

MTD2A & MTD2A_base

MTD2A: Model Train Detection And Action – arduino library <https://github.com/MTD2A/MTD2A>

Jørgen Bo Madsen / V1.7 / 04-10-2025

MTD2A er en samling af brugervenlige, avancerede og funktionelle C++ klasser - byggeklodser - til tidsstyret håndtering af input og output, og understøtter parallel processering samt asynkron eksekvering.

Biblioteket er tiltænkt Arduino interesserede uden større programmeringserfaring, der har elektronikstyring og -automatisering, samt modeltog som interesse.

Fælles for alle byggeklodser:

- Understøtter en bred vifte af inputsensorer og outputenheder
- Er enkle at bruge til at bygge komplekse løsninger med få kommandoer
- Fungere Ikke-blokerende, procesorienteret og tilstandsrevet
- Tilbyder omfattende kontrol- og fejlfindingsinformation
- Grundigt dokumenterede med eksempler

Indhold

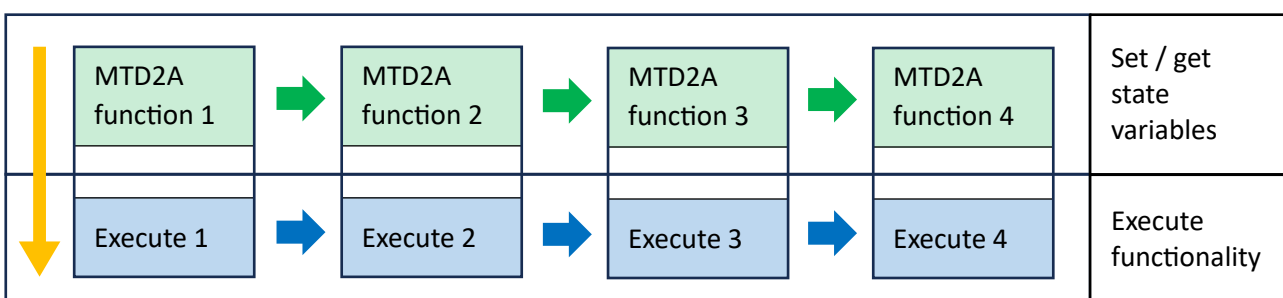
MTD2A & MTD2A_base.....	1
Virkemåde.....	1
To typer af process flow.....	3
Bibliotek.....	4
MTD2A_const.h.....	4
MTD2A_base (h og cpp).....	5
Tidsstyring og kadance.....	5
Fejl – og advarselsbeskeder.....	6
Aliases.....	7
Produktion.....	7
Eksempel på parallel processering.....	8

Virkemåde

MTD2A er en såkaldt tilstandsmaskine. Det betyder MTD2A objekter gennemløbes **hurtigt** i en uendelig løkke, og hvert gennemløb er opdelt i to faser:

1. Ændring af tilstandsvariable (ændre og/eller aflæse og kontrollere)
2. Eksekvere funktionalitet og kontrollere tidsforløb.

Konceptdiagram



Den øverste grønne proces gennemføres sammen med brugerdefineret kode og andre biblioteker. Den nederste blå process gennemføres via `MTD2A_loop_execute ();` og som det sidste i `void loop ();`

På denne måde opnåes der en tilnærmet parallelisering, hvor flere funktioner i praksis kan eksekveres samtidigt. Fx to blinkende LED, hvor den ene er synkron, og den anden er asynkron. [Eksempel på Youtube](#)

De enkelte instatierede (aktiverede) objekter eksekveres i den rækkefølge de instantieres (aktiveres).

```
// Execution order MTD2A_loop_execute ();
MTD2A_binary_output LED_1 ("LED_1", 500, 500); // Execute first [1]
MTD2A_binary_output LED_2 ("LED_2", 500, 500); // Execute next [2]
MTD2A_binary_output LED_3 ("LED_2", 500, 500); // Execute last [3]
```

MTD2A biblioteket kan uden videre blandes med brugerdefineret kode og andre biblioteker, så længe ekeveringen foregår ikke blokerende (non blocking). Men det kræver en lidt anderledes tankegang når der udvikles kode, idet der hele tiden skal tages højde for, at den uendelig og hurtige løkke ikke må forsinkes, men også at brugerkode ikke eksekveres flere gange, end hvad der er tiltænkt. Det er ofte nødvendig at benytte forskellige former for logiske styringsflag. Eksempel:

```
oneTimeFlag == true;
void loop() {
  // user or MTD2A code
  if (oneTimeFlag == true) {
    do_something_once ();
    oneTimeFlag = false;
  }
  // user or MTD2A code
}
```

Ikke blokerende eksekvering

Det er helt afgørende, at der ikke benyttes forsinkende eller blokerende kode sammen med MTD2A biblioteket. Der må ikke benyttes forsinkende funktioner som fx `delay ();` samt biblioteker der forhinde hurtige gennemløb af den uendelig løkke. Dog tillades som standard forsinkelser op til maksimalt 10 millisekunder pr. loop gennemløb. Det er i et fleste tilfælde tilstrækkelig tid til at eksekvere brugerdefineret kode og forskellige former for biblioteker samtidigt. Se uddybende forklaring i senere afsnit.

I stedet for at benytte `delay ();` kan der tælles på antal gennemløb. Optælling er det mest simple, men er ikke præcis. timing er en smule hurtigere end `loopCount` og forskellen vokser over tid.

ATmega328: 1% - 5% og ESP32 under 1%. Brug `MTD2A_timer ();` for præcis timing.

```
long loopCount = 0;
// standard MTD2A loop execution time is 10 Miliseconds

void loop() {
  switch (loopCount) {
    case 200: do_things_1 (); break; // 2 seconds after startup
    case 400: do_things_2 (); break; // 4 seconds after startup
    case 6000: do_things_3 (); break; // 60 seconds after startup
  }
  loopCount++;
}
```

Se uddybende forklaring her: [Finite-state machine - Wikipedia](#)

To typer af process flow

Der er grundlæggende to typer procesflows, og de kan kombineres

1. Forudsigeligt procesflow – tidsstyret
2. Uforudsigeligt procesflow – hændelsesstyret

Forudsigeligt procesforløb

Velegnet til forudsigelige processer, der følger en ikke-kritisk tidslinje. Eksempel:

```
// Timed cascading (round robin) 4 LEDs.
switch (loopCount) {
    case 0: green_LED_1.activate(); break; // Start at once
    case 50: green_LED_2.activate(); break; // 0.5 second
    case 100: red_LED_1.activate(); break; // 1 second
    case 150: red_LED_2.activate(); break; // 1.5 second
}
loopCount++;
if (loopCount >= 200) {
    loopCount = 0;
}
```

Uforudsigeligt procesforløb

Velegnet til hændelsesstyret procesforløb som f.eks. objekt-detektion med en sensor. Eksempel:

```
// Read infrared sensor en blink LED.
switch (stepCount) {
    case 0:
        if (IR_sensor.get_processState() == ACTIVE) {
            // do something while detecting
            stepCount = 1;
        }
        // do something while waiting on ACTIVE
        break;
    case 1:
        // Add 500 millisecond pause delay to blink LED
        timer_GL1.timer (START_TIMER, 500);
        stepCount = 2;
        break;
    case 2:
        if (timer_GL1.get_processState() == COMPLETE) {
            // do something once
            green_LED_1.activate();
            stepCount = 3;
        }
        // do something while waiting on COMPLETE
        break;
    case 3:
        if (green_LED_1.get_processState() == COMPLETE) {
            // do something once
            stepCount = 0; // Start forfra
        }
        // do something while waiting on COMPLETE
        break;
}
```

```
}
```

Syv eksempler til inspiration på, hvordan man koder tids- og hændelsesprocesforløb

<https://github.com/MTD2A/MTD2A/tree/main/examples>

DEMO video: https://youtu.be/UU4k4_8GWfM

Bibliotek

```
#include <MTD2A.h> // Inkluder alle biblioteksfiler
using namespace MTD2A_const; // Brugervenlig navnekonstanter
```

Nuværende byggeklodser

- MTD2A_binary_input
- MTD2A_binary_output
- MTD2A_timer

Yderligere planlagte byggeklodser

- | | |
|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none">• MTD2A_tone• MTD2A_sound | <ul style="list-style-type: none">• MTD2A_display• MTD2A_DCC_input |
|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|

MTD2A_const.h

Indeholde brugervenlig og letforståelige globale konstanter til anvendelse som parametre til de forskellige funktioner og i den kode brugeren skriver. Herunder et udvalg af globale logiske konstanter.

ENABLE	DISABLE
ACTIVE	COMPLETE
FIRST_TRIGGER	LAST_TRIGGER
TIME_DELAY	MONO_STABLE
NORMAL	INVERTED
PULSE	FIXED
BINARY	P_W_M
RESTART_TIMER	STOP_TIMER

Procesfaser tilhørende de enkelte moduler

RESET_PHASE	= 0		
BEGIN_PHASE	= 1,	OUTPUT_PHASE	= 2, END_PHASE = 3
FIRST_TIME_PHASE	= 1,	LAST_TIME_PHASE	= 2, BLOCKING_PHASE = 3
COMPLETE_PHASE	= 4		

Brugen af namespace er en metode til at håndtere navnesammenfald med andre biblioteker. Som udgangspunkt kan alle de globale konstanter gøres tilgængelige ved i starten af brugerprogrammet ved at skrive: `using namespace MTD2A_const;` Hvis der er navnsammenfald, skal hver af de globale konstanter der benyttes gøres tilgængelige i brugerprogrammet. Det kan gøres på to mådersom vist på eksemplet herunder:

`MTD2A_const::ENABLE;` Benyttes hver gang ENABLE benyttes.

`using MTD2A_const::ENABLE;` Gør ENABLE til en global konstant i brugerprogrammet.

`::` = Scope Resolution Operator.

MTD2A_base (h og cpp)

Indeholder styringsfunktioner, fælles interne funktioner og globale fælles funktioner.

MTD2A_loop_execute (); eksekverer alle instantierede objekter (Linked list of function pointers)
Absolut sidste kommando i **void loop ();** og skal altid kaldes en - og kun - en gang.

MTD2A::get_globalObjectCount (); Funktionen returner antallet af instantierede (aktiverede) MTD2A objekter.

MTD2A_globalDebugPrint (); Aktiver fejl og status meddelelser til Arduiono IDE Serial Monitor, og er Initialiseret til **DISABLE** Logisk parameter = { **ENABLE** [true] | **DISABLE** [false] } Udelades parameter sættes default = **ENABLE**.

MTD2A_globalErrorPrint (); Aktiver fejl meddelelser til Arduiono IDE Serial Monitor. Initialiseret til **ENABLE** Logisk parameter = { **ENABLE** [true] | **DISABLE** [false] } Udelades parameter sættes default = **ENABLE**

Tidsstyring og kadance


Synkronisering

For at sikre at alle instantierede (aktiverede) objekter "går i takt", benytter alle objekter et fælles udgangspunkt for tidsfastsættelse. Tiden sættes én gang i hvert **void loop ();** som det første trin i funktionen **MTD2A_loop_execute ();** Dvs. efter at brugerkode og MTD2A funktioner er eksekveret, og før alle interne styrings- og tidskontrol funktioner er eksekveret.

Den aktuelle tidstagnung kan aflæses med funktion: **MTD2A::globalSyncTimeMS**

Kadance

For at sikre at alle instantierede (aktiverede) objekter følger en forudsigelig og ensartet tidsperiode, beregner funktionen **MTD2A_loop_execute ();** gennemløbstiden af alle instantierede MTD2A opbjekter, den kode som brugeren har tilføjet, og de biblioteker brugeren benytter. Funktionen modregner forsinkelser op til 10 millisekunder pr. samlet gennemløb **void loop ();**, ved standard kadance = **DELAY_10MS**

Standard kadance = 10 millisekunder = loop time. Rød linje: Ny fælles tidstagnung til tidskontroller. Gul linje: Modregning af forsinkelser. Styret tidsforsinkelse sikre ensartet kadance.		User code, libraries & MTD2A functions	Elapsed time A
		MTD2A_loop_execute ()	Elapsed time B
		MTD2A delay()	Delay (10 - A - B)

Hvis beregningstiden går ud over de 10 millisekunder kan der opstå problemer med at fastholde en ensartede kadance. Jo længere beregningstiden går ud over de 10 millisekunder, jo længere og mere uensartede kadance, og det kan opstå asynkrone problemer. Især ved tidskritiske og hurtigt eksekverende funktioner. I praksis kan det i mange tilfælde fungere, med mindre variationer, ved en beregningstid på op til 100 millisekunder.

Præcisionen i tidsmålingerne er +/- 10 millisekunder for **DELAY_10MS** og +/- 1 millisekund for **DELAY_1MS**

MTD2A::set_globalDelayTimeMS (); Sætter kadancen i millisekunder hvormed alle instantierede (aktiverede) objekter eksekveres. Parameter = { **DELAY_10MS** | **DELAY_5MS** | **DELAY_1MS** }
Udelades parameter sættes default **DELAY_10MS**

MTD2A::get_globalDelayTimeMS (); Returnere den aktuelle tid for eksekveringskadance i millisekunder.
Default **DELAY_10MS**.

MTD2A::get_MaxElapsedTimeUS (); Funktionen returnerer den maksimale målte tidsforsinkelse i mikrosekunder for hvert gennemløb totalt set, og nulstiller samtidigt målingen. Dvs. tidsforsinkelsen for gennemløb **void loop ();** af instantierede MTD2A objekter (kode), samt den kode som brugeren har tilføjet. Funktionen er primært rettet mod at identificere bruger kode, og de biblioteker brugen benytter, der forsinker gennemløbstiden mere end **MTD2::get_globalDelayTimeMS ();** Millisekunder. Det kan medføre kadance- og synkroniseringsproblemer.

MTD2A::get_timeOverrunCount (); Tæller hvor mange gange gennemløbstiden **void loop ();** overstiger **MTD2::get_globalDelayTimeMS ();** Der sker ofte hvis der skrives længere tekster til *IDE Serial Monitor* og især hvis kadance er defineret til = **DELAY_1MS**, og hvis anvendes mindre kraftige Arduino boards

Eksekveringshastighed

Som udgangspunkt er de mindre kraftige Arduino boards tilstrækkelige fx MEGA, UNO og NANO. Hvis der benyttes mange instatierede (aktiverede) MTD2A objekter, anbefales at benytte de mere kraftige Arduino boards (MCU) som fx ESP32 serien.

Gennemløbstiden kan stige betydeligt ved benyttelse af sensore med egen MCU (fx VL53L4CD laser afstandsmåler), kommunikationsprotokoller fx UART samt skrivning til Arduino IDE Serial Monitor.

Fx tager det 7-8 millisekunder at forspørge VL53L4CD om data er klar, og efterfølgende aflæse data. Kommunikationsprotokoller bør, om muligt, konfigureres til høj hastighed. Det samme gælder for Serial Monitor, der som standard benytter den relativt langsomme 9600 BAUD hastighed.

Hvis der er behov for meget hurtig aflæsning og eksekvering, kan **DELAY_1MS** vælges. Fx hurtige aflæsning af infrarød sensor til nødstop, eller mere præcis tidsstyring. Her anbefales det at benytte de mere kraftige Arduino boards (MCU) som fx ESP32 serien.

Fejl – og advarselsbeskeder

Number	Error text
1	Pin number not set (255)
2	Digital pin number out of range
3	Analog pin number out of range
4	Output pin already in use
5	Pin does not support PWM
6	tone() conflicts with PWM pin
7	Pin does not support interrupt
8	Must be INPUT or INPUT_PULLUP
9	Time must be >= globalDelayTimeMS
11	Pin write is disabled
12	Proces state must be COMPLETE
13	Select STOP_TIMER or RESET_TIMER
14	Unknown TIMER argument
15	globalDelayTimeMS must be 1 - 10 MS

Number	Warning text
128	Digital pin check not possible
129	Analog pin check not possible
130	Pin used more than once
131	PWM pin check not possible
132	Interrupt pin check not possible
140	Timer value is zero
150	Output timer value is zero
151	All three timers are zero
152	Binary pin value > 1. Set to HIGH
153	Undefined PWM curve
154	Use RISING curve instead of FALLING
155	Use FALLING curve instead of RISING

Aliases

`MTD2A_loop_execute ();` Er et mere brugervenligt alias for `MTD2A::loop_execute ();`
`MTD2A_globalDebugPrint ();` Er et mere brugervenligt alias for `MTD2A::globalDebugPrint ();`
`MTD2A_globalErrorPrint ();` Er et mere brugervenligt alias for `MTD2A::globalErrorPrint ();`

`MTD2A_print_conf ();`

```
MTD2A_base:
-----
globalDebugPrint : DISABLE
globalErrorPrint : ENABLE
globaldelayTimeMS: 10
globalSyncTimeMS : 10034
lastElapsedTimeUS: 132
maxLoopTimeUS    : 1452
timeOverrunCount : 0
globalObjectCount: 4
MS/US = Milli/Microseconds
```

Produktion

For at minimere forsinkelser og sikre maksimal præcision, bør den relative langsomme skrivning til *Serial Monitor* blokeres som det sidste, før programmet overgår til produktion. [MTD2A_base.h](#)

```
// Development and debug
#define PortPrint(x)    Serial.print(x)
#define PortPrintln(x) Serial.println(x)
// Optimized production
// #define PortPrint(x)
// #define PortPrintln(x)
```

Eksempel på parallel processing

```
// Two flashing LEDs. One with symmetric interval and another with asymmetric interval.
// Jørgen Bo Madsen / may 2025 / https://github.com/jebmdk

#include <MTD2A.h>
using namespace MTD2A_const;

MTD2A_binary_output red_LED ("Red LED", 400, 400);
// 0.4 sec light, 0.4 sec no light
MTD2A_binary_output green_LED ("Green LED", 300, 700, 0, PWM, 96);
// 0.3 sec light, 0.7 sec no light, PWM dimmed

void setup() {
  Serial.begin(9600);
  while (!Serial) { delay(10); } // ESP32 Serial Monitor ready delay

  red_LED.initialize (9); // Output pin 9
  green_LED.initialize (10); // Output pin 10

  Serial.println("Two LED blink");
}

void loop() {
  if (red_LED.get_processState() == COMPLETE) {
    red_LED.activate();
  }
  if (green_LED.get_processState() == COMPLETE) {
    green_LED.activate();
  }

  MTD2A_loop_execute();
}
```

Flere eksempler og youtube video:

<https://github.com/MTD2A/MTD2A/tree/main/examples>

DEMO video: <https://youtu.be/eyGRazX9Bko>