# FIR Filter IP Library

The AMD FIR IP block can be called within a C++ design using the library `hls_fir.h`. This section explains how the FIR can be configured in your C++ code.

**RECOMMENDED:** *AMD highly recommends that you review the FIR Compiler LogiCORE IP Product Guide (PG149) for information on how to implement and use the features of the IP.*

To use the FIR in your C++ code:

1. Include the `hls_fir.h` library in the code.
2. Set the static parameters using the predefined struct `hls::ip_fir::params_t`.
3. Call the FIR function.
4. Optionally, define a runtime input configuration to modify some parameters dynamically.

The following code examples provide a summary of how each of these steps is performed. Each step is discussed in more detail below.

First, include the FIR library in the source code. This header file resides in the include directory in the Vitis HLS installation area. This directory is automatically searched when Vitis HLS executes. There is no need to specify the path to this directory if compiling inside Vitis HLS.

```
#include "hls_fir.h"
```

Define the static parameters of the FIR. This includes such static attributes such as the input width, the coefficients, the filter rate (`single`, `decimation`, `hilbert`). The FIR library includes a parameterization struct `hls::ip_fir::params_t` which can be used to initialize all static parameters with default values.

In this example, the coefficients are defined as residing in array `coeff_vec` and the default values for the number of coefficients, the input width and the quantization mode are over-ridden using a user a user-defined struct `myconfig` based on the predefined struct.

```
struct myconfig : hls::ip_fir::params_t {
    static const double coeff_vec[sg_fir_srrc_coeffs_len];
    static const unsigned num_coeffs = sg_fir_srrc_coeffs_len;
    static const unsigned input_width = INPUT_WIDTH;
    static const unsigned quantization = hls::ip_fir::quantize_only;
};
```

Create an instance of the FIR function using the HLS namespace with the defined static parameters (`myconfig` in this example) and then call the function with the `run` method to execute the function. The function arguments are, in order, input data and output data.

```
static hls::FIR<param1> fir1;
fir1.run(fir_in, fir_out);
```

Optionally, a runtime input configuration can be used. In some modes of the FIR, the data on this input determines how the coefficients are used during interleaved channels or when coefficient reloading is required. This configuration can be dynamic and is therefore defined as a variable. For a complete description of which modes require this input configuration, refer to the *FIR Compiler LogiCORE IP Product Guide* (PG149).

When the runtime input configuration is used, the FIR function is called with three arguments: input data, output data and input configuration.

```
// Define the configuration type
typedef ap_uint<8> config_t;
// Define the configuration variable
config_t fir_config = 8;
// Use the configuration in the FFT
static hls::FIR<param1> fir1;
fir1.run(fir_in, fir_out, &fir_config);
```

**TIP:** *The example above shows the use of scalar values and arrays, but the FIR function also supports the use of `hls::stream` for arguments. Refer to Vitis HLS Introductory Examples for more information.*

# FIR Static Parameters

The static parameters of the FIR define how the FIR IP is parameterized and specifies non-dynamic items such as the input and output widths, the number of fractional bits, the coefficient values, the interpolation and decimation rates. Most of these configurations have default values: there are no default values for the coefficients.

The `hls_fir.h` header file defines a `struct hls::ip_fir::params_t` that can be used to set the default values for most of the static parameters.

**IMPORTANT!** *There are no defaults defined for the coefficients. Therefore, AMD does not recommend using the pre-defined struct to directly initialize the FIR. A new user defined struct which specifies the coefficients should always be used to perform the static parameterization.*

In this example, a new user `struct my_config` is defined and with a new value for the coefficients. The coefficients are specified as residing in array `coeff_vec`. All other parameters to the FIR use the default values.

```
struct myconfig : hls::ip_fir::params_t {
    static const double coeff_vec[sg_fir_srrc_coeffs_len];
};
static hls::FIR<myconfig> fir1;
fir1.run(fir_in, fir_out);
```

### Fir Struct Parameters

The following table describes the parameters used for the parametrization struct `hls::ip_fir::params_t`and lists the default values for the parameters as well as the possible values.

*Table 57:* **FIR Struct Parameter Values**

| Parameter | Description | C Type | Default Value | Valid Values |
|---|---|---|---|---|
| input_width | Data input port width | unsigned | 16 | No limitation |
| input_fractional_bits | Number of fractional bits on the input port | unsigned | 0 | Limited by size of input_width |
| output_width | Data output port width | unsigned | 24 | No limitation |
| output_fractional_bits | Number of fractional bits on the output port | unsigned | 0 | Limited by size of output_width |
| coeff_width | Bit-width of the coefficients | unsigned | 16 | No limitation |
| coeff_fractional_bits | Number of fractional bits in the coefficients | unsigned | 0 | Limited by size of coeff_width |
| num_coeffs | Number of coefficients | bool | 21 | Full |
| coeff_sets | Number of coefficient sets | unsigned | 1 | 1-1024 |
| input_length | Number of samples in the input data | unsigned | 21 | No limitation |
| output_length | Number of samples in the output data | unsigned | 21 | No limitation |
| num_channels | Specify the number of channels of data to process | unsigned | 1 | 1-1024 |
| total_num_coeff | Total number of coefficients | unsigned | 21 | num_coeffs * coeff_sets |
| coeff_vec[total_num_coeff] | The coefficient array | double array | None | Not applicable |
| filter_type | The type implementation used for the filter | unsigned | single_rate | single_rate, interpolation, decimation, hilbert_filter, interpolated |
| rate_change | Specifies integer or fractional rate changes | unsigned | integer | integer, fixed_fractional |
| interp_rate | The interpolation rate | unsigned | 1 | 1-1024 |
| decim_rate | The decimation rate | unsigned | 1 | 1-1024 |
| zero_pack_factor | Number of zero coefficients used in interpolation | unsigned | 1 | 1-8 |
| rate_specification | Specify the rate as frequency or period | unsigned | period | frequency, period |
| hardware_oversampling_rate | Specify the rate of over-sampling | unsigned | 1 | No Limitation |
| sample_period | The hardware oversample period | bool | 1 | No Limitation |
| sample_frequency | The hardware oversample frequency | unsigned | 0.001 | No Limitation |
| quantization | The quantization method to be used | unsigned | integer_coefficients | integer_coefficients, quantize_only, maximize_dynamic_range |
| best_precision | Enable or disable the best precision | unsigned | false | false<br>true |

*Table 57:* **FIR Struct Parameter Values** *(cont'd)*

| Parameter | Description | C Type | Default Value | Valid Values |
|---|---|---|---|---|
| coeff_structure | The type of coefficient structure to be used | unsigned | non_symmetric | inferred, non_symmetric, symmetric, negative_symmetric, half_band, hilbert |
| output_rounding_mode | Type of rounding used on the output | unsigned | full_precision | full_precision, truncate_lsbs, non_symmetric_rounding_down, non_symmetric_rounding_up, symmetric_rounding_to_zero, symmetric_rounding_to_infinity, convergent_rounding_to_even, convergent_rounding_to_odd |
| filter_arch | Selects a systolic or transposed architecture | unsigned | systolic_multiply_accumulate | systolic_multiply_accumulate, transpose_multiply_accumulate |
| optimization_goal | Specify a speed or area goal for optimization | unsigned | area | area, speed |
| inter_column_pipe_length | The pipeline length required between DSP columns | unsigned | 4 | 1-16 |
| column_config | Specifies the number of DSP module columns | unsigned | 1 | Limited by number of DSP macrocells used |
| config_method | Specifies how the DSP module columns are configured | unsigned | single | single, by_channel |
| coeff_padding | Specifies if zero padding is added to the front of the filter | bool | false | false<br>true |

**IMPORTANT!** *When specifying parameter values which are not integer or boolean, the HLS FIR namespace should be used. For example the possible values for* `rate_change` *are shown in the table to be* `integer` *and* `fixed_fractional`*. The values used in the C program should be* `rate_change = hls::ip_fir::integer` *and* `rate_change = hls::ip_fir::fixed_fractional`*.*

# Using the FIR Function with Array Interface

The FIR function is defined in the HLS namespace and can be called as follows:

```
// Create an instance of the FIR
static hls::FIR<STATIC_PARAM> fir1;
// Execute the FIR instance fir1
fir1.run(INPUT_DATA_ARRAY, OUTPUT_DATA_ARRAY);
```

The `STATIC_PARAM` is the static parameterization struct that defines most static parameters for the FIR.

Both the input and output data are supplied to the function as arrays (`INPUT_DATA_ARRAY` and `OUTPUT_DATA_ARRAY`). In the final implementation, these ports on the FIR IP will be implemented as AXI4-Stream ports. AMD recommends always using the FIR function in a region using the dataflow optimization (`set_directive_dataflow`), because this ensures the arrays are implemented as streaming arrays. An alternative is to specify both arrays as streaming using the `set_directive_stream` command.

**IMPORTANT!** *The FIR cannot be used in a region which is pipelined. If high-performance operation is required, pipeline the loops or functions before and after the FIR then use dataflow optimization on all loops and functions in the region.*

The multichannel functionality of the FIR is supported through interleaving the data in a single input and single output array.

- The size of the input array should be large enough to accommodate all samples: `num_channels * input_length`.
- The output array size should be specified to contain all output samples: `num_channels * output_length`.

The following code example demonstrates, for two channels, how the data is interleaved. In this example, the top-level function has two channels of input data (`din_i`, `din_q`) and two channels of output data (`dout_i`, `dout_q`). Two functions, at the front-end (fe) and back-end (be) are used to correctly order the data in the FIR input array and extract it from the FIR output array.

```
void dummy_fe(din_t din_i[LENGTH], din_t din_q[LENGTH], din_t
out[FIR_LENGTH]) {
    for (unsigned i = 0; i < LENGTH; ++i) {
        out[2*i] = din_i[i];
        out[2*i + 1] = din_q[i];
    }
}
void dummy_be(dout_t in[FIR_LENGTH], dout_t dout_i[LENGTH], dout_t
dout_q[LENGTH]) {
    for(unsigned i = 0; i < LENGTH; ++i) {
        dout_i[i] = in[2*i];
        dout_q[i] = in[2*i+1];
    }
}
void fir_top(din_t din_i[LENGTH], din_t din_q[LENGTH],
             dout_t dout_i[LENGTH], dout_t dout_q[LENGTH]) {

 din_t fir_in[FIR_LENGTH];
    dout_t fir_out[FIR_LENGTH];
    static hls::FIR<myconfig> fir1;

    dummy_fe(din_i, din_q, fir_in);
    fir1.run(fir_in, fir_out);
    dummy_be(fir_out, dout_i, dout_q);
}
```

# Optional FIR Runtime Configuration

In some modes of operation, the FIR requires an additional input to configure how the coefficients are used. For a complete description of which modes require this input configuration, refer to the *FIR Compiler LogiCORE IP Product Guide* (PG149).

This input configuration can be performed in the C code using a standard ap_int.h 8-bit data type. In this example, the header file `fir_top.h` specifies the use of the FIR and `ap_fixed` libraries, defines a number of the design parameter values and then defines some fixed-point types based on these:

```
#include "ap_fixed.h"
#include "hls_fir.h"

const unsigned FIR_LENGTH   = 21;
const unsigned INPUT_WIDTH = 16;
const unsigned INPUT_FRACTIONAL_BITS = 0;
const unsigned OUTPUT_WIDTH = 24;
const unsigned OUTPUT_FRACTIONAL_BITS = 0;
const unsigned COEFF_WIDTH = 16;
const unsigned COEFF_FRACTIONAL_BITS = 0;
const unsigned COEFF_NUM = 7;
const unsigned COEFF_SETS = 3;
const unsigned INPUT_LENGTH = FIR_LENGTH;
const unsigned OUTPUT_LENGTH = FIR_LENGTH;
const unsigned CHAN_NUM = 1;
typedef ap_fixed<INPUT_WIDTH, INPUT_WIDTH - INPUT_FRACTIONAL_BITS> s_data_t;
typedef ap_fixed<OUTPUT_WIDTH, OUTPUT_WIDTH - OUTPUT_FRACTIONAL_BITS>
m_data_t;
typedef ap_uint<8> config_t;
```

In the top-level code, the information in the header file is included, the static parameterization struct is created using the same constant values used to specify the bit-widths, ensuring the C code and FIR configuration match, and the coefficients are specified. At the top-level, an input configuration, defined in the header file as 8-bit data, is passed into the FIR.

```
#include "fir_top.h"

struct param1 : hls::ip_fir::params_t {
    static const double coeff_vec[total_num_coeff];
    static const unsigned input_length = INPUT_LENGTH;
    static const unsigned output_length = OUTPUT_LENGTH;
    static const unsigned num_coeffs = COEFF_NUM;
    static const unsigned coeff_sets = COEFF_SETS;
};
const double param1::coeff_vec[total_num_coeff] =
    {6,0,-4,-3,5,6,-6,-13,7,44,64,44,7,-13,-6,6,5,-3,-4,0,6};

void dummy_fe(s_data_t in[INPUT_LENGTH], s_data_t out[INPUT_LENGTH],
                config_t* config_in, config_t* config_out)
{
    *config_out = *config_in;
    for(unsigned i = 0; i < INPUT_LENGTH; ++i)
        out[i] = in[i];
}

void dummy_be(m_data_t in[OUTPUT_LENGTH], m_data_t out[OUTPUT_LENGTH])
```

```
{
    for(unsigned i = 0; i < OUTPUT_LENGTH; ++i)
        out[i] = in[i];
}

// DUT
void fir_top(s_data_t in[INPUT_LENGTH],
             m_data_t out[OUTPUT_LENGTH],
             config_t* config)
{

    s_data_t fir_in[INPUT_LENGTH];
    m_data_t fir_out[OUTPUT_LENGTH];
    config_t fir_config;
    // Create struct for config
    static hls::FIR<param1> fir1;

    //================================================
// Dataflow process
    dummy_fe(in, fir_in, config, &fir_config);
    fir1.run(fir_in, fir_out, &fir_config);
    dummy_be(fir_out, out);
    //================================================
}
```

## Using FIR Function with Streaming Interface

The `run()` function with streaming interfaces and without config input is defined in the HLS namespace similar to this:

```
 void run(
 hls::stream<in_data_t> &in_V,
 hls::stream<out_data_t> &out_V);
```

With config input, it is defined similar to this:

```
 void run(
 hls::stream<in_data_t> &in_V,
 hls::stream<out_data_t> &out_V,
 hls::stream<config_t> &config_V);
```

The FIR function is defined in the HLS namespace and can be called as follows:

```
// Create an instance of the FIR
static hls::FIR<STATIC_PARAM> fir1;
// Execute the FIR instance fir1
fir1.run(INPUT_DATA_STREAM, OUTPUT_DATA_STREAM);
```

The STATIC_PARAM is the static parameterization struct that defines most static parameters for the FIR. Both the input and output data are supplied to the function as `hls::stream<>`. These ports on the FIR IP will be implemented as AXI4-Stream ports.

AMD recommends always using the FIR function in a dataflow region using `set_directive_dataflow` or `#pragma HLS dataflow`.

**IMPORTANT!** *The FIR cannot be used in a region which is pipelined. If high-performance operation is required, pipeline the loops or functions before and after the FIR, and use the DATAFLOW pragma or directive on all loops and functions in the region, as shown in the exampe below.*

The multichannel functionality of the FIR is supported through interleaving the data in a single input and single output stream.

- The size of the input stream should be large enough to accommodate all samples: `num_channels * input_length`
- The output stream size should be specified to contain all output samples: `num_channels * output_length`

The following code example demonstrates how the FIR IP function can be used.

```
template<typename data_t, int LENGTH>
void process_fe(data_t in[LENGTH], hls::stream<data_t> &out)
{
    for(unsigned i = 0; i < LENGTH; ++i)
        out.write(in[i]);
}

template<typename data_t, int LENGTH>
void process_be(hls::stream<data_t> &in, data_t out[LENGTH])
{
    for(unsigned i = 0; i < LENGTH; ++i)
        out[i] = in.read();
}

// TOP function
void fir_top(
    data_t in[FIR1_LENGTH],
    data_out_t out[FIR2_LENGTH])
{

    #pragma HLS dataflow

    hls::stream<data_t> fir1_in;
    hls::stream<data_intern_t> fir1_out;
    hls::stream<data_out_t> fir2_out;

    // Create FIR instance
    static hls::FIR<config1> fir1;
    static hls::FIR<config2> fir2;

    //================================================
    // Dataflow process
    process_fe<data_t, FIR1_LENGTH>(in, fir1_in);
    fir1.run(fir1_in, fir1_out);
    fir2.run(fir1_out, fir2_out);
    process_be<data_out_t, FIR2_LENGTH>(fir2_out, out);
    //================================================
}
```

**TIP:** *To avoid bubbles in the execution of the FIR you need to ensure the FIFO depth is sufficient for the data throughput in the dataflow region using the `config_dataflow -fifo_depth` command, and might need to apply loop rewind to the loops in `process_fe` and `process_be` as described in the PIPELINE pragma or directive.*