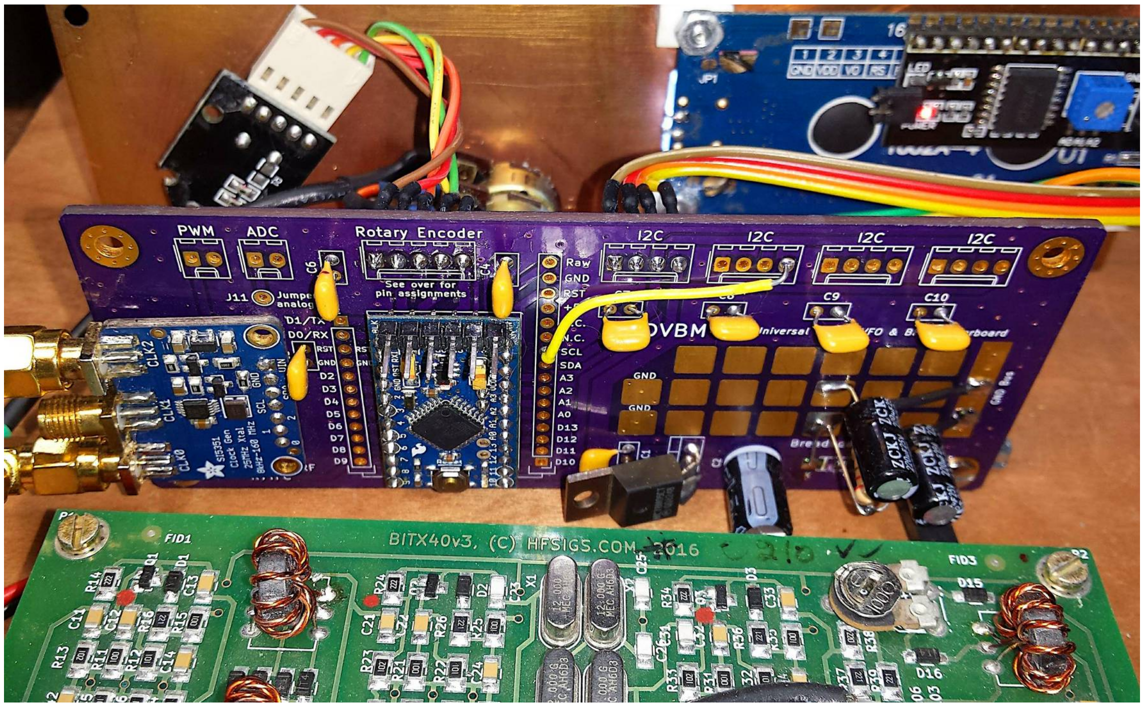
mostly DIY RF™

UDVBM-1

Universal Digital VFO and BFO Motherboard


Early prototype of the UDVBM-1 used with a BITX40 module

User's Guide

See the glossary in Appendix B for underlined words or terms

The UDVBM-1 is a universal motherboard for Arduino-compatible micro-controller modules and an Adafruit-compatible Si5351 PLL clock module that can be used in any rig for either the VFO, the BFO, or (usually) both. Additionally, the I2C, SPI, and 1-wire digital buses of the MCU are brought out to ports along the board edge to use for inputs (rotary encoders, buttons, and switches) and outputs (LCDs or other digital displays, panel indicators, buzzers, and cooling-fan control). To top it all off, the MCU's analog-to-digital converter (ADC) inputs and pulse-width modulation (PWM) outputs are made available as well.

Currently, we have three versions of the UDVBM-1 available: one for the Arduino Pro-Mini, one for the Arduino Nano, and one for Seeed Studio's (yes, three "e"s) family of Xiao (pronounced she-ow) MCU modules (based on the SAMD21, the RP2040, or the ESP32

MCU chips). In addition to Arduino’s C/C++ language, the Xiaos can be programmed in MicroPython or CircuitPython.

The UDVBM-1 offers these features:

- Board-edge connections (for use with or without headers) for I2C bus, SPI bus, 1-Wire bus, rotary encoder, ADC input, and PWM output.
- A manageable size, neither too big nor too small.
- Onboard linear regulator with back filter to keep digital noise out of the rig’s main supply.
- Available for Arduino ProMini or Nano, or for the family of Seeed Studio's Xiao microcontroller modules.
- Free, open-source, and ready-to-use software included. Of course, you can also program the Arduino with your own software.
- The Xiao version can be programmed in Arduino C/C++, MicroPython, and CircuitPython.
- Versions for the Arduino include a small breadboard area of isolated pads for additional user circuitry.
- Available as a bare board, as a kit of PCB and parts to use with your own Arduino (or Xiao) and Si5351 modules, or as a fully-populated and tested assembly.

SYSTEM OVERVIEW

The UDVBM-1 is a “motherboard” supporting three important sub-systems:

- A 12V to 5VDC (3.3V for the Xiao version) power supply filtered to prevent both incoming and outgoing noise (i.e., it keeps digital noise generated from or through the UDVBM-1 from getting on the 12VDC line).
- A micro-controller module (Arduino or Xiao) to take user input to set the frequency and other operating characteristics (phase, drive level, etc.) of each channel of the Adafruit Si5351 clock generator module (or compatible), and to readout the results and status on the user's front panel (an LCD or OLED character display, or a TFT or similar graphics display).
- Adafruit Si5351 module (or compatible) using the Silicon Labs Si5351A PLL clock-generator IC. The chip itself requires 3.3VDC power, but it has an on-board LDO regulator and level shifter with automatic MOSFET switching to allow for 5VDC operation.

The micro-controller (MCU) module and the Si5351 module are connected on the board to the Arduino's I2C "two-wire" data bus and no further connections are necessary if you will be using genuine-branded Arduino Pro Minis or "clones" sold by Adafruit or Sparkfun. Currently-sold clones made by HiLetGo, AITRIP, or other off-shore suppliers will require two jumper connections in order to use pins A6 and A7. See Appendix D for details.

Additional connection ports to the I2C bus are provided along the top edge of the motherboard. These could be used for serial "backpack" LCD displays, other displays that communicate via the I2C bus, or any other device that can use the bus such as PCF8574 eight-bit I/O expander chip modules. These provide eight lines of I/O for SPST switch inputs, panel-mounted indicator LEDs, cooling-fan relay control, and a large number of other uses.

The motherboard also provides easy-to-use connections for the Arduino's SPI "three-wire" serial bus and for the relatively-uncommon Dallas/Maxim "1-Wire" bus. Though I2C has become the most common microcontroller serial bus in use, there are some important devices that use SPI, including several OLED and TFT displays. There are also I/O expander modules for SPI based on chips such as the MCP23017 (16-bits) and the MAX7301 (28-bits). There are Arduino libraries available for the expanders.

The Dallas/Maxim 1-Wire bus has limited applications, but it could still have uses in radio equipment. The Adafruit DS2413 1-Wire breakout module, for instance, provides two general-purpose I/O pins that could come in handy for a couple of front-panel or relay connections. Adafruit also offers the DS18B20, a 1-Wire digital temperature sensor in a TO-92 housing. It could easily be epoxied to the heat sink of a transmitter finals amp for monitoring or alarm purposes.

In addition to these digital I/O buses, The UDVBM-1 motherboard has a dedicated rotary encoder port with "A" and "B" inputs (on some encoder modules labeled "data" and "Clk"), an input for the push-button contact available on most encoders, as well as +5V and GND connections. As with all the ports along the top and right-hand edges of the motherboard, the encoder port can be used with 0.1"-spaced headers (facing either side of the PCB) or with wires soldered directly to the PCB pads. Ribbon cabling is particularly handy for that purpose.

To top off the UDVBM-1's complement of break-out connections, there is a two-channel 10-bit ADC (12-bit with the Xiao version)input port and a single PWM output connection. One or both ADC channels could be used to read the position of a potentiometer, the voltage level of a battery, or for an analog one-wire bus (see Appendix C). If filtered with a simple RC network, the PWM output could be used to drive an annunciator (buzzer or small speaker) or even an analog meter movement. Without filtering, it could be used to vary the apparent brightness of a front-panel LED. MCU module pin assignments for each of these buses and

I/O ports are silk screened on the back of the UDVBM-1's PC board for easy reference (Fig. 1).

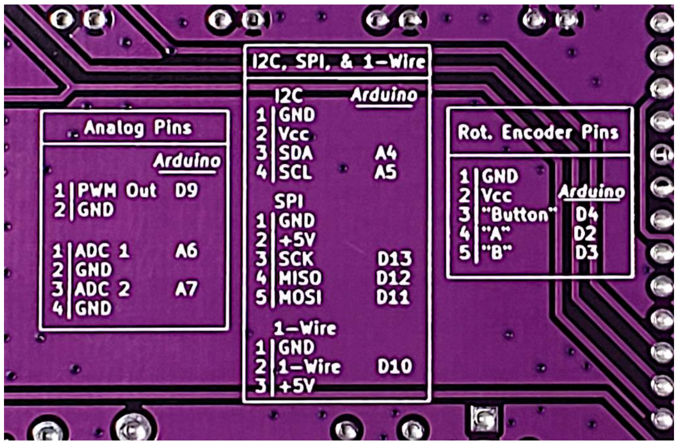


Figure 1: Silkscreen legend of Arduino-pin assignments.

Just to be clear, these pin assignments cannot be altered on the UDVBM-1, either because they're hardwired in the microcontroller module itself (e.g., the I2C pins), or because they are physically connected by traces printed on the motherboard PCB. The ADC1 and ADC2 pins (A6 and A7) were chosen because they cannot be used as regular digital pins, and though any digital pin can be used for *software-based* (and resource-wasting) pulse-width output, D9 is the Pro-Mini's and Nano's *hardware-based* PWM pin. The rotary-encoder and 1-Wire pins were chosen for ease of PCB layout.

This leaves some pins unassigned and available for use via easy access to the Arduino breakout headers. It's worth noting, though, that with even-easier access to the digital buses and to analog breakouts, directly accessing the Arduino pins would not ordinarily be necessary or even desirable. They're there, though, should you want them. One possible use would be be to add a second rotary encoder to your application.

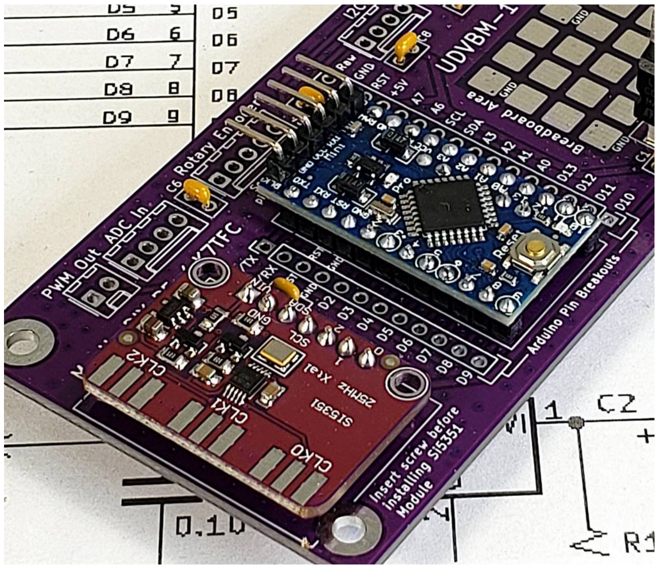
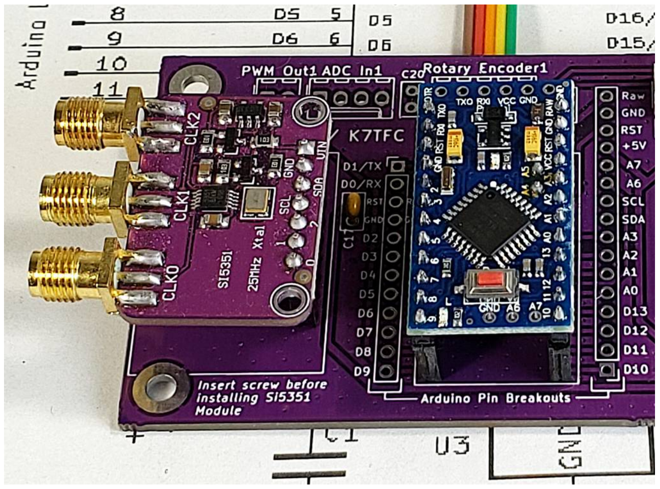
The best way to establish these pin assignments in your Arduino "sketch" (the Arduino term for "program") is to use the #define directive near the top so they're usable globally (i.e., anywhere in your sketch).

```
#define I2C_SDA       A4
#define I2C_SCL       A5
#define SPI_SCK       13
#define SPI_MISO      12
#define SPI_MOSI      11
```

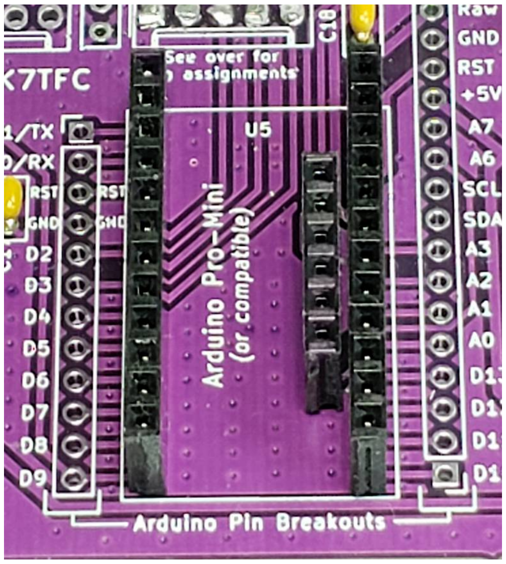
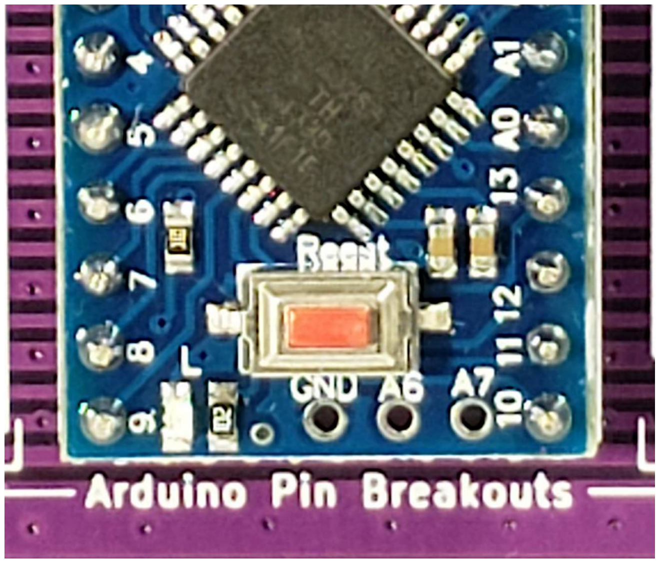
```
#define 1WIRE         10

#define ENCODER_A     2
#define ENCODER_B     3
#define ENCODER_BTN   4

#define PWM_out       9
#define ADC1pin       A6
#define ADC2pin       A7
```

MOUNTING THE ARDUINO AND SI5351

There are a two basic ways of mounting the Arduino or Xiao and Si5351 modules to the UDVBM-1. Both depend on the pin headers that are usually included with them when purchased. These should be soldered with the pins pointing down. The modules can then be soldered directly to the motherboard (Fig. 2), or they can be plugged into female socket headers which have been soldered into place (Figs. 3 & 4). In general, the latter is the recommended method. A burned-out or otherwise malfunctioning Arduino or Si5351 module could then be easily replaced (or removed for use in another project). Directly-soldered modules, however, would be very difficult to remove (certainly the Arduino would be), but they result in a lower profile and they are somewhat more rugged.

 Fig. 2: Modules soldered directly to UDVBM.	 Fig. 3: Modules inserted into female socket headers (easily removable).

If you are using an Arduino Pro-Mini clone with pins A6 and A7 along the bottom of the PCB, you will need to run jumpers from these pads to the corresponding pads of the Arduino breakout header in order to use them with the ADC port of the UDVBM-1. On some clones

 Fig. 4: Female socket headers. Note inner header for pins A4 - A7.	 Fig. 5: Note alternate location of A6 and A7 on some Pro-Mini clones. Requires jumpers to Arduino pin break-out pads.

pins A4 & A5 are also along the bottom. Jumpers will absolutely be necessary in that case since those are Pro-Mini's I2C I/O and clock pins. Be aware, also, that even if the clone has any of the A4 - A7 pins in the "official" position, they may be slightly off the grid used by Arduino, requiring those clone pins to be bent inward a few degrees to fit into their holes on the UDVBM or female sockets.

Mounting the two modules on the Arduino Nano version will require some special considerations in order to avoid blocking access to the Nano's mini-USB (*not* micro-USB) socket. If you are using female headers for either the Nano or the Si5351 module (or both), you can simply unplug either to gain easy access to the socket.

If you wish to avoid repeated unpluggings for the purpose of reprogramming the Nano, you can mount the Si5351 lower in its female header by removing the plastic strip that holds its male header pins together before they're soldered in place (Figs. 6 and 7), and by clipping about 0.10" (2.5mm) off the ends of the pins (Fig. 8). On the Nano, you would leave the plastic strips in place (Fig. 9). The Si5351 module would then sit lower on the motherboard than the Nano. Additional clearance for the USB connector can be obtained by closely clipping the tops of the pin header on the Si5351 (Fig. 10), though this may be unnecessary if you are not using SMA connectors on the module. As a last resort, you may need to shave down the bottom of the mini-USB cable shroud for the needed clearance (Fig. 11).

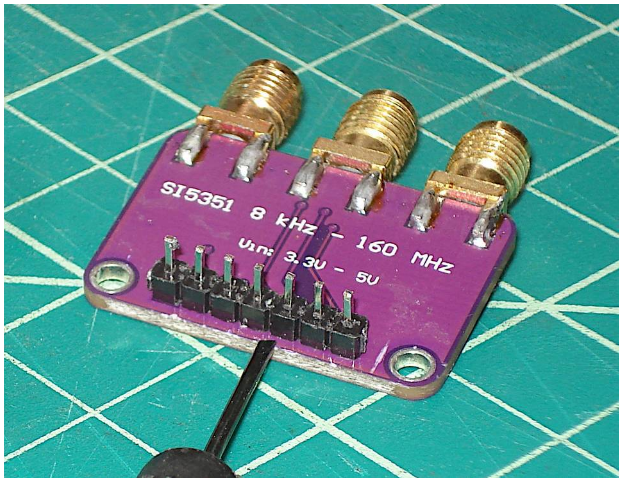


Fig. 6: Prying plastic strip from Si5351 pins.

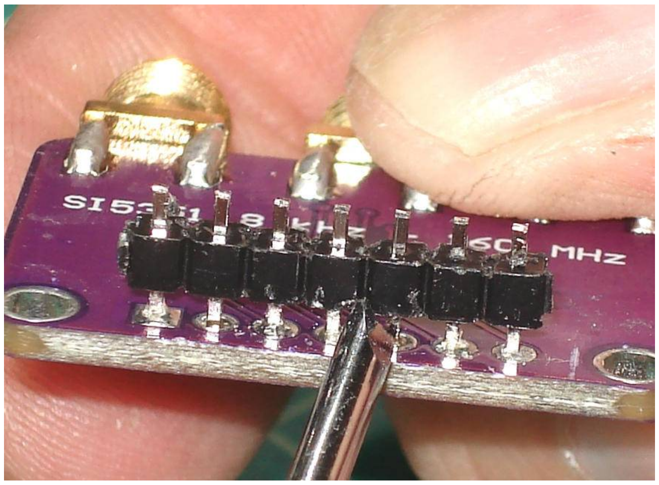


Fig. 7: Prying plastic strip from Si5351 pins.

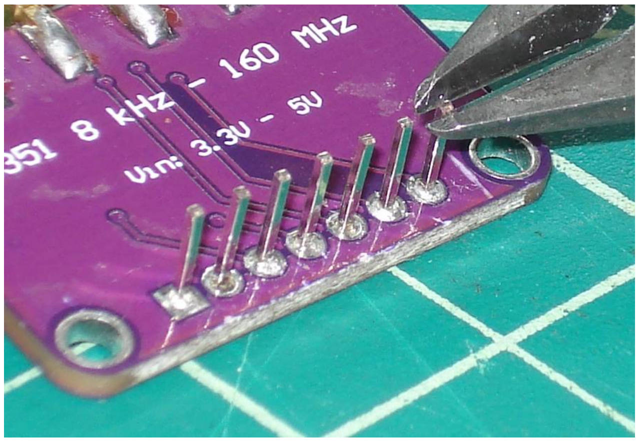


Fig. 8: Clip about 0.1" from Si5351 pins.

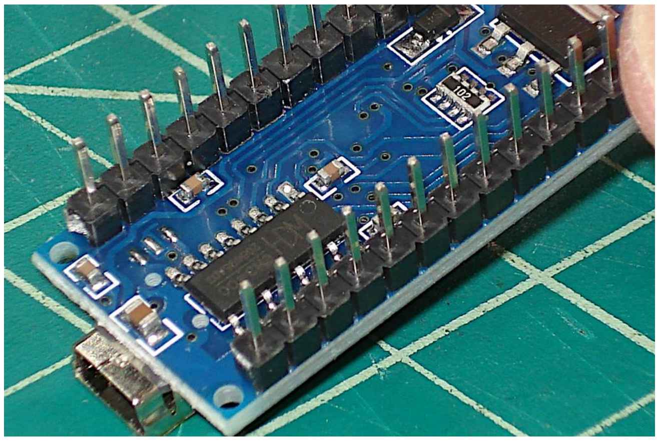


Fig. 9: Leave plastic strips on Nano pins.

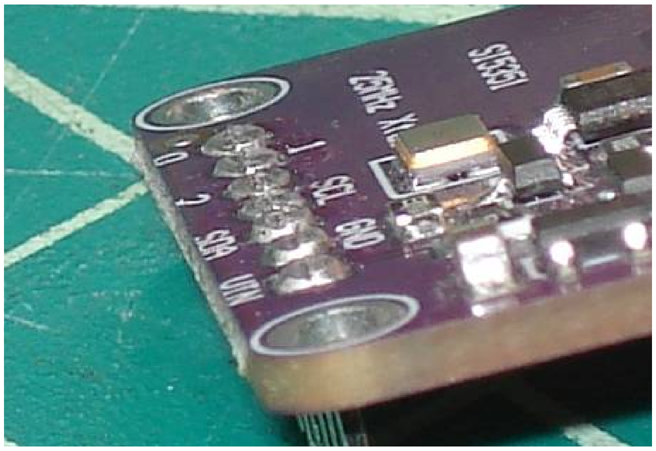


Fig. 10: Closely clip Si5351 pins on top.

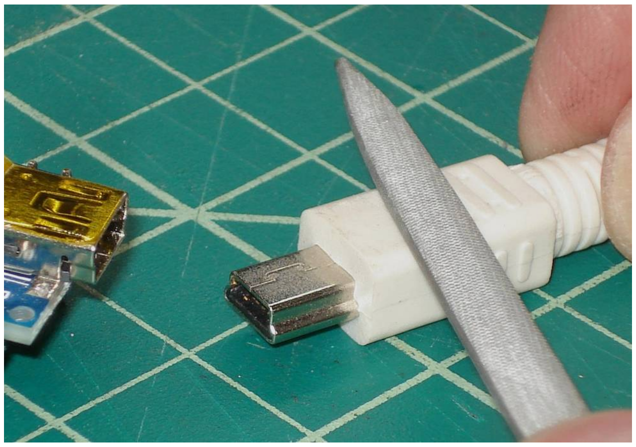


Fig. 11: If needed, shave down USB shroud.

MAKING CONNECTIONS TO THE UDVBM-1

All of the pin headers and ports on the UDVBM are on 0.1" (2.54mm) centers. Male pin headers (e.g., "break-apart" type), female socket headers, Molex-type connectors, or any other connectors having 0.1" spacing are suitable. Additionally, wires can be directly soldered to the header pads, and both connectors or wires soldered to either side of the board (ribbon cable is well-suited to this). Which you use will depend on how and where you mount the UDVBM-1, and on its relation relative to the rest of the rig. See Figs. 12 & 13 for examples.

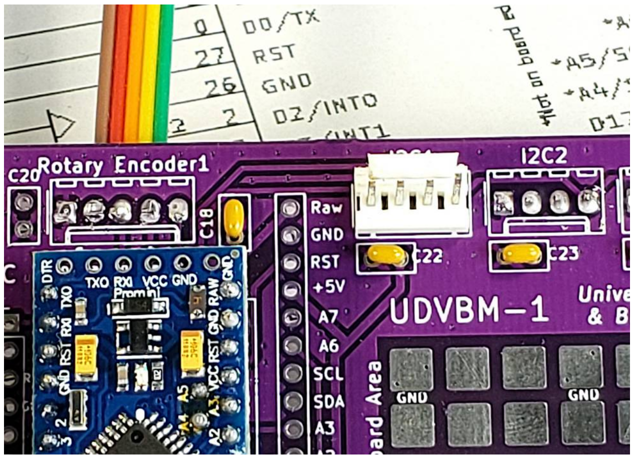


Fig. 12: Molex-type locking connector. Any type of connector with 0.1" (2.54mm) pin spacing is suitable.

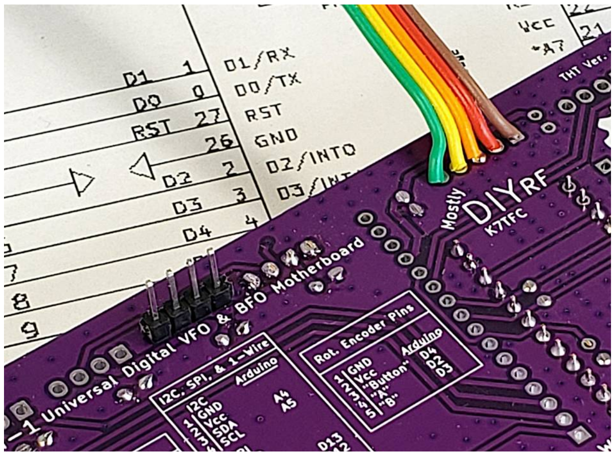


Fig. 13: Male pin header (left) and directly-soldered ribbon cable (right) on the back of the UDVBM-1 board.

For RF connections to the Si5351, you can solder small-gauge coax (such as RG174 or RG316) directly to the clock-output pads. If you do this, it's best to keep the unshielded center conductor as short as possible, and to solder the braided shield to the adjacent GND pad (Fig. 14).

Alternately, you can solder board-edge-type female SMA connectors (not included) to the same pads (Fig. 15). These have the theoretical virtue of being true 50Ω connections, but it's worth noting that the PCB traces on both the Adafruit and clone modules are not designed as 50Ω transmission lines. Whatever impedance discontinuity may exist because of soldered vs. SMA connections will have no discernible impact on practical radio applications.

SOFTWARE

There are no special requirements associated with programming the UDVBM-1's Arduino Nano. As noted above, it has a built-in USB interface for serial communication, including

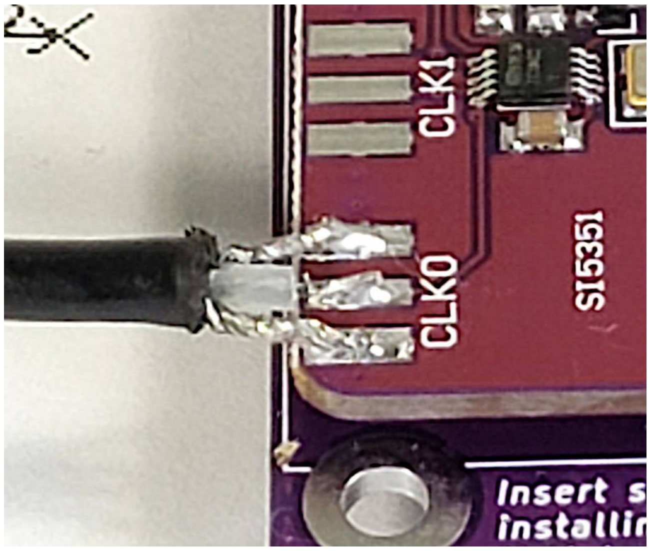


Fig. 14: RG174 soldered directly to pads.

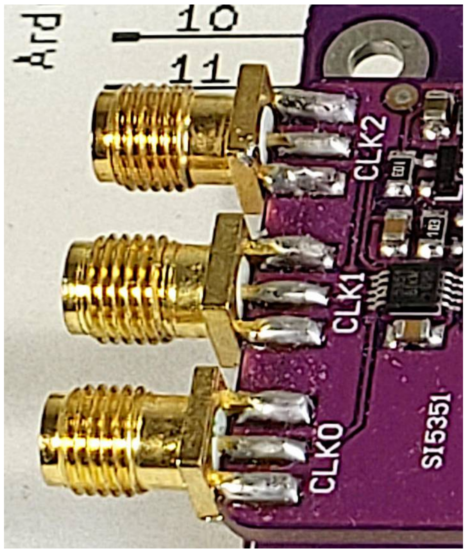


Fig. 15: SMA board-edge connectors.

programming. For the Pro-Mini version, though, you must use a USB-to-Serial conversion module to upload your sketches. These use an FT232RL, CP2102N, CH340, or similar chip to convert the USB protocol to serial RX/TX signals. These modules commonly have a 6-pin female header that plugs directly into the right-angle male programming header on the Pro Mini. Fortunately, one can be had for less than ten dollars, and they are useful for a number of other Arduinos and for other systems that need a USB-to-serial interface.

Seeed Studio Xiao

Seeed Studio was founded by a former Intel engineer in Schenzhen, China, and it's in its second decade as an open-source hardware manufacturer and developer. It introduced the family of Xiao devices in 2019. In addition to its very-small size, the Xiao can be programmed in either Arduino C/C++ or in Python (either MicroPython or CircuitPython). For Arduino's C/C++, you can program it using the regular IDE if you add the Xiao to the IDE's "board manager" (for detailed instructions, see the "Setting up the Arduino IDE for the Seeeduino XIAO" section of Bill Jamshedji's https://dronebotworkshop.com/seeeduino-xiao-intro/). All of the code libraries you would use with Arduinos are usable with the Xiaos.

The Xiao is also programmable using MicroPython or Adafruit's CircuitPython. Unlike the compiled C/C++ code of the Arduinos, both Python variants are interpreted by run-time firmware you install in the micro-controller's flash memory. Once done, the Xiaos appear just like USB drives on your computer, and there is no compile process before uploading your code. There is also no dedicated IDE--you simple write your programs in any text editor, save them to the Xiao, and run the code directly.

Though compiled code is theoretically faster than interpreted code, in practice (for our purposes certainly) it is a meaningless distinction. Current versions of Python and other interpreted languages are used successfully for a number of speed-critical real-time applications including facial recognition and robotics. Partly this is due to improvements made in data structure handling and byte-code based JIT ("just in time") compiling, but mostly because of multi-core 32-bit processors, processor clock speeds several times faster than generations of previous devices, and by the exceptionally cheap, compact, and fast working memory (SRAM) now used in very-inexpensive MCUs.

The slowest of the Xiao variants is based on the SAMD21 MCU chip. This is a dual-core, 32-bit device running at 48MHz, and sporting 32KB of working SRAM and 256KB of flash (program) memory. By contrast, the classic Arduino Uno (and the Nano, Pro-Mini, and some other Arduinos) have single-core 16MHz MCUs with 2KB of SRAM and 32KB of flash memory.

The Xiao RP2040 uses the same dual-core MCU chip as the Raspberry Pi Pico (the RP2040 chip was, in fact, developed by the Raspberry Pi Foundation) running at 133MHz with 264KB of SRAM and up to 16MB of flash memory. The fastest of the Xiaos is based on the ESP32-C3 MCU running at 160MHz with 400KB of SRAM and 4MB of flash. It additionally has built-in WIFI and Bluetooth connectivity.

Not only will these faster devices easily handle an interpreted language such as Python--and do so much faster than compiled C/C++ code on the older Arduino devices--but they are fast enough to allow for simpler "polling" methods of taking user input, rendering the "interrupt" methods unnecessary for any conceivable amateur-radio use (certainly for the UDVBM-1's uses).

Program or "Sketch" Structure

For either the Arduinos or the Xiaos, depending on the type of peripheral input and display devices you use (rotary encoder, LCD, OLED, TFT, etc.) you will need to "include" the appropriate code "libraries" that greatly simplify interactions with external hardware. It *is* possible to directly address, read, and write to hardware registers, and to "bit-bang" communication on the three serial data buses available on the UDVBM-1. Except as an educational exercise, though, it's totally absurd to even try. Professional programmers don't "reinvent the wheel," and neither should you. Arduino and Python libraries are mature and well-tested, and they have been optimized over numerous revisions.

In order to control the on-board Si5351 module, you'll use the Etherkit Si5351 library by Jason Milldrum NT7S. This is available using the Arduino IDE's Library Manager or from Github (https://github.com/etherkit/Si5351Arduino). Jason's user guide (the README.md file at the Github site) is excellent and very thorough.

As with any Arduino program (a.k.a., "sketch"), the code you develop will be divided into at least three parts. At the top are the "global" entries where you'll "include" libraries, and set global variables, constants, and "defines." These lines will be executed only once when the Arduino starts up. Then comes the setup() function that can contain many of the same things as the global lines, but because it's defined as a function, variables used within it are "local" and not global. It, too, will only run once.

The third major part of a sketch is the loop() function, so named because everything in it will run over and over--sequentially--branching off conditionally to other functions, before returning to the loop to pick up where it left off. Much of what takes place in the loop is executed behind the scenes in functions defined in the libraries you've included. You can, of course, define you're own functions. Like the library functions, they will be executed (a.k.a., "called") from within setup() or loop(). Due to the nature of Arduino's implementation of C/C++, the setup() and loop() functions are predefined (though of course empty) and required. With Python, you'd create these functions yourself.

For radio applications, the most common thing you'll do in the loop is to "tune" the rig according to input from a rotary encoder, numeric-keyboard array, or some other frequency-change device, and then to visually indicate the frequency and other settings (LSB or USB, for instance) on a panel display or computer screen. There are two principle ways of accomplishing this: using "polling" or employing "interrupts."

Polling

The polling method relies on checking the state or contents of a hardware register in the Arduino's microcontroller each time through the main loop and then acting accordingly. When you turn the rig's rotary-encoder knob, for instance, functions in the encoder library you're using will raise a "flag" and then set a register variable you can "poll" (check the value of) from within the loop. The flag will indicate there's been a change since the last pass though the loop, and the register variable will store the direction of change as either "increment" or "decrement." The library functions then "expose" this event and state change in a simple variable you can test in a conditional statement to either increase the frequency or decrease it. If nothing has changed, you don't want anything done at all, and the loop goes back to the top and does it all over again.

In a typical radio application, this polling methods works fine as long as there's not too much else going on in the loop() function. If there is, the loop will run too slowly and it may miss changes that have taken place in the hardware functions. If you are using a Xiao MCU, this will never happen. With an Arduino, it is possible but still very unlikely. With each pass, the Arduino will read the state of the encoder flag/register, increment or decrement the frequency of the VFO clock of the Si5351, display the corresponding digits on the display, then go back to the top of the loop again.

A typical loop in this kind of application will take a tenth of a millisecond or less to execute (i.e., ten-thousand times a second or more). You probably can't turn the dial fast enough to get ahead of the polling and subsequent changes that are happening each pass through. In reality, most of the time there's no change to process at all. Unless you're actively turning the dial or inputting some other kind of change, you want the main loop to just spin its wheels.

This is particularly important because you also want to minimize the introduction of digital noise impulses that can radiate from interconnecting wires and that can be conducted on your power supply line to the rest of the radio's circuits. For this reason, you should poll for inputs at the very top of the loop, and if there are no changes, you should skip everything else--the Si5351 and the display--and start the loop over again. Once set at the desired frequencies, the Si5351 will not require an update until there's a change. There's no reason to continually send the same frequency data each time through the loop.

Likewise, and LCD or other digital displays will continually show the last data they received until they receive an update. You shouldn't send them the same data over and over, either. If for some reason (nostalgia, for instance) you want to use 7-segment LED displays, don't multiplex or "charlieplex" them. Instead, use a serial interface with latching buffers (or use a second MCU) that will hold what they last received.

Interrupts

The second basic method of taking and processing user input involves *interrupts*. These are of two kinds: *external* and *internal*. External interrupts (sometimes called *hardware interrupts*) are built into the hardware of the microcontroller chip itself. Triggering an external interrupt on either the rising edge or falling edge of an input signal immediately stops the microcontroller from processing the main loop it's running, save its place and variables, and then to run another function called an *interrupt service routine* (ISR). In our use case, the ISR would process the rotary-encoder input, updating the Si5351 and the frequency display, and then returning control to the main loop. On ATMEGA328-based Arduinos (Uno, Nano, Pro-Mini, etc.), there are two external-interrupt pins: 2 (D2) and 3 (D3). These can be used for ordinary digital I/O as well as for external interrupts.

So-called *internal* interrupts--also called *pin-change* interrupts--will also respond to an input trigger, but their implementation depends entirely on software and not on hardwired structures in the microcontroller chip itself. This makes them generally slower to respond than external interrupts. In addition, they also require more processing in their associated ISR.

With the two external interrupt pins, it is easy to determine which pin was triggered and therefore what ISR to jump to, but the internal-interrupts are *port* based. The Arduino bootloader and IDE generally obscures the hardware-port structure of the ATMEGA328, but in this case it's necessary to pay attention to it. Ports are groups of pins that can be read or set as a byte of 8 bits. On the Pro-Mini used in the UDVBM-1, Port D, for instance, consists of pins 0 (RX), 1 (TX), and pins 2 through 7. If you are using one or more of them for pin-change interrupts, you have to read the entire port as a byte and use *bitwise* operations to find out which of the individual pins triggered the interrupt and then to call the appropriate ISR in response.

Because the use of interrupts is not usually necessary in the kind of applications to which the UDVBM-1 would be put, we will not go into additional detail here. For more on interrupts and the Arduino platform, see: <https://www.best-microcontroller-projects.com/arduino-interrupt.html> and <https://thewanderingengineer.com/2014/08/11/arduino-pin-change-interrupts/>

USE IN A RADIO

How you use the UDVBM-1 will depend upon the system architecture and frequency scheme you've chosen. For a typical direct-conversion receiver (a.k.a., "zero IF"), you would use a single clock channel of the Si5351 for the "LO"--the *l*ocal *o*scillator to be mixed with the incoming RF and converted directly to audio.

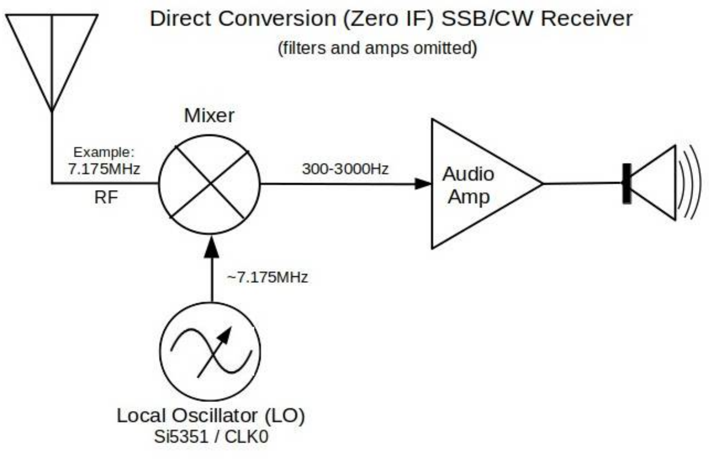


Fig. 10

For a “single-conversion” (more-accurately called single IF) AM or SSB superhet rig, you would use two of the Si5351’s outputs—one (on receive) for the VFO/LO (to mix with the incoming RF), and one for the BFO (to demodulate the IF down to audio). On transmit, the BFO (a.k.a, carrier oscillator) signal is mixed with the audio input from a mic or sound card and converted to the IF frequency. After filtering and amplification, the IF is then mixed with the VFO signal, amplified, and sent out the antenna.

It has become customary with Adafruit-compatible Si5351 modules to use clock-channels “0” and “2” for single-IF schemes, avoiding channel 1 in order to minimize possible “cross-talk” between channels. This is sensible and without cost, though not really necessary for most amateur-radio uses.

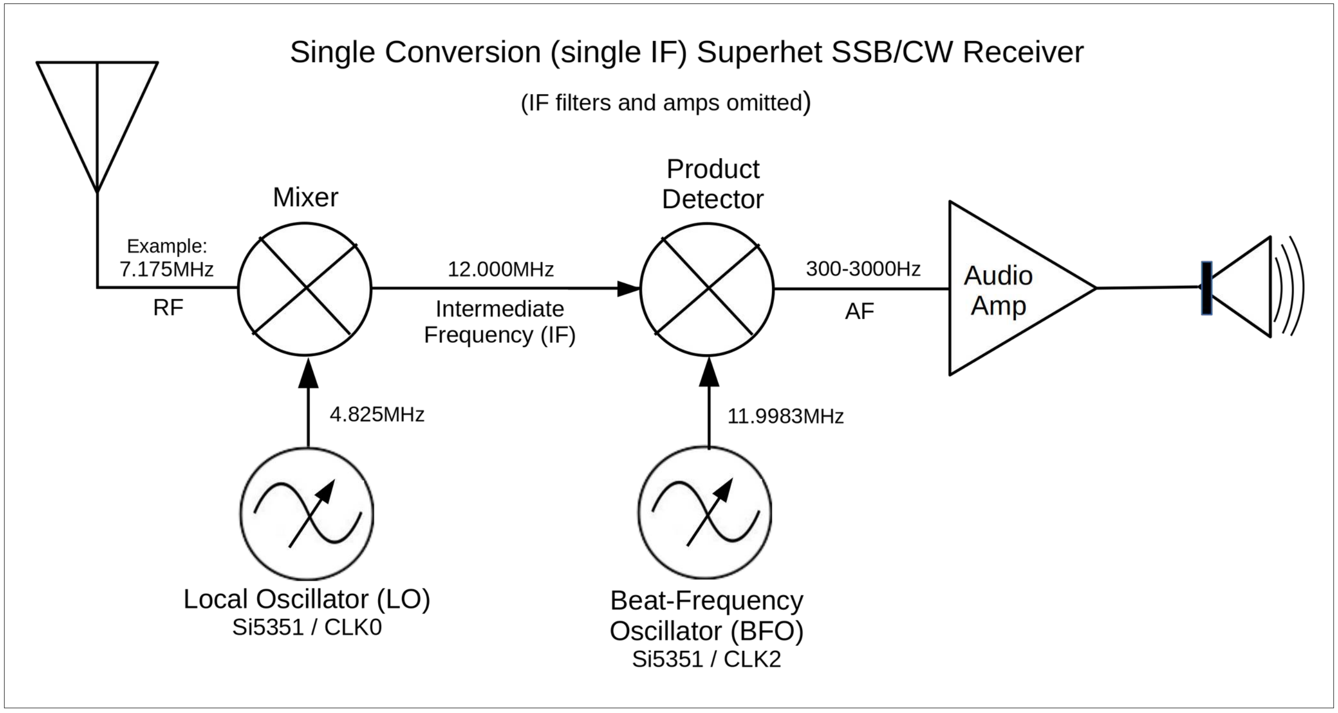

Single Conversion (single IF) Superhet SSB/CW Receiver
(IF filters and amps omitted)
Mixer
Product Detector
Example: 7.175MHz
RF
12.000MHz
Intermediate Frequency (IF)
300-3000Hz
AF
Audio Amp
4.825MHz
11.9983MHz
Local Oscillator (LO)
Si5351 / CLK0
Beat-Frequency Oscillator (BFO)
Si5351 / CLK2

Fig. 11

For dual-IF frequency schemes, all three clock channels are used: one for the VFO (1st IF) conversion, one for the 2nd IF conversion, and one for the BFO. Such schemes are used for a variety of purposes, including using a single SSB filter with multiple bands by employing a 1st IF higher than any one of the bands. This is the architecture used in Ashar Farhan's μBITX.

If implementing a dual-IF scheme with the UDVBM-1, you could use CLK0 and CLK1 for the VFO and 2nd local oscillator, and use CLK2 for the BFO. In this case, you would not be able to physically separate the clock signals on the Si5351 module. This is not really a problem since whatever cross-talk that may be present between the clock channels (measurable only with lab-grade instruments) has no practical impact on amateur-radio communications. It is, in fact, completely swamped out in a receiver or transceiver by the high band noise now characteristic of all-but the most remote places in the 21st century.

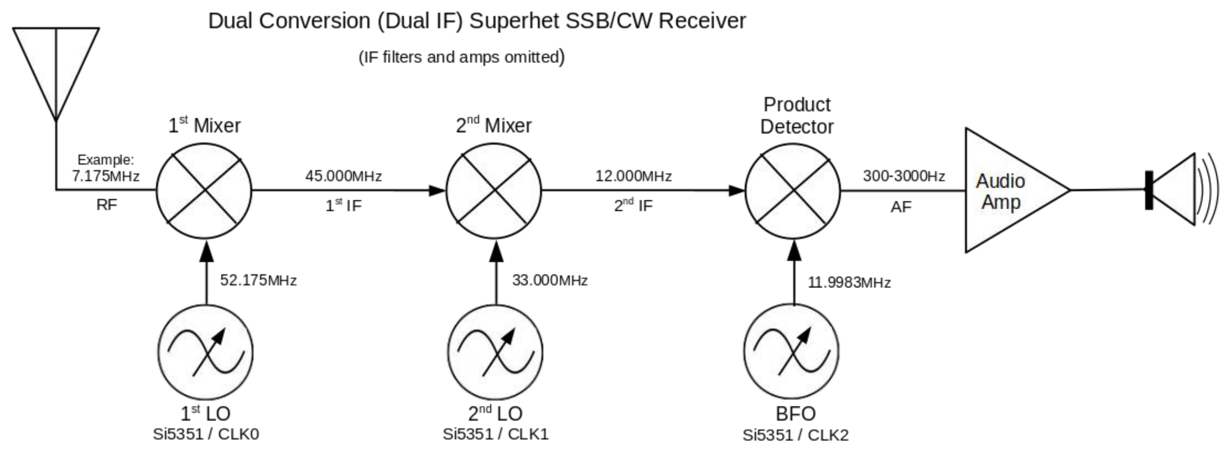


Fig. 12

Historically, there have been triple-IF schemes, with the third IF usually at a very-low frequency (less than 100KHz). They are still sometimes used for specialized purposes today. To use the UDVBM-1 for such schemes, you would need an additional Si5351 module connected to one of the I2C ports, and you would need to set the additional module to a different I2C address than the built-in one.

For full discussions of mixer and IF schemes, see:
en.wikipedia.org/wiki/Intermediate_frequency
web.ece.ucsb.edu/~long/ece145a/Introduction_to_Receivers.pdf
analog.com/media/en/training-seminars/design-handbooks/Basic-Linear-Design/Chapter4.pdf

Si5351 Output Drive Level

Because you are using the outputs of the Si5351 to drive the LO input of the mixer(s) and the product detector (itself a kind of mixer), your Arduino program will have to set the proper drive level of those outputs depending on the requirements of the mixers you're using. An output-drive level of 2mA (the lowest available) will result in approximately +7dBm into 50Ω. 8mA (the highest available) will result in approximately +14dBm.

According to manufacturers such as Mini-Circuits, under-driving diode-ring double-balanced mixer (DBM) LO inputs will result in higher intermodulation distortion (and lower dynamic range) than if properly driven. However, especially for manufactured miniaturized DBMs such as the ADE-1 and SBL-1, care should be taken not to greatly exceed specified drive levels. Doing so can damage their fine transformer wires or small diodes. A homebrewed Schottky-diode-ring DBM, though, can be driven at the highest level the Si5351 can produce with much better performance than lower drive levels. If ordinary silicon-junction diodes (e.g., 1N914, 1N4148, etc.) are used, buffering the Si5351 output with a 5V CMOS inverter for higher LO drive may give better results.

For more on LO drive levels, see:

H.P. Walker, "Sources of Intermodulation in Diode-Ring Mixers" *The Radio and Electronic Engineer* 46: 247-255 (https://www.scribd.com/document/457060559/Sources-of-Intermodulation-in-Diode-ring-Mixers-H-P-Walker).

Hewlett Packard, "The Schottky Diode Mixer" (Application Note 995, 1986).

Shankar Joshi, "Taking the Mystery out of Diode Double-Balanced Mixers" *QST* (Dec. 1993) 32-36.

A Few Final Thoughts

Though *Mostly DIY RF* encourages and hopes to facilitate hands-on homebrewing, we also believe practical trade-offs can be made along the way. While completely-analog VFO and BFO oscillators are interesting and fun projects in and of themselves, they are also difficult to keep stable and linear in their operation. We see no meaningful conceptual problems associated with adopting an easy-to-use and very-stable digital solution such as the UDVBM-1 while also employing traditional analog approaches to other parts of a receiver or transmitter: filters, amplifiers, mixers, and (de)modulation methods.

In homebrewing as in life, one choses one's battles carefully, knowing some are worth fighting, some not. Determining which is which cannot be prescribed--it's an entirely personal descision, and it can change from project to project.

Appendix A

Example Arduino Code

Here's what a bare-bones sketch might look like (to download from GitHub repository, go to: https://github.com/mostlydiyrf/UDVBM-1). The sketch can be further simplified by removing the functions (including statements calling them) used to save frequency settings, set tuning increments, and position increment cursors on the 16x2 LCD display.

```
// Arduino "sketch" for use with UDVBM-1 VFO/BFO. Version. 6.21.2022
// (C) T.F. Carney (K7TFC). For use under the terms of the
// Creative Commons Attribution-ShareAlike 4.0 International license.
// See https://creativecommons.org/licenses/by-sa/4.0/legalcode

//************************************
// SET Si5351 CALIBRATION FACTOR ***** // Using NT7S's calibration sketch:
const uint32_t cal_factor = 149600;    // File-> Examples-> Etherkit Si5351-> si5351_calibration
//************************************

// *** EEPROM byte addresses for VFO and BFO memory ***
// Addr Use
// 0-3: VFO address write tally
// 4-5: VFO address
// 6-9: BFO frequency
// 10-13: VFO frequency (1)
// 14-17: VFO frequency (2)
// 18-1020: Subsequent 4-byte segments
//
// EEPROM bytes have a maximum number of reliable write cycles. 100,000 is a safe number. Because
// the VFO frequency data is saved each time it's incremented (by rotary encoder), it might not
// take long to use up a single block of 4 byte used to store the 32-bit integer. A few hundred
// cycles could be used in a single operating/listening session. Consequently, this sketch abandons
// each block of 4-byte VFO addresses every 100,000 cycles, beginning with addresses 10-13. Sub-
// sequent addresses (14 - 17, 18 - 21, etc.) are saved as a 2-byte integer in addresses 4 & 5. On
// a ATMEGA328P-based Arduino, there are 1024 bytes available allowing for 254 4-byte blocks to use
// for 100,000 writes before being abandoned for the next one. A total of 25,400,000 write cycles.
// This should be sufficient.
//
// Because the BFO frequency will be changed *much* less frequently (perhaps as few as a dozen
// times), a single block of 4 bytes can be used for the life of the system. The addresses used for
// the write tally and VFO address bytes can also remain the same.
```

```
#define NOP __asm__ __volatile__ ("nop\n\t")  // NOP needed to follow "skip" labels.
#define relayPin 6
#define PWMout 9
#define i2cSDA A4
#define i2cSCL A5
#define spiMISO 12
#define spiMOSI 11
#define spiSCK 13
#define encoderBTN 2
#define encoderA 4
#define encoderB 3
// For custom LCD characters:
#if defined(ARDUINO) && ARDUINO >= 100
#define printByte(args)  write(args);
#else
#define printByte(args)  print(args,BYTE);
#endif

//========================================
//=============  LIBRARIES ===============
//========================================
#include <EEPROM.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>    // Author: Schwartz
#include <si5351.h>               // Author: Jason NT7S. V.2.1.4
#include <ClickEncoder.h>         // Author: 0xPIT, but using the ver. from soligen2010's repo
#include <TimerOne.h>             // Required by ClickEncoder

//========================================
//======== GLOBAL DECLARATIONS ===========
//========================================
uint32_t lastUsedVFO;
uint32_t lastUsedBFO;
uint32_t maxWriteCycles = 100000;
int steps[] = {10,100,1000,10000}; // Tuning steps to increment frequency (in Hz) each detent.
int step = 1000;                   // Step on startup.
byte stepsPerNotch = 4;
byte detent = 0;                   // Tuning encoder. THIS *MUST* REMAIN A *BYTE* TYPE.
byte encoder = 0;
byte single_click;
byte btn = 0;                      // Tuning encoder button.
byte downcaret[] = {0x00,0x00,0x00,0x00,0x11,0x0A,0x04,0x00};
```

```
//========================================
//============ INSTANTIATIONS ============
//========================================
Si5351 si5351;
ClickEncoder tuningEncoder(encoderA,encoderB,encoderBTN,stepsPerNotch);
LiquidCrystal_I2C lcd(0x27, 16, 2);

////=====================================
////******* FUNCTION: saveInt ********
////=====================================
void saveInt(int address, int number) {

  EEPROM.write(address, number >> 8);
  EEPROM.write(address + 1, number & 0xFF);
}

////=====================================
////******* FUNCTION: readInt ********
////=====================================
int readInt(int address) {

  byte byte1 = EEPROM.read(address);
  byte byte2 = EEPROM.read(address + 1);
  return (byte1 << 8) + byte2;
}

////=====================================
////******* FUNCTION: saveUint32 ********
////=====================================
void saveUint32(int address, uint32_t number) {

  EEPROM.write(address, (number >> 24) & 0xFF);      // These lines code the 32-bit variable into
  EEPROM.write(address + 1, (number >> 16) & 0xFF);  // 4 bytes (8-bits each) and writes them to
  EEPROM.write(address + 2, (number >> 8) & 0xFF);   // consecutive eeprom bytes starting with the
  EEPROM.write(address + 3, number & 0xFF);          // address byte. Using write() instead of
                                                     // update() to save time and because update()
}                                                    //  won't help  save write cycles for *each*
                                                     // byte of the 4-byte blocks.
```

```
////===================================
//// ***** FUNCTION: readUint32 *******
///===================================
uint32_t readUint32(int address) {

  return ((uint32_t)EEPROM.read(address) << 24) +      // These lines decode 4 consecutive eeprom
         ((uint32_t)EEPROM.read(address + 1) << 16) +  // bytes (8-bits each) into a 32-bit
         ((uint32_t)EEPROM.read(address + 2) << 8) +   // variable (starting with the address byte)
         (uint32_t)EEPROM.read(address + 3);           // and returns to the calling statement.
}                                                      // Example: uint32_t myNum = readUint32(0);

///=======================================
////***** FUNCTION: saveBFO  **************
////=======================================
void saveBFO() {

  saveUint32(6, lastUsedBFO);

}

////=======================================
////***** FUNCTION: saveVFO  ***************
////=======================================
void saveVFO() {

  uint32_t tally = readUint32(0);
  int vfoAddress = readInt(4);

  if(tally < maxWriteCycles) {
    tally++;
    goto skip;
  }
  else {
    vfoAddress = vfoAddress + 4;
    tally = 0;
  }

  skip:

  saveUint32(vfoAddress, lastUsedVFO);
  tally++;
  saveUint32(0, tally);

}
```

```
////========================================
////***** FUNCTION: lcdClearLine ***********
////========================================
int lcdClearLine(byte lineNum) {

    lcd.setCursor(0,lineNum);
    lcd.print("                ");
    lcd.setCursor(0,lineNum);
}

////========================================
////***** FUNCTION: displayFreqLine ********
////========================================
void displayFreqLine(byte lineNum, uint32_t freqValue) {

  char padspace = 32;  // ASCII for blank space.
  String valueStr;
  String lineTag;
  String khzOnly;
  String decKHZ;

  if(lineNum == 0){
    lineTag = "VFO:";
  }
  else if(lineNum == 1) {
    lineTag = "BFO:";
  }
  else {                 // Only 2-line displays allowed.
    lineNum = 0;         // Add more else-if statements for 4-line displays.
    lineTag = "Err!";
  }

  valueStr = String(freqValue);
  khzOnly = valueStr.substring(0, valueStr.length() - 3);                   // Takes all but the
last 3 digits.
  decKHZ = valueStr.substring(valueStr.length() - 3, valueStr.length()-1);  // Takes the last 3
digits and cuts
                                                                            //  off the last digit
(0-9Hz).
                                                                            // I.e., keeps only
tenths and hundredths
                                                                            // of KHz (hundreds and
tens of Hz).
  lcdClearLine(1);
  lcd.setCursor(0, lineNum);
  if(valueStr.length() == 8) {
    lcd.print(lineTag + khzOnly + "." + decKHZ);  // For frequencies >=10,000KHz (no leading blank
space).
  }
  else {
    lcd.print(lineTag + padspace + khzOnly + "." + decKHZ);  // For frequencies <=9,999KHz (adds
leading blank space).
  }

  Serial.print("ValueStr length: "); Serial.println(valueStr.length());

  lcd.setCursor(13, lineNum);
  lcd.print("KHz");

} // End displayFreqLine()

////=======================================
////***** FUNCTION: displayStepCursor ******
////=======================================
void displayStepCursor(int Step, byte lineNum) {

  char stepCursor;

  if(lineNum == 1) {             // lineNum is the line on which the cursor is to appear.
    stepCursor = 94;             // ASCII "up" caret.
  }
  else if(lineNum == 0) {
    stepCursor = 0;              // 0 is the LCD custom-character location for a "down" caret
  }
  else {
    stepCursor = 'X';            // To indicate error of lineNum.
  }

  switch (Step) {
    case 10:
      lcdClearLine(lineNum);
      lcd.setCursor(11, lineNum);
      lcd.print(stepCursor);
      break;
    case 100:
      lcdClearLine(lineNum);
      lcd.setCursor(10, lineNum);
      lcd.print(stepCursor);
      break;
    case 1000:
      lcdClearLine(lineNum);
      lcd.setCursor(8, lineNum);
      lcd.print(stepCursor);
      break;
    case 10000:
      lcdClearLine(lineNum);
      lcd.setCursor(7, lineNum);
      lcd.print(stepCursor);
      break;
  }
}

////========================================
////****FUNCTION: timerISR ***********
////========================================
void timerIsr() {

  tuningEncoder.service();      // Used by ClickEncoder for timer-based interrupts.

  }

////========================================
////****** FUNCTION: bfoFreq() *******
////========================================
void bfoFreq() {

    lastUsedBFO = readUint32(6);
    int bfoStep = steps[0];
    uint32_t bfoValue = lastUsedBFO;
    btn = 0;
    String valueStr;

    lcdClearLine(0);
    displayFreqLine(1, bfoValue);          // Parameters: LCD line (0 or 1), frequency value.
    displayStepCursor(bfoStep, 0);

  while (btn != 4) {                       // Loop until a long press-and-release of encoder button
    // Reset button variables for this pass through the loop.
    int flag = 0;
    int item = 0;
    int encoder = 0;

    // Read tuning encoder and set Si5351 accordingly
    ////////////////////////////////////////////////
    btn = tuningEncoder.getButton();
    detent = tuningEncoder.getValue();    // ClickEncoder "gets" can only be done once.
    if (detent == 255) {                  // A (-1) from the encoder rolls back the byte-type
      encoder = -1;                       // 'encoder' byte from 0 to 255. Doing it this way
    }                                     // eliminates otherwise testing for CW or CCW movement.
    else {
      encoder = detent;
    }
    // Skip to end of loop() unless there's change on either encoder or button
    // so LCD and Si5351 aren't constantly updating (and generating RFI).
    if (encoder == 0 && btn == 0) {
      goto skip;
    }
    else {
      bfoValue += (encoder * bfoStep);
      // Si5351 is set in 0.01 Hz increments.
      // "value" is in integer Hz.
      si5351.set_freq(bfoValue * 100, SI5351_CLK2);
      lastUsedBFO = bfoValue;
      saveBFO();
    }
    // LCD display ///////////////////////////////
    ////////////////////////////////////////////
    if (btn == 5 && bfoStep == steps[3]) {
    bfoStep = steps[0];
    displayStepCursor(bfoStep, 0);
    }
   else if (btn == 5 && bfoStep == steps[0]) {
    bfoStep = steps[1];
    displayStepCursor(bfoStep, 0);
    }
   else if (btn == 5 && bfoStep == steps[1]) {
    bfoStep = steps[2];
    displayStepCursor(bfoStep, 0);
    }
   else if (btn == 5 && bfoStep == steps[2]) {
    bfoStep = steps[3];
    displayStepCursor(bfoStep, 0);
   }
   displayFreqLine(1,bfoValue);  //Parameters: LCD line (0 or 1), frequency value.

  skip:
  NOP;
  }

  // At this point (after long press-and-hold of encoder button),
  // restore the VFO display before returning.
  lcdClearLine(1);lcd.setCursor(0,1);
```

```
  displayFreqLine(0,lastUsedVFO);  //Parameters: LCD line (0 or 1), frequency value.
  displayStepCursor(step, 1);

  return;
}

////=======================================
////******** FUNCTION: setup ***************
////=======================================
void setup() {

  lcd.init();
  lcd.backlight();
  lcd.createChar(0, downcaret);

  Serial.begin(115200);
  Wire.begin();

  Timer1.initialize(1000);
  Timer1.attachInterrupt(timerIsr);

  tuningEncoder.setAccelerationEnabled(false);

  // Initialize the Si5351
  si5351.init(SI5351_CRYSTAL_LOAD_8PF, 0, 0);
  si5351.set_correction(cal_factor, SI5351_PLL_INPUT_XO);
  si5351.drive_strength(SI5351_CLK0, SI5351_DRIVE_2MA);

  // Set default VFO & BFO frequencies for first-time use.
  byte vfoMem = (EEPROM.read(10));           // Get first byte of first VFO memory location.
  byte bfoMem = (EEPROM.read(6));
  if((vfoMem >> 6) != 0) {                   // This would be true only if never used before.
    lastUsedVFO = 5000000;                   // First startup default.
    saveUint32(10, lastUsedVFO);
    saveInt(4, 10);
  }
  if((bfoMem >> 6) != 0) {
    lastUsedBFO = 11000000;                  // First startup default.
    saveUint32(6, lastUsedBFO);
  }
  lastUsedVFO = readUint32(10);
  lastUsedBFO = readUint32(6);
  si5351.set_freq(lastUsedVFO * 100, SI5351_CLK0);
  si5351.set_freq(lastUsedBFO * 100, SI5351_CLK2);
```

```
// LCD display
  displayFreqLine(0,lastUsedVFO);          // Parameters: LCD line (0 or 1), frequency value.
  displayStepCursor(step, 1);              // Parameters: displayStepCursor(int Step, byte lineNum)
}

//=======================================
//********* FUNCTION: (main)loop *********
//=======================================
void loop() {

  lastUsedVFO = readUint32(10);

  uint32_t vfoValue = lastUsedVFO;

  // Reset button variables for this pass through the loop.
  int flag = 0;
  int item = 0;
  int encoder = 0;

  // Read tuning encoder and set Si5351 accordingly
  ///////////////////////////////////////////////////
  btn = tuningEncoder.getButton();
  detent = tuningEncoder.getValue();            // ClickEncoder "gets" can only be done once.
                                                // Getting them also clears them.

  if (detent == 255) {                          // A (-1) from the encoder rolls back the byte-type
    encoder = -1;                               // 'encoder' byte from 0 to 255. Doing it this way
  }                                             // eliminates testing for CW or CCW movement.
  else {
    encoder = detent;
   }

  // Skip to end of loop() unless there's change on either encoder or button
  // so LCD and Si5351 aren't constantly updating (and generating RFI).
  if (encoder == 0 && btn == 0) {
    goto skip;
  }
  else {
    vfoValue += (encoder * step);
    Serial.print("vfoValue: "); Serial.println(vfoValue);
    // Si5351 is set in 0.01 Hz increments.
    // "vfoValue" is in integer Hz.
    si5351.set_freq(vfoValue * 100, SI5351_CLK0);
    lastUsedVFO = vfoValue;
    saveVFO();   }
  // LCD display //////////////////
  displayFreqLine(0,lastUsedVFO);           // Parameters: LCD line (0 or 1), frequency value.
  // Button activity on tuning encoder      // ClickEncoder button returns: 4==released (after
                                            // long press), 5==clicked, 6==double-clicked.
   if (btn == 4) {
    bfoFreq();                              // Long press-and-release calls BFO-setting function.
   }
   else if(btn == 5 && step == steps[3]) {  // These else-if statements respond to single (short
     step = steps[0];                       // click) button pushes to step-through
     displayStepCursor(step, 1);            // the tuning increments (10Hz, 100Hz, 1KHz, 10KHz)
   }                                        // for each detent of the tuning encoder
   else if (btn == 5 && step == steps[0]) { // and moves the cursor caret to the corresponding
    step = steps[1];                        // digit. A short click on 10KHz loops back
    displayStepCursor(step, 1);             // to 10Hz. The default step is 1KHz. The bfoFreq()
    }                                       // function uses the same structure.
   else if (btn == 5 && step == steps[1]) {
    step = steps[2];
    displayStepCursor(step, 1);
    }
   else if (btn == 5 && step == steps[2]) {
    step = steps[3];
    displayStepCursor(step, 1);
   }

skip:   // This label is where the loop goes if there are no inputs.
NOP;    // C/C++ rules say a label must be followed by something. This "something" does nothing.

}       // closes main loop()
```

Appendix B

Glossary

ADC Analog to digital converter. A device that takes an input voltage and converts it to a digital number that represents that voltage's level relative to a reference voltage. See <eetimes.com/analog-to-digital-converters> for more information.

AM Amplitude modulation. A method in which an audio (a.k.a. "base-band") signal is used to modulate the signal strength of a radio-frequency carrier. in the process, audio-pitch information is carried by sidebands both above and below the carrier frequency. SSB modulation is a variant of AM.

Arduino A popular open-source micro-controller module based on the ATmega-328P or similar MCU chips. It is program-ed using a dedicated IDE in a variant of the C++ language. See <www.arduino.cc>

BFO Beat-frequency oscillator. Used for reception of a CW signal to make it audible by mixing ("beating") the received signal with the BFO signal offset by an audible frequency (typically 700Hz or so). For SSB reception, the BFO recreates the original carrier (suppressed in transmission) without which the sidebands would be unintelligible. Not used for AM or FM reception.

BNC Bayonet Neill–Concelman connector. A miniature quick connect and disconnect radio-frequency connector used for coaxial cable. It uses a twist-type "bayonet" locking mechanism. Named after its inventors.

Clock, **CLK** In this context, a *clock* is a signal, typically a square wave, that drives the timing of digital logic and communication circuits. The Si5351 was intended for use as a clock-signal generator. We use it here for RF purposes. *CLK* (usually pronounced "clock") is the timing signal for the I2C serial protocol.

Flag A variable or single bit of a byte that is set (or cleared) to record that a particular event in software (or hardware) has occurred (or not occurred). The value of the flag variable or bit is then used in if-then-else statements to determine what the program is to do next.

FM Frequency modulation. In amateur radio, used mostly in the VHF and UHF bands, but also for 10-meter band. The UDVBM-1 can be used as the LO for any FM TX/RX scheme that uses an IF.

I/O A short-hand version of input/output.

I2C Commonly-used version of *IIC*, itself an acronym for *inter-integrated-circuit*. It is a two-wire serial-communication data bus and protocol intended for use between integrated circuits on the same PC board, but also usable up to ~100cm by wire. On the UDVBM-1, I2C is used by the Arduino to communicate with the Si5351 and often with panel displays such as LCDs and OLEDs.

IDE Integrated development environment. In this context, an IDE is used to program the Arduino by providing a code editor, Arduino-specific extensions of the C/C++ language, a manager for Arduino libraries, and a USB-based hardware interface with the microcontroller. The Arduino Foundation provides the most-commonly-used (and open source) IDE free of charge. Other suitable IDEs for Arduinos include the PlatformIO plugin available for Microsoft's VisualStudio.

IF Intermediate frequency as used in superheterodyne-type radio receivers. Superhets use one or more IFs to gain filtering and selectivity benefits, and to allow for "single-signal" reception.

LCD Liquid-crystal display. Typically used for front-panel display of frequency and other information. As implemented for the UDVBM-1, LCD displays would use the I2C serial bus instead of their native 4- or 8-bit parallel input. This is typically implemented through the use of I2C-to-parallel "backpacks."

LO Local oscillator. Typically the name of the oscillator that supplies the signal to be mixed with the incoming RF to create the IF frequency. For this purpose, the LO is usually a variable-frequency oscillator, so it's is sometimes called the VFO, though more often so in a transmitter than a receiver. LO is one of several terms in radio work (amateur and professional) that are sometimes used loosely or inaccurately and are best understood in context.

LSB Lower sideband. The modulation of a radio-frequency carrier results in three separate signals: 1) the carrier itself (which in spite of its name carries no speech or data information), 2) a narrow band of frequencies that extends below the carrier (LSB), and 3) a separate narrow band above the carrier (upper-sideband or "USB"). The frequencies of these sidebands vary according to the instantaneous frequency of the audio signal used to modulate the carrier (typically from 300 to 3000Hz). Other than their position relative to the carrier frequency, these sidebands are identical. Only one is needed to transmit the audio signal. The carrier itself is needed only to reconstitute the original audio, and in an SSB system this is supplied by the BFO in the receiver. The carrier, then, can be suppressed along with the opposite sideband in transmission. This results in great power saving in the transmitter and narrower selectivity in the receiver.

OLED Organic light-emitting diode. Used for small graphics-oriented panel displays, often with one or more colors.

PLL Phase-locked loop. A closed-loop arrangement of VCO (voltage-controlled oscillator), phase detector, and low-pass filter that maintains tight frequency control while still allowing for deliberate frequency changes. PLLs are among the core mechanisms of the Si5351 and similar devices.

Poll To check the value of a variable or input state in a microcontroller such as an Arduino and acting accordingly. Such polling is done within an endless program loop. For switch or rotary-encoder inputs, polling works satisfactorily if the program loop run quickly enough such that it can keep up with human input. Provided this is so, polling is a simpler method of taking and acting on input that the use of interrupts.

PWM Pulse-width modulation. This technique varies the "width" (duration) of the "high" level of a square wave relative to its "low" level. Its typical use is for speed control of DC-operated motors, and for light devices such as LEDs and filament bulbs. When filtered, PWM can approximate an analog voltage (when DC coupled) or an audio signal (when AC coupled).

RF Radio frequency. In radio architecture, RF typically refers to the incoming signal picked up by an antenna and delivered to the front-end of a receiver rather than any signal (such as an IF) that might be considered in the radio-frequency (in contrast to audio) range. RF can also refer to that part of a transmitter signal chain in which modulated IF frequencies have been mixed to produce the signal which, after amplification, will be sent out the antenna.

SPI Serial peripheral interface. It is a three-wire serial-communication data bus and protocol intended for use between integrated circuits on the same PC board, but also usable up to ~100cm by wire. On the UDVBM-1, SPI can be used by the Arduino to communicate with panel displays that require it such as LCDs and OLEDs.

SSB Single sideband. See LSB.

TFT Thin-film-transistor liquid-crystal display. One of several device types used by graphics-based panel displays. These are usual available in much larger sizes than OLED-based displays.

USB Upper sideband. See LSB.

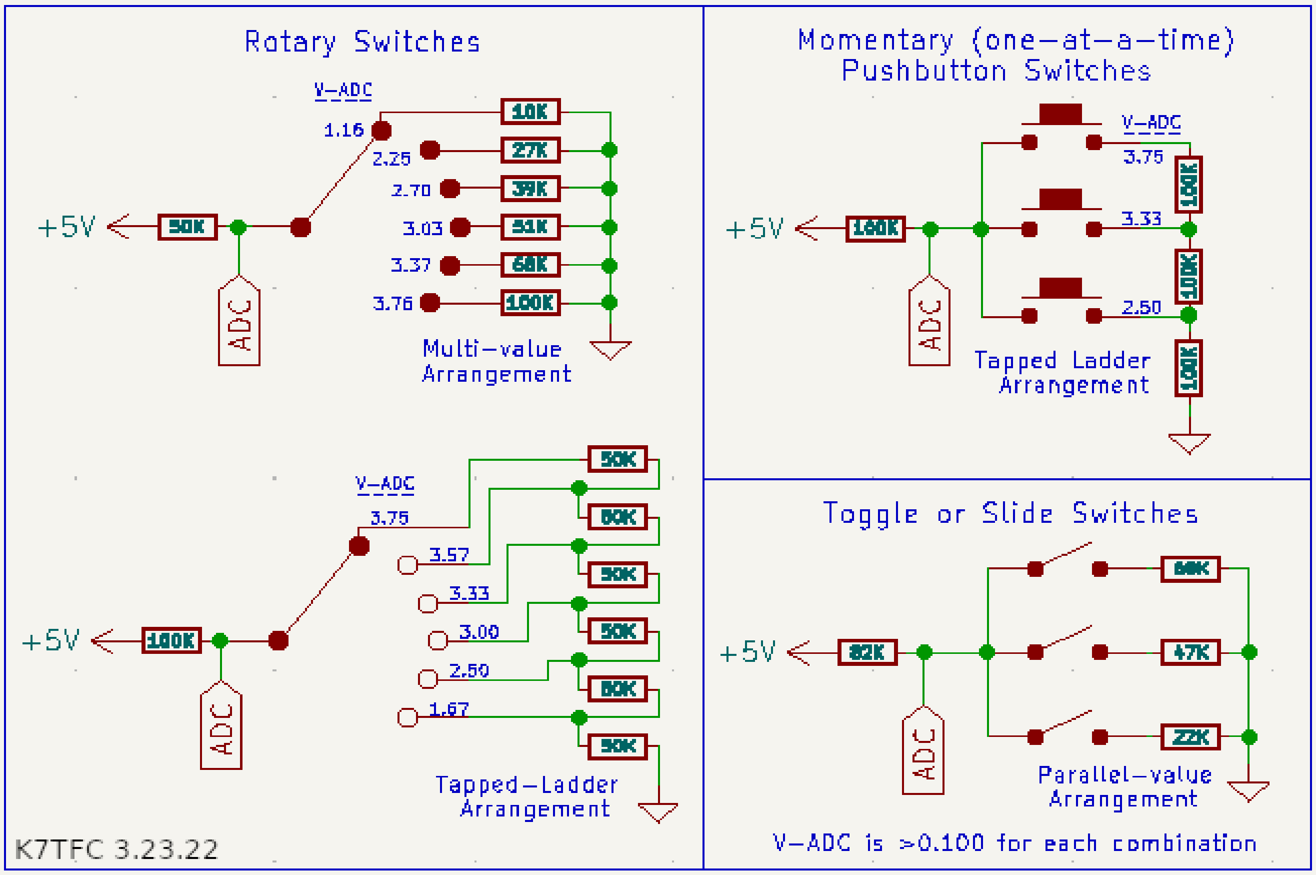
VFO Variable-frequency oscillator. Depending on how it's used, may also be called an LO, or *l*ocal *o*scillator. This is the primary oscillator used to "tune" a receiver and/or transmitter to the desired frequency. In most cases, the frequency of a VFO/LO is mixed with that of a fixed-frequency oscillator to create the desired frequency. With the UDVBM-1, the VFO/LO function is synthesized by the Si5351.

Appendix C

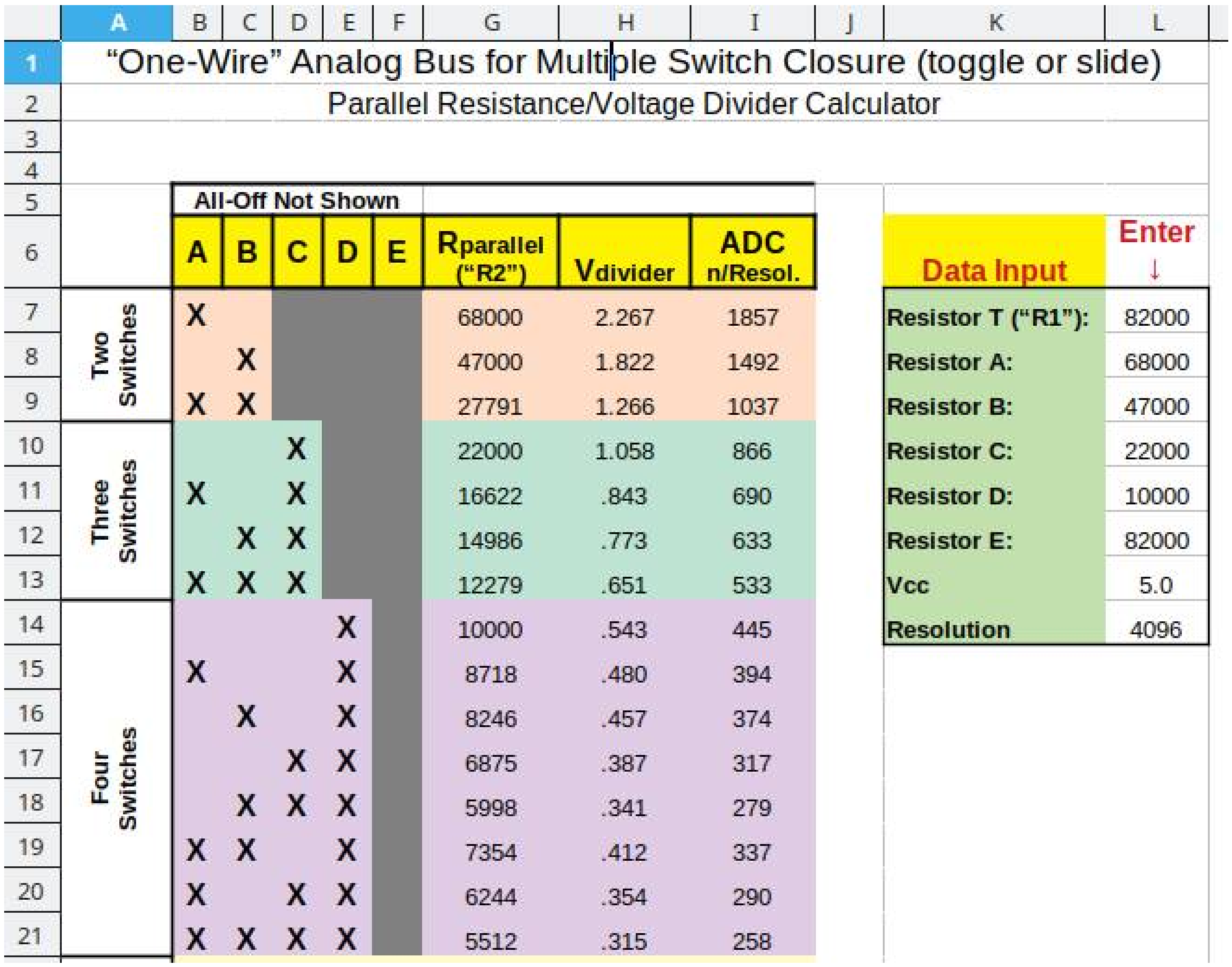
One-Wire Analog Input Bus

A what? Well, that's what I call it. It's another one-wire solution. It's implemented by using a single analog-input pin (ADC) to read a voltage divider, the bottom of which consists of a string of resistors, one at each switch contact that, when closed, grounds the string at that point.

This is almost a no-brainer for rotary or other multi-position switches. It would likewise be a simple matter for individual momentary switches that are always actuated one at a time. It's a little-more complicated for toggle or slide switches, some or all of which might be closed at any given time. By careful resistance choice, though, it's possible to determine which switches are closed by their parallel resistance and hence the voltage read by the analog input. The object would be to select resistances such that each possible switch combination would have a unique parallel resistance the ADC could resolve.



Switches on an Analog Bus (resistors soldered directly to switch lugs)



“One-Wire” Analog Bus for Multiple Switch Closure (toggle or slide)

Parallel Resistance/Voltage Divider Calculator

	All-Off Not Shown							
	A	B	C	D	E	Rparallel (“R2”)	Vdivider	ADC n/Resol.
Two Switches	X					68000	2.267	1857
		X				47000	1.822	1492
	X	X				27791	1.266	1037
Three Switches			X			22000	1.058	866
	X		X			16622	.843	690
		X	X			14986	.773	633
	X	X	X			12279	.651	533
Four Switches				X		10000	.543	445
	X			X		8718	.480	394
		X		X		8246	.457	374
			X	X		6875	.387	317
		X	X	X		5998	.341	279
	X	X		X		7354	.412	337
	X		X	X		6244	.354	290
	X	X	X	X		5512	.315	258

Data Input	Enter ↓
Resistor T (“R1”):	82000
Resistor A:	68000
Resistor B:	47000
Resistor C:	22000
Resistor D:	10000
Resistor E:	82000
Vcc	5.0
Resolution	4096

Sample spreadsheet calculation of multiple switch closure ADC values.
For live spreadsheet, go to: bit.ly/analog_bus
(save a copy to your Google Drive or download)

With an Arduino's 10-bit ADC and a 5V reference voltage, more than three switches becomes a problem of maintaining adequate divider-voltage separation to account for Vcc variation and resistance-value tolerance. However, if you use an external 12-bit ADC (or the internal one for the Xiaos), up to five toggle or slide switches can be attached to the same analog bus with little problem.

The overall object: to reduce the spaghetti and hay wiring required inside a homebrewed rig.

Implementing a one-wire analog bus on the Arduino (or any MCU with a ADC and capable of being programmed in C/C++) is straightforward and simple. In the declarations and global variable section of your sketch (i.e., before the `setup()` function), assign the analog pin you will use to a variable:

```
unsigned int analogBus = A0; // Only positive integers needed
```

You can use any of the analog pins, but avoid A4 and A5 as they are used (on the Nano) for the I2C bus. Then, like any input, `analogBus` is read within the main loop:

```
loop() {
. . .
unsigned int aBus = 0;
aBus = analogRead(analogBus);
. . .
}
```

It is now a matter of using the value of aBus in conditional statements to determine which switch or rotary position is detected. Since resistor tolerances and wiring will result in a range of values for each position, upper and lower values of the range can be established and used to compare with the value of aBus. Since this comparison will be used several times each loop, it's best to create a very-simple function:

```
bool inRange(unsigned int val, unsigned int min, unsigned int max) {
   return ((min <= val) && (val <= max));
}
```

Place this function above `setup()` in your sketch. It returns either a TRUE or a FALSE that can be used in conditional statements. Here's an example of usage for a six position rotary switch using the tapped-ladder resistor scheme shown above:

```
loop() {
. . .
unsigned int aBus = 0;
aBus = analogRead(analogBus);

if (inRange(aBus, 757, 778)) {
  . . . // do position 1 stuff
}
else if (inRange(aBus, 717, 743)) {
  . . . // do position 2 stuff
{
else if (inRange(aBus, 670, 693)) {
  . . . // do position 3 stuff
{
```

```
else if (inRange(aBus, 600, 626)) {
  . . . // do position 4 stuff
{
else if (inRange(aBus, 498, 524)) {
  . . . // do position 5 stuff
{
else if (inRange(aBus, 326, 354)) {
  . . . // do position 6 stuff
{
else {
  . . . // do something-is-wrong stuff
}

. . .

} // closing bracket of loop()
```

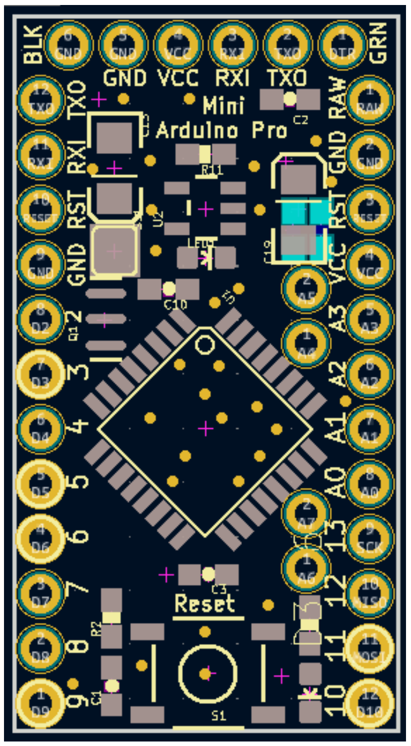
The ranges of `aBus` values given in each conditional are the calculated ADC readings using ±5% for the six 50KΩ ladder resistors, +5V as the ADC reference voltage, and a 10-bit ADC (such as in a Nano). It's sensible to wire a rotary switch with the resistors and +5V and then confirm the actual values using a simple testing sketch that displays the analogRead values. Adjust the range values for the appication sketch accordingly.

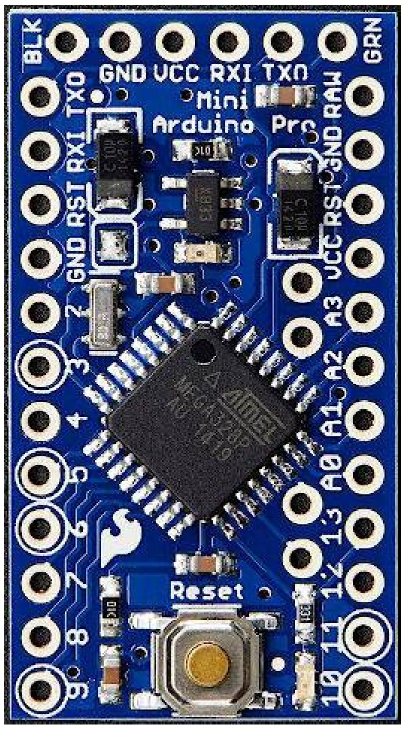
Appendix D

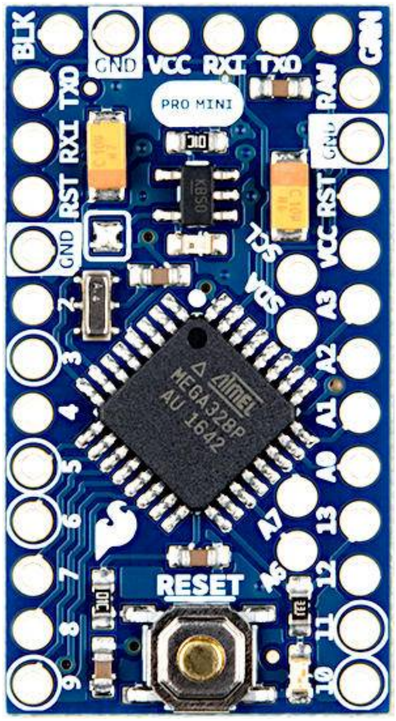
Arduino Pro-Mini and Clone Variations

Like all of Arduino's dozens of different models, the Pro-Mini is an open-hardware design. This means that its hardware and specifications are available free to anyone--or any manufacturer--to produce them in addition to the Arduino Foundation itself. By and large, this has resulted in fully-compatible "clones" and "knockoffs," and that has resulted in wide availability and acceptable quality at low cost.

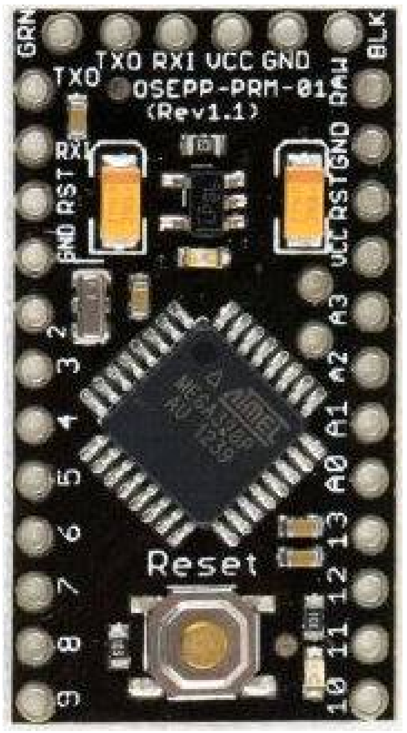
Occasionally, though, some differences have crept in. For the Pro-Mini, this has resulted in a few variations in the placement of pins A4, A5, A6, and A7. To minimize the footprint of the overall module, the original Arduino Pro-Mini did not place these pins on the outer perimeter of the board. Instead, they were located in a separate row inboard of the right edge. Pins A4 and A5 lie above the right corner of the MCU chip, A6 and A7 below that corner (see first image below). Pro-Mini clones made by Adafruit and Sparkfun duplicate the original exactly, and so does the PCB footprint for the Pro-Mini on the UDVBM-1.


Official Arduino layout. Note the column of four pads divided by the right corner of the MCU chip. From top to bot-tom, these are pins A5, A4, A7, and A6.

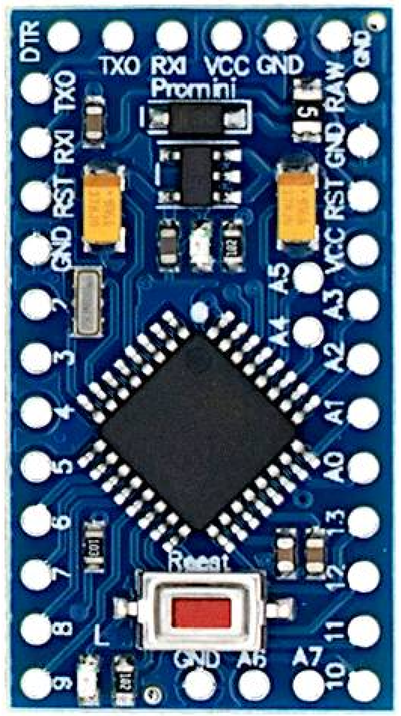

Adafruit Pro-Mini. Note the A4 to A7 pads in the same position as the official Arduino.


Sparkfun Pro-Mini. Also conforms to the Arduino layout.

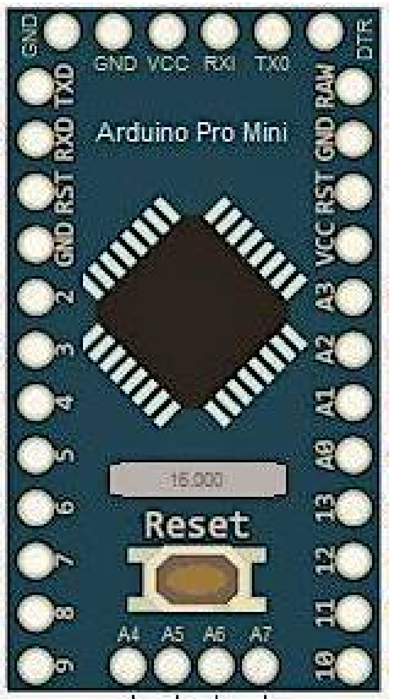
Pro-Mini clones made by various other manufacturers may or may not be identical to the original with regard to these pins. See the photos below for variations.



OSEPP clone. Pins A5 & A4 are present, but there are no A6 or A7 pins.



Various clones frequently found on eBay, Amazon, Ali Express, or Banggood. Brands include HiLetgo, AITRIP, Teyleten, etc. Pins A5 & A4 are in the official position, but A6 & A7 are on the bottom edge.



Various clones sometimes found online. Pins A4 - A7 are all on the bottom edge.

If you will not be using the ADC functions of the Pro-Mini, you need not be concerned with the location of pins A6 and A7. If you are using a clone with those pins located in a place other than the original, though, you will need to add short jumpers from those pads on the clone itself to the corresponding Arduino-breakout pads on the UDVBM-1. This will connect them to the ADC port found along the top edge of the UDVBM-1 board.

If your clone has pins A4-A7 all along the bottom edge (as in the last image above), you will definitely need to jumper A4 and A5 to their breakout pads since they are needed to I2C functionality. Fortunately, this clone variation is rare.