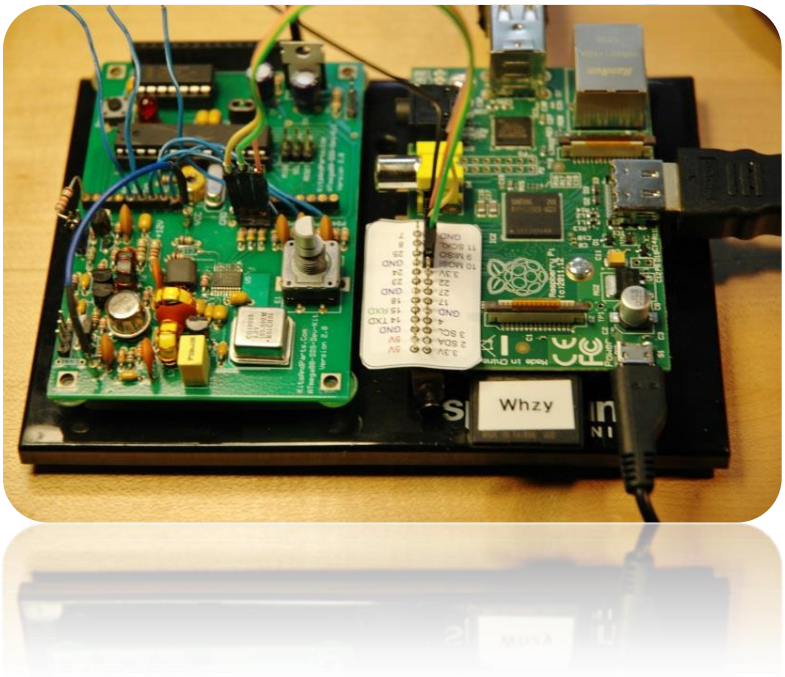


Controlling your VFO with a Raspberry Pi

Bruce E. Hall, W8BH

1) INTRODUCTION

Many home-brew rigs use a VFO with direct-digital synthesis (DDS). These DDS devices are typically controlled by on-board microcontrollers. In this tutorial I will show you how you can control your VFO from a Raspberry Pi instead.

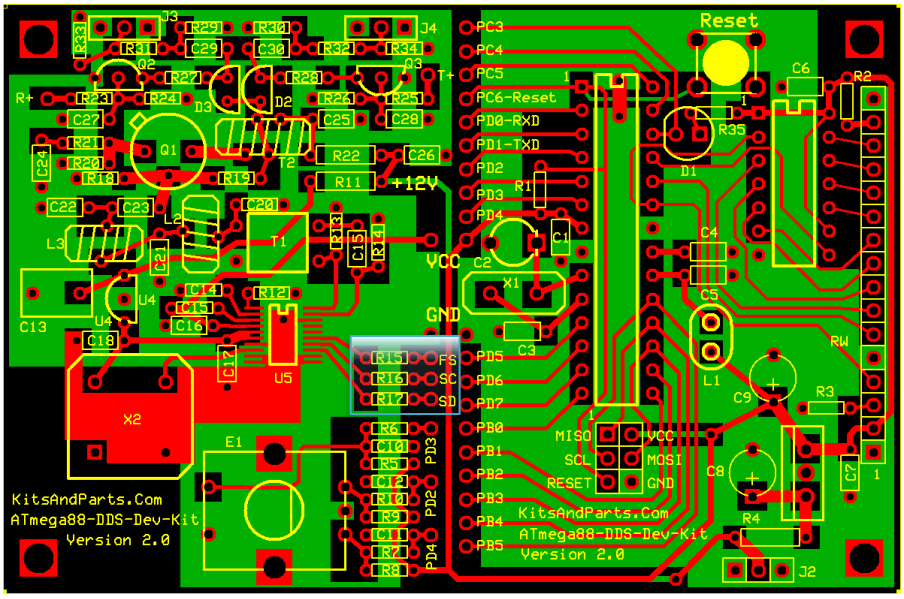
The Raspberry Pi is a credit card size computer, built for learning and fun. In addition to the USB and Ethernet ports, it can connect to other hardware, such as a DDS, through its general-purpose IO (GPIO) interface.

But why use a Pi?

- You want remote/internet-based control
- You want to reprogram your DDS VFO
- You have a Pi and don't know what to do with it
- You like to tinker, like me.

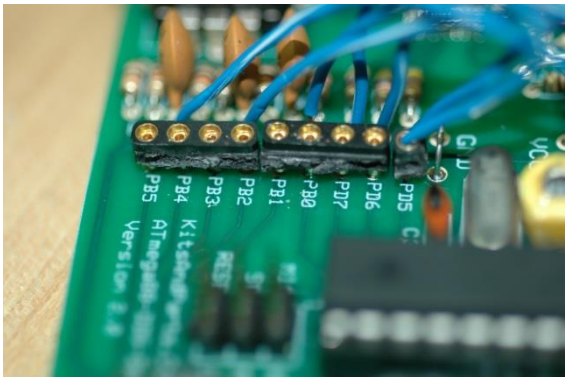
If you like learning by doing, get out your DDS and Pi. In this tutorial, the DDS I will be using is the AD9834 from Analog Devices. The DDS and analog circuitry are part of the "DDS-Development-Kit" by W8DIZ at kitsandparts.com. Other DDS boards will likely require different interfacing and programming, but the concepts will be the same.

2) PREPARING THE DDS BOARD



The DDS board is really two boards in one: the left half of the board contains the DDS and analog output circuitry; the right half is the ATmega88 microcontroller and its supporting components. The center of the board is literally two rows of holes allowing connections between the two. Nearly all of the ATmega88 I/O lines are brought out for our use. But for this project, we will ignore the right half of the board and concentrate on the DDS input lines, highlighted in the blue box. There are three data lines here, marked 'FS' for FSYNC, 'SC' for SCLK, and 'SD' for SDATA. Normally these lines are jumpered to the microcontroller PD0, PD1, and PD5 lines. Remove the three jumper wires.

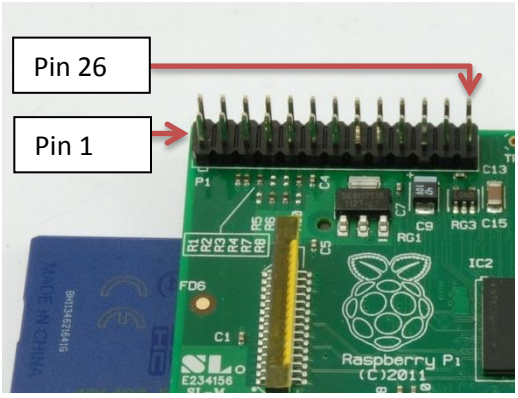
On my board I put 'machine-pin' IC sockets in the PCB holes where the jumpers should go. These work great with regular wire jumpers. You could also use male or female headers, depending on how you want to connect to them. You choose.

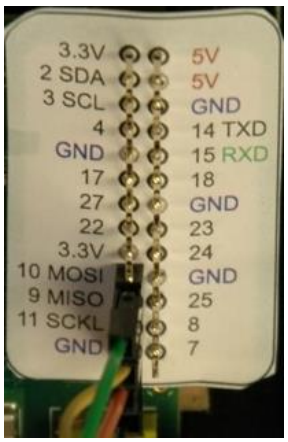


The final connection we'll need is ground. You can take this from the ground jumper, just above the DDS lines. You can even remove the LCD display and use pins 1 or 16 on the female header.

3) THE GPIO CONNECTOR

The GPIO connector is a block of 26 male pins in the upper-left corner of the Raspberry Pi board, arranged 2x13. Pin 1 is lower-left pin (marked P1) and pin 26 is upper-right. The bottom row are odd- numbered 1 though 25, and the top row are even- numbered 2 through 26. The pins are spaced 0.1" apart. Search "26p female





connector" to find mating connectors, which cost about $0.50 each. With a little care, even oversized, 40-pin computer ribbon cables can be attached. This project only needs four wires, so I used individual female-to-male breadboard jumber wires.

Keeping track of these pins can be tricky, so I put a paper label on my pins. See the 'raspberry leaf' at doctormonk.com to get a copy of this really helpful device. We will use GPIO lines 9, 10, 11 and ground, all conveniently located together (pins 19, 21, 23, and 25).

4) MAKING THE CONNECTIONS

Connect your raspberry pi to the DDS using four jumper wires. Simple! I used female-to-male jumpers, which are handy for any breadboard work with the pi. On my board, the AD9834 uses a 5V supply, but it tolerant of the 3.3V logic from the Pi, even through the inline 4.7K resistors. Note that the reverse is not true: the Pi will not tolerate 5V logic inputs. We are OK here because the Pi is output only.

DDS	Raspberry Pi
FSYNC (input)	GPIO 9 (output)
SCLK (input)	GPIO 10 (output)
SDATA (input)	GPIO 11 (output)
GND	GND

5) SOFTWARE

The most popular programming language for the Raspberry Pi is Python. I must admit, before I got my Pi, I'd never heard of Python. But it isn't hard to learn. Python shares many common elements other popular languages, such as C and Java. I have written a few tutorials on programming the Pi, which are on my website at w8bh.net. I also have a tutorial on using the GPIO ports, which you can find at w8bh.net/pi/GPIO1.pdf

Let's focus on the DDS requirements. The three inputs to the AD9834 give us an SPI-compatible interface. It should be possible (and perhaps simpler) to configure your Pi for SPI and use the Pi's SPI interface. I have not done that yet, and so I will use a simple "bit-banging" approach.

Check out the AD9834 datasheet for the signal timing requirements. Now put it away, 'cause you won't need it! GPIO signals from the Pi, controlled by a Python application, are much slower than the minimum timing requirements of the DDS.

Data is sent to the chip serially, one bit at a time, in 16 bit chunks. Each bit is clocked into the chip by briefly taking the SCLK line low. Here is the algorithm:

1. Before data transfer, FSYNC and SCLK should be high
2. Take FSYNC (data enable line) low
3. Put data bit on the SDATA (serial data) line
4. Pulse the SCLK (serial clock) low. Data is input on the high-to-low transition
5. Repeat steps 3 & 4 for all 16 bits
6. Return FSYNC to logic high

Look at the original source code for Diz' development kit, and you will see that his SHIFT16 routine follows these steps exactly. Let's try it in Python. Here it is:

```
SetPin(FSYNC,0);                #enable data input to DDS
for b in range(16):             #loop for 16 data bits
       value = data & 0x8000    #look at left-most bit
       SetPin(SDAT,value)       #puts its value on data line
       data <<= 1               #shift data bits to left
       data &= 0xFFFF           #limit data to 16 bits
       PulseClock()             #clock in the data bit
SetPin(FSYNC,1)                 #brink FSYNC high after word sent
```

Look at the outer shell first: Take FSYNC low, do a loop for 16 data bits, and return FSYNC high. The loop itself is a little trickier. We need to send that data right-to-left. That is, the most significant bit first, the least significant bit last. To get a value of the most significant bit, we do a logical AND with binary value of that bit: 1000.0000.0000.000 (hex 8000). The result of that is either $8000 if the bit is one, or zero if the bit is zero. SetPin will put a logic one on the SDATA line for any nonzero number, including $8000. We are ready to clock in the data now. I'm sure I could put the call to PulseClock right after SetPin, but I've delayed it for two instructions just to give the data a bit more time to settle. Not necessary, but can't hurt.

Finally, shift the data word one bit to the left, using the '<<' operator. This moves the next bit into the most-significant bit position. Normally we would write it like this: data = data << 1. But when the variable appears on both sides of the equals sign we can use the abbreviation: data <<= 1. Similarly, data = data +1 would be data +=1.

PulseClock will briefly take the SCLK line low:

```
SetPin(SCLK,0)                  #bit clocked on high-low transition
SetPin(SCLK,1)                  #no delay since python is slow
```

In assembler you would add a short timing delay to lengthen the clock pulse, but they are unnecessary in Python.

Now we have duplicated the serial data routine from the original development kit. How do we convert a desired frequency into a set of AD9834 instructions?

Let's start with an example. If we want a frequency output of 7.040 MHz, we'll need a register value of 18,897,856. See my tutorial on AD9834 programming for more information. Python gives us floating point arithmetic, so converting from frequency to register value is very easy:

```
factor = 2.68435456              #assumes 100 MHz oscillator
reg = int(hz * factor)           #convert Hz to DDS register value
```

Variable reg now contains the necessary register value, but the AD9834 requires us to break up the 28-bit value into two 14-bit halves. To get the lower half, we do a logical AND operation with a number that contains 14 one's (binary 0011.1111.1111.1111 = hex 3FFF). To get the upper 14 bits, we shift the register value 14 bits to the right. The right-shift operator is '>>'

```
#divide 28-bit register value into 14-bit halves
regLo = reg & 0x3FFF
regHi = reg >> 14
```

Finally, the AD9834 requires each 14-bit half to be prefixed by a two-bit command. The command to put this data into the reg0 register is '01'. We can do that logically, by doing a local OR operation with the binary value 0100.0000.0000.0000 (hex 4000).

```
#now prefix each half with reg0 command
regLo |= 0x4000
regHi |= 0x4000
```

Now the variables regLo and regHi contain 16 bits of data. They are ready to send to the DDS, lower half first. The function Shift16 will serially send 16-bits of data to the DDS.

```
#send both halves to the DDS
Shift16(regLo)
Shift16(regHi)
```

The complete program script is given at the end of this article. I call mine 'dds'. Under raspian/linux you make a file executable by changing the file permissions. Then run it, prefixing the filename with a period & slash:

```
➢  Chmod +x dds

➢  ./dds 7.040
➢  Setting DDS = 7.040000 MHz
➢  ./dds 7030
➢  Setting DDS = 7.030000 MHz
➢  ./dds 7
➢  Setting DDS = 7.000000 MHz
➢  ./dds 30m
➢  Setting DDS = 10.106000 MHz
```

That's it. Enable VFO output with a wire jumper from +R or +T to Vcc. The script allows for Hz, KHz, MHz, or band select inputs. You can enter frequencies manually, or call the script from your own applications. Put your Pi on a network, and run the script from computers across the room or miles away. Have Fun!

6) PYTHON SCRIPT for DDS CONTROL:

```
#!/usr/bin/python

##########################################################################
#
#   A Python script for controlling a DDS VFO from a Raspbery Pi.
#
#   Author :   Bruce E. Hall, W8BH    <bhall66@gmail.com>
#   Date   :   23 Apr 2013
#
#   The DDS used is the Analog Devices AD9834, part of the
#   DDS-Development kit from KitsAndParts.com.
#
#   For more information, see w8bh.net
#
##########################################################################


import RPi.GPIO as GPIO
import sys


#There are three DDS to RPi GPIO connections (plus ground):
#DDS     GPIO
FSYNC  = 9
SCLK   = 10
SDAT   = 11
pins   = [FSYNC,SCLK,SDAT]

#set up a few band defaults = standard qrp frequencies
dict = {'80m':3.560,'40m':7.040,'30m':10.106,'20m':14.060,'15m':21.060}


##########################################################################
#
#   Low-level routines
#   These routines access GPIO directly
#

def InitIO():
    GPIO.setmode(GPIO.BCM)
    GPIO.setwarnings(False)
    for pin in pins:
        GPIO.setup(pin,GPIO.OUT)

def SetPin(pinNumber,value):
    #sets the GPIO pin to desired value (1=on,0=off)
    GPIO.output(pinNumber,value)


##########################################################################
#
#   DDS routines:
#
#

def InitDDS():
       #Start with FSYNC & SCLK lines high
       SetPin(FSYNC,1)
       SetPin(SCLK,1)

def PulseClock():
```

```
	#pulses the DDS serial clock line LOW
	SetPin(SCLK,0)                    #bit clocked on high-low transition
	SetPin(SCLK,1)                    #no delay since python is slow

def Shift16 (data):
	#Send data word = 16 serial bits to DDS
	#FSYNC is low for duration of data transfer
	#SCLK is pulsed low to clock in each bit

	SetPin(FSYNC,0);                  #enable data input to DDS
	for b in range(16):               #loop for 16 data bits
		value = data & 0x8000     #look at left-most bit
		SetPin(SDAT,value)        #puts its value on data line
		data <<= 1                #shift data bits to left
		data &= 0xFFFF            #limit data to 16 bits
		PulseClock()              #clock in the data bit
	SetPin(FSYNC,1)                   #brink FSYNC high after word sent

def ResetDDS():
	Shift16(0x2100)                    #this is DDS reset command

def OutputA():
	Shift16(0x2000)                   #sends DDS register0 to output

def OutputB():
	Shift16(0x2800)                   #sends DDS register1 to output

def GetFrequency(s):
	#convert string input to a frequency value, in Hz
	#Examples:
	#  Band defaults:  '30m' --> 10,106,000
	#  KHz inputs:    '7040' -->  7,040,000
	#  MHz inputs:   '7.040' -->  7,040,000
	#  Hz inputs: '10106000' --> 10,106,000

	s = dict.get(s,s)                 #look for band defaults
	try:                              #dont have a cow with typos
		x = float(s)              #convert string to number in Hz
		if (x<99):                #assume 1-2 digit numbers are MHz
			x *= 1000000
		elif (x<99999):           #assume 3-5 digit numbers are KHz
			x *= 1000
	except ValueError:
		x = 0
		print "Bad input"
	return x

def SetFrequency(hz):
	#input = frequency in Hz;  result: sends command to DDS
	#this routine converts requested Hz into a DDS register value
	#the conversion factor depends on the master oscillator input
	#My DDS uses a 100 MHz, so factor = 2^28 steps/100,000,000 Hz

	factor = 2.68435456               #assumes 100 MHz oscillator
	if (hz<0) or (hz>30000000):       #my DDS is good for 30 MHz max.
		print "input out of range"
	else:
		reg = int(hz * factor)    #convert Hz to DDS register value

		#divide 28-bit register value into 14-bit halves
		regLo = reg & 0x3FFF
		regHi = reg >> 14
```

```
                #now prefix each half with reg0 command
                regLo |= 0x4000
                regHi |= 0x4000

                #send both halves to the DDS
                Shift16(regLo)
                Shift16(regHi)


##########################################################################
#
#   Main Program
#

InitIO()
InitDDS()

if len(sys.argv)<2:
       print "Need desired frequency (Examples: 7.040, 7040, 40m)"
else:
       value = GetFrequency(sys.argv[1].lower())
       if value>0:
              print "Setting DDS = %f MHz" % (value/1000000)
              ResetDDS()
              SetFrequency(value)
              OutputA()

#   END  ###################################################################
```