EmonLibCM V2.2.0 Release Note

The sole change in this release resolves a major shortcoming with the way pulse counting is handled.

Sketches compiled with this release of the library and with no further changes might not count pulses correctly, or at all. It will almost certainly be necessary to change the value of the pulse counting parameter ‘period’ taken by EmonLibCM_setPulseMinPeriod( ) to ensure correct operation.

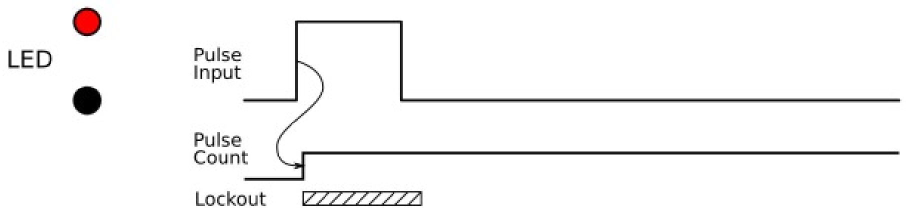
The pulse counting algorithm used by the emonTx and emonPi, and adopted by previous versions of this library, uses a hardware interrupt to recognise and count on the rising edge of the voltage pulse appearing on the input connector. When the input source is a ‘perfect’ electronic switch, either the optical pulse detector or (say) the S0 output from a flow meter, this arrangement works perfectly well. If the switching source is metallic contacts that make and break a connection, it is very likely that the contacts will bounce off each other, or there will be a wiping action, either of which is likely to give rise to one or more fleeting connections before the final state (the switch is properly open or properly closed) is achieved.

The algorithm attempts to ignore multiple input pulses coming from contact bounce by locking out and ignoring everything following the first rising edge for a time defined by setPulseMinPeriod( ) – usually 110 ms.

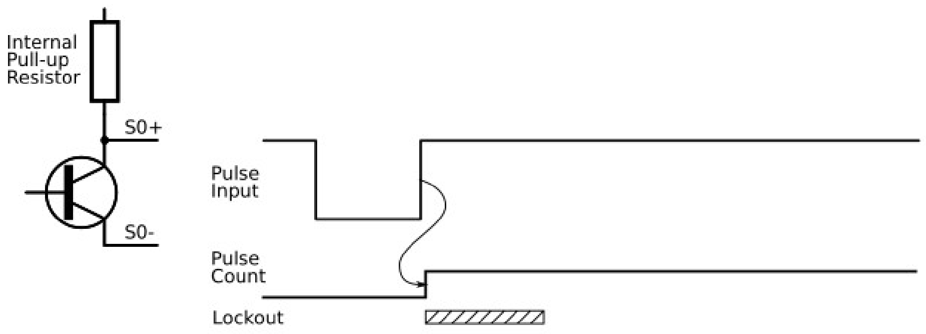
If “contact bounce” (which usually refers to any intermittent contact) occurs when opening as well as closing, it is impossible for the algorithm to work correctly, and the software will count multiple pulses where only one was intended.

In the diagrams below, the first three show how different types of input switch react. The first and second are ‘perfect’ electronic switches that have no contact bounce, these will always give the correct count. The third shows a magnetic reed relay that generates several pulses as the contacts close and bounce. The first bounce generates a pulse that is counted, but the lockout period prevents the remainder from being seen. However, the damage has been done – the first pulse and the first count should not have been there at all. The switch will remain closed long after the lockout has expired, with the result that the switch opening will generate the second count, but this should have been the first and only time the count increments. The result is the count increments by two for each switch closing and opening sequence.

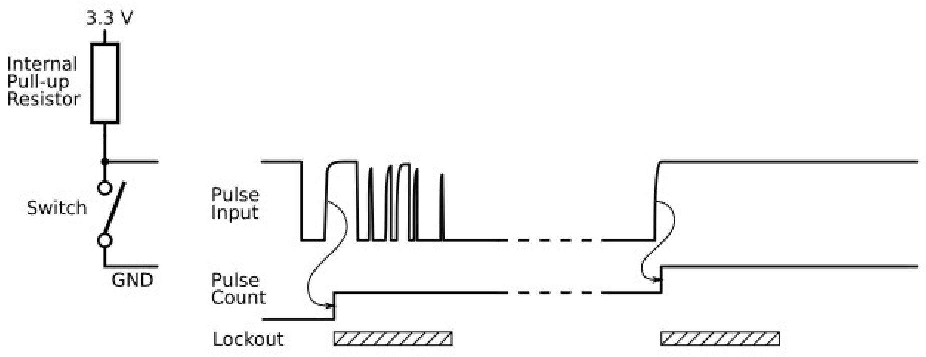
The fourth diagram shows how the new algorithm reacts. It responds to *any* change, and waits while the switch contacts settle, then it compares the input state with the state it remembers from last time, and increments the count only when there is a change and it is in the desired direction. It will not count a pulse that is shorter than the waiting period – the minimum pulse length.



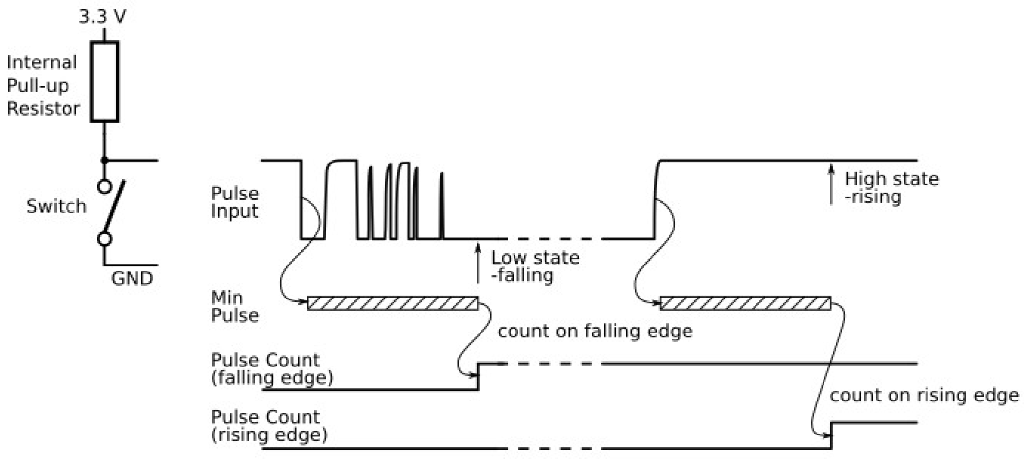
1. Optical Pulse Sensor - count on rising edge



2. S0 pulse output - count on rising edge



3. Magnetic Reed Switch - count on rising edge



4. Magnetic Reed Switch - new algorithm

Change to the API

The API function now takes one, two or three parameters:

void EmonLibCM_setPulseMinPeriod([byte channel,] int _period, byte _edge=FALLING);

The most important change is ‘period’ is now exactly what the function name implies, it is the absolute minimum duration of the pulse. In physical terms, it is the delay between the first change and the instant when the ‘settled’ state of the input is read. If the pulse width is less than this, as will be the case if the pulse comes from an energy meter and is 100 ms long, and the sketch has set the pulse minimum period to the old standard value of 110 ms, then ***pulses will not be counted***. The period should be set to less than the pulse width but longer than the duration of any contact bounce. A value of 20 – 30 (milliseconds) should be suitable in all but the most extreme cases. A value of zero may be used (but should not be necessary) if a ‘perfect’ electronic switch is used to generate the pulse.

The parameter ‘_edge’ has been added to define the edge on which the count is incremented. It is unlikely to be necessary to specify this, as it makes no practical difference on which edge of the pulse the count increments. The value can be either ‘FALLING’ (the new default) or ‘RISING’. These symbolic constants should be cast to the type ‘byte’ to avoid an ambiguity that the compiler is unable to resolve.

Two channels for handling pulse inputs are available to cater for the new non-radio version of the emonPi shield that can have two pulse inputs, and ‘channel’ must also be cast to a byte if it is not explicitly defined.