Mode of operation:

Controlling the Isolog Mobile first requires to obtain a connection handle using the FTDI D2XX driver or an equivalent library (like libftdi). The Isolog Mobile can be identified by the following USB IDs:

```
USB VID = 0x0403
USB PID = 0xe8de
```

Please refer to the FTDI D2XX programmers guide (or the documentation of your library) for the exact way how to establish a connection (this may differ based on OS).

If the connection is established it needs to be configured properly (using the D2XX API as example, skipping error checking for readability):

```
FT_SetUSBParameters(handle, 4096, 4096);
FT_SetChars(handle, 0, 0, 0, 0);
FT_SetTimeouts(handle, 0, 5000);
FT_SetLatencyTimer(handle, 1);
FT_SetFlowControl(handle, 0x0100, 0, 0);
FT_SetBitMode(handle, 0, 0);
FT_SetBitMode(handle, 0, 0x02);
```

As the control protocol differs between firmware versions it is necessary to extract and store the version before attempting to actually communicate with the device. The firmware version is embedded in the serial string of the FTDI EEPROM, which should have the form

XXXXXX-YY-ZZZZZZ   (number of X and Z digits may vary, but Y should always be a 2-digit value)

The firmware version is the second Y digit. The firmware version can be extracted in the following way for example:

```
FT_DEVICE dev;
DWORD devID;
char serial[16];
char desc[64];
FT_GetDeviceInfo(handle, &dev, &devID, serial, desc);
int firmware_ver = 0;
for (int i = 0; i < 14 && serial[i] != '\0'; i++) {
      if (serial[i] == '-' && serial[i+2] != '\0') {
            firmware_ver = serial[i+2] - '0';
            break;
      }
}
```

(for reference: the XXX part of the USB serial identifies the product revision, the ZZZ part the actual device serial number, and the first Y digit is the hardware version)

For better readability in the next sections a helper function around FT_Write to send individual bytes is recommended:

```
FT_Status FT_WriteBytes(FT_HANDLE handle, int count, char byte1, char byte2,
char byte3) {
      char buffer[3];
      int written;
      buffer[0] = byte1;
      buffer[1] = byte2;
      buffer[3] = byte3;
      return FT_Write(handle, buffer, count, &written);
}
```

Now is the time to initialize the FTDI MPSSE engine on the device that will translate incoming data into actual control signals:

```
FT_WriteBytes(handle, 1, 0x84);
char adbus = 0x09;      // this field is modified later on and therefore should
be stored
FT_WriteBytes(handle, 3, 0x80, adbus, 0xfb);
FT_WriteBytes(handle, 3, 0x86, 0x64, 0x00);
```

At this point the IsoLOG Mobile operating state can be modified. Any modification requires sending a complete control frame containing the full desired state, it is not possible to only modify individual fields. The control frame is a 16 bit value with the following sematics (counting from LSB to MSB):

```
     Bit  0: axis == X or axis == Chopper
     Bit  1: axis == Y or axis == Chopper
     Bit  2: antenna == Loop
     Bit  3: lna1 == enabled and firmware_ver < 3 (else reserved)
     Bit  4: remote LED == enabled and firmware_ver >= 2 (else reserved)
     Bit  5: lna1 == enabled and firmware_ver >= 3 (else reserved)
     Bit  6: lna2 == enabled and firmware_ver >= 3 (else reserved)
     Bit  7: reserved
     Bit  8-15: chopper rate index (0-255)
```

If bit 0 and 1 are both set the antenna enters "chopper mode" and cycles between all three axes with the frequency defined by "chopper rate index" (see below). If neither bit 0 or 1 are set the Z-axis is selected.
„remote LED“ status also controls whether the physical hardware buttons are locked. Please note that this state is enabled when establishing a USB connection and is NOT disabled automatically on disconnect.

The actual cycle frequency can be configured between 1 Hz (index = 0) and 50 kHz (index = 255). The physical switch for controlling the cycle frequency ranges from 0-15, which is then multiplied by 17 to get the actual chopper rate index. Refer to the following table for frequency values at each switch positions (intermediate index values are scaled accordingly, but aren't specified exactly).

```
Switch position  0 = Index   0: 1 Hz
Switch position  1 = Index  17: 2.06 Hz
Switch position  2 = Index  34: 4.23 Hz
Switch position  3 = Index  51: 8.71 Hz
Switch position  4 = Index  68: 17.91 Hz
Switch position  5 = Index  85: 36.84 Hz
Switch position  6 = Index 102: 75.79 Hz
Switch position  7 = Index 119: 155.9 Hz
Switch position  8 = Index 136: 320.71 Hz
Switch position  9 = Index 153: 659.74 Hz
Switch position 10 = Index 170: 1357.22 Hz
Switch position 11 = Index 187: 2791.74 Hz
Switch position 12 = Index 204: 5742.18 Hz
Switch position 13 = Index 221: 11820.33 Hz
Switch position 14 = Index 238: 24330.90 Hz
Switch position 15 = Index 255: 50125.31 Hz
```

Sending the control frame involves splitting it into 4-bit pieces which are individually sent to the device. After each transfer a delay is required for the device to actually process the incoming data:

```
void PulseRemoteClock() {
      adbus |= 0x40;
      FT_WriteBytes(handle, 3, 0x80, adbus, 0xfb);
      adbus &= 0xb0;
      FT_WriteBytes(handle, 3, 0x80, adbus, 0xfb);
      usleep(100000); // The device requires time to process the command, but
doesn't provide any feedback when done
}

adbus |= 0x80;
FT_WriteBytes(handle, 3, 0x80, adbus, 0xfb);
FT_WriteBytes(handle, 3, 0x82, frame & 0x000f, 0xff);
PulseRemoteClock();
if (firmware_ver >= 2) {
      adbus &= 0x7f;
      FT_WriteBytes(handle, 3, 0x80, adbus, 0xfb);
      FT_WriteBytes(handle, 3, 0x82, (frame >> 4) & 0x000f, 0xff);
      PulseRemoteClock();
}
adbus &= 0x7f;
FT_WriteBytes(handle, 3, 0x80, adbus, 0xfb);
FT_WriteBytes(handle, 3, 0x82, (frame >> 8) & 0x000f, 0xff);
PulseRemoteClock();
FT_WriteBytes(handle, 3, 0x82, (frame >> 12) & 0x000f, 0xff);
PulseRemoteClock();
```

It is not possible to extract the current hardware state, so you must initialize the state and track all modifications yourself.