

# 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 is a collection of user friendly advanced and functional C++ classes - building blocks - for time-controlled handling of input and output. The library support parallel processing and asynchronous execution. 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:

- They support a wide range of input sensors and output devices
- Are simple to use to build complex solutions with few commands
- They operate non-blocking, process-oriented and state-driven
- Offers extensive control and troubleshooting information
- Thoroughly documented with examples

## Content

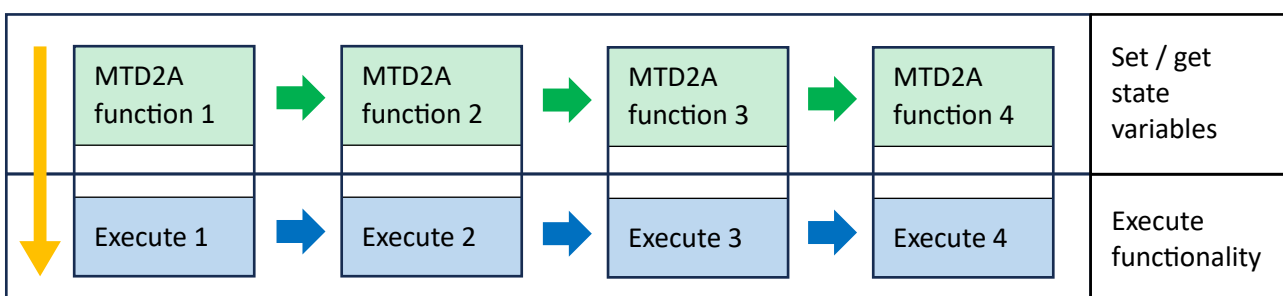
MTD2A & MTD2A_base.....	1
Mode of operation.....	1
Two types of process flow .....	3
Library .....	4
MTD2A_const.h .....	4
MTD2A_base (H and CPP) .....	5
Time management and cadence .....	5
Produktion .....	7
Errors and warning messages .....	6
Aliases .....	7
Example of parallel processing .....	8

## Mode of operation

MTD2A is a so-called state machine. This means MTD2A objects are quickly traversed in an infinite loop, and each traversal is divided into two phases:

1. Changing state variables (changing and/or reading and checking)
2. Execute functionality and control the passage of time.

## Concept Diagram



The top **green process** is carried out together with user-defined code and other libraries.

The lower **blue process** is carried out via `MTD2A_loop_execute ();` and as the last in `void loop ();`

In this way, an approximate parallelization is achieved, where several functions can in practice be executed simultaneously. For example, two flashing LEDs, where one is synchronous and the other is asynchronous.

[Example on Youtube](#)

The individual instantiated (activated) objects are executed in the order in which they are instantiated (activated).

```
// 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]
```

The MTD2A library can be mixed with custom code and other libraries without further ado, as long as the execution is done non-blocking. But it requires a slightly different mindset when developing code, as it must always be taken into account that the infinite and fast loop must **not** be delayed, but also that user code is not executed more times than what is intended. It is often necessary to use different types of logic control flags.

#### Example

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

#### Non-blocking execution

It is absolutely crucial that no delaying or blocking code is used with the MTD2A library. Do not use delaying functions such as `delay ();` as well as libraries that prevent rapid passages of the infinite loop. Use `MTD2A_timer ();` However, by default, delays up to a maximum of 10 milliseconds are allowed per loop pass. In most cases, this is sufficient of time to execute custom code and different types of libraries simultaneously.

Instead of using `delay ();` it is possible to count the number of timed loop passes. Counting is the simplest but is not accurate. Timing is a bit faster than `loopCount` and differences grow over time. ATmega328: 1% - 5% and ESP32 below 1%. Use `MTD2A_timer ();` for precision timing. See example below.

```
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++;
}
```

See more detailed information here: [Finite-state machine - Wikipedia](#)

## Two types of process flow

There are basically two types of process flows and they can be combined

1. Predictable process flow – time controlled
2. Unpredictable process flow – event controlled

### Predictable process flow

Suitable for predictable processes following a non critical time line. Example:

```
// 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;
}
```

### Unpredictable process flow

Suitable for event driven process flows such as object detection with a sensor. Example:

```
// 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 all over again
        }
        // do something while waiting on COMPLETE
        break;
}
```

Seven examples on to inspire how to code time and event process flows

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

DEMO video: [https://youtu.be/UU4k4\\_8GWfM](https://youtu.be/UU4k4_8GWfM)

## Library

```
#include <MTD2A.h> // Include all library files
using namespace MTD2A_const; // Include user friendly constant names
```

### Current building blocks

- MTD2A\_binary\_input
- MTD2A\_binary\_output
- MTD2A\_timer

### Additional planned building blocks

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

## MTD2A\_const.h

Contain user-friendly and easy-to-understand global constants for use as parameters for the various functions and in the code the user writes. Below a selection of global bool constants.

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

### Process phases of the individual modules

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		

The use of namespace is a way to handle name coincidence with other libraries. As a starting point, all the global constants can be made available by typing at the beginning of the user application:

using namespace MTD2A\_const; If there is a name conflict, each of the global constants specified must be made available in the user program. This can be done in two ways, as shown in the example below:

MTD2A\_const::ENABLE; Used every time ENABLE is used.

using MTD2A\_const::ENABLE; Make ENABLE a global constant in the user application.

:: = Scope Resolution Operator.

## MTD2A\_base (H and CPP)

Provides governance capabilities, common internal capabilities, and global common capabilities.

**MTD2A\_loop\_execute ();** executes all instantiated objects (Linked list of function pointers)  
Absolute last command in **void loop ();** and must always be called once - and only - once.

**MTD2A::get\_globalObjectCount ();** Returns the number of instantiated (activated) MTD2A objects.

**MTD2A\_globalDebugPrint ();** Enable error and status notifications for the Arduiono IDE Serial Monitor and is initialized to **DISABLE** bool parameter = { **ENABLE** [true] | **DISABLE** [true] } Omit parameter set default = **ENABLE**

**MTD2A\_globalErrorPrint ();** Enable error notifications for the Arduiono IDE Serial Monitor an is initialized to **DISABLE** bool parameter = { **ENABLE** [true] | **DISABLE** [true] } Omit parameter set default = **DISABLE**

## Time management and cadence


### Synchronization

To ensure that all instantiated (activated) objects "walk in step", all objects use a common starting point for timing. Time is set once in each **void loop ();** as the first step in the function In function **MTD2A\_loop\_execute ();** That is, after user code and MTD2A functions have been executed, and before all internal control and time control functions have been executed.

The current timing can be read with function: **MTD2::globalSyncTimeMS**

### Cadence

To ensure that all instantiated (activated) objects follow a predictable and consistent time period, the function **MTD2A\_loop\_execute ();** calculates the throughput time of all instantiated MTD2A objects, the code added by the user, and the libraries used by the user. The function offsets delays up to 10 milliseconds per total throughput **void loop ();**, at standard cadence = **DELAY\_10MS**

Standard cadence = 10 milliseconds = loop time. <b>Red line:</b> New joint timing for time controls. <b>Yellow linje:</b> Offsetting delays. Controlled time delay ensure uniform cadence.		User code, libraries & MTD2A functions	Elapsed time A
		<b>MTD2A_loop_execute ()</b>	Elapsed time B
		<b>MTD2A delay()</b>	Delay (10 - A - B)

If the calculation time goes beyond the 10 milliseconds, problems may arise in maintaining a consistent cadence. The longer the calculation time goes beyond the 10 milliseconds, the longer and more disparate the cadence, and asynchronous problems can arise. Especially for time-critical and fast-executing functions. In practice, in many cases, it can work, with minor variations, at a calculation time of up to 100 milliseconds.

The precision of the time measurements is +/- 10 milliseconds for **DELAY\_10MS** and +/- 1 millisecond for **DELAY\_1MS**

**MTD2A::set\_globalDelayTimeMS ();** Sets the cadence in milliseconds at which all instantiated (activated) objects are executed. Parameter = { **DELAY\_10MS** | **DELAY\_5MS** | **DELAY\_1MS** } Omitted parameter set default **DELAY\_10MS**

**MTD2A::get\_globalDelayTimeMS ();** Return the current time execution cadence in milliseconds.  
Default **DELAY\_10MS**.

`MTD2A::get_maxElapsedTimeUS ();` The function returns the maximum measured time delay in microseconds for each pass overall, and simultaneously resets the measurement, i.e. the time delay for the throughput `void loop ();` of instantiated MTD2A objects (code), as well as the code that the user has added.

The function is primarily aimed at identifying user code, and the libraries the user uses, which delays the throughput time more than `MTD2A::get_globalDelayTimeMS ();` This can cause cadence and synchronisation problems.

`MTD2A::get_timeOverrunCount ();` Counts how many times the lead time `void loop ();` exceeds `MTD2::get_globalDelayTimeMS ();` This often happens if longer texts are written to *IDE Serial Monitor*, and especially if cadence is defined to = `DELAY_1MS`, and if less powerful Arduino boards are used

### Execution speed

As a starting point, the less powerful Arduino boards are sufficient, e.g. MEGA, UNO and NANO. If many instantiated (activated) MTD2A objects are used, it is recommended to use the more powerful Arduino boards (MCU) such as the ESP32 series.

The throughput time can be increased significantly when using sensors with their own MCU (e.g. VL53L4CD laser rangefinder), communication protocols e.g. UART and IIC as well as writing to the Arduino IDE Serial Monitor.

For example, it takes 7-8 milliseconds to ask VL53L4CD if data is ready, and subsequently read the data. Communication protocols should, if possible, be configured for high speed. The same goes for Serial Monitor, which uses the relatively slow 9600 BAUD speed by default.

If very fast reading and execution is required, can `DELAY_1MS` is selected. For example, fast reading of infrared sensors for emergency stops. The disadvantage is that no set-off is made. Here it is recommended to use the more powerful Arduino boards (MCU) such as the ESP32 series.

### Errors and warning messages

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 ();` Is a more user-friendly alias for `MTD2A::loop_execute ();`  
`MTD2A_globalDebugPrint ();` Is a more user-friendly alias for `MTD2A::globalDebugPrint ();`  
`MTD2A_globalErrorPrint ();` Is a more user-friendly 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

To minimize delays and ensure maximum precision, the relatively slow writing to the Serial Monitor should be blocked as the last thing before the program goes into production. [MTD2A\\_base.h](#)

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

## Example of parallel processing

```
// Two flashing LEDs. One with symmetric interval and another with asymmetric interval.
// Jorgen 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 flashes");
}

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

    MTD2A_loop_execute();
}
```

*More examples and youtube video:*

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

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