Introduction

The WB7NFX Repeater Controller is a module designed to support a Westell DRB25 VHF repeater. The controller consists of 3 components, a 2-board module and front panel containing speaker, LED's and push button controls.

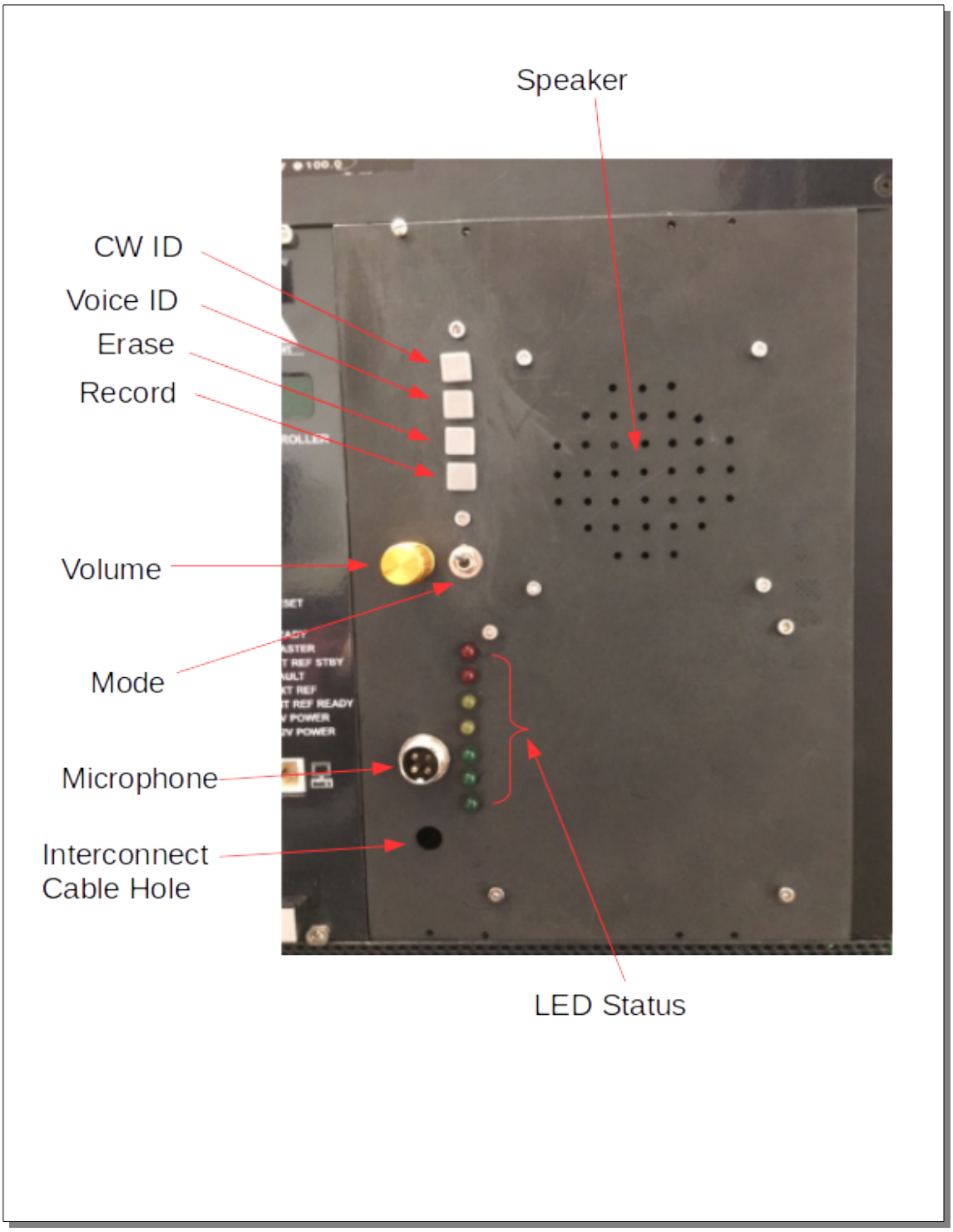
The controller interfaces with the VHF transceiver board via an RJ45 connector. The controller generates CW ID and Voice ID at 10m and 30m intervals respectively. The CW ID interval follows part 97 rules pertaining to Repeater Identification. Power is supplied from the VHF transceiver board.

The controller consists of a 3 channel audio mixer that takes CW, Voice and Microphone feed from the front panel and interfaces to the VHF repeater module via its Microphone input. Levels for each channel are made via trim pots on the controller module.

To maintain time/state information the controller utilizes a small micro-controller called a Teensy3.2 AVR. The micro-controller is responsible for driving PTT to the repeater and generates a Carrier Operated Squelch input to determine repeater activity. The micro-controller drives LED for status indication and senses push button controls from the microphone and push buttons on the front panel. The micro-controller also generates the CW ID which has been programmed via software. Updates to the software can be made via USB interface on the Teensy3.2 sub-module.

When the repeater is idle a Voice ID can be configured to be sent every 30m. When COS is first detected a CW ID is sent at 18wpm and again every 10m while the machine is active.

Front Panel, Interconnect and Controls



Operation

Operation of the controller is simplistic. There are several buttons on the front panel which perform the following operations :

- **CW ID** – Sends CW ID which has been programmed into the software.
- **Voice ID** – Sends the Voice ID which has been recorded in the Voice ID board
- **Erase** – Erases the recorded Voice ID
- **Record** – Records a new Voice ID. Note that you must first Erase the stored ID before a new one is recorded. Recording of the message uses the front panel microphone however PTT is not required to be pressed.
- **Mode** – Toggles whether the controller sends CW ID only or Both Voice ID and CW ID. CW ID is sent after the 1st time the repeater goes active then again in 10m per part 97 rule. Voice ID is sent at 30m intervals when the repeater is not active.

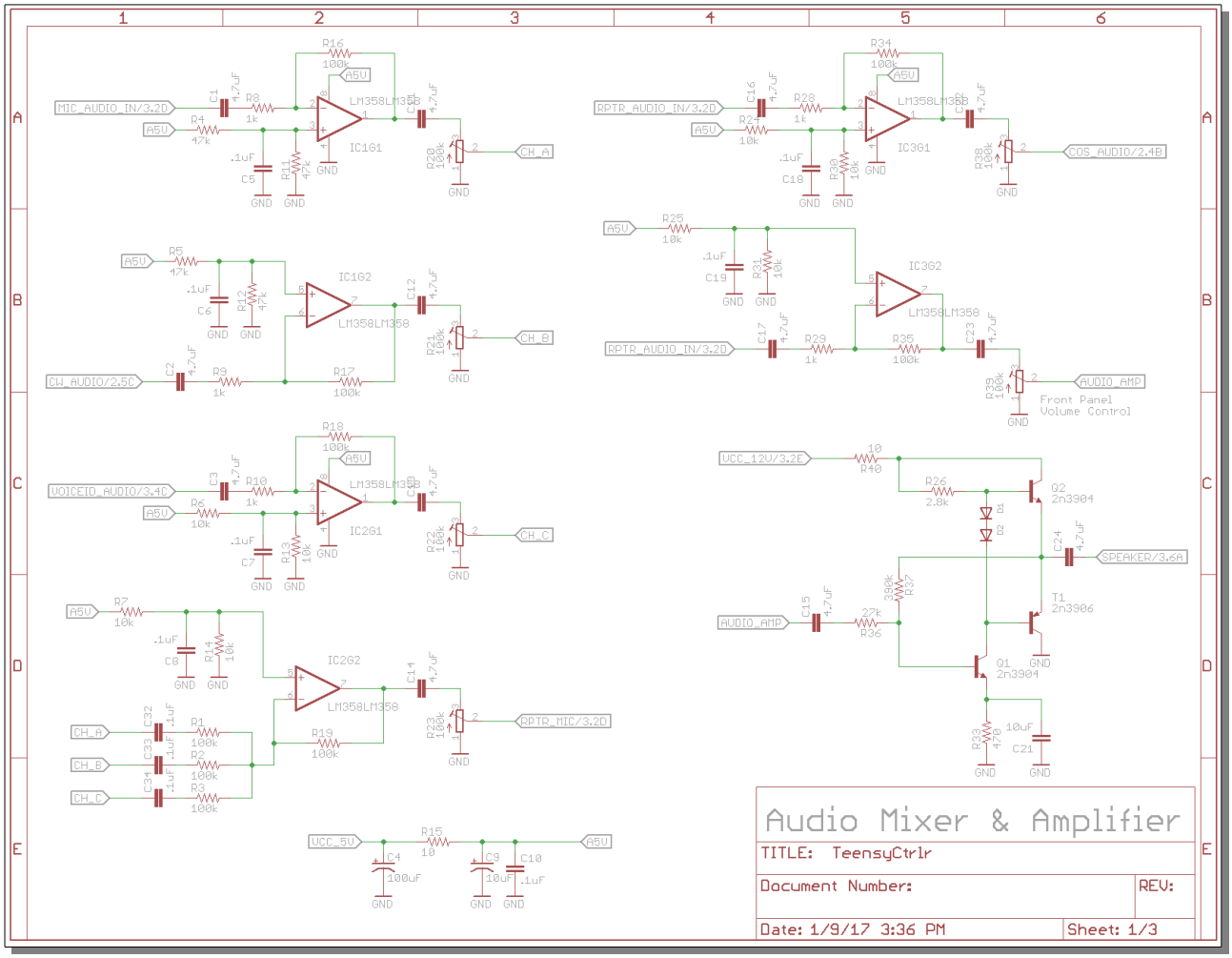
The front panel contains a speaker and small audio amplifier with volume control. This allows to hear the repeater activity. When using the front panel microphone to transmit voice is not heard over the speaker.

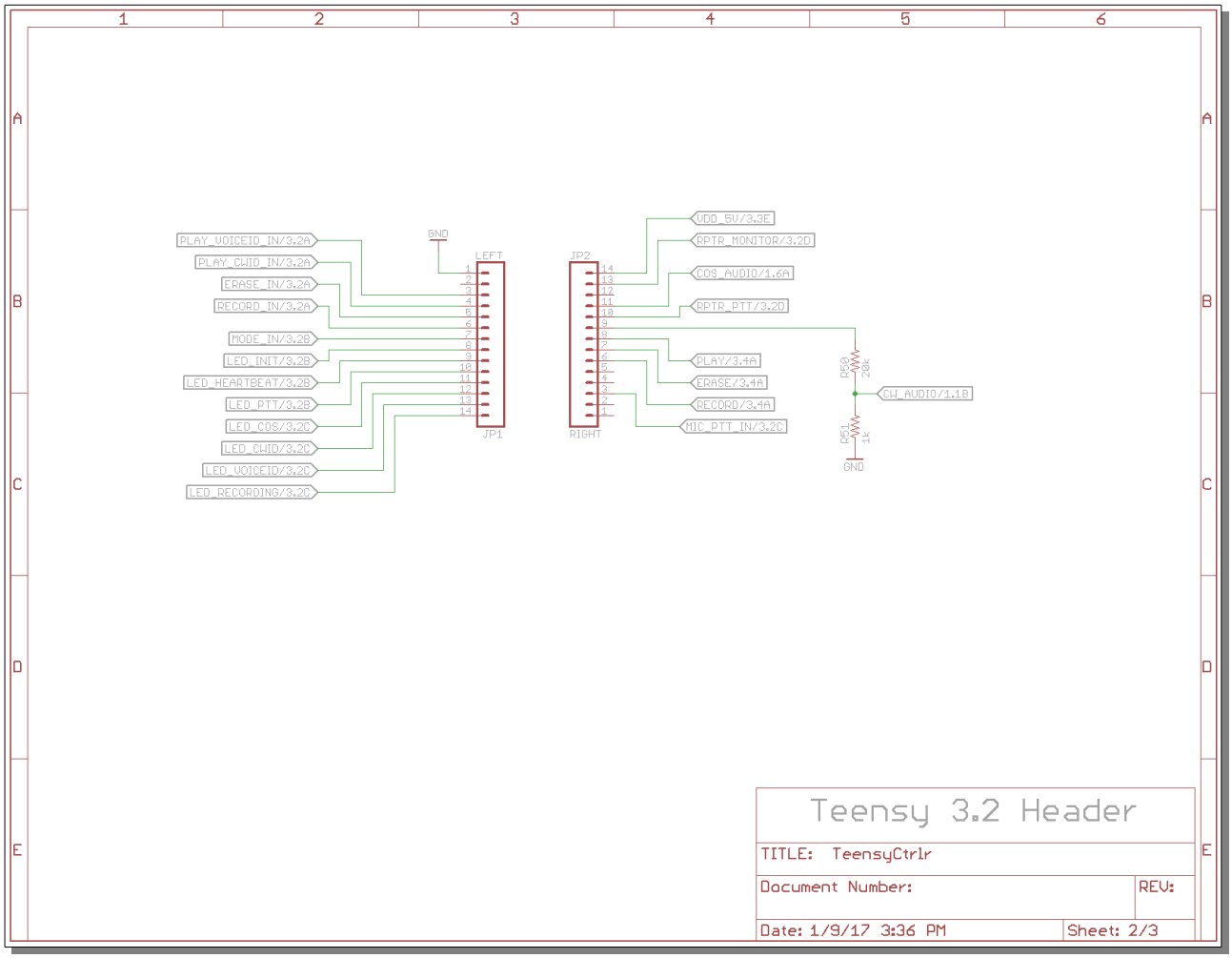
Note : *If using a handheld transceiver and volume is up you may hear feedback unless the microphone has been disconnected from the front panel or the volume is turned down enough to prevent feedback.*

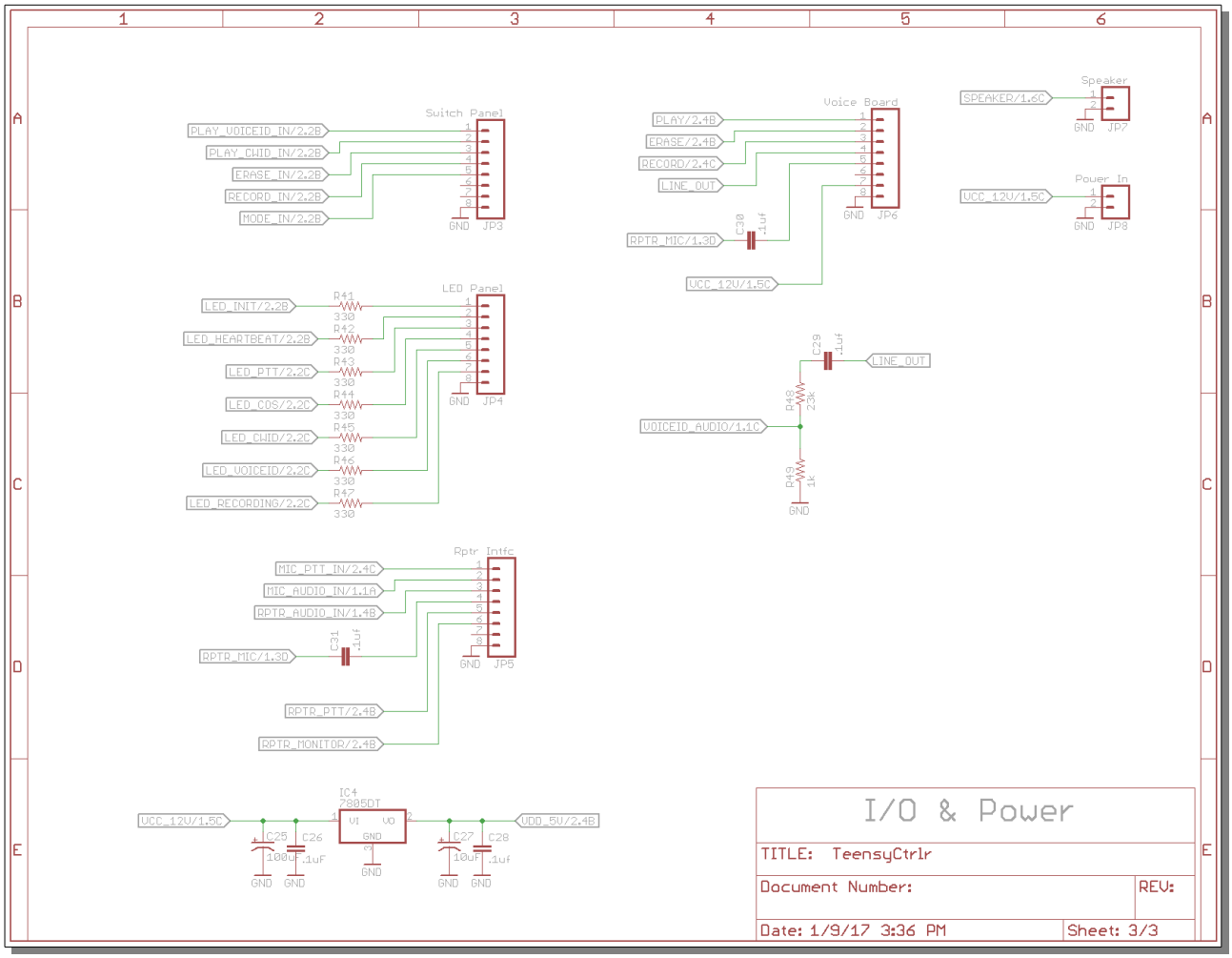
Front panel has 7 LED's that represent status of the controller. They are as follows (top to bottom) :

LED	Description
INIT	Lights up when the setup function completes in software
HEARTBEAT	Pulses approx 1s intervals indicating main loop is operational
PTT	Microphone PTT is active
COS	Repeater Tx is busy, Carrier Operated Squelch via software
CW ID	CW ID is being sent
Voice ID	Voice ID is being sent
Recording	Recording new voice ID in process

Schematic


Audio Mixer & Amplifier
TITLE: TeensyCtrlr
Document Number:
REV:
Date: 1/9/17 3:36 PM
Sheet: 1/3
MIC_AUDIO_IN/3.2D
CW_AUDIO/2.5C
VOICEID_AUDIO/3.4C
RPTR_AUDIO_IN/3.2D
VCC_12V/3.2E
CH_A
CH_B
CH_C
RPTR_MIC/3.2D
COS_AUDIO/2.4B
AUDIO_AMP
SPEAKER/3.6A
Front Panel Volume Control
LM358LM358
2n3904


PLAY_VOICEID_IN/3.2A
PLAY_CWID_IN/3.2A
ERASE_IN/3.2A
RECORD_IN/3.2A
MODE_IN/3.2B
LED_INIT/3.2B
LED_HEARTBEAT/3.2B
LED_PTT/3.2B
LED_COS/3.2C
LED_CWID/3.2C
LED_VOICEID/3.2C
LED_RECORDING/3.2C
GND
LEFT
JP1
JP2
RIGHT
VDD_5V/3.3E
RPTR_MONITOR/3.2D
COS_AUDIO/1.6A
RPTR_PTT/3.2D
PLAY/3.4A
ERASE/3.4A
RECORD/3.4A
MIC_PTT_IN/3.2C
D50
20k
CW_AUDIO/1.1B
D51
1k
GND
Teensy 3.2 Header
TITLE: TeensyCtrlr
Document Number:
REV:
Date: 1/9/17 3:36 PM
Sheet: 2/3



Software

Software can be found at : https://github.com/dtheriault/TeensyRptrCtrlr

The software is offered under the GPL and is considered Open Source available for general use and modification following GPL agreement.

Software is written in C langage, and uses the Ardruino IDE supporting the Teensy3.2 AVR chipset.

Updates to the software are made using the micro USB connector on the Teensy3.2 module located on the controller module.

```
//
// File: TeensyRptrCtrlr.ino
//
// Author:  NO1D - Doug Theriault
// Dated:   20170112
//
// Description:
//
// This sketch is a very basic/simple repeater ID controller which was developed
// for WB7NFX.  The interface is to a Westel DRB25 VHF repeater however this
// software is generic and can be modified to fit other requirements.
//
// The software handles button presses to activate CW ID, Voice ID and record
// and erase functions and mode switch.  In addition it outputs series of 7 LEDs
// to track state/function.  It also outputs control lines to a Voice ID board
// and the repeater itself.
//
// Since the Westel Repeater did not have means to detect COS (carrier operated
// squelch) the repeaters audio is fed into a pin configured with ADC so its
// voltage can be read.
//
// The controller was based on Teensy3.2 board.  I started with an ATiny85 but
// quickly ran out of pins as I wanted to add more/more features...
//
// References:
//
// The CW ID portion of the code was developed by SV1DJG which was a great start
// for me to get something working on an ATiny85 chip.  Special thanks goes out
// to Nick for producing this software.  I include his file header below.
//
// Bugs/Issues/ToDo:
//
// Ideally this would all be run off interrupts/timers.  Someday perhaps.
// This is not a very powerful repeater controller.  While it will ID per part 97
// rules, its been deployed only recently and may still contain bugs.
// There are some printf statements for debugging left in the code and do affect
// accuracy of the timing.  You might need to modify delay loops when removing
// them.
//
// Hardware:
//
// Schematic for simple 3-channel audio mixer, 3-transistor audio amp, Teensy I/O
// LED's etc are available on github.
//
// Voice ID is handled by external board similar to:
//   https://ludens.cl/Electron/voiceID/voiceID.html
//
// Contact info:
//
// email:  no1d.doug@gmail.com
// repository:  https://github.com/dtheriault/TeensyRptrCtrlr
//
// License:
//
// Copyright (C) 2017 Douglas Theriault - NO1D
//
// TeensyRptrCtrlr.ino is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
```

```
//
// TeensyRptrCtrlr.ino is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
// or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
// more details.
//
// You should have received a copy of the GNU General Public License
// along with IDtimer.ino.  If not, see <http://www.gnu.org/licenses/>.
//
//
////////////////////////////////////////////////
//
// A simple CW Blinker (aka LED Beacon)
// by SV1DJG (c.2016)
//
// This is a ultra simple blinker in CW for the
// digispark mini board (attiny85).
// change the led port if using a different
// board (e.g. Arduino)
//
// Just set your message into the msg array and you
// are ready to go.Make sure to use ONLY the set
// of the supported characters.
// the supported set of characters is capital
// letters,numbers and a few symbols.
//
// You can also modify the transmission speed by
// changing the speedWPM variable.
//
// HAVE FUN!!
//
// This code is provided as-is and may contain bugs!
//
// (codesize:1132 bytes + message length)
//
////////////////////////////////////////////////

#include <avr/pgmspace.h>

//
// PIN Definitions:  Input and Outputs
//
#define PLAY_VOICE_IN  1          // Send the Voice ID, Play on Voice Bd
#define PLAY_CW_IN     2          // Send a CW ID from the Teensy
#define ERASE_IN       3          // Erase the Voice ID, Erase on Voice Bd
#define RECORD_IN      4          // Record New Voice ID, Record on Voice Bd
#define MODE_IN        5          // Toggle switch for CW only or CW & Voice ID
//
#define LED_INIT       6          // Setup has been executed, stays lit
#define LED_HEARTBEAT  7          // Loop interval, one second pulse
#define LED_PTT        8          // PTT is activated, Microphone front panel
#define LED_COS        9          // COS detected from Repeater Audio feed
#define LED_CWID      10          // CW ID in process
#define LED_VOICEID   11          // Voice ID in process
#define LED_RECORDING 12          // Recording new Voice ID in process
//
#define RPTR_MONITOR  14          // Repeater Monitor Active Low (not used)
#define PTT_IN        15          // Front Panel Microphone, Active Low
//
#define RECORD_OUT    18          // Button on Voice Bd
#define ERASE_OUT     19          // Button on Voice Bd
#define PLAY_OUT      20          // Button on Voice Bd
```

```
//
#define CW_AUDIO      21          // PWM Teensy output of CW Audio tone
#define PTT_OUT       22          // PTT to Repeater, Active Low
#define COS_IN        23          // Carrier Detect Input
//
//
//
// General Defines
//
#define CW_TONE      750          // CW ID Tone in Hertz
//
#define MODE_CW_ONLY   0          // Does not play Voice ID
#define MODE_BOTH      1          // Plays both Voice and CW ID's
//
// State Machine Defines
//
#define IDLE              1       // IDLE State
#define TIMING_CW         2       // Timer State:  Detected COS/PTT
#define TIMING_VOICE      3       // Timer State:  Detected COS/PTT
#define CWID              4       // CW ID: Send CW ID now, timer expired
#define VOICEID           5       // Voice ID:  Send Voice ID
#define RPTR_ACTIVE       6       // Time COS/PTT been active
#define ID_FIRST          7       // First ID > 10m on first PTT/COS from IDLE period
//
// Timeouts are in seconds
//
#define CW_TIMEOUT    600         // Play CW ID every 10m after COS or PTT go active
#define VOICE_TIMEOUT 1800        // Play Voice ID every 30m when IDLE
#define COS_TIMEOUT   300         // COS/PTT 3m timeout
#define VID_TIMEOUT   1200        // 20m after CWID, if we're idle, we VoiceID
//
// All delays are in ms
//
#define RECORDING_DELAY 10000     // Record for 10s
#define ERASE_DELAY     10000     // Erase for 10s
#define VOICEID_DELAY   10000     // 10s for voice id to play
#define PTT_DELAY       1500      // 1.5s PTT delay
#define COS_DELAY       100       // delay 100ms between consecutive COS ADC reads
//
#define LED_OFF      LOW
#define LED_ON      HIGH

#define TRUE           0
#define FALSE          1

// this defines the keying level and provides a quick way to change the keying method.
// when we use a LED we need HIGH to turn the LED on and LOW to turn the LED off
// but if we select to connect an external switching hardware , this may need to be reversed
#define MORSE_KEY_DOWN  HIGH
#define MORSE_KEY_UP    LOW

// delay before repeating the message (in Ms)
#define repeatDelayMs 300000

// transmission speed (in words per minute)
#define speedWPM 18

// element duration is according to Farnsworth Technique
// see : http://www.arrl.org/files/file/Technology/x9004008.pdf
#define dotDuration  1200 / speedWPM          // element duration in milliseconds
#define dashDuration        dotDuration * 3
```

```
#define interSymbolDuration dotDuration
#define interCharDuration   dotDuration * 3
#define interWordDuration   dotDuration * 7


//
// Global Variables Here
//
boolean mic_ptt;
boolean play_voice;
boolean play_cw;
boolean erase_voice;
boolean record;
boolean voice_bd_reset;

int mode;
int state;
int last_state;

int cw_timer;
int voice_timer;
int activity_timer;
int last_id;

int cos_in;

boolean last_ptt;
boolean last_cos;

char dState[8][16] =
  {
    "UNKNOWN",
    "IDLE",
    "TIMING CW",
    "TIMING VOICE",
    "CW ID",
    "VOICE ID",
    "RPTR ACTIVE",
    "ID FIRST"
  };

////////////////////////////////////////////////////////////////////////////////////////////
//
// this is the beacon message (only capital letters, numbers and a few symbols - see below)
//
////////////////////////////////////////////////////////////////////////////////////////////

//char const msg[] = "TEST DE SV1DJG";
char const msg[] = "DE WB7NFX/R";
char const msgTimeout[] = "TIMEOUT";

//
// Morse code table
// ----------------
// Morse code elements are variable length and transmitted Msb first.
// we need to know the length of each element so we save
// it along with the element as the first part of the pair in the table.
//
// Each element is made up from 1's and 0's where:
// 1 means DOT
// 0 mean DASH
// for example, A is DOT-DASH which is translated to 10.
```

```
// B is DASH-DOT-DOT-DOT which is translated to 0111 etc.
// these are encoded as binary values and paired with their length
// for example
//    A -> 2,B10
//    B -> 4,B0111
// etc
//
// table of elements for Morse cose numbers do not follow this as all elements are 5 digits long
// (so we save space in the symbol table)
//
byte const letters[] PROGMEM =
{
  2,B10,   // A
  4,B0111, // B
  4,B0101, // C
  3,B011,  // D
  1,B1,    // E
  4,B1101, // F
  3,B001,  // G
  4,B1111, // H
  2,B11,   // I
  4,B1000, // J
  3,B010,  // K
  4,B1011, // L
  2,B00,   // M
  2,B01,   // N
  3,B000,  // O
  4,B1001, // P
  4,B0010, // Q
  3,B101,  // R
  3,B111,  // S
  1,B0,    // T
  3,B110,  // U
  4,B1110, // V
  3,B100,  // W
  4,B0110, // X
  4,B0100, // Y
  4,B0011, // Z
};

byte const numbers[] PROGMEM =
{
  B00000, // 0
  B10000, // 1
  B11000, // 2
  B11100, // 3
  B11110, // 4
  B11111, // 5
  B01111, // 6
  B00111, // 7
  B00011, // 8
  B00001, // 9
};


byte const symbols[] PROGMEM =
{
  6,B101010,  // Full-stop (period)
  6,B001100,  // Comma
  6,B110011,  // Question mark (query)
  5,B01101 ,  // Slash
  5,B01110 ,  // Equals sign
```

```
};

// transmits a morse element (either a dot or a dash)
void sendSymbol(boolean sendDot)
{
  digitalWrite(LED_CWID, MORSE_KEY_DOWN);
  tone(CW_AUDIO, CW_TONE);
  delay(sendDot ? dotDuration : dashDuration);
  digitalWrite(LED_CWID, MORSE_KEY_UP);
  noTone(CW_AUDIO);
}

// transmits an ASCII character in Morse Code
void sendCharacter(char c)
{
  byte numberOfBits = 0;
  byte elementCode  = 0;

  if (c >= 'A' && c <='Z')
  {
    int elementIndex = (int)(c-'A');
    numberOfBits = pgm_read_byte(&letters[elementIndex * 2]);
    elementCode  = pgm_read_byte(&letters[elementIndex * 2 + 1]);
  }
  else if (c >= '0' && c <='9')
  {
    int elementIndex = (int)(c-'0');
    numberOfBits = 5;
    elementCode  = pgm_read_byte(&numbers[elementIndex]);
  }
  else if (c == ' ')
  {
    // adjust delay because we have already "waited" for a character separation duration
    // NOTE:this delay will not be 100% accurate if the first character in the message is SPACE
    delay(interWordDuration-interCharDuration);
  }
  else
  {
    int elementIndex = -1;

    if (c == '.')      elementIndex = 0;
    else if (c == ',') elementIndex = 1;
    else if (c == '?') elementIndex = 2;
    else if (c == '/') elementIndex = 3;
    else if (c == '=') elementIndex = 4;

    if (elementIndex != -1)
    {
      numberOfBits = pgm_read_byte(&symbols[elementIndex * 2]);
      elementCode  = pgm_read_byte(&symbols[elementIndex * 2 + 1]);
    }
  }


  if (numberOfBits > 0)
  {
    byte mask = 0x01 << (numberOfBits - 1);

    while (numberOfBits > 0 )
    {

      sendSymbol((elementCode & mask) == mask);
```

```
      delay(interSymbolDuration);
      mask = mask >> 1;
      --numberOfBits;
    }
     // adjust delay because we have already "waited" for a dot duration
     delay(interCharDuration - interSymbolDuration);
   }

}

// transmits an ASCII message in Morse Code
void sendCWID(const char* msg)
{
  // trigger PTT so we send ID over repeater
  digitalWrite(PTT_OUT, LOW);
  digitalWrite(LED_PTT, LED_ON);

  delay(PTT_DELAY);

  while (*msg)
    sendCharacter(*msg++);

  delay(PTT_DELAY);

  // turn off PTT at the end
  digitalWrite(PTT_OUT, HIGH);
  digitalWrite(LED_PTT, LED_OFF);
}

// transmits Voice ID via external board
void sendVoiceID()
{
  Serial.print("Sending Voice ID\n");
  digitalWrite(PTT_OUT, LOW);
  digitalWrite(LED_PTT, LED_ON);

  delay(PTT_DELAY);

  digitalWrite(PLAY_OUT, LOW);
  digitalWrite(LED_VOICEID, LED_ON);

  // TODO: Ideally we want program to continue while output is low
  //       as believe voice play stops when we toggle back high

  delay(VOICEID_DELAY);

  digitalWrite(PLAY_OUT, HIGH);
  digitalWrite(LED_VOICEID, LED_OFF);

  delay(PTT_DELAY);

  digitalWrite(PTT_OUT, HIGH);
  digitalWrite(LED_PTT, LED_OFF);

  Serial.print("End Voice ID\n");
}

//
// Reset the external Voice ID board
//
void resetVoiceBd()
{
```

```
  //  digitalWrite(RESET_OUT, LOW);
  delay(500);
  //  digitalWrite(RESET_OUT, HIGH);
}

//
// Erase stored Voice ID
//
void eraseVoice()
{
  digitalWrite(ERASE_OUT, LOW);
  delay(ERASE_DELAY);
  digitalWrite(ERASE_OUT, HIGH);
  Serial.print("Voice ID Erased\n");
}

//
// Record New Voice ID
//
void recordVoiceID()
{
  // first erase Voice ID
  eraseVoice();

  // Enable recording for max of 7 seconds
  digitalWrite(LED_RECORDING, LOW);
  digitalWrite(RECORD_OUT, LOW);

  delay(RECORDING_DELAY);

  digitalWrite(RECORD_OUT, HIGH);

  // delay a bit between operations
  delay(2000);

  // Now play without transmitting
  digitalWrite(PLAY_OUT, LOW);

  delay(VOICEID_DELAY);

  // Turn off LED indicator at end of recording
  digitalWrite(LED_RECORDING, HIGH);

  Serial.print("Voice ID Recorded\n");
}

//
// readCOS()
//
boolean readCOS()
{
  int rValue = 0;

  rValue = analogRead(COS_IN);
  delay(COS_DELAY);
  rValue += analogRead(COS_IN);

  rValue = rValue/2;

  //  Serial.printf("COS: %3d\n", rValue);

  if (rValue > 30) return HIGH;
```

```
  return LOW;
}

//
// readInputs()
//
void readInputs()
{

  mic_ptt = digitalRead(PTT_IN);
  play_voice = digitalRead(PLAY_VOICE_IN);
  play_cw = digitalRead(PLAY_CW_IN);
  erase_voice = digitalRead(ERASE_IN);
  record = digitalRead(RECORD_IN);
  mode = digitalRead(MODE_IN);

  cos_in = readCOS();
}


//
// execCommands()
//
void execCommands()
{
  // Low Priority:  Handle Front Panel Buttons
  //
  if (play_cw == LOW)
      state = CWID;

  else if (play_voice == LOW)
      state = VOICEID;

  //  if (voice_bd_reset == LOW)
  //      resetVoiceBd();

  if (record == LOW)
      recordVoiceID();

  else if (erase_voice == LOW)
      eraseVoice();

  // Med Priority:  Handle COS input
  //
  if (cos_in == HIGH) {
    if (last_cos == LOW)
      Serial.print("COS Active\n");
    last_cos = HIGH;
    // light the COS LED
    digitalWrite(LED_COS, LED_ON);
    }
  else {
      if (last_cos == HIGH)
       Serial.print("COS Deactivated\n");
      last_cos = LOW;
      digitalWrite(LED_COS, LED_OFF);
    }

  // High Priority:  Handle PTT
  //
  if (mic_ptt == LOW) {
```

```
    if (last_ptt == HIGH)
      Serial.print("PTT Active\n");
    last_ptt = LOW;
    digitalWrite(LED_PTT, LED_ON);
    digitalWrite(PTT_OUT, LOW);
  }
  else {
    digitalWrite(LED_PTT, LED_OFF);
    digitalWrite(PTT_OUT, HIGH);
    if (last_ptt == LOW)
      Serial.print("PTT Deactivated\n");
    last_ptt = HIGH;
  }

  //  Serial.printf("execCmd:  [COS]: %d, [PTT]: %d\n", cos_in, mic_ptt);
}

//
// Initialization Function - executed once
//
void setup()
{
  int x = 0;
  int value = 0;

  // Initilize Pins
  //
  pinMode(PTT_IN, INPUT_PULLUP);
  pinMode(PLAY_VOICE_IN, INPUT_PULLUP);
  pinMode(PLAY_CW_IN, INPUT_PULLUP);
  pinMode(ERASE_IN, INPUT_PULLUP);
  pinMode(RECORD_IN, INPUT_PULLUP);
  pinMode(MODE_IN, INPUT_PULLUP);

  analogReference(DEFAULT);
  for (x = 0; x < 100; x++) {
    value += analogRead(COS_IN);
  }

  pinMode(LED_INIT, OUTPUT);
  pinMode(LED_HEARTBEAT, OUTPUT);
  pinMode(LED_PTT, OUTPUT);
  pinMode(LED_COS, OUTPUT);
  pinMode(LED_CWID, OUTPUT);
  pinMode(LED_VOICEID, OUTPUT);
  pinMode(LED_RECORDING, OUTPUT);

  // pinMode(CW_AUDIO, ????);

  pinMode(PTT_OUT, OUTPUT);
  pinMode(PLAY_OUT, OUTPUT);
  pinMode(RECORD_OUT, OUTPUT);
  pinMode(ERASE_OUT, OUTPUT);
  //  pinMode(RESET_OUT, OUTPUT);


  // preset digital outputs
  digitalWrite(PTT_OUT, HIGH);
  digitalWrite(PLAY_OUT, HIGH);
  digitalWrite(RECORD_OUT, HIGH);
  digitalWrite(ERASE_OUT, HIGH);
```

```
  // Initialize State Machine to IDLE
  //
  state = IDLE;
  last_state = DEFAULT;

  // Voice_Bd_Reset the timer counts
  //
  cw_timer = CW_TIMEOUT;
  voice_timer = VOICE_TIMEOUT;
  activity_timer = COS_TIMEOUT;
  last_id = CW_TIMEOUT;

  // voice_bd_reset globals
  //
  play_voice = HIGH;
  play_cw = HIGH;
  erase_voice = HIGH;
  record = HIGH;
  mode = MODE_BOTH;
  voice_bd_reset = HIGH;

  // initialize carrier operated squelch
  //
  cos_in = 0;
  mic_ptt = HIGH;

  // initialize states for printing
  //
  last_ptt = HIGH;
  last_cos = LOW;

  // Flash LED's during Init
  //
  digitalWrite(LED_INIT, LED_ON);
  digitalWrite(LED_HEARTBEAT, LED_ON);
  digitalWrite(LED_PTT, LED_ON);
  digitalWrite(LED_COS, LED_ON);
  digitalWrite(LED_CWID, LED_ON);
  digitalWrite(LED_VOICEID, LED_ON);
  digitalWrite(LED_RECORDING, LED_ON);

  delay(2000);

  digitalWrite(LED_INIT, LED_OFF);
  digitalWrite(LED_HEARTBEAT, LED_OFF);
  digitalWrite(LED_PTT, LED_OFF);
  digitalWrite(LED_COS, LED_OFF);
  digitalWrite(LED_CWID, LED_OFF);
  digitalWrite(LED_VOICEID, LED_OFF);
  digitalWrite(LED_RECORDING, LED_OFF);

  delay(2000);

  // Leave this LED on at end of INIT function
  digitalWrite(LED_INIT, LED_ON);

  Serial.begin(9600);
}

// Main Repeater Controller Loop
//
// Description:
```

```
//
// This routine will loop endlessely executing a simple state machine
// that performs necessary ID'ng of the 2m repeater for Part 97 rules.
// When the repeater is idle, every 30m the repeater will key a voice ID
// board to play a Voice ID message.  If from IDLE, the repeater generates
// transmission, Carrier Detect is determined at which time an ID will be
// sent when PTT is non-active.  Then, 10m later a CW Id will be sent.
//
// The routine calls series of functions to read input from some switches,
// from the PTT and Monitor lines to perform specific actions such as
// recording a message, playing a Voice or CW Id or erasing a Voice ID.
//
// Note:  Currently the CW Id is burned into the code.  User must keep the
//        voice ID same as CW Id.
//
// See the design document for schematic and details on operational control.
//
void loop()
{

  // Read Inputs
  readInputs();

  // Execute FrontPanel commands
  execCommands();

  // Execute State Machine
  switch (state)
    {
    case IDLE:
      last_state = IDLE;
      // reset timeout counters
      cw_timer = CW_TIMEOUT;
      voice_timer = VOICE_TIMEOUT;
      activity_timer = COS_TIMEOUT;
      state = TIMING_VOICE;
      break;

    case TIMING_CW:
      last_state = TIMING_CW;
      if (--cw_timer <= 0)
       state = CWID;
      else if ((mic_ptt == LOW) || (cos_in == HIGH)) {
       state = RPTR_ACTIVE;
       activity_timer = COS_TIMEOUT;
      }
      break;

    case TIMING_VOICE:
      last_state = TIMING_VOICE;
      // if we've timed out, immediately send appropriate ID
      if (--voice_timer <= 0) {
         if (mode == MODE_BOTH)
           state = VOICEID;
         else
           state = IDLE;
      }
      // Check to see if Repeater has gone active
      else if ((mic_ptt == LOW) || (cos_in == HIGH)) {
       if (last_id >= CW_TIMEOUT) {
         state = ID_FIRST;
         last_id = 0;
```

```
       }
       else
         state = RPTR_ACTIVE;
      }
      break;

    case ID_FIRST:
      voice_timer = VOICE_TIMEOUT;
      last_state = ID_FIRST;
      if ((mic_ptt == HIGH) && (cos_in == LOW)) {
       sendCWID(msg);
       activity_timer = COS_TIMEOUT;
       state = RPTR_ACTIVE;
       last_id = 0;
      }
      else if (--activity_timer <= 0) {
       Serial.printf("Sending Timeout Msg\n");
       sendCWID(msgTimeout);
       sendCWID(msg);
       activity_timer = COS_TIMEOUT;
       state = RPTR_ACTIVE;
       last_id = 0;
      }
      break;

    case RPTR_ACTIVE:
      last_state = RPTR_ACTIVE;
      if ((mic_ptt == HIGH) && (cos_in == LOW)) {
       state = TIMING_CW;
       activity_timer = COS_TIMEOUT;
      }
      else if (--activity_timer <= 0) {
       Serial.printf("Sending Timeout Msg\n");
       sendCWID(msgTimeout);
       activity_timer = COS_TIMEOUT;
      }
      else if (--cw_timer == 0) {
       state = CWID;
      }
      break;

    case CWID:
      Serial.print("CW ID\n");
      last_state = CWID;
      sendCWID(msg);
      last_id = 0;
      state = IDLE;
      break;

    case VOICEID:
      Serial.print("VOICE ID\n");
      last_state = VOICEID;
      sendVoiceID();
      state = IDLE;
      break;

    default:
      break;
    }

  // This counter keeps track of last time we ID'd
  ++last_id;
```

```
  //
  // Blinky Heartbeat LED
  //
  digitalWrite(LED_HEARTBEAT, LED_ON);

  // wait and repeat
  delay(500);

  digitalWrite(LED_HEARTBEAT, LED_OFF);

  // wait and repeat
  delay(400);


    Serial.printf("DEBUG: last state: %s, new state: %s, cwtmr=%d, atmr=%d, vtmr=%d, last_id: %d\n",
dState[last_state], dState[state], cw_timer, activity_timer, voice_timer, last_id);
 }
```