```
// preamp_01.h
// Port definitions, constants and global variables


#define BT_RX         0                 // Bluetooth / serial receive data (Arduino defaults)
#define BT_TX         1                 // Bluetooth / serial transmit data
#define ENC_1         3                 // two (interrupt-enabled) pins to read the encoder
#define ENC_2         2                 // ...interrupt-enabled pins make the library more
efficient...
#define ENC_SWITCH    4                 // the encoder push-button switch
#define LCD_PORTS    10, 9, 8, 7, 6, 5 // two control and four data
#define IR_PORT      A0                // The Infrared sensor
#define VOL_VAL      11                // pwm output 0 - 3.3V
#define RELAY_DS     A3                // shift register data send
#define RELAY_CLK    A4                // shift register clock
#define RELAY_STO    A5                // shift register latch (store) data
#define RELAY_COUNT   8                // for error checking
#define FLAG         13                // show activity


#define OFF           LOW               // 0 is off (no current in the relay)
#define ON            HIGH              // 1 is on (relay is pulling current)

#define DSP_RELAY      7                // enable / disable the MiniDSP on relay / register bit 7

#define ACTIVE_INPUT   0               // EEPROM address for non-volatile storage of active input
#define ACTIVE_DSP     1               // remember if the dsp was active
#define ACTIVE_VOLUME  2               // remember if tape monitor was active
#define ACTIVE_BALANCE 3

#define IR_DSP          4              // IR codes (replace with actual values)
#define IR_VB_UP        5              // Volume up and balance right
#define IR_VB_DOWN      3              // Volume down and balance left
#define IR_INPUT_UP     6              // Next input
#define IR_INPUT_DOWN   1              // Previous input
#define IR_VOL_BAL      2              // Toggle focus between volume and balance

#define COMMAND_TIMEOUT 5000           // 5 seconds

#define VOL_MAX      168              // corresponds to 5V from the top of the potentiometer
#define VOL_DELTA      4              // Volume change speed
#define DSP_RELAY      7              // Relay for switching the DSP


// Temporary (test) port definitions using the Uno

/*
#define IR_PORT       8                // The Infrared sensor
#define LCD_PORTS     6, 7, 2, 3, 4, 5  // two control and four data
#define ENC_SWITCH    9
#define ENC_1        10
#define ENC_2        11
#define VOL_CS       A0                // analog in to read pot. position
#define IR           A2                // The Infrared sensor
#define RELAY_DS     A3                // shift register data send
#define RELAY_CLK    A4                // shift register clock
#define RELAY_STO    A5                // shift register latch (store) data
#define RELAY_COUNT   8                // for error checking
```

```
//  THE MAIN SKETCH

#include "globals.h"
#include <LiquidCrystal.h>
#include <IRremote.h>
#include <Encoder.h>
#include <EEPROM.h>
// #include <SPI.h>

// Global Variables

boolean Standby = false;               // 0 means all is off and quiet; 1 is operating
LiquidCrystal Lcd(LCD_PORTS);          // LCD in 4-pin mode
IRrecv Irrecv(IR_PORT);                // Single pin to read IR data
decode_results Results;                // Output from IR receiver
boolean IrVolMode = true;              // IR has volume and balance modes with the Apple remote
Encoder Enc(ENC_1, ENC_2);             // Create an encoder object
long EncPos;                           // Accumulator for encoder state change recognition
byte EncMode = 0;                      // Encode mode: volume, input, balance, DSP
int Volume = 0;                        // Volume value from 0 to 255
int Balance = 0;                       // Balance - a delta to subtract from left, add to right
int Input = 1;                         // Default to CD input for first boot-up
byte CommandMode = 1;                  // IR and Rotary Controller (and maybe Bluetooth) "focus"
boolean CommandActive = false;         // true if we're processing a command
unsigned long EncCommandActiveTime;    // Time of start of new command active state
boolean Monitor = false;               // Default to source, not monitor
boolean Dsp = true;                    // DSP is active by default
boolean Mute = true;                   // MUTE is active at startup
byte RelayState = 0b10000000;          // Initialize to default state of the 8 relays
static char * Relays[] = {
  "Phono", "CD  ", "DAC  ", "Tuner", "Aux. ", "DSP  "};
int Source = 0;

void inputSet()                         // select a specific input and mute the others
//////////////////////////
{
    relaySetInput();
    displayInput();
}

void dspSet()                           // Cosmetic again - enable or disable the DSP
{
  relaySetRelay(DSP_RELAY, !Dsp);       // Wired inverted: normally on
  displayDSP();
}

// Action functions

void goStandby()
////////////////
{
  displayStatus("goStandby");

  //   Store volume and source values for next time if they have changed
  if (EEPROM.read(ACTIVE_INPUT) != Input)
    EEPROM.write(ACTIVE_INPUT, Input);
  if (EEPROM.read(ACTIVE_DSP) != Dsp)
    EEPROM.write(ACTIVE_DSP, Dsp);
```

```
  if (EEPROM.read(ACTIVE_VOLUME) != Volume)
    EEPROM.write(ACTIVE_VOLUME, Volume);
  if (EEPROM.read(ACTIVE_BALANCE) != Balance)
    EEPROM.write(ACTIVE_BALANCE, Volume);

  // Turn off all the relays to conserve power

  relayUpdateShiftRegister(0b00000000);

  // Display a message
  displaySignoff();

  Standby = true;
}

// Command functions



void setup()
////////////
{
  // Initialize relay ports and set all to their default states (using initialized value of
RelayState)

relayInit(RelayState);
  // Mute is 1 (enabled); all others are zero

  // Initialize display ports and print splash screen
  displayInit();
  displaySplash(1000);                              // Show splash screen for 1 second

  // Initialize rotary encoder ports
  //displayStatus("Encoder Init");
  encoderInit();

  // Initialize IR port
  //displayStatus("IR Init");
  IRInit();

  /* Disable for now
   // Initialize bluetooth

   Serial.begin(9600);     // Bluetooth dongle attached to the serial port
   */

  // Initialize volume control

  volumeInit();

  // Recall source, DSP settings and set signal path
  //    EEPROM.write(ACTIVE_VOLUME, 180);
  if (EEPROM.read(ACTIVE_INPUT == 255))
  {                                  // First time: set default values into EEPROM
    EEPROM.write(ACTIVE_INPUT, Input);
    EEPROM.write(ACTIVE_DSP, Dsp);
    EEPROM.write(ACTIVE_VOLUME, Volume);
    EEPROM.write(ACTIVE_BALANCE, Balance);
```

```
  }
  // Now get values (either as saved or new values just written)
  Input   = EEPROM.read(ACTIVE_INPUT);
  Dsp     = EEPROM.read(ACTIVE_DSP);
  Volume  = EEPROM.read(ACTIVE_VOLUME);
  Balance = EEPROM.read(ACTIVE_BALANCE);

  inputSet();                     // set the selected input to be active (connected)
  dspSet();                       // set the DSP on or off

  // Unmute the output

  volumeRamp(0, Volume);          // ramp up the volume slowly

  // Display source, volume, DSP flag

  displayInit();
}


void loop()
////////////{
  // Operating assumption is that everything is in a valid state.
  // The job of the loop is to recognize commands that change system state,
  // and to implement the change.
  // Each input source tests for an input, interprets it, calls the
  // appropriate implementation function, and updates the display.
  // Each subroutine blocks until it's finished processing a command, if there's one to handle.

  if (0) //(Standby)
    Standby = !encoderSwitch();             // check to see if the encoder switch has been pressed
  else
  {  // Process commands from each source in turn
    encoderCommand();                      // Used for volume, source selection, balance and DSP
select.
    // We store the volume and display it for the user.
    IRCommand();                           // Listen for a command: volume, source, other controls.
//    bluetoothCommand();                    // Bluetooth interface: same commands as IR.
  }
}
```

```
// Display Management
//////////////////////

void displayStatus(char * message)                      // utility function for debugging
///////////////////////////////////
{
  Lcd.setCursor(0, 1);
  Lcd.print("                    ");
  Lcd.setCursor(0, 1);
  Lcd.print(message);
  delay(1000);
//  Lcd.setCursor(0, 1);
//  Lcd.print("*                   ");
}

void displayValue(int val)
{
  Lcd.setCursor(12, 1);
  Lcd.print(val);
}

void printError(char* error)                    // Print an error message on the display
////////////////////////////
{
  //  Lcd.setCursor(0,0);
  //  Lcd.print(error);
}

void displayBalance()
////////////////////////////////////
{
  Lcd.setCursor(0,1);
  Lcd.print("----------------");
  if (Balance == 0)
  {
    Lcd.setCursor(7, 1);
    Lcd.print("><");
  }
  else
  {
  Lcd.setCursor(7 + map(Balance, 0, 128, 0, 15), 1);
  Lcd.print("<>");                    // A different symbol when we're not centered
  }
}

void displayVolume()
////////////////////
{
//  displayStatus("displayVolume");
  int graphic = map(Volume, 0, VOL_MAX, 0, 16);
  int v = (31.5 - (0.5 * (VOL_MAX - Volume)));
  Lcd.setCursor(11, 0);
  Lcd.print("    ");
  Lcd.setCursor(11, 0);
  if (v >= 0)
  {
    Lcd.print(" ");
    if (v < 10)
```

```
      Lcd.print("0");
  }
  Lcd.print(v);
  Lcd.setCursor(0, 1);
  for (int i = 0; i < graphic; i++)
    Lcd.print(">");
  for (int i = graphic; i <= 15; i++)
    Lcd.print("-");
}

void displayInput()
////////////////////////////
{
  static const char* inputs[] = {
    "DAC     ",
    "CD      ",
    "Tuner   ",
    "Aux.    ",
    "Phono   "        };  // padding spaces to make sure we overwrite whatever was there before
  Lcd.setCursor(0, 0);
  if (Input < 5 && Input >= 0)
    Lcd.print(inputs[Input]);
  else
  {
    Lcd.print(Input);
    Lcd.print("!!!!");
  }
}

void displayDSP()
////////////////////////////
{
  Lcd.setCursor(14, 0);
  if (Dsp)
    Lcd.print(" D");
  else
    Lcd.print(" -");
}

void displayMode(int val)
////////////////////////////
{
    switch (val)
    {
      case 0:
      {
        displayVolume();
        break;
      }
      case 1:
      {
        Lcd.setCursor(0, 1);
        Lcd.print("     Input      ");
        break;
      }
      case 2:
      {
        displayBalance();
```

```
        break;
      }
      case 3:
      {
        Lcd.setCursor(0, 1);
        Lcd.print("      DSP       ");
      }
    }
}

void displaySplash(int ms)
//////////////////////////        Show splash screen, then pause for ms milliseconds
{
  Lcd.begin(16, 2);                       // LCD size may change in final
  Lcd.setCursor(0, 0);
  Lcd.print(" Arduino & ESP");
  Lcd.setCursor (2, 1);
  Lcd.print(" RC Preamp");
  delay(2000);
  Lcd.clear();
}

void displaySignoff()
//////////////////////
{
  Lcd.clear();
  Lcd.setCursor(0, 1);
  Lcd.print("Powering down");
  delay(2000);
}

void displayInit()
//////////////////
{
  Lcd.clear();
  displayVolume();
  displayInput();
  displayDSP();
}
```

```
// RELAYS
/////////

void relayInit(byte state)
//////////////////////////
{
  pinMode(RELAY_STO, OUTPUT);
  pinMode(RELAY_CLK, OUTPUT);
  pinMode(RELAY_DS, OUTPUT);
  relayUpdateShiftRegister(state);
}

void relayUpdateShiftRegister(byte value)   // Send new values to the relays
//////////////////////////////////////
{
  digitalWrite(RELAY_STO, LOW);                   // Turn on the enable
  shiftOut(RELAY_DS, RELAY_CLK, MSBFIRST, value); // Arduino function to send data
  digitalWrite(RELAY_STO, HIGH);                  // Turn off the enable
}

void relaySetInput()
////////////////////////////
{
  RelayState = RelayState & 0b11100000;           // zero out all inputs, leaving monitor, DSP and
mute alone
  bitSet(RelayState, Input);                      // turn on the appropriate input, from 0 to 4
  relayUpdateShiftRegister(RelayState);           // send the updated relay settings
}


void relaySetRelay(int relay, boolean value) // Set one bit of the relay data and send via the
shift register
// Used for monitor, DSP, and Mute functions
{
  if (value)                          // We're going to turn the relay on
  {
    bitSet(RelayState, relay);
  }
  else                                // We're going to turn the relay off
  {
    bitClear(RelayState, relay);
  }
  relayUpdateShiftRegister(RelayState);    // Send all eight relay bits out at once
}

void relayInputChange(int inc)
////////////////////////////
{
  Input = (Input + inc + 5) % 5;
  relaySetInput();
  displayInput();
}
```

```
// Rotary Encoder command handling
////////////////////////////////

void encoderInit()
//////////////////
{
  pinMode(ENC_SWITCH, INPUT);                       // Encoder's push button
  digitalWrite(ENC_SWITCH, HIGH);                   // Set pullup resistor
//  pinMode(ENC_1, INPUT);                            // First input
//  digitalWrite(ENC_1, HIGH);
//  pinMode(ENC_2, INPUT);                            // Second input
//  digitalWrite(ENC_2, HIGH);
  EncPos = Enc.read();
}

boolean encoderSwitch()
///////////////////////

// Return true if buttonpress, false if no button is pressed
{
  static boolean lastval;
  boolean esw = digitalRead(ENC_SWITCH);
  if ((esw == HIGH) || (esw == lastval))
  {
    lastval = esw;                      // save the value we read
    return false;                       // high or no change
  }
  delay(25);                            // it's low, wait for debounce
  if (digitalRead(ENC_SWITCH) == LOW)
  {
    lastval = LOW;
    delay(20);
    return true;
  }
}

void encProcessRotation()
/////////////////////////
{
  long newEnc = Enc.read();
  int inc;
 // Lcd.setCursor(7, 0);
 // Lcd.print(EncMode);
  delay(200);
  if (newEnc != EncPos)                  // rotation detected
  {
    EncCommandActiveTime = millis();     // Retart the timer
    inc = (newEnc < EncPos) ? -1 : 1;
//    if (inc == -1) displayStatus("L"); else displayStatus("R");
    switch (EncMode)
    {
    case 0: // volume mode
      if (inc < 0)
        volumeDown();
      else
        volumeUp();
      break;
    case 1: // input mode
```

```
      if (inc < 0)
      {
        relayInputChange(-1);
      }
      else
      {
        relayInputChange(1);
      }
      break;
    case 2: // balance mode
      if (inc < 0)
        volumeBalanceLeft();
      else
        volumeBalanceRight();
      break;
    case 3: // DSP mode
      Dsp = !Dsp;
      dspSet();
      break;
    }
    EncPos = newEnc;
 // Lcd.setCursor(9, 0);
 // Lcd.print(EncMode);

  }
}

void encoderCommand()
//////////////////////
{
  // The rotary encoder is by default in Volume mode. Rotating left or right will alter the volume.
  // First press shifts it into Input mode. Rotating left or right will alter the source input.
  // Next press shifts it into Balance mode. Rotating left or right will alter the balance.
  // Next press shifts it into DSP mode. Rotation left disables the DSP; right enables it.
  // Next press shifts back to Volume mode.
  // After COMMAND_TIMEOUT milliseconds active without change, the mode returns to Volume - the
default display.

  if (encoderSwitch())
  {
    EncMode = (EncMode + 1) % 4;
    displayMode(EncMode);
    if (EncMode != 0)
      EncCommandActiveTime = millis();     // Start the timer
      switch (EncMode)
      {
        case 0:
        {
          displayVolume();
          break;
        }
        case 1:
        {
          displayInput();
          break;
        }
        case 2:
        {
```

```
          displayBalance();
          break;
        }
        case 3:
        {
          displayDSP();
          break;
        }
      }
    return;
  }

  // no button press, so check timer and process rotations

  if ((EncMode > 0) && (millis() - EncCommandActiveTime) > COMMAND_TIMEOUT)
  {
    EncMode = 0;
    displayVolume();
      return;
  }

  // mode still active, so process rotation

    encProcessRotation();
}
```

```
// Infrared remote control
//////////////////////////

void IRInit()
//////////////
{
  Irrecv.enableIRIn();    // start the receiver
//  displayStatus("Enable IR");
}

boolean IRCommand()
///////////////////
{
//  displayStatus("IRCommand");
  static byte cmd;
  if (Irrecv.decode(&Results))
  //displayStatus("IR");
  {
    cmd = (Results.value & 0x00007000) / 0x1000;
//    Lcd.print(cmd);
    delay(500);
    switch (cmd)                // mask out the required bits
    {
    case IR_DSP:
      Dsp = !Dsp;               // Invert the DSP global
      dspSet();                 // Set the relay
      break;
    case IR_VB_UP:
      if (IrVolMode)            // In volume mode,
        volumeUp();             // increment the volume
      else
        volumeBalanceRight();   // otherwise adjust balance
      break;
    case IR_VB_DOWN:
      if (IrVolMode)
        volumeDown();
      else
        volumeBalanceLeft();
      break;
    case IR_INPUT_UP:
      relayInputChange(1);      // Next input, with wraparound
      break;
    case IR_INPUT_DOWN:
      relayInputChange(-1);     // Previous input, with wraparound
      break;
    case IR_VOL_BAL:
      if (IrVolMode = !IrVolMode)   // Flip between volume and balance
        displayVolume();
      else
        displayBalance();
      break;
    default:
      break;
    }
    Irrecv.resume();
  }
}
```

```
// Volume control MiniDSP
//////////////////////////

void volumeInit()
//////////////////
{
  pinMode(VOL_VAL, OUTPUT);                      // Chip select
  analogWrite(VOL_VAL, 0);                       // 0 = mute.
  volumeSet(Volume, Balance);
  displayVolume();
}

void volumeUp()
//////////////////////
{
  volumeSet(Volume + VOL_DELTA, Balance);
//  displayStatus("Volume Up");
}

void volumeDown()
////////////////////////
{
//  displayStatus("Volume Down");
  volumeSet(Volume - VOL_DELTA, Balance);
}

void volumeBalanceRight()
//////////////////////////   Not functional with the MiniDSP - there's only a single vol. control
{
  Balance = Balance + 5;
  volumeSet(Volume, Balance);
  displayBalance();
}

void volumeBalanceLeft()
////////////////////////
{
  Balance = Balance - 10;
  volumeSet(Volume, Balance);
  displayBalance();
}

void volumeSet(int vol, int bal)
////////////////////////
{
  vol = min(VOL_MAX, max(0, vol));              // don't allow it to go out of range
  analogWrite(VOL_VAL, vol);
//  displayValue(vol);
  Volume = vol;                                 // Set the globals
  Balance = bal;
  displayVolume();
}

void volumeRamp(int start, int vol)
///////////////////////////////////
{
  for (int i=start; i<vol; i++)
  {
```

```
    volumeSet(vol, Balance);
    delay(100);
  }
}
```