d:x count -- d:x)
\ logically shifts the double number on the stack left by
\ "count" bits
\ Dependencies: None
    *sp+  r0  mov,          \ pop shift-count
    r0  r0  mov,            \ test shift-count
    gt if,                  \ if shift-count>0...
        sp **  r1  mov,     \ get high word
        2 sp ()  r2  mov,   \ get low word
        r2  r6  mov,        \ copy low word
        r1 0  sla,          \ shift high word "r0" bits
        r2 0  sla,          \ shift low word "r0" bits
        r0  neg,            \ negate shift count
        r0 16  ai,          \ add 16
        r6 0  srl,          \ shift low word copy r0 bits right
        r6 r1  a,           \ add to high word
        r1  sp **  mov,     \ put high word back
        r2  2 sp ()  mov,   \ put low word back
    endif,
;ASM
```

Code Version

```
CODE: D<< C034 C000 1501 100E C054 C0A4 0002 C182 0A01 0A02 0500 0220 0010
0906 A046 C501 C902 0002 ;CODE
```

10.1.8 D>>

```
ASM: D>> ( d:x shift_count -- d:x)
\ logically shifts the double number on the stack right by
\ "count" bits
\ Dependencies: None
    *sp+  r0  mov,          \ get shift-count
    r0  r0  mov,            \ test shift-count
    gt if,                  \ if shift-count>0...
        sp **  r1  mov,     \ get high word
        2 sp ()  r2  mov,   \ low word
        r1  r6  mov,        \ copy high word
        r1  0  srl,         \ shift high word r0 bits
        r2  0  srl,         \ shift low word r0 bits
        r0  neg,            \ negate shift shift count
        r0  16  ai,         \ add 16
        r6  0  sla,         \ shift copy of high word r0 bits
        r6  r2  a,          \ add to low word
        r1  sp **  mov,     \ write high word
        r2  2 sp ()  mov,   \ write low word
    endif,
;ASM
```

Code Version

```
CODE: D>> C034 C000 1501 100E C054 C0A4 0002 C181 0901 0902 0500 0220 0010
0A06 A086 C501 C902 0002 ;CODE
```

10.1.9 D2/

```
ASM: D2/ ( d:x -- d:x/2)
\ arithmetically shifts double value x right one bit position
\ Dependencies: None
    sp **  r1  mov,          \ get high word
    r1  r0  mov,             \ copy it
    2 sp ()  r2  mov,        \ get low word
    r2  1  srl,              \ shift low word
    r1  1  sra,              \ shift high word
    r0  1  andi,             \ check lsb of high word
    ne if,                   \ if not zero...
        r2  $8000  ori,      \ ...switch on msb of low word
    endif,
    r2  2 sp ()  mov,        \ write low word
    r1  sp **  mov,          \ write high word
;ASM
```

Code Version

```
CODE: D2/ C054 C001 C0A4 0002 0912 0811 0240 0001 1302 0262 8000 C902 0002
C501 ;CODE
```

10.1.10 D+

```
ASM: D+ ( d:x d:y -- d:x+y)
\ adds the double numbers x and y leaving a double result
\ Dependencies: None
    4 sp ()  r0  mov,        \ get x high
    6 sp ()  r1  mov,        \ get x low
    *sp  r2  mov,            \ get y high
    2 sp ()  r7  mov,        \ get y low
    r2  r0  a,               \ add most significant words
    r7  r1  a,               \ add least significant words
    oc if,                   \ if a carry then...
        r0  inc,             \ ...increase msw of result
    endif,
    sp  4  ai,               \ adjust stack
    r0  *sp  mov,            \ write msw of result
    r1  2 sp ()  mov,        \ write lsw of result
;ASM
```

Code Version

```
CODE: D+ C024 0004 C064 0006 C094 C1E4 0002 A002 A047 1701 0580 0224 0004
C500 C901 0002 ;CODE
```

10.1.11 D-

```
ASM: D- ( d:x d:y -- d:x-y)
\ subtracts the double number y from x leaving a double result
\ Dependencies: None
    4 sp ()  r0  mov,       \ get x high
    6 sp ()  r1  mov,       \ get x low
    *sp  r2  mov,           \ get y high
    2 sp ()  r7  mov,       \ get y low
    r7  r1  s,              \ subtract low words
    nc if,                  \ if no carry then...
        r0  dec,            \ ...reduce x high word
    endif,
    r2  r0  s,              \ subtract high words
    sp  4  ai,              \ adjust stack
    r0  *sp  mov,           \ write msw of result
    r1  2 sp ()  mov,       \ write lsw of result
;ASM
```

Code Version

```
CODE: D- C024 0004 C064 0006 C094 C1E4 0002 6047 1801 0600 6002 0224 0004
C500 C901 0002 ;CODE
```

10.1.12 DU*

```
ASM: DU* ( d:x d:y  -- d:x*y)
\ multiplies unsigned-double values x and y leaving an unsigned-
\ double result.
    r3  r13  mov,
    r4  r14  mov,
    r5  r15  mov,       \ save

    r0  clr,
    *sp  r6  mov,       \ how(l2)
    2 sp ()  r7  mov,   \ low(l2)
    4 sp ()  r3  mov,
    ne if,              \ check how(l1)=0
        r0  inc,
    endif,
    r6  r3  mov,
    ne if,              \ check how(l2)=0
        r0  inc,
    endif,
    r0  1  ci,
    lte if,
        r0  clr,                \ overflow flag
        6 r14 ()  r1  mov,      \ low(l1)
        r1  r3  mov,            \ copy low(l1)
        r7  r1  mpy,            \ low(l2)*low(l1)
        r6  r3  mpy,            \ how(l2)*low(l1)
        r4  r1  a,
        oc if,
            r0  seto,           \ overflow
        else,
            r3  abs,
            ne if,
                r0  seto,       \ overflow
            else,
                4 r14 ()  r3  mov,      \ how(l1)
                r7  r3  mpy,            \ low(l2)*how(l1)
                r4  r1 a,
                oc if,
                    r0  seto,   \ overflow
                else,
                    r3  abs,
                    ne if,
                        r0  seto,   \ overflow
                    else,
                        r14  4  ai,           \ adjust stack
                        r1  r14 **  mov,      \ how(result)
                        r2  2 r14 ()  mov,    \ low(result)
                    endif,
                endif,
            endif,
        endif,
    else,
        \ both how(l1) and how(l2) are > 0...
        r0  seto,       \ overflow
```

```
    endif,
    r0  r0  mov,        \ check overflow flag
    ne if,
        \ set overflow
        r14  2  ai,     \ adjust stack
        r14 **  seto,
        2 r14 ()  seto, \ push -1
    endif,

    r13  r3  mov,
    r14  r4  mov,
    r15  r5  mov,       \ restore
;ASM
```

Code Version

```
CODE: DU* C343 C384 C3C5 04C0 C194 C1E4 0002 C0E4 0004 1301 0580 C0C6 1301
0580 0280 0001 151F 04C0 C06E 0006 C0C1 3847 38C6 A044 1702 0700 1014 0743
1302 0700 1010 C0EE 0004 38C7 A044 1702 0700 1009 0743 1302 0700 1005 022E
0004 C781 CB82 0002 1001 0700 C000 1305 022E 0002 071E 072E 0002 C0CD C10E
C14F ;CODE
```

10.1.13 DU/

```
ASM: DU/ ( d:dividend d:divisor -- d:remainder d:quotient)
\ divides the double divisor into the double dividend,
\ leaving a double result and double remainder
\ Dependencies: None
    R3 R13 MOV,  R4 R14 MOV,  R5 R15 MOV,  \ save registers

    R14 *+  R6  MOV,     \ DIVISOR HI
    R14 *+  R7  MOV,     \ DIVISOR LO
    R14 *+  R8  MOV,     \ DIVIDEND HIGH
    R14 *+  R9  MOV,     \ DIVIDEND LOW


    R2 CLR,  R3 CLR,     \ modulo how & low
    R4 CLR,  R5 CLR,     \ quotient how & low
    R6  R1  MOV,         \ divide by 0?
    R7  R1  SOC,
    EQ IF,
        \ div by 0. set overflow and exit
        R4 SETO,
        R5 SETO,
    ELSE,
        R1  32  LI,      \ 32 bit positions
        BEGIN,
            R4  1  SLA,
            R5  1  SLA,
            OC IF,
                R4 INC,
            ENDIF,
            R2  1  SLA,
            R3  1  SLA,
            OC IF,
                R2 INC,
            ENDIF,
            R8  1  SLA,
            OC IF,
                R3 INC,
            ENDIF,
            R9  1  SLA,
            OC IF,
                R8 INC,
            ENDIF,
            R10 CLR,
            R2  R6  CMP,
            HE IF,
                R10 SETO,
                R2  R6  CMP,
                EQ IF,
                    R3  R7  CMP,
                    LO IF,
                        R10 CLR,
                    ENDIF,
                ENDIF,
                R10  R10  MOV,
```

```
                NE IF,
                    R7  R3  S,
                    NC IF,
                        R2 DEC,
                    ENDIF,
                    R6 R2 S,
                    R5 INC,
                    OC IF,
                        R4 INC,
                    ENDIF,
                ENDIF,
            ENDIF,
            R1 DEC,
        EQ UNTIL,
    ENDIF,

    R14 DECT,
    R3  R14 **  MOV,
    R14 DECT,
    R2  R14 **  MOV,    \ result

    R14 DECT,
    R5  R14 **  MOV,
    R14 DECT,
    R4  R14 **  MOV,    \ remainder

    R13  R3  MOV,
    R14  R4  MOV,
    R15  R5  MOV,       \ restore
;ASM
```

Code Version

```
CODE: DU/ C343 C384 C3C5 C1BE C1FE C23E C27E 04C2 04C3 04C4 04C5 C046 E047
1603 0704 0705 1024 0201 0020 0A14 0A15 1701 0584 0A12 0A13 1701 0582 0A18
1701 0583 0A19 1701 0588 04CA 8182 1A0F 070A 8182 1603 81C3 1401 04CA C28A
1307 60C7 1801 0602 6086 0585 1701 0584 0601 16DE 064E C783 064E C782 064E
C785 064E C784 C0CD C10E C14F ;CODE
```

10.1.14 D*

```
: D* ( d:x d:y  -- d:x*y)
\ multiplies signed-double values x and y leaving an signed-
\ double result.
\ Dependencies: DNEGATE 2SWAP D< DU*
  DUP               \ get sign of y
  3 PICK            \ get sign of x
  XOR               \ flag. if <0 then signs are different
  >R                \ move flag to return stack
  DABS              \ negate y if negative
  2SWAP DABS        \ negate x if negative
  DU*               \ perfom the multiplication
  R>                \ get the flag
  0< IF DNEGATE THEN \ if flag <0 then negate result
;
```

10.1.15 D/

```
\ the following double variables are required for D/
2VARIABLE _dend \ store signed dividend
2VARIABLE _dsor \ store signed divisor

: D/ ( d:dividend d:divisor  -- d:remainder d:quotient)
\ divides signed-double values x and y leaving a floored
\ signed-double quotient and signed remainder.
\ Dependencies: _dend _dsor 2DUP 2! 2SWAP DABS DU/ 2@ 2XOR D<
\               DNEGATE D1- 2NIP
  2DUP _dsor 2!     \ store divisor
  2SWAP
  2DUP _dend 2!     \ store dividend
  DABS              \ force dividend positive
  2SWAP DABS        \ force divisor positive
  DU/               \ perform unsigned division
  \ check if floor rule should be applied
  _dend 2@ _dsor 2@ \ get signed dividend and divisor
  2XOR 0. D< IF     \ if signs differ then apply floor rule
    DNEGATE         \ negate quotient
    D1-             \ floor quotient
    \ compute remainder: r=(divisor*quotient)-dividend
    2NIP            \ discard original remainder
    2DUP DABS       \ DUP quotient
    _dsor 2@ DABS   \ get divisor
    DU*             \ divisor*quotient
    _dend 2@ DABS   \ get dividend
    D-              \ subtract dividend
    2SWAP
  THEN
  \ apply sign of divisor to remainder
  _dsor 2@ .0 D< IF \ negative divisor?
    2SWAP DABS DNEGATE 2SWAP \ then negate remainder
  THEN
;
```

10.2 Comparison Words

10.2.1 D=

```
ASM: D= ( d:x d:y -- true|false)
\ returns true if double values x and y are equal, otherwise
\ returns false
\ Dependencies: None
    *sp+  r1  mov,          \ yl
    *sp+  r0  mov,          \ yh
    *sp+  r7  mov,          \ xl
    *sp   r6  mov,          \ xh
    r0  r1  xor,            \ check xh & yh
    r6  r7  xor,            \ check xl & yl
    r1  r7  xor,            \ combine results
    eq if,                  \ equal?
        *sp seto,           \ set true
    else,
        *sp clr,            \ set false
    endif,
;ASM
```

Code Version

```
CODE: D= C074 C034 C1F4 C194 2840 29C6 29C1 1602 0714 1001 04D4 ;CODE
```

10.2.2 D0=

```
ASM: D0= ( d:x -- true|false)
\ returns true (-1) if the double value x is 0 else returns 0
\ Dependencies: None
    r0  clr,                \ assume false
    *sp+  r1  mov,          \ check high word
    eq if,                  \ if 0 then...
        *sp  r1  mov,       \ ...check low word
        eq if,              \ if 0 then...
            r0  seto,       \ ...load true flag
        endif,
    endif,
    r0  *sp  mov,           \ write result
;ASM
```

Code Version

```
CODE: D0= 04C0 C074 1603 C054 1601 0700 C500 ;CODE
```

10.2.3 D<

```
ASM: D< ( d:x d:y -- true|false)
\ returns true if double value x is arithmetically less than
\ double value y
\ Dependencies: None
    r0  clr,                \ assume false
    sp 6  ai,               \ point to xl
    \ check high words first
    -2 sp ()  -6 sp ()  cmp, \ compare xh & yh
    lt if,
        r0 seto,            \ yh is < xh. set true
    else,
        gt if,
            r0 clr,         \ higher. set false
        else,
            \ equal - check low words
            *sp  -4 sp ()  cmp, \ compare xl & yl
            lo if,
                r0  seto,    \ x is < y. set true
            else,
                r0  clr,     \ higher or equal. set false
            endif,
        endif,
    endif,
    r0  *sp  mov,
;ASM
```

Code Version

```
CODE: D< 04C0 0224 0006 8924 FFFE FFFA 1101 1002 0700 100A 1501 1002 04C0
1006 8914 FFFC 1402 0700 1001 04C0 C500 ;CODE
```

10.2.4 DU<

```
ASM: DU< ( d:x d:y -- true|false)
\ returns true if unsigned double value x is less than unsigned
\ double value y
\ dependencies: none
    r0 clr,                 \ assume false
    sp  6  ai,              \ point to xl
    \ check high words first
    -2 sp ()  -6 sp ()  cmp, \ compare xh & yh
    lo if,
        r0 seto,            \ xh is < yh. set true
    else,
        hi if,
            r0 clr,         \ higher. set false
        else,
            \ equal - check low words
            *sp  -4 sp ()  cmp, \ compare xl & yl
            lo if,
                r0 seto,    \ x is < y. set true
            else,
                r0 clr,
            endif,
        endif,
    endif,
    r0  *sp  mov,
;ASM
```

Code Version

```
CODE: DU< 04C0 0224 0006 8924 FFFE FFFA 1402 0700 1009 1202 04C0 1006 8914
FFFC 1402 0700 1001 04C0 C500 ;CODE
```

10.2.5 D>

```
: D> ( d:x d:y -- true|false)
\ returns true if double value x is arithmetically greater than
\ double value y
\ Dependencies: 2SWAP D<
  2SWAP D< ;
```

10.2.6 DU>

```
: DU> ( d:x d:y -- true|false)
\ returns true if unsigned double value x is greater than
\ unsigned double value y
\ Dependencies: 2SWAP DU<
  2SWAP DU< ;
```

10.2.7 DMAX

```
: DMAX ( d:x d:y -- d:x|d:y)
\ drops the arithmetically lower of the double values x & y,
\ leaving the higher value on the stack
\ Dependencies: 2OVER 2NIP
  2OVER 2OVER D< IF 2NIP ELSE 2DROP THEN ;
```

10.2.8 DMIN

```
: DMIN ( d:x d:y -- d:x|d:y)
\ drops the arithmetically greater of the double values x & y,
\ leaving the lower value on the stack
\ Dependencies: 2OVER 2NIP
  2OVER 2OVER D< IF 2DROP ELSE 2NIP THEN ;
```

10.3 Logical Words

10.3.1 DNOT

```
ASM: DNOT ( d:x -- ~d:x)
\ inverts the double number on the stack
\ Dependencies: None
    *sp inv,                \ invert high word
    2 sp () inv,           \ invert low word
;ASM
```

Code Version

```
CODE: DNOT 0554 0564 0002 ;CODE
```

10.3.2 2OR

```
ASM: 2OR ( d:x d:y -- d:(x||y))
\ logically ORs double values x and y
\ Dependencies: None
    *sp+  r1  mov,              \ yh
    *sp+  r0  mov,              \ yl
    *sp+  r7  mov,              \ xh
    *sp   r6  mov,              \ xl
    sp  dect,                   \ correct stack
    r0  r6  soc,                \ yl||xl
    r1  r7  soc,                \ yh||xh
    r7  *sp  mov,               \ high result to stack
    r6  2 sp ()  mov,           \ low result to stack
;ASM
```

Code Version

```
CODE: 2OR C074 C034 C1F4 C194 0644 E180 E1C1 C507 C906 0002 ;CODE
```

10.3.3 2AND

```
ASM: 2AND ( d:x d:y -- d:(x&&y))
\ logically ANDs double values x and y
\ Dependencies: None
    *sp+  r1  mov,              \ yh
    *sp+  r0  mov,              \ yl
    *sp+  r7  mov,              \ xh
    *sp   r6  mov,              \ xl
    sp  dect,                   \ correct stack
    r1 inv,                     \ invert xh for szc
    r0 inv,                     \ same for xl
    r0  r6  szc,                \ yl&&xl
    r1  r7  szc,                \ yh&&xh
    r7  *sp  mov,               \ high result to stack
    r6  2 sp ()  mov,           \ low result to stack
;ASM
```

Code Version

```
CODE: 2AND C074 C034 C1F4 C194 0644 0541 0540 4180 41C1 C507 C906 0002 ;CODE
```

10.3.4 2XOR

```
ASM: 2XOR ( d:x d:y -- d:(x XOR y))
\ Dependencies: None
    *sp+  r1  mov,              \ yh
    *sp+  r0  mov,              \ yl
    *sp+  r7  mov,              \ xh
    *sp   r6  mov,              \ xl
    sp  dect,                   \ correct stack
    r0  r6  xor,                \ yl||xl
    r1  r7  xor,                \ yh||xh
    r7  *sp  mov,               \ high result to stack
    r6  2 sp ()  mov,           \ low result to stack
;ASM
```

Code Version

```
CODE: 2XOR C074 C034 C1F4 C194 0644 2980 29C1 C507 C906 0002 ;CODE
```

10.4 Programming and Stack Management Words

10.4.1 2VARIABLE

```
: 2VARIABLE ( -- address)
\ creates a double-cell (32-bit) variable
\ Dependencies: None
  CREATE 0. , , ;
```

10.4.2 2!

```
ASM: 2! ( d:x addr --)
\ stores the double value x at address addr & addr+1
\ note that the low word is stored in the lower address
\ Dependencies: None
    *sp+  r0  mov,          \ pop address
    *sp+  r0 *+  mov,       \ pop and write high word
    *sp+  r0 *+  mov,       \ pop and write low word
;ASM
```

Code Version

```
CODE: 2! C034 CC34 CC34 ;CODE
```

10.4.3 2@

```
ASM: 2@ ( addr -- d:x)
\ fetches a double value from addr
\ Dependencies: None
    *sp  r1  mov,           \ get address
    2 r1 ()  *sp  mov,      \ get low word
    sp  dect,               \ new stack entry
    r1 **  *sp  mov,        \ get high word
;ASM
```

Code Version

```
CODE: 2@ C054 C521 0002 0644 C511 ;CODE
```

10.4.4 2CONSTANT

```
: 2CONSTANT ( -- value)
\ creates a double-cell (32-bit) constant
\ Dependencies: 2@
  CREATE  2,  DOES> 2@ ;
```

10.4.5 2LITERAL

```
: 2LITERAL ( d:x -- )
\ compiles the double value x as a literal
\ Dependencies: None
  SWAP          \ xh xl
  COMPILE LIT , \ xl
  COMPILE LIT , \ xh
; IMMEDIATE
```

10.4.6 2,

```
: 2, ( d:x -- )
\ compiles the double value x to memory
\ Dependencies: None
  , \ xh
  , \ xl
;
```

10.4.7 2SWAP

```
ASM: 2SWAP ( d:x d:y -- d:y d:x)
\ swaps the double values x and y on the stack
\ Dependencies: None
    2 sp ()  r0  mov,       \ save y lo
    6 sp ()  2 sp ()  mov,  \ move x lo
    r0  6 sp ()  mov,       \ move y lo
    *sp  r0  mov,           \ save y hi
    4 sp ()  *sp  mov,      \ move x hi
    r0  4 sp ()  mov,       \ move y hi
;ASM
```

Code Version

```
CODE: 2SWAP C024 0002 C924 0006 0002 C900 0006 C014 C524 0004 C900 0004
;CODE
```

10.4.8 2OVER

```
ASM: 2OVER ( d:x d:y -- d:x d:y d:x)
\ copies the double value x to the top of the stack
\ Dependencies: None
    6 sp ()  -2 sp ()  mov, \ move x lo
    4 sp ()  -4 sp ()  mov, \ move x hi
    sp -4  ai,              \ adjust stack pointer
;ASM
```

Code Version

```
CODE: 2OVER C924 0006 FFFE C924 0004 FFFC 0224 FFFC ;CODE
```

10.4.9 2NIP

```
ASM: 2NIP ( d:x d:y -- d:y)
\ removes the double x from the stack
\ Dependencies: None
    *sp  4 sp ()  mov,      \ move y hi
    2 sp ()  6 sp ()  mov,  \ move y lo
    sp 4  ai,               \ adjust stack pointer
;ASM
```

Code Version

```
CODE: 2NIP C914 0004 C924 0002 0006 0224 0004 ;CODE
```

10.4.10 2ROT

```
ASM: 2ROT ( d:x d:y d:z -- d:y d:z d:x)
\ rotates doubles x y & z left, x goes to the top
\ Dependencies: None
    8 sp ()   r0  mov,      \ save xh
    10 sp ()  r1  mov,      \ save xl
    4 sp ()   8 sp ()  mov, \ move yh to xh position
    6 sp ()  10 sp ()  mov, \ move yl to xl position
    *sp  4 sp ()  mov,      \ move zh to yh position
    2 sp ()  6 sp ()  mov,  \ move zl to yl position
    r0  *sp  mov,           \ move xh to zh position
    r1  2 sp ()  mov,       \ move xl to zl position
;ASM
```

Code Version

```
CODE: 2ROT C024 0008 C064 000A C924 0004 0008 C924 0006 000A C914 0004 C924
0002 0006 C500 C901 0002 ;CODE
```

10.4.11 -2ROT

```
: -2ROT ( d:x d:y d:z -- d:z d:x d:y)
\ rotates doubles x y & z right, z goes to the bottom
\ Dependencies: 2ROT
  2ROT 2ROT ;
```

10.4.12 S>D

```
: S>D ( s:x -- d:x)
\ converts the single number on the stack to a double number,
\ maintaining the sign of the original single number.
\ Dependencies: None
  DUP 0< IF -1 ELSE 0 THEN ;
```

10.5 Double-Precision Numeric Display words

10.5.1 <#

```
CREATE _PNOB 36 CHARS ALLOT  \ pno buffer
0 VALUE _PNOLEN              \ pno length
_PNOB 33 + VALUE _PNOBP      \ pno current position pointer

: <# ( -- )
\ begins pictured numeric output by initialising the PNO length
\ and buffer pointers. Also sets internal SIGN variable.
\ Dependencies: _PNOLEN _PNOB _PNOBP
  0 TO _PNOLEN
  [ _PNOB 33 + LITERAL ] TO _PNOBP ;
```

10.5.2

```
: # ( d:x -- d:y)
\ converts a single digit of double value x using the current
\ value of BASE. The least significant digit of x is copied to
\ the pictured numeric output buffer. x is reduced accordingly,
\ producing y.
\ Dependencies: DU/ _PNOBP _PNOLEN DPL 2SWAP
\  DPL @ _PNOLEN = IF
\    ASCII . _PNOBP C! -1 +TO _PNOBP 1 +TO _PNOLEN
\  THEN
  BASE @ S>D \ get base as a double
  DU/        \ divide x by base
  2SWAP
  \ stack: d:quotient d:remainder
  \ convert the remainder to an ascii digit:
  OVER DUP 9 <= IF 48 ELSE 55 THEN +
  _PNOBP C!  -1 +TO _PNOBP  1 +TO _PNOLEN
  2DROP ;
```

10.5.3 #S

```
: #S ( d:x -- d:0)
\ converts the double x into digits in the pictured numeric
\ output buffer. Stops when all digits have been converted.
\ Note: 0 (double) will be left on the stack
\ Dependencies: D0=
  BEGIN # 2DUP D0= UNTIL ;
```

10.5.4 #>

```
: #> ( d:x -- addr len)
\ ends pictured numeric processing and replaces the double
\ value on the stack (which should be 0 if all digits have
\ been converted) with the address and length of the ASCII
\ version of the converted number in the pictured numeric
\ output buffer. The address and length can be used with TYPE.
\ Dependencies: _PNOB _PNOLEN
  2DROP  _PNOB 34 + _PNOLEN -  _PNOLEN ;
```

10.5.5 HOLD

```
: HOLD ( ascii --)
\ compiles the ascii character on the stack into the pictured
\ numeric output buffer
\ Dependencies: _PNOBP _PNOLEN
  _PNOBP C!  -1 +TO _PNOBP  1 +TO _PNOLEN ;
```

10.5.6 INS:

```
: INS: ( ascii --)
\ inserts ascii character immediately following it into the
\ pictured numeric output buffer. Represents a (non-standard)
\ alternative to using the rather wordy [CHAR] and HOLD.
\ Consider:
\ .1234567 <# # # # CHAR , HOLD # # # CHAR , HOLD # #>
\ and .1234567 <# # # # INS: , # # # INS: , # #>
\ Dependencies: HOLD
  [COMPILE] ASCII  COMPILE HOLD ; IMMEDIATE
```

10.5.7 SIGN

```
: SIGN ( n -- )
\ inserts a negative sign into the pictured numeric output
\ string if the number n is negative.
\ Dependencies: _SIGN HOLD
  0< IF ASCII - HOLD THEN ;
```

10.5.8 D.

```
: D. ( d:x -- )
\ displays the double value x as a signed value
\ Dependencies: DABS <# #S #> SIGN
  DUP >R DABS <# #S R> SIGN #> TYPE SPACE ;
```

10.5.9 UD.

```
: UD. ( d:x -- )
\ displays the double value x as an unsigned value
\ Dependencies: <# #S #>
  <# #S #> TYPE SPACE ;
```

10.5.10 D.R

```
: D.R ( d:x s:w -- )
\ displays the double value x as an signed value formatted to
\ w characters wide by inserting the appropriate number of
\ spaces in front of the number.
\ If the number will not fit the width, then the least
\ significant width-1 digits are displayed, with a leading #
\ sign. E.g. 1234567. 3 D.R will display #67
\ Dependencies: <# #S #> DNEGATE SIGN
    DUP 0<= IF DROP 1 THEN  \ if width <=0 then replace with 1
    DUP >R                  \ save width
    -ROT DUP >R             \ save sign
    <# DABS #S R> SIGN #>   \ convert the number
    ROT OVER -              \ calculate number of spaces
    DUP 0>= IF              \ if 0 or more spaces
        SPACES  TYPE        \ display spaces and number
        R> DROP             \ drop width
    ELSE                    \ addr len -spaces
        DROP                \ drop spaces
        ASCII # EMIT        \ display a leading # sign
        + 1+                \ point to end of number
        R@ - R> 1- TYPE     \ display the end of the number
    THEN
    SPACE
;
```

10.5.11 UD.R

```
: UD.R ( ud:x s:w -- )
\ displays the double value x as an unsigned value formatted to
\ w characters wide by inserting the appropriate number of
\ spaces.
\ If the number will not fit the width, then the least
\ significant width-1 digits are displayed, with a leading #
\ sign. E.g. 1234567. 3 UD.R will display #67
\ Dependencies: <# #S #>
    DUP 0<= IF DROP 1 THEN  \ if width <=0 then replace with 1
    DUP >R                  \ save width
    -ROT <# #S #>           \ convert the number
    ROT OVER -              \ calculate number of spaces
    DUP 0>= IF              \ if 0 or more spaces
        SPACES  TYPE        \ display spaces and number
        R> DROP             \ drop width
    ELSE                    \ addr len -spaces
        DROP                \ drop spaces
        ASCII # EMIT        \ display a leading # sign
        + 1+                \ point to end of number
        R@ - R> 1- TYPE     \ display the end of the number
    THEN
    SPACE
;
```

10.5.12 D$.

```
: D$. ( d:x -- )
\ displays the double value x in hexadecimal. The effective
\ base remains unchanged.
\ Dependencies: UD.
  BASE @ -ROT  16 BASE !  UD.  BASE ! ;
```

10.5.13 .DS

```
' D. VALUE _D. \ vector for DS.
: .DS ( -- )
\ displays the contents of the stack as double numbers (the
\ double equivalent of .S). The actual routine to call to
\ physically display the values is vectored via _D..
\ Thus the display mode can be changed by ticking the
\ appropriate display word (e.g. D. D$. etc) and storing the
\ address in _D.
\ Dependencies: _D.
  DEPTH 1 > IF
    CR ." Stack (as doubles):" CR
    DEPTH 2/ 0 DO
      S0 6 - I 4 * - 2@ _D. EXECUTE
    LOOP
    ." <TOP"
  ELSE
    ." Empty"
  THEN CR ;
```

11 Source Code – Blocks Version

```
   -----+----1----+----2----+----3----+----4----+----5----+----6---
00|CR .( Loading 32-bit library...)
01|   .(   Loading machine code...) HEX
02|CODE: 2! C034 CC34 CC34 ;CODE CODE: 2@ C054 C521 0002 0644 C511
03|;CODE CODE: 2SWAP C024 0002 C924 0006 0002 C900 0006 C014 C524 0
04|004 C900 0004 ;CODE CODE: 2OVER C924 0006 FFFE C924 0004 FFFC 02
05|24 FFFC ;CODE CODE: 2NIP C914 0004 C924 0002 0006 0224 0004 ;COD
06|E CODE: 2ROT C024 0008 C064 000A C924 0004 0008 C924 0006 000A C
07|914 0004 C924 0002 0006 C500 C901 0002 ;CODE CODE: D1+ 05A4 0002
08| 1701 0594 ;CODE CODE: D1- 0624 0002 1801 0614 ;CODE CODE: d2+ 0
09|5E4 0002 1701 0594 ;CODE CODE: D2- 0664 0002 1801 0614 ;CODE COD
10|E: DNEGATE C014 C064 0002 A040 1305 0554 0524 0002 1701 0594 ;CO
11|DE CODE: DABS C014 1101 1005 0554 0524 0002 1701 0594 ;CODE CODE
12|: D<< C034 C000 1501 100E C054 C0A4 0002 C182 0A01 0A02 0500 022
13|0 0010 0906 A046 C501 C902 0002 ;CODE CODE: D>> C034 C000 1501 1
14|00E C054 C0A4 0002 C181 0901 0902 0500 0220 0010 0A06 A086 C501
15|C902 0002 ;CODE CODE: DNOT 0554 0564 0002 ;CODE -->

   -----+----1----+----2----+----3----+----4----+----5----+----6---
00|CODE: D2/ C054 C001 C0A4 0002 0912 0811 0240 0001 1302 0262 8000
01| C902 0002 C501 ;CODE CODE: D+ C024 0004 C064 0006 C094 C1E4 000
02|2 A002 A047 1701 0580 0224 0004 C500 C901 0002 ;CODE CODE: D- C0
03|24 0004 C064 0006 C094 C1E4 0002 6047 1801 0600 6002 0224 0004 C
04|500 C901 0002 ;CODE CODE: D< 04C0 0224 0006 8924 FFFE FFFA 1101
05|1002 0700 100A 1501 1002 04C0 1006 8914 FFFC 1402 0700 1001 04C0
06| C500 ;CODE CODE: DU< 04C0 0224 0006 8924 FFFE FFFA 1402 0700 10
07|09 1202 04C0 1006 8914 FFFC 1402 0700 1001 04C0 C500 ;CODE CODE:
08| D= C074 C034 C1F4 C194 2840 29C6 29C1 1602 0714 1001 04D4 ;CODE
09| CODE: D0= 04C0 C074 1603 C054 1601 0700 C500 ;CODE CODE: 2OR C0
10|74 C034 C1F4 C194 0644 E180 E1C1 C507 C906 0002 ;CODE CODE: 2AND
11| C074 C034 C1F4 C194 0644 0541 0540 4180 41C1 C507 C906 0002 ;CO
12|DE CODE: 2XOR C074 C034 C1F4 C194 0644 2980 29C1 C507 C906 0002
13|;CODE -->
14|
15|

   -----+----1----+----2----+----3----+----4----+----5----+----6---
00|CODE: DU* C343 C384 C3C5 04C0 C194 C1E4 0002 C0E4 0004 1301 0580
01| C0C6 1301 0580 0280 0001 151F 04C0 C06E 0006 C0C1 3847 38C6 A04
02|4 1702 0700 1014 0743 1302 0700 1010 C0EE 0004 38C7 A044 1702 07
03|00 1009 0743 1302 0700 1005 022E 0004 C781 CB82 0002 1001 0700 C
04|000 1305 022E 0002 071E 072E 0002 C0CD C10E C14F ;CODE
05|CODE: DU/ C343 C384 C3C5 C1BE C1FE C23E C27E 04C2 04C3 04C4 04C5
06| C046 E047 1603 0704 0705 1024 0201 0020 0A14 0A15 1701 0584 0A1
07|2 0A13 1701 0582 0A18 1701 0583 0A19 1701 0588 04CA 8182 1A0F 07
08|0A 8182 1603 81C3 1401 04CA C28A 1307 60C7 1801 0602 6086 0585 1
09|701 0584 0601 16DE 064E C783 064E C782 064E C785 064E C784 C0CD
10|C10E C14F ;CODE  DECIMAL .(   Compliling Forth code...)
11|CREATE _PNOB 36 CHARS ALLOT  0 VALUE _PNOLEN  _PNOB 33 + VALUE _
12|PNOBP  : 2VARIABLE ( --addr) CREATE 0. , , ; : 2, ( dx--) , , ;
13|: 2LITERAL ( dx--) SWAP COMPILE LIT , COMPILE LIT , ; IMMEDIATE
14|: S>D ( sx --dx) DUP 0< IF -1 ELSE 0 THEN ; : 2CONSTANT ( --dn)
15|CREATE 2, DOES> 2@ ; -->
```

```
   -----+----1----+----2----+----3----+----4----+----5----+----6---
00|: -2ROT ( dx dy dz--dz dx dy) 2ROT 2ROT ; : D> ( dx dy--true|fal
01|se) 2SWAP D< ; : DU> ( dx dy--true|false) 2SWAP DU< ; : DMAX ( d
02|x dy--dx|dy) 2OVER 2OVER D< IF 2NIP ELSE 2DROP THEN ; : DMIN ( d
03|x dy--dx|dy) 2OVER 2OVER D< IF 2DROP ELSE 2NIP THEN ; : <# ( -)
04|0 TO _PNOLEN [ _PNOB 33 + LITERAL ] TO _PNOBP ; : # ( dx--dy) BA
05|SE @ S>D DU/ 2SWAP OVER DUP 9 <= IF 48 ELSE 55 THEN + _PNOBP C!
06|-1 +TO _PNOBP 1 +TO _PNOLEN 2DROP ; : #S ( dx--d0) BEGIN # 2DUP
07|D0= UNTIL ; : #> ( dx--addr len) 2DROP _PNOB 34 + _PNOLEN - _PNO
08|LEN ; : HOLD ( ascii--) _PNOBP C! -1 +TO _PNOBP 1 +TO _PNOLEN ;
09|: INS: ( ascii--)  [COMPILE] ASCII COMPILE HOLD ; IMMEDIATE : SI
10|GN ( n--) 0< IF ASCII - HOLD THEN ; : D. ( dx--) DUP >R DABS <#
11|#S R> SIGN #> TYPE SPACE ; : UD. ( dx--)  <# #S #> TYPE SPACE ;
12|: D.R ( dx sw--) DUP 0<= IF DROP 1 THEN DUP >R -ROT DUP >R <# DA
13|BS #S R> SIGN #> ROT OVER - DUP 0>= IF SPACES TYPE R> DROP ELSE
14|DROP ASCII # EMIT + 1+ R@ - R> 1- TYPE THEN SPACE ;
15|2VARIABLE _dend 2VARIABLE _dsor -->

   -----+----1----+----2----+----3----+----4----+----5----+----6---
00|: UD.R ( udx sw--) DUP 0<= IF DROP 1 THEN DUP >R -ROT <# #S #> R
01|OT OVER - DUP 0>= IF SPACES TYPE R> DROP ELSE DROP ASCII # EMIT
02|+ 1+ R@ - R> 1- TYPE THEN SPACE ; : D$. ( dx--) BASE @ >R 16 BAS
03|E ! UD. R> BASE ! ;  ' D. VALUE _D. : .DS ( -) DEPTH 1 > IF CR .
04|" Stack (as doubles):" CR DEPTH 2/ 0 DO S0 6 - I 4 * - 2@ _D. EX
05|ECUTE LOOP ." <TOP" ELSE ." Empty" THEN CR ;
06|: D* ( dx dy--dx*y) DUP 3 PICK XOR >R DABS 2SWAP DABS DU* R> 0<
07|IF DNEGATE THEN ;
08|: D/ ( ddend ddisor--dr dr) 2DUP _dsor 2! 2SWAP 2DUP _dend 2! DA
09|BS 2SWAP DABS DU/ _dend 2@ _dsor 2@ 2XOR 0. D< IF DNEGATE D1- 2N
10|IP 2DUP DABS _dsor 2@ DABS DU* _dend 2@ DABS D- 2SWAP THEN _dsor
11| 2@ .0 D< IF 2SWAP DABS DNEGATE 2SWAP THEN ;
12|
13|.( 32-Bit Library loaded.)
14|.( For documentation please see)
15|.( http://turboforth.net)
```

11.1 Downloadable Version

The above code is downloadable from the TurboForth website: http://turboforth.net

It is located on blocks 31 to 35 inclusive of the utilities disk. The disk is available in various downloadable formats.

12 ASM>CODE Utility

The **CODE:** equivalents of the assembly language words were produced automatically with a small utility written specifically for the development of the double-precision library.

ASM>CODE (pronounced “assembly to code”) takes a *pre-loaded* assembly language word, as assembled by the TurboForth assembler, and produces a **CODE:** equivalent word in a DV80 disk file specified by the user.

12.1 ASM>CODE syntax

The syntax for the ASM>CODE utility is very simple:

```
ASM>CODE <word-name> <file>
```

Where <word-name> is the name of a word in memory, assembled with the TurboForth assembler, and file is any valid file name. For example:

```
ASM>CODE D1+ DSK1.D1PLUS
```

Would produce a text file on DSK1 called D1PLUS that contained the following:

```
CODE: D1+
05A4 FFFE 1701 0594 ;CODE
```

Note that ASM>CODE opens the output file in append mode, so it is possible to (for example) create a blocks file containing many ASM>CODE statements (i.e. a script) and have them directed cleanly and simply to the same text file.

The source code for ASM.CODE is presented in section 12.2. Two free disk blocks are required to host the code.

12.1.1 ASM>CODE in Classic99 Emulator Environment

If you are running TurboForth in Classic99 (a free TI-99/4A emulator for Windows systems – see http://harmlesslion.com) ASM>CODE can place its output directly into the Windows clipboard, ready for instant pasting into text-files/documents on the 'Windows side', which is very convenient. Simply use CLIP as the output device.

For example:

```
ASM>CODE D1+ CLIP
```

The CODE equivalent of D1+ is now sitting in the Windows copy/paste buffer, and can be pasted into a Windows document in the normal way.

12.2 ASM>CODE Source Code

The above code is downloadable from the TurboForth website, as per section 11.1.

```
   -----+----1----+----2----+----3----+----4----+----5----+----6---
00|.( Loading ASM>CODE)
01|FORGET CFA      0 VALUE CFA     0 VALUE FNADDR  0 VALUE FNLEN
02|0 VALUE STRPOS  0 VALUE CCOUNT  FBUF: FileOut
03|: OpenFile  HERE COUNT FileOut FILE   FileOut #OPEN ;
04|: ClearHERE  HERE 1+ 64 BL FILL   64 HERE C! ;
05|: SetFileName  ClearHERE  FNADDR HERE 1+ FNLEN CMOVE ;
06|: SetLen  FNLEN 7 + HERE C! ;
07|: SetAppend  SetFileName S" DV80AS" HERE 1+ FNLEN + 1+ SWAP
08|  CMOVE SetLen OpenFile ;
09|: SetSeq  SetFileName S" DV80SO" HERE 1+ FNLEN + 1+ SWAP CMOVE
10|  SetLen OpenFile ;  : Asm? CFA @ CFA 2+ = ;
11|: SetName  ClearHERE  S" CODE: " HERE 1+ SWAP CMOVE
12|  CFA >LINK 2+ DUP @ 15 AND SWAP 2+ SWAP HERE 7 + SWAP 0 DO
13|  OVER C@ OVER C! 1+ SWAP 1+ SWAP LOOP 2DROP ;
14|: FlushLine  HERE COUNT FileOut #PUT ABORT" Can't write to file"
15|  ClearHERE  0 TO STRPOS ; -->

   -----+----1----+----2----+----3----+----4----+----5----+----6---
00|: PlaceCell  CFA @ N>S  STRPOS 5 * HERE 1+ + SWAP CMOVE
01|  1 +TO STRPOS  2 +TO CFA ;
02|: &;  S" ;CODE" STRPOS 5 * HERE 1+ + SWAP CMOVE ;
03|: ProcessWord  SetName FlushLine BASE @ >R   16 BASE !
04|  ZEROS @ >R   TRUE ZEROS !  UNSIGNED @ >R  TRUE UNSIGNED !
05|  2 +TO CFA  BEGIN  CFA @ $045C <> WHILE PlaceCell STRPOS 12 =
06|  IF FlushLine THEN REPEAT STRPOS 0> IF &; FlushLine ELSE CFA @
07|  $045C = IF &; FlushLine THEN THEN R> UNSIGNED !  R> ZEROS !
08|  R> BASE ! ; : ASM>CODE  CR  ' TO CFA   BL WORD TO FNLEN
09|  TO FNADDR  CFA IF Asm? IF SetAppend IF SetSeq
10|  ABORT" Cant open file" THEN ProcessWord FileOut #CLOSE ELSE
11|  TRUE   ABORT" Not an assembly language word" THEN ELSE TRUE
12|  ABORT" Word not found" THEN ;
13|.( ASM>CODE loaded.)
14|.( Usage: ASM>CODE <name> <file>)
15|.( E.g.: ASM>CODE EXPECT DSK1.EXPECT)
```

13 Questions, Bug Reports etc.

Any questions, bug reports, tips, hints etc. should be posted to the TurboForth mailing list, which can be found at http://tinyurl.com/turboforth