MTD2A & MTD2A_base

MTD2A: Model **Train Detection And Action – Arduino library** https://github.com/MTD2A/MTD2A
Jørgen Bo Madsen / V1.3.2 / 16-10-2026

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
- They operate non-blocking, process-oriented and state-driven
- Offers extensive control and troubleshooting information
- Thoroughly documented with examples

Content

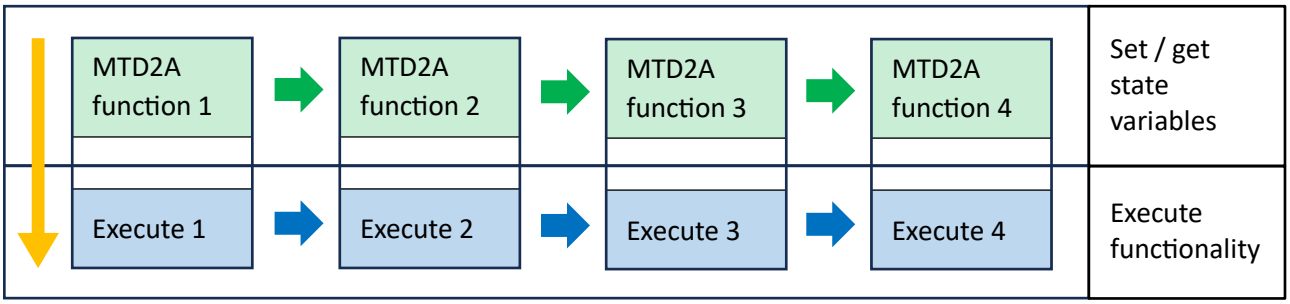
MTD2A & MTD2A_base 1
Mode of operation 2
Two types of process flow 3
Library 5
MTD2A_const.h 5
MTD2A_base (H and CPP) 6
Time management and cadence 6
MTD2A_loop_execute() — Design and Function 7
Errors and warning messages 10
Aliases 11
Set and get functions 11
MTD2A_print_conf (); 11
Production 11
Example of parallel processing 12

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 total loop 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.

```
// 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, at standard cadence = `DELAY_10MS`. In most cases, this is sufficient time to execute custom code and different types of libraries simultaneously.

Instead of using `delay ();` you can count the number of passes `loopCount`. Counting is simple and easy to understand, but counting is not precise. The deviation from the MCU system time grows a little over time.

If precision or long processing times are needed, it is recommended to use `MTD2A_timer ();` and ESP32 MCU.

```
unsigned int loopCount = 0;
// standard MTD2A loop execution time is 10 Milliseconds

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 and blink LED.
switch (stepCount) {
  case 0:
    if (IR_sensor.get_processState() == ACTIVE) {
      // do something once when detected
      stepCount = 1;
    }
    // do something while waiting on sensor is 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 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

It is also possible to use **for** and **while** loops, as long as `MTD2A_loop_execute ();` is part of the loop.

```
void loop() {
  do { // Wait for sensor to activate
    MTD2A_loop_execute ();
  } while (IR_sensor.get_processState() != ACTIVE);
  //
  for (i = 0; i < maxIndex; i++) {
    train_forward.set_timers (stepArr[i].curveMS, stepArr[i].delayMS, stepArr[i].maintainMS);
    train_forward.activate (stepArr[i].beginValue, stepArr[i].endValue, stepArr[i].PWMcurve);
    do {
      MTD2A_loop_execute ();
    } while (train_forward.get_processState() == ACTIVE);
  }
}  // loop ()
```

See example here: https://github.com/MTD2A/MTD2A/tree/main/examples/pendulum_H_bridge

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

- MTD2A_tone
- MTD2A_DCC_input

MTD2A_const.h

Contains user-friendly and easy-to-understand global constants for use as parameters for the various functions and in the code the user writes. Below is 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

Namespace

The use of namespace is a way to handle name conflict 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. `::` = Scope Resolution Operator.

`using MTD2A_const::ENABLE;` Make ENABLE a global constant in the user application.

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), and must always be called once as the last command in void loop (); MTD2A_loop_execute (); (); MTD2A_loop_execute (); can be called multiple times in the same loop if needed, but be aware that several state flags may change status before they can be read and acted upon. E.g. processState and phaseChange

MTD2A:: get_globalObjectCount (); Returns the number of instantiated (activated) MTD2A objects.

MTD2A_globalDebugPrint (); Enable error and status notifications for the Arduino IDE Serial Monitor and is initialized to **DISABLE** bool parameter = { ENABLE | DISABLE } Omit parameter set default = **ENABLE**

MTD2A_globalErrorPrint (); Enable error notifications for the Arduino IDE Serial Monitor and is initialized to **ENABLE** bool parameter = { ENABLE | DISABLE } Omit parameter set default = **ENABLE**

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 (epoch) is set once in each void loop (); as the first step in the 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 (instantiated MTD2A objects).

Cadence

To ensure that all instantiated (activated) objects follow a predictable and consistent time period, the function MTD2A_loop_execute (); calculates the loop time of all instantiated MTD2A objects, the code added by the user, and the libraries used by the user. The total time consumption is corrected after each loop in relation to a parallel precisely calculated total time measurement. This ensures precision.

The function offsets delays up to 10 milliseconds per total loop void loop (); , at standard cadence = DELAY_10MS

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

If the calculation time goes beyond globalDelayTimeMS, problems may arise in maintaining a consistent cadence. Compensation is provided for spontaneous unforeseen time delays in addition to globalDelayTimeMS To synchronize with elapsed time, totalLoopCount is incremented by the number corresponding to the overrun time. Persistent delays create an inconsistent cadence, and asynchronous problems can occur. Especially for time-critical and fast-executing functions

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 by which all instantiated (activated) objects are executed. The function resets the time measurement and calls MTD2A_reset_stats ();
Parameter = { DELAY_10MS | DELAY_5MS | DELAY_2MS | DELAY_1MS }
Omitted parameter set default **DELAY_10MS**

MTD2A::get_globalDelayTimeMS (); Return the current time execution cadence in milliseconds. Default DELAY_10MS.

MTD2A:: get_maxElapsedTimeMS (); Returns the maximum measured time delay in milliseconds for each loop overall. 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 loop time more than MTD2A:: get_globalDelayTimeMS (); This can cause cadence and synchronization problems.

MTD2A::get_timeOverrunCount (); Counts how many times the loop time void loop (); exceeds MTD2A::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 e.g. NANO.

MTD2A::get_totalLoopCount (); Returns the number of loops. Resets after approx. 72 minutes time measurement.

MTD2A::get_globalSyncTimeMS (); Returns the current time that all instantiated (activated) objects use for time control. Resets after approx. 72 minutes of time measurement.

MTD2A_reset_stats (); Resets the statistical measurement variables: lastElapsedTimeMS, maxElapsedTimeMS, timeOverrunCount

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, or CPU demanding PWM curves calculations, 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 I2C as well as writing to the Arduino IDE Serial Monitor.

For example, it takes 7-8 milliseconds to inquire 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 often uses the slow 9600 BAUD speed by default.

If very fast reading and execution is required, DELAY_1MS can be chosen. For example, fast reading of infrared sensors for emergency stops. Here it is recommended to use the more powerful Arduino boards (MCU) such as the ESP32 series.

MTD2A_loop_execute() — Design and Function

MTD2A_loop_execute (); is the central timing engine of the MTD2A library. It is called repeatedly at the end from the Arduino void loop (); function and provides:

- A **constant, predictable cadence** (e.g. exactly 10 milliseconds per loop iteration)
- **A synchronized reference time** (globalSyncTimeUS) shared by all instantiated objects
- **Automatic drift correction** so that accumulated time remains accurate over thousands of loops
- **Diagnostics** for monitoring code execution time and detecting overruns

Design Principle — Epoch-Anchored Timing

Traditional Arduino timing loops measure how long each iteration took and delay the remainder:

```
delay (interval - elapsed)   // relative: errors accumulate.
```

This approach accumulates rounding errors, interrupt jitter, and measurement overhead over thousands of loops. After 60,000 loops at 10ms, the total elapsed time can drift by hundreds of milliseconds.

MTD2A_loop_execute (); uses an **epoch-anchored** reference approach instead. On the first call, it records an absolute start time (the *epoch*). The target time for each subsequent loop is calculated directly from this anchor:

targetTimeUS = epochTimeUS + (totalLoopCount × delayTimeUS)

This means the target for loop N is always relative to the original epoch, **not** to the previous loop. If one loop runs 200 microseconds long, the next loop automatically gets 200 microseconds less delay. Errors from individual loops do not accumulate.

Result: After N loops, the total elapsed wall-clock time converges to N × interval, regardless of per-loop jitter.

Execution Flow

1. Capture current time (globalSyncTimeUS)
2. First-call initialization (epoch anchor)
3. Execute all registered instantiated objects
4. Measure code execution time
5. Overrun correction
6. Gap detection and re-anchoring
7. Epoch-anchored delay
8. Advance loop counter (totalLoopCount)

NOTE: MTD2A_loop_execute (); Uses Arduino micros(); for precision calculation, and will overflow (return to zero) after 71.58 minutes ≈ 4,294,967,295 (0xFFFFFFFF) microseconds. MTD2A handles both overflow and underflow without problems. All time calculations are accurate. However, it will have an impact on readings and printing of various variables for time measurement, which will change to a very small number in case of overflow. Integer overflow - Wikipedia

Overflow example from MTD2A_timer (); (rounded milliseconds - not microseconds)

Last loop before overflow	Next loop
MTD2A_timer: ---------- startTimeMS : 4.294.952 stopTimeMS : 4.294.962	MTD2A_timer: ---------- startTimeMS : 4.294.962 stopTimeMS : 5

Initial reset timers for calculations

The anchor and reference time measurement is reset on the first loop. However, at this point, the time delay is unknown, due to the user's code and the libraries the user is using. This means that the first loop may take longer than expected. To ensure accurate time measurement, MTD2A_loop_execute (); can be called in void setup ();

```
void setup() {
  Serial.begin(250000);
  while (!Serial) { delay(1); } // ESP32 Serial Monitor ready delay
  // One initialization loop to set epoch and calculate correct target time.
  MTD2A_loop_execute();
}
```

Example of overflow handling

```
uint32_t beginTime, elapsedTime, endTime;

void setup() {
  Serial.begin(250000);
  while (!Serial) { delay(10); } // ESP32 Serial Monitor ready delay

  Serial.println("micros() and millis() test of correct overflow handling");
  Serial.println("Elapsed\tBegin\t\tEnd");
  beginTime = 4294967293;   endTime = 4294967294;
  for (byte index = 0; index < 6; index++) {
    elapsedTime = endTime - beginTime;
    Serial.print(elapsedTime); Serial.print(":\t"); Serial.print(beginTime);
    if (beginTime < 10)   Serial.print("\t\t");   else   Serial.print("\t");
    Serial.println(endTime);
    beginTime = endTime;
    endTime++;
  }
} // setup

void loop() {
  delay (10000);
}
```

Output from overflow handling

```
micros() and millis() test of correct overflow handling
Elapsed         Begin                End
1:        4294967293        4294967294
1:        4294967294        4294967295
1:        4294967295                 0
1:                 0                 1
1:                 1                 2
1:                 2                 3
```

Errors and warning messages

Number	Error text
1	Pin number not defined (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	Timer set to globalDelayTimeMS = X
10	Timer set to MAX_TIME_MS
11	Pin write is disabled
12	Process state must be COMPLETE
13	Select STOP_TIMER or RESET_TIMER
14	Unknown TIMER argument
15	globalDelayTimeMS must be: 1 - 10 MS
16	Process state must be ACTIVE
17	Unforeseen system error
18	Must be ACTIVE and timer configured
19	Already initialized
20	Object instantiation error / warning

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
133	PWM disabled: Timer claimed by another library
140	Timer value is zero
141	Time + pause exceeds MAX_TIME_MS
142	setCountDownMS argument is ignored
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
156	BeginValue = endValue => NO_CURVE
157	Time = globalDelayTime => NO_CURVE
158	PAUSE already active
159	loopActivate ENABLE is ignored
160	Value not stepping by globalDelayTimeMS: X

Aliases

MTD2A_loop_execute ();	Is a more user-friendly alias for	MTD2A::loop_execute ();
MTD2A_globalDebugPrint ();	Is a more user-friendly alias for	MTD2A::set_globalDebugPrint ();
MTD2A_globalErrorPrint ();	Is a more user-friendly alias for	MTD2A::set_globalErrorPrint ();
MTD2A_print_conf ();	Is a more user-friendly alias for	MTD2A::print_conf ();
MTD2A_reset_stats ();	Is a more user-friendly alias for	MTD2A::reset_stats ();

Set and get functions

Set functions Green is the default, by no parameter	**Comment**
set_globalDebugPrint ({ ENABLE | **DISABLE** });	Print phase state number and state text
set_globalErrorPrint ({ **ENABLE** | DISABLE });	Print error and warning text
set_globalDelayTimeMS ({ DELAY_1MS | DELAY_2MS | DELAY_5MS | DELAY_10MS });	Main loop delay in milliseconds and calls reset_stats ();

Get functions	**Comment**
get_globalDelayTimeMS (); Return uint8_t milliseconds	Main loop delay time
get_globalSyncTimeMS (); Return uint32_t milliseconds	Current common reference time Resets after approx. 72 minutes time delay
get_globalObjectCount (); Return uint8_t count	Number of instantiated MTD2A objects
get_maxElapsedTimeMS (); Return uint32_t milliseconds	max MTD2A code, user code and other library loop execution time
get_timeOverrunCount (); Return uint32_t count	Number of times loop time that exceeds globalDelayTimeMS
get_totalLoopCount (); Return uint64_t count	Number of loop count since startup. Resets after approx. 72 minutes time delay

MTD2A_print_conf ();

```
MTD2A_base:
--------------------
  globalDebugPrint : DISABLE
  globalErrorPrint : ENABLE
  globalObjectCount: 4
  globalDelayTimeMS: 10
  globalSyncTimeMS : 10978
  lastElapsedTimeMS: 0.00
  maxElapsedTimeMS : 0.00
  timeOverrunCount : 0
  totalLoopCount   : 839
```

Production

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_print.h**

```
// Development and 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 & another with asymmetric interval.
// https://github.com/MTD2A/MTD2A/blob/main/examples/blink_LED/blink_LED.ino

#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, P_W_M, 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