

# DaisySP

## Contents

<b>Core</b>	<b>6</b>
Memory Section Macros . . . . .	6
GPIO . . . . .	6
<b>Daisy Patch BSP</b>	<b>8</b>
Description . . . . .	8
Credits . . . . .	8
Data Types . . . . .	8
ctrl . . . . .	8
led . . . . .	8
Init . . . . .	9
Audio Helpers . . . . .	9
StartAudio . . . . .	9
ChangeAudioCallback . . . . .	9
Analog Control Reads . . . . .	10
LED helpers . . . . .	10
SetLed . . . . .	10
ClearLeds . . . . .	10
UpdateLeds . . . . .	10
GetCtrl . . . . .	10
Public Members . . . . .	10
<b>AnalogControl</b>	<b>18</b>
Description . . . . .	18
Credit: . . . . .	18
General Functions . . . . .	18
Init . . . . .	18

Init_bipolar_cv . . . . .	18
Process . . . . .	18
Value . . . . .	19
<b>Encoder</b>	<b>20</b>
Description . . . . .	20
Files . . . . .	20
Credits . . . . .	20
General Functions . . . . .	20
Init . . . . .	20
Debounce . . . . .	20
Increment . . . . .	20
RisingEdge . . . . .	21
FallingEdge . . . . .	21
Pressed . . . . .	21
TimeHeldMs . . . . .	21
<b>GateIn</b>	<b>22</b>
Description . . . . .	22
Files . . . . .	22
Credits . . . . .	22
General Functions . . . . .	22
Init . . . . .	22
<b>MidiHandler</b>	<b>23</b>
Description . . . . .	23
Credit . . . . .	23
Data . . . . .	23
MidiMessageType . . . . .	23
NoteOnEvent . . . . .	23
ControlChangeEvent . . . . .	24
MidiEvent . . . . .	24
MidiHandler . . . . .	24
Midi IO Modes . . . . .	24
Functions . . . . .	25
Init . . . . .	25
StartReceive . . . . .	25
Parse . . . . .	25

HasEvents . . . . .	25
PopEvent . . . . .	26
<b>Switch</b>	<b>27</b>
Description . . . . .	27
Files . . . . .	27
Credits . . . . .	27
Data Types . . . . .	27
Type . . . . .	27
Polarity . . . . .	27
Pull . . . . .	28
General Functions . . . . .	28
Init . . . . .	28
Debounce . . . . .	28
RisingEdge . . . . .	29
FallingEdge . . . . .	29
Pressed . . . . .	29
TimeHeldMs . . . . .	29
<b>hid_usb</b>	<b>30</b>
Description . . . . .	30
Credits . . . . .	30
Data Types . . . . .	30
UsbPeriph . . . . .	30
General Functions . . . . .	30
Init . . . . .	30
TransmitInternal . . . . .	30
TransmitExternal . . . . .	31
<b>Audio</b>	<b>32</b>
Description . . . . .	32
Credit . . . . .	32
SampleRate . . . . .	32
<b>Parameter</b>	<b>33</b>
parameter class . . . . .	33
Data Types . . . . .	33
Curve . . . . .	33
init . . . . .	33

process . . . . .	33
value . . . . .	34
<b>PER_I2C</b>	<b>36</b>
Types . . . . .	36
dsy_i2c_periph . . . . .	36
dsy_i2c_pin . . . . .	36
dsy_i2c_speed . . . . .	37
dsy_i2c_handle . . . . .	37
Functions . . . . .	37
dsy_i2c_init . . . . .	37
<b>per_tim</b>	<b>39</b>
init . . . . .	39
start . . . . .	39
tick . . . . .	39
get_tick . . . . .	39
delay_tick . . . . .	39
ms . . . . .	39
get_ms . . . . .	40
delay_ms . . . . .	40
us . . . . .	40
get_us . . . . .	40
delay_us . . . . .	40
<b>UART</b>	<b>41</b>
Description . . . . .	41
Credit . . . . .	41
TODO: . . . . .	41
Functions . . . . .	41
Init . . . . .	41
PollReceive . . . . .	41
StartRx . . . . .	41
PollTx . . . . .	42
PopRx . . . . .	42
Readable . . . . .	42
CheckError . . . . .	42
<b>RingBuffer</b>	<b>44</b>

Description . . . . .	44
Credit . . . . .	44

## **Core**

### **Memory Section Macros**

Macro for area of memory that is configured as cacheless This should be used primarily for DMA buffers, and the like.

THE DTCM RAM section is also non-cached. However, is not suitable for DMA transfers. Performance is on par with internal SRAM w/ cache enabled.

### **GPIO**

Enums and a simple struct for defining a hardware pin on the MCU These correlate with the stm32 datasheet, and are used to configure the hardware. 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 functions 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)
```

## Analog Control Reads

Starts the ADC conversions on the DMA.

without this the knobN.Process() functions will always return 0.0

```
void StartAdc()
```

## 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-

finer. 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](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-

lsl/daisy chained 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(); }
```

## **GateIn**

### **Description**

Generic Class for handling gate inputs through GPIO.

### **Files**

hid\_gatein.h

### **Credits**

**Author:** Stephen Hensley

**Date:** March 2020

### **General Functions**

#### **Init**

Initializes the gate input with specified hardware pin

```
void Init(dsy_gpio *gatepin);
```

#### **Trig**

Checks current state of gate input. Returns FALSE if pin is low, and TRUE if high

```
bool Trig();
```

## MidiHandler

### Description

Simple MIDI Handler

Parses bytes from an input into valid MidiEvents.

The MidiEvents fill a FIFO queue that the user can pop messages from.

### Credit

*author:* shensley *date added:* March 2020

### Data

#### MidiMessageType

Parsed from the Status Byte, these are the common Midi Messages that can be handled.

At this time only 3-byte messages are correctly parsed into MidiEvents.

```
enum MidiMessageType
{
    NoteOff,
    NoteOn,
    PolyphonicKeyPressure,
    ControlChange,
    ProgramChange,
    ChannelPressure,
    PitchBend,
    MessageLast, // maybe change name to MessageUnsupported
};
```

#### NoteOnEvent

Struct containing note, and velocity data for a given channel.

Can be made from MidiEvent

```
struct NoteOnEvent
{
```

```

    int      channel;
    uint8_t  note;
    uint8_t  velocity;
};

```

### **ControlChangeEvent**

Struct containing control number, and value for a given channel.

Can be made from MidiEvent

```

struct ControlChangeEvent
{
    int      channel;
    uint8_t  control_number;
    uint8_t  value;
};

```

### **MidiEvent**

Simple MidiEvent with message type, channel, and data[2] members. Newer ish.

### **AsNoteOn**

Returns the data within the MidiEvent as a NoteOnEvent struct.

```

NoteOnEvent  AsNoteOn()

```

### **AsNoteOn**

Returns the data within the MidiEvent as a NoteOnEvent struct.

```

ControlChangeEvent  AsControlChange()

```

## **MidiHandler**

### **Midi IO Modes**

Input and Output can be configured separately Multiple Input modes can be selected by OR'ing the values.



```

enum MidiInputMode
{
    INPUT_MODE_NONE      = 0x00,
    INPUT_MODE_UART1     = 0x01,
    INPUT_MODE_USB_INT   = 0x02,
    INPUT_MODE_USB_EXT   = 0x04,
};

enum MidiOutputMode
{
    OUTPUT_MODE_NONE     = 0x00,
    OUTPUT_MODE_UART1    = 0x01,
    OUTPUT_MODE_USB_INT  = 0x02,
    OUTPUT_MODE_USB_EXT  = 0x04,
};

```

## Functions

### Init

Initializes the MidiHandler

```
void Init(MidiInputMode in_mode, MidiOutputMode out_mode);
```

### StartReceive

Starts listening on the selected input mode(s). MidiEvent Queue will begin to fill, and can be checked with

### Parse

Feed in bytes to state machine from a queue.

Populates internal FIFO queue with MIDI Messages

For example with uart: midi.Parse(uart.PopRx());

```
void Parse(uint8_t byte);
```

### HasEvents

Checks if there are unhandled messages in the queue

```
bool HasEvents() const { return event_q_.readable(); }
```

### **PopEvent**

Pops the oldest unhandled MidiEvent from the internal queue

## 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
```

## hid\_usb

### Description

Interface for initializing and using the USB Peripherals on the daisy

### Credits

**Author:** Stephen Hensley

**Date Added:** December 2019

### Data Types

#### UsbPeriph

Specified which of the two USB Peripherals to initialize.

FS External D- pin is Pin 37 (GPIO31)

FS External D+ pin is Pin 38 (GPIO32)

```
FS_INTERNAL,  
FS_EXTERNAL,  
FS_BOTH,
```

### General Functions

#### Init

Initializes the specified peripheral(s) as USB CDC Devices

```
void Init(UsbPeriph dev);
```

#### TransmitInternal

Transmits a buffer of 'size' bytes from the on board USB FS port.

```
void TransmitInternal(uint8_t* buff, size_t size);
```

### **TransmitExternal**

Transmits a buffer of 'size' bytes from a USB port connected to the external USB Pins of the daisy seed.

```
void TransmitExternal(uint8_t* buff, size_t size);
```

TODO: - Add support for other USB classes (currently only CDC is supported) - Add support for Receiving data (currently it is handled and tested,

## Audio

### Description

Class for accessing the Audio Hardware.

Both the built-in, and any external audio can be accessed here.

The Built In Audio will use the SAI1 peripheral on the Daisy Seed, while external audio can be connected to the SAI2 Peripheral.

Previously, INT/EXT needed to be specified for everything, but now an object can be created separately for each if desired.

Several Samplerrates are available, but not all devices support all samplerates. The onboard WM8731 only supports up to 96kHz (but the firmware currently only supports up to 48kHz)

Static Buffers will be allocated in a cache-less section of RAM for DMA transfers.

### Credit

Written by Stephen Hensley

TODO Fix hard coding of these parameters

### SampleRate

Available Sample Rates, not all devices support all samplerates. Stops transmitting/receiving audio. Returns the real sample rate of the audio engine. Default Callbacks Stuff from the old, separate struct within .c 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



## 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**

Specifies 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);
```

Section Attributes Currently Sample Rates are not correctly 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);
```



# UART

## Description

Uart Peripheral

## Credit

Written by: shensley Date Added: March 2020

## TODO:

- Add flexible config for:
  - data size, stop bits, parity, baud, etc.
  - dma vs interrupt (or not).
- Error handling
- Transmit function improvements.
- Other UART Peripherals (currently only handles USART1 in UART mode.
- Overflow handling, etc. for Rx Queue.

## Functions

### Init

Initializes the UART Peripheral

```
void Init();
```

### PollReceive

Reads the amount of bytes in blocking mode with a 10ms timeout.

```
int PollReceive(uint8_t *buff, size_t size);
```

### StartRx

Starts a DMA Receive callback to fill a buffer of specified size.

Data is populated into a FIFO queue, and can be queried with the functions below. Maximum Buffer size is defined above.

If a value outside of the maximum is specified, the size will be set to the maximum.

```
int StartRx(size_t size);
```

### **PollTx**

Sends an amount of data in blocking mode.

```
int PollTx(uint8_t *buff, size_t size);
```

### **PopRx**

Pops the oldest byte from the FIFO.

```
uint8_t PopRx();
```

### **Readable**

Checks if there are any unread bytes in the FIFO

```
size_t Readable();
```

### **CheckError**

Returns the result of HAL\_UART\_GetError() to the user.

```
int CheckError();
```

Sets clock speeds, etc. externs of HAL handles... GPIO Map I2C MAp

## **RingBuffer**

### **Description**

Utility Ring Buffer

### **Credit**

imported from pichenettes/stmlib

Read enough samples to make it possible to read 1 sample.