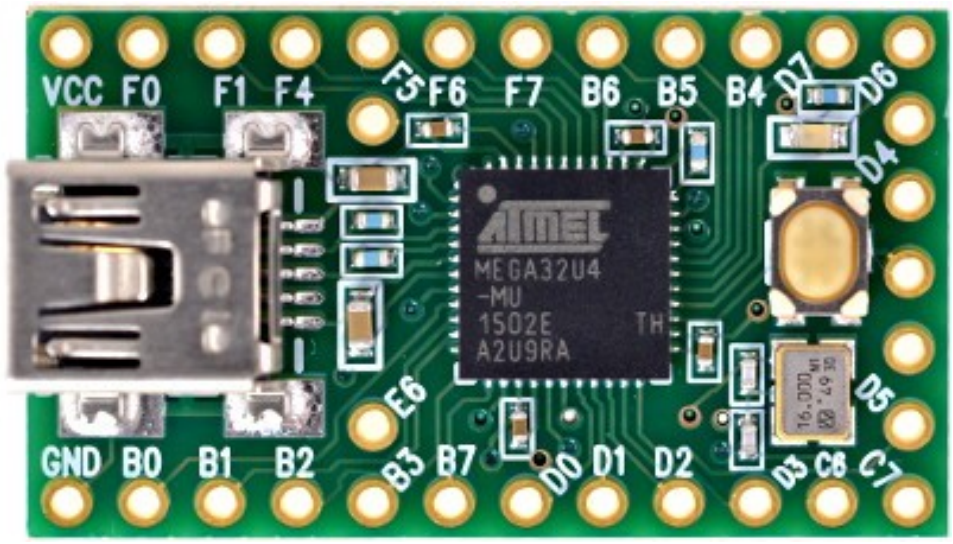
LinHPSDR is capable of sidetone for CW signals with latency in the order of 12 mSec which is quick enough for CW at 35 wpm or maybe more without the paddle to ear delays causing extra dots and dashes to be inserted by the operator. In addition it can send information via Hamlib to the common log programs like **CQRLOG**, **Xlog** and **tlf** for the contesters. The following describes how to set it all up.

Set up midi CW

At present linHPSDR will only work with Pulse Audio although work is underway to extend this to alsa in the future. linHPSDR expects to find a text file which maps the midi notes sent from the key/paddles/keyer and for brevity this is a sample of one for just keying only which must be saved to /home/username/.local/share/linhpsdr/ and is automatically loaded when linHPSDR starts up. You can send other commands via midi like ptt or volume up/down depending on what linHPSDR has been programmed to accept. At present it only understands the key/keyer input but piHPSDR understands many other midi commands.

```
# filename = midi.props
#
# save in /home/username/.local/share/linhpsdr/
#
# Sample midi.inp file, suitable for a teensy MIDI controller
#
DEVICE=Teensy MIDI
#
# CW Key
KEY=64 ACTION=CWR ONOFF
```

The Arduino UNO, Arduino pro-micro and teensy are suitable microprocessors for a keyer's TTL output to USBmidi conversion. In the case of the pro-micro and the teensy these microprocessors can operate with their usb port being a native midi port and key presses can be detected on the standard Digital I/O pins of the microprocessor. Shown below is a sample code for generating midi notes which will work on linHPSDR and as we are using KEY = 64 we find signals between GND & pin B7 will trigger that response. Programming the teensy is outside the scope of this help file but all the information needed can be found at PRJC web site https://www.pjrc.com/teensy/ At present I am leaning towards the pro-micro and have one on order for evaluation as it is a little cheaper and takes less messing around with the Arduino IDE but I feel that we should quickly come to a consensus on what we as a group default to as each USB midi has a different device identification and switching between them means editing the midi.props file.



// This code actually runs & can be compiled & loaded with the Arduino IDE
//

```
/* Buttons to USB MIDI Example

   You must select MIDI from the "Tools > USB Type" menu

   To view the raw MIDI data on Linux: aseqdump -p "Teensy MIDI"

   This example code is in the public domain.
*/

#include <Bounce.h>

// the MIDI channel number to send messages
const int channel = 1;

// Create Bounce objects for each button.  The Bounce object
// automatically deals with contact chatter or "bounce", and
// it makes detecting changes very simple.
Bounce button0 = Bounce(0, 5);
Bounce button1 = Bounce(1, 5);  // 5 = 5 ms debounce time
Bounce button2 = Bounce(2, 5);  // which is appropriate for good
Bounce button3 = Bounce(3, 5);  // quality mechanical pushbuttons
Bounce button4 = Bounce(4, 5);
Bounce button5 = Bounce(5, 5);  // if a button is too "sensitive"
Bounce button6 = Bounce(6, 5);  // to rapid touch, you can
Bounce button7 = Bounce(7, 5);  // increase this time.
Bounce button8 = Bounce(8, 5);
Bounce button9 = Bounce(9, 5);
Bounce button10 = Bounce(10, 5);
//Bounce button11 = Bounce(11, 5);

void setup() {
  // Configure the pins for input mode with pullup resistors.
  // The pushbuttons connect from each pin to ground.  When
  // the button is pressed, the pin reads LOW because the button
  // shorts it to ground.  When released, the pin reads HIGH
  // because the pullup resistor connects to +5 volts inside
  // the chip.  LOW for "on", and HIGH for "off" may seem
  // backwards, but using the on-chip pullup resistors is very
  // convenient.  The scheme is called "active low", and it's
  // very commonly used in electronics... so much that the chip
  // has built-in pullup resistors!
  pinMode(0, INPUT_PULLUP);
  pinMode(1, INPUT_PULLUP);
  pinMode(2, INPUT_PULLUP);
  pinMode(3, INPUT_PULLUP);
  pinMode(4, INPUT_PULLUP);
  pinMode(5, INPUT_PULLUP);
  pinMode(6, INPUT_PULLUP);  // Teensy++ 2.0 LED, may need 1k resistor pullup
  pinMode(7, INPUT_PULLUP);
  pinMode(8, INPUT_PULLUP);
  pinMode(9, INPUT_PULLUP);
  pinMode(10, INPUT_PULLUP);
  pinMode(11, OUTPUT); // Teensy 2.0 LED, may need 1k resistor pullup
}

void loop() {
  // Update all the buttons.  There should not be any long
```

```
  // delays in loop(), so this runs repetitively at a rate
  // faster than the buttons could be pressed and released.
  button0.update();
  button1.update();
  button2.update();
  button3.update();
  button4.update();
  button5.update();
  button6.update();
  button7.update();
  button8.update();
  button9.update();
  button10.update();
//  button11.update();

  // Check each button for "falling" edge.
  // Send a MIDI Note On message when each button presses
  // Update the Joystick buttons only upon changes.
  // falling = high (not pressed - voltage from pullup resistor)
  //           to low (pressed - button connects pin to ground)
  if (button0.fallingEdge()) {
    usbMIDI.sendNoteOn(60, 127, channel);  // 60 = C4
  }
  if (button1.fallingEdge()) {
    usbMIDI.sendNoteOn(61, 99, channel);  // 61 = C#4
  }
  if (button2.fallingEdge()) {
    usbMIDI.sendNoteOn(62, 99, channel);  // 62 = D4
  }
  if (button3.fallingEdge()) {
    usbMIDI.sendNoteOn(63, 99, channel);  // 63 = D#4
  }
  if (button4.fallingEdge()) {
    usbMIDI.sendNoteOn(64, 127, channel);  // 64 = E4
    digitalWrite(11, HIGH);
  }
  if (button5.fallingEdge()) {
    usbMIDI.sendNoteOn(65, 99, channel);  // 65 = F4
  }
  if (button6.fallingEdge()) {
    usbMIDI.sendNoteOn(66, 99, channel);  // 66 = F#4
  }
  if (button7.fallingEdge()) {
    usbMIDI.sendNoteOn(67, 99, channel);  // 67 = G4
  }
  if (button8.fallingEdge()) {
    usbMIDI.sendNoteOn(68, 99, channel);  // 68 = G#4
  }
  if (button9.fallingEdge()) {
    usbMIDI.sendNoteOn(69, 99, channel);  // 69 = A5
  }
  if (button10.fallingEdge()) {
    usbMIDI.sendNoteOn(70, 99, channel);  // 70 = A#5
  }
//  if (button11.fallingEdge()) {
//    usbMIDI.sendNoteOn(71, 99, channel);  // 71 = B5
//  }

  // Check each button for "rising" edge
  // Send a MIDI Note Off message when each button releases
```

```
  // For many types of projects, you only care when the button
  // is pressed and the release isn't needed.
  // rising = low (pressed - button connects pin to ground)
  //          to high (not pressed - voltage from pullup resistor)
  if (button0.risingEdge()) {
    usbMIDI.sendNoteOff(60, 0, channel);  // 60 = C4
  }
  if (button1.risingEdge()) {
    usbMIDI.sendNoteOff(61, 0, channel);  // 61 = C#4
  }
  if (button2.risingEdge()) {
    usbMIDI.sendNoteOff(62, 0, channel);  // 62 = D4
  }
  if (button3.risingEdge()) {
    usbMIDI.sendNoteOff(63, 0, channel);  // 63 = D#4
  }
  if (button4.risingEdge()) {
    usbMIDI.sendNoteOff(64, 0, channel);  // 64 = E4
    digitalWrite(11, LOW);
  }
  if (button5.risingEdge()) {
    usbMIDI.sendNoteOff(65, 0, channel);  // 65 = F4
  }
  if (button6.risingEdge()) {
    usbMIDI.sendNoteOff(66, 0, channel);  // 66 = F#4
  }
  if (button7.risingEdge()) {
    usbMIDI.sendNoteOff(67, 0, channel);  // 67 = G4
  }
  if (button8.risingEdge()) {
    usbMIDI.sendNoteOff(68, 0, channel);  // 68 = G#4
  }
  if (button9.risingEdge()) {
    usbMIDI.sendNoteOff(69, 0, channel);  // 69 = A5
  }
  if (button10.risingEdge()) {
    usbMIDI.sendNoteOff(70, 0, channel);  // 70 = A#5
  }
//  if (button11.risingEdge()) {
//    usbMIDI.sendNoteOff(71, 0, channel);  // 71 = B5
//  }

  // MIDI Controllers should discard incoming MIDI messages.
  // http://forum.pjrc.com/threads/24179-Teensy-3-Ableton-Analog-CC-causes-midi-
crash
  while (usbMIDI.read()) {
    // ignore incoming messages
  }
}
```

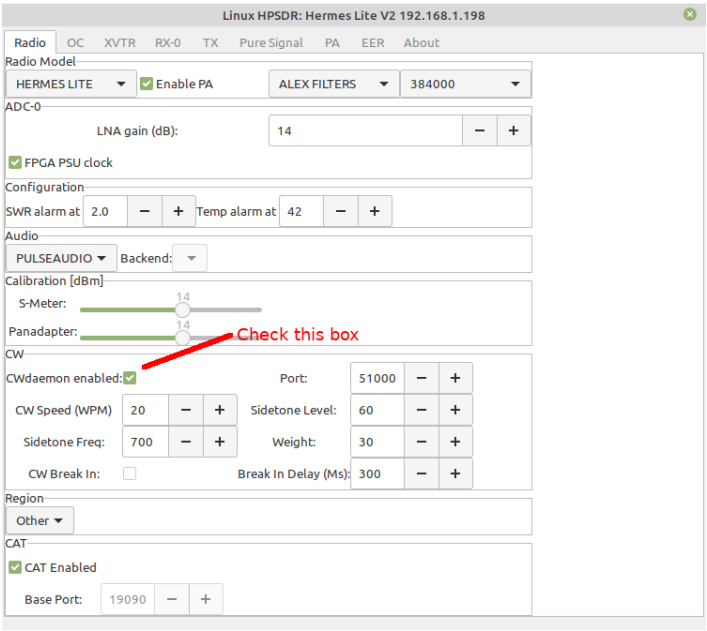
When the microprocessor program (teensy in this example) has been loaded, it can be tested by plugging into the computer and issuing terminal commands on linux as follows

```
~ $ aseqdump -l
 Port    Client name                      Port name
  0:0    System                           Timer
  0:1    System                           Announce
 14:0    Midi Through                     Midi Through Port-0
 20:0    Teensy MIDI                      Teensy MIDI MIDI 1
~ $
```

On the bottom line the name of the midi device shows and is “Teensy MIDI” in this case. The device can now be tested for correct operation now that the name is known and connecting a key between GND and B7 on the Teensy will produce the following output on key down followed by key up.

```
$ aseqdump -p "Teensy MIDI"
Waiting for data. Press Ctrl+C to end.
Source  Event                  Ch  Data
 20:0   Note on                 0, note 64, velocity 127
 20:0   Note off                0, note 64, velocity 0
^C ~ $
```

At this stage you are ready to set up linHPSDR for midi keying which is very simple.
Put the midi.props file in /home/username/.local/share/linhpsdr/
Start linHPSDR
Check the CWdaemon enabled box on the linHPSDR configure screen



Switch to CW mode and press the key. You should hear the sidetone in your speaker/headphones with no noticeable lag. The sidetone volume is also adjustable in the configure screen.

Set up Logging

Hamlib is a library which is designed to take a standard set of input signals and convert them to signals understood by various radios. The advantage of this is that a program such as a logging program can talk to a multiplicity of radios from its one set of commands that are translated by Hamlib. The communication to the radio can be via a serial port or a TCP connection which is what a software defined radio PC Console would use. To effect the connection to the radio the rigctld daemon is run in the background furnished with the information of what type of radio it is and what address and port is being used. In the case of linHPSDR or piHPSDR the radio is type 240 and if running the log program on the same computer you can use the loopback *localhost* address corresponding to *127.0.0.1* with the port specified by the PC Console e.g. linHPSDR in this case.

The following example will show how to connect Xlog and linHPSDR using Hamlib and the rigctld daemon. This is a typical set-up and will not vary much with other log software.

Most distros have Hamlib in their repository but you need to be careful which version that you have as for example Linux Mint 19.3 has libhamlib-utils 3.1.7build1 which does not support the rig 240 which is linHPSDR and piHPSDR on version 3.3.xbuildx. Just to further complicate things the recent versions of hamlib appear to have reorganized the rig#'s so the lin and pi HPSDR radios may not necessarily be 240.

In my case I had CQRLOG installed from my distro package and the early version of hamlib which did not support rig# 240 so I removed hamlib which due to the associated removal of rigctld caused the removal of CQRLOG which I figured I would reinstall after I loaded the hamlib 3.3 from github. Those who know will be smirking as catch 22 applies here and when I went to reinstall CQRLOG it wanted to overwrite rigctld grrr. I used checkinstall to reinstall hamlib and let CQRLOG install using whatever defaults it wanted and the net result was that rigctld now listed rig#240. I don't want to scare you off as Linux Mint 19.3 is only a month away from a major point upgrade and all this will go away. I tried MX Linux and Arche with both of these being OK and Ubuntu 20 has been reported to work.

The report above on the pitfalls are not likely to affect the installation and hopefully will be edited out in a couple of months and from here on is the usual method of installation.

- From your source install your log software. In my case it was from the distro packages XLog
- From your distro packages install hamlib
- check that linHPSDR is supported as follows ...

  ~$ rigctl -l | grep 240
  240 OpenHPSDR PiHPSDR 20170121 Beta
  2401 RFT EKD-500 0.4 Alpha
  ~ $
  piHPSDR and linHPSDR use the same format.

- Start linHPSDR from a terminal and from the config screen on the bottom left, check the Cat Enabled check box and note the port number (should be 19090)
- open a terminal and type in the following
  `rigctld -m 240 -r localhost:19090`

On the terminal that you ran linHPSDR from you will see the connection

```
rigctl_client: starting rigctl_client: socket=18
RIGCTL: CTLA INC cat_contro=0
RIGCTL: Leaving rigctl_client threadsetting SO_LINGER to 0 for client_socket: 18
```

The final step is to configure your log program and you will need to find your cat/rigctl/hamlib options and set them to radio type 2, localhost:4532 (the Hamlib standard address). Don't forget to enable polling as hamlib does not send a stream of data but responds to requests for data.

Please modify and add if better information is available.