LogiCORE™ IP
LTE RACH Detector v1.0

C Model User Guide

UG736 April 19, 2010

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/19/10	1.0	Initial Xilinx release.

Table of Contents

Preface

About This Guide

The *LogiCORE IP LTE RACH Detector v1.0 User Guide* provides information about generating a LogiCORE™ IP LTE RACH Detector core, customizing and simulating the core utilizing the provided example design, and running the design files through implementation using the Xilinx tools.

Guide Contents

This guide contains the following chapters:

- Chapter 1, "Introduction," introduces the core and provides related information, including recommended design experience, additional resources, technical support, and submitting feedback to Xilinx.
- Chapter 2, "Functional Description," describes the function of the core and related details.
- Chapter 3, "Interfaces," describes the configuration, input and output.

Abbreviations

The following abbreviations are used in this manual:

Abbreviation	Description
CP	Cyclic Prefix
NZC	Prime length of Zadoff-Chu sequence
OFDM	Orthogonal Frequency Division Multiplexing
RACH	Random Access Channel
RTD	Round Trip Delay
UE	User Equipment
ZC	Zadoff-Chu

Conventions

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

Typographical

The following typographical conventions are used in this document:

<table>
<tr><th>Convention</th><th>Meaning or Use</th><th>Example</th></tr>
<tr><td><code>Courier font</code></td><td>Messages, prompts, and program files that the system displays</td><td><code>speed grade: - 100</code></td></tr>
<tr><td><code>Courier bold</code></td><td>Literal commands that you enter in a syntactical statement</td><td><code>ngdbuild</code> design_name</td></tr>
<tr><td rowspan="2">Helvetica bold</td><td>Commands that you select from a menu</td><td>File →Open</td></tr>
<tr><td>Keyboard shortcuts</td><td>Ctrl+C</td></tr>
<tr><td rowspan="3">Italic font</td><td>Variables in a syntax statement for which you must supply values</td><td><code>ngdbuild</code> design_name</td></tr>
<tr><td>References to other manuals</td><td>See the User Guide for more information.</td></tr>
<tr><td>Emphasis in text</td><td>If a wire is drawn so that it overlaps the pin of a symbol, the two nets are not connected.</td></tr>
<tr><td>Dark Shading</td><td>Items that are not supported or reserved</td><td>This feature is not supported</td></tr>
<tr><td>Square brackets [ ]</td><td>An optional entry or parameter. However, in bus specifications, such as <code>bus[7:0]</code>, they are required.</td><td><code>ngdbuild</code> <code>[option_name] design_name</code></td></tr>
<tr><td>Braces { }</td><td>A list of items from which you must choose one or more</td><td><code>lowpwr ={on|off}</code></td></tr>
<tr><td>Vertical bar |</td><td>Separates items in a list of choices</td><td><code>lowpwr ={on|off}</code></td></tr>
<tr><td>Angle brackets < ></td><td>User-defined variable for directory names or in code samples</td><td><directory name></td></tr>
<tr><td>Vertical ellipsis
.
.
.</td><td>Repetitive material that has been omitted</td><td><code>IOB #1: Name = QOUT'</code>
<code>IOB #2: Name = CLKIN'</code>
.
.
.</td></tr>
<tr><td>Horizontal ellipsis . . .</td><td>Repetitive material that has been omitted</td><td><code>allow block</code> block_name loc1 loc2 ... locn;</td></tr>
</table>

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. See "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.

Chapter 1

Introduction

The Xilinx LogiCORE™ IP 3GPP LTE RACH Detector v1.0 core has a bit accurate C model designed for system modeling. This allows the user to model the effect of different core parameters on performance. This document describes how to use the bit accurate C model.

Overview

This user guide provides information about the Xilinx LogiCORE IP 3GPP LTE RACH Detector 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, "Interfaces."

The model is bit-accurate but not cycle-accurate, so it produces exactly the same output data as the core on a block-by-block basis. However, it does not model core latency or interface signals.

About the Core

The LTE RACH Detector core is a Xilinx CORE Generator™ software IP core, included in the latest IP Update on the Xilinx IP Center. For detailed information about the core, see the product page.

Recommended Design Experience

Although the LTE RACH Detector core is a fully-verified solution, the challenge associated with implementing a complete design varies depending on the configuration and functionality of the application. Before implementing the LTE RACH Detector core, it is highly recommended that the designer gain knowledge of the LTE standard, in particular its definition of the Random Access Channel. For best results, previous experience building high performance, pipelined FPGA designs using Xilinx implementation software and UCF is also recommended.

Contact your local Xilinx representative for a closer review and estimation for your specific requirements.

Additional Core Resources

For detailed information about LTE RACH Detector technology and updates to the LTE RACH Detector core, see the following:

Documentation

From the product page:

- *LTE RACH Detector Data Sheet*

From the document directory after generating the core:

- *LTE RACH Detector Release Notes*
- *LTE RACH Detector Data Sheet*

LTE RACH Detector Technology

For information about LTE RACH Detector technology basics, including features, FAQs, the LTE RACH Detector interface, typical applications, specifications, and other important information, see the product page.

References

1. 3GPP TS 36.211 v9.0.0 (2009-12), *Physical Channels and Modulation (Release 8).*
2. LogiCORE IP *DDS Compiler Data Sheet v4.0*, DS558.

Technical Support

For technical support, visit www.xilinx.com/support. Questions are routed to a team of engineers with expertise using the LTE RACH Detector core.

Xilinx cannot guarantee timing, functionality, or support of this product for designs that do not follow these guidelines.

Feedback

Xilinx welcomes comments and suggestions about the LTE RACH Detector core and the documentation supplied with the core.

Core

For comments or suggestions about the LTE RACH Detector core, submit a WebCase from www.xilinx.com/support. Be sure to include the following information:

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

Document

For comments or suggestions about this document, submit a WebCase from www.xilinx.com/support. Be sure to include the following information:

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

Chapter 2

Functional Description

The function of the RACH is to find a known preamble in a RACH sequence which it receives from none, one or more UEs in a cell. This is achieved by correlating the preamble root sequence against the received RACH sequence.

The received RACH sequence is equivalent to a whole number of subframes. It consists of a CP, a useful part and a guard period. The CP is sized according to the radius of the cell, and the useful part is made up of multiples of a 24576 sample sequence.

The CP (cyclic prefix) length is defined in the LTE standard, 3GPP TS 36.211, v8.6.0, so that a RACH signal from a UE with the maximum round trip delay (RTD) will overlap in time with any other UE and allow the correlation to be carried out successfully.

The useful RACH sequence is generated from a Zadoff-Chu sequence [Ref 1] with a prime length of NZC=839 or NZC=139, cyclic shifted to provide orthogonality between UEs. The cyclic shift represents a rotation in frequency, or a delay in the time domain.

The cyclic shifts are dictated by the higher layers in the LTE system. Once all the cyclic shifts have been allocated, other sequences, based on the Uth Z-C root, are allocated by the higher layers. A maximum of 64 roots can be used in the LTE cell.

Up to 64 different RACH preambles must be supported in an LTE cell. These preambles are made from combinations of Z-C roots and their associated cyclic shifts. The RACH preambles are distributed across the roots which are enabled in the RACH system. The RACH receiver needs to process each root to fully extract the RACH preamble data.

Each new root requires a new RACH correlation to be carried out, so the number of roots is directly related to the size and complexity of the resulting design.

The RACH preamble is restricted to 72 sub-carriers, giving the RACH channel a bandwidth of 1.08 MHz. The RACH burst can be considered as an OFDM signal with NZC sub-carriers spaced by $\Delta f' = (N/N_{PRE})$. $\Delta f = 1.25$ kHz.

At transmission, this is transformed into the frequency domain, and then IFFT'ed for transmission. Doing this interpolates it into the 30.72 MHz transmission frequency.

At the receiver, this interpolation means that the received sequence can be digitally down converted to reduce the number of samples and the complexity of the RACH processing.

The correlation process requires six main processing stages. Firstly, the RACH spectrum is centered around DC. Secondly, it is down converted to reduce the sample rate for processing. Down conversion is followed by transformation of the received sequence to the frequency domain. The Zadoff-Chu sequence, which is being correlated against, must be generated in the frequency domain. The received signal and the Z-C sequence are then multiplied in the frequency domain (equivalent to correlation in the time domain). Finally, the result of this multiplication must be transformed back into the time domain to generate a result. These stages are explained in more detail in the following section.

Center RACH Spectrum around DC

A frequency shift is introduced by the UEs; this must be removed so that the received RACH sequence is centered around DC. This is achieved by multiplying the received sequence by a DDS, which is operating at the correct frequency to remove the UE shift.

Down Conversion

The RACH channel has a low bandwidth of 1.08 MHZ, and can be down-sampled in the RACH receiver. To make the subsequent FFT easier, the down-sampling process must result in an M sample sequence, where M is a power of 2. Also the down-sampling factor D must be an integer to preserve orthogonality in the RACH preambles. D must be less than N_{PRE}/N_{ZC}=28.4 to preserve the RACH bandwidth.

For a 20 MHz system, a down-sampling rate D=24 will reduce the 24576 sample RACH sequence to 1024 samples. This gives a sample rate of 1.28 MHz (30.72/24). The RACH preamble occupies a bandwidth of 1.08 MHz, which will be maintained at this sample rate. For a 10 MHz system operating at 15.36 MHz, D=12, and for a 5 MHz system, D=6.

For RACH formats 1, 2 and 3, the 24576 sample useful portion becomes 2x24576 samples. This will be down converted to 2x1024 samples, ready for FFT processing.

Figure 2-1 shows the proposed architecture for the DDC filter. For a 20 MHz system, all four part filters would be used; for a 10 MHz system, H1 would be bypassed; and for a 5 MHz system, H1 and H2 would be bypassed. In each filter, the pass band will be set to 1.08 MHz, and the stop band accordingly, dependent on the down-sampling factor.

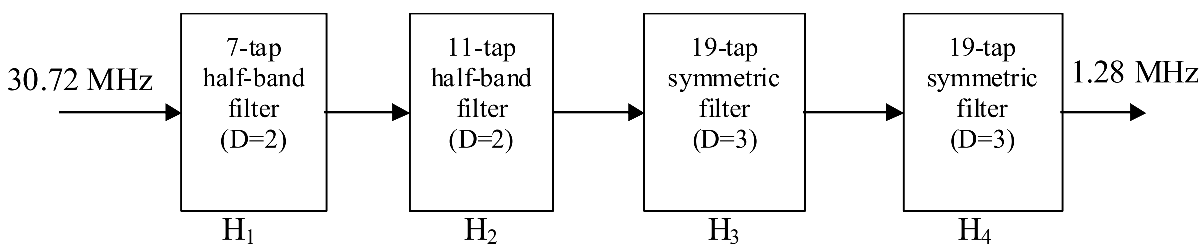

Figure 2-1: **Proposed DDC Filter Design**

M-point FFT

To perform the correlation as efficiently as possible, it is carried out in the frequency domain. The first stage of this process is to FFT the down-sampled RACH sequence. The RACH sequence has been down-sampled to 1024 samples so an FFT can be used.

Normally, when correlating in the frequency domain, a zero pad is added to allow the correlation to grow to 2x, and prevent "leakage" of energy due to the cyclic properties of an FFT. However, for the RACH, we can use this to our advantage. Multiple UEs use cyclic shifted versions of the same Z-C root. When we correlate the received sequence against the original Z-C root, without padding the FFT, all of the cyclic shifts of the Z-C root are found – by not zero padding, the correlation wraps round the end of the sequence.

The FFT will be performed using the Xilinx FFT core (v7.0); this will also perform the IFFT.

Z-C Root Generation in the Frequency Domain

To perform the correlation in the frequency domain, the DFT of the Z-C root sequence is required. Rather than generating the Z-C root, and then taking its DFT, it is possible to generate the DFT of the Z-C sequence directly. The N_{ZC}-point DFT of the uth root sequence can be expressed according to

$$X_u(k) = X_u(0) * e^{j\pi v \frac{k(v*(k+1))}{N_{ZC}}} \quad \text{for } k=0,\ldots N_{ZC}-1$$ *Equation 2-1*

where v is the smallest integer such that (u.v mod N_{ZC}) = 1, and

$$X_u(0) = \sum_{n=0}^{N_{ZC}-1} x_u(n)$$ *Equation 2-2*

The following recursive formula can be further derived:

$$X_u(k+1) = X_u(k) * e^{j\pi \frac{u.v}{N_{ZC}}(2v*(k+v+1))}$$ *Equation 2-3*

Consequently, the real time DFT computation of any root sequence can be implemented as follows:

- Storage of the $X_u(0)$ and v data for all N_{ZC} roots
- Generation of the complex exponential, using a DDS
- Complex multiplication

Correlation (Multiplication in Frequency Domain)

To perform the correlation, the DFT of the Z-C root is multiplied by the FFT of the RACH sequence. This is performed on a sample-by-sample basis using a complex multiplier. The complex multiplication will be performed using a streamed complex multiplier.

K-point IFFT

To get back to the time domain, the result of the multiplication must be IFFT'ed. To achieve the time resolution required, which is specified as 0.52 us in the LTE specification, the IFFT is performed over 2048 samples, which gives a resolution of (800 us/2048 = 0.39 us). To achieve this resolution, the result of the multiplication is zero padded to 2048 samples before the IFFT is performed.

The zero padding is done by inserting a block of 1024 zeros to the output of the complex multiply.

The same FFT core will be used for both the M-point FFT and the K-point IFFT.

Post-Processing

The output from the IFFT will be a 2048 sample sequence. This result may be subject to some post-processing, or output from the core. In either case, a block RAM large enough to store this is required after the IFFT, either to act as a buffer between the interface-X output, or to allow the results processing.

In the post-processing, the results are combined non-coherently. In the case of a single antenna, for format 0,1 or 4, the I and Q results from the IFFT are squared and added. The output is the square of the absolute value. The result will be 2048 samples long.

If there are multiple antennas, the results across the antenna, for the same root sequence, are combined. This is achieved by adding the square of the absolute value for each antenna. The result accumulated across the 2 or 4 antenna in the output block RAM. The result will be 2048 samples long.

For formats 2 and 3, the transmitted RACH contains two copies of the same sequence. In this case, the square of the absolute value for each RACH sequence are added together, to form a single output spectrum of 2048 samples. Combination across antennas is also possible for these formats. So potentially 8 (2 subframes x 4 antennas) correlation results are non-coherently combined to form a single 2048 sample output spectrum.

Non-coherently combining in this way allows us to compress the amount of output data from the RACH; however, it also removes information about the distribution of peaks across antennas and subframes. The information could be useful in implementing the RACH peak detection algorithm (especially under Doppler conditions).

The output of the RACH can be a significant amount of data, so much that it is expensive to store the results internally. The output from the RACH must be read in a timely manner by the next stage (not part of the core), to prevent the RACH from stalling. If the RACH stalls it will not be able to process the sequence within the 2 ms limit. The definition of “timely” is fairly implementation dependent because the smaller implementations produce much less data, and so do not need to be read as often.

Interfaces

The C model takes three structures as an input: configuration, register settings, and the input data stream. Roughly, configuration corresponds to the generics in VHDL designs, registers settings are the control register writes that are sent to the core to control its operation on a per detection basis, and input represents the antenna samples that the core would process. The output corresponds to the data read from the output pins of the IP core.

The header file `lte_rach_detector_v1_0_bitacc_cmodel.h` contains the data structures and the function declarations.

Generation

The function `xip_lte_rach_detector_v1_0_create` returns the pointer to a newly created model. This pointer is used to access a particular instance of the model when it is required. The first parameter is a pointer to the `xip_lte_rach_detector_v1_0_config` data structure. The contents and generation of this structure are described in detail in the following section. It contains the configuration data for the model and is equivalent to specifying the generics in generated core. The second and third parameters can be used for getting the error messages from the model.

```
/**
 * Create a new instance of the core based on some configuration values.
 *
 * @param     config      Pointer to a xip_lte_rach_detector_v1_0_config structure
 * @param     handler     Callback function for errors and warnings (providing a null
 *                        pointer means no messages are output)
 * @param     handle      Optional argument to be passed back to callback function
 *
 * @returns   Exit code   XIP_RACH_STATUS_*
 */
xip_lte_rach_detector_v1_0 *xip_lte_rach_detector_v1_0_create(
  const xip_lte_rach_detector_v1_0_config   *config,
  msg_handler                       handler,
  void                             *handle
);
```

Configuration

The following structure defines the configuration of the RACH core:

```
typedef struct
{
  /**
   * lte_rach_detector_v1_0 Core Generics
   *
   * These are the only generics that influence the operation of this bit-accurate model.
   */
  const char *name;           //@- Instance name (arbitrary)
  int c_antenna_width;        //@- Data width for each antenna, default: 8
  int c_output_width;         //@- Data width for each antenna, default: 8
  int c_arch;                 //@- Specify the fft arch
} xip_lte_rach_detector_v1_0_config;
```

The fields in the structure are described in Table 3-1. The structure can be created in either of two ways:

1. Calling the function

   ```
   xip_rach_status xip_lte_rach_detector_v1_0_default_config(xip_lte_rach_detector_v1_0_config *config)
   ```

   which will set the fields to their default values; or

2. Directly setting the fields to the desired values, which should be the same as the core generics.

The config data should be set before calling `xip_lte_rach_detector_v1_0_create`. Changing the values of the fields requires that the `xip_lte_rach_detector_v1_0_create` is recalled.

Table 3-1: **xip_lte_rach_detector_v1_0_config Fields**

Configuration Field	Range	Default	Description
c_antenna_width	8-14	8	Defines the width of the RACH antenna input. The width value describes the size of each I or Q sample.
c_output_width	12-20	16	Defines the width of the Rach Output. Determines the range of the correlation peaks generated by the RACH.
c_arch	1,2,3,4	2	Defines the architecture of the FFT inside the LTE RACH detector. This varies according to the RACH configuration, depending on the processing speed required. 1 = Radix 4 2 = Radix 2 3 = Streamed 4 = Radix 2 lite

Registers

The C model includes the function:

```
/**
 * Set RACH registers
 *
 * @param    s             Pointer to xip_lte_rach_detector_v1_0 state structure
 * @param    num_ant       the number of antenna being processed in this RACH
 * @param    format        The rach format 0,1,2,3 or 4
 * @param    bw            The BW of the RACH being processed 5,10, 20(MHz).
 * @param    num_fc        The number of frequency channels to be processed.
 * @param    fc            The demodulating frequencies for each frequency channel
 * @param    number_roots  The number ZC roots the RACH is searching for
 * @param    root          The ZC root values for the RACH detector. Maximum of 64 root,
                           with values from 1-839(formats 0-3) and 1-139(format 4)
 * @returns  Exit code     XIP_RACH_STATUS_*
*/
Ip_xilinx_ip_lte_rach_detector_v1_0_DLL
xip_rach_status xip_lte_rach_detector_v1_0_ctrl_set_registers
  ( xip_lte_rach_detector_v1_0 *s,
    int num_ant,
    int format,
    int bw,
    unsigned num_fc,
    double *fc,
    size_t number_roots,
    unsigned *root
  );
```

This function allows the control registers in the RACH core to be replicated inside the model. It takes a pointer to an instance of the model and the new register values. Table 3-2 describes the inputs to the function in greater detail. The function returns a status indicator.

The function is called every time the registers need to be updated with new control information. It can be called repeatedly for a single instance of the C model, and should generally be called before each new RACH detection operation (unless the register contents is not being changed, in which case it is unnecessary). There is no internal FIFOing of the register writes, so each new call will overwrite the previous settings.

Table 3-2: **Configuration Register Values**

Name	Type	Range	Default Value	Corresponding Register Field (see core data sheet DS761)	Description
num_ant	int	1,2,4	1	NUM_ANT	Defines the number of antenna to be processed for the current RACH transmission. This is equivalent to the "NUM_ANT" register value in the LTE RACH detector core.
format	int	0,1,2,3,4	0	FORMAT	Defines the RACH format being processed by the LTE RACH detector C model.
bw	int	5,10, 20	20	BW	Defines the bandwidth in which the RACH detector is operating. The values are in MHz.

Table 3-2: **Configuration Register Values *(Continued)***

Name	Type	Range	Default Value	Corresponding Register Field (see core data sheet DS761)	Description
num fc	Unsigned	1-6	1	NUM_FC	The number of frequency channels the RACH detector is processing.
fc[]	Double		0.0	FC[0-5]	Defines the carrier frequency being used to de-multiplex each of the channels in the frequency multiplex. The multiplexing frequencies (f_{mux}) used in the RACH transmission are generated according to TS 36.211, Sections 5.7.1 and 5.7.3 [Ref 1]. To de-multiplex, the negative of the multiplexing frequency must be used, (($f_{demux} = -f_{mux}$). In the RACH receiver, the FC is used to drive a DDS inside the core. Within the DDS, the generated frequency is defined as a phase increment per sample (see the LogiCORE™ IP DDS compiler v4.0, DS558 [Ref 2]). To create the correct de-multiplexing frequency, the FC should be set to the phase increment for f_{demux}. For the RACH core, the sample rate is 30.72 MHz. For a de-multiplexing frequency of f_{demux}, the phase increment per sample is: $P_{inc} = f_{demux}/30720000$ This result will be a real value between -1.0 and 1.0.
number_ roots	Unsigned	1-64	1	NUMBER_ ROOTS	Defines the maximum number of roots that the LTE RACH detector core is configured to process. This differs from the number of roots that is processed for a given received RACH, which is configured on a per RACH basis (see below).
root[]	Unsigned		0	ZC_ROOT_ {0-63}_INDEX	Array of the roots that the RACH detector is detecting.

Input

The input data to the model is defined in the structure:

```
typedef struct
{
  /**
   * Data port input structure. Contains a two-dimensional array of pointers
   * to sample data buffers - one pointer per sector, per antenna.
   */

  xip_rach_data *din_i[XIP_RACH_MAX_ANTENNAS]; //@- Real part of sample data
  xip_rach_data *din_q[XIP_RACH_MAX_ANTENNAS]; //@- Imaginary part of sample data

} xip_lte_rach_detector_v1_0_data_req;
```

The fields inside this structure are described in Table 3-3. The input antenna data should mapped into this structure and scaled to fit within the expected range.

Table 3-3: **Input Data Fields**

Field	Range	Description
din_i [] []	[-1.0, 1.0)	Array of arrays of the Real input samples. Each entry in the top level array contains an array of input data for one antenna. The length of this array depends on the RACH format and the system bandwidth. Each sample represents a received antenna sample in the LTE RACH detector core. Each input sample is a double in the range -1.0 to 1.0, which corresponds to the range [-2^(c_antenna_width-1) to 2^(c_antenna_width-1)-1] in the LTE RACH detector core. Generating the samples for the input may involve combining multiple "UE" transmissions. To make this compatible with the input, it is necessary to scale the input samples so they fit within the range expected by the model.
din_q [] []	[-1.0, 1.0)	Array of arrays of the Imaginary input samples. The description is the same as for din_i.

The input data structure comes with the utility functions:

```
/**
 * Allocate appropriate buffers in a data request structure.
 *
 * @param     s           Pointer to xip_lte_rach_detector_v1_0 state structure
 * @param     r           Pointer to request structure to set up
 * @returns   Exit code   XIP_RACH_STATUS_*
 *
 */
Ip_xilinx_ip_lte_rach_detector_v1_0_DLL
xip_rach_status xip_lte_rach_detector_v1_0_alloc_data_req
  ( xip_lte_rach_detector_v1_0          *s,
    xip_lte_rach_detector_v1_0_data_req *r
  );
```

and

```
/**
 * Deallocate the buffers in a data request structure allocated by
   xip_lte_rach_detector_v1_0_alloc_data_req
 *
 * @param     s           Pointer to xip_lte_rach_detector_v1_0 state structure
 * @param     r           Pointer to request structure to free
 * @returns   Exit code   XIP_RACH_STATUS_*
 *
 */
Ip_xilinx_ip_lte_rach_detector_v1_0_DLL
xip_rach_status xip_lte_rach_detector_v1_0_free_data_req
  ( xip_lte_rach_detector_v1_0          *s,
    xip_lte_rach_detector_v1_0_data_req *r
  );
```

Both functions take a pointer to the C model instance (`xip_lte_rach_detector_v1_0`) and a pointer to an input data structure (`xip_lte_rach_detector_v1_0_data_req`). `xip_lte_rach_detector_v1_0_alloc_data_req` points the data input structure to an area of memory large enough to accommodate the input data for a RACH detection. `xip_lte_rach_detector_v1_0_free_data_req` frees the memory block.

Output

The output data to the model is defined in the structure:

```
typedef struct
{
  /**
   * Data port output structure. Contains an array of arrays of output samples. 1 array per
     root, per frequency channel.
   */

  size_t num_roots_out;     //@- Number of samples
  size_t dout_size;
  xip_rach_data *dout_i[XIP_RACH_MAX_RACHOUT]; //@- Real part of sample data
  xip_rach_data *dout_q[XIP_RACH_MAX_RACHOUT]; //@- Imaginary part of sample data

} xip_lte_rach_detector_v1_0_data_resp;
```

The fields inside this structure are described in Table 3-4.

Table 3-4: **Input Data Fields**

Field	Range	Description
num_roots_out	1-384	The number of output RACH correlation results. This figure is the product of the number of frequency channels and the number of roots. Potentially there are 6 channels, each containing 64 roots, giving a maximum value of 384.
dout_size	512, 2048	The length of the RACH correlation. For formats 0,1,2 and 3, the output length is 2048 samples to provide a temporal resolution that meets the LTE specification. For format 4, the output is 512 samples long to achieve sufficient resolution.
dout_i[][]	[0.0, 1.0)	Array of the real output data samples. It consists of an array of Num_roots_out, one for each RACH correlation result. Each entry in this array contains an array of dout_size samples that make up the RACH correlation result. The RACH results are normalized. Each output sample is a double in the range 0 to 1.0. The RACH output is a power, so always positive. The maximum output value is the equivalent to (2^(c_output_width) -1) in the RACH core.
dout_q[][]		The output contains an imaginary component; however because the RACH output is a magnitude only, the Imaginary component is always 0.

The output data structure comes with the utility functions:

```
/**
 * Allocate appropriate buffers in a data response structure.
 *
 * @param     s           Pointer to xip_lte_rach_detector_v1_0 state structure
 * @param     r           Pointer to response structure to set up
 * @returns   Exit code   XIP_RACH_STATUS_*
 *
 */
Ip_xilinx_ip_lte_rach_detector_v1_0_DLL
xip_rach_status xip_lte_rach_detector_v1_0_alloc_data_resp
  ( xip_lte_rach_detector_v1_0           *s,
    xip_lte_rach_detector_v1_0_data_resp *r
  );
```

and

```
/**
 * Deallocate the buffers in a data request structure allocated by
xip_lte_rach_detector_v1_0_alloc_data_resp
 *
 * @param     s           Pointer to xip_lte_rach_detector_v1_0 state structure
 * @param     r           Pointer to response structure to free
 * @returns   Exit code   XIP_RACH_STATUS_*
 *
 */
Ip_xilinx_ip_lte_rach_detector_v1_0_DLL
xip_rach_status xip_lte_rach_detector_v1_0_free_data_resp
  ( xip_lte_rach_detector_v1_0           *s,
    xip_lte_rach_detector_v1_0_data_resp *r
    );
```

Both functions take a pointer to the C model instance (`xip_lte_rach_detector_v1_0`) and a pointer to an output data structure (`xip_lte_rach_detector_v1_0_data_resp`). `xip_lte_rach_detector_v1_0_alloc_data_resp` points the data output structure to an area of memory large enough to accommodate the output data for a RACH detection. `xip_lte_rach_detector_v1_0_free_data_resp` frees the memory block.

Running the Model

Running the C model is achieved by calling the following function. This function accepts a pointer to an instance of the C model, which is created using `xip_lte_rach_detector_v1_0_create`, a pointer to an input data structure created using `xip_lte_rach_detector_v1_0_alloc_data_req` which has been loaded with suitable input samples, and a pointer to an output data structure, created using `xip_lte_rach_detector_v1_0_alloc_data_resp`. Calling the model will return this output structure having been filled with the RACH output.

The `xip_lte_rach_detector_v1_0_ctrl_set_registers` function should be called prior to calling the model to ensure the correct register set up. Failure to do so will result in the default register values being used.

```
/**
 * Apply a transaction on the data port.
 *
 * @param     s          Pointer to xip_lte_rach_detector_v1_0 state structure
 * @param     req        Pointer to xip_lte_rach_detector_v1_0_data_req request structure
 * @param     resp       Pointer to xip_lte_rach_detector_v1_0_data_resp response
                         structure
 * @returns   Exit code  XIP_RACH_STATUS_*
 */
Ip_xilinx_ip_lte_rach_detector_v1_0_DLL
xip_rach_status xip_lte_rach_detector_v1_0_data_do
  ( xip_lte_rach_detector_v1_0           *s,
    xip_lte_rach_detector_v1_0_data_req  *req,
    xip_lte_rach_detector_v1_0_data_resp *resp
  );
```