# DaisySP

## Contents

Maps Daisy interface to STM32 HAL – I'd like to get all of this stuff tucked away somewhere inbetween the HAL, and User level So that I can start to slowly replace HAL stuff over time. Also I don't like that this throws a warning for every library file that doesn't use it… Possible solution: Move this to dsy_hal_interface.h – and the explicitly include it in the lower-level files that need it. Hardware related defines. keyboard switches shift register UART for

MIDI via TRS jacks on Field CD4051 Select Pin controls enums for controls, etc. All knobs connect to ADC1_INP10 via CD4051 mux Init Daisy Seed TODO: decide if this should be a part of the bsp init, or if users should have to init seed and board. Init Switches Init Gate Input Init Gate Output Init LED Driver 2x PCA9685 addresses 0x00, and 0x01 TODO: add multidriver support Init Keyboard Switches TODO: add cd4021 with parallel data support Init ADC (currently in daisy_seed).

# Daisy Patch BSP

## Description

Class that handles initializing all of the hardware specific to the Daisy Patch Board.

Helper funtions are also in place to provide easy access to built-in controls and peripherals.

## Credits

**Author:** Stephen Hensley **Date Added:** November 2019

## Data Types

### ctrl

These are the hardware controls accessed via hid_ctrl objects.

They can be accessed directly, or via the GetCtrl() function

```
enum ctrl
{
    KNOB_1,
    KNOB_2,
    KNOB_3,
    KNOB_4,
    CV_1,
    CV_2,
    CV_3,
    CV_4,
    CV_LAST,
    KNOB_LAST = CV_1,
};
```

### led

These are the LEDs connected to the LED Driver peripheral

They can be accessed via the dsy_led_driver module, or using the LED Helpers below

```
enum led
{
```

```
    LED_A4,
    LED_A3,
    LED_A2,
    LED_A1,
    LED_B4,
    LED_B3,
    LED_B2,
    LED_B1,
    LED_C4,
    LED_C3,
    LED_C2,
    LED_C1,
    LED_D4,
    LED_D3,
    LED_D2,
    LED_D1,
    LED_LAST
};
```

#### Init

Initializes the daisy seed, and patch hardware.

```
void Init();
```

### Audio Helpers

#### StartAudio

Starts the audio calling the specified callback

```
void StartAudio(dsy_audio_callback cb)
```

#### ChangeAudioCallback

Changes what callback is being called when audio is ready for new data.

```
void ChangeAudioCallback(dsy_audio_callback cb)
```

## LED helpers

Worth noting that all changes to LED brightness only apply UpdateLeds() is called.

#### SetLed

Sets the brightness of one of the LEDs

```
inline void SetLed(led ld, float bright)
```

#### ClearLeds

Sets the brightness of all LEDs to 0

```
inline void ClearLeds()
```

#### UpdateLeds

Writes the changes in brightness to the actual LEDs

```
inline void UpdateLeds()
```

#### GetCtrl

Returns an AnalogControl KNOB_1 through CV_4

```
inline AnalogControl GetCtrl(ctrl c)
```

## Public Members

These are in place to keep everything working for now.

All of these members can be accessed directly, and used with the rest of the C-Based libdaisy library. alternate AnalogCtrl with public access Order of ADC Channels for accessing dsy_adc.h Mapping of LEDs via dsy_leddriver.h ADC related Hardware related de-

fines. Switches Encoder Extra Peripherals enums for controls, etc. All knobs connect to ADC1_INP10 via CD4051 mux Init Daisy Seed TODO: decide if this should be a part of the bsp init, or if users should have to init seed and board. Init Switches Encoder TODO Add Encoder support Init LED Driver 2x PCA9685 addresses 0x00, and 0x01 ADC Init Switches

LEDs are just going to be on/off for now. TODO: Add PWM support Encoder This is a Board

Specific File I don't think it actually belongs in the library. Any new piece of hardware can just have their own board file. This will allow minor pin changes, etc. not to require changing the library in a million places. Specifies whether generic initialization will be done within the daisy_seed_init, or not. Allows for more selective init Probably should move this to a dsy_handle.h So that it can be used in the other peripheral initializations, etc. (E.g. Audio needs both SAI, and I2C for most devices.) THIS BREAKS WHEN ITS INLINED? QSPI FMC SAI - Serial Audio Interface SAI1 - config SAI2 - config I2C - Inter-Integrated Circuit TODO: Add Config for I2C3 and I2C4 I2C 1 - (On daisy patch this controls the LED Driver, and the WM8731). I2C 2 - (On daisy patch this controls the on-board WM8731) ADC DAC GPIO Audio System Initialization (optional) WM8371 Codec support. TODO: Abstract the

colors of this driver. SDRAM for 32MB AS4C16M16SA (and 64MB equivalent). Thanks to

whoever this awesome person is: http://main.lv/writeup/stm32f4_sdram_configuration.md The Init function is basically a copy and paste. He has references to timing, etc. for now we're configuring the RAM to run at 108MHz To use these the .sdram_data/_bss sections must be configured correctly in the LINKER SCRIPT. using BSS is advised for most things, since the DATA section must also fit in flash in order to be initialized. Determines whether chip is initialized, and activated. This is only the pins that can change on a board-to-board basis. Pins that have functions that cannot be moved to another pin will be hardcoded into the driver.

- SDNWE is the only pin that i've seen move, Fixed maximums for paral-

lel/daisychained use These could be expanded TODO Fix hard coding of

these parameters Thinking about getting rid of this… If initialized to a single channel, its just that. If both initialized, then you get a quad callback. Stops transmitting/receiving audio. If the device supports hardware bypass, enter that mode. If the device supports hardware bypass, exit that mode. Default Callbacks

# AnalogControl

## Description

Hardware Interface for control inputs

Primarily designed for ADC input controls such as potentiometers, and control voltage.

## Credit:

**Author:** Stephen Hensley **Date Added:** November 2019

## General Functions

### Init

Initializes the control *adcptr is a pointer to the raw adc read value – This can acquired with dsy_adc_get_rawptr(), or dsy_adc_get_mux_rawptr()

sr is the samplerate in Hz that the Process function will be called at.

slew_seconds is the slew time in seconds that it takes for the control to change to a new value.

slew_seconds defaults to 0.002 seconds if not specified.

```
void Init(uint16_t *adcptr, float sr);
void Init(uint16_t *adcptr, float sr, float slew_seconds);
```

### Init_bipolar_cv

This Initializes the AnalogControl for a -5V to 5V inverted input

All of the Init details are the same otherwise

```
void InitBipolarCv(uint16_t *adcptr, float sr);
```

### Process

filters, and transforms a raw ADC read into a normalized range.

this should be called at the rate of specified by samplerate at Init time.

Default Initializations will return 0.0 -> 1.0

Bi-polar CV inputs will return -1.0 -> 1.0

```
float Process();
```

**Value**

Returns the current stored value, without reprocessing

```
inline float Value() const { return val_; }
```

# Encoder

## Description

Generic Class for handling Quadrature Encoders

## Files

hid_encoder.*

## Credits

**Author:** Stephen Hensley

**Date:** December 2019

Inspired/influenced by Mutable Instruments (pichenettes) Encoder classes

## General Functions

### Init

Initializes the encoder with the specified hardware pins.

Update rate should be the rate at which Debounce() gets called in Hertz.

```
void
Init(dsy_gpio_pin a, dsy_gpio_pin b, dsy_gpio_pin click, float update_rate);
```

### Debounce

Called at update_rate to debounce and handle timing for the switch.

In order for events not to be missed, its important that the Edge/Pressed checks be made at the same rate as the debounce function is being called.

```
void Debounce();
```

### Increment

Returns +1 if the encoder was turned clockwise, -1 if it was turned counter-clockwise, or 0 if it was not just turned.

```
inline int32_t Increment() const { return inc_; }
```

#### RisingEdge

Returns true if the encoder was just pressed.

```
inline bool RisingEdge() const { return sw_.RisingEdge(); }
```

#### FallingEdge

Returns true if the encoder was just released.

```
inline bool FallingEdge() const { return sw_.FallingEdge(); }
```

#### Pressed

Returns true while the encoder is held down.

```
inline bool Pressed() const { return sw_.Pressed(); }
```

#### TimeHeldMs

Returns the time in milliseconds that the encoder has been held down.

```
inline float TimeHeldMs() const { return sw_.TimeHeldMs(); }
```

# Switch

## Description

Generic Class for handling momentary/latching switches

## Files

hid_switch.*

## Credits

**Author:** Stephen Hensley

**Date:** December 2019

Inspired/influenced by Mutable Instruments (pichenettes) Switch classes

## Data Types

### Type

Specifies the expected behavior of the switch

```
enum Type
{
    TYPE_TOGGLE,
    TYPE_MOMENTARY,
};
```

### Polarity

Specifies whether the pressed is HIGH or LOW.

```
enum Polarity
{
    POLARITY_NORMAL,
    POLARITY_INVERTED,
};
```

#### Pull

Specifies whether to use built-in Pull Up/Down resistors to hold button at a given state when not engaged.

```
enum Pull
{
    PULL_UP,
    PULL_DOWN,
    PULL_NONE,
};
```

### General Functions

#### Init

Initializes the switch object with a given port/pin combo.

Parameters:

- pin: port/pin object to tell the switch which hardware pin to use.
- update_rate: the rate at which the Debounce() function will be called. (used for timing).
- t: switch type – Default: TYPE_MOMENTARY
- pol: switch polarity – Default: POLARITY_INVERTED
- pu: switch pull up/down – Default: PULL_UP

```
void
Init(dsy_gpio_pin pin, float update_rate, Type t, Polarity pol, Pull pu);

void Init(dsy_gpio_pin pin, float update_rate);
```

#### Debounce

Called at update_rate to debounce and handle timing for the switch.

In order for events not to be missed, its important that the Edge/Pressed checks be made at the same rate as the debounce function is being called.

```
void Debounce();
```

#### RisingEdge

Returns true if a button was just pressed.

```
inline bool RisingEdge() const { return state_ == 0x7f; }
```

#### FallingEdge

Returns true if the button was just released

```
inline bool FallingEdge() const { return state_ == 0x80; }
```

#### Pressed

Returns true if the button is held down (or if the toggle is on).

```
inline bool Pressed() const { return state_ == 0xff; }
```

#### TimeHeldMs

Returns the time in milliseconds that the button has been held (or toggle has been on)

```
inline float TimeHeldMs() const
```

# Parameter

```
Simple parameter mapping tool that takes a 0-1 input from an hid_ctrl.
```

TODO: Move init and process to .cpp file - i was cool with them being in the h file until math.h got involved for the log stuff.

## parameter class

## Data Types

### Curve

Curves are applied to the output signal

```
enum Curve
{
    LINEAR,
    EXP,
    LOG,
    CUBE,
    LAST,
};
```

### init

initialize a parameter using an hid_ctrl object.

hid_ctrl input - object containing the direct link to a hardware control source.

min - bottom of range. (when input is 0.0)

max - top of range (when input is 1.0)

curve - the scaling curve for the input->output transformation.

```
inline void init(AnalogControl input, float min, float max, Curve curve)
```

### process

processes the input signal, this should be called at the samplerate of the hid_ctrl passed in.

returns a float with the specified transformation applied.

```
inline float process()
```

**value**

returns the current value from the parameter without processing another sample. this is useful if you need to use the value multiple times, and don't store the output of process in a local variable.

```
inline float value() { return val_; }
```

Limitations: - For now speed is fixed at ASYNC_DIV128 for ADC Clock, and SAMPLE-TIME_64CYCLES_5 for each conversion. - No OPAMP config for the weird channel - No oversampling built in yet These are getters for multiplexed inputs on a single channel (up to 8 per ADC input).

# PER_I2C

Driver for controlling I2C devices

TODO:

- Add DMA support
- Add timing calc based on current clock source freq.
- Add discrete rx/tx functions (currently other drivers still need to call ST HAL functions).

Errata:

- 1MHZ (FastMode+) is currently only 886kHZ

## Types

#### dsy_i2c_periph

Specifices the internal peripheral to use (these are mapped to different pins on the hardware).

```
typedef enum
{
    DSY_I2C_PERIPH_1,
    DSY_I2C_PERIPH_2,
    DSY_I2C_PERIPH_3,
    DSY_I2C_PERIPH_4,
} dsy_i2c_periph;
```

#### dsy_i2c_pin

List of pins associated with the peripheral. These must be set in the handle's pin_config.

```
typedef enum
{
    DSY_I2C_PIN_SCL,
    DSY_I2C_PIN_SDA,
    DSY_I2C_PIN_LAST,
} dsy_i2c_pin;
```

#### dsy_i2c_speed

Rate at which the clock/data will be sent/received. The device being used will have maximum speeds.

**1MHZ Mode is currently 886kHz**

```
typedef enum
{
    DSY_I2C_SPEED_100KHZ,
    DSY_I2C_SPEED_400KHZ,
    DSY_I2C_SPEED_1MHZ,
    DSY_I2C_SPEED_LAST,
} dsy_i2c_speed;
```

#### dsy_i2c_handle

this object will be used to initialize the I2C interface, and can be passed to dev_ drivers that require I2C.

```
typedef struct
{
    dsy_i2c_periph periph;
    dsy_gpio_pin  pin_config[DSY_I2C_PIN_LAST];
    dsy_i2c_speed speed;
} dsy_i2c_handle;
```

### Functions

#### dsy_i2c_init

initializes an I2C peripheral with the data given from the handle.

Requires a dsy_i2c_handle object to initialize.

```
void dsy_i2c_init(dsy_i2c_handle *dsy_hi2c);
```

#### dsy_i2c_hal_handle

Returns a pointer to the HAL I2C Handle, for use in device drivers.

```
I2C_HandleTypeDef *dsy_i2c_hal_handle(dsy_i2c_handle *dsy_hi2c);
```

Section Attributes Currently Sample Rates are not correclty supported. We'll get there.

## per_tim

General purpose timer for delays and general timing.

TODO:

- Add configurable tick frequency – for now its set to the APB1 Max Freq (200MHz)
- Add ability to generate periodic callback functions

#### init

initializes the TIM2 peripheral with maximum counter autoreload, and no prescalers.

```
void dsy_tim_init();
```

#### start

Starts the timer ticking.

```
void dsy_tim_start();
```

### tick

These functions are specific to the actual clock ticks at the timer frequency which is currently fixed at 200MHz

#### get_tick

Returns a number 0x00000000-0xffffffff of the current tick

```
uint32_t dsy_tim_get_tick();
```

#### delay_tick

blocking delay of cnt timer ticks.

```
void     dsy_tim_delay_tick(uint32_t cnt);
```

### ms

These functions are converted to use milliseconds as their time base.

#### get_ms

returns the number of milliseconds through the timer period.

```
uint32_t dsy_tim_get_ms();
```

#### delay_ms

blocking delay of cnt milliseconds.

```
void     dsy_tim_delay_ms(uint32_t cnt);
```

### us

These functions are converted to use microseconds as their time base.

#### get_us

returns the number of microseconds through the timer period.

```
uint32_t dsy_tim_get_us();
```

#### delay_us

blocking delay of cnt microseconds.

```
void     dsy_tim_delay_us(uint32_t cnt);
```

Sets clock speeds, etc.