MedianFilter: sliding window median filter
0.0

Features

Filter_N
MedianFilter
int32
size=15

- Implements sliding window median algorithm
- Range: int8, int16, int32, uint8, uint16, uint32
- Doesn't use hardware resources
- Has fixed execution time
- Grows linear with size
- Non-decimating

General description

The MedianFilter(*) component implements Phil Extrom's sliding window median algorithm(†) for scalar data [1]. Sampled signals with few amounts of erroneous data can be effectively de-noised using this filter. Component doesn't consume hardware resources, performing all operations by CPU, which is useful for systems with little resources, such as PoC4. The filter is non-decimating, producing the output on every sample added. Multiple instances of the component can be added to the project for processing independent signal streams.

When to use MedianFilter component

Component was developed for a weight scale project, where ADC signals from several load cells must be filtered from erroneous spikes occurring during weight load/unload operation. It can be useful whenever a digitized signal needs to be cleaned out from erroneous artefacts such as digital noise leaking into the analog path or mechanical noise from potentiometer slider. Component is useful for a system with limited hardware resources, such as PSoC4. Component was tested using CY8KIT-059 PSoC5LP Prototyping Kit and CY8KIT-042 PSoC4 Pioneer Kit. Demo projects are provided.

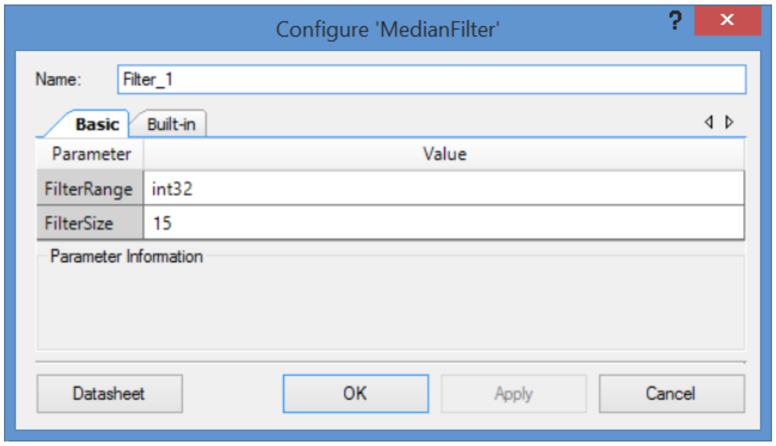
* Hereafter referred to as "Filter"
† Also known as moving median, running median and rolling median filter

Input-output connections

Component does not have any input/output connections. It is, in essence, a software library, performing all operations by API. Unlike library, multiple instances of the component can run simultaneously in the project.

Parameters and Settings

Basic dialog provides following parameters(*):



Name: Filter_1

Parameter	Value
FilterRange	int32
FilterSize	15

FilterRange (int8 / int16 / int32 / uint8 / uint16 / uint32)

Selects filter range. Valid options are int8, int16, int32, uint8, uint16, uint32. The input data type must be a subset of the filter range type. See **Application Programming Interface** section for details. Default setting is int32, which fits most input data types. The value can't be changed during the run-time.

FilterSize (int16)

Sets filter window length. Filter allocates FIFO buffer of this length to keep last incoming data samples. The value must be odd in the range [3, 5, … , 255]. Default value is 15. Thought the algorithm works both for odd and even sizes, the Dialog input is restricted to only odd values (N=2k+1) to avoid ambiguity. The value can't be changed during the run-time.

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

Application Programming Interface

Function	Description
`Filter_AddValue()`	Add next data sample

range_t ADC_AddValue(range_t value)

Description: Calculates median value for last N data samples added, where N is filter size. Note that the MedianFilter needs priming, and first N-1 outputs after the start should be discarded. The range_t represents selected Filter range: int8, int16, int32, uint8, uint16, uint32.

Parameters: input data. The data type should be a subset of the Filter range (the input data must fit into the Filter range), see Table 1. To avoid possible collision of the input data with the STOPPER parameter value, select larger Filter range. See **Implementation** section for details.

Table 1. Useful Filter ranges for various input data types[*].

Input data range	Filter range					
	int8	int16	int32	uint8	uint16	uint32
int8	±	+	+			
int16		±	+			
int32			±			
uint8		+	+	±		
uint16			+		±	±
uint32						±

[*] Shaded cells indicate potential collision of input data with STOPPER value.

Return Value: filtered output.

Functional Description

The median filter is a non-linear filtering technique in digital signal processing, often used for removing impulsive signal noise while maintaining signal trends. Unlike a linear FIR filter, it requires sorting of the array to extract a median. The median filter always selects an actual data point from the input signal, whereas the FIR filter returns calculated value.

Performance comparison of the median vs linear filter depends on the signal noise shape. For Gaussian noise shape (“normal” noise), the median filtering offers no advantages over linear (FIR) filtering techniques [1]. In many practical applications, however, the noise shape is often far from normal, being polluted with “outliers” - random spikes, caused by noise from a different source. Typical example is digital noise of a microcontroller, leaking into the ADC analog interface. Another example is a mechanical potentiometer with a slider moving over resistive element; any imperfections on its way cause voltage spikes on top of the smooth analog signal. Conventional linear filters can’t filter out such noise.

A comparison of median vs FIR filter for randomly distributed noise(*) is shown on Figure 1. For 100% random noise, FIR filter outperforms the median; FIR output also looks “smoother” than the median, because it returns a calculated value. When random noise becomes 50% sparse (the rest 50% of the time the signal has no noise), the median filter rejects noise entirely, dramatically outperforming FIR filter.

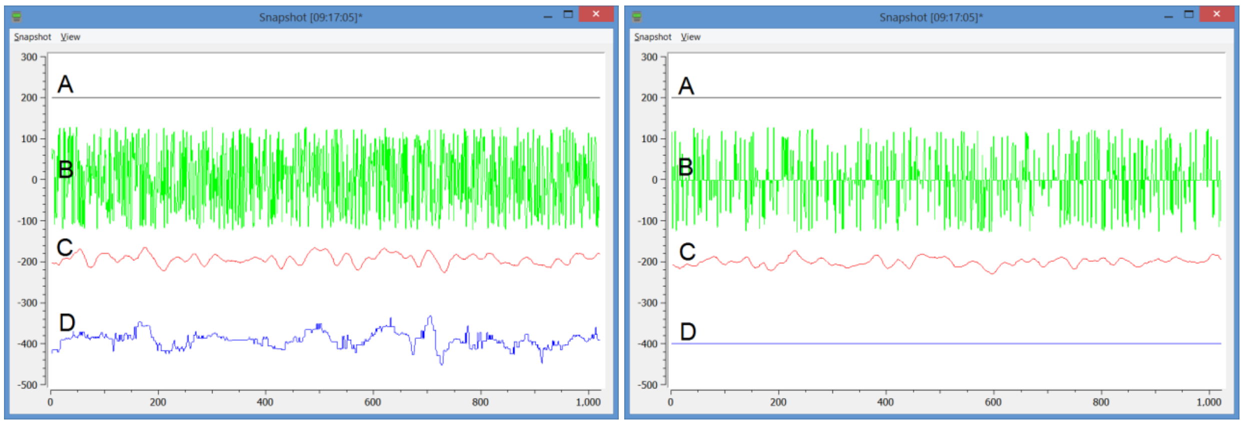


Figure 1. Effect of noise shape on median and FIR filter outputs: Left – signal with added 100% density random flat noise; Right – noise density is 50%. Filter size is 31. A – source signal w/o noise; B – signal with noise added; C – FIR output; D – median filter output. Vertical offsets added for clarity. Notice that at 100% noise density FIR filter outperforms median filter. At 25% noise density, however, median filter dramatically outperforms FIR, removing noise entirely.

(*) Noise was generated using rand() function, with flat distribution in range -32768 to 32767.

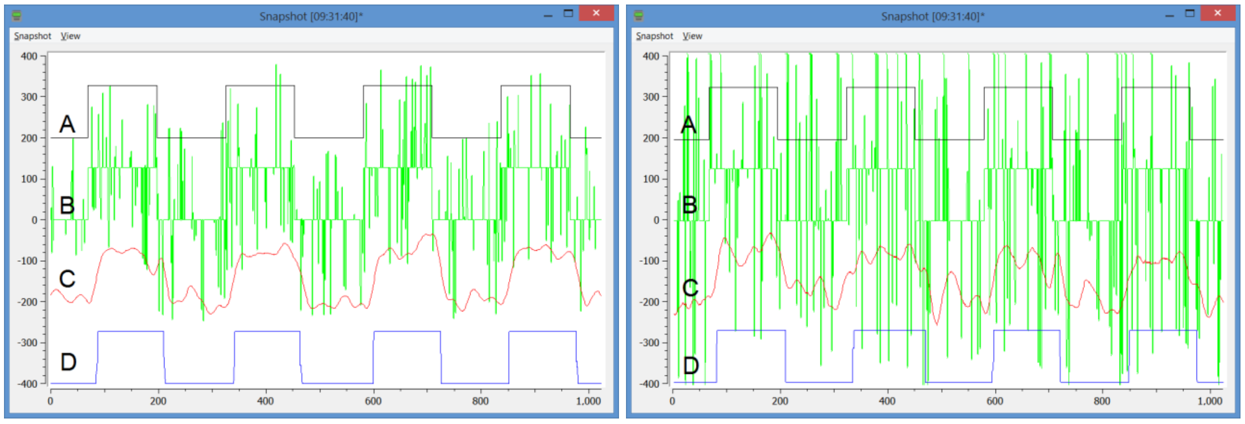


Figure 2. Effect of noise amplitude on median and FIR filter outputs. Left - random noise of amplitude 256, density 25%; Right - noise amplitude 512. Filter size is 31. A – source signal w/o noise; B – signal with noise added; C – FIR output; D – median filter output. Vertical offsets added for clarity. Notice that median filter recovers original signal amplitude and wavefronts, and output is not affected by the noise amplitude. Contrary to it, FIR's slew rate is bandwidth-limited, and its performance gets worse with noise.

All median filters exhibit a delay of (FilterSize / 2 + 1) samples before responding to a step change. When they do respond, the output is also a step change [2]. By contrast a classic linear filter will slew to the stepped value at a rate governed by the band of the filter (Figures 2, 3). Median filter excels in rejecting pulsed noise due to its unique property of being insensitive to the noise amplitude while preserving the shape of the recovered signal, unlike the FIR filter, which output deviation scales up with noise (Figure 3, 4).

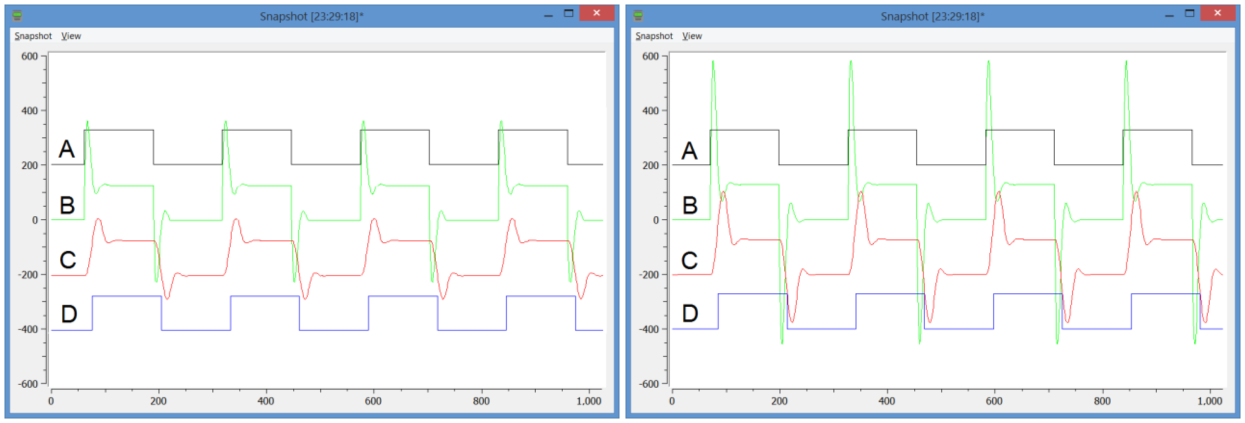


Figure 3. Effect of pulsed noise on median and FIR filter outputs. Left – exponentially fading coherent pulsed noise of amplitude 512; Right – amplitude is 1024. Filter size is 31. A – source signal w/o noise; B – signal with noise added; C – FIR output; D – median filter output. Vertical offsets added for clarity. Notice that median filter rejects pulsed noise and recovers signal amplitude and wavefronts. Performance of the FIR filter gets worse with noise amplitude, and its slew rate is bandwidth-limited.

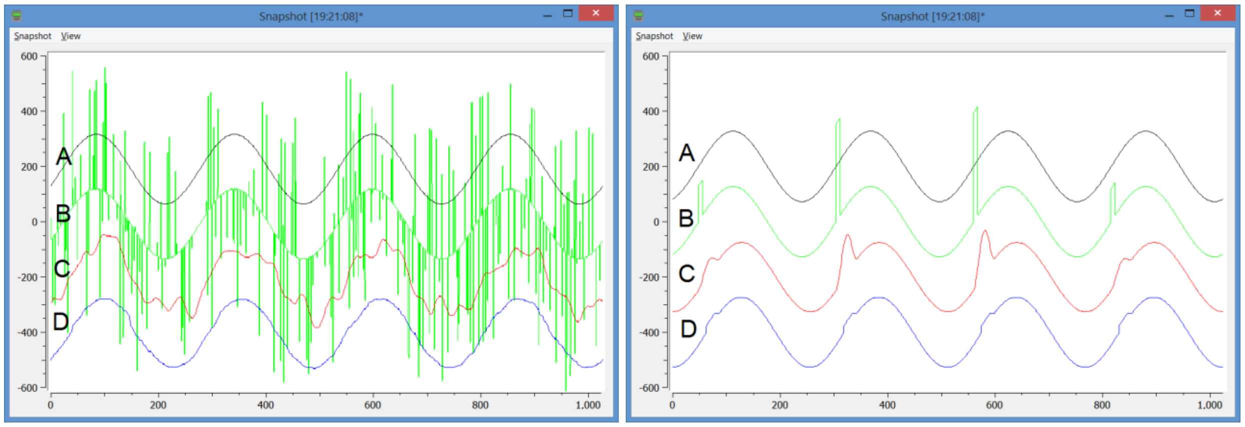


Figure 4. Effect of median and FIR filters on noisy sinusoidal input signal. Left – with random noise of amplitude 256, density 25%; Right – with coherent PWM pulsed noise of random amplitude. Filter size is 31. A – signal w/o noise; B – signal with noise added; C – FIR output; D – median filter output. Vertical offsets added for clarity. Notice that median filter outperforms FIR, effectively rejecting all spikes, irrespective of the amplitude. Performance of the FIR filter gets worse with rise of spike amplitude.

Despite its advantages, the median filtering comes at a price. It is usually slower than a FIR filter of the same size, as it based on some sorting algorithm, which execution time typically grows with size as N*log(N). For that reason, the median filtering is rarely used for large window sizes. The Extrom algorithm, however, scales up linearly with size because previous buffer is already sorted. See **Performance** section for details.

Implementation

The median filter algorithm

The Filter algorithm was published by Phil Extrom [1]. The algorithm uses a double linked list to keep sorted order of the incoming values. The code is provided in **Appendix 1**. Compared to several other algorithms [3], it shows best average performance on large data sets. The unique feature of the algorithm is that it has same execution time independent of the data sequence.

Compared to others, the Extrom algorithm is quite mindboggling, heavily using pointers to the double linked list. It is also spoiled by the STOPPER parameter, which is discussed below.

The STOPPER parameter

The Extrom algorithm uses internal STOPPER parameter, which value must be lower than any possible input sample. Selected STOPPER values are listed in Table 2. If incoming sample accidentally equals STOPPER, it is incremented by +1 to avoid the collision (see **Appendix 1** for details). In practice, this is rarely an issue unless the input data gets continuously saturated at the bottom of the selected range. For example if both ADC and the Filter were configured for uint8 data range (STOPPER = 0u) and ADC output saturated at 0u, the output of the Filter will be 1u instead of expected 0u! One simple way to avoid this issue is to set Filter range to int16 or int32 data, which guarantees that the STOPPER value is outside of the input data range. For example, selecting int32 Filter range guarantees correct performance for any of the 8, 16 or 24-bit input signal, signed or unsigned (see Table 1).

Table 2. Value of the STOPPER for different Filter ranges.

Filter range	int8	int16	int32	uint8 / uint16 / uint32
STOPPER	-2^{7}	-2^{15}	-2^{31}	0

Features not implemented

- Filter reset
- Output valid flag

Performance

Component was tested using CY8KIT-059 PSoC5LP Prototyping Kit and CY8KIT-042 PSoC4 Pioneer Kit. The component doesn't use UDB, performing all operation entirely by CPU. Such approach saves valuable hardware resources, but may be limited to low sampling rates. Results for PSoC5LP are presented below. Results for PSoC4 are typically ~20% slower.

Filter execution time[*] for random dataset is shown on Figure 5. At small sizes it grows linearly as $51 + 18 \times N$ (inset), while over the entire range it better fits $61 + 17 \times N$. Such linear performance is due to the fact that insertion of the new value into previously sorted buffer needs maximum of N steps.

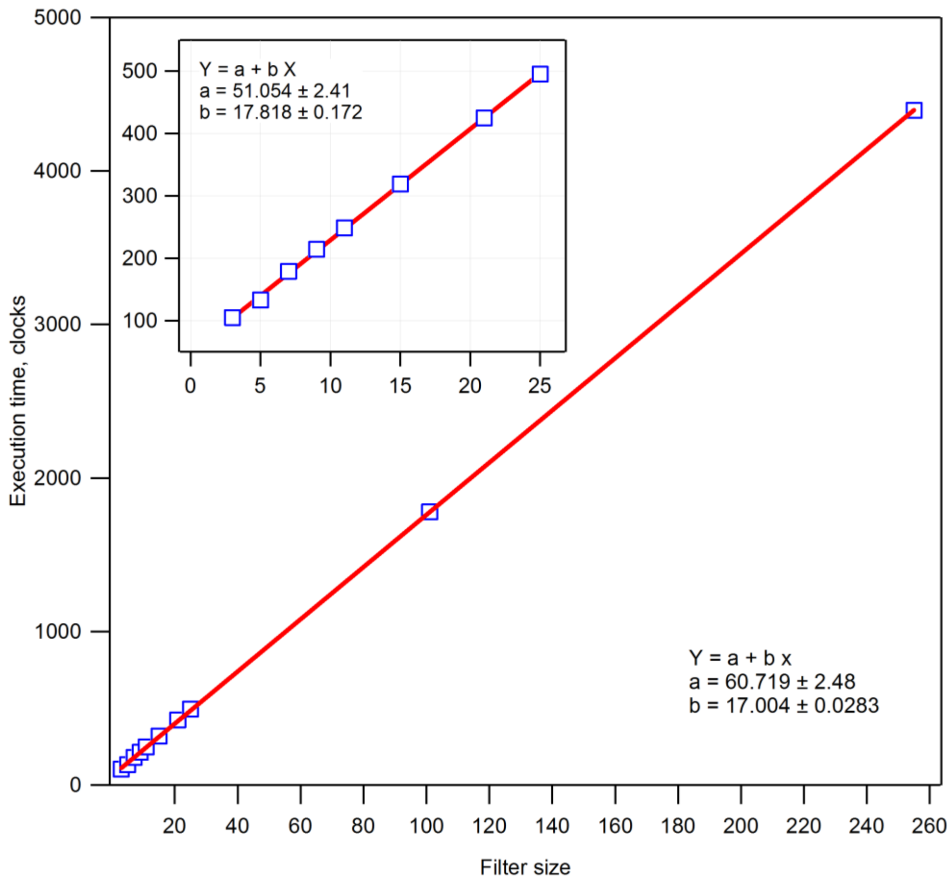


Figure 5. Execution time (CPU clocks) vs. filter size, measured over random input data using PSoC5. Input data were generated using rand() function in the int16 range (-32768 ÷ 32767). Filter range was set to int32.

[*] Compiled in release mode, with compiler optimization set to speed

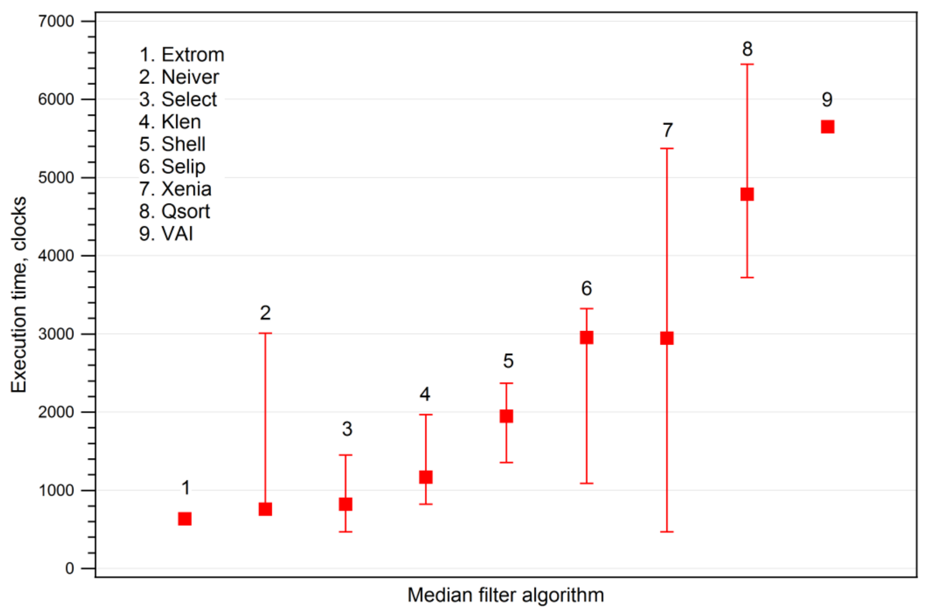


Figure 6. Median algorithms performance results, tested over large dataset. Filter size 31. Squares – average performance (processor clocks); error bars show dispersion (minimum to maximum execution time).

The algorithms performance comparison is shown on the Figure 6 [3]. For medium-sized filters the Extrom algorithm outperformed several other tested over large data set. It also shows minimal dispersion, which guarantees stable execution time, independent of the data sequence. The Extrom algorithm was reported to have better performance for any filter size larger than 3 [4].

Resources

Component does not consume hardware resources. It doesn’t use interrupts, clocks or UDB. Component does not have built-in DMA capabilities.

Sample Firmware Source Code

Demo project is provided, see **Appendix 2** for details.

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. Phil Extrom, Better than Average, Embedded Systems Programming, Nov 2000, pp.100-110, https://www.embedded.com/better-than-average/ https://m.eet.com/media/1173225/f-eckstro.pdf
2. Wesley Bylsma, Algorithm Alley, https://www.drdobbs.com/parallel/algorithm-alley/184411079
3. Forum Electronix.ru, Median algorithm comparison test, 2013, https://electronix.ru/forum/index.php?app=forums&module=forums&controller=topic&id=114436&page=4&tab=comments#comment-1182441
4. Nigel Jones, Median Filter Performance Results, 2010 https://embeddedgurus.com/stack-overflow/2010/11/median-filter-performance-results/
5. Nigel Jones, Median Filtering, 2010 https://embeddedgurus.com/stack-overflow/2010/10/median-filtering/

Appendix 1

The median filter algorithm

Component implemented entirely in code. The actual code was taken from the Ref. [5]:

```
int16 Filter_1_AddValue(int16 datum)
{
  struct pair
  {
    struct pair *point;                        // Pointers list linked in sorted order
    int16   value;                             // Values to sort
  };

  static struct pair buffer[FilterSize]={};    // Buffer of nwidth pairs
  static struct pair *datpoint = buffer;       // Pointer into circular buffer of data
  static struct pair small = {NULL,STOPPER};   // Chain stopper
  static struct pair big = {&small, 0};        // Pointer to head of linked list.

  struct pair *successor;                      // Ptr to successor of replaced  item
  struct pair *scan;                           // Ptr used to scan down the sorted list
  struct pair *scanold;                        // Previous value of scan
  struct pair *median;                         // Pointer to median

  if (datum == STOPPER) {datum=STOPPER+1;}     // No stoppers allowed

  if ( (++datpoint-buffer) >= FilterSize) {
      datpoint = buffer;                       // Increment and wrap data in pointer
  }

  datpoint->value = datum;                     // Copy in new datum
  successor = datpoint->point;                 // Save ptr to old value's successor
  median = &big;                               // Median initially to first in chain
  scanold = NULL;                              // Scanold initially null
  scan = &big;                                 // Points to ptr to first (largest)
                                               // datum in chain

  if (scan->point == datpoint) {               // Handle chain-out of first item in
    scan->point = successor;                   // chain as special case
  }

  scanold = scan;                              // Save this pointer and
  scan = scan->point ;                         // step down chain

  uint16 i;
  for (i = 0; i < FilterSize; ++i)             // Loop through the chain, normal loop
  {                                            // exit is via break

    if (scan->point == datpoint) {             // Handle odd-numbered item in chain
        scan->point = successor;               // Chain out the old datum
    }

    if (scan->value < datum)                   // If data is larger than scanned value
    {
      datpoint->point = scanold->point;        // Chain it in here
      scanold->point  = datpoint;              // Mark it chained in
        datum = STOPPER;
    };
    // Step median pointer down chain
    // after doing odd-numbered element
    median = median->point;                     // Step median pointer

    if (scan == &small) break;                  // Break at end of chain

    scanold = scan;                             // Save this pointer and
    scan = scan->point;                         // step down chain

    if (scan->point == datpoint) {              // Handle even-numbered item in chain
        scan->point = successor;
    }

    if (scan->value < datum)
    {
      datpoint->point = scanold->point;
      scanold->point  = datpoint;
      datum = STOPPER;
    }

    if (scan == &small) break;

    scanold = scan;
    scan = scan->point;
  }

  return median->value;
}
```

Using the Filter API in the main loop, where the *sample* comes from ADC or signal generator routine:

```
for(;;)                                         // main loop
{
  if (isrTimer_flag != 0)                       // Timer interrupt
  {
    median = Filter_1_AddValue(sample);         // add next data sample
    . . .
  }
}
```

Appendix 2

Component demo project

The PSoC5 project example using MedianFilter component is shown on Figure 7. Test data samples are generated on clock timer and added to the Filter. The Filter output is streamed along with original data samples to the plotting software$^{(*)}$ using USB-UART bridge, built into the KitProg.

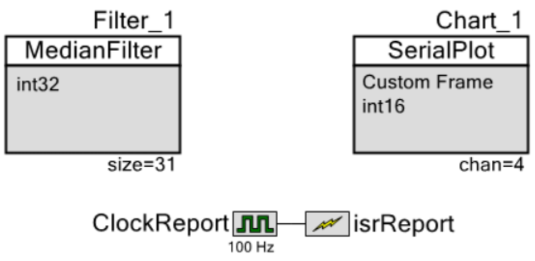
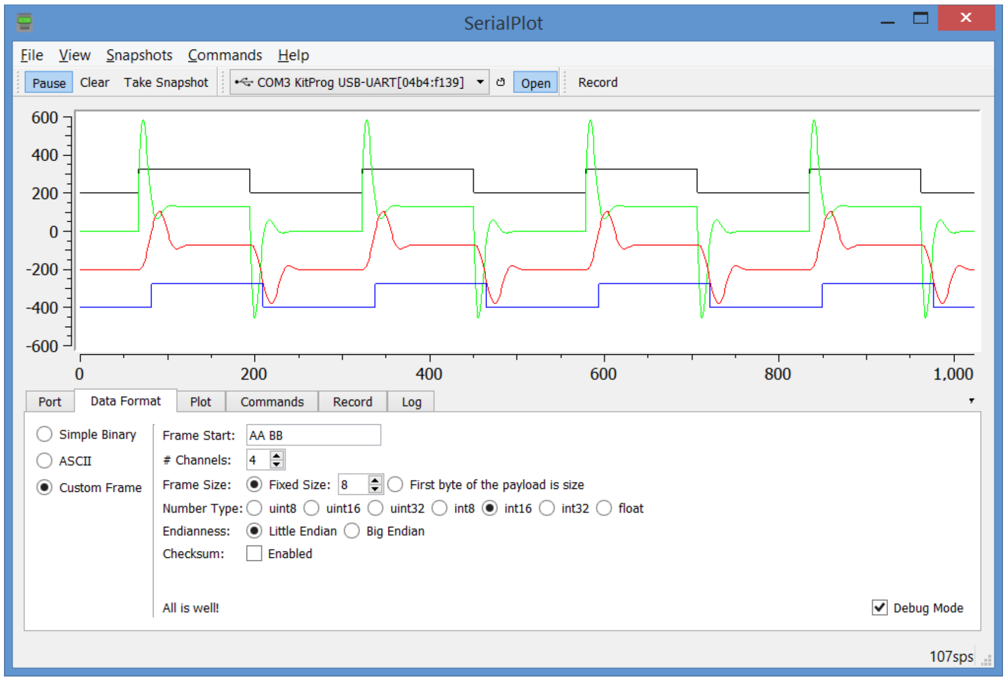


Figure 7. MedianFilter project schematic.

The Filter input data is a square wave with large transition spikes. The Filter effectively rejects the spikes and recovers the square wave. The FIR filter output is obtained using PSoC5 built-in Digital Filter and provided for comparison (this option is absent in PSoC4 basic demo).


SerialPlot
File View Snapshots Commands Help
Pause Clear Take Snapshot
COM3 KitProg USB-UART[04b4:f139]
Open Record
600 400 200 0 -200 -400 -600
0 200 400 600 800 1,000
Port Data Format Plot Commands Record Log
107sps

Simple Binary
ASCII
[✓ Custom Frame]

Frame Start: AA BB
# Channels: 4
Frame Size: [✓ Fixed Size:] 8 First byte of the payload is size
Number Type: uint8 uint16 uint32 int8 [✓ int16] int32 float
Endianness: [✓ Little Endian] Big Endian
Checksum: Enabled

All is well!

[✓ Debug Mode]

Figure 8. Filter response to the square wave with transition artifacts. Filter length is 31. Black line – signal w/o noise; Green – signal with noise; Red – FIR output; Blue – median filter output (recovered data). Vertical offsets are added for clarity. Data Format panel displays configuration settings.

* SerialPlot – Real time Plotting Software for UART/Serial Port, https://hasanyavuz.ozderya.net/?p=244

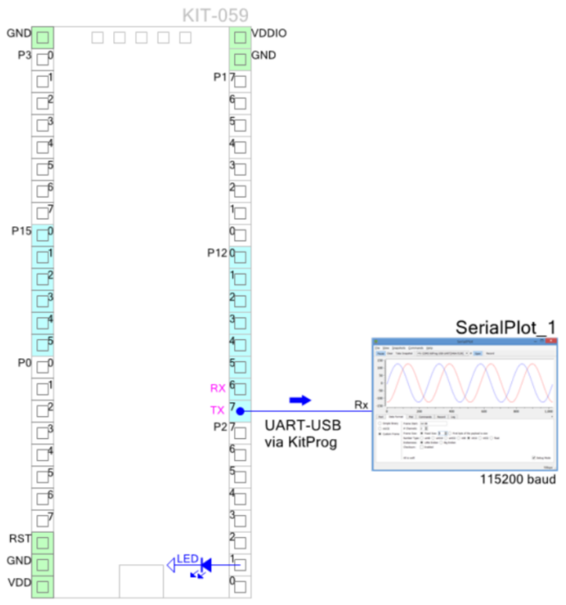


Figure 9. Project annotation for PSoC5 CY8CKIT-059 using PSoC Annotation library[*].

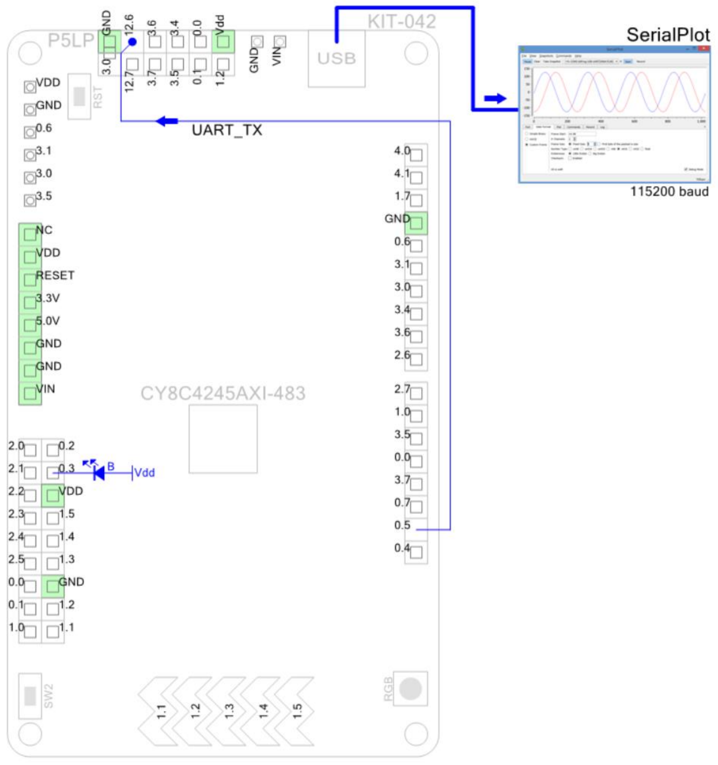


Figure 10. Project annotation for PSoC4 Pioneer Kit (CY8CKIT-042) using KIT-042[†] stub.

[*] PSoC Annotation Library v1.0, https://community.cypress.com/thread/48049
[†] KIT-042: annotation component for CY8CKIT-042 Pioneer Kit, https://community.cypress.com/thread/48741