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 er en samling af brugervenlige, avancerede og funktionelle C++ byggeklodser til tidsstyret håndtering af input og output ved hjælp af et tilstandsmaskinesystem til parallel behandling. Biblioteket er beregnet til Arduino-entusiaster uden megen programmeringserfaring, der er interesserede i elektronisk styring og automatisering, samt modeltog.

Fælles for alle byggeklodser:

- Bygger på et tilstandsmaskine system til parallel processering og synkron timing
- Understøtter en bred vifte af inputsensorer og outputenheder
- Er enkle at bruge til at bygge komplekse løsninger med få kommandoer
- Fungerer ikke-blokerende, procesorienterede og tilstandsdrevne
- Tilbyder omfattende information om styring og fejlfinding
- Grundigt dokumenteret med eksempler

Indholdsfortegnelse

MTD2A_binary_input 1
Funktionsbeskrivelse 2
Input detektering og aktivering 4
Pin Input mode 5
Simpel binær funktion 6
Time delay – first trigger 7
Time delay – last trigger 8
Monostable – first trigger 9
Monostable – last trigger 10
Eksempel på configuration 10
Kontaktprel og anti-prel 11
Knap holdes nede 12
Kort og lang knaptryk 13
Gentagne knaptryk 14
Faldende og stigende flanke 15
Hurtige knaptryk 15
Set og get funktionsoversigt 16
Print_conf(); 17

Funktionsbeskrivelse

MTD2A_binary_input processen består af 3 funktioner:

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 });
   Kaldes en gang i void setup (); og efter Serial.begin (”Hastighed”);

3. MTD2A_loop_execute (); Kaldes som det sidste I void loop ();

Alle funktioner benytter default værdier og kan derfor kaldes med ingen og op til max antal parametre. Dog skal parametre angives i stigende rækkefølge startende fra den første. Se eksempel herunder:

```
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);
```

Eksempel

```
// 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;

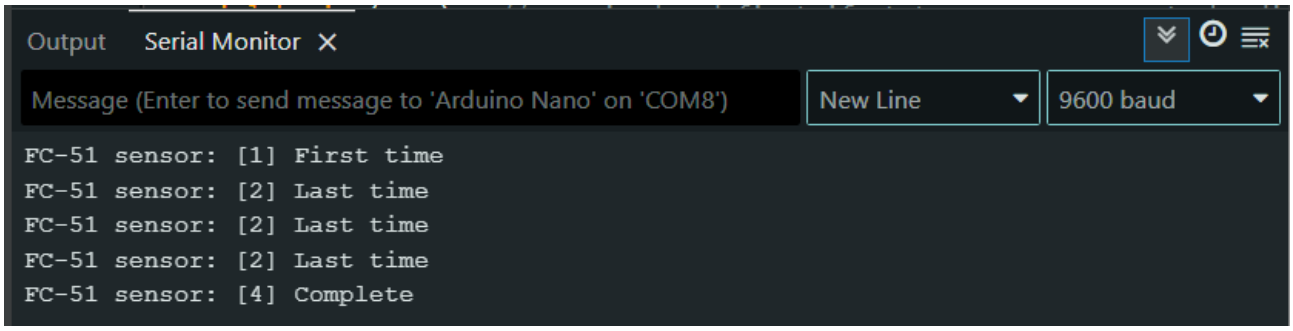
MTD2A_binary_input FC_51_sensor ("FC_51 sensor", 5000);
// "FC-51 sensor" = Sensor (object) name, which is displayed together with status messages
// 5000 = Time delay in milliseconds (5 seconds)
// default: LAST_TRIGGER = Start calculating time from last impulse (LOW->HIGH)
// default: TIME_DELAY =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

Eksempel på udskrift til IDE Serial Monitor:



Flere eksempler og youtube demo video:

https://github.com/MTD2A/MTD2A/tree/main/examples
DEMO video: https://youtu.be/eyGRazX9Bko

Proces faser

Afhængig af den aktuelle konfiguration gennemføres processen i mellem 3 og 5 faser.

object_name.get_processState (); = { ACTIVE | COMPLETE }



object_name.get_phaseNumber (); = { 0 – 4 }



0. 0) Når funktion object_name.reset (); kaldes. 4) Den initelle fase når programmet starter.
1. Første gang at der er sket en ændring i input (sensor eller programmet selv).
2. Sidste gang at der skete en ændring i input. Kan forekommer flere gange.
3. Blokering af input fra sensor eller programmet selv i en tidsbestemt periode.
4. Afventer ny input (tilstandsændring) fra sensor eller programmet selv.

Globale nummerkonstanter:
RESET_PHASE, FIRST_TIME_PHASE, LAST_TIME_PHASE, BLOCKING_PHASE & COMPLETE_PHASE

Det øjeblikkelige faseskift kan identificeres med funktion: object_name.get_phaseChange (); = { true | false }

Proces status

Ved overgang til FIRST_TIME_PHASE eller LAST_TIME_PHASE skifter ProcessState til ACTIVE.
Ved overgang til COMPLETE_PHASE skifter processState til COMPLETE.

Timing

Tidsperiodene sættes som standard når objektet instantieres (aktiveres).
Det er muligt at definere nye tidsperioder for begge timere:

object_name.set_delayTimeMS ({0 - MAX_TIME_MS });

object_name.set_pinBlockMS ({0 - MAX_TIME_MS });

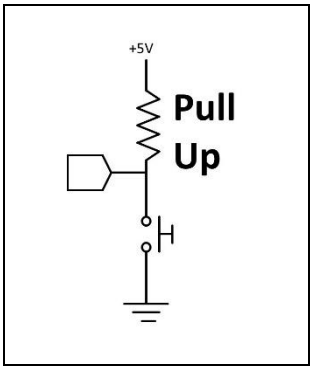
Se dokumentet MTD2A.PDF og afsnittet ”Kadence”, ”Synkronisering” samt ”Eksekveringshastighed”.

Input detektering og aktivering

MTD2A_binary_input læser input fra

1) Digital benforbindelse på Arduino boardet
2) Programmet selv.

Input fra digital benforbindelse på Arduino board er som standard konfigureret med INPUT_PULLUP. Det betyder at der er tilsluttet en modstand på typisk 10 K ohm mellem input benforbindelsen og + (plus) = HIGH. Aktivering sker ved at forbinde benforbindelsen til – (minus) = LOW.
Se mere her: INPUT | INPUT_PULLUP | OUTPUT | Arduino Documentation



Der kan benyttes alle former for bryde og slutte kontakter, momentan og skifte kontakter, relæer og alle former for kredsløb med Open Collector transistor NPN - binært og analogt. Ved analogt kredsløb skifter status efter spændingsniveauerne som beskrevet her: HIGH | LOW | Arduino Documentation

Hvis input kredsløbet også benytter egen pullup modstand, vil det som udgangspunkt virke som det skal.
I modsat fald kan INPUT_PULLUP vælges fra ved at angive INPUT i funktionen:
object_name.initialize (pinNumber, {NORMAL | INVERTED}, {INPUT_PULLUP | INPUT });.

Der er to mulige input til funktionen: 1. Input fra digital benforbindelse på Arduino board pinState => {HIGH | LOW} pin read only. 2. Input fra selve programmet inputState = {HIGH | LOW} write & read.	pinState	inputState	CurrState
	HIGH	HIGH	HIGH
	HIGH	LOW	LOW
	LOW	HIGH	LOW
	LOW	LOW	LOW

Input aflæses fra det digitale benforbindelsesnummer, der er specificeret i object_name.initialize (pinNumber.);.
Kaldes funktionen ikke, bliver benforbindelsen ikke aflæst pinReadToggl = disable og pinNumber = 255.

object_name.initialize (pinNumber); Må kun kaldes **en gang** og kun i Arduino void setup ();

Hvis benforbindelsenummer er initialiseret korrekt med ovenstående funktion, er det muligt løbende at styre om benforbindelsen skal aflæses eller ej med funktionen:
object_name.set_pinReadToggl ({ENABLE | DISABLE});

Under BLOCKING_PHASE er aflæsning af benforbindelsen midlertidigt suspenderet af processen selv. Et kald til object_name.set_pinReadToggl ({ENABLE | DISABLE}); i denne fase afbryder ikke den aktive blokeringsperiode — indstillingen gemmes og træder i kraft, når blokeringsperioden ophører.

Input kan også komme fra programmet selv:
object_name.set_inputState ({HIGH | LOW}, {PULSE | FIXED});

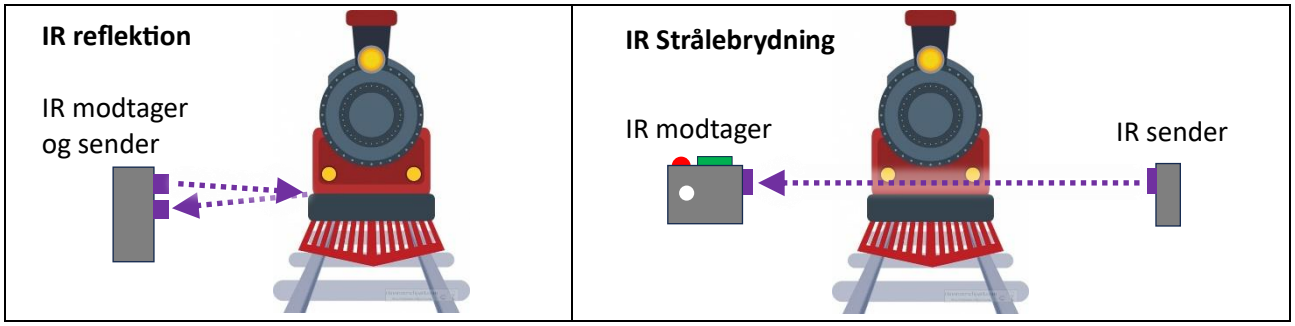
Pulse angiver en enkelt impuls (kort monostabilt) og fixed virker permanent og indtil Pulse angives.

Pin Input mode

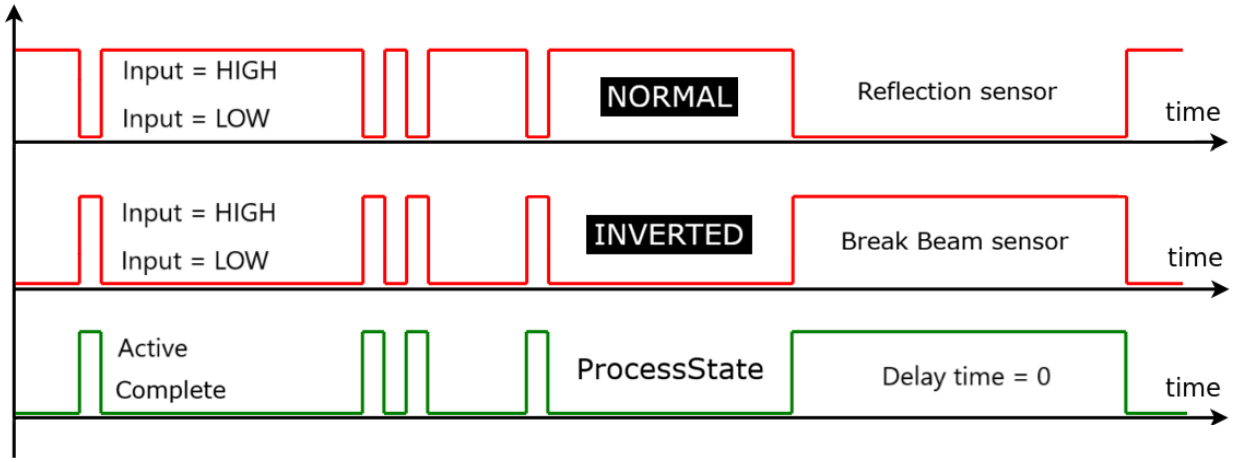
Der er to måder af aflæse input på:

1. NORMAL Trigger sker ved HIGH -> LOW. Output spejler input. Fx reflektionssensor.
2. INVERTED Trigger sker ved LOW -> HIGH. Output følger input. Fx strålebrydningssensor.

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



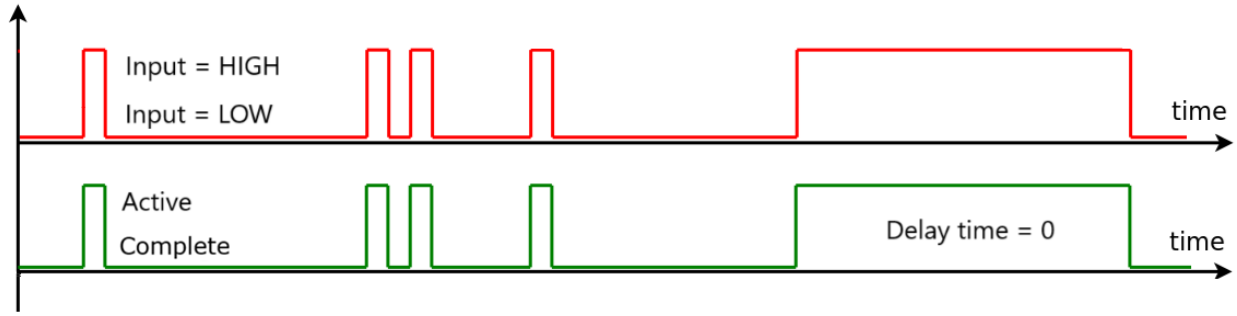
I de følgende eksempler tages der udgangspunkt i FC-51 binær strålebrydning sensor, hvor sender er placeret på den ene side af toget, og modtager er placeret på den anden side. Det betyder at sensor input er LOW i hvile tilstand og HIGH når toget bryder strålen. Der benyttes INVERTED.



Simpel binær funktion

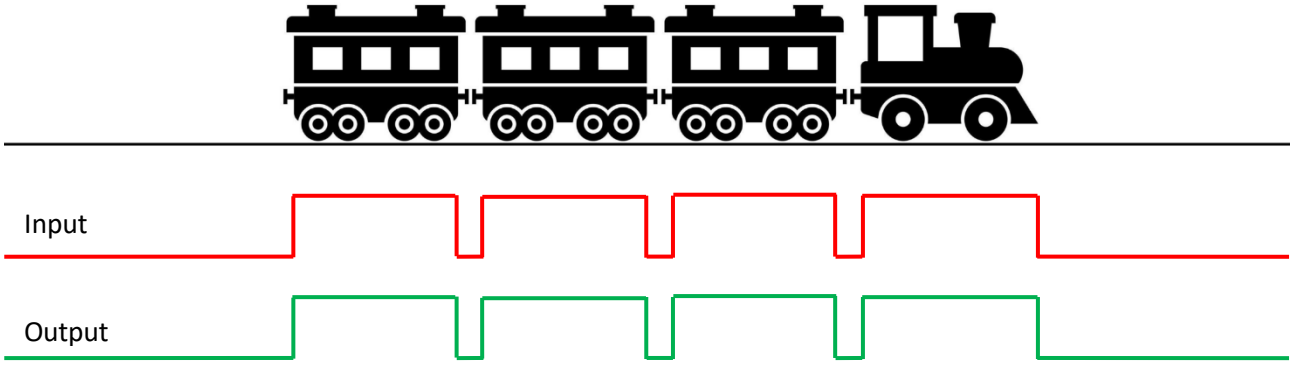
```
MTD2A_binary_input FC_51_sensor (”FC_51_sensor”);
```

Når input går fra LOW til HIGH gør output præcis det samme, og omvendt.



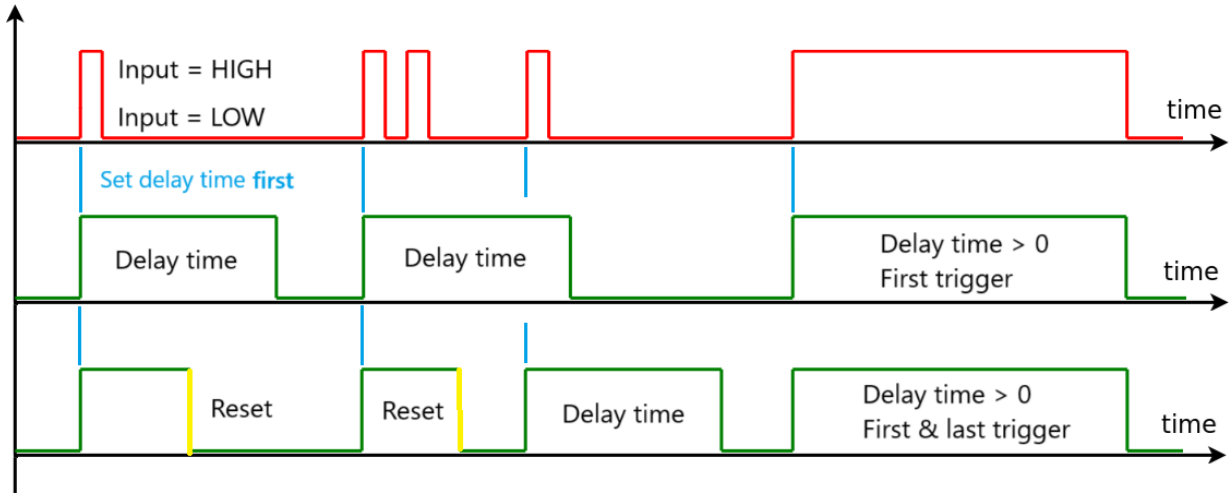
Sensordetektering af et togsæt i bevægelse vil uundgåeligt medføre et antal impulser grundet variationer i opbygning af togvognene og lokomotiv, samt ”huller” ved sammenkoblinger med mere. Disse variationer kan medføre uforudsete genaktivering af funktioner og fejl i den efterfølgende logik proces.

Eksempel på kørende tog



Time delay – first trigger

Når input går fra LOW til HIGH fastholdes output HIGH ind til at den definerede tidsperiode ophører. Er input HIGH ved tidsperiodens ophør, forbliver output HIGH, ind til at input går fra HIGH til LOW.




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

Nulstiller alle styrings og process variable og gør klar til ny start. Alle funktionskonfigurerede variable og standard værdier bibeholdes. Procesfasen skifter til RESET_PHASE

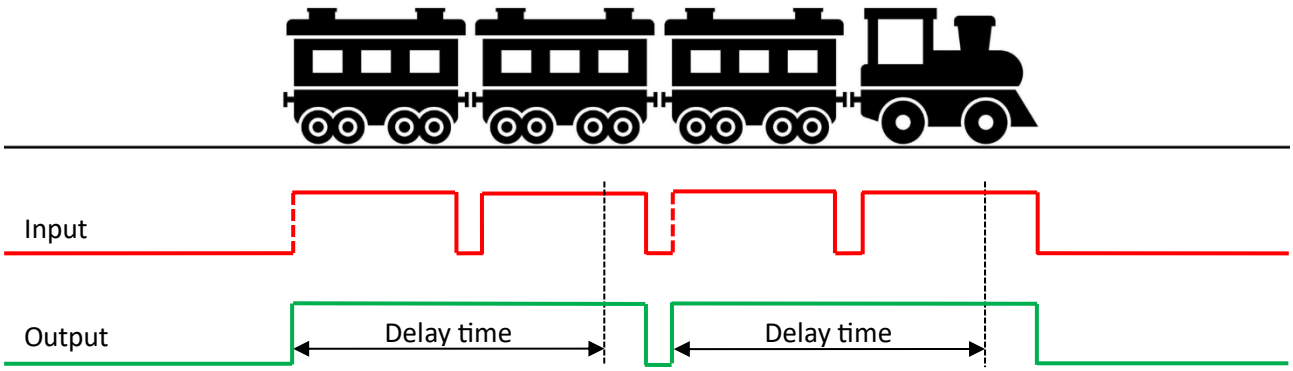
Kaldes `object_name.reset ();` under BLOCKING_PHASE, annulleres blokeringsperioden, og aflæsning af benforbindelsen gendannes til indstillingen fra før blokeringsperioden.

Eksempel på kørende tog

Hvis uhensigtsmæssig genaktivering skal undgås, skal delayTimeMS være lang for at sikre, at det også fungerer ved langsomt kørende tog. I eksemplet herunder er delayTimeMS for kort, hvilket medfører to aktiveringer istedt for en.

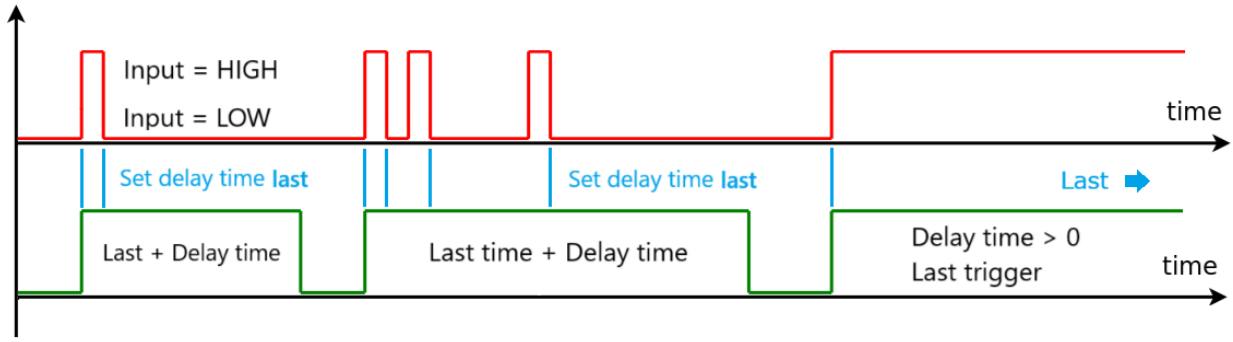
delayTimeMS = 5.000 millisekunder og triggerMode = **FIRST**_TRIGGER.
5 sekund forsinkelse målt fra første detektering af tog.

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



Time delay – last trigger

Når input går fra LOW til HIGH fastholdes output HIGH ind til at den definerede tidsperiode ophører. Hver gang input går fra HIGH til LOW forskydes starten for tidsperioden til det nye tidspunkt. Er input HIGH ved tidsperiodens ophør, forbliver output HIGH, ind til at input går fra HIGH til LOW.

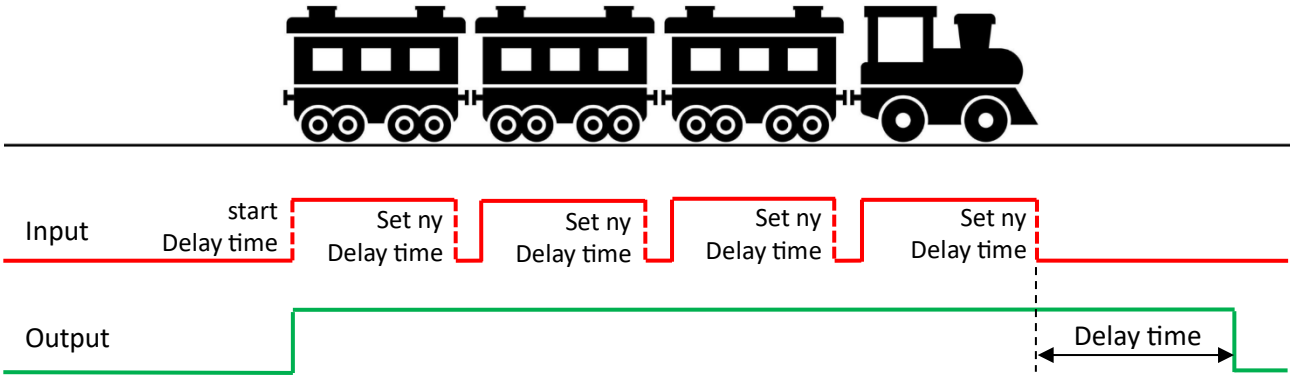


Eksempel på kørende tog

Denne metode er bedst egnet til at detektere hurtigt- og langsomt kørende tog uden uhensigtsmæssige genaktiveringer.

delayTimeMS = 5.000 millisekunder og triggerMode = LAST_TRIGGER.
5 sekunder forsinkelse målt fra sidste detektering af tog (HIGH til LOW).

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

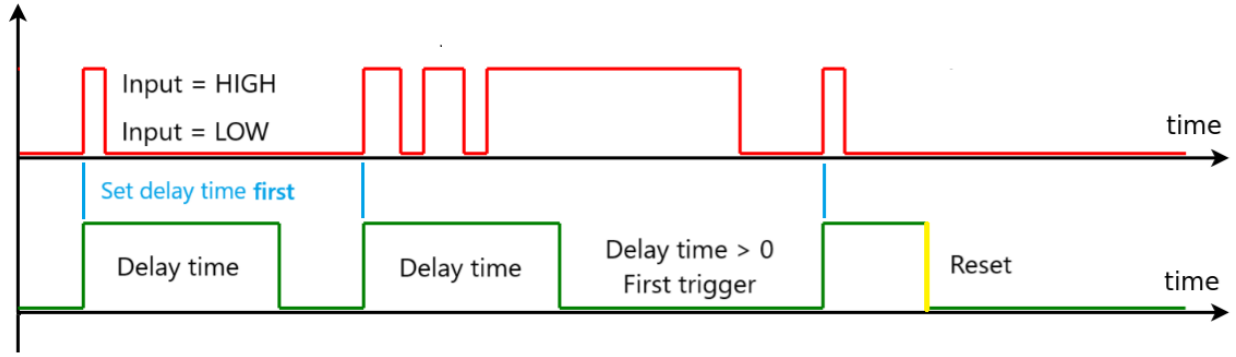


object_name.set_stopDelayTimer ( ); Stopper øjeblikkeligt forsinkelsesperioden. I MONO_STABLE tilstand går processen straks videre til næste fase. I TIME_DELAY tilstand gælder samme regel som ved normalt tidsudløb: er input HIGH, forbliver output HIGH, indtil input går fra HIGH til LOW.

Det kræver, at processen er ACTIVE, og at den tilhørende timer er konfigureret (delayTimeMS > 0). Ellers ignoreres kaldet, og fejl 18 ERR_TIMER_NOT_IN_USE rapporteres. En stopanmodning gælder kun den aktuelle procescyklus — den nulstilles, når processen når COMPLETE_PHASE, og overføres aldrig til næste aktivering.

Monostable – first trigger

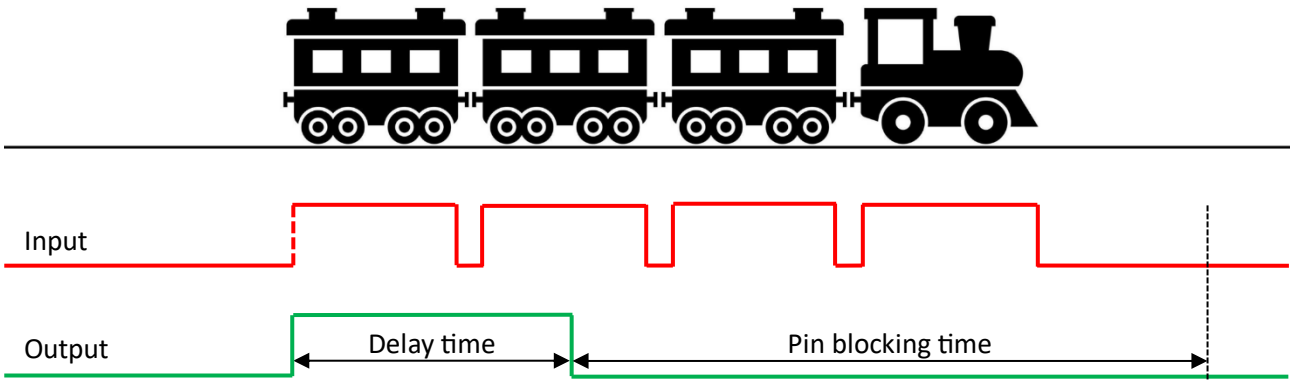
Monostabilt fastholder altid den definerede tidsperiode, uanset om input skifter mellem HIGH og LOW i tidsperioden, og forbliver input enten HIGH eller LOW, ændre det ikke tidsperioden.



Eksempel på kørende tog

delayTimeMS = 5.000 millisekunder, triggerMode = FIRST_TRIGGER, timerMode = MONO_STABLE, pinBlockMS = 12.000 millisekunder (tidsperiode hvor input fra benforbindelsen blokeres fra delayTimeMS afslutning og frem til monostabil afslutning).
5 sekunder forsinkelse målt fra første detektering af tog (LOW til HIGH).

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

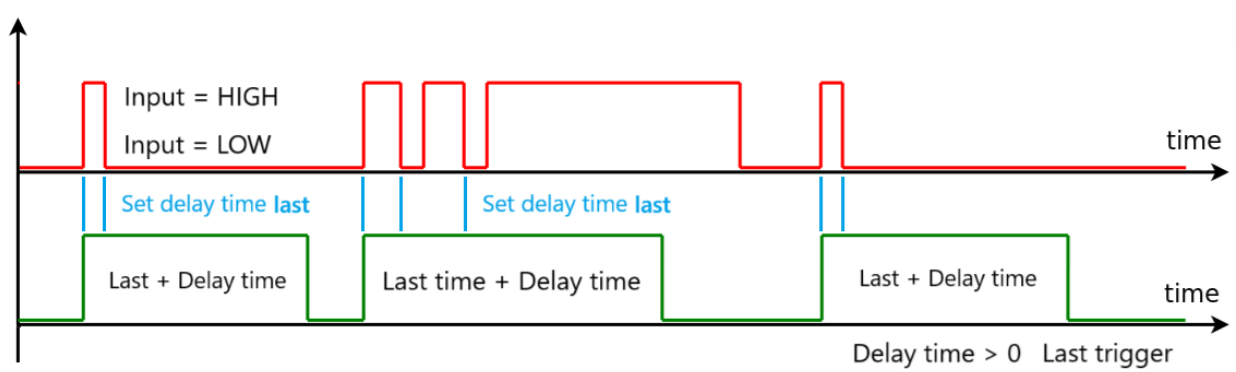


`object_name.set_stopBlockTimer ();` Stopper øjeblikkeligt blokeringspersioden og går videre til næste fase.

Det kræver, at processen er ACTIVE, og at den tilhørende timer er konfigureret (pinBlockTimeMS > 0). Ellers ignoreres kaldet, og fejl 18 ERR_TIMER_NOT_IN_USE rapporteres. En stopanmodning gælder kun den aktuelle procescyklus — den nulstilles, når processen når COMPLETE_PHASE, og overføres aldrig til næste aktivering.

Monostable – last trigger

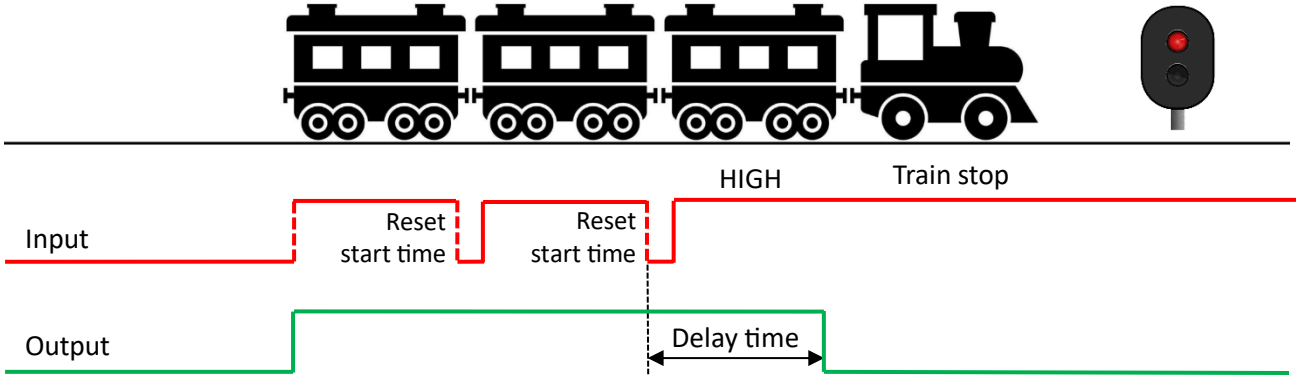
Monostabilt fastholder altid den definerede tidsperiode, men skifter input i mellem HIGH og LOW i tidsperioden, forlænges tidsperioden hver gang. Efter den samlede tidsperiode skifter output til LOW, uanset om input enten er HIGH eller LOW.



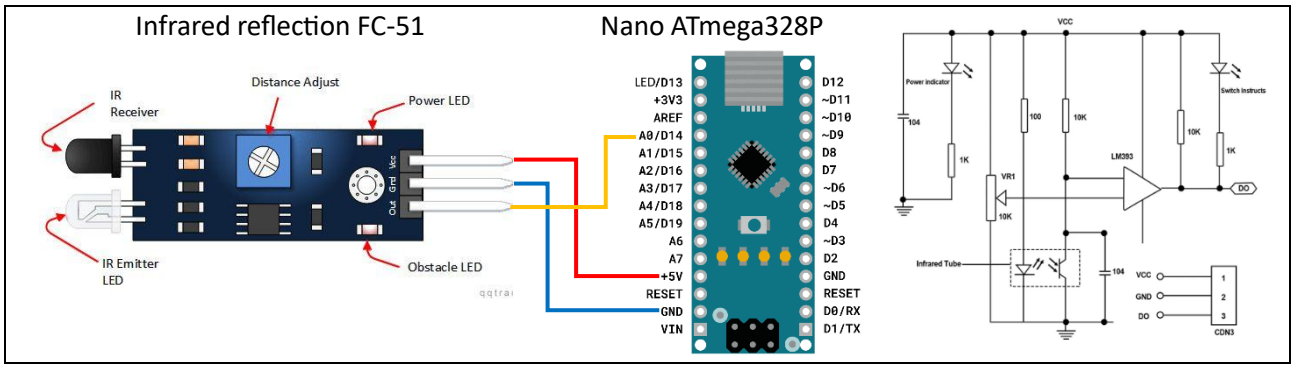
Eksempel på kørende tog der standser over sensor

delayTimeMS = 3.000 millisekunder, triggerMode = LAST_TRIGGER, timerMode = MONO_STABLE.
3 sekunder forsinkelse målt fra første detektering af tog (LOW til HIGH).

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



Eksempel på 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 ();

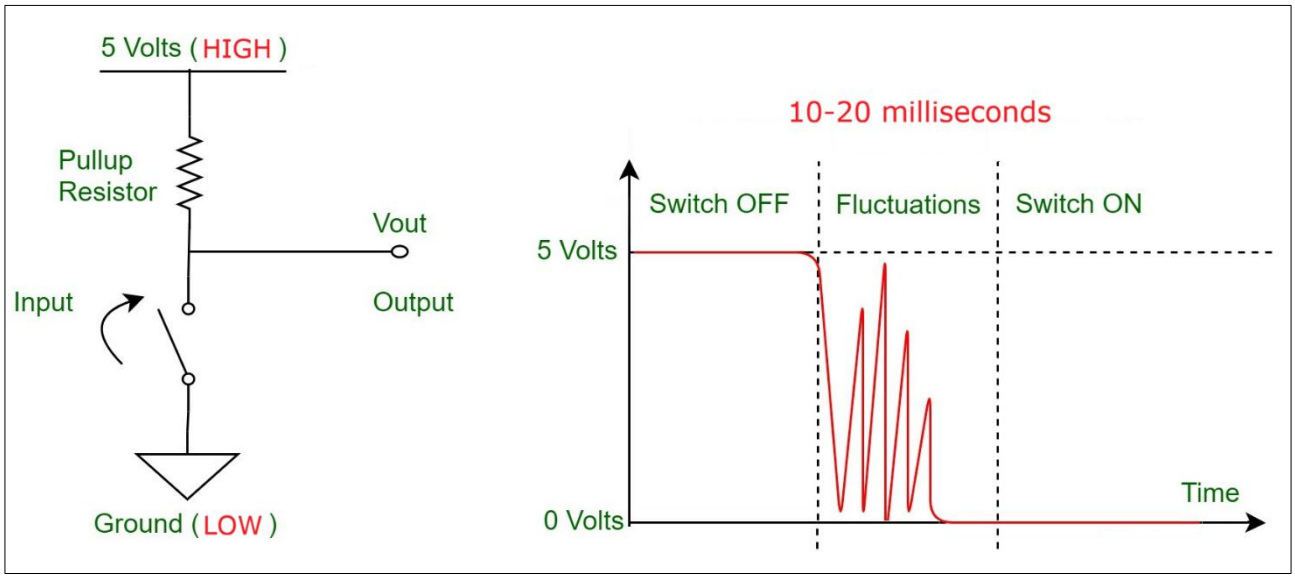
Kontaktprel og anti-prel

Kontaktprel Kontaktprel (eller "chatter") – et almindeligt problem ved mekaniske afbrydere, knapper og relæer. Da der er tale om elektromekaniske komponenter, påvirkes kontaktpunkterne af bevægelsesmængde og elasticitet. Disse kræfter skaber tilsammen hurtige svingninger i kontakten ("prel"), når en afbryder "åbnes" eller "lukkes".

Varigheden af kontaktprel (bounce) varierer afhængigt af kontakttypen, og kontakter preller både ved lukning og åbning. Hver kontakttype har sit eget "prel-mønster" bestående af hurtige svingninger.

Prellet varer typisk 1–20 ms, afhængigt af kontakten. På baggrund af en række tests og analyser vurderes det, at 50 millisekunder er et passende tidsrum, før kontakten er helt stabiliseret.

Problemet med kontaktprel kan løses i enten software eller hardware. Fx med MAX6816 anti-prel IC.



En mere detaljeret analyse om debouncing, skrevet af Jack G. Ganssle, kan læses her: debouncing.pdf

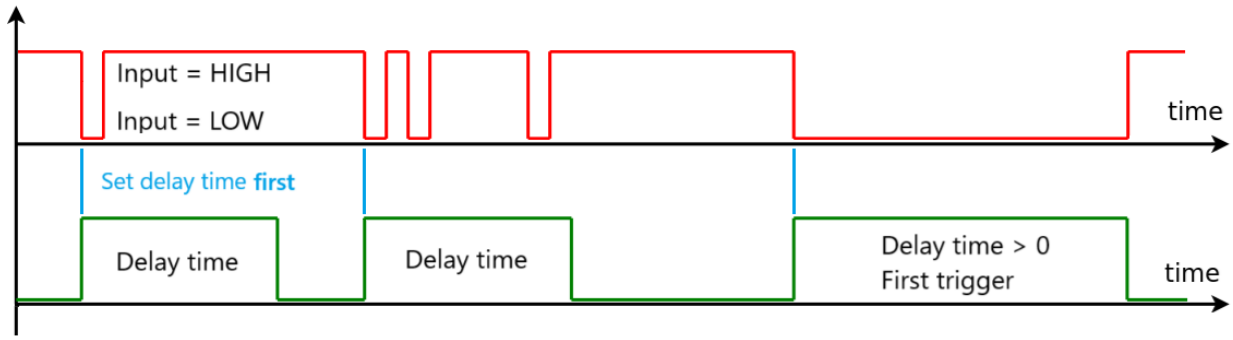
`MTD2A_binary_input ();` håndterer debouncing i kraft af sit design!
Standardindstillingen for MTD2A `DEBOUNCE_MS` es 50 millisekunder.

Se eksempel 1 her: GitHub: MTD2A/examples/push_button

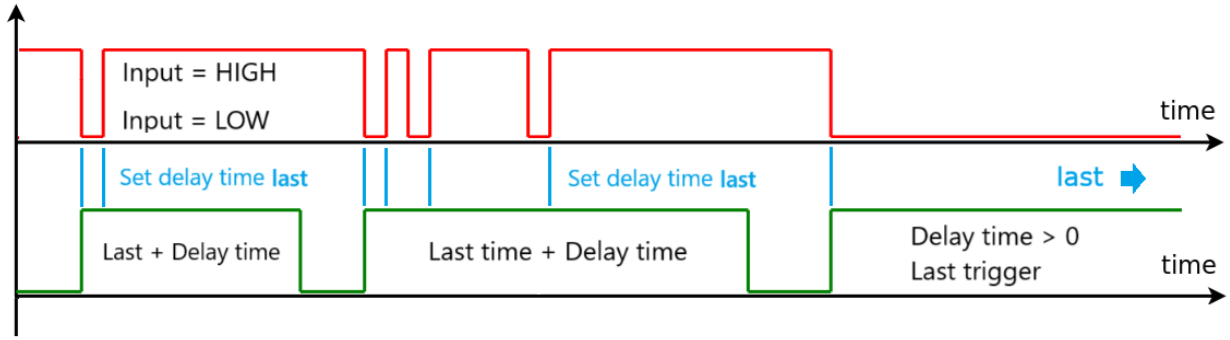
Knap holdes nede

object_name.get_activeTimeMS (); Måler knappens aktiveringstid i millisekunder. En timer starter, når den første aktivering af knappen registreres. Den mindste tidsperiode bestemmes af delayTimeMS. Tidsmålingen fortsætter, så længe knappen er aktiveret.

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
```

Se eksempel 2 and 3 her: [Github: MTD2A/examples/push_button]

Kort og lang knaptryk

Følgende gennemsnitlige tidsværdier udgør et godt udgangspunkt for at registrere korte og lange knaptryk. Tidsværdierne varierer dog fra gang til gang, hvilket gør det nødvendigt at definere et interval omkring disse gennemsnitsværdier.

`SHORT_PRESS_MS` er baseret på en række tests målt til 200 millisekunder. I praksis: > 50 && < 500
`LONG_PRESS_MS` er baseret på en række tests målt til 1000 millisekunder. I praksis: > 500 && < 1500

Eksempel

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

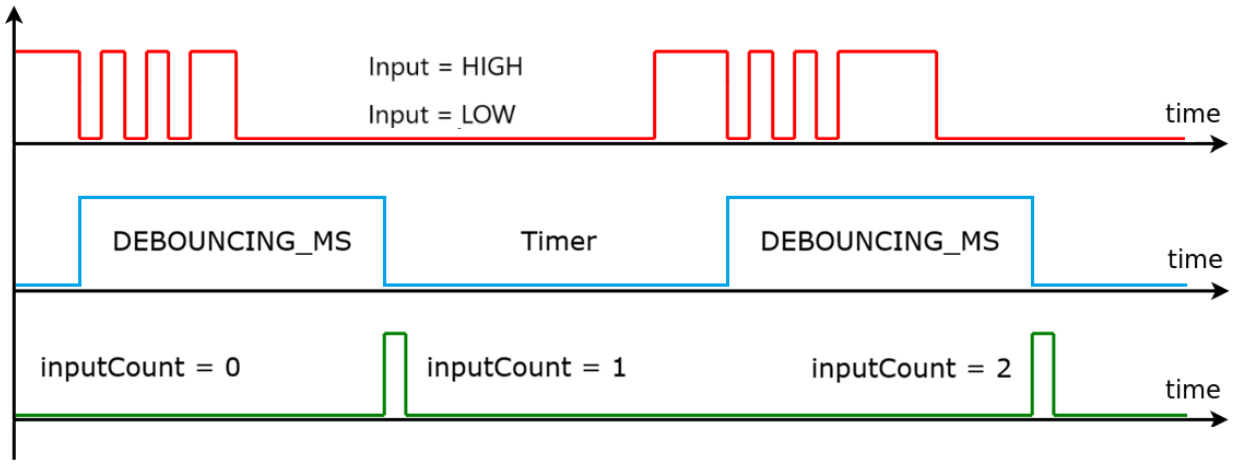
void setup() {
  Serial.begin(9600);
  buttonTimer.initialize (pushButton);
}

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
```

Se eksempel 4 her: [Github: MTD2A/examples/push_button]

Gentagne knaptryk

object_name.get_inputCount (); Tæller antallet af knaptryk og forhindrer fejlagtige aktiveringer grundet debouncing. En timer starter, når den første knapaktivering registreres. Efter en stabiliseringsperiode på DEBOUNCE_MS millisekunder kontrolleres det, om knappen stadig er aktiveret. Hvis dette er tilfældet, lægges 1 til inputCount. Den samme proces gentages ved efterfølgende knaptryk. delayTimeMS skal være større end eller lig med DEBOUNCE_MS, for at optælling kan finde sted. I modsat fald sættes inputCount til 1.



Eksemper

Tæl antallet af knaptryk inden for 2 sekunder. Hvis knappen holdes nede i mere end 2 sekunder, vil den samlede varighed overstige 2 sekunder.

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

Tæl antallet af knaptryk inden for 2 sekunder, og nulstil timeren, hver gang knappen slippes.
Dette gør det muligt at trykke på knappen op til 255 gange, forudsat at hvert tryk fuldføres inden for 2 sekunder. Dermed vil den samlede varighed overstige 2 sekunder, og det endelige resultat afhænger af brugerens adfærd.

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

Registrerer øjeblikkeligt præcis ét knaptryk og sætter inputCount til 1. Processen starter forfra, når knappen slippes. Hvis knappen holdes nede, øges den samlede varighed, indtil knappen slippes.

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

Eksempel

```
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
```

I tilstanden MONO_STABLE registreres et knaptryk inden for de sidste DEBOUNCE_MS af forsinkelsen ikke.

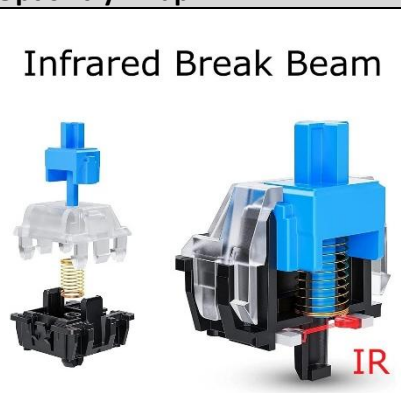
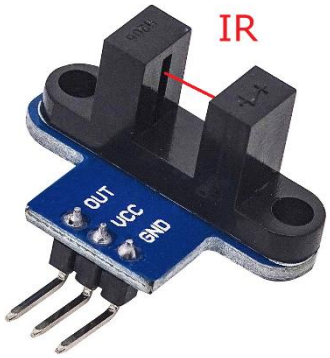
Faldende og stigende flanke

Knappen trykkes ned: `get_inputGoLow ();` Returnerer bool {true | false} Er kun sand i én løkke-cyklus.
Knappen slippes: `get_inputGoHigh ();` Returnerer bool {true | false} Er kun sand i én løkke-cyklus.

Hurtige knaptryk

`object_name.get_inputCount ();` understøtter ikke hurtige knaptryk på grund af stabiliseringstiden `DEBOUNCE_MS`. Inputtet vil blive fortolket forkert, og resultatet vil ligge et sted mellem 0 og det faktiske antal knaptryk. Ved først at tælle, når `DEBOUNCE_MS` millisekunder er gået, sikres stabilitet, filtrering af støj samt frasortering af urealistisk hurtige skift.

Hvis det er nødvendigt at registrere hurtige tastetryk, kan man benytte optiske knapper og kontakter. Optiske kontakter anvender lysstråler i stedet for metalkontakter, hvilket giver en næsten øjeblikkelig responstid, eliminerer prel, og sikrer en usædvanlig lang holdbarhed på over 100 millioner tastetryk. Optiske kontakter anvendes blandt andet i visse typer tastaturer.

Optisk trykknap	**Optiskt micro switch**	**Optisk slidsføler**
		

Set og get funktionsoversigt

Set functions Grøn er standard, ved ingen parameter	Comment
set_pinReadToggl ({ENABLE | DISABLE});	Enable or disable pin reading
set_pinReadMode ({NORMAL | INVERTED});	Configure pin trigger input mode
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 or last timer process immediately.
set_stopBlockTimer ();	Stop blocking timer process immediately.
set_debugPrint ({ENABLE | DISABLE});	Activate print phase number and text.
set_errorPrint ({ENABLE | DISABLE});	Activate 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
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
```