Blackstomp Library

gaindoubler.ino -- simple effectModule Pedal controlling only volume

~~~~~~

stereochorus.ino -- effectModule Pedal chorus effect. Uses classes fractionalDelay and oscillator.

~~~~~~

effectmodule.h .cpp -- defines the Base effectModule Class. Instead of constructor and destructor methods use "virtual" init() and deInit() methods defined in each descendant effectModule object. Introduces a virtual process() method and various virtual Controller and Button methods. "virtual" means that the method will be defined in detail (and perhaps differently) in each descendant object, not here in the Base Class where it is just declared.

Also uses typedef enum's inside struct's to define various mode, controller, button, encoder, and ble variables. See example of use below:

~~~~~~

```
typedef enum                    // NORTH=0, SOUTH=1, EAST=2, WEST=3    An enum assigns numbers to Labels.
                                //  The computer uses the numbers while you can use the more descriptive Labels.
                                // Typedef allows you to define your own type instead of using the usual int, bool, float, etc.
{ NORTH,
  SOUTH,
  EAST,
  WEST,
} DIRECTIONS;
```
~~~~~~

```
DIRECTIONS directions;    // declare "directions" as type "DIRECTIONS" defined above (a special enum type)
directions = EAST;        //set directions = 2
directions = 3;           //set directions = WEST

struct WIND               // a struct is essentially a Class definition with various type attributes
{  DIRECTIONS wind_direction;
   int speed;
   int min;
   int max;
   bool rain;
   int temperature;
};

WIND monday_wind;                          //declare a Monday wind weather object
monday_wind.wind_direction = NORTH;
monday_wind.speed = 35;
monday_wind.rain = true;

WIND wind_week[7];                         //declare a Monday through Sunday weather array
wind_week[0].wind_direction = NORTH;  //set Monday's wind direction
wind_week[1].speed = 35;                   //set Tuesday's wind speed
```

~~~~~~

control.h .cpp /   ledindicator.h .cpp   -- Defines the classes **controlInterface** and **ledIndicator**. These functions manage all possible LEDs and controllers such as potentiometers. An LED can not only be turned on and off but can also be set up to blink in any number of ways. The pot controllers can be set up as a control voltage with certain ranges, as a selector switch, as a toggle switch, as a momentary switch or as a tap-tempo switch. (The **button_task**( ) is found in **blackstomp.cpp**.)

Within each of these two classes a rather involved function is declared to continuously manage the LEDs and Controllers in any of their various modes of operation:
~~~~~~

```
friend void blinktask(void* arg);
friend void controltask( void* arg);
```

The "friend" declaration means that **controltask** is not actually a **controlInterface** Class method, and **blinktask** is not an **ledIndicator** Class method. They are defined outside the scope of the Class but are given access to all the public and protected members of the Class in which they are declared.

Note that in **ledindicator**.cpp, **blinktask** is defined with "void **blinktask**(void* arg) { }" and not "void **ledIndicator**::**blinktask**(void* arg) { }" The same is done with **controltask**. They are not Class methods but can access the Class as set up by the "friend" declaration.

Each of these functions show up again inside their class Init( ) in a special "xTask" function:

```
xTaskCreatePinnedToCore(blinktask, "blinktask",4096,(void*)this,priority,NULL,0);
xTaskCreatePinnedToCore(controltask, "controltask",4096,(void*)this,priority,NULL,0);
```

The ESP32 microprocessor is special in that it has 2 cores that allows two program threads to run simultaneously. The **xTaskCreatePinnedToCore**( ) is an ESP32 Library function used to set up specific tasks (functions) to run in one of the two ESP32 cores. The last argument "0" in the xTask functions above indicate that **blinktask** and **controltask** are both assigned to the ESP32 Core 0. Several other tasks defined in blackstomp.cpp are also assigned to Core 0. The only task assigned to ESP32 Core 1 will be the **i2s_task** which manages the audio data streams to and from the codec; in this way the user control functions will not interrupt the audio into and out of the codec ADC and DACs.

blinktask and **controltask** are set up to run continuously, always watching for user control input to alter the audio streams through an effectModule( ) in some way or to change the LED indicators. **blinktask( )** sets up a somewhat infinite loop with "while(! I -> terminaterequest) { }" **controltask( )** sets up an infinite loop with "while(true) { }"

All the tasks running in Core0 must be programmed for real-time response. The user must feel that any button press or knob turn has an immediate effect even with all the tasks competing for processor time. The main programming tool for creating this real-time response is **FreeRTOS** (Free Real Time Operating System). The ESP32 development board comes with FreeRTOS firmware already installed on it which is supported by the Arduino IDE as well. The FreeRTOS is a Real-time Operating System used to run multiple tasks individually. This firmware allows the ESP32 board to multitask via

API functions. Both tasks described above use FreeRTOS commands to improve responses to user input.

For one example, since the human reaction time is around 150 ms, the tasks can be slowed down to allow other tasks to do their thing without affecting the perceived user response time. This is easily done with an RTOS Delay command at the start of the task's infinite loop. **controltask( )** uses "vTaskDelay(1)" and **blinktask( )** uses "vTaskDelay(10)", delays of about 1ms and 10ms, depending upon the processor "Tick" rate. Unlike the Arduino delay( ), vTaskDelay() is a non-blocking delay; it lets other tasks continue working while the one task is idle.

blinktask( ) uses another FreeRTOS construct, the Semaphore. A Semaphore is a permit to access the processor which is passed between functions. It is created in **ledIndicator::init( )** with the line "**xSemaphore = xSemaphoreCreateBinary( );**" Each ledIndicator class method then must check on the xSemaphore's availability before it can "Take" it, perform its function, and then "Give" the Semaphore back. The "Take" and "Give" lines framing the method's main lines are shown here:

```
if(xSemaphoreTake(l->xSemaphore,(TickType_t)1) == pdTRUE);
...
xSemaphoreGive(xSemaphore);
```

**~~~~~~**

bsdsp.h .cpp defines the following 3 classes used for digital signal processing:

biquadFilter -- implements a direct-form-2 biquad iir filter

oscillator -- implements an oscillator from a 256 element table of one waveform cycle.
It can use the **sine_table[ ]** from the file **dsptable.h**
It uses the function **lookupLinear( )** which can interpolate a fractional index into a wave table.
Value = table[integer part of index] + (fraction part of index) * (table[index + 1] - table[index])

fractionalDelay -- implements a delayed output by building a circular sample buffer sized for a given maxDelayInMs. You can then request a sample read at any delay value up to the maxDelay.
It uses the same fractional index interpolation implemented in the lookupLinear( ) shown above.

~~~~~~
~~~~~~

blackstomp.h .cpp -- Main file in Library

button_task( ) -- Along with **blinktask( )** and **controltask( )**, **button_task( )** completes the trio of user controller tasks. This one manages pushbuttons and switches. It is specifically programmed for the 3 push buttons on the ESP32_A1S_ES8388 board or the 4 push buttons on the ESP32_A1S_AC101 board. If not using either of these boards, you can still wire up 3 or 4 ESP32 pins as switches and use this task function to manage them. You will need to define each switch's mode of operation within an effectModule class instance - BM_TOGGLE, BM_MOMENTARY, BM_TAPTEMPO or EM_BUTTONS (rotary encoder mode).

The **button_task( )** is set up similar to **blinktask( )** and **controltask( )**. After configuring the ESP32 pins used for the switches as inputs with pull-ups, the task starts up an infinite loop with "while(true){ }" along with "vTaskDelay(1);" It then continuously services each switch according to its selected mode of operation. As with the other two user control tasks, this one is assigned to the ESP32 Core0 with the following statement found in blackstompSetup(effectModule* module) :

xTaskCreatePinnedToCore(**button_task**, "button_task", 4096, NULL, AUDIO_PROCESS_PRIORITY, NULL,0);

ESP32 #defines -- The blackstomp.cpp file starts off with the following ESP32 pin defines:

6 ADC Pot or Slider Controller pins: **P1_PIN** through **P6_PIN**
4 Digital Button pins: **RE-BUTTON_PIN**, **RE_PHASE0_PIN**, **RE_PHASE1_PIN**, **FS_PIN**
2 Digital LED pins: **MAINLED_PIN**, **AUXLED_PIN**
2 OLED I2C Display pins: **SCK_PIN**, **SDA_PIN**

7 pin Button assignments for the AC101 or ES8388 boards

7 pin Codec assignments for the AC101 or ES8388 boards: I2S { **MCLK**, **BCK**, **WS**, **DO**, **DI** }, I2C { **SDA**, **SCK** }, plus **ADDR** and **NUM**

The pin assignments shown above may need to be changed for other ESP32/Codec board configurations. To allow for easier library updating these assignments can be left as is while another .h file is created with #define overrides in the following format:

```
 #ifdef __RESETPIN__
 #undef __RESETPIN__
#define __RESETPIN__ 1
       #endif
```

Audio Processing constants:
FRAMELENGTH 64 - 32 samples for Right, 32 samples for Left (4 bytes per sample give a total of 256 Bytes)
SAMPLECOUNT FRAMELENGTH/2 - 32 samples per channel
CHANNELCOUNT 2 - Left and Right
FRAMESIZE FRAMELENGTH*4   (4 bytes per sample give a total of 256 Bytes)
AUDIO_PROCESS_PRIORITY 10 - highest priority given in xTaskCreatePinnedToCore( )
DMABUFFERLENGTH 32 - 32 bytes (4 samples Left, 4 samples Right) fed to codec DMA
DMABUFFERCOUNT 20 - 640 bytes, 160 samples (80 for Left, 80 for Right) fed to codec DMA
DEVICE_TYPE_deviceType = **DT_ESP32_A1S_ES8388**;

The following are declared "static" variables which means that they are effectively "global". Note the "_"

```
//  instances of classes created
static codec*              _acodec;
static effectModule*       _module     = NULL;
static bt_terminal*         btt;           // bluetooth BLE terminal
static controlInterface    _control ;
static ledIndicator        _mainLed ;
static ledIndicator        _auxLed  ;

static bool                _es8388Mode = true;              //codec
static uint8_t             _codecAddress = 0;               //codec
static bool                _outCorrectionGain = 1;  //controlInterface
static int                 _optimized Range = 2;     //controlInterface
```

```
//Buffers  used in i2s_task( )
static float                    wleft[ ] = {0,0};
static float                    wright[ ] = {0,0};
static int32_t                  inbuffer   [FRAMELENGTH];
static int32_t                  outbuffer [FRAMELENGTH];
static float                    inleft       [FRAMESIZE];
static float                    inright      [FRAMESIZE];
static float                    outleft      [FRAMESIZE];
static float                    outright   [FRAMESIZE];


struct EEPROMBUFFER            //See EEPROM functions below
{
        int controlvalue[6];
        int buttonvalue[4];
};
static EEPROMBUFFER    eeprombuffer ;    //eeprombuffer.controlvalue[i], eeprombuffer.buttonvalue[i]
static unsigned long         eepromupdatetime = 0;
static bool                     eepromrequestupdate = false;


static char*                    debugStringPtr = "None";
static float                    debugVars[ ] = {0, 0, 0, 0};

static volatile unsigned int  processedframe ;
static unsigned int             audiofps;    //frames per second

// calculated in i2s_task( ) using xthal_get_ccount( )
static unsigned int             usedticks;
static unsigned int             availableticks;
static unsigned int             availableticks_start;
static unsigned int             availableticks_end;
static unsigned int             usedticks_start;
```

static unsigned int **usedticks_end**;

~~~~~~

// Functions in blackstomp.cpp

Void **setDeviceType**(DEVICE_TYPE dt) //_deviceType = dt:
void **button_task**(void* arg) //task assigned to Core0 in blackstompSetup( ). Described above.
void **blackstompSetup**(effectModule* module) //placed in Setup( ) of all user Effects Pedal sketches
void **enableBleTerminal**(void) //enable Bluetooth

As an aid to debugging your own effects sketches the following Debug utilities can be used. The SystemMonitor can print out Debug variables, several System readings, and controller readings, all using the Serial.print( ) command of the Arduino Serial Monitor Tool. In addition, a Scope utility allows you to print out the codec's audio output waveform using the Arduino's Serial Plotter Tool.

void **setDebugStr**(const char* str) //debugStringPtr = (char* str)
void **setDebugVars**(...) //fill debugVars[0 through 3]
void **sysmon_task**(void *arg) //task assigned to Core0 in runSystemMonitor( ).
void **runSystemMonitor**(...) //Serial.print Monitor of System values for debugging
void **scope_task**(void *arg) //task assigned to Core0 in runScope( ).
void **runScope**(...)
void **scopeProbe**(float sample, int channel)

The I2S serial interface is used to move audio data between the ESP32 Microprocessor and the Codec, out of the Analog to Digital Converters and into the Digital to Analog Converters. This Codec Audio Input/Output is the most time intensive and time sensitive task of the whole program. As such, it is the only task that is assigned to Core 1 of the ESP32 dual core processor.

void **i2s_task**(void* arg) //task assigned to Core1 in blackstompSetup( ).
void **i2s_setup**() //run in blackstompSetup( ).
~~~~~~

Controller and button values are periodically stored in the ESP32 EEPROM memory so that the Pedal Effect can be powered up in the same state it was in when powered down.

void **eepromupdate_task**(void* arg)	//store current button values in ESP32 eeprom
void **eepromsetup_task**(void* arg)	//task assigned to Core0 in blackstompSetup( ).

Base class **codec** public virtual functions (to be overriden by more specific functions defined in **codec**::**AC101Codec** and **codec**::**ES8388Codec).**

void **codecsetup_task**(void* arg)	//task assigned to Core0 in blackstompSetup( ).
bool **analogBypass**(...) /	/return _acodec->analogBypass(bypass, bm);
bool **analogSoftBypass**(...)	//return _acodec->analogSoftBypass(bypass, bm);
void **setMicGain**(int gain)	//_acodec->setMicGain(gain)
int **getMicGain**( )	//return _acodec->getMicGain()
void **setInGain**(int gain)	//_acodec->setInGain(gain)
void **optimizeConversion**(int range)	//_acodec->optimizeConversion(range) else _optimizedRange = range
void **setMicNoiseGate**(int gate)	//_acodec->setMicNoiseGate(gate)
int **getMicNoiseGate**( )	//return _acodec->getMicNoiseGate( )
int **getInGain**( )	//return _acodec->getInGain( )
void **setOutVol**(int vol)	//_acodec->setOutVol(vol)
int **getOutVol**( )	//return _acodec->getOutVol( )

int **getTotalCpuTicks**( )	//return availableticks;
int **getUsedCpuTicks**( )	//return usedticks;
float **getCpuUsage**( )	//return usedticks/availableticks
int **getAudioFps**( )	//return audiofps
void **framecounter_task**(void* arg)	//task assigned to Core0 in blackstompSetup( ).

codec.h .cpp   Audio Codec Drivers.

Codecs use two interface protocols. A 2-wire I2C interface is used to configure the chip, and the I2S is used to move the audio data. In the ES8388 codec the I2C interface is used to load and read 53 user programmable 8-bit registers that set up I/O connections, sampling rate, sample format, sample size, volume, filters, effects, etc. Only a few of these registers are of interest to the user and these are provided get and set virtual functions in the base codec class (listed above). Others can be set and left in their default settings. The **codec.h** and **.cpp** files deal with setting up the codec by loading its registers using the I2C interface.

Three possible codecs are indicated in the typedef enum **DEVICE_TYPE** below. Used in the creation of the **_deviceType** variable: **DEVICE_TYPE _deviceType** = DT_ESP32_A1S_ES8388; (in **blackstomp.cpp**). Also used in **setDeviceType**(DT_ESP32_A1S_ES8388) to set the variable **_deviceType** (in any user effect pedal sketch)**.** The **_deviceType** variable is then implemented in **i2s_setup**( ) and **codecsetup_task**( ).

```
typedef enum
{       DT_ESP32_A1S_AC101,
        DT_ESP32_A1S_ES8388,
        DT_WROVER_WM8776
}  DEVICE_TYPE;

typedef enum
{
        BM_LR,
        BM_L,
        BM_R
}  BYPASS_MODE;
```

The codec.cpp file starts out with #define constants that name the many codec registers found on the AC101 and the ES3833. These are the registers used to set up the codec ADCs and DACs (Analog to Digital and Digital to Analog Converters) and other parameters. Several of the data values loaded into these registers, such as SampleRate, Word Size, Data Format, Clock Division, etc, are named and defined using c++ enum.

bool **codecBusInit**(int sdaPin, int sclPin, int frequency) //_codecWire = new TwoWire(0); using I2C Wire.h Library.

Next, the base codec class is created with some basic virtual functions defined above in blackstomp.cpp. Then two more specific codec classes are derived from the base class. These are specific to the AC101 Codec and the ES3833 Codec. Several functions defined in these derived classes override the virtual functions defined in the base class since they are specific to a particular codec. A number of other functions are defined to access specific codec setup registers. The 3 main codec functions are **init**( ), **readReg**( ), and **WriteReg**( ) (Initialize the codec, read a register, write to a register).

class **codec** { } // the base class, with all the virtual functions shown above in **blackstomp.cpp**
class **AC101Codec**: public **codec**
class **ES8388Codec**:public **codec**

~~~~~~
~~~~~~