class Encoder – Quadrature Encoder for i.MXRT MCUs

This class provides the Quadrature Encoder Service.

Example usage:

```
# Samples for Teensy
#

from machine import Pin, Encoder

qe = Encoder(0, Pin(0), Pin(1)) # create Quadrature Encoder object
qe.value()                      # get current counter values
qe.value(0)                     # Set value and cycles to 0
qe.init(cpc=128)                # Specify 128 counts/cycle
qe.init(index=Pin(3))           # Specify Pin 3 as Index pulse input
qe.deinit()                     # turn off the Quadrature Encoder
qe.init(compare=64)             # Set create a compare match event at count 64
qe.irq(qe.COMPARE, handler)     # Call the function handler at a compare match

qe                              # show the Encoder object properties
```

Constructors

class machine.**Encoder**(*id*, *input_a*, *input_b*, ***, *keyword_arguments*)

Construct and return a new quadrature encoder object using the following parameters:

- id: The number of the Encoder. The range is board-specific, starting with 0. For i.MX RT1015 and i.MX RT1021 based boards, this is 0..1, for i.MX RT1052, i.MX RT106x and i.MX RT11xx based boards it is 0..3.
- *input_a* and *input_b* are the Quadrature encoder inputs, which are usually machine.Pin objects, but a port may allow other values, like integers or strings, which designate a Pin in the machine.PIN class.

Keyword arguments:

- *phase_a*=value. A Pin specifier telling to which pin the phase a of the encoder is connected. This is also a positional argument for the constructor and init() function.
- *phase_b*=value. A Pin specifier telling to which pin the phase b of the encoder is connected. This is also a positional argument for the constructor and init() function.
- *filter_ns*=value. Specifies a ns-value for the minimal time a signal has to be stable at the input to be recognized. The code does the best effort to configure the filter. The largest value is 20400 for the i.MXRT102x and 17000 for the i.MXRT105x/i.MXRT106x (1000000000 * 2550 * 4 / CPU_CLK). A value of 0 sets the filter sets the filter off.
- *index*=value. A Pin specifier telling to which pin the index pulse is connected. At a rising slope of the index pulse the position counter is set to the init value and the cycles counter in increased by one.

- *home*=value. A Pin specifier telling to which pin the home pulse is connected. At a rising slope of the home pulse the position counter is set to the init value, but the cycles counter is not changed.
- *match*=value. A Pin specifier telling to which pin the match output is connected. This output will have a high level as long as the position counter matches the compare value. The signal is generated by the encoder logic and requires no further software support. The pulse width is defined by the input signal frequency and can be very short, like 20ns, or stay, if the counter stops at the match position.
- *cpc*=value. Specify the number of counts per cycle. Since the Encoder counts all four phases of the input signal, the cpc value has to be four time the ppr value given in the encoder data sheet. The position counter will count up from the 0 up to cpc - 1, and then reset to the init value and increase the cycles counter by one. The default is: no cpc set. In that case the position counter overflows at 2**32 - 1. When counting down, the cycles counter changes at the transition from 0 to cpc - 1.
- *compare*=value. Sets a position counter compare value. When the counter matches this value, a callback function can called and the match output will get a high level.
- *signed*=False|True tells, whether the value return by Encoder.value() is signed or unsigned. The default is `True`.

The arguments phase_a, phase_b and filter are generic across ports, all other arguments are port-specific.

Methods

`Encoder.`**`init`**`(`*`keyword_arguments`*`)`

Modify settings for the Encoder object. See the above constructor for details about the parameters.

`Encoder.`**`deinit`**`()`

Stops the Encoder, disables interrupts and releases the resources used by the encoder. On Soft Reset, all instances of Encoder and Counter are deinitialized.

`value=Encoder.value([value])`

Get or set the current position counter of the Encoder as signed or unsigned 32 bit integer, depending on the signed=xxx keyword option of init().

With no arguments the actual position counter value is returned.

With a single *value* argument the position counter is set to that value and the cycles counter is cleared. The methods returns the previous value.

`cycles=Encoder.cycles([value])`

Get or set the current cycles counter of the Encoder as signed 16 bit integer.

With no arguments the actual cycles counter value is returned.

With a single *value* argument the cycles counter is set to that value. The position counter is not changed. The methods returns the previous value.

If the value returned by Encoder.value() is unsigned, the total position can be calculated as cycles() * cpc + value(). If the total position range is still too small, you can create your own cycles counter using the irq() callback method.

`Encoder.`**`compare`**`([`*`value`*`])`

Get or set the position compare value.

With no arguments the actual compare value is returned. With a single *value* argument the compare value is set to that value.

`Encoder.`**`irq`**`(`*`trigger=event, handler=handler, hard=False`*`)`

Specifies, that the *handler* is called when the respective *event* happens.

event may be:

- Encoder.COMPARE Triggered when the position counter matches the compare value.
- Encoder.ROLL_OVER Triggered when the position counter rolls over from the highest to the lowest value.
- Encoder.ROLL_UNDER Triggered when the position counter rolls under from the lowest to the highest value.

The callback function *handler* receives a single argument, which is the Encoder object. All events share the same callback. The event which triggers the callback can be identified with the Encoder.status() method. The argument *hard* specifies, whether the callback is called as a hard interrupt or as regular scheduled function. Hard interrupts have always a short latency, but are limited in that they must not allocate memory. Regular scheduled functions are not limited in what can be used, but depending on the load of the device execution may be delayed. Under low load, the difference in latency is minor.

The default arguments values are trigger=0, handler=None, hard=False. The callback will be disabled, when called with handler=None.

The position match event is triggered as long as the position and compare value match. Therefore the position match callback is run in a one-shot fashion, and has to be enabled again when the position has changed.

`Encoder.`**`status`**`()`

Returns the event status flags of the recent handled Encoder interrupt as a bitmap. The assignment of events to the bits are:

- 0: Transition at the HOME signal. (*)
- 1: Transition at the INDEX signal. (*)
- 2: Watchdog event. (*)
- 3 or Encoder.COMPARE: Position match event.
- 4: Phase_A and Phase_B changed at the same time. (*)
- 5 or Encoder.ROLL_OVER: Roll-Over event of the position counter.
- 6 or Encoder.ROLL_UNDER: Roll-Under event of the position counter.
- 7: Direction of the last count. 1 for counting up, 0 for counting down.

(*) These flags are defined, but not (yet) enabled.

The Encoder was tested to work up to 25MHz on a Teensy. It may work at higher frequencies as well, but that was the limit of the test set-up.

class Counter– Signal counter for i.MXRT MCUs

This class provides a Counter service using the Quadrature Encoder module

Example usage:

```
# Samples for Teensy
#

from machine import Pin, Counter

counter = Counter(0, Pin(0))          # create Counter object
counter.value()                       # get current counter value
counter.value(0)                      # Set the counter to 0
counter.init(cpc=128)                 # Specify 128 counts/cycle
counter.deinit()                      # turn off the Counter
counter.init(compare=1000)            # Set create a compare match event at count 1000
counter.irq(Counter.COMPARE, handler) # Call the function handler at a compare match

counter                               # show the Counter object properties
```

Constructors

class machine.**Counter**(*id*, *input*, **, *keyword_arguments*)

Construct and return a new Counter object using the following parameters:

- id: The number of the Counter. The range is board-specific, starting with 0. For i.MX RT1015 and i.MX RT1021 based boards, this is 0..1, for i.MX RT1052, i.MX RT106x and i.MX RT11xx based boards it is 0..3.
- *input* is the Counter input pin, which is usually a machine.Pin object, but a port may allow other values, like integers or strings, which designate a Pin in the machine.PIN class.

Keyword arguments:

- *input*=value. A Pin specifier telling to which pin the input of the counter is connected. This is also a positional argument for the constructor and init() function.
- *direction*=value. Specifying the direction of counting. Suitable values are:
  - Counter.UP: Count up, with a roll-over to 0 at 2**48-1.
  - Counter.DOWN: Count down, with a roll-under to 2**48-1 at 0.
  - a machine.Pin object. The level at that pin controls the counting direction. Low: Count up, High: Count down.
- *filter_ns*=value. Specifies a ns-value for the minimal time a signal has to be stable at the input to be recognized. The code does the best effort to configure the filter. The largest value is 20400 for the i.MXRT102x and

17000 for the i.MXRT105x/i.MXRT106x (1000000000 * 2550 * 4 / CPU_CLK). A value of 0 sets the filter off.

- *match*=value. A Pin specifier telling to which pin the match output is connected. This output will have a high level as long as the lower 32 bit of the counter value matches the compare value. The signal is generated by the encoder logic and requires no further software support.
- *cpc*=value. Specify the number of counts per cycle.The counter will count up from the 0 up to cpc - 1, and then reset to 0 and increase the cycles counter by one. The default is: no cpc set. In that case the counter overflows at 2**32 - 1. If the counting direction is DOWN, then the cycles counter is decreased when counting from 0 to cpc-1.
- *compare*=value. Sets counter compare value. When the counter matches this value, a callback function can called and the match output will get a high level.
- *signed*=False|True tells, whether the value returned by Counter.value() is signed or unsigned. The default is `True`.

The arguments input, direction and filter are generic across ports, all other arguments are port-specific.

Methods

`Counter.`**`init`**`(`*`keyword_arguments`*`)`

Modify settings for the Counter object. See the above constructor for details about the parameters.

`Counter.`**`deinit`**`()`

Stops the Counter, disables interrupts and releases the resources used by the encoder. On Soft Reset, all instances of Encoder and Counter are deinitialized.

`value=Counter.value([value])`

Get or set the current event value of the Counter. The value is returned as a signed or unsigned 32 bit integer, as defined with the signed=True/False option of init()

With a single *value* argument the counter is set to the lower 32 bits of that value, and the cycles counter to the bits 32-47 of the supplied number. The methods returns the previous value.

`cycles=Counter.cycles([value])`

Get or set the current cycles counter of the counter as signed 16 bit integer. The value represents the overflow or underflow events of the 32bit basic counter. A total count can be calculated as cycles() * 0x100000000 + value(). If the total count range is still too small, you can create your own overflow counter using the irq() callback method.

With no arguments the actual cycles counter value is returned.

With a single *value* argument the cycles counter is set to that value. The base counter is not changed. The methods returns the previous value.

`Counter.`**`compare`**`([`*`value`*`])`

Get or set the counter compare value.

With no arguments the actual compare value is returned. With a single *value* argument the compare value is set to that value. Note that only the lower 32 bit are significant.

`Counter.`**`irq`**`(`*`trigger=event, handler=handler, hard=False`*`)`

Specifies, that the *handler* is called when the respective *event* happens.

event may be:

- Counter.COMPARE Triggered when the positions counter matches the compare value.
- Counter.ROLL_OVER Triggered when the position counter rolls over from the highest to the lowest value.
- Counter.ROLL_UNDER Triggered when the position counter rolls under from the lowest to the highest value.

The callback function *handler* receives a single argument, which is the Counter object. All events share the same callback. The event which triggers the callback can be identified with the Counter.status() method. The argument *hard* specifies, whether the callback is called as a hard interrupt or as regular scheduled function. Hard interrupts have always a short latency, but are limited in that they must not allocate memory. Regular scheduled functions are not limited in what can be used, but depending on the load of the device execution may be delayed. Under low load, the difference in latency is minor.

The default arguments values are trigger=0, handler=None, hard=False. The callback will be disabled, when called with handler=None.

The counter match event is triggered as long as the lower 32 bit of the counter and compare value match. Therefore the counter match callback is run in a one-shot fashion, and has to be enabled again when the counter value has changed.

`Counter.`**`status`**`()`

Returns the event status flags of the recent handled Counter interrupt as a bitmap. The assignment of events to the bits are:

- 0: Transition at the HOME signal. (*)
- 1: Transition at the INDEX signal. (*)
- 2: Watchdog event. (*)
- 3 or Counter.COMPARE: Position match event.
- 4: Phase_A and Phase_B changed at the same time. (*)
- 5 or Counter.ROLL_OVER: Roll-Over event of the counter.
- 6 or Counter.ROLL_UNDER: Roll-Under event of the counter.
- 7: Direction of the last count. 1 for counting up, 0 for counting down.

(*) These flags are defined, but not (yet) enabled.

The counter was tested up to 25MHz. It may work at higher frequencies as well, but that was the limit of the test set-up.

Pin Assignment

Pins are specified in the same way as for the Pin class. The pins available for an assignment to the Encoder or Counter are:

IMXRT1010_EVK:

Not supported.

IMXRT1020_EVK:

Pins D0 and D1.

IMXRT1050_EVK, **IMXRT1050_EVKB**, **IMXRT1060_EVK**, **IMXRT1064_EBK**:

Pins D2, D4, D5, D8, D9, D10, D11, D12, D13, D14, D15, A4, A5. Depending on the board configuration, not all pins may be wired. Pins D2, D4 and D5 cannot be used for the match output.

IMXRT1170_EVK:

Pins D0, D1, D2.

D2 is connected to the 1G PHY chip as well. So levels may be distorted.

Teensy 4.0:

Pins 0, 1, 2, 3, 4, 5, 7, 8, 26, 27, 30, 31, 32, 33. Pin 0 and 5 share the same signal and cannot be used independently. Pins 26, 27, 30 and 31 cannot be used for the match output.

Teensy 4.1:

Pins 0, 1, 2, 3, 4, 5, 7, 8, 26, 27, 30, 31, 32, 33, 37, 42, 43, 44, 45, 46 and 47. Pins 26, 27, 30 and 31 cannot be used for the match output. Some pins are assigned to the same signal and cannot be used independently. These are:

- Pin 0, 5 and 37,
- Pin 2 and 43,
- Pin 3 and 42, and
- Pin 4 and 47.

Seeed ARCH MIX

Pins J3_14, J3_15, J4_19, J4_20, J5_15, J5_16, J5_17, J5_22, J5_23, J5_24, J5_25 and J5_26. Pins J3_14 and J3_15 cannot be used for the match output.