Russian Peasant Multiplication In MyForth

19Jun10 SVFIG Presentation by Bob Nash

CONTENTS

Russian Peasant Multiplication In MyForth

19Jun10 SVFIG Presentation by Bob Nash

Introduction

The code presented here is part of an application that generates a variable frequency signal and displays the frequency on an LCD. This Variable Frequency Oscillator (VFO) application uses a Direct Digital Synthesizer (DDS) chip to generate the signal, an incremental encoder to change the frequency and an 8-character LCD to display the frequency.

One problem to be solved was the accurate display of the generated frequency without having to measure it with additional hardware (e.g., a separate frequency counter). I solved the problem using multiple-precision multiplication and a “shift and add” method commonly known as the Russian Peasant algorithm.

My presentation discusses the algorithm and its coding in MyForth. See the Program Listing section for the multiplication source code.

Overview

The application uses an incremental encoder to set the VFO frequency. Encoder counts are accumulated in a 32 bit accumulator. To change the frequency, the encoder counts are periodically captured, converted to values compatible with the DDS chip registers and sent to the chip.

To display the frequency, the encoder counts must be converted to a Hertz value and displayed on an LCD. This is relatively straightforward: the accumulator counts are converted to Hertz and the result is converted to decimal digits and written to the LCD display.

The most challenging part of this process is the calculation of the Hertz value. Although a number of complicated approaches were considered, I finally decided to use the most direct method which requires multiple-precision multiplication.

Russian Peasant Multiplication In MyForth

19Jun10 SVFIG Presentation by Bob Nash

Frequency Calculation

For a fixed DDS reference frequency, the 28-bit value loaded into the DDS chip's registers is based on a Hertz/count value. This is a constant which depends on the DDS reference frequency.

Thus, the frequency to be displayed, in Hertz, can be calculated by multiplying the encoder value by a "counts per Hertz" constant. The calculation of this constant, based on the DDS frequency, is given in the DDS chip's datasheet.

The chip I used is an AD9833 from Analog Devices. The chip will generate sine, square or triangle wave signals in the megahertz range with a 25 MHz crystal and costs approximately $7. Refer to its datasheet for more information on how the 28-bit frequency control data is loaded into the chip.

The frequency calculation from the datasheet, expressed in GForth, is:

```
268435456 constant 2^28
24577966   constant freq       \ crystal frequency, 24.577 MHz (same as CPU)
                               \ 24577966/268435456 = 0.091560058 Hz/count

    : U*/MOD  >R UM* R> UM/MOD ;
    : U*/  U*/MOD NIP ;
    : (hz)  ( n1 – n2)   2^28 freq U*/ ;
    : hz  ( freq -)   (hz) hex . decimal ;
```

Frequency Changes

To provide the user with various frequency change rates, the encoder's accumulator is incremented in increments of 1, 10, 100, 1000 and 10,000. The application allows this increment to be set as a user menu option.

As noted above, the Hz/count value is approximately 0.0916, close to one tenth of a Hertz. Although the application can display to a higher resolution, the accuracy of the display is only slightly better than one tenth of a Hertz.

Display

To calculate the current frequency, the encoder count must be multiplied by the Hertz/count constant. Initially I was reluctant to perform a straightforward direct multiplication to avoid multiple precision multiplication on an eight bit processor.

I also thought that multiple precision multiplication on an 8-bit microprocessor, even with a 50 MHz clock speed, might be too slow.

The concern for speed was unwarranted. The multiplication only had to beat the display speed of an LCD and the perception of the user. The incremental encoder accumulator, although it can accumulate a large number of counts on each knob turn, needs only to be sampled periodically.

After the user stops rotating the encoder knob, the sampled encoder value will be static and the displayed frequency will be exact. During rapid changes, the LCD digits change too fast to easily read but do provide a “rate of change” indication.

Methodology

Using multiplication to directly calculate the frequency was not as big a problem as I initially thought. One factor simplifying the direct multiplication was that the Hertz/count multiplier is a constant. Using a shift and add multiplication technique simplified the remainder of the multiplication “problem.”

Because the project used a C8051F410 processor, the 8051 MUL instruction was available to perform an 8X8 unsigned multiply with a 16-bit product.

Instead of using the MUL instruction for a multiplication requiring a 40 bit (or larger) result and a multiplier of 32 bits, I chose to use a shift and add algorithm which was easier for me to understand and code. An added advantage is that the algorithm is easily extended for any number of multiplicand bytes.

Russian Peasant Multiplication In MyForth
19Jun10 SVFIG Presentation by Bob Nash

Russian Peasant Algorithm

The shift and add algorithm I used is commonly known as the Russian Peasant algorithm. For those (few) of you not familiar with this algorithm, I will attempt to provide a simple description:

1. In this description, I use the term “multiplier” to designate one of the numbers to be multiplied and “multiplicand” to designate the other number. Of course, both numbers are multiplicands.
2. Multiplication is performed in a sequence of steps. For each step, one multiplicand (which I call the multiplier) is shifted right one bit (divided by 2) and the other multiplicand is shifted left one bit (multiplied by 2).
3. Thus, at each step, the product of the “multiplier” and the multiplicand is the same. When the multiplier is divided down to 1, the multiplicand contains the approximate result.
4. The result is approximate because each time a 1 is shifted right from the least significant bit of the multiplier, there is a remainder that is equal to the value of the previous multiplicand. This is because one multiplicand’s worth of the result is “lost” when the multiplier bit is shifted right.
5. The remainders generated when shifting odd numbered multipliers can be accumulated as the algorithm progresses. When shifting is complete, the accumulated remainders can be added to the multiplicand result to yield an accurate result.
6. In the general case, the number of additions is variable and depends on the number of bits in the multiplier. Also, the total number of shifts varies, depending on the position of the most significant bit in the multiplier.

Because the multiplier is a known constant, it is possible to simplify the shifting calculation by noting the position of the most significant 1 bit and performing a fixed number of shifts.

The algorithm I coded calculates the position of the most significant 1 bit and is accurate regardless of the multiplier size. This allows for the use of a different crystal frequency or other changes in the Hertz/count constant (e.g., adjusting for a new crystal). Thus, the algorithm presented here should work for any multiplier, provided that the result accumulator is large enough.

I considered checking the multiplier for a value of 1 (or 0) at each step of the algorithm in order to determine algorithm termination. The only methods I could think of involved ORing multiple bytes which takes a fair number of cycles that had to be repeated at each step. It seemed that a one-time calculation using shifting would be more efficient. Most likely, there is a better way to do this than the shifting method I use.

In the program listing, the Word “msteps” calculates the number of multiply steps by counting the number of leading zeros in the multiplier and subtracting that from 32 (the multiplier is a four-byte constant). This yields the number of right shifts that are required to shift out the most significant bit in the multiplier.

For the calculation in the code, which uses the multiplier for a 10 Hertz increment, the total number of shifts is 14.

Here is how msteps works:

1. One of my naming conventions is that Words that perform initialization begin with a single or double “slash” (“/” or “//”). Thus, the “slash” in “|/quadb10” designates “initialization”, not division. Another convention is that a “bar” in the name indicates a macro, not a callable routine. A later section provides more notes related to MyForth coding.

2. The macro “|/quadb10” initializes the multiplier, quadb, by loading the scaled Hertz/step value into it. The quadb multiplier is 32 bits and consists of four 8-bit direct memory cells quadb, quadb1, quadb2 and quadb3. The most significant byte is quadb.

3. After loading the multiplier constant, msteps puts 32 on the stack (with “32 #”) and enters a 32 step loop that left shifts quadb through the carry using quadb2*’ (in MyForth, a “tick” in the name usually indicates “carry”). When the carry bit is set, the loop exits. Otherwise, one is subtracted from the top of stack for each zero shifted into carry (i.e., with “-1 # +”).

The Word “pentd*quadb” performs the multiplication, as follows:

1. Both pentc and pentd are 5-byte numbers (hence the “pent” in the names).
2. The result is accumulated in “pentc” and “pentd” contains the frequency bits from the encoder. The Word ie>pentd moves the incremental encoder accumulator into pentd. This Word is executed in an interrupt service routine so that the encoder value loaded into pentd is an accurate “snapshot” and not a partial result.
3. The Word “0pentc” zeros out the result accumulator, pentc. It is assumed that pentd, the frequency Word from the encoder accumulator, is pre-loaded.
4. As described above, msteps calculates the total number of loop steps needed to complete the multiplication. The macro |/quad10 loads the 4-byte Hertz/count multiplier into quadb, as described above.

5. Inside the loop, the Word pentd2*' left shifts pentd by one bit. Similarly, quadb2/' shifts the multiplier, quadb, one bit to the right through the carry bit.
6. In the definition of quadb2', the pentd multiplicand is added to the pentc result accumulator if a 1 is right shifted out of quadb into carry (hence the need to always clear the carry on entry with "clrc").
7. Astute readers may have noted that quadb2/' is executed after pentd2*' so that the remainder added to pentc is always twice as large as the actual remainder. After shifting is complete, pentc/2' divides the result by two to correct for this. This method is used instead of reversing the order of multiplication because quadb2*' operates on a shift of a 1 into carry. Thus, when the last 1 is shifted out, quadb is zero, not 1. There is probably a better way to do this calculation.

Accuracy

This Hertz/count constant depends on the crystal oscillator frequency that is used by both the microprocessor and DDS chip (about 24.5 MHz). The constant (or the oscillator) can be adjusted so that the value displayed by the LCD more closely agrees with the actual frequency.

Typically, without calibration, the displayed frequency is accurate to within a tenth of a Hertz over the range of the oscillator (approximately 600 kHz).

The application limits the output frequency to 25 kHz. As currently coded, the calculation is good up to approximately 600 kHz. However, the algorithm can be easily extended to provide a result larger than 5 bytes. I intend to do this when I use a DDS chip with a larger frequency register (e.g., an AD9851).

Russian Peasant Multiplication In MyForth
19Jun10 SVFIG Presentation by Bob Nash

MyForth Notes
For those not familiar with MyForth the following notes may help understand the code in the attached listing.

General
- MyForth is a minimalist 8-bit Forth written in Gforth by Charley Shattuck. MyForth uses many ideas from ColorForth and is designed specifically for 8051 processors. Charley has developed another MyForth-like system for the Arduino (Arduous Forth? ArForth?) that he will describe at this meeting.
- The top of stack is the 8051 accumulator and is designated ”t” for ”top of stack.” You can see this used in ”assembly” routines such as |pentc+pentd in the program listing
- MyForth uses two vocabularies. Using ”[” sets Gforth first in the search order and ”]” sets MyForth first in the order.
- Colon definitions generate a callable routine. A semicolon generates an 8051 RET instruction. MyForth is optimized to eliminate unneccesary returns. There can be multiple semicolons in a definition (this is unstructured but corresponds with typical assembly coding). Macros are defined with a ”:m ... m;” pair and cannot be executed from the command line. Macros are often identified with a ”bar” as the first character in the name (e.g., :m |my-macro dup drop m;). Macros can be made callable by encapsulating them in a colon definition (e.g., : my-call |my-macro ;).
- Parameters for logic operators such is ”if” are left on the stack and must be explicitly dropped. Conditionals operating on bits and carry do not require any stack manipulation. There is no ”else” conditional.

Numbers
- Numbers are put on the processor’s stack at run time with the Word ”#” following the number. Double numbers (16 bit numbers) are put on the stack with ”##.”
- For example, ”22 #” puts the number 22 on the stack and ”$ABCD ## ” puts two bytes on the stack, $AB (MSB) on the top, $CD under it.

Allocating & Naming A Direct Cell

- Direct cells (RAM bytes) are allocated by getting the current RAM pointer with cpuHere and alloting cells at that location.
- For example, to name and allocate a direct cell called ”foo”, use: cpuHERE constant foo 1 cpuALLOT
- Multi-byte numbers are allocated similarly. To allocate a 4-byte number called ”bar”, use: cpuHERE constant bar 4 cpuALLOT. Individual cells can be addressed using macros and the Gforth compiler. For example: :m bar-lsb [bar 3 +] m;

Loops

- MyForth implements ”for ... next” loops, which have a range of an unsigned 8-bit number. The register to use (1 to 7) for looping is specified before the ”#for” and ”#next” keywords. The number of looping steps is specified by the value on the top of the stack.
- For example, loop repeated 250 times using register 5 would be expressed as: ”250 # 5 #for doit 5 #next” .

Fetch and Store

- A number is put on the stack by specifying the 8051 direct cell number (or register) followed by the Word ”#@” after the cell.
- To store a number to a cell, use the Word ”#!” after the cell.
- For example, storing the number 22 to direct cell 34 is expressed as ”22 # 34 #!” Similarly, putting the contents of direct cell 34 on the stack would be expressed as ”34 #@” .
- Assuming that direct cell 34 is named ”foo” (see above), store the value 22 to it using ”22 foo #!” .

Conclusion

Although using the 8051 MUL instruction may have produced faster and/or smaller code, Russian Peasant multipication was easier for me to understand and implement. Extending the algorithm for larger multipicands is very straightforward, which is important to me for future applications.

Implementation of the algorithm in MyForth resulted in a small amount of simple code. The process was also very instructive.

If you are interested in the VFO application, particularly the (somewhat tricky) driver for the AD9833 DDS chip, please contact me at Bob.Nash1@GMail.com . Another item of interest may be the code for decimal digit conversion and LCD scaling. Please feel to contact me for copies of this code.

Russian Peasant Multiplication In MyForth

19Jun10 SVFIG Presentation by Bob Nash

Program Listing

```
\ multiply.fs -- Multi-byte multiply by a constant -- 12May09 rjn
\
]  \ target forth
\
0 [if] ----------[ Russian Peasant Multipication ]----------

1. Multiplies a 5-byte number by a 4-byte constant using the Russian
   Peasant algorithm (shifts and adds).
2. Uses the pentd (penta) register for the multiplicand and
   the quadb (quad) register for the multiplier.  The result is in the pentc
   register ready for decimal digit conversion with .pdigits .
------------------------------------------------------------------------ [then]
\
: pentd2*'  \ multiply contents of pentd register by 2, keep carry result
   clrc   pentd4 #@  2*' pentd4 #!  pentd3 #@  2*'  pentd3 #!
          pentd2 #@  2*' pentd2 #!  pentd1 #@  2*'  pentd1 #!
          pentd  #@  2*' pentd  #! ;

\ : set-pentd   $A7 # $E4 # $0B # $54 # $02 #  pentd #  5 # !cells ;
\ : set-pentd1  $A9 # $2A # $00 #  $00 # $00 #  pentd #  5 # !cells ;

0 [if] \ ---------- FYI, from registers.fs ----------

:m |pentc+pentd
     [ t push  clrc
             pentc4 t mov  pentd4 addc  t pentc4 mov
          pentc3 t mov  pentd3 addc  t pentc3 mov
             pentc2 t mov  pentd2 addc  t pentc2 mov
          pentc1 t mov  pentd1 addc  t pentc1 mov
          pentc  t mov  pentd  addc  t pentc  mov
         t pop ]
m;
\
: pentc+pentd  |pentc+pentd ;

:m |/quadb \ 916 decimal
   $00 # quadb #!  $00 # quadb1 #!  $03 # quadb2 #!  $94 # quadb3 #! m;
```

Russian Peasant Multiplication In MyForth

19Jun10 SVFIG Presentation by Bob Nash

```
:m |/quadb10  \ 9156 decimal -- rounded 0.091557
  $00 # quadb #!  $00 # quadb1 #!  $23 # quadb2 #!  $C4 # quadb3 #! m;
[then]  \ ------------------------------------------------------------
\ : set-quadb   $A7 # $E4 # $0B # $54 #  quadb #  4 # !cells ; \ test

: quadb2/' \ divide quadb register by 2, keeping carry result
  clrc  quadb  #@  2/'  quadb  #!  quadb1 #@  2/' quadb1 #!
       quadb2 #@  2/'  quadb2 #!  quadb3 #@  2/' quadb3 #!
  if' |pentc+pentd ; then ;

: quadb2*'  \ multiply quadb register by 2, keeping carry result
  clrc  quadb3 #@ 2*' quadb3 #!  quadb2 #@ 2*' quadb2 #!
       quadb1 #@ 2*' quadb1 #!  quadb  #@ 2*' quadb  #! ;

: msteps  ( - n)  \ calculate multiply steps by looking for the last "1"
  |/quadb10  32 #  32 #   4 #for quadb2*' if' ; then -1 # + 4 #next ;

\ note: 31 bytes, this is the last step: remainders are accumulated X 2
: pentc/2'  \ divide contents of pentc register by 2, keep carry info
  clrc   pentc  #@  2/' pentc  #!  pentc1 #@  2/'  pentc1 #!
        pentc2 #@  2/' pentc2 #!  pentc3 #@  2/'  pentc3 #!
        pentc4 #@  2/' pentc4 #! ;

: pentd*quadb  \ multiply the two registers, set pentd first
  0pentc
  [ 4 push ]  \ belt and suspenders
    msteps |/quadb10  4 #for pentd2*' quadb2/' 4 #next
  [ 4 pop  ]  pentc/2' ;

: ie>pentd  \ frequency value to send to the DDS is accumulated in ie-acc
  ie-acc3 #@  pentd4 #!  ie-acc2 #@  pentd3 #!  ie-acc1 #@  pentd2 #!
  ie-acc  #@  pentd1 #!  0 # pentd #! ;

\ notes:
\   1. ($2AA9)($394)=$98A4B4  .pdigits -->  00099992676 (999.93 Hertz)
\   2. ($2AAA)($394)=$98A848
\ : multiply  pentd*quadb  ( .pentc) ;
\ : test-multiply  set-pentd1  multiply ;  \ test-multiply .pdigits
```