LogiCORE™ IP DUC/DDC Compiler v1.0 Bit Accurate C Model

User Guide

UG742 (v1.0) July 23, 2010

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/23/10	1.0	Initial Xilinx release.

Table of Contents

About This Guide

This user guide provides information about the Xilinx LogiCORE™ IP DUC/DDC Compiler v1.0 bit-accurate C model for 32-bit and 64-bit Linux platforms and 32-bit and 64-bit Windows platforms.

Guide Contents

This manual contains the following chapters:

- Chapter 1, "Introduction," contains an overview of the DUC/DDC Compiler v1.0 bit-accurate C model.
- Chapter 2, "User Instructions," provides information on the C model contents and installation.
- Chapter 3, "DUC/DDC Bit-Accurate C Model," contains information on using the interface and compiling with the C model.

Additional Resources

To find additional documentation, see the Xilinx website at:

www.xilinx.com/support/documentation/index.htm.

To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support WebCase, see the Xilinx website at:

www.xilinx.com/support/mysupport.htm.

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *design_name*
Helvetica bold	Commands that you select from a menu	**File → Open**
	Keyboard shortcuts	**Ctrl+C**
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *design_name*
	References to other manuals	See the *User Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Dark Shading	Items that are not supported or reserved	This feature is not supported
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **`bus[7:0]`**, they are required.	**`ngdbuild`** [*option_name*] *design_name*
Braces { }	A list of items from which you must choose one or more	**`lowpwr =`**{**`on`**|**`off`**}
Vertical bar |	Separates items in a list of choices	**`lowpwr =`**{**`on`**|**`off`**}
Angle brackets < >	User-defined variable or in code samples	<directory name>

Vertical ellipsis
.
.
.
	Repetitive material that has been omitted	`IOB #1: Name = QOUT'`
		`IOB #2: Name = CLKIN'`
		.
		.
		.

Horizontal ellipsis ...	Repetitive material that has been omitted	**`allow block`** *block_name loc1 loc2 ... locn;*

Convention	Meaning or Use	Example
Notations	The prefix '0x' or the suffix 'h' indicate hexadecimal notation	A read of address 0x00112975 returned 45524943h.
	An '_n' means the signal is active low	**usr_teof_n** is active low.

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
Blue, underlined text	Hyperlink to a website (URL)	Go to www.xilinx.com for the latest speed files.

XILINX®

Chapter 1

Introduction

The Xilinx LogiCORE™ IP DUC/DDC Compiler v1.0 core has a bit accurate C model designed for system modeling. This allows you to model the effect of different core parameters on performance. For purposes of abbreviation, DUC/DDC compiler v1.0 is used interchangeably with DUC/DDC throughout this document.

Features

- Bit-accurate to DUC/DDC Compiler v1.0 core
- Available for 32-bit and 64-bit Linux platforms
- Available for 32-bit and 64-bit Windows platforms
- Provides sample-level timing accuracy
- Designed for integration into a larger system model
- Example C code provided 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 DUC/DDC Compiler v1.0 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, "DUC/DDC Bit-Accurate C Model."

The model is bit-accurate but not cycle-accurate. It produces exactly the same output data as the core on a sample-by-sample basis. However, it does not model the core clock cycle latency, its interface signals or its register map.

The DUC/DDC core contains a number of internal FIR filter stages. Each filter stage has a sample history pipeline that contributes to the overall sample latency of the core. The C model does not attempt to model the contents of these pipelines at initialization and after reset events. Bit-accuracy is only guaranteed once these pipelines have flushed, which takes a number of samples equal to the total sample latency.

Additional Core Resources

For detailed information on the DUC/DDC Compiler v1.0 core, see the following documents:

- *DUC/DDC Compiler v1.0 Data Sheet* (DS766)
- *DUC/DDC Compiler v1.0 Release Notes*

Technical Support

For technical support, go to www.xilinx.com/support.

Xilinx provides technical support for use of this product as described in the *DUC/DDC Compiler v1.0 Bit Accurate C Model User Guide* (UG742) and the *DUC/DDC Compiler v1.0 Data Sheet* (DS766). Xilinx cannot guarantee functionality or support of this product for designs that do not follow these guidelines.

Be sure to include the following information:

- Product name
- C model version number
- Which operating system and compiler you are using
- Explanation of the problem observed
- Instructions on how to reproduce the problem
- Any source code and/or stimulus data files required to reproduce the problem

Feedback

Xilinx welcomes comments and suggestions about the DUC/DDC Compiler v1.0 core and C model documentation.

Core and C Model

For comments or suggestions about the DUC/DDC Compiler core or C model, submit a WebCase at www.xilinx.com/support/clearexpress/websupport.htm.

Be sure to include the following information:

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

Documents

For comments or suggestions about the DUC/DDC Compiler v1.0 documentation, submit WebCase at www.xilinx.com/support/clearexpress/websupport.htm.

Be sure to 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

Unpacking and Model Contents

Unzipping the DUC/DDC C model ZIP file produces the directory structure and files shown in Table 2-1.

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

File	Description
README.txt	Release notes
duc_ddc_compiler_bitacc_cmodel_ug742.pdf	This file
duc_ddc_compiler_v1_0_bitacc_cmodel.h	Model header file
libIp_duc_ddc_compiler_v1_0_bitacc_cmodel.so	Model shared object library (Linux platforms only)
libstlport.so.5.1	STL portability library (Linux platforms only)
libIp_duc_ddc_compiler_v1_0_bitacc_cmodel.dll	Model dynamically linked library (Windows platforms only)
libIp_duc_ddc_compiler_v1_0_bitacc_cmodel.lib	Model library file for static linking (Windows platforms only)
stlport.5.1.dll	STL portability library (Windows platforms only)
duc_ddc_compiler_v1_0_bitacc_mex.cc	MATLAB® MEX function source
@duc_ddc_compiler_v1_0_bitacc	MATLAB class directory
make_mex.m	MEX function compilation script
test_mex.m	Example MATLAB source file
test_max_mat.m	Test vectors for example MATLAB source file
run_bitacc_cmodel.c	Example source file
ducddc_testcases.dat	Test vectors for example source file

Installation

- On Linux platforms, ensure that the directory in which the files libIp_duc_ddc_compiler_v1_0_bitacc_cmodel.so and libstlport.so.5.1 are located appears as an entry in your LD_LIBRARY_PATH environment variable.
- On Windows platforms, ensure that the directory in which the files libIp_duc_ddc_compiler_v1_0_bitacc_cmodel.dll and libstlport.5.1.dll are located either:
  a. Appears as an entry in your PATH environment variable, or
  b. Is the directory in which you will run your executable that calls the DUC/DDC C model.

XILINX®

Chapter 3

DUC/DDC Bit-Accurate C Model

DUC/DDC C Model Interface

The Application Programming Interface (API) of the C model is defined in the header file `duc_ddc_compiler_v1_0_bitacc_cmodel.h`. The interface consists of three user-visible structures, ten functions to manipulate the model and perform simulation, and four functions to assist in allocating input and output data structures. The header file also contains some useful constants and type definitions.

Type Definitions

The interface contains the type definitions shown in Table 3-1.

Table 3-1: **Type Definitions**

Type Definition	Purpose
typedef double xip_ducddc_data;	Defines the data type used to represent input and output samples
typedef int xip_ducddc_status;	Defines the data type used to represent status codes
typedef void (*msg_handler)(void* handle, int error,const char* msg);	Defines the interface to a message handler function
typedef struct _xip_ducddc_v1_0 xip_ducddc_v1_0;	Declares a handle type to refer to instances of the C model object

Structures

The interface contains the following structures.

Configuration

The xip_ducddc_v1_0_config structure specifies the configuration that should apply to the modelled core. See Table 3-2.

Note: See the core data sheet (DS766) for a full description of the relevant parameters.

Table 3-2: **Configuration Structure**

Field	Type	Description
name	char*	Instance name (arbitrary)
core_type	int	0 = DUC, 1 = DDC
ch_bandwidth	int	Channel bandwidth (1)
if_passband	int	IF passband (in MHz)
digital_if	int	0 = Digital IF at 0Hz, 1 = Digital IF at FS/4
rf_rate	int	RF sample rate (in Hz)
clock_rate	int	Clock speed (in Hz) (2)
n_carriers	int	Number of carriers (1 - 18)
n_antennas	int	Number of antennas (1 - 8)
din_width	int	Bit width (precision) of input data (11 - 18)
dout_width	int	Bit width (precision) of output data (11 - 18)
rounding_mode	int	0 = Round ties up (POSIX/MATLAB®), 1 = Round ties to even (IEEE)

1. Valid values are (1, 3, 5, 10, 15, 20) for LTE DUC/DDC configurations, and (2) for TD-SCDMA configurations. The value 1 corresponds to an actual bandwidth of 1.4MHz; the value 2 corresponds to 1.6MHz.
2. While the DUC/DDC C model is not cycle accurate, the implementation clock frequency must be known to correctly determine how the internal filter structure will be decomposed (which affects both filter coefficient values and latency).

Request

The xip_ducddc_v1_0_data_req structure is used to specify input complex sample data for the C model. See Table 3-3.

Table 3-3: **Request Structure**

Field	Type	Description
din_size	size_t	Number of (I,Q) sample pairs (per carrier, per antenna)
din_dim0	size_t	Number of antennas
din_dim1	size_t	Number of carriers
din_i	xip_ducddc_data*[][]	Real part of sample data
din_q	xip_ducddc_data*[][]	Imaginary part of sample data

The `din_i` and `din_q` arrays are two-dimensional arrays in which each element is a pointer to a sample buffer. The first array dimension is the number of antennas, and the second dimension is the number of carriers. The arrays are statically dimensioned to hold data for the maximum numbers of antennas and carriers (8 and 18 respectively). The number of valid entries in each dimension should be indicated in the `din_dim0` and `dim_dim1` fields. Unused pointer entries in each dimension are ignored and should be filled with null pointers. This mechanism is illustrated in Figure 3-1. This example shows a configuration with six carriers and four antennas.

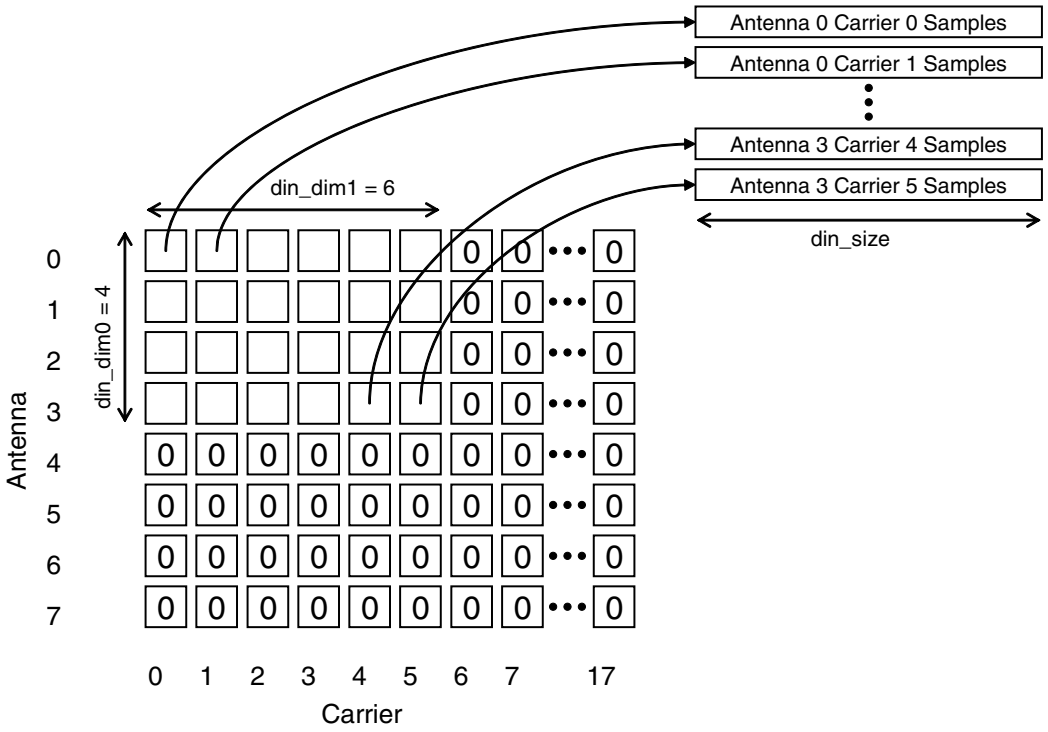

Figure 3-1: **Request Structure Example**

Allocation of the arrays is your responsibility. They may be allocated statically or dynamically. If they are allocated dynamically, then you are responsible for their deallocation. The buffers for every carrier and antenna should be at least as large as `din_size` samples. If they are larger, excess samples will be ignored.

The most efficient way to perform dynamic allocation of such a multi-dimensional buffer structure is to allocate one large buffer with enough sample storage for all carriers and antennas in use, then logically divide the allocated region into smaller blocks and compute pointers to the start of these blocks. The function `xip_ducddc_v1_0_alloc_data_req` performs this operation for data request structures. Its use is recommended, but not mandatory.

Input sample data is in double-precision format, and should be scaled to lie within the range -1.0 to (just less than) +1.0. The input data is expected to be quantized according to the `din_width` that was specified in the model configuration. If the model detects an input value with excessive range or precision, it will saturate and/or round to the nearest valid value and issue a warning message.

For a DDC core, the input is a single sample stream per antenna, even if there is more than one carrier. For a DDC, only the first entry of the carrier dimension is used (that is, `din_dim1` must be equal to 1).

For a DDC configured with a digital IF of FS/4, the input data is real-valued. In these cases, the model ignores the contents of the `dout_q` array.

Response

The xip_ducddc_v1_0_data_resp structure is used to specify output sample data from the C model. See Table 3-4.

Table 3-4: **Response Structure**

Field	Type	Description
dout_size	size_t	Number of (I,Q) sample pairs (per carrier, per antenna) output
dout_max_size	size_t	Number of samples for which space is allocated
dout_clean	size_t	Number of samples which are clean (bit-accurate)
dout_dim0	size_t	Number of antennas
dout_dim1	size_t	Number of carriers
dout_i	xip_ducddc_data*[][]	Real part of sample data
dout_q	xip_ducddc_data*[][]	Imaginary part of sample data

The `dout_i` and `dout_q` arrays are two-dimensional arrays in which each element is a pointer to a sample buffer, as previously described in detail for the "Request" structure. All comments regarding the dimensioning and allocation of arrays apply equally and equivalently to the response structure.

The `dout_max_size` field should be used by the caller to specify the amount of space that has been allocated in each output sample buffer. When processing a transaction, the C model will check that the output arrays have sufficient space for the resulting data before updating the `dout_size` field with the actual number of output samples that were written.

The dout_clean field is updated by the C model when a transaction is performed. It indicates how many samples in each output sample buffer are bit-accurate. At initialization and after a reset, the simulated state of the sample histories of the DUC/DDC constituent FIR filters may differ from those that would be observed in the hardware. Until new input data has propagated through these sample histories, the outputs of the model and the hardware may not match.

When the value of dout_clean is the same as the value of dout_size, all output samples are bit-accurate. If dout_clean is less than dout_size, only the last dout_clean output samples in the output buffer are bit-accurate; earlier samples may not be bit-accurate. If dout_clean is zero, none of the output samples are guaranteed to be bit-accurate.

Output sample data is in double-precision format. It will be scaled to lie within the range -1.0 to (just less than) +1.0, and will be quantized according to the dout_width that was specified in the model configuration.

For a DUC core, the output is a single sample stream per antenna, even if there is more than one carrier. For a DUC, only the first entry of the carrier dimension is used (that is, dout_dim1 will always equal 1).

Functions

The application programming interface contains the following functions. Most functions return a value of type xip_ducddc_status to indicate success (0, XIP_DUCDDC_STATUS_OK) or failure (1, XIP_DUCDDC_STATUS_ERROR).

Get Version

```
const char*
xip_ducddc_v1_0_get_version(void);
```

This function returns a string describing the version of the DUC/DDC Compiler core modeled by the C library (for example “1.0beta,” “1.0,” “1.0patch1”).

Get Default Configuration

```
xip_ducddc_status
xip_ducddc_v1_0_default_config(xip_ducddc_v1_0_config *config);
```

This function populates the xip_ducddc_v1_0_config configuration structure pointed to by config with the default configuration data for the DUC/DDC core. The data in this structure can be further customized to specify user parameters.

Create Model Object

```
xip_ducddc_v1_0 *
xip_ducddc_v1_0_create(
        const xip_ducddc_v1_0_config *config,
        msg_handler handler,
        void *handle
);
```

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

The `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 will always be the value that was passed in as the `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.

Reset Model Object

```
xip_ducddc_status
xip_ducddc_v1_0_reset(xip_ducddc_v1_0 *s);
```

This function resets the internal state of the DUC/DDC C model object pointed to by `s`. (This is not equivalent to a hardware reset.)

A reset causes the phases of any DDS and/or IF mixer components to return to their default starting states. It clears the sample histories of all constituent filters to zero, and causes any partially-accumulated sample data to be discarded. However, a reset does not alter any of the carrier frequencies, phase offsets or gain settings that may have been applied.

Simulate

```
xip_ducddc_status
xip_ducddc_v1_0_data_do(
        xip_ducddc_v1_0 *s,
        xip_ducddc_v1_0_data_req *req,
        xip_ducddc_v1_0_data_resp *resp
);
```

This function applies the data specified in the request structure pointed to by `req` to the model object pointed to by `s`. Any output data resulting from the processing of this transaction is written into the response structure pointed to by `resp`.

The dimensions of the request and response buffer arrays must match the number of antennas and carriers as configured when the model object was created, as previously described in "Structures."

The `dout_i` and `dout_q` buffers and the corresponding `dout_max_size` field of the response structure must be sufficiently large to accommodate the output samples that will be produced. If `dout_max_size` indicates that the buffers are too small, the transaction is

ignored and an error message is issued; `dout_size` will be updated with the number of samples that would have been produced.

Calculate Output Sizes

```
xip_ducddc_status
xip_ducddc_v1_0_data_calc_size(
        xip_ducddc_v1_0 *s,
        xip_ducddc_v1_0_data_req *req,
        xip_ducddc_v1_0_data_resp *resp
);
```

This function determines the output buffer sizes required for a call to the simulation function.

The array dimensions of the request and response structures pointed to by `req` and `resp` should be appropriately initialized as if for a call to `xip_ducddc_v1_0_data_do`, as should the `din_size` field of the request structure. The C model will update the `dout_size` field with the number of samples that would be returned by a simulation operation. You can then use this information to ensure that the output buffers are sufficiently large to accept this data.

If the structures supplied are invalid, the function issues an error message and returns a non-zero error code.

Get Raster Parameters

```
xip_ducddc_status
xip_ducddc_v1_0_ctrl_get_raster(
        xip_ducddc_v1_0 *s,
        double *freq_raster,
        double *phase_raster,
        double *gain_step
);
```

This function retrieves information about the quantization of frequency, phase and gain information within the DUC/DDC core. The double-precision values pointed to by `freq_raster`, `phase_raster` and `gain_step` are overwritten with the corresponding information retrieved from the model object pointed to by `s`. If any of the pointers is null, no value is returned for that parameter.

- The frequency raster gives the resolution to which the carrier frequencies can be set, in Hz.
- The phase raster gives the resolution to which the carrier phases can be set, in radians.
- The gain step gives the minimum non-zero gain that can be programmed for a carrier. Valid gain values are all power-of-two multiples of this value, up to and including 1.0. This quantity is meaningful only for DUC configurations.

If the model object has only a single carrier, then no mixer will be present in the core and, consequently, raster information will not be available. Calling this function on such an object will result in an error.

Set Carrier Parameters

```
xip_ducddc_status
xip_ducddc_v1_0_ctrl_set_carrier(
        xip_ducddc_v1_0 *s,
        int index,
        double f,
        double phi,
        double beta
);
```

This function configures the carrier addressed by index within the model object pointed to by `s` with the frequency, phase and gain settings supplied in `f`, `phi` and `beta`.

The internal quantization of the floating-point values supplied was previously described in "Get Raster Parameters." If the values provided are not already quantized, they will be adjusted by the C model. The actual values that were ultimately set can be retrieved by the "Get Carrier Parameters" function described in the following section.

If the model object has only a single carrier, then no mixer will be present in the core and, consequently, carrier parameters cannot be set. Calling this function on such an object will result in an error unless the values specified are 0.0 for the frequency and phase rasters and 1.0 for the gain step.

The default carrier parameters are a phase offset of 0.0, a gain of 1.0, and a set of frequencies symmetrically distributed around the 0Hz center point, spaced out according to the specified channel bandwidth.

Get Carrier Parameters

```
xip_ducddc_status
xip_ducddc_v1_0_ctrl_get_carrier(
        xip_ducddc_v1_0 *s,
        int    index,
        double *f,
        double *phi,
        double *beta
);
```

This function retrieves the configuration of the carrier addressed by index within the model object pointed to by `s`, placing the frequency, phase and gain settings into the double-precision values pointed to by `f`, `phi` and `beta`.

If the model object has only a single carrier, then no mixer will be present in the core and, consequently, carrier parameters cannot be set. Calling this function on such an object will result in values of 0.0 for the frequency and phase rasters and a value of 1.0 for the gain step.

Destroy Model Object

```
xip_ducddc_status
xip_ducddc_v1_0_destroy(xip_ducddc_v1_0 *s);
```

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

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.

Utility Functions

Allocate Data Request

```
xip_ducddc_status
xip_ducddc_v1_0_alloc_data_req(
    xip_ducddc_v1_0 *s,
    xip_ducddc_v1_0_data_req *r,
    unsigned n_samples
);
```

This function allocates buffers with space for `n_samples` input samples in a request structure pointed to by `r` suitable for use with the model object pointed to by `s`.

Allocation is performed as previously described in "Structures." The `din_dim0`, `din_dim1` and `din_size` fields are filled in appropriately. Any pointers already present in the `din_i` and `din_q` arrays are overwritten. This function should be called only on an uninitialized request structure.

Allocate Data Response

```
xip_ducddc_status
xip_ducddc_v1_0_alloc_data_resp(
    xip_ducddc_v1_0 *s,
    xip_ducddc_v1_0_data_resp *r,
    unsigned n_samples
);
```

This function allocates buffers with space for `n_samples` output samples in a response structure pointed to by `r` suitable for use with the model object pointed to by `s`.

Allocation is performed as previously described in "Structures." The `dout_dim0`, `dout_dim1` and `dout_max_size` fields are filled in appropriately. Any pointers already present in the `dout_i` and `dout_q` arrays are overwritten. This function should be called only on an uninitialized response structure.

Deallocate Data Request

```
xip_ducddc_status
xip_ducddc_v1_0_free_data_req(
        xip_ducddc_v1_0 *s,
        xip_ducddc_v1_0_data_req *r
);
```

This function deallocates a request structure allocated by the `alloc_data_req()` function described previously, and clears its constituent fields to an empty state. It should not be used to deallocate a request structure that was allocated in any other way.

Deallocate Data Response

```
xip_ducddc_status
xip_ducddc_v1_0_free_data_resp(
        xip_ducddc_v1_0 *s,
        xip_ducddc_v1_0_data_resp *r
);
```

This function deallocates a response structure allocated by the `alloc_data_resp()` function described previously, and clears its constituent fields to an empty state. It should not be used to deallocate a response structure that was allocated in any other way.

Compiling

To compile code using the C model, the compiler requires access to the duc_ddc_compiler_v1_0_bitacc_cmodel.h header file.

This can be achieved in either of two ways:

1. Add the directory containing the header file to the include search path of the compiler; or,
2. Copy the header file to a location already on the include search path of the compiler.

Linking

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

Linux

The executable must be linked against the libIp_duc_ddc_compiler_v1_0_bitacc_cmodel.so shared object library.

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

```
-L. -lIp_duc_ddc_compiler_v1_0_bitacc_cmodel
```

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

Windows

The executable must be linked against the libIp_duc_ddc_compiler_v1_0_bitacc_cmodel.dll dynamic link library.

Depending on the compiler, the import library libIp_duc_ddc_compiler_v1_0_bitacc_cmodel.lib may be required.

Using Microsoft Visual Studio.NET, linking is typically achieved by adding the import library to the **Additional Dependencies** edit box under the **Linker** tab of **Project Properties**.

Example

The run_bitacc_cmodel.c file contains some example code to illustrate the basic operation of the C model.

MATLAB

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.

MEX Function

Compiled MEX functions are provided for MATLAB R2008a for each platform. They can be found in the C model matlab directory and have MATLAB MEX platform extensions (for example, mexw32, mex64, mexglx and mexa64).

Source code for the MEX function is also provided so that it can be compiled for other MATLAB software releases if necessary. To compile the MEX function within MATLAB, change to the C model matlab directory and run the `make_mex.m` script.

MATLAB Class

The @duc_ddc_compiler_v1_0_bitacc class handles the create/destroy semantics on the C model. An instance of the class can exist in two states – empty and created. An empty instance has not yet been attached to an instance of the C model and so cannot be used to simulate. A created instance has been attached to a C model instance and can simulate.

The class provides the following methods:

Constructor

[model]=duc_ddc_compiler_v1_0_bitacc

[model]=duc_ddc_compiler_v1_0_bitacc(config)

[model]=duc_ddc_compiler_v1_0_bitacc(field, value [, field, value]*)

The first version of the class constructor method constructs an empty model object that can be created later using the create method. The second version constructs a fully-created 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 in the "Configuration" subsection of "Structures."

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.

Create

[model]=create(model, config)

[model]=create(model, field, value [, field, value]*)

This method creates an instance of the C model using the supplied configuration parameters and attaches it to the model object. If the object has already been created, the existing model is automatically destroyed and a new one created and attached in its place.

The first version creates a model object from a structure that specifies the configuration parameter values to use. The second version is the same as the first, but allows the configuration to be specified as a series of (field, value) pairs rather than a single structure.

The names and valid values of configuration parameters are identical to those previously described in the "Configuration" subsection of "Structures."

Destroy

[mode]=destroy(model)

This method destroys any C model instance attached to the model object.

If the model object is attached to a C model instance, the instance is destroyed and the model object state updated to empty. It is safe to call the destroy method on an empty model object. After a call to the destroy method, the model object will always be empty.

Created

[created]=is_created(model)

This method returns true if the model object is in the created state, and false if it is in the empty state.

Simulate

[model, resp]=simulate(model, req)

[model, resp]=simulate(model, field, value [, field, value]*)

This method processes a data transaction on the model object.

The first version uses a structure to pass the input data into the model. The second version allows the data to be supplied using (field, value) pairs.

The request is a structure with the following members:

Member	Type	Description
din	3-dimensional complex scalar array	Input sample data (combined I and Q) • First dimension is sample index • Second dimension is antenna index • Third dimension is carrier index

The response is a structure with the following members:

Member	Type	Description
dout	3-dimensional complex scalar array	Output sample data (combined I and Q) • First dimension is sample index • Second dimension is antenna index • Third dimension is carrier index

Get Raster

[raster]=get_raster(model)

This method returns the raster parameters of the model object. The output is a structure with the following members:

Member	Type	Description
freq_raster	Real scalar	Frequency raster, in Hz
phase_raster	Real scalar	Phase raster, in radians
gain_step	Real scalar	Minimum non-zero carrier gain (dimensionless)

Get Carrier

[carrier]=get_carrier(model, index)

This method returns the parameters for the carrier with the specified index within the model object. The output is a structure with the following members:

Member	Type	Description
f	Real scalar	Carrier frequency, in Hz
phi	Real scalar	Carrier phase offset, in radians
beta	Real scalar	Carrier gain (dimensionless)

Set Carrier

[model]=set_carrier(model, index, carrier_info)

[model]=set_carrier(model, index, field, value [,field, value]*)

This method configures the parameters for the carrier with the specified index within the model object.

The first version takes a structure of the form returned by the "Get Carrier" function previously described. The second version allows the carrier settings to be specified individually as (field, value) pairs.

Installation

To use the MEX function and class, the files in the C model matlab class directory must be present on the MATLAB search path. This can be achieved in either of two ways:

1. Add the C model matlab directory 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).

Example

The `test_mex.m` script is provided as an example of how to use the MATLAB class. It passes some test data samples through the model and verifies that the results are as expected.