MTD2A & MTD2A_base

MTD2A: Model **Train Detection And Action – arduino library** https://github.com/MTD2A/MTD2A
Jørgen Bo Madsen / V1.4 / 09-07-2025

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

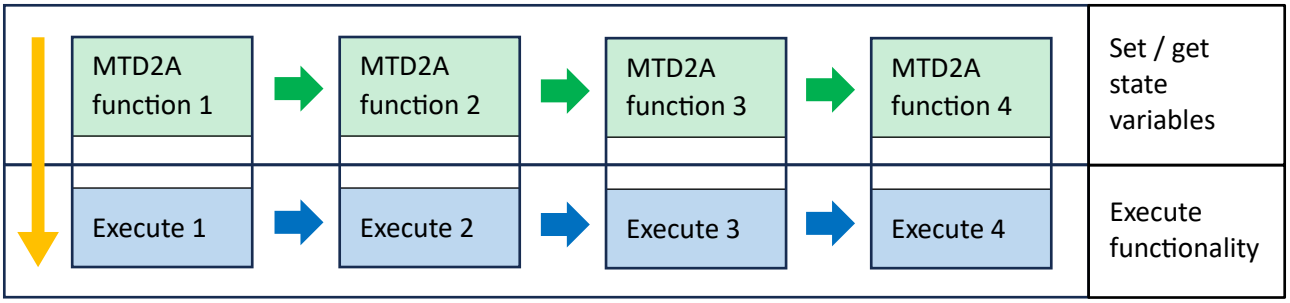
Content

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 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 bibliographies that prevent rapid passages of the infinite loop. However, by default, delays up to a maximum of 10 milliseconds are allowed per pass. In most cases, this is sufficient of time to execute custom code and different types of libraries simultaneously. See further explanation in later sections.

See more here: Finite-state machine - Wikipedia and here: Real-time operating system - Wikipedia

Library

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

Current building blocks

- MTD2A_binary_input.h
- MTD2A_binary_output.h

Additional planned building blocks

• MTD2A_timer • MTD2A_astable • MTD2A_flip_flop • MTD2A_tone • MTD2A_sound • MTD2A_servo	• MTD2A_stepper • MTD2A_display • MTD2A_ultrasonic • MTD2A_laser • MTD2A_IR_ranging • MTD2A_DCC_input

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.

ENABLE	DISABLE
ACTIVE	COMPLETE
FIRST_TRIGGER	LAST_TRIGGER
TIME_DELAY	MONO_STABLE
NORMAL	INVERTED
PULSE	FIXED
BINARY	P_W_M
RESTART_TIMER	STOP_TIMER
DELAY_1MS	DELAY_10MS

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_globalDebugPrint ();` Enable error and status notifications for the Arduiono IDE Serial Monitor.
Parameter = ({ENABLE | DISABLE}) Omit parameter set default = ENABLE

`MTD2A_globalErrorPrint ();` Enable error notifications for the Arduiono IDE Serial Monitor.
Parameter = ({ENABLE | DISABLE}) Omit parameter set default = DISABLE

`MTD2A_globalDelayTimeMS ();` Sets the cadence at which all instantiated (activated) objects are executed.
Parameter = ({ DELAY_10MS | DELAY_1MS}) Omitted parameter set default DELAY_10MS

`MTD2A_globalMaxLoopMS ();` The function returns the maximum measured time delay 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_globalDelayTimeMS ();` This can cause cadence and synchronisation problems.

`MTD2A_globalObjectCount ();` The function returns the number of instantiated (activated) MTD2A objects.

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 ();`, after the user code has been executed and before the first object is executed in function `MTD2A_loop_execute ();`
The current timing can be read with function: `MTD2A_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 up to 10 milliseconds per total throughput `void loop ();`, at standard cadence = `DELAY_10MS`

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.

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.

Phase change delay

Each phase change results in an extra throughput of 1 - 10 milliseconds. If the timing is to be completely accurate, it must be offset against the times specified when the object is instantiated (activated).

Example
A precise total time consumption of 1,000 milliseconds is desired.
`MTD2A_binary_output green_LED ('Green LED', 490, 490);` 490 + 10 + 490 + 10 = 1,000 milliseconds.

If very fast reading and execution is required, kan `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
	Warning text
128	Digital Pin check not possible
129	Anaog Pin check not possible
130	Pin used more than once
131	PWM Pin check not possible
132	Interupt 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

```
MTD2A_print_conf ();
```

```
MTD2A_base:
--------------------
 globalDebugPrint  : DISABLE
 globalErrorPrint  : ENABLE
 globalSyncTimeMS  : 10034
 globalDelayTimeMS: DELAY_10MS
 globalLastTimeMS  : 10044
 globalLoopTimeMS  : 0
 globalMaxLoopMS   : 1
 globalObjectCount: 4
```

Example of parallel processing

```
// Two flashing LEDs. One with symmetric interval and another with asymetric 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();
}
```