How to program the AD9834 in your DDS Development Kit

By Bruce Hall, W8BH

At the heart of your DDS development kit is the Analog Devices chip AD9834. That tiny, surface-mount piece of plastic handles all of the frequency synthesis on board. Did you know that your DDS kit has been programmed to output a signal of 12.5 MHz when it's first turned on? If you'd like to learn more, read on.

The AD9834 is a 75 MHz device, overclocked to 100 MHz. It divides its master clock by a 28 bit number, yielding an incredible 100 MHz/2^28 = 0.37 Hz resolution. Internally there are two frequency and two phase registers, which can be set and selected by software. The output frequency is determined by the value of the active frequency register, according to the formula:

Freq (MHz) = Register * 100/2^28.

Most of the time, however, we want to know the register value for a certain frequency. This equation would be:

Register = (Freq/100)*2^28
= 2684354.56 * Freq (in MHz)

For example, the register value for 7.040 MHz would be (7.040/100)*2^28 = 18897856 decimal or $01205BC0 hexadecimal. It would be great if we could just program this number into our microcontroller and then send it to the AD9834. Unfortunately it's not quite that easy! For starters, $01205BC0 is a really big number for an 8-bit controller. Our ATmega88 can only handle numbers up to 256, which is one byte (or 2 hex digits) in size. The number we want is 4 bytes long, so we'll need to send it in four byte-sized chunks:

Send this:	$01	(The first byte is called the MSB, 'most-significant byte')
Then this:	$20	
Then this:	$5B	
Then this:	$C0	(The last byte is called the LSB, 'least-significant byte')

There is another wrinkle, too: the AD9834, for some obscure reason, wants us to embed some bits to tell it which register it should load. That right: we have to chop up this huge number, and stuff it with a couple ‘01’s if we want frequency register0 or a couple ‘10’s if we want frequency register1. If you get squeamish at the sight of binary 0’s and 1’s, you can skip the next few paragraphs. Suffice it to say that it takes humans a while to figure out what bits to send to our AD9834. It is much easier for our AVR microcontroller.

To set a frequency register in the AD9834, we have to send it a total of 32 bits (8 bytes). These 32 bits will include the 28-bit value that we want in the register, plus 4 bits for the register address. The bits must be sent in the following order:

2 bits for register address
14 upper bits for the register value
2 bits for the register address (again)
14 lower bits for the register value

We send the register address twice because the AD9834 accepts data in 16-bit chunks. Every time we send it data, the first two bits identify what the data is for:

00 command follows
01 frequency register0 data follows
10 frequency register1 data follows
11 phase register data follows

Getting back to our 7.040 MHz example, what number do we need to enter? Here is the plan: write out the number in binary, cut it into 14 bit sections, put in ‘01’ for register0 in the appropriate spots, and then send it to the DDS chip in byte-sized chunks.

Step1: write out the number ($01205BC0) in binary. I like to put dots between every four bits, which divide it into hex-digits. You may like to use a space, or something else.

$1205BC0 = 0001.0010.0000.0101.1011.1100.0000 (28 bits)

-	-	-	-	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	1	1	0	1	1	1	1	0	0	0	0	0	0

Step2: divide it into two 14-bit chunks

upper chunk: 00010010000001 (14 bits)
lower chunk: 01101111000000 (14 bits)

-	-	-	-	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	1	1	0	1	1	1	1	0	0	0	0	0	0

Step3: add ‘01’ to start of each chunk, so that the DDS knows it’s for register0

upper chunk: 0100010010000001 (16 bits)
lower chunk: 0101101111000000 (16 bits)

0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	1	0	1	1	0	1	1	1	1	0	0	0	0	0	0

Step 4: group the 32 bits into four byte-sized values that the micro can handle:

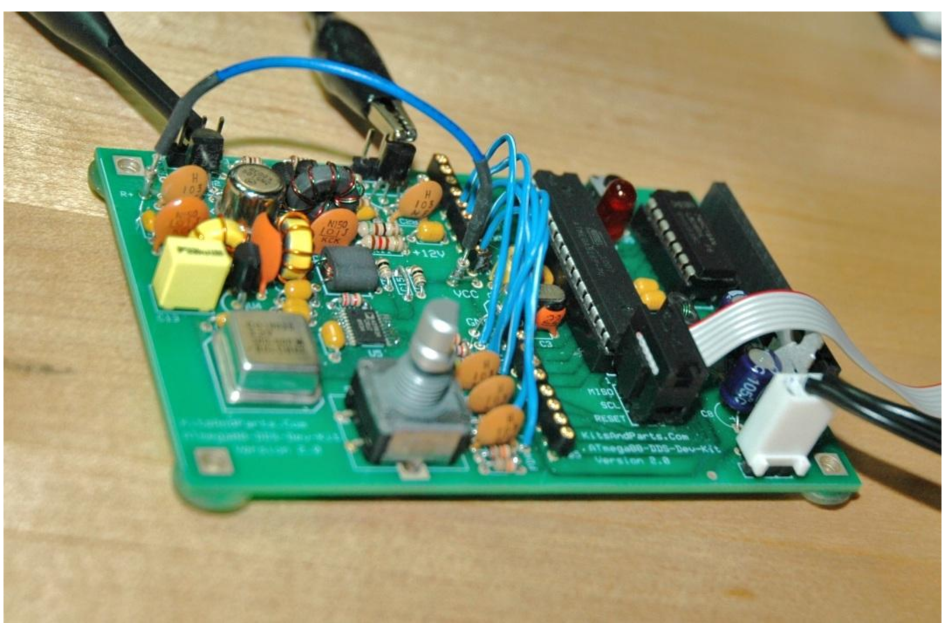
0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	1	0	1	1	0	1	1	1	1	0	0	0	0	0	0
4				4				8				1				5				B				C				0			

Final result = $4481.5BC0. Send four bytes: $44, $81, $5B, $C0

That's it for the binary stuff. If we send the four bytes $44, $81, $5B, and $C0 to the DDS, we will get 7.040 MHz out the other end.   Diz has given us a routine to send stuff to the DDS, called Shift_16. (Respectfully ignore the comments in the Shift_16 code, since they are obviously a typo meant for something else).

Finding the 12.5 MHz output

Let's try some experiments. Hook up a frequency counter to your output, and enable output with a jumper from Vcc to T+ or R+. Here is a picture of my setup. If you don't have a frequency counter handy, terminate the output with a 50 ohm resistor, extend a wire from the output towards your receiver and tune to the indicated frequency.   Make sure that you are able to see RF output at 10 MHz, or whatever other frequency is indicated by your LCD.



Notice that I don't even have my LCD connected! It doesn't do anything in the following experiments – in fact, we halt the microcontroller before the LCD is initialized. Looking at a non-functioning LCD was bothering me, so I removed it.

Now find a group of lines, near the beginning of the program, that call SHIFT_16 several times. The last group of 3 is listed here. Add a single instruction below it, like this:

```
        ldi    temp1,$20           ;enable output
        ldi    temp2,$00
        rcall  SHIFT_16            ;output to DDS chip

;*********************************************************************
; W8BH  -  START OF INSERTED CODE
;*********************************************************************

halt: rjmp     halt                ;hard stop.
```

The 'halt' line forms an infinite loop. The microcontroller cannot advance beyond this point. When you run this code, your output should read 12.5 MHz. The lines above the hard stop send data to the AD9834 that specify this peculiar frequency. Why 12.5 MHz?? I dunno.

Experiment #1

It's time to send our own data. From all the binary discussion above, we know the number for an output of 7.040 MHz. Here is the code:

```
     rcall     exp01               ;call our experiment first, then halt
halt: rjmp     halt                ;hard stop.


Exp01:
;    sends the value $4481.5BC0 to the DDS chip
;    the corresponding DDS output frequency is 7.040 MHz
     ldi    temp1,$5B
     ldi    temp2,$C0
     rcall  SHIFT_16
     ldi    temp1,$44
     ldi    temp2,$81
     rcall  SHIFT_16
     ret
```

Run it, and your frequency counter should jump to 7.040 MHz. That's nice, but it is really easy to get the bytes mixed up. I'd like to see the bytes in proper order. Here is a more user-friendly routine that takes the four input bytes from temp1 through temp4.

Experiment #2

```
     rcall     exp02               ;call our experiment first, then halt
halt: rjmp     halt                ;hard stop.


Exp02:
;    sends the value $4481.5BC0 to the DDS chip
;    the corresponding DDS output frequency is 7.040 MHz
```

```
;     same result as Exp01, except easier to read
      ldi     temp1,$44
      ldi     temp2,$81
      ldi     temp3,$5B
      ldi     temp4,$60
      rcall   DDSSendData
      ret

DDSSendData:
;     sends a 4-byte value to the DDS chip,
;     uses temp1 (MSB) to temp4 (LSB)
;     note: this is a 32-bit value (28bit freq + 4bit register select)
      push    temp1                  ;save MSW for now
      push    temp2
      mov     temp1,temp3
      mov     temp2,temp4
      rcall   SHIFT_16               ;send LSW first
      pop     temp2
      pop     temp1
      rcall   SHIFT_16               ;send MSW now
      ret
```

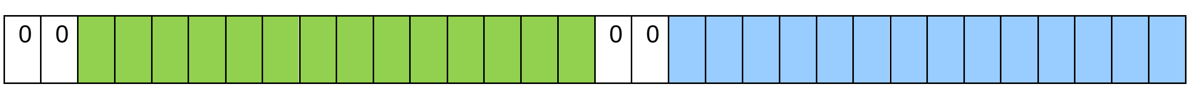
It does the exact same thing, but easier to use.   If you get tired of looking at the 7.040 MHz output, try some other values. I've put an appendix at the end with a table of frequencies and their 'numbers'. For example, for 10 MHz use the numbers $46, $66, $59, and $99.

It is quite tedious doing all of the bit manipulations in Steps 1-4 above. I found myself making lots of mistakes when trying to cut and splice those 1's and 0's. Instead, we can write code to do the bit-stuffing.   In our case we need to put 2 bits in front of and 2-bits in the middle of our 28-bit value. Let's do that by shifting the whole thing 2 bits to the left, and then shifting the lower 2 bytes to the right.   Got that? Me neither! Here is a diagram of the steps, just like before:

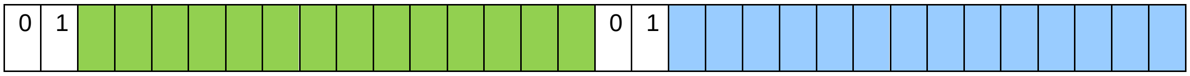
Unused   14 bits in upper chunk   14 bits in lower chunk

After shifting everything 2 bits to the left, it will look like this:

Now shift the lower 16 bits back to the right, and it will look like this:



Now we have space for our register selection bits. To send the value to register0, our main register, we just have to put '01' at both '00' locations.



An easy way to set these two bits is with the ORI (OR-immediate) instruction. An alternative method would be to use the SBR (set bit in register) instruction. Now the 32 bits are ready, and can be send them to the DDS. Here is the code:

Experiment #3

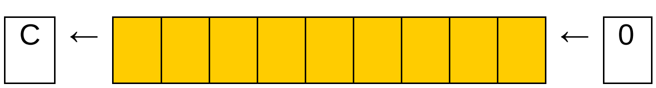
```
       rcall     exp03            ;call our experiment first, then halt
halt: rjmp     halt               ;hard stop.

Exp03:
;     sends the 28-bit value $0120.5BC0 to DDS Freq Register0
;     the corresponding DDS output frequency is 7.040 MHz
      ldi     temp1,$01
      ldi     temp2,$20
      ldi     temp3,$5B
      ldi     temp4,$C0
      rcall   DDSSetFreq28
      ret

DDSSetFreq28:
;     sends the 28-bit magic number to Freq Register0
      lsl     temp4                ;shift 1 bit left
      rol     temp3
      rol     temp2
      rol     temp1
      lsl     temp4                ;shift 1 bit left again
      rol     temp3
      rol     temp2
      rol     temp1
      lsr     temp3                ;undo shifts in LSW
      ror     temp4
      lsr     temp3
      ror     temp4
      ori     temp3,$40            ;add reg0 select bits to byte3
      ori     temp1,$40            ;add reg0 select bits to byte1
      rcall   DDSSendData          ;send Reg0 output# to DDS chip
      ret
```

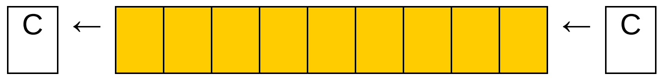
Did you notice how we use 1 shift and 3 rotate instructions to shift everything one bit to the left? The first instruction, LSL (logical shift left), works on the least significant byte (temp4). The 'rightmost' bit, bit0, gets a 0. Bit1 gets whatever bit0 was, bit2 gets whatever bit1 was, etc. What happens to the leftmost bit, Bit7? It gets bumped off to a carry bit.

LSL:



The carry bit is important, because bit7 needs to be shifted into the byte above it. The instruction that does that job is ROL (rotate left). It does the same thing as LSL, except that instead of putting a zero into bit0, it puts the carry bit.

ROL: new carry old carry



We can extend this one-bit shift over as many bytes as we want, adding a ROL for each more-significant byte. The two ORI instructions at the end specify that the data goes to frequency register 0. If we want to send that data to the second frequency register instead, we'd ORI with $80 instead of $40.

With DDSSetFreq28, calculate the number we need from the formula on page one, load it into temp1-temp4, and the desired frequency is output. The appendix lists some example numbers for the QRP calling frequencies.

Great, now it's time for another wrinkle. Diz does NOT use these 28-bit numbers to calculate frequency! And for a good reason, too. Although the number is fairly precise, it still has an inherent inaccuracy of up to 0.37 Hz. As we turn the encoder, those inaccuracies accumulate. For example, suppose that the 1 KHz number has an rounding error of +0.2 Hz. As we turn at 1 KHz increments from 7.040 to 7.080, we add 40 * 0.2 Hz = 8 Hz to our operating frequency. The more we turn the encoder, the larger the error gets.

We can minimize the rounding error if we start with more precise numbers. Diz uses larger (x32) numbers that give us 5 more bits of precision. So instead of an accuracy of 0.37 (1E8/2^28) Hz, we now get an accuracy of 0.01 Hz (1E8/2^33). You can turn the encoder knob all day and not build up any significant error. The key is to do all computations at the higher precision, and fill the frequency register with the result/32. It is super-simple to divide by 32: just shift your result 5 bits to the right. In binary, shifting right divides by 2 and shifting left multiplies by two. Five left shifts = /32.

Let's make a new routine, using Diz' 32-bit "super" numbers instead of the regular 28-bit numbers. All we have to do is shift our supernumber 5 bits to the right (divide-by-32) and then use our old routine. Maybe something like this:

```
DDSSetFreq32idea:
;    sends the 32-bit magic number to Freq Register0
;    divides the number by 32 (5 shifts right), then calls DDSSetFreq28
     ldi    temp5,3                  ;do 3 bit-shifts to the right
dd1: lsr    temp1                    ;shift LSB
     ror    temp2
     ror    temp3
     ror    temp4                    ;shift MSB
     dec    temp5                    ;all done?
     brne   dd1                      ;no, do another bit-shift
```

```
        rcall   DDSSetFreq28
        ret
```

It works, but can be better. Look at the 28-bit routine again. The first thing it does is shift everything 2 bits to the left. Why bother going right 5 bits and then immediately back (left) 2bits? It is simpler and faster to go right by 3 bits:

```
DDSSetFreq32:
;       sends the 32-bit "super" magic number to Freq Register0
        ldi     temp5,3                 ;do 3 bit-shifts to the right
dd0:    lsr     temp1                   ;shift LSB
        ror     temp2
        ror     temp3
        ror     temp4                   ;shift MSB
        dec     temp5                   ;all done?
        brne    dd0                     ;no, do another bit-shift
        lsr     temp3                   ;create 2-bit space in LSW
        ror     temp4
        lsr     temp3
        ror     temp4
        ori     temp3,$40               ;add reg0 select bits to byte3
        ori     temp1,$40               ;add reg0 select bits to byte1
        rcall   DDSSendData             ;send Reg0 output# to DDS chip
        ret
```

Compare our new routine to Diz' FREQ_OUT routine. They accomplish the exact same function, and are written very similarly. Check it out with the last experiment. The code below can use the same frequency numbers used by the main program, which are stored in the 4-byte variable rcve0.

Experiment #4

```
        rcall     exp04              ;call our experiment first, then halt
halt:   rjmp      halt               ;hard stop.

Exp04:
;       sends the 32-bit value $240B.7803 to DDS Freq Register0
;       the corresponding DDS output frequency is 7.030 MHz
        ldi     temp1,$24
        ldi     temp2,$0B
        ldi     temp3,$78
        ldi     temp4,$03
        rcall   DDSSetFreq32
        ret
```

That's it: you now know how to program the DDS chip. One thing I didn't cover is using the second frequency register. I have put a few routines into the source code that let you select output from either register. Having a second frequency is quite handy for things like dual VFO's/split operation, RIT/XIT, FSK, RTTY, etc.

If you try any of the above experiments with your DDS kit, don't forget to remove the code (especially the hard stop) when you're done. Your DDS kit won't work until you remove the hard stop.

Appendix

In my examples I used numbers for 40 meters. Here are some more numbers you can try.

DDS Output Frequency (MHz)	Experiment #2 Direct Entry for Reg 0	Experiment #3 28-bit Magic Number	Experiment #4 32-bit Magic Number
3.560	4247514E	0091D14E	123A29C7
7.030	447F72E4	011FF2E4	23FE5C91
7.040	44815BC0	01205BC0	240B7803
10.000	46665999	01999999	33333333
10.106	46777117	019DF117	33BE22E5
14.060	48FF65C9	023FE5C9	47FCB923
18.096	4B947650	02E53650	5CA6CA03
21.060	4D7A5E1B	035E9E1B	6BD3C361
24.906	4FF06656	03FC2656	7F84CAD5
28.060	51F5566C	047D566C	8FAACD9E

Source Code

Find the following lines in the original source code, near the beginning of the program. Insert new code as indicated.

```
       ldi           temp1,$21 ;reset AD9834 and init all registers
       ldi           temp2,$00
       rcall  SHIFT_16 ;output to DDS chip


       ldi           temp1,$7F ;freq0 ls 14 bits
       ldi           temp2,$29
       rcall  SHIFT_16 ;output to DDS chip

       ldi           temp1,$47 ;freq0 ms 14 bits
       ldi           temp2,$FF
       rcall  SHIFT_16 ;output to DDS chip

       ldi           temp1,$80 ;freq1 ls 14 bits
       ldi           temp2,$00
       rcall  SHIFT_16 ;output to DDS chip

       ldi           temp1,$80 ;freq1 ms 14 bits
       ldi           temp2,$80
       rcall  SHIFT_16 ;output to DDS chip

       ldi           temp1,$C0 ;clear phase0
       ldi           temp2,$00
       rcall  SHIFT_16 ;output to DDS chip

       ldi           temp1,$E0 ;clear phase1
       ldi           temp2,$00
       rcall  SHIFT_16 ;output to DDS chip

       ldi           temp1,$20 ;enable output
       ldi           temp2,$00
       rcall  SHIFT_16 ;output to DDS chip

;*********************************************************************
; W8BH  -  START OF INSERTED CODE
;*********************************************************************

     rcall     exp01              ;change this to exp02, exp03, or exp04
halt: rjmp     halt               ;hard stop.


Exp01:
;    sends the value $4481.5BC0 to the DDS chip
;    the corresponding DDS output frequency is 7.040 MHz
     ldi    temp1,$5B
     ldi    temp2,$C0
     rcall  SHIFT_16
     ldi    temp1,$44
     ldi    temp2,$81
     rcall  SHIFT_16
     ret

Exp02:
;    sends the value $4481.5BC0 to the DDS chip
;    the corresponding DDS output frequency is 7.040 MHz
```

```
;     same result as Exp01, except easier to read
      ldi     temp1,$44
      ldi     temp2,$81
      ldi     temp3,$5B
      ldi     temp4,$60
      rcall   DDSSendData
      ret

DDSSendData:
;     sends a 4-byte value to the DDS chip,
;     uses temp1 (MSB) to temp4 (LSB)
;     note: this is a 32-bit value (28bit freq + 4bit register select)
      push    temp1                   ;save MSW for now
      push    temp2
      mov     temp1,temp3
      mov     temp2,temp4
      rcall   SHIFT_16                ;send LSW first
      pop     temp2
      pop     temp1
      rcall   SHIFT_16                ;send MSW now
      ret

Exp03:
;     sends the 28-bit value $0120.5BC0 to DDS Freq Register0
;     the corresponding DDS output frequency is 7.040 MHz
      ldi     temp1,$01
      ldi     temp2,$20
      ldi     temp3,$5B
      ldi     temp4,$C0
      rcall   DDSSetFreq28
      ret

DDSSetFreq28:
;     sends the 28-bit magic number to Freq Register0
      lsl     temp4                   ;shift 1 bit left
      rol     temp3
      rol     temp2
      rol     temp1
      lsl     temp4                   ;shift 1 bit left again
      rol     temp3
      rol     temp2
      rol     temp1
      lsr     temp3                   ;undo shifts in LSW
      ror     temp4
      lsr     temp3
      ror     temp4
      ori     temp3,$40               ;add reg0 select bits to byte3
      ori     temp1,$40               ;add reg0 select bits to byte1
      rcall   DDSSendData             ;send Reg0 output# to DDS chip
      ret

Exp04:
;     sends the 32-bit value $240B.7803 to DDS Freq Register0
;     the corresponding DDS output frequency is 7.030 MHz
      ldi     temp1,$24
      ldi     temp2,$0B
      ldi     temp3,$78
      ldi     temp4,$03
      rcall   DDSSetFreq32
      ret

DDSSetFreq32:
;     sends the 32-bit "super" magic number to Freq Register0
```

```
     ldi    temp5,3                ;do 3 bit-shifts to the right
dd0: lsr    temp1                  ;shift LSB
     ror    temp2
     ror    temp3
     ror    temp4                  ;shift MSB
     dec    temp5                  ;all done?
     brne   dd0                    ;no, do another bit-shift
     lsr    temp3                  ;create 2-bit space in LSW
     ror    temp4
     lsr    temp3
     ror    temp4
     ori    temp3,$40              ;add reg0 select bits to byte3
     ori    temp1,$40              ;add reg0 select bits to byte1
     rcall  DDSSendData            ;send Reg0 output# to DDS chip
     ret

DDSOutputA:
;    set output according to frequency register 0
     ldi    temp1,$20              ;enable output A
     ldi    temp2,$00
     rcall  SHIFT_16               ;output to DDS chip
     ret

DDSOutputB:
;    set output according to frequency register 1
     ldi    temp1,$28              ;enable output B
     ldi    temp2,$00
     rcall  SHIFT_16               ;output to DDS chip
     ret

DDSReset:
;    initializes the AD9834 DDS chip, clears all registers
;    note: output is disabled until reset bit is cleared.
     ldi    temp1,$21              ;reset command
     ldi    temp2,$00
     rcall  SHIFT_16               ;output to DDS chip
     ret


;*******************************************************************
; W8BH  -  END OF INSERTED CODE
;*******************************************************************

       rcall  DEFAULT_FREQ  ;move default freq to buffers
       rcall  FREQ_OUT                    ;output freq bits to DDS chip




; FURTHER DOWN IN THE SOURCE CODE ARE THESE ORIGINAL ROUTINES…

FREQ_OUT:
 ldi temp1,$20 ;28 bits FREQ0 to AD9834
 ldi temp2,$00
 rcall SHIFT_16 ;output to DDS chip
 ldi yl,low(rcve0)

    rcall  AdjFreq          ;!!W8BH - Added for IF adjustment

 ld temp4,y+   ;LSB
 ld temp3,y+   ;
 ld temp2,y+   ;
 ld temp1,y+   ;MSB
```

```
 lsr temp1                  ;MSB-high
 ror temp2                  ;MSB-low
 ror temp3                  ;LSB-high
 ror temp4                  ;LSB-low

 lsr temp1                  ;MSB-high
 ror temp2                  ;MSB-low
 ror temp3                  ;LSB-high
 ror temp4                  ;LSB-low

 lsr temp1                  ;MSB-high
 ror temp2                  ;MSB-low
 ror temp3                  ;LSB-high
 ror temp4                  ;LSB-low

 lsr temp3
 ror temp4
 lsr temp3
 ror temp4

 push temp1
 push temp2
 mov temp1,temp3
 mov temp2,temp4

 ori temp1,0b01000000
 rcall SHIFT_16             ;send 14 bits
 pop temp2
 pop temp1

 push temp1
 push temp2
 ori temp1,0b01000000
 rcall SHIFT_16             ;send 14 bits

 mov temp1,temp3
 mov temp2,temp4
 ori temp1,0b10000000
 rcall SHIFT_16             ;send 14 bits

 pop temp2
 pop temp1
 ori temp1,0b10000000
 rcall SHIFT_16             ;send 14 bits

 ret

;**********************************
;* SHIFT_16
;* display HEX bytes
;* at active LCD position
;* byte count in temp2
;***********************************
SHIFT_16:
;16 bit serial out msb first in temp1, then lsb in temp2
 push temp3
 cbi PORTD,DDSenable        ;FSYNC goes LOW
 ldi temp3,16                             ;16 bits bit counter
shift8:
 sbi PORTD,DDSdata          ;set port bit
 rol temp1                                       ;shift dds address byte
 brcs clockit                     ;check for 1/0
```

```
 cbi PORTD,DDSdata         ;clear port bit
clockit:
 nop
 cbi PORTD,DDSclock        ;clock dds
 nop
 sbi PORTD,DDSclock
 dec temp3                                          ;decrement bit counter
 breq sox                                                 ;exit if done
 cpi temp3,8                                  ;check byte counter
 brne shift8                                  ;output more bits
 mov temp1,temp2                       ;get lsb
 rjmp shift8                                  ;write data bits
sox:
 sbi PORTD,DDSenable              ;FSYNC goes HIGH
 pop temp3
 ret
```