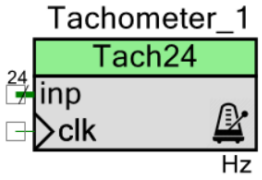
Tach24: 24-channel tachometer
0.0

Features



- Up to 24 input channels
- Doesn't consume UDB Datapath resources
- Optional output median filter
- Optional timeout detection
- Optional stall detection

General description

The Tach24[(*)] component implements a tachometer, which allows parallel monitoring of up to 24 digital signals. It was specifically designed to consume very little hardware resources per channel, leaving PSoC5 UDB Datapath available for other tasks. Component uses reciprocal counters technique, combining both hardware and firmware to obtain results, and provides uniform accuracy across entire operation frequency range. The component is not a substitute for the standard Creator FanContoller component [1], which can monitor up to 16 input channels sequentially.

When to use Tachometer component

Component was developed for monitoring array of DC fans equipped with digital tachometer output. It can be used whenever multiple digital frequency sources need to be sampled in parallel, such as arrays of fans in the rack, flow meters, motors or wheels. Component is compatible with PSoC5 and was tested using CY8CKIT-059 PSoC5LP Prototyping Kit. Demo projects are provided. Component is not compatible with PSoC4.

[*] Hereafter referred to as "Tachometer"

Input-output connections

inp[N:0] – input

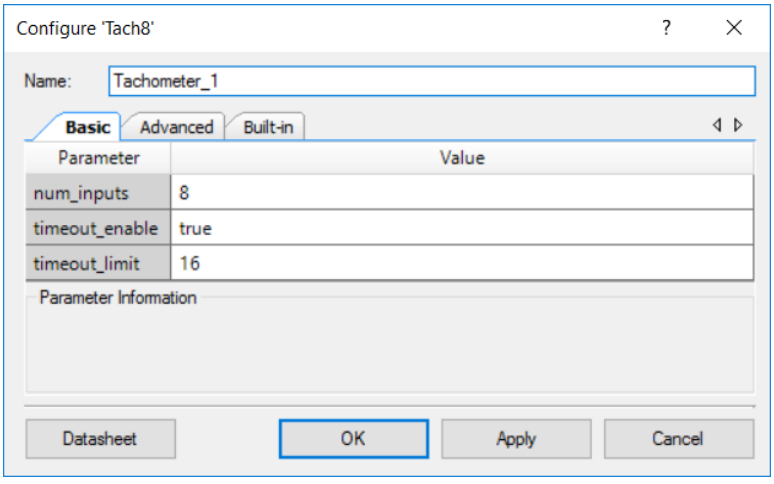
Digital input bus of up to 24-bit width. The capture of the tachometer counters is triggered on the rising edge on this line. The number of inputs is set by the **num_inputs** parameter. The input signal should be synchronized to the BUS_CLK to avoid “Asynchronous path exist…” compiler warning. When displayed, each input line must be connected to valid digital source.

clk – capture clock input

Capture clock input. The rising edge of this line resets the capture of the input line. The pin must be connected to a clock signal or valid digital source. The clock doesn’t need to be 50% duty cycle. Tachometer output doesn’t depend on the accuracy of the clock, as this parameter is not involved in the calculations. However, when detecting Timeout and Stall conditions the input frequency is compared against that clock, which may affect the performance. See **Implementation** section for details.

Parameters and Settings

Basic dialog provides following parameters[*]:



num_inputs (unt8)

Number of tachometer channels. Valid range: [1 to 24]. Default value is 8.

[*] Component was intentionally compiled using Creator 4.0 for compatibility with older versions

timeout_enable (bool)

Enables timeout interrupt. When enabled, the component will respond on every clock, returning either a valid capture result or a timeout status. The timeout mode is preferred when multiple channels must be read simultaneously and in timely manner. It is also protected against DWT timer overrun. When timeout is disabled, the data will be available only when valid capture occurs. The DWT timer is not protected from overrun in this mode, which may lead to erroneous readings if the input frequency falls below BUS_CLK/2^{32}. See **Functional description** section for details.

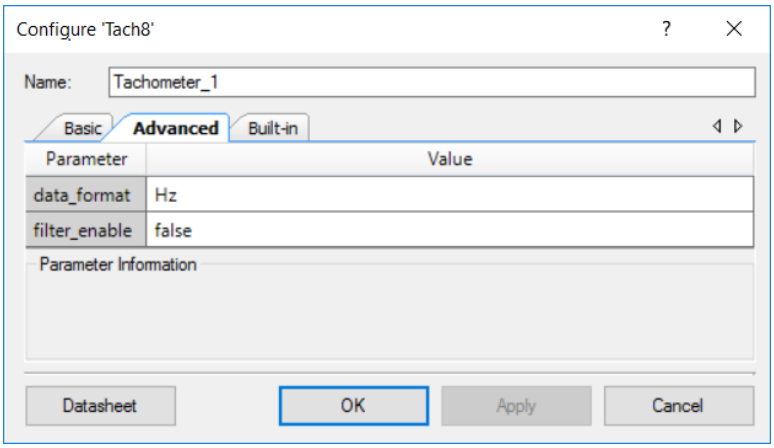
Parameter setting	Description
timeout_enabled = *true*	The **DataReady** flag raised on every clock. The returned **Timeout** flag is *false* if capture occurred, and *true* – if timeout detected (no capture).
timeout_enabled = *false*	The **DataReady** flag raised at the lowest out of the input and clock rates. The **Timeout** flag is not active in this mode.

timeout_limit (unt16)

Number of consecutive timeouts in a row before the stall alert. This parameter has effect only when timeout option is enabled. Allowed range is 2 to 255. This parameter isn't automatically checked by the validator; user should make sure that the value doesn't exceed ($2^{32} \times F_{CLK}/BUS\_CLK$) to avoid DWT timer overrun. The table below shows maximum value of **timeout_limit** for typical settings. See **Implementation** section for details.

capture frequency, Hz	BUS_CLK, MHz	2^{32} x F_{CLK} / BUS_CLK
1	80	53
1	48	89
2	48	178
5	48	443
10	48	890
20	48	1789
50	24	8947

Advanced dialog provides following parameters:



data_format [Hz / mHz / RPM]

Selects return data format. Valid options are Hz, mHz (millihertz) or RPM (rounds per minute). When the Hz option is selected, frequency calculations are performed using float precision. When mHz or RPM options are selected, calculations are performed with uint32 precision. The integer calculations are faster and sufficient in most practical cases. It is recommended when number of inputs is large. Calculations with float precision are more time-consuming and recommended when processor load is light. See **Performance** sections for details.

filter_enable (bool)

Enables 3-point median filter on each of the Tachometer channels. This helps to eliminate occasional spurious readouts, improving accuracy of the measurements when input frequency is steady. When input frequency has considerable noise, such as DC fan tachometer output, the filter may have little or no effect. When enabled, the output is delayed by one reading. On startup, the filter content is initialized to zero, thus making the first output value is 0 by default. To discard that data point, the code can check the FilterPrimed flag. See **Application Programming Interface** section for details.

Application Programming Interface

The Tachometer component designed as a wrapper to a set of the tachometer cores, one for each channel, named Chan0, Chan1, etc. Tachometer variables can be accessed using common API and through each channel directly.

Tachometer common API

Functions and Variables	Description
`void Tachometer_Start()`	Initializes and starts component
`void Tachometer_Stop()`	Stops component
`uint32 Tachometer_MASK`	Channels bit mask

void Tachometer_Start(void)

Description: Initializes and starts all channels. Enables DMA channels and starts DWT timer.
Parameters: none
Return Value: none

void Tachometer_Stop(void)

Description: Stops all channels. Disables all DMA channels. Doesn't stop DWT timer.
Parameters: none
Return Value: none

uint32 Tachometer_MASK

Description: Bit mask, where each bit representing a channel. Bit value 1 indicates that the channel is enabled. It can be used for quick check if all channels have reported their status by comparing the common DataReady flag and MASK. In case of a single channel, the MASK equals 1u; when Tachometer is configured for 24 channels, the MASK equals 0xFFFFFFu.

Access: The mask is a design-time constant and can't be changed at runtime.

Tachometer single channel configuration

When Tachometer is configured for a single channel, its variables can be accessed directly:

Functions and Variables	Description
`uint8 Tachometer_DataReady`	Data ready flag
`uint8 Tachometer_Timeout`	Timeout flag
`uint16 Tachometer_TimeoutCnt`	Timeout counter
`uint8 Tachometer_Stall`	Stall flag
`data_t(*) Tachometer_Frequency`	Measured frequency
`uint8 Tachometer_FiterPrimed`	Filter primed status

uint8 Tachometer_DataReady

Description: Data ready semaphore flag, indicating that either valid data capture or a timeout has occurred. The flag shall set as soon valid data capture occurs. If no data has been captured during capture period, the flag shall set at the end, signaling missing data. The flag is latching and must be cleared. It will automatically reset upon next capture clock. When timeout option is disabled, the flag shall set only on valid data capture, and shall stop responding if channel input stalled.

Access: read/write

uint8 Tachometer_Timeout

Description: The timeout semaphore flag, indicating that no data capture occurred during entire capture period. When Timeout flag is *true*, the frequency reading is 0 and must be discarded. The flag is latching and must be cleared. It will automatically reset upon next capture clock. The Timeout flag is active only when timeout option is selected in component properties dialog.

Access: read/write

* data_t – data type: float, when selected **data_format** is Hz; uint32, when **data_format** is mHz or RPM.

uint16 Tachometer_TimeoutCnt

Description: The timeout counter, indicating how many consecutive timeouts occurred since the last valid capture event. This value will saturate at 0xFFFF. The timeout counter is acive only when timeout option is selected in component properties dialog.

Access: read only

uint8 Tachometer_Stall

Description: Tachometer Stall semaphore flag, indicating that the timeout counter exceeded the **timeout_limit**. The flag is latching and must be cleared. It will automatically reset upon next rising edge of the capture clock. The flag is acive only when the timeout option is selected in component properties dialog.

Access: read/write

data_t Tachometer_Frequency

Description: Measured input frequency. It encapsulates a call to the procedure GetFrequency(), which calculates frequency from the counters values. If the Timeout or Stall flag is *true*, the returned frequency value is zero. The user must check if return result is valid by checking the Timeout flag. The calculation precision depends on selected output data format.

Access: read only

Return value: When **data_format** is Hz, calculations are performed using float arithmetic and return data type is float. When **data_format** is mHz or RPM, calculations are performed using integer arithmetic and return data type is uint32.

uint8 Tachometer_FilterPrimed

Description: Median filter semaphore flag, indicating that the filter has been primed, and the output value represents valid result. The flag shall set only after channel frequency has been calculated using the GetFrequency() procedure. The flag

is active only when the filter option is enabled, otherwise returning default value 0 (*false*). The median filter has delay of one sample. On startup, the filter content is automatically reset to 0, thus making by default the first output value is 0. Upon the second reading of the frequency, the flag shall set 1 (*true*), indicating that output is valid. Thus, the flag can be used to discard that first zero value, if necessary.

Access: read only

Tachometer multichannel configuration

Common functions and variables

When component is configured for multiple channels, these common flags return bitwise status for all channels, one bit per channel. They are useful when all channels should be processed at once, for example sending results by UART as a packet. They should be used only with timeout option enabled.

Variable	Description
`uint32 Tachometer_DataReady`	Combined data ready status flag
`uint32 Tachometer_Timeout`	Combined timeout status flag
`uint32 Tachometer_Stall`	Combined stall status flag
Function	**Description**
`void Tachometer_GetAllFreq()`	Get frequency for all channels

uint32 Tachometer_DataReady

Description: Combined DataReady semaphore flag, where each bit representing a channel. Bit value 1 indicates that channel status is available. This shall happen once channel captures the data, or at the end of the capture period if no capture occurred. The flag is latching and must be cleared. It will automatically reset upon next capture clock. This flag should be used only when timeouts are enabled. When used with timeout option disabled, Tachometer may stop responding if any channel stalls.

Access: read/write

uint32 Tachometer_Timeout

Description: Combined Timeout semaphore flag, where each bit representing a channel. Bit value 1 indicates that channel timeout occurred. It can be used for quick check if timeout occurred on any of the channels. A non-zero value indicates that one or more channels have missed data during capture period. The flag is latching and must be cleared. It will automatically reset upon next capture clock. This flag is active only when timeout option is selected in component properties dialog.

Access: read/write

uint32 Tachometer_Stall

Description: Combined Stall semaphore flag, where each bit representing a channel. Bit value 1 indicates that channel input has stalled. It can be used for quick check if stall occurred on any of the channels. A non-zero value indicates that one or more channels have stalled. The flag is latching and must be cleared. It will automatically reset upon next capture clock. This flag is active only when timeout option is selected in component properties dialog.

Access: read/write

void Tachometer_GetAllFreq(data_t Freq[])

Description: Calculate input frequencies for all channels at once, and return a results array. The calculation precision depends on selected output data format. This function should be called only after all channels reported data ready status.

Access: Input/output: pointer to an array Freq [**num_inputs**]. The array data type should match the selected accuracy of calculations. When **data_format** is Hz, the data type is float and calculations are performed using floating arithmetic. When **data_format** is mHz or RPM, the data type is uint32, and calculations are performed using integer arithmetic.

Return value: none

Tachometer per-channel variables

In multichannel configuration, the variables of each individual channel can be selectively accessed either by channel index or by direct API calls using channel name, similar to the single channel configuration. Individual channel access may be useful in control applications, providing immediate response without waiting for other channels to report.

Channel access by the index[*]:

Variable	Description
`uint8 Tachometer_DataReady(i)`	Channel data ready flag
`uint8 Tachometer_Timeout(i)`	Channel timeout flag
`uint16 Tachometer_TimeoutCnt(i)`	Channel timeout counter
`uint8 Tachometer_Stall(i)`	Channel stall flag
`data_t Tachometer_Frequency(i)`	Channel measured frequency
`uint8 Tachometer_FilterPrimed(i)`	Channel filter primed status

Direct access by the channel name[†]:

Variable	Description
`uint8 Tachometer_Chan0_DataReady`	Channel data ready flag
`uint8 Tachometer_Chan0_Timeout`	Channel timeout flag
`uint16 Tachometer_Chan0_TimeoutCnt`	Channel timeout counter
`uint8 Tachometer_Chan0_Stall`	Channel stall flag
`data_t Tachometer_Chan0_Frequency`	Channel measured frequency
`uint8 Tachometer_Chan0_FilterPrimed`	Channel filter primed status

[*] Where index i range is from 0 to (**num_inputs**-1)
[†] Where other channels: 1, 2, … can be accessed similarly, by changing to Chan1, Chan2, etc.

Functional description

Introduction

This experimental component was designed with the goal of fitting up to 24 parallel tachometer channels into a single PSoC5 project. PSoC paradigm dictates offloading digital operations to the hardware logic. This way, hardware frequency meter needs at least one 16-bit Counter in capture mode, or a pair of 16-bit Counters in reciprocal mode [2-4]. For PSoC5, which has 24 of 8-bit UDB Datapath blocks, this limits the amount of tachometer channels to 12 and 6 accordingly. To fit 24 channels into single PSoC5 the Tachometer component utilizes rarely used resources: the DMA internal 12-bit counter, and 32-bit Design Wide Timer (DWT), available to Cortex M3 microcontrollers. Total 24 DMA channels are available to PSoC5, which makes up to 24 tachometer channels feasible. There is, however, only single DWT timer, which has to be shared among all tachometer channels. The drawback of this approach is that neither DMA counter, nor the DWT timer feature hardware capture option [6]. They remain free-rolling and have to be polled using software interrupt, an approach susceptible to unpredictable delays caused by concurrent tasks run by a microcontroller. This puts such design in between pure hardware and software approach used by traditional microcontrollers. It, however, may suffice for some applications because tachometers typically have much narrow specifications than general purpose frequency meters[(*)].

This component was designed for testing the performance of the Cortex M3 microcontroller when multiple DMAs and software interrupts have to operate simultaneously and concurrently, creating a stampede. It appears, however, that PSoC5's M3 core is able of handling such load, providing adequate accuracy for all tachometer channels, while leaving UDB Datapath resources available to other tasks.

[*] See Table 1 in **Functional description** section.

What is a Tachometer?

Tachometer is a frequency meter for monitoring rotational speed of mechanical devices: motor shafts, wheels, fans, flow meters, etc. In a result, tachometers have much narrower specifications than general purpose frequency meter:

■ Mechanical devices usually operate at low frequencies. Their typical range of operation is rarely going above the 1000 Hz. For example, BLDC motors may have upper frequency about 120 thousands PRM (2000 Hz), but majority of other AC and DC motors operate under 3-18 thousands RPM (50-300 Hz). Furthermore, the maximum frequency can't increase beyond the mechanical limit of the system, which caps tachometer's upper limit.

■ Mechanical devices usually operate within limited dynamic range. For example, typical DC fan frequency falls in range 20 to 400 Hz (dynamic range 26dB). Operation range for flow meters is approx. 20-200 Hz (20dB).

■ Due to the inertia, mechanical devices don't need to be monitored faster than 1-100 Hz. Tachometer, however, should tolerate conditions when the input frequency falls below the sampling rate (for example, when motor is idling after power turn-off).

■ The intrinsic instability of the mechanical motion makes it unnecessary of having accuracy better than 10-bits of resolution, and in most cases even 8-bits is sufficient.

■ Tachometers usually have some additional features like stall detection or frequency limits crossing, accessible by controller independently from the frequency measurement.

Table 1. Tachometer typical specs

Parameter	Specifications
Maximum input frequency	2000 Hz
Minimum input frequency	1 Hz
Dynamic frequency range (max/min)	2000 (11-bit)
Capture rate range	1 – 100 Hz
Frequency resolution	1000 ppm (10-bit)
Frequency limits	yes
Stall detection	yes

Theory of operation

The Tachometer implementation uses reciprocal counters technique [2-4] using a pair of free-rolling counters (Figure 1). This approach offers uniform accuracy across entire operation frequency range [2].

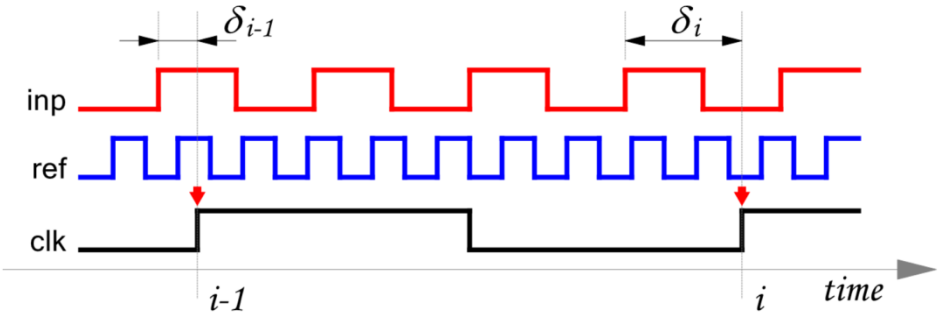


Figure 1. Tachometer timing diagram. Red – input signal to the counter A from the unknown frequency source; blue – reference clock to the counter B; black – capture clock. Sampling of the counters A and B occurs on the rising edges of the capture clock (marked by red arrows). The signals are not synchronized, resulting in uncertainty of at least of 1 count for both counters.

During selected capture period the counter A is counting pulses from the input source, and counter B - from the reference clock. Once a pair of measurements over two consecutive clock periods has been obtained, the input frequency can be calculated using equation:

$$F_A \cong F_B \cdot \frac{cnt_i^A - cnt_{i-1}^A}{cnt_i^B - cnt_{i-1}^B} = F_B \cdot \frac{\Delta cnt_i^A}{\Delta cnt_i^B} \quad (1)$$

where $\boldsymbol{F_A}$ – measured input frequency, $\boldsymbol{F_B}$ - reference frequency; $\boldsymbol{cnt^A}$ and $\boldsymbol{cnt^B}$ - captured values of counter A and counter B respectfully, and indexes $(\boldsymbol{i})$ and $(\boldsymbol{i-1})$ are referring to the consecutive capture periods. Since counters ignore the fractional portion of the periods elapsed, both the nominator $\Delta\boldsymbol{cnt_i^A}$ and denominator $\Delta\boldsymbol{cnt_i^B}$ bear uncertainty of up to 1 clock, resulting in some inaccuracy of the result.

Notice that the capture frequency is not present in the equation. However, it affects accuracy of the measurement indirectly: the longer the interval between the captures, the higher the accuracy. Indeed, as capture period increases, both nominator and denominator of the Equation 1 are growing large, and the fraction value approaching the limit.

Further improvement of the accuracy can be achieved if sampling of the counters is synchronized with the rising edges of the input signal (Figure 2). This way the nominator value $\Delta \boldsymbol{cnt}_i^A$ returns exact number of periods, leaving the only source of error is the uncertainty of one reference clock count in the denominator. The later can be reduced by utilizing a reference clock of higher frequency and longer capture periods.

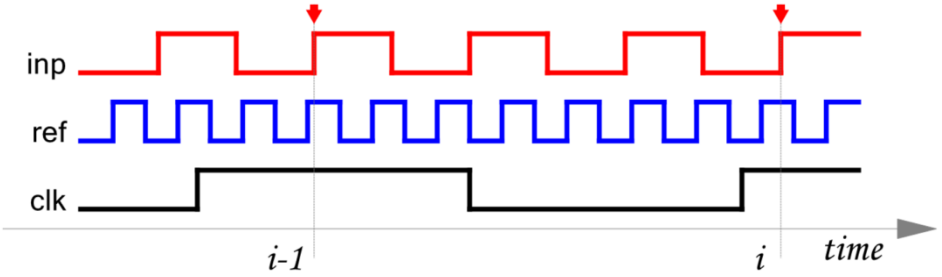


Figure 2. Tachometer timing diagram using input latch. Red – input signal to the counter A from the unknown frequency source; blue – reference clock to the counter B; black – capture clock. The sampling occurs on the rising edges of the counter A instead of the capture clock, resulting in exact round number of periods of the input signal. The uncertainty of the counter B is still at least 1 count.

Implementation

Tachometer implementation is shown on Figure 3. In basic form it utilizes only a single DMA, an interrupt and DFF logic element per channel. The EdgeDetector and the capture interrupt can be shared among all tachometer channels. The DMA counter is used as a counter A, sampling input signal, and Design Wide Timer (DWT), running at system clock, is used as a counter B.

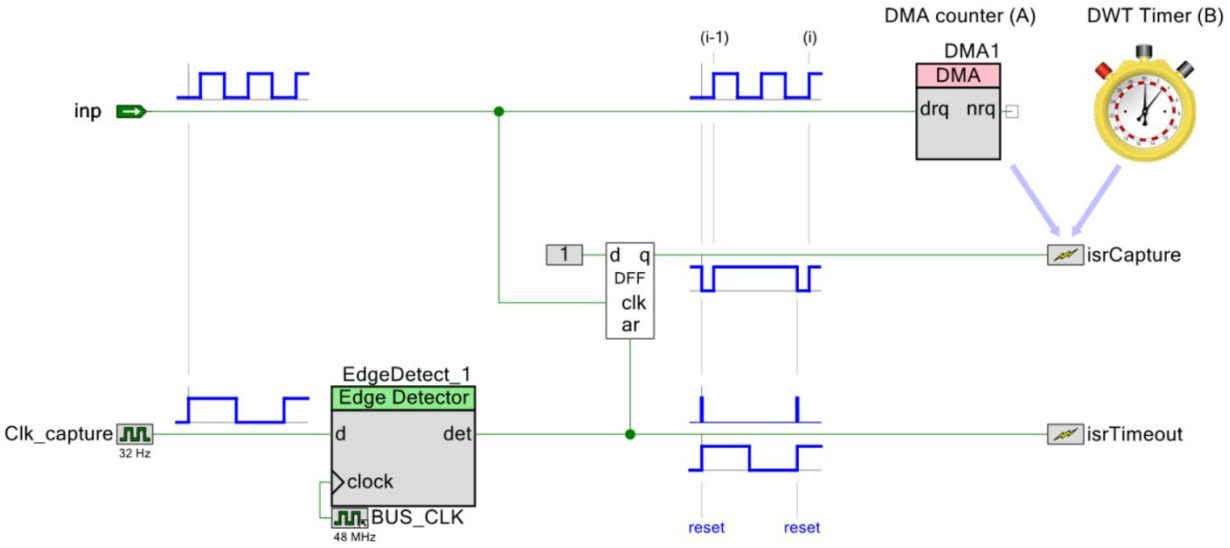


Figure 3. Tachometer schematic (simplified). The DFF, DMA and capture interrupt constitute an individual tachometer channel. The common section, which includes capture clock, edge detector, timeout interrupt and DWT timer can service multiple input channels simultaneously.

The DMA is configured to continuously sample input signal on the rising edge. DMA has internal approx. 12-bit down counter, which decrements DMA TransferCount register from 0xFFFF down to 0x0001, bypassing the 0 value. The DWT is a free-rolling 32-bit up-counting timer operating at the BUS_CLK rate.

The logic of the Tachometer operates as following. The rising edge of the capture clock resets the DFF, waiting for the first rising edge of the input to set it high, triggering the interrupt. The interrupt handler routine reads and backs-up the DMA and DWT counters, providing all the data necessary to calculate frequency using Equation 1. To reduce the amount of time spent in the interrupt, the Frequency value is not calculated immediately, but reserved for the later time, when the result is requested in the main loop. See code example in **Appendices 1-5**.

The state machine

Tachometer's state machine has only of two states: SET and RESET (Figure 4). On the rising edge of the capture clock it falls into the RESET state, raising the timeout interrupt. It shall remain in that state until the input pulse arrives, raising a timeout interrupt on each capture clock. The first rising edge of the input pulse puts the machine into the SET state, raising the cature interrupt, in which processor can capture DMA and DWT counters. Any consecutive input pulses are ignored, and the state remains until the next capture clock. Then cycle repeats. The state machine has to be primed by at least two consecutive capture events before it can return valid results.

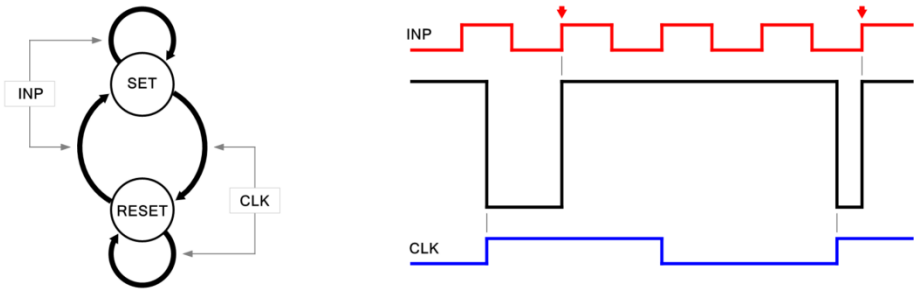


Figure 4. The state machine diagram and timing profile. Red – input signal, Blue – capture clock, Black – machine state (Hi=SET, Lo=RESET). The capture events are marked by red arrows.

Implementation constrains

For stable operation of the Tachometer its counters should not overflow. For DMA counter not to overflow over the capture period, the input frequency should remain below

$$F_{max} = F_{CLK} \times 4095 \quad (2)$$

The minimum frequency, which can be measured without occurrence of the timeout is $\boldsymbol{F_{CLK}}$. Once the input frequency falls below the $\boldsymbol{F_{CLK}}$, the timeout shall be reported every time the capture is missing. The Tachometer shall continue to correctly measure the input frequency, but the results will be available not on every capture clock, but at the input frequency.

The lowest frequency that can be measured by Tachometer with timeout option enabled, is defined by the **timeout_limit** parameter:

$$F_{lim} = F_{CLK}/timeout\_limit \quad (3)$$

The DWT is a 32-bit timer with overflow period (at BUS_CLK = 48 MHz):

$$T_{DWT} = 2^{32}/BUS\_CLK = 2^{32}/48\ MHz \approx 89\ sec \quad (4)$$

That sets the limit for the lowest input frequency, which can be potentially measured by the Tachometer

$$F_{min} = (1/T_{DWT}) = BUS\_CLK/2^{32} = 48\ MHz/2^{32} \approx 0.011\ Hz \quad (5)$$

Tachometer frequency constrains calculated for typical operational parameters BUS_CLK 48 MHz, F_{CLK} 20 Hz, and timeout limit 10 are summarized on Figure 5. These parameters are typically dictated by the external hardware specs and performance requirements.

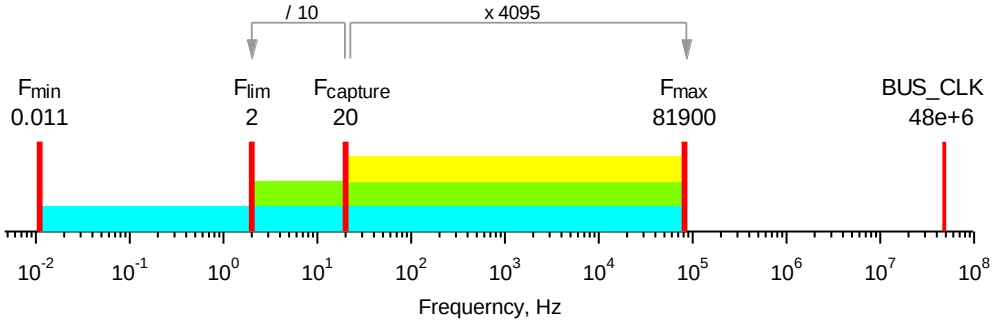


Figure 5. Tachometer input frequency ranges calculated with parameters: BUS_CLK 48 MHz, capture frequency 20 Hz, timeout limit 10. Yellow – timeout-free input range; Green – input range with timeout enabled; Blue – maximum achievable input range.

Performance

Tachometer does not feature hardware capture of the counters, and thus prone to the delays caused by concurrent tasks run by processor. Assuming that DMA counter has been captured exactly using latching logic, the only error of the measurement may come from the timing delay of the DWT capture. Since the DWT increment equals number of the BUS_CLK cycles during the capture period $\Delta cnt_i^B \approx F_{CLK}/BUS\_CLK$, the error can be estimated from Equation 1:

$$\frac{\delta F_A}{F_A} = \frac{\delta}{\Delta cnt_i^B} \approx \delta \times \frac{F_{CLK}}{BUS\_CLK} \quad (6)$$

where δ- DWT capture error, measured in number of system bus clocks. In ideal conditions $\delta \leq 1$, thus best possible accuracy is

$$\delta F_A/F_A \leq F_{CLK}/BUS\_CLK = 32Hz/48MHz \cong 0.7\ ppm \quad (7)$$

However, in real conditions, controller may perform multiple concurrent tasks, causing random jitter of DWT capture moments, inducing some errors. For example, an error of δ =±10 clocks shall increase relative error tenfold: $\delta F_A/F_A \approx \pm 7ppm$.

For example, Figure 6 shows Tachometer measurement of the 1 kHz clock with compiler configured in either debug or release modes. In release mode, the frequency noise was negligibly small and could not be measured. When configured for debugging mode, the frequency noise was approx. ±7.10^{-3} Hz, which corresponds to frequency resolution of $\delta F_A/F_A \approx \pm 7ppm$, as discussed above. Such accuracy (16-bit) is overkill for practical applications, where 10-bit is mostly sufficient.

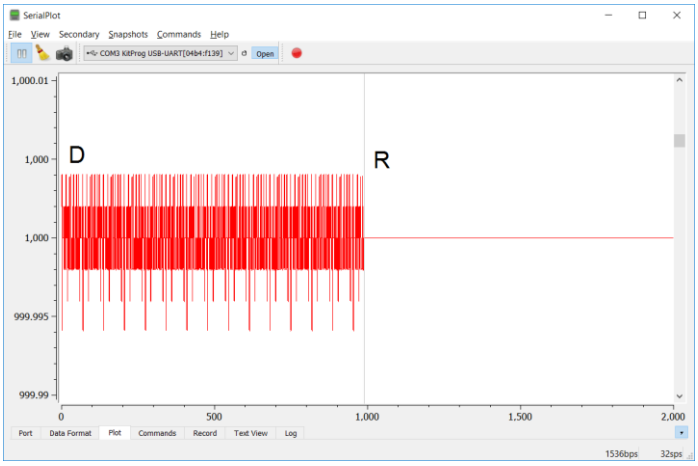


Figure 6. Effect of the debug (D) *vs* release (R) compiler mode on the output accuracy. Tachometer is configured for single channel, float output precision. Input frequency – 1 kHz, BUS_CLK 48 MHz, capture clock frequency 32 Hz.

Tachometer resolution in debugging mode (ppm) as a function of the capture frequency is shown on Figure 7. The observed error is proportional to the capture frequency in agreement with the Equation 6. The slope of the linear fit (0.35 ppm/Hz) corresponds to persistent error of 2δ=17 bus clocks. No errors were detected in the release mode.

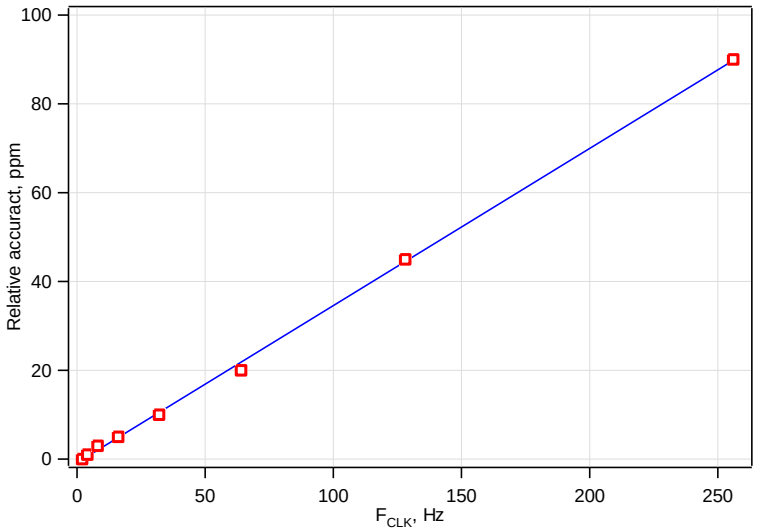


Figure 7. Tachometer resolution (ppm) *vs.* capture clock rate. Red squares – test data, blue line – linear fit. Tachometer is configured for a single channel, float precision, timeout enabled. Input frequency 1 kHz, BUS_CLK 48 MHz Compiler mode is debugging.

Tachometer performance was tested in various conditions using 8, 16 and 24 parallel inputs. Performance results using 8 parallel inputs are shown on Figure 8. Each input was provided with individual source of frequency with slight offset from each other: $F_k = 200Hz - 0.04Hz \times k$, where k- is the channel number. Tachometer raw readings show some spurious artifacts of less than ±0.04 Hz amplitude, corresponding to frequency resolution of 200 ppm (12.3-bits). Enabling the median filter rejects spurious noise almost entirely.

Tachometer performance was about the same when number of inputs increased to 16 and 24. Result of sampling 16 inputs is shown on Figure 9. Due to the charting software limitations [7], only 8 out of total 16 channels measured could be displayed at a time. Chart shows data for channels 0, 2, 4, 6, 8, 10, 12 and 14. The output resolution of the raw data was about 200 ppm (12.3-bit), which is sufficient for most practical applications. When enabled, the median filter was able to clean up spurious artifacts almost entirely.

Tachometer performance sampling 24 parallel inputs is shown in **Appendix 4**.

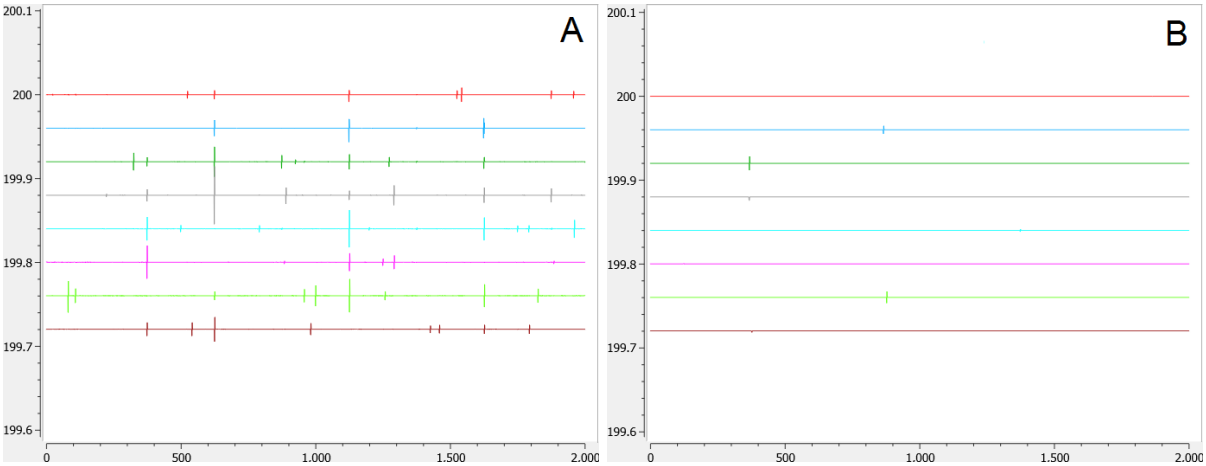


Figure 8. Tachometer performance sampling 8 channels: A – raw output, B - with filter enabled. Input signal frequency for the first channel is 200 Hz, other channels are offset by 0.04 Hz (200 ppm) from each other. The noise figure doesn't exceed 0.04 Hz (200 ppm) Tachometer is configured for 8 channels, float precision, timeout enabled. Capture frequency 20 Hz, BUS_CLK 48 MHz. Compiler is set for release mode.

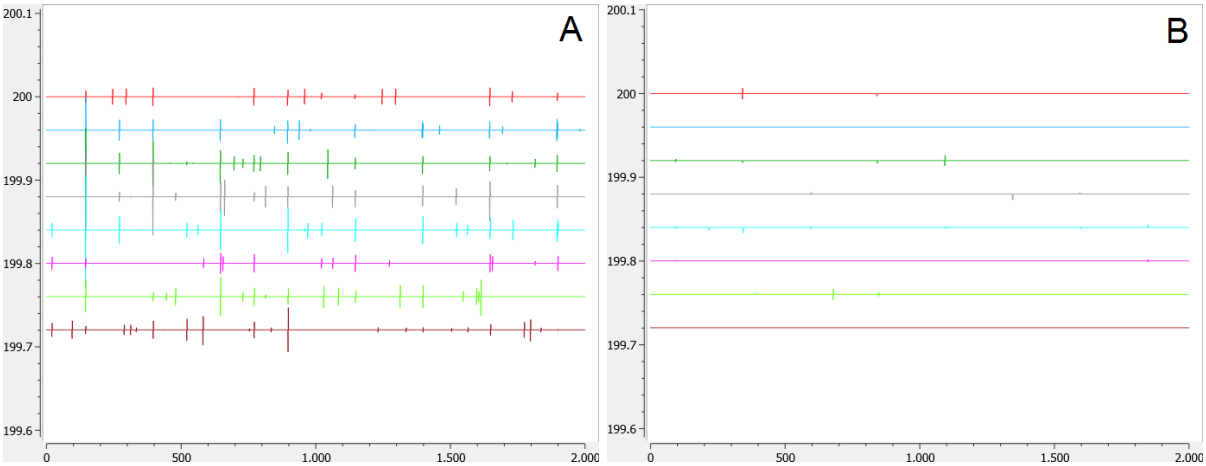


Figure 9. Tachometer performance sampling 16 channels; A – raw output, B - with filter enabled. Only 8 channels (#0, #2, #4, #6, #8, #10, #12 and #14) out of total 16 are displayed. Input frequency for the first channel is 200 Hz; the noise doesn't exceed 0.04 Hz (200 ppm). Tachometer is configured for 16 channels, float precision, timeout enabled. Capture frequency 20 Hz, BUS_CLK 48 MHz. Compiler is set for release mode.

Timeout and Stall

Tachometer can automatically detect when signal frequency is below the clock rate without performing actual calculations of the frequency, raising a Timeout flag. It can also detect a stall condition, when frequency becomes so low, that no data could be captured during a number of timeouts in a row. These features require the timeout option enabled in the Properties Dialog. Timing diagrams for input frequency above and below the clock rate are shown on Figure 10.

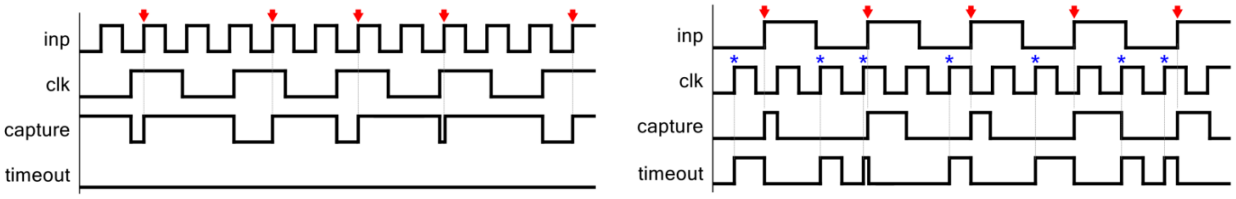


Figure 10. Tachometer timing diagrams. Left: $F_{INP} > F_{CLK}$; Right: $F_{INP} < F_{CLK}$. Arrows mark the capture events, stars – timeout events.

When input frequency is above the clock, the data are being captured on every clock period without timeouts occurring. When input frequency is below the clock rate, some periods may miss the capture, ending up with the timeout interrupt. In this case, the data is captured at the input rate (below the clock), while combined rate of capture and timeouts remains the same and equals clock rate. By testing the Timeout status flag one can detect when input frequency falls below the sampling clock without performing frequency calculations. Tachometer capture rate dependency on the input frequency is shown on Figure 11. The capture rate grows linearly with the input frequency, capped by the clock. When timeout is enabled, no capture occurs below the stall limit ($F_{CLK}/timeout\_limit$).

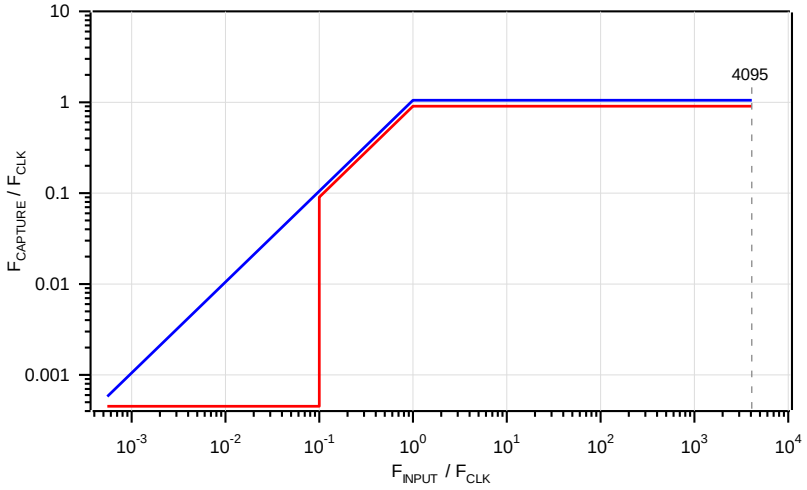


Figure 11. Tachometer data capture rate *vs*. input frequency. Red – timeout enabled, Blue – timeout disabled. Capture clock frequency 20 Hz, timeout limit 10, BUS_CLK 48 MHz.

Tachometer operation with input frequency slowly dropping below the capture frequency is shown on Figure 12. Tachometer was configured for single channel with timeout enabled, and timeout limit set to 10. The capture clock frequency was 20 Hz. Tachometer input was provided with a signal, whose frequency was ramped down from 25 Hz to 2 Hz in 5 sec, crossing the capture clock. Tachometer returned data at the capture clock rate, including captured Frequency value, Timeout and Stall status. When input frequency is above F_{CLK}, the Timeout status is always false, and valid Frequency value is returned on every capture clock. When input frequency falls below the F_{CLK}, the results returned are valid only when Timeout status is *false.* If Timeout status is *true*, no capture has occurred during the capture period, and returned Frequency value is invalid. In such case, no Frequency result could be obtained, and return value is zero.

As the input frequency gets lower, the timeout events become repetitive, with number of timeouts in a row increasing as $N \cong F_{CLK}/F_{inp}$. When the number of timeouts in a row exceeds the **timeout_limit**, the Stall flag shall raise, signaling potential lockdown of the motor.

The chart reveals some deviation between the input and the measured values at lower range, which is due to the fact that the input frequency is rapidly changing, while the returned value represents the average frequency between the capture events.

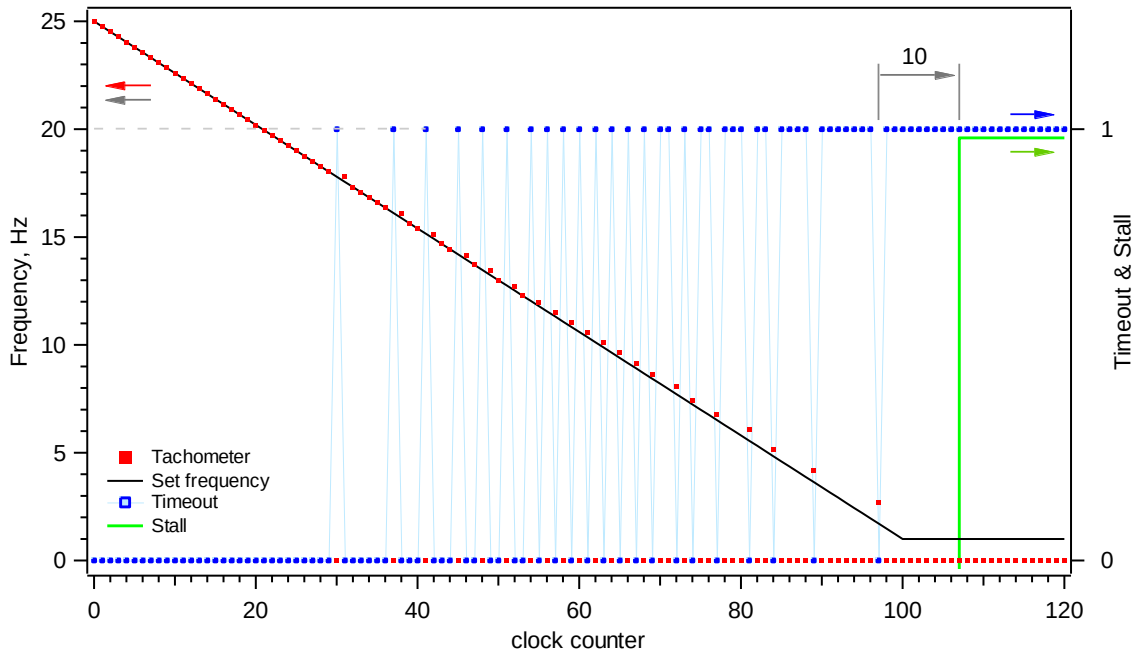


Figure 12. Tachometer output Frequency, Timeout and Stall status flags. Black line – input frequency; Red – output result; Blue – Timeout flag (1=true, 0=false); Green – Stall flag. Input frequency ramps down from 25 Hz to 2 Hz in 5 sec; capture frequency 20 Hz. Tachometer timeout is enabled, timeout limit 10. When Timeout flag=1, the Frequency result is invalid.

Timeout disabled mode

Tachometer can operate with timeout option disabled. The Timeout and Stall status flags are not active in this mode, and data are returned not at deterministic times but only when valid capture occurred. When input frequency is above the capture clock rate, tachometer operation is same as in the timeout enabled mode, and frequency result is available on each capture clock. When input frequency falls below the capture clock, the results are available only when valid capture occurs - at input rate, which is below the clock. No status information is available when the data is missing (Figure 13).

Use of per-channel API is recommended in this mode of operation, as with common API a stall of a single channel shall stop reporting of all other channels also.

The lowest input frequency that can be potentially measured by the Tachometer in this mode is defined only by the system clock. For example, for BUS_CLK of 48 MHz

$$F_{min} = BUS\_CLK/2^{32} = 48\ MHz/2^{32} \approx 0.011\ Hz \quad (8)$$

The drawback of this mode is that it has no control over DWT timer overrun. Once the input frequency falls below the F_{min}, the DWT timer overruns, and resulting output frequency becomes unpredictable. Therefore, this mode is not recommended for critical applications.

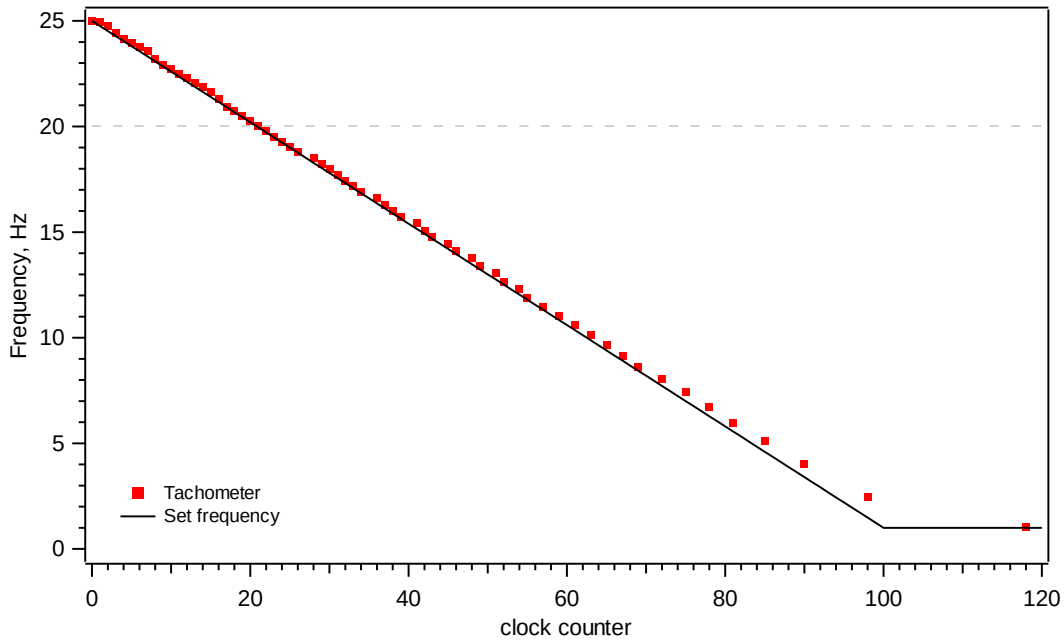


Figure 13. Tachometer operation with timeout option disabled. Black line – input frequency; Red squares – tachometer output. Tachometer is configured for single channel, timeout option disabled. Input frequency ramps down from 25 Hz to 2 Hz in 5 sec; capture frequency 20 Hz.

Resources

Component resources consumption with timeout interrupt enabled is shown below. Multiple instances of the component can fit into design, limited by the amount of available DMA channels (24) and interrupts (32). Component doesn't use UDB Datapath.

channels	DMA	Interrupts	macrocells	P-terms	status cell	DWT	BUS_CLK
1	1	2	3	2	1	1	108.7 MHz
2	2	3	4	2	1	1	108.4 MHz
8	8	9	10	2	1	1	90.9 MHz
16	16	17	18	2	1	1	81.9 MHz
24	24	25	26	2	1	1	63.5 MHz

Performance

Component interrupts are lightweight with no calculations inside. Actual calculation of the frequency is performed in the main loop using GetFrequency() procedure, which has overload versions for each output format. Component performance was obtained in release mode with compiler optimization set for speed, link time optimization enabled.

Table 2. Typical average processor clocks used by the GetFrequency() procedure

Output data format	Filter disabled	Filter enabled
Hz	350	630
mHz	190	215
RPM	200	260

Sample Firmware Source Code

Examples of Tachometer code are provided in **Appendices 1-5**. Demo projects are provided.

Component Changes

Version	Description of changes	Reason for changes/impact
0.0	Version 0.0 is the first beta release of the component	

References

1. Cypress AN66627. PSoC® 3 and PSoC 5LP Intelligent Fan Controller
2. Cypress AN2283, PsoC1 Measuring Frequency.
3. How a Reciprocal Counter works, https://youtu.be/V8HGeaGBN2A
4. Hendrik Lipka, PSoC5LP Frequency Counter, https://hendriklipka.de/hardware/freqcounter.html, https://youtu.be/4oK3syZszUU
5. Leonard Poma, Multi-Input Frequency Measurement Tutorial,
6. Leonard Poma, PSoC5LP DMA from DWT_CYCLE_COUNT (0xE0001004) to RAM
7. SerialPlot v0.0, SerialPlot: interface to real-time data charts

Appendix 1

Basic demo

Project schematic of basic Tachometer demo is shown on Figure 14[(*)].

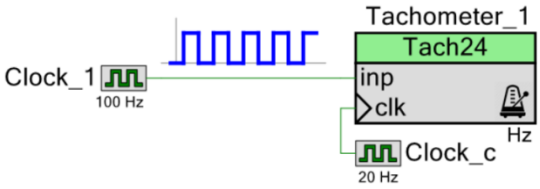


Figure 14. Project schematic. Tachometer is configured for Hz output format with timeout enabled.

Tachometer is configured for a single channel, float precision output in Hz format. The input is driven by a test clock of 100 Hz; the capture clock is set to 20 Hz. Tachometer results are transferred to the external Terminal using UART on each DataReady event. Tachometer's single channel status flag can be accessed directly:

```
Int main()
{
    Tachometer_1_Start();
    char buf[255];

    for(;;)
    {
        if (Tachometer_1_DataReady)        // capture or timeout
        {
            Tachometer_1_DataReady=0;      // reset flag

            sprintf(buf, "%.3g\t%d\t%lu\r\n",  Tachometer_1_Frequency, \
                         Tachometer_1_Timeout, Tachometer_1_TimeoutCnt);
            UART_1_PutString(buf);
        }
    }
}
```

When input clock frequency (100 Hz) is set above the capture rate, the data is captured on each clock; Timeout flag and Timeout counter are 0, as no timeouts occurred (Figure 15). When input clock frequency (4.5 Hz) is set below the capture rate, the data can't be captured on each clock, causing timeouts. When timeout occurs, the value returned is 0, Timeout flag is 1 (*true*) and Timeout counter returns number of timeouts since the last data capture (Figure 16). The average rate of the frequency capture in this case equals the input frequency.

[*] Associated project: Tach24_x1_basic_01a.cyproj

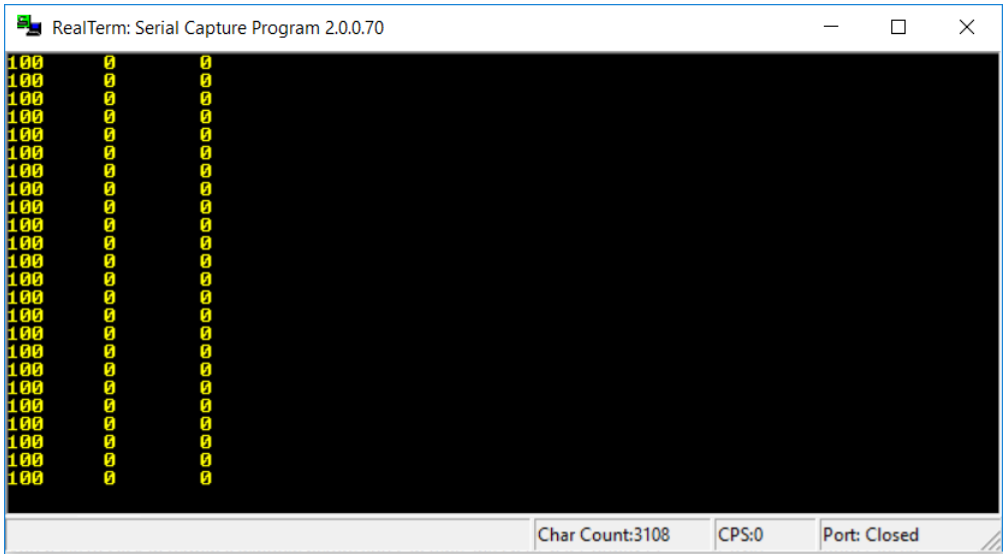


Figure 15. Terminal output when input frequency is above F_{CLK}. Columns 1, 2 and 3 show Frequency, Timeout status and Timeout counter. Input frequency 100 Hz, capture clock frequency 20 Hz.

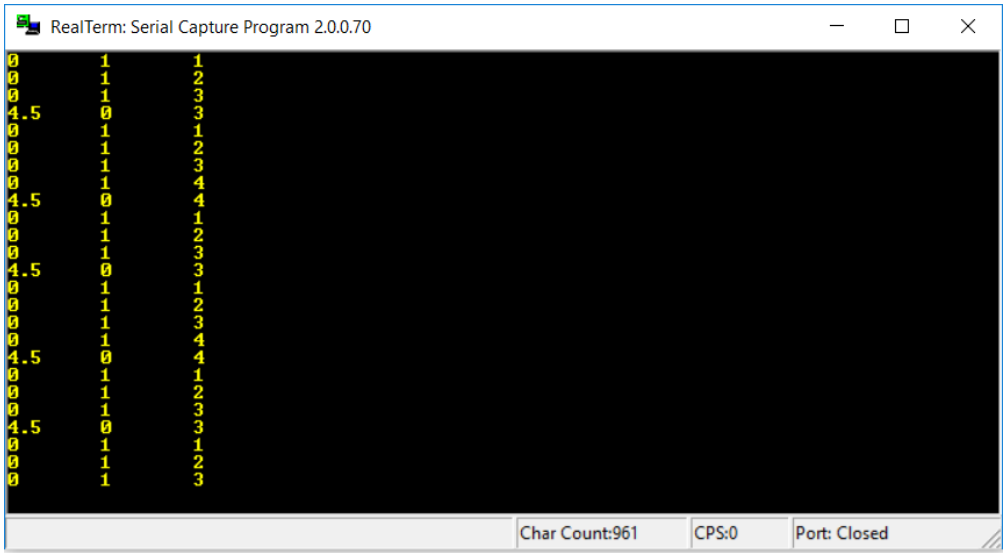


Figure 16. Terminal output when input frequency is below F_{CLK}. Columns 1, 2 and 3 show Frequency, Timeout status and Timeout counter. Input frequency 4.5 Hz, capture clock frequency 20 Hz.

Appendix 2

Independent reading of channels

Example of independent reading of Tachometer channels is shown on Figure 21[(*)].

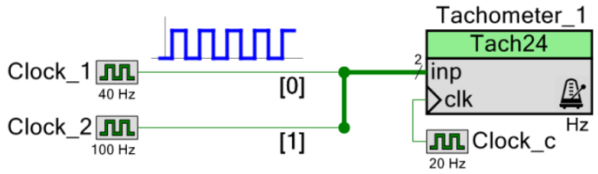


Figure 17. Tachometer example of sampling of 2 inputs. Tachometer is configured for Hz output with timeout enabled.

In this demo, tachometer channels are read independently from each other. This mode of operation can be useful for control applications, where immediate response on each channel is required. Tachometer is configured for 2 channels, float precision output in Hz format, with timeout enabled. The inputs are connected to the test clocks of 40 Hz and 100 Hz. The capture frequency is set to 20 Hz. Tachometer channels are read in the main loop on DataReady flag independently for each channel, and results are printed on the external Terminal using UART (Figure 22).

```
int main()
{
    Tachometer_1_Start();
    char buf[255];

    for(;;)
    {
        // read channel 0->
        if (Tachometer_1_Chan0_DataReady)  // capture or timeout
        {
           Tachometer_1_Chan0_DataReady=0; // clear flag

           sprintf(buf, "ch0: %.3g; ", Tachometer_1_Chan0_Frequency);
           UART_1_PutString(buf);
         }

        // read channel 1->
        if (Tachometer_1_Chan1_DataReady)  // capture or timeout
        {
           Tachometer_1_Chan1_DataReady=0; // clear flag

           sprintf(buf, "ch1: %.3g\r\n", Tachometer_1_Chan1_Frequency);
           UART_1_PutString(buf);
         }
     }
}
```

[*] Associated project: Tach24_x2_basic_01a.cyproj

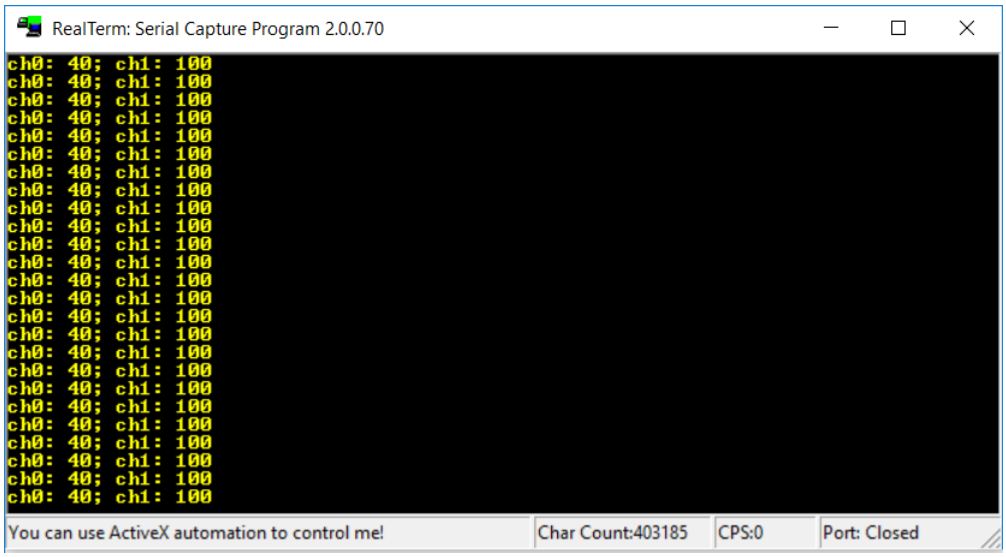


Figure 18. Terminal output of 2 independently sampled inputs.

Appendix 3

Sampling of 8 inputs

Basic example of Tachometer sampling 8 parallel inputs is shown on Figure 19[(*)].

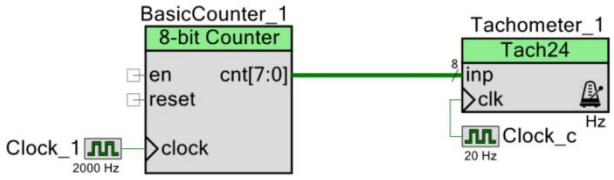


Figure 19. Tachometer example of sampling of 8 inputs. Tachometer is configured for Hz output format with timeout enabled.

Tachometer is configured for 8 channels, float precision output in Hz format, with timeout enabled. The input bus is connected to the test source, 8-bit BasicCounter, driven by 2 kHz clock, providing eight digital signals with frequencies: 1000, 500, 250, 125, 62.5, 31.25, 15.625 and 7.8125 Hz. The capture clock is set to 20 Hz. Tachometer results are transferred to the external Terminal using UART on each DataReady event (Figure 18).

```
int main()
{
    Tachometer_1_Start();
    char buf[255];
    float F0, F1, F2, F3, F4, F5, F6, F7;

    for(;;)
    {
        if (Tachometer_1_DataReady == Tachometer_1_MASK)
        {
            Tachometer_1_DataReady = 0; // reset flag

            // force calculation->
            F0 = Tachometer_1_Frequency(0);
            F1 = Tachometer_1_Frequency(1);
            F2 = Tachometer_1_Frequency(2);
            F3 = Tachometer_1_Frequency(3);
            F4 = Tachometer_1_Frequency(4);
            F5 = Tachometer_1_Frequency(5);
            F6 = Tachometer_1_Frequency(6);
            F7 = Tachometer_1_Frequency(7);

            sprintf(buf, "%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\r\n", \
                          F0, F1, F2, F3, F4, F5, F6, F7 );

            UART_1_PutString(buf);
         }
    }
}
```

[*] Associated project: Tach24_x8_basic_01a.cyproj

Alternative code using GetAllFreq() API:

```
int main()
{
    Tachometer_1_Start();
    char buf[255];
    float F[Tachometer_1_NumInput];     // results array

    for(;;)
    {
        if (Tachometer_1_DataReady == Tachometer_1_MASK)
        {
            Tachometer_1_DataReady = 0; // reset flag

            Tachometer_1_GetAllFreq(F); // calculate results

            sprintf(buf, "%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\r\n", \
                         F[0],F[1],F[2],F[3],F[4],F[5],F[6],F[7] );

            UART_1_PutString(buf);
         }
    }
}
```

When input frequency is above the capture rate, the data is returned on every capture clock. When input frequency is below the capture frequency (channels 6 and 7), the data is captured at the input rate, returning a zero when timeout detected.

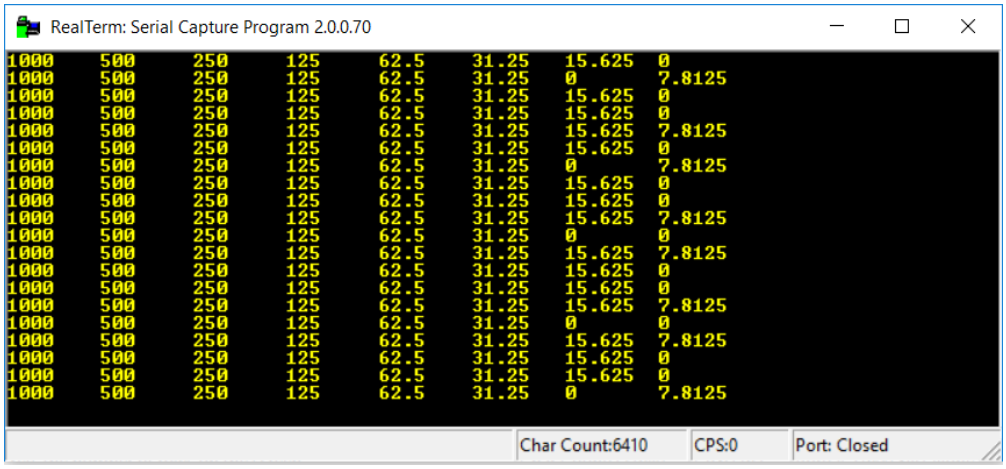


Figure 20. Terminal output of 8 sampled inputs. The frequency value is zero when timeout is detected.

Appendix 4

Sampling of 24 inputs

PSoC5 example of Tachometer sampling 24 parallel inputs is shown on Figure 21[(*)].

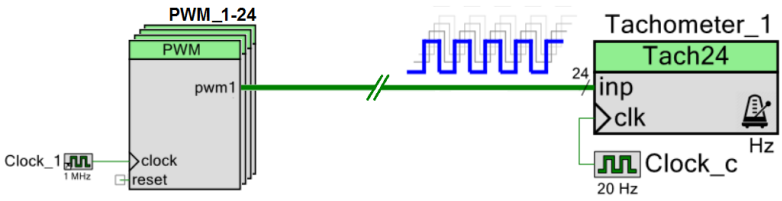


Figure 21. Example of sampling 24 inputs using three Tachometers. Tachometer is configured for Hz output format with timeout enabled.

Tachometer is configured for 24 channels, float precision output in Hz format, with timeout enabled. The input bus is sourced by the array of PWMs with output frequencies: 200 Hz, 199.96 Hz, 199.92 Hz, etc. derived from a design-wide clock of 1 MHz using dividers 5000, 5001, 5002, etc. The capture clock is set to 20 Hz.

Frequency results for the selective channels are plotted using SerialPlot charting tool [7] (Figure 20). Due to its limitations only 8 out of total 24 channels could be plotted at a time. The unfiltered data show spurious artifacts of less than ±0.04 Hz amplitude, corresponding to resolution 200 ppm, or 12.3 bits. When enabled, the median filter rejects spurious artifacts almost entirely. Associated source code is provided below.

```
int main()
{
    Initialize();

    float F[Tachometer_1_NumInputs];   // results array

    for(;;)
    {
        if (Tachometer_1_DataReady == Tachometer_1_MASK)
        {
            Tachometer_1_DataReady=0; // reset flag

            Tachometer_1_GetAllFreq(F);

            // plot selected data->
            Chart_1_Plot( F[0],F[3],F[6],F[9],F[12],F[15],F[18],F[21] );
        }
    }
}
```

[*] Associated demo project: Tach24_x24_01a.cyproj

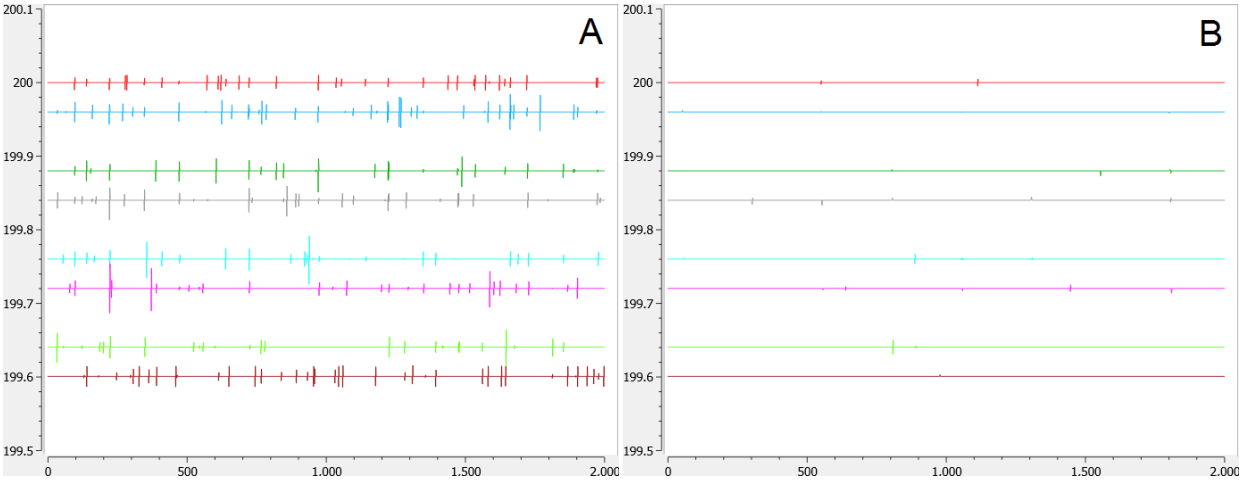


Figure 22. Tachometer performance simultaneously sampling 24 channels; A - raw data, B - with filter enabled. Only 8 channels (#0, #3, #6, #9, #12, #15, #18 and #21) out of total 24 are displayed. Input frequency for the first channel is 200 Hz; the noise doesn't exceed 0.04 Hz (200 ppm). Tachometer is configured for float precision, timeout enabled. Capture clock frequency 20 Hz, BUS_CLK 48 MHz. Compiler is set for release mode.

Appendix 5

Using per-channel API

The demo code using per-channel API is shown below. It checks if frequency of any channel exceeds 200 Hz. It works in any timeout mode, responding as soon as a valid capture occurs on any channel.

```
int main()
{
   for(;;)
   {
      for(int i=0; i<Tachometer_1_NumInputs; i++)    // scan Tachometer channels
      {
         if (Tachometer_1_DataReady(i) && !Tachometer_1_Timeout(i))  // valid capture
         {
            Tachometer_1_DataReady(i)=0;             // clear channel flag

            float F = Tachometer_1_Frequency(i);     // get result

            if (F > 200) {                           // check high limit
               BlinkLED();                           // do something..
            }
         }
      }
   }
}
```

The alternative code using common API works only when timeout is enabled, waiting for all channels to report their statuses before processing:

```
int main()
{

   for(;;)
   {
      if (Tachometer_1_DataReady==Tachometer_1_MASK) // all channels reported status
      {
         Tachometer_1_DataReady=0;                    // clear common flag

         for(int i=0; i<Tachometer_1_NumInputs; i++) // scan channels
         {
            if (!Tachometer_1_Timeout(i))             // if valid capture
            {
               float F = Tachometer_1_Frequency(i);  // get result

               if (F > 200)                           // check high limit
               {
                   BlinkLED();                        // do something..
               }
            }
         }
      }
   }
}
```