LogiCORE™ IP Video On-Screen Display v2.0 Bit Accurate C Model

User Guide

UG816 June 10, 2011

XILINX®

Xilinx is providing this product documentation, hereinafter “Information,” to you “AS IS” with no warranty of any kind, express or implied. Xilinx makes no representation that the Information, or any particular implementation thereof, is free from any claims of infringement. You are responsible for obtaining any rights you may require for any implementation based on the Information. All specifications are subject to change without notice.

XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE INFORMATION OR ANY IMPLEMENTATION BASED THEREON, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF INFRINGEMENT AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Except as stated herein, none of the Information may be copied, reproduced, distributed, republished, downloaded, displayed, posted, or transmitted in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx.

© 2011 Xilinx, Inc. XILINX, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/10/11	1.0	Initial Xilinx release.

Table of Contents

Revision History 2

Preface: About This Guide

Guide Contents 5
Additional Resources 5
Conventions 6
Typographical 6
Online Document 7

Chapter 1: Introduction

Features 9
Overview 9
Additional Core Resources 10
Technical Support 10
Feedback 10
Video On-Screen Display v2.0 Bit Accurate C Model and IP Core 10
Document 10

Chapter 2: User Instructions

Unpacking and Model Contents 11
Installation 12
Software Requirements 13

Chapter 3: Interface

OSD Generics Structure 17
OSD Inputs Structure 18
Frame Configuration 18
Layer Configuration 19
Graphics Configuration 19
OSD Outputs Structure 21
OSD Video Structure 22
Working With Video_struct Containers 23
Delete the Video Structure 24

Chapter 4: C Model Example Code

Config File Format 26
Color LUT File Format 30
Font File Format 30
String File Format 31
Instruction File Format 32

Initializing the OSD Input Video Structure 33
Bitmap Image Files 33
YUV Image/Video Files 34
Binary Image/Video Files 35
Compiling on 32-bit and 64-bit Windows Platforms 35
Compiling under 64-bit Linux Platforms 36
Example Demonstration 36
Configurable Demonstration 36
Compiling on 32-bit Linux Platforms 37
Example Demonstration 37
Configurable Demonstration 38
Running the Executables 38
Example Demonstration 38
Configurable Demonstration 39

About This Guide

This user guide provides information about the Xilinx® LogiCORE IP Video On-Screen Display (OSD) v2.0 bit accurate C model 32-bit Windows, 64-bit Windows, 32-bit Linux, and 64-bit Linux platforms.

Guide Contents

This manual contains the following chapters:

- Chapter 1, Introduction introduces the bit accurate C model for the Xilinx® LogiCORE™ IP Video On-Screen Display v2.0 core, which has been developed primarily for system level modeling.
- Chapter 2, User Instructions provides information on the C model directory structure, files, installation, and software requirements.
- Chapter 3, Interface provides information on the C model interface, including defining the inputs, generics and output of the OSD.
- Chapter 4, C Model Example Code provides an example C file along with the Win32 version of the executable for this example.

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

This document introduces the bit accurate C model for the Xilinx® LogiCORE™ IP Video On-Screen Display (OSD) v2.0 core, which has been developed primarily for system level modeling.

Features

- Bit accurate with OSD v1.0 and v2.0 core
- Statically linked library (.lib, .o, .obj)
- Available for 32-bit Windows, 64-bit Windows, 32-bit Linux and 64-bit Linux platforms
- Supports all features of the OSD core that affect numerical results
- Designed for rapid integration into a larger system model
- Example C code is provided to show to use the function

Overview

The Xilinx LogiCORE IP Video OSD v2.0 has a bit accurate C model for 32-bit Windows, 64-bit Windows, 32-bit Linux and 64-bit Linux platforms. The model has an interface consisting of a set of C functions, which reside in a statically link library (shared library). Full details of the interface are given in "OSD v2.0 C Model Interface". An example piece of C code is provided to show how to call the model.

The model is bit accurate, as it produces exactly the same output data as the core on a frame-by-frame basis. However, the model is not cycle accurate, as it does not model the core's latency or its interface signals. The model is backward compatible with the Xilinx LogiCORE IP Video OSD v1.0 feature set. The Xilinx LogiCORE IP Video OSD v1.0 does not support 10 and 12 bit data widths (the v2.0 core and C-model *do* support). Consequently, the 10 and 12 bit C model configurations are invalid for the Xilinx LogiCORE IP Video OSD v1.0.

The latest version of the model is available for download on the Xilinx™ LogiCORE IP Video OSD web page at:

http://www.xilinx.com/products/ipcenter/EF-DI-OSD.htm

Additional Core Resources

For detailed information and updates about the Xilinx LogiCORE IP Video OSD v2.0 core, see:

http://www.xilinx.com/products/ipcenter/EF-DI-OSD.htm

- *Video On-Screen Display v2.0 User Guide (UG801)*
- *Video On-Screen Display v2.0 Data Sheet (DS837)*
- *Video On-Screen Display v2.0 Release Notes*

Technical Support

For technical support, go to www.xilinx.com/support. Questions are routed to a team with expertise using the Video On-Screen Display v2.0 core.

Xilinx provides technical support for use of this product as described in this user guide (*LogiCORE IP Video On-Screen Display Bit Accurate C Model User Guide*).

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

Feedback

Xilinx welcomes comments and suggestions about the Video On-Screen Display v2.0 core and the accompanying documentation.

Video On-Screen Display v2.0 Bit Accurate C Model and IP Core

For comments or suggestions about the Video On-Screen Display v2.0 core and bit accurate C model, submit a WebCase from http://www.xilinx.com/support/clearexpress/websupport.htm. Be sure to include the following information:

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

Document

For comments or suggestions about the documentation for the Video On-Screen Display v2.0 core and bit accurate C model, submit a WebCase from http://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

Unzip the `v_osd_v2_0_bitacc_model.zip` file, containing the bit accurate models for the On-Screen Display IP Core. This creates the directory structure and files in Table 2-1.

Table 2-1: **Directory Structure and Files of the Video On-Screen Display v2.0 Bit Accurate C Model**

File Name	Contents
README.txt	Release notes
ug816_v_osd.pdf	LogiCORE IP Video On-Screen Display Bit Accurate C Model User Guide
Makefile	Makefile for running GCC via make for 32-bit and 64-bit Linux platforms
v_osd_v2_0_bitacc_cmodel.h	Model header file
rgb_utils.h	Header file declaring the RGB image/video container type and support functions
yuv_utils.h	Header file declaring the YUV (.yuv) image file I/O functions
bmp_utils.h	Header file declaring the bitmap (.bmp) image file I/O functions
video_utils.h	Header file declaring the generalized image/video container type, I/O and support functions
video_fio.h	Header file declaring support functions for test bench stimulus file I/O
run_bitacc_cmodel.c	Example code calling the C model
run_bitacc_cmodel_config.c	Example code calling the C model; uses command line and config file arguments
/lin64	Precompiled bit accurate ANSI C reference model for simulation on 64-bit Linux platforms
libIp_v_osd_v2_0_bitacc_cmodel.so	Model shared object library
libstlport.so.5.1	STL library, referenced by libIp_ v_osd_v2_0_bitacc_cmodel.so
run_bitacc_cmodel	64-bit Linux fixed configuration executable
run_bitacc_cmodel_config	64-bit Linux programmable configuration executable
/lin	Precompiled bit accurate ANSI C reference model for simulation on 32-bit Linux platforms.

Table 2-1: **Directory Structure and Files of the Video On-Screen Display v2.0 Bit Accurate C Model**

libIp_v_osd_v2_0_bitacc_cmodel.so	Model shared object library
libstlport.so.5.1	STL library, referenced by libIp_ v_osd_v2_0_bitacc_cmodel.so
run_bitacc_cmodel	32-bit Linux fixed configuration executable
run_bitacc_cmodel_config	32-bit Linux programmable configuration executable
/nt64	Precompiled bit accurate ANSI C reference model for simulation on 64-bit Windows platforms
libIp_v_osd_v2_0_bitacc_cmodel.lib	Precompiled library file for 64-bit Windows platforms compilation
run_bitacc_cmodel.exe	64-bit Windows fixed configuration executable
run_bitacc_cmodel_config.exe	64-bit Windows programmable configuration executable
/nt	Precompiled bit accurate ANSI C reference model for simulation on 32-bit Windows platforms
libIp_v_osd_v2_0_bitacc_cmodel.lib	Precompiled library file for 32-bit Windows platforms compilation
run_bitacc_cmodel.exe	32-bit Windows fixed configuration executable
run_bitacc_cmodel_config.exe	32-bit Windows programmable configuration executable
examples	Example input files to be used with the run_bitacc_cmodel_config executable
example0.cfg	Example config file; internal test patterns, no graphics controller and BMP output
example1.cfg	Example config file; no input, internal test patterns , no graphics controller and YUV output
example2.cfg	Example config file; BMP input, no graphics controller and BMP output
example3.cfg	Example config file., BMP input, graphics overlay and BMP output
clut.txt	Example graphics controller color look-up table/pallet file
string.txt	Example graphics controller text/strings file
font.txt	Example graphics controller font file
instructions.txt	Example graphics controller instruction list
bridge.bmp	Example 24-bit 576x720 bitmap

Installation

For Linux, make sure the following files are in a directory that is in your $LD_LIBRARY_PATH environment variable:

- libIp_v_osd_v2_0_bitacc_cmodel.so
- libstlport.so.5.1

Software Requirements

The Video On-Screen Display v2.0 C models were compiled and tested with the software listed in Table 2-2.

Table 2-2: **Compilation Tools for the Bit Accurate C Models**

Platform	C Compiler
64-bit Linux	GCC 3.4.6 & 4.1.1
32-bit Linux	GCC 3.4.6 & 4.1.1
64-bit Windows	Microsoft Visual Studio 2005
32-bit Windows	Microsoft Visual Studio 2005

Chapter 3

Interface

The video OSD bit accurate C model core function is a statically linked library. This model is accessed through a set of functions and data structures that are declared in the `v_osd_v2_0_bitacc_cmodel.h` file. A higher level software project can make function calls to this function:

```
/**
 * Create a new state structure for this C-Model.
 *
 * IMPORTANT: Client is responsible for calling
 *            xilinx_ip_v_osd_v2_0_destroy_state()
 *            to free state memory.
 *
 * @param generics    Generics to be used to configure C-Model
 *                    state.
 *
 * @returns xilinx_ip_v_osd_v2_0_state*  Pointer to the internal
 *                                       state.
 */
struct xilinx_ip_v_osd_v2_0_state*
xilinx_ip_v_osd_v2_0_create_state(struct xilinx_ip_v_osd_v2_0_generics generics);

/**
 * Simulate this bit-accurate C-Model.
 *
 * @param     state      Internal state of this C-Model. State
 *                       may span multiple simulations.
 * @param     inputs     Inputs to this C-Model.
 * @param     outputs    Outputs from this C-Model.
 *
 * @returns   Exit code   Zero for SUCCESS, Non-zero otherwise.
 */
int xilinx_ip_v_osd_v2_0_bitacc_simulate
(
 struct xilinx_ip_v_osd_v2_0_state*   state,
 struct xilinx_ip_v_osd_v2_0_inputs   inputs,
 struct xilinx_ip_v_osd_v2_0_outputs* outputs
);
```

Before using the model, the structures holding the generics, inputs, and outputs of the OSD instance must be defined:

```
struct  xilinx_ip_v_osd_v2_0_generics generics;
struct  xilinx_ip_v_osd_v2_0_inputs   inputs;
struct  xilinx_ip_v_osd_v2_0_outputs  outputs;
```

The declaration of these structures is in the `v_osd_v2_0_bitacc_cmodel.h` file.

Before making the function call, complete these steps:

1. Populate the *generics* structure. It defines the values of build time parameters. See OSD Generics Structure for more information on the structure and an example of how to initialize.
2. Populate the *inputs* structure. It defines the values of run time parameters. See OSD Inputs Structure for more information on the structure and an example of how to initialize.
3. Populate the *outputs* structure. See OSD Outputs Structure for more information on the structure and an example of how to initialize.

After the inputs are defined and all `video_structs` are initialized, the model can be simulated by calling the following functions:

```
state = xilinx_ip_v_osd_v2_0_create_state(generics);
if (state == NULL) {
  printf("ERROR: could not create state object\n");
  return 1;
}

// Simulate the core
printf("Running the C model...\n");
if(xilinx_ip_v_osd_v2_0_bitacc_simulate(state, inputs, &outputs) != 0) {
  printf("ERROR: simulation did not complete successfully\n");
  return 1;
} else {
  printf("Simulation completed successfully\n");
}
```

The results are provided in the outputs structure, which contains only one member of type `video_struct`. See OSD Video Structure for more information on `video_struct`.

The successful execution of all provided functions return a value of 0, otherwise a non-zero error code indicates that problems occurred during function calls.

OSD Generics Structure

The Xilinx LogiCORE IP Video OSD Core bit accurate C model takes multiple generic parameters. All generic parameters are integers or integer arrays. See Table 3-1 for generic definitions.

Table 3-1: **OSD Generics Structure**

Generic	Designation
C_DATA_WIDTH	Data width of each color component channel; valid values are 8, 10 and 12.
C_NUM_LAYERS	The number of layers.
C_LAYER_TYPE[8]	Defines the layer type of each layer: 1=Graphics Controller 2=VFBC 3=XSVI All other values are reserved.
C_LAYER_INS_BOX_EN[8]	Enable box instructions.
C_LAYER_INS_TEXT_EN[8]	Enable text instructions.
C_LAYER_CLUT_SIZE[8]	Maximum number of colors.
C_LAYER_TEXT_NUM_STRINGS[8]	Maximum number of strings.
C_LAYER_TEXT_MAX_STRING_LENGTH[8]	Maximum string length.
C_LAYER_FONT_NUM_CHARS[8]	Maximum number of characters.
C_LAYER_FONT_WIDTH[8]	Maximum font width.
C_LAYER_FONT_HEIGHT[8]	Maximum font height.
C_LAYER_FONT_BPP[8]	Font bits per pixel.
C_LAYER_FONT_ASCII_OFFSET[8]	The ASCII value of the first character in the font file.

Calling `xilinx_ip_v_osd_v2_0_get_default_generics()` initializes the generics structure, `xilinx_ip_v_osd_v2_0_generics`, with the OSD defaults. An example of initialization of the generics structure with layer two configured as a graphics controller is as follows:

```
generics = xilinx_ip_v_osd_v2_0_get_default_generics(); //Get Defaults
generics.C_NUM_LAYERS = 3;
generics.C_LAYER_TYPE[2] = 1; // Graphics Controller

// Setup Graphics Controller
generics.C_LAYER_INS_BOX_EN[2]  = 1;
generics.C_LAYER_INS_TEXT_EN[2] = 1;
generics.C_LAYER_CLUT_SIZE[2]   = 256;

// Setup Font RAM
generics.C_LAYER_FONT_NUM_CHARS[2]    = 128;
generics.C_LAYER_FONT_WIDTH[2]        = 8;
generics.C_LAYER_FONT_HEIGHT[2]       = 8;
generics.C_LAYER_FONT_BPP[2]          = 1;
generics.C_LAYER_FONT_ASCII_OFFSET[2] = 0;
```

```
// Setup Text RAM
generics.C_LAYER_TEXT_NUM_STRINGS[2] = 16; // Set number of strings
generics.C_LAYER_TEXT_MAX_STRING_LENGTH[2] = 64; //Set max string length
```

OSD Inputs Structure

The structure `xilinx_ip_v_osd_v2_0_inputs` defines the values of run time parameters and the actual input video frames/images for each layer.

```
struct xilinx_ip_v_osd_v2_0_inputs
{
  struct  video_struct video_in[OSD_MAX_LAYERS];

  struct frame_cfg_struct * frame_cfg;
  struct layer_cfg_struct *layer_cfg[OSD_MAX_LAYERS];
  struct graphics_cfg_struct * gfx_cfg[OSD_MAX_LAYERS];

  int     num_frames;
  int     color_space;

}; // end xilinx_ip_v_osd_v2_0_inputs
```

The `video_in` variable is an array of `video_struct` structures, one structure per layer. See the OSD Video Structure for a description of the `video_in` structure. The `video_in` structure must be initialized if neither the internal graphics controller nor the test pattern generator is used.

Frame Configuration

The `frame_cfg` variable is a pointer to the `frame_cfg_struct`. The `frame_cfg_struct` is defined as:

```
struct frame_cfg_struct
{
  int y_size;
  int x_size;
  int bg_color[3];

  struct frame_cfg_struct * next; // For Changing parameters each Frame
};
```

The `frame_cfg` variable points to the first element in the frame config linked list. For each frame, the OSD model reads the x and y size of output frame and the background color from the `frame_cfg_struct` pointed to by `frame_cfg`. At the end of the frame, if the next pointer is not NULL, the OSD model updates the background color and the output size from the next structure in the linked list. Consequently, if the number of video frames is more than the number of elements in the linked list, the last element is used for the remaining frames. The user is responsible for initializing the linked list.

Layer Configuration

The `layer_cfg` variable is an array of pointers to the `layer_cfg_struct` structure, one pointer per layer. The `layer_cfg_struct` is defined as:

```
struct layer_cfg_struct
{
  int enable;
  int g_alpha_en;
  int priority;
  int alpha;
  int x_pos;
  int y_pos;
  int x_size;
  int y_size;

  int chan_mode[4];
  int chan_color[4];

  struct layer_cfg_struct * next; // For Changing parameters each Frame
};
```

Each pointer must be initialized to point to the first element in the layer config linked list. For each frame, the OSD model reads the layer registers and the test parameter arrays (`chan_mode[4]` and `chan_color[4]`) from the `layer_cfg_struct` pointed to by the `layer_cfg` pointer. This linked list enables the user to change the layer configuration (size, position, transparency, z-plane, and so on) for each video frame.

At the end of the frame, if the next pointer is not NULL, the OSD model updates the layer configuration from the next structure in the linked list. Consequently, if the number of video frames is more than the number of elements in the linked list, the last element is used for the remaining frames. The user is responsible for initializing the linked list.

Graphics Configuration

The `gfx_cfg` variable is an array of pointers to the `graphics_cfg_struct` structure, one pointer per layer. This variable is only used if the layer is configured for graphics controller input. The `graphics_cfg_struct` is defined as:

```
struct graphics_cfg_struct
{
  int layer_num;

  uint16 * clut; // Color Table
  char * text_ram; // Text Ram
  int * font_ram; // Font Ram

  struct graphics_list * graph_instruction;

  struct graphics_cfg_struct * next; // For Changing parameters each
Frame
};
```

Each pointer must be initialized to point to the first element in the graphics config linked list. For each frame, the OSD model reads the graphics controller memories from the `graphics_cfg_struct` pointed to by the `gfx_cfg` pointer. This linked list enables the user to change the graphics controller output (boxes, text, color, size, position, transparency, font and strings) for each video frame.

The CLUT pointer points to an array of 16-bit unsigned integers. This array contains the color entries for the current video frame. Each color entry contains four integers, one for each color component and one for alpha. The CLUT array must contain 4*16 or 4*256 integers.

The `text_ram` pointer points to an array of characters. This array contains all strings for the current video frame. The number of characters in the array must equal the (maximum string length) * (the number of strings).

The `font_ram` pointer points to an array of integers. This array contains the font for the current video frame. The number of integers in the array must equal the (number of characters) * (font width) * (font height). The number of bits used in each integer is 8, 16 or 32 depending on the setting of the `font_width` and `font_bpp`.

The `graph_instruction` pointer points to a linked list of graphics instructions (defined by the `graphics_list` structure). This linked list contains the graphics instructions for the current video frame. The Graphics Controller draws each instruction in the linked list until a NULL pointer is encountered. The `graphic_list` structure is defined as:

```
struct graphics_list
{
  int opcode;
  int xstart;
  int xstop;
  int ystart;
  int ystop;
  int color_index;
  int text_index;
  int object_size;

  struct graphics_list * next;

};
```

This structure contains the same fields as in the instruction file defined previously. The opcode variable can be `OSD_INS_BOX`, `OSD_INS_TEXT` or `OSD_INS_BOXTEXT` (each defined in the `v_osd_v_2_0_bitacc_cmodel.h` file). See *UG801 - Video On-Screen Display v2.0 User Guide* for more information on `xstart`, `xstop`, `ystart`, `ystop`, `color_index`, `text_index`.

At the end of the frame, if the next pointer is not NULL, the OSD model updates the graphics controller configuration from the next structure in the linked list. Consequently, if the number of video frames is more than the number of elements in the linked list, the last element is used for the remaining frames. The user is responsible for initializing the linked list. Example initialization code of the inputs structure is as follows:

```
  inputs.frame_cfg   = (struct frame_cfg_struct *) calloc(1,
sizeof(struct frame_cfg_struct));
  inputs.frame_cfg->x_size      = 1280;
  inputs.frame_cfg->y_size      =  720;
  inputs.frame_cfg->bg_color[0] = 0x88;
  inputs.frame_cfg->bg_color[1] = 0x3a;
  inputs.frame_cfg->bg_color[2] = 0xbd;
  inputs.frame_cfg->next        = NULL; // End of Frame Config

  // Setup Layer 0 Configuration
   inputs.layer_cfg[0] = (struct layer_cfg_struct *) calloc(1,
sizeof(struct layer_cfg_struct));
  inputs.layer_cfg[0]->enable      = 1;
  inputs.layer_cfg[0]->g_alpha_en  = 0;
```

```
inputs.layer_cfg[0]->priority    = 2;
inputs.layer_cfg[0]->alpha       = 0x80;
inputs.layer_cfg[0]->x_pos       = 0;
inputs.layer_cfg[0]->y_pos       = 0;
inputs.layer_cfg[0]->x_size      = 1280;
inputs.layer_cfg[0]->y_size      = 720;

inputs.layer_cfg[0]->chan_mode[0] = OSD_SOLID_MODE;
inputs.layer_cfg[0]->chan_mode[1] = OSD_SOLID_MODE;
inputs.layer_cfg[0]->chan_mode[2] = OSD_SOLID_MODE;
inputs.layer_cfg[0]->chan_mode[3] = OSD_HRAMP_MODE;

inputs.layer_cfg[0]->chan_color[0] = 0xe0;
inputs.layer_cfg[0]->chan_color[1] = 0x5a;
inputs.layer_cfg[0]->chan_color[2] = 0xbf;
inputs.layer_cfg[0]->chan_color[3] = 0x80; // Alpha
inputs.layer_cfg[0]->next = NULL;
```

OSD Outputs Structure

The structure `xilinx_ip_v_osd_v2_0_outputs` provides the actual output video frames/images of the OSD core. This structure is a wrapper to the standard `video_struct` used by other Xilinx video core C models.

```
struct xilinx_ip_v_osd_v2_0_outputs
{
  struct  video_struct video_out;
}; // xilinx_ip_v_osd_v2_0_outputs
```

The `video_out` structure must be initialized. The following code shows a typical `video_out` initialization.

```
// Setup Output Video Buffer
outputs.video_out.frames          = inputs.num_frames;
outputs.video_out.rows            = inputs.frame_cfg->y_size;
outputs.video_out.cols            = inputs.frame_cfg->x_size;
outputs.video_out.mode            = FORMAT_C444;
outputs.video_out.bits_per_component = generics.C_DATA_WIDTH;
outputs.video_out.data[0]         = NULL;
outputs.video_out.data[1]         = NULL;
outputs.video_out.data[2]         = NULL;
```

OSD Video Structure

Input images or video streams can be provided to the OSD v2.0 reference model using the `video_struct` structure, defined in `video_utils.h`. Output images or video streams are also placed within a `video_struct` structure. The video_struct is defined as:

```
struct video_struct{
  int       frames, rows, cols, bits_per_component, mode;
  uint16*** data[5]; };
```

The structure member variables are defined in Table 3-2.

Table 3-2: **Member Variables of the Video Structure**

Member Variable	Designation
frames	Number of video/image frames in the data structure
rows	Number of rows per frame Pertains to the image plane with the most rows and columns, such as the luminance channel for YUV data. Frame dimensions are assumed constant through the all frames of the video stream, however different planes, such as y, u and v can have different smaller dimensions.
cols	Number of columns per frame Pertains to the image plane with the most rows and columns, such as the luminance channel for YUV data. Frame dimensions are assumed constant through the all frames of the video stream, however different planes, such as y, u and v can have different smaller dimensions.
bits_per_component	Number of bits per color channel/component. All image planes are assumed to have the same color/component representation. Maximum number of bits per component is 16.
mode	Contains information about the designation of data planes. Named constants to be assigned to `mode` are listed in Table 3-3.
data	Set of 5 pointers to 3 dimensional arrays containing data for image planes. `data` is in 16 bit unsigned integer format accessed as data[plane][frame][row][col]

Table 3-3: **Named Constants for Video Modes With Corresponding Planes and Representations**

Mode	Planes	Video Representation
FORMAT_MONO	1	Monochrome – luminance only
FORMAT_RGB	3	RGB image/video data
FORMAT_C444	3	444 YUV, or YCrCb image/video data
FORMAT_C422	3	422 format YUV video, (u, v chrominance channels horizontally sub-sampled)
FORMAT_C420	3	420 format YUV video, (u, v sub-sampled both horizontally and vertically)
FORMAT_MONO_M	3	Monochrome (luminance) video with motion
FORMAT_RGBA	4	RGB image/video data with alpha (transparency) channel
FORMAT_C420_M	5	420 YUV video with motion or alpha
FORMAT_C422_M	5	422 YUV video with motion or alpha
FORMAT_C444_M	5	444 YUV video with motion or alpha
FORMAT_RGBM	5	RGB video with motion

Working With Video_struct Containers

The `video_utils.h` file defines functions to simplify access to video data in `video_struct`.

```
int video_planes_per_mode(int mode);
int video_rows_per_plane(struct video_struct* video, int plane);
int video_cols_per_plane(struct video_struct* video, int plane);
```

Function `video_planes_per_mode` returns the number of component planes defined by the mode variable, as described in Table 3-3. Functions `video_rows_per_plane` and `video_cols_per_plane` return the number of rows and columns in a given plane of the selected video structure. The following example demonstrates using these functions in conjunction to process all pixels within a video stream stored in variable `in_video`, with this construct:

```
for (int frame = 0; frame < in_video->frames; frame++) {
  for (int plane = 0; plane < video_planes_per_mode(in_video->mode); plane++) {
    for (int row = 0; row < rows_per_plane(in_video,plane); row++) {
      for (int col = 0; col < cols_per_plane(in_video,plane); col++) {
   // User defined pixel operations on
// in_video->data[plane][frame][row][col]
      }
    }
  }
}
```

Delete the Video Structure

Finally, large arrays such as the `video_in` element in the video structure must be deleted to free up memory. As an example, the following function is defined as part of the `video_utils` package.

```
void free_video_buff(struct video_struct* video )
{
  int plane, frame, row;

  if (video->data[0] != NULL) {
    for (plane = 0; plane <video_planes_per_mode(video->mode); plane++) {
      for (frame = 0; frame < video->frames; frame++) {
        for (row = 0; row<video_rows_per_plane(video,plane); row++) {
          free(video->data[plane][frame][row]);
        }
        free(video->data[plane][frame]);
      }
      free(video->data[plane]);
    }
  }
}
```

This function can be called in the following way to free the video input buffers (up to eight) and the video output buffer:

```
// Free Layer Buffers
for(i=0; i < generics.C_NUM_LAYERS; i++)
{
  printf("Freeing Layer Video Buffer #%d...\n", i);
  free_video_buff(&inputs.video_in[i]);
}
printf("Freeing Output Buffer...\n");
free_video_buff(&outputs.video_out);
```

XILINX®

Chapter 4

C Model Example Code

Two example C files, `run_bitacc_cmodel.c` and `bitacc_cmodel_config.c`,are provided. The 32-bit and 64-bit Windows and Linux executables for these examples are also included. This C file has these characteristics:

The `run_bitacc_cmodel` example executable provides:

- Shows a fixed implementation of the OSD, including two VFBC layers populated from the internal test pattern generator and one graphics controller layer.
- Contains an example of how to write an application that makes all necessary function calls to the OSD C model core function.
- Contains an example of how to populate the video structures at the input and output, including allocation of memory to these structures.
- Uses a YUV file reading function to extract video information from YUV files for use by the model.
- Uses a YUV file writing function to provide an output YUV file, which allows the user to visualize the result of the core.

The `run_bitacc_cmodel` example executable does not use command line parameters. To run the executable:

1. Use the **`cd`** command to go to the platform directory (lin64, lin, win64 or win32).
2. Enter this command at the shell or DOS prompt:

```
run_bitacc_cmodel
```

The `run_bitacc_cmodel_config` example executable provides:

- Shows configurable implementations of the OSD configured from a config file or command line arguments.
- Includes a config file parser, allowing the user to pass parameters into the model for multiple test cases.
- Uses YUV or BMP file reading functions to extract video information from YUV or BMP files for use by the model.
- Uses YUV or BMP file writing functions to provide an output YUV or BMP file, which allows the user to visualize the result of the core.

The `run_bitacc_cmodel` _config example executable uses multiple command line parameters. To run the executable:

1. Use the **`cd`** command to go to the platform directory (lin64, lin, win64 or win32).
2. Enter this command at the shell or DOS prompt:

```
run_bitacc_cmodel_config -i <Config Filename> <-parameter=value …>
```

Config File Format

The config file defines configuration generics, register settings and test parameters for each video frame to be simulated by the C model. The basic file format is a series of lines each containing a parameter-value pair separated by an '='. An example config file snippet is provided here:

```
C_DATA_WIDTH = 8
C_NUM_LAYERS = 2
T_NUM_FRAMES = 2
# FORMAT_RGB
T_COLORSPACE = 8
C_NUM_DATA_CHANNELS = 3
C_OUTPUT_MODE = 1
C_LAYER0_TYPE = 2
C_LAYER1_TYPE = 2
T_OUTFILE = example0.bmp

[FRAME 1]
R_X_SIZE = 1280
R_Y_SIZE = 720
R_BGCOLOR0 = 0x10
R_BGCOLOR1 = 0x80
R_BGCOLOR2 = 0x80

R_LAYER0_ENABLE = 1
R_LAYER0_G_ALPHA_EN = 1
R_LAYER0_PRIORITY = 1
R_LAYER0_ALPHA = 0xff
R_LAYER0_X_POS = 0
R_LAYER0_Y_POS = 0
R_LAYER0_X_SIZE = 640
R_LAYER0_Y_SIZE = 720

T_LAYER0_CHAN0_MODE = 5
T_LAYER0_CHAN1_MODE = 5
T_LAYER0_CHAN2_MODE = 5
T_LAYER0_CHAN3_MODE = 5
T_LAYER0_CHAN0_COLOR = 2
T_LAYER0_CHAN1_COLOR = 0xa0
T_LAYER0_CHAN2_COLOR = 0xb0
T_LAYER0_CHAN3_COLOR = 0xc0
```

Configuration generics are prefixed with "C_", OSD hardware registers are prefixed with "R_" and test parameters are prefixed with "T_". Settings can be changed for each video frame. Video frame settings are delineated by a single line containing "[FRAME <*num*>]", where <*num*> is an integer denoting the frame number. Global parameters (generics and some test parameters) must be before the first "[FRAME <*num*>]" line. Comment lines are those lines in which the first non-white-space character is '#' or ';'. See Table 4-1 for a full list of all valid parameters.

Table 4-1: **Global Parameters**

Parameter	Valid Range	Description
Global Parameters		Global parameters must be outside of [FRAME <*num*>] sections.
C_DATA_WIDTH	8,10,12	Data width of each color component channel.
C_NUM_LAYERS	1-8	Number of layers.
C_LAYER<*num*>_TYPE	1,2,3	Defines the layer type: 1 = Graphics Controller 2 = VFBC. Loads data from a file or from an internally generated test pattern. The T_LAYER<*num*>_CHAN0_MODE (see below) defines if the layer data is from internal test pattern or from file. If the T_COLORSPACE is set to 8, the file format expected is .bmp. If T_COLOR_SPACE is set to 1,2 or 3, the file format expected is .yuv. 3 = XSVI. Same as VFBC.
C_LAYER<*num*>_INS_BOX_EN	0,1	Enable Box instructions. If 0, then all box instructions in the instruction files are ignored.
C_LAYER<*num*>_INS_TEXT_EN	0,1	Enable Text Instructions. If 0, then all text instructions in the instruction files are ignored. Both C_LAYER<*num*>_INS_BOX_EN and C_LAYER<*num*>_INS_TEXT_EN must be enabled to enable the box text instruction.
C_LAYER<*num*>_IMEM_SIZE	4-4096	Maximum number of instructions .
C_LAYER<*num*>_CLUT_SIZE	16 or 256	Maximum number of colors.
C_LAYER<*num*>_TEXT_NUM_STRINGS	1 – 256	Maximum number of strings.
C_LAYER<*num*>_TEXT_MAX_STRING_LENGTH	32,64,128,256	Maximum string length.
C_LAYER<*num*>_FONT_NUM_CHARS	1-256	Maximum number of characters.
C_LAYER<*num*>_FONT_WIDTH	8,16	Maximum Font Width.
C_LAYER<*num*>_FONT_HEIGHT	8,16	Maximum Font Height.
C_LAYER<*num*>_FONT_BPP	1,2	Font bits per pixel. 1 corresponds to 2 color font and 2 corresponds to 4 color font.
C_LAYER<*num*>_FONT_ASCII_OFFSET	0 - (C_LAYER<*num*>_FONT_NUM_CHARS) -1	ASCII value of the first character in the font file.
T_NUM_FRAMES	1-	Number of frames to simulate
T_COLORSPACE	1,2,3,8	Color space: 1 = YUV 4:2:0 2 = YUV 4:2:2 3 = YUV 4:4:4 8 = RGB
T_OUTFILE	Any String	Destination file name to write output data. If the T_COLORSPACE is set to 8, this file will be in 24-bit .bmp format, otherwise this file is a planar .yuv file.

Table 4-1: **Global Parameters**

T_LAYER<*num*>_VIDEO_FILE	Any String	Defines the .bmp or .yuv file used to read layer data if the C_LAYER<*num*>_TYPE is set to 2.
T_LAYER<*num*>_INSTRUCTION_FILE	Any String	File name of instruction file. The OSD C model does not include a default set of instructions internally. This parameter must be set if using the graphics controller. See Instruction File Format.
T_LAYER<*num*>_CLUT_FILE	Any String	File name of color LUT file. The OSD C model does not include a default color LUT internally. This parameter must be set if using the graphics controller. See Color LUT File Format.
T_LAYER<*num*>_FONT_FILE	Any String	File name of font file. The OSD C model does not include a default font internally. This parameter must be set if using the graphics controller. See Font File Format.
T_LAYER<*num*>_TEXT_FILE	Any String	File name of string file. The OSD C model does not include a default set of strings internally. This parameter must be set if using the graphics controller. See String File Format.
Frame Parameters		Frame Parameters can be defined and redefined for each frame.
R_X_SIZE	1 – 4096	Width of OSD output frames.
R_Y_SIZE	1 – 4096	Height of OSD output frames.
R_BGCOLOR0	0x00 – 0xfff	Background color component 0 – R or Y. Maximum value for data width of 8 is 0xff. Maximum value for data width of 10 is 0x3ff. Maximum value for data width of 12 is 0xfff.
R_BGCOLOR1	0x00 – 0xfff	Background color component 1 – G or U. Maximum value for data width of 8 is 0xff. Maximum value for data width of 10 is 0x3ff. Maximum value for data width of 12 is 0xfff.
R_BGCOLOR2	0x00 – 0xfff	Background color component 2 – B or V.Maximum value for data width of 8 is 0xff. Maximum value for data width of 10 is 0x3ff. Maximum value for data width of 12 is 0xfff.
R_LAYER<*num*>_ENABLE	0,1	Enables layer when 1.
R_LAYER<*num*>_G_ALPHA_EN	0,1	Enables global alpha when 1. When 0, pixel alpha values are used.
R_LAYER<*num*>_PRIORITY	0-7	Z-plane order. Lower values denotes layers that are below layers with higher priority.
R_LAYER<*num*>_ALPHA	0-0xfff	Alpha value for 100% opaque to 100% transparent. Maximum value for data width of 8 is 0xff. Maximum value for data width of 10 is 0x3ff. Maximum value for data width of 12 is 0xfff.
R_LAYER<*num*>_X_POS	0 – (R_X_SIZE-1)	X position of upper-left corner of layer.
R_LAYER<*num*>_Y_POS	0 – (R_Y_SIZE-1)	Y position of upper-left corner of layer.
R_LAYER<*num*>_X_SIZE	0 – R_X_SIZE	Width of layer.
R_LAYER<*num*>_Y_SIZE	0 – R_Y_SIZE	Height of layer.

Table 4-1: **Global Parameters**

T_LAYER<*num*>_CHAN0_MODE	0 – 7	The test mode of color channel/component 0 (R or Y) 0 = OSD_PREFILL_MODE: Denotes that the layer buffer is pre-filled with data before the OSD core simulation begins. The OSD model will expect to read input data from the T_LAYER<*num*>_VIDEO_FILE in this mode. 1 = OSD_GRAPHICS_MODE: Denotes that the layer data will be generated from the graphics controller. All the graphics controller files must be setup. 2 = OSD_CHECKER_MODE: Channel data is generated from internal test pattern generator. Channel data filled with T_LAYER<*num*>_CHAN0_COLOR in the upper-left and lower-right quadrants and filled with the bit-reversed color in the upper-right and lower-left quadrants. 3 = OSD_RAND_MODE: Channel data is generated from internal test pattern generator. Channel data is filled with random data. The value of T_LAYER<*num*>_CHAN0_MODE is used as the seed. 4 = OSD_SOLID_MODE: Channel data is generated from internal test pattern generator. Channel data is filled with the value of T_LAYER<*num*>_CHAN0_MODE. 5 = OSD_HRAMP_MODE: Channel data is generated from internal test pattern generator. Channel data is filled with a horizontal ramp, values incremented every pixel. 6 = OSD_VRAMP_MODE: Channel data is generated from internal test pattern generator. Channel data is filled with a vertical ramp, values incremented every line. 7 = OSD_TEMPR_MODE: Channel data is generated from internal test pattern generator. Channel data is filled with a temporal ramp, values incremented every frame. NOTE: If T_LAYER<*num*>_CHAN0_MODE is set to 0 or 1, then T_LAYER<*num*>_CHAN1_MODE through T_LAYER<*num*>_CHAN3_MODE is ignored.
T_LAYER<*num*>_CHAN1_MODE	0 - 7	Same as T_LAYER<*num*>_CHAN0_MODE for channel 1
T_LAYER<*num*>_CHAN2_MODE	0 - 7	Same as T_LAYER<*num*>_CHAN0_MODE for channel 2
T_LAYER<*num*>_CHAN3_MODE	0 - 7	Same as T_LAYER<*num*>_CHAN0_MODE for channel 3 (alpha)
T_LAYER<*num*>_CHAN0_COLOR	0 – 0xfff	Used when T_LAYER<*num*>_CHAN0_MODE is set to 2-7. Used to set the color or to configure the internal test pattern generator for channel 0. Maximum value for data width of 8 is 0xff. Maximum value for data width of 10 is 0x3ff. Maximum value for data width of 12 is 0xfff.
T_LAYER<*num*>_CHAN1_COLOR	0 – 0xfff	Same as T_LAYER<*num*>_CHAN0_COLOR for channel 1 Maximum value for data width of 8 is 0xff. Maximum value for data width of 10 is 0x3ff. Maximum value for data width of 12 is 0xfff.
T_LAYER<*num*>_CHAN2_COLOR	0 – 0xfff	Same as T_LAYER<*num*>_CHAN0_COLOR for channel 2 Maximum value for data width of 8 is 0xff. Maximum value for data width of 10 is 0x3ff. Maximum value for data width of 12 is 0xfff.

Table 4-1: **Global Parameters**

T_LAYER<*num*>_CHAN3_COLOR	0 – 0xfff	Same as T_LAYER<*num*>_CHAN0_COLOR for channel 3 (alpha) Maximum value for data width of 8 is 0xff. Maximum value for data width of 10 is 0x3ff. Maximum value for data width of 12 is 0xfff.

Color LUT File Format

The color LUT file defines the color pallet used by the graphics controller. Each graphics controller can have a different color LUT file just as the OSD hardware can have different color LUT memory. The format of the file is plain text containing a series of decimal or hexadecimal numbers separated by white space or new line characters. Only the lower 8-bits of each number are used.

The order of the file is channel0, channel1, channel2, and alpha for each color entry starting at entry zero. Here is an example color LUT file:

```
0x00 0x00 0x00 0x00
0x10 0x80  128 0xc0
0x51 0x5a 0xef 0x80
0x89 0x52 0x46  128
0x6b 0xba 0x65 0x80
```

The first line shows all color 0 and has all channels including alpha set to zero. The second line defines color 1 to be black in YUV with an alpha of 192. The remaining lines define color 2, 3 and 4 as red, green and blue in YUV, all with an alpha of 128 or 50% transparent.

The OSD can have a color LUT with 16 colors or 256 colors (64 or 1024 separate numbers for all channels). Not all entries need to be defined. Those entries not defined are set to zero. Consequently, the previous example defines only color entries 0, 1, 2, 3 and 4. Entries 5 through to the end of the table are zero.

Colors can be changed for each video frame (just as in the OSD hardware) by providing multiple color LUTs within the file. The first C_LAYER<*num*>_CLUT_SIZE numbers are used for frame 1, the next C_LAYER<*num*>_CLUT_SIZE numbers are used for the next frame, and so on. If the number of frames is more than the number of color LUTs in the file, then the last color LUT is used for all remaining frames.

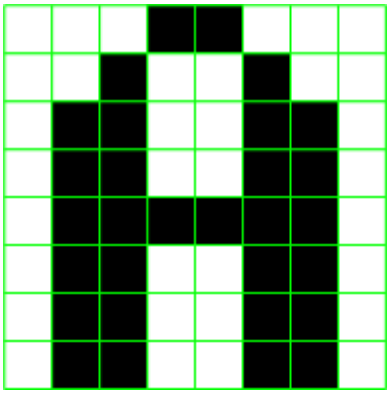
The Xilinx LogiCORE IP Video OSD C model does not include a default color LUT internally. The color LUT must be initialized from file if using the graphics controller.

Font File Format

The font file defines the bits used to define each pixel of each line of each character used by the graphics controller. Each graphics controller can have a different font file just as the OSD hardware can have different font memory. The format of the file is plain text containing a series of decimal or hexadecimal numbers separated by white space or new-line characters.

The order of the file is line 0, line 1, line 2, etc for each character. The number of lines for each character is defined by the C_LAYER<num>_FONT_HEIGHT parameter. The number of bits for each line is defined by C_LAYER<*num*>_FONT_WIDTH * C_LAYER<*num*>_FONT_BPP. The first character in the font file does not have to define character 0. Instead, the first character is set by the C_LAYER<*num*>_FONT_ASCII_OFFSET. Here is an example font file:

```
0x18
0x24
0x66
0x66
0x7e
0x66
0x66
0x66
```



This example shows a snippet of the font file for C_LAYER<*num*>_FONT_WIDTH=8, C_LAYER<*num*>_FONT_HEIGHT=8, C_LAYER<*num*>_FONT_BPP=1 and C_LAYER<*num*>_FONT_ASCII_OFFSET=32. The eight lines shown are for the capital letter 'A', ASCII 65. These lines would be the 33rd (65-32) character definition and lines 265 through 272 in the font file.

Fonts can be changed in each video frame (just as in the OSD hardware) by providing multiple fonts within the file. The first C_LAYER<*num*>_FONT_NUM_CHARS * C_LAYER<*num*>_FONT_WIDTH * C_LAYER<*num*>_FONT_HEIGHT numbers are used for frame 1, the next C_LAYER<*num*>_FONT_NUM_CHARS *C_LAYER<*num*>_FONT_WIDTH * C_LAYER<*num*>_FONT_HEIGHT numbers are used for the next frame, and so on. If the number of frames is more than the number of fonts in the file, then the last font is used for all remaining frames.

The Xilinx LogiCORE IP Video OSD C model does not include a default font internally. The font must be initialized from a file if using the graphics controller.

String File Format

The string file defines the text strings used by the graphics controller. Each graphics controller can have a different font file just as the OSD hardware can have different font memory. The format of the file is plain text containing one string of characters including spaces per line.

The order of the file is string 0, string 1, string 2, and so on, again, one string per line. The number of strings for each graphics controller is defined by this parameter:

C_LAYER<*num*>_TEXT_NUM_STRINGS

The maximum number of characters (including the terminating NULL character) is defined by this parameter:

C_LAYER<*num*>_TEXT_MAX_STRING_LENGTH parameter

Here is an example string file:

```
This is String # 0.  It is on one line!
String 1
Xilinx
OSD
Menu
!&^%!@#*
```

In the previous example file, only the first lines (up to C_LAYER<num>_TEXT_NUM _STRINGS number of lines) are used. All other lines are ignored. Also, the first characters of each line (up to C_LAYER<num>_TEXT_MAX_STRING_LENGTH) are used. All other characters are ignored. If the maximum string length was set to 8, the first string would be truncated to "This is\0".

Note: In the OSD hardware, any character after the first NULL character in a string is ignored and not displayed.

Strings can be changed in each video frame by providing multiple sets of strings within the string file. The first C_LAYER<*num*>_TEXT_NUM_STRINGS number of lines are used for frame 1, the next C_LAYER<*num*>_TEXT_NUM_STRINGS number of lines are used for the next frame, and so on. If the number of frames is more than the number of sets of strings in the file, then the last set of strings are used for all remaining frames.

The Xilinx LogiCORE IP Video OSD C model does not include a default set of strings internally. The text strings must be initialized from a file if using the graphics controller.

Instruction File Format

The instruction file defines the instructions used by the graphics controller. Each graphics controller can have a different instruction file just as the OSD hardware can have different instruction memory. The format of the file is plain text containing one string of characters including spaces per line. One full instruction is contained on each line.

The order of the file is instruction, x_start, x_stop, y_start, y_stop, color_index, text_index and object_size on each line. The instruction field is a text string describing the graphics instruction. All other fields are either decimal or hexadecimal numbers for the parameters of the instruction.

Here is an example instruction file:

```
##########################
# Frame 1 Instructions
##########################
BOX      10  20  40  80 1 0 4
BOX      40  60  70  90 3 0 4
TEXT    100 100  50  50 2 1 0x40
BOXTEXT  30  40  30  40 2 2 0x14
END
##########################
# Frame 2 Instructions
##########################
TEXT    100 100  50  50 2 1 0x40
BOX      20  40  40  80 1 0 4
BOX      40  80  70  90 3 0 4
TEXT    200 100  50  50 2 1 0x40
BOXTEXT  30  40  30  40 2 2 0x14
END
```

Each field is described in Table 4-2.

Table 4-2: **Instruction File Fields**

Field	Valid Range	Description
Instruction	BOX, TEXT, BOXTEXT, END	The graphics instruction.
X_start	0 – end of line	Starting draw x position of the instruction.
X_stop	0 – end of line	Ending draw x position of the instruction.
Y_start	0 – end of frame	Starting draw y position of the instruction.
Y_stop	0 – end of frame	Ending draw y position of the instruction.
Color index	0 – 15 or 255	The color to be used for the graphics object. For boxes, this color index is used directly. For Text with BPP=1, the color index is used for the background and the color index + 1 is used for the foreground. For Text with BPP=2, the color index is used for bits “00” in the font, color index + 1 for bits “01”, color index + 2 for “10” and color index + 3 for “11”.
Text index	0 – (number of strings -1)	The text string to draw.
Object Size	0 – 0xff	For BOX, Size of boxes. For BOXTEXT, [3:0] size of boxes, [7;4] size of text. For TEXT, bits [7:4] size of text.

See *UG801- LogiCORE IP On-Screen Display User Guide* for more information on the format of each instruction.

There are two "END"s in the example instruction file because the file is used to describe the instructions for each video frame. All instructions from the beginning of the file to the first END are displayed on frame 1. For each frame following, the instructions between each subsequent "END" are displayed. If the number of frames is more than the number of "END"s in the file, then the last set of instructions are displayed for all remaining frames.

The Xilinx LogiCORE IP Video OSD C model does not include a default set of instructions internally. The instructions must be initialized from a file if using the graphics controller.

Initializing the OSD Input Video Structure

The easiest way to assign stimuli values to the input video structure is to initialize it with an image or video. The `bmp_util.h`, `yuv_utils.h`, `rgb_utils.h` and `video_util.h` header files packaged with the bit accurate C models contain functions to facilitate file I/O.

Bitmap Image Files

The `rgb_utils.h` and `bmp_utils.h` files declare functions that help access files in Windows bitmap format (http://en.wikipedia.org/wiki/BMP_file_format). However, this format limits color depth to a maximum of 8 bits per pixel, and operates on images with three planes (R,G,B). Consequently, the following functions operate on arguments

type rgb8_video_struct, which is defined in `rgb_utils.h`. Also, both functions support only true color, non-indexed formats with 24 bits per pixel.

```
int write_bmp(FILE *outfile, struct rgb8_video_struct *rgb8_video);
int read_bmp(FILE *infile, struct rgb8_video_struct *rgb8_video);
```

These functions are used to dynamically allocate and free memory for RGB structure storage:

```
int alloc_rgb8_frame_buff(struct rgb8_video_struct* rgb8video );
void free_rgb_frame_buff(struct rgb_video_struct* rgb_video );
```

Exchanging data between rgb8_video_struct and general video_struct type frames/videos is facilitated by functions:

```
int copy_rgb8_to_video(struct rgb8_video_struct* rgb8_in,
struct video_struct* video_out );
int copy_video_to_rgb8( struct video_struct* video_in,
struct rgb8_video_struct* rgb8_out );
```

Note: All image / video manipulation utility functions expect both input and output structures initialized; for example, pointing to a structure that has been allocated in memory, either as static or dynamic variables. Additionally, the input structure must have the dynamically allocated containers (data, r, g, b, y, u, and v arrays) already allocated and initialized with the input frame(s). If the output container structure is pre-allocated at the time of the function call, the utility functions verify and issue an error if the output container size does not match the size of the expected output. If the output container structure is not pre-allocated, the utility functions create the appropriate container to hold results.

YUV Image/Video Files

The `yuv_utils.h` file declares functions that support file access in YUV format. To view and play YUV files, access YUV players at http://www.yuvplayer.com/ or http://dsplab.diei.unipg.it/pyuv_raw_video_sequence_player.

These functions are used to dynamically allocate and free memory for YUV structure storage:

```
int alloc_yuv8_frame_buff(struct yuv8_video_struct* yuv8video );
void free_yuv_frame_buff(struct yuv_video_struct* yuv_video );
```

These functions allow reading and writing of YUV functions (used to initialize or write yuv8_video data):

```
int write_yuv(FILE *outfile, struct yuv8_video_struct *yuv8_video);
int read_yuv(FILE *infile, struct yuv8_video_struct *yuv8_video);
```

Exchanging data between yuv8_video_struct and general video_struct type frames/videos is facilitated by functions:

```
int copy_yuv8_to_video(struct yuv8_video_struct* yuv8_in,
struct video_struct* video_out );
int copy_video_to_yuv8( struct video_struct* video_in,
struct yuv8_video_struct* yuv8_out );
```

YUV formats (4:2:0, 4:2:2 and 4:4:4) can be converted with these functions:

```
int yuv8_420to444(struct yuv8_video_struct* video_in, struct yuv8_video_struct* video_out);
int yuv8_422to444(struct yuv8_video_struct* video_in, struct yuv8_video_struct* video_out);
int yuv8_444to420(struct yuv8_video_struct* video_in, struct yuv8_video_struct* video_out);
int yuv8_444to422(struct yuv8_video_struct* video_in, struct yuv8_video_struct* video_out);
```

Binary Image/Video Files

The `video_utils.h` file declares functions that help load and save generalized video files in raw, uncompressed format. These functions effectively serialize the video_struct structure:

```
int read_video( FILE* infile,  struct video_struct* in_video);
int write_video(FILE* outfile, struct video_struct* out_video);
```

The corresponding file contains a small, plain text header defining, "Mode", "Frames", "Rows", "Columns", and "Bits per Pixel". The plain text header is followed by binary data, 16-bits per component in scan line continuous format. Subsequent frames contain as many component planes as defined by the video mode value selected. Also, the size (rows, columns) of component planes can differ within each frame as defined by the actual video mode selected.

These functions are used to dynamically allocate and free memory for video structure storage:

```
int alloc_video_buff(struct video_struct* video );
void free_video_buff(struct video_struct* video );
```

Compiling on 32-bit and 64-bit Windows Platforms

Precompiled library `v_osd_v2_0_bitacc_cmodel.lib`, top level demonstration code `run_bitacc_cmodel_config.c` and example code `run_bitacc_cmodel.c` must be compiled with an ANSI C compliant compiler under Windows 32-bit or Windows 64-bit. This section describes an example using Microsoft Visual Studio.

In Visual Studio create a new, empty Win32 Console Application project. As existing items, add:

- `libIpv_osd_v2_0_bitacc_cmodel.lib` to the "Resource Files" folder of the project
- `run_bitacc_cmodel.c` or the `run_bitacc_cmodel_config.c` to the "Source Files" folder of the project
- `v_osd_v2_0_bitacc_cmodel.h` header file to the "Header Files" folder of the project
- `bmp_utils.h` file to the "Header Files" folder of the project
- `rgb_utils.h` file to the "Header Files" folder of the project
- `video_fio.h` file to the "Header Files" folder of the project
- `video_utils.h` file to the "Header Files" folder of the project
- `yuv_utils.h` file to the "Header Files" folder of the project

To build the x64 executable for 64-bit Windows platforms, perform these steps. These steps can be skipped if building the Win32 executable.

1. Right-click on the solution in the Solution Explorer and click **Properties** at the bottom of the pop-up menu.
2. Click **Configuration Manager**.
3. In the Active solution platform drop-down box, select **<New…>**.
4. In the new platform drop-down box, select **x64** and click **OK**.

   Make sure that all the projects now have x64 as the default platform in the Configuration Manager.

5. After the project is created and populated, it must be compiled and linked (built) to create a Win32 or x64 executable. To perform the build step, select **Build Solution** from the Build menu. An executable matching the project name is created either in the Debug or Release subdirectories under the project location based on whether "Debug" or "Release" has been selected in the "Configuration Manager" under the Build menu.

Note: The `run_bitacc_cmodel.c` file is an example demonstration that reads no input but generates an output .yuv file from internally generated test patterns. The `run_bitacc_cmodel_config.c` file is a configurable demonstration and requires several input files to run. See Running the Executables for information on command line arguments and input file formats.

Compiling under 64-bit Linux Platforms

Example Demonstration

To compile the example demonstration, go to the directory where the header files, the library files and `run_bitacc_cmodel.c` were unpacked. The libraries and header files are referenced during the compilation and linking process. In this directory, perform these steps:

1. Set your LD_LIBRARY_PATH environment variable to include the root directory where you unzipped the model zip file. For example:

   ```
   setenv LD_LIBRARY_PATH <unzipped_c_model_dir>:${LD_LIBRARY_PATH}
   ```

2. Copy these files from the /lin64 directory to the root directory:

   libstlport.so.5.1

   libIp_v_osd_v2_0_bitacc_cmodel.so

3. In the root directory, compile using the GNU C Compiler by typing this command at the shell prompt:

```
gcc -x c++ run_bitacc_cmodel.c -o run_bitacc_cmodel -L. -lIp_v_osd_v2_0_bitacc_cmodel -Wl,-rpath,.
```

4. This results in the creation of the executable run_bitacc_cmodel, which can be run using this command:

   ```
   ./run_bitacc_cmodel
   ```

   A make file is also included that runs GCC. To clean the executable and compile the example code, enter this command at the shell prompt:

   ```
   make clean all
   ```

Configurable Demonstration

To compile the configurable demonstration, go to the directory where the header files, the library files and `run_bitacc_cmodel_config.c` were unpacked. The libraries and header files are referenced during the compilation and linking process. In this directory, perform these steps:

1. Set your LD_LIBRARY_PATH environment variable to include the root directory where you unzipped the model zip-file. For example:

   ```
   setenv LD_LIBRARY_PATH <unzipped_c_model_dir>:${LD_LIBRARY_PATH}
   ```

2. Copy these files from the /lin64 directory to the root directory:

   libstlport.so.5.1

   libIp_v_osd_v2_0_bitacc_cmodel.so

3. In the root directory, compile using the GNU C Compiler by entering this command at the shell prompt:

```
gcc -x c++ run_bitacc_cmodel_config.c -o run_bitacc_cmodel_config -L. -
lIp_v_osd_v2_0_bitacc_cmodel -Wl,-rpath,.
```

4. This results in the creation of the executable run_bitacc_cmodel, which can be run using this command:

   ```
   ./run_bitacc_cmodel_config -i <Config Filename> <-parameter=value ...>
   ```

   A make file is also included that runs GCC. To clean the executable and compile the example code, enter this following command at the shell prompt:

   ```
   make clean run_bitacc_cmodel_config
   ```

Compiling on 32-bit Linux Platforms

Example Demonstration

To compile the example demonstration, go to the directory where the header files, the library files and `run_bitacc_cmodel.c` were unpacked. The libraries and header files are referenced during the compilation and linking process. In this directory, perform these steps:

1. Set your LD_LIBRARY_PATH environment variable to include the root directory where you unzipped the model zip-file. For example:

   ```
   setenv LD_LIBRARY_PATH <unzipped_c_model_dir>:${LD_LIBRARY_PATH}
   ```

2. Copy these files from the /lin directory to the root directory:

   libstlport.so.5.1

   libIp_v_osd_v2_0_bitacc_cmodel.so

3. In the root directory, compile using the GNU C Compiler by entering this command at the shell prompt:

```
gcc -x c++ run_bitacc_cmodel.c -o run_bitacc_cmodel -L. -lIp_v_osd_v2_0_bitacc_cmodel -Wl,-
rpath,.
```

4. This results in the creation of the executable run_bitacc_cmodel, which can be run using this command:

   ```
   ./run_bitacc_cmodel
   ```

A make file is also included that runs GCC. To compile the example code, enter this command at the shell prompt:

```
make clean all
```

Configurable Demonstration

To compile the configurable demonstration, go to the directory where the header files, the library files and `run_bitacc_cmode_config.c` were unpacked. The libraries and header files are referenced during the compilation and linking process. In this directory, perform these steps:

1. Set your LD_LIBRARY_PATH environment variable to include the root directory where you unzipped the model zip-file. For example:

   ```
   setenv LD_LIBRARY_PATH <unzipped_c_model_dir>:${LD_LIBRARY_PATH}
   ```

2. Copy these files from the /lin directory to the root directory:

   libstlport.so.5.1

   libIp_v_osd_v2_0_bitacc_cmodel.so

3. In the root directory, compile using the GNU C Compiler by entering this command at the shell prompt:

```
gcc -x c++ run_bitacc_cmodel_config.c -o run_bitacc_cmodel_config -L. -
lIp_v_osd_v2_0_bitacc_cmodel -Wl,-rpath,.
```

4. This results in the creation of the executable run_bitacc_cmodel, which can be run using this command:

   ```
   ./run_bitacc_cmodel_config -i <Config Filename> <-parameter=value ...>
   ```

   A make file is also included that runs GCC. To compile the example code, enter this command at the shell prompt:

   ```
   make run_bitacc_cmodel_config
   ```

Running the Executables

Included in the zip file are precompiled executable files for use with 32-bit and 64-bit Windows and Linux platforms. The instructions for running on each platform are included in this section.

Example Demonstration

The example demonstration does not use command line parameters. To run on a 32-bit or 64-bit Linux platform, perform these steps:

1. Set your $LD_LIBRARY_PATH environment variable to include the root directory where you unzipped the model zip file. For example:

   ```
   setenv LD_LIBRARY_PATH <unzipped_c_model_dir>:${LD_LIBRARY_PATH}
   ```

2. Copy these files from the /lin64 (for 64-bit Linux) or from the /lin (for 32-bit Linux) directory to the root directory:

   libstlport.so.5.1

   libIp_v_osd_v2_0_bitacc_cmodel.so

   run_bitacc_cmodel

3. Execute the model. From the root directory, enter this command at a shell prompt:

   ```
   run_bitacc_cmodel
   ```

To run on a 32-bit or 64-bit Windows platform, perform these steps:

1. Copy this file from the /nt64 (for 64-bit Windows) or from the /nt (for 32-bit Windows) directory to the root directory:

   `run_bitacc_cmodel.exe`

2. Execute the model. From the root directory, enter this command at a DOS prompt:

   `run_bitacc_cmodel`

During successful execution, the `test.yuv` file is created in the directory containing the run_bitacc_cmodel executable. This file is a planar YUV file in 4:4:4 format. The example demonstration is set up to generate three frames of video data at 1280x720 resolution. Each frame contains the output of three video layers and background color.

Figure 4-1 shows frame 1 of the `test.yuv` file. The image shows a background color of orange, a video layer with a horizontal ramp, another video layer with random data, and a graphics controller layer with text and boxes.

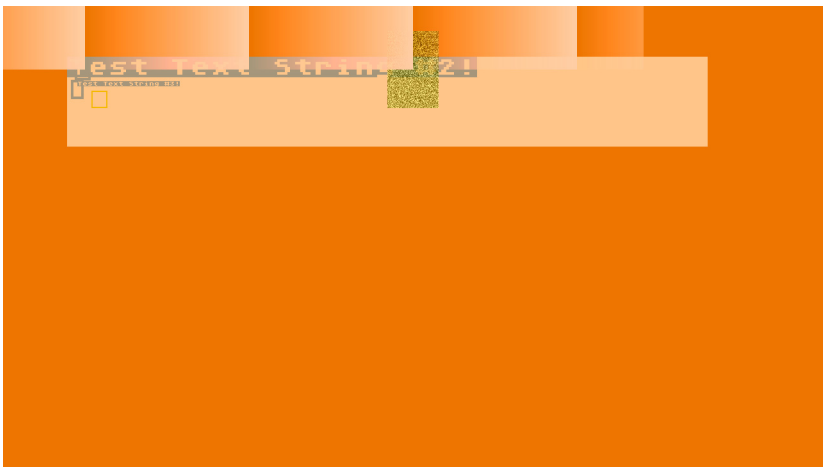


Figure 4-1: **Example Demonstration Output Image**

Configurable Demonstration

The configurable demonstration takes multiple command line parameters. To run on a 32-bit or 64-bit Linux platform, perform these steps:

1. Set your $LD_LIBRARY_PATH environment variable to include the root directory where you unzipped the model zip-file. For example:

   `setenv LD_LIBRARY_PATH <unzipped_c_model_dir>:${LD_LIBRARY_PATH}`

2. Copy these files from the /lin64 (for 64-bit Linux) or from the /lin (for 32-bit Linux) directory to the root directory:

   libstlport.so.5.1

   libIp_v_osd_v2_0_bitacc_cmodel.so

   run_bitacc_cmodel_config

3. Execute the model. From the root directory, enter this command at a shell prompt:

   `run_bitacc_cmodel_config -i <Config Filename> <-parameter=value …>`

To run on a 32-bit or 64-bit Windows platform, perform these steps:

1. Copy this file from the /nt64 (for 64-bit Windows) or from the /nt (for 32-bit Windows) directory to the root directory:

   run_bitacc_cmodel_config.exe

2. Execute the model. From the root directory, enter this command at a DOS prompt:

   ```
   run_bitacc_cmodel_config -i <Config Filename> <-parameter=value …>
   ```

The configurable demonstration reads parameters from the config file specified with the -i <config_file> argument where <config_file> is the relative path and filename of the config file. See Config File Format for more information. Parameters in the config file can be overridden on the command line by prefixing the parameter with a dash ('-') and removing white spaces. For example, the number of frames to simulate can be overridden with this command line argument "-T_NUM_FRAMES=2". Config parameters set on the command line must be set after the -i argument to take effect.

Figure 4-2 shows frame 1 of the output of the configurable demonstration from this command line:

```
run_bitacc_cmodel_config -i examples/example0.cfg
```

The image shows a background color of green, a video layer with a horizontal ramp and another video layer with random data.

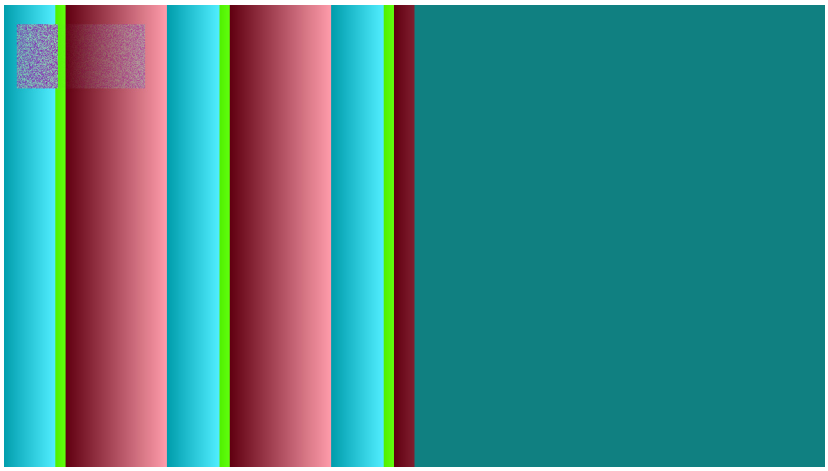


Figure 4-2: **Configurable Demonstration Output Image (Example 0)**

Figure 4-3 shows frame 1 of the output of the configurable demonstration from this command line:

```
run_bitacc_cmodel_config -i examples/example1.cfg
```

The image shows a background color of grey, a video layer with a horizontal ramp and another video layer with a vertical ramp. Each ramp layer (vertical and horizontal) have different ramp starting values for each color component.

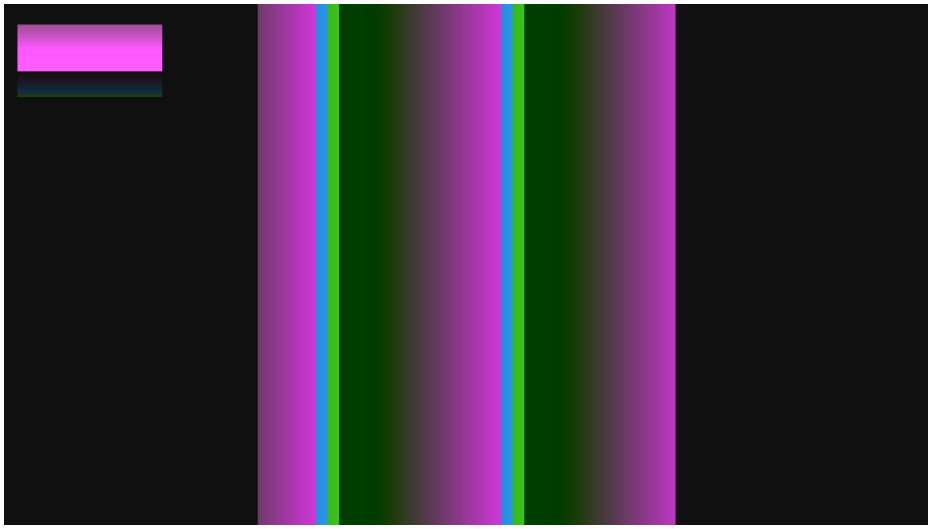


Figure 4-3: **Configurable Demonstration Output Image (Example 1)**

Figure 4-4 shows frame 1 of the output of the configurable demonstration from this command line:

```
run_bitacc_cmodel_config -i examples/example2.cfg
```

The image shows a background color of red, a video layer from a BMP file input and another video layer with random data.

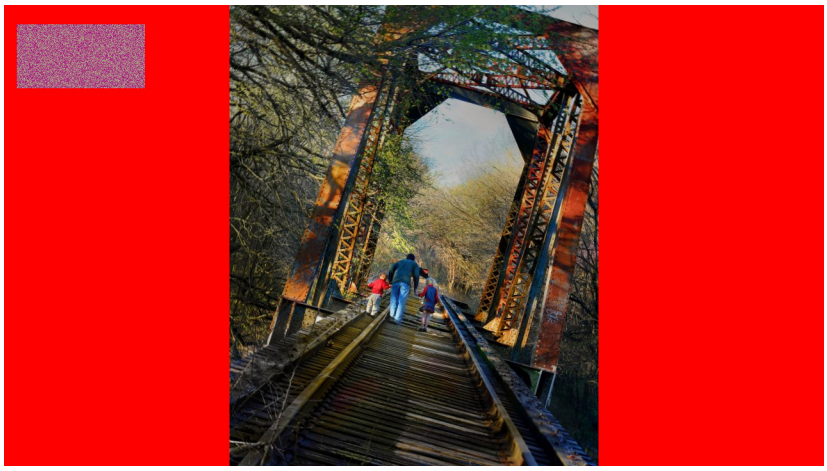


Figure 4-4: **Configurable Demonstration Output Image (Example 2)**

Figure 4-5 shows frame 1 of the output of the configurable demonstration from this command line:

```
run_bitacc_cmodel_config -i examples/example3.cfg
```

The image shows a background color of grey, a video layer from a BMP file input, six other video layers with checkerboard, horizontal ramp and vertical ramp patterns. One graphics controller layer is also displayed generating multi-colored lines, boxes and text.

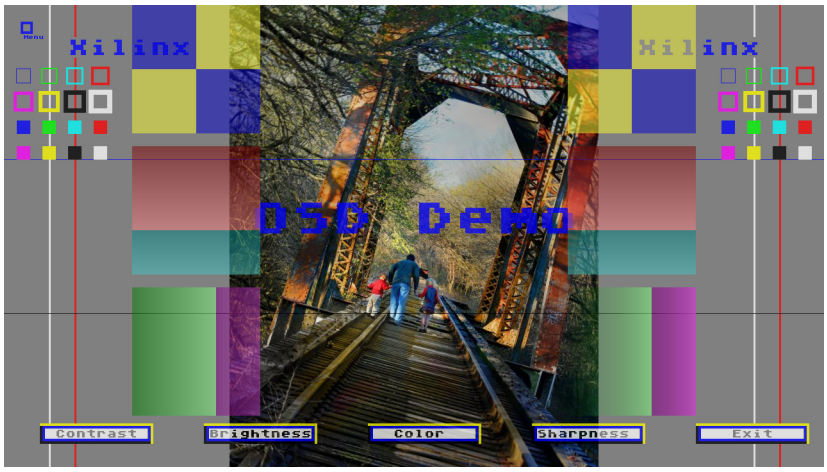


Figure 4-5: **Configurable Demonstration Output Image (Example 3)**