LogiCORE IP FIR Compiler v6.3 Bit Accurate C Model

User Guide

UG853 (v1.2) April 24, 2012

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/19/11	1.0	Initial Xilinx release.
1/18/12	1.1	MATLAB MEX function added
4/24/12	1.2	Updated for ISE Release 14.1. CORE Generator delivery of C Model.

Table of Contents

XILINX®

Chapter 1

Introduction

The Xilinx® LogiCORE™ IP FIR Compiler v6.3 core bit accurate C model is a self-contained, linkable, shared library that models the functionality of this core with finite precision arithmetic. This model provides a bit accurate representation of the various modes of the FIR Compiler v6.3 core, and it is suitable for inclusion in a larger framework for system-level simulation or core-specific verification.

Features

- Bit accurate with FIR Compiler v6.3 core
- Available for 32-bit and 64-bit Linux platforms
- Available for 32-bit and 64-bit Windows platforms
- Supports all features of the FIR Compiler v6.3 core
- Designed for integration into a larger system model
- Example C code showing how to use the C model functions
- MEX function and class to support MATLAB® software integration

Overview

This user guide provides information about the Xilinx LogiCORE IP FIR Compiler v6.3 bit accurate C model for 32-bit and 64-bit Linux, and 32-bit and 64-bit Windows platforms.

The model consists of a set of C functions that reside in a shared library. Example C code is provided to demonstrate how these functions form the interface to the C model. Full details of this interface are given in Chapter 3 of this document.

The model is bit accurate but not cycle-accurate; it performs exactly the same operations as the core. However, it does not model the core's latency or its interface signals.

Feedback

Xilinx welcomes comments and suggestions about the FIR Compiler v6.3 core and the accompanying documentation.

FIR Compiler v6.3 Bit Accurate C Model and IP Core

For comments or suggestions about the FIR Compiler v6.3 core and bit accurate C model, submit a WebCase from www.xilinx.com/support/clearexpress/websupport.htm. Be sure to include the following information:

- Product name
- Core version number
- Explanation of your comments

Documentation

For comments or suggestions about the FIR Compiler v6.3 core documentation, submit a WebCase from www.xilinx.com/support/clearexpress/websupport.htm. Include the following information:

- Document title
- Document number
- Page number(s) to which your comments refer
- Explanation of your comments

XILINX®

Chapter 2

User Instructions

The C model ZIP files are delivered in the `cmodel` directory of a generated core or can be accessed from the FIR Compiler product page.

Unpacking and Model Contents

There are separate ZIP files containing all the files necessary for use with a specific computing platform. Each ZIP file contains:

- The C model shared library
- Multiple Precision Integers and Rationals (MPIR) [Ref 2] shared libraries and header files.
- The C model header file
- The example code showing customers how to call the C model
- Documentation

Note: The C model uses MPIR libraries, which is provided in the ZIP files. MPIR is an interface-compatible version of the GNU Multiple Precision (GMP) [Ref 3] library, with greater support for Windows platforms. MPIR has been compiled using its GMP compatibility option, so the MPIR library and header file use GMP file names.

Table 2-1: **Example C Model ZIP File Contents - Linux**

File	Description
fir_compiler_v6_3_bitacc_cmodel.h	Header file which defines the C model API
libIp_fir_compiler_v6_3_bitacc_cmodel.so	Model shared object library
libstlport.so.5.1	STL portability library
libgmp.so.7	MPIR library, used by the C model
libgmpxx.so.1	MPIR Class library, used internally by the C model
gmp.h	MPIR header file, used by the C model
run_bitacc_cmodel.c	Example program for calling the C model
README.txt	Release notes
ug853_fir_compiler_cmodel.pdf	This user guide
fir_compiler_v6_3_bitacc_mex.cpp	MATLAB MEX function source
make_fir_compiler_v6_3_mex.m	MATLAB MEX function compilation script

Table 2-1: **Example C Model ZIP File Contents - Linux *(Cont'd)***

File	Description
run_fir_compiler_v6_3_mex.m	MATLAB MEX function example script
@fir_compiler_v6_3_bitacc	MATLAB MEX function class directory

Table 2-2: **Example C Model ZIP File Contents - Windows**

File	Description
fir_compiler_v6_3_bitacc_cmodel.h	Header file which defines the C model API
libIp_fir_compiler_v6_3_bitacc_cmodel.dll	Model dynamically linked library
libIp_fir_compiler_v6_3_bitacc_cmodel.lib	Model .lib file for compiling
stlport.5.1.dll	STL portability library
libgmp.dll	MPIR library, used by the C model
libgmp.lib	MPIR .lib file for compiling
gmp.h	MPIR header file, used by the C model
run_bitacc_cmodel.c	Example program for calling the C model
README.txt	Release notes
ug812_fir_compiler_cmodel.pdf	This user guide
fir_compiler_v6_3_bitacc_mex.cpp	MATLAB MEX function source
make_fir_compiler_v6_3_mex.m	MATLAB MEX function compilation script
run_fir_compiler_v6_3_mex.m	MATLAB MEX function example script
@fir_compiler_v6_3_bitacc	MATLAB MEX function class directory

Installation

Linux

- Unpack the contents of the ZIP file.
- Ensure that the directory where the `libIp_fir_compiler_v6_3_bitacc_cmodel.so`, `libgmp.so.7` and `libgmpxx.so.1` files reside is included in the path of the environment variable LD_LIBRARY_PATH.

Windows

- Unpack the contents of the ZIP file.
- Ensure that the directory where the `libIp_fir_compiler_v6_3_bitacc_cmodel.dll` and `libgmp.dll` files reside is
  a. included in the path of the environment variable PATH or
  b. the directory in which the executable that calls the C model is run.

XILINX®

Chapter 3

FIR Compiler v6.3 Bit Accurate C Model

C Model Interface

The Application Programming Interface (API) of the C model is defined in the header file `fir_compiler_v6_3_bitacc_cmodel.h`. The interface consists of data structures and functions as described in the sections:

- Constants
- Type Definitions
- Dynamic Arrays
- Structures
- Functions

An example C file, `run_bitacc_cmodel.c`, is included with the C libraries. This file demonstrates how to call the C model.

Constants

Table 3-1: **Constants**

Name	Value
Error codes	
XIP_STATUS_OK	0
XIP_STATUS_ERROR	1
Filter Types	
XIP_FIR_SINGLE_RATE	0
XIP_FIR_INTERPOLATION	1
XIP_FIR_DECIMATION	2
XIP_FIR_HILBERT	3
XIP_FIR_INTERPOLATED	4
Rate Change	
XIP_FIR_INTEGER_RATE	0
XIP_FIR_FRACTIONAL_RATE	1

Table 3-1: **Constants *(Cont'd)***

Name	Value
Channel Sequence	
XIP_FIR_BASIC_CHAN_SEQ	0
XIP_FIR_ADVANCED_CHAN_SEQ	1
Quantization	
XIP_FIR_INTEGER_COEFF	0
XIP_FIR_QUANTIZED_ONLY	1
XIP_FIR_MAXIMIZE_DYNAMIC_RANGE	2
Output Rounding	
XIP_FIR_FULL_PRECISION	0
XIP_FIR_TRUNCATE_LSBS	1
XIP_FIR_SYMMETRIC_ZERO	2
XIP_FIR_SYMMETRIC_INF	3
XIP_FIR_CONVERGENT_EVEN	4
XIP_FIR_CONVERGENT_ODD	5
XIP_FIR_NON_SYMMETRIC_DOWN	6
XIP_FIR_NON_SYMMETRIC_UP	7
Configuration Method	
XIP_FIR_CONFIG_SINGLE	0
XIP_FIR_CONFIG_BY_CHANNEL	1

Type Definitions

Table 3-2: **Type Definitions**

Field Name	Description
General types	
typedef double xip_real	Scalar type alias
typedef unsigned integer xip_uint	Integer type alias
typedef struct xip_complex	Complex data type
typedef mpz_t xip_mpz	MPIR [Ref 2] integer type alias
typedef struct xip_mpz_complex	Complex type based on xip_mpz
typedef int xip_status	Result code from API functions
typedef struct _xip_fir_v6_3 xip_fir_v6_3	Handle type to refer to an instance of the model
typdef enum xip_fir_v6_3_pattern	Defines enumerated values for all the advance channel patterns

Table 3-2: **Type Definitions *(Cont'd)***

Field Name	Description
Handler function signatures	
typedef void (*xip_msg_handler)(void* handle, int error,const char* msg)	Interface to a message handler function
typedef void (*xip_array_real_handler) (const xip_array_real* data, void* handle, void* opt_arg)	Interface to data packet handler function
Structures	
typedef struct xip_fir_v6_3_config	Model configuration structure, Table 3-4
typedef struct xip_fir_v6_3_cnfg_packet	Configuration packet structure, Table 3-5
typedef struct xip_fir_v6_3_rld_packet	Reload packet structure, Table 3-6
Array types	
typedef struct xip_array_uint	See Dynamic Arrays
typedef struct xip_array_real	
typedef struct xip_array_complex	
typedef struct xip_array_mpz	

Dynamic Arrays

The C model represents input and output data using multi-dimensional dynamic arrays. The `xip_array_<type>` structure is used to specify a multi-dimensional dynamic array containing elements of type `xip_<type>`. A number of utility functions are provided that allow creation, allocation and destruction of array instances.

For each array type, the DECLARE_XIP_ARRAY(<type>) macro can be used to declare the structure and utility function prototypes. The C model header already contains declarations for the following array types:

- `xip_array_real` for arrays of `xip_real`
- `xip_array_complex` for arrays of `xip_complex`
- `xip_array_uint` for arrays of `xip_uint`
- `xip_array_mpz` for arrays of `xip_mpz`
- `xip_array_mpz_complex` for arrays of `xip_mpz_complex`

The utility functions for each array type can be defined using the DEFINE_XIP_ARRAY(<type>) macro. The utility function must be defined somewhere within user code before the functions can be used; see the `run_bitacc_cmodel.c` file for examples.

Further utility functions, specific to the FIR Compiler C Model, can be declared and defined using the DECLARE_FIR_XIP_ARRAY(<type>) and DEFINE_FIR_XIP_ARRAY(<type>) macros. The C model header already contains declarations for the following array types:

- `xip_array_real`
- `xip_array_complex`

- xip_array_mpz
- xip_array_mpz_complex

Structure

The xip_array_<type> structure is used to specify a multi-dimensional array of data with type <type>. The content is summarized in Table 3-3.

Table 3-3: **xip_array_<type>**

Field Name	Type	Description
data	xip_<type>*	Pointer to array of data
data_size	size_t	Current number of elements in the data array
data_capacity	size_t	Max number of elements in the data array
dim	size_t*	Pointer to dimension array
dim_size	size_t	Current number of elements in the dimension array, dim
dim_capacity	size_t	Max number of elements in dim array
owner	unsigned int	Ownership control. A value of 0 indicates that the structure and associated memory (for the data and dim fields) is allocated and owned by the xip_array_<type>_* functions, in which case the model can automatically resize arrays as required. Any other value indicates that the memory is owned by the user, and the model must report an error if an array is of insufficient capacity.

This data structure is defined for types:

- xip_real
- xip_complex
- xip_uint
- xip_mpz
- xip_mpz_complex

General Functions

Create Array

```
xip_array_<type>*
xip_array_<type>_create();
```

This function allocates and initializes an empty array for holding values of type <type>. The function returns a pointer to the created structure, or null if the structure cannot be created. The structure fields are all initialized to zero indicating an empty array, with ownership associated with the xip_array_<type>_* functions.

Reserve Data Memory

```
xip_status
xip_array_<type>_reserve_data(
    xip_array_<type>* p,
    size_t max_nels
);
```

This function ensures that array `p` has sufficient space to store up to `max_nels` elements of data. If the current `data_capacity` is insufficient and the current owner is zero, the function attempts to allocate or reallocate space to meet the request. The function returns XIP_STATUS_OK if the array capacity is now sufficient or XIP_STATUS_ERROR if memory could not be allocated.

Note: This function does not change the data or dimensions held within the array in any way; the contents of the array after calling the function are equivalent to the contents before calling the function, even if memory is reallocated. Also, this function never reduces memory allocation; use xip_array_<type>_destroy to release memory.

Reserve Dimension Memory

```
xip_status
xip_array_<type>_reserve_dim(
    xip_array_<type>* p,
    size_t max_nels
);
```

This function ensures that array `p` has sufficient space to store up to `max_ndims` dimensions. If the current `dim_capacity` is insufficient and the current owner is zero, the function attempts to allocate or reallocate space to meet the request. The function returns XIP_STATUS_OK if the array capacity is now sufficient or XIP_STATUS_ERROR if memory could not be allocated.

Note: This function does not change the data or dimensions held within the array in any way; the contents of the array after calling the function are equivalent to the contents before calling the function, even if memory is reallocated. Also, this function never reduces memory allocation; use xip_array_<type>_destroy to release memory.

Destroy Array

```
xip_array_<type>*
xip_array_<type>_create(
    xip_array_<type>* p
);
```

This function attempts to release all memory associated with array `p`. If the owner field is zero, the function releases the memory associated with `data`, `dim` and `p`, and returns null indicating success. If owner is non-zero the function returns `p`, indicating failure.

FIR Compiler Specific Functions

The following functions have been added to aid the use of the array types with the FIR Compiler C Model and, specifically, the advanced channel patterns.

Set Channel

```
xip_status
xip_array_<type>_set_chan(
    xip_array_<type>* p
    const <type> value,
    size_t path,
    size_t chan,
    size_t index
    xip_fir_v6_3_pattern pattern
);
```

This function maps an array index for one channel, specified by `path` and `chan`, onto the 3-D structure of `xip_array_<type>` structure expected by the model's `xip_fir_v6_3_data_send` (see Send DATA Packet, page 20) and `xip_fir_v6_3_data_get` (see Get DATA Packet, page 22) functions.

This function should be particularly useful for the Advanced Interleaved Channels feature; see [Ref 1] for details of this feature, where locations in the input array are remapped to duplicate entries for some channels. Figure 3-2 illustrates this requirement.

`pattern` should be set to `P_BASIC` for a Basic Interleaved Channel model configuration and set to the current pattern ID for an Advanced Interleaved Channel model configuration. See [Ref 1] for a list of all the supported patterns and see `fir_compiler_v6_3_bitacc_cmodel.h` for the enumerated pattern IDs.

Note: If the value of index exceeds the current capacity (data_capacity) of p then the function issues a XIP_STATUS_ERROR. If the value of index exceeds number of elements (data_size) of p then the function sets the new size of the array.

Get Channel

```
xip_status
xip_array_<type>_get_chan(
    xip_array_<type>* p
<type>* value,
    size_t path,
    size_t chan,
    size_t index
    xip_fir_v6_3_pattern pattern
);
```

This function is the reciprocal of the `xip_array_<type>_set_chan` function and extracts an individual channel's value for a given index, path and channel. The function issues an XIP_STATUS_ERROR if the index exceeds the array capacity or size.

Structures

The `xip_fir_v6_3_config` structure contains all parameters that affect the filter configuration. Most are duplicates of the core's XCO parameters. The model's filter coefficients are provided using the `coeff` field and are quantized (if specified) in the same manner as by the core GUI.

Table 3-4: **xip_fir_v6_3_config**

Field Name	Type	Description
name	const char*	
filter_type	unsigned int	Select from: XIP_FIR_SINGLE_RATE XIP_FIR_INTERPOLATION XIP_FIR_DECIMATION XIP_FIR_HILBERT XIP_FIR_INTERPOLATED
rate_change	unsigned int	Select from: XIP_FIR_INTEGER_RATE XIP_FIR_FRACTIONAL_RATE
interp_rate	unsigned int	Specifies the interpolation (or up-sampling) factor

Table 3-4: **xip_fir_v6_3_config *(Cont'd)***

Field Name	Type	Description
decim_rate	unsigned int	Specifies the decimation (or down-sampling) factor
zero_pack_factor	unsigned int	Specifies the zero packing factor for Interpolated filters
coeff	const double*	Pointer to coefficient array
coeff_padding	unsigned int	Specifies the amount of zero padding added to the front of the filter. ***Note:*** The core GUI reports this value for a given core configuration.
num_coeffs	unsigned int	Specifies the number of coefficients in one filter
coeff_sets	unsigned int	Specifies the number of coefficient sets in the coeff array
reloadable	unsigned int	Specifies if the coefficients are reloadable; 0 = No, 1 = Yes
is_halfband	unsigned int	Specifies if halfband coefficients have been specified; 0 = No, 1 = Yes
quantization	unsigned int	Select from: XIP_FIR_INTEGER_COEFF XIP_FIR_QUANTIZED_ONLY XIP_FIR_MAXIMIZE_DYNAMIC_RANGE
coeff_width	unsigned int	The model uses these parameters, if requested, to quantize the supplied coefficients
coeff_fract_width	unsigned int	
chan_seq	unsigned int	Select from: XIP_FIR_BASIC_CHAN_SEQ XIP_FIR_ADVANCED_CHAN_SEQ
num_channels	unsigned int	Specifies the number of data channels supported
init_pattern	xip_fir_v6_3_pattern	Specifies the initial channel pattern used by the model when Advanced Interleaved Channels have been selected
num_paths	unsigned int	Specifies the number of data paths supported
data_width	unsigned int	The model uses these parameters to quantize the model's input samples
data_fract_width	unsigned int	

Table 3-4: **xip_fir_v6_3_config *(Cont'd)***

Field Name	Type	Description
output_rounding_mode	unsigned int	Select from: XIP_FIR_FULL_PRECISION XIP_FIR_TRUNCATE_LSBS XIP_FIR_SYMMETRIC_ZERO XIP_FIR_SYMMETRIC_INF XIP_FIR_CONVERGENT_EVEN XIP_FIR_CONVERGENT_ODD XIP_FIR_NON_SYMMETRIC_DOWN XIP_FIR_NON_SYMMETRIC_UP
output_width	unsigned int	Ignored when XIP_FIR_FULL_PRECISION
output_fract_width	unsigned int	**READ ONLY** Provides the number of fractional bits present in the output word
config_method	unsigned int	Select from: XIP_FIR_CONFIG_SINGLE XIP_FIR_CONFIG_BY_CHANNEL

The `xip_fir_v6_3_cnfg_packet` structure is supplied to the `xip_fir_v6_3_config_send` function (see Send CONFIG Packet, page 20) to update the channel pattern and coefficient set used by the model.

Table 3-5: **xip_fir_v6_3_cnfg_packet**

Field Name	Type	Description
chanpat	xip_fir_v6_3_pattern	Specifies the Advanced Interleaved Channel pattern to be used
fsel	xip_array_uint*	Filter set to use, 1-D array; specifies one value for all channels (XIP_FIR_CONFIG_SINGLE) or individually for each interleaved channel (XIP_FIR_CONFIG_BY_CHANNEL)

The `xip_fir_v6_3_rld_packet` structure is supplied to the `xip_fir_v6_3_reload_send` function (see Send RELOAD Packet, page 20) to update a given coefficient set with new filter coefficients. As with the core, a configuration packet must be processed by the model to apply any pending reload packets.

Table 3-6: **xip_fir_v6_3_rld_packet**

Field Name	Type	Description
fsel	Int	Filter set to reload
coeff	xip_array_real*	Pointer to an array containing the new coefficients to be loaded, 1-D array.

Functions

Model Configuration Functions

Get Version

```
const char* xip_fir_v6_3_get_version(void);
```

The function returns a string describing the version of the model.

Get Default Configuration

```
xip_status
xip_fir_v6_3_get_default_config(
    xip_fir_v6_3_config* config
)
```

This function populates the `xip_fir_v6_3_config` configuration structure pointed to by `config` with the default configuration of the FIR Compiler v6.3 core.

Create Model Object

```
xip_fir_v6_3
xip_fir_v6_3_create(
    const xip_fir_v6_3_config* config,
    xip_msg_handler msg_handler,
    void* msg_handle
)
```

This function creates a new model instance, based on the configuration data pointed to by `config`.

The `msg_handler` argument is a pointer to a function taking three arguments as previously defined in Type Definitions. This function pointer is retained by the model object and is called whenever the model wishes to issue a note, warning or error message. Its arguments are:

1. A generic pointer (void*). This is always the value that was passed in as the `msg_handle` argument to the create function.
2. An integer (int) indicating whether the message is an error (1) or a note or warning (0).
3. The message string itself.

If the `handler` argument is a null pointer, then the C model outputs no messages at all. Using this mechanism, you may choose whether to output messages to the console, log them to a file or ignore them completely.

The `create` function returns a pointer to the newly created object. If the object cannot be created, then a diagnostic error message is emitted using the supplied handler function (if any) and a null pointer is returned.

If the data and coefficient widths, number of coefficients and output precision result in an output precision greater than supported by the double (`xip_real`) data type then the model uses the `mpz_t` data type [Ref 2] (`xip_mpz`) and issues a warning indicating this requirement when this function is executed.

Get Model Configuration

```
xip_status
xip_fir_v6_3_get_config (
    xip_fir_v6_3* model,
    xip_fir_v6_3_config* config
)
```

This function returns the full configuration of the model. The function is intended to be primarily used to determine the output width and output fractional width of the model.

Note: The coeff pointer of the returned xip_fir_v6_3_config structure is set to NULL.

Reset Model Object

```
xip_status
xip_fir_v6_3_reset(
    xip_fir_v6_3* model
);
```

This function resets in the internal state of the FIR Compiler model object pointed to by `model`. A reset causes all data and pending configuration packets to be cleared from the model. As per the core, any pending reload packets are retained.

Destroy Model Object

```
xip_status
xip_fir_v6_3_destroy(
    xip_fir_v6_3* model
);
```

This function deallocates the model object pointed to by `model`.

Any system resources or memory belonging to the model object are released on return from this function. The model object becomes undefined, and any further attempt to use it is an error.

Set Output Data Array

```
xip_status
xip_fir_v6_3_set_data_sink(
    xip_fir_v6_3* model,
    xip_array_real* data,
    xip_array_complex* cmplx_data
);
xip_status
xip_fir_v6_3_set_data_sink_mpz(
    xip_fir_v6_3* model,
    xip_array_mpz* data,
    xip_array_mpz_complex* cmplx_data
);
```

This function registers an array (the data sink), pointed to by `data` or `cmplx_data`, to push the generated filter output when the `xip_fir_v6_3_data_send` function is called. Only `data` or `cmplx_data` may be set, the other should be set to NULL (or 0).

If the data sink is undefined the filter output must be explicitly pulled using the `xip_fir_v6_3_data_get` function.

The array is automatically sized by the model given the size of the input request. The owner field of `xip_array_<type>` is ignored and forced to 0.

Note: The complex data sink is intended for the Hilbert filter type but is populated for other filter types with im set to 0.

Set Data Handler

```
xip_status
xip_fir_v6_3_set_data_handler(
    xip_fir_v6_3* model,
    xip_array_real_handler data_handler,
    void* handle,
    void* opt_arg
);
```

This function registers a data handler call back function that is called when the output data array is filled following a call to `xip_fir_v6_3_data_send`. The FIR Compiler C model API contains a function, `xip_fir_v6_3_data_send_handler` (see Send DATA Packet, page 20), to send data to an instance of the model whose signature matches that of a data handler.

The intention of this facility is to enable multiple instances of the model to be chained together such that only the first and last instance of the chain need to be directly controlled using the `xip_fir_v6_3_data_send` and `xip_fir_v6_3_data_get` functions.

The model only supports data handlers for output data arrays of type `xip_array_real` and the value passed to the (`*xip_array_real_handler`) function for the `data` argument is the value set by the `xip_fir_v6_3_set_data_sink` function. See Type Definitions for details of the data handler function signature. Its arguments are:

1. `data`: A pointer to the `xip_array_real` type containing the data to be processed. The array registered by the `xip_fir_v6_3_set_data_sink` function.
2. `handle`: A void pointer used to point to the next model instance in the filter chain.
3. `opt_arg`: An extra generic argument not currently used by the FIR Compiler C model.

Calculate Output Size

```
xip_status
xip_fir_v6_3_data_calc_size(
    xip_fir_v6_3* model,
    const xip_array_real* data_in,
    xip_array_real* data_out,
    xip_array_complex* cmplx_data_out
)
xip_status
xip_fir_v6_3_data_calc_size_mpz(
    xip_fir_v6_3* model,
    const xip_array_real* data_in,
    xip_array_mpz* data_out,
    xip_array_mpz_complex cmplx_data_out
)
```

This function calculates the size of an output packet/array given the size of the supplied input packet/array.

The `data_out` or `cmplx_data_out` array dimensions are modified to reflect the size of output the model produces, given the `data_in` array. The array dimensions, `dim` and `data_size` element are updated but the function does not allocate more space. Ensure that the correct amount of space is allocated for the `data` element of the array.

Note: Only one of data_out or cmplx_data_out can be set; the other should be set to NULL (or 0).

Model Operation Functions

Send CONFIG Packet

```
xip_status
xip_fir_v6_3_config_send(
    xip_fir_v6_3* model,
    const xip_fir_v6_3_cnfg_packet* cnfg_packet
)
```

This function passes a configuration packet, pointed to by `cnfg_packet` (see Table 3-5, page 16), to the model. The model implements an internal fifo/queue. A configuration packet is consumed from the queue for every data packet processed, that is, every call to `xip_fir_v6_3_data_send`.

Note: If the fsel field of the cnfg_packet is not sized correctly the function returns XIP_STATUS_ERROR.

Send RELOAD Packet

```
xip_status
xip_fir_v6_3_reload_send(
    xip_fir_v6_3* model,
    const xip_fir_v6_3_rld_packet* rld_packet
)
```

This function passes a reload packet, pointed to by `rld_packet` (see Table 3-6, page 16), to the model.

Note: If the coeff field of the rld_packet is not sized correctly the function returns XIP_STATUS_ERROR.

Send DATA Packet

```
xip_status
xip_fir_v6_3_data_send(
    xip_fir_v6_3* model,
    const xip_array_real* data
);
void
xip_fir_v6_3_data_send_handler(
    const xip_array_real* data,
    void* model,
    void* dummy
);
```

This function sends a new data packet, pointed to by `data`, to the model for processing.

The second version of the function, `xip_fir_v6_3_data_send_handler`, is supplied to be used as a (`*xip_array_real_handler`) call back function, see Set Data Handler, page 19 for further details.

Input data is provided using the `xip_array_real` structure pointed to by `data` and is expected to be sized:
Number of paths x *Number of interleaved channels* x *number of input vectors*.

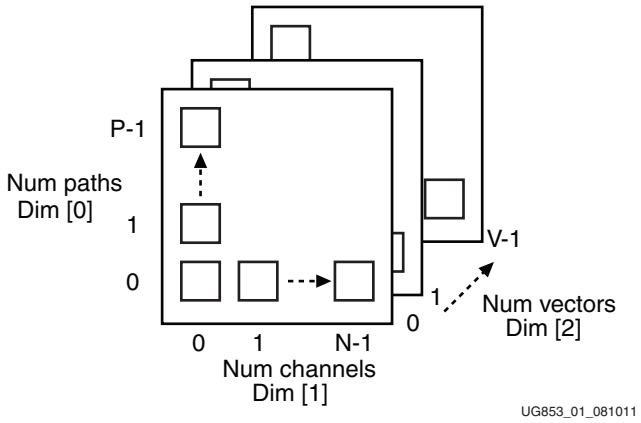

Figure 3-1: **Input and Output Data Packet Structure**

The 3-D structure illustrated in Figure 3-1 is translated to the 1-D array of the `xip_array_<type>` `data` element in the order; Paths, Channels, Vectors. The helper functions, `xip_array_<type>_set_chan` (Set Channel, page 13) and `xip_array_<type>_get_chan` (Get Channel, page 14) implement this translation.

The Advanced Channel implementation requires redundant channel positions to be remapped to higher rate channels. The helper functions, `xip_array_<type>_set_chan` (Set Channel, page 13) and `xip_array_<type>_get_chan` (Get Channel, page 14), simplify referencing each channel by presenting a flat index for each channel.

Figure 3-1 illustrates the remapping for three different pattern sequences:

- P4_4 (4 channels x 1/4fs);
- P4_3 (1 channel x 1/2fs and 2 channels x 1/4fs);
- P4_2 (1 channel x 3/4fs and 1 channel x 1/4fs).

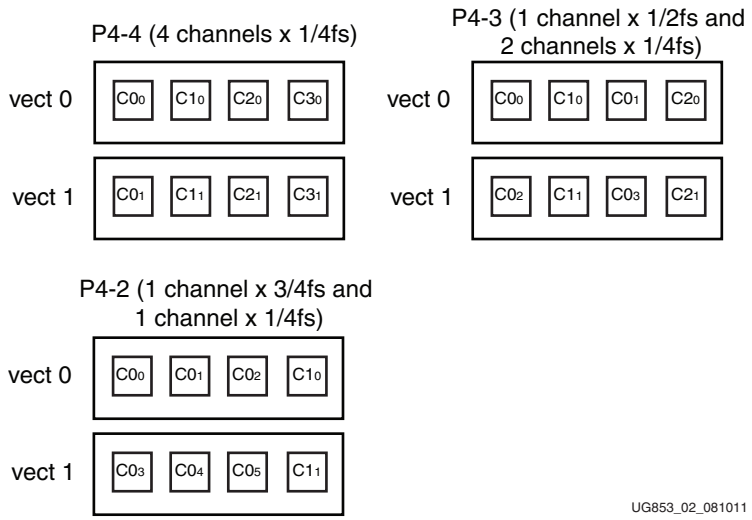

Figure 3-2: **Advanced Channel Pattern Data Packet Remapping**

Get DATA Packet

```
xip_status
xip_fir_v6_3_data_get(
    xip_fir_v6_3* model,
    xip_array_real* data,
    xip_array_complex* cmplx_data
);
xip_status
xip_fir_v6_3_data_get_mpz(
    xip_fir_v6_3* model,
    xip_array_mpz* data,
    xip_array_mpz_complex* cmplx_data
);
```

This function retrieves a filtered data packet from the model into the `xip_array_<type>` pointed to by `data` or `cmplx_data`. Only one of `data` or `cmplx_data` maybe set, the other should be set to NULL (or 0).

The size of the array `dim[2]` (Figure 3-1) determines how much data is fetched from the model. If the request is greater than available, then the array size is reduced to reflect this. The model does not modify the amount of space allocated. Both versions of the functions maybe used regardless of the model's internal implementation method. If double data (`xip_real`) is requested when `mpz_t (xip_mpz)` has been used internally by the model the output data is truncated, as per the `mpz_get_d` function (see [Ref 3]).

`mpz_t (xip_mpz)` is an integer type so the model scales the input data and coefficients by their specified fractional width to use an integer representation. The output is also supplied as an integer value when `mpz_t` is requested. To correctly interpret the `mpz_t` output the model configuration, returned by the `xip_fir_v6_3_get_config` function (see Get Model Configuration, page 17), should be interrogated to determine the output fractional width.

Compiling

Compilation of user code requires access to the `fir_compiler_v6_3_bitacc_cmodel.h` header file and the header file of the MPIR [Ref 2] dependent library, `gmp.h`. The header files should be copied to a location where they are available to the compiler. Depending on the location chosen, the 'include' search path of the compiler might need to be modified.

The `fir_compiler_v6_3_bitacc_cmodel.h` header file includes the MPIR header file, so these do not need to be explicitly included in source code that uses the C model. When compiling on Windows, the symbol `NT` must be defined, either by a compiler option, or in user source code before the `fir_compiler_v6_3_bitacc_cmodel.h` header file is included.

Linking

To use the C model the user executable must be linked against the correct libraries for the target platform.

Note: The C model uses the MPIR library. Pre-compiled MPIR libraries are provided with the C model. It is also possible to use GMP or MPIR, libraries from other sources, for example, compiled from source code. For details, see Dependent Libraries.

Linux

The executable must be linked against the following shared object libraries:

- `libgmp.so.7`
- `libIp_fir_compiler_v6_3_bitacc_cmodel.so`

Using GCC, linking is typically achieved by adding the following command line options:

```
-L. -lgmp -lIp_fir_compiler_v6_3_bitacc_cmodel
```

This assumes the shared object libraries are in the current directory. If this is not the case, the `-L.` option should be changed to specify the library search path to use.

Using GCC, the provided example program `run_bitacc_cmodel.c` can be compiled and linked using the following command:

```
gcc run_bitacc_cmodel.c -o run_bitacc_cmodel -I. -L. -lgmp
-lIp_fir_compiler_v6_3_bitacc_cmodel
```

Note: The C model dynamically links to gmpxx.so.1 and therefore must be visible to the model while running.

Windows

The executable must be linked against the following dynamic link libraries:

- `libgmp.dll`
- `libIp_fir_compiler_v6_3_bitacc_cmodel.dll`

Depending on the compiler, the import libraries might also be required:

- `libgmp.lib`
- `libIp_fir_compiler_v6_3_bitacc_cmodel.lib`

Using Microsoft Visual Studio, linking is typically achieved by adding the import libraries to the Additional Dependencies entry under the Linker section of Project Properties.

Example

The `run_bitacc_cmodel.c` file contains example code to show the basic operation of the C model in various configurations.

MATLAB Interface

A MEX function and MATLAB® software class are provided to simplify the integration with MATLAB. The MEX function provides a low-level wrapper around the underlying C model, while the class file provides a convenient interface to the MEX function.

Compiling

Source code for a MATLAB MEX function is provided. This can be compiled within MATLAB by changing to the directory that contains the code and running the `make_fir_compiler_v6_3_bitacc_mex.m` script.

Installation

To use the MEX function, the compiled MEX function must be present on the MATLAB search path. This can be achieved in either of two ways:

1. Add the directory where the compiled MEX function is located to the MATLAB search path (see the MATLAB addpath function)

   or

2. Copy the files to a location already on the MATLAB search path.

As with all uses of the C model, the correct C model libraries also need to be present on the platform library search path (that is, PATH or LD_LIBRARY_PATH).

MATLAB Class Interface

The `@fir_compiler_v6_3_bitacc` class handles the create/destroy semantics on the C model. The class provides objects for each of the data, configuration and control structures, defined for the C model and previously described in Structures, page 14. All structure elements have MATLAB type double. MATLAB arrays are used with the mapping of types as in Table 3-7.

Table 3-7: **MATLAB to C Model Type Mapping**

C Model Type	MATLAB Type
xip_uint32	uint32
xip_complex	complex double
xip_real	double

The class provides the methods:

Constructor

```
[model]=fir_compiler_v6_3_bitacc
[model]=fir_compiler_v6_3_bitacc(config)
[model]=fir_compiler_v6_3_bitacc(field, value [, field,value]*)
```

Note: * indicates an optional parameter.

The first version of the function call constructs a model object using the default configuration.

The second version constructs a model object from a structure that specifies the configuration parameter values to use.

The third version is the same as the second, but allows the configuration to be specified as a series of (parameter name, value) pairs rather than a single structure.

The names and valid values of configuration parameters are identical to those previously described for the C model in Structures, page 14.

The MATLAB configuration structure can contain an additional element, `PersistentMemory`. When the element is set to `true` the model's internal data memory state is retained following a call to the Filter function. Otherwise, the model is Reset after the filtered data is returned. `PersistentMemory` is set to `false` by default.

Get Version

```
[version]=get_version(model)
```

This method returns the version string of the C model library used.

Get Configuration

```
[config]=get_configuration(model)
```

This method returns the current parameters structure of a model object. If the model object is empty, the method returns the default configuration. If the model object has been created, the method returns the configuration parameters that were used to create it.

Reset

```
[model]=reset(model)
```

This function resets the model, see Reset Model Object, page 18 for further details.

Send CONFIG Packet

```
[model]=config_send(model,cnfg_packet)
```

This function passes a configuration packet (see Table 3-5, page 16), to the model. See Send CONFIG Packet, page 20 for further details.

Send RELOAD Packet

```
[model]=reload_send(model,rld_packet)
```

This function passes a reload packet (see Table 3-6, page 16), to the model. See Send RELOAD Packet, page 20 for further details.

Filter

```
[model,data_out]=filter(model,data_in)
```

This function passes a MATLAB double array to the model and returns the filtered output. `data_in` can be a 1, 2 or 3 dimensional array:

- A 1-D array is only supported by a single channel, single path filter configuration.
- A 2-D array is only supported by a multichannel, single path filter configuration.
- All filter configurations support a 3-D array.

See Send DATA Packet, page 20 and Figure 3-1, page 21 for further details on the data array structure.

Example

The `run_fir_compiler_v6_3_bitacc_mex.m` file contains a MATLAB script with an example of how to run the C model using the MEX function.

To run the sample script:

1. Compile the MEX function with the `make_fir_compiler_v6_3_bitacc_mex.m` script (see Compiling, page 23).
2. Install the MEX function (see Installation, page 24).
3. Execute the `run_fir_compiler_v6_3_bitacc_mex.m` script.

Dependent Libraries

The C model uses MPIR libraries. Pre-compiled MPIR libraries are provided with the C model, using the following versions of the libraries:

- MPIR 2.2.1

Because MPIR is a compatible alternative to GMP, the GMP library can be used in place of MPIR. It is possible to use GMP or MPIR libraries from other sources, for example, compiled from source code.

GMP and MPIR in particular contain many low level optimizations for specific processors. The libraries provided are compiled for a generic processor on each platform, not using optimized processor-specific code. These libraries work on any processor, but run more slowly than libraries compiled to use optimized processor-specific code. For the fastest performance, compile libraries from source on the machine on which you run the executables.

Source code and compilation scripts are provided for the version of MPIR that were used to compile the provided libraries. Source code and compilation scripts for any version of the libraries can be obtained from the GMP [Ref 3] and MPIR [Ref 2] web sites.

Note: If compiling MPIR using its configure script (for example, on Linux platforms), use the --enable-gmpcompat option when running the configure script. This generates a libgmp.so library and a gmp.h header file that provide full compatibility with the GMP library.

XILINX®

Appendix A

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx Support website at:

www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

www.xilinx.com/company/terms.htm

References

1. LogiCORE IP FIR Compiler data sheet DS795
2. The Multiple Precision Integers and Rationals (MPIR) Library: www.mpir.org/
3. The GNU Multiple Precision Arithmetic (GMP) Library: gmplib.org/

Additional Core Resources

For detailed information and updates about the FIR Compiler v6.3 core, see the following document:

- FIR Compiler v6.3 Release Notes

Technical Support

For technical support, go to www.xilinx.com/support. Questions are routed to a team with expertise using the FIR Compiler v6.3 core.

Xilinx provides technical support for use of this product as described in *FIR Compiler v6.3 Bit Accurate C Model User Guide (UG853)* and the *LogiCORE IP FIR Compiler v6.3 Product Specification (DS795)*. Xilinx cannot guarantee functionality or support of this product for designs that do not follow these guidelines.