MTD2A_binary_output

MTD2A: Model **T**rain **D**etection **A**nd **A**ction – **Arduino** library https://github.com/MTD2A/MTD2A
Jørgen Bo Madsen / V1.3.3 / 17-10-2026

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

Table of contents

MTD2A_binary_output 1
Feature Description 1
Process phases 3
Initialization 4
Activation 4
Stop and reset (restart) timer 5
outputValue 6
PWM curves 7
Scaling 14
outputValue and PWM 15
Examples of configuration 17
Set get functions 18
print_conf(); 19

Feature Description

MTD2A_binary_output process consists of 4 functions:

1. `MTD2A_binary_output object_name ("object_name" , outputTimeMS, beginTimeMS, endTimeMS, outputMode, beginValue, endValue, loopActivate, ActivateNumber);`

2. `object_name.initialize ( pinNumber, pinWriteMode, startPinValue );`
   Executed once in `void setup ();` and after `Serial.begin (9600);`

3. `object_name.activate();` Activates once. Can be reactivated once the process is `(COMPLETE)`

4. `MTD2A_loop_execute ();` Called as the last instruction in `void loop ();`

All functions use default values and can therefore be called with none and up to nine parameters. However, parameters must be specified in ascending order. See example below:

```
MTD2A_binary_output object_name;
MTD2A_binary_output object_name
  ("object name");
  ("o n" , outputTimeMS);
  ("o n" , outputTimeMS, beginTimeMS);
  ("o n" , outputTimeMS, beginTimeMS, endTimeMS);
  ("o n" , outputTimeMS, beginTimeMS, endTimeMS, outputMode);
  ("o n" , outputTimeMS, beginTimeMS, endTimeMS, outputMode, beginValue);
  ("o n" , outputTimeMS, beginTimeMS, endTimeMS, outputMode, beginValue, endValue);
  ("o n " , outputTimeMS, beginTimeMS, endTimeMS, outputMode, beginValue, endValue, loopActivate);
  ("o n" , outputTimeMS, beginTimeMS, endTimeMS, outputMode, beginValue, endValue, loopActivate, ActivateNumber);
```

Defaults:

```
  ("o n" , 0          , 0          , 0          , BINARY     , HIGH     , LOW     , DISABLE     , 0);
```

Example

```
// Two blinking LEDs. One with symmetric interval and another with asymmetric interval.

#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 / 0.7 sec PWM dimmed

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

  byte RED_LED_PIN   = 9;   // Arduino board pin number
  byte GREEN_LED_PIN = 10;  // Arduino board pin number
  red_LED.initialize (RED_LED_PIN);
  green_LED.initialize (GREEN_LED_PIN);
  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();
}
```

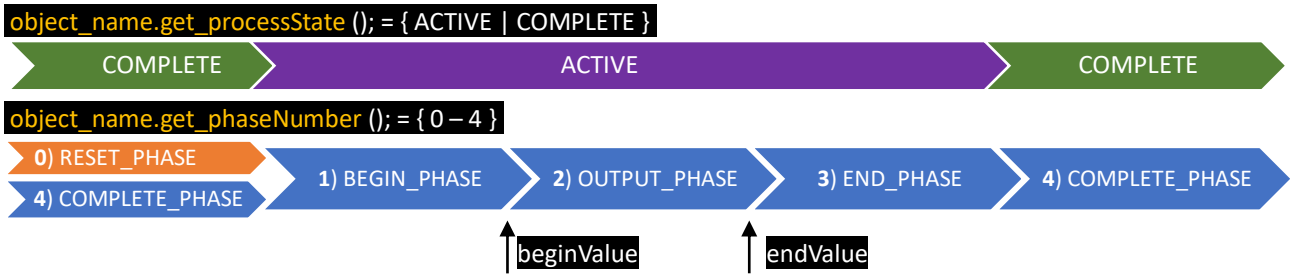
More examples and Youtube video:

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

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

Process phases

Depending on the current configuration, the process is carried out in between 1 and 5 stages.



0. The initial phase when the program starts and when the function reset (); is called or 4) COMPLETE_PHASE.
1. Start delay. If set to 0 or not defined, the stage will be skipped.
2. Output time period. Starts with beginValue and ends with endValue.
   If set to 0 or not defined, the stage will be skipped, and a warning is written.
3. End of delay. If set to 0 or not defined, the stage will be skipped.
4. The process is completed COMPLETE and ready for reactivation object_name.activate ();

Global number constants: RESET_PHASE, BEGIN_PHASE, OUTPUT_PHASE, END_PHASE & COMPLETE_PHASE

The immediate phase shift can be identified by function: object_name.get_phaseChange (); = { true | false }

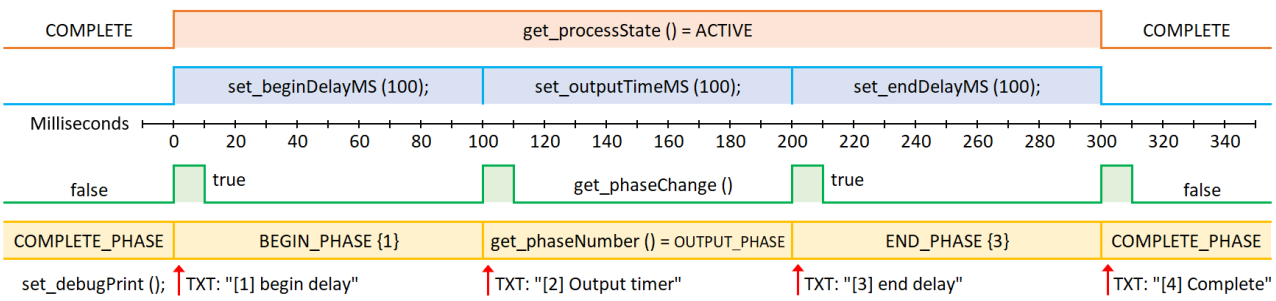
Process status

When transitioning to BEGIN_PHASE, OUTPUT_PHASE, or END_PHASE, ProcessState switches to ACTIVE.
When transitioning to COMPLETE_PHASE or RESET_PHASE, the processState switches to COMPLETE.

The **immediate** phase shift can be identified by function: object_name.get_phaseChange (); = { true | false }
Status information can be printed with the function: object_name.set_debugPrint (); = { ENABLE | DISABLE }

Example of different configuration methods

```
MTD2A_binary_output relay ("Relay", 100, 100, 100, BINARY, HIGH, LOW);
// is equivalent to
MTD2A_binary_output relay ("Relay", 100, 100, 100); // default: BINARY, HIGH, LOW
// is equivalent to
MTD2A_binary_output relay ("Relay"); // default: 0, 0, 0, BINARY, HIGH, LOW
relay.set_outputTimeMS (100);
relay.set_beginDelayMS (100);
relay.set_endDelayMS   (100);
// Show phase change information
relay.set_debugPrint   (); // default enable
```



Timing

See the document MTD2A.PDF, section "Cadence" and "Synchronization" as well as "Execution speed".

Initialization

object_name.initialize (pinNumber, pinWriteMode, startPinValue); Is executed only once in void setup (); and after Serial.begin (9600); This is important for printing possible errors and warnings, including from previous instantiating of objects.
The output is written to the digital PIN connection number specified in object_name.initialize (pinNumber);
If an unsupported PIN connection is specified, the pin connection will not be initialized, and it will **not** be possible to write to the pin connection subsequently: pinWriteToggl = DISABLE and pinNumber = NO_PIN.
object_name.initialize (); is equivalent to object_name.initialize (**NO_PIN**); NO_PIN = 255.

The output on the pin connection can be inverted (reversed) by setting the pinWriteMode to INVERTED
This means that HIGH and LOW are reversed and the PWM value is calculated to 255 minus PWM value.
object_name.initialize (pinNumber, {NORMAL | **INVERTED**});

The PIN connection is initialized with the value specified in the startPinValue.
startPinValue is written **instantly** to pinNumber If the parameter is omitted, LOW is specified.

object_name.initialize ({ pinNumber | **NO_PIN** }, {**NORMAL** | INVERTED}, {startPinValue **|** **LOW**});

It is possible to continuously control whether or not to write to the PIN connection with the function:
object_name.set_pinWriteToggl ({**ENABLE** | DISABLE});

It is also possible to write directly to the PIN connection with the functions:
object_name.set_pinWriteValue (PinValue); {HIGH | LOW} / PWM {0-255}
object_name.set_pinWriteValue (PinValue, {BINARY | P_W_M}); Optional parameter. No default value.
This happens **instantly**, regardless of whether the process is active or not, and independently of MTD2A_loop_execute ();

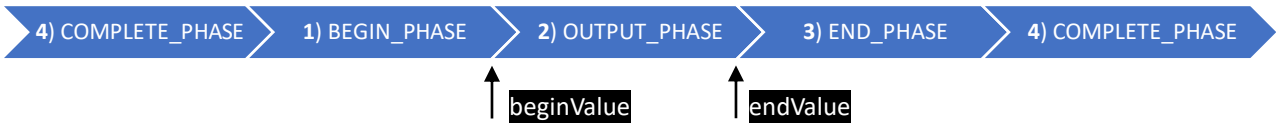
As a starting point, the PIN connection is **undefined**. See Digital Pins | Arduino Documentation

Activation

The process is activated with the function: object_name.activate (); This switches the process State to ACTIVE
As soon as the processState switches to COMPLETE, the process can be activated again.

The process can be reset at any time with the function: object_name.reset(); The function resets all control and process variables **instantly**, and before MTD2A_loop_execute ();), and prepares for a fresh start. All functionally configured variables and default values are retained. The process phase switches to RESET_PHASE
If the PIN connection is defined correctly, the startPinValue is written to the PIN connection **instantly**.

It is possible to write beginning values beginValue and end values endValue to the PIN connection.



Activate functions use "function overloading". This means that the function can be called with none and up to 4 parameters. However, parameters must be specified in ascending order, starting from the first.

Default values are **not** used. All existing values remain the same unless values are specified as parameters in the function call. See example below:

object_name.activate ();
object_name.activate (beginValue);
object_name.activate (beginValue, endValue);
object_name.activate (beginValue, endValue, PWMcurveType);
object_name.activate (beginValue, endValue, PWMcurveType, outputTimeMS);

PWMcurveType specifies which curve to follow in the time period outputTimeMS The curve starts with the value beginValue and ends with the value endValue

outputTimeMS indicates the time span between beginValue and endValue in milliseconds.

Activation can be configured to be repeated indefinitely with the function:
object_name.set_loopActivate ({ ENABLE | DISABLE }); Repeats the process that is subsequently started with activate (); This feature is particularly suitable for flashing LEDs and where a constant pulsing output is required. It is possible to add how many times activate (); should be repeated by specifying the number in ActivateNumber. Zero means ininite. set_loopActivate ({ ENABLE | DISABLE }, ActivateNumber);

When activate (); looping is enabled, the process runs through all phases get_phaseNumber (); in every cycle, including phase COMPLETE_PHASE. The Complete phase lasts one cycle, after which the next cycle starts immediately — the timing of the cycles is not affected.

get_processState (); stays ACTIVE during all loop cycles and only returns COMPLETE when the process stops loopActivate (DISABLE); or activateCount has reached ActivateNumber. DISABLE is accepted at any time.

If ActivateNumber is greater than 0, the number of loop cycles is incremented in activateCount, and the current count can be retrieved using the function get_activateCounter ();

Stop and reset (restart) timer

It is possible to set the new start time for the current timer process, and stop the current timer process prematurely.
RESET_TIMER Sets a new start time which is retrieved from the globally synchronized time globalSyncTimeMS
STOP_TIMER interrupts the time control process

```
object_name.set_outputTimer ( {STOP_TIMER | RESET_TIMER} );
object_name.set_beginTimer ( {STOP_TIMER | RESET_TIMER} );
object_name.set_endTimer ( {STOP_TIMER | RESET_TIMER} );
```

The new start time is retrieved from the globally synchronized time and can be read with the function:
MTD2A::get_globalSyncTimeMS ();

The time period is set by default when the object is instantiated (activated). It is possible to define new time periods for all three time periods:

```
object_name.set_outputTimeMS ( {0 - MAX_TIME_MS} );
object_name.set_beginDelayMS ( {0 - MAX_TIME_MS} );
object_name.set_endDelayMS   ( {0 - MAX_TIME_MS} );
```

object_name.set_timers combines all three timers into one function. "function overloading" is used. This means that the function can be called with one and up to 3 parameters. However, parameters must be specified in ascending order, starting from the first.

Default values are **not** used . All existing values remain the same unless they are specified as parameters in the function call. See example below:

```
object_name.set_timers (outputTimeMS);
object_name.set_timers (outputTimeMS, beginDelayMS);
object_name.set_timers (outputTimeMS, beginDelayMS, endDelayMS);
```

outputValue

Write to outputValue and the PIN connection happens in the same millisecond the MTD2A_loop_execute (); is started. **But** the value can only be read with object_name.get_outputValue (); After MTD2A_loop_execute (); is completed, and thus only at the next pass.

There are therefore two logical status flags that can be used to identify when there is data to be read:

1. object_name.get_outputState (); Is true every time there is new data that can be read.
1. object_name.get_outputProcess (); Is true when data can be read. That is, throughout the output period.

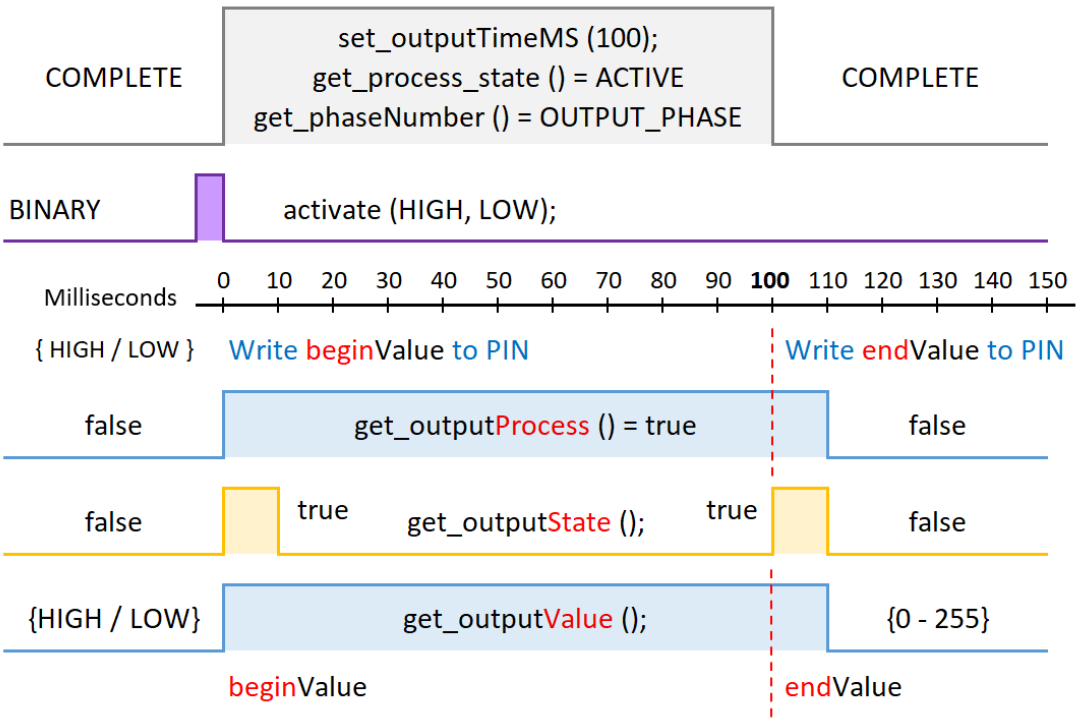
The output value for the PIN connection is specified regardless of whether the PIN connection is defined or not. See explanation of initialization without specifying PIN connection under the section initialization.

The value that can be read from object_name.get_outputValue (); is always the last specified value.

NOTE: If object_name.pinWriteMode is set to INVERTED, this is the inverted value — in !value BINARY mode and 255 − value in P_W_M mode — not the value passed to object_name.activate (); or object_name.get_outputValue (); With the default object_name.pinWriteMode (NORMAL); the two are identical.

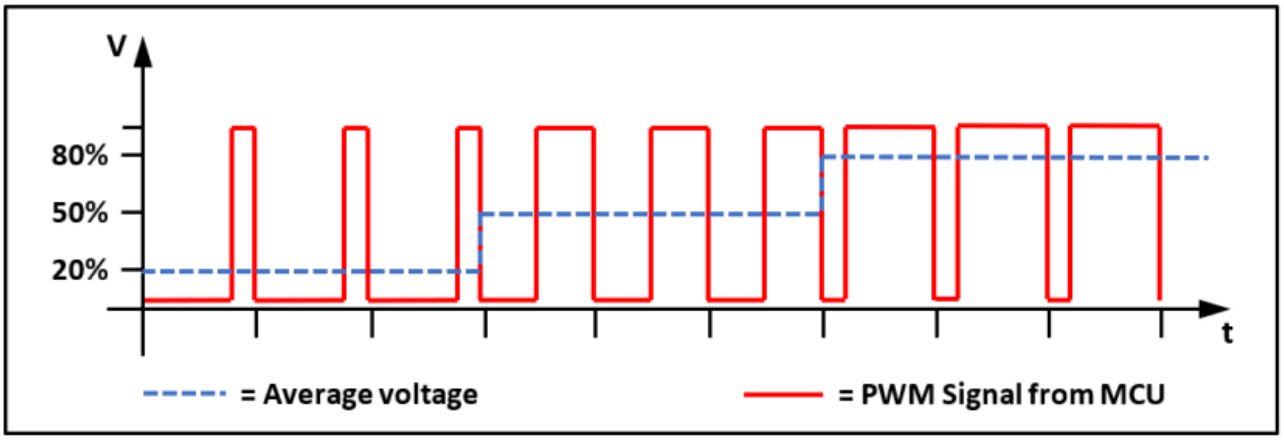
Example:

```
if (relay.get_outputState() == true) {
  outputValue = relay.get_outputValue();
  Serial.print(”PIN output value: ”); Serial.println (outputValue);
}
```



PWM curves

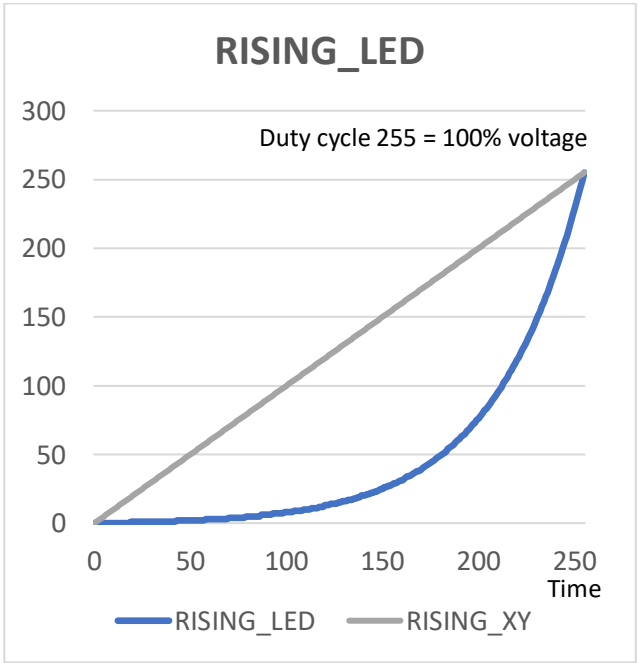
PWM (Pulse Width Modulation) is a technique used to control the average power delivered to an electrical device by varying the width of a series of pulses. Essentially, it's a digital method of creating an analog effect by rapidly switching a signal on and off

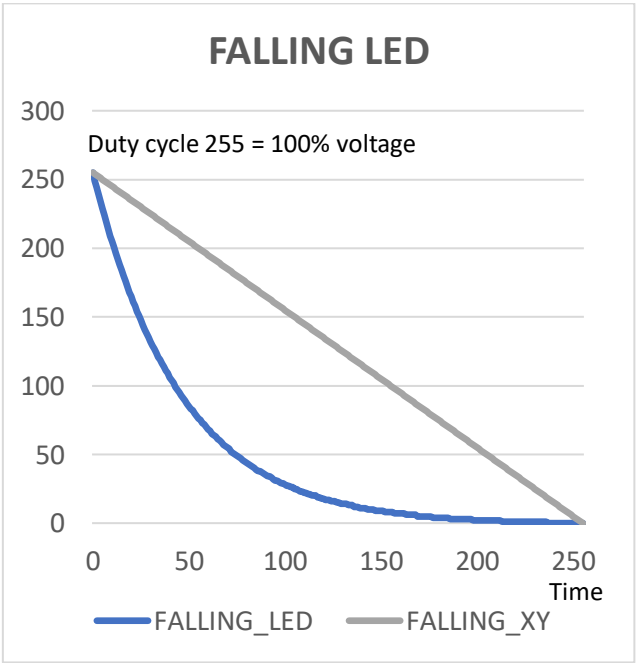


24 different mathematical PWM curves are designed: 2 linear, 4 exponential, 10 power, 4 Bézier and 4 Sine functions. They are intended to smooth out physical conditions that have a nonlinear relationship between time and function. E.g. acceleration and deceleration DC motors (trains) and servo.

LED fading

PWM is a good way to control the brightness of LEDs. However, the relationship between the duty cycle and the perceived brightness is not linear at all. Humans perceive the brightness change non-linearly. The human eye responds to light in a logarithmic fashion, and has a better sensitivity at low luminance than high luminance.





Arduino example: MTD2A/examples/math_fade_LED at main · MTD2A/MTD2A
DEMO video: https://youtu.be/8TV6nOdXBno

Example of a soft blink

```
MTD2A_binary_output red_LED   ("Red LED", 400, 400);  // 0.4 sec light, 0.4 sec no light
bool switchUpDown = true, printFlag = true;

void setup() {
  Serial.begin(9600);
  while (!Serial) { delay(10); } // ESP32 Serial Monitor ready delay
  byte RED_LED_PIN   = 9;
  red_LED.initialize   (RED_LED_PIN);
}

void loop() {
  if (printFlag) {
    Serial.println (F("Red LED SOFT blink (like an incandescent bulb) using PWM curves"));
    printFlag = false;
  }

  if (switchUpDown) {
    if (red_LED.get_processState () == COMPLETE) {
      // Light up during 200 milliseconds and keep full lightning for 300 milliseconds
      red_LED.set_timers (200, 0, 300);
      red_LED.activate(MIN_PWM_VALUE, MAX_PWM_VALUE, RISING_LED);
      switchUpDown = false;
    }
  }
  else {
    if (red_LED.get_processState () == COMPLETE) {
      // Light down during 200 milliseconds and keep zero lightning for 300 milliseconds
      red_LED.set_timers (200, 0, 300);
      red_LED.activate(MAX_PWM_VALUE, MIN_PWM_VALUE, FALLING_LED);
      switchUpDown = true;
    }
  } // switchUpDown
  MTD2A_loop_execute();
} // loop
```

Arduino example: [MTD2A/examples/blink_LED at main · MTD2A/MTD2A]
DEMO video: [https://youtu.be/nufAm2gE6pM]

Power functions

Power function is for smoothing acceleration and deceleration.

For example, using power functions is suitable for shuttle service with model trains. The locomotive accelerates smoothly up to speed, runs at a fixed speed for a while, decelerates smoothly down to speed and comes to a complete stop. After a short pause, the process is repeated in the opposite direction.

Inertia and friction

Starting and stopping of train (locomotive and cars) involves both inertia and friction.

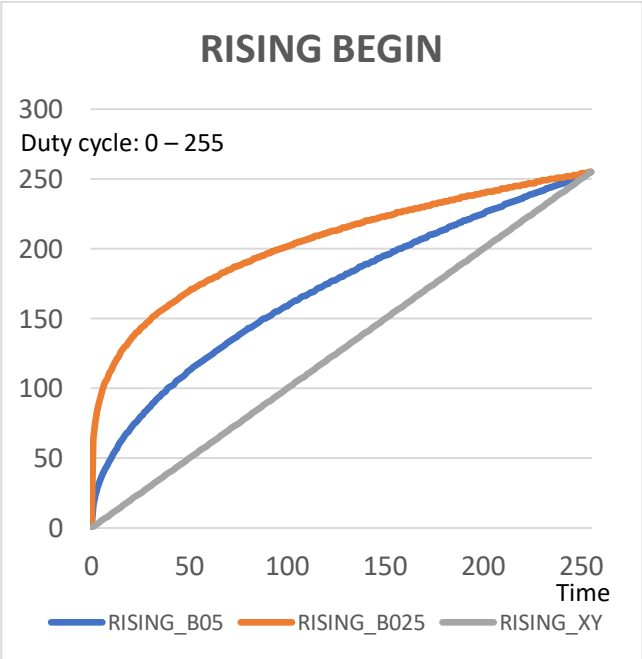
Inertia is the property of an object that causes it to resist changes to its state of motion. To start or stop motion, an unbalanced force is needed to overcome inertia and, in most real-world scenarios, friction.

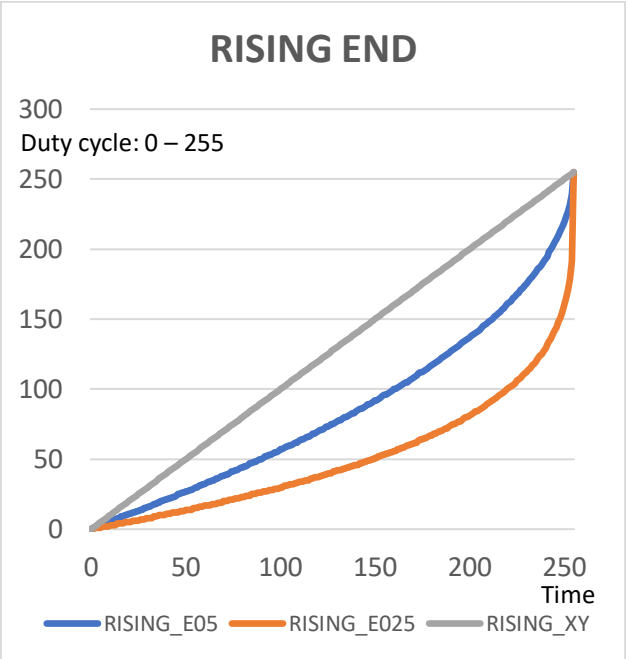
- A car accelerating requires a force to overcome the inertia of the car, which wants to remain stationary.
- A rolling ball eventually stops due to friction, but without friction, it would continue rolling indefinitely.
- Pushing a box: You need to push with enough force to overcome static friction and get the box moving. Once moving, you need to continue applying force to overcome kinetic friction and maintain the speed.

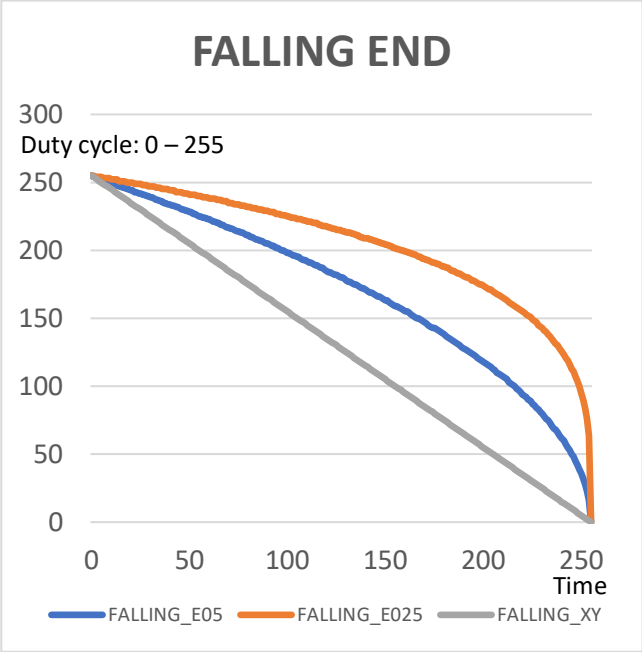
Inertia and friction are fundamental concepts in physics that explain why objects move (or don't move) and how motion can be started and stopped, and the relationship between time and function is **nonlinear**.

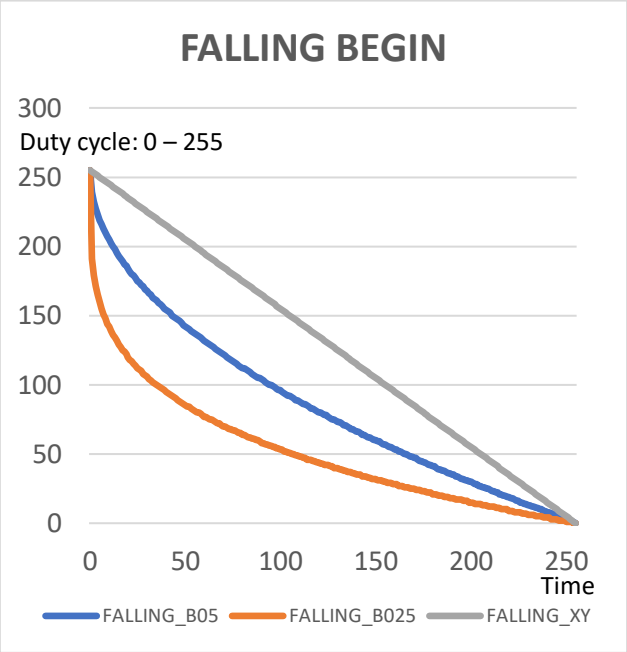
Furthermore there can be an effect in DC brushed motors called "armature reaction." Depending on how the motor is manufactured there can be an asymmetry between forward and reverse rotation.

The following 8 power functions will result in more smooth train acceleration and deceleration.





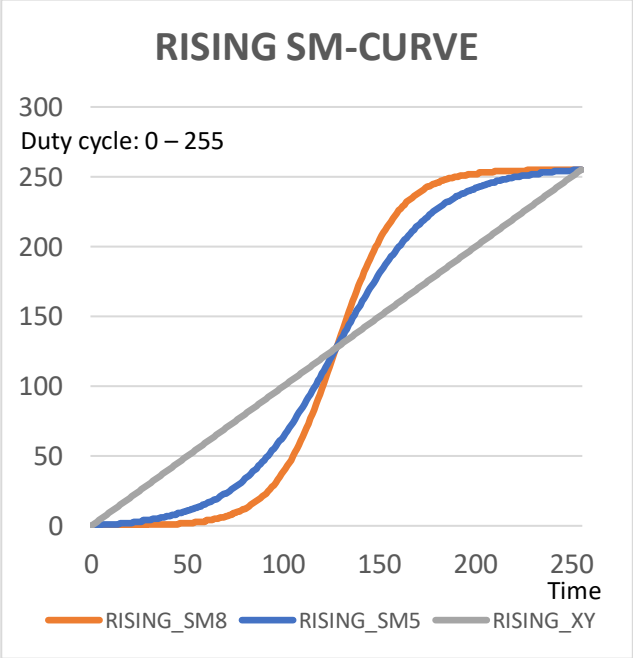


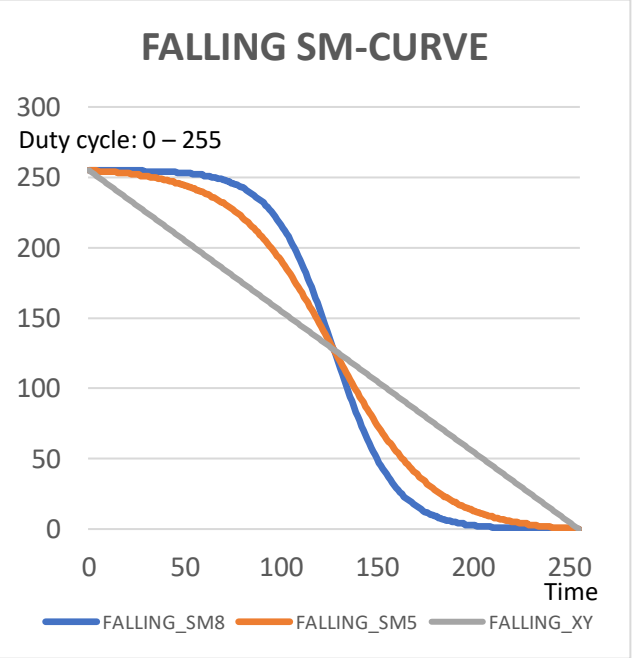


Arduino example: [MTD2A/examples/PWM_power_curves at main · MTD2A/MTD2A]
DEMO video: https://youtu.be/Fi9D1hrzT9M

S-curves

Sigmoid S-curves can be used where soft acceleration and deceleration is preferable. For example, to control a (heavily loaded) servo motor from point a to b, or a soft start acceleration of a locomotive and a soft end deceleration to continued constant speed.

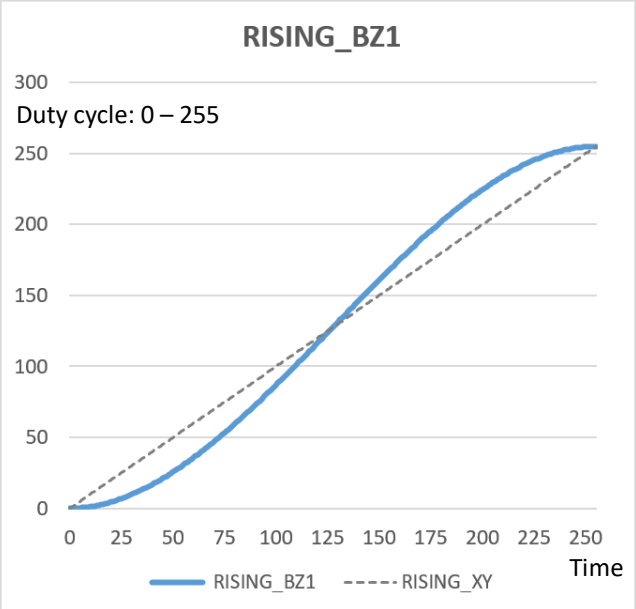


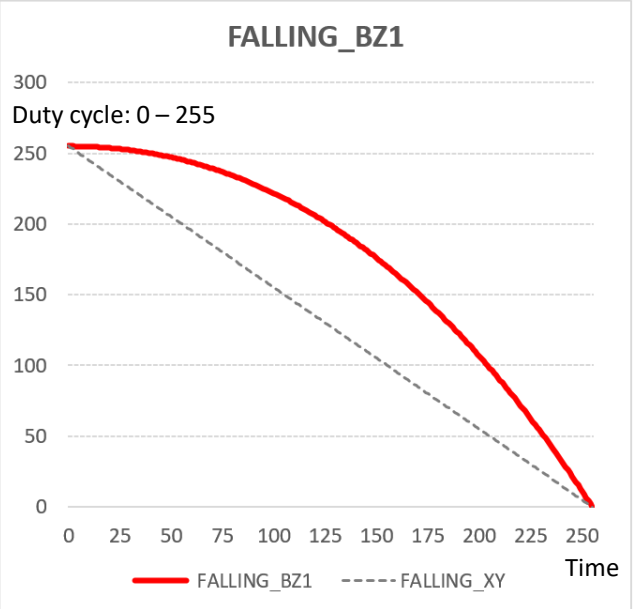


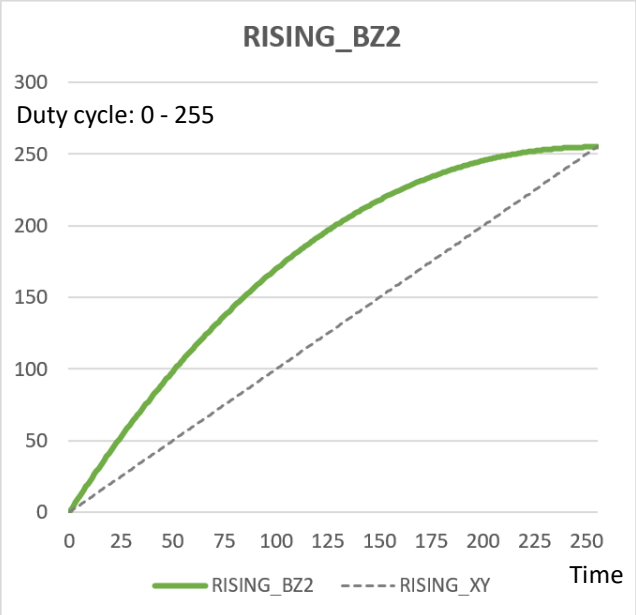
Arduino example: [MTD2A/examples/servo_math_curve at main · MTD2A/MTD2A]
DEMO video: https://youtu.be/rhQtu0iKFl8

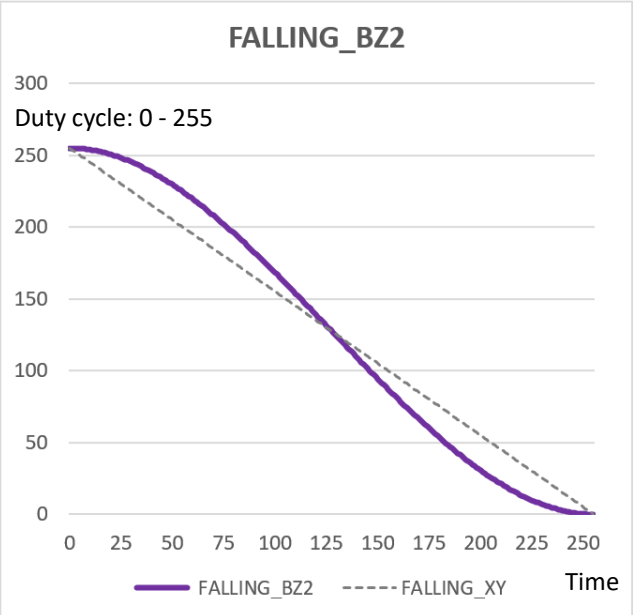
Bézier curves

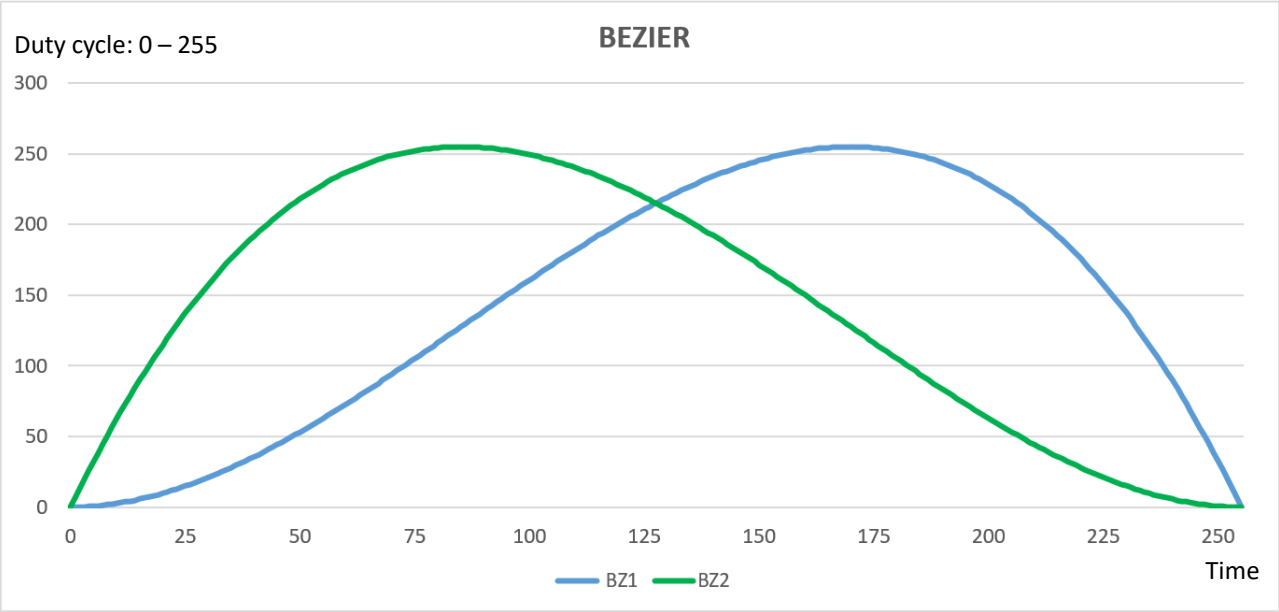
Bézier curves resemble Sigmoid S-curves and can be used where a smoother acceleration and deceleration is preferred. For example, to control a (heavily loaded) servo motor from point a to b, or a soft start acceleration of a locomotive and a soft end deacceleration to continued constant speed.







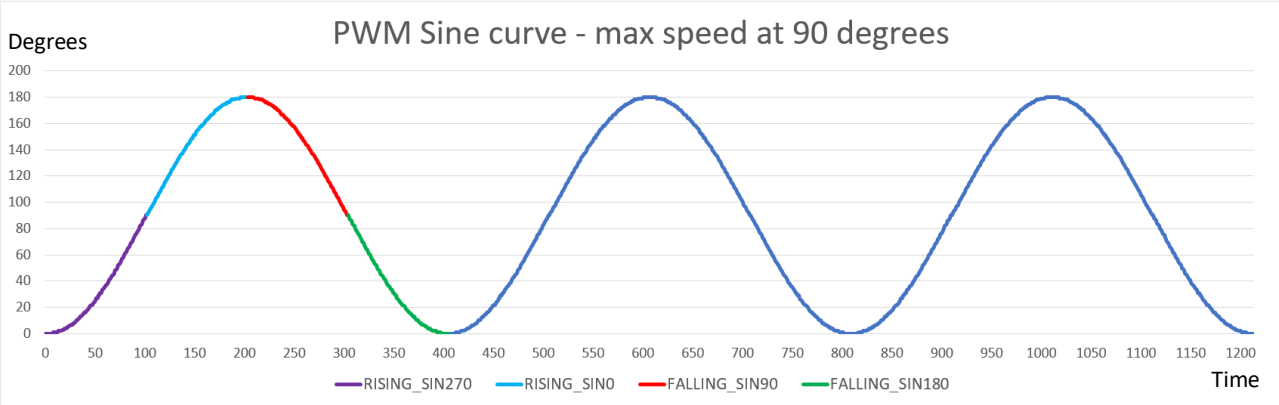


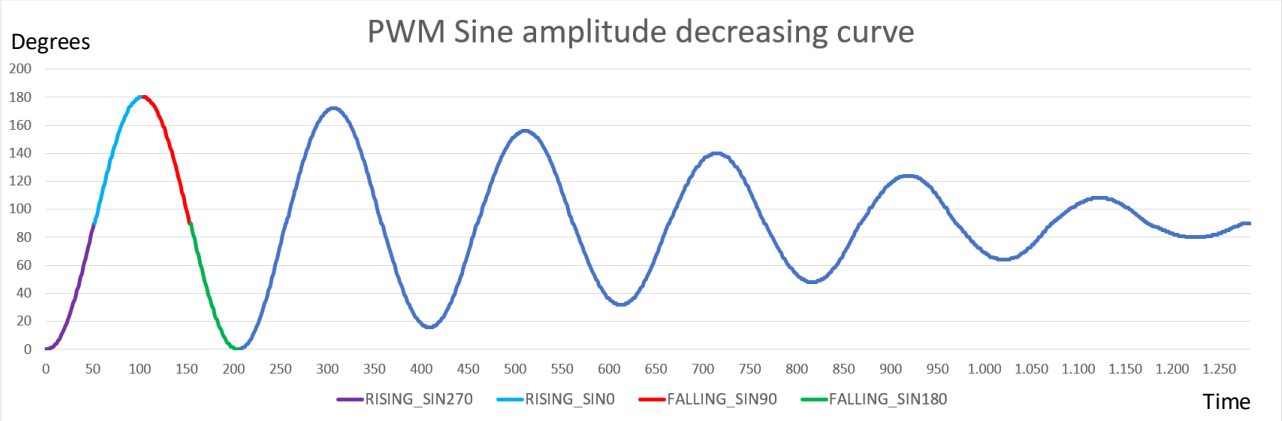


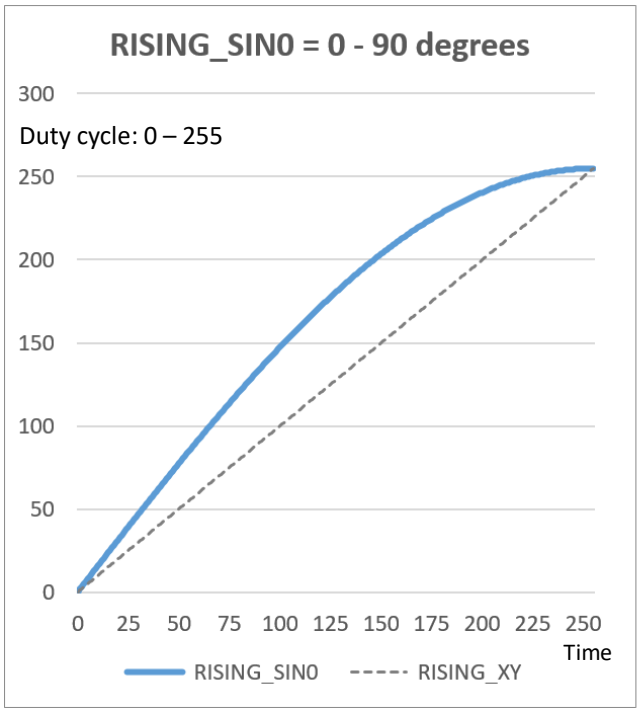
Arduino example: [MTD2A/examples/advanced_servo_curves at main · MTD2A/MTD2A]
DEMO video: https://youtu.be/parVpl98mDk

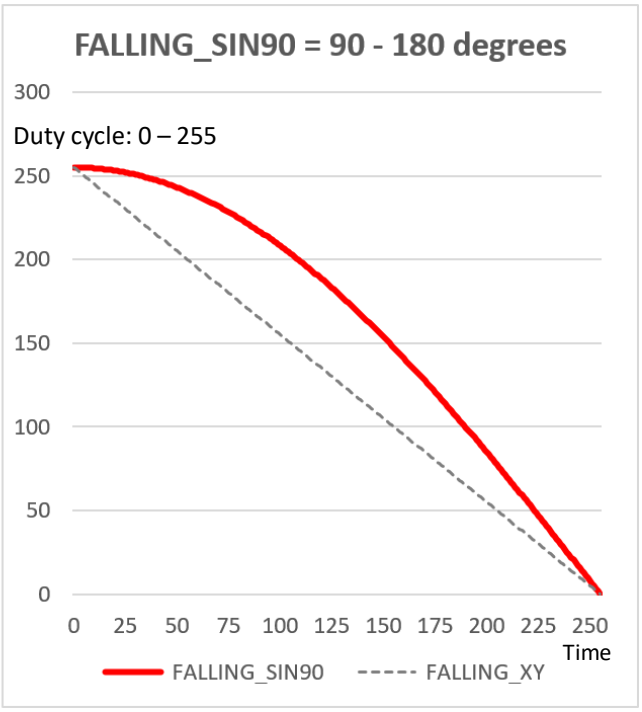
Sine curve

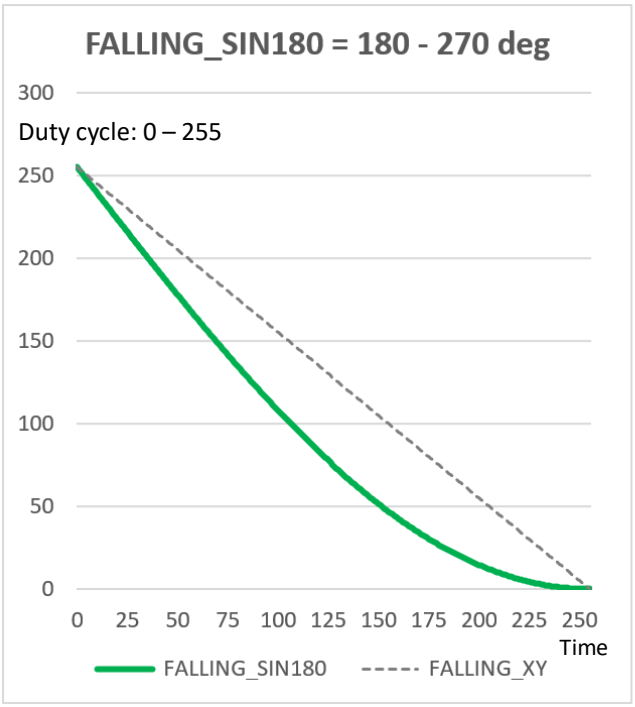
Sine curve is divided into four parts of each of 90 degrees, and distributed together in the range 0 – 360 degrees. In this way, it is possible to combine the individual parts into various new functions.

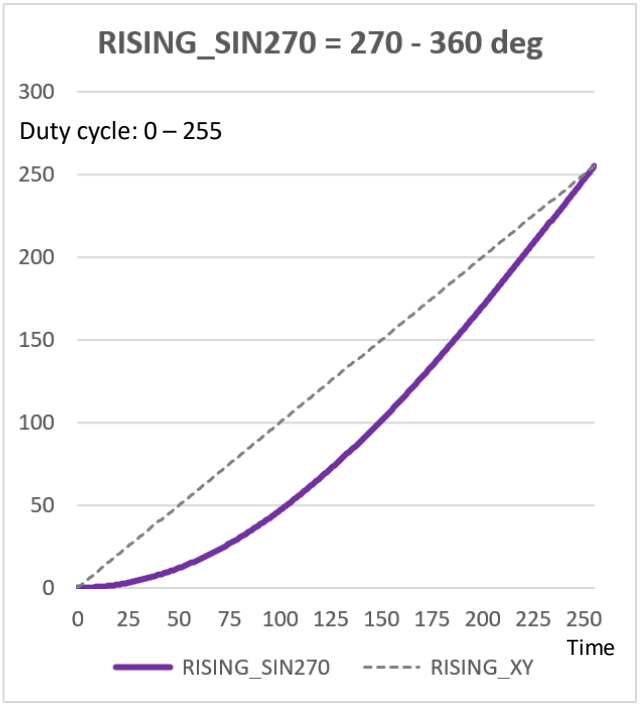


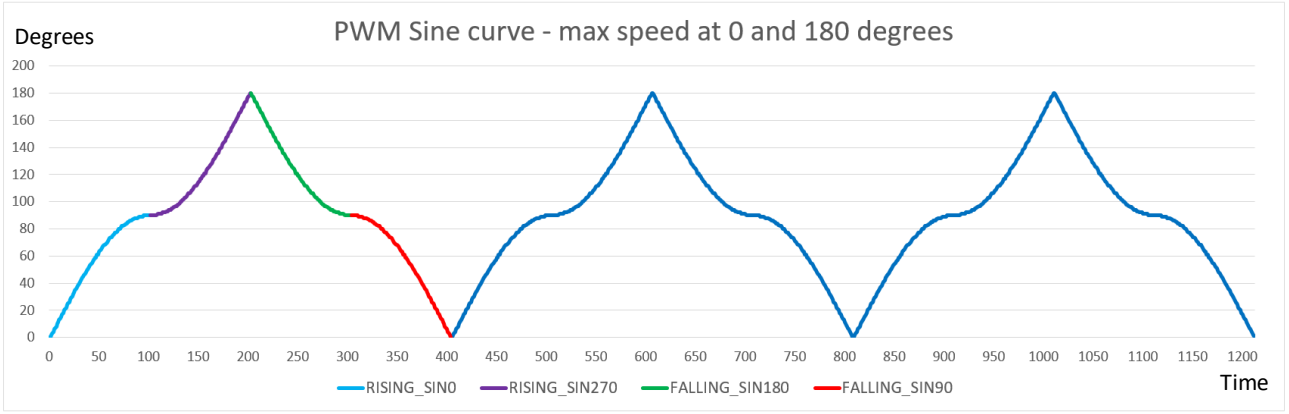












Arduino example: [MTD2A/examples/advanced_servo_curves at main · MTD2A/MTD2A]
DEMO video: [https://youtu.be/parVpl98mDk]

Pendulum operation using math PWM curves and H-bridges: https://youtu.be/1i-cGc6Dk4E

The different curves are named as global constants (MTD2A_const.h):

MIN_PWM_VALUE

MAX_PWM_VALUE

NO_CURVE
RISING_XY
RISING_LED
RISING_SM8
RISING_B05
FALLING_B05
RISING_BZ1
RISING_SIN0

FALLING_XY
FALLING_LED
RISING_SM5
RISING_B025
FALLING_B025
FALLING_BZ1
FALLING_SIN90

FALLING_SM8
RISING_E05
FALLING_E05
RISING_BZ2
RISING_SIN270

FALLING_SM5
RISING_E025
FALLING_E025
FALLING_BZ2
FALLING_SIN180

Please write if there is a desire for other PWM curves: MTD2A@jorgen-madsen.dk

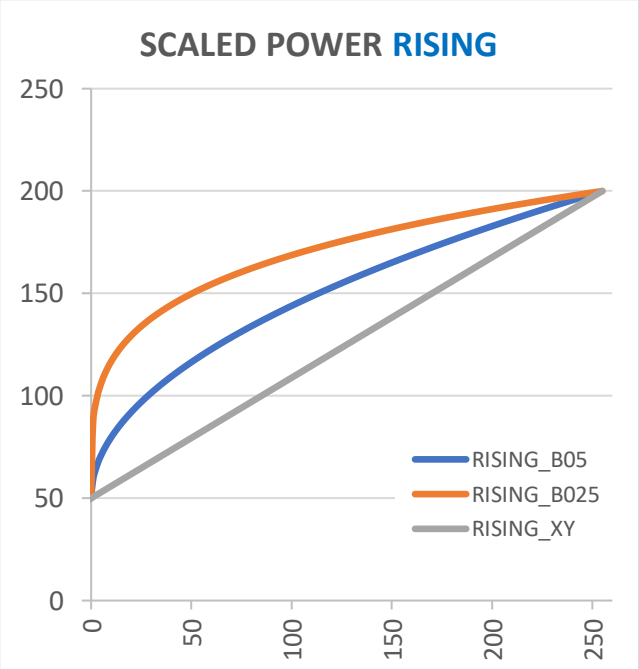
Scaling

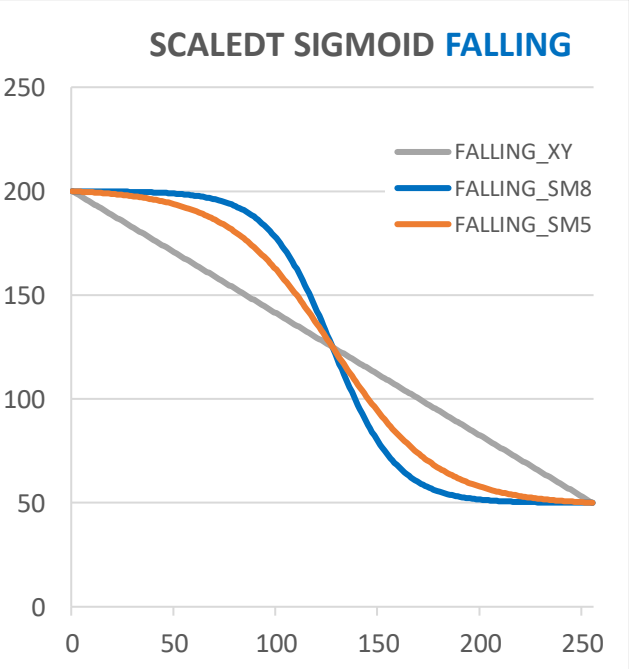
All curves can be scaled. If the value beginValue and the value endValue are defined, the entire curve is scaled within the time period outputTimeMS

Examples of scaling:

```
motor.activate (50, 200, RISING_XY);
motor.activate (50, 200, RISING_B05);
motor.activate (50, 200, RISING_B025);
```

```
servo.activate (200, 50, FALLING_XY);
servo.activate (200, 50, FALLING_SM8);
servo.activate (200, 50, FALLING_SM5);
```





outputValue and PWM

Refer to the outputValue section for an introduction.

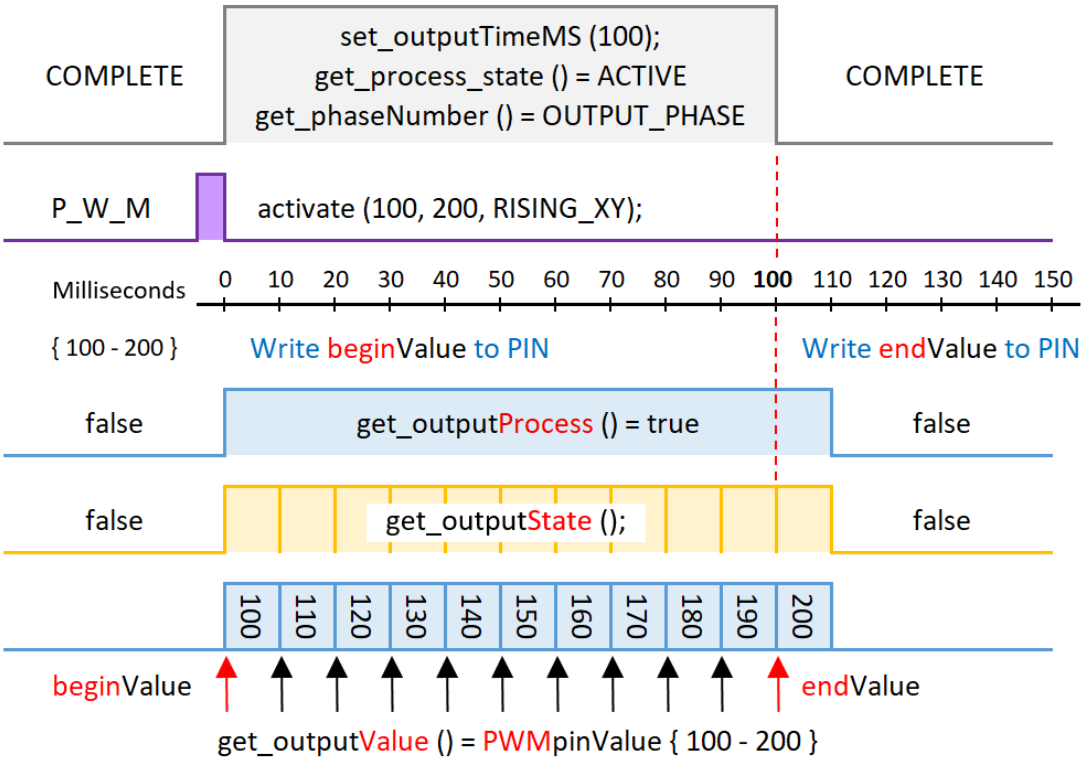
The individual PWM curves are calculated pointwise and evenly over `outputTimeMS` period, and with temporal steps determined by the `globalDelayTimeMS` See the in-depth explanation of time management and synchronization in the document MTD2A_base.pdf.

The individual calculated points on the curve can be read with the function: `object_name.get_outputValue ();`

`object_name.get_outputState ();` is `true` when there is **new data** to read. This means that if the same value occurs several times in a row, the function only returns `true` the first time the value occurs.

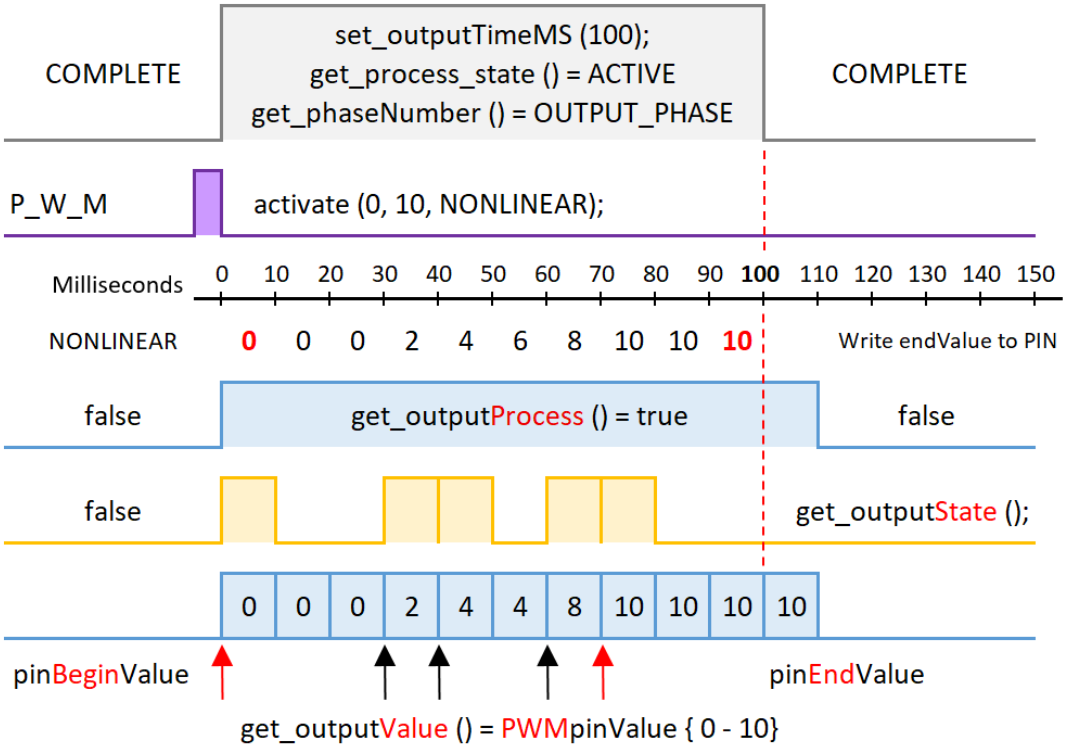
Whether or not to optimize is controlled with the function:
`object_name.set_outputOptimize ( {ENABLE | DISABLE} );`

The current configuration can be read with the function: `object_name.get_outputOptimize ( );`



Example of linear PWM function RISING_XY

```
MTD2A_binary_output PWM_curve ("PWM_curve", 3000); // 3 seconds slow movement
Servo myServo;
// Initialize and loop etc.
if (PWM_curve.get_processState() == COMPLETE) {
      PWM_curve.activate (0, 180, RISING_XY); // begin 0 and end 180 degrees
}
if (PWM_curve.get_outputState () == true) {
  myServo.write( PWM_curve.get_outputValue() );
}
```

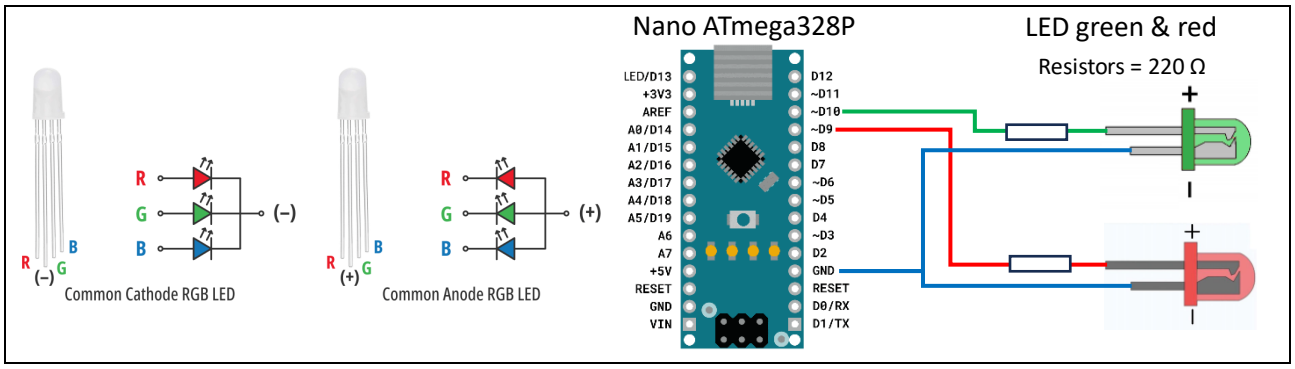


Example of nonlinear PMW Function RISING_SM5

```
MTD2A_binary_output PWM_curve ("PWM_curve", 3000); // 3 seconds slow movement
Servo myServo;
// Initialize and loop etc.
if (PWM_curve.get_processState() == COMPLETE) {
      PWM_curve.activate (0, 180, RISING_SM5); // begin 0 and end 180 degrees
}
if (PWM_curve.get_outputState () == true) {
  myServo.write( PWM_curve.get_outputValue() );
}
```

Examples of configuration

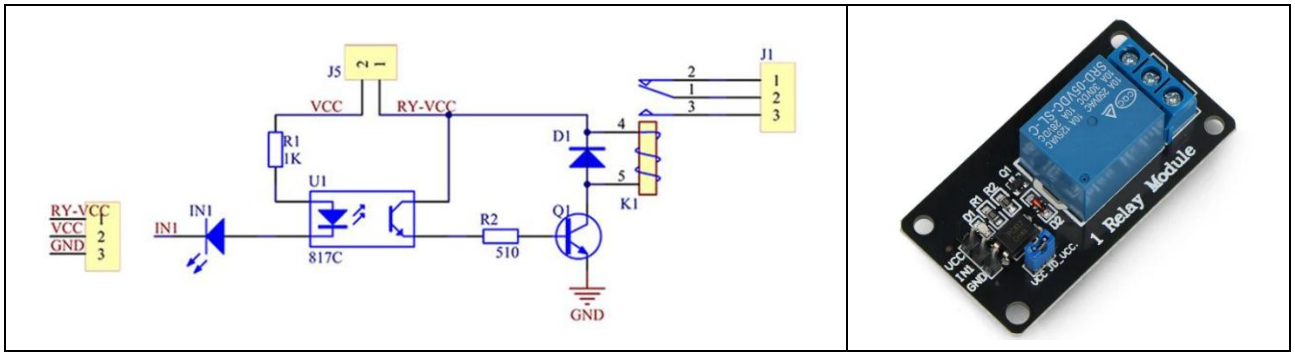
A multi-colored LED with a common **cathode** is activated by changing from LOW to HIGH. PWM 0 -> 255.
A multi-colored LED with a common **anode** is activated by changing from HIGH to LOW. PWM 255 -> 0.



Example of a green LED that waits for half a second and then lights up for half a second.

```
1. MTD2A_binary_output green_LED ("Green LED", 500, 500);
2. green_LED.initialize (10);
3. green_LED.activate ();
4. MTD2A_loop_execute ();
```

A standard relay with optocoupler that is activated by going from HIGH to LOW.



Example of an optocoupled relay that waits for half a second and is then activated for half a second.
All default values are the inverse of what is the default. Therefore, all parameters must be specified.

```
1. MTD2A_binary_output opto_relay ("Opto relay",  500, 500, 0,  BINARY, LOW, HIGH);
2. opto_relay.initialize (12, NORMAL, HIGH);
3. opto_relay.activate ();
4. MTD2A_loop_execute ();
```

Alternatively, INVERTED, which inverts (reverses) all PIN output values.

```
1. MTD2A_binary_output opto_relay ("Opto relay",  500, 500);
2. opto_relay.initialize(12, INVERTED);
3. opto_relay.activate ();
4. MTD2A_loop_execute ();
```

Set get functions

Set functions Green is default when no parameter	Comment
set_outputMode ({ BINARY | P_W_M });	Select binary or PWM output
set_PWMcurveType (PWM curve number/name)	Select PWM curve to use in PWM mode
set_outputOptimize ({ENABLE | DISABLE});	ENABLE : outputState is only when outputValue change from last outputValue DISABLE: outputState always true
set_pinWriteValue ({BINARY {HIGH | LOW} / PWM {0-255}) , {BINARY | P_W_M});	Write directly to output PIN (if enabled) **Optional** parameter. No default.
set_pinWriteToggl ({ENABLE | DISABLE});	Enable or disable PIN writing
set_pinWriteMode ({NORMAL | INVERTED});	Invert all output (PIN writing)
set_outputTimeMS ({0 - MAX_TIME_MS })	Set new **output** time period milliseconds
set_beginDelayMS ({0 - MAX_TIME_MS })	Set new **begin** delay time period millisec.
set_endDelayMS ({0 - MAX_TIME_MS })	Set new **end** delay time period millisec.
set_timers (outputTimeMS, beginDelayMS, endDelayMS)	Set one, two or three timers milliseconds **All optional** parameters. No defaults.
set_outputTimer ({ STOP_TIMER | RESET_TIMER})	Stop timer process immediately or reset
set_beginTimer ({ STOP_TIMER | RESET_TIMER})	Stop timer process immediately or reset
set_endTimer ({ STOP_TIMER | RESET_TIMER})	Stop timer process immediately or reset
set_loopActivate ({ENABLE | DISABLE}, {0-65.535})	Repeat the activation process until DISABLE or activateCount has reached activateNumber **Optional** parameter. Default = 0
set_debugPrint ({ENABLE | DISABLE});	Enable print phase number and text
set_errorPrint ({ENABLE | DISABLE});	Enable error messages

Get functions	Comment
get_outputMode (); return bool { BINARY | P_W_M }	Currently selected binary or PWM output
get_outputOptimize (); return bool {ENABLE | DISABLE}	Currently selected active or not
get_PWMcurveType (); Return uint8_t curve number	Current selected PWM curve
get_processState (); return bool {ACTIVE | COMPLETE}	Process state.
get_pinWriteMode (); return bool {NORMAL | INVERTED}	Output is normal or inverted
get_pinWriteToggl (); return bool {ENABLE | DISABLE}	Write to PIN is enabled or disabled
get_outputValue () return uint8_t {HIGH | LOW} / {0- 255}	Current output value
get_outputState ();return bool {true | false}	Indicates when to call get_outputValue ();
get_outputProcess (); return bool {ACTIVE | COMPLETE}	Output process state
get_phaseChange (); return bool {true | false}	Momentarily phase change (one loop time)
get_phaseNumber (); return uint8_t {0- 4}	Reset = 0, begin = 1, output = 2, end = 3, complete = 4.
get_setBeginMS (); return uint32_t milliseconds	Start time for **begin delay** process
get_setOutputMS (); return uint32_t milliseconds	Start time for **output** process
get_setEndMS (); return uint32_t milliseconds	Start time for **end delay** proces
get_loopActivate (); return bool {ENABLE | DISABLE}	Repeat the whole process until DISABLE
get_activateNumber (); return uint16_t number	Number of activations to perform
get_activateCounter ();return uint16_t number	Current number of activations.
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_output:
--------------------
  objectName    : LED_1
  processState  : ACTIVE
  phaseText     : [3] End delay
  debugPrint    : DISABLE
  globalDebugPr: DISABLE
  errorPrint    : DISABLE
  globalErrorPr: ENABLE
  errorNumber   : 0 OK
  outputTimeMS  : 2000
  beginDelayMS  : 0
  endDelayMS    : 2000
  outputMode    : PWM
  PWMcurveType  : 0(NO_CURVE)
  PWMloopCount  : 200
  beginValue    : 10
  endValue      : 0
  outputValue   : 255
  loopActivate  : DISABLE
  activateNumbr: 0(infinite)
  activateCount: 0
  pinNumber     : 9
  pinWriteToggl: ENABLE
  pinWriteMode  : NORMAL
  pinStartValue: 0
  PinWriteValue: 0
  setOutputMS   : 2014
  setBeginMS    : 0
  setEndMS      : 4015
```