G8R

Generated by Doxygen 1.9.1

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

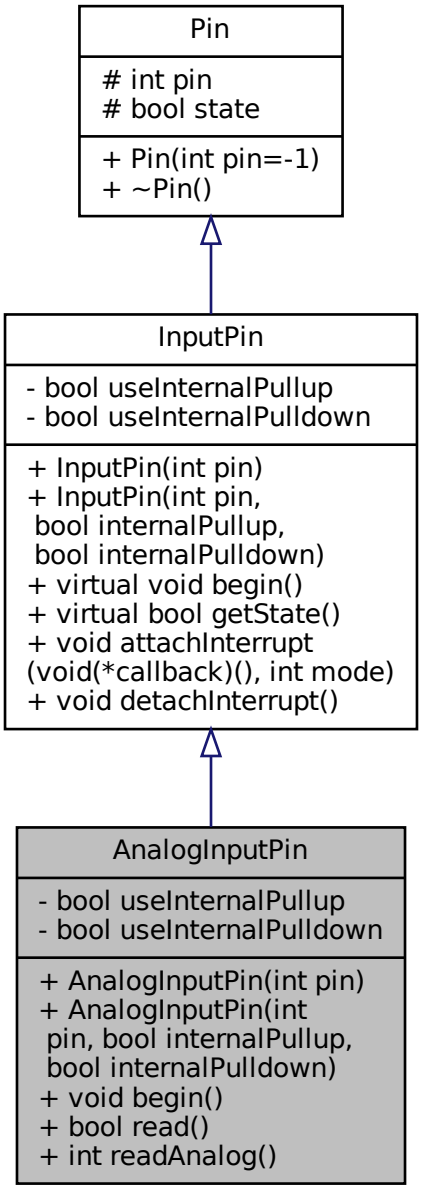
Class Documentation

4.1 AnalogInputPin Class Reference

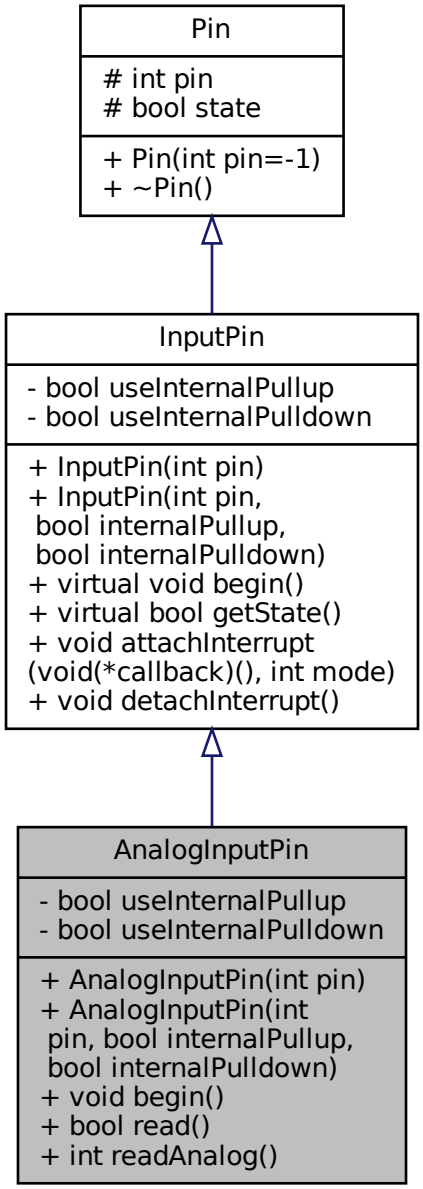
This class represents an analog input pin on the microcontroller.

`#include <Pin.h>`

Inheritance diagram for AnalogInputPin:



Collaboration diagram for AnalogInputPin:



Public Member Functions

- AnalogInputPin (int pin)

  Construct a new Analog Input Pin:: Analog Input Pin object.
- AnalogInputPin (int pin, bool internalPullup, bool internalPulldown)

  Construct a new Analog Input Pin:: Analog Input Pin object This constructor initializes the pin and sets the internal pullup and pulldown flags to the specified values. Use this constructor if you want to use the internal pullup or pulldown resistors.
- void begin ()

This function is used to initialize the analog input pin. It is intended to be called in the setup() function of the main sketch.

- bool read ()

  This function is used to read the digital value of the analog input pin.

- int readAnalog ()

  This function is used to read the value of the analog input pin.

Private Attributes

- bool useInternalPullup
- bool useInternalPulldown

Additional Inherited Members

4.1.1 Detailed Description

This class represents an analog input pin on the microcontroller.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 AnalogInputPin() [1/2]

```
AnalogInputPin::AnalogInputPin (
            int pin )
```

Construct a new Analog Input Pin:: Analog Input Pin object.

Parameters

pin	

4.1.2.2 AnalogInputPin() [2/2]

```
AnalogInputPin::AnalogInputPin (
            int pin,
            bool internalPullup,
            bool internalPulldown )
```

Construct a new Analog Input Pin:: Analog Input Pin object This constructor initializes the pin and sets the internal pullup and pulldown flags to the specified values. Use this constructor if you want to use the internal pullup or pulldown resistors.

4.1.3 Member Function Documentation

4.1.3.1 begin()

```
void AnalogInputPin::begin ( )  [virtual]
```

This function is used to initialize the analog input pin. It is intended to be called in the setup() function of the main sketch.

Reimplemented from InputPin.

4.1.3.2 read()

```
bool AnalogInputPin::read ( )
```

This function is used to read the digital value of the analog input pin.

Returns

bool

4.1.3.3 readAnalog()

```
int AnalogInputPin::readAnalog ( )
```

This function is used to read the value of the analog input pin.

Returns

int

4.1.4 Member Data Documentation

4.1.4.1 useInternalPulldown

```
bool AnalogInputPin::useInternalPulldown  [private]
```

4.1.4.2 useInternalPullup

```
bool AnalogInputPin::useInternalPullup  [private]
```

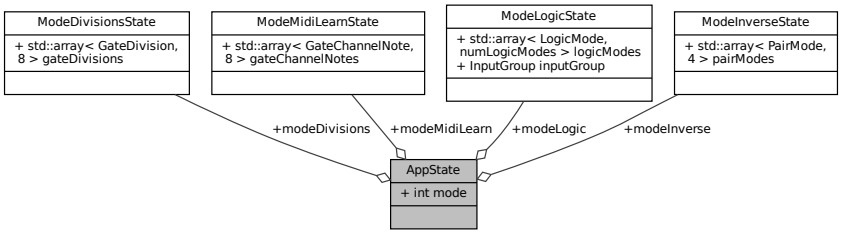
The documentation for this class was generated from the following files:

- include/Pin.h
- src/Pin.cpp

4.2 AppState Struct Reference

```
#include <AppState.h>
```

Collaboration diagram for AppState:



Public Attributes

- int mode
- ModeDivisionsState modeDivisions
- ModeMidiLearnState modeMidiLearn
- ModeInverseState modeInverse
- ModeLogicState modeLogic

4.2.1 Member Data Documentation

4.2.1.1 mode

```
int AppState::mode
```

4.2.1.2 modeDivisions

```
ModeDivisionsState AppState::modeDivisions
```

4.2.1.3 modeInverse

```
ModeInverseState AppState::modeInverse
```

4.2.1.4 modeLogic

```
ModeLogicState AppState::modeLogic
```

4.2.1.5 modeMidiLearn

```
ModeMidiLearnState AppState::modeMidiLearn
```

The documentation for this struct was generated from the following file:

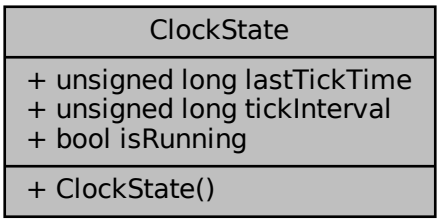
- include/AppState.h

4.3 ClockState Struct Reference

The ClockState struct is used to store the current state of the clock.

```
#include <EurorackClock.h>
```

Collaboration diagram for ClockState:



Public Member Functions

- ClockState ()

Public Attributes

- unsigned long lastTickTime
- unsigned long tickInterval
- bool isRunning

4.3.1 Detailed Description

The ClockState struct is used to store the current state of the clock.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 ClockState()

```
ClockState::ClockState ( )  [inline]
```

4.3.3 Member Data Documentation

4.3.3.1 isRunning

```
bool ClockState::isRunning
```

4.3.3.2 lastTickTime

```
unsigned long ClockState::lastTickTime
```

4.3.3.3 tickInterval

```
unsigned long ClockState::tickInterval
```

The documentation for this struct was generated from the following file:

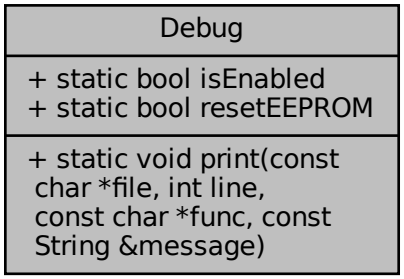
- include/EurorackClock.h

4.4 Debug Class Reference

Helper class for debugging. This makes it easier to enable/disable debug messages. It is used in conjunction with the DEBUG_PRINT macro. Which adds the file, line, and function name to the debug message so that the developer can easily find where the message is coming from.

`#include <Debug.h>`

Collaboration diagram for Debug:



Static Public Member Functions

- static void print (const char ∗file, int line, const char ∗func, const String &message)

Static Public Attributes

- static bool isEnabled = false
- static bool resetEEPROM = false

4.4.1 Detailed Description

Helper class for debugging. This makes it easier to enable/disable debug messages. It is used in conjunction with the DEBUG_PRINT macro. Which adds the file, line, and function name to the debug message so that the developer can easily find where the message is coming from.

4.4.2 Member Function Documentation

4.4.2.1 print()

```
void Debug::print (
            const char * file,
            int line,
            const char * func,
            const String & message )  [static]
```

4.4.3 Member Data Documentation

4.4.3.1 isEnabled

```
bool Debug::isEnabled = false  [static]
```

4.4.3.2 resetEEPROM

```
bool Debug::resetEEPROM = false  [static]
```

The documentation for this class was generated from the following files:

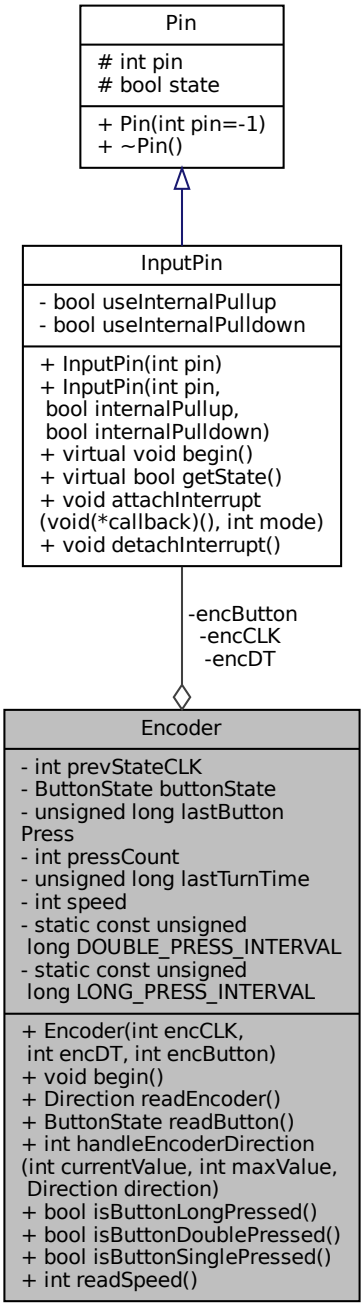
- include/Debug.h
- src/Debug.cpp

4.5 Encoder Class Reference

This class is used to read the encoder and button inputs.

```
#include <Encoder.h>
```

Collaboration diagram for Encoder:



Public Types

- enum Direction { NONE , CW , CCW }
- enum ButtonState { OPEN , PRESSED }

Public Member Functions

- Encoder (int encCLK, int encDT, int encButton)

- void begin ()

  This is intended to be called in the setup() function of the main sketch.
- Direction readEncoder ()

  Read the encoder and return the direction of the turn. It uses some constants to determine the speed of the turn.
- ButtonState readButton ()

  This is used to read the state of the button. It also handles debouncing and double-click detection.
- int handleEncoderDirection (int currentValue, int maxValue, Direction direction)

  This is a helper function to handle the encoder direction.
- bool isButtonLongPressed ()

  Check if the button has been long pressed.
- bool isButtonDoublePressed ()

  Check if the button has been double pressed.
- bool isButtonSinglePressed ()

  Check if the button has been single pressed.
- int readSpeed ()

  Read the speed of the encoder turn.

Private Attributes

- InputPin encCLK
- InputPin encDT
- InputPin encButton
- int prevStateCLK
- ButtonState buttonState
- unsigned long lastButtonPress
- int pressCount
- unsigned long lastTurnTime
- int speed

Static Private Attributes

- static const unsigned long DOUBLE_PRESS_INTERVAL = 500
- static const unsigned long LONG_PRESS_INTERVAL = 1000

4.5.1 Detailed Description

This class is used to read the encoder and button inputs.

4.5.2 Member Enumeration Documentation

4.5.2.1 ButtonState

`enum Encoder::ButtonState`

Enumerator

OPEN	
PRESSED	

4.5.2.2 Direction

```
enum Encoder::Direction
```

Enumerator

NONE	
CW	
CCW	

4.5.3 Constructor & Destructor Documentation

4.5.3.1 Encoder()

```
Encoder::Encoder (
            int encCLK,
            int encDT,
            int encButton )
```

4.5.4 Member Function Documentation

4.5.4.1 begin()

```
void Encoder::begin ( )
```

This is intended to be called in the setup() function of the main sketch.

4.5.4.2 handleEncoderDirection()

```
int Encoder::handleEncoderDirection (
            int currentValue,
            int maxValue,
            Direction direction )
```

This is a helper function to handle the encoder direction.

Parameters

currentValue	
maxValue	
direction	

Returns

int

4.5.4.3 isButtonDoublePressed()

```
bool Encoder::isButtonDoublePressed ( )
```

Check if the button has been double pressed.

Returns

true

false

4.5.4.4 isButtonLongPressed()

```
bool Encoder::isButtonLongPressed ( )
```

Check if the button has been long pressed.

Returns

true

false

4.5.4.5 isButtonSinglePressed()

```
bool Encoder::isButtonSinglePressed ( )
```

Check if the button has been single pressed.

Returns

true

false

4.5.4.6 readButton()

```
Encoder::ButtonState Encoder::readButton ( )
```

This is used to read the state of the button. It also handles debouncing and double-click detection.

Returns

Encoder::ButtonState

4.5.4.7 readEncoder()

```
Encoder::Direction Encoder::readEncoder ( )
```

Read the encoder and return the direction of the turn. It uses some constants to determine the speed of the turn.

TODO: I should probably move the constants to the constructor and make them configurable.

Returns

Encoder::Direction

4.5.4.8 readSpeed()

```
int Encoder::readSpeed ( )
```

Read the speed of the encoder turn.

Returns

int

4.5.5 Member Data Documentation

4.5.5.1 buttonState

```
ButtonState Encoder::buttonState  [private]
```

4.5.5.2 DOUBLE_PRESS_INTERVAL

```
const unsigned long Encoder::DOUBLE_PRESS_INTERVAL = 500  [static], [private]
```

4.5.5.3 encButton

```
InputPin Encoder::encButton  [private]
```

4.5.5.4 encCLK

```
InputPin Encoder::encCLK  [private]
```

4.5.5.5 encDT

```
InputPin Encoder::encDT  [private]
```

4.5.5.6 lastButtonPress

```
unsigned long Encoder::lastButtonPress  [private]
```

4.5.5.7 lastTurnTime

```
unsigned long Encoder::lastTurnTime  [private]
```

4.5.5.8 LONG_PRESS_INTERVAL

```
const unsigned long Encoder::LONG_PRESS_INTERVAL = 1000  [static], [private]
```

4.5.5.9 pressCount

```
int Encoder::pressCount  [private]
```

4.5.5.10 prevStateCLK

```
int Encoder::prevStateCLK  [private]
```

4.5.5.11 speed

```
int Encoder::speed  [private]
```

The documentation for this class was generated from the following files:

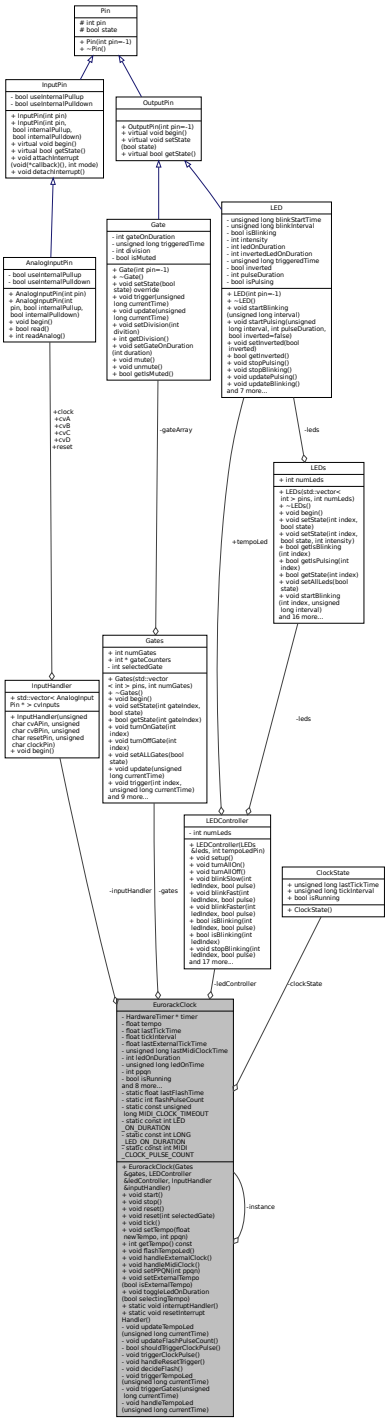
- include/Encoder.h
- src/Encoder.cpp

4.6 EurorackClock Class Reference

The EurorackClock class is used to handle the clock and tempo of the device. It utilizes an interrupt to handle the clock ticks, and can be set to an external tempo.

```
#include <EurorackClock.h>
```

Collaboration diagram for EurorackClock:



Public Member Functions

- EurorackClock (Gates &gates, LEDController &ledController, InputHandler &inputHandler)

  Constructor.
- void start ()

  This function is used to start the clock.
- void stop ()

This function is used to stop the clock.

- void reset ()

  Reset the clock and gates.

- void reset (int selectedGate)

  Reset the selected gate. Useful for syncing the gates.

- void tick ()

  This is the main tick() function, which is used to update the clock and its components. Don't confuse this with the pulse() function, which is used to trigger a clock pulse.

- void setTempo (float newTempo, int ppqn)

  This function is used to set the tempo of the clock.

- int getTempo () const

  This function is used to toggle the clock.

- void flashTempoLed ()
- void handleExternalClock ()

  This function is used to handle the external clock.

- void handleMidiClock ()

  This function is used to handle the MIDI clock.

- void setPPQN (int ppqn)

  Update the local ppqn value.

- void setExternalTempo (bool isExternalTempo)

  sets the external tempo mode when the external clock is used.

- void toggleLedOnDuration (bool selectingTempo)

  This function is used to toggle how long the LED flashes for.

Static Public Member Functions

- static void interruptHandler ()
- static void resetInterruptHandler ()

Private Member Functions

- void updateTempoLed (unsigned long currentTime)

  This function is used to flash the tempo LED at the correct interval.

- void updateFlashPulseCount ()

  This function determines if we need to send a trigger or reset them.

- bool shouldTriggerClockPulse ()

  Evaluate if a clock pulse should be triggered.

- void triggerClockPulse ()

  This triggers on clock pulses. Not to be confused with the tick() function.

- void handleResetTrigger ()

  Handle the trigger signal from a reset pin.

- void decideFlash ()

  Check if it's time to flash the LED.

- void triggerTempoLed (unsigned long currentTime)

  Responsible for flashing the tempo LED.

- void triggerGates (unsigned long currentTime)

  tigger the gates and LEDs.

- void handleTempoLed (unsigned long currentTime)

  Used to determine if the tempo LED should be flashed according to the clock pulse and defined PPQN.

Private Attributes

- ClockState clockState
- HardwareTimer * timer
- Gates & gates
- LEDController & ledController
- InputHandler & inputHandler
- float tempo
- float lastTickTime
- float tickInterval
- float lastExternalTickTime
- unsigned long lastMidiClockTime
- int ledOnDuration = LONG_LED_ON_DURATION
- unsigned long ledOnTime = 0
- int ppqn
- bool isRunning
- bool isExternalTempo
- bool isMidiClock
- bool timeToFlash
- bool resetTriggered
- float externalTempo
- int lastClockState
- unsigned long lastClockTime
- int tickCount

Static Private Attributes

- static EurorackClock * instance = nullptr
- static float lastFlashTime = 0

  Static variables initialization.
- static int flashPulseCount = 0
- static const unsigned long MIDI_CLOCK_TIMEOUT = 1000
- static const int LED_ON_DURATION = 10
- static const int LONG_LED_ON_DURATION = 50
- static const int MIDI_CLOCK_PULSE_COUNT = 24

4.6.1 Detailed Description

The EurorackClock class is used to handle the clock and tempo of the device. It utilizes an interrupt to handle the clock ticks, and can be set to an external tempo.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 EurorackClock()

```
EurorackClock::EurorackClock (
            Gates & gates,
            LEDController & ledController,
            InputHandler & inputHandler )
```

Constructor.

4.6.3 Member Function Documentation

4.6.3.1 decideFlash()

```
void EurorackClock::decideFlash ( )  [private]
```

Check if it's time to flash the LED.

4.6.3.2 flashTempoLed()

```
void EurorackClock::flashTempoLed ( )
```

4.6.3.3 getTempo()

```
int EurorackClock::getTempo ( ) const
```

This function is used to toggle the clock.

4.6.3.4 handleExternalClock()

```
void EurorackClock::handleExternalClock ( )
```

This function is used to handle the external clock.

4.6.3.5 handleMidiClock()

```
void EurorackClock::handleMidiClock ( )
```

This function is used to handle the MIDI clock.

4.6.3.6 handleResetTrigger()

```
void EurorackClock::handleResetTrigger ( )  [private]
```

Handle the trigger signal from a reset pin.

4.6.3.7 handleTempoLed()

```
void EurorackClock::handleTempoLed (
            unsigned long currentTime )  [private]
```

Used to determine if the tempo LED should be flashed according to the clock pulse and defined PPQN.

4.6.3.8 interruptHandler()

```
static void EurorackClock::interruptHandler ( )  [inline], [static]
```

4.6.3.9 reset() [1/2]

```
void EurorackClock::reset ( )
```

Reset the clock and gates.

4.6.3.10 reset() [2/2]

```
void EurorackClock::reset (
            int selectedGate )
```

Reset the selected gate. Useful for syncing the gates.

Parameters

selectedGate	

4.6.3.11 resetInterruptHandler()

```
static void EurorackClock::resetInterruptHandler ( )  [inline], [static]
```

4.6.3.12 setExternalTempo()

```
void EurorackClock::setExternalTempo (
            bool isExternalTempo )
```

sets the external tempo mode when the external clock is used.

Parameters

isExternalTempo	

4.6.3.13 setPPQN()

```
void EurorackClock::setPPQN (
            int ppqn )
```

Update the local ppqn value.

4.6.3.14 setTempo()

```
void EurorackClock::setTempo (
            float newTempo,
            int ppqn )
```

This function is used to set the tempo of the clock.

Parameters

newTempo	The new tempo to set.
ppqn	The PPQN (Pulses Per Quarter Note) to set.

4.6.3.15 shouldTriggerClockPulse()

```
bool EurorackClock::shouldTriggerClockPulse ( )  [private]
```

Evaluate if a clock pulse should be triggered.

Returns

bool shouldTrigger

4.6.3.16 start()

```
void EurorackClock::start ( )
```

This function is used to start the clock.

4.6.3.17 stop()

```
void EurorackClock::stop ( )
```

This function is used to stop the clock.

4.6.3.18 tick()

```
void EurorackClock::tick ( )
```

This is the main tick() function, which is used to update the clock and its components. Don't confuse this with the pulse() function, which is used to trigger a clock pulse.

4.6.3.19 toggleLedOnDuration()

```
void EurorackClock::toggleLedOnDuration (
            bool selectingTempo )
```

This function is used to toggle how long the LED flashes for.

4.6.3.20 triggerClockPulse()

```
void EurorackClock::triggerClockPulse ( )  [private]
```

This triggers on clock pulses. Not to be confused with the tick() function.

4.6.3.21 triggerGates()

```
void EurorackClock::triggerGates (
            unsigned long currentTime )  [private]
```

tigger the gates and LEDs.

Parameters

currentTime	

4.6.3.22 triggerTempoLed()

```
void EurorackClock::triggerTempoLed (
            unsigned long currentTime )  [private]
```

Responsible for flashing the tempo LED.

4.6.3.23 updateFlashPulseCount()

```
void EurorackClock::updateFlashPulseCount ( )  [private]
```

This function determines if we need to send a trigger or reset them.

4.6.3.24 updateTempoLed()

```
void EurorackClock::updateTempoLed (
            unsigned long currentTime )  [private]
```

This function is used to flash the tempo LED at the correct interval.

4.6.4 Member Data Documentation

4.6.4.1 clockState

```
ClockState EurorackClock::clockState  [private]
```

4.6.4.2 externalTempo

```
float EurorackClock::externalTempo  [private]
```

4.6.4.3 flashPulseCount

```
int EurorackClock::flashPulseCount = 0  [static], [private]
```

4.6.4.4 gates

```
Gates& EurorackClock::gates  [private]
```

4.6.4.5 inputHandler

```
InputHandler& EurorackClock::inputHandler  [private]
```

4.6.4.6 instance

```
EurorackClock * EurorackClock::instance = nullptr  [static], [private]
```

4.6.4.7 isExternalTempo

```
bool EurorackClock::isExternalTempo  [private]
```

4.6.4.8 isMidiClock

```
bool EurorackClock::isMidiClock  [private]
```

4.6.4.9 isRunning

```
bool EurorackClock::isRunning  [private]
```

4.6.4.10 lastClockState

```
int EurorackClock::lastClockState  [private]
```

4.6.4.11 lastClockTime

```
unsigned long EurorackClock::lastClockTime  [private]
```

4.6.4.12 lastExternalTickTime

```
float EurorackClock::lastExternalTickTime  [private]
```

4.6.4.13 lastFlashTime

```
float EurorackClock::lastFlashTime = 0  [static], [private]
```

Static variables initialization.

4.6.4.14 lastMidiClockTime

```
unsigned long EurorackClock::lastMidiClockTime  [private]
```

4.6.4.15 lastTickTime

```
float EurorackClock::lastTickTime  [private]
```

4.6.4.16 LED_ON_DURATION

```
const int EurorackClock::LED_ON_DURATION = 10  [static], [private]
```

4.6.4.17 ledController

```
LEDController& EurorackClock::ledController  [private]
```

4.6.4.18 ledOnDuration

```
int EurorackClock::ledOnDuration = LONG_LED_ON_DURATION  [private]
```

4.6.4.19 ledOnTime

```
unsigned long EurorackClock::ledOnTime = 0  [private]
```

4.6.4.20 LONG_LED_ON_DURATION

```
const int EurorackClock::LONG_LED_ON_DURATION = 50  [static], [private]
```

4.6.4.21 MIDI_CLOCK_PULSE_COUNT

```
const int EurorackClock::MIDI_CLOCK_PULSE_COUNT = 24  [static], [private]
```

4.6.4.22 MIDI_CLOCK_TIMEOUT

```
const unsigned long EurorackClock::MIDI_CLOCK_TIMEOUT = 1000  [static], [private]
```

4.6.4.23 ppqn

```
int EurorackClock::ppqn  [private]
```

4.6.4.24 resetTriggered

```
bool EurorackClock::resetTriggered  [private]
```

4.6.4.25 tempo

```
float EurorackClock::tempo  [private]
```

4.6.4.26 tickCount

```
int EurorackClock::tickCount  [private]
```

4.6.4.27 tickInterval

```
float EurorackClock::tickInterval  [private]
```

4.6.4.28 timer

```
HardwareTimer* EurorackClock::timer  [private]
```

4.6.4.29 timeToFlash

```
bool EurorackClock::timeToFlash  [private]
```

The documentation for this class was generated from the following files:

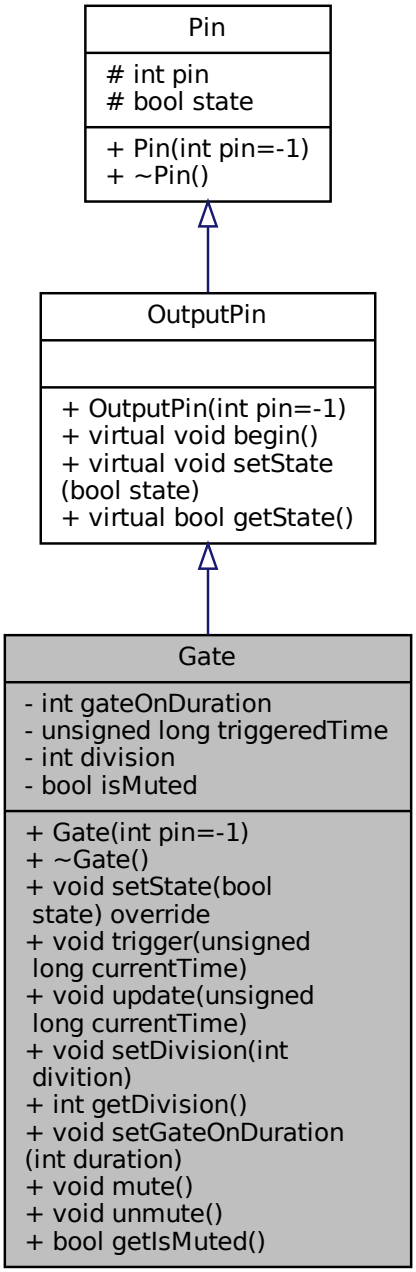
- include/EurorackClock.h
- src/EurorackClock.cpp

4.7 Gate Class Reference

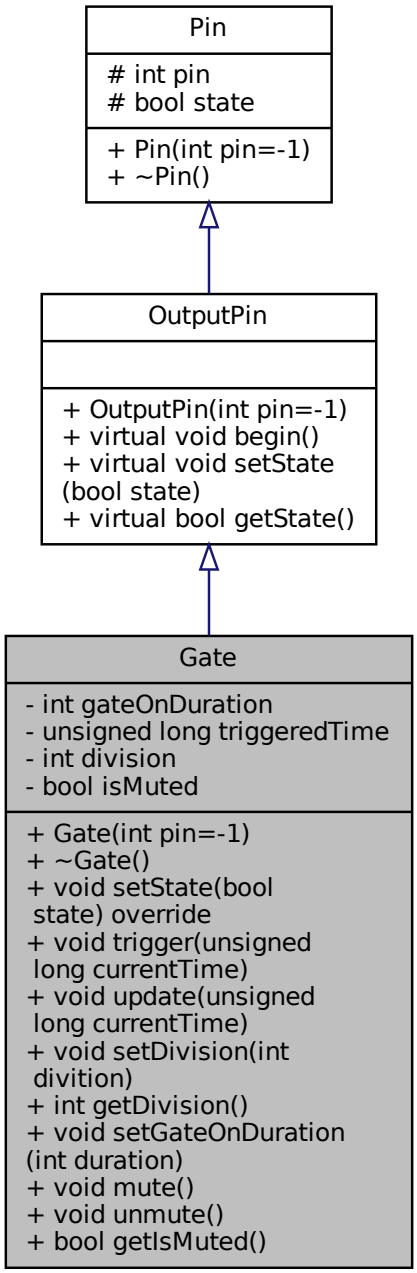
This class defines what a gate is and how it should behave. It inherits from the OutputPin class, which provides the basic functionality for a pin including setting state to HIGH or LOW, getting the current state, etc.

```
#include <Gate.h>
```

Inheritance diagram for Gate:



Collaboration diagram for Gate:



Public Member Functions

- Gate (int pin=-1)

  Constructor.
- ~Gate ()

  Destructor.
- void setState (bool state) override

This function is used to set the state of the output pin. Possible states are HIGH or LOW.

- void trigger (unsigned long currentTime)

  This function is used to execute a trigger signal. It sets the state of the gate to HIGH and records the time of the trigger. Then, the gate will automatically turn off after the gateOnDuration has passed.

- void update (unsigned long currentTime)

  This function is used to update the state of the gate. If the gate is currently HIGH and the gateOnDuration has passed, the gate will be turned off. It is meant to be called in every loop iteration.

- void setDivision (int divition)

  This function is used to set the division of the gate.

- int getDivision ()

  Returns the division configured for the gate.

- void setGateOnDuration (int duration)

  This function is used to set the duration of the gate being on.

- void mute ()

  This function is used to mute the gate.

- void unmute ()

  This function is used to unmute the gate.

- bool getIsMuted ()

  This function is used to check if the gate is currently muted.

Private Attributes

- int gateOnDuration = 10
- unsigned long triggeredTime = 0
- int division = internalPPQN
- bool isMuted = false

Additional Inherited Members

4.7.1 Detailed Description

This class defines what a gate is and how it should behave. It inherits from the OutputPin class, which provides the basic functionality for a pin including setting state to HIGH or LOW, getting the current state, etc.

4.7.2 Constructor & Destructor Documentation

4.7.2.1 Gate()

```
Gate::Gate (
            int pin = -1 )
```

Constructor.

4.7.2.2 ~Gate()

```
Gate::~Gate ( )
```

Destructor.

4.7.3 Member Function Documentation

4.7.3.1 getDivision()

```
int Gate::getDivision ( )
```

Returns the division configured for the gate.

Returns

int

4.7.3.2 getIsMuted()

```
bool Gate::getIsMuted ( )
```

This function is used to check if the gate is currently muted.

Returns

true
false

4.7.3.3 mute()

```
void Gate::mute ( )
```

This function is used to mute the gate.

4.7.3.4 setDivision()

```
void Gate::setDivision (
            int newDivision )
```

This function is used to set the division of the gate.

Parameters

newDivision	

4.7.3.5 setGateOnDuration()

```
void Gate::setGateOnDuration (
            int duration )
```

This function is used to set the duration of the gate being on.

Parameters

duration	

4.7.3.6 setState()

```
void Gate::setState (
            bool state )  [override], [virtual]
```

This function is used to set the state of the output pin. Possible states are HIGH or LOW.

Parameters

newState	

Reimplemented from OutputPin.

4.7.3.7 trigger()

```
void Gate::trigger (
            unsigned long currentTime )
```

This function is used to execute a trigger signal. It sets the state of the gate to HIGH and records the time of the trigger. Then, the gate will automatically turn off after the gateOnDuration has passed.

Parameters

currentTime	

4.7.3.8 unmute()

```
void Gate::unmute ( )
```

This function is used to unmute the gate.

4.7.3.9 update()

```
void Gate::update (
            unsigned long currentTime )
```

This function is used to update the state of the gate. If the gate is currently HIGH and the gateOnDuration has passed, the gate will be turned off. It is meant to be called in every loop iteration.

Parameters

currentTime	

4.7.4 Member Data Documentation

4.7.4.1 division

```
int Gate::division = internalPPQN  [private]
```

4.7.4.2 gateOnDuration

```
int Gate::gateOnDuration = 10  [private]
```

4.7.4.3 isMuted

```
bool Gate::isMuted = false  [private]
```

4.7.4.4 triggeredTime

```
unsigned long Gate::triggeredTime = 0  [private]
```

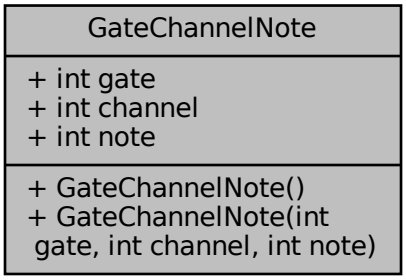
The documentation for this class was generated from the following files:

- include/Gate.h
- src/Gate.cpp

4.8 GateChannelNote Struct Reference

```
#include <AppState.h>
```

Collaboration diagram for GateChannelNote:



Public Member Functions

- GateChannelNote ()
- GateChannelNote (int gate, int channel, int note)

Public Attributes

- int gate
- int channel
- int note

4.8.1 Constructor & Destructor Documentation

4.8.1.1 GateChannelNote() [1/2]

```
GateChannelNote::GateChannelNote ( )  [inline]
```

4.8.1.2 GateChannelNote() [2/2]

```
GateChannelNote::GateChannelNote (
            int gate,
            int channel,
            int note )  [inline]
```

4.8.2 Member Data Documentation

4.8.2.1 channel

```
int GateChannelNote::channel
```

4.8.2.2 gate

```
int GateChannelNote::gate
```

4.8.2.3 note

```
int GateChannelNote::note
```

The documentation for this struct was generated from the following file:

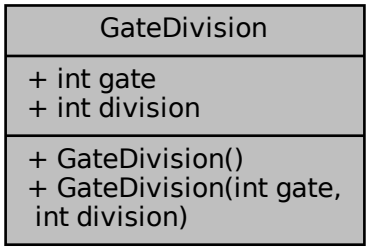
- include/AppState.h

4.9 GateDivision Struct Reference

This is a global struct that holds the state of the application. It mainly holds items that need to persist after a power cycle. The object is initialized managed by the StateManager class.

`#include <AppState.h>`

Collaboration diagram for GateDivision:



Public Member Functions

- GateDivision ()
- GateDivision (int gate, int division)

Public Attributes

- int gate
- int division

4.9.1 Detailed Description

This is a global struct that holds the state of the application. It mainly holds items that need to persist after a power cycle. The object is initialized managed by the StateManager class.

The fefault values are set in the StateManager class, in the initializeEEPROM() function. To avoid issues with the EEPROM memory, make sure you initialize all values in the StateManager class.

This object is updated through out the app, however saving to EEPROM is only done when the app is in mode selection mode. It saves when long pressing and also when the mode is successfully changed.

4.9.2 Constructor & Destructor Documentation

4.9.2.1 GateDivision() [1/2]

```
GateDivision::GateDivision ( )  [inline]
```

4.9.2.2 GateDivision() [2/2]

```
GateDivision::GateDivision (
            int gate,
            int division )  [inline]
```

4.9.3 Member Data Documentation

4.9.3.1 division

```
int GateDivision::division
```

4.9.3.2 gate

```
int GateDivision::gate
```

The documentation for this struct was generated from the following file:

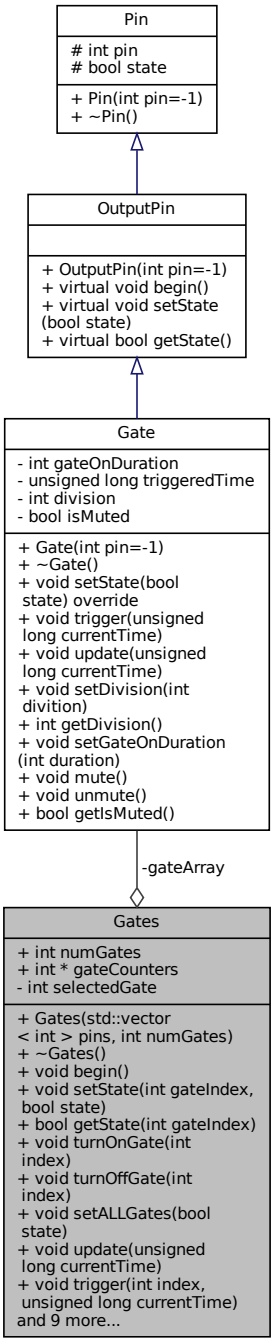
- include/AppState.h

4.10 Gates Class Reference

This is a collection of gates and thus the main thing we are working with in this project. Very rarely will you need to interact with the Gate class directly, as most of the functionality is handled by the Gates class.

```
#include <Gates.h>
```

Collaboration diagram for Gates:



Public Member Functions

- Gates (std::vector< int > pins, int numGates)

  Construct a new Gates:: Gates object.
- ~Gates ()

  Destroy the Gates:: Gates object.
- void begin ()

This is intended to be called in the setup() function of the main sketch. It mainly initializes the pins for the gates as outputs.

- void setState (int gateIndex, bool state)

  This function is used to set the state of a specific gate. State is either HIGH or LOW.
- bool getState (int gateIndex)

  This function is used to get the state of a specific gate.
- void turnOnGate (int index)

  This function is used to turn on a specific gate.
- void turnOffGate (int index)

  This function is used to turn off a specific gate.
- void setALLGates (bool state)

  This function is sets the satet of all gates.
- void update (unsigned long currentTime)

  This is used to update the state of the gates. It is meant to be called in every loop iteration. It is needed in order to evaluate if the gate should be turned off.
- void trigger (int index, unsigned long currentTime)

  This function is used to start a trigger signal on a specific gate. The gate will automatically turn off after the gate↩ OnDuration has passed. Which is why the currentTime is needed.
- void setDivision (int index, int division)

  This method is used to se the division of a specific gate.
- int getDivision (int index)

  This function returns the division of a specific gate.
- void setSelectedGate (int gate)

  This is a helper function used when working with a specific gate.
- int getSelectedGate ()

  returns the selected gate.
- void setGateOnDuration (int index, int duration)

  Sets the duration of the gate being on when sending trigger signals.
- void mute (int index)

  This function is used to mute a specific gate.
- void unmute (int index)

  This function is used to unmute a specific gate.
- bool isMuted (int index)

  This function is used to check if a specific gate is muted.
- void unMuteAll ()

  This function is used to unmute all gates.

Public Attributes

- int numGates
- int * gateCounters

Private Attributes

- Gate * gateArray
- int selectedGate

4.10.1 Detailed Description

This is a collection of gates and thus the main thing we are working with in this project. Very rarely will you need to interact with the Gate class directly, as most of the functionality is handled by the Gates class.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 Gates()

```
Gates::Gates (
            std::vector< int > pins,
            int numGates )
```

Construct a new Gates:: Gates object.

Parameters

pins	
numGates	

4.10.2.2 ~Gates()

```
Gates::~Gates ( )
```

Destroy the Gates:: Gates object.

4.10.3 Member Function Documentation

4.10.3.1 begin()

```
void Gates::begin ( )
```

This is intended to be called in the setup() function of the main sketch. It mainly initializes the pins for the gates as outputs.

4.10.3.2 getDivision()

```
int Gates::getDivision (
            int index )
```

This function returns the division of a specific gate.

Parameters

index	

Returns

int

4.10.3.3 getSelectedGate()

```
int Gates::getSelectedGate ( )
```

returns the selected gate.

Returns

int

4.10.3.4 getState()

```
bool Gates::getState (
            int gateIndex )
```

This function is used to get the state of a specific gate.

Parameters

gateIndex	

Returns

state of the gate as a boolean

4.10.3.5 isMuted()

```
bool Gates::isMuted (
            int index )
```

This function is used to check if a specific gate is muted.

Parameters

index	

Returns

true

4.10.3.6 mute()

```
void Gates::mute (
            int index )
```

This function is used to mute a specific gate.

Parameters

index	

4.10.3.7 setALLGates()

```
void Gates::setALLGates (
            bool state )
```

This function is sets the satet of all gates.

Parameters

state	

4.10.3.8 setDivision()

```
void Gates::setDivision (
            int index,
            int division )
```

This method is used to se the division of a specific gate.

Parameters

index	
division	

4.10.3.9 setGateOnDuration()

```
void Gates::setGateOnDuration (
            int index,
            int duration )
```

Sets the duration of the gate being on when sending trigger signals.

Parameters

index	
duration	

4.10.3.10 setSelectedGate()

```
void Gates::setSelectedGate (
            int index )
```

This is a helper function used when working with a specific gate.

Parameters

index	

4.10.3.11 setState()

```
void Gates::setState (
            int gateIndex,
            bool state )
```

This function is used to set the state of a specific gate. State is either HIGH or LOW.

Parameters

gateIndex	
state	

4.10.3.12 trigger()

```
void Gates::trigger (
```

```
            int index,
            unsigned long currentTime )
```

This function is used to start a trigger signal on a specific gate. The gate will automatically turn off after the gate↩ OnDuration has passed. Which is why the currentTime is needed.

Parameters

index	
currentTime	

4.10.3.13 turnOffGate()

```
void Gates::turnOffGate (
            int index )
```

This function is used to turn off a specific gate.

Parameters

index	

4.10.3.14 turnOnGate()

```
void Gates::turnOnGate (
            int index )
```

This function is used to turn on a specific gate.

Parameters

index	

4.10.3.15 unmute()

```
void Gates::unmute (
            int index )
```

This function is used to unmute a specific gate.

Parameters

index	

4.10.3.16 unMuteAll()

```
void Gates::unMuteAll ( )
```

This function is used to unmute all gates.

Returns

true

4.10.3.17 update()

```
void Gates::update (
            unsigned long currentTime )
```

This is used to update the state of the gates. It is meant to be called in every loop iteration. It is needed in order to evaluate if the gate should be turned off.

Parameters

currentTime	

4.10.4 Member Data Documentation

4.10.4.1 gateArray

```
Gate* Gates::gateArray  [private]
```

4.10.4.2 gateCounters

```
int* Gates::gateCounters
```

4.10.4.3 numGates

```
int Gates::numGates
```

4.10.4.4 selectedGate

```
int Gates::selectedGate  [private]
```

The documentation for this class was generated from the following files:

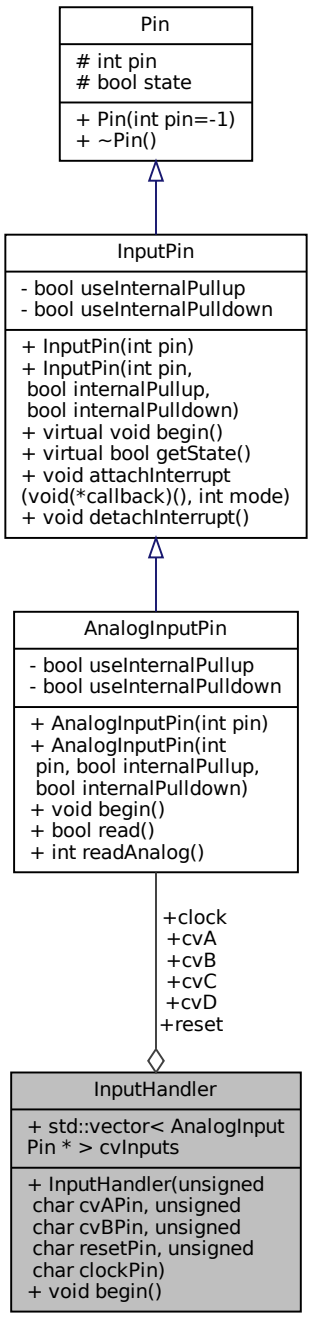
- include/Gates.h
- src/Gates.cpp

4.11 InputHandler Class Reference

This class is used to read the CV inputs. It is a simple class that uses the AnalogInputPin class to read the CV inputs. Alias the reset and clock inputs to cvC and cvD respectively. cvC is the reset input and cvD is the clock input.

```
#include <InputHandler.h>
```

Collaboration diagram for InputHandler:



Public Member Functions

- InputHandler (unsigned char cvAPin, unsigned char cvBPin, unsigned char resetPin, unsigned char clockPin)

  Construct a new Input Handler:: Input Handler object.
- void begin ()

  This is intended to be called in the setup() function of the main sketch. It initializes the CV inputs.

Public Attributes

- AnalogInputPin cvA
- AnalogInputPin cvB
- AnalogInputPin cvC
- AnalogInputPin cvD
- AnalogInputPin & reset = cvC
- AnalogInputPin & clock = cvD
- std::vector< AnalogInputPin * > cvInputs

4.11.1 Detailed Description

This class is used to read the CV inputs. It is a simple class that uses the AnalogInputPin class to read the CV inputs. Alias the reset and clock inputs to cvC and cvD respectively. cvC is the reset input and cvD is the clock input.

4.11.2 Constructor & Destructor Documentation

4.11.2.1 InputHandler()

```
InputHandler::InputHandler (
            unsigned char cvAPin,
            unsigned char cvBPin,
            unsigned char resetPin,
            unsigned char clockPin )
```

Construct a new Input Handler:: Input Handler object.

Parameters

cvAPin	
cvBPin	

4.11.3 Member Function Documentation

4.11.3.1 begin()

```
void InputHandler::begin ( )
```

This is intended to be called in the setup() function of the main sketch. It initializes the CV inputs.

4.11.4 Member Data Documentation

4.11.4.1 clock

```
AnalogInputPin& InputHandler::clock = cvD
```

4.11.4.2 cvA

```
AnalogInputPin InputHandler::cvA
```

4.11.4.3 cvB

```
AnalogInputPin InputHandler::cvB
```

4.11.4.4 cvC

```
AnalogInputPin InputHandler::cvC
```

4.11.4.5 cvD

```
AnalogInputPin InputHandler::cvD
```

4.11.4.6 cvInputs

```
std::vector<AnalogInputPin*> InputHandler::cvInputs
```

4.11.4.7 reset

```
AnalogInputPin& InputHandler::reset = cvC
```

The documentation for this class was generated from the following files:

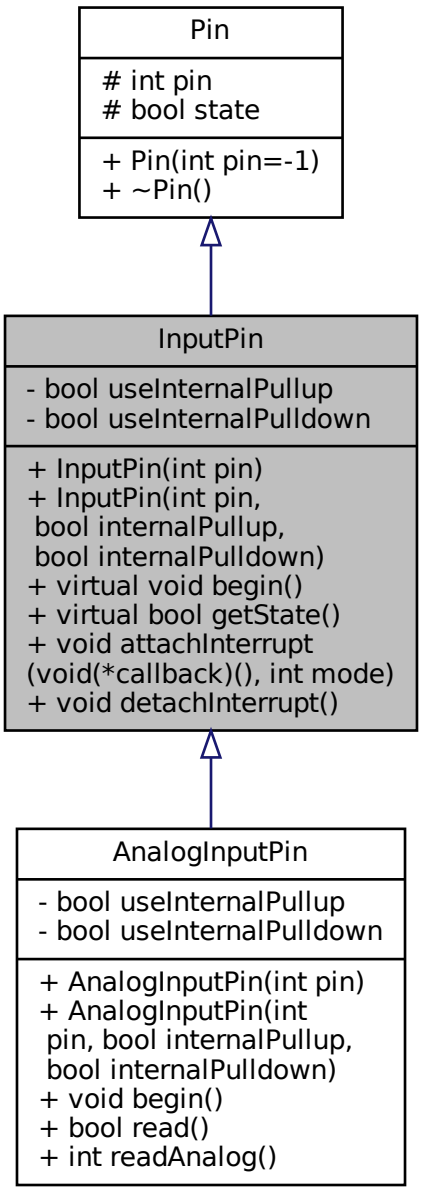
- include/InputHandler.h
- src/InputHandler.cpp

4.12 InputPin Class Reference

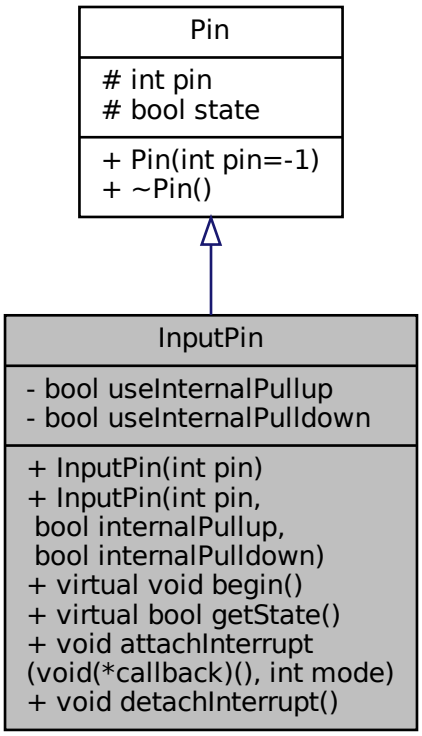
This class represents an input pin on the microcontroller.

`#include <Pin.h>`

Inheritance diagram for InputPin:



Collaboration diagram for InputPin:



Public Member Functions

- InputPin (int pin)

  Construct a new Input Pin:: Input Pin object This constructor initializes the pin and sets the internal pullup and pulldown flags to false. Use this constructor if you do not want to use the internal pullup or pulldown resistors.
- InputPin (int pin, bool internalPullup, bool internalPulldown)

  Construct a new Input Pin:: Input Pin object This constructor initializes the pin and sets the internal pullup and pulldown flags to the specified values. Use this constructor if you want to use the internal pullup or pulldown resistors.
- virtual void begin ()

  This function is used to initialize the input pin. It is intended to be called in the setup() function of the main sketch. However, we don't call it directly, instead we use the begin() function of the derived classes.
- virtual bool getState ()

  This function is used to read the state of the input pin.
- void attachInterrupt (void(∗callback)(), int mode)

  This function is used to attach an interrupt to the input pin.
- void detachInterrupt ()

  This function is used to detach an interrupt from the input pin.

Private Attributes

- bool useInternalPullup
- bool useInternalPulldown

Additional Inherited Members

4.12.1 Detailed Description

This class represents an input pin on the microcontroller.

4.12.2 Constructor & Destructor Documentation

4.12.2.1 InputPin() [1/2]

```
InputPin::InputPin (
            int pin )
```

Construct a new Input Pin:: Input Pin object This constructor initializes the pin and sets the internal pullup and pulldown flags to false. Use this constructor if you do not want to use the internal pullup or pulldown resistors.

Parameters

pin	

4.12.2.2 InputPin() [2/2]

```
InputPin::InputPin (
            int pin,
            bool internalPullup,
            bool internalPulldown )
```

Construct a new Input Pin:: Input Pin object This constructor initializes the pin and sets the internal pullup and pulldown flags to the specified values. Use this constructor if you want to use the internal pullup or pulldown resistors.

Parameters

pin	
internalPullup	
internalPulldown	

4.12.3 Member Function Documentation

4.12.3.1 attachInterrupt()

```
void InputPin::attachInterrupt (
            void(*)() callback,
            int mode )
```

This function is used to attach an interrupt to the input pin.

Parameters

callback	
mode	

4.12.3.2 begin()

```
void InputPin::begin ( )  [virtual]
```

This function is used to initialize the input pin. It is intended to be called in the setup() function of the main sketch. However, we don't call it directly, instead we use the begin() function of the derived classes.

Reimplemented in AnalogInputPin.

4.12.3.3 detachInterrupt()

```
void InputPin::detachInterrupt ( )
```

This function is used to detach an interrupt from the input pin.

4.12.3.4 getState()

```
bool InputPin::getState ( )  [virtual]
```

This function is used to read the state of the input pin.

Returns

bool

4.12.4 Member Data Documentation

4.12.4.1 useInternalPulldown

```
bool InputPin::useInternalPulldown  [private]
```

4.12.4.2 useInternalPullup

```
bool InputPin::useInternalPullup  [private]
```

The documentation for this class was generated from the following files:

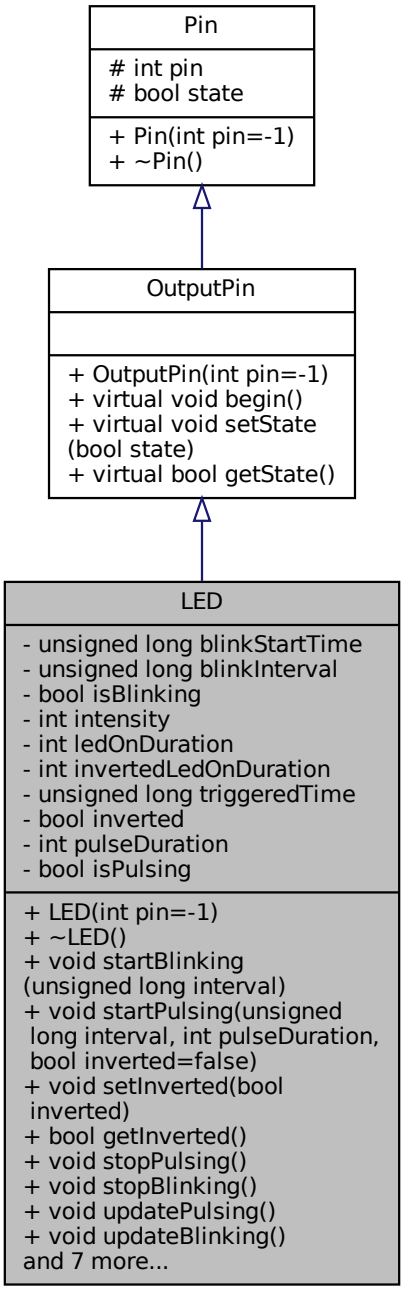
- include/Pin.h
- src/Pin.cpp

4.13 LED Class Reference

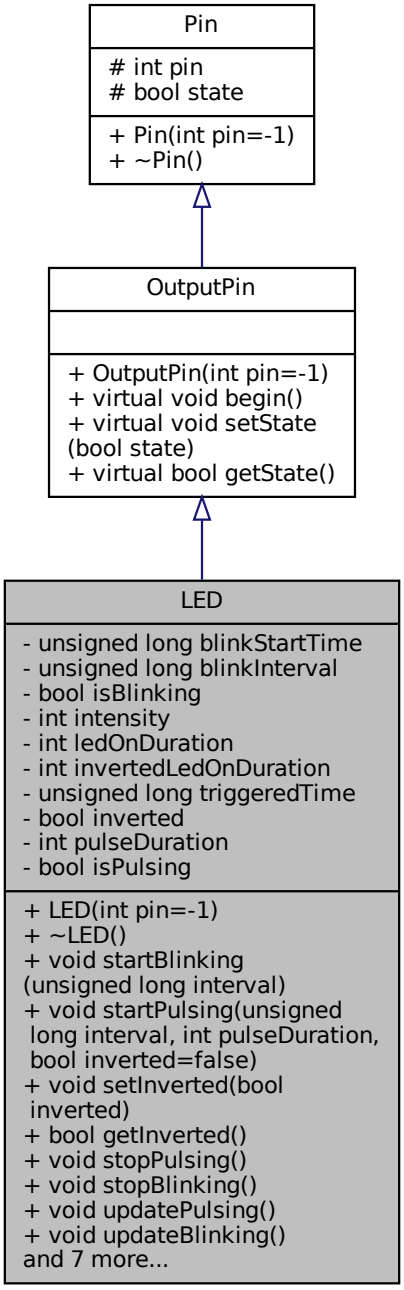
This class defines what an LED is and how it should behave.

```
#include <LED.h>
```

Inheritance diagram for LED:



Collaboration diagram for LED:



Public Member Functions

- LED (int pin=-1)

  Constructor.
- ~LED ()

  Destructor.
- void startBlinking (unsigned long interval)

This function is used to start blinking the LED. The LED's blink based on the interval provided. The blinking is then updated in the updateBlinking() function.

- void startPulsing (unsigned long interval, int pulseDuration, bool inverted=false)

  This function is used to start pulsing the LED. The LED's pulse based on the interval and pulse duration provided.
- void setInverted (bool inverted)
- bool getInverted ()

  Get Inverted state of the LED.
- void stopPulsing ()

  This function is used to stop the LED from pulsing.
- void stopBlinking ()

  This function is used to stop the LED from blinking.
- void updatePulsing ()

  This function is used to update the pulsing of the LED. It is meant to be called in every loop iteration.
- void updateBlinking ()

  This function is used to update the blinking of the LED. It is meant to be called in every loop iteration.
- void setIntensity (int intensity)

  Set the intensity of the LED. I'm 99% sure I"m not actually using this but it is here for future use.
- void trigger (unsigned long currentTime, bool inverted=false)

  This function is used to trigger the LED. The LED will stay on for ledOnDuration milliseconds.
- void update (unsigned long currentTime)

  This function is used to update the state of the LED. It is meant to be called in every loop iteration.
- void resetIvernted ()

  This function is used to reset the inverted state of the LED. The inverted state is used to determine is used to provide a visual feedback when the gate/LED is selected.
- void setLedOnDuration (int duration)

  This function is used to set the duration that the LED should stay on.
- bool getIsBlinking ()

  This function is used to check if the LED is currently blinking.
- bool getIsPulsing ()

  This function is used to check if the LED is currently pulsing.

Private Attributes

- unsigned long blinkStartTime
- unsigned long blinkInterval
- bool isBlinking
- int intensity = 255
- int ledOnDuration = 25
- int invertedLedOnDuration = 40
- unsigned long triggeredTime = 0
- bool inverted = false
- int pulseDuration = 0
- bool isPulsing = false

Additional Inherited Members

4.13.1 Detailed Description

This class defines what an LED is and how it should behave.

4.13.2 Constructor & Destructor Documentation

4.13.2.1 LED()

```
LED::LED (
            int pin = -1 )
```

Constructor.

4.13.2.2 ~LED()

```
LED::~LED ( )
```

Destructor.

4.13.3 Member Function Documentation

4.13.3.1 getInverted()

```
bool LED::getInverted ( )
```

Get Inverted state of the LED.

4.13.3.2 getIsBlinking()

```
bool LED::getIsBlinking ( )
```

This function is used to check if the LED is currently blinking.

Returns

true

false

4.13.3.3 getIsPulsing()

```
bool LED::getIsPulsing ( )
```

This function is used to check if the LED is currently pulsing.

Returns

true

false

4.13.3.4 resetIvernted()

```
void LED::resetIvernted ( )
```

This function is used to reset the inverted state of the LED. The inverted state is used to determine is used to provide a visual feedback when the gate/LED is selected.

4.13.3.5 setIntensity()

```
void LED::setIntensity (
            int intensity )
```

Set the intensity of the LED. I'm 99% sure I"m not actually using this but it is here for future use.

Parameters

intensity	

4.13.3.6 setInverted()

```
void LED::setInverted (
            bool inverted )
```

4.13.3.7 setLedOnDuration()

```
void LED::setLedOnDuration (
            int duration )
```

This function is used to set the duration that the LED should stay on.

Parameters

duration	

4.13.3.8 startBlinking()

```
void LED::startBlinking (
            unsigned long interval )
```

This function is used to start blinking the LED. The LED's blink based on the interval provided. The blinking is then updated in the updateBlinking() function.

Parameters

interval	

4.13.3.9 startPulsing()

```
void LED::startPulsing (
            unsigned long interval,
            int pulseDuration,
            bool inverted = false )
```

This function is used to start pulsing the LED. The LED's pulse based on the interval and pulse duration provided.

4.13.3.10 stopBlinking()

```
void LED::stopBlinking ( )
```

This function is used to stop the LED from blinking.

4.13.3.11 stopPulsing()

```
void LED::stopPulsing ( )
```

This function is used to stop the LED from pulsing.

4.13.3.12 trigger()

```
void LED::trigger (
            unsigned long currentTime,
            bool inverted = false )
```

This function is used to trigger the LED. The LED will stay on for ledOnDuration milliseconds.

Parameters

currentTime	
inverted	

4.13.3.13 update()

```
void LED::update (
            unsigned long currentTime )
```

This function is used to update the state of the LED. It is meant to be called in every loop iteration.

Parameters

currentTime	

4.13.3.14 updateBlinking()

```
void LED::updateBlinking ( )
```

This function is used to update the blinking of the LED. It is meant to be called in every loop iteration.

4.13.3.15 updatePulsing()

```
void LED::updatePulsing ( )
```

This function is used to update the pulsing of the LED. It is meant to be called in every loop iteration.

4.13.4 Member Data Documentation

4.13.4.1 blinkInterval

```
unsigned long LED::blinkInterval  [private]
```

4.13.4.2 blinkStartTime

```
unsigned long LED::blinkStartTime  [private]
```

4.13.4.3 intensity

```
int LED::intensity = 255  [private]
```

4.13.4.4 inverted

```
bool LED::inverted = false  [private]
```

4.13.4.5 invertedLedOnDuration

```
int LED::invertedLedOnDuration = 40  [private]
```

4.13.4.6 isBlinking

```
bool LED::isBlinking  [private]
```

4.13.4.7 isPulsing

```
bool LED::isPulsing = false  [private]
```

4.13.4.8 ledOnDuration

```
int LED::ledOnDuration = 25  [private]
```

4.13.4.9 pulseDuration

```
int LED::pulseDuration = 0  [private]
```

4.13.4.10 triggeredTime

```
unsigned long LED::triggeredTime = 0  [private]
```

The documentation for this class was generated from the following files:

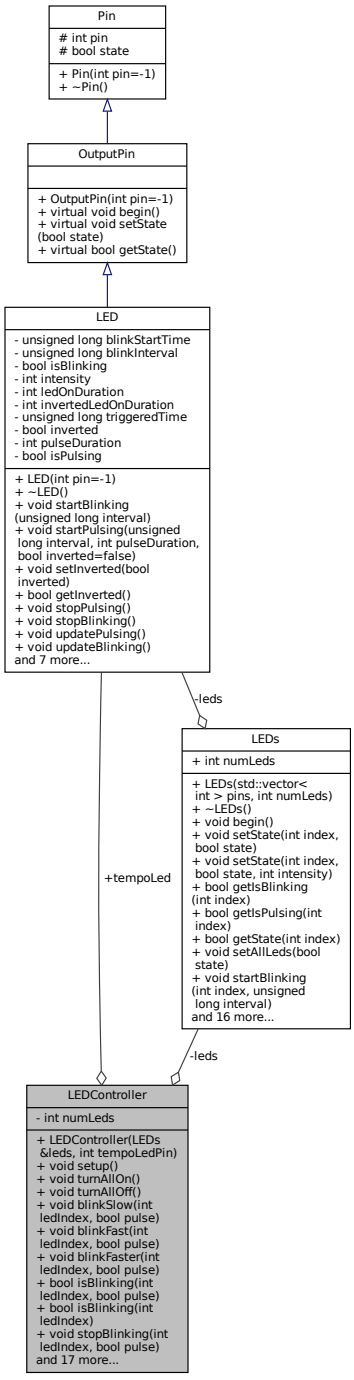
- include/LED.h
- src/LED.cpp

4.14 LEDController Class Reference

This class is used as the main interface for controlling the LEDs.

```
#include <LEDController.h>
```

Collaboration diagram for LEDController:



Public Member Functions

- LEDController (LEDs &leds, int tempoLedPin)

  Construct a new LED Controller:: LED Controller object. By default, all LEDs are turned off.
- void setup ()

  This function is used to setup the LED controller. It is meant to be called in the setup() function of the main sketch.
- void turnAllOn ()

This function is used to turn all LEDs on.

- void turnAllOff ()

  This function is used to turn all LEDs off.
- void blinkSlow (int ledIndex, bool pulse)

  This function is used to start blinking an LED at a slow rate.
- void blinkFast (int ledIndex, bool pulse)

  This function is used to start blinking an LED at a fast rate.
- void blinkFaster (int ledIndex, bool pulse)

  This function is used to start blinking an LED at a faster rate.
- bool isBlinking (int ledIndex, bool pulse)

  Check if an LED is currently blinking.
- bool isBlinking (int ledIndex)
- void stopBlinking (int ledIndex, bool pulse)

  This function is used to stop blinking an LED.
- void stopBlinking (int ledIndex)

  This function is used to stop blinking an LED.
- void stopAllBlinking (bool pulse)

  This function is used to stop all LEDs from blinking.
- void resetInverted ()

  This function is used to reset the inverted state of all LEDs.
- void resetInverted (int ledIndex)

  This function is used to reset the inverted state of a specific LED.
- void setInverted (int ledIndex, bool inverted)

  This function is used to set the inverted state of an LED.
- void setAllInverted (bool inverted)

  This function is used to set the inverted state of all LEDs.
- bool getInverted (int ledIndex)

  This function is used to get the inverted state of an LED.
- int getNumLeds ()

  returns the total number of LEDs in the LED array.
- void update ()

  This function updates the LEDs. It is meant to be called in every loop iteration.
- void update (unsigned long currentTime)

  This function updates the LEDs. It is meant to be called in every loop iteration.
- void update (int ledIndex, unsigned long currentTime)

  This function updates a specific LED. It is meant to be called in every loop iteration.
- void clearAndResetLEDs ()

  Helper function to clear and reset all LEDs.
- void clearLEDs ()

  Helper function to clear all LEDs.
- void updateBlinking ()

  This function is used to update the blinking of the LED. It is meant to be called in every loop iteration.
- void updatePulsing ()

  This function is used to update the pulsing of the LED. It is meant to be called in every loop iteration.
- void setState (int ledIndex, bool state)

  This function is used to set the state of an LED.
- void trigger (int index, unsigned long currentTime, bool inverted=false)

  This function is used to trigger an LED. The LED will stay on for ledOnDuration milliseconds.

Public Attributes

- LED tempoLed

Private Attributes

- LEDs & leds
- int numLeds

4.14.1 Detailed Description

This class is used as the main interface for controlling the LEDs.

4.14.2 Constructor & Destructor Documentation

4.14.2.1 LEDController()

```
LEDController::LEDController (
            LEDs & leds,
            int tempoLedPin )
```

Construct a new LED Controller:: LED Controller object. By default, all LEDs are turned off.

Parameters

leds	

4.14.3 Member Function Documentation

4.14.3.1 blinkFast()

```
void LEDController::blinkFast (
            int ledIndex,
            bool pulse = false )
```

This function is used to start blinking an LED at a fast rate.

Parameters

ledIndex	

4.14.3.2 blinkFaster()

```
void LEDController::blinkFaster (
            int ledIndex,
            bool pulse = false )
```

This function is used to start blinking an LED at a faster rate.

Parameters

ledIndex	

4.14.3.3 blinkSlow()

```
void LEDController::blinkSlow (
            int ledIndex,
            bool pulse = false )
```

This function is used to start blinking an LED at a slow rate.

Parameters

ledIndex	

4.14.3.4 clearAndResetLEDs()

```
void LEDController::clearAndResetLEDs ( )
```

Helper function to clear and reset all LEDs.

4.14.3.5 clearLEDs()

```
void LEDController::clearLEDs ( )
```

Helper function to clear all LEDs.

4.14.3.6 getInverted()

```
bool LEDController::getInverted (
            int ledIndex )
```

This function is used to get the inverted state of an LED.

Parameters

ledIndex	

Returns

bool

4.14.3.7 getNumLeds()

```
int LEDController::getNumLeds ( )
```

returns the total number of LEDs in the LED array.

Returns

int

4.14.3.8 isBlinking() [1/2]

```
bool LEDController::isBlinking (
            int ledIndex )
```

4.14.3.9 isBlinking() [2/2]

```
bool LEDController::isBlinking (
            int ledIndex,
            bool pulse = false )
```

Check if an LED is currently blinking.

4.14.3.10 resetInverted() [1/2]

```
void LEDController::resetInverted ( )
```

This function is used to reset the inverted state of all LEDs.

4.14.3.11 resetInverted() [2/2]

```
void LEDController::resetInverted (
            int ledIndex )
```

This function is used to reset the inverted state of a specific LED.

Parameters

ledIndex	

4.14.3.12 setAllInverted()

```
void LEDController::setAllInverted (
            bool inverted )
```

This function is used to set the inverted state of all LEDs.

Parameters

inverted	

4.14.3.13 setInverted()

```
void LEDController::setInverted (
            int ledIndex,
            bool inverted )
```

This function is used to set the inverted state of an LED.

Parameters

ledIndex	
inverted	

4.14.3.14 setState()

```
void LEDController::setState (
            int ledIndex,
            bool state )
```

This function is used to set the state of an LED.

Parameters

ledIndex	
state	

4.14.3.15 setup()

```
void LEDController::setup ( )
```

This function is used to setup the LED controller. It is meant to be called in the setup() function of the main sketch.

4.14.3.16 stopAllBlinking()

```
void LEDController::stopAllBlinking (
            bool pulse = false )
```

This function is used to stop all LEDs from blinking.

4.14.3.17 stopBlinking() [1/2]

```
void LEDController::stopBlinking (
            int ledIndex )
```

This function is used to stop blinking an LED.

Parameters

ledIndex	

4.14.3.18 stopBlinking() [2/2]

```
void LEDController::stopBlinking (
            int ledIndex,
            bool pulse = false )
```

This function is used to stop blinking an LED.

Parameters

ledIndex	

4.14.3.19 trigger()

```
void LEDController::trigger (
            int index,
```

```
            unsigned long currentTime,
            bool inverted = false )
```

This function is used to trigger an LED. The LED will stay on for ledOnDuration milliseconds.

Parameters

index	
currentTime	
inverted	

4.14.3.20 turnAllOff()

```
void LEDController::turnAllOff ( )
```

This function is used to turn all LEDs off.

4.14.3.21 turnAllOn()

```
void LEDController::turnAllOn ( )
```

This function is used to turn all LEDs on.

4.14.3.22 update() [1/3]

```
void LEDController::update ( )
```

This function updates the LEDs. It is meant to be called in every loop iteration.

4.14.3.23 update() [2/3]

```
void LEDController::update (
            int ledIndex,
            unsigned long currentTime )
```

This function updates a specific LED. It is meant to be called in every loop iteration.

Parameters

ledIndex	
currentTime	

4.14.3.24 update() [3/3]

```
void LEDController::update (
            unsigned long currentTime )
```

This function updates the LEDs. It is meant to be called in every loop iteration.

Parameters

currentTime	

4.14.3.25 updateBlinking()

```
void LEDController::updateBlinking ( )
```

This function is used to update the blinking of the LED. It is meant to be called in every loop iteration.

4.14.3.26 updatePulsing()

```
void LEDController::updatePulsing ( )
```

This function is used to update the pulsing of the LED. It is meant to be called in every loop iteration.

4.14.4 Member Data Documentation

4.14.4.1 leds

```
LEDs& LEDController::leds  [private]
```

4.14.4.2 numLeds

```
int LEDController::numLeds  [private]
```

4.14.4.3 tempoLed

```
LED LEDController::tempoLed
```

The documentation for this class was generated from the following files:

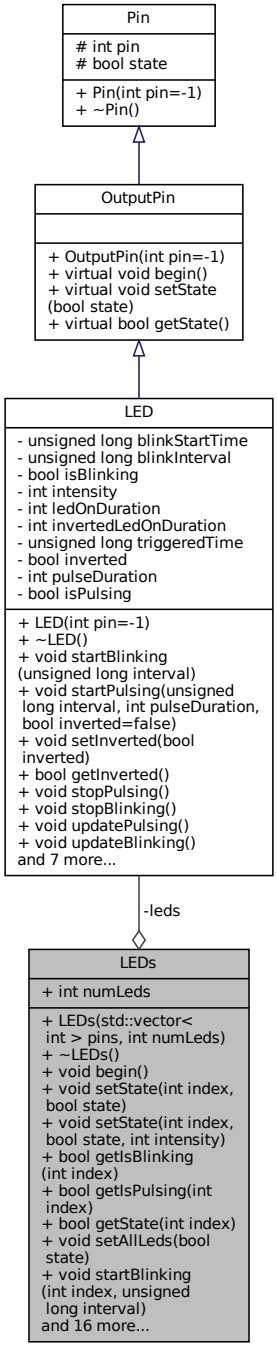
- include/LEDController.h
- src/LEDController.cpp

4.15 LEDs Class Reference

This is a collection of LEDs and mainly used by the LEDController class. Use that if you need to interact with the LEDs.

```
#include <LEDs.h>
```

Collaboration diagram for LEDs:



Public Member Functions

- LEDs (std::vector< int > pins, int numLeds)

  Constructor.
- ~LEDs ()

  Destructor.
- void begin ()

This function is used to initialize the pins for the LEDs as outputs. It is intended to be called in the setup() function of the main sketch.

- void setState (int index, bool state)

  This function is used to set the state of a specific LED. Possible states are HIGH or LOW.
- void setState (int index, bool state, int intensity)
- bool getIsBlinking (int index)
- bool getIsPulsing (int index)

  THis function is used to get the pulse state of a specific LED.
- bool getState (int index)

  This function is used to get the state of a specific LED.
- void setAllLeds (bool state)

  This function is used to set the state of all the LEDs. Possible states are HIGH or LOW.
- void startBlinking (int index, unsigned long interval)

  This function is used to start blinking an LED. The LED's blink based on the interval provided.
- void stopBlinking (int index)

  This function is used stop the LED from blinking.
- void stopAllBlinking ()

  This function is used to stop all LEDs from blinking.
- void updateBlinking ()

  This function is used to update the blinking of the LEDs. It is meant to be called in every loop iteration.
- void startPulsing (int index, unsigned long interval, int pulseDuration, bool inverted=false)

  This function is used to start pulsing an LED. The LED will pulse based on the interval and pulse duration provided.
- void stopPulsing (int index)

  This function is used to stop the LED from pulsing.
- void stopAllPulsing ()

  This function is used to stop all LEDs from pulsing.
- void updatePulsing ()

  This function is used to update the pulsing of the LEDs. It is meant to be called in every loop iteration.
- void setInverted (int index, bool inverted)

  This function is used to set the inverted state of a specific LED.
- void setAllInverted (bool inverted)

  This function is used to set the inverted state of all LEDs.
- bool getInverted (int index)

  This function is used to get the inverted state of a specific LED.
- void setIntensity (int index, int intensity)

  This function is used to set the intensity of a specific LED.
- void setAllintensity (int intensity)

  This function is used to set the intensity of all LEDs.
- void update (unsigned long currentTime)

  This function is used to update the state of the LEDs. It is meant to be called in every loop iteration.
- void update (int index, unsigned long currentTime)

  This function is used to update the state of a specific LED. It is meant to be called in every loop iteration.
- void trigger (int index, unsigned long currentTime, bool inverted=false)

  This function is used to trigger the LED. The LED will stay on for ledOnDuration milliseconds.
- void resetInverted (int index)

  This function is used to reset the inverted state of all LEDs.

Public Attributes

- int numLeds

Private Attributes

- LED * leds

4.15.1 Detailed Description

This is a collection of LEDs and mainly used by the LEDController class. Use that if you need to interact with the LEDs.

4.15.2 Constructor & Destructor Documentation

4.15.2.1 LEDs()

```
LEDs::LEDs (
            std::vector< int > pins,
            int numLeds )
```

Constructor.

4.15.2.2 ~LEDs()

```
LEDs::~LEDs ( )
```

Destructor.

4.15.3 Member Function Documentation

4.15.3.1 begin()

```
void LEDs::begin ( )
```

This function is used to initialize the pins for the LEDs as outputs. It is intended to be called in the setup() function of the main sketch.

4.15.3.2 getInverted()

```
bool LEDs::getInverted (
            int index )
```

This function is used to get the inverted state of a specific LED.

Parameters

index	

Returns

bool

4.15.3.3 getIsBlinking()

```
bool LEDs::getIsBlinking (
            int index )
```

4.15.3.4 getIsPulsing()

```
bool LEDs::getIsPulsing (
            int index )
```

THis function is used to get the pulse state of a specific LED.

Returns

bool

4.15.3.5 getState()

```
bool LEDs::getState (
            int index )
```

This function is used to get the state of a specific LED.

Parameters

index	

Returns

state of the LED as a boolean

4.15.3.6 resetInverted()

```
void LEDs::resetInverted (
            int index )
```

This function is used to reset the inverted state of all LEDs.

4.15.3.7 setAllintensity()

```
void LEDs::setAllintensity (
            int intensity )
```

This function is used to set the intensity of all LEDs.

Parameters

intensity	

4.15.3.8 setAllInverted()

```
void LEDs::setAllInverted (
            bool inverted )
```

This function is used to set the inverted state of all LEDs.

Parameters

inverted	

4.15.3.9 setAllLeds()

```
void LEDs::setAllLeds (
            bool state )
```

This function is used to set the state of all the LEDs. Possible states are HIGH or LOW.

Parameters

state	

4.15.3.10 setIntensity()

```
void LEDs::setIntensity (
            int index,
            int intensity )
```

This function is used to set the intensity of a specific LED.

Parameters

index	
intensity	

4.15.3.11 setInverted()

```
void LEDs::setInverted (
            int index,
            bool inverted )
```

This function is used to set the inverted state of a specific LED.

Parameters

index	
inverted	

4.15.3.12 setState() [1/2]

```
void LEDs::setState (
            int index,
            bool state )
```

This function is used to set the state of a specific LED. Possible states are HIGH or LOW.

Parameters

index	
state	

4.15.3.13 setState() [2/2]

```
void LEDs::setState (
            int index,
```

```
            bool state,
            int intensity )
```

4.15.3.14 startBlinking()

```
void LEDs::startBlinking (
            int index,
            unsigned long interval )
```

This function is used to start blinking an LED. The LED's blink based on the interval provided.

Parameters

index	
interval	

4.15.3.15 startPulsing()

```
void LEDs::startPulsing (
            int index,
            unsigned long interval,
            int pulseDuration,
            bool inverted = false )
```

This function is used to start pulsing an LED. The LED will pulse based on the interval and pulse duration provided.

Parameters

index	
interval	
pulseDuration	
inverted	

4.15.3.16 stopAllBlinking()

```
void LEDs::stopAllBlinking ( )
```

This function is used to stop all LEDs from blinking.

4.15.3.17 stopAllPulsing()

```
void LEDs::stopAllPulsing ( )
```

This function is used to stop all LEDs from pulsing.

4.15.3.18 stopBlinking()

```
void LEDs::stopBlinking (
            int index )
```

This function is used stop the LED from blinking.

Parameters

index	

4.15.3.19 stopPulsing()

```
void LEDs::stopPulsing (
            int index )
```

This function is used to stop the LED from pulsing.

Parameters

index	

4.15.3.20 trigger()

```
void LEDs::trigger (
            int index,
            unsigned long currentTime,
            bool inverted = false )
```

This function is used to trigger the LED. The LED will stay on for ledOnDuration milliseconds.

Parameters

index	
currentTime	
inverted	

4.15.3.21 update() [1/2]

```
void LEDs::update (
            int index,
            unsigned long currentTime )
```

This function is used to update the state of a specific LED. It is meant to be called in every loop iteration.

Parameters

index	
currentTime	

4.15.3.22 update() [2/2]

```
void LEDs::update (
            unsigned long currentTime )
```

This function is used to update the state of the LEDs. It is meant to be called in every loop iteration.

Parameters

currentTime	

4.15.3.23 updateBlinking()

```
void LEDs::updateBlinking ( )
```

This function is used to update the blinking of the LEDs. It is meant to be called in every loop iteration.

4.15.3.24 updatePulsing()

```
void LEDs::updatePulsing ( )
```

This function is used to update the pulsing of the LEDs. It is meant to be called in every loop iteration.

4.15.4 Member Data Documentation

4.15.4.1 leds

```
LED* LEDs::leds  [private]
```

4.15.4.2 numLeds

```
int LEDs::numLeds
```

The documentation for this class was generated from the following files:

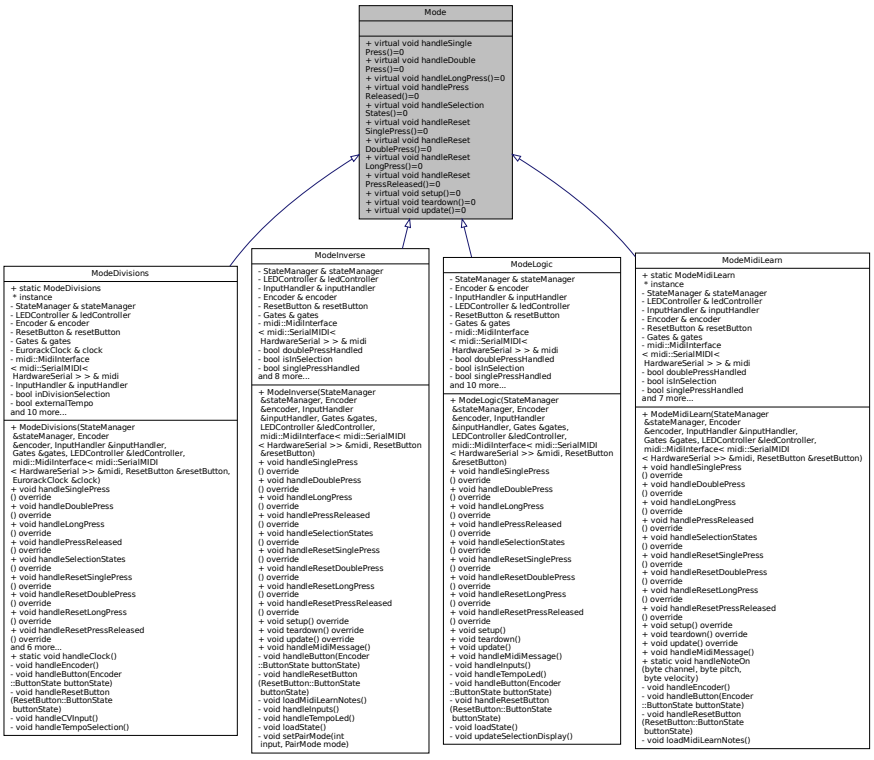
- include/LEDs.h
- src/LEDs.cpp

4.16 Mode Class Reference

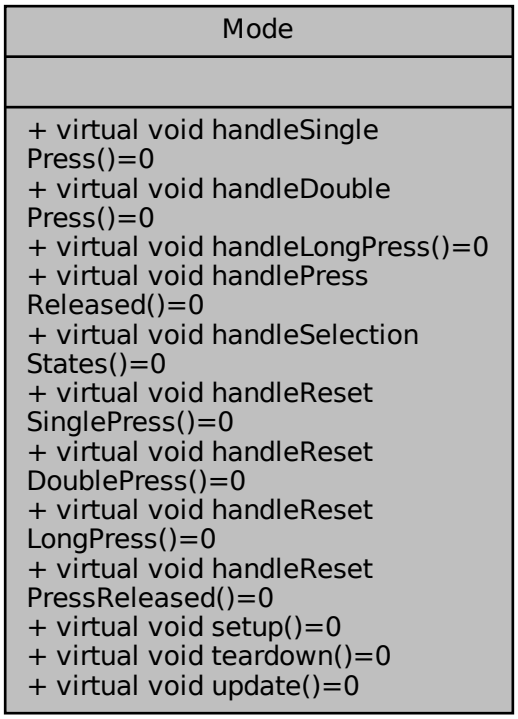
This class is the base for our application modes.

```
#include <Mode.h>
```

Inheritance diagram for Mode:



Collaboration diagram for Mode:



Public Member Functions

- virtual void handleSinglePress ()=0
- virtual void handleDoublePress ()=0
- virtual void handleLongPress ()=0
- virtual void handlePressReleased ()=0
- virtual void handleSelectionStates ()=0
- virtual void handleResetSinglePress ()=0
- virtual void handleResetDoublePress ()=0
- virtual void handleResetLongPress ()=0
- virtual void handleResetPressReleased ()=0
- virtual void setup ()=0
- virtual void teardown ()=0
- virtual void update ()=0

4.16.1 Detailed Description

This class is the base for our application modes.

4.16.2 Member Function Documentation

4.16.2.1 handleDoublePress()

```
virtual void Mode::handleDoublePress ( )  [pure virtual]
```

Implemented in ModeMidiLearn, ModeLogic, ModeInverse, and ModeDivisions.

4.16.2.2 handleLongPress()

```
virtual void Mode::handleLongPress ( )  [pure virtual]
```

Implemented in ModeMidiLearn, ModeLogic, ModeInverse, and ModeDivisions.

4.16.2.3 handlePressReleased()

```
virtual void Mode::handlePressReleased ( )  [pure virtual]
```

Implemented in ModeMidiLearn, ModeLogic, ModeInverse, and ModeDivisions.

4.16.2.4 handleResetDoublePress()

```
virtual void Mode::handleResetDoublePress ( )  [pure virtual]
```

Implemented in ModeMidiLearn, ModeLogic, ModeInverse, and ModeDivisions.

4.16.2.5 handleResetLongPress()

```
virtual void Mode::handleResetLongPress ( )  [pure virtual]
```

Implemented in ModeMidiLearn, ModeLogic, ModeInverse, and ModeDivisions.

4.16.2.6 handleResetPressReleased()

```
virtual void Mode::handleResetPressReleased ( )  [pure virtual]
```

Implemented in ModeMidiLearn, ModeLogic, ModeInverse, and ModeDivisions.

4.16.2.7 handleResetSinglePress()

```
virtual void Mode::handleResetSinglePress ( )  [pure virtual]
```

Implemented in ModeMidiLearn, ModeLogic, ModeInverse, and ModeDivisions.

4.16.2.8 handleSelectionStates()

```
virtual void Mode::handleSelectionStates ( )  [pure virtual]
```

Implemented in ModeMidiLearn, ModeLogic, ModeInverse, and ModeDivisions.

4.16.2.9 handleSinglePress()

```
virtual void Mode::handleSinglePress ( )  [pure virtual]
```

Implemented in ModeMidiLearn, ModeLogic, ModeInverse, and ModeDivisions.

4.16.2.10 setup()

```
virtual void Mode::setup ( )  [pure virtual]
```

Implemented in ModeMidiLearn, ModeInverse, ModeDivisions, and ModeLogic.

4.16.2.11 teardown()

```
virtual void Mode::teardown ( )  [pure virtual]
```

Implemented in ModeMidiLearn, ModeInverse, ModeDivisions, and ModeLogic.

4.16.2.12 update()

```
virtual void Mode::update ( )  [pure virtual]
```

Implemented in ModeMidiLearn, ModeInverse, ModeDivisions, and ModeLogic.

The documentation for this class was generated from the following file:

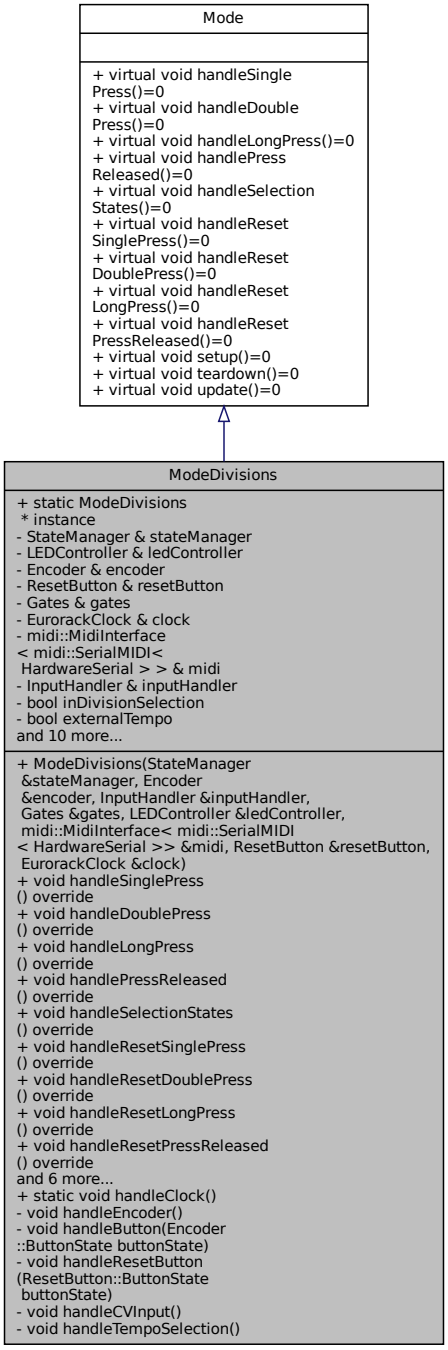
- include/Mode.h

4.17 ModeDivisions Class Reference

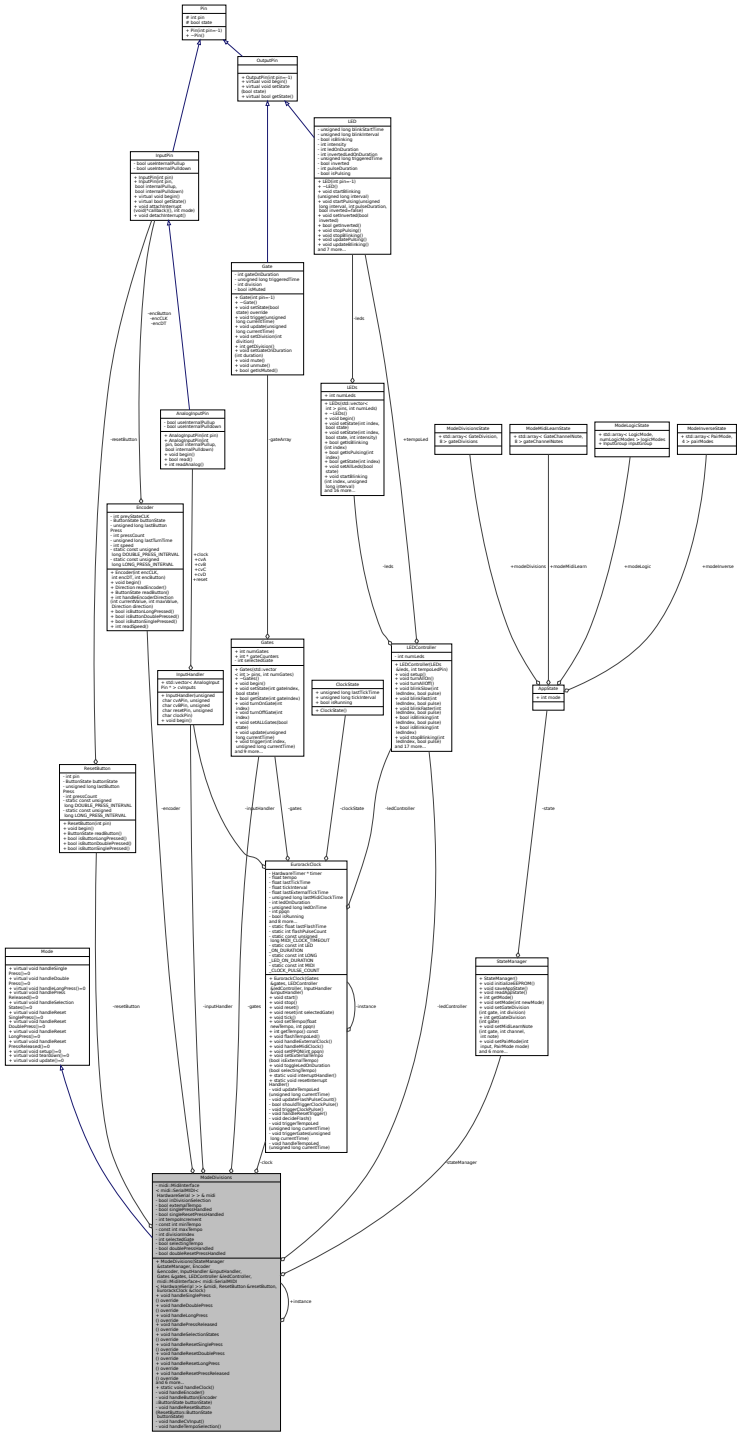
This class uses the eurorack clock to provide us pullses with selectable division. It can be synced to a clock too, internal and external.

```
#include <ModeDivisions.h>
```

Inheritance diagram for ModeDivisions:



Collaboration diagram for ModeDivisions:



Public Member Functions

- ModeDivisions (StateManager &stateManager, Encoder &encoder, InputHandler &inputHandler, Gates &gates, LEDController &ledController, midi::MidiInterface< midi::SerialMIDI< HardwareSerial >> &midi, ResetButton &resetButton, EurorackClock &clock)

  Construct a new Mode 0:: Mode 0 object.
- void handleSinglePress () override

Handle single press. Default behavior is to toggle between division selection and gate selection.

- void handleDoublePress () override

  Handle double press. Default behavior is to enter or exit tempo selection mode.

- void handleLongPress () override

  This function is used to handle long press of the button. However, it doesn't do anything yet.

- void handlePressReleased () override

  Handle press released. Default behavior is to do nothing.

- void handleSelectionStates () override

  Handle selection states. Default behavior is to handle tempo selection.

- void handleResetSinglePress () override

  Handle reset single press. Default behavior is to reset the selected gate so it can by synced with the clock.

- void handleResetDoublePress () override

  Handle reset double press. Default behavior is to reset the clock so it can be synced with an external clock.

- void handleResetLongPress () override

  This function is used to handle long press of the reset button. However, it doesn't do anything yet.

- void handleResetPressReleased () override

  This function is used to handle reset press released. However, it doesn't do anything yet.

- void setup () override

  Setup and teardown methods are meant to be called when Mode selector switches modes. This is where you can put code that should only run once when the mode is switched to. It is configured to run once when the mode is switched to and once when the mode is switched from.

- void teardown () override

  This block of code is executed once whenever we switch modes. The code here is intended to be cleanup code. This is where you can put code that should only run once when the mode is switched from.

- void update () override

  The update method is meant to be called every loop iteration. This is where you can put code that should run every loop iteration.

- void setDivisionPPQN (int ppqn)
- void setDefaultDivisionIndex ()

  Set the default division index based on the internal PPQN value, only used by the constructor to avoid compile errors.

- void handleMidiMessage ()

  Handle MIDI messages. This function is called by the update method.

Static Public Member Functions

- static void handleClock ()

  This function is used to handle MIDI clock messages.

Static Public Attributes

- static ModeDivisions * instance = nullptr

  This is the instance of the ModeDivisions class. We need this in order to work with the MIDI library. The library requires a static function to be called when a MIDI message is received.

Private Member Functions

- void handleEncoder ()

  Detects the direction of the encoder and updates the selected gate or division based on the direction.
- void handleButton (Encoder::ButtonState buttonState)

  This block of code is used to handle button presses. It is called by the update method.
- void handleResetButton (ResetButton::ButtonState buttonState)

  This block of code is used to handle reset button presses. It is called by the update method.
- void handleCVInput ()

  block of code is here to handle inputs from the CV Input Jacks. It doesn't do anything now but is here for future use.
- void handleTempoSelection ()

  Handle tempo selection. Default behavior is to increase or decrease the tempo based on the encoder direction.

Private Attributes

- StateManager & stateManager
- LEDController & ledController
- Encoder & encoder
- ResetButton & resetButton
- Gates & gates
- EurorackClock & clock
- midi::MidiInterface< midi::SerialMIDI< HardwareSerial > > & midi
- InputHandler & inputHandler
- bool inDivisionSelection = false
- bool externalTempo = false
- bool singlePressHandled = false
- bool singleResetPressHandled = false
- int tempoIncrement = 1
- const int minTempo = 20
- const int maxTempo = 340
- int divisionIndex = 24
- int selectedGate = 0
- bool selectingTempo = false
- bool doublePressHandled = false
- bool doubleResetPressHandled = false

4.17.1 Detailed Description

This class uses the eurorack clock to provide us pullses with selectable division. It can be synced to a clock too, internal and external.

4.17.2 Constructor & Destructor Documentation

4.17.2.1 ModeDivisions()

```
ModeDivisions::ModeDivisions (
            StateManager & stateManager,
            Encoder & encoder,
            InputHandler & inputHandler,
            Gates & gates,
            LEDController & ledController,
            midi::MidiInterface< midi::SerialMIDI< HardwareSerial >> & midi,
            ResetButton & resetButton,
            EurorackClock & clock )
```

Construct a new Mode 0:: Mode 0 object.

Parameters

stateManager	
encoder	
inputHandler	
gates	
ledController	
midi	
resetButton	
clock	

4.17.3 Member Function Documentation

4.17.3.1 handleButton()

```
void ModeDivisions::handleButton (
            Encoder::ButtonState buttonState )  [private]
```

This block of code is used to handle button presses. It is called by the update method.

Parameters

buttonState	

4.17.3.2 handleClock()

```
void ModeDivisions::handleClock ( )  [static]
```

This function is used to handle MIDI clock messages.

4.17.3.3 handleCVInput()

```
void ModeDivisions::handleCVInput ( )  [private]
```

block of code is here to handle inputs from the CV Input Jacks. It doesn't do anything now but is here for future use.

4.17.3.4 handleDoublePress()

```
void ModeDivisions::handleDoublePress ( )  [override], [virtual]
```

Handle double press. Default behavior is to enter or exit tempo selection mode.

Implements Mode.

4.17.3.5 handleEncoder()

```
void ModeDivisions::handleEncoder ( )  [private]
```

Detects the direction of the encoder and updates the selected gate or division based on the direction.

4.17.3.6 handleLongPress()

```
void ModeDivisions::handleLongPress ( )  [override], [virtual]
```

This function is used to handle long press of the button. However, it doesn't do anything yet.

Long press is used by modeSelector, so don't use that here.

Implements Mode.

4.17.3.7 handleMidiMessage()

```
void ModeDivisions::handleMidiMessage ( )
```

Handle MIDI messages. This function is called by the update method.

4.17.3.8 handlePressReleased()

```
void ModeDivisions::handlePressReleased ( )  [override], [virtual]
```

Handle press released. Default behavior is to do nothing.

Mode 0 specific press released handling

Implements Mode.

4.17.3.9 handleResetButton()

```
void ModeDivisions::handleResetButton (
            ResetButton::ButtonState buttonState )  [private]
```

This block of code is used to handle reset button presses. It is called by the update method.

Parameters

buttonState	

4.17.3.10 handleResetDoublePress()

```
void ModeDivisions::handleResetDoublePress ( )  [override], [virtual]
```

Handle reset double press. Default behavior is to reset the clock so it can be synced with an external clock.

Implements Mode.

4.17.3.11 handleResetLongPress()

```
void ModeDivisions::handleResetLongPress ( )  [override], [virtual]
```

This function is used to handle long press of the reset button. However, it doesn't do anything yet.

Does nothing yet but it could. :)

Implements Mode.

4.17.3.12 handleResetPressReleased()

```
void ModeDivisions::handleResetPressReleased ( )  [override], [virtual]
```

This function is used to handle reset press released. However, it doesn't do anything yet.

Does nothing yet but it could. :)

Implements Mode.

4.17.3.13 handleResetSinglePress()

```
void ModeDivisions::handleResetSinglePress ( )  [override], [virtual]
```

Handle reset single press. Default behavior is to reset the selected gate so it can by synced with the clock.

Implements Mode.

4.17.3.14 handleSelectionStates()

```
void ModeDivisions::handleSelectionStates ( )  [override], [virtual]
```

Handle selection states. Default behavior is to handle tempo selection.

Implements Mode.

4.17.3.15 handleSinglePress()

```
void ModeDivisions::handleSinglePress ( )  [override], [virtual]
```

Handle single press. Default behavior is to toggle between division selection and gate selection.

If in division selection update the division for the selected gate

Toggle between division selection and gate selection

Implements Mode.

4.17.3.16 handleTempoSelection()

```
void ModeDivisions::handleTempoSelection ( )  [private]
```

Handle tempo selection. Default behavior is to increase or decrease the tempo based on the encoder direction.

If externalTempo, exit external tempo mode and increase the tempo

Enter external tempo mode when the tempo reaches the minimum

4.17.3.17 setDefaultDivisionIndex()

```
void ModeDivisions::setDefaultDivisionIndex ( )
```

Set the default division index based on the internal PPQN value, only used by the constructor to avoid compile errors.

4.17.3.18 setDivisionPPQN()

```
void ModeDivisions::setDivisionPPQN (
            int ppqn )
```

4.17.3.19 setup()

```
void ModeDivisions::setup ( )  [override], [virtual]
```

Setup and teardown methods are meant to be called when Mode selector switches modes. This is where you can put code that should only run once when the mode is switched to. It is configured to run once when the mode is switched to and once when the mode is switched from.

Implements Mode.

4.17.3.20 teardown()

```
void ModeDivisions::teardown ( )  [override], [virtual]
```

This block of code is executed once whenever we switch modes. The code here is intended to be cleanup code. This is where you can put code that should only run once when the mode is switched from.

Implements Mode.

4.17.3.21 update()

```
void ModeDivisions::update ( )  [override], [virtual]
```

The update method is meant to be called every loop iteration. This is where you can put code that should run every loop iteration.

Implements Mode.

4.17.4 Member Data Documentation

4.17.4.1 clock

```
EurorackClock& ModeDivisions::clock  [private]
```

4.17.4.2 divisionIndex

```
int ModeDivisions::divisionIndex = 24  [private]
```

4.17.4.3 doublePressHandled

```
bool ModeDivisions::doublePressHandled = false  [private]
```

4.17.4.4 doubleResetPressHandled

```
bool ModeDivisions::doubleResetPressHandled = false  [private]
```

4.17.4.5 encoder

```
Encoder& ModeDivisions::encoder  [private]
```

4.17.4.6 externalTempo

```
bool ModeDivisions::externalTempo = false  [private]
```

4.17.4.7 gates

```
Gates& ModeDivisions::gates  [private]
```

4.17.4.8 inDivisionSelection

```
bool ModeDivisions::inDivisionSelection = false  [private]
```

4.17.4.9 inputHandler

```
InputHandler& ModeDivisions::inputHandler  [private]
```

4.17.4.10 instance

```
ModeDivisions * ModeDivisions::instance = nullptr  [static]
```

This is the instance of the ModeDivisions class. We need this in order to work with the MIDI library. The library requires a static function to be called when a MIDI message is received.

4.17.4.11 ledController

```
LEDController& ModeDivisions::ledController  [private]
```

4.17.4.12 maxTempo

```
const int ModeDivisions::maxTempo = 340  [private]
```

4.17.4.13 midi

```
midi::MidiInterface<midi::SerialMIDI<HardwareSerial> >& ModeDivisions::midi  [private]
```

4.17.4.14 minTempo

```
const int ModeDivisions::minTempo = 20  [private]
```

4.17.4.15 resetButton

```
ResetButton& ModeDivisions::resetButton  [private]
```

4.17.4.16 selectedGate

```
int ModeDivisions::selectedGate = 0  [private]
```

4.17.4.17 selectingTempo

```
bool ModeDivisions::selectingTempo = false  [private]
```

4.17.4.18 singlePressHandled

```
bool ModeDivisions::singlePressHandled = false  [private]
```

4.17.4.19 singleResetPressHandled

```
bool ModeDivisions::singleResetPressHandled = false  [private]
```

4.17.4.20 stateManager

```
StateManager& ModeDivisions::stateManager  [private]
```

4.17.4.21 tempoIncrement

```
int ModeDivisions::tempoIncrement = 1  [private]
```

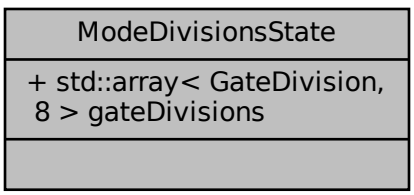
The documentation for this class was generated from the following files:

- include/ModeDivisions.h
- src/ModeDivisions.cpp

4.18 ModeDivisionsState Struct Reference

```
#include <AppState.h>
```

Collaboration diagram for ModeDivisionsState:



Public Attributes

- std::array< GateDivision, 8 > gateDivisions

4.18.1 Member Data Documentation

4.18.1.1 gateDivisions

```
std::array<GateDivision, 8> ModeDivisionsState::gateDivisions
```

The documentation for this struct was generated from the following file:

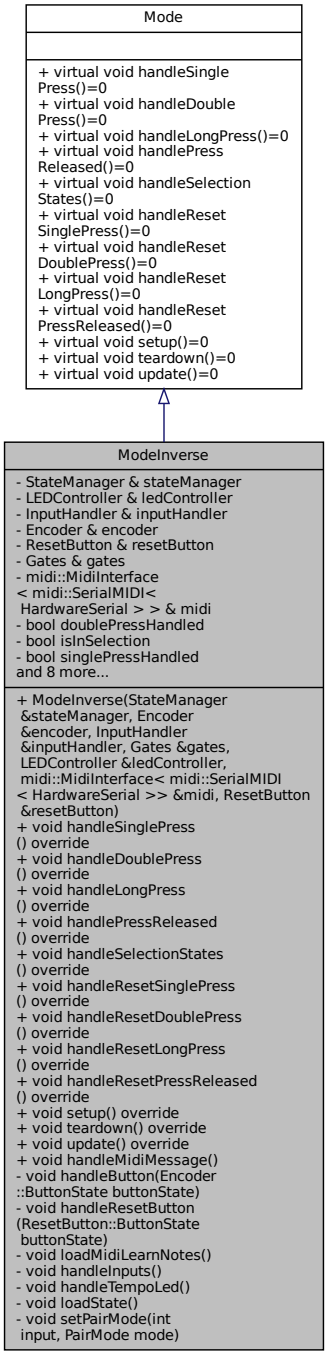
- include/AppState.h

4.19 ModeInverse Class Reference

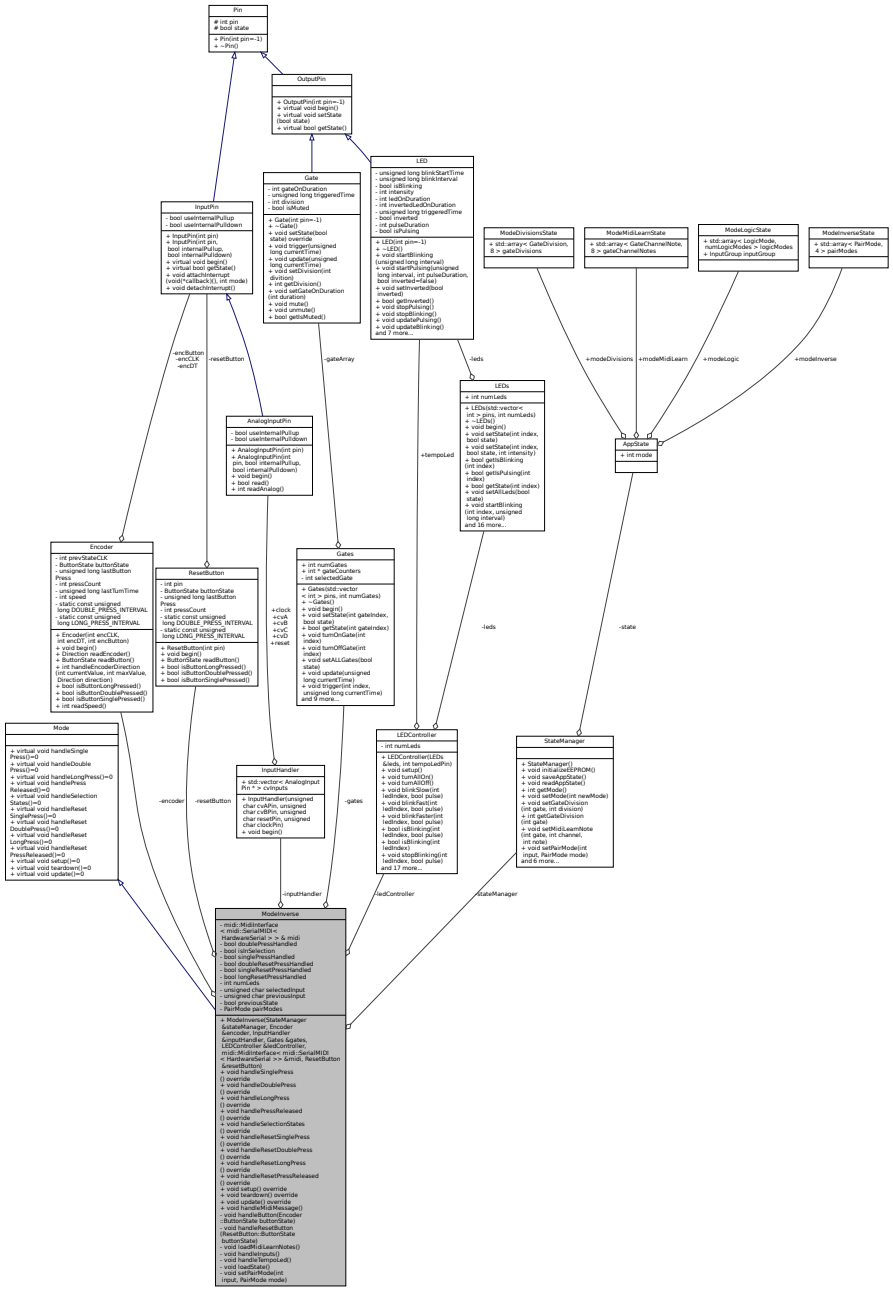
This mode is for inverting the gates. If the gate is high, it will be low and vice versa. The user can select the gate pairs and change the behaviour of the gates. So instead of sending gates, it will send triggers on the separate gates for the rising edge and falling edge of the gate.

```
#include <ModeInverse.h>
```

Inheritance diagram for ModeInverse:



Collaboration diagram for ModeInverse:



Public Member Functions

- ModeInverse (StateManager &stateManager, Encoder &encoder, InputHandler &inputHandler, Gates &gates, LEDController &ledController, midi::MidiInterface< midi::SerialMIDI< HardwareSerial >> &midi, ResetButton &resetButton)

  Construct a new Mode Inverse:: Mode Inverse object.

- void handleSinglePress () override

  Handle the single press event. How in this mode we don't use it.

- void handleDoublePress () override

  This function is used to handle the double press event. How in this mode we don't use it.

- void handleLongPress () override

  This function is used to handle the long press event. How in this mode we don't use it.
- void handlePressReleased () override

  This function is used to handle the press released event. How in this mode we don't use it.
- void handleSelectionStates () override

  This function is used to handle the encoder rotation event.
- void handleResetSinglePress () override

  This function is used to handle the reset single press event. In this mode, we use it to mute/unmute the selected input pairs.
- void handleResetDoublePress () override

  This function is used to handle the reset double press event. How in this mode we don't use it.
- void handleResetLongPress () override

  This function is used to handle the reset long press event. How in this mode we don't use it.
- void handleResetPressReleased () override

  This function is used to handle the reset press released event. How in this mode we don't use it.
- void setup () override

  This function is used to setup the current mode object. Setup and teardown methods are meant to be called when Mode selector switches modes. This is where you can put code that should only run once when the mode is switched to.
- void teardown () override

  Like the setup method, this function is used to teardown the current mode object.
- void update () override

  This function is used to update the current mode object. The update method is meant to be called every loop iteration. This is where you can put code that should run every loop iteration.
- void handleMidiMessage ()

  This function is used to handle the MIDI messages. This is where the MIDI messages are handled. In this mode we only use it to forward messages to the MIDI output. Basically, a soft MIDI thru.

Private Member Functions

- void handleButton (Encoder::ButtonState buttonState)

  This block of code is used to handle button presses. It is called by the update method.
- void handleResetButton (ResetButton::ButtonState buttonState)

  This block of code is used to handle reset button presses. It is called by the update method.
- void loadMidiLearnNotes ()
- void handleInputs ()

  This function is used to handle the inputs. This is where the magic happens.
- void handleTempoLed ()

  This function is used to handle the tempo LED. This is where the tempo LED is updated.
- void loadState ()

  This function is used to load the state of the ModeInverse object.
- void setPairMode (int input, PairMode mode)

  This function is used to set the pair mode for the inputs. This is stored in the pairModes array.

Private Attributes

- StateManager & stateManager
- LEDController & ledController
- InputHandler & inputHandler
- Encoder & encoder
- ResetButton & resetButton
- Gates & gates
- midi::MidiInterface< midi::SerialMIDI< HardwareSerial > > & midi
- bool doublePressHandled = false
- bool isInSelection = false
- bool singlePressHandled = false
- bool doubleResetPressHandled = false
- bool singleResetPressHandled = false
- bool longResetPressHandled = false
- int numLeds = 8
- unsigned char selectedInput = 0
- unsigned char previousInput = 0
- bool previousState [4]
- PairMode pairModes [4]

4.19.1 Detailed Description

This mode is for inverting the gates. If the gate is high, it will be low and vice versa. The user can select the gate pairs and change the behaviour of the gates. So instead of sending gates, it will send triggers on the separate gates for the rising edge and falling edge of the gate.

4.19.2 Constructor & Destructor Documentation

4.19.2.1 ModeInverse()

```
ModeInverse::ModeInverse (
            StateManager & stateManager,
            Encoder & encoder,
            InputHandler & inputHandler,
            Gates & gates,
            LEDController & ledController,
            midi::MidiInterface< midi::SerialMIDI< HardwareSerial >> & midi,
            ResetButton & resetButton )
```

Construct a new Mode Inverse:: Mode Inverse object.

Parameters

stateManager	
encoder	
inputHandler	
gates	
ledController	
midi	
resetButton	

4.19.3 Member Function Documentation

4.19.3.1 handleButton()

```
void ModeInverse::handleButton (
            Encoder::ButtonState buttonState )  [private]
```

This block of code is used to handle button presses. It is called by the update method.

Parameters

buttonState	

4.19.3.2 handleDoublePress()

```
void ModeInverse::handleDoublePress ( )  [override], [virtual]
```

This function is used to handle the double press event. How in this mode we don't use it.

Implements Mode.

4.19.3.3 handleInputs()

```
void ModeInverse::handleInputs ( )  [private]
```

This function is used to handle the inputs. This is where the magic happens.

4.19.3.4 handleLongPress()

```
void ModeInverse::handleLongPress ( )  [override], [virtual]
```

This function is used to handle the long press event. How in this mode we don't use it.

Implements Mode.

4.19.3.5 handleMidiMessage()

```
void ModeInverse::handleMidiMessage ( )
```

This function is used to handle the MIDI messages. This is where the MIDI messages are handled. In this mode we only use it to forward messages to the MIDI output. Basically, a soft MIDI thru.

NOTE: If you need more functionality, you will need to implement callback functions. However, those will need to be static functions. This is because the MIDI library requires static functions. Just like the handleNoteOn and handleNoteOff functions in the ModeMidiLearn class. Remember that you'll need to create an instance of this class if you do that. You can use ModeMidiLearn as a reference.

4.19.3.6 handlePressReleased()

```
void ModeInverse::handlePressReleased ( )  [override], [virtual]
```

This function is used to handle the press released event. How in this mode we don't use it.

Implements Mode.

4.19.3.7 handleResetButton()

```
void ModeInverse::handleResetButton (
            ResetButton::ButtonState buttonState )  [private]
```

This block of code is used to handle reset button presses. It is called by the update method.

Parameters

buttonState	

4.19.3.8 handleResetDoublePress()

```
void ModeInverse::handleResetDoublePress ( )  [override], [virtual]
```

This function is used to handle the reset double press event. How in this mode we don't use it.

Implements Mode.

4.19.3.9 handleResetLongPress()

```
void ModeInverse::handleResetLongPress ( )  [override], [virtual]
```

This function is used to handle the reset long press event. How in this mode we don't use it.

Implements Mode.

4.19.3.10 handleResetPressReleased()

```
void ModeInverse::handleResetPressReleased ( )  [override], [virtual]
```

This function is used to handle the reset press released event. How in this mode we don't use it.

Implements Mode.

4.19.3.11 handleResetSinglePress()

```
void ModeInverse::handleResetSinglePress ( )  [override], [virtual]
```

This function is used to handle the reset single press event. In this mode, we use it to mute/unmute the selected input pairs.

Implements Mode.

4.19.3.12 handleSelectionStates()

```
void ModeInverse::handleSelectionStates ( )  [override], [virtual]
```

This function is used to handle the encoder rotation event.

Implements Mode.

4.19.3.13 handleSinglePress()

```
void ModeInverse::handleSinglePress ( )  [override], [virtual]
```

Handle the single press event. How in this mode we don't use it.

Implements Mode.

4.19.3.14 handleTempoLed()

```
void ModeInverse::handleTempoLed ( )  [private]
```

This function is used to handle the tempo LED. This is where the tempo LED is updated.

4.19.3.15 loadMidiLearnNotes()

```
void ModeInverse::loadMidiLearnNotes ( )  [private]
```

4.19.3.16 loadState()

```
void ModeInverse::loadState ( )  [private]
```

This function is used to load the state of the ModeInverse object.

4.19.3.17 setPairMode()

```
void ModeInverse::setPairMode (
            int input,
            PairMode mode )  [private]
```

This function is used to set the pair mode for the inputs. This is stored in the pairModes array.

4.19.3.18 setup()

```
void ModeInverse::setup ( )  [override], [virtual]
```

This function is used to setup the current mode object. Setup and teardown methods are meant to be called when Mode selector switches modes. This is where you can put code that should only run once when the mode is switched to.

This is where you'd read the eeprom for the ModeInverse settings. However, we don't have any settings for ModeInverse yet.

Implements Mode.

4.19.3.19 teardown()

```
void ModeInverse::teardown ( )  [override], [virtual]
```

Like the setup method, this function is used to teardown the current mode object.

Implements Mode.

4.19.3.20 update()

```
void ModeInverse::update ( )  [override], [virtual]
```

This function is used to update the current mode object. The update method is meant to be called every loop iteration. This is where you can put code that should run every loop iteration.

Implements Mode.

4.19.4 Member Data Documentation

4.19.4.1 doublePressHandled

```
bool ModeInverse::doublePressHandled = false  [private]
```

4.19.4.2 doubleResetPressHandled

```
bool ModeInverse::doubleResetPressHandled = false  [private]
```

4.19.4.3 encoder

```
Encoder& ModeInverse::encoder  [private]
```

4.19.4.4 gates

```
Gates& ModeInverse::gates  [private]
```

4.19.4.5 inputHandler

```
InputHandler& ModeInverse::inputHandler  [private]
```

4.19.4.6 isInSelection

```
bool ModeInverse::isInSelection = false  [private]
```

4.19.4.7 ledController

```
LEDController& ModeInverse::ledController  [private]
```

4.19.4.8 longResetPressHandled

```
bool ModeInverse::longResetPressHandled = false  [private]
```

4.19.4.9 midi

```
midi::MidiInterface<midi::SerialMIDI<HardwareSerial> >& ModeInverse::midi  [private]
```

4.19.4.10 numLeds

```
int ModeInverse::numLeds = 8  [private]
```

4.19.4.11 pairModes

```
PairMode ModeInverse::pairModes[4]  [private]
```

4.19.4.12 previousInput

```
unsigned char ModeInverse::previousInput = 0  [private]
```

4.19.4.13 previousState

```
bool ModeInverse::previousState[4]  [private]
```

4.19.4.14 resetButton

```
ResetButton& ModeInverse::resetButton  [private]
```

4.19.4.15 selectedInput

```
unsigned char ModeInverse::selectedInput = 0  [private]
```

4.19.4.16 singlePressHandled

```
bool ModeInverse::singlePressHandled = false  [private]
```

4.19.4.17 singleResetPressHandled

```
bool ModeInverse::singleResetPressHandled = false  [private]
```

4.19.4.18 stateManager

```
StateManager& ModeInverse::stateManager  [private]
```

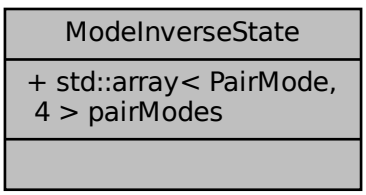
The documentation for this class was generated from the following files:

- include/ModeInverse.h
- src/ModeInverse.cpp

4.20 ModeInverseState Struct Reference

```
#include <AppState.h>
```

Collaboration diagram for ModeInverseState:



Public Attributes

- std::array< PairMode, 4 > pairModes

4.20.1 Member Data Documentation

4.20.1.1 pairModes

```
std::array<PairMode, 4> ModeInverseState::pairModes
```

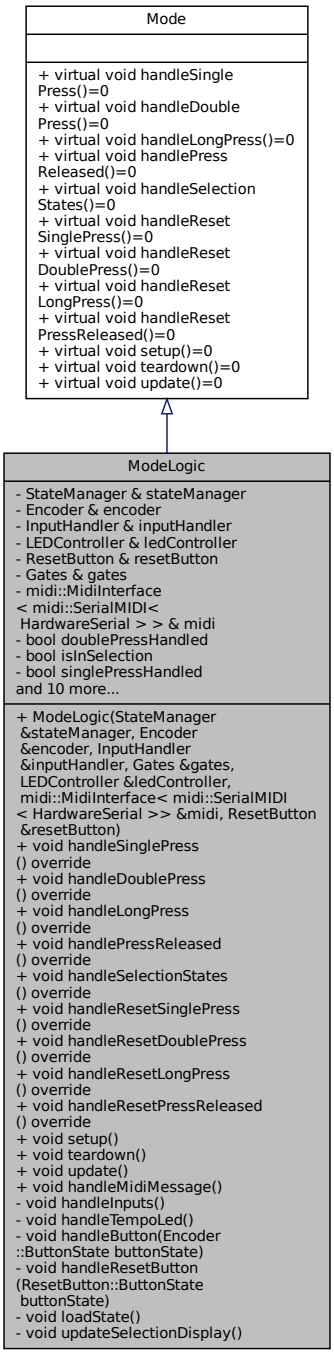
The documentation for this struct was generated from the following file:

- include/AppState.h

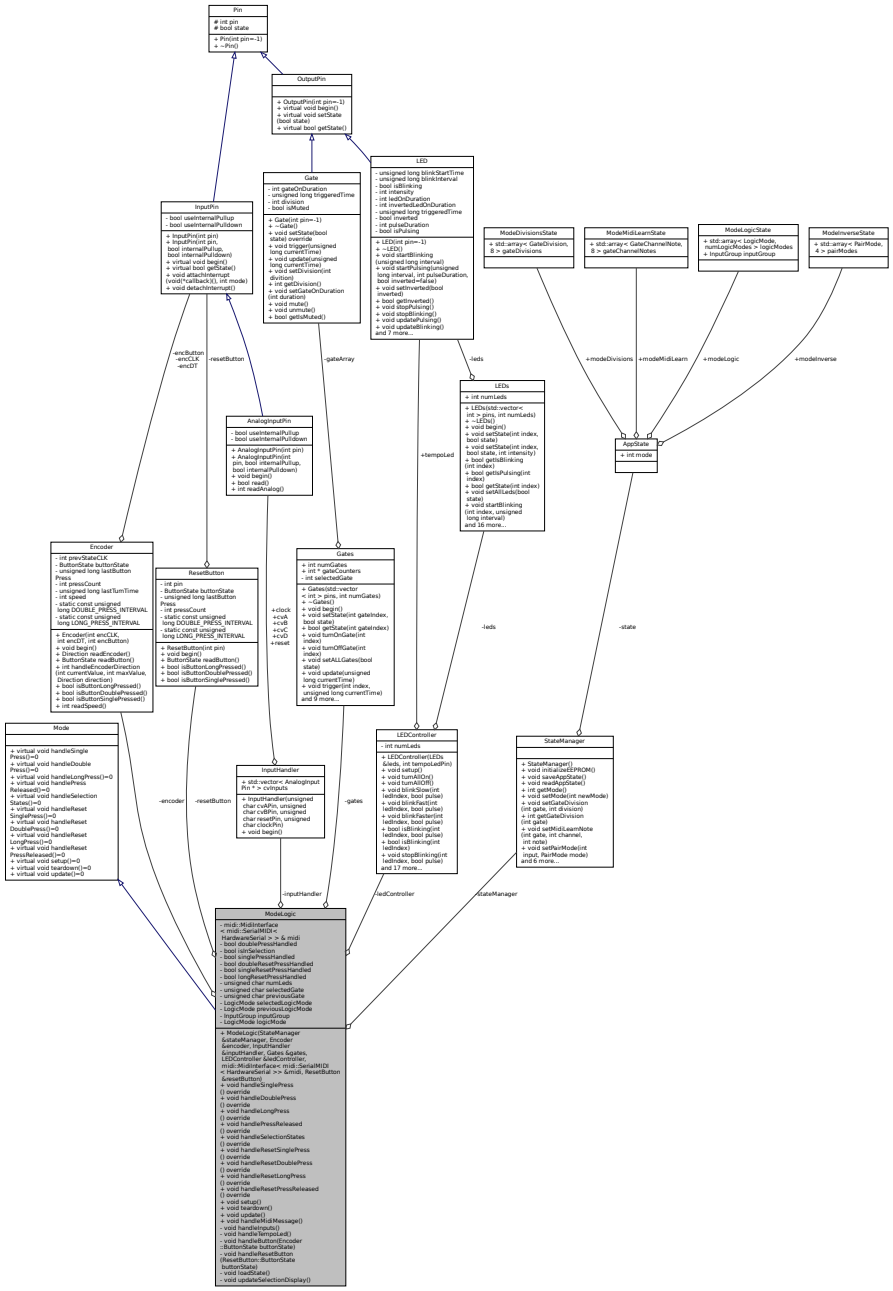
4.21 ModeLogic Class Reference

```
#include <ModeLogic.h>
```

Inheritance diagram for ModeLogic:



Collaboration diagram for ModeLogic:



Public Member Functions

- ModeLogic (StateManager &stateManager, Encoder &encoder, InputHandler &inputHandler, Gates &gates, LEDController &ledController, midi::MidiInterface< midi::SerialMIDI< HardwareSerial >> &midi, ResetButton &resetButton)

  Construct a new Mode Inverse:: Mode Inverse object.

- void handleSinglePress () override

  Handle the single press event. How in this mode we don't use it.

- void handleDoublePress () override

  This function is used to handle the double press event. How in this mode we don't use it.

- void handleLongPress () override

  This function is used to handle the long press event. How in this mode we don't use it.
- void handlePressReleased () override

  This function is used to handle the press released event. How in this mode we don't use it.
- void handleSelectionStates () override

  This function is used to handle the encoder rotation event.
- void handleResetSinglePress () override

  This function is used to handle the reset single press event. In this mode, we use it to mute/unmute the selected input pairs.
- void handleResetDoublePress () override

  This function is used to handle the reset double press event. How in this mode we don't use it.
- void handleResetLongPress () override

  This function is used to handle the reset long press event. How in this mode we don't use it.
- void handleResetPressReleased () override

  This function is used to handle the reset press released event. How in this mode we don't use it.
- void setup ()

  This function is used to setup the current mode object. Setup and teardown methods are meant to be called when Mode selector switches modes. This is where you can put code that should only run once when the mode is switched to.
- void teardown ()

  Like the setup method, this function is used to teardown the current mode object.
- void update ()

  This function is used to update the current mode object. The update method is meant to be called every loop iteration. This is where you can put code that should run every loop iteration.
- void handleMidiMessage ()

  This function is used to handle the MIDI messages. This is where the MIDI messages are handled. In this mode we only use it to forward messages to the MIDI output. Basically, a soft MIDI thru.

Private Member Functions

- void handleInputs ()

  This function is used to handle the inputs. This is where the magic happens.
- void handleTempoLed ()

  This function is used to handle the tempo LED. This is where the tempo LED is updated.
- void handleButton (Encoder::ButtonState buttonState)

  This block of code is used to handle button presses. It is called by the update method.
- void handleResetButton (ResetButton::ButtonState buttonState)

  This block of code is used to handle reset button presses. It is called by the update method.
- void loadState ()

  This function is used to load the state of the ModeLogic object.
- void updateSelectionDisplay ()

  Update the selection display.

Private Attributes

- StateManager & stateManager
- Encoder & encoder
- InputHandler & inputHandler
- LEDController & ledController
- ResetButton & resetButton
- Gates & gates

- midi::MidiInterface< midi::SerialMIDI< HardwareSerial > > & midi
- bool doublePressHandled = false
- bool isInSelection = false
- bool singlePressHandled = false
- bool doubleResetPressHandled = false
- bool singleResetPressHandled = false
- bool longResetPressHandled = false
- unsigned char numLeds = 8
- unsigned char selectedGate = 0
- unsigned char previousGate = 0
- LogicMode selectedLogicMode = AND
- LogicMode previousLogicMode = AND
- InputGroup inputGroup = GROUP_ALL
- LogicMode logicMode [8] = {AND, AND, AND, AND, AND, AND, AND, AND}

4.21.1 Constructor & Destructor Documentation

4.21.1.1 ModeLogic()

```
ModeLogic::ModeLogic (
            StateManager & stateManager,
            Encoder & encoder,
            InputHandler & inputHandler,
            Gates & gates,
            LEDController & ledController,
            midi::MidiInterface< midi::SerialMIDI< HardwareSerial >> & midi,
            ResetButton & resetButton )
```

Construct a new Mode Inverse:: Mode Inverse object.

Parameters

stateManager	
encoder	
inputHandler	
gates	
ledController	
midi	
resetButton	

4.21.2 Member Function Documentation

4.21.2.1 handleButton()

```
void ModeLogic::handleButton (
            Encoder::ButtonState buttonState )  [private]
```

This block of code is used to handle button presses. It is called by the update method.

Parameters

buttonState	

4.21.2.2 handleDoublePress()

```
void ModeLogic::handleDoublePress ( )  [override], [virtual]
```

This function is used to handle the double press event. How in this mode we don't use it.

Implements Mode.

4.21.2.3 handleInputs()

```
void ModeLogic::handleInputs ( )  [private]
```

This function is used to handle the inputs. This is where the magic happens.

4.21.2.4 handleLongPress()

```
void ModeLogic::handleLongPress ( )  [override], [virtual]
```

This function is used to handle the long press event. How in this mode we don't use it.

Implements Mode.

4.21.2.5 handleMidiMessage()

```
void ModeLogic::handleMidiMessage ( )
```

This function is used to handle the MIDI messages. This is where the MIDI messages are handled. In this mode we only use it to forward messages to the MIDI output. Basically, a soft MIDI thru.

NOTE: If you need more functionality, you will need to implement callback functions. However, those will need to be static functions. This is because the MIDI library requires static functions. Just like the handleNoteOn and handleNoteOff functions in the ModeMidiLearn class. Remember that you'll need to create an instance of this class if you do that. You can use ModeMidiLearn as a reference.

4.21.2.6 handlePressReleased()

```
void ModeLogic::handlePressReleased ( )  [override], [virtual]
```

This function is used to handle the press released event. How in this mode we don't use it.

Implements Mode.

4.21.2.7 handleResetButton()

```
void ModeLogic::handleResetButton (
            ResetButton::ButtonState buttonState )  [private]
```

This block of code is used to handle reset button presses. It is called by the update method.

Parameters

buttonState	

4.21.2.8 handleResetDoublePress()

```
void ModeLogic::handleResetDoublePress ( )  [override], [virtual]
```

This function is used to handle the reset double press event. How in this mode we don't use it.

Implements Mode.

4.21.2.9 handleResetLongPress()

```
void ModeLogic::handleResetLongPress ( )  [override], [virtual]
```

This function is used to handle the reset long press event. How in this mode we don't use it.

Implements Mode.

4.21.2.10 handleResetPressReleased()

```
void ModeLogic::handleResetPressReleased ( )  [override], [virtual]
```

This function is used to handle the reset press released event. How in this mode we don't use it.

Implements Mode.

4.21.2.11 handleResetSinglePress()

```
void ModeLogic::handleResetSinglePress ( )  [override], [virtual]
```

This function is used to handle the reset single press event. In this mode, we use it to mute/unmute the selected input pairs.

Implements Mode.

4.21.2.12 handleSelectionStates()

```
void ModeLogic::handleSelectionStates ( )  [override], [virtual]
```

This function is used to handle the encoder rotation event.

Implements Mode.

4.21.2.13 handleSinglePress()

```
void ModeLogic::handleSinglePress ( )  [override], [virtual]
```

Handle the single press event. How in this mode we don't use it.

Implements Mode.

4.21.2.14 handleTempoLed()

```
void ModeLogic::handleTempoLed ( )  [private]
```

This function is used to handle the tempo LED. This is where the tempo LED is updated.

4.21.2.15 loadState()

```
void ModeLogic::loadState ( )  [private]
```

This function is used to load the state of the ModeLogic object.

4.21.2.16 setup()

```
void ModeLogic::setup ( )  [virtual]
```

This function is used to setup the current mode object. Setup and teardown methods are meant to be called when Mode selector switches modes. This is where you can put code that should only run once when the mode is switched to.

This is where you'd read the eeprom for the ModeLogic settings. However, we don't have any settings for ModeLogic yet.

Implements Mode.

4.21.2.17 teardown()

```
void ModeLogic::teardown ( )  [virtual]
```

Like the setup method, this function is used to teardown the current mode object.

Implements Mode.

4.21.2.18 update()

```
void ModeLogic::update ( )  [virtual]
```

This function is used to update the current mode object. The update method is meant to be called every loop iteration. This is where you can put code that should run every loop iteration.

Implements Mode.

4.21.2.19 updateSelectionDisplay()

```
void ModeLogic::updateSelectionDisplay ( )  [private]
```

Update the selection display.

4.21.3 Member Data Documentation

4.21.3.1 doublePressHandled

```
bool ModeLogic::doublePressHandled = false  [private]
```

4.21.3.2 doubleResetPressHandled

```
bool ModeLogic::doubleResetPressHandled = false  [private]
```

4.21.3.3 encoder

```
Encoder& ModeLogic::encoder  [private]
```

4.21.3.4 gates

```
Gates& ModeLogic::gates  [private]
```

4.21.3.5 inputGroup

```
InputGroup ModeLogic::inputGroup = GROUP_ALL  [private]
```

4.21.3.6 inputHandler

```
InputHandler& ModeLogic::inputHandler  [private]
```

4.21.3.7 isInSelection

```
bool ModeLogic::isInSelection = false  [private]
```

4.21.3.8 ledController

```
LEDController& ModeLogic::ledController  [private]
```

4.21.3.9 logicMode

```
LogicMode ModeLogic::logicMode[8] = {AND, AND, AND, AND, AND, AND, AND, AND}  [private]
```

4.21.3.10 longResetPressHandled

```
bool ModeLogic::longResetPressHandled = false  [private]
```

4.21.3.11 midi

```
midi::MidiInterface<midi::SerialMIDI<HardwareSerial> >& ModeLogic::midi  [private]
```

4.21.3.12 numLeds

```
unsigned char ModeLogic::numLeds = 8  [private]
```

4.21.3.13 previousGate

```
unsigned char ModeLogic::previousGate = 0  [private]
```

4.21.3.14 previousLogicMode

```
LogicMode ModeLogic::previousLogicMode = AND  [private]
```

4.21.3.15 resetButton

```
ResetButton& ModeLogic::resetButton  [private]
```

4.21.3.16 selectedGate

```
unsigned char ModeLogic::selectedGate = 0  [private]
```

4.21.3.17 selectedLogicMode

```
LogicMode ModeLogic::selectedLogicMode = AND  [private]
```

4.21.3.18 singlePressHandled

```
bool ModeLogic::singlePressHandled = false  [private]
```

4.21.3.19 singleResetPressHandled

```
bool ModeLogic::singleResetPressHandled = false  [private]
```

4.21.3.20 stateManager

```
StateManager& ModeLogic::stateManager  [private]
```

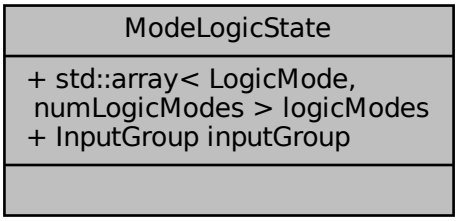
The documentation for this class was generated from the following files:

- include/ModeLogic.h
- src/ModeLogic.cpp

4.22 ModeLogicState Struct Reference

```
#include <AppState.h>
```

Collaboration diagram for ModeLogicState:



Public Attributes

- std::array< LogicMode, numLogicModes > logicModes
- InputGroup inputGroup

4.22.1 Member Data Documentation

4.22.1.1 inputGroup

```
InputGroup ModeLogicState::inputGroup
```

4.22.1.2 logicModes

```
std::array<LogicMode, numLogicModes> ModeLogicState::logicModes
```

The documentation for this struct was generated from the following file:

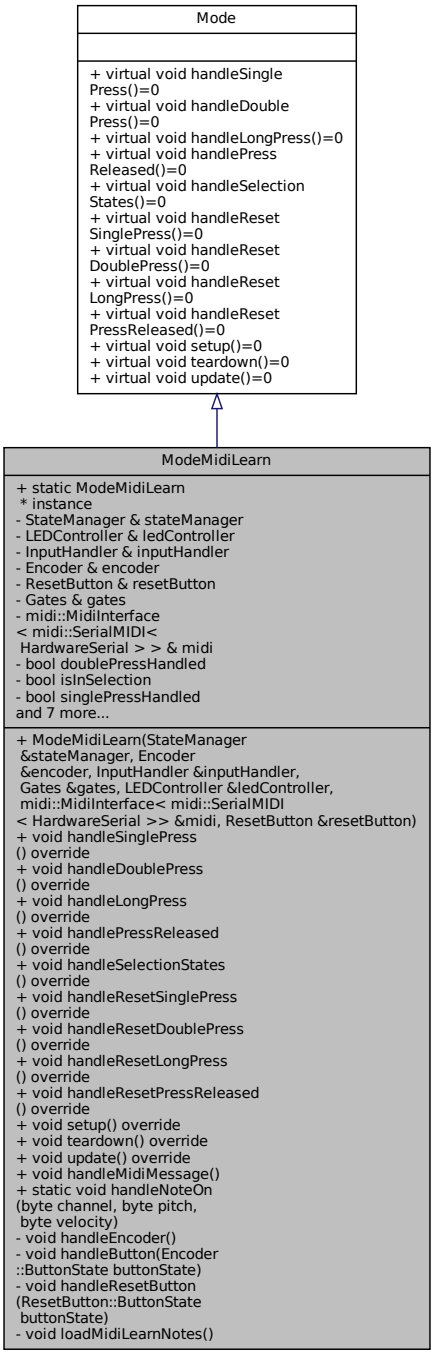
- include/AppState.h

4.23 ModeMidiLearn Class Reference

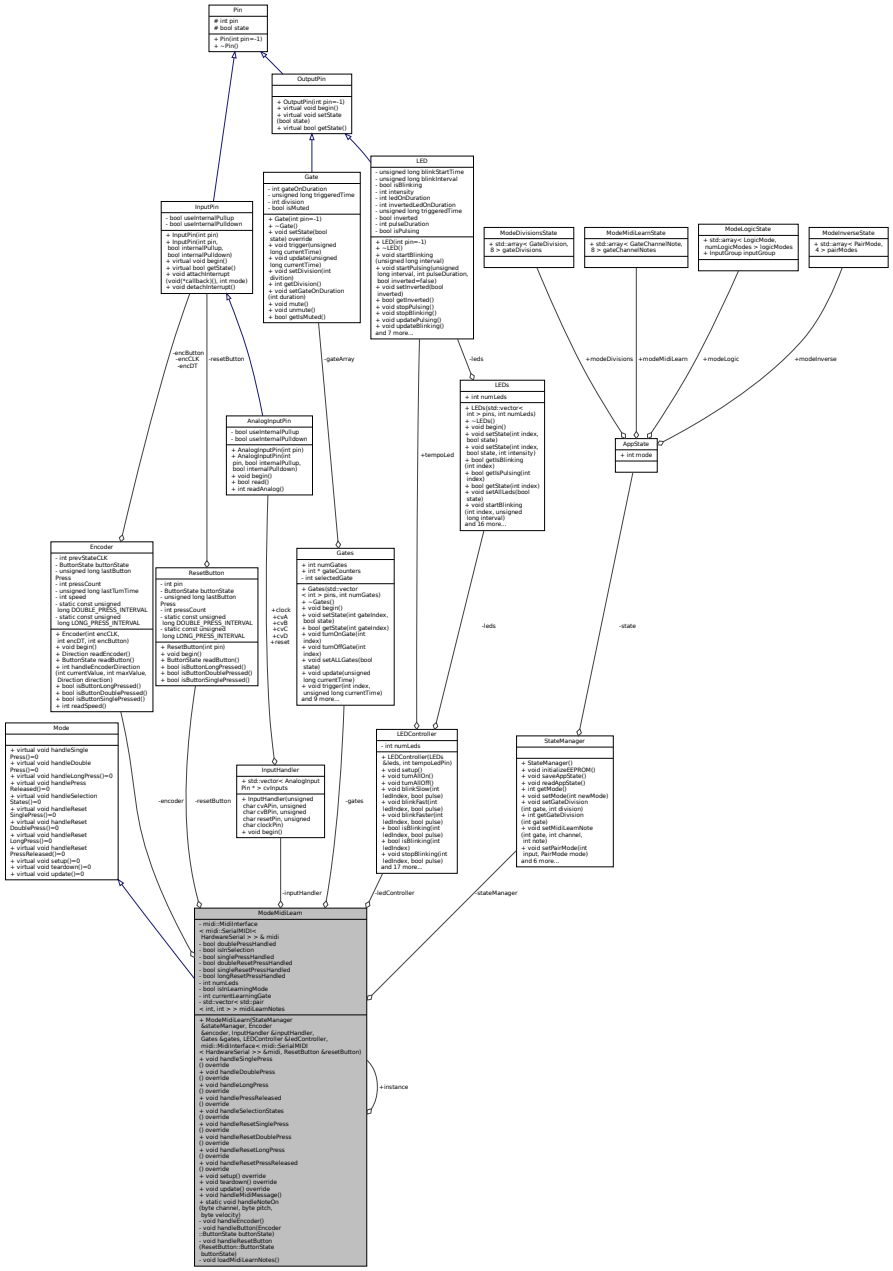
This is a MIDI to Trigger class for Note On but it only cares about the channel number.

```
#include <ModeMidiLearn.h>
```

Inheritance diagram for ModeMidiLearn:



Collaboration diagram for ModeMidiLearn:



Public Member Functions

- ModeMidiLearn (StateManager &stateManager, Encoder &encoder, InputHandler &inputHandler, Gates &gates, LEDController &ledController, midi::MidiInterface< midi::SerialMIDI< HardwareSerial >> &midi, ResetButton &resetButton)
- void handleSinglePress () override

  Handle the single press event. How in this mode we don't use it.
- void handleDoublePress () override

  This function is used to handle the double press event. How in this mode we don't use it.
- void handleLongPress () override

  This function is used to handle the long press event. How in this mode we don't use it.

- void handlePressReleased () override

  This function is used to handle the press released event. How in this mode we don't use it.
- void handleSelectionStates () override

  This function is used to handle the encoder rotation event. How in this mode we don't use it.
- void handleResetSinglePress () override

  This function is used to handle the reset single press event. How in this mode we don't use it.
- void handleResetDoublePress () override

  This function is used to handle the reset double press event. How in this mode we don't use it.
- void handleResetLongPress () override

  This function is used to handle the reset long press event. How in this mode we don't use it.
- void handleResetPressReleased () override

  This function is used to handle the reset press released event. How in this mode we don't use it.
- void setup () override

  This function is used to setup the current mode object. Setup and teardown methods are meant to be called when Mode selector switches modes. This is where you can put code that should only run once when the mode is switched to.
- void teardown () override

  Like the setup method, this function is used to teardown the current mode object.
- void update () override

  This function is used to update the current mode object. The update method is meant to be called every loop iteration. This is where you can put code that should run every loop iteration.
- void handleMidiMessage ()

  Handle MIDI messages. This function is called by the update method.

Static Public Member Functions

- static void handleNoteOn (byte channel, byte pitch, byte velocity)

  Static callback function for handling MIDI Note On messages.

Static Public Attributes

- static ModeMidiLearn * instance = nullptr

  This is the instance of the ModeMidiLearn class. It is used to access the ModeMidiLearn class from the static MIDI callback function.

Private Member Functions

- void handleEncoder ()
- void handleButton (Encoder::ButtonState buttonState)

  This block of code is used to handle button presses. It is called by the update method.
- void handleResetButton (ResetButton::ButtonState buttonState)

  This block of code is used to handle reset button presses. It is called by the update method.
- void loadMidiLearnNotes ()

  Update midiLearnNotes from stateManager.

Private Attributes

- StateManager & stateManager
- LEDController & ledController
- InputHandler & inputHandler
- Encoder & encoder
- ResetButton & resetButton
- Gates & gates
- midi::MidiInterface< midi::SerialMIDI< HardwareSerial > > & midi
- bool doublePressHandled = false
- bool isInSelection = false
- bool singlePressHandled = false
- bool doubleResetPressHandled = false
- bool singleResetPressHandled = false
- bool longResetPressHandled = false
- int numLeds = 8
- bool isInLearningMode = false
- int currentLearningGate = 0
- std::vector< std::pair< int, int > > midiLearnNotes

4.23.1 Detailed Description

This is a MIDI to Trigger class for Note On but it only cares about the channel number.

4.23.2 Constructor & Destructor Documentation

4.23.2.1 ModeMidiLearn()

```
ModeMidiLearn::ModeMidiLearn (
            StateManager & stateManager,
            Encoder & encoder,
            InputHandler & inputHandler,
            Gates & gates,
            LEDController & ledController,
            midi::MidiInterface< midi::SerialMIDI< HardwareSerial >> & midi,
            ResetButton & resetButton )
```

4.23.3 Member Function Documentation

4.23.3.1 handleButton()

```
void ModeMidiLearn::handleButton (
            Encoder::ButtonState buttonState )  [private]
```

This block of code is used to handle button presses. It is called by the update method.

Parameters

buttonState	

4.23.3.2 handleDoublePress()

```
void ModeMidiLearn::handleDoublePress ( )  [override], [virtual]
```

This function is used to handle the double press event. How in this mode we don't use it.

Implements Mode.

4.23.3.3 handleEncoder()

```
void ModeMidiLearn::handleEncoder ( )  [private]
```

4.23.3.4 handleLongPress()

```
void ModeMidiLearn::handleLongPress ( )  [override], [virtual]
```

This function is used to handle the long press event. How in this mode we don't use it.

Implements Mode.

4.23.3.5 handleMidiMessage()

```
void ModeMidiLearn::handleMidiMessage ( )
```

Handle MIDI messages. This function is called by the update method.

4.23.3.6 handleNoteOn()

```
void ModeMidiLearn::handleNoteOn (
            byte channel,
            byte pitch,
            byte velocity )  [static]
```

Static callback function for handling MIDI Note On messages.

Parameters

channel	
pitch	
velocity	

4.23.3.7 handlePressReleased()

```
void ModeMidiLearn::handlePressReleased ( )  [override], [virtual]
```

This function is used to handle the press released event. How in this mode we don't use it.

Implements Mode.

4.23.3.8 handleResetButton()

```
void ModeMidiLearn::handleResetButton (
            ResetButton::ButtonState buttonState )  [private]
```

This block of code is used to handle reset button presses. It is called by the update method.

Parameters

buttonState	

4.23.3.9 handleResetDoublePress()

```
void ModeMidiLearn::handleResetDoublePress ( )  [override], [virtual]
```

This function is used to handle the reset double press event. How in this mode we don't use it.

Implements Mode.

4.23.3.10 handleResetLongPress()

```
void ModeMidiLearn::handleResetLongPress ( )  [override], [virtual]
```

This function is used to handle the reset long press event. How in this mode we don't use it.

Implements Mode.

4.23.3.11 handleResetPressReleased()

```
void ModeMidiLearn::handleResetPressReleased ( )  [override], [virtual]
```

This function is used to handle the reset press released event. How in this mode we don't use it.

Implements Mode.

4.23.3.12 handleResetSinglePress()

```
void ModeMidiLearn::handleResetSinglePress ( )  [override], [virtual]
```

This function is used to handle the reset single press event. How in this mode we don't use it.

Implements Mode.

4.23.3.13 handleSelectionStates()

```
void ModeMidiLearn::handleSelectionStates ( )  [override], [virtual]
```

This function is used to handle the encoder rotation event. How in this mode we don't use it.

Implements Mode.

4.23.3.14 handleSinglePress()

```
void ModeMidiLearn::handleSinglePress ( )  [override], [virtual]
```

Handle the single press event. How in this mode we don't use it.

Implements Mode.

4.23.3.15 loadMidiLearnNotes()

```
void ModeMidiLearn::loadMidiLearnNotes ( )  [private]
```

Update midiLearnNotes from stateManager.

4.23.3.16 setup()

```
void ModeMidiLearn::setup ( )  [override], [virtual]
```

This function is used to setup the current mode object. Setup and teardown methods are meant to be called when Mode selector switches modes. This is where you can put code that should only run once when the mode is switched to.

This is where you'd read the eeprom for the ModeMidiLearn settings. However, we don't have any settings for ModeMidiLearn yet.

Implements Mode.

4.23.3.17 teardown()

```
void ModeMidiLearn::teardown ( )  [override], [virtual]
```

Like the setup method, this function is used to teardown the current mode object.

Implements Mode.

4.23.3.18 update()

```
void ModeMidiLearn::update ( )  [override], [virtual]
```

This function is used to update the current mode object. The update method is meant to be called every loop iteration. This is where you can put code that should run every loop iteration.

Implements Mode.

4.23.4 Member Data Documentation

4.23.4.1 currentLearningGate

```
int ModeMidiLearn::currentLearningGate = 0  [private]
```

4.23.4.2 doublePressHandled

```
bool ModeMidiLearn::doublePressHandled = false  [private]
```

4.23.4.3 doubleResetPressHandled

```
bool ModeMidiLearn::doubleResetPressHandled = false  [private]
```

4.23.4.4 encoder

```
Encoder& ModeMidiLearn::encoder  [private]
```

4.23.4.5 gates

```
Gates& ModeMidiLearn::gates  [private]
```

4.23.4.6 inputHandler

```
InputHandler& ModeMidiLearn::inputHandler  [private]
```

4.23.4.7 instance

```
ModeMidiLearn * ModeMidiLearn::instance = nullptr  [static]
```

This is the instance of the ModeMidiLearn class. It is used to access the ModeMidiLearn class from the static MIDI callback function.

4.23.4.8 isInLearningMode

```
bool ModeMidiLearn::isInLearningMode = false  [private]
```

4.23.4.9 isInSelection

```
bool ModeMidiLearn::isInSelection = false  [private]
```

4.23.4.10 ledController

```
LEDController& ModeMidiLearn::ledController  [private]
```

4.23.4.11 longResetPressHandled

```
bool ModeMidiLearn::longResetPressHandled = false  [private]
```

4.23.4.12 midi

```
midi::MidiInterface<midi::SerialMIDI<HardwareSerial> >& ModeMidiLearn::midi  [private]
```

4.23.4.13 midiLearnNotes

```
std::vector<std::pair<int, int> > ModeMidiLearn::midiLearnNotes  [private]
```

4.23.4.14 numLeds

```
int ModeMidiLearn::numLeds = 8  [private]
```

4.23.4.15 resetButton

```
ResetButton& ModeMidiLearn::resetButton  [private]
```

4.23.4.16 singlePressHandled

```
bool ModeMidiLearn::singlePressHandled = false  [private]
```

4.23.4.17 singleResetPressHandled

```
bool ModeMidiLearn::singleResetPressHandled = false  [private]
```

4.23.4.18 stateManager

```
StateManager& ModeMidiLearn::stateManager  [private]
```

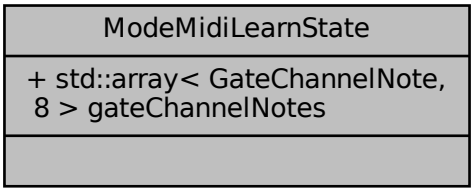
The documentation for this class was generated from the following files:

- include/ModeMidiLearn.h
- src/ModeMidiLearn.cpp

4.24 ModeMidiLearnState Struct Reference

```
#include <AppState.h>
```

Collaboration diagram for ModeMidiLearnState:



Public Attributes

- std::array< GateChannelNote, 8 > gateChannelNotes

4.24.1 Member Data Documentation

4.24.1.1 gateChannelNotes

```
std::array<GateChannelNote, 8> ModeMidiLearnState::gateChannelNotes
```

The documentation for this struct was generated from the following file:

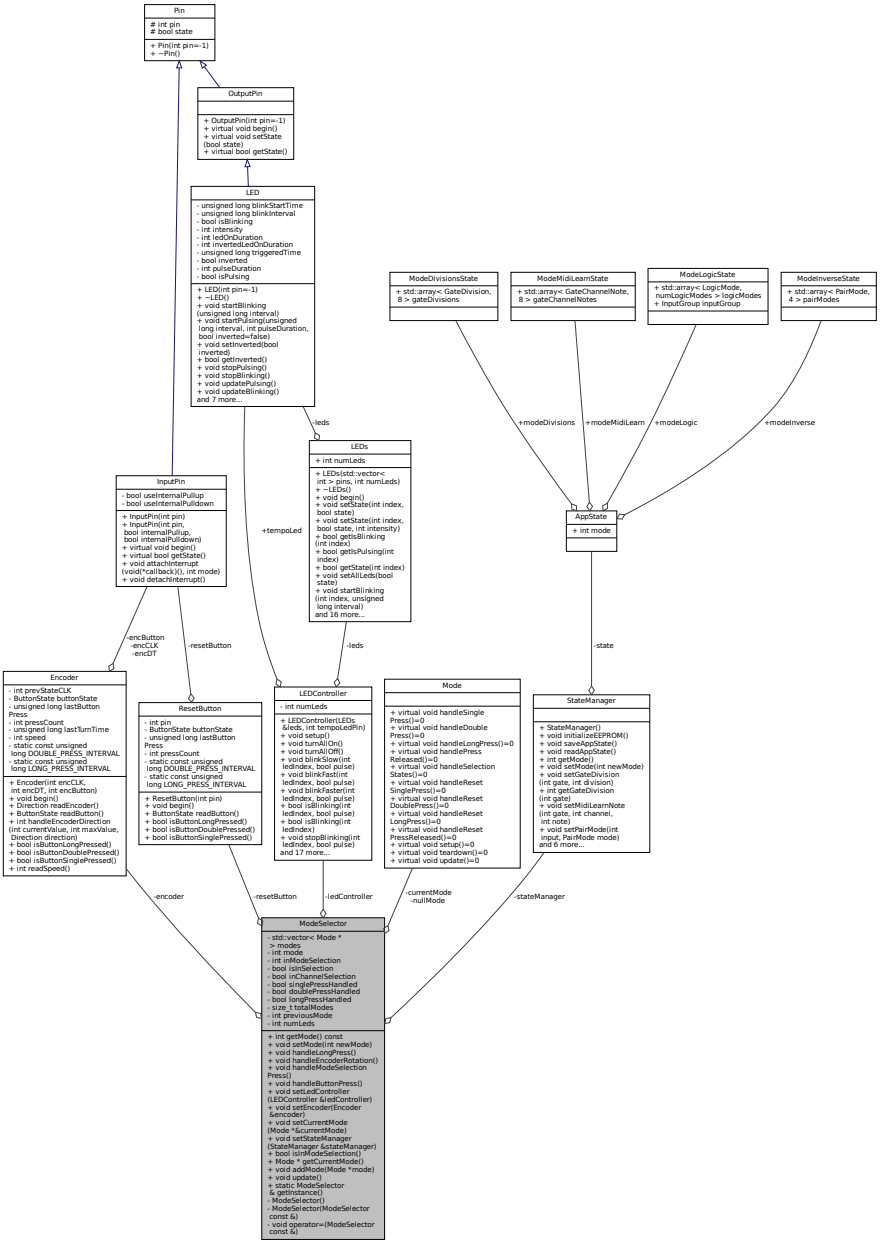
- include/AppState.h

4.25 ModeSelector Class Reference

Mode Selector Singleton. This class is responsible for managing the different modes of the device. It provides methods to add modes, set the current mode, and handle mode selection.

`#include <ModeSelector.h>`

Collaboration diagram for ModeSelector:



Public Member Functions

- int getMode () const

  Returns the current mode as an integer.

- void setMode (int newMode)

  This function is used to set the current mode. IMPORTANT: The order of the modes in the vector matters. The index of the mode in the vector is the mode number.
- void handleLongPress ()

  Handles the long press of the encoder button. This is how the user enters the mode selection state. IMPORTANT: It is best to avoid using long press for other purposes. I've considered removing long press from the individual modes, if this becomes a problem we can remove it from the modes and make sure it is only used for mode selection.
- void handleEncoderRotation ()

  This function is used to handle the encoder rotation. It reads the encoder and handles the rotation. The encoder is used to cycle through the different modes displaying the corresponding LED.
- void handleModeSelectionPress ()

  This function is used to handle the mode selection press. It clars the LEDs and gets them ready for the next mode selection.
- void handleButtonPress ()

  This function is used to handle the button press. It reads the encoder and handles button presses.
- void setLedController (LEDController &ledController)

  This function is used to set the LED controller for the ModeSelector. It is meant to be called by the main sketch to set the LED controller for the ModeSelector in the setup() function.
- void setEncoder (Encoder &encoder)

  This function is used to set the encoder for the ModeSelector. It is meant to be called by the main sketch to set the encoder for the ModeSelector in the setup() function.
- void setCurrentMode (Mode *&currentMode)

  This function is used to set the current mode.
- void setStateManager (StateManager &stateManager)

  Configures the StateManager for the ModeSelector which is used to save the current mode to EEPROM.
- bool isInModeSelection ()

  Helper function to check if the mode selection state is active.
- Mode * getCurrentMode ()

  This function is used to get the current mode.
- void addMode (Mode *mode)

  This function is used to add a mode to the ModeSelector. This is why the order of the modes in the vector matters.
- void update ()

  This function is used to update the mode selector. It is intended to be called in the loop() function of the main sketch.

Static Public Member Functions

- static ModeSelector & getInstance ()

  Sets the instance of the ModeSelector class.

Private Member Functions

- ModeSelector ()

  Constructor is private.
- ModeSelector (ModeSelector const &)
- void operator= (ModeSelector const &)

Private Attributes

- std::vector< Mode * > modes
- Mode * nullMode = nullptr
- Mode *& currentMode
- int mode
- int inModeSelection = false
- LEDController * ledController
- Encoder * encoder
- StateManager * stateManager
- ResetButton * resetButton
- bool isInSelection
- bool inChannelSelection
- bool singlePressHandled
- bool doublePressHandled
- bool longPressHandled
- size_t totalModes = modes.size()
- int previousMode = -1
- int numLeds

4.25.1 Detailed Description

Mode Selector Singleton. This class is responsible for managing the different modes of the device. It provides methods to add modes, set the current mode, and handle mode selection.

4.25.2 Constructor & Destructor Documentation

4.25.2.1 ModeSelector() [1/2]

```
ModeSelector::ModeSelector ( )  [private]
```

Constructor is private.

Construct a new Mode Selector:: Mode Selector object.

4.25.2.2 ModeSelector() [2/2]

```
ModeSelector::ModeSelector (
            ModeSelector const &  )  [private]
```

4.25.3 Member Function Documentation

4.25.3.1 addMode()

```
void ModeSelector::addMode (
            Mode * mode )
```

This function is used to add a mode to the ModeSelector. This is why the order of the modes in the vector matters.

Parameters

mode	

4.25.3.2 getCurrentMode()

```
Mode * ModeSelector::getCurrentMode ( )
```

This function is used to get the current mode.

Returns

Mode*

4.25.3.3 getInstance()

```
ModeSelector & ModeSelector::getInstance ( )  [static]
```

Sets the instance of the ModeSelector class.

Returns

ModeSelector&

4.25.3.4 getMode()

```
int ModeSelector::getMode ( ) const
```

Returns the current mode as an integer.

Returns

int

4.25.3.5 handleButtonPress()

```
void ModeSelector::handleButtonPress ( )
```

This function is used to handle the button press. It reads the encoder and handles button presses.

4.25.3.6 handleEncoderRotation()

```
void ModeSelector::handleEncoderRotation ( )
```

This function is used to handle the encoder rotation. It reads the encoder and handles the rotation. The encoder is used to cycle through the different modes displaying the corresponding LED.

4.25.3.7 handleLongPress()

```
void ModeSelector::handleLongPress ( )
```

Handles the long press of the encoder button. This is how the user enters the mode selection state. IMPORTANT: It is best to avoid using long press for other purposes. I've considered removing long press from the individual modes, if this becomes a problem we can remove it from the modes and make sure it is only used for mode selection.

4.25.3.8 handleModeSelectionPress()

```
void ModeSelector::handleModeSelectionPress ( )
```

This function is used to handle the mode selection press. It clars the LEDs and gets them ready for the next mode selection.

4.25.3.9 isInModeSelection()

```
bool ModeSelector::isInModeSelection ( )
```

Helper function to check if the mode selection state is active.

Returns

bool

4.25.3.10 operator=()

```
void ModeSelector::operator= (
            ModeSelector const &  )  [private]
```

4.25.3.11 setCurrentMode()

```
void ModeSelector::setCurrentMode (
            Mode *& currentMode )
```

This function is used to set the current mode.

Parameters

currentMode	

4.25.3.12 setEncoder()

```
void ModeSelector::setEncoder (
            Encoder & encoder )
```

This function is used to set the encoder for the ModeSelector. It is meant to be called by the main sketch to set the encoder for the ModeSelector in the setup() function.

Parameters

encoder	

4.25.3.13 setLedController()

```
void ModeSelector::setLedController (
            LEDController & ledController )
```

This function is used to set the LED controller for the ModeSelector. It is meant to be called by the main sketch to set the LED controller for the ModeSelector in the setup() function.

Parameters

ledController	

4.25.3.14 setMode()

```
void ModeSelector::setMode (
            int newMode )
```

This function is used to set the current mode. IMPORTANT: The order of the modes in the vector matters. The index of the mode in the vector is the mode number.

Parameters

newMode	

4.25.3.15 setStateManager()

```
void ModeSelector::setStateManager (
            StateManager & stateManager )
```

Configures the StateManager for the ModeSelector which is used to save the current mode to EEPROM.

Parameters

stateManager	

4.25.3.16 update()

```
void ModeSelector::update ( )
```

This function is used to update the mode selector. It is intended to be called in the loop() function of the main sketch.

4.25.4 Member Data Documentation

4.25.4.1 currentMode

```
Mode*& ModeSelector::currentMode  [private]
```

4.25.4.2 doublePressHandled

```
bool ModeSelector::doublePressHandled  [private]
```

4.25.4.3 encoder

```
Encoder* ModeSelector::encoder  [private]
```

4.25.4.4 inChannelSelection

```
bool ModeSelector::inChannelSelection  [private]
```

4.25.4.5 inModeSelection

```
int ModeSelector::inModeSelection = false  [private]
```

4.25.4.6 isInSelection

```
bool ModeSelector::isInSelection  [private]
```

4.25.4.7 ledController

```
LEDController* ModeSelector::ledController  [private]
```

4.25.4.8 longPressHandled

```
bool ModeSelector::longPressHandled  [private]
```

4.25.4.9 mode

```
int ModeSelector::mode  [private]
```

4.25.4.10 modes

```
std::vector<Mode*> ModeSelector::modes  [private]
```

4.25.4.11 nullMode

```
Mode* ModeSelector::nullMode = nullptr  [private]
```

4.25.4.12 numLeds

```
int ModeSelector::numLeds  [private]
```

4.25.4.13 previousMode

```
int ModeSelector::previousMode = -1  [private]
```

4.25.4.14 resetButton

```
ResetButton* ModeSelector::resetButton  [private]
```

4.25.4.15 singlePressHandled

```
bool ModeSelector::singlePressHandled  [private]
```

4.25.4.16 stateManager

```
StateManager* ModeSelector::stateManager  [private]
```

4.25.4.17 totalModes

```
size_t ModeSelector::totalModes = modes.size()  [private]
```

The documentation for this class was generated from the following files:

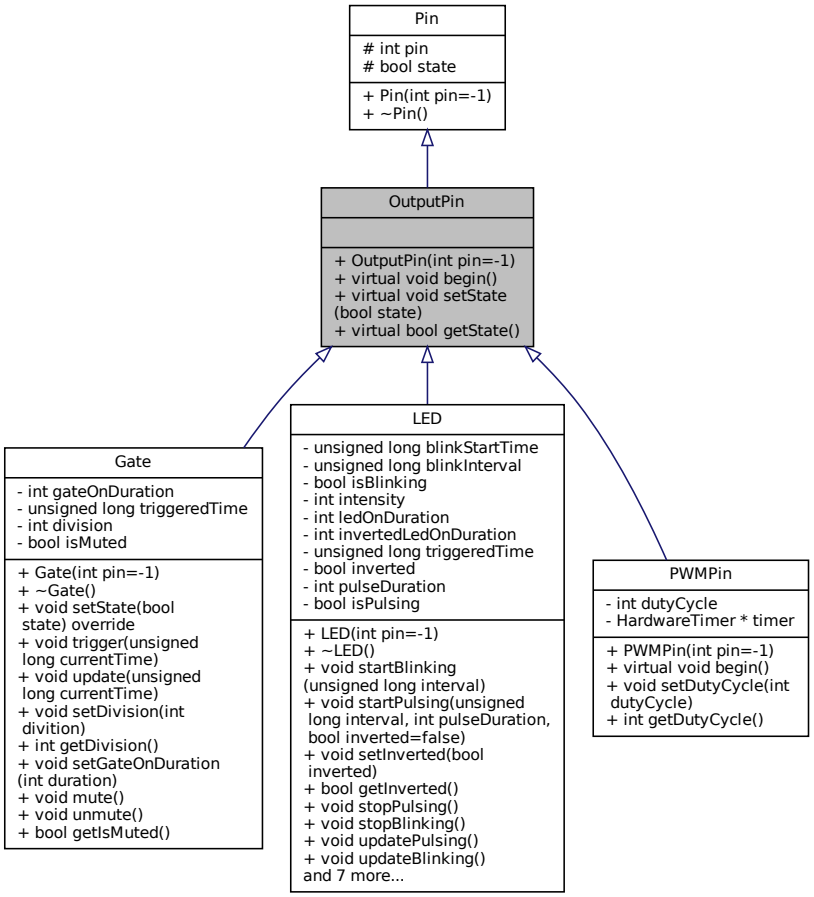
- include/ModeSelector.h
- src/ModeSelector.cpp

4.26 OutputPin Class Reference

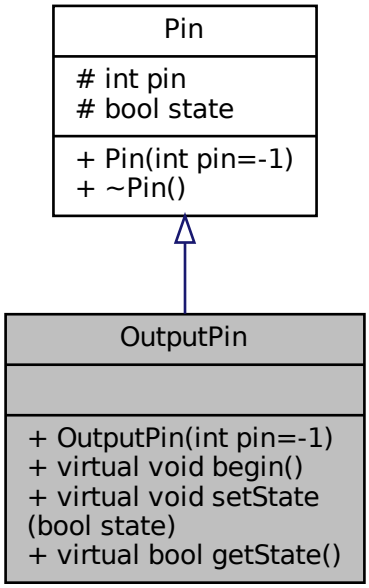
This class represents an output pin on the microcontroller.

`#include <Pin.h>`

Inheritance diagram for OutputPin:



Collaboration diagram for OutputPin:



Public Member Functions

- OutputPin (int pin=-1)

  Construct a new Output Pin:: Output Pin object.
- virtual void begin ()

  This function is used to initialize the output pin. It is intended to be called in the setup() function of the main sketch.
- virtual void setState (bool state)

  This function is used to set the state of the output pin. Possible states are HIGH or LOW.
- virtual bool getState ()

  This function is used to get the state of the output pin.

Additional Inherited Members

4.26.1 Detailed Description

This class represents an output pin on the microcontroller.

4.26.2 Constructor & Destructor Documentation

4.26.2.1 OutputPin()

```
OutputPin::OutputPin (
            int pin = -1 )
```

Construct a new Output Pin:: Output Pin object.

Parameters

pin	

4.26.3 Member Function Documentation

4.26.3.1 begin()

```
void OutputPin::begin ( )  [virtual]
```

This function is used to initialize the output pin. It is intended to be called in the setup() function of the main sketch.

Reimplemented in PWMPin.

4.26.3.2 getState()

```
bool OutputPin::getState ( )  [virtual]
```

This function is used to get the state of the output pin.

Returns

bool

4.26.3.3 setState()

```
void OutputPin::setState (
            bool newState )  [virtual]
```

This function is used to set the state of the output pin. Possible states are HIGH or LOW.

Parameters

newState	

Reimplemented in Gate.

The documentation for this class was generated from the following files:

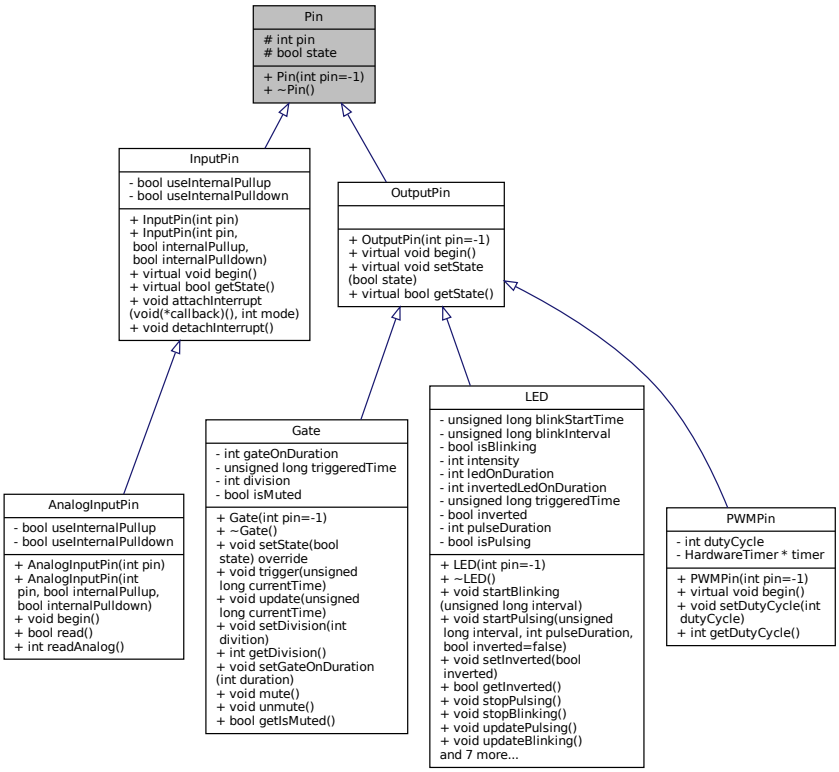
- include/Pin.h
- src/Pin.cpp

4.27 Pin Class Reference

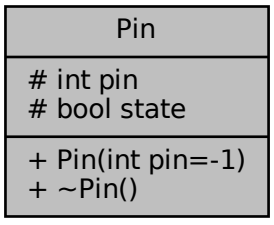
This class represents a pin on the microcontroller.

`#include <Pin.h>`

Inheritance diagram for Pin:



Collaboration diagram for Pin:



Public Member Functions

- Pin (int pin=-1)

  Construct a new Pin:: Pin object, which is the base class for all pin.
- ~Pin ()

  Destroy the Pin:: Pin object.

Protected Attributes

- int pin
- bool state

4.27.1 Detailed Description

This class represents a pin on the microcontroller.

4.27.2 Constructor & Destructor Documentation

4.27.2.1 Pin()

```
Pin::Pin (
            int pin = -1 )
```

Construct a new Pin:: Pin object, which is the base class for all pin.

Parameters

pin	

4.27.2.2 ~Pin()

```
Pin::~Pin ( )
```

Destroy the Pin:: Pin object.

4.27.3 Member Data Documentation

4.27.3.1 pin

```
int Pin::pin  [protected]
```

4.27.3.2 state

```
bool Pin::state  [protected]
```

The documentation for this class was generated from the following files:

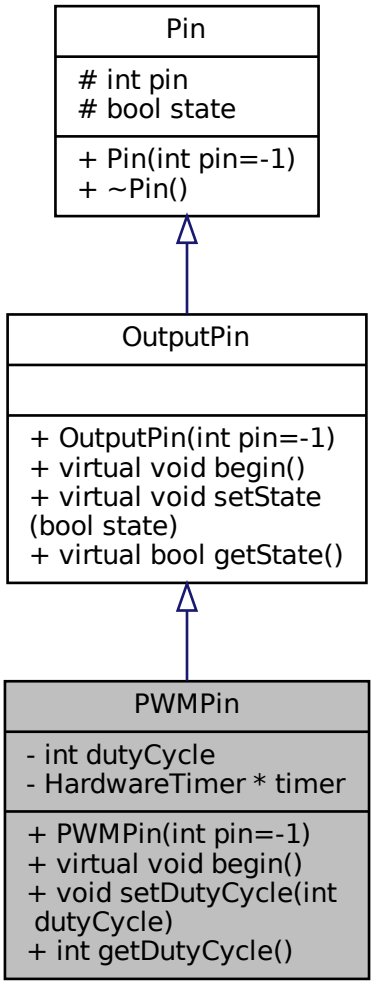
- include/Pin.h
- src/Pin.cpp

4.28 PWMPin Class Reference

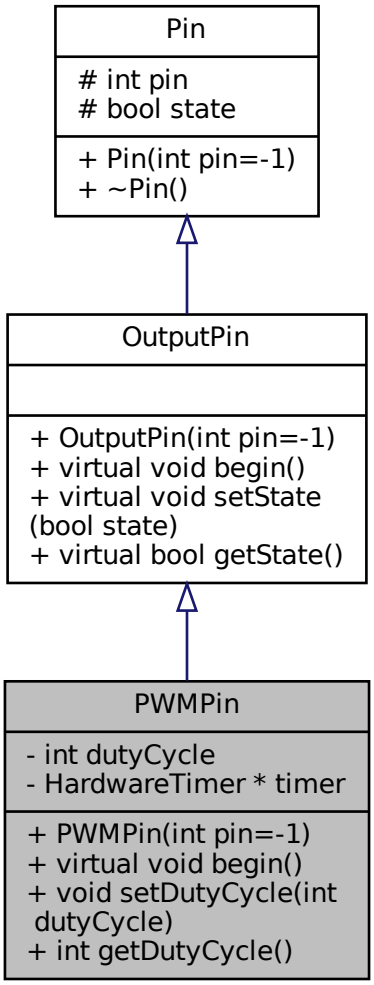
This class represents a PWM output pin on the microcontroller.

```
#include <Pin.h>
```

Inheritance diagram for PWMPin:



Collaboration diagram for PWMPin:



Public Member Functions

- PWMPin (int pin=-1)

  Construct a new PWMPin::PWMPin object. We are using the HardwareTimer library for STM32 boards.
- virtual void begin ()

  This function is used to initialize the PWM pin. It is intended to be called in the setup() function of the main sketch.
- void setDutyCycle (int dutyCycle)

  This function is used to set the duty cycle of the PWM pin.
- int getDutyCycle ()

  This function is used to get the duty cycle of the PWM pin.

Private Attributes

- int dutyCycle
- HardwareTimer * timer

Additional Inherited Members

4.28.1 Detailed Description

This class represents a PWM output pin on the microcontroller.

4.28.2 Constructor & Destructor Documentation

4.28.2.1 PWMPin()

```
PWMPin::PWMPin (
            int pin = -1 )
```

Construct a new PWMPin::PWMPin object. We are using the HardwareTimer library for STM32 boards.

Parameters

pin	

4.28.3 Member Function Documentation

4.28.3.1 begin()

```
void PWMPin::begin ( )  [virtual]
```

This function is used to initialize the PWM pin. It is intended to be called in the setup() function of the main sketch.

Reimplemented from OutputPin.

4.28.3.2 getDutyCycle()

```
int PWMPin::getDutyCycle ( )
```

This function is used to get the duty cycle of the PWM pin.

Returns

int

4.28.3.3 setDutyCycle()

```
void PWMPin::setDutyCycle (
            int dutyCycle )
```

This function is used to set the duty cycle of the PWM pin.

Parameters

dutyCycle	

4.28.4 Member Data Documentation

4.28.4.1 dutyCycle

```
int PWMPin::dutyCycle  [private]
```

4.28.4.2 timer

```
HardwareTimer* PWMPin::timer  [private]
```

The documentation for this class was generated from the following files:

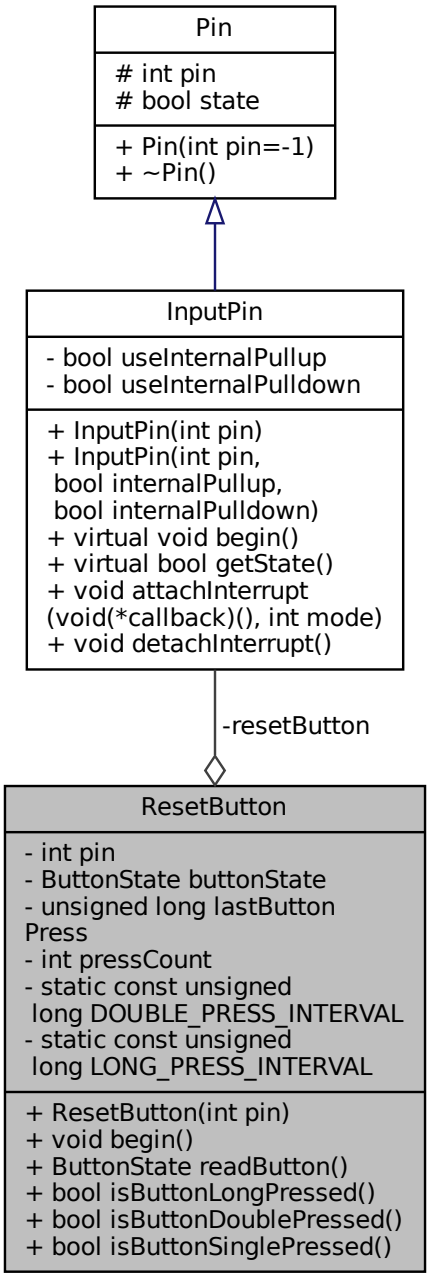
- include/Pin.h
- src/Pin.cpp

4.29 ResetButton Class Reference

This class is used to read the reset button input.

```
#include <ResetButton.h>
```

Collaboration diagram for ResetButton:



Public Types

- enum ButtonState { OPEN , PRESSED }

Public Member Functions

- ResetButton (int pin)

Construct a new Reset Button:: Reset Button object.

- void begin ()

  This function is used to initialize the reset button. This is intended to be called in the setup() function of the main sketch. TODO: ResetButton should probably inherit from Pin. When I first wrote this class I had issues with the order of operation of the various objects and the order of initialization. I think I have resolved those issues but I never went back and refactored this class. This is a good candidate for refactoring, but it also works fine so I have not bothered to do it.

- ButtonState readButton ()

  This function is used to read the state of the reset button.

- bool isButtonLongPressed ()

  This function is used to check if the button has been long pressed.

- bool isButtonDoublePressed ()

  This function is used to check if the button has been double pressed.

- bool isButtonSinglePressed ()

  This function is used to check if the button has been single pressed.

Private Attributes

- int pin
- InputPin resetButton
- ButtonState buttonState
- unsigned long lastButtonPress
- int pressCount

Static Private Attributes

- static const unsigned long DOUBLE_PRESS_INTERVAL = 500
- static const unsigned long LONG_PRESS_INTERVAL = 1000

4.29.1 Detailed Description

This class is used to read the reset button input.

4.29.2 Member Enumeration Documentation

4.29.2.1 ButtonState

```
enum ResetButton::ButtonState
```

Enumerator

OPEN	
PRESSED	

4.29.3 Constructor & Destructor Documentation

4.29.3.1 ResetButton()

```
ResetButton::ResetButton (
            int pin )
```

Construct a new Reset Button:: Reset Button object.

Parameters

pin	

4.29.4 Member Function Documentation

4.29.4.1 begin()

```
void ResetButton::begin ( )
```

This function is used to initialize the reset button. This is intended to be called in the setup() function of the main sketch. TODO: ResetButton should probably inherit from Pin. When I first wrote this class I had issues with the order of operation of the various objects and the order of initialization. I think I have resolved those issues but I never went back and refactored this class. This is a good candidate for refactoring, but it also works fine so I have not bothered to do it.

4.29.4.2 isButtonDoublePressed()

```
bool ResetButton::isButtonDoublePressed ( )
```

This function is used to check if the button has been double pressed.

Returns

bool

4.29.4.3 isButtonLongPressed()

```
bool ResetButton::isButtonLongPressed ( )
```

This function is used to check if the button has been long pressed.

Returns

bool

4.29.4.4 isButtonSinglePressed()

```
bool ResetButton::isButtonSinglePressed ( )
```

This function is used to check if the button has been single pressed.

Returns

bool

4.29.4.5 readButton()

```
ResetButton::ButtonState ResetButton::readButton ( )
```

This function is used to read the state of the reset button.

Returns

ResetButton::ButtonState

4.29.5 Member Data Documentation

4.29.5.1 buttonState

```
ButtonState ResetButton::buttonState  [private]
```

4.29.5.2 DOUBLE_PRESS_INTERVAL

```
const unsigned long ResetButton::DOUBLE_PRESS_INTERVAL = 500  [static], [private]
```

4.29.5.3 lastButtonPress

```
unsigned long ResetButton::lastButtonPress  [private]
```

4.29.5.4 LONG_PRESS_INTERVAL

```
const unsigned long ResetButton::LONG_PRESS_INTERVAL = 1000  [static], [private]
```

4.29.5.5 pin

```
int ResetButton::pin  [private]
```

4.29.5.6 pressCount

```
int ResetButton::pressCount  [private]
```

4.29.5.7 resetButton

```
InputPin ResetButton::resetButton  [private]
```

The documentation for this class was generated from the following files:

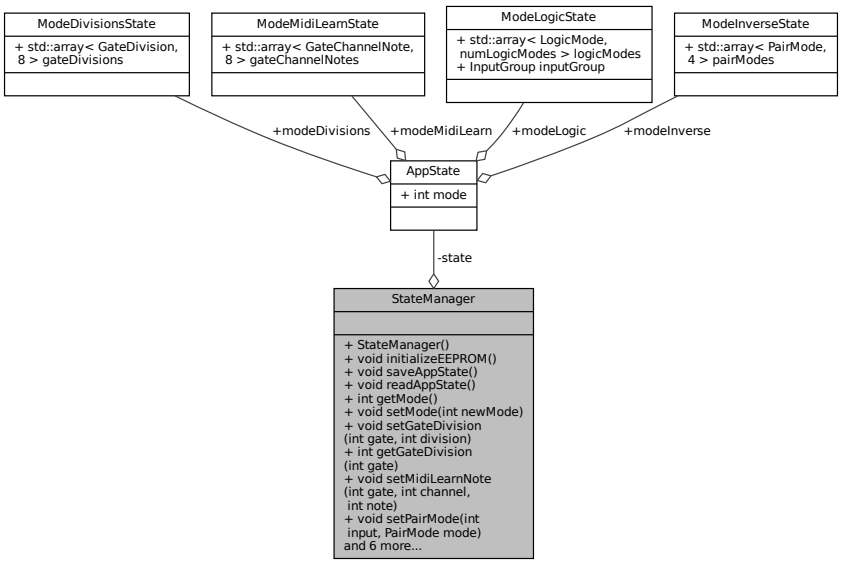
- include/ResetButton.h
- src/ResetButton.cpp

4.30 StateManager Class Reference

The StateManager class is used to manage the application state. It is used to save and read the application state from EEPROM. It uses the AppState struct to hold the state of the application while the application is running. The state is saved to EEPROM when the app is in mode selection mode.

```
#include <StateManager.h>
```

Collaboration diagram for StateManager:



Public Member Functions

- StateManager ()
- void initializeEEPROM ()

  Initializes the EEPROM memory with the default AppState values if the EEPROM has not been initialized.
- void saveAppState ()

  Saves the current AppState object 'state' to the EEPROM memory.
- void readAppState ()

  Reads the AppState object 'state' from the EEPROM memory.
- int getMode ()

  Returns the current mode stored in the AppState object 'state'.
- void setMode (int newMode)

  Sets the current mode in the AppState object 'state'.
- void setGateDivision (int gate, int division)

  Sets the gate division for a specific gate in the AppState object 'state'.
- int getGateDivision (int gate)

  Returns the gate division for a specific gate from the AppState object 'state'.
- void setMidiLearnNote (int gate, int channel, int note)

  Sets the MIDI learn note and channel for a specific gate in the AppState object 'state'.
- void setPairMode (int input, PairMode mode)

  Sets the pair mode for a specific input in the AppState object 'state'.

- PairMode getPairMode (int input)

  Returns the pair mode for a specific input from the AppState object 'state'.
- std::pair< int, int > getMidiLearnNote (int gate)

  Returns the MIDI learn note and channel for a specific gate from the AppState object 'state'.
- void setLogicMode (int index, LogicMode mode)

  Sets the logic mode for a specific index in the AppState object 'state'.
- LogicMode getLogicMode (int index)

  Returns the logic mode for a specific index from the AppState object 'state'.
- void setInputGroup (InputGroup group)

  Sets the input group in the AppState object 'state'.
- InputGroup getInputGroup ()

  Returns the input group from the AppState object 'state'.

Private Attributes

- AppState state

4.30.1 Detailed Description

The StateManager class is used to manage the application state. It is used to save and read the application state from EEPROM. It uses the AppState struct to hold the state of the application while the application is running. The state is saved to EEPROM when the app is in mode selection mode.

4.30.2 Constructor & Destructor Documentation

4.30.2.1 StateManager()

```
StateManager::StateManager ( )
```

Parameters

state	- The AppState object to be saved to the EEPROM

Empty constructor - we will initialize the AppState object in the setup() function this way we can print debug messages.

4.30.3 Member Function Documentation

4.30.3.1 getGateDivision()

```
int StateManager::getGateDivision (
            int gate )
```

Returns the gate division for a specific gate from the AppState object 'state'.

Parameters

gate	- The gate to get the division for

Returns

int - The division for the gate

4.30.3.2 getInputGroup()

```
InputGroup StateManager::getInputGroup ( )
```

Returns the input group from the AppState object 'state'.

Returns

InputGroup - The input group

4.30.3.3 getLogicMode()

```
LogicMode StateManager::getLogicMode (
            int index )
```

Returns the logic mode for a specific index from the AppState object 'state'.

Parameters

index	- The index to get the logic mode for

Returns

LogicMode - The logic mode for the index

4.30.3.4 getMidiLearnNote()

```
std::pair< int, int > StateManager::getMidiLearnNote (
            int gate )
```

Returns the MIDI learn note and channel for a specific gate from the AppState object 'state'.

Parameters

gate	- The gate to get the note and channel for

Returns

std::pair<int, int> - The note and channel for the gate

4.30.3.5 getMode()

```
int StateManager::getMode ( )
```

Returns the current mode stored in the AppState object 'state'.

Returns

int - The current mode

4.30.3.6 getPairMode()

```
PairMode StateManager::getPairMode (
            int input )
```

Returns the pair mode for a specific input from the AppState object 'state'.

Parameters

input	- The input to get the pair mode for

Returns

PairMode - The pair mode for the input

4.30.3.7 initializeEEPROM()

```
void StateManager::initializeEEPROM ( )
```

Initializes the EEPROM memory with the default AppState values if the EEPROM has not been initialized.

Read the current state from the EEPROM

Initialize the EEPROM with the default state

Set the mode to 0 by default. This is the first item int he vector.

Initialize Mode Divisions

Initialize Mode MIDI Learn

Initialize Mode Inverse

Initialize Mode Logic

Save the default state to the EEPROM

4.30.3.8 readAppState()

```
void StateManager::readAppState ( )
```

Reads the AppState object 'state' from the EEPROM memory.

By using get we don't have to read each byte individually

4.30.3.9 saveAppState()

```
void StateManager::saveAppState ( )
```

Saves the current AppState object 'state' to the EEPROM memory.

By using put we don't have to write each byte individually

4.30.3.10 setGateDivision()

```
void StateManager::setGateDivision (
            int gate,
            int division )
```

Sets the gate division for a specific gate in the AppState object 'state'.

Parameters

gate	- The gate to set the division for
division	- The division to set

4.30.3.11 setInputGroup()

```
void StateManager::setInputGroup (
            InputGroup group )
```

Sets the input group in the AppState object 'state'.

Parameters

group	- The group to set

4.30.3.12 setLogicMode()

```
void StateManager::setLogicMode (
            int index,
            LogicMode mode )
```

Sets the logic mode for a specific index in the AppState object 'state'.

Parameters

index	- The index to set the logic mode for
mode	- The mode to set

4.30.3.13 setMidiLearnNote()

```
void StateManager::setMidiLearnNote (
            int gate,
            int note,
            int channel )
```

Sets the MIDI learn note and channel for a specific gate in the AppState object 'state'.

Parameters

gate	- The gate to set the note and channel for
note	- The note to set
channel	- The channel to set

4.30.3.14 setMode()

```
void StateManager::setMode (
            int newMode )
```

Sets the current mode in the AppState object 'state'.

Parameters

newMode	- The new mode to set

Save the new mode to the EEPROM

4.30.3.15 setPairMode()

```
void StateManager::setPairMode (
            int input,
            PairMode mode )
```

Sets the pair mode for a specific input in the AppState object 'state'.

Parameters

input	- The input to set the pair mode for
mode	- The mode to set

4.30.4 Member Data Documentation

4.30.4.1 state

```
AppState StateManager::state  [private]
```

The documentation for this class was generated from the following files:

- include/StateManager.h
- src/StateManager.cpp

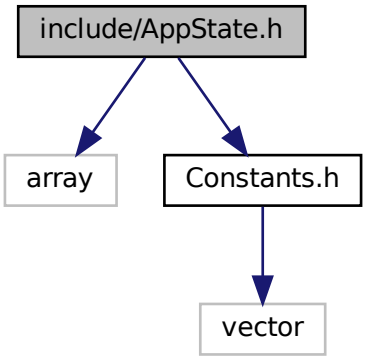
Chapter 5

File Documentation

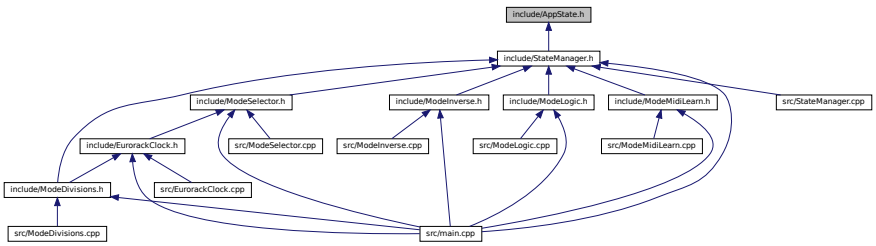
5.1 include/AppState.h File Reference

```
#include <array>
#include "Constants.h"
```

Include dependency graph for AppState.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct GateDivision

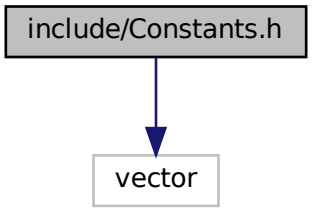
  This is a global struct that holds the state of the application. It mainly holds items that need to persist after a power cycle. The object is initialized managed by the StateManager class.
- struct GateChannelNote
- struct ModeDivisionsState
- struct ModeMidiLearnState
- struct ModeInverseState
- struct ModeLogicState
- struct AppState

5.2 include/Constants.h File Reference

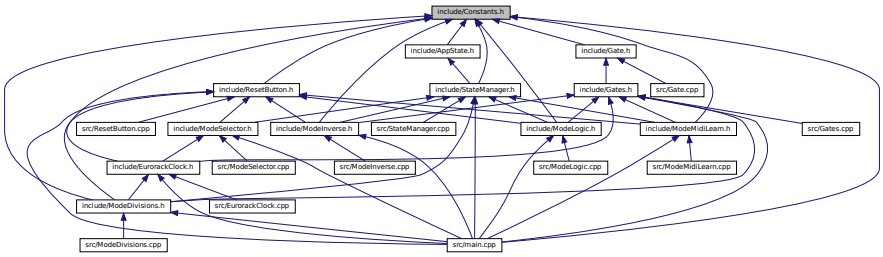
This file contains the constants used throughout the application. I adopted this approach to make the main.cpp file cleaner and easier to read. This file is included in main.cpp and MIDIHandler.cpp among others. There are probably other items to add here, but I'm starting with the musical intervals and PPQN. These are meant to be constants, so they are declared as extern here and defined in Constants.cpp.

```
#include <vector>
```

Include dependency graph for Constants.h:



This graph shows which files directly or indirectly include this file:



Enumerations

- enum PairMode { NORMAL , INVERT , RISE_FALL }
- enum InputGroup { GROUP_TWO , GROUP_ALL }
- enum LogicMode {
  AND , OR , XOR , NAND ,
  NOR , XNOR }

Variables

- std::vector< int > musicalIntervals
- const int musicalIntervalsSize
- unsigned char internalPPQN
- const unsigned char numLogicModes = 6

5.2.1 Detailed Description

This file contains the constants used throughout the application. I adopted this approach to make the main.cpp file cleaner and easier to read. This file is included in main.cpp and MIDIHandler.cpp among others. There are probably other items to add here, but I'm starting with the musical intervals and PPQN. These are meant to be constants, so they are declared as extern here and defined in Constants.cpp.

5.2.2 Enumeration Type Documentation

5.2.2.1 InputGroup

```
enum InputGroup
```

Enumerator

GROUP_TWO	
GROUP_ALL	

5.2.2.2 LogicMode

```
enum LogicMode
```

Enumerator

AND	
OR	
XOR	
NAND	
NOR	
XNOR	

5.2.2.3 PairMode

```
enum PairMode
```

Enumerator

NORMAL	
INVERT	
RISE_FALL	

5.2.3 Variable Documentation

5.2.3.1 internalPPQN

```
unsigned char internalPPQN  [extern]
```

5.2.3.2 musicalIntervals

```
std::vector<int> musicalIntervals  [extern]
```

5.2.3.3 musicalIntervalsSize

```
const int musicalIntervalsSize  [extern]
```

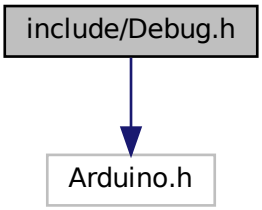
5.2.3.4 numLogicModes

```
const unsigned char numLogicModes = 6
```

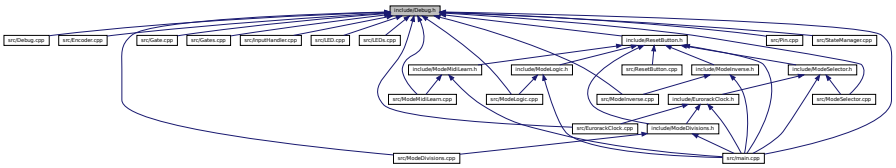
5.3 include/Debug.h File Reference

```
#include <Arduino.h>
```

Include dependency graph for Debug.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Debug

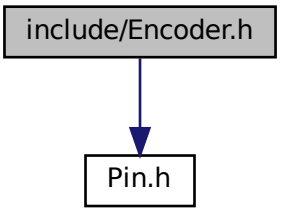
  Helper class for debugging. This makes it easier to enable/disable debug messages. It is used in conjunction with the DEBUG_PRINT macro. Which adds the file, line, and function name to the debug message so that the developer can easily find where the message is coming from.

5.4 include/Encoder.h File Reference

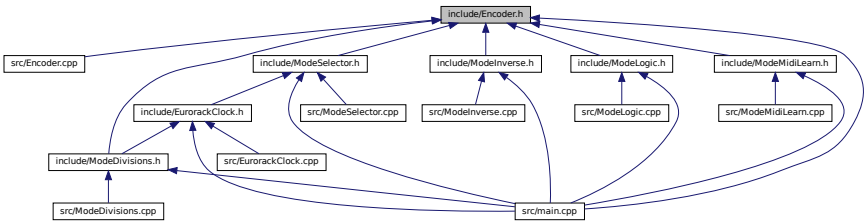
This file contains the Encoder class which manages the physical encoder and button inputs.

```
#include "Pin.h"
```

Include dependency graph for Encoder.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Encoder

  This class is used to read the encoder and button inputs.

5.4.1 Detailed Description

This file contains the Encoder class which manages the physical encoder and button inputs.

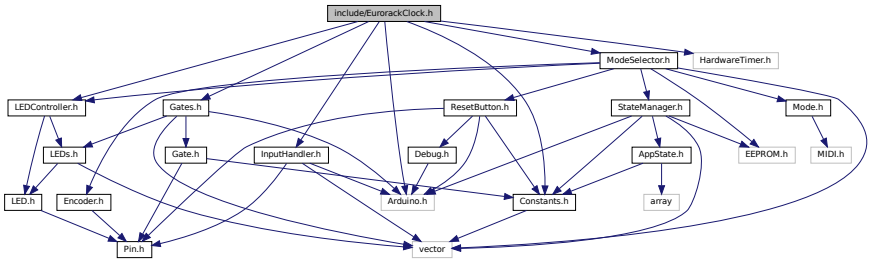
5.5 include/EurorackClock.h File Reference

This file contains the EurorackClock class, which is used to handle the clock and tempo of the device. This is one of the first classes I wrote for the project, and it has been refactored a few times. It probably could use with a bit more refactoring love, but it works well enough for now.

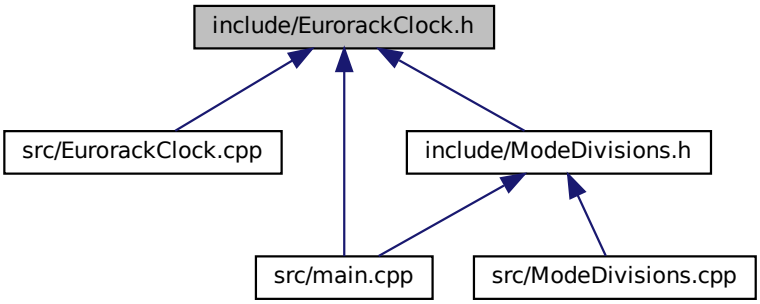
```
#include <Arduino.h>
#include <HardwareTimer.h>
#include "Gates.h"
#include "LEDController.h"
#include "Constants.h"
```

```
#include "InputHandler.h"
#include "ModeSelector.h"
```

Include dependency graph for EurorackClock.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct ClockState

  The ClockState struct is used to store the current state of the clock.
- class EurorackClock

  The EurorackClock class is used to handle the clock and tempo of the device. It utilizes an interrupt to handle the clock ticks, and can be set to an external tempo.

5.5.1 Detailed Description

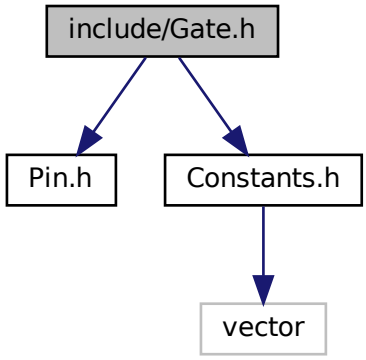
This file contains the EurorackClock class, which is used to handle the clock and tempo of the device. This is one of the first classes I wrote for the project, and it has been refactored a few times. It probably could use with a bit more refactoring love, but it works well enough for now.

5.6 include/Gate.h File Reference

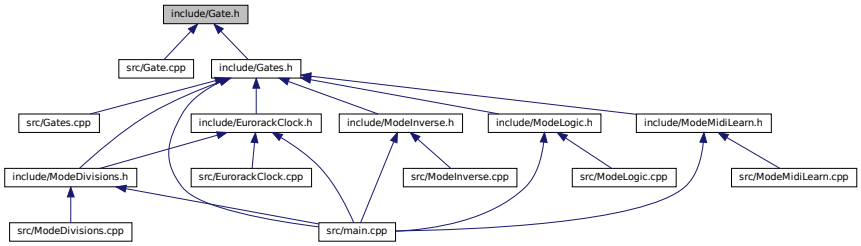
This file contains the Gate class, which is used to control the gates. Quick note: This class has a data member to hold a "Mute" state. I purposely left out the implementation of the mute functionality within the gate object. This is because we have more flexibility and less risk of bugs if we handle the mute functionality in the mode classes. See ModeInverse.h for an example of how to mute the gates.

```
#include "Pin.h"
#include "Constants.h"
```

Include dependency graph for Gate.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Gate

  This class defines what a gate is and how it should behave. It inherits from the OutputPin class, which provides the basic functionality for a pin including setting state to HIGH or LOW, getting the current state, etc.

5.6.1 Detailed Description

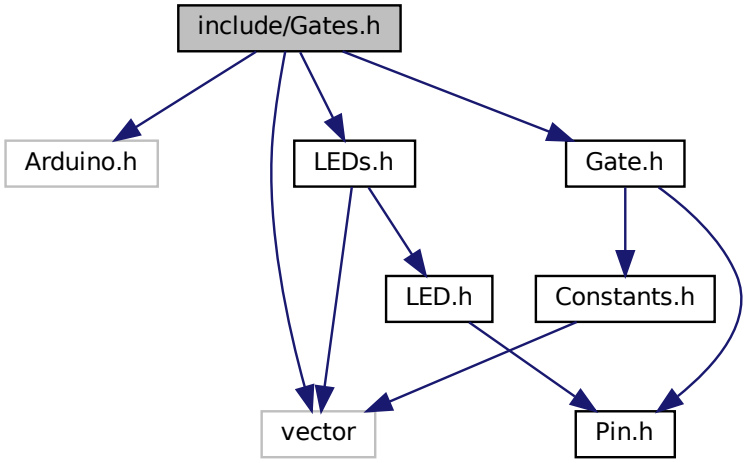
This file contains the Gate class, which is used to control the gates. Quick note: This class has a data member to hold a "Mute" state. I purposely left out the implementation of the mute functionality within the gate object. This is because we have more flexibility and less risk of bugs if we handle the mute functionality in the mode classes. See ModeInverse.h for an example of how to mute the gates.

5.7 include/Gates.h File Reference

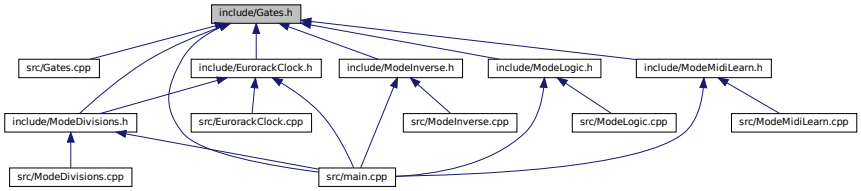
This file contains the Gates class, which is used to control the gates in the system.

```
#include <Arduino.h>
#include "Gate.h"
#include "LEDs.h"
#include <vector>
```

Include dependency graph for Gates.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Gates

  This is a collection of gates and thus the main thing we are working with in this project. Very rarely will you need to interact with the Gate class directly, as most of the functionality is handled by the Gates class.

5.7.1 Detailed Description

This file contains the Gates class, which is used to control the gates in the system.

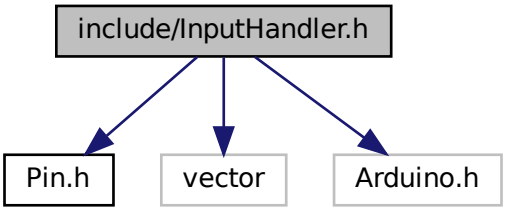
This is also an early class, and could use some refactoring love. But again, this works so it's fine for now. TODO: We could probably move the division logic out of this, but it doesn't interfere with the other modes so it's fine for now.

5.8 include/InputHandler.h File Reference

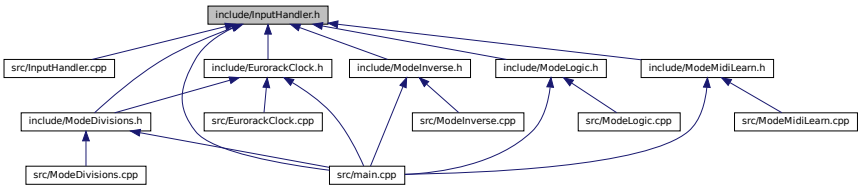
This file contains the InputHandler class, which is used to read the CV inputs. Right now it only reads the CV inputs but it could be expanded to handle other inputs in the future.

```
#include "Pin.h"
#include <vector>
#include <Arduino.h>
```

Include dependency graph for InputHandler.h:



This graph shows which files directly or indirectly include this file:



Classes

- class InputHandler

  This class is used to read the CV inputs. It is a simple class that uses the AnalogInputPin class to read the CV inputs. Alias the reset and clock inputs to cvC and cvD respectively. cvC is the reset input and cvD is the clock input.

5.8.1 Detailed Description

This file contains the InputHandler class, which is used to read the CV inputs. Right now it only reads the CV inputs but it could be expanded to handle other inputs in the future.

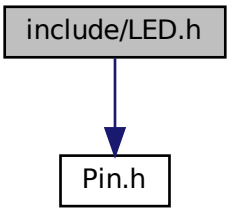
TODO: This is one of the last classes I created and would probably be a better home for the Clock and Reset input handling. I might refactor this in the future to include those features, but again the code works now so it's fine for now.

5.9 include/LED.h File Reference

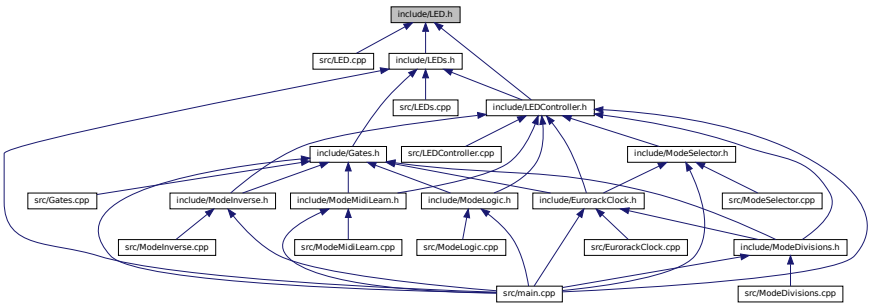
This file contains the LED class, which is used to control the LEDs associated with the gates. This of this like a UI matrix as well. The LEDs are used to indicate the state of the gates, as well as, to provide feedback when selecting modes, gates, etc.

```
#include "Pin.h"
```

Include dependency graph for LED.h:



This graph shows which files directly or indirectly include this file:



Classes

- class LED

  This class defines what an LED is and how it should behave.

5.9.1 Detailed Description

This file contains the LED class, which is used to control the LEDs associated with the gates. This of this like a UI matrix as well. The LEDs are used to indicate the state of the gates, as well as, to provide feedback when selecting modes, gates, etc.

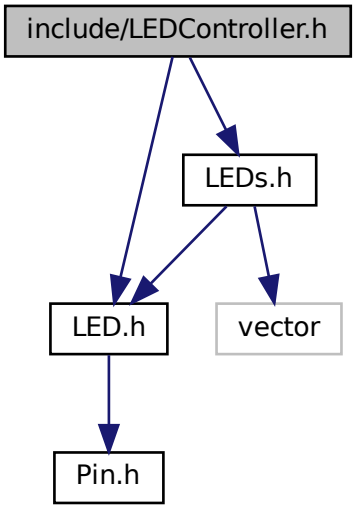
Like the Gate and Gates classes, you should not really interact with this clas directly, but rather through the LEDController class.

5.10 include/LEDController.h File Reference

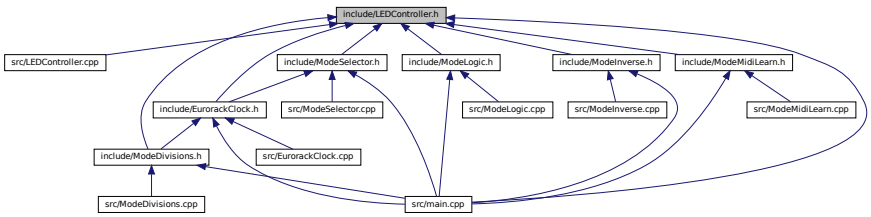
I originally had the LEDs class handling the LED control, but I ran into issues making coding difficult all the modes needed to interact with the LEDs and maintain some sort of state. To help facilityate state and management of the leds I created this class to handle the LED control and to provide a more user-friendly interface.

```
#include "LED.h"
#include "LEDs.h"
```

Include dependency graph for LEDController.h:



This graph shows which files directly or indirectly include this file:



Classes

- class LEDController

  This class is used as the main interface for controlling the LEDs.

5.10.1 Detailed Description

I originally had the LEDs class handling the LED control, but I ran into issues making coding difficult all the modes needed to interact with the LEDs and maintain some sort of state. To help facilityate state and management of the leds I created this class to handle the LED control and to provide a more user-friendly interface.

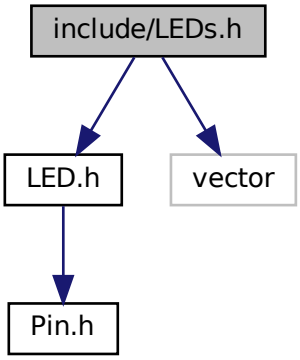
TODO: The tempo LED is still handled by the EurorackClock class, which is a bit of a mess. I should move that to this class and refactor the EurorackClock class to be more of a clock manager. But again, it works so it's fine for now.

5.11 include/LEDs.h File Reference

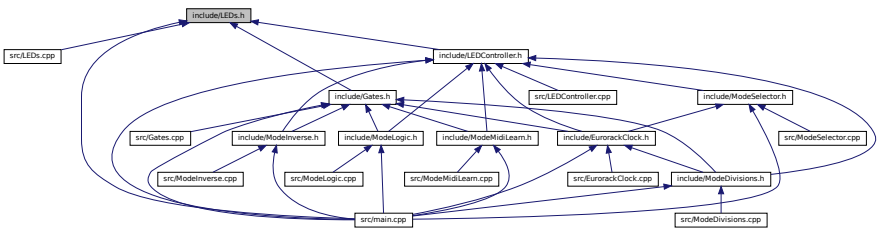
This is a collection of LEDs whicle you could interact with this class directly, it is recommended to use the LEDController class to interact with the LEDs.

```
#include "LED.h"
#include <vector>
```

Include dependency graph for LEDs.h:



This graph shows which files directly or indirectly include this file:



Classes

- class LEDs

  This is a collection of LEDs and mainly used by the LEDController class. Use that if you need to interact with the LEDs.

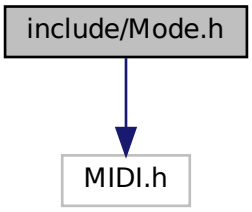
5.11.1 Detailed Description

This is a collection of LEDs whicle you could interact with this class directly, it is recommended to use the LEDController class to interact with the LEDs.

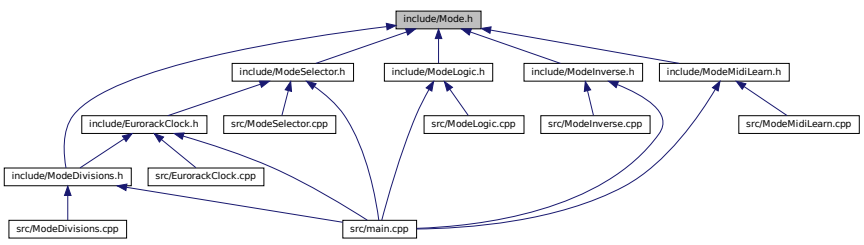
5.12 include/Mode.h File Reference

This is the base class for the various modes that the module can be in. It defines the interface that all modes must implement.

`#include <MIDI.h>`
Include dependency graph for Mode.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Mode

  This class is the base for our application modes.

5.12.1 Detailed Description

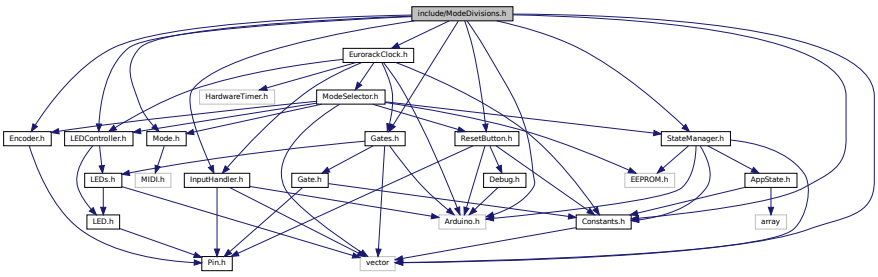
This is the base class for the various modes that the module can be in. It defines the interface that all modes must implement.

5.13 include/ModeDivisions.h File Reference

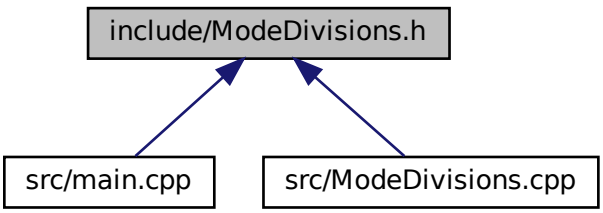
This mode is the main mode for the Eurorack Clock module.

```
#include "Mode.h"
#include "Encoder.h"
#include "Gates.h"
#include "LEDController.h"
#include "EurorackClock.h"
#include "Constants.h"
#include "ResetButton.h"
#include "InputHandler.h"
#include <vector>
#include <Arduino.h>
#include "StateManager.h"
```

Include dependency graph for ModeDivisions.h:



This graph shows which files directly or indirectly include this file:



Classes

- class ModeDivisions

  This class uses the eurorack clock to provide us pullses with selectable division. It can be synced to a clock too, internal and external.

5.13.1 Detailed Description

This mode is the main mode for the Eurorack Clock module.

In this mode, the user can set the tempo, select the division of the clock signal, and select the gate output. It works with the Encoder, Gates, LEDController, MIDIHandler, ResetButton, and EurorackClock classes.

This mode utilizes an internal clock and can be synchronized with an external clock signal as well as MIDI clock. When the mode is active, the user can set the tempo by turning the encoder knob. The tempo can be set between 20 and 340 BPM. This is done by turning the encoder knob to the left to decrease the tempo or to the right to increase the tempo when in tempo selection mode.

Tempo selection mode is activated by pressing the encoder knob twice in quick succession. Then to exit this mode the user can press the encoder knob twice again.

The user can also select the division of the clock signal for each gate output. The division can be set to 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192, or 256 PPQN. This is done by first selecting the gate output by rotating the encoder. The selected gate output will be indicated by the LED corresponding to the gate output. Then the user can press the encoder knob to enter the division selection mode. The division can be set by rotating the encoder knob. Pese the encoder knob again to exit the division selection mode.

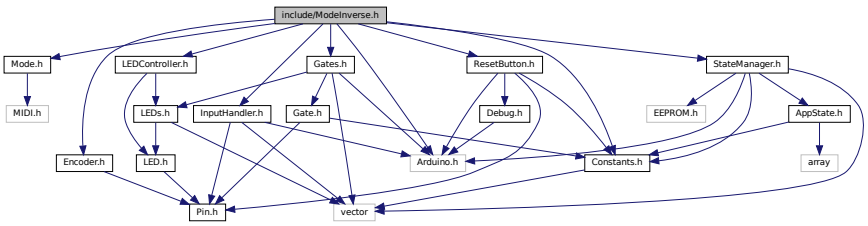
The internal clock is done by using the EurorackClock class. The clock signal is sent to the gate outputs using the Gates class. It's all complicated stuff but I'm working on making it easier to understand. The MIDIHandler class is used to handle MIDI clock signals. The LEDController class is used to control the LEDs on the module.

TODO: The internal clock works with a PPQN of 24 by default. This can be changed by pressing the reset button and rotating the encoder knob to select the desired PPQN.

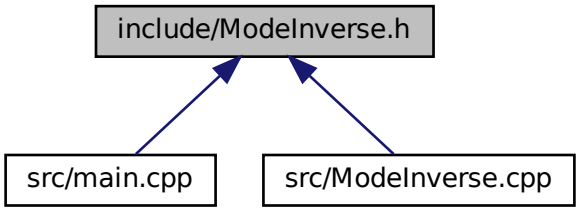
5.14 include/ModeInverse.h File Reference

```
#include "Mode.h"
#include <Arduino.h>
#include "LEDController.h"
#include "Encoder.h"
#include "Gates.h"
#include "Constants.h"
#include "InputHandler.h"
#include "ResetButton.h"
#include "StateManager.h"
```

Include dependency graph for ModeInverse.h:



This graph shows which files directly or indirectly include this file:



Classes

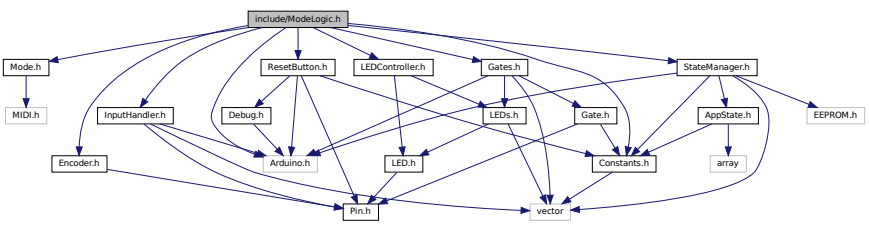
- class ModeInverse

  This mode is for inverting the gates. If the gate is high, it will be low and vice versa. The user can select the gate pairs and change the behaviour of the gates. So instead of sending gates, it will send triggers on the separate gates for the rising edge and falling edge of the gate.

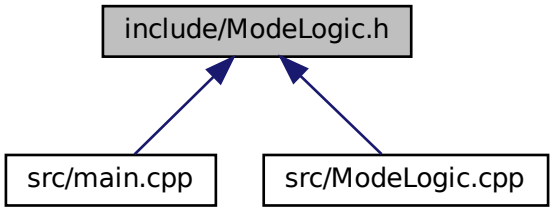
5.15 include/ModeLogic.h File Reference

```
#include "Mode.h"
#include <Arduino.h>
#include "Constants.h"
#include "Encoder.h"
#include "Gates.h"
#include "InputHandler.h"
#include "LEDController.h"
#include "ResetButton.h"
#include "StateManager.h"
```

Include dependency graph for ModeLogic.h:



This graph shows which files directly or indirectly include this file:



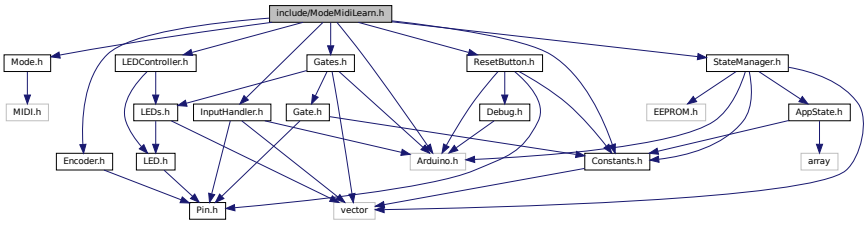
Classes

- class ModeLogic

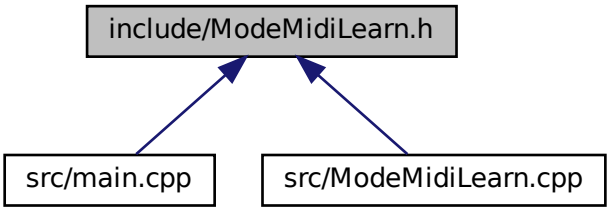
5.16 include/ModeMidiLearn.h File Reference

```
#include "Mode.h"
#include <Arduino.h>
#include "LEDController.h"
#include "Encoder.h"
#include "Gates.h"
#include "Constants.h"
#include "InputHandler.h"
#include "ResetButton.h"
#include "StateManager.h"
```

Include dependency graph for ModeMidiLearn.h:



This graph shows which files directly or indirectly include this file:



Classes

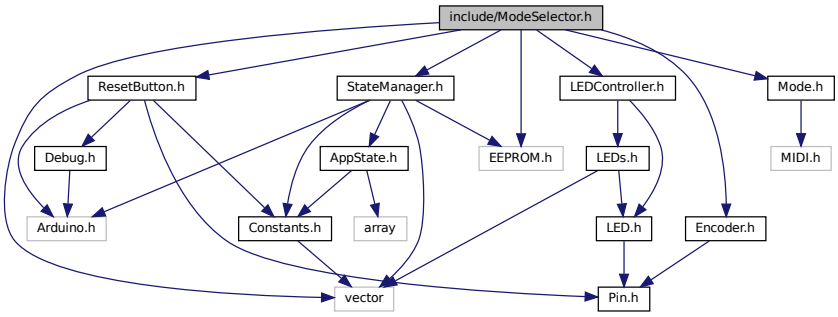
- class ModeMidiLearn

  This is a MIDI to Trigger class for Note On but it only cares about the channel number.

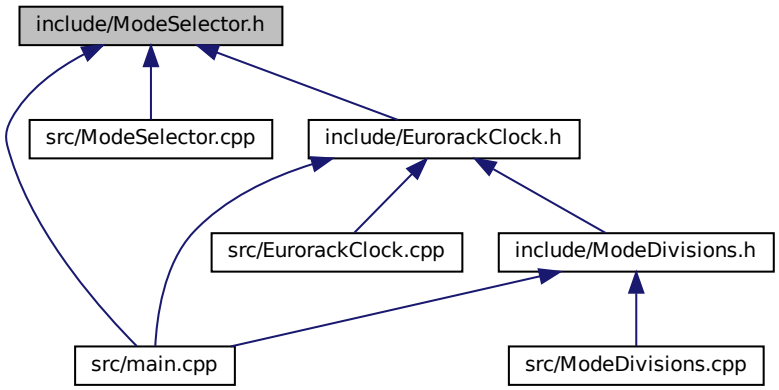
5.17 include/ModeSelector.h File Reference

```
#include <vector>
#include <EEPROM.h>
#include "LEDController.h"
#include "Encoder.h"
#include "Mode.h"
#include "ResetButton.h"
#include "StateManager.h"
```

Include dependency graph for ModeSelector.h:



This graph shows which files directly or indirectly include this file:



Classes

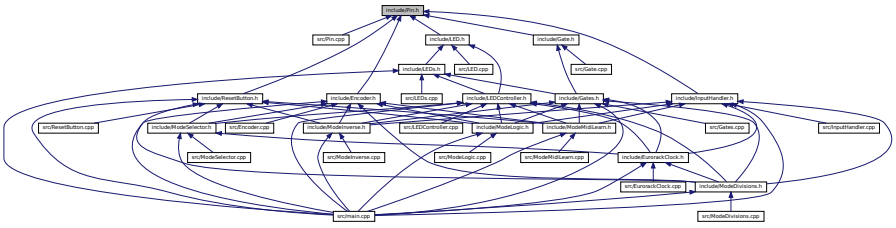
- class ModeSelector

  Mode Selector Singleton. This class is responsible for managing the different modes of the device. It provides methods to add modes, set the current mode, and handle mode selection.

5.18 include/Pin.h File Reference

This file contains the pin base class and its derived classes for input, output, and PWM pins.

This graph shows which files directly or indirectly include this file:



Classes

- class Pin

  This class represents a pin on the microcontroller.
- class InputPin

  This class represents an input pin on the microcontroller.
- class AnalogInputPin

  This class represents an analog input pin on the microcontroller.
- class OutputPin

  This class represents an output pin on the microcontroller.
- class PWMPin

  This class represents a PWM output pin on the microcontroller.

5.18.1 Detailed Description

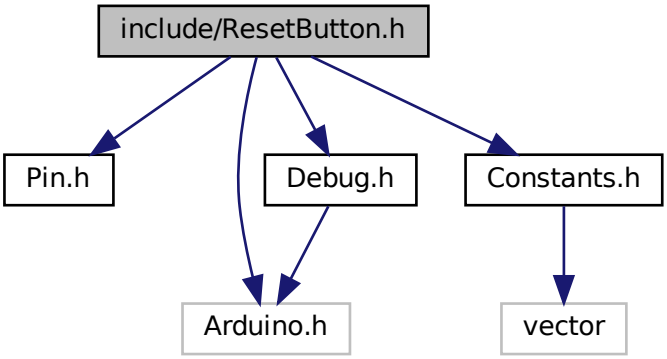
This file contains the pin base class and its derived classes for input, output, and PWM pins.

5.19 include/ResetButton.h File Reference

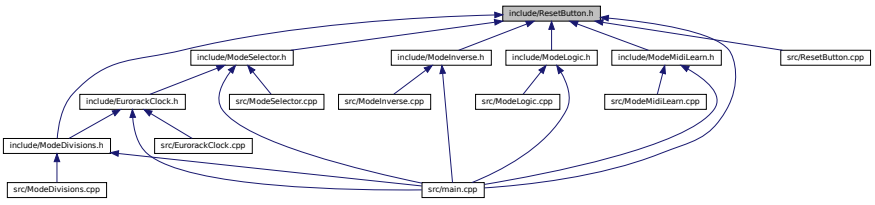
This file contains the ResetButton class which manages the physical reset button input.

```
#include "Pin.h"
#include <Arduino.h>
#include "Debug.h"
#include "Constants.h"
```

Include dependency graph for ResetButton.h:



This graph shows which files directly or indirectly include this file:



Classes

- class ResetButton

  This class is used to read the reset button input.

5.19.1 Detailed Description

This file contains the ResetButton class which manages the physical reset button input.

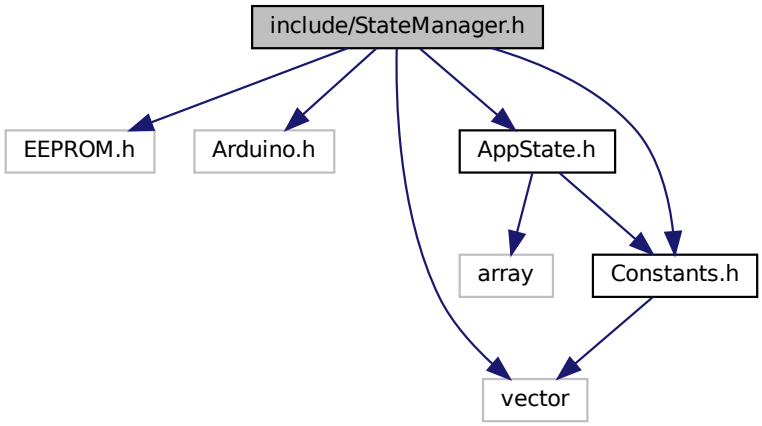
TODO: This class could probably be combined with the Encoder class to create a more generic Button class as the functionality is very similar. But it works now so it's fine for now.

5.20 include/StateManager.h File Reference

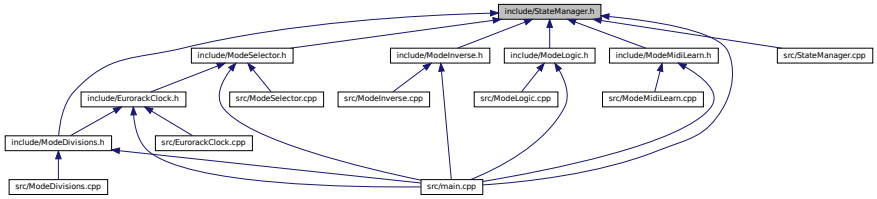
This file contains the StateManager class, which is used to manage the application state. It is used to save and read the application state from EEPROM.

```
#include <EEPROM.h>
#include <Arduino.h>
#include <vector>
#include "AppState.h"
#include "Constants.h"
```

Include dependency graph for StateManager.h:



This graph shows which files directly or indirectly include this file:



Classes

- class StateManager

  The StateManager class is used to manage the application state. It is used to save and read the application state from EEPROM. It uses the AppState struct to hold the state of the application while the application is running. The state is saved to EEPROM when the app is in mode selection mode.

5.20.1 Detailed Description

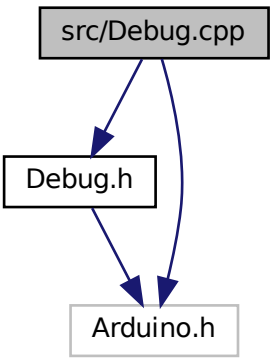
This file contains the StateManager class, which is used to manage the application state. It is used to save and read the application state from EEPROM.

5.21 src/Debug.cpp File Reference

helper class for debugging

```
#include "Debug.h"
#include <Arduino.h>
```

Include dependency graph for Debug.cpp:



5.21.1 Detailed Description

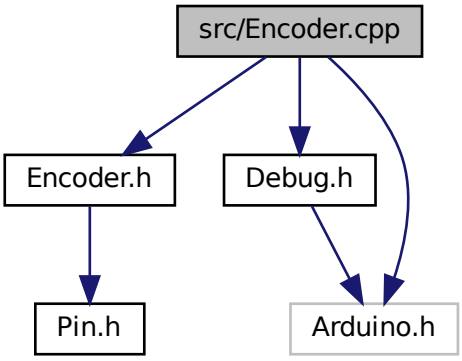
helper class for debugging

5.22 src/Encoder.cpp File Reference

This file contains the implementation of the Encoder class which manages the physical encoder and button inputs.

```
#include "Encoder.h"
#include "Debug.h"
#include <Arduino.h>
```

Include dependency graph for Encoder.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.22.1 Detailed Description

This file contains the implementation of the Encoder class which manages the physical encoder and button inputs.

5.22.2 Macro Definition Documentation

5.22.2.1 DEBUG_PRINT

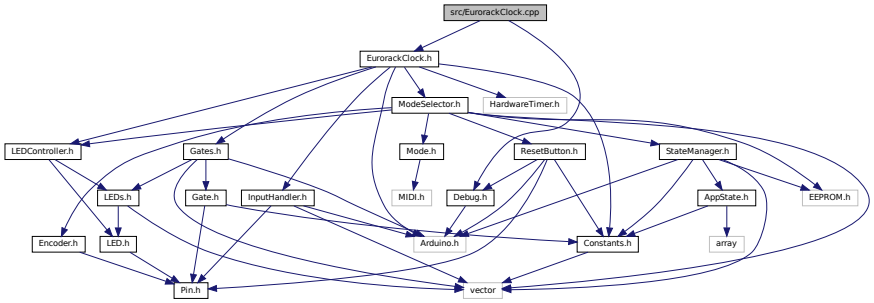
```
#define DEBUG_PRINT(
             message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.23 src/EurorackClock.cpp File Reference

This file contains the implementation of the EurorackClock class, which is used to manage the clock and gates of the Eurorack module.

```
#include "EurorackClock.h"
#include "Debug.h"
```

Include dependency graph for EurorackClock.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.23.1 Detailed Description

This file contains the implementation of the EurorackClock class, which is used to manage the clock and gates of the Eurorack module.

5.23.2 Macro Definition Documentation

5.23.2.1 DEBUG_PRINT

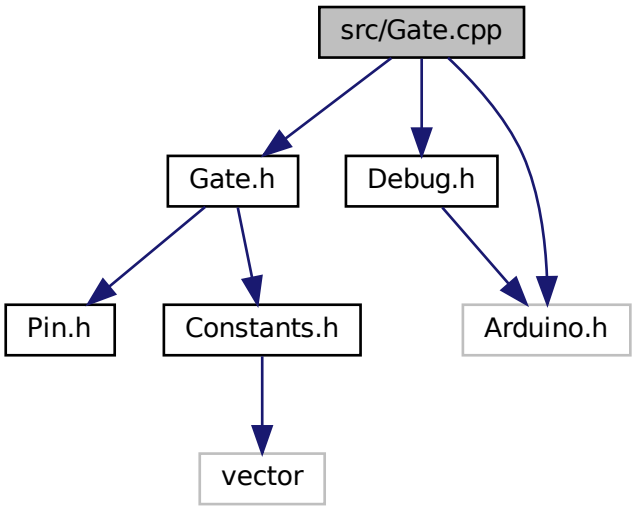
```
#define DEBUG_PRINT(
            message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.24 src/Gate.cpp File Reference

This file contains the implementation of the Gate class, which is used to manage the gates of the Eurorack module.

```
#include "Gate.h"
#include "Debug.h"
#include <Arduino.h>
```

Include dependency graph for Gate.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.24.1 Detailed Description

This file contains the implementation of the Gate class, which is used to manage the gates of the Eurorack module.

5.24.2 Macro Definition Documentation

5.24.2.1 DEBUG_PRINT

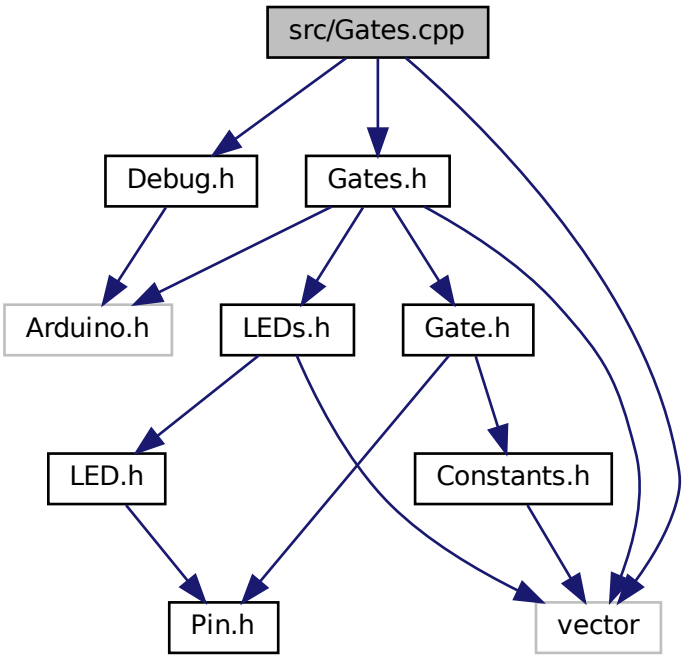
```
#define DEBUG_PRINT(
             message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.25 src/Gates.cpp File Reference

This file contains the implementation of the Gates class, which is used to manage the gates of the Eurorack module.

```
#include "Gates.h"
#include "Debug.h"
#include <vector>
```

Include dependency graph for Gates.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.25.1 Detailed Description

This file contains the implementation of the Gates class, which is used to manage the gates of the Eurorack module.

5.25.2 Macro Definition Documentation

5.25.2.1 DEBUG_PRINT

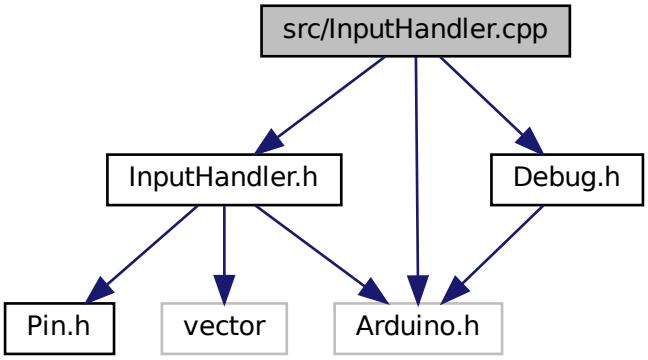
```
#define DEBUG_PRINT(
             message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.26 src/InputHandler.cpp File Reference

This file contains the implementation of the InputHandler class, which is used to manage the CV inputs of the Eurorack module.

```
#include "InputHandler.h"
#include "Debug.h"
#include <Arduino.h>
```

Include dependency graph for InputHandler.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.26.1 Detailed Description

This file contains the implementation of the InputHandler class, which is used to manage the CV inputs of the Eurorack module.

5.26.2 Macro Definition Documentation

5.26.2.1 DEBUG_PRINT

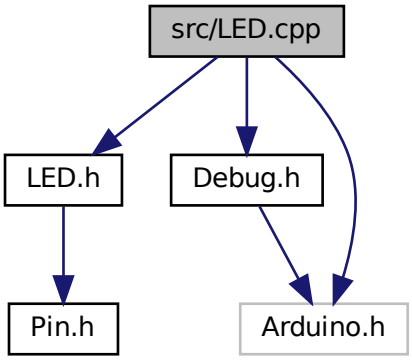
```
#define DEBUG_PRINT(
            message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.27 src/LED.cpp File Reference

This file contains the implementation of the LED class, which is used to manage the LEDs of the Eurorack module.

```
#include "LED.h"
#include "Debug.h"
#include <Arduino.h>
```

Include dependency graph for LED.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.27.1 Detailed Description

This file contains the implementation of the LED class, which is used to manage the LEDs of the Eurorack module.

5.27.2 Macro Definition Documentation

5.27.2.1 DEBUG_PRINT

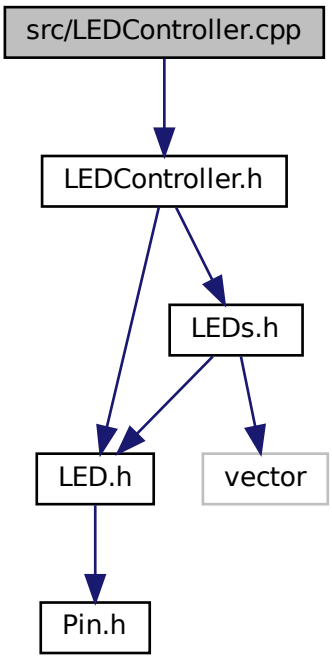
```
#define DEBUG_PRINT(
            message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.28 src/LEDController.cpp File Reference

This file contains the implementation of the LEDController class, which is used to manage the LEDs of the Eurorack module.

```
#include "LEDController.h"
```

Include dependency graph for LEDController.cpp:



5.28.1 Detailed Description

This file contains the implementation of the LEDController class, which is used to manage the LEDs of the Eurorack module.

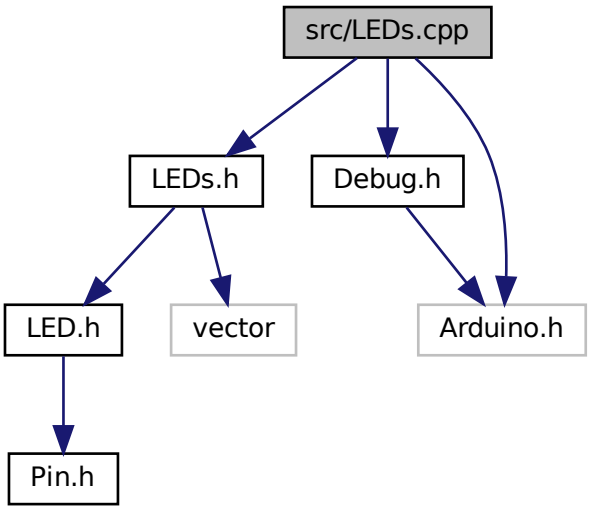
5.29 src/LEDs.cpp File Reference

This is the implementation file for the LEDs class, which is used to manage the LEDs of the Eurorack module.

```
#include "LEDs.h"
#include "Debug.h"
```

```
#include <Arduino.h>
```

Include dependency graph for LEDs.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.29.1 Detailed Description

This is the implementation file for the LEDs class, which is used to manage the LEDs of the Eurorack module.

5.29.2 Macro Definition Documentation

5.29.2.1 DEBUG_PRINT

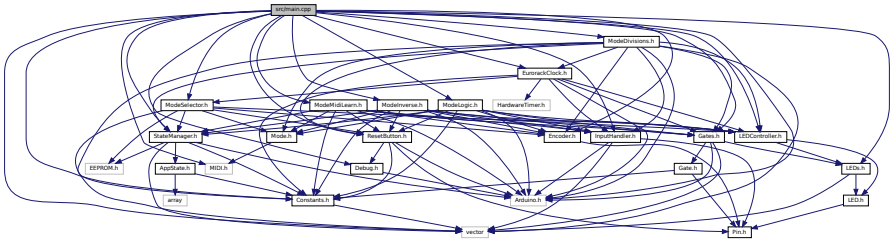
```
#define DEBUG_PRINT(
            message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.30 src/main.cpp File Reference

This is the main entrypoint of the G8R application. I'm trying to keep this file as clean as possible, so most of the logic is in the Mode classes.

```
#include <Arduino.h>
#include <vector>
#include <MIDI.h>
#include "Gates.h"
#include "ModeSelector.h"
#include "LEDs.h"
#include "Debug.h"
#include "Encoder.h"
#include "EurorackClock.h"
#include "Constants.h"
#include "LEDController.h"
#include "ResetButton.h"
#include "InputHandler.h"
#include "StateManager.h"
#include "ModeDivisions.h"
#include "ModeMidiLearn.h"
#include "ModeInverse.h"
#include "ModeLogic.h"
```

Include dependency graph for main.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

  Debug print macro, used to make debugging easier.
- #define RX_PIN PA3
- #define TX_PIN PA2
- #define ENCODER_PINA PB13
- #define ENCODER_PINB PB14
- #define ENCODER_BUTTON PB12
- #define CLOCK_PIN PB10
- #define RESET_PIN PB11
- #define RESET_BUTTON PB15
- #define TEMPO_LED PA8
- #define CV_A_PIN PA4
- #define CV_B_PIN PA5

Functions

- midi::SerialMIDI< HardwareSerial > midiSerial (Serial2)

  Instance of the EurorackClock class.
- midi::MidiInterface< midi::SerialMIDI< HardwareSerial > > midiInterface (midiSerial)
- void midiSetup ()

  Instance of ModeLogic class.
- void setup ()

  Setup function for the Arduino sketch.
- void loop ()

  Main loop function for the Arduino sketch.

Variables

- std::vector< int > pins = {PA15, PB3, PB4, PB5, PB6, PB7, PB8, PB9}
- const int numPins = pins.size()
- std::vector< int > ledPins = {PA12, PA11, PB1, PB0, PA7, PA6, PA1, PA0}
- int numLedPins = ledPins.size()
- bool inModeSelection = false
- bool isInSelection = false
- unsigned char internalPPQN = 24
- std::vector< int > musicalIntervals = {1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 72, 96, 128, 144, 192, 288, 384, 576, 768, 1152, 1536}
- const int musicalIntervalsSize = musicalIntervals.size()
- Gates gates = Gates(pins, numPins)
- LEDs leds = LEDs(ledPins, numLedPins)
- StateManager stateManager = StateManager()
- Encoder encoder = Encoder(ENCODER_PINA, ENCODER_PINB, ENCODER_BUTTON)

  Instance of the StateManager class used to manage state of the device in EEPROM.
- ResetButton resetButton = ResetButton(RESET_BUTTON)

  Instance of the Encoder class.
- LEDController ledController (leds, TEMPO_LED)

  Instance of the ResetButton class.
- InputHandler inputHandler = InputHandler(CV_A_PIN, CV_B_PIN, RESET_PIN, CLOCK_PIN)

  Instance of the LEDController class.
- EurorackClock clock (gates, ledController, inputHandler)

  Instance of the InputHandler class.
- ModeSelector & modeSelector = ModeSelector::getInstance()
- Mode * currentMode = nullptr

  Instance of the ModeSelector class.
- Mode * previousMode = nullptr

  Pointer to the current mode.
- ModeDivisions modeDivisions (stateManager, encoder, inputHandler, gates, ledController, midiInterface, resetButton, clock)

  Pointer to the previous mode.
- ModeMidiLearn modeMidiLearn (stateManager, encoder, inputHandler, gates, ledController, midiInterface, resetButton)

  Instance of ModeDivisions class.
- ModeInverse modeInverse (stateManager, encoder, inputHandler, gates, ledController, midiInterface, resetButton)

  Instance of ModeMidiLearn class.
- ModeLogic modeLogic (stateManager, encoder, inputHandler, gates, ledController, midiInterface, resetButton)

  Instance of ModeInverse class.

5.30.1 Detailed Description

This is the main entrypoint of the G8R application. I'm trying to keep this file as clean as possible, so most of the logic is in the Mode classes.

5.30.2 Macro Definition Documentation

5.30.2.1 CLOCK_PIN

```
#define CLOCK_PIN PB10
```

5.30.2.2 CV_A_PIN

```
#define CV_A_PIN PA4
```

5.30.2.3 CV_B_PIN

```
#define CV_B_PIN PA5
```

5.30.2.4 DEBUG_PRINT

```
#define DEBUG_PRINT(
              message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

Debug print macro, used to make debugging easier.

5.30.2.5 ENCODER_BUTTON

```
#define ENCODER_BUTTON PB12
```

5.30.2.6 ENCODER_PINA

```
#define ENCODER_PINA PB13
```

5.30.2.7 ENCODER_PINB

```
#define ENCODER_PINB PB14
```

5.30.2.8 RESET_BUTTON

```
#define RESET_BUTTON PB15
```

5.30.2.9 RESET_PIN

```
#define RESET_PIN PB11
```

5.30.2.10 RX_PIN

```
#define RX_PIN PA3
```

5.30.2.11 TEMPO_LED

```
#define TEMPO_LED PA8
```

5.30.2.12 TX_PIN

```
#define TX_PIN PA2
```

5.30.3 Function Documentation

5.30.3.1 loop()

```
void loop ( )
```

Main loop function for the Arduino sketch.

This function is called repeatedly as long as the Arduino is powered on. It contains the main logic of the sketch.

5.30.3.2 midiInterface()

```
midi::MidiInterface<midi::SerialMIDI<HardwareSerial> > midiInterface (
            midiSerial  )
```

5.30.3.3 midiSerial()

```
midi::SerialMIDI<HardwareSerial> midiSerial (
            Serial2  )
```

Instance of the EurorackClock class.

5.30.3.4 midiSetup()

```
void midiSetup ( )
```

Instance of ModeLogic class.

This function is used to setup the MIDI interface. It is intended to be called in the setup() function of the main sketch. We'll set the actual callback functions in the mode classes setup() and teardown() methods.

5.30.3.5 setup()

```
void setup ( )
```

Setup function for the Arduino sketch.

This function is called once when the sketch starts. It is used to initialize variables, input and output pin modes, and start using libraries. Initialize the debug settings

Add the modes to the ModeSelector. IMPORTANT: Add the modes in the order you want them to be selected via the encoder.

This gets the current mode from the ModeSelector and sets it as the current mode. It then updates the pointer to the current mode. This way we can switch modes easily at runtime by changing the the pointer to the current mode. Basically a State Pattern.

5.30.4 Variable Documentation

5.30.4.1 clock

```
EurorackClock clock(gates, ledController, inputHandler) (
            gates ,
            ledController ,
            inputHandler  )
```

Instance of the InputHandler class.

5.30.4.2 currentMode

```
Mode* currentMode = nullptr
```

Instance of the ModeSelector class.

5.30.4.3 encoder

```
Encoder encoder = Encoder(ENCODER_PINA, ENCODER_PINB, ENCODER_BUTTON)
```

Instance of the StateManager class used to manage state of the device in EEPROM.

5.30.4.4 gates

```
Gates gates = Gates(pins, numPins)
```

5.30.4.5 inModeSelection

```
bool inModeSelection = false
```

5.30.4.6 inputHandler

```
InputHandler inputHandler = InputHandler(CV_A_PIN, CV_B_PIN, RESET_PIN, CLOCK_PIN)
```

Instance of the LEDController class.

5.30.4.7 internalPPQN

```
unsigned char internalPPQN = 24
```

5.30.4.8 isInSelection

```
bool isInSelection = false
```

5.30.4.9 ledController

```
LEDController ledController(leds, TEMPO_LED) (
            leds ,
            TEMPO_LED  )
```

Instance of the ResetButton class.

5.30.4.10 ledPins

```
std::vector<int> ledPins = {PA12, PA11, PB1, PB0, PA7, PA6, PA1, PA0}
```

5.30.4.11 leds

```
LEDs leds = LEDs(ledPins, numLedPins)
```

5.30.4.12 modeDivisions

```
ModeDivisions modeDivisions(stateManager, encoder, inputHandler, gates, ledController, midiInterface,
resetButton, clock) (
            stateManager ,
            encoder ,
            inputHandler ,
            gates ,
            ledController ,
            midiInterface ,
            resetButton ,
            clock  )
```

Pointer to the previous mode.

5.30.4.13 modeInverse

```
ModeInverse modeInverse(stateManager, encoder, inputHandler, gates, ledController, midiInterface,
resetButton) (
             stateManager ,
             encoder ,
             inputHandler ,
             gates ,
             ledController ,
             midiInterface ,
             resetButton  )
```

Instance of ModeMidiLearn class.

5.30.4.14 modeLogic

```
ModeLogic modeLogic(stateManager, encoder, inputHandler, gates, ledController, midiInterface,
resetButton) (
             stateManager ,
             encoder ,
             inputHandler ,
             gates ,
             ledController ,
             midiInterface ,
             resetButton  )
```

Instance of ModeInverse class.

5.30.4.15 modeMidiLearn

```
ModeMidiLearn modeMidiLearn(stateManager, encoder, inputHandler, gates, ledController, midiInterface,
resetButton) (
             stateManager ,
             encoder ,
             inputHandler ,
             gates ,
             ledController ,
             midiInterface ,
             resetButton  )
```

Instance of ModeDivisions class.

5.30.4.16 modeSelector

```
ModeSelector& modeSelector = ModeSelector::getInstance()
```

5.30.4.17 musicalIntervals

```
std::vector<int> musicalIntervals = {1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 72, 96, 128,
144, 192, 288, 384, 576, 768, 1152, 1536}
```

5.30.4.18 musicalIntervalsSize

```
const int musicalIntervalsSize = musicalIntervals.size()
```

5.30.4.19 numLedPins

```
int numLedPins = ledPins.size()
```

5.30.4.20 numPins

```
const int numPins = pins.size()
```

5.30.4.21 pins

```
std::vector<int> pins = {PA15, PB3, PB4, PB5, PB6, PB7, PB8, PB9}
```

5.30.4.22 previousMode

```
Mode* previousMode = nullptr
```

Pointer to the current mode.

5.30.4.23 resetButton

```
ResetButton resetButton = ResetButton(RESET_BUTTON)
```

Instance of the Encoder class.

5.30.4.24 stateManager

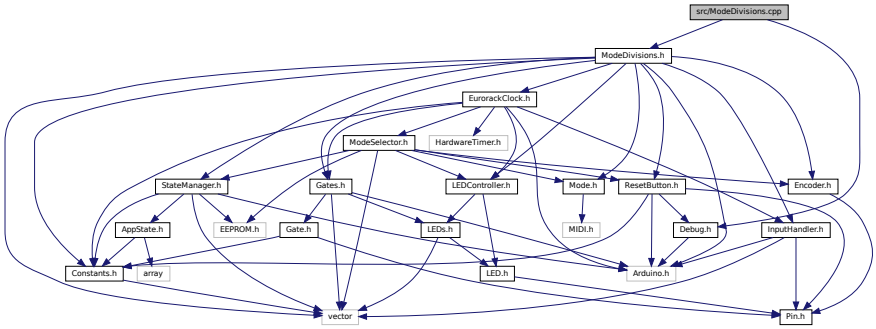
```
StateManager stateManager = StateManager()
```

5.31 src/ModeDivisions.cpp File Reference

Implementation file for ModeDivisions, Please see ModeDivisions.h for more information.

```
#include "ModeDivisions.h"
#include "Debug.h"
```

Include dependency graph for ModeDivisions.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.31.1 Detailed Description

Implementation file for ModeDivisions, Please see ModeDivisions.h for more information.

5.31.2 Macro Definition Documentation

5.31.2.1 DEBUG_PRINT

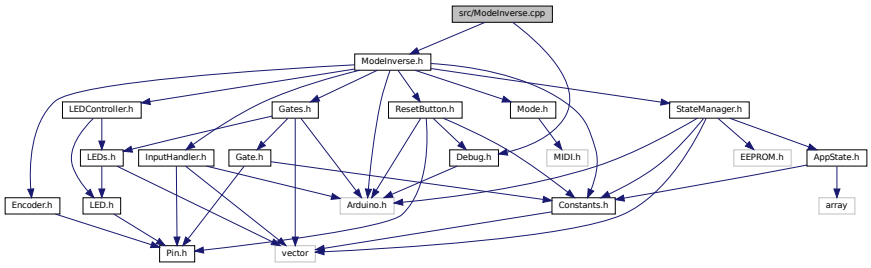
```
#define DEBUG_PRINT(
             message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.32 src/ModeInverse.cpp File Reference

This file contains the implementation of the ModeInverse class, which is used to manage the second mode of the Eurorack module.

```
#include "ModeInverse.h"
#include "Debug.h"
```

Include dependency graph for ModeInverse.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.32.1 Detailed Description

This file contains the implementation of the ModeInverse class, which is used to manage the second mode of the Eurorack module.

5.32.2 Macro Definition Documentation

5.32.2.1 DEBUG_PRINT

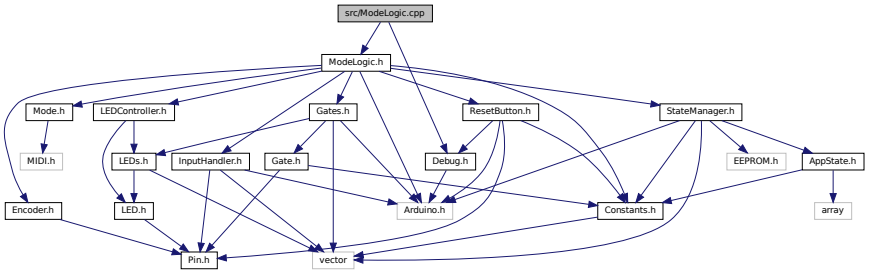
```
#define DEBUG_PRINT(
             message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.33 src/ModeLogic.cpp File Reference

This file contains the implementation of the ModeLogic class, which is used to manage the second mode of the Eurorack module.

```
#include "ModeLogic.h"
#include "Debug.h"
```

Include dependency graph for ModeLogic.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.33.1 Detailed Description

This file contains the implementation of the ModeLogic class, which is used to manage the second mode of the Eurorack module.

5.33.2 Macro Definition Documentation

5.33.2.1 DEBUG_PRINT

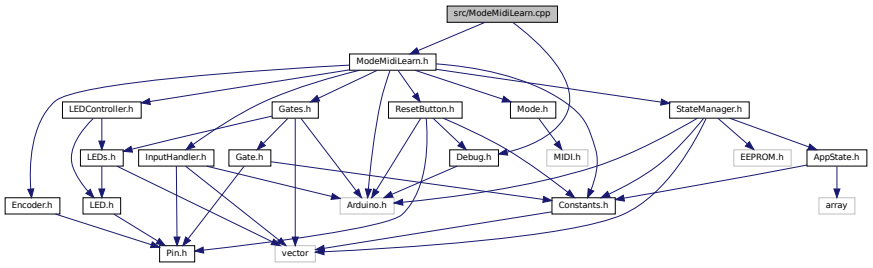
```
#define DEBUG_PRINT(
            message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.34 src/ModeMidiLearn.cpp File Reference

This file contains the implementation of the ModeMidiLearn class, which is used to manage the second mode of the Eurorack module.

```
#include "ModeMidiLearn.h"
#include "Debug.h"
```

Include dependency graph for ModeMidiLearn.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.34.1 Detailed Description

This file contains the implementation of the ModeMidiLearn class, which is used to manage the second mode of the Eurorack module.

5.34.2 Macro Definition Documentation

5.34.2.1 DEBUG_PRINT

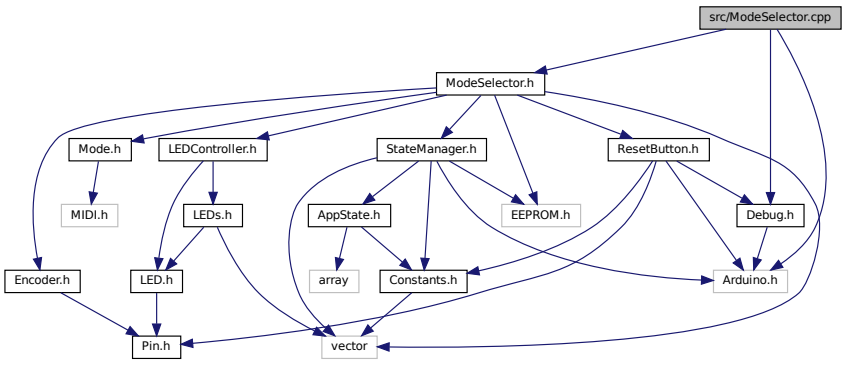
```
#define DEBUG_PRINT(
             message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.35 src/ModeSelector.cpp File Reference

This file contains the implementation of the ModeSelector class, which is used to manage the different modes of the Eurorack module. ModeSelector is a singleton class that is used to manage the different modes of the Eurorack module. It is responsible for handling the mode selection state, button presses, and encoder rotation.

```
#include "ModeSelector.h"
#include <Arduino.h>
#include "Debug.h"
```

Include dependency graph for ModeSelector.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.35.1 Detailed Description

This file contains the implementation of the ModeSelector class, which is used to manage the different modes of the Eurorack module. ModeSelector is a singleton class that is used to manage the different modes of the Eurorack module. It is responsible for handling the mode selection state, button presses, and encoder rotation.

5.35.2 Macro Definition Documentation

5.35.2.1 DEBUG_PRINT

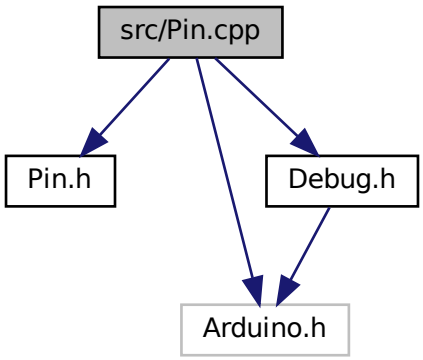
```
#define DEBUG_PRINT(
            message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.36 src/Pin.cpp File Reference

This file contains the implementation of the Pin class and its derived classes.

```
#include "Pin.h"
#include <Arduino.h>
#include "Debug.h"
```

Include dependency graph for Pin.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.36.1 Detailed Description

This file contains the implementation of the Pin class and its derived classes.

5.36.2 Macro Definition Documentation

5.36.2.1 DEBUG_PRINT

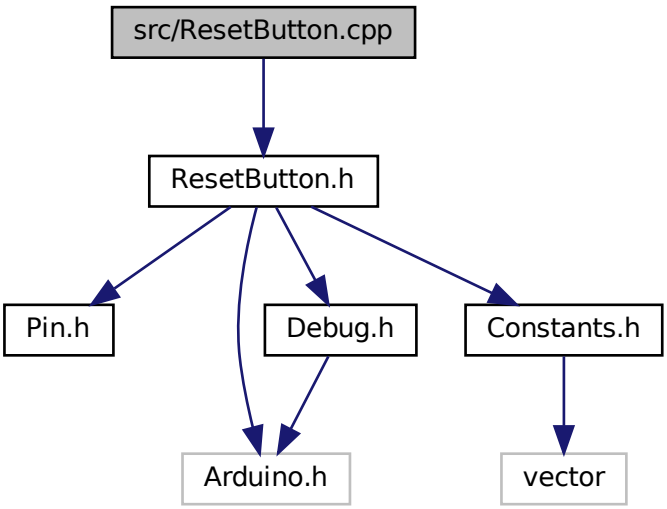
```
#define DEBUG_PRINT(
             message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

5.37 src/ResetButton.cpp File Reference

This file contains the implementation of the ResetButton class, which is used to manage the reset button of the Eurorack module.

```
#include "ResetButton.h"
```

Include dependency graph for ResetButton.cpp:



5.37.1 Detailed Description

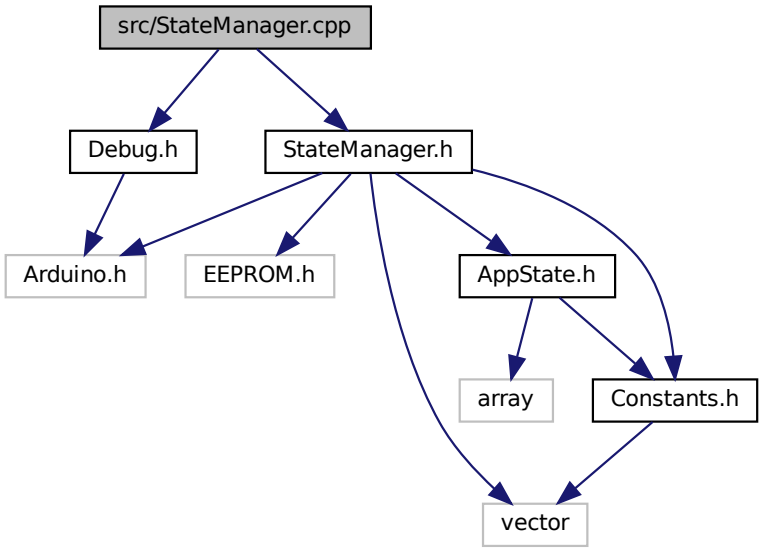
This file contains the implementation of the ResetButton class, which is used to manage the reset button of the Eurorack module.

5.38 src/StateManager.cpp File Reference

"This class manages reading and writing state to the EEPROM memory."

```
#include "StateManager.h"
#include "Debug.h"
```

Include dependency graph for StateManager.cpp:



Macros

- #define DEBUG_PRINT(message) Debug::print(__FILE__, __LINE__, __func__, String(message))

5.38.1 Detailed Description

"This class manages reading and writing state to the EEPROM memory."

5.38.2 Macro Definition Documentation

5.38.2.1 DEBUG_PRINT

```
#define DEBUG_PRINT(
            message ) Debug::print(__FILE__, __LINE__, __func__, String(message))
```

Debug macro