MTD2A_binary_input

MTD2A: Model **T**rain **D**etection **A**nd **A**ction **– Arduino** library https://github.com/MTD2A/MTD2A
Jørgen Bo Madsen / V1.3.5 / 18-10-2026

Model Train Detection And Action is a collection of user friendly advanced and functional C++ building blocks for time controlled handling of input and output, using a state machine system for parallel processing. The library is intended for Arduino enthusiasts without much programming experience who are interested in electronics control and automation, as well as model trains.

Common to all building blocks:

- Build on a state machine system for parallel processing and synchronous timing
- They support a wide range of input sensors and output devices
- Are simple to use to build complex solutions with few commands
- Operate non-blocking, process-oriented and state-driven
- Offers extensive control and troubleshooting information
- Thoroughly documented with examples

Table of contents

MTD2A_binary_input 1
Feature Description 2
Input detection and activation 4
Pin Input mode 5
Simple binary function 6
Time delay – first trigger 7
Time delay – last trigger 8
Monostable – first trigger 9
Monostable – last trigger 10
Examples of configuration 10
Contact bouncing and debouncing 11
Button pressed for duration 12
Short and long button press 13
Repeated button presses 14
Falling and rising edge 15
Rapid button presses 15
Get and set functions 16
print_conf(); 17

Feature Description

MTD2A_binary_input process consists of 3 functions:

1. MTD2A_binary_input object_name ("object_name" , delayTimeMS, { LAST_TRIGGER | FIRST_TRIGGER }, {TIME_DELAY | MONO_STABLE}, pinBlockTimeMS);

2. object_name.initialize (pinNumber, {NORMAL | INVERTED}, { INPUT_PULLUP | INPUT });
   Called once in void setup (); and after Serial.begin ("Velocity");

3. MTD2A_loop_execute (); Called as the last instruction in void loop ();

All functions use default values and can therefore be called with none and up to the maximum number of parameters. However, the parameter must be specified in ascending order. See example below:

```
MTD2A_binary_input object_name;
MTD2A_binary_input object_name ("object_name");
MTD2A_binary_input object_name ("object_name" , delayTimeMS);
MTD2A_binary_input object_name ("object_name" , delayTimeMS, triggerMode);
MTD2A_binary_input object_name ("object_name" , delayTimeMS, triggerMode, timerMode);
MTD2A_binary_input object_name ("object_name" , delayTimeMS, triggerMode, timerMode, pinBlockTimeMS);
Default: (                              "Object name" , 0              , LAST_TRIGGER,  TIME_DELAY,  0);
```

Example

```
// Read sensor and write phase state information to Arduino IDE serial monitor
// https://github.com/MTD2A/FC-51
#include <MTD2A.h>
using namespace MTD2A_const;

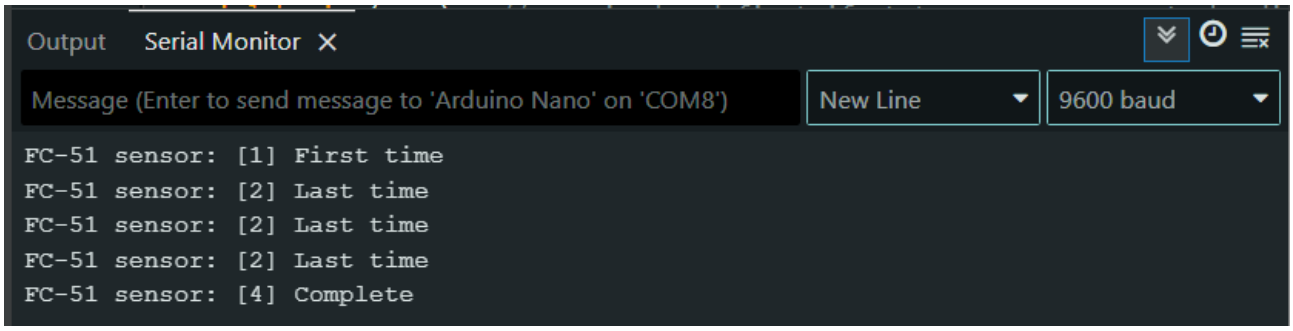
MTD2A_binary_input FC_51_sensor ("FC_51 sensor", 5000);
// "FC-51 sensor" = Sensor (object) name, which is displayed together with status messages
// 5000 = Time delay in milliseconds (5 seconds)
// default: LAST_TRIGGER = Start calculating time from last impulse (LOW->HIGH)
// default: TIME_DELAY = Use time delay (timer function)

void setup () {
  Serial.begin (9600); // Required and first if status messages are to be displayed
  while (!Serial) { delay(10); } // ESP32 Serial Monitor ready delay

  byte FC51_SENSOR_PIN = 2;
  FC_51_sensor.initialize (FC51_SENSOR_PIN); // Arduino board pin 2 input.
  FC_51_sensor.set_debugPrint (); // Display status messages
}

void loop () {
  MTD2A_loop_execute ();
}
```

Sample printout for IDE Serial Monitor:

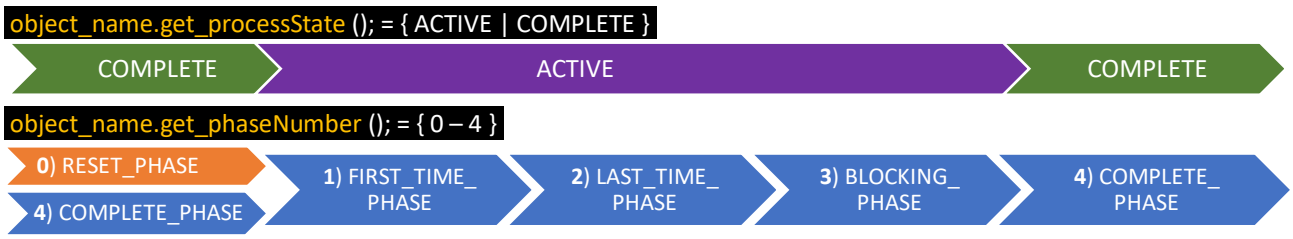


More examples and youtube demo video:

https://github.com/MTD2A/MTD2A/tree/main/examples
DEMO video: https://youtu.be/RDFgEbhYUzE

Process phases

Depending on the current configuration, the process is carried out in between 3 and 5 stages.



0. 0) when the function reset (); is called. 4) The initial phase when the program starts.
1. The first time that there was a change in the input (sensor or the program itself).
2. Last time there was a change in input. May occur several times.
3. Blocking input from the sensor or the program itself for a timed period.
4. Awaiting new input (change of state) from the sensor or the program itself.

Global number constants:

RESET_PHASE, FIRST_TIME_PHASE, LAST_TIME_PHASE, BLOCKING_PHASE & COMPLETE_PHASE

The instantaneous phase shift can be identified by the function: object_name.get_phaseChange (); = { true | false }

Process status

When transitioning to FIRST_TIME_PHASE or LAST_TIME_PHASE, ProcessState switches to ACTIVE.
When transitioning to COMPLETE_PHASE, the processState switches to COMPLETE.

Timing

The time periods are set by default when the object is instantiated (activated).
It is possible to define new time periods for both timers:
object_name.set_delayTimeMS ({0 – MAX_TIME_MS});
object_name.set_pinBlockMS ({0 – MAX_TIME_MS});

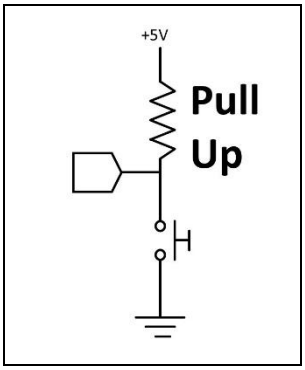
See the document MTD2A.PDF and the section "Cadence", "Synchronization" as well as "Execution speed".

Input detection and activation

MTD2A_binary_input read input from

1) Digital Pin connection on the Arduino Board
2) The program itself.

Input from digital pin connection on the Arduino board is configured with INPUT_PULLUP by default. This means that a resistor of typically 10 K ohms is connected between the input pin connection and + (plus) = HIGH. Activation is done by connecting the pin connection to – (minus) = LOW.
See more here: INPUT | INPUT_PULLUP | OUTPUT | Arduino Documentation



All types of switches, relays and all kinds of circuits can be used with the Open Collector transistor NPN - binary and analogue. In analog circuitry, the status changes according to the voltage levels as described here: HIGH | LOW | Arduino Documentation

If the input circuit also uses its own pullup resistor, it will generally work as it should. Otherwise, INPUT_PULLUP can be deselected by entering INPUT in the function:
`object_name.initialize ( pinNumber, {NORMAL | INVERTED}, {INPUT_PULLUP | INPUT } );`.

There are two possible inputs to the function:

1. Input from digital leg connection on Arduino board
   `pinState` => {HIGH | LOW} pin read only.
2. Input from the program itself
   `inputState` = {HIGH | LOW} write & read.

pinState	inputState	CurrState
HIGH	HIGH	HIGH
HIGH	LOW	LOW
LOW	HIGH	LOW
LOW	LOW	LOW

Input is read from the digital pin connection number specified in `object_name.initialize ( pinNumber );`
If the function is not called, the pin connection will not be read `pinReadToggl = disable` and `pinNumber` = 255.

`object_name.initialize ( pinNumber );` Can only be called **once** and only in Arduino `void setup ();`

If the pin connection is initialized correctly with the above function, it is possible to continuously control whether the pin connection should be read or not with the function:
`object_name.set_pinReadToggl ( {ENABLE | DISABLE} );`

During `BLOCKING_PHASE`, pin reading is temporarily suspended by the process itself. A call to `object_name.set_pinReadToggl ( {ENABLE | DISABLE} );` in this phase does not interrupt the active blocking period — the setting is stored and takes effect when the blocking period ends.

Input can also come from the program itself:
`object_name.set_inputState ( {HIGH | LOW}, {PULSE | FIXED} );`

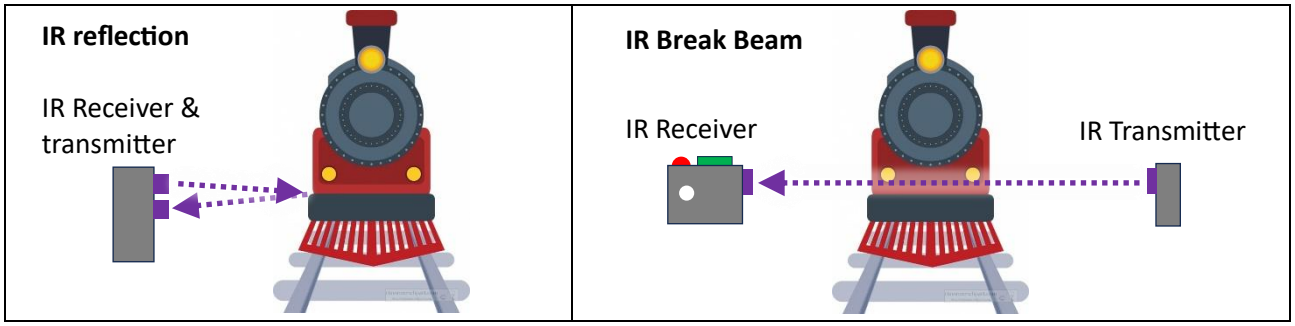
Pulse indicates a single pulse (short monostable) and fixed works permanently and until Pulse is specified.

Pin Input mode

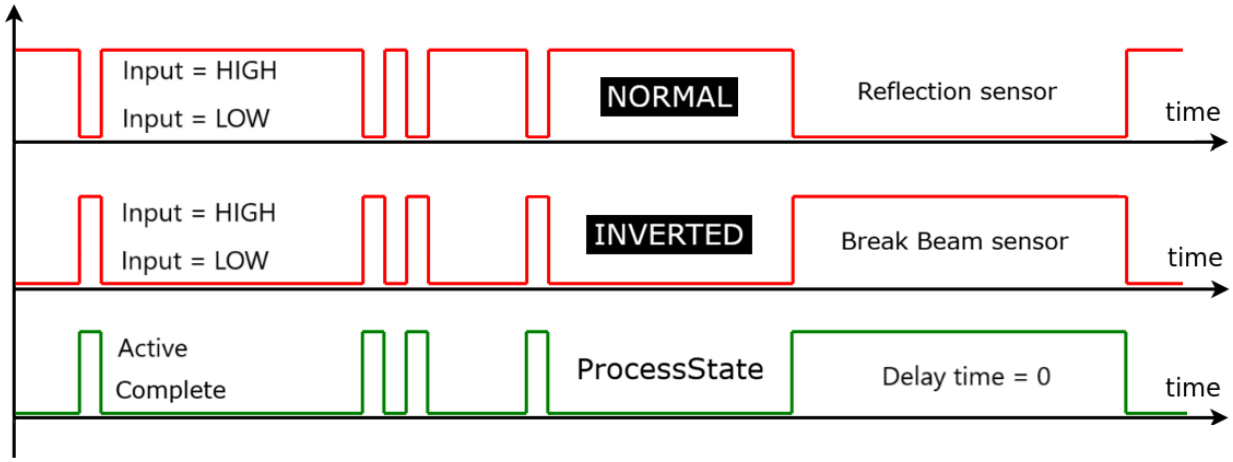
There are two ways to read input:

1. NORMAL Trigger occurs at HIGH -> LOW. The output mirrors the input. For example, a reflection sensor.
2. INVERTED Trigger occurs at LOW -> HIGH. Output follows input. E.g. break beam sensor.

Default normal `object_name.initialize (pinNumber);` or `object_name.initialize (pinNumber, INVERTED);`



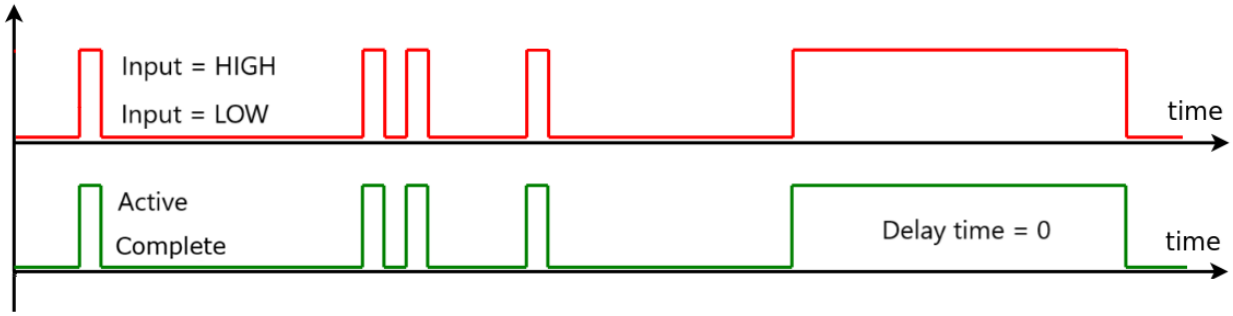
The following examples are based on the FC-51 binary break beam sensor, where the transmitter is placed on one side of the train, and the receiver is placed on the other side. This means that the sensor input is LOW in rest mode and HIGH when the train breaks the beam. INVERTED is used.



Simple binary function

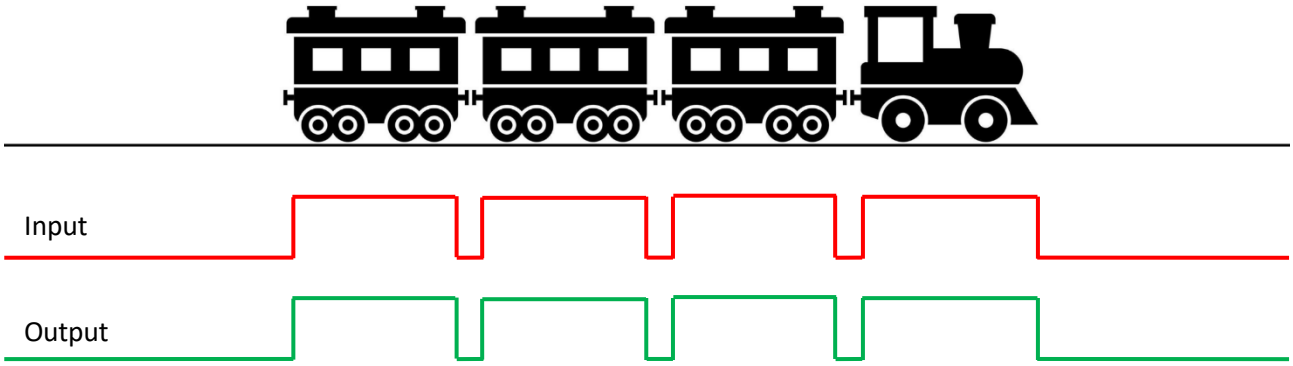
```
MTD2A_binary_input FC_51_sensor ("FC_51_sensor");
```

When the input goes from LOW to HIGH, the output does exactly the same, and vice versa.



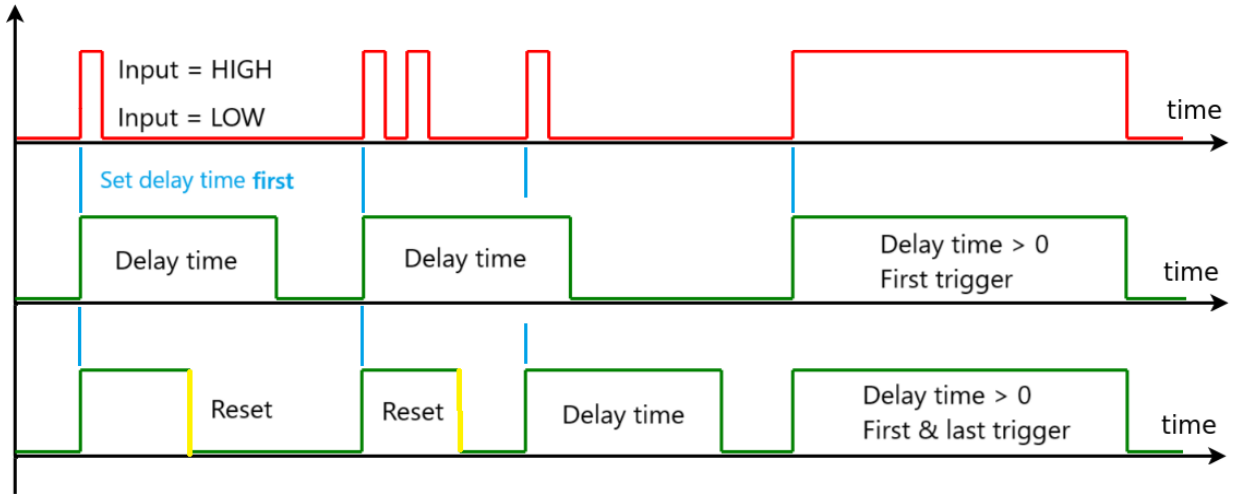
Sensor detection of a moving train set will inevitably cause a number of pulses due to variations in the construction of the train cars and locomotive, as well as "holes" at connections and more. These variations can cause unforeseen reactivation of functions and errors in the subsequent logic process.

Example of moving trains



Time delay – first trigger

When the input goes from LOW to HIGH, the output HIGH is maintained until the defined time period ends. If the input is HIGH at the end of the time period, the output remains HIGH until the input goes from HIGH to LOW.




```
object_name.reset ();
```

Resets all control and process variables and prepares for a fresh start. All functionally configured variables and default values are retained. The process phase switches to RESET_PHASE

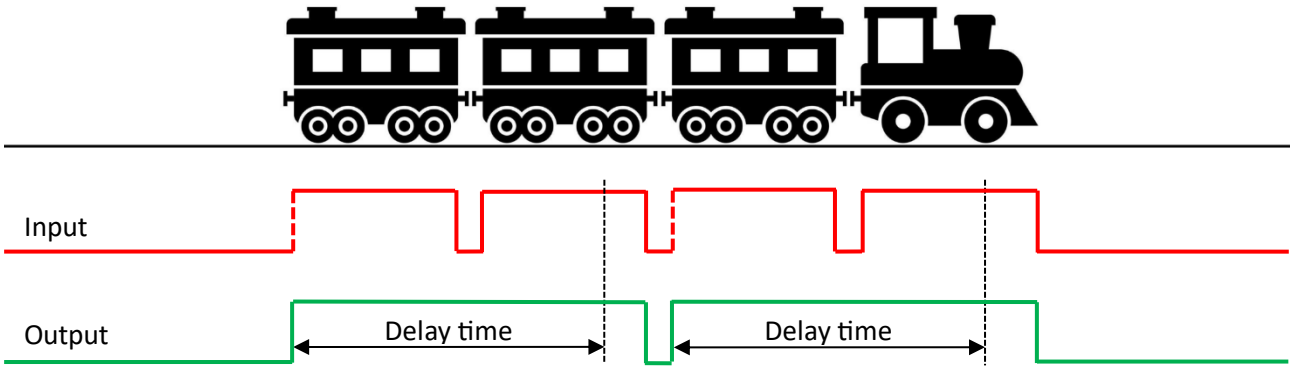
If object_name.reset (); is called during BLOCKING_PHASE, the blocking period is cancelled and pin reading is restored to its setting from before the blocking period.

Example of moving trains

To avoid inappropriate reactivation, delayTimeMS must be long to ensure that it also works on slow-moving trains. In the example below, delayTimeMS is too short, which results in two activations instead of one.

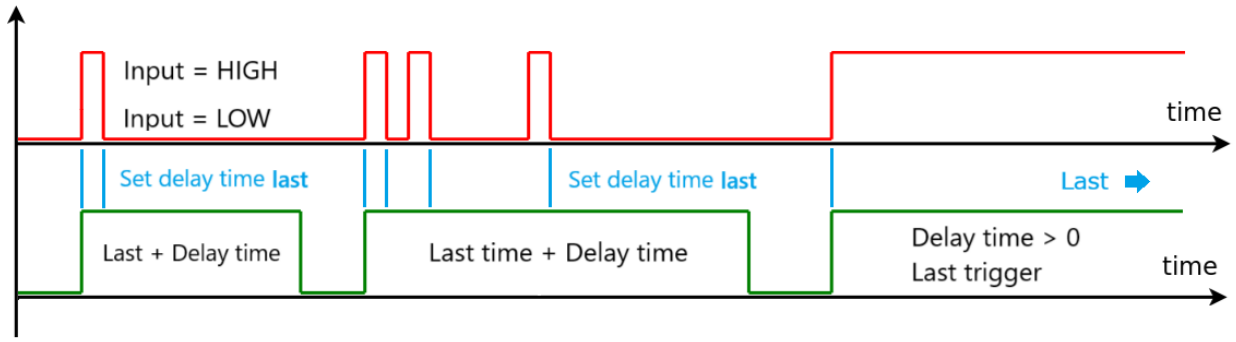
delayTimeMS = 5.000 milliseconds and triggerMode = **FIRST**_TRIGGER.
5 second delay measured from first detection of trains.

```
MTD2A_binary_input FC_51_sensor ("FC_51_sensor", 5000, FIRST_TRIGGER);
```



Time delay – last trigger

When the input goes from LOW to HIGH, the output HIGH is maintained until the defined time period ends. Each time the input goes from HIGH to LOW, the start of the time period is shifted to the new time. If the input is HIGH at the end of the time period, the output remains HIGH until the input goes from HIGH to LOW.

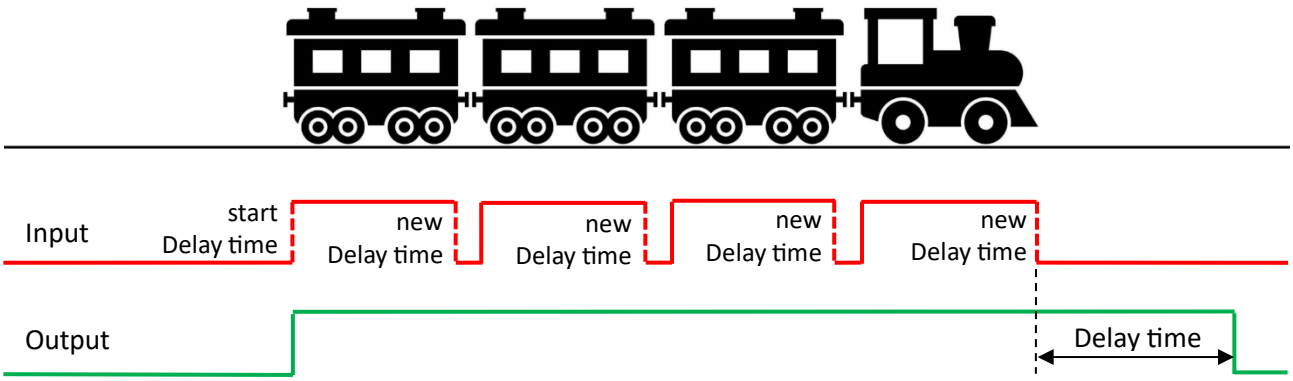


Example of moving trains

This method is best suited for detecting fast and slow-moving trains without inappropriate reactivations.

delayTimeMS = 5.000 milliseconds and triggerMode = **LAST**_TRIGGER.
5 seconds delay measured from the last detection of trains (HIGH to LOW).

```
MTD2A_binary_input FC_51_sensor ("FC_51_sensor", 5000, LAST_TRIGGER);
```

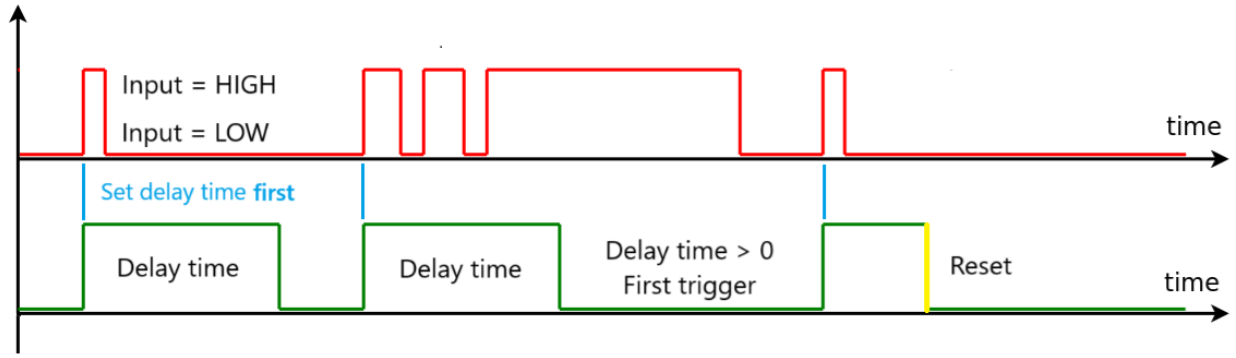


object_name.set_stopDelayTimer ( ); Instantly stops the delay period. In MONO_STABLE mode, the process moves on to the next stage immediately. In TIME_DELAY mode, the same rule applies as at normal time expiry: if the input is HIGH, the output remains HIGH until the input goes from HIGH to LOW.

It requires that the process is ACTIVE and that the corresponding timer is configured (delayTimeMS > 0). Otherwise the call is ignored and error 18 ERR_TIMER_NOT_IN_USE is reported. A stop request only applies to the current process cycle — it is cleared when the process reaches COMPLETE_PHASE and never carries over to the next activation.

Monostable – first trigger

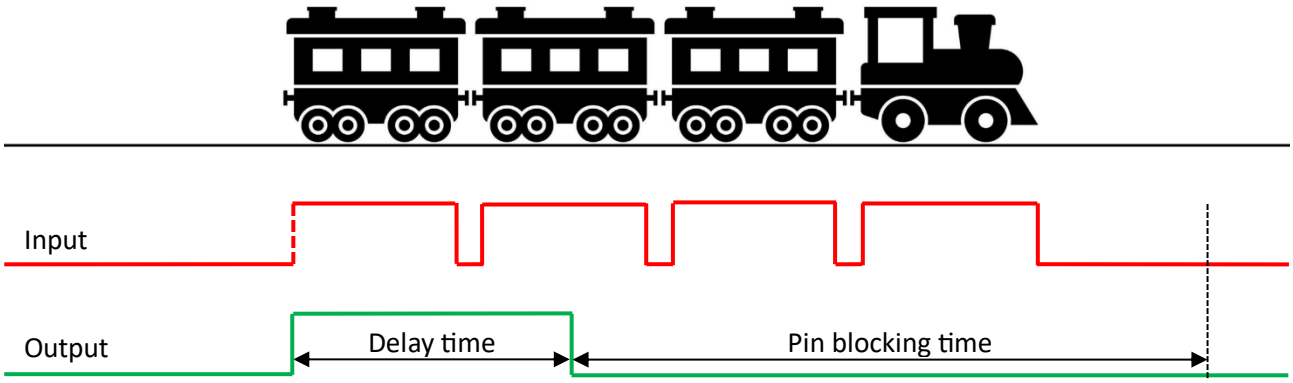
Monostable always maintains the defined time period, regardless of whether the input changes between HIGH and LOW during the time period, and if the input remains either HIGH or LOW, it does not change the time period.



Example of moving trains

delayTimeMS = 5.000 milliseconds, triggerMode = FIRST_TRIGGER, timerMode = MONO_STABLE, pinBlockMS = 12.000 milliseconds (time period where input from the pin connection is blocked from delayTimeMS termination until monostable termination).
5 seconds delay measured from first detection of trains (LOW to HIGH).

```
MTD2A_binary_input FC_51_sensor ("FC_51_sensor", 5000, FIRST_TRIGGER, MONO_STABLE, 12000);
```

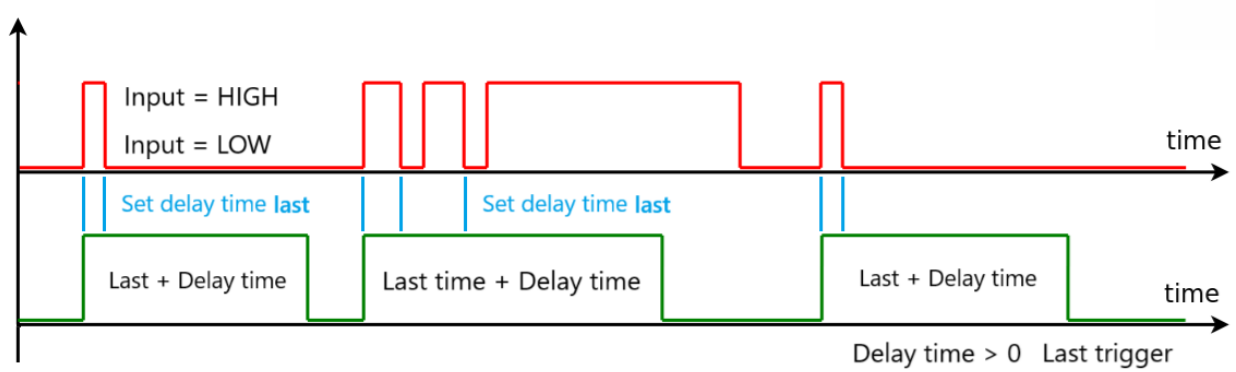


object_name.set_stopBlockTimer ( ); Instantly stops the blocking period and moves on to the next stage.

It requires that the process is ACTIVE and that the corresponding timer is configured (pinBlockTimeMS > 0). Otherwise the call is ignored and error 18 ERR_TIMER_NOT_IN_USE is reported. A stop request only applies to the current process cycle — it is cleared when the process reaches COMPLETE_PHASE and never carries over to the next activation.

Monostable – last trigger

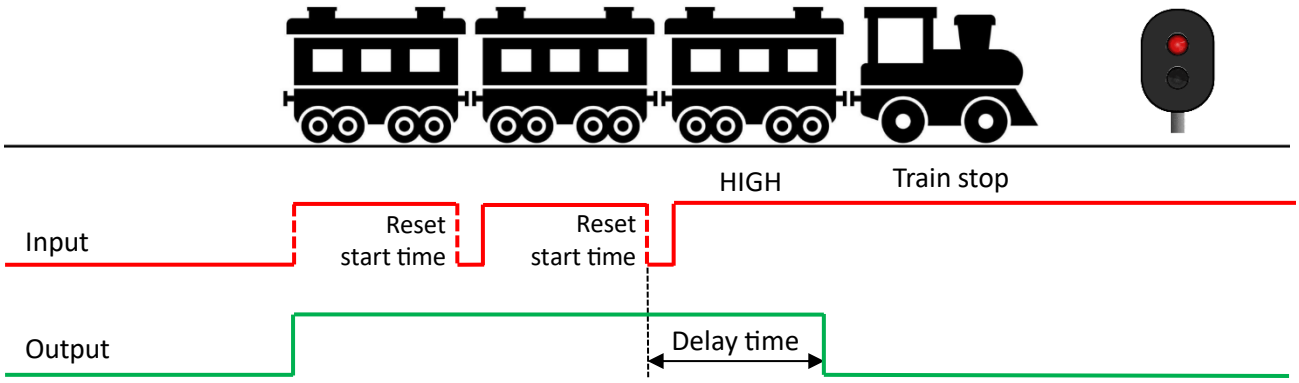
Monostable always maintains the defined time period, but switching inputs between HIGH and LOW during the time period, the time period is extended each time. After the total time period, the output will change to LOW regardless of whether the input is either HIGH or LOW.



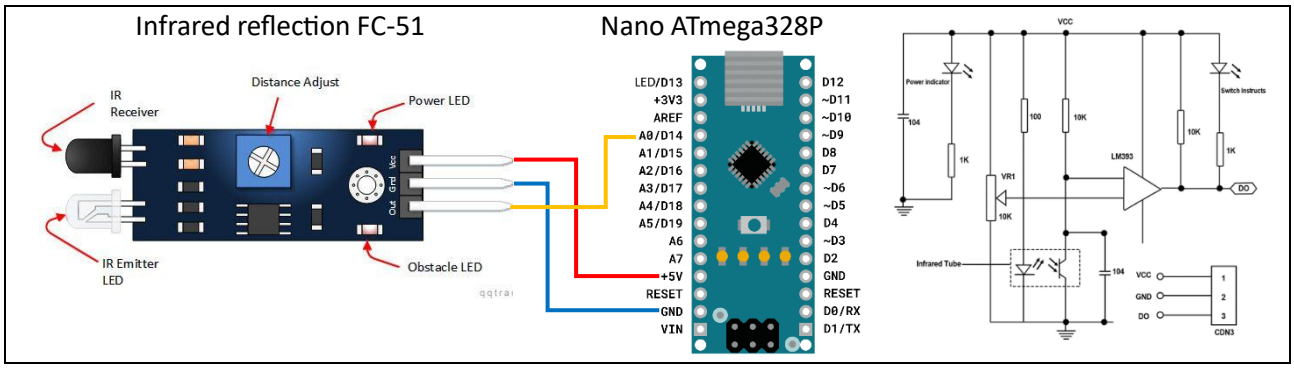
Example of moving train stopping over sensor

delayTimeMS = 3.000 milliseconds, triggerMode = LAST_TRIGGER, timerMode = MONO_STABLE.
3 seconds delay measured from first detection of trains (LOW to HIGH).

```
MTD2A_binary_input FC_51_sensor ("FC_51_sensor", 3000, LAST_TRIGGER, MONO_STABLE);
```



Examples of configuration



1. MTD2A_binary_input FC_51_sensor ("FC_51_sensor", 5000);
2. FC_51_sensor.initialize (A0);
3. FC_51_sensor.set_debugPrint ();
4. MTD2A_loop_execute ();

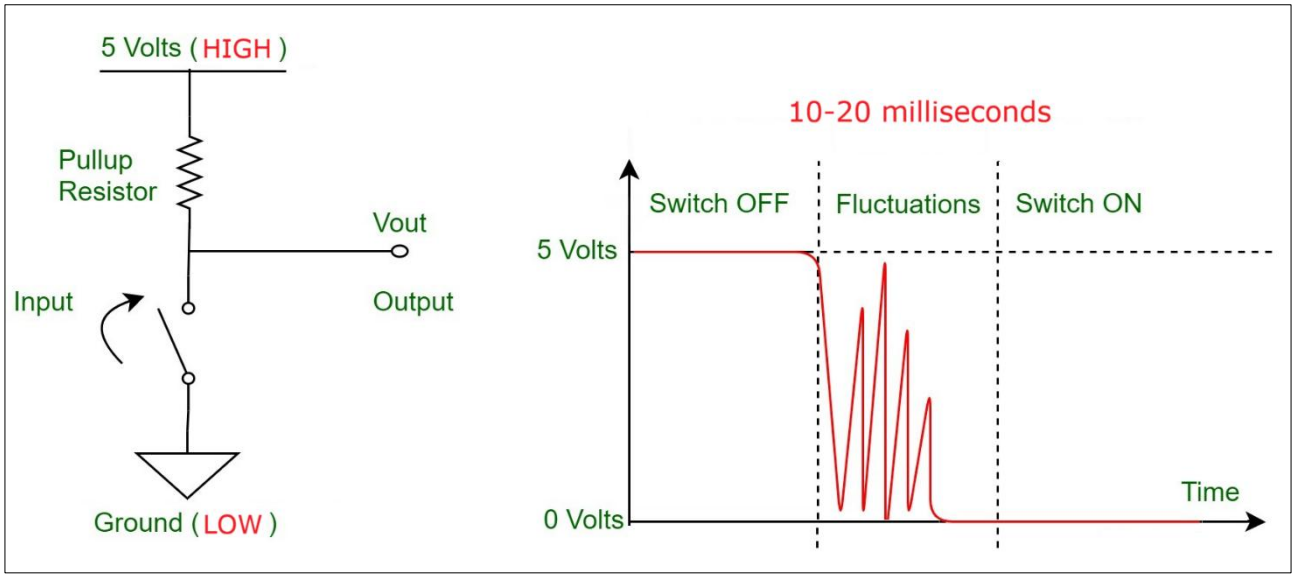
Contact bouncing and debouncing

Contact bouncing (or chatter)—a common problem with mechanical switches, buttons, and relays. As electromechanical devices, switch contact points are subject to momentum and elasticity. These forces act together to create rapid contact oscillations (“bounces”) when a switch is “opened” or “closed”.

The bounce period varies depending on the switch type and switches bounce both when closing and opening. Each switch type has its own “bounce pattern” of rapid oscillations.

The bouncing typically lasts 1–20 ms depending on the switch. Based on several tests and analyses, the assessment is that 50 milliseconds is an appropriate period before the bouncing period is fully stabilized.

The bouncing problem can be solved in software or hardware. E.g. the MAX6816 debouncer IC.



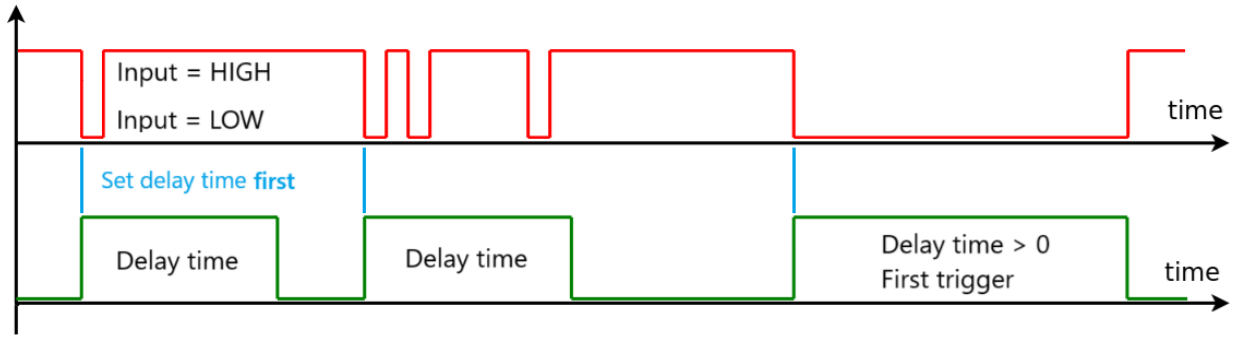
A more detailed analysis report on debouncing by Jack G. Ganssle can be read here: debouncing.pdf

`MTD2A_binary_input ();` handles debouncing by design! MTD2A default `DEBOUNCE_MS` is 50 milliseconds.
See example 1 here: GitHub: MTD2A/examples/push_button

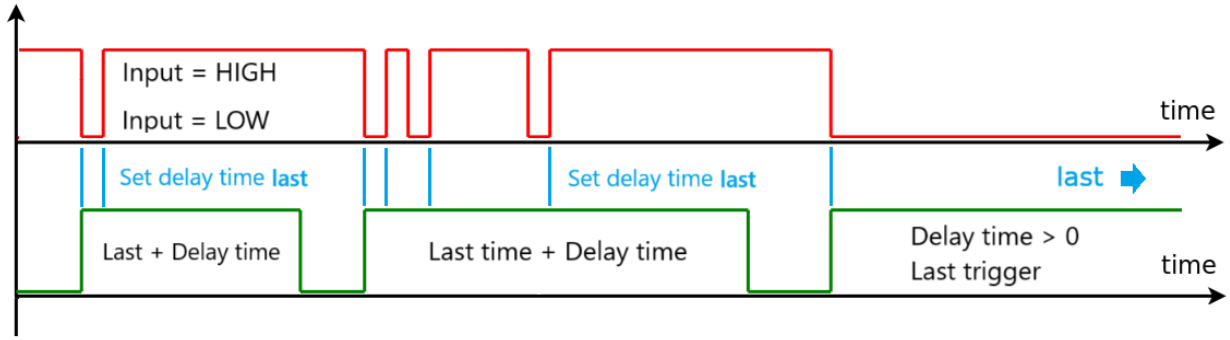
Button pressed for duration

`object_name.get_activeTimeMS ();` Measures the button's activation time in milliseconds. A timer starts when the first button activation is detected. Minimum timing at set by `delayTimeMS`. Timing continues as long as the button is activated.

First trigger



Last trigger




```
MTD2A_binary_input buttonTimer ("Button timer", DEBOUNCE_MS, FIRST_TRIGGER, TIME_DELAY);

void setup() {
  Serial.begin(9600);
  buttonTimer.initialize (pushButton);
}

void loop() {
  if (buttonTimer) {
    Serial.print (F("Button down milliseconds: "));
    Serial.println (buttonTimer.get_activeTimeMS ());
  }
  MTD2A_loop_execute ();
} // loop
```

See examples 2 and 3 here: [Github: MTD2A/examples/push_button]

Short and long button press

The following average time values serve as a good starting point for detecting short and long button presses. However, the time values vary from instance to instance, making it necessary to define a range of values around these averages.

SHORT_PRESS_MS is, based on a series of tests, measured at 200 milliseconds. In practice: > 50 && < 500
LONG_PRESS_MS is, based on a series of tests, measured at 1000 milliseconds. In practice: > 500 && < 1500

Example

```
MTD2A_binary_input buttonTimer ("Button timer", DEBOUNCE_MS, FIRST_TRIGGER, TIME_DELAY);

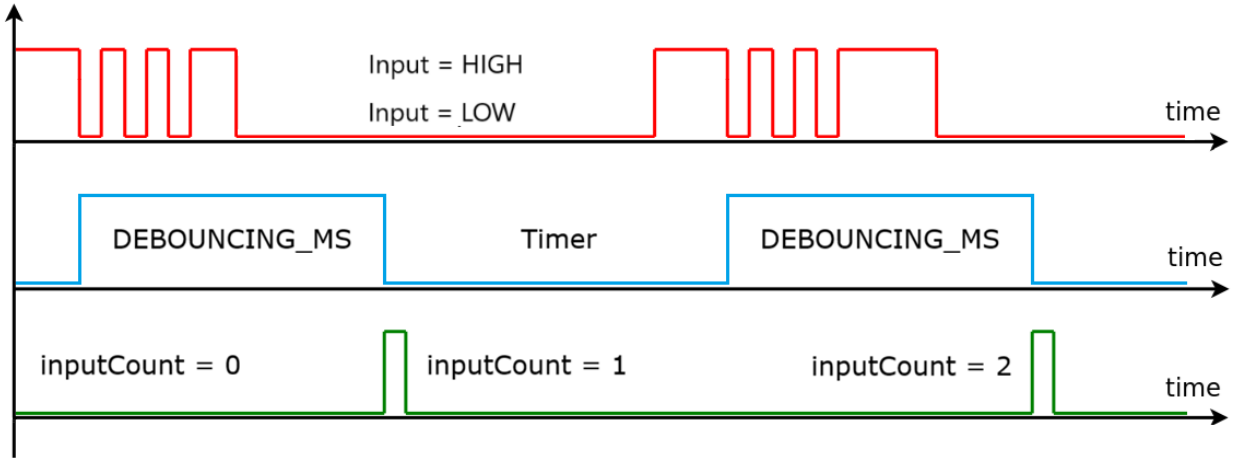
void setup() {
  Serial.begin(9600);
  buttonTimer.initialize (pushButton);
}

void loop() {
  if (buttonTimer) {
    unsigned long activeTimeMS = buttonTimer.get_activeTimeMS ();
    if (activeTimeMS > 50  && activeTimeMS <= 500) {
      Serial.print (F("SHORT button press: ")); Serial.println (activeTimeMS);
    }
    else if (activeTimeMS > 500  && activeTimeMS <= 1500) {
      Serial.print (F("LONG button press: ")); Serial.println (activeTimeMS);
    }
    else if (activeTimeMS > 1500  && activeTimeMS <= 2500) {
      Serial.print (F("EXTRA long button press: ")); Serial.println (activeTimeMS);
    }
  }
  MTD2A_loop_execute ();
} // loop
```

See example 4 here: [Github: MTD2A/examples/push_button]

Repeated button presses

object_name.get_inputCount (); Counts number of button presses and prevents erroneous activations due to debouncing. A timer starts when the initial button activation is detected. After DEBOUNCE_MS milliseconds of stabilization, a check is performed to see if the button is still activated. If this is the case, 1 is added to inputCount. The same process is repeated with subsequent button presses. delayTimeMS must be greater than or equal to DEBOUNCE_MS before counting can take place. Otherwise inputCount is set to 1.



Examples

Count number of button presses within 2 seconds. If the button is held down longer than 2 seconds, the total duration will exceed 2 seconds.

```
MTD2A_binary_input buttonCounter ("Button first press counter", 2000, FIRST_TRIGGER, TIME_DELAY);
```

Count number of button presses within 2 seconds, restarting the timer after each button release.
That makes it possible to press the button up to 255 times, provided each button press is completed within 2 seconds. As a result, the total duration will exceed 2 seconds and the final result depends on user behavior.

```
MTD2A_binary_input buttonCounter ("Button last press counter", 2000, LAST_TRIGGER, TIME_DELAY);
```

Immediately detects exactly one button press and set inputCount to 1. The process will restart when the button is released. If the button is held down, the total duration will increase until button is released.

```
MTD2A_binary_input buttonCounter ("One fast button press", 0);
```

Example

```
void setup() {
  Serial.begin(9600);
  buttonCounter.initialize (pushButton);
}

void loop() {
  if (buttonCounter) {
    Serial.print   (F("Number of button presses: "));
    Serial.println (buttonCounter.get_inputCount ());
  }
  MTD2A_loop_execute ();
} // loop
```

In MONO_STABLE mode a button press within the last DEBOUNCE_MS of the delay is not counted.

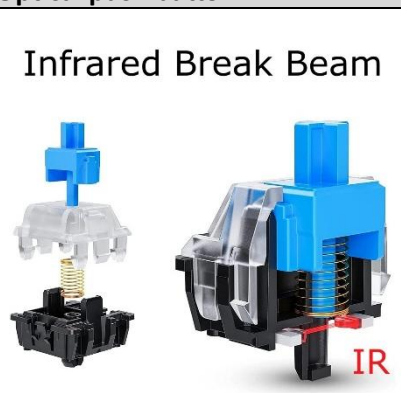
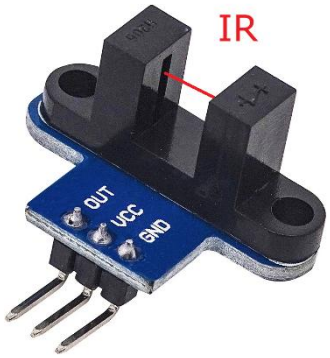
Falling and rising edge

Button is pressed: `get_inputGoLow ();` Return bool {true | false} True only in one loop cycle.
Button is released: `get_inputGoHigh ();` Return bool {true | false} True only in one loop cycle.

Rapid button presses

`object_name.get_inputCount ();` does not support rapid button presses due to `DEBOUNCE_MS` time to stabilization. The input will be interpreted incorrectly and the result will be between 0 and the number of button presses. Counting after `DEBOUNCE_MS` milliseconds ensures stability, noise cancellation and rejection of unrealistically fast switching.

If it is necessary to detect rapid button presses, optical buttons and switches can be used. Optical switches use light beams instead of metal contacts to provide near-instantaneous response times, zoro debounce delay, and exceptional durability of over 100 million button presses. Optical switches are used in, among other things, some types of keyboards.

Optical push button	**Optical micro switch**	**Optical slotted sensor**
		

Get and set functions

Set functions	Comment
set_pinReadToggl ({ENABLE | DISABLE});	Enable or disable pin reading
set_pinReadMode ({NORMAL | INVERTED});	Configure pin trigger input
set_inputState ({HIGH | LOW}, {PULSE | FIXED});	Activate input state and set input mode
set_delayTimeMS ({0 - MAX_TIME_MS });	Set new delay time after instantiation
set_pinBlockMS ({0 - MAX_TIME_MS });	Set new blocking time after instantiation
set_stopDelayTimer ();	Stop first and last timer process immediately.
set_stopBlockTimer ();	Stop blocking timer process immediately.
set_debugPrint ({ENABLE | DISABLE});	Enable print phase number and text
set_errorPrint ({ENABLE | DISABLE});	Enable error messages

Get functions	Comment
get_processState (); return bool {ACTIVE | COMPLETE}	Process state
get_pinState (); return bool {HIGH | LOW}	Current pin input state
get_phaseChange (); return bool {true | false}	Momentary phase change (one loop time)
get_phaseNumber (); return uint8_t {0 - 4}	Reset = 0, firstTime = 1, lastTime = 2, blocking = 3, complete = 4
get_firstTimeMS (); return uint32_t milliseconds.	First time trigger time
get_lastTimeMS (); return uint32_t milliseconds	Last time trigger time (LAST_TRIGGER only)
get_endTimeMS (); return uint32_t milliseconds.	End time (sync time at completion)
get_activeTimeMS (); return uint32_t milliseconds	Push button down time + DEBOUNCE_MS
get_inputCount (); return uint8_t {0 - 255}	Number of button presses
get_inputGoLow (); return bool {true | false}	Falling edge detected
get_inputGoHigh (); return bool {true | false}	Rising edge detected
get_reset_error (); return uint8_t {0-255}	Get error/warning number and reset number: Error [1 – 127] warning [128 – 255]

Operator overloading	Function
object_name & !objectName e.g. if (objectName) { activate };	bool phaseChange == true && phaseNumber == COMPLETE_PHASE
object_name_1 == object_name_2	bool processState_1 == processState_2
object_name_1 != object_name_2	Bool processState_1 != processState_2
object_name_1 > object_name_2	bool processState_1 == ACTIVE && processState_2 == COMPLETE
object_name_1 < object_name_2	bool processState_1 == COMPLETE && processState_2 == ACTIVE

print_conf();

```
object_name.print_conf ();
```

```
MTD2A_binary_input:
-------------------
  objectName    : Left
  processState  : ACTIVE
  phaseText     : [2] Last time
  debugPrint    : ENABLE
  globalDebugPr: DISABLE
  errorPrint    : DISABLE
  globalErrorPr: ENABLE
  errorNumber   : 0 OK
  triggerMode   : LAST_TRIGGER
  timerMode     : TIME_DELAY
  pinNumber     : 2
  pinType       : INPUT_PULLUP
  pinReadToggl  : ENABLE
  pinReadMode   : NORMAL
  inputMode     : PULSE
  inputCount    : 3
  activeTimeMS  : 12310
  delayTimeMS   : 10000
  firstTimeMS   : 4517
  lastTimeMS    : 7919
  endTimeMS     : 0
  blockTimeMS   : 0
  pinState      : HIGH
  inputState    : HIGH
```